小图图 >_<

建图

```
const int MAX_NODE = 100 + 10;
const int MAX_EDGE = 10000 + 10;

struct Edge {
  int m, n, next; // m可以去除
  int w;
} E[MAX_EDGE];

int nodes[MAX_NODE]; // 顶点第一条边在E数组里的下标
int node_count, edge_count;

void add_edge(int m, int n, int w) {
  E[edge_count] = Edge { m, n, nodes[m], w };
  nodes[m] = edge_count++;
};
```

遍历

最短路

Dijkstra

```
// 最短路 Dijkstra
// 思想: 贪心算法, 维护 dis 数组为 S 到每个点的最短距离
class CmpDijkstra {
public:
  bool operator () (const PII &a, const PII &b) {
    return a.first > b.first; // 最小堆, 类似 greater<T>
  }
};

int Dijkstra(int S, int T) {
  priority_queue<PII, vector<PII>, CmpDijkstra> q;
  vector<int> dis(node_count, INF);
  q.push(make_pair(0, S)); // S 到 S 的距离为 0
  while (!q.empty()) {
    PII pair = q.top(); q.pop();
    int m = pair.second;
    if (dis[m] != INF) continue;
    dis[m] = pair.first; // S 到 m 的最短路径就是 dis[m]
    for (int i = nodes[m]; i != -1; i = E[i].next) {
      // 确定 S 到 E[i].to 的最短路径
      // 一般的 Dijkstra 会在这里有判断、赋值 dis, 但本算法直接粗暴地把结果 push 到优先队列
      // 在上面可以看到, 取出 dis[m] 后, 如果不等于 INF, 就认为 dis[m] 已经是最小值了
      q.push(make_pair(dis[m] + E[i].w, E[i].n));
    }
  }
  return dis[T] == INF ? -1 : dis[T];
};
```

Bellman-Ford

思路:

1. 定义 `OPT[i][v]` = 使用少于等于 `i` 条边的从 `v` 到 `t` 的最短路径的长度;
2. 第一种情况: 从 `v` 到 `t` 的最短路径使用了少于等于 `i - 1` 条边, 则 `OPT[i][v] = OPT[i - 1][v]`;
3. 第二种情况: 从 `v` 到 `t` 的最短路径使用了恰好 `i` 条边, 假设 `(v, w)` 是一条权重为 `W(v, w)` 的边, 则我们能够求出 `OPT[i - 1][w] + W(v, w)`, 对于所有的 `w`, 我们可以求出其最小值;
4. `OPT[i][v]` = 2 和 3 的最小值.

窝写的 Bellman-Ford 的简单实现:

```
/**
 * 求存在负权重边的图的最短路的 Bellman-Ford 算法
 * @param S 起点
 * @param T 终点
 * @return S 到 T 的最短路径
 */
int BellmanFord(int S, int T) {
  vector<int> dis(node_count, INF);
  dis[S] = 0; // S 作为源点
  for (int i = 1; i < node_count; i++) {
    for (int j = 0; j < edge_count; j++) {
      Edge e = E[j];
      if (dis[e.m] + e.w < dis[e.n]) {
        dis[e.n] = dis[e.m] + e.w;
      }
    }
  }
  return dis[T];

// 再运行一遍，可以找出负环
//  for (int j = 0; j < edge_count; j++) {
//    Edge e = E[j];
//    if (dis[e.m] + e.w < dis[e.n]) {
//      return false; // 有负环
//    }
//  }
//  return true;
}
```

SPFA

```cpp
/**
 * 求存在负权重边的图的最短路的 SPFA 算法，优化版的 Bellman-Ford
 * 同样可以用来查找负环
 * @param S 起点
 * @param T 终点
 * @return S 到 T 的最短路径
 */
int spfa(int S, int T) {
  vector<int> dis(node_count, INF);
  vector<bool> in_queue(node_count, false);
  vector<int> in(node_count, 0); // 记录一个点进入队列的次数
  dis[S] = 0;
  queue<int> q;
  q.push(S);
  while (!q.empty()) {
    int m = q.front(); q.pop();
    in_queue[m] = false;
    for (int i = nodes[m]; i != -1; i = E[i].next) {
      int n = E[i].n;
      if (dis[n] > dis[m] + E[i].w) {
        dis[n] = dis[m] + E[i].w;
        if (!in_queue[n]) {
          q.push(n);
          in_queue[n] = true;
          in[n]++;
          if (in[n] > node_count) {
            // 有负环
          }
        }
      }
    }
  }
  return dis[T] == INF ? -1 : dis[T];
}
```

最小生成树

Kruskal

```cpp
int fa[MAX_NODE];
int get_fa(int x) {
  if (x == fa[x]) return x;
  else return fa[x] = get_fa(fa[x]);
}
bool cmp_kruskal(const Edge &a, const Edge &b) {
  return a.w < b.w;
}

int kruskal() {
  int ans = 0;
  sort(E, E + edge_count, cmp_kruskal);
  for (int i = 0; i < node_count; i++) fa[i] = i; // init fa
  for (int i = 0; i < edge_count; i++) {
    int a = get_fa(E[i].m);
    int b = get_fa(E[i].n);
    if (a != b) {
      ans += E[i].w;
      fa[a] = b;
    }
  }
  int k = get_fa(0);
  for (int i = 1; i < node_count; i++) if (get_fa(i) != k) return -1;
  return ans;
}
```

拓扑排序

```cpp
/**
 * 拓扑排序
 * @param s 保存入度为 0 的节点，在调用该函数时，需要已经先把入度为 0 的节点 push 到其中
 * @param list 排序结果
 * @param in_degree 每个节点的入度
 */
void topological_sort(queue<int> &s, vector<int> &list, vector<int> &in_degree) {
  vector<bool> visited(node_count, false);
  while (!s.empty()) {
    int m = s.front(); s.pop();
    if (visited[m]) continue;
    list.push_back(m);
    visited[m] = true;
    for (int i = nodes[m]; i != -1; i = E[i].next) {
      int n = E[i].n;
      in_degree[n]--;
      if (in_degree[n] == 0) s.push(n);
    }
  }
};
```