

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 25 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	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	12
C 堆的实现	16

第一部分 正文

第 1 章 实验背景

1.1 实验目的

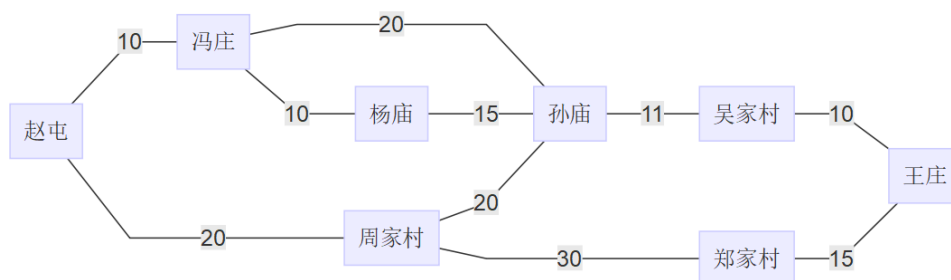
实际问题 我们常常会遇到求整体最短、最近、最省钱这些方面的问题，就比如下列问题：

- 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。

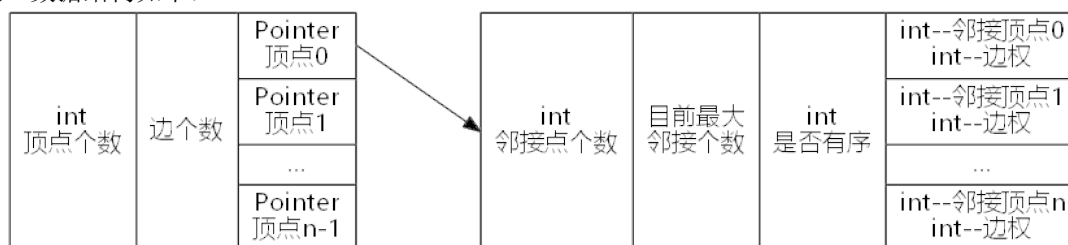
2.2.2 算法步骤

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:



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

- *Prim* 算法实现 (伪代码)

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 (伪代码)

KRUSKAL ALGORITHM(G)

```

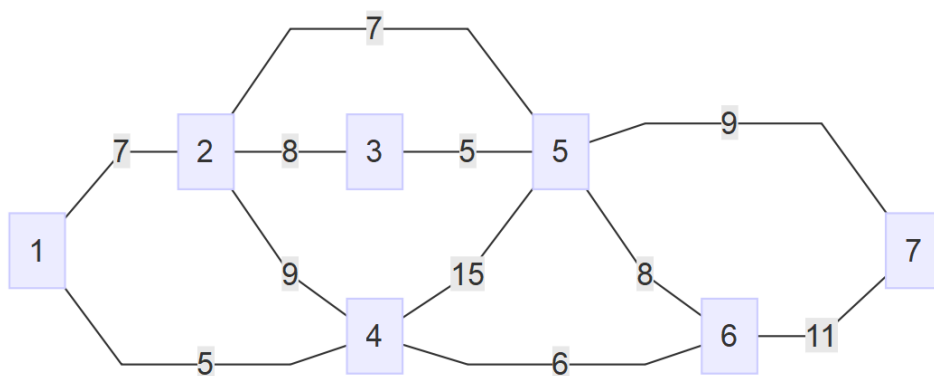
1   $G_{new}(V_{new}, E_{new}, W_{new}), V_{new} = V$ 
2  Sort  $W$ 
3  Set  $v_0 \in S$ 
4  while  $\exists u, v \in V_{new}, \text{Set}_u \neq \text{Set}_v$ 
5       $\langle u, v \rangle = \text{Dis}(W).Min$ 
6      if  $\text{Set}_u \neq \text{Set}_v$ 
7          Set  $W(u, v) \in W_{new}$ 
8   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

```

第 4 章 实验结果

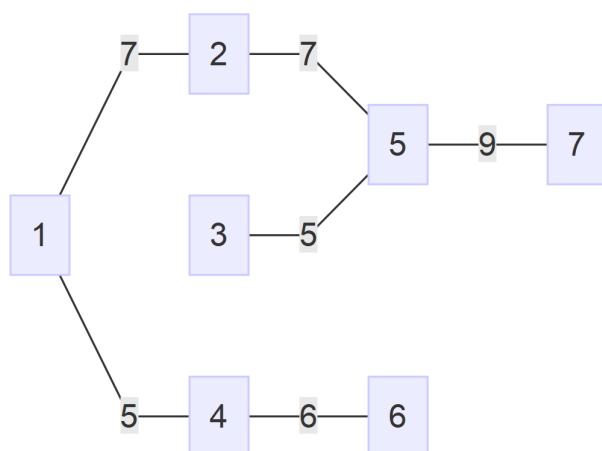
4.1 数据输入

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 // 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
12 // 图的定义
13 typedef struct w_graph *WGraph;
14
15 // 创建一个图
16 WGraph w_graph_create(int n);
17
18 // 删除一个图
19 void w_graph_destroy(WGraph);
20
21 // 添加一条边
22 void w_graph_add_edge(WGraph, int source, int sink, int weight);
23
24 // 顶点的数目
25 int w_graph_vector_count(WGraph);
26
27 // 边的数目
28 int w_graph_edge_count(WGraph);
29
30 // 顶点的度
31 int w_graph_out_degree(WGraph, int source);
32
33 // 两个顶点是否邻接
34 int w_graph_has_edge(WGraph, int source, int sink);
35
36 // 边的权
37 int w_graph_weight_edge(WGraph, int source, int sink);
38
39 // 提供遍历数据的接口
40 void w_graph_foreach(WGraph g, int source,
```



```

41     void (*f)(WGraph, int src, int sink, int weight, void *),
42     void *data);
43
44 #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邻接矩阵
15
16
17 typedef struct __list {
18     int vec;           //邻接顶点
19     int weight;        //权
20 } link_list;
21
22 struct w_graph {
23     int n;             //顶点个数
24     int m;             //边个数
25     struct successors {
26         int d;         //邻接点个数
27         int len;       //最大邻接点个数
28         char is_sorted; //
29         link_list list[1]; //邻接列表
30     } *v_list[1];
31 };
32
33
34 //创建一个顶点从 0 ~ n-1 的带权图
35 WGraph w_graph_create(int n) {
36     WGraph g;
37     int i;
38
39     g = malloc(sizeof(struct w_graph) +
40               sizeof(struct successors *) * (n - 1));

```

```

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

```

```

86     g->v_list[u]->d++;
87
88     g->v_list[u]->is_sorted = 0;
89
90     //边数+1
91     g->m++;
92 }
93
94 //返回顶点个数
95 int w_graph_vector_count(WGraph g) {
96     return g->n;
97 }
98
99 //返回边个数
100 int w_graph_edge_count(WGraph g) {
101     return g->m;
102 }
103
104 //返回顶点的度
105 int w_graph_out_degree(WGraph g, int source) {
106     assert(source >= 0);
107     assert(source < g->n);
108
109     return g->v_list[source]->d;
110 }
111
112 //是否需要进行二分搜索和排序
113 #define BSEARCH_THRESHOLD (10)
114
115 static int list_cmp(const void *a, const void *b) {
116     return ((const link_list *) a)->
117         vec - ((const link_list *) b)->vec;
118 }
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  // 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
9 #include "WeightGraph.h"
10
11 // tools
12 // 将图打印出来
13 void w_graph_show(WGraph);
14
15 // 将图成 Graphviz 的格式输出文件以便于生成图片
16 void w_graph_show_dot(WGraph, char path[]);
17
18 // 封装：由于是无权图，一次添加二条边
19 void w_graph_add_edge2(WGraph, int source, int sink, int weight);
20
21 // 求最小生成树
22 // Prim 算法
23 WGraph Prim(WGraph, int start);
24
25 // Kruskal 算法 (其中用到了堆)
26 WGraph Kruskal(WGraph);
27
28 #endif //GRAPH_GRAPH_TOOLS_H

```

```

1 //
2 // Graph_tools.c
3 // Created by Along on 2017/5/13.
4 // https://github.com/AlongWY/Graph
5 //
6
7 #include <stddef.h>
8 #include <assert.h>
9 #include <malloc.h>
10 #include <stdio.h>
11 #include "WeightGraph.h"
12 #include "Graph_tools.h"
13 #include "Binaryheap.h" //堆的实现
14
15 //辅助工具
16 //边遍历工具
17 struct need_data {
18     int *near;
19     int *Len;
20     int *S;
21 };
22

```

```

23 static void update_edge(WGraph g, int src, int sink
24                          , int weight, void *data) {
25     assert(g);
26     if (!((struct need_data *) data)->S[sink]) {
27         if (weight < ((struct need_data *) data)->Len[sink]) {
28             ((struct need_data *) data)->Len[sink] = weight;
29             ((struct need_data *) data)->near[sink] = src;
30         }
31     }
32 }
33
34 WGraph Prim(WGraph g, int start) {
35     int i, j;
36     int vec_num = w_graph_vector_count(g);
37     WGraph res = w_graph_create(vec_num);
38     assert(res);
39     assert(start >= 0);
40     assert(start < vec_num);
41
42     struct need_data Data;
43     Data.S = calloc((size_t) vec_num, sizeof(int)); //逐步增加的新顶点集
44     Data.Len = calloc((size_t) vec_num, sizeof(int)); //到树的最小边
45     Data.near = calloc((size_t) vec_num, sizeof(int)); //最近临接顶点
46     for (i = 0; i != vec_num; ++i)
47         Data.Len[i] = INFINITY;
48     Data.S[start] = 1;
49     Data.Len[start] = 0;
50     int curr = start;
51
52     for (i = 1; i != vec_num; ++i) {
53         //通过curr更新各边最短值
54         w_graph_foreach(g, curr, update_edge, &Data);
55         int near_len = INFINITY;
56         for (j = 0; j != vec_num; ++j) {
57             if (!Data.S[j] && Data.Len[j] < near_len) {
58                 near_len = Data.Len[j];
59                 curr = j;
60             }
61         }
62         Data.S[curr] = 1;
63         w_graph_add_edge2(res, curr, Data.near[curr], Data.Len[curr]);
64     }
65     free(Data.near);
66     free(Data.Len);
67     free(Data.S);

```

```

68     return res;
69 }
70
71 // 存入堆的数据类型
72 typedef struct __Edge {
73     int from;
74     int to;
75     int len;
76 } Edge;
77
78 // 比较边的大小
79 int edge_cmp(const void *a, const void *b) {
80     return ((Edge *) a)->len - ((Edge *) b)->len;
81 }
82
83 // 初始时将各边插入堆中
84 static void insertEdge(WGraph g, int src, int sink,
85                       int weight, void *heap) {
86     assert(g);
87     Edge *edge = malloc(sizeof(Edge));
88     edge->from = src;
89     edge->to = sink;
90     edge->len = weight;
91     Heap_insert((BinaryHeap) heap, edge, edge_cmp);
92 }
93
94 // 并查集+堆优化 Kruskal 算法
95 WGraph Kruskal(WGraph g) {
96     int i, SetType;
97     int vec_num = w_graph_vector_count(g);
98     WGraph res = w_graph_create(vec_num);
99     BinaryHeap heap = Heap_create((size_t) w_graph_edge_count(g));
100     int *S = calloc((size_t) vec_num, sizeof(int));
101     assert(S);
102
103     for (i = 0; i != vec_num; ++i) {
104         w_graph_foreach(g, i, insertEdge, heap);
105         S[i] = i; // 森林
106     }
107     Edge *e = NULL;
108
109     while (w_graph_vector_count(res) != (w_graph_edge_count(res) / 2 + 1))
110     {
111         e = Heap_delete_key(heap, edge_cmp);
112         // 如果加入这条边不会形成圈

```



```

113         if (S[e->from] != S[e->to]) {
114             //接收此边并合并
115             w_graph_add_edge2(res, e->from, e->to, e->len);
116             SetType = S[e->to];
117             for (i = 0; i != vec_num; ++i) {
118                 if (S[i] == SetType)
119                     S[i] = S[e->from];
120             }
121         }
122         free(e);
123     }
124     Heap_delete(heap);
125     return res;
126 }

```

C 堆的实现

```

1 //
2 // Binaryheap.h
3 // 堆的简单实现
4 // Created by Along on 2017/5/14.
5 //
6
7 #ifndef GRAPH_BINARYHEAP_H
8 #define GRAPH_BINARYHEAP_H
9
10 // 堆的定义
11 typedef struct BinaryHeap *BinaryHeap;
12
13 // 创建一个堆
14 BinaryHeap Heap_create(size_t n);
15
16 // 堆的数据量
17 int Heap_size(BinaryHeap);
18
19 // 插入数据
20 void Heap_insert(BinaryHeap, void *key,
21                 int (*cmp)(const void *a, const void *b));
22
23 // 删除并取出堆顶元素
24 void *Heap_delete_key(BinaryHeap,
25                      int (*cmp)(const void *a, const void *b));
26
27 // 删除堆

```

```
28 void Heap_delete(BinaryHeap);
29
30 #endif //GRAPH_BINARYHEAP_H
```

```
1 //
2 // Binaryheap.c
3 // Created by Along on 2017/5/14.
4 //
5
6 #include <stddef.h>
7 #include <malloc.h>
8 #include <assert.h>
9 #include "Binaryheap.h"
10
11 // 堆的数据结构
12 struct BinaryHeap {
13     int n;
14     int len;
15     void *data[1];
16 };
17
18 // 创建堆
19 BinaryHeap Heap_create(size_t n) {
20     BinaryHeap h = malloc(n * sizeof(BinaryHeap) +
21                             sizeof(void *) * (n - 1));
22     assert(h);
23     h->n = 0;
24     h->len = (int) n;
25     return h;
26 }
27
28 //插入数据
29 void Heap_insert(BinaryHeap h, void *key, int (*cmp)
30                  (const void *, const void *)) {
31     assert(key);
32     //动态扩展堆大小
33     if (h->n >= h->len) {
34         h->len *= 2;
35         h = realloc(h, sizeof(BinaryHeap) + sizeof(void *) * (h->len - 1));
36     }
37
38     //上滤
39     int hole = ++h->n;
40     void *copy = key;
41
```

```

42     h->data[0] = copy;
43     for (; cmp(key, h->data[hole / 2]) < 0; hole /= 2) {
44         h->data[hole] = h->data[hole / 2];
45     }
46     h->data[hole] = h->data[0];
47     h->data[0] = NULL;
48 }
49
50 // 元素上滤，保证堆的平衡
51 static void percolateDown(BinaryHeap h, int hole,
52     int (*cmp)(const void *, const void *)) {
53     int child;
54     void *tmp = h->data[hole];
55     for (; hole * 2 <= h->n; hole = child) {
56         child = hole * 2;
57         if (child != h->n && cmp(h->data[child + 1], h->data[child]) < 0)
58             ++child;
59         if (cmp(h->data[child], tmp) < 0)
60             h->data[hole] = h->data[child];
61         else
62             break;
63     }
64     h->data[hole] = tmp;
65 }
66
67 // // 删除并取出堆顶元素
68 void *Heap_delete_key(BinaryHeap h, int (*cmp)
69     (const void *, const void *)) {
70     assert(h->n > 0);
71     void *res = h->data[1];
72
73     h->data[1] = h->data[h->n--];
74     percolateDown(h, 1, cmp);
75     return res;
76 }
77
78 // 堆的数据量
79 int Heap_size(BinaryHeap h) {
80     return h->n;
81 }
82
83 // 删除堆
84 void Heap_delete(BinaryHeap h) {
85     free(h);
86 }

```