Chapter 5 - Solutions

5.1 (a) 19

(b) 2

(c)

Edge included	Cut
AE	$\{A, B, C, D\}$ & $\{E, F, G, H\}$
EF	$\{A, B, C, D, E\}$ & $\{F, G, H\}$
BE	$\{A, E, F, G, H\}$ & $\{B, C, D\}$
FG	$\{A, B, E\}$ & $\{C, D, F, G, H\}$
GH	$\{A, B, E, F, G\}$ & $\{C, D, H\}$
CG	$\{A, B, E, F, G, H\}$ & $\{C, D\}$
GD	$\{A, B, C, E, F, G, H\}$ & $\{D\}$

5.2 (a)

Vertex included	Edge included	Cost
A		0
B	AB	1
C	BC	3
G	CG	5
D	GD	6
F	GF	7
H	GH	8
E	AE	9

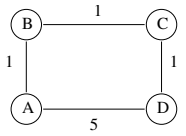
5.3 Since the graph is given to be connected, it will have an edge whose removal still leaves it connected, if and only if it is not a tree i.e. has more than $|V| - 1$ edges. We perform a DFS on the graph until we see $|V|$ edges. If we can find $|V|$ edges then the answer is "yes" else it is "no". In either case, the time taken is $O(|V|)$.

5.4 Let e_i, n_i denote the number of edges and vertices in the ith component. Since a connected graph on t vertices must have at least $t - 1$ edges,

$$|E| = \sum_{i=1}^{k} e_i \geq \sum_{i=1}^{k} (n_i - 1) = n - k$$

5.5 (a) The minimum spanning tree does not change. Since, each spanning tree contains exactly $n - 1$ edges, the cost of each tree is increased $n - 1$ and hence the minimum is unchanged.

(b) The shortest paths may change. In the following graph, the shortest path from A to D changes from $AB - BC - CD$ to AD if each edge weight is increased by 1.



5.6 Suppose the graph has two different MSTs T_1 and T_2. Let e be the lightest edge which is present in exactly one of the trees (there must be some such edge since the trees must differ in at least one edge). Without loss of generality, say $e \in T_1$. Then adding e to T_2 gives a cycle. Moreover, this cycle must contain an edge e' which is (strictly) heavier that e, since all lighter edges are also present in T_1, where e does not induce a cycle. Then adding e to T_2 and removing e' gives a (strictly) better spanning tree than T_2 which is a contradiction.

5.7 Multiply the weights of all the edges by -1. Since both Kruskal's and Prim's algorithms work for positive as well as negative weights, we can find the minimum spanning tree of the new graph. This is the same as the maximum spanning tree of the original graph.

5.8 Consider the edge $e = (s, u)$ having the minimum weight among all the edges incident on s. Since all the edge weights are positive, this is the unique shortest path from s to u and hence must be present in any shortest path tree. Also, Prim's algorithm includes this edge in construction of the MST. Since by problem 5.6, the MST must be unique (because of distict edge weights), all the shortest path trees and the MST must share at least one edge.

5.9 (a) False, consider the case where the heaviest edge is a bridge (is the only edge connecting two connected components of G.

(b) True, consider removing e from the MST and adding another edge belonging to the same cycle. Then we get e new tree with less total weight.

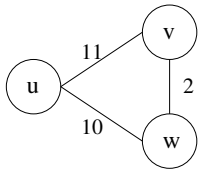
(c) True, e will belong to the MST produced by Kruskal.

(d) True, if not there exists a cycle connecting the two endpoints of e, so adding e and removing another edge of the cycle, produces a lightest tree.

(e) True, consider the cut that has u in one side and v in the other, where $e = (u, v)$.

(f) False,assume the graph consists of two adjacent 4-cycles,the one with very heavy edges of weight M and the other with very light edges of weight m. Let e be the edge in the middle with weight $m < w(e) < M$. Then the MST given by Kruskal, will pick all edges of weight m first and will not include e in the MST.

(g) False. In the following graph, the MST has edges (u, w) and (v, w) while Dijkstra's algorithm gives $(u, v), (u, w)$.



(h) False. In the previous graph the shortest path between u and v is the edge (u, v) but the only MST is $(u, w), (v, w)$

(i) True

(j) True. Suppose that the path from s to t has an edge longer than r. Then one of the edges of the r-path must be absent (otherwise there is cycle). Including this and removing the longer edge gives a better tree which is a contradiction.

5.10 Let $T \cup H = \{e_1, \ldots, e_k\}$. We use the cut property repeatedly to show that there exists an MST of H containing $T \cap H$.

Suppose for $i \geq 0$, $X = \{e_1, \ldots, e_i\}$ is contained in some MST of H. Removing the edge e_{i+1} from T divides T in two parts giving a cut $(S, G \backslash S)$ and a corresponding cut $(S_1, H \backslash S_1)$ of H with $S_1 = S \cap H$. Now, e_{i+1} must be the lightest edge in G (and hence also in H) crossing the cut , else we can include the lightest and remove e_{i+1} getting a better tree. Also, no other edges in T, and hence also in X, cross this cut. We can then apply the cut property to get that $X \cup e_{i+1}$ must be contained in some MST of H. Continuing in this manner, we get the result for $T \cap H = \{e_1, \ldots, e_k\}$.

5.11 The figure below shows the data-structure after the operations.

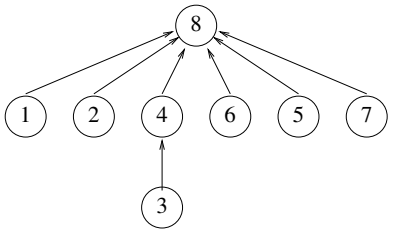
5.12 For the sake of convenience, assume that in case of a tie, we make the *higher* numbered root point to the *lower* numbered root. Let $n = 2^k$. Consider the following operations starting from the singleton sets $\{1\}, \ldots, \{2^k\}$

`union(1,2), union(3,4), ...,` `union`$(2^k - 1, 2^k)$

`union(2,4), union(6,8), ...,` `union`$(2^k - 2, 2^k)$

$\vdots$

`union`$(2^i, 2 \cdot 2^i)$`, union`$(3 \cdot 2^i, 4 \cdot 2^i)$`, ...,` `union`$(2^k - 2^i, 2^k)$



$\vdots$

`union(`$2^{k-1}, 2^k$`)` The above are $O(2^k)$ operations which make a tree in which the depth of element i is $\lfloor \log i \rfloor$ (the depth of element i increases in the first j steps, if 2^j is the largest power of 2 less that i). The cost of the above is $\sum_{i=1}^{k-1} i \cdot 2^{k-i} = O(2^k)$. We now perform a `find` for every element.
`find(1), ..., find(`2^k`)`
The cost of the find operations is $\sum_{i=1}^{2^k} \lfloor \log i \rfloor = O(k2^k) = O(n \log n)$. Hence, the above is a sequence of $O(n)$ operations taking time $O(n \log n)$.

5.13 Huffman's algorithm assigns codewords of length 1 to D, length 2 to A and length 3 to B and C. So, one possible encoding can be 0 for D, 10 for A, 110 for B and 111 for C.

5.14 (a) $a \rightarrow 0$, $b \rightarrow 10$,$c \rightarrow 110$, $d \rightarrow 1110$, $e \rightarrow 1111$.

(b) length $= \frac{1000000}{2} \cdot 1 + \frac{1000000}{4} \cdot 2 + \frac{1000000}{8} \cdot 3 + 2 \cdot \frac{1000000}{16} \cdot 4 = 1875000$

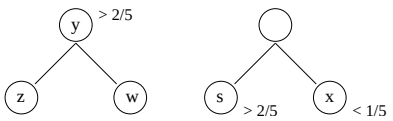
5.15 (a) $(f_a, f_b, f_c) = (2/3, 1/6, 1/6)$ gives the code $\{0, 10, 11\}$.

(b) This encoding is not possible, since the code for a (0), is a prefix of the code for c (00).

(c) This code is not optimal since $\{1, 01, 00\}$ gives a shorter encoding. Also, it does not correspond to a *full* binary tree and hence cannot be given by the Huffman algorithm.

5.16 a) Let s be the symbol with the highest frequency (probability) $p(s) > 2/5$ and suppose that it merges with some other symbol during the process of constructing the tree and hence does not correspond to a codeword of length 1. To be merged with some node, the node s and some other node x must be the two with minimum frequencies. This means there was at least one other node y (formed by merging of other nodes), with $p(y) > p(s)$ and $p(y) > p(x)$. Thus, $p(y) > 2/5$ and hence $p(x) < 1/5$.

Now, y must have been formed by merging some two nodes z and w with at least one of them having probability greater than $1/5$ (as they add up to more than $2/5$). But this is a contradiction - $p(z)$ and $p(w)$ could not have been the minimum since $p(x) < 1/5$.



b) Suppose this is not the case. Let x be a node corresponding to a single character with $p(x) < 1/3$ such that the encoding of x is of length 1. Then x must not merge with any other node till the end. Consider the stage when there are only three leaves - x, y and z left in the tree. At the last stage y, z must merge to form another node so that x still corresponds to a codeword of length 1. But, $p(x) + p(y) + p(z) = 1$ and $p(x) < 1/3$ implies $p(y) + p(z) > 2/3$. Hence, at least one of $p(y)$ or $p(z)$, say $p(z)$, must be greater than $1/3$. But then these two cannot merge since $p(x)$ and $p(y)$ would be the minimum. This leads to a contradiction.

5.17 The longest codeword can be of length $n - 1$. An encoding of n symbols with $n - 2$ of them having probabilities $1/2, 1/4, \ldots, 1/2^{n-2}$ and two of them having probability $1/2^{n-1}$ achieves this value.

5.18 ———————————

5.19 (a) It is enough to show that a symbol with frequency $p_i = 2^{-k_i}$ will be encoded with $log\frac{1}{p_i} = k_i$ bits. This is easily proved by induction : since all frequencies are a power of 1/2, the two least probable symbols will have the same probability $f = \frac{1}{2^k}$. Merging them according to the Huffman procedure into a new symbol, of probability $\frac{1}{2^{k-1}}$, will allow us to use our inductive hypothesis for the new set of $n-1$ symbols. The length of the first $n-2$ symbols is the same in the old and new tree, equal to $log\frac{1}{p_i}, i = 1, \cdots n-2$ and the length of the last two in the old tree is $1 + log\frac{1}{2^{k-1}} = k - 1 + 1 = k$.

(b) The largest possible entropy is when all the symbols have the same probability ,$1/n$. The entropy in that case is $logn$.The entropy is smallest when there exists a unique symbol with probability 1 (and all the others have probability 0). The entropy in that case is 0.

5.20 For each leaf of the tree, the edge incident on the leaf must be in the matching. Also, if we remove all these leaves, the leaves of the resulting tree have already been covered and the edges incident on them must *not* be in the matching. We can then construct the matching bottom up, including and excluding edges at alternate steps. If at any step, we need to include two edges incident on the same vertex, then no matching exists.

This can be done in linear time by maintaining the degree of every vertex and maintaining the leaves (nodes with degree 1) in a list. At each step, we update the degree of the endpoints of edges that we delete.

5.21 For a connected graph, removing the feedback arc set leaves a spanning tree. Hence, to find the minimum feedback arc set, we need to find the maximum spanning tree for every connected component of G. To do that, we run Kruskal's algorithm, negating the edge weights of the original graph. Once we have the Maximum Spanning Tree T, we output E' = the set of the edges that don't belong to T. Running time is the same as Kruskal.

5.22 (a) Consider an MST T which contains e. Removing e breaks the tree into two connected components say S and $V \backslash S$. Since all the vertices of the cycle cannot still be connected after removing e, at least one edge, say e' in the cycle must cross from S to $V \backslash S$. However, then replacing e by e' gives a tree T' such that $cost(T') \leq cost(T)$. Since T is an MST, T' is also an MST which does not contain e.

(b) If e is the heaviest edge in some cycle of G, then there is some MST T not containing e. However, then T is also an MST of $G - e$ and so we can simply search for an MST of $G - e$. At every step, the algorithm creates a new graph $(G - e)$ such that an MST of the new graph is also an MST of the old graph (G). Hence the output of the algorithm (when the new graph becomes a tree) is an MST of G.

(c) An undirected edge (u, v) is part of cycle iff u and v are in the same connected component of $G - e$. Since the components can be found by DFS (or BFS), this gives a linear time algorithm.

(d) The time for sorting is $O(|E| \log |E|)$ and checking for a cycle at every step takes $O(|E|)$ time. Finally, we remove $|E| - |V| + 1$ edges and hence the running time is $O(|E| \log |E| + (|E| - |V|) * |E|) = O(|E|^2)$.

5.23 To solve this problem, we shall use the characterization that T is an MST of G, if and only if for every cut of G, at least one least weight edge across the cut is contained in T.

The "only if" direction is easy since if the lightest edge across a cut is not in T, then we can include it and remove some edge in T that crosses the cut (there must be at least one) to get a better tree. To prove the "if" part, note that at each in Prim's algorithm, we include the lightest edge across some cut, which can be chosen from T. Since T is a possible output of Prim's algorithm, it must be an MST.

(a) Since the change only increases the cost of some other spanning trees (those including e) and the cost of T is unchanged, it is still an MST.

(b) We include e in the tree, thus creating a cycle. We then remove the heaviest edge e' in the cycle, which can be found in linear time, to get a new tree T'. It is intuitively clear that this algorithm should work, but a rigorous proof seems surprisingly tricky. To prove this, we argue that T' contains a least weight edge across every cut of G and is hence an MST.

Note that since the only changed edge is e, $T \cup \{e\}$ already includes a least weight edge across every cut. We only removed e' from this. However, any cut crossed by e', must also be crossed by at least one more edge of the cycle, which must have weight less than or equal to e'. Since this edge is still present in T', it contains a least weight edge across every cut.

(c) The tree is still an MST if the weight of an edge in the tree is reduced. Hence, no changes are required.

(d) We remove e from the tree to obtain two components, and hence a cut. We then include the lightest edge across the cut to get a new tree T'. We can now "build up" T' using the cut property to show that it is an MST.

Let $X \subseteq T'$ be a set of edges that is part of some MST, and let $e_1 \in T' \backslash X$. Then, $T' \backslash \{e_1\}$ gives a cut which is not crossed by any edge of X and across which e_1 is the lightest edge. Hence, $X \cup \{e_1\}$ is also a part of some MST. Continuing this, we can grow X to $X = T'$, which must be then an MST.

5.24 We first note that each $u \in U$ must have at least one neighbor in $V \backslash U$, else the problem has no solution. If T is the optimal tree, then $T \backslash U$ must be a spanning tree of $G \backslash U$. Moreover, it must be a minimum spanning tree since the nodes in U can be attached as leaves to *any* spanning tree. Hence, we first find an MST of $G \backslash U$ (in time $O(|E| \log |V|)$) and then for each $u \in U$, we add the lightest edge between u and $G \backslash U$ (in time $O(|E|)$).

5.25 To implement a counter of unspecified length, suppose we maintain it as a list of bits, with a pointer to the most significant bit. Each time we need to increase the length of the binary string, we can add a new element to the front of the list and update this pointer.

To increment, we set the first 0 from the right to 1 and set all the 1s to the right of it to 0. Since we start from the all 0 string, each bit is set to 1 once before it is set to 0. Hence, the total cost of k increments is at most twice the number of bits set to 1, which is $O(k)$ since each increment sets exactly one bit to 1.

Finally, note that the cost of a reset (setting all bits to 0), is at most the number of significant digits (if we maintain a pointer to the most significant 1 bit). This is at most the number of increment operations since the previous reset operation. Hence, instead of charging a reset operation, we double the cost of each increment operation which gives the cost of n increment and reset operations as $O(n)$.

5.26 We construct two graphs $G_{eq} = (V, E_{eq})$ and $G_{neq} = (V, E_{neq})$ where V $\{1, \ldots, n\}$. We have $(i, j) \in E_{eq}$ if $x_i = x_j$ is a constraint and $(i, j) \in E_{neq}$ if $x_i \neq x_j$ is a constraint. Since equality is an equivalence relation, all the variables in each connected component of G_{eq} must have the same value. The system of constraints is satisfiable if and only if there is no edge $(i, j) \in E_{neq}$ such that x_i and x_j are in the same component in G_{eq}. The decomposition in components and checking for edges can both be done in $O(m + n)$ time.

5.27 (a) $(3, 3, 1, 1)$ and $(100, 2, 1, 1)$ are both valid examples.

(b) i. Suppose the neighbors of v_1 in G are not $v_2, v_3, \ldots, v_{d_1+1}$. Therefore, there is some $i \in \{2, 3, \ldots, d_1 + 1\}$ such that $(v_1, v_i) \notin E$. In addition, since v_1 has d_1 neighbors, there must be some $j > d_1 + 1$ such that $(v_1, v_j) \in E$. In particular, $2 \leq i < j$. Next, note that the vertices v_i and v_j have d_i and $d_j - 1$ neighbors in $V - \{d_1\}$ respectively. Now, $i < j$ implies $d_i \geq d_j \geq 1$. Hence, $d_i > d_j - 1$, so there exists some vertex $u \in V - \{d_1\}$ that is a neighbor of v_i but not of v_j.

ii. Delete the edges (v_1, v_j) and (u, v_i) and add the edges (v_1, v_i) and (u, v_j).

iii. We repeatedly apply the above argument till v_1 has neighbors $v_2, \ldots, v_{d+1}$.

(c) From (b), we know that there exists a graph on n vertices with degree sequence $(d_1, \ldots, d_n)$ iff there exists a graph on $n-1$ vertices with degree sequence $(d_2 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n)$. The algorithm is as follows: sort (and relabel) the numbers $d_1, \ldots, d_n$ in decreasing order. Remove d_1 from the list; decrease $d_2, \ldots, d_{d_1+1}$ by 1 (output "no" if $d_1 + 1 > n$); remove any 0's in the resulting list, and recurse on the list of at most $n-1$ numbers. The base case is the empty list, where you output "yes". The running time is $O(n^2+m)$ (since you can sort numbers in a bounded range in linear time); with the appropriate data structure, you can avoid sorting at every stage, and obtain a $O(m+n)$ running time.

5.28 We construct a graph $G = (V, E)$ as follows. The vertex set consists of a vertex for each person. An edge $e = (u, v)$ between two vertices represents the relation 'person v knows person u' . The problem reduces of finding a subset V' of V such that in the induced graph each vertex has degree more than 5 and less than $|V'| - 5$. We do that iteratively : at the beginning, we look at all nodes and delete those with degree more than $|V| - 5$ and those with degree less than 5. We update our graph to be the induced graph with vertex set the remaining vertices. Then, we do the same procedure again (with the updated degrees for every node). We repeat the above until we end up with a graph G' where after running the procedure, we don't delete any vertices. The algorithm takes time $O(n)$ for every iteration of the above procedure, therefore a total of $O(n^2)$ time.

5.29 If we consider the binary tree with all strings of length k at level k and the left and right branches representing adding a 0 and 1 respectively, it contains all the binary strings and hence all the strings in the encoding (we only need to have number of levels equal to maximum length of a string). Also, since all intermediate nodes in a path from the root to a node v are prefixes of v, the strings of the encoding must be all at leaves since it is prefix-free.

To argue that the tree must be full, suppose for contradiction that a node u corresponding to a string s has only one child v corresponding to $s0$. Since a codeword is a leaf in the subtree rooted at u iff it has s, and hence also $s0$ as a prefix, replacing $s0$ by s in all these codewords gives a better encoding. However, this is a contradiction since we assumed our encoding to be minimal.

5.30 We first prove that if the number of symbols is odd then the encoding must correspond to a full ternary tree. We then make the number of symbols odd by adding a symbol with frequency 0 if needed. Once it is guaranteed that the optimal tree is full, we can simply proceed as in the binary case combining the three nodes with least frequencies at every step to form a single node. Since this reduces the number of nodes by 2, it still remains odd and we can carry on.

Suppose now that the number of symbols is odd and some non-leaf node u in the optimal tree does not have three children. Without loss of generality, we can assume that all children of u are leaves (else we can remove a leaf attached to some other node and add it as a child of u) and that u has two children (else we can delete its only child getting a better code).

Removing all children of nodes (say $u_1, \ldots, u_k$) which have two children, gives a full ternary tree. A full ternary tree can be built by always adding 3 children to some existent leaf and hence must have an odd number of leaves, since we started with just the root and the number of leaves increased by two at each step. Now, adding the children of $u_1, \ldots, u_k$ adds k more leaves (removes k and adds $2k$). Since the final number of leaves must still be odd, k must be even. We can then pair up $u_1, \ldots, u_k$ and transfer children between the pair so that half of them have 3 children, while the other half has 1 each. For each node u having only one child leaf v, we can now delete v getting a better code, which is a contradiction.

5.31 The algorithm in the chapter minimizes the sum $\sum_i f_i l_i$. To minimize $\sum_i c_i f_i l_i$, we simply treat $f_i c_i$ as the frequency of the ith word.

5.32 We simply proceed by a greedy strategy, by sorting the customers in the increasing order of service times and servicing them in this order. The running time is $O(n \log n)$. To prove the correctness, for any ordering of the customers, let $s(j)$ denote the jth customer in the ordering. Then

$$T = \sum_{i=1}^{n} \sum_{j=1}^{i-1} t_{s(j)} = \sum_{i=1}^{n} (n-i) t_{s(i)}$$

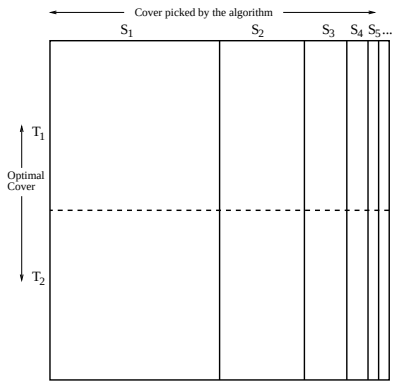
For any ordering, if $t_{s(i)} > t_{s(j)}$ for $i < j$, then swapping the positions of the two customers gives a better ordering. Since, we can generate all possible orderings by swaps, an ordering which has the property that $t_{s(1)} \leq \ldots \leq t_{s(n)}$ must be the global optimum. However, this is exactly the ordering we output.

5.33 We create a graph with a node for every variable. Also, for every implication, say $(x \wedge y \wedge z) \implies w$, we add the directed edges $(x, w), (y, w)$ and (z, w). With the edges, we also store which clause they belong to. Finally, at each variable we store a counter corresponding each clause in which appears, which is initially set to the number of variables on the LHS of the implication.

We start with assigning the value `false` to all the variables and then making some true as required by the implications. For each variable that is set to true, we decrement the counters of all its neighbors corresponding to the respective clauses. Any variable for which the counter corresponding to any clause becomes zero, is added to a list L, which contains the variable to be set to true. The list L starts with variables which start with counters zero i.e. clauses of the form (x).

5.34 Consider the base set $U = \{1, 2, \ldots, 2^k\}$ for some $k \geq 2$. Let $T_1 = \{1, 3, \ldots, 2^k - 1\}$ and $T_2 = \{2, 4, \ldots, 2^k\}$. These two sets comprise an optimal cover. We add sets $S_1, \ldots, S_{k-1}$ by defining $l_i = 2 + \sum_{j=1}^{i} 2^{k-j}$ and letting $S_i = \{l_{i-1} + 1, \ldots, l_i\}$ (take $l_0 = 0$).

Thus, S_1 contains $2^{k-1} + 2$ elements and the greedy algorithm picks this first. After the algorithm has picked i sets, each of T_1 and T_2 covers $2^{k-i-1} - 1$ new elements while S_{i+1} covers 2^{k-i-1} new elements. Hence, the algorithm picks the cover $S_1, \ldots, S_{k-1}$ containing $k - 1 = \log n - 1$ sets.



5.35 Since the algorithm in the box titled "A randomized algorithm for minimum cut" produces every minimum cut with probability at least $\frac{1}{n(n-1)}$ and the probabilities must sum to 1, the number of minimum cuts is at most $n(n-1)$.