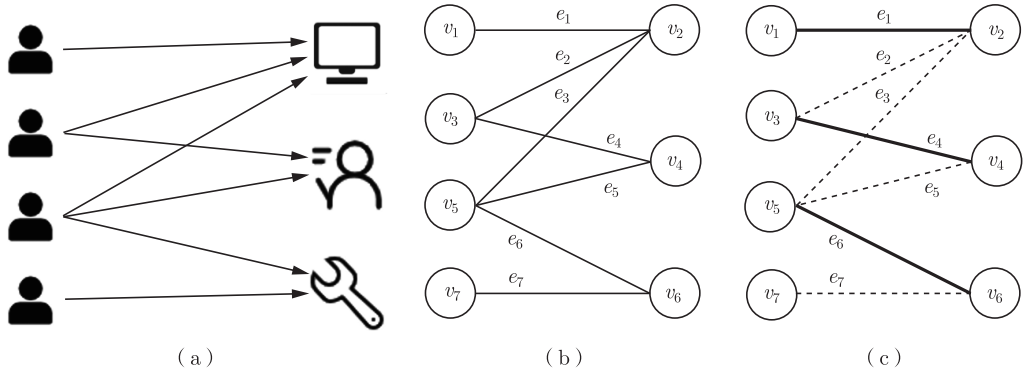
第 5 章　匹　　配

某企业有多项任务急需完成，每项任务最多由一人承担，每个人最多承担一项任务。企业内的每位职员可胜任其中若干项任务，如图 5.1（a）所示。应如何将这些任务分配给可胜任的职员，使尽可能多的任务有人承担？我们可以用图建模这个问题：将职员和任务表示为顶点，将可胜任关系表示为边，如图 5.1（b）所示，图中两两不相邻的边最多有多少条？对于这个例子，通过观察很容易发现：边子集 $\{e_1, e_4, e_6\}$ 表示一种最佳分配方式，3 项任务均由不同的职员承担，如图 5.1（c）所示。然而，一般意义上，如何解决这个问题呢？



（a）　　（b）　　（c）

图 5.1　任务分配问题

（a）4 位职员和他们可胜任的 3 项任务；（b）将可胜任关系表示为图；（c）最佳分配粗实线—最佳分配

某比赛要求双人组队参赛，一个人不允许加入多支队伍。某班级组织同学们组队参赛，由于这次比赛要求参赛选手具有较高的默契度，同学们商议决定：只有曾经搭档参加过类似比赛的同学，这次才可组队参赛。应如何组队才能派出最多数量的参赛队伍呢？我们可以用图建模这个问题：将同学表示为顶点，将过往搭档关系表示为边，如图 5.2（a）所示，图中两两不相邻的边最多有多少条？对于这个例子，通过观察很容易发现：边子集 $\{e_1, e_3, e_6, e_9, e_{11}, e_{13}\}$ 表示一种最佳组队方式，12 位同学全部组队参赛，如图 5.2（b）所示。然而，一般意义上，如何解决这个问题呢？

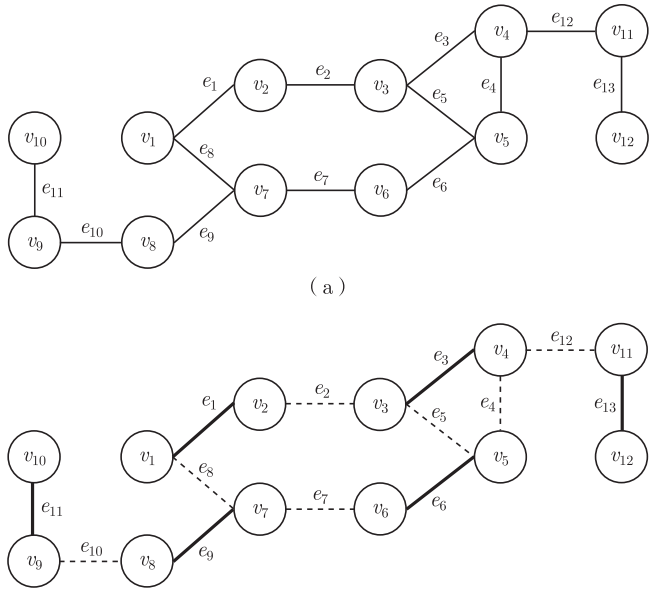


图 5.2　组队参赛问题

（a）表示过往搭档关系的图；（b）最佳组队
粗实线—最佳组队

上述两个问题都是顶点配对问题，区别在于任务分配问题被表示为二分图，而组队参赛问题未必被表示为二分图，例如，图 5.2（a）就不是二分图。如何设计算法解决这些问题？这便是本章讨论的主题：匹配。

本章共分 2 节：5.1节介绍匹配和最大匹配；5.2节介绍特殊的最大匹配——完美匹配。

若无特殊说明，则本章讨论的图都是简单图。

5.1　匹配和最大匹配

匹配是一组不相邻的边。我们通常感兴趣的是最大匹配，即包含边的数量最多的匹配。可以通过匈牙利算法或霍普克罗夫特-卡普算法找出二分图中的最大匹配，可以通过花算法找出非二分图中的最大匹配。现在，首先给出匹配的数学定义。

5.1.1　理论

对于图 $G=\langle V,E\rangle$ 和边子集 $M\subseteq E$，若 M 中的边两两不相邻，则 M 称作 G 的**匹配**（matching），M 中边的端点称作被 M **饱和**（saturated），又称**已匹配**（matched）。对于匹配 M，若 M 不是 G 的任何匹配的真子集，则 M 称作 G 的**极大匹配**（maximal

matching）。边的数量最多的匹配称作 G 的**最大匹配**（maximum matching）。例如，对于图 5.2（a）所示的图，边子集 $\{e_1, e_3\}$ 是匹配，它不是极大匹配，更不是最大匹配；边子集 $\{e_5, e_8, e_{10}, e_{12}\}$ 是极大匹配，它不是最大匹配；边子集 $\{e_1, e_3, e_6, e_9, e_{11}, e_{13}\}$ 是最大匹配，如图 5.2（b）所示。

✍ **思考题 5.1**　每个图都有匹配吗？

✍ **思考题 5.2**　图 1.6（a）所示的彼得森图的最大匹配有多少条边？

✍ **思考题 5.3**　阶为 n 的图的最大匹配至多有多少条边？

✍ **思考题 5.4**　完全图 K_n 的最大匹配有多少条边？

✍ **思考题 5.5**　完全二分图 $K_{m,n}$ 的最大匹配有多少条边？

✍ **思考题 5.6**　图的两个匹配的并集的边导出子图的每个连通分支的结构有什么特征？

✍ **思考题 5.7**　图的两个匹配的对称差的边导出子图的每个连通分支的结构有什么特征？

对于图 $G = \langle V, E\rangle$ 和匹配 $M \subseteq E$，若路 P 交替经过集合 M 和 $E \setminus M$ 中的边，则 P 称作 M **交错路**（alternating path）。起点和终点未被 M 饱和的非平凡 M 交错路称作 M **增广路**（augmenting path）。例如，对于图 5.2（a）所示的图，若 $M = \{e_1, e_3\}$，则路 $v_7, e_8, v_1, e_1, v_2, e_2, v_3, e_3, v_4$ 是 M 交错路，它不是 M 增广路；$v_7, e_8, v_1, e_1, v_2, e_2, v_3, e_3, v_4, e_4, v_5$ 是 M 增广路。

✍ **思考题 5.8**　每个匹配都有交错路和增广路吗？若有，则唯一吗？

✍ **思考题 5.9**　如何利用增广路得到一个更大的匹配？

增广路和最大匹配有着紧密联系。下述**贝尔热定理**（Berge's theorem）[15] 为最大匹配给出了一个充要条件。

定理 5.1（贝尔热定理）　对于图 $G = \langle V, E\rangle$ 和匹配 $M \subseteq E$，M 是最大匹配当且仅当 G 不含 M 增广路。

✍ **思考题 5.10**（◊♦）　证明定理 5.1。

人物简介

贝尔热（Claude Jacques Berge），1926 年出生于法国。

5.1.2　算法

如何找出图中的最大匹配？对于二分图，这个问题可以通过两种不同的算法解决：匈牙利算法，或更高效的霍普克罗夫特-卡普算法；对于非二分图，这个问题可以通过花算

法解决。

1. 匈牙利算法

匈牙利算法（Hungarian algorithm）[16] 逐步构造最大匹配，每步利用当前匹配的一条增广路得到一个更大的匹配。

人物简介

匈牙利算法的作者：库恩（Harold William Kuhn），1925 年出生于美国。他命名“匈牙利算法”是由于算法主要基于之前两位匈牙利数学家的工作。

匈牙利算法伪代码如算法 5.1所示。对于二分图 $G=\langle X\cup Y,E\rangle$，该算法从初值为空集的匹配 M 开始，每轮 do-while 循环尝试找一条 M 增广路 P（第 6 行），直至 G 中不存在 M 增广路，即 P 为 null（第 1、10 行）：若能找到，即 P 不为 null（第 7 行），则计算 P 经过的边的集合和 M 的对称差，得到一个包含边的数量更多的匹配（第 8 行），本轮尝试提前中止（第 9 行）并进入下轮尝试（第 10 行）。算法运行结束时，输出最大匹配 M（第 11 行）。其中，找 M 增广路的方法是调用 DFSAP 算法，这是一种扩展的 DFS 算法。具体而言，从顶点子集 X 中每个在本轮 do-while 循环中未被 DFSAP 算法访问过（即 visited 属性值为 false）且未被 M 饱和的顶点 r 出发运行 DFSAP 算法，尝试找一条以 r 为起点的 M 增广路 P（第 4~6 行）。

算法 5.1: 匈牙利算法伪代码

输入: 二分图 $G=\langle X\cup Y,E\rangle$
初值: 集合 M 初值为 $\varnothing$

```
1  do
2      foreach u ∈ (X ∪ Y) do
3          u.visited ← false;
4      foreach r ∈ X do
5          if r.visited = false 且 r 未被 M 饱和 then
6              P ← DFSAP(G, r, M);
7              if P ≠ null then
8                  M ← P经过的边的集合 Δ M;
9                  中止 foreach 循环;
10 while P ≠ null;
11 输出 (M);
```

✍ **思考题 5.11** do-while 循环运行多少轮？

DFSAP 算法伪代码如算法 5.2所示。该算法从图中的一个指定顶点 u 出发，按 DFS

的方式有序地遍历图，并返回一条找到的 M 增广路，或返回 null 表示未找到。相比于算法 2.1所示的 DFS 算法，DFSAP 算法的第一项扩展是限制了可访问的邻点，使 DFS 树中以根顶点（即匈牙利算法中的顶点 r）为起点的每条路都是 M 交错路（第 6 行）。DFSAP 算法的第二项扩展是：若非根顶点 u 未被 M 饱和（第 2 行），则 DFS 树中从根顶点到 u 的 M 交错路是 M 增广路，算法返回这条路（第 3 行）；否则，若从 u 对邻点 v 的递归调用找到 M 增广路 P_v，则算法返回 P_v（第 7~9 行）；否则，未找到 M 增广路，算法返回 null（第 10 行）。

✍ **思考题 5.12**　如何高效地判定 DFS 树中从根顶点到顶点 v 的路是 M 交错路？

算法 5.2: DFSAP 算法伪代码

```
   输入: 图 G = ⟨X ∪ Y, E⟩，顶点 u，匹配 M
 1 u.visited ← true;
 2 if u 未被 M 饱和 且 u ≠ DFS 树的根顶点 then
 3 |   return DFS 树中从根顶点到 u 的路;
 4 else
 5 |   foreach (u, v) ∈ E do
 6 |   |   if v.visited = false 且 DFS 树中从根顶点到 v 的路是 M 交错路 then
 7 |   |   |   P_v ← DFSAP(G, v, M);
 8 |   |   |   if P_v ≠ null then
 9 |   |   |   |   return P_v;
10 |   return null;
```

例如，对于图 5.1（b）所示的二分图，匈牙利算法输出的最大匹配可能是 $\{e_1, e_4, e_6\}$，也可能是 $\{e_2, e_5, e_7\}$ 等，具体输出取决于顶点的访问顺序。以输出 $\{e_1, e_4, e_6\}$ 为例，该算法运行过程示意图如图 5.3 所示。

下述思考题有助于理解匈牙利算法的正确性。

✍ **思考题 5.13** (♦)　对顶点 r 调用 DFSAP 算法返回 null 时，是否已尝试所有以 r 为起点的 M 交错路？会遗漏 M 增广路吗？（可以推迟至读完第 7章后再思考）。

✍ **思考题 5.14** (◇♦)　只从顶点子集 X 中的顶点出发运行 DFSAP 算法，会遗漏 M 增广路吗？

定理 5.2　匈牙利算法输出的集合 M 是图 G 的最大匹配。

✍ **思考题 5.15** (♦)　证明定理 5.2。

对于阶为 n、边数为 m 的图，匈牙利算法 do-while 循环的轮数为 $O(n)$；每轮 do-while 循环的时间复杂度为 $O(n+m)$，和 DFS 算法相同。因此，该算法的时间复杂度为 $O(n(n+m))$。

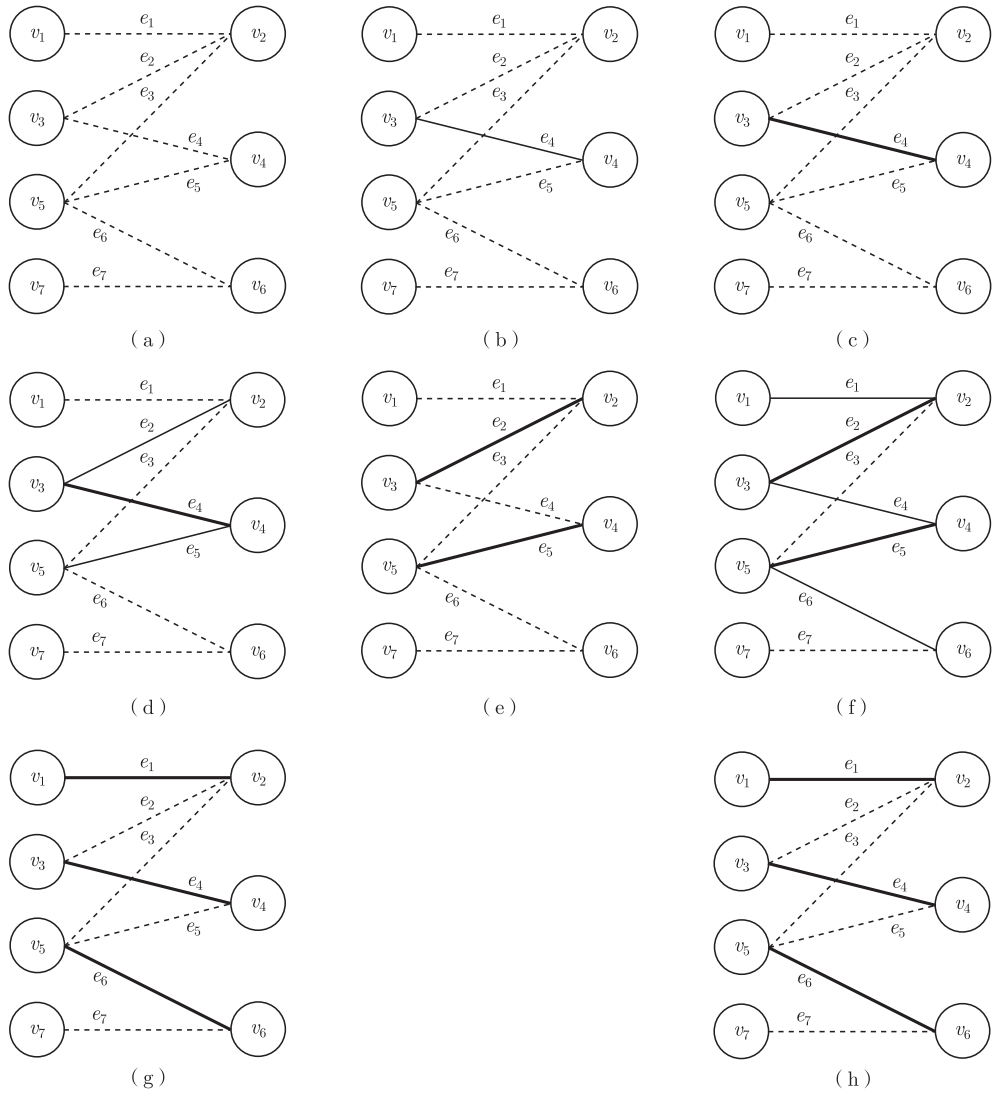


图 5.3　匈牙利算法运行过程示意图

（a）第 1 轮 do-while 循环开始前的 M；（b）第 1 轮 do-while 循环开始后的 P；（c）第 2 轮 do-while 循环开始前的 M；（d）第 2 轮 do-while 循环开始后的 P；（e）第 3 轮 do-while 循环开始前的 M；（f）第 3 轮 do-while 循环开始后的 P；（g）第 4 轮 do-while 循环开始前的 M；（h）第 4 轮 do-while 循环开始后的 P（不存在）

实线—M 增广路 P 经过的边　粗实线—匹配 M

2. 霍普克罗夫特-卡普算法

霍普克罗夫特-卡普算法（Hopcroft-Karp algorithm）[17] 对匈牙利算法做出了两项改进：第一项改进是通过扩展 BFS 算法找增广路，比通过扩展 DFS 算法找到的增广路更

短，从而减少单轮 do-while 循环的时间；第二项改进是每轮 do-while 循环尝试同时找多条增广路，匹配的规模增幅更大，从而减少 do-while 循环的轮数。

人物简介

卡普（Richard Manning Karp），1935 年出生于美国。由于在算法理论方面的持续贡献，尤其是对 NP 完全理论的贡献，他于 1985 年获得图灵奖。

霍普克罗夫特-卡普算法伪代码如算法 5.3所示。对于二分图 $G = \langle X \cup Y, E \rangle$，该算法从初值为空集的匹配 M 开始，每轮 do-while 循环尝试找一个 M 增广路的集合 $\mathcal{P}$（第 4 行），直至 G 中不存在 M 增广路，即 $\mathcal{P}$ 为空（第 1、7 行）：首先，HKInit 算法初始化 BFS 并返回队列 Q，存储顶点子集 X 中所有未被 M 饱和的顶点子集作为 BFS 的出发点（第 2 行）；接下来，HKBFS 算法运行扩展的 BFS 算法并返回通过 M 交错路找到的顶点子集 Y 中未被 M 饱和的顶点子集 Y'（第 3 行）；然后，HKPaths 算法找出以 Y' 中的顶点为终点的一组不经过相同顶点的 M 增广路 $\mathcal{P}$（第 4 行）；最后，相继计算 $\mathcal{P}$ 中每条路 P 经过的边的集合和 M 的对称差，得到包含边的数量更多的匹配（第 5~6 行）。算法运行结束时，输出最大匹配 M（第 8 行）。

算法 5.3: 霍普克罗夫特-卡普算法伪代码

输入: 二分图 $G = \langle X \cup Y, E \rangle$
初值: 集合 M 初值为 $\varnothing$

```
1 do
2     Q ← HKInit(G, M);
3     Y′ ← HKBFS(G, M, Q);
4     P ← HKPaths(G, Y′);
5     foreach P ∈ P do
6         M ← P经过的边的集合 Δ M;
7 while P ≠ ∅;
8 输出 (M);
```

HKInit 算法伪代码如算法 5.4所示。相比于算法 2.3所示的 BFS 算法，HKInit 算法同时从顶点子集 X 中所有未被匹配 M 饱和的顶点 u 出发（第 2~5 行）。相应地，顶点集 $X \cup Y$ 中每个顶点的 d 属性的含义也有所扩展：表示该顶点和所有出发点 u 间的最短距离。

HKBFS 算法伪代码如算法 5.5所示。相比于算法 2.3所示的 BFS 算法，HKBFS 算法沿用了 DFSAP 算法对 DFS 算法的两项扩展：限制了可访问的邻点（第 10 行）；将未被匹配 M 饱和的非根顶点 v 作为 M 增广路的终点（第 5 行）。与 DFSAP 算法不同的是，HKBFS 算法并不立刻返回 BFS 树中从根顶点到 v 的 M 增广路，而是先用集合 Y'

存储这些终点 v（第 6 行）。HKBFS 算法的第三项扩展是有可能提前中止 BFS：引入初值为 ∞ 的整数型变量 d'，表示被访问的首个未被 M 饱和的非根顶点的 d 属性值（第 7 行），即最短的 M 增广路的长度；在访问 d 属性值不超过 d' 的所有顶点后，不再访问 d 属性值更大的顶点，算法提前中止（第 3~4 行）。最终，Y' 存储所有 d 属性值为 d' 的未被 M 饱和的非根顶点。

算法 5.4: HKInit 算法伪代码

输入: 二分图 $G = \langle X \cup Y, E \rangle$，匹配 M
初值: 队列 Q 初值为空

```
1 foreach u ∈ (X ∪ Y) do
2     if u ∈ X 且 u 未被 M 饱和 then
3         u.visited ← true;
4         u.d ← 0;
5         入队列 (Q, u);
6     else
7         u.visited ← false;
8         u.d ← ∞;
9 return Q;
```

算法 5.5: HKBFS 算法伪代码

输入: 二分图 $G = \langle X \cup Y, E \rangle$，匹配 M，队列 Q
初值: 顶点子集 Y' 初值为 $\varnothing$；变量 d' 初值为 ∞

```
1  while Q 非空 do
2      v ← 出队列 (Q);
3      if v.d > d′ then
4          中止 while 循环;
5      else if v 未被 M 饱和 且 v.d > 0 then
6          Y′ ← Y′ ∪ {v};
7          d′ ← v.d;
8      else
9          foreach (v, w) ∈ E do
10             if w.visited = false 且 BFS 树中从根顶点到 w 的路是 M 交错路 then
11                 w.visited ← true;
12                 w.d ← v.d + 1;
13                 入队列 (Q, w);
14 return Y′;
```

HKPaths 算法的伪代码较简单，这里略去。该算法相继从集合 Y' 中的每个顶点出

发，沿顶点 d 属性值递减的方向，找出并返回极多的一组 M 增广路 $\mathcal{P}$，满足这些增广路不经过相同顶点，即之后找到的增广路不允许经过先找到的增广路经过的顶点。由于这组 M 增广路不经过相同顶点，它们经过的边的集合可以互不干扰地和匹配 M 计算对称差，因此，霍普克罗夫特-卡普算法每轮 do-while 循环中 M 的规模增幅有可能超过 1。

例如，对于图 5.1（b）所示的二分图，霍普克罗夫特-卡普算法输出的最大匹配可能是 $\{e_1, e_4, e_6\}$，也可能是 $\{e_2, e_5, e_7\}$ 等，具体输出取决于顶点的访问顺序和增广路的选择。以输出 $\{e_1, e_4, e_6\}$ 为例，该算法运行过程示意图如图 5.4 所示。

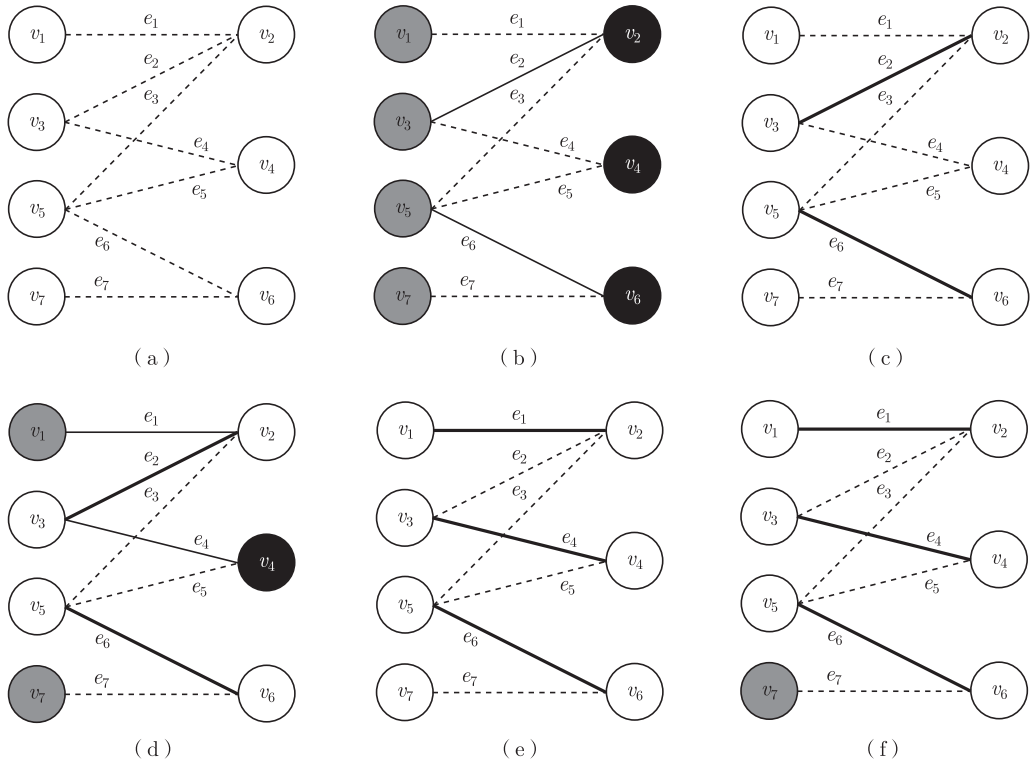


图 5.4　霍普克罗夫特-卡普算法运行过程示意图

（a）第 1 轮 do-while 循环开始前的 M；（b）第 1 轮 do-while 循环开始后的 Q、Y'、$\mathcal{P}$；（c）第 2 轮 do-while 循环开始前的 M；（d）第 2 轮 do-while 循环开始后的 Q、Y'、$\mathcal{P}$；（e）第 3 轮 do-while 循环开始前的 M；（f）第 3 轮 do-while 循环开始后的 Q、Y'、$\mathcal{P}$（空集）

实线—$\mathcal{P}$ 中的 M 增广路经过的边　粗实线—匹配 M　灰色顶点—队列 Q　黑色顶点—Y'

霍普克罗夫特-卡普算法的正确性较容易理解，其核心原理和匈牙利算法相似，只是将从单个顶点出发的 DFS 改为了从多个顶点同时出发的 BFS。

要理解霍普克罗夫特-卡普算法的时间复杂度，关键在于注意定理 5.3 和定理 5.4 给

出的事实。

定理 5.3 随着 do-while 循环轮数的增加，变量 d' 的值严格单调增。

定理 5.4 对于阶为 n 的图，所有 do-while 循环的变量 d' 有至多 $2\left\lfloor\sqrt{\frac{n}{2}}\right\rfloor+2$ 种值。

因此，对于阶为 n、边数为 m 的图，霍普克罗夫特-卡普算法 do-while 循环的轮数为 $O(\sqrt{n})$；每轮 do-while 循环的时间复杂度为 $O(n+m)$，和匈牙利算法相同。因此，该算法的时间复杂度为 $O(\sqrt{n}(n+m))$。

3. 花算法

匈牙利算法和霍普克罗夫特-卡普算法只适用于二分图。对于非二分图，这些算法未必能找到最大匹配。例如，对于图 5.5（a）所示的非二分图，匈牙利算法某轮 do-while 循环结束后得到匹配 $M=\{e_2,e_4,e_6\}$，下轮 do-while 循环从未被 M 饱和的顶点 v_1 出发运行 DFSAP 算法。事实上，此时存在 M 增广路 $v_1,v_2,v_3,v_4,v_5,v_6,v_7,v_8$。然而，当从顶点 v_3 对其邻点 v_4 和 v_7 递归调用 DFSAP 算法时，若先访问 v_7 而非 v_4，则匈牙利算法本轮 do-while 循环将找不到任何 M 增广路，算法中止并输出的不是最大匹配。霍普克罗夫特-卡普算法也存在同样问题，读者可以自行验证。

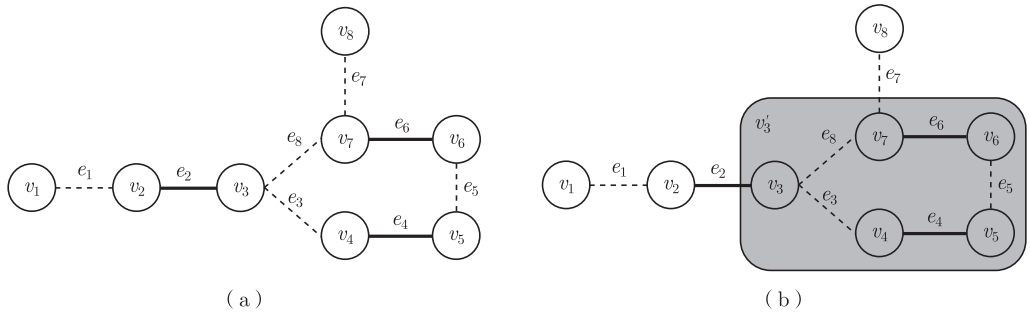


图 5.5 花算法原理示意图

（a）非二分图的匹配；（b）将花 $\{v_3,v_4,v_5,v_6,v_7\}$ 收缩为顶点 v_3'

粗实线—匹配　圆角矩形—花收缩成的新顶点

上述问题的成因是从顶点 v_1 到 v_7 存在两条 M 交错路：匈牙利算法若选择 M 交错路 $v_1,v_2,v_3,v_4,v_5,v_6,v_7$，则可找到 M 增广路；若选择 M 交错路 v_1,v_2,v_3,v_7，则找不到 M 增广路。然而，当从顶点 v_3 对其邻点 v_4 和 v_7 递归调用 DFSAP 算法时，并不知道应当先访问哪个邻点，有可能做出错误选择。