

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。

关键字： *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	3
3.3	实现最小生成树算法	4
第 4 章	实验结果	5
4.1	数据输入	5
4.2	结果输出	5
第 5 章	实验分析	5
5.1	效率分析	5
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

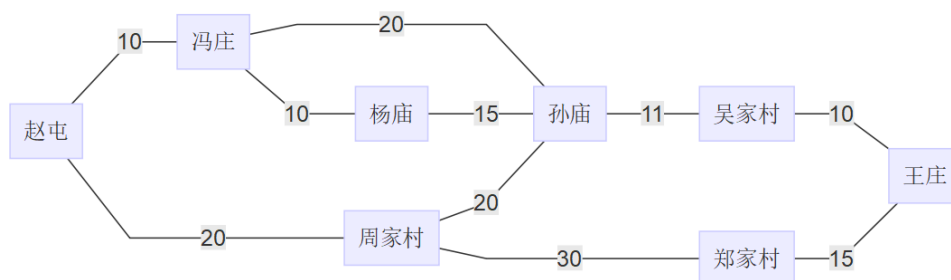
**实际问题** 我们常常会遇到求整体最短、最近、最省钱这些方面的问题，就比如下列问题：

- 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。

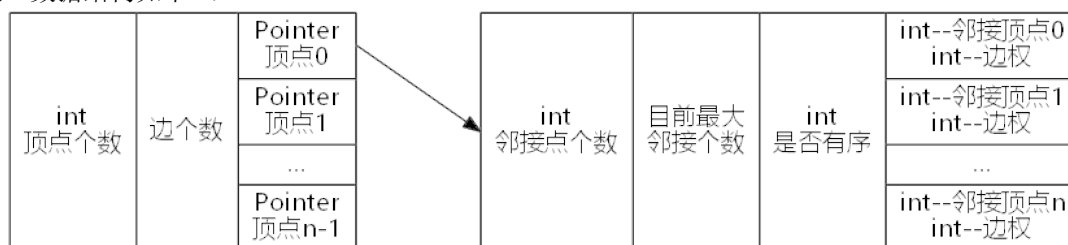
### 2.2.2 算法步骤

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, 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 实现最小生成树算法

- *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  $v \in V_{new}$ , Set  $\langle u, v \rangle \in E_{new}$ , Set  $W(u, v) \in W_{new}$

- *Kruskal* 算法实现（实际代码中使用了堆）

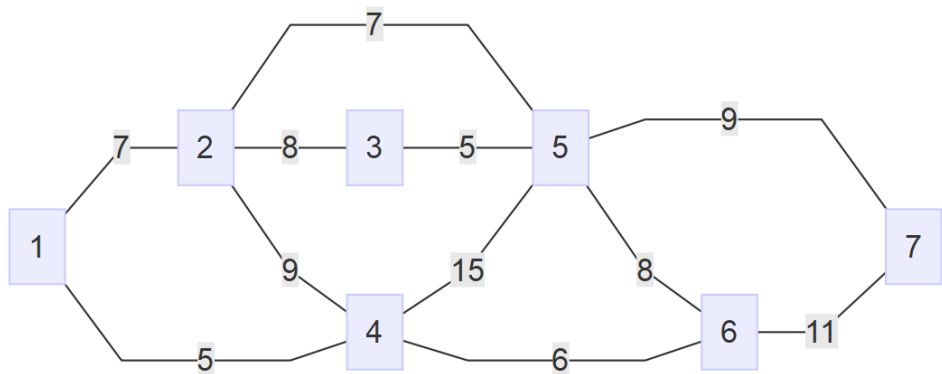
PRIM ALGORITHM( $G$ )

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

第 4 章 实验结果

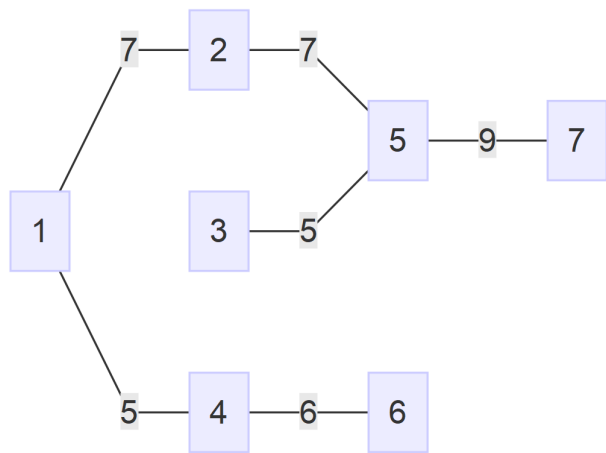
4.1 数据输入

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  //
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 最小生成树

```

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

```

```

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

```

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