Chapter 9 – Solutions

9.1 The algorithm first fixes any variables appearing as a 1-literal clause and then chooses a value for a variable in a 2-clause. We prove that the time taken is $O(n^2)$, where n is the number of variables.

Given a formula φ with n variables, we can first fix any variables appearing in a clause with just one variable. This takes $O(n)$ time. After this we pick a variable x_i appearing in a 2-clause and fix to value v_i. We have the following cases:

1. This may create other clauses with 1 variable, which may then be fixed. Suppose the number of clauses remaining is m' ($m' \leq m$). If this did not create an empty clause, the algorithm will proceed with m'.

   An important observation is that the formula φ' formed by these m' clauses is a *subformula* of φ. Since we have already satisfied all clauses in φ that are not in φ', φ is satisfiable *if and only if* φ' is satisfiable. Hence, we can simply proceed with φ' and will *never* need to backtrack to change the value of x_i.
2. If we derive an empty clause by fixing $x_i = v_i$, we know that that x_i should be set to $\bar{v}_i$. If setting $x_i = \bar{v}_i$ does not derive the empty clause, we again get a subformula φ' and the previous argument applies.
3. If both $x_i = v_i$ and $x_i = \bar{v}_i$ lead to the empty clause, the formula is unsatisfiable and we stop.

Notice that when we choose a value for a variable, either we derive an empty clause immediately (after fixing variables in the 1-clauses thus created) in time $O(n)$, or we simply carry on with the subformula derived and never backtrack. In the worst case, we derive an empty clause for one value of each variable and carry on with the other value. Hence the maximum time is $n * O(n) = O(n^2)$.

9.2 a) A subproblem is a new instance of the `RUDRATA PATH` on a subgraph of the original graph, starting at a given vertex $v \in V$. This allows us to progressively construct a path by removing vertices which have already been visited.

b) `choose` can be implemented by picking the subproblem defined on the smaller graph. This is reasonable as smaller graphs will be evaluated faster. Moreover, this will reduce the space complexity of the algorithm, as a single active path must be kept in the tree of possible Rudrata paths.

c) To `expand` a subproblem P defined as `RUDRATA PATH` starting at s on a graph G, break it down into subproblems $P_1, ..., P_{deg_G(s)}$, where the ith subproblem is an instance of `RUDRATA PATH` on $G - s$ starting at the ith neighbor of s in G.

9.3 a) A suproblem is a new instance of `SET COVER` in which the set of elements is a subset of the original set of elements, and the collection of sets is a subset of the original collection of sets.

b) `choose` can be implemented by picking the subproblem with the least number of elements.

c) `expand` a subproblem by considering all subproblems obtained by letting one set S into the cover. Then, each subproblem will consists of all the elements not in S and all the original sets except S.

d) Many `lowerbound` techniques exist. A simple way to bound the cost of a subproblem is to consider the ratio between the number of elements and the cardinality of the smallest set. More accurate ways would be to use the greedy algorithm described in Chapter 5. If such algorithm produces a solution of cost ALG on the subproblem, then, by the analysis in the book, the optimal cost for the subproblem must be at least $\frac{ALG}{\log n}$, where n is the number of elements in the subproblem.

9.4 Initially, let G be the original graph and $I = \emptyset$. Repeat the process below until $G = \emptyset$.

```
    Pick the node $v$ with the smallest degree and let $I = I \cup \{v\}$.
    Delete $v$ and all its neighbors from the graph.
    Let G be the new graph.
```

Notice that I is an independent set by construction. At each step, I grows by one vertex and we delete at most $d + 1$ vertices from the graph (since v has at most d neighbors). Hence there are at least $|V|/(d + 1)$ iterations. Let K be the size of the maximum independent set. Then the previous argument implies that

$$|I| \geq \frac{|V|}{d+1} \geq \frac{K}{d+1}$$

9.5 a) Let e be an edge present in T' but not in T. This creates a unique cycle. Also, this cycle must contain at least one edge e' which is not in T' (all the edges in the cycle cannot be from T' since T' is a tree!). Swapping (e, e') gives a tree T_1 which has one more edge from T' as compared to T. We can continue this procedure, until we replace all the edges in T'. Since we always remove edges not in T', and T' has $|V| - 1$ edges, this takes at most $|V| - 1$ swaps.

(b) Given T and T', we include an edge $e = (u, v)$ from T' which is not present in T. This creates a cycle. Now we obtain T_1 by removing one edge from this cycle. However, we need to argue that there exists at least one edge in the cycle which has at least as much weight as e, so that cost(T_1) $\geq$ cost(T_0).

To argue this, consider a cut which separates u and v. Then edge (u, v) goes across this cut. Since u and v are on opposite sides of the cut, and the other edges of the cycle form a path between u and v, at least one more edge e' from the cycle must cross the cut. Also, since an edge in the MST must be the lightest edge across a cut separating its endpoints, $w(e') > w(e)$. Thus, we can remove e'. Continuing this argument gives the desired sequence.

(c) We will now strengthen the previous argument to say that if T is not an MST, then there must be an edge e in an MST T' and an edge e' in T, such that swapping (e, e') strictly *reduces* the cost. Suppose for every edge e in the MST, all the edges in the cycle produced by including e have weight greater than or equal to e. Then by the previous argument, T contains the lightest edge across every cut, and hence must be an MST. Since we assumed T is not an MST, there must be a cost-reducing swap. This proves that there are no local minima and our algorithm always finds an MST.

To bound the running time, note that after each swap, one edge in the tree is swapped with a lower weight edge. Consider an edge e in the tree. This may be swapped with some edge e_1, e_1 with e_2 and so on. Since the weights must decrease in this sequence, the length of sequence is at most $|E|$. Also, the tree has $|V| - 1$ edges to begin with. Hence the total number of swaps is $O(|V||E|)$. Also, at each iteration, all possible swaps can be checked in $O(|V||E|)$ time (why?). Hence the total running time is at most $O(|V|^2|E|^2)$.

9.6 Let T be the minimum Steiner tree having cost C. Then we can follow the shape of the Steiner tree to obtain a tour of cost $2C$ which passes through all the vertices in the Steiner tree. Let i, j, k be adjacent vertices in the path. Using the triangle inequality, we know that $d_{ik} \leq d_{ij} + d_{jk}$. Hence, we can "bypass" an intermediate vertex j and connect i and k directly if we want, without increasing the cost.

Using this trick, we can bypass all the vertices *not* in V' that are present in the path, and also the vertices of V' which are being visited twice. This gives a path of cost at most $2C$, which passes though all the vertices of V' exactly once. Hence, this path is a spanning tree for V' of cost $2C$. This implies $cost(MST) \leq 2C$, thus giving the desired approximation guarentee.

9.7 a) This is the same problem as finding a (s_1, s_2)min-cut, which can be done by a maximum flow computation in polynomial time.

b) Find a (s_1, s_2)mincut $E_1 \subseteq E$ using maximum flow. Suppose s_1 and s_3 fall on the same side of the cut (the other case is symmetric). Compute then a (s_1, s_3)mincut $E_2 \subseteq E$ and output $E_1 \cup E_2$. To see this is a 2-approximation, consider the optimal multiway cut E^*: because $E*$ is both a (s_1, s_2)cut and a (s_1, s_3) cut, we have $|E_1| \leq E^*$ and $|E_2| \leq E^*$. Hence, $|E_1 \cup E_2| \leq |E_1| + |E_2| \leq 2|E^*|$, as required.

c) We need to define a neighborhood structure on the subsets of E whose removal disconnects the input terminals. The most natural choice is to have two subsets be neighbors if the size of their symmetric difference is less than some fixed number t. Notice that the size of each neighborhood is at most $|E|^t$.

9.8 a) Given a formula φ with m clauses as an instance of SAT, it is satisfiable if and only if the maximum number of satisfiable clauses is exactly m. Hence, an algorithm for MAX-SAT easily gives one for SAT.

b) Let X_j be a random variable obtaining value 1 if the j th clause is satisfied and 0 otherwise. X_j gets a value 0 only when *all* the literals appearing in the clause are simultaneously false in a random assignment, which happens only with probability $\frac{1}{2^{k_j}}$ where k_j is the number literals in the clause. Hence, $E[X_j] = 1 - \frac{1}{2^{k_j}} \geq 1/2$. If X is the number of satisfied clauses in the formula, then $X = \sum_{j=1}^{m} X_j$ and we have by linearity of expectation

$$E[X] = \sum_{j=1}^{m} E[X_j] = \sum_{j=1}^{m} \left(1 - \frac{1}{2^{k_j}}\right) \geq \frac{m}{2}$$

If all clauses contain exactly k literals, then $E[X] = m\left(1 - \frac{1}{2^k}\right)$, which gives an approximation factor of at least $1/\left(1 - \frac{1}{2^k}\right) = 1 + 1/(2^k - 1)$.

c) Let N_i denote the number of clauses containing at least one of the variables $x_1, \ldots, x_i$. We claim that after i variables have been assigned, the number of satisfied clauses is greater than $N_i/2$. The statement is true for x_1 by definition of the algorithm. When assigning x_i, we satisfy more than half of *all* the currently unsatisfied clauses containing x_i, which is more than $N_i - N_{i-1}$. Hence, this algorithm satisfies more than half the total number of clauses, achieving an approximation ratio of at least 2.

9.9 a) We first show that any solution S that the algorithm outputs must have the property that $w(S) \geq \frac{1}{2}\sum_{e \in E} w(e)$. Since the size of the maximum cut can at most be $\sum e \in Ew(e)$, this guarantees an approximation ratio of 2.

For contradiction suppose that $w(S) < \frac{1}{2}\sum_{e \in E} w(e)$. For each vertex $v \in V$, define $c(v)$ as the total weight of edges incident on v that cross the cut and $w(v)$ as the sum of weights of all the edges incident to v. Then

$$\sum_{v \in V} c(v) = 2w(S) < \sum_{e \in E} w(e) = \frac{1}{2}\sum_{v \in V} w(v)$$

since we count each edge twice, once for each of its endpoints. Thus, there must be at least one vertex v such that $c(v) < \frac{1}{2}w(v)$. If $v \in S$, define S' as $S \backslash \{v\}$, else define S' as $S \cup \{v\}$. All the edges incident to v that were not crossing the cut, will now cross the cut (adding $w(v) - c(v)$), but the ones that were crossing earlier may not (removing $c(v)$),hence

$$w(S') \geq w(S) - c(v) + (w(v) - c(v)) = w(S) + w(v) - 2c(v) > w(S)$$

Hence, there exists an S', which differs from S only by one element and $w(S') > w(S)$. This means that the algorithm could not have stopped at S, which is a contradiction.

b) At each iteration, we take $O(n)$ time to find if an S' exists with the required properties. Also, at each step the value of the cut increases and hence the total number of iterations is at most $O(\sum_{e \in E} w(e))$. Thus the running time is $O(n\sum_{e \in E} w(e))$. This is polynomial in the special case when all the edge weights are 1, but not in general.

9.10