图算法

图的邻接表表示：

```
std::map<int, std::list<int>>  graph;
```

对图问题，首先要有一个`api adj`:

```
// 输入节点s,返回节点s的相邻节点
std::list<int> adj(int s) {
  return graph[s];
}
```

对于加权图，需要知道其权值，可以抽象出`weight`方法：

```
// 返回节点from到节点to的权重
int weight(int from, int to);
```

从二叉树层序遍历推到`Dijkstra`算法

二叉树层序遍历框架：

```
void levelTravel(TreeNode *root) {
  if(root == nullptr) {
    return ;
  }

  std::list<TreeNode *> que;
  que.push_back(root);

  int level = 1;

  // 从上到下遍历二叉树节点
  while(!que.size()){
    int sz = que.size(); // 当前层的节点数，当前层的节点已全部写入que

    // 遍历每层的节点
    for(int i=0;i<sz;i++){
      TreeNode *t = que.front();
      que.pop_front();

      if(t->left != nullptr) {
        que.push_back(t->left);
      }
```

```
      if(t->right != nullptr) {
        que.push_back(t->right);
      }
    }
    level++;
  }
}
```

从二叉树层序遍历可以得到多叉树层序遍历：

```
void levelTraveser(mTreeNode *root){
  if(nullptr == root) {
    return ;
  }
  std::list<mTreeNode *> que;
  que.push_back(root);
  int level = 1;

  // 从上到下
  while(que.size()) {
    int sz = que.size();
    for(int i = 0; i < sz; i++) {
      mTreeNode *t = que.front();
      que.pop_front();

      for(auto iter : t->children){
        que.push_back(iter);
      }
    }

    level++;
  }

}
```

多叉树可以推导出BFS框架：

```
void bfs(node s) {
  std::list<node> que;
  std::set<node> visited; // 防止走回头路

  que.push_back(s); // 加入起点
  visited.insert(s);

  while(que.size()) {
    int sz = que.size();
```

```
      for(int i =0;i<sz;i++) {
        Node t = que.front();
        que.pop_front();

        // t 的相邻节点如队列
        for(auto iter : adj(t)){
          if(visited.count(iter) == 0) {
            que.push_back(iter);
            visited.insert(iter);
          }
        }
      }
      step++;
    }

  }
```

`Dijkstra`算法框架：

```
#include <climits>
#include <list>
#include <map>
#include <queue>
#include <set>
#include <vector>

class graph {
public:
  // 输入一幅图graph,和一个起点start,计算start到其他节点的距离
  std::vector<int> dijkstra(int start) {
    // 途中节点个数
    int v = graph.size();
    // 记录最短路径的权重数组
    std::vector<int> dis = std::vector<int>(v, INT_MAX);

    // base case
    dis[0] = 0;

    // 按照distFromStart排序的小根堆
    std::priority_queue<state, std::vector<state>, greator> heap;

    // 从起点开始进行BFS
    heap.push(state(start, 0));
    while (heap.size()) {
      state curstate = heap.top();
      heap.pop();
      int currid            = curstate.id;
      int currdistfromstart = curstate.distFromStart;

      if (currdistfromstart > dis[currid]) {
        // 已经有一条最短路径到达当前节点
        continue;
```

```
        }

        for (auto iter : adj(currid)) {
          int distCurr = dis[currid] + weight(iter, currid);
          if (dis[iter] > distCurr) {
            // 更新结果
            dis[iter] = distCurr;
            // 将节点和距离加入到堆中
            heap.push(state(iter, distCurr));
          }
        }
      }

      return dis;
    }

  private:
    std::list<int> adj(int i) {
      return graph[i];
    }

    // 获取权重
    int weight(int from, int to);

    struct state {
      state(int id, int dis) : id(id), distFromStart(dis) {
      }

      int id;               // 图的节点
      int distFromStart;   // 从start节点到当前节点的距离
    };

    struct greator {
      bool operator()(const state &s1, const state &s2) {
        return s1.distFromStart > s2.distFromStart;
      }
    };

    // 邻接表表示的图，key为图的当前节点，value为<临边节点，权重>
    std::map<int, std::list<int>> graph;
  };
```

时间复杂度：`O(ElogE)`

习题：

1. 网络延迟时间(743)

```
有 n 个网络节点，标记为 1 到 n。

给你一个列表 times，表示信号经过 有向 边的传递时间。 times[i] = (ui, vi, wi)，其
```

```
中 ui 是源节点，vi 是目标节点， wi 是一个信号从源节点传递到目标节点的时间。

现在，从某个节点 K 发出一个信号。需要多久才能使所有节点都收到信号？如果不能使所有节点收到
信号，返回 -1 。
```

分析： `dijkstra`距离中的最大值

```
#include <climits>
#include <list>
#include <map>
#include <queue>
#include <vector>

// @lc code=start
class Solution {
public:
  int networkDelayTime(std::vector<std::vector<int>> &times, int n, int k)
{
    createGraph(times);
    std::vector<int> res = dijkstra(k, n);

    int t = 0;
    for (int i = 1; i <= n; i++) {
      if (INT_MAX == res[i]) {
        // 存在节点不可达
        return -1;
      } else {
        t = (t > res[i] ? t : res[i]);
      }
    }

    return t;
  }

private:
  std::vector<int> dijkstra(int start, int n) {
    std::vector<int> res = std::vector<int>(n + 1, INT_MAX);
    std::priority_queue<state, std::vector<state>, greator> heap;  // 小根堆
    res[start] = 0;
    // base case
    heap.push(state(start, 0));

    while (heap.size()) {
      state curstate = heap.top();
      heap.pop();

      int currid   = curstate.id;
      int currdist = curstate.distfromstart;

      if (res[currid] < currdist) {
        continue;
      }
```

```
      for (auto item : adj(currid)) {
        int dist = currdist + item.second;

        if (dist < res[item.first]) {
          res[item.first] = dist;
          heap.push(state(item.first, dist));
        }
      }
    }

    return res;
  }

  void createGraph(std::vector<std::vector<int>> &times) {
    for (auto item : times) {
      graph[item[0]].push_back(std::make_pair(item[1], item[2]));
    }
  }

  std::list<std::pair<int, int>> adj(int v) {
    return graph[v];
  }

  struct state {
    state(int id, int dis) : id(id), distfromstart(dis) {
    }

    int id;
    int distfromstart;
  };

  struct greator {
    bool operator()(const state &s1, const state &s2) {
      return s1.distfromstart > s2.distfromstart;
    }
  };

  std::map<int, std::list<std::pair<int, int>>> graph;
};
// @lc code=end
```

2. 最小体力消耗路径

```
你准备参加一场远足活动。给你一个二维 rows x columns 的地图 heights ，其
中 heights[row][col] 表示格子 (row, col) 的高度。一开始你在最左上角的格子 (0,
0) ，且你希望去最右下角的格子 (rows-1, columns-1) （注意下标从 0 开始编号）。你每次
可以往 上，下，左，右 四个方向之一移动，你想要找到耗费 体力 最小的一条路径。

一条路径耗费的 体力值 是路径上相邻格子之间 高度差绝对值 的 最大值 决定的。
```

> 请你返回从左上角走到右下角的最小 体力消耗值 。

分析： 对二维矩阵运动问题，如果方向一定，可以选择向下，或向右，直接采用`dp`即可； 但对可以上下左右移动，只能采用`dijsktra`算法。

```
#include <climits>
#include <cmath>
#include <list>
#include <queue>
#include <vector>

// @lc code=start
class Solution {
public:
  int minimumEffortPath(std::vector<std::vector<int>> &heights) {
    int row = heights.size();
    if (0 == row) {
      return 0;
    }
    int col = heights[0].size();

    std::vector<std::vector<int>> effortTo =
        std::vector<std::vector<int>>(row, std::vector<int>(col, INT_MAX));

    std::priority_queue<state, std::vector<state>, greator> hp; // 小根堆
    hp.push(state(0, 0, 0));
    effortTo[0][0] = 0;

    while (hp.size()) {
      state cursate = hp.top();
      hp.pop();

      if (cursate.x == row - 1 && cursate.y == col - 1) {
        // 到达末尾
        return cursate.distfromstart;
      }

      if (effortTo[cursate.x][cursate.y] < cursate.distfromstart) {
        continue;
      }

      for (auto item : adj(cursate.x, cursate.y, row, col)) {
        // 计算从curstate.x,curstate.y 到 item.first, item.second的消耗
        int dist = std::max(effortTo[cursate.x][cursate.y],
                            abs(heights[item.first][item.second] -
                                heights[cursate.x][cursate.y]));

        if (dist < effortTo[item.first][item.second]) {
          effortTo[item.first][item.second] = dist;
```

```
          hp.push(state(item.first, item.second, dist));
        }
      }
    }

    return -1;
  }

private:
  std::list<std::pair<int, int>> adj(int x, int y, int row, int col) {
    std::list<std::pair<int, int>> neightors;
    for (auto item : dir) {
      int dx = x + item[0];
      int dy = y + item[1];

      if (isInArea(dx, dy, row, col)) {
        neightors.push_back(std::make_pair(dx, dy));
      }
    }

    return neightors;
  }

  bool isInArea(int x, int y, int row, int col) {
    return x >= 0 && x < row && y >= 0 && y < col;
  }

  struct state {
    int x, y;            // 二维平面中图的坐标
    int distfromstart;  // 从起点到达当前节点的最小距离

    state(int x, int y, int dis) : x(x), y(y), distfromstart(dis) {
    }
  };

  struct greator {
    bool operator()(const state &s1, const state &s2) {
      return s1.distfromstart > s2.distfromstart;
    }
  };

  const std::vector<std::vector<int>> dir = {{-1, 0},
                                             {1, 0},
                                             {0, -1},
                                             {0, 1}};  // 运动方向数组
};
```

3. 最大概率路径

给你一个由 `n` 个节点（下标从 `0` 开始）组成的无向加权图，该图由一个描述边的列表组成，其中 `edges[i] = [a, b]` 表示连接节点 `a` 和 `b` 的一条无向边，且该边遍历成功的概率为 `succProb[i]` 。

> 指定两个节点分别作为起点 start 和终点 end ，请你找出从起点到终点成功概率最大的路径，并返回其成功概率。
>
> 如果不存在从 start 到 end 的路径，请 返回 0 。只要答案与标准答案的误差不超过 1e-5 ，就会被视作正确答案。

分析：

1. 注意浮点数的比较形式；
2. 对`priority_queue`的默认为`less(a<b)`为大顶堆，`greator(a>b)`为小顶堆。

```
#include <cmath>
#include <list>
#include <map>
#include <queue>
#include <vector>

// @lc code=start
const double eps = 1e-5;

class Solution {
public:
  double maxProbability(int                              n,
                        std::vector<std::vector<int>>& edges,
                        std::vector<double>&           succProb,
                        int                              start,
                        int                              end) {
    std::vector<double> res   = std::vector<double>(n, -1);
    auto                graph = createGraph(edges, succProb);
    std::priority_queue<state, std::vector<state>, less> hp; // 大根堆
    hp.push(state(start, 1));
    res[start] = 1;

    while (hp.size()) {
      state currstate = hp.top();
      hp.pop();
      int    currid   = currstate.id;
      double currdist = currstate.distfromstart;

      if (currid == end) {
        return currdist;
      }

      if (currdist + eps < res[currid]) {
        continue;
      }

      for (auto item : graph[currid]) {
        double dist = item.second * currdist;
        if (dist > eps + res[item.first]) {
```

```
          res[item.first] = dist;
          hp.push(state(item.first, dist));
        }
      }
    }

    return 0.0;
  }

private:
  std::map<int, std::list<std::pair<int, double>>> createGraph(
      std::vector<std::vector<int>>& edges,
      std::vector<double>&           succProb) {
    std::map<int, std::list<std::pair<int, double>>> graph;
    int                                              row = edges.size();

    for (int i = 0; i < row; i++) {
      graph[edges[i][0]].push_back(std::make_pair(edges[i][1],
succProb[i]));
      graph[edges[i][1]].push_back(std::make_pair(edges[i][0],
succProb[i]));
    }

    return graph;
  }

  struct state {
    int    id;
    double distfromstart;
    state(int id, double d) : id(id), distfromstart(d) {
    }
  };

  struct less {
    bool operator()(const state& s1, const state& s2) {
      return s1.distfromstart + eps < s2.distfromstart;
    }
  };
};
```

`dijkstra`算法中求最大值用大根堆(`less(a<b)`),求最小值用小跟堆(`greator(a>b)`)

二分图

顶点集可以分为不想交的两个子集，图中的每条边依赖的两个顶点分别属于这两个子集，且两个子集内的顶点互不相邻。 二分图的遍历逻辑：

```
void traverse(Graph &graph, std::vector<bool> &visited, int v) {
  visited[v] = true;
  // 遍历节点v的所有相邻节点neightors
  for(int neightor : graph.adj(v)) {
```

```
      if(!visited[neightor]){
        // 相邻节点数据neightor没有被访问过，将neightor涂上与v不同的颜色
        traverse(graph, visited, neightor);
      }else{
        // 如果已被访问过一定不是二分图。
      }
    }
  }
```

习题：

1.判断二分图

> 给你输入一个 邻接表 表示一幅无向图，请你判断这幅图是否是二分图。

```
#include <iostream>
#include <vector>

// @lc code=start
class Solution {
public:
  bool isBipartite(std::vector<std::vector<int>> &graph) {
    bool              isbip = true;
    int               n     = graph.size();
    std::vector<bool> visited = std::vector<bool>(n, false);  // 是否被访问过
    std::vector<bool> color = std::vector<bool>(n, false);  // 染色

    for (int i = 0; i < n; i++) {
      if (!visited[i]) {
        isBipartite(graph, isbip, visited, color, i);
      }
    }

    return isbip;
  }

private:
  void isBipartite(std::vector<std::vector<int>> &graph,
                   bool                          &isbip,
                   std::vector<bool>             &visited,
                   std::vector<bool>             &color,
                   int                            v) {
    if (!isbip) {  // 已经确定结果，直接返回
      return;
    }
    visited[v] = true;

    for (auto iter : graph[v]) {
      if (!visited[iter]) {       // 临近节点未被访问过
```

```
          color[iter] = !color[v];  // 标记与当前节点的不同颜色
          isBipartite(graph, isbip, visited, color, iter);
        } else {
          // 如果已经被访问过
          // 判断颜色是否与当前节点颜色相同，如果相同，直接返回false
          if (color[iter] == color[v]) {
            isbip = false;
            return;
          }
        }
      }
    }
};
```

2. 可能的二分法

```
给定一组 N 人（编号为 1, 2, ..., N）， 我们想把每个人分进任意大小的两组。

每个人都可能不喜欢其他人，那么他们不应该属于同一组。

形式上，如果 dislikes[i] = [a, b]，表示不允许将编号为 a 和 b 的人归入同一组。

当可以用这种方法将所有人分进两组时，返回 true；否则返回 false。
```

分析： 将`dislike`看成一个图，如果这个图可以二分，则表示能够分成两组，否则不能够分成两组。

```
#include <list>
#include <vector>

// @lc code=start
class Solution {
public:
  bool possibleBipartition(int n, std::vector<std::vector<int>> &dislikes)
{
    std::vector<bool>           visited = std::vector<bool>(n + 1, false);
    std::vector<bool>           color   = std::vector<bool>(n + 1, false);
    std::vector<std::list<int>> graph   = buildGraph(n, dislikes);

    for (int i = 1; i <= n; i++) {
      if (!visited[i]) {
        if (!traver(graph, i, visited, color)) {
          return false;
        }
      }
    }

    return true;
  }
```

```
  private:
    bool traver(std::vector<std::list<int>> &graph,
                int                          v,
                std::vector<bool>            &visited,
                std::vector<bool>            &color) {
      visited[v] = true;
      for (auto iter : graph[v]) {
        if (!visited[iter]) {
          color[iter] = !color[v];
          traver(graph, iter, visited, color);
        } else {
          if (color[iter] == color[v]) {
            return false;
          }
        }
      }
      return true;
    }

    std::vector<std::list<int>> buildGraph(
        int                               n,
        std::vector<std::vector<int>> &dislike) {
      std::vector<std::list<int>> graph =
          std::vector<std::list<int>>(n + 1, std::list<int>());

      int row = dislike.size();
      for (int i = 0; i < row; i++) {
        int v1 = dislike[i][0];
        int v2 = dislike[i][1];
        graph[v1].push_back(v2);
        graph[v2].push_back(v1);
      }

      return graph;
    }
  };
```

`kruskal`最小生成树

最小生成树：在图生成的二叉树中，最小权值和的二叉树。 例如：

```
给你输入编号从0到n - 1的n个结点，和一个无向边列表edges（每条边用节点二元组表示），
请你判断输入的这些边组成的结构是否是一棵树。
```

分析：

1. 可以采用`union find`算法，判断是否能够构成唯一分量

```
```

```cpp
#include <vector>

// union find
class UF {
public:
  // 初始化
  UF(int n) {
    uf_     = std::vector<int>(n, -1);
    weight_ = std::vector<int>(n, 0);
    count_  = n;

    for (int i = 0; i < n; i++) {
      uf_[i]     = i;
      weight_[i] = 1;
    }
  }

  // 连接p,q节点
  void connect(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);
    if (rootQ == rootP) {
      return;
    }

    if (weight_[rootP] < weight_[rootQ]) {
      // 链接到小根节点
      weight_[rootP] += weight_[rootQ];
      uf_[rootQ] = rootP;
    } else {
      weight_[rootQ] += weight_[rootP];
      uf_[rootP] = rootP;
    }

    count_--;
  }

  // 判断是否连通
  bool isConnect(int p, int q) {
    int rootq = find(q);
    int rootp = find(p);
    return rootq == rootq;
  }

  // 查找父节点
  int find(int q) {
    if (q != uf_[q]) {
      uf_[q] = uf_[uf_[q]];  // 路径压缩
      q      = uf_[q];
    }

    return q;
```

```
  }

  int count() const {
    return count_;
  }

private:
  std::vector<int> uf_;       // 存储一颗树
  std::vector<int> weight_;  // 权重
  int              count_;   // 连通分量个数
};

class Solution {
public:
  // 给定一个边集，判断这个边集能否构成一棵树
  bool validTree(int n, std::vector<std::vector<int>> edges) {
    // 初始化0~n-1个节点
    UF uf(n);
    for (auto item : edges) {
      int p = item[0];
      int q = item[1];
      if (uf.isConnect(p, q)) {
        return false;
      }

      uf.connect(p, q);
    }

    return uf.count() == 1;
  }
};
```

2. `kruskal`算法。 所谓最小生成树，就是图中若干边的集合（我们后文称这个集合为 mst，最小生成树的英文缩写），你要保证这些边： 1、包含图中的所有节点。 2、形成的结构是树结构（即不存在环）。 3、权重和最小。

将所有边按照权重从小到大排序，从权重最小的边开始遍历， 如果这条边和 mst 中的其它边不会形成环，则这条边是最小生成树的一部分， 将它加入 mst 集合；否则，这条边不是最小生成树的一部分，不要把它加入 mst 集合。

最小代价连通城市

```
#include <algorithm>
#include <vector>

class UF {
public:
  UF(int n) {
    parent_ = std::vector<int>(n, -1);
    weight_ = std::vector<int>(n, 0);
    count_  = n;
```

```
    for (int i = 0; i < n; i++) {
      parent_[i] = i;
      weight_[i] = 0;
    }
  }

  void connect(int p, int q) {
    int rootp = find(p);
    int rootq = find(q);
    if (rootp == rootq) {
      return;
    }

    if (weight_[rootp] < weight_[rootq]) {
      weight_[rootp] += weight_[rootq];
      parent_[rootq] = rootp;
    } else {
      weight_[rootq] += weight_[rootq];
      parent_[rootp] = rootq;
    }
  }

  bool connected(int p, int q) {
    int rootp = find(p);
    int rootq = find(q);

    return rootp == rootq;
  }

  int count() const {
    return count_;
  }

  int find(int v) {
    while (v != parent_[v]) {
      parent_[v] = parent_[parent_[v]];
      v          = parent_[v];
    }

    return v;
  }

private:
  int              count_;   // 连通分量个数
  std::vector<int> weight_;  // 权重数组
  std::vector<int> parent_;  // 连通分量父节点数组
};

int minimumCost(int n, std::vector<std::vector<int>> &connections) {
  UF  uf(n + 1);
  int mst = 0;
  std::sort(connections.begin(),
            connections.end(),
            [](std::vector<int> &a, std::vector<int> &b) {
```

```
                return a[2] < b[2];
              });

    for (int i = 0; i < n; i++) {
      int u = connections[i][0];
      int v = connections[i][1];
      int w = connections[i][2];

      if (uf.connected(u, v)) {
        continue;
      }

      mst += w;
      uf.connect(u, v);
    }

    return uf.count() == 2 ? mst : -1;
  }
```

连接所有点的最小费用

```
给你一个points 数组，表示 2D 平面上的一些点，其中 points[i] = [xi, yi] 。

连接点 [xi, yi] 和点 [xj, yj] 的费用为它们之间的 曼哈顿距离 ：|xi - xj| + |yi -
yj| ，其中 |val| 表示 val 的绝对值。

请你返回将所有点连接的最小总费用。只有任意两点之间 有且仅有 一条简单路径时，才认为所有点
都已连接。
```

```
#include <algorithm>
#include <cmath>
#include <vector>

// @lc code=start
class Solution {
public:
  int minCostConnectPoints(std::vector<std::vector<int>>& points) {
    std::vector<std::vector<int>> graph = buildGraph(points);
    int                           n     = points.size();

    int mst = 0;
    UF  uf(n);

    std::sort(graph.begin(),
              graph.end(),
              [](std::vector<int>& a, std::vector<int>& b) {
                return a[2] < b[2];
              });
```

```
    for (int i = 0; i < graph.size(); i++) {
      int u = graph[i][0];
      int v = graph[i][1];
      int w = graph[i][2];
      if (uf.connected(u, v)) {
        continue;
      }

      mst += w;
      uf.connectTwoPoints(u, v);
    }

    return mst;
  }

private:
  std::vector<std::vector<int>> buildGraph(
      std::vector<std::vector<int>>& points) {
    std::vector<std::vector<int>> graph;

    int row = points.size();

    for (int i = 0; i < row; i++) {
      for (int j = i + 1; j < row; j++) {
        int dis =
            abs(points[i][0] - points[j][0]) + abs(points[i][1] - points[j]
[1]);
        graph.push_back(std::vector<int>{i, j, dis});
      }
    }

    return graph;
  }

  class UF {
  public:
    UF(int n)
        : count_(n),
          parent_(std::vector<int>(n, 0)),
          weight_(std::vector<int>(n, 0)) {
      for (int i = 0; i < n; i++) {
        parent_[i] = i;
      }
    }

    void connectTwoPoints(int p, int q) {
      int rootp = find(p);
      int rootq = find(q);

      if (rootq == rootp) {
        return;
      }

      if (weight_[rootp] < weight_[rootq]) {
```

```
        weight_[rootp] += weight_[rootq];
        parent_[rootq] = rootp;
      } else {
        weight_[rootq] += weight_[rootq];
        parent_[rootp] = rootq;
      }
    }

    bool connected(int p, int q) {
      int rootp = find(p);
      int rootq = find(q);

      return rootp == rootq;
    }

    int count() const {
      return count_;
    }

    int find(int p) {
      while (p != parent_[p]) {
        parent_[p] = parent_[parent_[p]];
        p          = parent_[p];
      }

      return p;
    }

  private:
    int              count_;
    std::vector<int> parent_;
    std::vector<int> weight_;
  };
};
```