云南大学数学与统计学院
《运筹学通论实验》上机实践报告

课程名称：运筹学实验	**年级：**2015 级	**上机实践成绩：**
指导教师：李建平	**姓名：**刘鹏	**专业：**信息与计算科学
上机实践名称：Prim 算法求图的支撑树与联通子图	**学号：**20151910042	**上机实践日期：**2018-07-08
上机实践编号：4	**组号：**	

一、 实验目的

1. 学习 Prim 算法的使用；
2. 了解 Prim 算法作为贪心算法能达最优的理论证明。

二、 实验内容

1. 写出 Prim（反圈法）算法[1]的伪码描述[2]；
2. 用 C 语言[3]编程实现 Prim 算法，找出一幅图的最小生成树；

三、 实验平台

Microsoft Windows 10 Pro Workstation 1803；
Microsoft Visual Studio 2017 Enterprise。

四、 算法设计

4.1 算法背景

普里姆算法（Prim 算法），图论中的一种算法，可在加权联通图里搜索最小生成树。意即由此算法搜索到的边子集所构成的树中，不但包括了连通图里的所有顶点，且其所有边的权值之和为最小。该算法于 1930 年由杰克数学家沃伊捷赫·亚尔尼克发现；并在 1957 年由美国计算机科学家罗伯特·普里姆独立发现；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。Prim 算法的工作原理与 Dijkstra 的最短路径算法相似。本策略属于贪心策略，因为每一步所加入的边都必须是使得树的总权重增加量最小的边。

4.2 时间复杂度

这个算法的时间复杂度与图的实现方法有关。设图$\boldsymbol{G}=(\boldsymbol{V},\ \boldsymbol{E})$，其中$\boldsymbol{V}$是图的所有节点的集合，$\boldsymbol{E}$是图的所有边的集合。图是由如果采用比较低级的邻接矩阵实现，那么 Prim 算法的时间复杂度是$O(|\boldsymbol{V}|^2)$；如果用二叉堆、邻接表来实现，那么时间复杂度是$O\big((|\boldsymbol{V}|+|\boldsymbol{E}|)\log|\boldsymbol{V}|\big)=O(|\boldsymbol{E}|\log|\boldsymbol{V}|)$；如果用斐波那契堆来实现复杂度可以降低至$O(|\boldsymbol{E}|+|\boldsymbol{V}|\log|\boldsymbol{V}|)$

五、 程序代码

5.1 程序描述

综合考虑了实现难易程度与算法的时间复杂度，我决定采用邻接映射（Adjacent Map）来做为主要的数据结构。邻接映射的主要思想是用哈希表这种快速查找表来代替遍历带来的高时间复杂度，本次实验的结构在仅仅采用哈希表的基础上进行了改进，把所有已经被占据的数组下标记录下来，放入一个动态数组里，为以后查找所有与某个节点相邻的节点或者找寻连接到该点的无向边提供便利。

邻接映射本身就是一张哈希表$\boldsymbol{M}$，key 是 vertex，value 是另外的一个子哈希表$\boldsymbol{m}$，在子哈希表里面，key 是与 vertex 相邻的 vertex（并且是有向的，只能从前者到后者），value 是两个 vertex 中间的边。为了能够在一个结构里实现有向图（directed graph）与无向图（undirected graph），这里采用两个大哈希表来实现有向图，其中一个记录的是$v_1 \to v_2$这种类型，第二个记录$v_1 \leftarrow v_2$这种类型，两者恰好反向。由于无向图的两种上述类型必然是同时存在的，而有向图则不然，所以可以通过这种方式进行处理有向图。由于普里姆算法主要关心无向图，所以代码中并没有实现针对有向图的算法。

普里姆算法实现的具体思路如下。

```
Algorithm   PRIM-MST
Input       无向图$\boldsymbol{G}$，初始节点$s$

Output      图$\boldsymbol{G}$的最小生成树
Begin
Step 1      f 存储一个节点集合$\boldsymbol{V}$，$\boldsymbol{V}$的初值为起点组成的单元素集合

Step 2      初始化一个图$\boldsymbol{G}$，$\boldsymbol{G}$的初值为空集，把$\boldsymbol{V}$中所有的节点添加到$\boldsymbol{G}$中

Step 3      通过对$\boldsymbol{V}$中所有的节点分别进行两次哈希表查询：第一次对$\boldsymbol{G}_{\text{in}}$进行查询，得到一个子哈希
            表；第二次对第一步得到的子哈希表进行查询，得到这个子哈希表中所有的边。每次查询
            的时间复杂度均为$O(1)$。这样就找到与集合$\boldsymbol{V}$中所有节点分别相邻的所有的边，将这些边
            添加到一个临时边集合$\boldsymbol{E}$里。

Step 4      对于边集合$\boldsymbol{E}$，进行如下处理：对所有终点不在$\boldsymbol{V}$里的边而言，将之权重添加到一个临时空
            泛型动态数组里，对这个数组进行排序，找到最小权重值；之后从所有的边中，任意挑选
            一个权重为最小值的边，将其终点添加到节点集合$\boldsymbol{V}$以及图$\boldsymbol{G}$中，然后把这条边也添加到图
            $\boldsymbol{G}$中。如果图$\boldsymbol{G}$的节点数量等于输入的图的节点数量，GOTO End，返回值为$\boldsymbol{G}$；否则，返
            回 Step 3
End
```

5.2 程序代码示例

源代码数量太多，这里仅仅给出核心文件 Prim.cpp 的代码，其他程序在本节报告的附录中给出，工程文件参看我的 GitHub 链接。

```
/*
* Copyright (c) 2018, Liu Peng, School of Mathematics and Statistics, YNU
* Apache License.
*
* 文件名称：Prim.cpp
* 文件标识：见配置管理计划书
* 摘 要：Prim 算法
*
* 当前版本：1.0
* 作 者：刘鹏
* 创建日期：2018 年 6 月 25 日
* 完成日期：2018 年 6 月 26 日
*
* 取代版本：
* 原作者 ：刘鹏
* 完成日期：
*/

/*
* A function based on Prim Algorithm to find the minimal spanning tree
* of a undirected graph.
*/

#include "Graph.h"

// Get the Minimal Spanning Tree of a connected graph.
// If the graph is not connected, exception would be raised.
Graph *MST_Prim_Jarnik(Graph *g, Vertex start, Function f) {
    map_t arg;      // make up the parameters hashmap_get fuc needing
    if (MAP_MISSING == hashmap_get(g, (char *)start, &arg)) {
        printf("fatal Error: bad input!\n");
        return NULL;
    }

    Graph *ans = Graph_init(false);            // the answer of this algorithm
    Dynamic_Array *V = Dynamic_Array_init();    // set of vertices have been found

    Dynamic_Array_append(V, (any)start);
    Graph_insert_vertex(ans, start);

    int i;
    int j;
    int Length_of_Graph_in = hashmap_length(g->outgoing);
    Vertex temp;
    map_t temp_map;
    Dynamic_Array *temp_Edge = Dynamic_Array_init();        // set of edges connected with V

    // Generally, loop will stop while len(V) = len(g).
    // If g is not connected, error will be raised.
    while (V->n < Length_of_Graph_in) {
        for (i = 1; i <= V->n; i++) {
            temp = (Vertex)Dynamic_Array_get_Element(V, i);
            temp_map = Graph_get_adjacent_Vertices(g, temp);        // submap
            Dynamic_Array *index = hashmap_used_index(temp_map);    // used slots of the submap

            Edge *tmp;
            int change_memo = temp_Edge->n;
            for (j = 1; j <= index->n; j++) {
                int addr = (int)Dynamic_Array_get_Element(index, j);
                tmp = (Edge *)hashmap_select(temp_map, addr);

                // only if the destination of Edge(tmp) is not
                // included in the map(G), the appending
                // operation can be done.
                if (hashmap_get(ans->outgoing, (char *)tmp->destination, (void **)&arg)
                    == MAP_MISSING) {
                    Dynamic_Array_append(temp_Edge, (any)tmp);
                }
            }
        }

        // find the minimal value
        int min_location = Dynamic_Array_min(temp_Edge, f_get_double);
        Edge *min = (Edge *)Dynamic_Array_get_Element(temp_Edge, min_location);
        temp = min->destination;
        Dynamic_Array_append(V, temp);
        Graph_insert_vertex(ans, temp);
        Graph_insert_edge(ans, min->origin, temp, min->element);
    }
    return ans;
}
```

程序代码 1

5.3 具体可运行的代码

```
/*
 * Copyright (c) 2018, Liu Peng, School of Mathematics and Statistics, YNU
 * Apache License.
 *
 * 文件名称：Source.cpp
 * 文件标识：见配置管理计划书
 * 摘  要：测试 Prim 算法
 *
 * 当前版本：1.0
 * 作  者: 刘鹏
 * 创建日期: 2018 年 7 月 9 日
 * 完成日期: 2018 年 7 月 9 日
 *
 * 取代版本: 0.9
 * 原作者 : 刘鹏
 * 完成日期:
 */


 // C / C++ program for Prim's MST for adjacency list representation of graph

 #include <stdio.h>
 #include <stdlib.h>
 #include <limits.h>
 #include <time.h>

 // A structure to represent a node in adjacency list
 struct AdjListNode {
     int dest;
     int weight;
     struct AdjListNode* next;
 };

 // A structure to represent an adjacency liat
 struct AdjList {
     struct AdjListNode *head; // pointer to head node of list
 };

 // A structure to represent a graph. A graph is an array of adjacency lists.
 // Size of array will be V (number of vertices in graph)
 struct Graph {
     int V;
     struct AdjList* array;
 };

 // A utility function to create a new adjacency list node
 struct AdjListNode* newAdjListNode(int dest, int weight) {
     struct AdjListNode* newNode =
         (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
     newNode->dest = dest;
     newNode->weight = weight;
     newNode->next = NULL;
     return newNode;
 }

 // A utility function that creates a graph of V vertices
 struct Graph* createGraph(int V) {
     struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
     graph->V = V;

     // Create an array of adjacency lists. Size of array will be V
     graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

     // Initialize each adjacency list as empty by making head as NULL
     for (int i = 0; i < V; ++i)
         graph->array[i].head = NULL;

     return graph;
 }

 // Adds an edge to an undirected graph
 void addEdge(struct Graph* graph, int src, int dest, int weight) {
     // Add an edge from src to dest. A new node is added to the adjacency
     // list of src. The node is added at the begining
     struct AdjListNode* newNode = newAdjListNode(dest, weight);
     newNode->next = graph->array[src].head;
     graph->array[src].head = newNode;

     // Since graph is undirected, add an edge from dest to src also
     newNode = newAdjListNode(src, weight);
     newNode->next = graph->array[dest].head;
     graph->array[dest].head = newNode;
 }

 // Structure to represent a min heap node
 struct MinHeapNode {
     int v;
     int key;
 };

 // Structure to represent a min heap
 struct MinHeap {
     int size;    // Number of heap nodes present currently
     int capacity; // Capacity of min heap
     int *pos;    // This is needed for decreaseKey()
     struct MinHeapNode **array;
 };

 // A utility function to create a new Min Heap Node
 struct MinHeapNode* newMinHeapNode(int v, int key) {
     struct MinHeapNode* minHeapNode =
         (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
     minHeapNode->v = v;
     minHeapNode->key = key;
     return minHeapNode;
 }

```

```c
// A utilit function to create a Min Heap
struct MinHeap* createMinHeap(int capacity) {
    struct MinHeap* minHeap =
        (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->pos = (int *)malloc(capacity * sizeof(int));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array =
        (struct MinHeapNode**) malloc(capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}

// A utility function to swap two nodes of min heap. Needed for min heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b) {
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx) {
    int smallest, left, right;
    smallest = idx;
    left = 2 * idx + 1;
    right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->array[left]->key < minHeap->array[smallest]->key)
        smallest = left;

    if (right < minHeap->size &&
        minHeap->array[right]->key < minHeap->array[smallest]->key)
        smallest = right;

    if (smallest != idx) {
        // The nodes to be swapped in min heap
        struct MinHeapNode *smallestNode = minHeap->array[smallest];
        struct MinHeapNode *idxNode = minHeap->array[idx];

        // Swap positions
        minHeap->pos[smallestNode->v] = idx;
        minHeap->pos[idxNode->v] = smallest;

        // Swap nodes
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if the given minHeap is ampty or not
int isEmpty(struct MinHeap* minHeap) {
    return minHeap->size == 0;
}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap) {
    if (isEmpty(minHeap))
        return NULL;

    // Store the root node
    struct MinHeapNode* root = minHeap->array[0];

    // Replace root node with last node
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    // Update position of last node
    minHeap->pos[root->v] = minHeap->size - 1;
    minHeap->pos[lastNode->v] = 0;

    // Reduce heap size and heapify root
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

// Function to decreasy key value of a given vertex v. This function
// uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int key) {
    // Get the index of v in heap array
    int i = minHeap->pos[v];

    // Get the node and update its key value
    minHeap->array[i]->key = key;

    // Travel up while the complete tree is not hepified.
    // This is a O(Logn) loop
    while (i && minHeap->array[i]->key < minHeap->array[(i - 1) / 2]->key) {
        // Swap this node with its parent
        minHeap->pos[minHeap->array[i]->v] = (i - 1) / 2;
        minHeap->pos[minHeap->array[(i - 1) / 2]->v] = i;
        swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

        // move to parent index
```

```c
        i = (i - 1) / 2;
    }
}

// A utility function to check if a given vertex
// 'v' is in min heap or not
bool isInMinHeap(struct MinHeap *minHeap, int v) {
    if (minHeap->pos[v] < minHeap->size)
        return true;
    return false;
}

// A utility function used to print the constructed MST
void printArr(int arr[], int n) {
    for (int i = 1; i < n; ++i)
        printf("%d - %d\n", arr[i], i);
}

// The main function that constructs Minimum Spanning Tree (MST)
// using Prim's algorithm
void PrimMST(struct Graph* graph) {
    int V = graph->V;   // Get the number of vertices in graph

    // Array to store constructed MST
    int *parent = (int *)calloc(V, sizeof(int));

    // Key values used to pick minimum weight edge in cut
    int *key = (int *)calloc(V, sizeof(int));

    // minHeap represents set E
    struct MinHeap* minHeap = createMinHeap(V);

    // Initialize min heap with all vertices. Key value of
    // all vertices (except 0th vertex) is initially infinite
    int v;
    for (v = 1; v < V; ++v) {
        parent[v] = -1;
        key[v] = INT_MAX;
        minHeap->array[v] = newMinHeapNode(v, key[v]);
        minHeap->pos[v] = v;
    }

    // Make key value of 0th vertex as 0 so that it
    // is extracted first
    key[0] = 0;
    minHeap->array[0] = newMinHeapNode(0, key[0]);
    minHeap->pos[0] = 0;

    // Initially size of min heap is equal to V
    minHeap->size = V;

    // In the followin loop, min heap contains all nodes
    // not yet added to MST.
    while (!isEmpty(minHeap)) {
        // Extract the vertex with minimum key value
        struct MinHeapNode* minHeapNode = extractMin(minHeap);
        int u = minHeapNode->v; // Store the extracted vertex number

                                // Traverse through all adjacent vertices of u (the extracted
                                // vertex) and update their key values
        struct AdjListNode* pCrawl = graph->array[u].head;
        while (pCrawl != NULL) {
            int v = pCrawl->dest;

            // If v is not yet included in MST and weight of u-v is
            // less than key value of v, then update key value and
            // parent of v
            if (isInMinHeap(minHeap, v) && pCrawl->weight < key[v]) {
                key[v] = pCrawl->weight;
                parent[v] = u;
                decreaseKey(minHeap, v, key[v]);
            }
            pCrawl = pCrawl->next;
        }
    }

    // print edges of MST
    // printArr(parent, V);
}

// Driver program to test above functions
int main() {
    // Let us create the graph given in above fugure
    FILE *file_input;
    file_input = fopen("../dataset/grafo-esparso-10000.txt", "r");
    struct Graph* graph;
    char a[999], b[999], c[999], v[999];
    if (file_input) {
        fscanf(file_input, "%s\n", v);
        graph = createGraph(atoi(v));
        while ((fscanf(file_input, "%s %s %s\n", a, b, c)) != EOF) {
            addEdge(graph, atoi(a), atoi(b), atoi(c));
        }
    }
    fclose(file_input);

    int i;
    for (i = 0; i < 173; ++i) {
        clock_t tStart = clock();
        PrimMST(graph);
        printf("Time taken: %fs\n", (double)(clock() - tStart) / CLOCKS_PER_SEC, i + 1);
    }

    return 0;
}
```

程序代码 2

六、 运行结果

6.1 运行结果

考虑到这里并不是用邻接矩阵做的，所以很难输出一个图。代码 5.2.1 就做了 173 便 Prim 算法，查看在一个很复杂的图里运行时间如何。

七、 实验体会

为了练习使用 Map 结构与了解 Prim 算法，本次实验采用的数据结构为邻接映射，核心的映射实现为 CRC32 哈希函数，基于这个函数，完成了 HashMap 的结构与一批相关函数，然后基于这些成果，进一步完成了泛型有向图、无向图结构。

遗憾的是时间有限加上我本人水平有限，程序不能做到尽善尽美，而且目前版本还没有一个良好的图输出程序，就像是采取最简单的邻接矩阵，输出也只是一堆数字，与实际的图没有视觉联系。接下来我可能花点时间完善一下这个输出。

感谢开源社区[4]提供的高质量代码。

八、 参考文献

[1] HILLIER F S, LIEBERMAN G J. 运筹学导论 [M]. 9th ed. 北京: 清华大学出版社, 2010.

[2] CORMEN T H, LEISERSON C E, RIVEST R L, et al. 算法导论 [M]. 3rd ed. 北京: 机械工业出版社, 2013.

[3] **林锐**. 高质量 C++/C 编程指南 [M]. 1.0 ed., 2001.

[4] https://github.com/jeffvfa/projeto-CEXP