

HARBIN INSTITUTE OF TECHNOLOGY

集合论实验报告：最小生成树

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，我们往往会遇到求整体最短、最近、最省钱的问题... 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小生成树的问题。

关键词： *Prim* *Kruskal* 最小生成树

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	<i>Prim</i> 算法	2
2.1.1	介绍	2
2.1.2	算法步骤	2
2.2	<i>Kruskal</i> 算法	3
2.2.1	介绍	3
2.2.2	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	4
3.3	实现最小生成树算法 B	5
第 4 章	实验结果	5
4.1	数据输入	5
4.2	结果输出	6
第 5 章	实验分析	6
5.1	效率分析	6
5.2	优化策略	6
5.3	算法比较	6
第二部分	附录	7
A	带权图实现	7
B	最小生成树	11

第一部分 正文

第 1 章 实验背景

1.1 实验目的

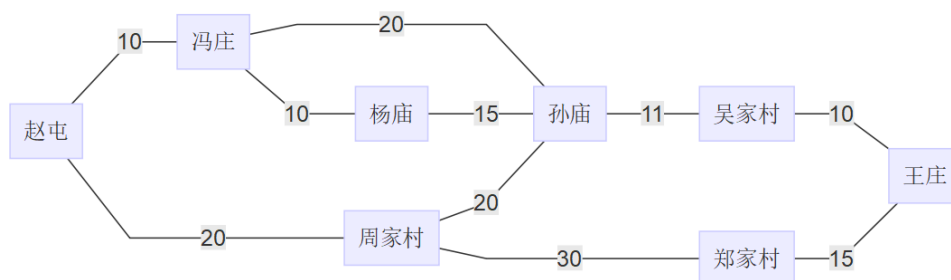
实际问题 我们常常会遇到求整体最短、最近、最省钱这些方面的问题，就比如下列问题：

- 镇子里要铺设自来水管，怎么铺用的水管最少，而且家家都能用到水。
- 在电路设计中，常常需要把一些电子元件的插脚用电线连接起来。如果每根电线连接两个插脚，把所有 n 个插脚连接起来，只要用 $n-1$ 根电线就可以了。在所有的连接方案中，我们通常对电线总长度最小的连接方案感兴趣。
- 要在 n 个城市之间铺设光缆，主要目标是要使这 n 个城市的任意两个之间都可以通信，找出一条使用最短的光纤连通这些城市的铺设方法。

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，上述一个铺设管道的问题，我们就可以将其按如下方式转化：取图 $G(V, E, W)$ ，城镇所对应的顶点集 $(V_0, V_1 \dots V_{n-1}) \in V$ ，若两个城镇 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 Prim 算法

2.1.1 介绍

普里姆算法 (Prim 算法) 图论中的一种算法，可在加权连通图里搜索最小生成树。

该算法于 1930 年由捷克数学家沃伊捷赫·亚尔尼克（英语：Vojtěch Jarník）发现；并在 1957 年由美国计算机科学家罗伯特·普里姆（英语：Robert C. Prim）独立发现；1959 年，艾兹格·迪科斯彻再次发现了该算法。

因此，在某些场合，普里姆算法又被称为 DJP 算法、亚尔尼克算法或普里姆—亚尔尼克算法。

2.1.2 算法步骤

1. 获得 $G(V, E, W)$ ，新建图 $G_{new}(V_{new}, E_{new}, W_{new})$ 。

2. 初始化: $V_{new} = \{x\}$, 其中 x 为集合 V 中的任一节点 (起始点), $E_{new} = \{\}$;
3. 重复下列操作, 直到 $V_{new} = V$:
 - (a) 在集合 E 中选取权值最小的边 $\langle u, v \rangle$, 其中 $u \in V_{new}, v \notin V_{new}$ (如果存在有多条满足前述条件即具有相同权值的边, 则可任意选取其中之一)。
 - (b) 将 v 加入集合 V_{new} 中, 将 $\langle u, v \rangle$ 边加入集合 E_{new} 中, 将 $W \langle u, v \rangle$ 加入到 W_{new} 。
4. 输出: $G_{new}(V_{new}, E_{new}, W_{new})$ 。

2.2 Kruskal 算法

2.2.1 介绍

Kruskal 算法 是一种用来寻找最小生成树的算法, 由 Joseph Kruskal 在 1956 年发表。用来解决同样问题的还有 Prim 算法和 Boruvka 算法等。三种算法都是贪婪算法的应用。和 Boruvka 算法不同的地方是, Kruskal 算法在图中存在相同权值的边时也有效。

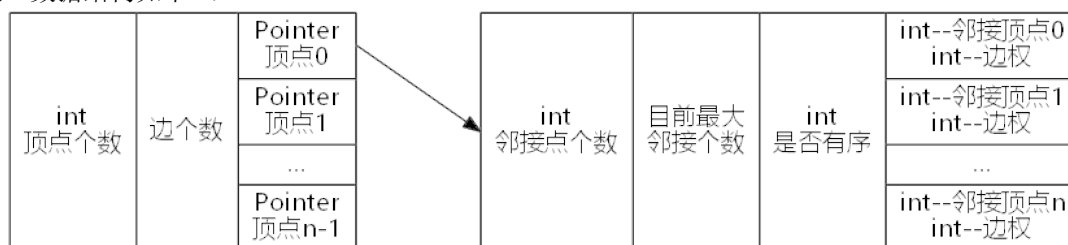
2.2.2 算法步骤

1. 获得 $G(V, E, W)$, 新建图 $G_{new}(V_{new}, E_{new}, W_{new})$, 使 $V_{new} = V$ 。
2. 将原图 $G(V, E, W)$ 中所有边按权值从小到大排序。
3. 重复以下操作, 直至图 G_{new} 中所有的节点都在同一个连通分量中。
 - (a) 获取 G 中的权值最小的边 (若获取过则不再获取)。
 - (b) 如果这条边连接的两个节点于图 G_{new} 中且不在同一个连通分量中, 则添加这条边到图 G_{new} 中。
4. 输出: $G_{new}(V_{new}, E_{new}, W_{new})$ 。

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A:



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 $0 \rightarrow n-1$ 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 *realloc* 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）。

3.3 实现最小生成树算法B

- *Prim* 算法实现

PRIM ALGORITHM(G, v_0)

```

1   $v_{new} = v_0, G_{new}(V_{new}, E_{new}, W_{new})$ 
2  while  $V_{new} \neq V$ 
3      Get  $u \in V_{new}, v \notin V_{new}$  Make  $W(u, v)$  min
4      Set  $v \in V_{new}, \text{Set } \langle u, v \rangle \in E_{new}, \text{Set } W(u, v) \in W_{new}$ 

```

- *Kruskal* 算法实现（实际代码中使用了堆）堆的实现在 Github 上，此处不再赘述。

PRIM ALGORITHM(G)

```

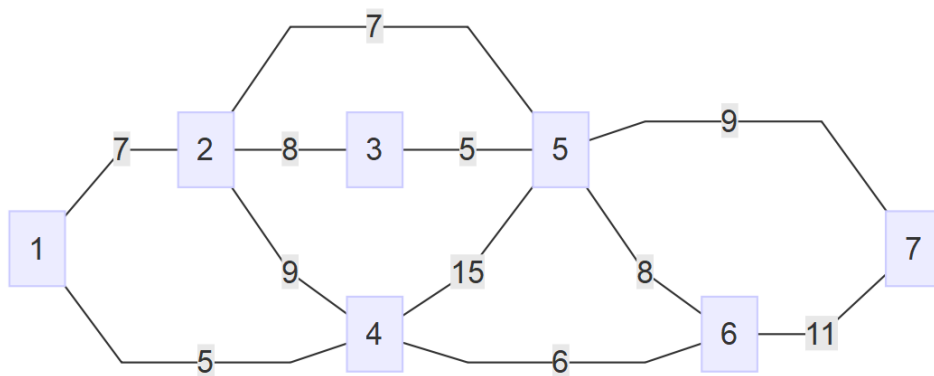
1   $G_{new}(V_{new}, E_{new}, W_{new}), V_{new} = V$ 
2  Sort  $W$ 
3  Set  $v_0 \in S$ 
4  while  $\exists u, v \in V_{new}, \text{Set}_u \neq \text{Set}_v$ 
5       $\langle u, v \rangle = \text{Dis}(W).Min$ 
6      if  $\text{Set}_u \neq \text{Set}_v$ 
7          Set  $W(u, v) \in W_{new}$ 
8          Merge ( $\text{Set}_u, \text{Set}_v$ )

```

第 4 章 实验结果

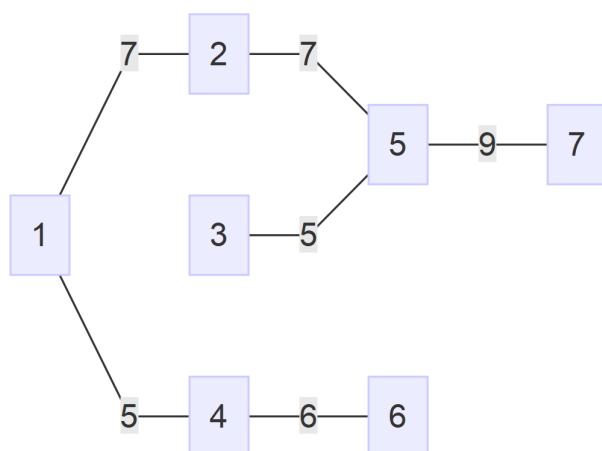
4.1 数据输入

输入如下无向带权图



4.2 结果输出

通过 *Prim* 算法或 *Kruskal* 算法可得到如下结果



第 5 章 实验分析

5.1 效率分析

Prim 算法 的时间复杂度取决于数据结构, 使用矩阵则为 $O(v^2)$, 邻接表为 $O(e \log_2 v)$ 。

Kruskal 算法 的时间复杂度 $O(e \log_2 e)$ 。

5.2 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。

5.3 算法比较

Prim 算法的时间复杂度为 $O(v^2)$ 或者 $O(e \log_2 v)$, 只和顶点的数目有关。而 *Kruskal* 算法的时间复杂度 $O(e \log_2 e)$, 只和边的数目有关。由此可见, *Kruskal* 算法适用于边稀疏的情形, 而 *Prim* 算法适用于边稠密的情形。

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 最小生成树

```

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 #include "Binaryheap.h" //堆的实现
13
14 //辅助工具
15 //边遍历工具
16 struct need_data {
17     int *near;
18     int *Len;
19     int *S;
20 };
21
22 static void update_edge(WGraph g, int src, int sink
23                        , int weight, void *data) {
24     assert(g);
25     if (!((struct need_data *) data)->S[sink]) {
26         if (weight < ((struct need_data *) data)->Len[sink]) {
27             ((struct need_data *) data)->Len[sink] = weight;
28             ((struct need_data *) data)->near[sink] = src;
29         }
30     }
31 }
32
33 WGraph Prim(WGraph g, int start) {
34     int i, j;
35     int vec_num = w_graph_vector_count(g);
36     WGraph res = w_graph_create(vec_num);
37     assert(res);
38     assert(start >= 0);
39     assert(start < vec_num);
40
41     struct need_data Data;
42     Data.S = calloc((size_t) vec_num, sizeof(int)); //逐步增加的新顶点集
43     Data.Len = calloc((size_t) vec_num, sizeof(int)); //到树的最小边
44     Data.near = calloc((size_t) vec_num, sizeof(int)); //最近临接顶点
45     for (i = 0; i != vec_num; ++i)
46         Data.Len[i] = INFINITY;
47     Data.S[start] = 1;
48     Data.Len[start] = 0;
49     int curr = start;
50

```

```

51     for (i = 1; i != vec_num; ++i) {
52         //通过 curr 更新各边最短值
53         w_graph_foreach(g, curr, update_edge, &Data);
54         int near_len = INFINITY;
55         for (j = 0; j != vec_num; ++j) {
56             if (!Data.S[j] && Data.Len[j] < near_len) {
57                 near_len = Data.Len[j];
58                 curr = j;
59             }
60         }
61         Data.S[curr] = 1;
62         w_graph_add_edge2(res, curr, Data.near[curr], Data.Len[curr]);
63     }
64     free(Data.near);
65     free(Data.Len);
66     free(Data.S);
67     return res;
68 }
69
70
71 typedef struct __Edge {
72     int from;
73     int to;
74     int len;
75 } Edge;
76
77 int edge_cmp(const void *a, const void *b) {
78     return ((Edge *) a)->len - ((Edge *) b)->len;
79 }
80
81
82 static void insertEdge(WGraph g, int src, int sink, int weight, void *heap) {
83     assert(g);
84     Edge *edge = malloc(sizeof(Edge));
85     edge->from = src;
86     edge->to = sink;
87     edge->len = weight;
88     Heap_insert((BinaryHeap) heap, edge, edge_cmp);
89 }
90
91 //并查集+堆优化 Kruskal 算法
92 WGraph Kruskal(WGraph g) {
93     int i, SetType;
94     int vec_num = w_graph_vector_count(g);
95     WGraph res = w_graph_create(vec_num);

```

```
96     BinaryHeap heap = Heap_create((size_t) w_graph_edge_count(g));
97     int *S = calloc((size_t) vec_num, sizeof(int));
98     assert(S);
99
100     for (i = 0; i != vec_num; ++i) {
101         w_graph_foreach(g, i, insertEdge, heap);
102         S[i] = i; // 森林
103     }
104     Edge *e = NULL;
105
106     while (w_graph_vector_count(res) != (w_graph_edge_count(res) / 2 + 1)) {
107         e = Heap_delete_key(heap, edge_cmp);
108         // 如果加入这条边不会形成圈
109         if (S[e->from] != S[e->to]) {
110             // 接收此边并合并
111             w_graph_add_edge2(res, e->from, e->to, e->len);
112             SetType = S[e->to];
113             for (i = 0; i != vec_num; ++i) {
114                 if (S[i] == SetType)
115                     S[i] = S[e->from];
116             }
117         }
118         free(e);
119     }
120     Heap_delete(heap);
121     return res;
122 }
```