figure). Therefore we are allowed to change the matching edges in B without affecting the rest of M, in order to include the augmented matching edge (u,b) in G' involving b.

On the other hand, if b is an internal vertex on p', there exist $(u,b) \notin M'$ and $(b,v) \in M'$ for some (distinct) vertices u,v outside B (see the right side of the above figure). Since b is matched by M', we have $(w,v) \in M$. In the augmented matching in G', we have $(u,b) \in M' \oplus p'$ and $(w,v) \notin M' \oplus p'$. Therefore we are again allowed to change the blossom because w becomes free.

Conversely, suppose that G contains an augmenting path p with respect to M. If p does not include any vertex of B, then this path is augmenting in G' too with respect to M'. So suppose that p involves one or more vertices of B. Since B itself does not contain an augmenting path, at least one endpoint x of p is outside B. We follow p starting from x until we reach the first vertex y in B. Under the hypothesis that w is free, b is free in G'. Therefore the x,y path in G with y replaced by b is an augmenting path in G' with respect to M'. •

The second statement of the theorem is not necessarily true if w is not free in M. In this case, all vertices in the blossom are matched. Therefore if G contains an augmenting path p involving one or more vertices of B (as given in the above proof), its both endpoints must lie outside B. This path p therefore starts at a vertex x outside B, enters B (possibly multiple times), alternates along the edges inside and outside B, and eventually leaves B to end at a vertex y outside B. These branchings of p away from B (to x and y) let the DFS locate augmenting paths in G itself (although the contracted graph G' fails to contain an augmenting path). Here, I prefer to skip further discussion on this topic.

21.6.3 Weighted bipartite matching

Let $G = (V,E)$ be a bipartite graph with the bipartitioning of V into two disjoint independent sets X and Y. We assume that $|X| = |Y| = n$ (so $|V| = 2n$). Each edge $(x,y) \in E$ with $x \in X$ and $y \in Y$ has a weight $w(x,y)$. We assume that all weights are positive (or non-negative). If some edge (x,y) is absent between $x \in X$ and $y \in Y$, we add that edge, and set $w(x,y) = 0$. Therefore, we may view G as the complete bipartite graph $K_{n,n}$. The task is to find a matching M in G for which the sum

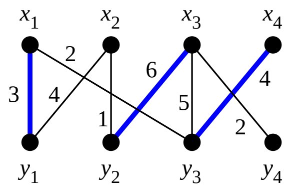
$$w(M) = \sum_{e \in M} w(e)$$

is as large as possible. This problem is known as the *weighted bipartite matching problem* or the *assignment problem*.

Harold Kuhn first proposes an algorithm to solve the problem in 1955. However, he attributes the algorithm to earlier work by Hungarian mathematicians, and named the algorithm as the *Hungarian algorithm*. In 1957, James Munkres establishes that the algorithm is strongly polynomial-time. Consequently, the Hungarian algorithm is also often referred to as the *Kuhn–Munkres algorithm*.

The Hungarian algorithm is based upon the concept of augmenting paths. We recall the definitions once again. Let M be a matching in G. A vertex $v \in V$ which is an endpoint of an edge in M is said to be *matched* by M. An unmatched vertex is also called *free*. A path in G on which edges alternate between M and $E \setminus M$ is called an *alternating path*. An alternating path is called *augmenting* if the first and the last vertices on the path are free.

The following figure demonstrates a bipartite graph. Only the edges with positive weights are shown. All missing edges may be assumed to be present and have zero weights. The thick edges show a matching in G. The path x_1, y_1, x_2 is alternating, but not augmenting because x_1 is not free. The path x_2, y_2, x_3, y_4 is augmenting. If we swap the thick and thin edges on this path, we get the matching $\{(x_1,y_1),(x_2,y_2),(x_3,y_4),(x_4,y_3)\}$ which has one more edge than the matching shown in the figure, but the weight of the matching is reduced by $6-(1+2)=3$ by the augmentation. This illustrates that augmentation in G is not a good idea toward solving the assignment problem. The Hungarian algorithm instead tries to find augmenting paths in a related graph $G_l = (V, E_l)$ on the same vertex set V as G.



A *labeling* of the vertices of G is a function $l : V \to \mathbb{R}$. A labeling of V is called *feasible* if for every $x \in X$ and $y \in Y$, we have

$$l(x) + l(y) \geqslant w(x,y).$$

The *equality (sub)graph* $G_l = (V, E_l)$ has the edge set E_l consisting only of the edges $(x,y) \in X \times Y$ for which

$$l(x) + l(y) = w(x,y).$$

We start with the initial feasible labeling

$$l(x) = \max\left\{ w(x,y) \mid y \in Y \right\} \text{ for all } x \in X,$$

and

$$l(y) = 0 \text{ for all } y \in Y.$$

It is easy to see that this is a feasible labeling. Moreover, each $x \in X$ is connected to at least one vertex in Y (actually, to all the vertices in Y for which the maximum is achieved). That is, $E_l \neq \emptyset$. Recall that a matching in a graph is called *perfect* if all vertices are matched. We plan to compute a perfect matching in G_l by finding augmenting paths in it. When there are no augmentation possibilities in G_l, we improve the labeling to a feasible labeling l' such that $G_{l'}$ contains edges in addition to those in G_l (that is, $E_l \subsetneqq E_{l'}$). This opens up new opportunities of detecting augmenting paths in the equality graph. The importance of the equality graph stems from the following theorem.

Kuhn–Munkres Theorem: Let l be a feasible labeling of G. If M is a perfect matching in G_l, then M is a maximum matching in G.

Proof Let M be any matching in G. We than have

$$w(M) = \sum_{(x,y)\in M} w(x,y) \leqslant \sum_{(x,y)\in M} [l(x)+l(y)] \leqslant \sum_{x\in X} l(x) + \sum_{y \in Y} l(y) = \sum_{v\in V} l(v).$$

It follows that the sum $\sum_{v\in V} l(v)$ is an upper bound on $w(M)$ for any matching M of G. If some matching M achieves this upper bound, it has to be maximum. If M is a perfect matching in G_l, it matches all vertices in V. Moreover, for any edge in E_l, we have $w(x,y) = l(x) + l(y)$. Therefore for a perfect matching M of G_l, we have

$$w(M) = \sum_{(x,y)\in M} w(x,y) = \sum_{(x,y)\in M} [l(x)+l(y)] = \sum_{x\in X} l(x) + \sum_{y \in Y} l(y) = \sum_{v\in V} l(v).$$

This completes the proof. •

To sum up, the Kuhn–Munkres theorem reduces the optimization problem of finding a maximum matching in G to the combinatorial problem of finding a perfect matching in G_l. All we need to do is to maintain and update a labeling function that leads to a solution of the combinatorial problem.

I have already described the initial construction of a feasible labeling l. Now, let me explain how a labeling can be improved to include additional edges in the equality graph. Let S be a non-empty subset of X. The *neighborhood* of S in the equality graph is defined as

$$N_l(S) = \{y \in Y \mid (x,y) \in E_l \text{ for some } x \in S\}.$$

Let M be a matching of G_l, which is not perfect. We choose any free vertex $x \in X$. We create a tree rooted at x such that all paths in the tree starting from the root are alternating. The set S consists of the vertices in the even levels of the tree, whereas the set T consists of the vertices in the odd levels of the tree. If a free vertex is ever discovered at an odd level, we have already obtained an augmenting path. Augmenting the matching M by swapping the edges of M and $E_l \setminus M$ on this path increases the number of edges in the matching by one. We start with the empty matching. After n augmentation steps, we get a perfect matching in G_l.

If $N_l(S) \neq T$, we can find $y \in N_l(S) \setminus T$. If y is a free vertex, an augmenting path is found. If y is matched, to a vertex $z \in X$, we add the edge (y,z) to the alternating tree (that is, we add y to T and z to S). We can no longer grow the alternating tree when we eventually have $N_l(S) = T$. This is when an improving of the labeling l becomes necessary. To that end, we compute the quantity

$$\alpha_l = \min\left\{ l(x) + l(y) - w(x,y) \mid x \in S, \text{ and } y \in Y \setminus T \right\}.$$

In this calculation, we consider all the edges in $S \times (Y \setminus T)$, including those of weight zero. Since $T = N_l(S)$, any edge (x,y) with $x \in S$ and $y \notin T$ is not in E_l, and so satisfies $l(x) + l(y) > w(x,y)$. Therefore α_l is a positive quantity. We decrement the label of each $x \in S$ by α_l, increment the label of each $y \in T$ by α, and keep the labels of other vertices unchanged.

For each $x \in S$ and $y \in T$, $l(x)$ decreases by the same quantity as $l(y)$ increases, and so the sum $l(x) + l(y)$ remains unchanged, and continues to satisfy $l(x) + l(y) = w(x,y)$. Thus, the edge (x,y) continues to stay in the equality graph.

For $(x,y) \in E_l$ with $x \in X \setminus S$ and $y \in Y \setminus T$, the labels $l(x)$ and $l(y)$ do not change, and continue to satisfy the equality $l(x) + l(y) = w(x,y)$, that is, the edge (x,y) continues to remain in the equality graph.

Finally, for each $x \in S$ and $y \in Y \setminus T$, the sum $l(x) + l(y)$ reduces by α_l. Therefore those edges $(x,y) \in S \times (Y \setminus T)$ for which the earlier sum $l(x) + l(y) - w(x,y) = \alpha_l$ are now included in the equality graph.

The steps of the Hungarian algorithm are now explicitly presented.

```
1. Start with the initial feasible mapping $l$, and the corresponding equality graph. Set $M = \emptyset$.
2. While $M$ is not a perfect matching in the equality graph, repeat:
    (a) Pick a free vertex $x \in X$.
    (b) Set $S = \{x\}$ and $T = \emptyset$.
    (c) While $N_l(S) \neq T$, repeat:
        (i) Pick $y \in N_l(S) \setminus T$.
        (ii) If $y$ is free, augment $M$, and go to Step 2.
        (iii) Otherwise, $y$ is matched, say, to $z \in X$. Set $T = T \cup \{y\}$ and $S = S \cup \{z\}$.
    (d) Here we have $N_l(S) = T$. Do the following two steps.
        (iv) Compute $\alpha$.
        (v) Decrement $l(x)$ by $\alpha$ for all $x \in S$, and increment $l(y)$ by $\alpha$ for all $y \in T$.
    (e) The condition $N_l(S) \neq T$ is again restored, so go to the top of the loop (c).
```

The working of the Hungarian algorithm is illustrated in Fig 83. The figure does not show the edges of weight zero, but these edges must be considered to be present in the graph. The initial labeling and the corresponding equality graph and shown at the top. For example, $l(x_3)$ is initialized to $\max\{w(x_3,y_1), w(x_3,y_2), w(x_3,y_3), w(x_3,y_4)\} = \max\{0,6,5,4\} = 6$. This maximum is achieved by y_2, so the initial equality graph contains the edge (x_3,y_2).

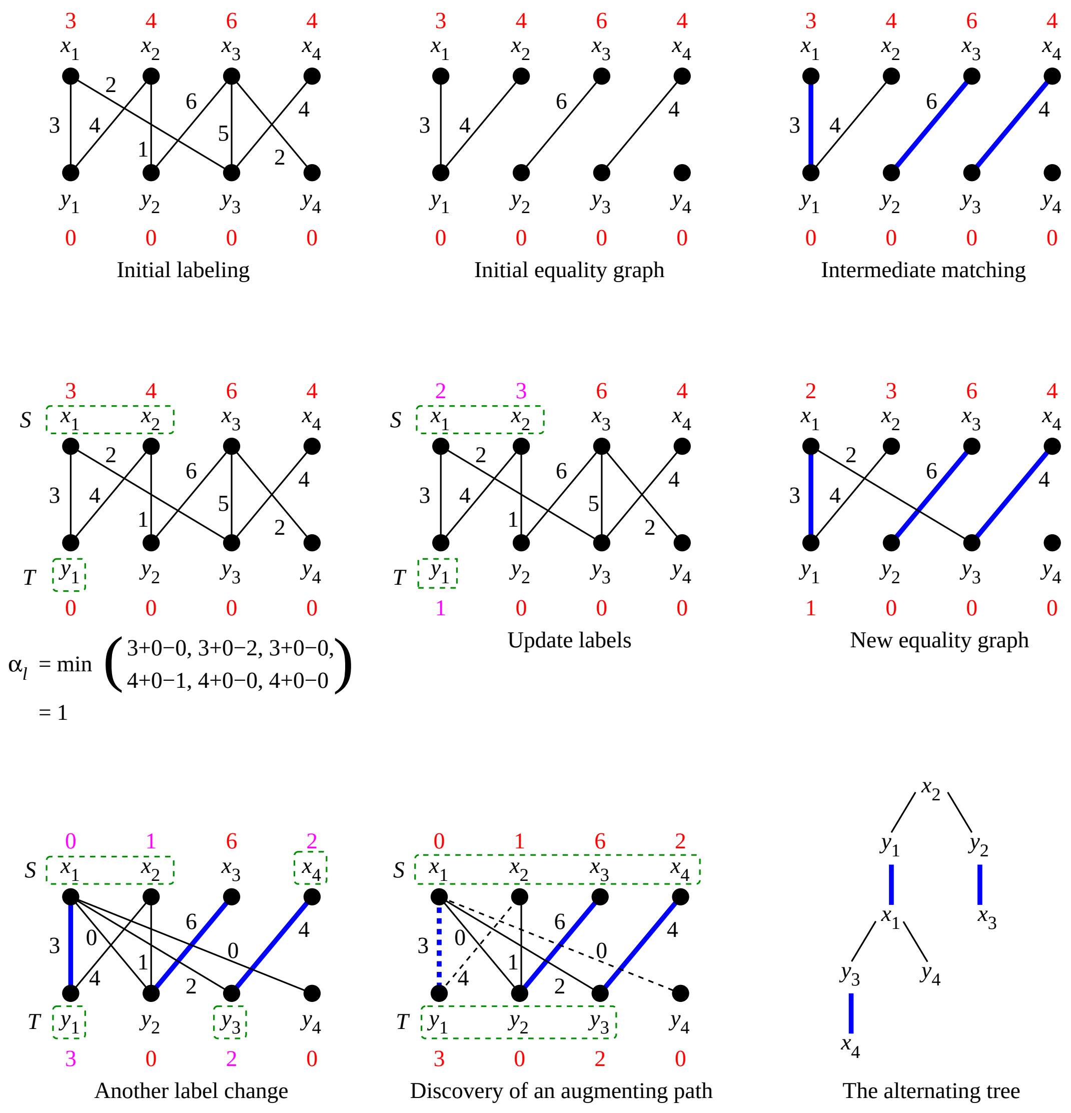
Now, suppose that a partial matching $\{(x_1,y_1),(x_3,y_2),(x_4,y_3)\}$ is computed by the algorithm. This was done by choosing $x = x_1,x_3,x_4$ in Step (b), by picking $y = y_1,y_2,y_3$ in Step (i), and by augmenting the initial empty matching three times in Step (ii). So far, there was no necessity of improving the initial labeling. The situation is described at the top right of Fig 83. The current matching is shown by the thick edges.

At this point, x_2 is free. So we start with $S = \{x_2\}$ and $T = \emptyset$. We have $N_l(S) = \{y_1\} \neq T$. We pick $y_1 \in N_l(S) \setminus T$, and see that y_1 is matched to x_1. So we update $S = \{x_1,x_2\}$, and $T = \{y_1\}$. Since $N_l(S) = \{y_1\} = T$ now, we need to improve the labeling. To this end, we compute $\alpha_l = 1$ as shown. We decrement the labels of x_1 and x_2 from 3 and 4 to 2 and 3, respectively. We also increment the label of y_1 from 0 to 1. This adds the new edge (x_1,y_3), and the neighborhood of S changes to $N_l(S) = \{y_1,y_3\}$.

We pick $y_3 \in N_l(S) \setminus T$. But y_3 is matched to x_4, so we update $S = \{x_1,x_2,x_4\}$ and $T = \{y_1,y_3\}$, and again face the condition $N_l(S) = T$ so another change of labeling is necessary. Now, we have

$$\begin{aligned}
\alpha_l &= \min\Big\{ l(x_1)+l(y_2)-w(x_1,y_2), l(x_1)+l(y_4)-w(x_1,y_4), \\
&\qquad l(x_2)+l(y_2)-w(x_2,y_2), l(x_2)+l(y_4)-w(x_2,y_4), \\
&\qquad l(x_4)+l(y_2)-w(x_4,y_2), l(x_4)+l(y_4)-w(x_4,y_4) \Big\} \\
&= \min\Big\{ 2+0-0, 2+0-0, 3+0-1, 3+0-0, 4+0-0, 4+0-0 \Big\} \\
&= \min\Big\{ 2,2,2,3,4,4 \Big\} \\
&= 2.
\end{aligned}$$

Figure 83: Working of the Hungarian algorithm



The minimum is achieved for three pairs, so the three edges (x_1,y_2), (x_1,y_4), and (x_2,y_2) are added to the equality graph after the relabeling of the vertices. Now, we have $S=\{x_1,x_2,x_4\}$, $T=\{y_1,y_3\}$, and $N_l(S)=\{y_1,y_2,y_3,y_4\}$. We first pick $y_2 \in N_l(S)\setminus T$ which is matched to x_3, so we update $S=\{x_1,x_2,x_3,x_4\}$ and $T=\{y_1,y_2,y_3\}$. Since both (x_1,y_2) and (x_2,y_2) are added, y_2 can be viewed as the child of either x_1 or x_2 in the alternating tree. Finally, we choose $y_4 \in N_l(S)\setminus T$, and notice that y_4 is free. Tracing the alternating tree gives the augmenting path x_2,y_1,x_1,y_4, and the matching augments to $\{(x_1,y_4),(x_2,y_1),(x_3,y_2),(x_4,y_3)\}$. This is a perfect matching in G_l, and corresponds to a maximum matching in G of total weight $0+4+6+4=14$.

Let us now analyze the performance of the Hungarian algorithm on a general bipartite graph with $|X|=|Y|=n$. First, let me argue that the algorithm terminates. That is, I want to show that so long as M is not perfect, we can eventually discover an augmenting path. If $N_l(S)\neq T$, we can grow the alternating tree, and the process cannot proceed indefinitely because there are only finitely many vertices. If we reach the state $N_l(S)=T$, we can improve the labeling to introduce new edges in G_l enlarging the neighborhood $N_l(S)$. Since Y contains free vertices, (at least) one such vertex must eventually end up in $N_l(S)$.

I now show that each augmentation of the matching can finish in $O(n^2)$ time. For each $y \in Y\setminus T$, define the quantity

$$\delta_y = \min\Big\{ l(x)+l(y)-w(x,y) \mid x \in S \Big\}.$$

All δ_y can be initially computed in $O(n^2)$ time.

In each step of growing the alternating tree, we add a new vertex to S. The δ_y value for each $y \in Y\setminus T$ can be updated in $O(1)$ time. Since $|Y\setminus T|\leqslant n$, and we can add at most n vertices to S, the total effort associated with growing the alternating tree is $O(n^2)$.

Let us now look at the relabeling stage. Since each relabeling introduces at least one new vertex of Y to $N_l(S)$, at most n relabeling stages are necessary. During each relabeling, one can compute $\alpha_l=\min\{\delta_y \mid y\in Y\setminus T\}$ in $O(n)$ time. After the relabeling, we subtract α_l from δ_y for all $y \in Y\setminus T$, again in $O(n)$ time. Therefore each relabeling can be done in $O(n)$ time.

It follows that each augmentation of M (by an edge) can be completed in $O(n^2)$ time. We start with the empty matching, and eventually arrive at a perfect matching having n edges, so the overall running time of the Hungarian algorithm is $O(n^3)=O(|V|^3)$.

21.6.4 Stable matching

Let $G=(V,E)$ be a complete bipartite graph with n vertices in each of the parts M and W. There are $n!$ perfect matchings possible for this graph, because any permutation of the vertices in W can be matched with the vertices of M.

Suppose that M represents a set of n men, and W a set of n women. They want to get engaged and marry. Each man has an order of preferences for the women in W. Likewise, each woman has an order of preferences for the men in W. A perfect matching N in G prescribes a set of n engagements for the men and the women. Take a pair $(m,w)\in E\setminus M$. Let (m,w') and (m',w) be chosen in N, where $m'\neq m$ and $w'\neq w$. The pair (m,w) is called *unstable* if *both* the following conditions hold.

(1) m prefers w to w'.
(2) w prefers m to m'.

A perfect matching N is called a *stable matching* if there are no unstable pairs for this matching. The problem of finding a stable matching is called the *stable matching problem* or the *stable marriage problem*. In what follows, I explain the *Gale–Shapley algorithm* to solve this problem.

The basic idea is that so long as a man is not engaged, he keeps on proposing to the women in the order of his preferences. A woman not yet engaged always accepts a proposal. However, if a proposal is made by m to a woman w already engaged to m', w checks which one of m and m' has higher preference for her. If m has higher preference, she changes her engagement with m' to a new engagement with m (so m' becomes not engaged again). On the other hand, if m' has higher preference than m, w rejects the proposal of m (so m continues to remain not engaged). The process stops when all men and women are engaged.

The engagements made during the running of the algorithm are provisional. Women accept proposals from men of higher preferences. Eventually, when all men and women are engaged, we obtain the final matching. The algorithm is more explicitly stated now.

```
Start with the empty matching $N$.
For each man, set the next-to-propose woman to the woman of his highest preference.
So long as $N$ contains less than $n$ edges, repeat:
    Take a man $m$ not engaged (a vertex of $M$ not matched by $N$).
    Let $w$ be the next-to-propose woman for $m$.
    If $w$ is not engaged (not matched by $N$), add the pair $(m,w)$ to $N$.
    else do the following:
        Let $(m',w) \in N$ currently.
        If $m$ has higher preference than $m'$ to $w$, replace $(m',w)$ by $(m,w)$ in $N$,
        else keep $N$ as it is.
    Update the next-to-propose woman for $m$ to his next preference.
Return $N$.
```

Before the analysis of the Gale–Shapley algorithm, let me furnish an example illustrating the working of the algorithm. Take $n = 4$. The men are named A, B, C, D, and the women E, F, G, H. To start with, the orders of preferences of the men and women are given. In order that the comparison of m with m' can be made by w in constant time, the preference lists for women should be maintained as illustrated in the third table below.

	1	2	3	4
A	E	H	F	G
B	H	G	E	F
C	E	H	F	G
D	H	F	E	G

Men's choices

	1	2	3	4
E	D	C	B	A
F	C	B	A	D
G	B	D	C	A
H	B	C	A	D

Women's choices

	A	B	C	D
E	4	3	2	1
F	3	2	1	4
G	4	1	3	2
H	3	1	2	4

Women's choices

We assume that the turns of the men come in the round-robin fashion. If so, the algorithm works as given in the following table. The matching N is shown only when it changes. Notice how A ends up with his last choice. G also gets her last preference. Feel sorry for A and G. E and F can improve upon their initial provisional engagements. H luckily gets her first choice at the beginning, and rejects all other proposals.

Man's action	Woman's reaction	Current matching (N)
–	–	$\emptyset$
A proposes to E	E accepts A	$\{(A,E)\}$
B proposes to H	H accepts B	$\{(A,E),(B,H)\}$
C proposes to E	E replaces A by C	$\{(B,H),(C,E)\}$
D proposes to H	H rejects D	
A proposes to H	H rejects A	
B is already engaged		
C is already engaged		
D proposes to F	F accepts D	$\{(B,H),(C,E),(D,F)\}$
A proposes to F	F replaces D by A	$\{(A,F),(B,H),(C,E)\}$
B is already engaged		
C is already engaged		
D proposes to E	E replaces C by D	$\{(A,F),(B,H),(D,E)\}$
A is already engaged		
B is already engaged		
C proposes to H	H rejects C	
D is already engaged		
A is already engaged		
B is already engaged		
C proposes to F	F replaces A by C	$\{(B,H),(C,F),(D,E)\}$
D is already engaged		
A proposes to G	G accepts A	$\{(A,G),(B,H),(C,F),(D,E)\}$

Let us now look at the correctness of the algorithm. First, observe that if a woman is already engaged, she continues to remain so, but she can improve her partner. On the contrary, a man having a provisional engagement may become not engaged in future. We first need to argue that no man runs out of options for proposing. If that happens for m, this means that he has proposed to all of the n women, and has been either rejected or replaced by each of them. Since n women cannot be engaged to $n-1$ men (m is excluded), there must exist a woman w not engaged. But m has proposed to all of the n women and, in particular, to w. Since w is not engaged, she must have accepted m, a contradiction. This argument also establishes that the size of the matching eventually reaches n, that is, the matching N produced by the Gale–Shapley algorithm is a perfect matching.

Next, we need to show that N is a stable matching. Choose any pair $(m,w) \in E \setminus M$. Let N contain the pairs (m,w') and (m',w) (N is a perfect matching). I show that at least one of the conditions (1) and (2) for an unstable pair does not hold for (m,w). Consider the two cases.

Case 1: m has never proposed to w.

Since m is eventually engaged with w', he must have proposed to w' at some point of time. Since men propose in the order of their preferences, w' has a higher preference to m than w. So Condition (1) does not hold.

Case 2: m has proposed to w.

If the decision of w is to reject m, she must be already engaged to a man of higher preference. That partner of w may or may not be m'. If not, she eventually improves her partner further to m'. On the other hand, if m is provisionally engaged to w, the engagement of w eventually changes to m' (in one or more replace steps), because she gets men of higher preferences (than m) later. In either case, Condition (2) does not hold.

To sum up, the Gale–Shapley algorithm terminates after producing a perfect matching which is stable. Each proposal by a man can be handled in $O(1)$ time. There can be at most n proposals from each of the n men. Therefore the running time of the algorithm is $O(n^2)$.

21.7 Huffman codes

Data compression is an important and useful tool that finds immense applications in many areas including archiving and multimedia systems. Let us here plan to compress text data only. A text string is a finite sequence of symbols. It is customary to use binary encoding of the symbols occurring in strings. For example, ASCII is a 7-bit encoding of characters (Roman letters, numerals, and punctuation and control symbols).

In many cases, different symbols come in a string with different frequencies. For example, in an English text, the letter e usually comes with the highest frequency, whereas the letters q and z come with very low frequencies. In order to compress English text, it is, therefore, useful to have a short code for e and longer codes for q and z. Since e occurs more often than q or z, the overall bit length of the encoded message shrinks under this strategy.

An encoded message needs to be decoded unambiguously. A fixed-length encoding is always unambiguous. However, if we use variable-length codes for different symbols, we may run into trouble. As an example, let a, e, q and z have the codes 001, 10, 1101001 and 101101, respectively. Then, an encoded string, that starts with 101101001, may be decoded as 10 followed by 1101001 implying eq or as 101101 followed by 001 implying za.

In order to avoid this difficulty, we may design the encoding in such a way that no prefix of the code of a symbol equals the code for another symbol. An encoding satisfying this property is called a *prefix code*. Fixed-length encoding evidently yields prefix codes. There exist other schemes too that enjoy this property. Our task is to arrive at a prefix code for which the overall length of a string is minimum, that is, we achieve the maximum amount of compression.

Binary trees can be used to represent a prefix code. The leaves of the tree correspond to the symbols being encoded. In order to obtain the code for a symbol, we follow the unique path from the root to the leaf corresponding to that symbol. We start with the empty string. Whenever we follow a left link, we append a 0, and whenever we follow a right link, we append a 1. Eventually, we reach the desired leaf and declare the string, obtained by the traversal, as the code of the symbol. For example, see Figure 84(f) and (g). Let us calculate the code for b. The unique path from the root to the leaf labeled b consists of moving right first and subsequently moving left twice. Thus, b has the code 100.

A binary tree T representing a prefix code is called a *prefix tree*. Let $a_1, a_2, \ldots, a_n$ denote all the leaves of the tree. Each leaf a_i stands for a symbol denoted also as a_i. Let w_i denote the relative frequency (also called *weight*) of a_i. We say that T is a prefix tree on the weights $w_1, w_2, \ldots, w_n$. Let l_i denote the length of the unique path from the root to the leaf a_i. Thus, l_i equals the length of the code for a_i. We take the weighted sum of these lengths.

$$L(T) = \sum_{i=1}^{n} w_i l_i.$$