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

第 1 章　实验背景

1.1　实验目的

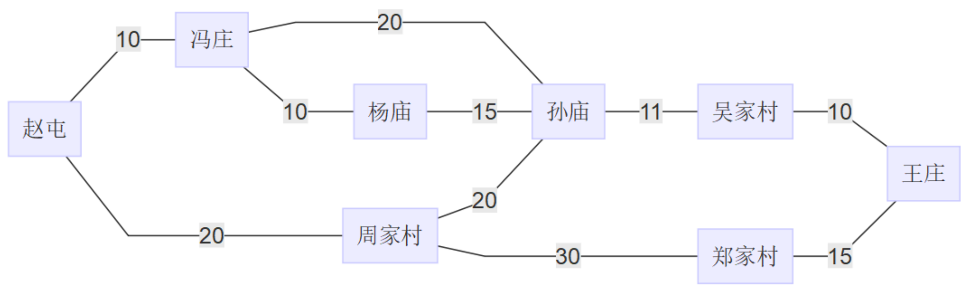
实际问题　我们常常会遇到求整体最短、最近、最省钱这些方面的问题，就比如下列问题：

- 镇子里要铺设自来水管道，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 $(V_0,V_1...V_{n-1})\in V$，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 $<u, v>$，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 $<u, v>$ 边加入集合 E_{new} 中，将 $W<u, v>$ 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。

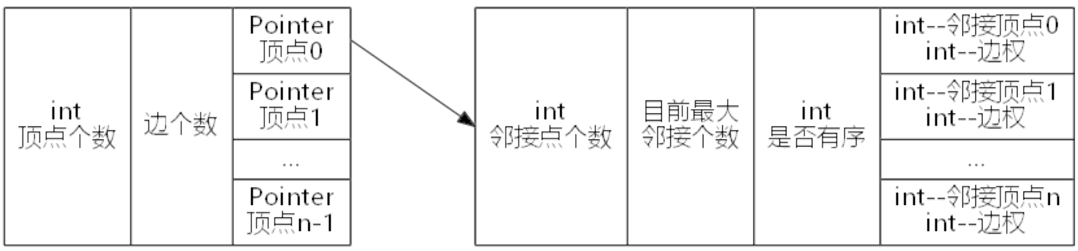
2.2.2　算法步骤

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 $0 \to n-1$ 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)

$v_{new} = v_0, G_{new}(V_{new}, E_{new}, W_{new})$
while $V_{new} \neq V$
　　$Get\ u \in V_{new}, v \notin V_{new}\ Make\ W(u,v)\ min$
　　$Set\ v \in V_{new},\ Set\ <u,v> \in E_{new},\ Set\ W(u,v) \in W_{new}$

- $Kruskal$ 算法实现（实际代码中使用了堆）

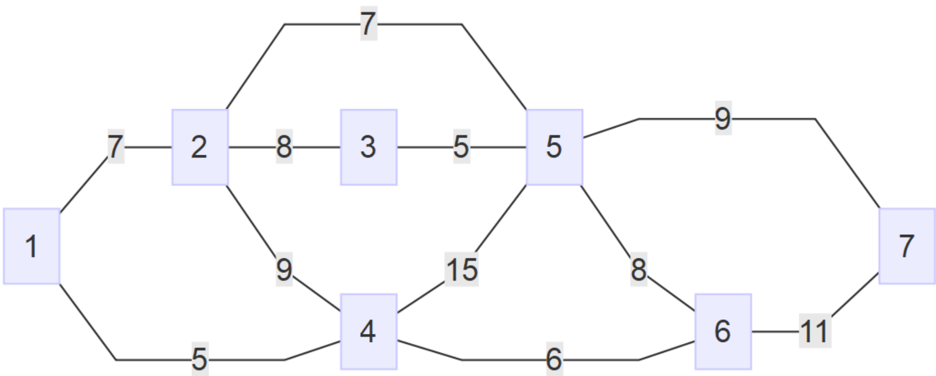
PRIM ALGORITHM(G)

$G_{new}(V_{new}, E_{new}, W_{new}), V_{new} = V$
$Sort\ W$
$Set\ v_0 \in S$
while $\exists u, v \in V_{new}, Set_u \neq Set_v$
　　$<u,v> = Dis(W).Min$
　　if $Set_u \neq Set_v$
　　　　$Set\ W(u,v) \in W_{new}$
　　　　$Merge\ (Set_u, Set_v)$

第 4 章　实验结果

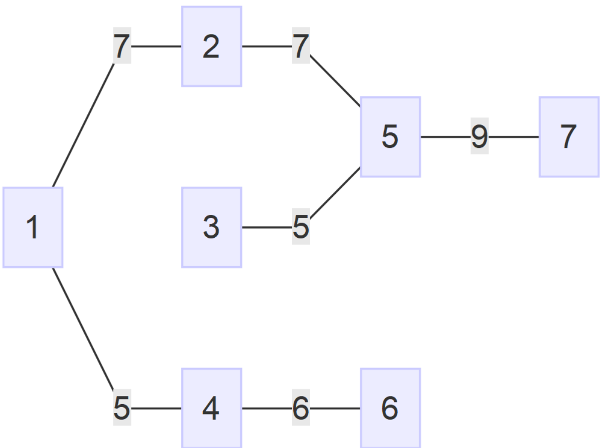
4.1　数据输入

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 $O(elog_2v)$。

$Kruskal$ **算法**　的时间复杂度 $O(elog_2e)$。

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 $O(elog_2v)$，只和顶点的数目有关。而$Kruskal$ 算法的时间复杂度 $O(elog_2e)$，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

```
//
// Created by Along on 2017/5/13.
//

#include <stdlib.h>
#include <assert.h>

#include "WeightGraph.h"

//基础带权图定义
//使用可变数组表示的临接矩阵


typedef struct _list {
    int vec;                //临接顶点
    int weight;             //权
} link_list;

struct w_graph {
    int n;                          //顶点个数
    int m;                          //边个数
    struct successors {
        int d;                      //临接点个数
        int len;                    //最大临接点个数
        char is_sorted;             //
        link_list list[1];                  //临接列表
    } *v_list[1];
};


//创建一个顶点从0 ~ n-1的带权图
WGraph w_graph_create(int n) {
    WGraph g;
    int i;

    g = malloc(sizeof(struct w_graph) +
    sizeof(struct successors *) * (n - 1));
    assert(g);

    g->n = n;
```

```
    g->m = 0;

    for (i = 0; i != n; i++) {
        g->v_list[i] = malloc(sizeof(struct successors));
        assert(g->v_list[i]);
        g->v_list[i]->d = 0;
        g->v_list[i]->len = 1;
        g->v_list[i]->is_sorted = 1;
    }

    return g;
}

//释放内存
void w_graph_destroy(WGraph g) {
    int i;

    for (i = 0; i != g->n; i++) {
        free(g->v_list[i]);
    };
    free(g);
}

//添加边和权
void w_graph_add_edge(WGraph g, int u, int v, int weight) {
    assert(u >= 0);
    assert(u < g->n);
    assert(v >= 0);
    assert(v < g->n);

    //是否需要增长list
    while (g->v_list[u]->d >= g->v_list[u]->len) {
        g->v_list[u]->len *= 2;
        g->v_list[u] =
                realloc(g->v_list[u], sizeof(struct successors) +
                 sizeof(link_list) * (g->v_list[u]->len - 1));
    }

    //添加新临接点
    g->v_list[u]->list[g->v_list[u]->d].vec = v;
    g->v_list[u]->list[g->v_list[u]->d].weight = weight;

    g->v_list[u]->d++;

    g->v_list[u]->is_sorted = 0;
```

```c

    //边数+1
    g->m++;
}

//返回顶点个数
int w_graph_vector_count(WGraph g) {
    return g->n;
}

//返回边个数
int w_graph_edge_count(WGraph g) {
    return g->m;
}

//返回顶点的度
int w_graph_out_degree(WGraph g, int source) {
    assert(source >= 0);
    assert(source < g->n);

    return g->v_list[source]->d;
}

//是否需要进行二分搜索和排序
#define BSEARCH_THRESHOLD (10)

static int list_cmp(const void *a, const void *b) {
    return ((const link_list *) a)->
    vec - ((const link_list *) b)->vec;
}


#include <stdio.h>

//二者之间有边则返回1
int w_graph_has_edge(WGraph g, int source, int sink) {
    int i;

    assert(source >= 0);
    assert(source < g->n);
    assert(sink >= 0);
    assert(sink < g->n);

    if (w_graph_out_degree(g, source) >= BSEARCH_THRESHOLD) {
        //确保已经被排序
```

```
        if (!g->v_list[source]->is_sorted) {
            qsort(g->v_list[source]->list,
                  g->v_list[source]->d,
                  sizeof(link_list),
                  list_cmp);
        }
        //使用二分查找
        link_list to_find;
        to_find.vec = sink;
        to_find.weight = 0;

        return bsearch(&to_find,
                       g->v_list[source]->list,
                       g->v_list[source]->d,
                       sizeof(link_list),
                       list_cmp) != 0;
    } else {
        //数据量很少，直接遍历
        int vec_degree = g->v_list[source]->d;
        for (i = 0; i != vec_degree; i++) {
            if (g->v_list[source]->list[i].vec == sink) {
                return 1;
            }
        }
    }
    return 0;
}

//返回权
int w_graph_weight_edge(WGraph g, int source, int sink) {
    int i;

    assert(source >= 0);
    assert(source < g->n);
    assert(sink >= 0);
    assert(sink < g->n);

    if (w_graph_out_degree(g, source) >= BSEARCH_THRESHOLD) {
        //确保已经被排序
        if (!g->v_list[source]->is_sorted) {
            qsort(g->v_list[source]->list,
                  g->v_list[source]->d,
                  sizeof(link_list),
                  list_cmp);
        }
```

```
        //使用二分查找
        link_list to_find;
        to_find.vec = sink;
        to_find.weight = 0;
        link_list *res = bsearch(&to_find,
                                 g->v_list[source]->list,
                                 g->v_list[source]->d,
                                 sizeof(link_list),
                                 list_cmp);
        return res->weight;
    } else {
        //数据量很少，直接遍历
        for (i = 0; i != g->v_list[source]->d; i++) {
            if (g->v_list[source]->list[i].vec == sink) {
                int res = g->v_list[source]->list[i].weight;
                return res;
            }
        }
        return INFINITY;
    }
}

//提供数据 遍历接口
void w_graph_foreach(WGraph g, int source,
void (*f)(WGraph, int, int, int, void *), void *data) {
    int i;

    assert(source >= 0);
    assert(source < g->n);

    for (i = 0; i != g->v_list[source]->d; ++i) {
        f(g, source, g->v_list[source]->list[i].vec,
          g->v_list[source]->list[i].weight, data);
    }
}
```

B 最小生成树

```
//
// Created by Along on 2017/5/13.
//

#include <stddef.h>
#include <assert.h>
```

```c
#include <malloc.h>
#include <stdio.h>
#include "WeightGraph.h"
#include "Graph_tools.h"

//辅助工具
//边遍历工具
struct need_data {
    int *near;
    int *Len;
    int *S;
};

static void update_edge(WGraph g, int src, int sink, int weight, void *data) {
    assert(g);
    if (!((struct need_data *) data)->S[sink]) {
        if (weight < ((struct need_data *) data)->Len[sink]) {
            ((struct need_data *) data)->Len[sink] = weight;
            ((struct need_data *) data)->near[sink] = src;
        }
    }
}

WGraph Prim(WGraph g, int start) {
    int i, j;
    int vec_num = w_graph_vector_count(g);
    WGraph res = w_graph_create(vec_num);
    assert(res);
    assert(start >= 0);
    assert(start < vec_num);

    struct need_data Data;
    Data.S = calloc((size_t) vec_num, sizeof(int));      //逐步增加的新顶点集
    Data.Len = calloc((size_t) vec_num, sizeof(int));    //到树的最小边
    Data.near = calloc((size_t) vec_num, sizeof(int));   //最近临接顶点
    for (i = 0; i != vec_num; ++i)
        Data.Len[i] = INFINITY;
    Data.S[start] = 1;
    Data.Len[start] = 0;
    int curr = start;

    for (i = 1; i != vec_num; ++i) {
        w_graph_foreach(g, curr, update_edge, &Data);    //通过curr更新各边最短值
        int near_len = INFINITY;
        for (j = 0; j != vec_num; ++j) {
```

```c
                if (!Data.S[j] && Data.Len[j] < near_len) {
                    near_len = Data.Len[j];
                    curr = j;
                }
            }
            Data.S[curr] = 1;
            w_graph_add_edge2(res, curr, Data.near[curr], Data.Len[curr]);
        }
        free(Data.near);
        free(Data.Len);
        free(Data.S);
        return res;
}


typedef struct _Edge {
        int from;
        int to;
        int len;
} Edge;

int edge_cmp(const void *a, const void *b) {
        return ((Edge *) a)->len - ((Edge *) b)->len;
}


static void insertEdge(WGraph g, int src, int sink, int weight, void *heap) {
        assert(g);
        Edge *edge = malloc(sizeof(Edge));
        edge->from = src;
        edge->to = sink;
        edge->len = weight;
        Heap_insert((BinaryHeap) heap, edge, edge_cmp);
}

//并查集+堆优化Kruskal算法
WGraph Kruskal(WGraph g) {
        int i, SetType;
        int vec_num = w_graph_vector_count(g);
        WGraph res = w_graph_create(vec_num);
        BinaryHeap heap = Heap_create((size_t) w_graph_edge_count(g));
        int *S = calloc((size_t) vec_num, sizeof(int));
        assert(S);

        for (i = 0; i != vec_num; ++i) {
```

```
        w_graph_foreach(g, i, insertEdge, heap);
        S[i] = i;                                   //森林
    }
    Edge *e = NULL;

    while (w_graph_vector_count(res) != (w_graph_edge_count(res) / 2 + 1)) {
        e = Heap_delete_key(heap, edge_cmp);
        //如果加入这条边不会形成圈
        if (S[e->from] != S[e->to]) {
            //接收此边并合并
            w_graph_add_edge2(res, e->from, e->to, e->len);
            SetType = S[e->to];
            for (i = 0; i != vec_num; ++i) {
                if (S[i] == SetType)
                    S[i] = S[e->from];
            }
        }
        free(e);
    }
    Heap_delete(heap);
    return res;
}
```