图论算法选讲

徐沐杰

南京大学

2023 年 7 月 12 日

目录

目录

图和子图

- 图：二元组 $G = (V, E)$，其中 V 为点集，E 为边集。

图和子图

- 图：二元组 $G = (V, E)$，其中 V 为点集，E 为边集。
- $|V|$ 是图的点数，$|E|$ 是图的边数，一般称图为 $|V|$-阶图。

图和子图

- 图：二元组 $G = (V, E)$，其中 V 为点集，E 为边集。
- $|V|$ 是图的点数，$|E|$ 是图的边数，一般称图为 $|V|$-阶图。
- 有向图：E 中是顶点的有序二元组，有向边也称作弧，把 (u, v) 中 u 称作起点，v 称作终点。

图和子图

- 图：二元组 $G = (V, E)$，其中 V 为点集，E 为边集。
- $|V|$ 是图的点数，$|E|$ 是图的边数，一般称图为 $|V|$-阶图。
- 有向图：E 中是顶点的有序二元组，有向边也称作弧，把 (u, v) 中 u 称作起点，v 称作终点。
- 无向图：E 中是顶点的无序二元组。

图和子图

- 图：二元组 $G = (V, E)$，其中 V 为点集，E 为边集。
- $|V|$ 是图的点数，$|E|$ 是图的边数，一般称图为 $|V|$-阶图。
- 有向图：E 中是顶点的有序二元组，有向边也称作弧，把 (u, v) 中 u 称作起点，v 称作终点。
- 无向图：E 中是顶点的无序二元组。
- 有权图：E 中每个元素有与其配对的权值 w。

图和子图

- 图：二元组 $G = (V, E)$，其中 V 为点集，E 为边集。
- $|V|$ 是图的点数，$|E|$ 是图的边数，一般称图为 $|V|$-阶图。
- 有向图：E 中是顶点的有序二元组，有向边也称作弧，把 (u, v) 中 u 称作起点，v 称作终点。
- 无向图：E 中是顶点的无序二元组。
- 有权图：E 中每个元素有与其配对的权值 w。
- 子图：边集和点集都是原图子集的图。

图和子图

- 图：二元组 $G = (V, E)$，其中 V 为点集，E 为边集。
- $|V|$ 是图的点数，$|E|$ 是图的边数，一般称图为 $|V|$-阶图。
- 有向图：E 中是顶点的有序二元组，有向边也称作弧，把 (u, v) 中 u 称作起点，v 称作终点。
- 无向图：E 中是顶点的无序二元组。
- 有权图：E 中每个元素有与其配对的权值 w。
- 子图：边集和点集都是原图子集的图。
- 真子图：不是原图的子图。

图和子图

- 图：二元组 $G = (V, E)$，其中 V 为点集，E 为边集。
- $|V|$ 是图的点数，$|E|$ 是图的边数，一般称图为 $|V|$-阶图。
- 有向图：E 中是顶点的有序二元组，有向边也称作弧，把 (u, v) 中 u 称作起点，v 称作终点。
- 无向图：E 中是顶点的无序二元组。
- 有权图：E 中每个元素有与其配对的权值 w。
- 子图：边集和点集都是原图子集的图。
- 真子图：不是原图的子图。
- 极大 X 子图：不是其他 X 子图真子图的子图。

图和子图

- 图：二元组 $G = (V, E)$，其中 V 为点集，E 为边集。
- $|V|$ 是图的点数，$|E|$ 是图的边数，一般称图为 $|V|$-阶图。
- 有向图：E 中是顶点的有序二元组，有向边也称作弧，把 (u, v) 中 u 称作起点，v 称作终点。
- 无向图：E 中是顶点的无序二元组。
- 有权图：E 中每个元素有与其配对的权值 w。
- 子图：边集和点集都是原图子集的图。
- 真子图：不是原图的子图。
- 极大 X 子图：不是其他 X 子图真子图的子图。
- 生成子图：点集是原图点集，边集是原图边集的子集的子图。

图和子图

- 图：二元组 $G = (V, E)$，其中 V 为点集，E 为边集。
- $|V|$ 是图的点数，$|E|$ 是图的边数，一般称图为 $|V|$-阶图。
- 有向图：E 中是顶点的有序二元组，有向边也称作弧，把 (u, v) 中 u 称作起点，v 称作终点。
- 无向图：E 中是顶点的无序二元组。
- 有权图：E 中每个元素有与其配对的权值 w。
- 子图：边集和点集都是原图子集的图。
- 真子图：不是原图的子图。
- 极大 X 子图：不是其他 X 子图真子图的子图。
- 生成子图：点集是原图点集，边集是原图边集的子集的子图。
- 导出子图：边集包含了原图中两点都在点集内的所有边的子图。

图和子图

- 图：二元组 $G = (V, E)$，其中 V 为点集，E 为边集。
- $|V|$ 是图的点数，$|E|$ 是图的边数，一般称图为 $|V|$-阶图。
- 有向图：E 中是顶点的有序二元组，有向边也称作弧，把 (u, v) 中 u 称作起点，v 称作终点。
- 无向图：E 中是顶点的无序二元组。
- 有权图：E 中每个元素有与其配对的权值 w。
- 子图：边集和点集都是原图子集的图。
- 真子图：不是原图的子图。
- 极大 X 子图：不是其他 X 子图真子图的子图。
- 生成子图：点集是原图点集，边集是原图边集的子集的子图。
- 导出子图：边集包含了原图中两点都在点集内的所有边的子图。
- 边导出子图：点集包含了原图中所有和边集中的边相交的点的子图。

有向图、无向图和度

- 定向图：把无向图的每条边只留下一个方向得到的有向图。

有向图、无向图和度

- 定向图：把无向图的每条边只留下一个方向得到的有向图。
- 基础图：把有向图的每条弧换成无向边得到的无向图。

有向图、无向图和度

- 定向图：把无向图的每条边只留下一个方向得到的有向图。
- 基础图：把有向图的每条弧换成无向边得到的无向图。
- 度：与一个点相交的边数。

有向图、无向图和度

- 定向图：把无向图的每条边只留下一个方向得到的有向图。
- 基础图：把有向图的每条弧换成无向边得到的无向图。
- 度：与一个点相交的边数。
- 出度：把一个点作为起点的边的数量。

有向图、无向图和度

- 定向图：把无向图的每条边只留下一个方向得到的有向图。
- 基础图：把有向图的每条弧换成无向边得到的无向图。
- 度：与一个点相交的边数。
- 出度：把一个点作为起点的边的数量。
- 入度：把一个点作为终点的边的数量。

有向图、无向图和度

- 定向图：把无向图的每条边只留下一个方向得到的有向图。
- 基础图：把有向图的每条弧换成无向边得到的无向图。
- 度：与一个点相交的边数。
- 出度：把一个点作为起点的边的数量。
- 入度：把一个点作为终点的边的数量。
- 有向图所有点的总入度 $=$ 总出度 $=$ 边数。

有向图、无向图和度

- 定向图：把无向图的每条边只留下一个方向得到的有向图。
- 基础图：把有向图的每条弧换成无向边得到的无向图。
- 度：与一个点相交的边数。
- 出度：把一个点作为起点的边的数量。
- 入度：把一个点作为终点的边的数量。
- 有向图所有点的总入度 $=$ 总出度 $=$ 边数。
- 无向图所有点度的和 $=$ 其定向图的总入度 $+$ 总出度 $= 2 *$ 边数。

链、迹、路、连通性

- 链路：一个边的有序元组，形如 $(u,t_1)\cdots(t_n,v)$ 称作一条 $u-v$ 链路。允许重复经过点和边。

链、迹、路、连通性

- 链路：一个边的有序元组，形如 $(u,t_1)\cdots(t_n,v)$ 称作一条 $u-v$ 链路。允许重复经过点和边。
- 回路：起点和终点相同的链路。

链、迹、路、连通性

- 链路：一个边的有序元组，形如 $(u, t_1)\cdots(t_n, v)$ 称作一条 $u-v$ 链路。允许重复经过点和边。
- 回路：起点和终点相同的链路。
- 轨迹：不重复经过边的链路。

链、迹、路、连通性

- 链路：一个边的有序元组，形如 $(u, t_1)\cdots(t_n, v)$ 称作一条 $u - v$ 链路。允许重复经过点和边。
- 回路：起点和终点相同的链路。
- 轨迹：不重复经过边的链路。
- 路径：不重复经过点的轨迹。

链、迹、路、连通性

- 链路：一个边的有序元组，形如 $(u,t_1)\cdots(t_n,v)$ 称作一条 $u-v$ 链路。允许重复经过点和边。
- 回路：起点和终点相同的链路。
- 轨迹：不重复经过边的链路。
- 路径：不重复经过点的轨迹。
- 最短路：两点间边数或权值和最短的链路，又称为测地线。

链、迹、路、连通性

- 链路：一个边的有序元组，形如 $(u, t_1)\cdots(t_n, v)$ 称作一条 $u-v$ 链路。允许重复经过点和边。
- 回路：起点和终点相同的链路。
- 轨迹：不重复经过边的链路。
- 路径：不重复经过点的轨迹。
- 最短路：两点间边数或权值和最短的链路，又称为测地线。
- 距离：两点间最短路的长度。

链、迹、路、连通性

- 链路：一个边的有序元组，形如 $(u,t_1)\cdots(t_n,v)$ 称作一条 $u-v$ 链路。允许重复经过点和边。
- 回路：起点和终点相同的链路。
- 轨迹：不重复经过边的链路。
- 路径：不重复经过点的轨迹。
- 最短路：两点间边数或权值和最短的链路，又称为测地线。
- 距离：两点间最短路的长度。
- 直径：图中最长的路径的长度。

链、迹、路、连通性

- 连通图：任何点对间均存在链路的无向图。

链、迹、路、连通性

- 连通图：任何点对间均存在链路的无向图。
- 连通分量/连通块：极大连通子图。

链、迹、路、连通性

- 连通图：任何点对间均存在链路的无向图。
- 连通分量/连通块：极大连通子图。
- 弱连通：任意点对间至少单向可达的有向图。

链、迹、路、连通性

- 连通图：任何点对间均存在链路的无向图。
- 连通分量/连通块：极大连通子图。
- 弱连通：任意点对间至少单向可达的有向图。
- 强连通：任意点对间双向可达的有向图。

链、迹、路、连通性

- 连通图：任何点对间均存在链路的无向图。
- 连通分量/连通块：极大连通子图。
- 弱连通：任意点对间至少单向可达的有向图。
- 强连通：任意点对间双向可达的有向图。
- 强连通分量：极大强连通子图。

链、迹、路、连通性

- 连通图：任何点对间均存在链路的无向图。
- 连通分量/连通块：极大连通子图。
- 弱连通：任意点对间至少单向可达的有向图。
- 强连通：任意点对间双向可达的有向图。
- 强连通分量：极大强连通子图。
- 欧拉回路：经过图中所有边的闭合轨迹。

链、迹、路、连通性

- 连通图：任何点对间均存在链路的无向图。
- 连通分量/连通块：极大连通子图。
- 弱连通：任意点对间至少单向可达的有向图。
- 强连通：任意点对间双向可达的有向图。
- 强连通分量：极大强连通子图。
- 欧拉回路：经过图中所有边的闭合轨迹。
- 哈密顿回路：经过图中所有点的闭合路径。

有根树

- 有根树：有层级关系的树，以一个节点为根，从它延展出这个树的层级结构，离根越远，层级越低。

有根树

- 有根树：有层级关系的树，以一个节点为根，从它延展出这个树的层级结构，离根越远，层级越低。
- 父亲：树节点的直接上级。

有根树

- 有根树：有层级关系的树，以一个节点为根，从它延展出这个树的层级结构，离根越远，层级越低。
- 父亲：树节点的直接上级。
- 祖先：树节点的间接上级。

有根树

- 有根树：有层级关系的树，以一个节点为根，从它延展出这个树的层级结构，离根越远，层级越低。
- 父亲：树节点的直接上级。
- 祖先：树节点的间接上级。
- 儿子：树节点的直接下级。

有根树

- 有根树：有层级关系的树，以一个节点为根，从它延展出这个树的层级结构，离根越远，层级越低。
- 父亲：树节点的直接上级。
- 祖先：树节点的间接上级。
- 儿子：树节点的直接下级。
- 后代：树节点的直接和间接下级。

有根树

- 有根树：有层级关系的树，以一个节点为根，从它延展出这个树的层级结构，离根越远，层级越低。
- 父亲：树节点的直接上级。
- 祖先：树节点的间接上级。
- 儿子：树节点的直接下级。
- 后代：树节点的直接和间接下级。
- 子树：树节点和其所有后代节点的导出子图。

有根树

- 有根树：有层级关系的树，以一个节点为根，从它延展出这个树的层级结构，离根越远，层级越低。
- 父亲：树节点的直接上级。
- 祖先：树节点的间接上级。
- 儿子：树节点的直接下级。
- 后代：树节点的直接和间接下级。
- 子树：树节点和其所有后代节点的导出子图。
- 二叉树：每个非叶子节点只有至多两个儿子的节点。

有根树

- 有根树：有层级关系的树，以一个节点为根，从它延展出这个树的层级结构，离根越远，层级越低。
- 父亲：树节点的直接上级。
- 祖先：树节点的间接上级。
- 儿子：树节点的直接下级。
- 后代：树节点的直接和间接下级。
- 子树：树节点和其所有后代节点的导出子图。
- 二叉树：每个非叶子节点只有至多两个儿子的节点。
- 重心：使得树最大子树的大小最小的根。

各种特殊图

- 完全图：任意两点间都有边的无向图。n 阶完全图记作 K_n。

各种特殊图

- 完全图：任意两点间都有边的无向图。n 阶完全图记作 K_n。
- 补图：$G_1 = (V, E_1)$，$E_1 = E^c_{K_{|V|}}$。

各种特殊图

- 完全图：任意两点间都有边的无向图。n 阶完全图记作 K_n。
- 补图：$G_1 = (V, E_1)$，$E_1 = E^c_{K_{|V|}}$。
- 竞赛图：对完全图的所有边只保留一个方向后形成的有向图。

各种特殊图

- 完全图：任意两点间都有边的无向图。n 阶完全图记作 K_n。
- 补图：$G_1 = (V, E_1)$，$E_1 = E^c_{K_{|V|}}$。
- 竞赛图：对完全图的所有边只保留一个方向后形成的有向图。
- 对称有向图：$\forall (u, v) \in E, (v, u) \in E$。

各种特殊图

- 完全图：任意两点间都有边的无向图。n 阶完全图记作 K_n。
- 补图：$G_1 = (V, E_1)$，$E_1 = E^c_{K_{|V|}}$。
- 竞赛图：对完全图的所有边只保留一个方向后形成的有向图。
- 对称有向图：$\forall (u, v) \in E, (v, u) \in E$。
- 空图：完全图的补图，没有任何边的图。

各种特殊图

- 完全图：任意两点间都有边的无向图。n 阶完全图记作 K_n。
- 补图：$G_1 = (V, E_1)$，$E_1 = E^c_{K_{|V|}}$。
- 竞赛图：对完全图的所有边只保留一个方向后形成的有向图。
- 对称有向图：$\forall (u, v) \in E, (v, u) \in E$。
- 空图：完全图的补图，没有任何边的图。
- 团：完全子图的别称。

各种特殊图

- 完全图：任意两点间都有边的无向图。n 阶完全图记作 K_n。
- 补图：$G_1 = (V, E_1)$，$E_1 = E^c_{K_{|V|}}$。
- 竞赛图：对完全图的所有边只保留一个方向后形成的有向图。
- 对称有向图：$\forall (u, v) \in E, (v, u) \in E$。
- 空图：完全图的补图，没有任何边的图。
- 团：完全子图的别称。
- 独立集：空导出子图的别称。

各种特殊图

- 环：所有点度数为 2 的连通无向图，或所有点出度和入度均为 1 的强连通有向图。

各种特殊图

- 环：所有点度数为 2 的连通无向图，或所有点出度和入度均为 1 的强连通有向图。
- 树：无向无环连通图。树永远只有 $|V| - 1$ 条边。

各种特殊图

- 环：所有点度数为 2 的连通无向图，或所有点出度和入度均为 1 的强连通有向图。
- 树：无向无环连通图。树永远只有 $|V|-1$ 条边。
- 生成树：原图的既是树又是生成子图的子图。

各种特殊图

- 环：所有点度数为 2 的连通无向图，或所有点出度和入度均为 1 的强连通有向图。
- 树：无向无环连通图。树永远只有 $|V| - 1$ 条边。
- 生成树：原图的既是树又是生成子图的子图。
- DAG：有向无环图，可以拓扑排序。

各种特殊图

- 环：所有点度数为 2 的连通无向图，或所有点出度和入度均为 1 的强连通有向图。
- 树：无向无环连通图。树永远只有 $|V| - 1$ 条边。
- 生成树：原图的既是树又是生成子图的子图。
- DAG：有向无环图，可以拓扑排序。
- 链：没有点度数超过 2 的树，或者所有点出度和入度至多为 1 的弱连通有向图（无向版本的一个定向）。

各种特殊图

- 环：所有点度数为 2 的连通无向图，或所有点出度和入度均为 1 的强连通有向图。
- 树：无向无环连通图。树永远只有 $|V|-1$ 条边。
- 生成树：原图的既是树又是生成子图的子图。
- DAG：有向无环图，可以拓扑排序。
- 链：没有点度数超过 2 的树，或者所有点出度和入度至多为 1 的弱连通有向图（无向版本的一个定向）。
- 菊花：存在一个点使得所有边都与它相交的树。

各种特殊图

- 环：所有点度数为 2 的连通无向图，或所有点出度和入度均为 1 的强连通有向图。
- 树：无向无环连通图。树永远只有 $|V|-1$ 条边。
- 生成树：原图的既是树又是生成子图的子图。
- DAG：有向无环图，可以拓扑排序。
- 链：没有点度数超过 2 的树，或者所有点出度和入度至多为 1 的弱连通有向图（无向版本的一个定向）。
- 菊花：存在一个点使得所有边都与它相交的树。
- 二分图：点集能够分成两个之间没有边相连的集合的图。

各种特殊图

- 环：所有点度数为 2 的连通无向图，或所有点出度和入度均为 1 的强连通有向图。
- 树：无向无环连通图。树永远只有 $|V|-1$ 条边。
- 生成树：原图的既是树又是生成子图的子图。
- DAG：有向无环图，可以拓扑排序。
- 链：没有点度数超过 2 的树，或者所有点出度和入度至多为 1 的弱连通有向图（无向版本的一个定向）。
- 菊花：存在一个点使得所有边都与它相交的树。
- 二分图：点集能够分成两个之间没有边相连的集合的图。
- 平面图：可以被画在平面上，并使得任意两条边不相交的图。

目录

邻接矩阵和邻接表

- 点是容易标号的，存图本质上就是在保存边。

邻接矩阵和邻接表

- 点是容易标号的，存图本质上就是在保存边。
- (我们只讨论有向边，无向图可以照对称有向图处理)

邻接矩阵和邻接表

- 点是容易标号的，存图本质上就是在保存边。
- (我们只讨论有向边，无向图可以照对称有向图处理)
- 但直接拿个数组无序地存边会拖累算法效率。

邻接矩阵和邻接表

- 点是容易标号的，存图本质上就是在保存边。
- (我们只讨论有向边，无向图可以照对称有向图处理)
- 但直接拿个数组无序地存边会拖累算法效率。
- 边是点的二元组 $\rightarrow$ 以点编号为下标的二维数组。

邻接矩阵和邻接表

- 点是容易标号的，存图本质上就是在保存边。
- (我们只讨论有向边，无向图可以照对称有向图处理)
- 但直接拿个数组无序地存边会拖累算法效率。
- 边是点的二元组 $\rightarrow$ 以点编号为下标的二维数组。
- 用 $G_{i,j}$ 表示 (i,j) 边的状态：不存在、存在、或具有某权值。

邻接矩阵和邻接表

- 点是容易标号的，存图本质上就是在保存边。
- (我们只讨论有向边，无向图可以照对称有向图处理)
- 但直接拿个数组无序地存边会拖累算法效率。
- 边是点的二元组 $\rightarrow$ 以点编号为下标的二维数组。
- 用 $G_{i,j}$ 表示 (i,j) 边的状态：不存在、存在、或具有某权值。
- 显然在稠密图上该方法的空间利用率很高，且支持随机访问。

邻接矩阵和邻接表

- 点是容易标号的，存图本质上就是在保存边。
- (我们只讨论有向边，无向图可以照对称有向图处理)
- 但直接拿个数组无序地存边会拖累算法效率。
- 边是点的二元组 $\rightarrow$ 以点编号为下标的二维数组。
- 用 $G_{i,j}$ 表示 (i,j) 边的状态：不存在、存在、或具有某权值。
- 显然在稠密图上该方法的空间利用率很高，且支持随机访问。
- 但是稀疏图（如树）上将会浪费平方级的空间，怎么办?

邻接矩阵和邻接表

- 点是容易标号的，存图本质上就是在保存边。
- (我们只讨论有向边，无向图可以照对称有向图处理)
- 但直接拿个数组无序地存边会拖累算法效率。
- 边是点的二元组 $\rightarrow$ 以点编号为下标的二维数组。
- 用 $G_{i,j}$ 表示 (i,j) 边的状态：不存在、存在、或具有某权值。
- 显然在稠密图上该方法的空间利用率很高，且支持随机访问。
- 但是稀疏图（如树）上将会浪费平方级的空间，怎么办?
- 放弃第二维上的随机访问，使用表来存储一个点的所有出边。

邻接矩阵和邻接表

- 点是容易标号的，存图本质上就是在保存边。
- (我们只讨论有向边，无向图可以照对称有向图处理)
- 但直接拿个数组无序地存边会拖累算法效率。
- 边是点的二元组 $\rightarrow$ 以点编号为下标的二维数组。
- 用 $G_{i,j}$ 表示 (i,j) 边的状态：不存在、存在、或具有某权值。
- 显然在稠密图上该方法的空间利用率很高，且支持随机访问。
- 但是稀疏图（如树）上将会浪费平方级的空间，怎么办？
- 放弃第二维上的随机访问，使用表来存储一个点的所有出边。
- 链表/动态数组！

邻接表：动态数组

```
// e.v 边的终点
// e.attr 边维护的其他信息（权值等）
vector <edge> G[MAXN];
for (edge e : G[u])
  printf("to: %d attr: %d\n", e.v, e.attr);
// 增加边 (u, v, attr)
G[u].push_back(edge(v, attr));
```

邻接表：边表

- 链表虽然原理简单，但是实现却是很多人的噩梦（包括我）。

邻接表：边表

- 链表虽然原理简单，但是实现却是很多人的噩梦（包括我）。
- 但是考虑到我们不太关心点的邻接边之间的顺序，所以只需要不断在每个点的链表表头插入新的边就行。

邻接表：边表

- 链表虽然原理简单，但是实现却是很多人的噩梦（包括我）。
- 但是考虑到我们不太关心点的邻接边之间的顺序，所以只需要不断在每个点的链表表头插入新的边就行。
- 此时我们可以任意规定存放边的顺序，所以通过把正向边和反向边放在一起，可以达到快速访问反边信息的效果（利好一些无向图，以及 dinic 等基于反边的网络流算法）。

边表的实现

- 相对来说实现还是简单的。边表也被称作链式前向星。

```
struct edge{
  int to, ne;
}e[MAXE];
int h[MAXN], ecnt=0;
void add(int a, int b) {
  e[++ecnt] = (edge) {.to = b, .ne = h[a]}; h[a] = ecnt;
}
void iter(int x) {
  for (int i = h[x]; i; i = e[i].ne)
    printf("%d\n", e[i].to);
}
```

图的遍历

- 学习搜索的时候已经知道：搜索的本质是对状态图的隐式遍历。

图的遍历

- 学习搜索的时候已经知道：搜索的本质是对状态图的隐式遍历。
- 因此对实际存在的图的遍历，只需同样的执行 BFS 和 DFS 算法。

图的遍历

- 学习搜索的时候已经知道：搜索的本质是对状态图的隐式遍历。
- 因此对实际存在的图的遍历，只需同样的执行 BFS 和 DFS 算法。
- 回忆：BFS 为广度优先，优先拓展先访问到的状态。

图的遍历

- 学习搜索的时候已经知道：搜索的本质是对状态图的隐式遍历。
- 因此对实际存在的图的遍历，只需同样的执行 BFS 和 DFS 算法。
- 回忆：BFS 为广度优先，优先拓展先访问到的状态。
- 回忆：DFS 为深度优先，立即拓展目前访问到的状态。

图的遍历

- 学习搜索的时候已经知道：搜索的本质是对状态图的隐式遍历。
- 因此对实际存在的图的遍历，只需同样的执行 BFS 和 DFS 算法。
- 回忆：BFS 为广度优先，优先拓展先访问到的状态。
- 回忆：DFS 为深度优先，立即拓展目前访问到的状态。
- 由于图的状态数（即点数）很少，图上可以很容易的把访问信息记录下来，防止重复遍历，且可以使用各种记忆化手段进行递推。

图的遍历：DFS

```
bool vis[MAXN];
void dfs(int x) {
  vis[x] = true;
  for (int i = h[x]; i; i = e[i].ne) {
    if (vis[e[i].to]) continue;
    dfs(e[i].to);
  }
  return ;
}
```

图的遍历：BFS

```
queue<int> q;
bool vis[MAXN];
void bfs(int s) {
  q.push(s);
  vis[s] = true;
  while (!q.empty()) {
    int u = q.front(); q.pop();
    for (int i = h[u]; i; i = e[i].ne) {
      if (!vis[e[i].to]) {
        q.push(e[i].to);
        vis[e[i].to] = true;
      }
    }
  }
}
```

拓扑排序

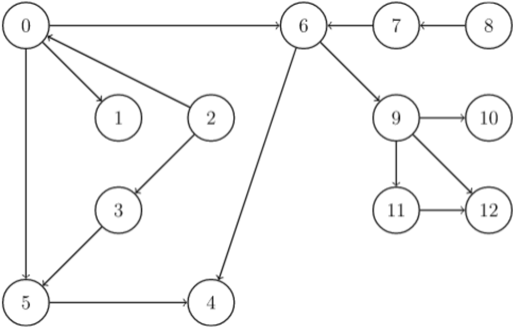
- 有向无环图（DAG）具有很好的性质，
  特别是当其表示某种依赖或顺序关系的时候。

拓扑排序

- 有向无环图（DAG）具有很好的性质，
  特别是当其表示某种依赖或顺序关系的时候。
- DAG 的单向可达关系定义了一种「偏序」，
  仅由“小的”到达“大的”那一边。

拓扑排序

- 有向无环图（DAG）具有很好的性质，
  特别是当其表示某种依赖或顺序关系的时候。
- DAG 的单向可达关系定义了一种「偏序」，
  仅由“小的”到达“大的”那一边。
- 可以用来解决各种无环形依赖的递推（DP）问题。



拓扑排序：代码实现

```
// q 起初存着所有入度为 0 的点
while (!q.empty()) {
  int now = q.front(); q.pop();
  for (int i = h[now]; i; i = e[i].ne) {
    // 某些递推关系
    f[e[i].to] = dp(f[now], f[e[i].to]);
    // 产生新的入度为 0 点就入队
    if (!--in[e[i].to]) {
      topo[e[i].to] = ++topo_time;
      q.push(e[i].to);
    }
  }
}
```

树的遍历

树的遍历比较特殊。因为树无环，所以只要不回头走就不会产生重复。如以下代码展示：

```
void dfs(int x, int fa) {
  dfs_time[x] = ++dfs_num;
  for (int i = h[x]; i; i = e[i].to) {
    // 不访问祖先
    if (e[i].to == fa) continue;
    dfs(e[i].to, x);
  }
}
```

树的 dfs 序

- 刚刚的遍历按照访问的顺序给每个节点记录了一个顺序，即 **dfs 序**。

树的 dfs 序

- 刚刚的遍历按照访问的顺序给每个节点记录了一个顺序，即 **dfs 序**。
- 按照深度优先搜索的规则知道，整个子树在 dfs 序上是连续的。

树的 dfs 序

- 刚刚的遍历按照访问的顺序给每个节点记录了一个顺序，即 **dfs 序**。
- 按照深度优先搜索的规则知道，整个子树在 dfs 序上是连续的。
- 所以维护子树信息可以转化为维护区间信息。

树的 dfs 序

- 刚刚的遍历按照访问的顺序给每个节点记录了一个顺序，即 **dfs 序**。
- 按照深度优先搜索的规则知道，整个子树在 dfs 序上是连续的。
- 所以维护子树信息可以转化为维护区间信息。
- 对于每个节点，我们记录以它为根的子树在 dfs 序列中最后出现的一次。这实际上就是此点在 dfs 中的退出时间。

树的 dfs 序

- 刚刚的遍历按照访问的顺序给每个节点记录了一个顺序，即 **dfs 序**。
- 按照深度优先搜索的规则知道，整个子树在 dfs 序上是连续的。
- 所以维护子树信息可以转化为维护区间信息。
- 对于每个节点，我们记录以它为根的子树在 dfs 序列中最后出现的一次。这实际上就是此点在 dfs 中的退出时间。
- 这实际上是一种隐式的欧拉序，下面我们来讨论另一种欧拉序。

树的欧拉序

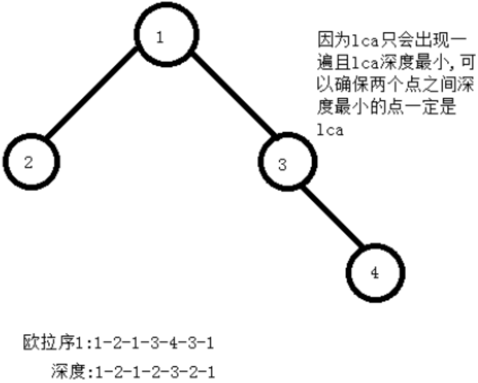
- 考虑在 dfs 序的基础之上，在每次子节点 dfs 退出重新进入自己的程序上下文时记录一次标记。

树的欧拉序

- 考虑在 dfs 序的基础之上，在每次子节点 dfs 退出重新进入自己的程序上下文时记录一次标记。
- 容易发现此时可以把 LCA 归结为欧拉序上的 RMQ 问题。

树的欧拉序

- 考虑在 dfs 序的基础之上，在每次子节点 dfs 退出重新进入自己的程序上下文时记录一次标记。
- 容易发现此时可以把 LCA 归结为欧拉序上的 RMQ 问题。



树的直径（选讲）

树的重心（选讲）

树上启发式合并（选讲）

目录

概念

- 回忆（生成树）：使原图连通的无环生成子图。

概念

- 回忆（生成树）：使原图连通的无环生成子图。
- 最小生成树：有权图上权值和最小的生成树。

概念

- 回忆（生成树）：使原图连通的无环生成子图。
- 最小生成树：有权图上权值和最小的生成树。
- 不像最短路/最长路，最大/最小生成树是可以容易互相转化的。

概念

- 回忆（生成树）：使原图连通的无环生成子图。
- 最小生成树：有权图上权值和最小的生成树。
- 不像最短路/最长路，最大/最小生成树是可以容易互相转化的。
- 今天用 MST 来统称最大/最小生成树问题。

目录

Prim

- 考虑从某起点出发不断拓展连通块。

Prim

- 考虑从某起点出发不断拓展连通块。
- 根据直觉，应该贪心的选择加入连通块代价最小的边。

Prim

- 考虑从某起点出发不断拓展连通块。
- 根据直觉，应该贪心的选择加入连通块代价最小的边。
- 复杂度 $O(n^2)$，利用堆优化可在稀疏图上做到 $O(m \log n)$。

Prim

- 考虑从某起点出发不断拓展连通块。
- 根据直觉，应该贪心的选择加入连通块代价最小的边。
- 复杂度 $O(n^2)$，利用堆优化可在稀疏图上做到 $O(m\log n)$。
- 证明正确性？(回忆：证明贪心正确性)

Prim

- 考虑从某起点出发不断拓展连通块。
- 根据直觉，应该贪心的选择加入连通块代价最小的边。
- 复杂度 $O(n^2)$，利用堆优化可在稀疏图上做到 $O(m\log n)$。
- 证明正确性？(回忆：证明贪心正确性)
- 考虑另一方案和 Prim 所做的第一次不同选择。

Prim

- 考虑从某起点出发不断拓展连通块。
- 根据直觉，应该贪心的选择加入连通块代价最小的边。
- 复杂度 $O(n^2)$，利用堆优化可在稀疏图上做到 $O(m\log n)$。
- 证明正确性？(回忆：证明贪心正确性)
- 考虑另一方案和 Prim 所做的第一次不同选择。
- 因为该方案没有选择最短边，假设选择了其余一边，那么该边一定在最短边两端点的路径上。

Prim

- 考虑从某起点出发不断拓展连通块。
- 根据直觉，应该贪心的选择加入连通块代价最小的边。
- 复杂度 $O(n^2)$，利用堆优化可在稀疏图上做到 $O(m\log n)$。
- 证明正确性？(回忆：证明贪心正确性)
- 考虑另一方案和 Prim 所做的第一次不同选择。
- 因为该方案没有选择最短边，假设选择了其余一边，那么该边一定在最短边两端点的路径上。
- 显然此时换为 Prim 的选择是合法的，且会更优。

目录

Kruskal

- 分布式贪心~~（很 Web 3.0，泰裤辣）~~：直接对边按照权值排序。用并查集维护当前的权值情况，能合并就合并，合并完了就输出答案。

Kruskal

- 分布式贪心~~（很 Web 3.0，泰裤辣）~~：直接对边按照权值排序。用并查集维护当前的权值情况，能合并就合并，合并完了就输出答案。
- 稀疏图复杂度 $O(m\log n)$，稠密图复杂度 $O(n^2\lg n)$。

Kruskal

- 分布式贪心~~（很 Web 3.0，泰裤辣）~~：直接对边按照权值排序。用并查集维护当前的权值情况，能合并就合并，合并完了就输出答案。
- 稀疏图复杂度 $O(m \log n)$，稠密图复杂度 $O(n^2 \lg n)$。
- 在稀疏图上表现和 Prim 基本一致，稠密图上略差些（因为需要排序）。

Kruskal

- 分布式贪心~~（很 Web 3.0，泰裤辣）~~：直接对边按照权值排序。用并查集维护当前的权值情况，能合并就合并，合并完了就输出答案。
- 稀疏图复杂度 $O(m \log n)$，稠密图复杂度 $O(n^2 \lg n)$。
- 在稀疏图上表现和 Prim 基本一致，稠密图上略差些（因为需要排序）。
- 在我初中的时候，我觉得这是最好写的图论算法（x）。

Kruskal

- 分布式贪心~~（很 Web 3.0，泰裤辣）~~：直接对边按照权值排序。用并查集维护当前的权值情况，能合并就合并，合并完了就输出答案。
- 稀疏图复杂度 $O(m\log n)$，稠密图复杂度 $O(n^2\lg n)$。
- 在稀疏图上表现和 Prim 基本一致，稠密图上略差些（因为需要排序）。
- 在我初中的时候，我觉得这是最好写的图论算法（x）。
- 证明正确性又是另外一个故事了（回忆：证明贪心算法的正确性）。

Kruskal

- 分布式贪心~~（很 Web 3.0，泰裤辣）~~：直接对边按照权值排序。用并查集维护当前的权值情况，能合并就合并，合并完了就输出答案。
- 稀疏图复杂度 $O(m \log n)$，稠密图复杂度 $O(n^2 \lg n)$。
- 在稀疏图上表现和 Prim 基本一致，稠密图上略差些（因为需要排序）。
- 在我初中的时候，我觉得这是最好写的图论算法（x）。
- 证明正确性又是另外一个故事了（回忆：证明贪心算法的正确性）。
- Kruskal 的求解过程极好地体现了 MST 的性质。藉由它可以证明最小生成树在图上优化/约束问题的众多应用的正确性。

正确性

定理 (Kruskal 引理)

从小到大考虑一个生成树的每条边，如果**严格不满足** *Kruskal* 算法流程 (即在两条严格不等的合并路径中选择较大的)，一定不是最小生成树。

正确性

定理 (Kruskal 引理)

从小到大考虑一个生成树的每条边，如果**严格不满足** *Kruskal* 算法流程 (即在两条严格不等的合并路径中选择较大的)，一定不是最小生成树。

证明.

考虑第一次被跳过的边。

正确性

定理 (Kruskal 引理)

从小到大考虑一个生成树的每条边，如果**严格不满足** *Kruskal* 算法流程 (即在两条严格不等的合并路径中选择较大的)，一定不是最小生成树。

证明.

考虑第一次被跳过的边。
由于小于它的边没有办法将它的两端点连接在同一个连通块里，于是是在该生成树上，此边的两端点的路径上一定存在一条比权值更大的边。

正确性

定理 (Kruskal 引理)

从小到大考虑一个生成树的每条边，如果**严格不满足** *Kruskal* 算法流程 (即在两条严格不等的合并路径中选择较大的)，一定不是最小生成树。

证明.

考虑第一次被跳过的边。
由于小于它的边没有办法将它的两端点连接在同一个连通块里，于是在该生成树上，此边的两端点的路径上一定存在一条比权值更大的边。
此时考虑添加此边并删去该权值更大的边，便得到了一个更小的生成树，所以该方案构造出的树不是最小生成树。 □

最小生成树计数

定理 (Kruskal 引理)

从小到大考虑一个生成树的每条边，如果**严格不满足** *Kruskal* 算法流程 (即在两条严格不等的合并路径中选择较大的)，一定不是最小生成树。

最小生成树计数

定理 (Kruskal 引理)

从小到大考虑一个生成树的每条边，如果**严格不满足** *Kruskal* 算法流程 (即在两条严格不等的合并路径中选择较大的)，一定不是最小生成树。

例 (MSTcount)

现有一个 n 阶有权无向图，问该图共有多少种最小生成树。

最小生成树计数

定理 (Kruskal 引理)

从小到大考虑一个生成树的每条边，如果**严格不满足** *Kruskal* 算法流程 (即在两条严格不等的合并路径中选择较大的)，一定不是最小生成树。

例 (MSTcount)

现有一个 n 阶有权无向图，问该图共有多少种最小生成树。

解 (MSTcount)

- 最小生成树的差异出现在同一权值大小的边的选择之上。

最小生成树计数

定理 (Kruskal 引理)

从小到大考虑一个生成树的每条边，如果**严格不满足** *Kruskal* 算法流程(即在两条严格不等的合并路径中选择较大的)，一定不是最小生成树。

例 (MSTcount)

现有一个 n 阶有权无向图，问该图共有多少种最小生成树。

解 (MSTcount)

- 最小生成树的差异出现在同一权值大小的边的选择之上。
- 考虑同一权值的边集合前后，无论如何选择图的连通性都将一致。

最小生成树计数

定理 (Kruskal 引理)

从小到大考虑一个生成树的每条边，如果**严格不满足** *Kruskal* 算法流程 (即在两条严格不等的合并路径中选择较大的)，一定不是最小生成树。

例 (MSTcount)

现有一个 n 阶有权无向图，问该图共有多少种最小生成树。

解 (MSTcount)

- 最小生成树的差异出现在同一权值大小的边的选择之上。
- 考虑同一权值的边集合前后，无论如何选择图的连通性都将一致。
- 用基尔霍夫矩阵树定理统计连接连通块的方案数即可。

货车运输问题

- Kruskal 优先按权值顺序考虑边的过程体现了 MST 本身的贪心性质。于是因此我们可以利用 MST 的贪心性质解决一些最优化问题。

货车运输问题

- Kruskal 优先按权值顺序考虑边的过程体现了 MST 本身的贪心性质。于是因此我们可以利用 MST 的贪心性质解决一些最优化问题。

例 (Transport)

现有一个 n 阶有权无向图，有 q 次询问，每次需要选择一条路径，使得该路径上的最小边权最大化。$n, q \leq 100000$。

货车运输问题

- Kruskal 优先按权值顺序考虑边的过程体现了 MST 本身的贪心性质。于是因此我们可以利用 MST 的贪心性质解决一些最优化问题。

例 (Transport)

现有一个 n 阶有权无向图，有 q 次询问，每次需要选择一条路径，使得该路径上的最小边权最大化。$n, q \le 100000$。

解 (Transport)

- 先考虑对于一对点，如何找到满足要求的路径。

货车运输问题

- Kruskal 优先按权值顺序考虑边的过程体现了 MST 本身的贪心性质。于是因此我们可以利用 MST 的贪心性质解决一些最优化问题。

例 (Transport)

现有一个 n 阶有权无向图，有 q 次询问，每次需要选择一条路径，使得该路径上的最小边权最大化。$n, q \leq 100000$。

解 (Transport)

- 先考虑对于一对点，如何找到满足要求的路径。
- 考虑 *Kruskal* 做最大生成树的过程：贪心地按边权从大到小加边。

货车运输问题

- Kruskal 优先按权值顺序考虑边的过程体现了 MST 本身的贪心性质。于是因此我们可以利用 MST 的贪心性质解决一些最优化问题。

例 (Transport)

现有一个 n 阶有权无向图，有 q 次询问，每次需要选择一条路径，使得该路径上的最小边权最大化。$n, q \le 100000$。

解 (Transport)

- 先考虑对于一对点，如何找到满足要求的路径。
- 考虑 *Kruskal* 做最大生成树的过程：贪心地按边权从大到小加边。
- 当两点间第一次连通时，我们可以证明此时令它们连通的唯一路径即为所求路径。因为在此时，原图中任意一条连通它们的路径都至少存在一条边权值比它们小。

货车运输问题

例 (Transport)

现有一个 n 阶有权无向图，有 q 次询问，每次需要选择一条路径，使得该路径上的最小边权最大化。$n, q \leq 100000$。

解 (Transport)

- 在进行最小生成树计数时发现，任意一颗 *MST* 都可以由 *Kruskal* 算法得出。于是我们可以证明对于任何两点，任意最大生成树上的路径对它们来说都是最优的。

例 (Transport)

现有一个 n 阶有权无向图，有 q 次询问，每次需要选择一条路径，使得该路径上的最小边权最大化。$n, q \leq 100000$。

解 (Transport)

- 在进行最小生成树计数时发现，任意一颗 *MST* 都可以由 *Kruskal* 算法得出。于是我们可以证明对于任何两点，任意最大生成树上的路径对它们来说都是最优的。
- 所以只需先求解最大生成树，再使用倍增预处理/计算树上最大值和 *LCA* 即可。

Kruskal 重构树

- 考虑把 Kruskal 算法的过程用数据结构保存下来。

Kruskal 重构树

- 考虑把 Kruskal 算法的过程用数据结构保存下来。
- 我们在 Kruskal 合并两个连通块时建立一个新点，把点权设置为连接的边的边权，并令两个连通块成为它的儿子。

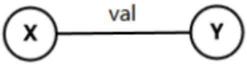
Kruskal 重构树

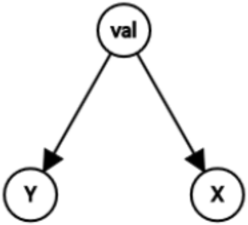
- 考虑把 Kruskal 算法的过程用数据结构保存下来。
- 我们在 Kruskal 合并两个连通块时建立一个新点，把点权设置为连接的边的边权，并令两个连通块成为它的儿子。
- 由此，最后一次合并的边在新树上就是根，而原节点都是叶子节点。越高的点的点权越大，显然，这是一个**二叉堆**。

Kruskal 重构树

- 考虑把 Kruskal 算法的过程用数据结构保存下来。
- 我们在 Kruskal 合并两个连通块时建立一个新点，把点权设置为连接的边的边权，并令两个连通块成为它的儿子。
- 由此，最后一次合并的边在新树上就是根，而原节点都是叶子节点。越高的点的点权越大，显然，这是一个**二叉堆**。
- 不难发现，不同叶子节点间的 LCA 就是它们在 Kruskal 算法执行过程中第一次连通的边权值，也就是它们路径上最大边的最小值。

- 考虑把 Kruskal 算法的过程用数据结构保存下来。
- 我们在 Kruskal 合并两个连通块时建立一个新点，把点权设置为连接的边的边权，并令两个连通块成为它的儿子。
- 由此，最后一次合并的边在新树上就是根，而原节点都是叶子节点。越高的点的点权越大，显然，这是一个**二叉堆**。
- 不难发现，不同叶子节点间的 LCA 就是它们在 Kruskal 算法执行过程中第一次连通的边权值，也就是它们路径上最大边的最小值。
- 用这一性质，我们可以解决比货车运输问题更进一步的类似问题。





爬山

例 (Climb)

给一个 n 阶有权（点和边都有权）无向图，询问某个点不经过边权超过 k 的边所能达到的所有点的权值和。

爬山

例 (Climb)

给一个 n 阶有权（点和边都有权）无向图，询问某个点不经过边权超过 k 的边所能达到的所有点的权值和。

解 (Climb)

- 我们发现在重构树上，满足这一条件的点就是和该点的 lca 权值不超过 k 的点。

爬山

例 (Climb)

给一个 n 阶有权（点和边都有权）无向图，询问某个点不经过边权超过 k 的边所能达到的所有点的权值和。

解 (Climb)

- 我们发现在重构树上，满足这一条件的点就是和该点的 lca 权值不超过 k 的点。
- 考虑该点最大且点权不超过 k 的祖先，满足条件的点都在以此祖先为根的子树里。

例 (Climb)

给一个 n 阶有权（点和边都有权）无向图，询问某个点不经过边权超过 k 的边所能达到的所有点的权值和。

解 (Climb)

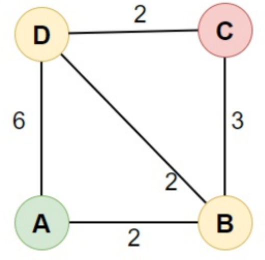
- 我们发现在重构树上，满足这一条件的点就是和该点的 lca 权值不超过 k 的点。
- 考虑该点最大且点权不超过 k 的祖先，满足条件的点都在以此祖先为根的子树里。
- 这些点在 DFS 序上是连续的，问题转化为区间求和问题。

目录

目录

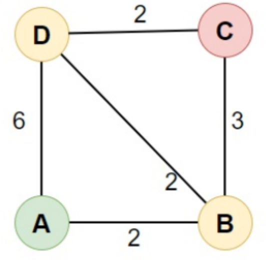
基础概念

- 问题定义：给出一个点，计算该点到其他所有点的最短路径。



基础概念

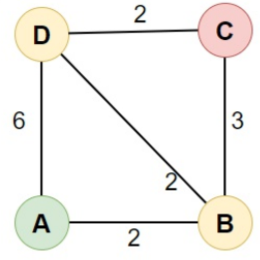
- 问题定义：给出一个点，计算该点到其他所有点的最短路径。



- 答案可以被写作一个函数 $d(u)$，表示从起点到 u 的最短路径。

基础概念

- 问题定义：给出一个点，计算该点到其他所有点的最短路径。



- 答案可以被写作一个函数 $d(u)$，表示从起点到 u 的最短路径。
- 性质：显然 $d(S)=0$，且满足 $\forall(u,v,w)\in E, d(u)+w\geq d(v)$。这叫做 **三角形不等式**，它说明了最短路的最短性（废话）。

描述问题状态

- 假设我们所知道的距离函数为 f。

描述问题状态

- 假设我们所知道的距离函数为 f。
- 一开始我们只知道「不经过任何其他点」的最短路径。

描述问题状态

- 假设我们所知道的距离函数为 f。
- 一开始我们只知道「不经过任何其他点」的最短路径。
- 进展：对于任何一点，如果满足 $\exists(u, v, w) \in E, f(u) + w < f(v)$，则令 $f(v) := f(u) + w$。这个操作叫做「松弛」，即引入更短的路径来松弛原来更长的路径。

描述问题状态

- 假设我们所知道的距离函数为 f。
- 一开始我们只知道「不经过任何其他点」的最短路径。
- 进展：对于任何一点，如果满足 $\exists(u, v, w) \in E, f(u) + w < f(v)$，则令 $f(v) := f(u) + w$。这个操作叫做「松弛」，即引入更短的路径来松弛原来更长的路径。
- 悲剧：松弛操作到底会执行多少次？松弛到最后能否算出最短路？

Bellman Ford

定理 (最短路判定引理)

如果 f 满足 $f(S)=0$，又 $\forall(u,v,w)\in E, f(u)+w\geq f(v)$，且 $\exists(u,v,w)\in E, f(u)+w=f(v)$，那么 $f=d$。

定理 (最短路判定引理)

如果 f 满足 $f(S)=0$，又 $\forall(u,v,w)\in E, f(u)+w\geq f(v)$，且 $\exists(u,v,w)\in E, f(u)+w=f(v)$，那么 $f=d$。

证明.

考虑一个从 S 到 u 的路径 $(S,t_1,w_1),\cdots,(t_n,u,w_n)$，发现必有 $\sum_{i\in[n]} w_i\geq f(u)-f(S)=f(u)$。且由题目条件也很容易知道长度为 $f(u)$ 的路径是存在的。证毕。 $\square$

Bellman Ford

定理 (最短路判定引理)

如果 f 满足 $f(S)=0$，又 $\forall(u,v,w)\in E, f(u)+w\geq f(v)$，且 $\exists(u,v,w)\in E, f(u)+w=f(v)$，那么 $f=d$。

证明.

考虑一个从 S 到 u 的路径 $(S,t_1,w_1),\cdots,(t_n,u,w_n)$，发现必有 $\sum_{i\in[n]} w_i \geq f(u)-f(S)=f(u)$。且由题目条件也很容易知道长度为 $f(u)$ 的路径是存在的。证毕。 $\square$

定理 (最短路松弛引理)

如果一个从 S 到 u 的路径 $(S,t_1,w_1),\cdots,(t_n,u,w_n)$ 是松弛边序列的子序列（即按序被松弛过），那么有 $f(u)-f(S)\leq\sum_{i\in n} w_i$。即整条路径都被松弛。

Bellman Ford

- 如果最短路径被松弛了，那么我们就成功求出了最短路。

Bellman Ford

- 如果最短路径被松弛了，那么我们就成功求出了最短路。
- 如何保证最短路径被松弛?

Bellman Ford

- 如果最短路径被松弛了，那么我们就成功求出了最短路。
- 如何保证最短路径被松弛?
- 最短路长度不超过 n，否则存在负环。

Bellman Ford

- 如果最短路径被松弛了，那么我们就成功求出了最短路。
- 如何保证最短路径被松弛?
- 最短路长度不超过 n，否则存在负环。
- 直接按顺序松弛所有边 n 次，任何长度为 n 的边序列都会是松弛序列的子序列。

Bellman Ford

- 如果最短路径被松弛了，那么我们就成功求出了最短路。
- 如何保证最短路径被松弛?
- 最短路长度不超过 n，否则存在负环。
- 直接按顺序松弛所有边 n 次，任何长度为 n 的边序列都会是松弛序列的子序列。
- 根据引理，算法是正确的。

SPFA：队列优化 BF

定理 (最短路判定引理)

如果 f 满足 $f(S)=0$，又 $\forall(u,v,w)\in E, f(u)+w\geq f(v)$，且 $\exists(u,v,w)\in E, f(u)+w=f(v)$，那么 $f=d$。

SPFA：队列优化 BF

定理 (最短路判定引理)

如果 f 满足 $f(S)=0$，又 $\forall(u,v,w)\in E, f(u)+w\geq f(v)$，且 $\exists(u,v,w)\in E, f(u)+w=f(v)$，那么 $f=d$。

- 根据引理，Bellman Ford 某轮发现没有边可以松弛时就可以停了。

SPFA：队列优化 BF

定理 (最短路判定引理)

如果 f 满足 $f(S)=0$，又 $\forall(u,v,w)\in E, f(u)+w\geq f(v)$，且 $\exists(u,v,w)\in E, f(u)+w=f(v)$，那么 $f=d$。

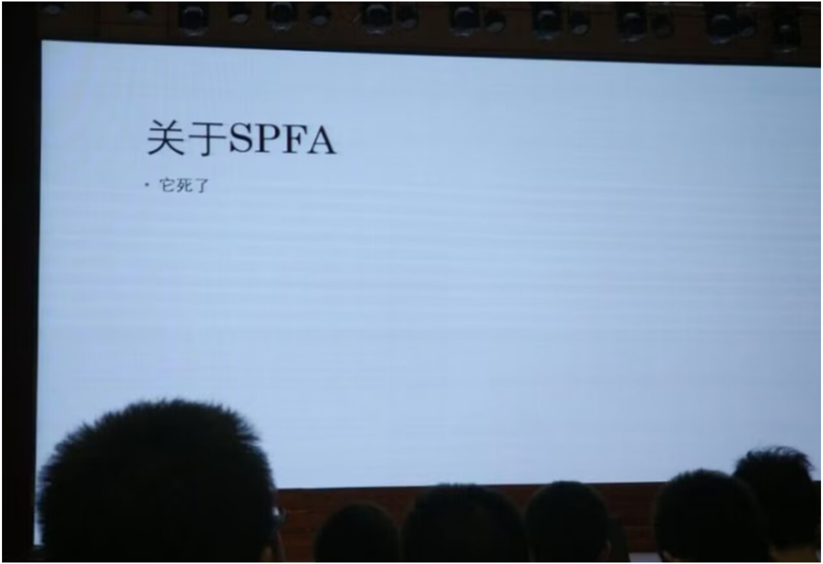
- 根据引理，Bellman Ford 某轮发现没有边可以松弛时就可以停了。
- 此外，Bellman Ford 也并没有利用上一轮的信息，我们可以借助广度优先搜索，每次拓展被更新的点，因为本轮没被更新的下一轮必定不会被更新。**SPFA 就是跳过不更新点的 Bellman Ford**。

定理 (最短路判定引理)

如果 f 满足 $f(S)=0$，又 $\forall(u,v,w)\in E, f(u)+w\geq f(v)$，且 $\exists(u,v,w)\in E, f(u)+w=f(v)$，那么 $f=d$。

- 根据引理，Bellman Ford 某轮发现没有边可以松弛时就可以停了。
- 此外，Bellman Ford 也并没有利用上一轮的信息，我们可以借助广度优先搜索，每次拓展被更新的点，因为本轮没被更新的下一轮必定不会被更新。**SPFA 就是跳过不更新点的 Bellman Ford**。
- 如果一个点被重新入队 n 轮，那么存在负环。

SPFA 已死?



SPFA 已死?

- 一点历史遗留问题：当年段老师在集训队论文里“发明”SPFA 时，错误地证明了它的复杂度是近线性的。

SPFA 已死?

- 一点历史遗留问题：当年段老师在集训队论文里“发明”SPFA 时，错误地证明了它的复杂度是近线性的。
- 事实上，SPFA 的渐进最坏复杂度是 $\Omega(nm)$ 的，和 BF 算法相同。

SPFA 已死?

- 一点历史遗留问题：当年段老师在集训队论文里“发明”SPFA 时，错误地证明了它的复杂度是近线性的。
- 事实上，SPFA 的渐进最坏复杂度是 $\Omega(nm)$ 的，和 BF 算法相同。
- 但是 SPFA 在部分图上仍然具有很好的表现，对于较弱的数据有很好的效果。~~（但是正式的赛事都会卡 SPFA）~~

SPFA 已死?

- 一点历史遗留问题：当年段老师在集训队论文里“发明”SPFA 时，错误地证明了它的复杂度是近线性的。
- 事实上，SPFA 的渐进最坏复杂度是 $\Omega(nm)$ 的，和 BF 算法相同。
- 但是 SPFA 在部分图上仍然具有很好的表现，对于较弱的数据有很好的效果。~~(但是正式的赛事都会卡 SPFA)~~
- 下午没卡 SPFA，因为反正也过不去......

SPFA 已死?

- 一点历史遗留问题：当年段老师在集训队论文里“发明”SPFA 时，错误地证明了它的复杂度是近线性的。
- 事实上，SPFA 的渐进最坏复杂度是 $\Omega(nm)$ 的，和 BF 算法相同。
- 但是 SPFA 在部分图上仍然具有很好的表现，对于较弱的数据有很好的效果。~~（但是正式的赛事都会卡 SPFA）~~
- 下午没卡 SPFA，因为反正也过不去……
- 几个优化：SLF，LLF，堆优化，DFS 优化。这些都是启发式的优化，看起来优化了，但可能令 SPFA 的复杂度退化至指数级。

SPFA 已死?

- 一点历史遗留问题：当年段老师在集训队论文里“发明”SPFA 时，错误地证明了它的复杂度是近线性的。
- 事实上，SPFA 的渐进最坏复杂度是 $\Omega(nm)$ 的，和 BF 算法相同。
- 但是 SPFA 在部分图上仍然具有很好的表现，对于较弱的数据有很好的效果。~~（但是正式的赛事都会卡 SPFA）~~
- 下午没卡 SPFA，因为反正也过不去……
- 几个优化：SLF，LLF，堆优化，DFS 优化。这些都是启发式的优化，看起来优化了，但可能令 SPFA 的复杂度退化至指数级。
- 我不建议在非乱搞情形下使用 SPFA（也不建议试图应用这些优化）。如果你一定要使用，请把它当作 Bellman Ford 的另一版本。

差分约束（选讲）

例 (bonds)

有 n 个变量和 m 条约束关系，形如 $x_i - x_j \geq b$，给出符合条件的取值。

差分约束（选讲）

例 (bonds)

有 n 个变量和 m 条约束关系，形如 $x_i - x_j \geq b$，给出符合条件的取值。

解 (bonds)

- 求取最短路的过程其实是一个满足三角不等式的过程。其形式 $h(u) - h(v) \geq -w(u, v)$ 事实上就是在描述两个离散变量的取值约束。

差分约束（选讲）

例 (bonds)

有 n 个变量和 m 条约束关系，形如 $x_i - x_j \geq b$，给出符合条件的取值。

解 (bonds)

- 求取最短路的过程其实是一个满足三角不等式的过程。其形式 $h(u) - h(v) \geq -w(u, v)$ 事实上就是在描述两个离散变量的取值约束。
- 因此对于题目所述的差分约束关系，直接建图跑最短路即可，如果图中有负环，说明这个差分约束关系无解。

所罗门王的宝藏（选讲）

贪心：Dijkstra

- 上面提到的最短路算法都没有对图作出除无负环外的要求，但实际中见到的图往往连负权边都没有，如何利用这个性质？

贪心：Dijkstra

- 上面提到的最短路算法都没有对图作出除无负环外的要求，但实际中见到的图往往连负权边都没有，如何利用这个性质？
- 当没有负权边时，容易发现当前 f 值最小的点再也没有可能被松弛了（因为其他的 f 加上一个非负值后不会变小），所以贪心地用这个点松弛其它点后就丢弃该点，不再考虑。

贪心：Dijkstra

- 上面提到的最短路算法都没有对图作出除无负环外的要求，但实际中见到的图往往连负权边都没有，如何利用这个性质？
- 当没有负权边时，容易发现当前 f 值最小的点再也没有可能被松弛了（因为其他的 f 加上一个非负值后不会变小），所以贪心地用这个点松弛其它点后就丢弃该点，不再考虑。
- 重复这个步骤 n 次，就可以得到所有节点的单源最短路，复杂度 $O(n^2)$。

贪心：Dijkstra

- 上面提到的最短路算法都没有对图作出除无负环外的要求，但实际中见到的图往往连负权边都没有，如何利用这个性质？
- 当没有负权边时，容易发现当前 f 值最小的点再也没有可能被松弛了（因为其他的 f 加上一个非负值后不会变小），所以贪心地用这个点松弛其它点后就丢弃该点，不再考虑。
- 重复这个步骤 n 次，就可以得到所有节点的单源最短路，复杂度 $O(n^2)$。
- 在稀疏图上，寻找最小 f 值可以使用优先队列进行优化，优化后复杂度 $O(m\log n)$。

模板：堆优化 Dij

```
void dijkstra(int n, int s) {
  // 除了起点初始要设为最大值
  memset(dis, 0x3f, sizeof(dis));
  dis[s] = 0; // 起点设置为 0
  q.push((node) {.dist = 0, .u = s});
  while (!q.empty()) {
    node now = q.top(); q.pop();
    if (dis[now.u] < now.dist) continue;
    for (auto ed : e[now.u]) {
      if (dis[ed.v] > dis[now.u] + ed.w) {
        dis[ed.v] = dis[now.u] + ed.w;
        q.push({dis[ed.v], ed.v});
      }
    }
  }
}
```

目录

基础概念

- 求出所有点对 (i, j) 间的最短路径 $d_{i,j}$。(这个形式有点像邻接矩阵)。

基础概念

- 求出所有点对 (i, j) 间的最短路径 $d_{i,j}$。(这个形式有点像邻接矩阵)。
- 感觉很简单，直接跑 n 遍单源最短路不就好了?

基础概念

- 求出所有点对 (i,j) 间的最短路径 $d_{i,j}$。(这个形式有点像邻接矩阵)。
- 感觉很简单，直接跑 n 遍单源最短路不就好了?
- 正权图复杂度 $O(nm\log n)$，有负权边 $O(n^2m)$。

基础概念

- 求出所有点对 (i, j) 间的最短路径 $d_{i,j}$。(这个形式有点像邻接矩阵)。
- 感觉很简单，直接跑 n 遍单源最短路不就好了?
- 正权图复杂度 $O(nm \log n)$，有负权边 $O(n^2 m)$。
- 对于稠密图（$m = \Omega(n^2)$）则分别是 $O(n^3 \log n)$ 和 $O(n^4)$ 的。

基础概念

- 求出所有点对 (i,j) 间的最短路径 $d_{i,j}$。(这个形式有点像邻接矩阵)。
- 感觉很简单，直接跑 n 遍单源最短路不就好了?
- 正权图复杂度 $O(nm\log n)$，有负权边 $O(n^2m)$。
- 对于稠密图（$m=\Omega(n^2)$）则分别是 $O(n^3\log n)$ 和 $O(n^4)$ 的。
- 对于有负边的图太慢了!

基础概念

- 求出所有点对 (i, j) 间的最短路径 $d_{i,j}$。(这个形式有点像邻接矩阵)。
- 感觉很简单，直接跑 n 遍单源最短路不就好了?
- 正权图复杂度 $O(nm\log n)$，有负权边 $O(n^2m)$。
- 对于稠密图（$m = \Omega(n^2)$）则分别是 $O(n^3\log n)$ 和 $O(n^4)$ 的。
- 对于有负边的图太慢了!
- 后跑的单源最短路能不能利用之前的最短路的信息?

基础概念

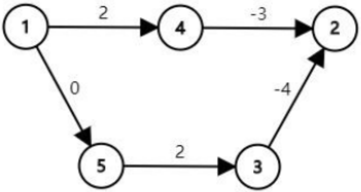
- 求出所有点对 (i, j) 间的最短路径 $d_{i,j}$。(这个形式有点像邻接矩阵)。
- 感觉很简单，直接跑 n 遍单源最短路不就好了?
- 正权图复杂度 $O(nm\log n)$，有负权边 $O(n^2m)$。
- 对于稠密图（$m = \Omega(n^2)$）则分别是 $O(n^3\log n)$ 和 $O(n^4)$ 的。
- 对于有负边的图太慢了!
- 后跑的单源最短路能不能利用之前的最短路的信息?
- 答案是可以，下面介绍 Johnson 算法。

Johnson 算法

- 你会怎么处理负权边?

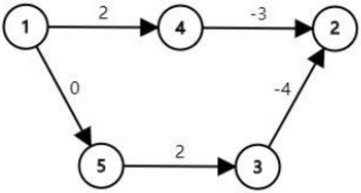
Johnson 算法

- 你会怎么处理负权边?
- ~~同时加上一个值?~~ (对长路径不公平)



Johnson 算法

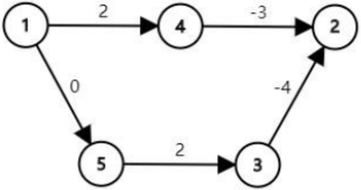
- 你会怎么处理负权边?
- ~~同时加上一个值?~~ (对长路径不公平)



- 如果有了一次最短路的结果，我们可以对图重新赋上边权。

Johnson 算法

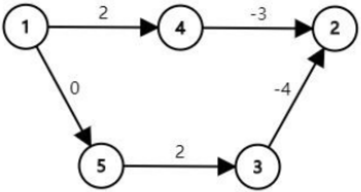
- 你会怎么处理负权边?
- ~~同时加上一个值?~~ (对长路径不公平)



- 如果有了一次最短路的结果，我们可以对图重新赋上边权。
- 考虑 (u, v, w)，令 $w' = h(u) - h(v) + w$。

Johnson 算法

- 你会怎么处理负权边?
- ~~同时加上一个值?~~ (对长路径不公平)



- 如果有了一次最短路的结果，我们可以对图重新赋上边权。
- 考虑 (u, v, w)，令 $w' = h(u) - h(v) + w$。
- 这个取法对于两定点间的任意路径都是公平的，考虑 $(u, t_1, w_1) \cdots (t_n, v, w_n)$，明显有

$$W' = \sum_{i \in [n]} w'_i = \sum_{i \in [n]} w_i + h(u) - h(v) = W + h(u) - h(v)$$

Johnson 算法

- 我们把 h 称作势能函数，它对每一个点有一个固定的取值。

Johnson 算法

- 我们把 h 称作势能函数，它对每一个点有一个固定的取值。
- 如何选取 h 使得 $w' = h(u) - h(v) + w \geq 0$?

Johnson 算法

- 我们把 h 称作势能函数，它对每一个点有一个固定的取值。
- 如何选取 h 使得 $w' = h(u) - h(v) + w \geq 0$?
- 明显地，我们希望 $h(u) + w \geq h(v)$......

Johnson 算法

- 我们把 h 称作势能函数，它对每一个点有一个固定的取值。
- 如何选取 h 使得 $w' = h(u) - h(v) + w \geq 0$?
- 明显地，我们希望 $h(u) + w \geq h(v)$......
- 好像有点眼熟？就是三角形不等式！

Johnson 算法

- 我们把 h 称作势能函数，它对每一个点有一个固定的取值。
- 如何选取 h 使得 $w' = h(u) - h(v) + w \geq 0$?
- 明显地，我们希望 $h(u) + w \geq h(v)$......
- 好像有点眼熟？就是三角形不等式！
- 选取 h 为第一次最短路的 d 数组即可。

Johnson 算法

- 我们把 h 称作势能函数，它对每一个点有一个固定的取值。
- 如何选取 h 使得 $w' = h(u) - h(v) + w \geq 0$?
- 明显地，我们希望 $h(u) + w \geq h(v)$……
- 好像有点眼熟？就是三角形不等式！
- 选取 h 为第一次最短路的 d 数组即可。
- 稀疏图复杂度 $O(nm \log n)$，稠密图复杂度 $O(n^3 \log n)$。

Johnson 算法

- 我们把 h 称作势能函数，它对每一个点有一个固定的取值。
- 如何选取 h 使得 $w' = h(u) - h(v) + w \geq 0$?
- 明显地，我们希望 $h(u) + w \geq h(v)$......
- 好像有点眼熟？就是三角形不等式！
- 选取 h 为第一次最短路的 d 数组即可。
- 稀疏图复杂度 $O(nm \log n)$，稠密图复杂度 $O(n^3 \log n)$。
- 还有提升空间吗？

又是 DP：Floyd

- 吐槽：DP 这种基础思想怎么最后才上。建议明年安排前头。

又是 DP：Floyd

- 吐槽：DP 这种基础思想怎么最后才上。建议明年安排前头。
- 状态设计：
  $dp_{k,i,j}$ 表示任意两点间可以经过 $(1,\cdots,k)$ 内点的最短路径。

又是 DP：Floyd

- 吐槽：DP 这种基础思想怎么最后才上。建议明年安排前头。
- 状态设计：
  $dp_{k,i,j}$ 表示任意两点间可以经过 $(1, \cdots, k)$ 内点的最短路径。
- 最短路显然是有最优子结构的（全局最优，子问题肯定也选最优），然后我们把划分选择讨论为 i, j 最短路经过 k 和不经过 k，有，

$$dp_{k,i,j} = \max\{dp_{k,i,j}, dp_{k-1,i,k} + dp_{k-1,j,k}\}$$

又是 DP：Floyd

- 吐槽：DP 这种基础思想怎么最后才上。建议明年安排前头。
- 状态设计：
  $dp_{k,i,j}$ 表示任意两点间可以经过 $(1, \cdots, k)$ 内点的最短路径。
- 最短路显然是有最优子结构的（全局最优，子问题肯定也选最优），然后我们把划分选择讨论为 i, j 最短路经过 k 和不经过 k，有，

$$dp_{k,i,j} = \max\{dp_{k,i,j}, dp_{k-1,i,k} + dp_{k-1,j,k}\}$$

- 简洁和舒适：

```
memcpy(dp[0], g, sizeof(g)); // 初始状态
for (int k = 1; k <= n; k++)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      dp[k][i][j] = max(dp[k-1][i][k], dp[k-1][k][j]);
```

又是 DP：Floyd

- 吐槽：DP 这种基础思想怎么最后才上。建议明年安排前头。
- 状态设计：
  $dp_{k,i,j}$ 表示任意两点间可以经过 $(1, \cdots, k)$ 内点的最短路径。
- 最短路显然是有最优子结构的（全局最优，子问题肯定也选最优），然后我们把划分选择讨论为 i, j 最短路经过 k 和不经过 k，有，

$$dp_{k,i,j} = \max\{dp_{k,i,j}, dp_{k-1,i,k} + dp_{k-1,j,k}\}$$

- 简洁和舒适：

```
memcpy(dp[0], g, sizeof(g)); // 初始状态
for (int k = 1; k <= n; k++)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      dp[k][i][j] = max(dp[k-1][i][k], dp[k-1][k][j]);
```

- 复杂度：就差把 $O(n^3)$ 写脸上了。

以邻接矩阵为基础的图算法很常见（如前述的基尔霍夫矩阵树）。典型是用类矩阵乘法运算来进行图上走 k 步信息的维护。下面看几道题。

例 (WalkSchema)

给定一个 n 阶有向图，求任意两点间长度为 k 的路径的条数（有模数）。$n \le 100, k \le 10^9$

解 (WalkSchema)

简单地，有：

$$dp_{r,i,j} = \sum_{l \in [n]} dp_{r-1,i,k} \cdot g_{k,j} \qquad dp_{1,i,j} = g_{i,j}$$

这其实就是矩阵乘法，$dp_r = dp_{r-1} \cdot g$。利用倍增（快速幂）计算即可。

来点计算几何?

例 (Polygon)

平面上有一个 n 边凸多边形，选择其中 k 个点，组成一个新的凸多边形，最小化这个凸多边形的周长。$n \le 100, k \le 1000$。

解 (Polygon)

- 按顺时针顺序给点编号，然后按照点编号给边定向（从小到大)。这样图就由完全图转化为一个竞赛图。

来点计算几何?

例 (Polygon)

平面上有一个 n 边凸多边形，选择其中 k 个点，组成一个新的凸多边形，最小化这个凸多边形的周长。$n \le 100, k \le 1000$。

解 (Polygon)

- 按顺时针顺序给点编号，然后按照点编号给边定向（从小到大）。这样图就由完全图转化为一个竞赛图。
- 发现原图任意一个 k 阶子环只有一条边是反向的，考虑破环为链。

来点计算几何?

例 (Polygon)

平面上有一个 n 边凸多边形，选择其中 k 个点，组成一个新的凸多边形，最小化这个凸多边形的周长。$n \le 100, k \le 1000$。

解 (Polygon)

- 按顺时针顺序给点编号，然后按照点编号给边定向（从小到大）。这样图就由完全图转化为一个竞赛图。
- 发现原图任意一个 k 阶子环只有一条边是反向的，考虑破环为链。
- 显然只需维护一个限制必须走的 k 条边区间最长路。

来点计算几何?

例 (Polygon)

平面上有一个 n 边凸多边形，选择其中 k 个点，组成一个新的凸多边形，最小化这个凸多边形的周长。$n \le 100, k \le 1000$。

解 (Polygon)

- 按顺时针顺序给点编号，然后按照点编号给边定向（从小到大）。这样图就由完全图转化为一个竞赛图。
- 发现原图任意一个 k 阶子环只有一条边是反向的，考虑破环为链。
- 显然只需维护一个限制必须走的 k 条边区间最长路。
- 有方程

$$dp_{r,i,j} = \max_k (dp_{r-1,i,k} + g_{k,j}) \qquad dp_{1,i,j} = g_{i,j}$$

来点计算几何?

例 (Polygon)

平面上有一个 n 边凸多边形，选择其中 k 个点，组成一个新的凸多边形，最小化这个凸多边形的周长。$n \le 100, k \le 1000$。

解 (Polygon)

- 易知由 dp_r 到 dp_{r+1} 的过程满足结合律，就是说当 $c + d = r$，

$$dp_{r,i,j} = \max_k (dp_{c,i,k} + dp_{d,k,j})$$

来点计算几何?

例 (Polygon)

平面上有一个 n 边凸多边形，选择其中 k 个点，组成一个新的凸多边形，最小化这个凸多边形的周长。$n \leq 100, k \leq 1000$。

解 (Polygon)

- 易知由 dp_r 到 dp_{r+1} 的过程满足结合律，就是说当 $c + d = r$，

$$dp_{r,i,j} = \max_k (dp_{c,i,k} + dp_{d,k,j})$$

- 到这里就很简单了，倍增即可，复杂度 $O(n^3 \log k)$。

来点计算几何?

例 (Polygon)

平面上有一个 n 边凸多边形，选择其中 k 个点，组成一个新的凸多边形，最小化这个凸多边形的周长。$n \le 100, k \le 1000$。

解 (Polygon)

- 易知由 dp_r 到 dp_{r+1} 的过程满足结合律，就是说当 $c + d = r$，

$$dp_{r,i,j} = \max_k (dp_{c,i,k} + dp_{d,k,j})$$

- 到这里就很简单了，倍增即可，复杂度 $O(n^3 \log k)$。
- 那如果我说 $n \le 1000$，阁下又该如何应对?

来点计算几何？（选讲）

例 (Polygon)

平面上有一个 n 边凸多边形，选择其中 k 个点，组成一个新的凸多边形，最小化这个凸多边形的周长。$n \leq 100, k \leq 1000$。

解 (Polygon)

- 这个类似矩阵乘法的 DP 非常讨人烦，考虑优化它。

来点计算几何？（选讲）

例 (Polygon)

平面上有一个 n 边凸多边形，选择其中 k 个点，组成一个新的凸多边形，最小化这个凸多边形的周长。$n \le 100, k \le 1000$。

解 (Polygon)

- 这个类似矩阵乘法的 DP 非常讨人烦，考虑优化它。
- 容易发现有四边形不等式成立，

$$\forall a < b < c < d,\ r,\ g_{a,c} + g_{b,d} \ge g_{a,d} + g_{b,c}$$

来点计算几何？（选讲）

例 (Polygon)

平面上有一个 n 边凸多边形，选择其中 k 个点，组成一个新的凸多边形，最小化这个凸多边形的周长。$n \leq 100, k \leq 1000$。

解 (Polygon)

- 这个类似矩阵乘法的 DP 非常讨人烦，考虑优化它。
- 容易发现有四边形不等式成立，

$$\forall a < b < c < d,\ \ r,\ \ g_{a,c} + g_{b,d} \geq g_{a,d} + g_{b,c}$$

- 所以令 $p_{i,j} = \arg\max_k(dp_{c,i,k} + dp_{d,k,j})$，有 $p_{i,j-1} \leq p_{i,j} \leq p_{i+1,j}$。

来点计算几何？（选讲）

例 (Polygon)

平面上有一个 n 边凸多边形，选择其中 k 个点，组成一个新的凸多边形，最小化这个凸多边形的周长。$n \leq 100, k \leq 1000$。

解 (Polygon)

- 这个类似矩阵乘法的 DP 非常讨人烦，考虑优化它。
- 容易发现有四边形不等式成立，

$$\forall a < b < c < d,\ r,\ g_{a,c} + g_{b,d} \geq g_{a,d} + g_{b,c}$$

- 所以令 $p_{i,j} = \arg\max_k (dp_{c,i,k} + dp_{d,k,j})$，有 $p_{i,j-1} \leq p_{i,j} \leq p_{i+1,j}$。
- 然后决策单调性优化就 $O(n^2 \log k)$ 了。

例 (Polygon)

平面上有一个 n 边凸多边形，选择其中 k 个点，组成一个新的凸多边形，最小化这个凸多边形的周长。$n \leq 100, k \leq 1000$。

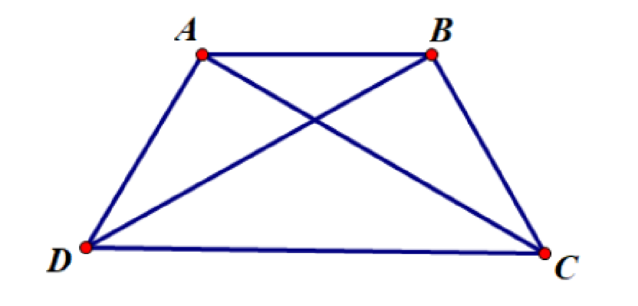
来点计算几何？（选讲）

例 (Polygon)

平面上有一个 n 边凸多边形，选择其中 k 个点，组成一个新的凸多边形，最小化这个凸多边形的周长。$n \le 100, k \le 1000$。

解 (Polygon)

- 这倒是真的四边形。



目录

单源次短路（选讲）

- 我们讨论「非严格次短路」，即方案和最短路不一样的最短路径。「严格次短路」是距离和最短路不一样的最短路径。并约定不存在负权边。容易发现，次短路只能由次短路和最短路拼成。
- 自然的想法是借鉴 Dijkstra 的思路。

单源次短路（选讲）

- 我们讨论「非严格次短路」，即方案和最短路不一样的最短路径。「严格次短路」是距离和最短路不一样的最短路径。并约定不存在负权边。容易发现，次短路只能由次短路和最短路拼成。
- 自然的想法是借鉴 Dijkstra 的思路。
- 记每个点的「状态」为其最短路和次短路。

单源次短路（选讲）

- 我们讨论「非严格次短路」，即方案和最短路不一样的最短路径。「严格次短路」是距离和最短路不一样的最短路径。并约定不存在负权边。容易发现，次短路只能由次短路和最短路拼成。
- 自然的想法是借鉴 Dijkstra 的思路。
- 记每个点的「状态」为其最短路和次短路。
- 为了区分方案，除了记录路径长度外还要记录路径的来源点。

单源次短路（选讲）

- 我们讨论「非严格次短路」，即方案和最短路不一样的最短路径。「严格次短路」是距离和最短路不一样的最短路径。并约定不存在负权边。容易发现，次短路只能由次短路和最短路拼成。
- 自然的想法是借鉴 Dijkstra 的思路。
- 记每个点的「状态」为其最短路和次短路。
- 为了区分方案，除了记录路径长度外还要记录路径的来源点。
- 由于 Dij 的贪心性质，每个点的最短/次短路最多各被松弛一次。

单源次短路（选讲）

- 我们讨论「非严格次短路」，即方案和最短路不一样的最短路径。「严格次短路」是距离和最短路不一样的最短路径。并约定不存在负权边。容易发现，次短路只能由次短路和最短路拼成。
- 自然的想法是借鉴 Dijkstra 的思路。
- 记每个点的「状态」为其最短路和次短路。
- 为了区分方案，除了记录路径长度外还要记录路径的来源点。
- 由于 Dij 的贪心性质，每个点的最短/次短路最多各被松弛一次。
- 复杂度 $O(m \log n)$。

单点对次短路（选讲）

- 假设 u, v 间最短路边的集合是 SP。

单点对次短路（选讲）

- 假设 u, v 间最短路边的集合是 SP。
- 对于次短路的任意一条边 (u, v, w)，有 $d_{S,u} + w + d_{v,T} = d'_{u,v}$

单点对次短路（选讲）

- 假设 u, v 间最短路边的集合是 SP。
- 对于次短路的任意一条边 (u, v, w)，有 $d_{S,u} + w + d_{v,T} = d'_{u,v}$
- 那么实际上，$d'_{u,v} = \min_{(u,v,w) \notin SP}(d_{S,u} + w + d_{v,T})$。

单点对次短路（选讲）

- 假设 u, v 间最短路边的集合是 SP。
- 对于次短路的任意一条边 (u, v, w)，有 $d_{S,u} + w + d_{v,T} = d'_{u,v}$
- 那么实际上，$d'_{u,v} = \min_{(u,v,w)\notin SP}(d_{S,u} + w + d_{v,T})$。
- 那就是要求一次 S 的单源最短路，一次 T 的“单目标最短路”。

单点对次短路（选讲）

- 假设 u, v 间最短路边的集合是 SP。
- 对于次短路的任意一条边 (u, v, w)，有 $d_{S,u} + w + d_{v,T} = d'_{u,v}$
- 那么实际上，$d'_{u,v} = \min_{(u,v,w)\notin SP}(d_{S,u} + w + d_{v,T})$。
- 那就是要求一次 S 的单源最短路，一次 T 的“单目标最短路”。
- 复杂度 $O(m \log n)$。

A*、最优路径搜索和单点对 k 短路（选讲）

目录

一点 BB

- 还是那两个字，你需要坚持**自学**。推荐使用 OI Wiki。
- 课下的习题起到的训练作用十分有限**（即使你订正了他们！）**。
- 为了掌握本次课的专题，你**至少应该**掌握所有模板和例题的实现。
- 更进一步地，你应该**使用搜索引擎**自行寻找同类的题目和相关的算法，磨练自己的代码能力，建立自己的知识体系。

关于练习题

- 还是四道题，从七八道题里选出来的，没出的都当例题了。
- 有一道选做题，还没想好放不放出来。
- 重点：单源最短路/次短路，最小生成树，树的 dfs 序。
- 部分题目没有唯一解，可以想想别的做法。
- 因为第一场比赛大家表现差强人意，所以临时更换了题目并降低了难度。~~又是通宵修题 真的都是签到题了啊这次~~
- **请至少学会一种最小生成树算法的实现方法。**
- 部分分还是给的很多的，不要轻易放弃。
- 数据出的比较放水，要是大家卡过了还请轻 D。
- 谢谢大家！祝大家午餐愉快，下午做题顺利。