堆 & 最短路

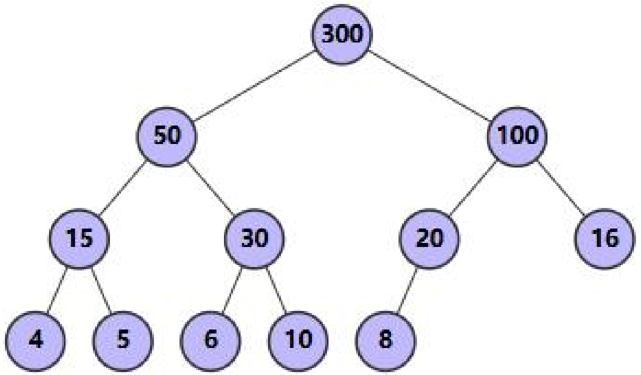
Kaslen_

关注乃琳Queen谢谢喵

堆

- 今天所讲的堆是二叉堆
- 它是一棵二叉树，并且是完全二叉树，每个结点中存有一个元素（或者说，有个权值）。

- 堆性质：父亲的权值不小于儿子的权值（大根堆）。同样的，我们可以定义小根堆。



支持的操作

- 显然，最基本需要
- 1、查询最大值
- 2、插入一个元素
- 3、删除最大值

核心：向上/向下调整

- 向上调整：如果这个结点的权值大于它父亲的权值，就交换，重复此过程直到不满足或者到根。
- 向下调整：在该结点的儿子中，找一个最大的，与该结点交换，重复此过程直到底层。

查询操作

- 很简单，显然树根就是最大值

插入操作

- 向堆最下一层最靠右的位置插入一个元素，然后向上调整。
- 如果最下一层已满，则新开一层。

删除操作

- 最朴素的想法是直接删除根堆，但这样就变成了两棵树，不好维护。
- 只需要把根节点向下调整，直到成为叶子节点，然后删除即可

复杂度分析

- 显然，由于是完全二叉树，单次插入或删除都是logn的，而查询是O1的
- 因此总复杂度就是O(qlogn)的

动画演示：https://gallery.selfboot.cn/zh/algorithms/heap

优先队列：stl库里的堆

- empty() 如果队列为空返回真

- pop() 删除队顶元素

- push() 加入一个元素

- size() 返回优先队列中拥有的元素个数

- top() 返回优先队列对顶元素

- 默认的优先队列是大根堆。
- 复杂度跟堆一样

如何改为小根堆?

- 第一种做法是把所有数*=-1，然后插入，查询的时候也*=-1即可。

- 当然，先进的stl有转换大小根堆的功能:
- 大根堆:

```
priority_queue<int,vector<int>,less<int> >q;
```

- 小根堆:

```
priority_queue<int, vector<int>, greater<int> > q
```

自定义排序方式

- 有些时候我们需要利用堆来维护结构体的关系，但自定义的结构体并没有预定义好”<”这个符号，因此在堆中使用小于号进行大小比较的时候会出错。

- 只需要利用重载运算符对小于号重载即可

- 其实改变大小根堆也可以通过对小于号的重载达到目的

灵活应用：最小函数值

- https://www.luogu.com.cn/problem/P2085

灵活应用：对顶堆

- https://www.luogu.com.cn/problem/P1801

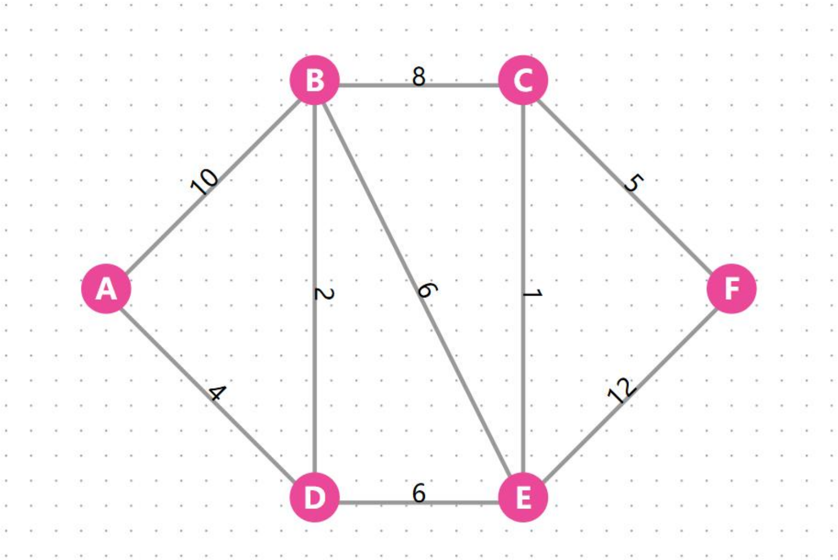
最短路

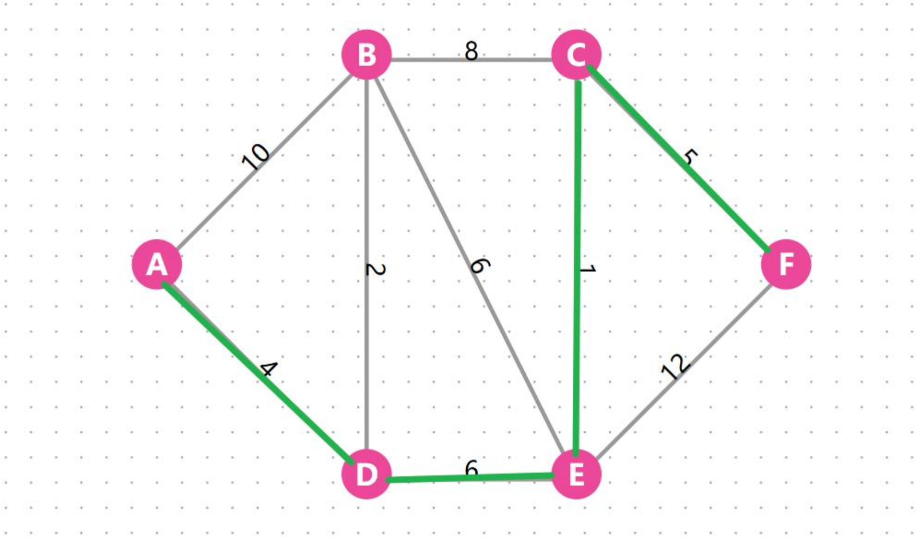
- 问题引入：

  Lisa肚子饿了，想去食堂吃晚餐，Lisa已经饿麻了，希望走最少的步数到达食堂，如果走的步数比期望的最少步数多，Lisa就饿死了，要怎么规划一条路径让Lisa在饿死之前吃到饭？

最短路

- 问题抽象：
  我们可以把某些路上的位置抽象为点，在两个点之间的距离视为边，所需走的步数抽象为边权，这样就抽象出来了一张图。那么问题就转化为在一张图上，找到一条路径使得从A点到F点的边权和最小





Dijkstra算法

- Dijkstra算法解决的问题是从单点出发，在非负权图上的单源最短路问题
- 可以先直观的感受一下：如果我们每次都从能到达的距离最近的点往外扩展，那么当到我们从目标点继续往外扩展的时候，此时从起点到目标点的距离一定是最小的

Dijkstra算法

- 这个结果是显然的，因为我们每次扩展出去的点都是从当前距离最小值点扩展。
- 这个结论建立在非负权图的基础上
- 特例：如果存在一个负权环，使得能从目标点出发，经过负权环并回到目标点，此时可能会使得从起点到终点的最短路径经过终点，再回到终点。

Dijkstra算法

- 算法流程：

将结点分成两个集合：已确定最短路长度的点集（记为 S 集合）的和未确定最短路长度的点集（记为 T 集合）。一开始所有的点都属于 T 集合。

初始化 $dis(s)=0$，其他点的 dis 均为 $+\infty$。

然后重复这些操作：

1. 从 T 集合中，选取一个最短路长度最小的结点，移到 S 集合中。
2. 对那些刚刚被加入 S 集合的结点的所有出边执行松弛操作。

直到 T 集合为空，算法结束。

演示动画：https://gallery.selfboot.cn/zh/algorithms/dijkstra

Dijkstra算法

- 正确性证明

设算法迭代每一次加入集合 S 的结点依次为 $v_{x_1}, v_{x_2} \ldots . v_{x_k}$ 证明每一次加入 S中的结点都是最短路径

1. 第一次加入的结点 v_{x_1} 显然是最短路径。
2. 假设第K次加入的结点 v_{x_k} 为最短路径，此时集合S为 $\{v_{x_1}, v_{x_2}, \ldots v_{x_k}\}$,现证明根据算法运行第 k + 1次加入的结点 $v_{x_{k+1}}$ 仍为最短路径。
3. 反证，假设加入的 $v_{x_{k+1}}$ 不是最短路径，即存在另外一条路径 为最短路径，设 v_u 为该路径中不在集合 S 的第一个结点，则 $dist\left[v_u\right] < dist\left[v_{x_{k+1}}\right]$ ，dijkstra的算法的第一个步骤是每次从集合T中基于 $dist$ 选取距离最小的结点加入集合 S ，因此 $dist\left[v_{x_{k+1}}\right] <= dist\left[v_u\right]$ ，因此产生矛盾。得证。

Dijkstra算法

- 由于每次都要选择最短的一个节点，故我们使用小根堆去动态维护大小关系。复杂度：O(mlogm)

```cpp
const int N = 10000 + 10;
const int INF = 0x3f3f3f3f;
int n,m;
vector<array<int,2>> edges[N];//所有边
void dijk(int start, int end){
    vector<int> dis(n+1,INF);
    vector<int> state(n + 1);//点是否入队
    priority_queue<array<int,2>, vector<array<int,2>>, greater<array<int,2>>> heap;  // 小根堆,first为最短距离,second为节点编号
    dis[start] = 0;  // 起点到自己的距离为0
    heap.push({ 0,start });  // 起点入堆

    while (heap.size()) {
        auto [d, u] = heap.top(); heap.pop();

        if (state[u]) continue;  // 节点u已确定最短距离,则跳过

        state[u] = true;  // 确定节点u到起点s的最短距离
        if(u == n) break;
        for (auto [v, w] : edges[u]) {  // 用节点u更新其能到达的节点到起点的最短距离
            if (dis[v] > d + w) {
                dis[v] = d + w;
                heap.push({ dis[v],v });  // 被更新的节点入堆
            }
        }
    }

    if(dis[end]>INF/2) cout<<-1<<endl;
    else cout<<dis[end]<<endl;
}

```

SPFA算法

- 前面讨论的Dijkstra算法是在非负权图上的最短路算法，那么如果图中存在负权，那么我们就要使用SPFA算法
- 在其之前，我们先简单的介绍一下SPFA的前身Bellman-Ford算法

Bellman-Ford算法

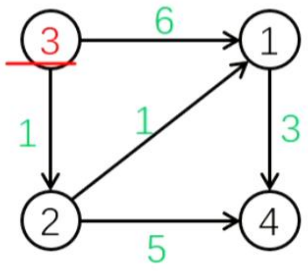
Bellman-Ford(贝尔曼-福特)算法 基于松弛操作的单源最短路算法。

e[u] 存u点的出边的邻点和边权，d[u] 存u点到源点的距离。

1. 初始化，d[s]=0，d[其它点]=+∞；
2. 执行多轮循环。每轮循环，对所有边都尝试进行一次松弛操作；
3. 当一轮循环中没有成功的松弛操作时，算法停止。

4 5 3
3 1 6
3 2 1
2 1 1
2 4 5
1 4 3

建图
3→1, 2
1→4
2→1, 4



i=1 u=1
u=2
u=3 d[1]=6 d[2]=1
u=4
i=2 u=1 d[4]=9
u=2 d[1]=2 d[4]=6
u=3
u=4
i=3 u=1 d[4]=5
u=2, 3, 4
i=4 u=1, 2, 3, 4
flag=false

```
struct Edge{//结构体存m条边
    int u,v,w;
};
void Bellman(int s){
    memset(dis,INF,so(dis));
    dis[s]=0;
    for(int i=1;i<n;i++){//n-1次松弛
        for(int j=0;j<m;j++){//遍历所有边进行
            auto &[u,v,w]=edges[j];
            if(dis[u]!=INF&&dis[v]>dis[u]+w)
                dis[v]=dis[u]+w;
        }
    }

    for(int i=1;i<=m;i++){
        Edge temp=egde[i];
        if(dis[temp.v]<dis[temp.u]+w){
          //若经过n-1轮迭代后还存在更短的路径，则存在负环
        }
    }

}
```

SPFA算法

- spfa是bellman算法的改良版，因为bellman算法是对每一条边进行了n-1次松弛操作，为了降低复杂度，只对当前点的所有出边进行更新，如果满足松弛条件，则改变dis

SPFA算法

Bellman-Ford 算法的优化——SPFA

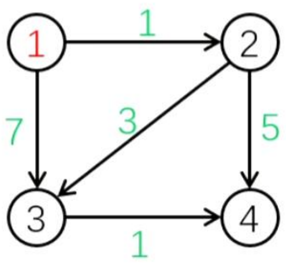
只有本轮被更新的点，其出边才有可能引起下一轮的松弛操作，因此用队列来维护被更新的点的集合。

vis[u] 标记u点是否在队内，cnt[v] 记录边数，判负环。

1. 初始化，s 入队，标记 s 在队内，d[s]=0，d[其它点]=+∞；
2. 从队头弹出u点，标记 u 不在队内；
3. 枚举 u 的所有出边，执行松弛操作。记录从 s 走到 v 的边数，并判负环。如果 v 不在队内则把 v 压入队尾，并打上标记；
4. 重复2, 3步操作，直到队列为空。

4 5 1
1 3 7
1 2 1
2 4 5
2 3 3
3 4 1

建图
1 → 3, 2
2 → 4, 3
3 → 4




```
struct edge{
    int to,w,next;
}edge[MAX];
int head[MAX],idx;
int n,m;

void add_edge(int u,int v,int w){
    edge[idx].to=v;
    edge[idx].w=w;
    edge[idx].next=head[u];
    head[u]=idx++;
}

int dis[MAX],vis[MAX],cnt[MAX];
bool spfa(){
    memset(dis,INF,so(dis));//最长路初始化为-INF，最短路初始化为INF
    queue<int> q;
    dis[0]=0;
    q.push(0);
    vis[0]=1;
    while(q.size()){
        int t=q.front();q.pop();
        vis[t]=0;

        for(int i=head[t];~i;i=edge[i].next){
            int j=edge[i].to;
            if(dis[j]>dis[t]+edge[i].w){//根据最长路或最短路改变符号
                dis[j]=dis[t]+edge[i].w;

                cnt[j]=cnt[t]+1;
                if(cnt[j]>=n+1) return false;//查询是否存在环

                if(!vis[j]){
                    q.push(j);
                    vis[j]=1;
                }
            }
        }
    }
    return true;
}
```

Floyd算法

- 前面我们讲的两种算法都是单源最短路问题，那么如果我们要查询q次任意两个点之间的最短路，要怎么处理呢？
- 显然不可能运行q次单源最短路算法

Floyd算法

- 先给出Floyd算法的核心式子：

```
int dist[N][N], n;
void Floyd(){
    for(int k = 1; k <= n; k++){
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= n; j++){
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
            }
        }
    }
}

```

Floyd算法

- 其本质思想就是如果从i -> k是最短的；从k -> j是最短的，那么从i -> k -> j就一定的最短的

注意：使用Floyd的时候，需要先将边权初始化进dist，并且
dist[i][i] = 0, 若i -> j不存在边时，初始化为INF

复杂度

- n为节点数，m为边数
- Dijkstra算法：O((n + m)logn)
- SPFA算法：O(nm)
- Floyd算法：O(n^3)