经典算法笔记

一、图论算法

1. 求有向图的强连通分量：

首先明确几个变量：开两个 pre[], post[]数组，代表着结点的前序编号和后序编号。然后是 clock：全局的一个指示变量，用于让 pre 和 post 数组的元素离散递增。

explore 函数根据 clock，DFS 这张图，然后填上各节点的 pre 和 post 值。

a) 将图 G 反向（边取反），得到 Gr
b) for 每个节点，对 Gr 调用 explore(v)，填完 pre 和 post 数组
c) 根据每个节点的 post 从大到小排序，按这个排序，对 G 调用无向图的连通分量算法即可

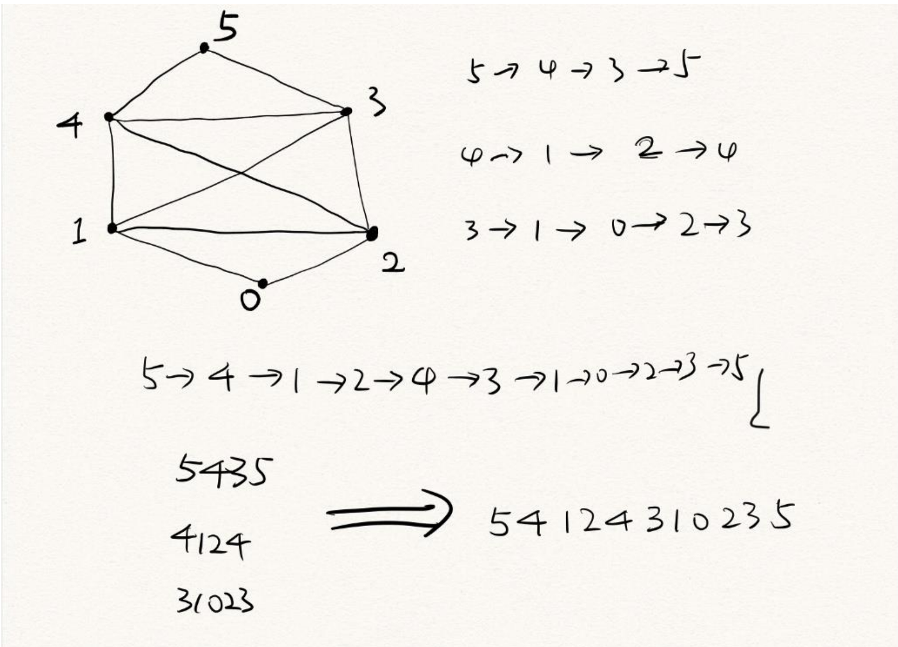
正确性是因为 Gr 里 post 最大的那个一定在“sink”里，所以按照 post 降序遍历可以得到正确的连通分量个数。

2. 欧拉回路：

首先判断是不是每个节点的度都是偶数，若不是，直接返回 false 输出空串

a) 随机选取一个节点进行没有回溯的 DFS（没有回溯就是没有那个 for 循环，而是不断地 DFS(下一条邻边)这样），直到返回了最初的节点（期间访问过的边不会再访问了，而访问过的结点还会）。
b) 若达到了所有边都访问过一次（即邻接表所有边的 vis 字段都设 true 了），那么相当于已经完成了一笔画，返回最终答案即可
c) 若没有，则选择路径中还有邻边没访问的节点，进行没有回溯的 DFS，并将得到的遍历序列合并到 ans 这个 string 中。
d) 重复 bc 直至所有边都被访问过

例如下面这样的：



3. **拓扑排序：**

   有向无环图（DAG）可以进行拓扑排序。具体做法是，先把度为 0 的所有节点入队。然后 while 队非空，出队一个节点，访问它（加入拓扑排序的序列），然后对每一个后继执行：后继入度-1，若此时后继入度为 0，那么后继入队。

4. **关键路径和关键节点（AOE 网）：**

   节点包含：a、后继列表，b、最早开工时间，c、最迟开工时间

   a) 对网络进行拓扑排序，得到拓扑排序序列。
   b) 计算最早开工时间：初始化各节点的最早开工时间为 0。对拓扑排序序列的节点顺序遍历，对节点的每一个后继执行：若当前节点到后继的边+当前节点的最早开工时间>后继节点的最早开工时间，更新后继节点的最早开工时间为更大的那个值。
   c) 计算最迟开工时间：初始化各节点的最迟开工时间为 b 步骤得到的汇节点的最早开工时间（这个就是汇节点的最晚开工时间，即最长路的长度）。对拓扑排序序列的节点逆序遍历，对每一个结点执行：将该节点的最迟开工时间更新为（它的某一后继节点的最迟开工时间-该节点到后继节点的边距离）中最小的那一个。
   d) 顺序遍历拓扑排序序列，找出那些最早开工时间等于最迟开工时间的节点，它们即是关键节点，它们组成的路径即是关键路径。

5. **最小生成树算法：**

   a) **Prim 算法：**

      从节点的角度来考虑，首先节点 S 为任一节点，U 为所有节点，然后从 S 内的所有节点中与 U-S 中节点相邻的边里，挑出边最小的那个，并将对应的节点加入 S 中，重复直至 S 包含所有节点。

   b) **Kruskal 算法：**

      更简单了，每一次都选择权值最小且加入之后不会形成环的那条边，加入 n-1 条边之后就得到了最小生成树。

6. **单源最短路（不含 BFS）：**

   首先声明，无向图中不能出现负权边，否则可以通过在负权边的两个端点之间反复横跳来不断减少 cost，而有向图中不能有负环。（可能有的话需要检测）

   a) **单源最短路的 Dijkstra 算法：（不能解决负权边）**

      首先初始化全部节点为 inf，然后设置初始节点距离为 0，并将其加入 S 集合。接下来重复以下步骤：

      ①、对新加入 S 集合的点的所有后继节点进行松弛操作

      ②、在所有非 S 集合的结点中找出距离源点最近的结点，加入 S 集合

      ③、若 S 集合元素个数到达点数 n，break，否则跳回①

      Dijkstra 算法的堆优化：

      我们在②步的时候，如果采用的是线性查找最小值，那么需要 O(n)，一共 n 轮循环，所以最终的复杂度是 $O(n^2)$。但是如果我们用一个最小化堆（优先级队列）来储存非 S 集合的节点即可。有一个实现的细节需要注意：当松弛操作发生后，已经在堆里面的节点要如何更新？事实上我们根本就不需要去更新它，因为只有在这个点离源点的最短距离减小时我们才需要更新它，那么我们其实只需要在优先级队列中加入一个新的（node_i, new_distance_i）即可，它一定

比之前的（node_i, old_distance_i）要先出队，我们把在 S 集合中的点做了标记，之后再出队遇到的是已经在 S 集合中的点时，把它删掉就好了。

代码注意事项：

1、首先是，dis[start] = 0;之后，不要把 vis[start]设为 true！！！这一点与 BFS 是不一样的！！！！！

2、然后是 while 循环里优先级队列 top 出来之后，记得判断 if(vis[u]) continue; 之后还要设 vis[u] = true;

3、使用小根堆的时候重载<时要写 return 自己大于对方的逻辑，并且不要忘了把函数定义为 const 的。

4、初始化 dis 数组的时候是用 0x3f 来初始化的，不是用 0。

代码见“模板代码/dijkstra.cpp”

复杂度分析：

时间复杂度是 O(ElongE)，如果是 n 个点，一般是稀疏图就是说 n^2 >>E，那其实复杂度大概是 nlogn 这样。

b) **SPFA 算法：（可以解决负权边和判断负环）**

在数据分布正常（没有用特殊的图卡你）的情况下，时间复杂度是比 Dijkstra 和 Bellman-Ford 算法低的。具体操作如下：

①、建立普通队列并将源点入队

②、每次取出队首节点 u，对它所有的后继节点 v 进行松弛操作，如果后继节点被松弛了，那么检查它是否已经在队列中，如果不在，就要将其入队。

③、重复①②，直到队列为空。

负环检测：

1、可以用一个数组记录每个节点途径的点数，超过 n+1 个点时就说明存在负环了。emmmm 不是可以，而是一定要，除非题目告诉你没有负环了，不然一定要这样，否则程序可能死循环。具体地，用一个 num 数组记录进行松弛的时候第 i 个点已经经过 num[i]个点。如果 nunm[i]>=n，那么一定有负环！因为没有负环的图是不可能经过超过或等于 n 个点来松弛某一个节点的（别问为什么，问就是不会）。在松弛的时候，若松弛成功，就 num[v] = num[u] + 1; 这样去维护 num 数组。

2、还有一种 DFS 版的 SPFA，其原始思想就是，不断地往被松弛的边的后继节点 DFS，在 DFS 某点前把 vis[某点]设为 true，这样如果在 DFS 函数开头碰到了当前点的 vis 为 true，那就说明绕了一圈回来了，这时就可以 return true 了，然后注意一旦有一个函数 return true 了，那么就都要 return true，其实在递归的时候写 if(spfa(下一个点)) return true; 就好了。

3、可以记录每个节点入队（被松弛）的次数，大于等于 n 次就说明有负环（其实就是 bellman-ford 的想法）。

好像 1 和 3 是差不多的意思，但是 1 更快一些。

代码注意事项：

①、把初始节点进队之后要设 vis[start] = true; //这是第③点的特殊情况

②、节点 pop 出来之后要设 vis[u] = false;

③、节点 push 进队之后要设 vis[v] = true;

代码见“模板代码/spfa.cpp”

c) **Bellman-Ford 算法：（可以解决负权边和判断负环）**

Bellman-Ford 是以边为依据进行松弛操作的。假如有 n 个点，m 条边，那么只要做 n-1 次以下的松弛操作即可：

对于每条边（u, v）（有向图就正常，无向图两个方向都要，其实用邻接表就没有这个问题），都去看看当前的 d(u) + w(u, v)是否小于 d(v)，如果小于就更新 d(v)的值。而每一轮更新松弛操作边的顺序没有关系。因为我们最多做 n-1 轮，在做的过程中其实如果某一轮没有边被松弛就可以停下了。而我们考虑，某个 Oracle 已经知道了实际的最短路离源点有 x 跳（即经过 x 条边），由于每一轮更新都必遍历了 m 条边，因此至少在第 i 轮更新的时候，距离源点的“客观存在”的最短路径为 i 跳的那些最短路是会被找出来的，而跳数最大为 n-1（除非存在负环），因此最多经过 n-1 次松弛操作所有的最短路都被找出来了。

而判断是否存在负环的依据就是，第 n 轮松弛仍有边被更新。

7. 多源最短路的 Floyd 算法：

   首先初始化 dis[i][i] = 0; dis[i][j] = weight(i,j)，之后：

   非常非常非常简单的三行代码：

```
for(int k=0; k<n; k++)
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            dis[i][j] = min(dis[i][j], dis[i][k]+dis[k][j])
```

   简单暴力的 O(n^3)算法……注意 Floyd 还有闭包传递的功能，即可以把具有传递性的等价关系全部传递出来。具体见《笔记》

   然后就是 Floyd 也可以判断负环，存在 dis[i][i]<0 就是有负环

8. 网络流相关：

   a) 网络最大流算法：Ford-Fulkerson 算法，垃圾，用 dinic

   直接看模板代码/maxflow.cpp 即可（还是别看这个了，看 GraphAlgorithm 里面的 LuoGu3376_MaxFlow 的代码好了，注释比较详细。）

   关键的地方就是理解 DFS(int now, int flow)这个函数本质上是递归，它的意义就是 now 这个节点流入了 flow 的流量，它能流出的最大流量。根据这个递归意义，代码也就很好理解了。分层不太好理解，记一下就好了。

   写一下代码的注意点吧：

   ①、首先是 BFS 里面不要忘记把距离更新一下……然后就是注意当到达终点 t 的时候要 return true

   ②、BFS 里不要忘记一开始把 cur 数组初始化为 head 数组，并且要注意如果点是从 1 开始的要<=n 而不是写<n！！！

   ③、DFS 里面不要忘记，当 dis[v] != dis[now]+1 的时候要直接 continue！！！

   ④、DFS 不要忘记如果 now==t 或者 flow==0 就 return flow;

   ⑤、DFS 不要忘记在循环后继边的时候判断 flow-used==0 就返回 used

   ⑥、DFS 函数里不要忘记当前边优化，循环后继边的时候是从 int i=cur[now]开始，然后循环一开始就要写 cur[now] = i;

   ⑦、main 函数里不要忘记先 memset(head, 0, sizeof(head));

   ⑧、边集的 cnt 是从 2 开始，不要忘记 main 函数里要加反向边！！！

   b) 最小费用最大流：

   直接看 GraphAlgorithm 里面的 MaxFlowMinCost 代码即可

   https://www.luogu.com.cn/problem/P3381 模板题

   写一下我的理解吧。

1、背景：

最小费用最大流的背景是，每一条边除了有一个容量 capa，还有一个单位流量的 cost。通过某条边的花费是 flow 的大小乘以这条边的 cost。要求的就是，在最大流的前提下，流量的总花费最小。

2、思路：

思路其实和 Ford-Fulkerson 差不多，我们现在做的是单路增广，不像 dinic 做的多路增广——我们使用 spfa，每一次都求出一条总 cost 最短(小)的增广路径，然后用这条路径的 bottleneck 去增加流量，同时算出花费(cost)。

具体地，我们使用了 5 个和点集大小一样大的 int 数组，分别是：head，cost，flow，pre，preedge。其意义分别是：链式前向星的 head 数组；cost 数组的意义和 spfa 的 dis 数组一样；flow 数组表示的是 flow[i]表流入 i 这个节点的流量值；pre 数组用来记录增广路上每个节点的前一个结点；preedge[i]表示的是增广路上 i 这个节点的前面一条边在链式前向星边数组中的下标。

我们的 spfa 算法返回值为 bool，表示有没有找到一条增广路。与普通的 spfa 不同的是，需要在松弛的时候额外维护 pre、preedge、flow 三个数组。其中 pre、preedge 很好维护，也就是用来记录路径的，而 flow 数组的维护就比较麻烦点，就是在松弛的时候要把 flow[v]更新为 flow[u]和 capacity of edge_uv 中更小的那一个，这样在最后我们才可以吧 flow[t]当做一条增广路径的 bottleneck。

至于 MFMC 函数里，只需要 while(spfa())然后利用 spfa 更新的信息去更新全局变量 maxflow 和 mincost 即可，同时也按照 pre 数组记录的路径更新一下 residual graph 的信息（增广路上的正向边减 flow[t]，反向边加 flow[t]这样）。

具体代码还是看 GraphAlgorithm 里面的 MaxFlowMinCost 代码啦。

3、代码注意点：

①、spfa 初始化要把 cost 和 flow 都初始化为 inf；把 pre[t]初始化为-1；最后 return 的值也是 pre[t] != -1

②、更新 mincost 的时候要用 mincost += flow[t]*cost[t]，表示流量值乘以路径长度（路径长度 cost[t]就是这条路上单位流量流过的 cost）。

③、初始化反向边的时候，反向边的 cost 要是正向边 cost 的相反数！！！这是因为我们在走反向边的时候，相当于减小了这条边流量，从而减小了 cost。

④、可以松弛的条件为 cost[v] > cost[u]+cost_uv && e[u_to_v(通常为 i)].capa>0 即不仅要距离变短，而且要 capacity 大于 0！

9. 二分图最大匹配算法：匈牙利算法

https://blog.csdn.net/young__fan/article/details/90719285 这篇蛮不错，前面很多废话，从增广路那里就是原理了。具体原理大概是，对于每一个 x 结点，都用一个 found 函数在 y 结点里找它是否存在匹配。而 found 函数利用了增广路的想法，即若能找到从某一未匹配的 x 结点到一个未匹配的 y 结点的增广路，就把增广路上的匹配状态都取反，即可增加一个匹配。具体原理看那篇博客，代码模板在下面给出了，我已经写好了注释：

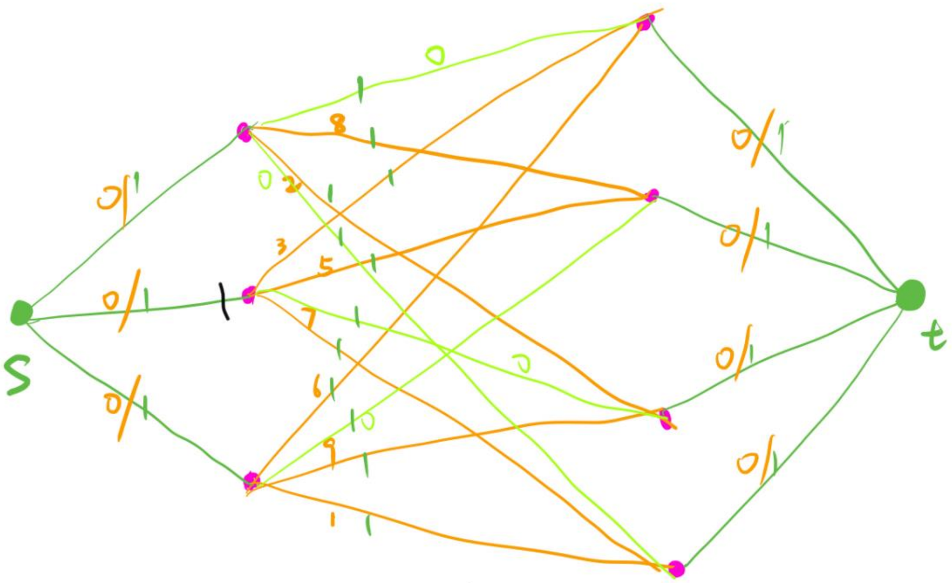
```cpp
const int N = 105;
bool line[N][N]; //line[x][y]记录x和y有没有连边（是否可以配对），1为可以，0为不可
bool used[N]; //在对每一次节点调用found函数时使用，记录这一次更新匹配有哪些y节点是在用的，即已经尝试过了吗
int result[N]; //记录第i个y节点的配对x是result[i]，0表示没有配对，但这个和used不是一样的意思
int n, m, s, v; //m是实际的y节点数

bool found(int x) //对于下标为x的x节点找出是否能够配对，并在找寻的过程中递归修改匹配状态result数组
{
    for(int i=1; i<=m; i++)
    {
        if(line[x][i] && !used[i])
        {
            used[i] = true; //当前这个节点铁定是要在这次found中被用到了
            if(result[i]==0 || found(result[i])) //如果当前探侧的i这个y节点没有原配，或者可以给它的原配找到新的配对者
            {
                result[i] = x; //把当前探侧的这个节点和x配对
                return true;
            }
        }
    }
    return false; //全部找完了都没有增广路，返回false
}
```

使用时如下：对于每一个 x 当中的结点，去 found 有没有能匹配的。

```cpp
int cnt = 0;
for(int i=1; i<=n; i++)
{
    memset(used, 0, sizeof(used));
    if(found(i)) cnt++;
}
```

其中 cnt 就是匹配个数。

10. 二分图的最大权匹配：用最小费用最大流实现



如果你连最大流（全为1的流量）都没有达到，那必然不会是最大权匹配！因为肯定可以加上至少一条边，使得 $\sum w_i$ 变大。

因此可以将每条边的容量设为1，cost设为权值的相反数，源点到X点集设容量1，权值0，Y点集到汇点权值0，容量1；建完图后跑最小费用最大流即可。最后输出最小费用的相反数即是匹配的最大权值和。