PS7

221300066 季千□

2024 年 6 月 12 日

1 Problem1

(a)

Vertex A: distance = 0, parent = None

Vertex B: distance = 1, parent = A

Vertex C: distance = 1, parent = B

Vertex D: distance = 3, parent = C

Vertex E: distance = 2, parent = B

Vertex F: distance = 3, parent = E

Vertex G: distance = 1, parent = A

Vertex H: distance = 1, parent = A

(b)

Vertex Q: start_time = 1, end_time = 20

Edge Q -> W: type = Tree

Edge Q -> S: type = Tree

Edge Q -> T: type = Tree

Edge Q -> Y: type = Tree

Vertex R: start_time = 2, end_time = 19

Edge R -> Y: type = Forward

Edge R -> U: type = Forward

Vertex S: start_time = 3, end_time = 18

Edge S -> Q: type = Forward

Edge S -> V: type = Tree

Edge S -> W: type = Tree

Vertex T: start_time = 4, end_time = 17
Edge T -> Q: type = Forward
Edge T -> X: type = Forward
Edge T -> Y: type = Forward
Vertex U: start_time = 5, end_time = 16
Edge U -> R: type = Forward
Edge U -> Y: type = Forward
Vertex V: start_time = 6, end_time = 15
Edge V -> W: type = Forward
Edge V -> S: type = Tree
Vertex W: start_time = 7, end_time = 14
Edge W -> Q: type = Forward
Edge W -> S: type = Forward
Edge W -> V: type = Forward
Vertex X: start_time = 8, end_time = 13
Edge X -> T: type = Forward
Edge X -> Z: type = Forward
Vertex Y: start_time = 9, end_time = 12
Edge Y -> Q: type = Forward
Vertex Z: start_time = 10, end_time = 11
(c)Final ordering:A B E F C D
Bonus
我的算法的思路如下：
从任意一个节点 s 开始，用 DFS 找到距离它最远的节点 x。
从节点 x 开始，用 DFS 找到距离它最远的节点 y。
节点 x 和 y 之间的距离就是树的直径。
伪代码如下：

```
function diameter (tree)
  s = any node in tree
  x = farthest node from s using DFS
  y = farthest node from x using DFS
  return dist (x, y)
```

时间复杂度是 O(|V| + |E|)，因为只需要两次 DFS，每次 DFS 的时间复杂

度是 O(|V| + |E|)。
为证明算法正确性，需要证明以下两个命题：
命题 1：树的直径一定是两个叶子节点之间的路径。
命题 2：如果 x 是从 s 开始的 DFS 的最远节点，那么 x 一定是一个叶子节点，并且是 DFS 的一个端点。
命题 1 的证明：
假设树的直径是两个非叶子节点 u 和 v 之间的路径，那么 u 和 v 都至少有一个子节点。
不妨设 u 的子节点是 w，那么 dist (u, w) = 1，所以 dist (w, v) > dist (u, v)。
这与 u 和 v 之间的路径是最长的矛盾，所以假设不成立。因此，树的直径一定是两个叶子节点之间的路径。
命题 2 的证明：
假设 x 不是一个叶子节点，那么它至少有一个子节点 y。
那么 dist (s, y) = dist (s, x) + 1，所以 y 比 x 更远，这与 x 是最远节点矛盾，所以假设不成立。因此，x 一定是一个叶子节点。
又因为 x 是从 s 开始的 DFS 的最远节点，所以它一定是 DFS 的一个端点，否则还有更远的节点可以访问到。因此，x 一定是一个叶子节点，并且是 DFS 的一个端点。
综上所述，我的算法是正确的。

2 Problem2

(a)Disprove

用反证法来证明这个命题的否定：不存在这样的有向图和顶点 u。
假设存在这样的有向图 G = (V, E) 和顶点 $u \in V$，那么 u 既有入度又有出度，意味着存在至少两条边 (v, u) 和 (u, w)，其中 $v, w \in V$。不妨设 u 是 DFS 的起始顶点，那么 DFS 会先访问 u，然后访问 u 的一个相邻顶点 w，然后继续沿着 w 的相邻顶点进行 DFS，直到遇到一个没有相邻顶点或者所有相邻顶点都已经访问过的顶点 x。此时，DFS 会回溯到 x 的父节点，也就是 x 的前驱节点，然后继续访问其它相邻顶点，直到回溯到 u。由于 u 还有一个相邻顶点 v，所以 DFS 会访问 v，然后继续沿着 v 的相邻顶点进行 DFS，直到遇到一个没有相邻顶点或者所有相邻顶点都已经访问过的顶

点 y。此时，DFS 会回溯到 y 的父节点，也就是 y 的前驱节点，然后继续访问其它相邻顶点，直到回溯到 u。由于 u 已经访问了所有的相邻顶点，所以 DFS 会结束对 u 的访问，然后继续访问图中其它没有访问过的顶点。

所以 DFS 会在访问 u 之后，访问 u 的两个相邻顶点 w 和 v，然后分别沿着 w 和 v 的路径进行 DFS，最后回溯到 u。这样，u 就不可能单独成为一棵树，而是和 w、v 以及它们的后继节点构成一棵树。这与题目的条件矛盾，假设不成立。因此，不存在这样的有向图和顶点 u。

(b)Disprove

用反证法来证明不存在这样的一个算法。

假设存在这样的算法 A，那么可以用它来解决一个已知的 NP 完全问题：哈密顿回路问题 1。哈密顿回路问题是指，给定一个无向图 G = (V, E)，判断 G 是否存在一个经过所有顶点恰好一次的回路 1。我们可以用以下步骤来利用算法 A 来解决哈密顿回路问题:

对于图 G 的每个顶点 v，将 v 复制成 v'，并添加一条边 (v, v')。

对于图 G 的每条边 (u, v)，将其删除，并添加两条边 (u, v') 和 (v, u')。

运行算法 A 判断修改后的图 G' 是否包含环，如果包含，则返回是，否则返回否。

不难证明，这个过程是正确的，图 G 存在哈密顿回路当且仅当图 G' 存在环。首先，假设图 G 存在哈密顿回路，那么这个回路可以表示为 $v1 - v2 - ... - vn - v1$，其中 vi 是图 G 的顶点，且不重复。那么，我们可以构造一个环 $v1 - v1' - v2 - v2' - ... - vn - vn' - v1$，这个环是图 G' 的子图，所以图 G' 存在环。其次，假设图 G' 存在环，那么这个环一定是由原来图 G 的顶点和它们的复制顶点交替组成的，否则就会有两个相邻的原顶点或复制顶点，这与图 G' 的构造方式矛盾。那么，我们可以将这个环中的复制顶点去掉，得到一个回路 $v1 - v2 - ... - vn - v1$，这个回路是图 G 的子图，且不重复，所以图 G 存在哈密顿回路。

根据以上，只需要 O(|V|) 的时间来修改图 G，然后用算法 A 来判断图 G' 是否包含环，这个算法 A 的时间复杂度也是 O(|V|)。所以总共只需要 O(|V|) 的时间来解决哈密顿回路问题。但是，哈密顿回路问题是一个 NP 完全问题，也就是说没有已知的多项式时间的算法能来解决它。这与假设矛盾，所以假设不成立。因此，不存在这样的一个算法 A。

(c) 不正确

考虑这样的一个有向图，这个图有三个强连通分量，分别是 $\{a, b, c\}$,$\{d, e\}$

和 $\{f\}$。按照 CLRS 的算法，首先对 G 进行 DFS，得到的完成时间 (按 a-f 的顺序) 为 12-7 的降序排列
然后，对 G 的转置图 G^{T} 进行 DFS，按照完成时间的降序扫描顶点，得到的强连通分量仍然是 $\{a,b,c\}$,$\{d,e\}$ 和 $\{f\}$
但是如果按照修改后的算法，首先对 G 进行 DFS，得到的完成时间和上面一样，然后对 G 进行 DFS，按照完成时间的升序扫描顶点，得到的强连通分量是 $\{e,d,c,b,a\}$ 和 $\{f\}$ 这是错误的结果，因为它把 $\{a,b,c\}$ 和 $\{d,e\}$ 合并成了一个强连通分量，而这两个分量之间是没有路径的。这个反例说明了修改后的算法是不正确的。

3 Problem Group I

3.1 ProblemI.1

(a) 算法描述如下：
首先，对图 G 进行 BFS，从任意一个顶点开始，记录每个顶点到起始顶点的距离，以及每个顶点的父节点。
然后，对于每个顶点 v，计算两个硬币到达 v 的总步数，即 dist(s, v) + dist(t, v)，其中 s 和 t 是两个硬币所在的顶点，dist(u, w) 表示 u 和 w 之间的最短路径长度。
最后，找出所有顶点中总步数的最小值，如果存在，则返回该值，否则返回无法达到的信息。
这个算法的时间复杂度是 O(|V| + |E|)，因为 BFS 的时间复杂度是 O(|V| + |E|)，计算总步数和最小值的时间复杂度是 O(|V|)。
(b) 算法描述如下：
首先，对图 G 进行三次 BFS，分别从三个硬币所在的顶点开始，记录每个顶点到每个硬币的距离，以及每个顶点的父节点。
然后，对于每个顶点 v，计算三个硬币到达 v 的总步数，即 dist(s, v) + dist(t, v) + dist(u, v)，其中 s, t, u 是三个硬币所在的顶点，dist(x, y) 表示 x 和 y 之间的最短路径长度。
最后，找出所有顶点中总步数的最小值，如果存在，则返回该值，否则返回无法达到的信息。
这个算法的时间复杂度是 O(|V| + |E|)，因为三次 BFS 的时间复杂度是 O(|V| + |E|)，计算总步数和最小值的时间复杂度是 O(|V|)。

(c) 算法描述如下：

首先，对图 G 进行一次 BFS，从任意一个顶点开始，记录每个顶点的层次，即到起始顶点的距离。

然后，对于每个层次 i，计算该层次的顶点数目 n_i，以及该层次的硬币数目 m_i。

最后，检查是否存在一个层次 i，使得 $m\_i \geq 42$，如果存在，则返回可能，否则返回不可能。

这个算法的时间复杂度是 O(|V| + |E|)，因为 BFS 的时间复杂度是 O(|V| + |E|)，计算层次和硬币数目的时间复杂度是 O(|V|)。

3.2 ProblemI.2

(a) 可以使用 DFS 或 BFS 来遍历图 G，并记录每个顶点的发现时间和完成时间。两种搜索算法都可以在 O(|V|+|E|) 时间内完成，其中 |V| 是顶点的数量，|E| 是边的数量。然后，可以检查每个顶点是否满足以下条件之一：

1. 它是根顶点，并且它有两个或更多的子节点。

2. 它不是根顶点，并且它有一个或更多的子节点，其发现时间大于或等于它的完成时间。

如果一个顶点满足以上两个条件之一，那么它就是一个关键顶点，因为它的删除会切断它的子树与其余图的连接。可以在 O(|V|) 时间内检查所有的顶点，所以总的时间复杂度仍然是 O(|V|+|E|)。如果找到一个不是关键顶点的顶点，就可以停止搜索并返回它。如果没有找到这样的顶点，那么说明图 G 是一个完全图，即任何一个顶点的删除都会断开图。

(b) 使用以下算法：

首先，用 DFS 或 BFS 遍历图 G，并记录每个顶点的发现时间和完成时间。

然后，对所有的顶点按照完成时间的降序进行排序，得到一个顶点的列表 L。

最后，从列表 L 的头部开始，依次删除每个顶点，并将它们加入到一个新的列表 R 中。

可以证明，这个算法可以得到一个满足要求的删除顶点的顺序。首先，如果一个顶点 v 在列表 L 中排在另一个顶点 u 的前面，那么 v 的完成时间一定大于 u 的完成时间。这意味着在遍历图 G 的过程中，v 一定是在 u 之后被访问的，也就是说，v 不可能是 u 的祖先。因此，删除 v 时，不会影响 u 与

其余图的连通性。其次，删除一个顶点 v 时，它可能会导致它的一些子节点与其余图断开连接。但是，由于是按照完成时间的降序删除顶点的，所以这些子节点一定已经被删除了，因为它们的完成时间一定小于 v 的完成时间。因此删除 v 时，不会影响图的连通性。综上所述，可以得到一个删除顶点的顺序，使得没有任何删除会断开图。

这个算法的时间复杂度是 O(|V|+|E|)，因为遍历图 G 需要 O(|V|+|E|) 时间，排序顶点需要 O(|V|log|V|) 时间，删除顶点需要 O(|V|) 时间，而 O(|V|log|V|) 是 O(|V|+|E|) 的上界。

3.3 ProblemI.3

(a) 根据题目给出的构造方法，可以得到如下的有向图：

$$\begin{tikzcd}
x1 \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & x2 \arrow[dl] \\
x3 \arrow[r, leftrightarrow] & x4
\end{tikzcd}$$

这个图有 8 个节点，分别是 x1, x2, x3, x4 及它们的否定。它有 10 条边，分别是：

从 x1 到 x2 和 x3，对应子句 $(x1 \vee x2)$ 和 $(x1 \vee x3)$

从 x2 到 x1 和 x3，对应子句 $(x1 \vee x2)$ 和 $(x1 \vee x3)$

从 x3 到 x4，对应子句 $(x3 \vee x4)$

从 x4 到 x1，对应子句 $(x1 \vee x4)$

从 x1 到 x4，对应子句 $(x1 \vee x4)$

从 x4 到 x3，对应子句 $(x3 \vee x4)$

从 x2 到 x1，对应子句 $(x1 \vee x2)$

从 x3 到 x1，对应子句 $(x1 \vee x3)$

从 x1 到 x2，对应子句 $(x1 \vee x2)$

从 x3 到 x2，对应子句 $(x1 \vee x3)$

(b) 用反证法来证明这个命题。

假设图 G 中存在一个强连通分量 C，包含了 x 和 x 的某个变量 x，且 I 有一个可满足赋值。那么，根据强连通分量的定义，C 中的任意两个节点之间都有一条有向路径。特别地，从 x 到 x 和从 x 到 x 都有一条有向路径。那么，沿着这两条路径，我们可以找到一系列的子句，它们的形式如下：

从 x 到 x 的路径上的子句：$(x \vee \alpha 1), (\alpha 1 \vee \alpha 2), ..., (\alpha k \vee x)$
从 x 到 x 的路径上的子句：$(x \vee \beta 1), (\beta 1 \vee \beta 2), ..., (\beta I \vee x)$
其中 αi 和 βj 是一些文字。由于 I 有一个可满足赋值，那么这些子句中至少有一个文字为真。但是，如果 x 为真，那么 x 为假，所以从 x 到 x 的路径上的所有子句都不满足；如果 x 为假，那么 x 为真，所以从 x 到 x 的路径上的所有子句都不满足。这就产生了矛盾。因此，不存在这样的强连通分量 C，或者 I 没有可满足赋值。
(c) 证明思路如下。
首先，假设图 G 中没有一个强连通分量包含了 x 和 x 的某个变量 y，那么可以用 Kosaraju 算法或 Tarjan 算法来找到图 G 的所有强连通分量，并将它们缩点为一个新的 DAG。
然后，对这个 DAG 进行拓扑排序，得到一个顺序。接着按照这个顺序，依次给每个强连通分量中的节点赋值。具体地，对于每个强连通分量，我们检查它是否已经有一个赋值，如果没有，就给它赋值为真，同时给它的否定赋值为假；如果已经有一个赋值，就保持不变。
最后，得到了一个对所有变量的赋值，可以证明这个赋值是可满足的：对于任意一个子句 $(x \vee y)$，它对应了图 G 中的两条边 $(x \vee y)$ 和 $(y \vee x)$。由于图 G 是一个 DAG，那么这两条边一定不会在同一个强连通分量中，否则就会形成一个环。那么，根据赋值方法，x 和 y 一定不会同时为假，因为如果它们同时为假，那么它们的否定就同时为真，这就意味着它们在同一个强连通分量中，与假设矛盾。所以，x 和 y 中至少有一个为真，这就意味着子句 $(x \vee y)$ 为真。因此，这个赋值是可满足的。
(d) 可以用以下算法来在 O(|V| + |E|) 时间内求解 2-SAT 问题：
首先根据题目给出的构造方法，把 2-SAT 问题转化为一个有向图的问题。这个过程需要 O(|V| + |E|) 时间，其中 |V| 是顶点的数量，|E| 是边的数量。
然后，使用 Kosaraju 算法或 Tarjan 算法来找到有向图的所有强连通分量。这两种算法都可以在 O(|V| + |E|) 时间内完成。
最后，检查有向图中是否存在一个强连通分量，包含了 x 和-x。如果存在，那么就输出无解；如果不存在，就按照上面的方法进行拓扑排序和赋值，并输出一个可满足的赋值。这个过程也需要 O(|V| + |E|) 时间。
因此，这个算法的总时间复杂度是 O(|V| + |E|)。

4 Problem Group II

4.1 ProblemII.1

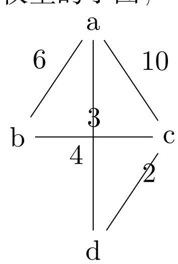
(a) 这个命题是正确的，可以用反证法来证明。
假设存在一个边 e，它属于某个最小生成树 T，但它不是任何一个切割的轻边。那么，对于任何一个包含 e 的切割 (S, V-S)，都存在另一个边 f，它也跨越这个切割，但它的权重比 e 小。那么，可以用 f 替换 e，得到一个新的生成树 T’，它的权重比 T 小，这与 T 是最小生成树矛盾。因此，不存在这样的边 e，所以命题成立。
(b) 这个命题是正确的，可以用数学归纳法来证明。
首先，当图 G 只有一个顶点时，它显然只有一个最小生成树，就是它自己。假设当图 G 有 n-1 个顶点时，命题成立，即 G 有一个唯一的最小生成树。当图 G 有 n 个顶点时，可以任意选择一个切割 (S, V-S)，并找到唯一的轻边 e，它跨越这个切割。由于 e 是轻边，它一定属于某个最小生成树 T。如果 T 不是唯一的，那么存在另一个最小生成树 T’，它不包含 e。那么可以在 T’ 中删除一个跨越切割的边 f，得到一个子图 G’，它有 n-1 个顶点。由于 f 不是轻边，它的权重比 e 大，所以 G’ 的权重比 T 减去 e 的权重小。根据归纳假设，G’ 有一个唯一的最小生成树 T’‘；那么 T’‘加上 e 就是 G 的一个最小生成树，它的权重比 T 小，这与 T 是最小生成树矛盾。因此，不存在这样的 T’，所以 T 是唯一的最小生成树。
(c) 这个命题是错误的。存在一个反例，如下图所示：

$$\begin{tikzcd}
A \arrow[r, dash, "e"] \arrow[d, dash, "f"'] & B \arrow[d, dash, "f"] \\
C \arrow[r, dash, "e"'] & D
\end{tikzcd}$$

这个图 G 有两条权重相等的边 e 和 f，但它只有一个最小生成树，就是包含所有边的图 G 本身。因为如果我们删除 e 或 f 中的任何一条，都会断开图 G，所以它们都是必须的。因此，这个图 G 有两条权重相等的边，但只有一个最小生成树。

4.2 ProblemII.2

(a) 这个问题不是最小生成树问题，因为最小生成树问题要求找到一个包含所有顶点的最小权重的子图，而这个问题只要求找到一个连通的最小权重的子图，不一定要包含所有顶点。例如，考虑下图：



这个图的最小生成树是包含所有边的图本身，它的权重是 10。但是这个图的最小权重连通子图是只包含 a, b, c, d 四个顶点和 a, b, b, c, c, d 三条边的子图，它的权重是 6。所以，这两个问题是不同的。

(b) 算法描述如下：

首先使用 Kruskal 算法或 Prim 算法来找到图 G 的最小生成树 T。这两种算法都可以在 O(|E|log|V|) 时间内完成，其中 |V| 是顶点的数量，|E| 是边的数量。

然后，对 T 的所有边按照权重的降序进行排序，得到一个边的列表 L。

最后，从列表 L 的头部开始，依次删除每条边，并检查删除后的子图是否仍然连通。如果是，就继续删除下一条边；如果不是，就停止删除，并返回当前的子图作为结果。

可以证明这个算法能得到一个最小权重连通子图：

首先，由于 T 是最小生成树，它的权重是所有连通子图中最小的。其次，当删除一条边时，如果子图仍然连通，那么就减少了子图的权重；如果子图不再连通，那么就找到了一个最小的连通分量，它不能再被划分。因此，删除边的过程结束时，就可以得到了一个最小权重连通子图。

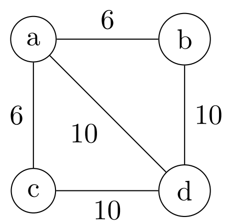
这个算法的时间复杂度是 O(|E|log|V|)，因为找到最小生成树需要 O(|E|log|V|) 时间，排序边需要 O(|E|log|E|) 时间，删除边需要 O(|E|) 时间，而 O(|E|log|E|) 是 O(|E|log|V|) 的上界。

4.3 ProblemII.3

(a) 用反证法来证明最小生成树是唯一的。

假设存在两个不同的最小生成树 T 和 T'，那么它们一定有至少一条不同的边，不妨设为 e 和 e'，且 w(e) < w(e')。那么可以用 e 替换 T' 中的 e'，得到一个新的生成树 T''，它的权重比 T' 小，这与 T' 是最小生成树矛盾。因此，不存在两个不同的最小生成树，所以最小生成树是唯一的。

然而，第二最小生成树不一定是唯一的。考虑下图：



这个图的最小生成树是包含所有边的图本身，它的权重是 10。但是这个图的第二最小生成树有两个，分别是只包含 $\{a, b, c, d\}$ 四个顶点和 $\{a, b\}, \{b, c\}, \{c, d\}$ 三条边的子图，它的权重是 6，以及只包含 $\{a, b, c, e\}$ 四个顶点和 $\{a, b\}, \{b, c\}, \{c, e\}$ 三条边的子图，它的权重也是 6。所以，这两个子图都是第二最小生成树，但它们是不同的。

(b) 用数学归纳法来证明。

首先，当图 G 只有两个顶点时，它显然只有一条边，所以不存在第二最小生成树，命题成立。

假设当图 G 有 n-1 个顶点时，命题成立，即存在一条边 (u, v) 属于 T，和一条边 (x, y) 不属于 T，使得 $(E(T) - \{(u, v)\}) \bigcup \{(x, y)\}$ 是 G 的一个第二最小生成树的边集。

当图 G 有 n 个顶点时，可以任意选择一个顶点 z，和一条连接 z 和 T 的边 (z, w)，并将它们从 G 和 T 中删除，得到一个子图 G' 和一个子树 T'。根据归纳假设，G' 中存在一条边 (u, v) 属于 T'，和一条边 (x, y) 不属于 T'，使得 $(E(T') - \{(u, v)\}) \bigcup \{(x, y)\}$ 是 G' 的一个第二最小生成树的边集。那么，可以在 G 中恢复 z 和 (z, w)，并得到 $(E(T) - \{(u, v)\}) \bigcup \{(x, y)\}$ 是 G 的一个第二最小生成树的边集。因为如果用 (x, y) 替换 (u, v)，那么就得到了 G' 的一个第二最小生成树，再加上 (z, w)，就得到了 G 的一个第二最小生成树。而如果用 (z, w) 替换 (u, v)，那就得到了 G' 的一个最小生成树，再加上 (z, w)，就得到了 G 的一个最小生成树。所以，这两种情况下，$(E(T) - \{(u, v)\}) \bigcup \{(x, y)\}$ 都是 G 的一个第二最小生成树的边集。

综上所述，命题成立。
(c) 我只能用动态规划的方法设计一个满足要求的算法：
使用一个二维数组 max[n][n] 来存储结果，其中 n = |V|，max[i][j] 表示 T 中顶点 i 和 j 之间的最大权重边。可以用以下递推公式来计算 max[i][j]：
如果 i = j，那么 max[i][j] = 0，因为同一个顶点之间没有边。
如果 i 和 j 是相邻的，那么 max[i][j] = w(i, j)，因为它们之间只有一条边。
如果 i 和 j 不是相邻的，那么 max[i][j] = maxmax[i][k], max[k][j]，其中 k 是 T 中 i 和 j 之间的任意一个顶点，因为它们之间的最大权重边要么是 i 和 k 之间的，要么是 k 和 j 之间的。
伪代码来实现这个算法如下：

```
n = |V|
max[n][n] // initialize with zeros
for each edge (i, j) in T
    max[i][j] = max[j][i] = w(i, j) // base case
for k = 1 to n // intermediate vertex
    for i = 1 to n // source vertex
        for j = 1 to n // destination vertex
            max[i][j] = max(max[i][j], max(max[i][k], max[k][j])) //update
```

这个算法的时间复杂度是 $O(|V|^2)$，因为它需要三层循环，每层循环需要 $O(|V|)$ 时间。
(d) 算法来计算 G 的一个第二最小生成树描述如下：
首先使用 Kruskal 算法或 Prim 算法来找到 G 的最小生成树 T。这两种算法都可以在 O(|E|log|V|) 时间内完成，其中 |V| 是顶点的数量，|E| 是边的数量。
然后，使用上面的算法来计算 T 中所有 $u, v \in V$ 的 max[u, v]，并将结果存储在一个二维数组 $max[n][n]$ 中，其中 $n = |V|$。这个算法需要 $O(|V|^2)$ 时间。
最后，遍历 G 中不属于 T 的所有边 (x, y)，并计算 $(E(T) - \{max[x, y]\}) \bigcup \{(x, y)\}$ 的权重，即用 (x, y) 替换 T 中 x 和 y 之间的最大权重边。记录下最小的权重，以及对应的边 (x, y) 和 max[x, y]。这个过程需要 O(|E|) 时间。返回 $(E(T) - \{max[x, y]\}) \bigcup \{(x, y)\}$ 作为 G 的一个第二最小生成树的边集。
这个算法的时间复杂度是 $O(|E|log|V| + |V|^2 + |E|) = O(|V|2)$，因为 $O(|E|log|V|)$ 是 $O(|V|2)$ 的上界。