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 21 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。

关键词： *Dijkstra* 最短路径

目录

第一部分　正文

第 1 章　实验背景

1.1　实验目的

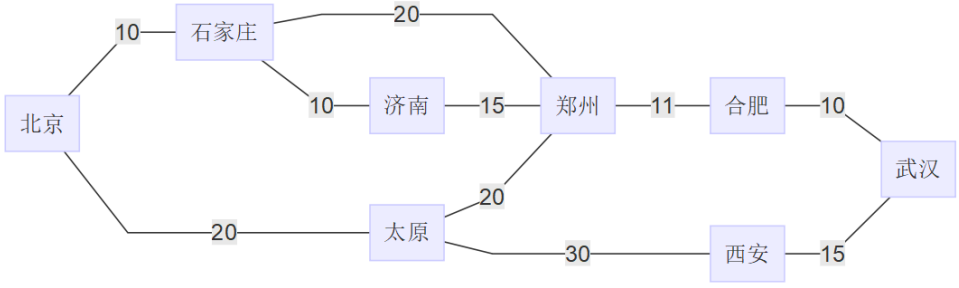
在实际生活中，我们常常会遇到关注点在于最短、最近、最省钱这些方面的问题，就比如下列问题：

- 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 $(V_0,V_1...V_{n-1}) \in V$，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。

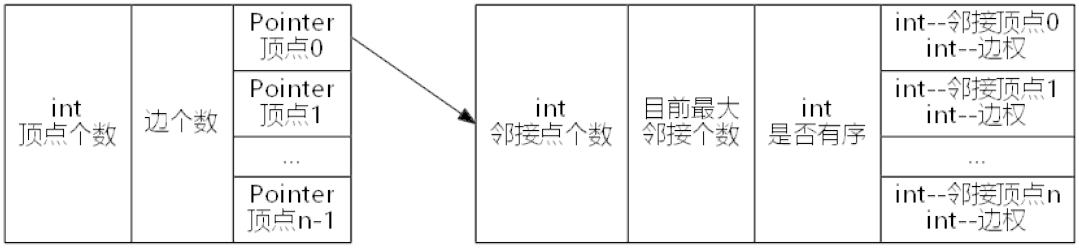
2.2　迪杰斯特拉算法步骤

1. 初始时，S 只包含源点，即 S v，$v-v$ 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 $<u,v>$ 正常有权值，若 u 不是 v 的出边邻接点，则 $<u,v>$ 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，修改后的距离值的顶点 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，又不会带来性能损失。数据结构如下：



结构代码实现如下：

```
1
2  typedef struct _list {
3      int vec;                    //临接顶点
4      int weight;                 //权
5  } link_list;
6
7  struct w_graph {
8      int n;                      //顶点个数
9      int m;                      //边个数
10     struct successors {
11         int d;                  //临接点个数
12         int len;                //最大临接点个数
13         char is_sorted;         //是否有序
14         link_list list[1];      //临接列表
15     } *v_list[1];               //注意：这是一个指针的数组
16 };
17
18 typedef struct w_graph *WGraph;//图是一个指针类型的
```

3.2　设计数据操作

创建一个顶点从 $0 \to n-1$ 的带权图　分配初始内存，此时分配内存时多分配 $n-1$ 块 $Struct\ Succesors^*$（顶点的邻接列表指针）大小的地址，并对每个指针分配默认大小的内存。

从内存中删去一个图　遍历释放各顶点邻接列表的内存，再释放图的内存。

添加边和权　首先确定是否需要对邻接列表的内存进行扩展（使用指数递增的策略扩展），使用 $relloc$ 函数对已经分配的内存进行改变，此时实际上对 $link\_list$ 数组进行了扩容。

返回顶点个数　返回 n。

返回边个数　返回 m。

返回顶点的度　返回 $v\_list[V].d$。

判断是否邻接　通过该点度的大小来确定是否需要对邻接列表排序，数据量少的时候进行遍历。

获取边的权　通过该点度的大小来确定是否需要对邻接列表排序，数据量少的时候进行遍历，而后返回权值。

提供读取顶点数据的接口　接受函数指针 $Func$ 和所需数据指针，内部对邻接列表里的每个顶点施加 $Func$ 操作（将原点，邻接点和权都传给该函数以供操作）。
完整代码参照附录 A。

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（伪代码）

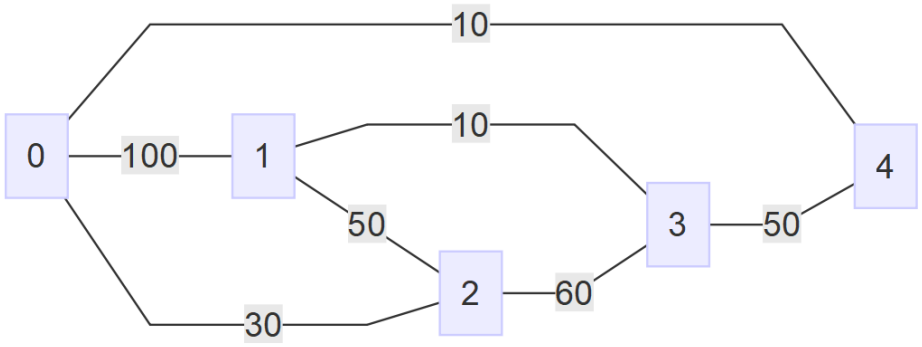
DIJKSTRA'S ALGORITHM(G, v_0)

```
1  for v ∈ G.V
2      Dis(v) = G.W(v, v_0)
3  Set v_0 ∈ S
4  while S ≠ G.V
5      k = Dis(V).V_m
6      Set k ∈ S
7      for v ∉ 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

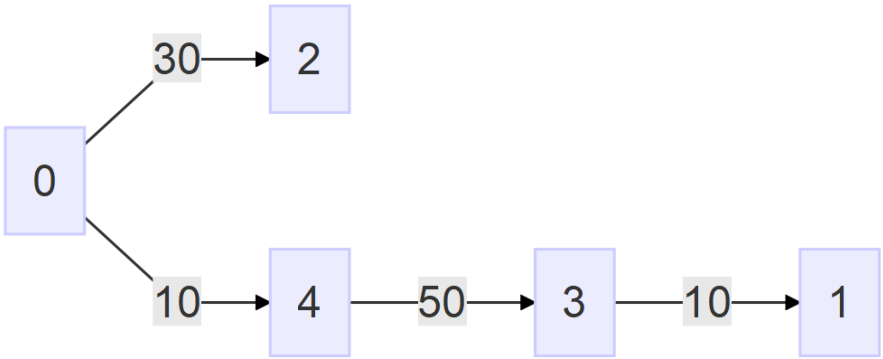
4.1　数据输入

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  // WeightGraph.h
3  // Created by Along on 2017/5/13.
4  //
5
6  #ifndef GRAPH_WEIGHTGRAPH_H
7  #define GRAPH_WEIGHTGRAPH_H
8
9  // 不可达时的返回值
10 #define INFINITY (65535)
11 // 图的定义
12 typedef struct w_graph *WGraph;
13
14 // 创建一个图
15 WGraph w_graph_create(int n);
16
17 // 删除一个图
18 void w_graph_destroy(WGraph);
19
20 // 添加一条边
21 void w_graph_add_edge(WGraph, int source, int sink, int weight);
22
23 // 顶点的数目
24 int w_graph_vector_count(WGraph);
25
26 // 边的数目
27 int w_graph_edge_count(WGraph);
28
29 // 顶点的度
30 int w_graph_out_degree(WGraph, int source);
31
32 // 两个顶点是否邻接
33 int w_graph_has_edge(WGraph, int source, int sink);
34
35 // 边的权
36 int w_graph_weight_edge(WGraph, int source, int sink);
37
38 // 提供遍历数据的接口
39 void w_graph_foreach(WGraph g, int source,
40       void (*f)(WGraph, int src, int sink, int weight, void *),
```

```
41          void *data);
42
43  #endif //GRAPH_WEIGHTGRAPH_H
```

```
1   //
2   // WeightGraph.c
3   // 图的封装
4   // Created by Along on 2017/5/13.
5   // https://github.com/AlongWY/Graph
6   //
7
8   #include <stdio.h>
9   #include <stdlib.h>
10  #include <assert.h>
11  #include "WeightGraph.h"
12
13  //基础带权图定义
14  //使用可变数组表示的临接矩阵
15
16  typedef struct _list {
17      int vec;              //临接顶点
18      int weight;           //权
19  } link_list;
20
21  struct w_graph {
22      int n;                          //顶点个数
23      int m;                          //边个数
24      struct successors {
25          int d;                      //临接点个数
26          int len;                    //最大临接点个数
27          char is_sorted;             //
28          link_list list[1];                  //临接列表
29      } *v_list[1];
30  };
31
32
33  //创建一个顶点从0 ~ n-1的带权图
34  WGraph w_graph_create(int n) {
35      WGraph g;
36      int i;
37
38      g = malloc(sizeof(struct w_graph) +
39      sizeof(struct successors *) * (n - 1));
40      assert(g);
41
```

```
42      g->n = n;
43      g->m = 0;
44
45      for (i = 0; i != n; i++) {
46          g->v_list[i] = malloc(sizeof(struct successors));
47          assert(g->v_list[i]);
48          g->v_list[i]->d = 0;
49          g->v_list[i]->len = 1;
50          g->v_list[i]->is_sorted = 1;
51      }
52
53      return g;
54  }
55
56  //释放内存
57  void w_graph_destroy(WGraph g) {
58      int i;
59
60      for (i = 0; i != g->n; i++) {
61          free(g->v_list[i]);
62      };
63      free(g);
64  }
65
66  //添加边和权
67  void w_graph_add_edge(WGraph g, int u, int v, int weight) {
68      assert(u >= 0);
69      assert(u < g->n);
70      assert(v >= 0);
71      assert(v < g->n);
72
73      //是否需要增长 list
74      while (g->v_list[u]->d >= g->v_list[u]->len) {
75          g->v_list[u]->len *= 2;
76          g->v_list[u] =
77                  realloc(g->v_list[u], sizeof(struct successors) +
78                   sizeof(link_list) * (g->v_list[u]->len - 1));
79      }
80
81      //添加新临接点
82      g->v_list[u]->list[g->v_list[u]->d].vec = v;
83      g->v_list[u]->list[g->v_list[u]->d].weight = weight;
84
85      g->v_list[u]->d++;
86
```

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

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

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

B 最短路径实现

```
1  //
2  // Graph_tools.h
3  // Created by Along on 2017/5/13.
4  //
5
6  #ifndef GRAPH_GRAPH_TOOLS_H
7  #define GRAPH_GRAPH_TOOLS_H
8
```

```c
#include "Graph.h"
#include "WeightGraph.h"

//tools
// 将图打印出来
void w_graph_show(WGraph);

// 将图成Graphviz的格式输出文件以便于生成图片
void w_graph_show_dot(WGraph, char path[]);

// 封装：由于是无权图，  一次添加二条边
void w_graph_add_edge2(WGraph, int source, int sink, int weight);

// 求最短路径部分
typedef struct min_len *Min_len;

// 图的顶点个数
int min_len_vector_count(Min_len);

// 删除最短路径生成数据
void min_len_destroy(Min_len);

// 提供数据遍历操作
void min_len_foreach(Min_len, void(*f)(Min_len, int dist,
                            int prev, void *), void *data);
// 打印最短路径数据
void min_len_show(Min_len);

// 迪杰斯特拉算法
Min_len Dijkstra(WGraph g, int source);
#endif //GRAPH_GRAPH_TOOLS_H
```

```c
//
// Graph_tools.c
// Created by Along on 2017/5/13.
// https://github.com/AlongWY/Graph
//

#include <stddef.h>
#include <assert.h>
#include <malloc.h>
#include <stdio.h>
#include "WeightGraph.h"
#include "Graph_tools.h"

```

```
14 //封装：由于是无权图，  一次添加二条边
15 void w_graph_add_edge2(WGraph g, int source, int sink, int weight) {
16     w_graph_add_edge(g, source, sink, weight);
17     w_graph_add_edge(g, sink, source, weight);
18 }
19
20 //最小生成树所要用到的数据结构和操作
21 struct min_len {
22     int n;
23     int vec;
24     struct list {
25         int dist;            //与所要求顶点的距离
26         int prev;            //前驱动点
27     } a_list[1];
28 };
29
30 //数据遍历接口
31 void min_len_foreach(Min_len m,
32 void (*f)(Min_len, int, int, void *), void *data) {
33     int i;
34     for (i = 0; i != m->n; i++) {
35         f(m, m->a_list[i].dist, m->a_list[i].prev, data);
36     }
37 }
38
39 //顶点个数
40 int min_len_vector_count(Min_len m) {
41     return m->n;
42 }
43
44 //释放内存
45 void min_len_destroy(Min_len m) {
46     free(m);
47 }
48
49 //打印数据
50 static void w_graph_show_vec(WGraph g, int src, int sink,
51 int weight, void *data) {
52     printf("%d:%d ", sink, weight);
53 }
54
55 //调用接口
56 void min_len_show(Min_len ml) {
57     int i, j;
58     for (i = 0; i != ml->n; ++i) {
```

```
59          printf("Dist:%d ", ml->a_list[i].dist);
60          for (j = i; j != ml->vec; j = ml->a_list[j].prev) {
61              printf("%d <- ", j);
62          }
63          printf("%d\n", ml->vec);
64      }
65  }
66
67  //迪杰斯特拉算法
68  Min_len Dijkstra(WGraph g, int source) {
69      int i, j, *S;
70      Min_len res;
71
72      int vec_num = w_graph_vector_count(g);
73
74      assert(source >= 0);
75      assert(source < vec_num);
76
77      res = malloc(sizeof(struct min_len) +
78      sizeof(struct list) * (vec_num - 1));
79      S = calloc((size_t) vec_num, sizeof(int));
80      assert(res);
81      assert(S);
82
83      res->n = vec_num;
84      res->vec = source;
85
86      //初始化各点
87      for (i = 0; i != vec_num; ++i) {
88          S[i] = 0;
89          if (w_graph_has_edge(g, source, i)) {
90              res->a_list[i].dist = w_graph_weight_edge(g, source, i);
91              res->a_list[i].prev = source;
92          } else {
93              res->a_list[i].prev = -1;
94              res->a_list[i].dist = INFINITY;
95          }
96      }
97
98      res->a_list[source].dist = 0;
99      res->a_list[source].prev = source;
100     S[source] = 1;
101
102     for (i = 1; i != vec_num; ++i) {
103         int min_dst = INFINITY;
```

```
104         int u = source;
105         //找出未使用过的点中 dist 最小的
106         for (j = 0; j != vec_num; ++j) {
107             if ((!S[j]) && res->a_list[j].dist < min_dst) {
108                 u = j;      //u是距离 source 最近的点
109                 min_dst = res->a_list[j].dist;
110             }
111         }
112 
113         S[u] = 1;     //将u标记为已使用
114 
115         for (j = 0; j != vec_num; ++j)
116         //j点未被使用且u,j之间有边
117             if ((!S[j]) && w_graph_has_edge(g, u, j)) {
118                 if (res->a_list[u].dist + w_graph_weight_edge(g, u, j)
119                  < res->a_list[j].dist) {
120                     res->a_list[j].dist = res->a_list[u].dist +
121                     w_graph_weight_edge(g, u, j);    //更新距离
122                     res->a_list[j].prev = u;         //更新路径
123                 }
124             }
125     }
126     free(S);
127     return res;
128 }
```