National Taiwan University
Department of Electrical Engineering
Algorithms, Fall 2019

Handout #32
January 4, 2020
TAs: Chen-Hao Hsu, Yu-Jie Cai, Chien-Hao Tsou

Sample Solutions to Homework #5

1. (10) The Edmonds-Karp algorithm.

   The corresponding residual networks are shown in Figure 1 step by step as follows.

   (1) Figure 1(a) shows the first augmenting path $s \to a \to d \to t$ and the residual capacity is 3.

   (2) Figure 1(b) shows the second augmenting path $s \to b \to d \to t$ and the residual capacity is 1.

   (3) Figure 1(c) shows the third augmenting path $s \to c \to e \to t$ and the residual capacity is 4.

   (4) Figure 1(d) shows the final residual network and the maximum flow is $3 + 1 + 4 = 8$.

   Figure 1(e) shows a cut with capacity $3 + 1 + 4 = 8$. By the max-flow min-cut theorem, the capacity of the cut is 8 which is an upper bound of flow. Thus, the maximum flow is 8.

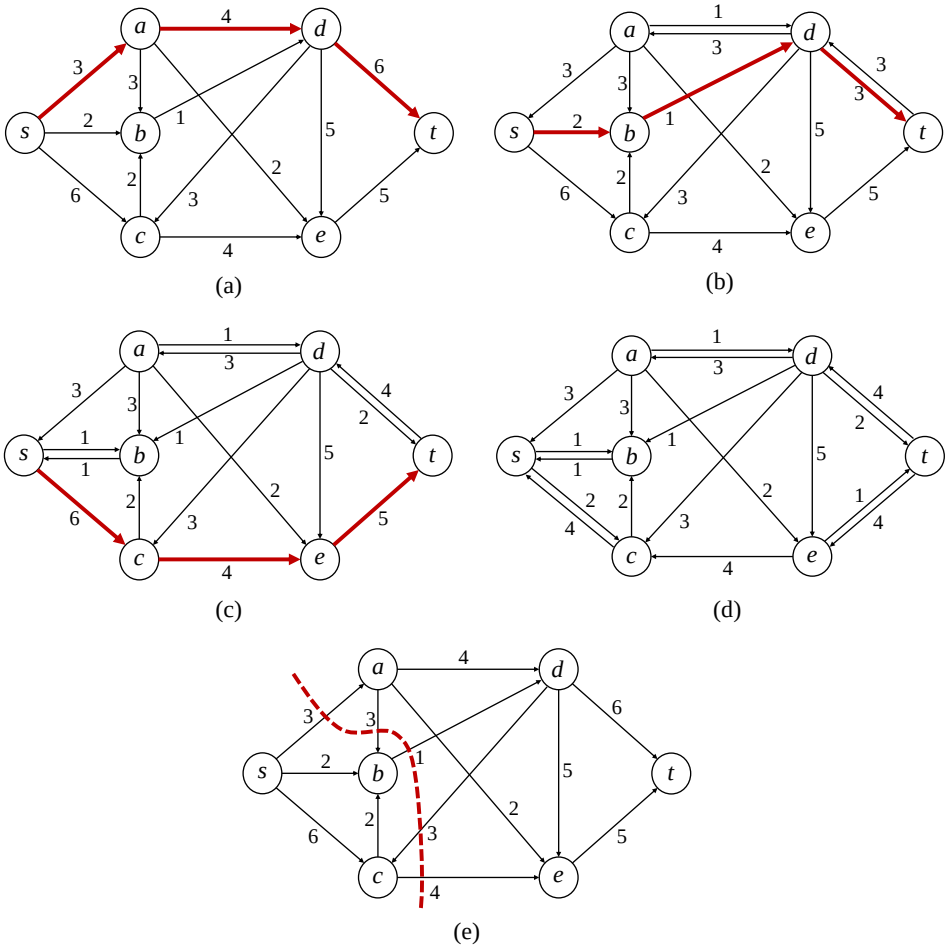

Figure 1: (a) The augmenting path with residual capacity 3. (b) The augmenting path with residual capacity 1. (c) The augmenting path with residual capacity 4. (d) Final residual network. (e) A cut with capacity 8.

2. (20) Problem 26-1 (pages 760–761).

a. To handle vertex capacity constraints, we can reduce a flow network $G = (V, E)$ with vertex and edge capacities to a flow network $G' = (V', E')$ with only edge capacities by the following steps. (1) Split each vertex v (with capacity c) into two vertices v' and v'' and then construct an edge from v' and v'' with capacity c (shown in Figure 2). (2) All the edges going into v are now going into v'; similarly, all the edges coming out of v are now coming out of v''. The new flow network $G' = (V', E')$ will have $|V'| = 2|V|$ vertices and $|E'| = |V| + |E|$ edges. The graph reduction takes $O(V + E)$ time.

$G' = (V', E')$ captures the vertex capacity constraints in $G = (V, E)$. Any flow going through vertex v with capacity c in $G = (V, E)$ must go through the edge (v', v'') in $G' = (V', E')$ which has capacity c due to the setting of v' and v'' (see step (2) above). Thus, we show that a maximum-flow problem with vertex and edge capacity constraints can be reduced to an ordinary maximum-flow problem (only edge capacity constraints).

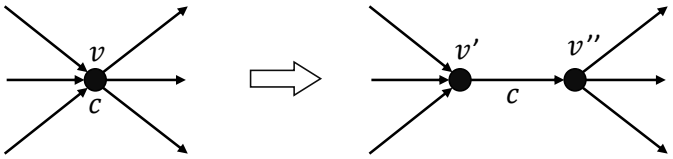

Figure 2: (a) Vertex v with capacity c. (b) The new flow network corresponding to the model in (a).

b. We can reduce the escape problem to a maximum-flow problem with edge and vertex capacity constraints. Construct a flow network $G = (V, E)$ by the following steps.

(1) Let each grid point be a vertex with unit capacity.
(2) Construct a bidirectional edge with unit capacity between two adjacent grid points.
(3) Create a source s and place a unit-capacity edge from s to each starting point.
(4) Create a sink t and place a unit-capacity edge from each boundary point to t.

Then, we can reduce the escape problem to a maximum-flow problem with vertex and edge capacity constraints. By performing the flow network reduction mentioned in (a), we can find the maximum flow by applying the Ford-Fulkerson algorithm. All the augmenting paths are the escape paths in the given grid. Because each vertex has unit capacity, all the augmenting paths are vertex-disjoint. Besides, all the augmenting paths are a unit-flow path since all the edges have unit capacity. Therefore, a solution to the maximum-flow problem corresponds to a solution to the escape problem. If the maximum flow is equal to m, there exist m vertex-disjoint paths from the starting points to the boundaries. Otherwise, if the maximum flow is less than m, it implies that we cannot find m vertex-disjoint escape paths in the given grid.

Now, let us analyze the running time of the algorithm. First, it takes $O(n^2)$ time to construct the flow network $G = (V, E)$ from an $n \times n$ grid. Second, reducing a flow network with edge and vertex capacities to an ordinary one (with only edge capacities) takes $O(V + E)$ time (see part (a)). Third, the Ford-Fulkerson algorithm takes $O(E|f^*|)$ time to find the maximum flow from s to t. Fourth, converting the solution to the maximum-flow problem into that to the corresponding escape problem takes $O(V + E)$ time. Finally, since $E = O(V)$, $V = O(n^2)$, and $|f^*| \leq 4n - 4$ (boundary points) for this problem, the total running time of the algorithm is $O(n^3)$ for an $n \times n$ grid.

3. (30) Problem 26-3 (pages 761–762).

a. We prove the statement by contradiction. Assume that there is a vertex $J_i \in T$ for a finite-capacity cut (S, T), and there exist some $A_k \in R_i$ such that $A_k \notin T$ (i.e., $A_k \in S$). By the construction of the flow network, there is an infinite-capacity edge from A_k to J_i if $A_k \in R_i$, which means that there

must be an infinite-capacity edge that crosses cut (S, T). Because of such infinite-capacity edges crossing the cut, the cut capacity is thus infinite, which contradicts the assumption. Therefore, we know that if $J_i \in T$ for a finite-capacity cut (S, T), then $A_k \in T$ for each $A_k \in R_i$.

b. Given a minimum finite-capacity cut (S, T). From part (a), we know that if $J_i \in T$, then $A_k \in T$ for each $A_k \in R_i$. If $p_i > \sum_{A_k \in R_i} c_k$, then job J_i should be accepted. The minimum finite-capacity cut (S, T) would cut the edge c_k. On the other hand, if $p_i \leq \sum_{A_k \in R_i} c_k$, then job J_i should not be accepted. (S, T) would cut the edge p_i. Therefore, Prof. Gore should accept the jobs $J_i \in T$ and hire experts in the subarea $A_k \in T$.

We define some notations here. Let $I_A = \{1, 2, \ldots, n\}$ and $I_J = \{1, 2, \ldots, m\}$ denote the sets of indices of all subareas and jobs respectively. Let $I_{A,S} = \{i : A_i \in S\}$, $I_{A,T} = \{i : A_i \in T\}$, $I_{J,S} = \{i : J_i \in S\}$, and $I_{J,T} = \{i : J_i \in T\}$. Let NR denote the net revenue. Then, we have

$$\begin{aligned} c(S,T) &= \sum_{\forall i \in I_{A,T}} c_i + \sum_{\forall i \in I_{J,S}} p_i \\ &= \sum_{\forall i \in I_{A,T}} c_i + \Big(\sum_{\forall i \in I_J} p_i - \sum_{\forall i \in I_{J,T}} p_i \Big) \\ &= \sum_{\forall i \in I_J} p_i - \Big(\sum_{\forall i \in I_{J,T}} p_i - \sum_{\forall i \in I_{A,T}} c_i \Big) \\ &= \sum_{\forall i \in I_J} p_i - NR. \end{aligned}$$

Thus, $NR = \sum_{\forall i \in I_J} p_i - c(S, T)$.

c. From part (b), we know that in order to maximize the net revenue, we have to find the minimum cut in the flow network $G = (V, E)$. First, apply the Edmonds-Karp algorithm to find a maximum flow. Second, perform the DFS (or BFS) algorithm on the final residual network to identify the vertices in S and those in T. Finally, hire the experts in the subareas in T and accept the jobs in T.

Now, let us analyze the time complexity of the algorithm. The Edmonds-Karp algorithm takes $O(VE^2)$ time to find a maximum flow. Next, the DFS (or BFS) algorithm takes $O(V + E)$ time to build S and T. Finally, since $V = 2 + n + m$ and $E = n + m + r$, the total running time is $O((m + n)(m + n + r)^2)$.

4. (10)

The flow network for the given instance is shown in Figure 3. The maximum flow from s to t gives the maximum number of apartments that can be sold.

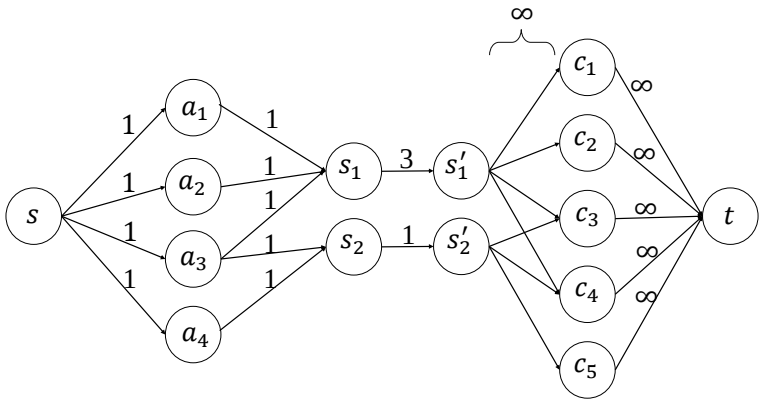

Figure 3: The flow network for the given instance in Problem 4. Let s, t, c_i, s_j, and a_k denote the source, the sink, customer i, salesman j, and apartment k, respectively.

5. (20) The 1-segment routing problem can be reduced to the maximum-flow problem by constructing a flow network $G = (V, E)$ as follows.

   (1) Let all the connections and segments be vertices in $G = (V, E)$.

   (2) If connection c_k can be placed within the column span of segment s_{ij}, place a unit-capacity edge from c_k to s_{ij}.

   (3) Create a source vertex s and place a unit-capacity edge from s to each connection vertex.

   (4) Create a sink vertex t and place a unit-capacity edge from each segment vertex to t.

   Figure 4 shows the flow network for the given instance. All the edges have unit capacity.

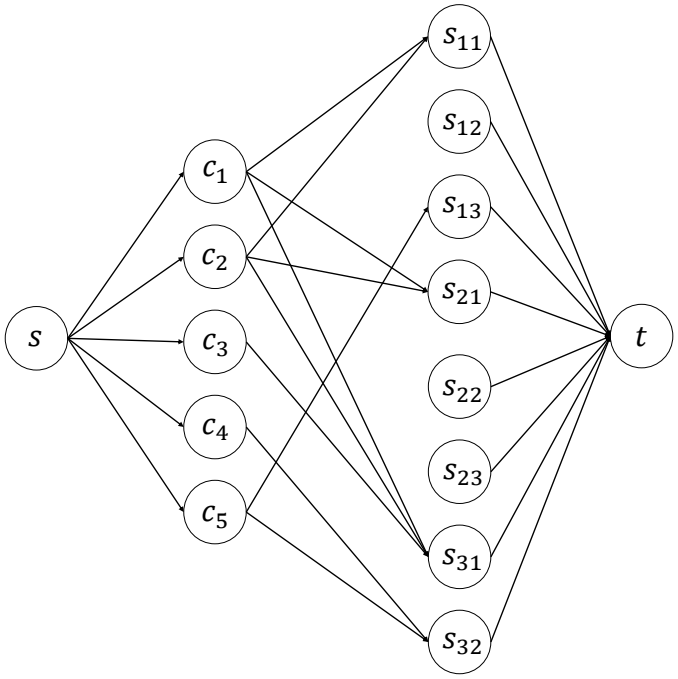

Figure 4: The flow network for the given 1-segment routing instance.

After the construction of the flow network, we can apply the Ford-Fulkerson algorithm to find the maximum flow from s to t. If the maximum flow is equal to the number of connections, there exists a feasible routing solution to the corresponding instance. We can simply route (place) connection c_k on segment s_{ij} if edge (c_k, s_{ij}) has flow. Each edge (c_k, s_{ij}) represents a legal routing solution for connection c_k. However, if the maximum flow is less than the number of connections, it implies that we cannot obtain a feasible routing solution.

Now, let us analyze the running time of the algorithm. Let n_c and n_s denote the number of connections and the number of segments respectively. First, it takes $O(n_c n_s)$ time to construct the flow network $G = (V, E)$. Second, the Ford-Fulkerson algorithm takes $O(E|f^*|)$ time to find the maximum flow from s to t. Third, converting the solution to the maximum-flow problem into that to the corresponding 1-segment routing problem takes $O(E)$ time. Finally, since $E = O(n_c n_s)$ and $|f^*| \leq \max(n_c, n_s)$ for this problem, the running time of the algorithm is $O((n_c n_s) \max(n_c, n_s))$.

6. (20) Concepts on polynomial-time complexity.

   (a) No. The time complexity of the algorithm for Exercise 16.2-2 is $O(nW)$, where n is the number of items and W is the maximum weight that the thief can put in his knapsack. For n items, there is a linear encoding to represent the index, value and weight of each item. However, the size of the encoding of W is only $k = \lg W$ bits. As a result, the time complexity of the given algorithm is actually $O(n2^k)$ in the size of input. It is not a polynomial-time algorithm.

(b) Yes. For V and E, there exists a linear encoding. Since C is an integer value, C is encoded in $k = \lg C$ bits. Therefore, the time complexity is actually $O(VE(\lg C)^2) = O(VEk^2)$ in the size of input, which is clearly polynomial.

7. (20) Exercise 34.4-7 (page 1086).

Let ϕ be an instance of 2-CNF-SAT, where each clause has exactly two literals. We can construct a directed graph $G_\phi = (V, E)$ as follows.

- The vertices in G_ϕ are the variables of ϕ and their negations. Thus, there are $2n$ vertices in V where n is the number of variables in ϕ.
- For each clause $(x \vee y)$ in ϕ, we place two edges $(\neg x, y)$ and $(\neg y, x)$ in G_ϕ. Thus, there are at most $2m$ edges in E where m is the number of clauses in ϕ.

Intuitively, the edges in G_ϕ capture the logical implications (i.e., $\rightarrow$) of ϕ. As a result, G_ϕ has a symmetry property: if there is an edge (x, y) in G_ϕ, then there is an edge $(\neg y, \neg x)$ in G_ϕ. Besides, by the transitivity property of implications, paths in G_ϕ are also valid implications. In other words, if $x \rightarrow y$ and $y \rightarrow z$, then $x \rightarrow z$.

Theorem 1 *ϕ is unsatisfiable* if and only if *there is a variable x such that there exist a path $x \rightsquigarrow \neg x$ and a path $\neg x \rightsquigarrow x$ in G_ϕ.*

Proof: We prove the two directions as follows.

- **There is a variable x such that there exist a path $x \rightsquigarrow \neg x$ and a path $\neg x \rightsquigarrow x$ in G_ϕ. $\Rightarrow$ ϕ is unsatisfiable.**
  Suppose that there is a variable x such that there exist a path $x \rightsquigarrow \neg x$ and a path $\neg x \rightsquigarrow x$ in G_ϕ. Since each path in G_ϕ is a valid implication, the path $x \rightsquigarrow \neg x$ implies the assignment of x must be assigned **False**. On the other hand, the path $\neg x \rightsquigarrow x$ implies the assignment of x must be assigned **True**. Clearly, the two contradictory assignments of x make ϕ unsatisfiable. Thus, if there is a variable x such that there exist a path $x \rightsquigarrow \neg x$ and a path $\neg x \rightsquigarrow x$ in G_ϕ, then ϕ is unsatisfiable.
- **ϕ is unsatisfiable. $\Rightarrow$ There is a variable x such that there exist a path $x \rightsquigarrow \neg x$ and a path $\neg x \rightsquigarrow x$ in G_ϕ.**
  Suppose that there is no variable with such two paths in G_ϕ. We are going to construct a satisfying assignment of ϕ to prove the statement. We repeat the following steps:

  i. Pick a vertex (literal) u with undefined value.

  ii. Assign **True** to all the vertices reachable from u and also assign **False** to their negations.

  This process is well-defined since if there were paths $u \rightsquigarrow v$ and $u \rightsquigarrow \neg v$, then there would be paths $v \rightsquigarrow \neg u$ and $\neg v \rightsquigarrow \neg u$ by the symmetry property, thus causing a contradiction with the assumption. Furthermore, if there were a path from u to a vertex already assigned **False** in a previous step, then u is a predecessor of that node and must be assigned **False** at that step.

  We repeat the step (i) and (ii) until all the vertices in G_ϕ have a truth assignment. Since we assume that there are no paths from any variable x to its negation $\neg x$ and back, all vertices will be assigned a truth value. Thus, the truth assignment satisfies ϕ.

□

We can test the condition of the Theorem by checking if there exists a variable x in G_ϕ such that the two paths $x \rightsquigarrow \neg x$ and $\neg x \rightsquigarrow x$ are both in G_ϕ. In other words, we can reduce a 2-CNF boolean formula ϕ to an implication graph $G_\phi = (V, E)$ and then assign a truth assignment to all the vertices in G_ϕ by the truth assignment process mentioned above.

Now, let us analyze the running time of the algorithm. First, constructing $G_\phi = (V, E)$ takes $O(n + m)$ time where n and m is the number of variables and the number of clauses in ϕ respectively. Second, the truth assignment process takes $O(V + E)$ time. Finally, since $V = O(n)$ and $E = O(m)$, the total time complexity is $O(n + m)$. Thus, 2-CNF-SAT$\in P$.

8. (40) Problem 34-1 (pages 1101–1102).

a. The decision problem can be formulated as follows: Given a graph $G = (V, E)$ and a positive integer $k \leq |V|$, is there an independent set of size $\geq k$ in G?

Now, let us prove that the independent-set problem (IS) is NP-complete. First, to show that the problem is in NP, we can use $V' \subseteq V$ as a certificate. Then, we can check whether V' is an independent set in $O(V + E)$ time by checking whether every edge in E is incident on at most one of the vertices in V'.

Second, we select the clique problem (CLIQUE) which is a known NP-complete problem. To prove that IS is NP-hard, we need to show that CLIQUE $\leq_P$ IS. In other words, we have to show how to reduce any instance of CLIQUE to an instance of IS in polynomial time. We define the complement of G: $\bar{G} = (V, \bar{E})$, where $\bar{E} = \{(u, v) : u, v \notin V, u \neq v, \text{and } (u, v) \notin E\}$. Here, we will show that G has a clique of size k *iff* $\bar{G}$ has an independent set of size k.

- **G has a clique of size k. $\Rightarrow$ $\bar{G}$ has an independent set of size k.**
  Suppose that G has a clique $V' \subseteq V$ with $|V'| = k$. That is, for each vertex pair $\langle u, v\rangle$ where $u, v \in V'$, $(u, v) \in E$. Then $(u, v) \notin \bar{E}$ by the definition of $\bar{G}$. Thus, V' is an independent set of size k in $\bar{G}$.
- **$\bar{G}$ has an independent set of size k. $\Rightarrow$ G has a clique of size k.**
  Suppose $\bar{G}$ has an independent set $V' \subseteq V$ with $|V'| = k$. For each vertex pair $\langle u, v\rangle$ where $u, v \in V'$, $(u, v) \notin \bar{E}$. Then $(u, v) \in E$ by the definition of $\bar{G}$. Thus, V' is a clique of size k in G.

Therefore, we prove that G has a clique of size k *iff* $\bar{G}$ has an independent set of size k.

The reduction algorithm takes as input an instance $\langle G, k\rangle$ of the clique problem. It computes the complement $\bar{G}$, which obviously takes polynomial time, specifically, $O(V^2)$. The output of the reduction algorithm is the instance $\langle \bar{G}, k\rangle$ of the independent-set problem. Therefore, we complete the proof that the independent-set problem is NP-complete.

b. Given a graph $G = (V, E)$ and the described black-box $B(G, k)$, we can find a maximum independent set as follows:

(1) Find the size of maximum independent set in G, which is denoted by k^*, by binary search with B.
(2) Initialize an empty set I.
(3) Repeat the following subroutine on each vertex $v \in V$:
If v is removed from G, just skip. Otherwise, construct a graph $G' = (V', E')$ by removing v and its associated edges from $G = (V, E)$. If $B(G', k^*)$ is true, then $G \leftarrow G'$; otherwise, $I \leftarrow I \cup \{v\}$ and $G \leftarrow G''$,where G'' is derived from G by removing all the vertices connected to v and their associated edges.
Finally, I is a maximum independent set in G with $|I| = k^*$.

Now, let us analyze the running time of the algorithm. First, step (1) and step (2) take $O(\lg V)$ time and $O(1)$ time respectively. Second, step (3) takes $O(V + E)$. Therefore, the running time of the algorithm is $(V + E)$.

c. If each vertex in graph $G = (V, E)$ has degree 2, then G must consist of some disjoint cycles. For each cycle, we can start from an arbitrary vertex on the cycle and sequentially label the vertices $0, 1, 0, 1, \ldots$ along the cycle. Finally, all the vertices that are labelled "1" form a maximum independent set in G.

A maximum independent set in G must be the union of the maximum independent sets in each cycle since the cycles are disjoint. By the proposed labelling method, we can always obtain an independent set in a cycle since no two vertices labelled "1" in the cycle such that there is an edge between them. For an n-vertex cycle where n is even (odd), the maximum size of independent set is $n/2$ ($\lfloor n/2 \rfloor$). Again, by the labelling method, the size of independent set in each cycle is always maximum since the number of vertices labelled "1" is $n/2$ ($\lfloor n/2 \rfloor$) for an even (odd) cycle. Thus the union of the maximum independent sets in each cycle must be a maximum independent set in G.

Now, let us analyze the running time of the algorithm. First, labelling all the vertices takes $O(V)$ time. Then, collecting the vertices labelled "1" takes $O(V)$ time as well. Therefore, the time complexity of the algorithm is $O(V)$.

d. To find an independent set in a bipartite graph $G = (V, E)$ (undirected), we can first find a maximum matching $M \subseteq E$ in G by the method in Section 26.3. Next, let $V_{unmatch}$ be the set of vertices on which no edge in M is incident. Let $V_{match} = V - V_{unmatch}$. For each edge $(u, v) \in M$, at least one vertex (i.e., u or v) is not connected to any vertices in $V_{unmatch}$ since if both u and v have an edge with the vertices in $V_{unmatch}$, then M is not a maximum matching. Then we select such the vertex from each edge $(u, v) \in M$ to form a subset $V'_{match} \subseteq V_{match}$, where $|V'_{match}| = |V_{match}|/2$. Finally, we can obtain a maximum independent set $V' = V_{unmatch} \cup V'_{match}$ in G.

We can prove that $V' = V_{unmatch} \cup V'_{match}$ is a maximum independent set in G by arguing that (1) $|V'|$ is the maximum size of an independent set in G and (2) V' is an independent set in G.

(1) For each edge $(u, v) \in M$, it is clear that at most one vertex can be in an independent set. In other words, it is impossible that both u and v are in an independent set. Thus, an upper bound of the size of an independent set is $|V| - |M| = |V_{unmatch}| + |V'_{match}| = |V'|$.

(2) First, $V_{unmatch}$ is an independent set in G, if not, there exists an edge (u, v) where $u, v \in V_{unmatch}$ and $u \neq v$ such that $(u, v) \cup M$ is also a matching, which contradicts that M is a maximum matching. Second, V'_{match} is an independent set in G, if not, M is not a maximum matching. Third, since V'_{match} is composed of the vertices that has no edges with the vertices in $V_{unmatch}$, we have that for each pair $(u, v) \in \{(s, t) : s \in V'_{match}, t \in V_{unmatch}, \}$, $(u, v) \notin E$. Thus, $V' = V_{unmatch} \cup V'_{match}$ is an independent set in G.

From (1) and (2), we prove that V' is a maximum independent set in G.

Now, let us analyze the running time of the algorithm. First, it takes $O(VE)$ time to find a maximum matching by the method in Section 26.3. Second, building $V_{unmatch}$ and V_{match} takes $O(V + E)$ time and building $V'_{unmatch}$ also takes $O(V + E)$ time. Therefore, the running time of the algorithm is $O(VE)$.

9. (60) Problem 34-3 (pages 1103–1104).

a. First, label every vertex in $G = (V, E)$ *unlabeled*. Pick an arbitrary vertex v_s in G as the starting vertex and label v_s as '1'. Then, apply the BFS algorithm to visit each vertex in G starting from v_s. Each time an unlabeled vertex v_i is visited, check the labels of its neighboring vertices. If v_i has two neighbors which are labeled as '1' and '2' respectively, then it implies G is not 2-colorable. Otherwise, if v_i's visited neighbors are labeled the same type, then label v_i the other type. If all the vertices are visited without any violation, a feasible solution for 2-coloring of G is determined. The time complexity is $O(V + E)$ if we use the adjacency list representation for the graph.

b. The decision problem of the graph-coloring problem L: Given an undirected graph $G = (V, E)$ and a positive integer k, is there a k-coloring for G?

If we can determine whether a k-coloring for a given graph $G = (V, E)$ exists or not in polynomial time, the graph-coloring problem can be solved by trying k from 1 to $|V|$ and applying at most $|V|$ times of L, which implies the graph-coloring problem can be also solved in polynomial time. On the other hand, if the graph-coloring problem can be solved in polynomial time, then the minimum number of colors k^* can be found in polynomial time. L can be solved by checking if $k \geq k^*$. Therefore, we prove that L is solvable in polynomial time if and only if the graph-coloring problem is solvable in polynomial time.

c. First, L can easily be verified in polynomial time by visiting all the vertices in $G = (V, E)$ to count the number of colors used and check if there are two neighboring vertices with the same color. Thus, we prove that $L \in$ NP. Second, given that 3-COLOR is NP-complete, L is NP-hard since 3-COLOR is a subset of L. To sum up, if 3-COLOR is NP-complete, the decision problem L is NP-complete.

d. Since a variable x_i and its negation $\neg x_i$ are both connected to the vertex RED, they cannot be colored c(RED) and thus have to be colored either c(TRUE) or c(FALSE). Moreover, since a variable x_i is connected to its negation $\neg x_i$, exactly one of them is colored c(TRUE) and the other is colored c(FALSE).

If a variable x_i is assigned TRUE/FALSE in ϕ, then we color x_i c(TRUE)/c(FALSE) and color its negation $\neg x_i$ c(FALSE)/c(TRUE). Thus, each triangle $\{x_i, \neg x_i, (\text{RED})\}$ are properly 3-colored. The graph containing just the literal edges consists of exactly these triangles and thus can be 3-colored.

e. Figure 5 shows Figure 34.20 in the textbook with the five shaded vertices labelled 1, 2, 3, 4, 5, respectively.

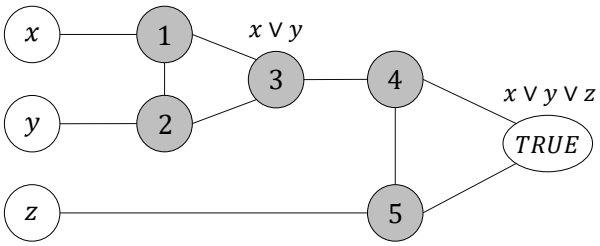

Figure 5: Figure 34.20 in the textbook.

By part (d), none of the vertices x, y and z can be colored $c(\text{RED})$. That is, x, y and z can be colored either $c(\text{TRUE})$ or $c(\text{FALSE})$. Now, we are going to argue that the widget is 3-colorable *iff* at least one of x, y and z is colored $c(\text{TRUE})$.

- **The widget is 3-colorable. ⇒ At least one of x, y and z is colored $c(\textbf{TRUE})$.**
  Suppose that none of x, y and z is colored $c(\text{TRUE})$, which implies that x, y and z are all colored $c(\text{FALSE})$. Since x and y are colored $c(\text{FALSE})$, vertex 1 and vertex 2 must be colored either $c(\text{TRUE})$ or $c(\text{RED})$. Besides, because vertex 1 and vertex 2 should be colored differently, exactly one of the two vertices is colored $c(\text{TRUE})$ while the other one is colored $c(\text{RED})$. Then, due to the colors of vertex 1 and vertex 2, vertex 3 should be colored $c(\text{FALSE})$. Similarly, since vertex 3 and vertex z are both colored $c(\text{FALSE})$, exactly one of vertex 4 and vertex 5 is colored $c(\text{TRUE})$ and the other one is colored $c(\text{RED})$, which indicates that the widget is not 3-colorable. Therefore, we have that if the widget is 3-colorable, then at least one of x, y and z is colored.
- **At least one of x, y and z is colored $c(\textbf{TRUE})$. ⇒ The widget is 3-colorable.**
  Table 1 gives the 3-coloring of the widget for all possible cases that at least one of x, y and z is colored $c(\text{TRUE})$. Therefore, we show that if at least one of x, y and z is colored $c(\text{TRUE})$, then the widget is 3-colorable.

Table 1: The 3-coloring of the widget. Let "x" denote don't-care color ($c(\text{TRUE})$ or $c(\text{FALSE})$).

x	y	z	1	2	3	4	5
$c(\text{TRUE})$	x	x	$c(\text{FALSE})$	$c(\text{RED})$	$c(\text{TRUE})$	$c(\text{FALSE})$	$c(\text{RED})$
x	$c(\text{TRUE})$	x	$c(\text{RED})$	$c(\text{FALSE})$	$c(\text{TRUE})$	$c(\text{FALSE})$	$c(\text{RED})$
$c(\text{FALSE})$	$c(\text{FALSE})$	$c(\text{TRUE})$	$c(\text{RED})$	$c(\text{TRUE})$	$c(\text{FALSE})$	$c(\text{RED})$	$c(\text{FALSE})$

Therefore, we prove that the widget is 3-colorable *iff* at least one of x, y and z is colored $c(\text{TRUE})$.

f. First, to show that the 3-COLOR problem (3-COLOR) is in NP, we can use a coloring c for a graph $G = (V, E)$ as a certificate. Then, we can check whether c is a 3-coloring in polynomial time (specifically, $O(V + E)$ time) by visiting all the vertices and edges in G to count the number of colors used and check if there are two neighboring vertices with the same color. Thus, we prove that 3-COLOR $\in$ NP.

Second, we select the 3-CNF-SAT problem (3-CNF-SAT) which is a known NP-complete problem. To prove that 3-COLOR is NP-hard, we need to show that 3-CNF-SAT $\leq_P$ 3-COLOR. In other words, we need to show how to reduce any instance of 3-CNF-SAT to an instance of 3-COLOR in polynomial time. Given a formula ϕ of m clauses and n variables, we construct a graph $G = (V, E)$ by the following steps described in the problem description, where $|V| = 2n + 5m + 3$ and $|E| = 3n + 10m + 3$. Clearly, the reduction takes polynomial time.

Now, let us prove that ϕ is satisfiable *iff* $G(V, E)$ is 3-colorable.

- **ϕ is satisfiable. ⇒ $G(V, E)$ is 3-colorable.**
  By part (d), for any truth assignment, G is 3-colorable regardless of the clause edges. Thus,

there is no literal edge that is incident on two same-color vertices. Since ϕ is satisfiable, every clause is **True**, which means that at least one of three literals in a clause is assigned **True**. Besides, for a truth assignment, each literal can only be **True** or **False**. By part (e), for each clause, if at least one of three literal vertices is colored $c(\text{TRUE})$, then the corresponding widget is 3-colorable. Therefore, $G(V, E)$ is 3-colorable.

- $G(V, E)$ **is 3-colorable.** $\Rightarrow$ ϕ **is satisfiable.**
  Since $G(V, E)$ is 3-colorable, every widget is 3-colorable. By part (e), if a widget is 3-colorable, then at least one of three literal vertices is colored $c(\text{TRUE})$, which implies that the corresponding clause in ϕ is **True**. Therefore, ϕ is satisfiable since all the clauses are **True**.

Therefore, we prove that ϕ is satisfiable *iff* $G(V, E)$ is 3-colorable.

In conclusion, we prove that 3-COLOR $\in$ NP and 3-COLOR $\in$ NP-hard. Thus, the 3-COLOR problem is NP-complete.

10. (30) Amortized analyses.

(a) Exercise 17.1-3 (page 456).

Let the cost of the ith operation be $c(i)$ where

$$c(i) = \begin{cases} i & \text{, if } i \text{ is an exact power of 2} \\ 1 & \text{, otherwise.} \end{cases}$$

For n operations, there are exactly $\lfloor \lg n \rfloor + 1$ operations with cost i and thus $n - \lfloor \lg n \rfloor - 1$ operations with cost 1. The total cost of n operations:

$$\begin{aligned} \sum_{i=1}^{n} c(i) &= \sum_{k=0}^{\lfloor \lg n \rfloor} 2^k + \sum_{k=1}^{n - \lfloor \lg n \rfloor - 1} 1 \\ &= (2^{\lfloor \lg n \rfloor + 1} - 1) + (n - \lfloor \lg n \rfloor - 1) \\ &\leq 2n + n \\ &= 3n \\ &= O(n). \end{aligned}$$

To find the average cost of each operation, we divide the total complexity of n operations by n. Thus, the amortized running time per operation is $O(1)$.

(b) Exercise 17.2-2 (page 459).

For the accounting method, let us charge an amortized cost of 3 dollars for each operation. Now, we need to show that $\sum_{i=0}^{n} \hat{c}(i) - \sum_{i=0}^{n} c(i) \geq 0$ for all sequences of n operations, where $\hat{c}(i)$ and $c(i)$ denote the amortized cost and the actual cost of the ith operation respectively. There are two cases as follows.

(1) n **is *not* an exact power of 2:**

$$\begin{aligned} \sum_{i=0}^{n} \hat{c}(i) - \sum_{i=0}^{n} c(i) &= 3n - \Big(\sum_{i=0}^{\lfloor \lg n \rfloor} 2^i + \sum_{i=1}^{n - \lfloor \lg n \rfloor - 1} 1 \Big) \\ &= 3n - \Big((2^{\lfloor \lg n \rfloor + 1} - 1) + (n - \lfloor \lg n \rfloor - 1) \Big) \\ &\geq 3n - (2^{\lg(n)+1} - 1) - (n - \lfloor \lg n \rfloor - 1) \\ &= 3n - 2n + 1 - n + \lfloor \lg n \rfloor + 1 \\ &= \lfloor \lg n \rfloor + 2 \\ &\geq 0 \end{aligned}$$

(2) n **is an exact power of 2:**

$$\begin{aligned}
\sum_{i=0}^{n}\hat{c}(i)-\sum_{i=0}^{n}c(i) &= 3n-\Big(\sum_{i=0}^{\lfloor\lg n\rfloor}2^i+\sum_{i=1}^{n-\lfloor\lg n\rfloor-1}1\Big)\\
&= 3n-\Big((2^{\lfloor\lg n\rfloor+1}-1)+(n-\lfloor\lg n\rfloor-1)\Big)\\
&= 3n-\Big((2n-1)+(n-\lg n-1)\Big)\\
&= \lg n+1\\
&\geq 0
\end{aligned}$$

The total credit never becomes negative for all sequences of n operations: $\sum_{i=0}^{n}\hat{c}(i)-\sum_{i=0}^{n}c(i)\geq 0$. Thus, for n operations, the total amortized cost is $O(n)$, which bounds the total actual cost. The amortized cost per operation is $O(1)$.

(c) Exercise 17.3-2 (page 462).

Define the potential function $\Phi(D_i)$:

$$\Phi(D_i)=\begin{cases}2\cdot i-2^{\lfloor\lg i\rfloor+1} & \text{, if } i>0\\ 0 & \text{, if } i=0.\end{cases}$$

Clearly, $\Phi(D_i)\geq 0,\forall i$.

There are 2 cases in the potential analysis as follows.

(1) i **is *not* an exact power of 2:**

$$\begin{aligned}
\hat{c_i} &= c_i+\Phi_i-\Phi_{i-1}\\
&= 1+\Big(2\cdot i-2^{\lfloor\lg i\rfloor+1}\Big)-\Big(2\cdot(i-1)-2^{\lfloor\lg(i-1)\rfloor+1}\Big)\\
&= 3+2^{\lfloor\lg(i-1)\rfloor+1}-2^{\lfloor\lg i\rfloor+1}\\
&\leq 3+2^{\lg(i-1)+1}-2^{\lg(i)+1}\\
&= 3+(2i-2)-2i\\
&= 1
\end{aligned}$$

(2) i **is an exact power of 2:**

$$\begin{aligned}
\hat{c_i} &= c_i+\Phi_i-\Phi_{i-1}\\
&= i+\Big(2\cdot i-2^{\lfloor\lg i\rfloor+1}\Big)-\Big(2\cdot(i-1)-2^{\lfloor\lg(i-1)\rfloor+1}\Big)\\
&= i+2+2^{\lfloor\lg(i-1)\rfloor+1}-2^{\lfloor\lg i\rfloor+1}\\
&= i+2+2^{(\lg(i)-1)+1}-2^{\lg(i)+1}\\
&= i+2+i-2i\\
&= 2
\end{aligned}$$

Therefore, the amortized cost per operation is $O(1)$.

11. (10) Exercise 17.4-3 (page 471).

There are two cases in potential analysis. The ith operation is a deletion: $num_i=num_{i-1}-1$.

(1) $\alpha_i\geq\frac{1}{3}$: no contraction $\Rightarrow size_i=size_{i-1}$.

$$\begin{aligned}
\hat{c_i} &= c_i+\Phi_i-\Phi_{i-1}\\
&= 1+|2\cdot num_i-size_i|-|2\cdot num_{i-1}-size_{i-1}|\\
&= 1+|2\cdot num_i-size_i|-|2\cdot(num_i+1)-size_i|\\
&= 1+|2\cdot num_i-size_i|-|2\cdot num_i-size_i+2|\\
&\leq 1+|-2|\ (\because \text{triangle inequality})\\
&= 3.
\end{aligned}$$

(2) $\alpha_i < \frac{1}{3}$: with contraction $\Rightarrow size_i = \frac{2}{3} size_{i-1}$ and $size_{i-1}/3 = num_{i-1} \Rightarrow size_i = 2 \cdot (num_i + 1)$.

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= (num_i + 1) + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
&= (num_i + 1) + |2 \cdot num_i - size_i| - |2 \cdot (num_i + 1) - \frac{3}{2} size_i| \\
&= (num_i + 1) + |2 \cdot num_i - 2 \cdot (num_i + 1)| - |2 \cdot (num_i + 1) - \frac{3}{2} \cdot 2 \cdot (num_i + 1)| \\
&= (num_i + 1) + |-2| - |-num_i - 1| \\
&= (num_i + 1) + |-2| - |num_i + 1| \\
&\leq (num_i + 1) + |-2| - \Big(|num_i| - |-1|\Big) \ (\because \text{triangle inequality}) \\
&= num_i + 1 + 2 - num_i + 1 \\
&= 4.
\end{aligned}$$

Therefore, by using the strategy, the amortized cost of a TABLE-DELETE is bounded above by a constant.

12. (40) Problem 17-3 (pages 473–474).

(a) We use a divide-and-conquer algorithm. First, we store the inorder traversal of the subtree rooted at x in an array, which needs $O(x.size)$ auxiliary space and takes $\Theta(x.size)$. Second, to construct a 1/2-balanced binary tree, we can select the middle element of the array to be the root and then recurse on the two halves of the array. Since the middle element of an array (a subarray) can be accessed in $O(1)$ time by indexing, we have the recursion $T(n) = T(n/2) + 1$, which has a solution $T(n) = \Theta(n)$. For this problem, $n = x.size$ so rebuilding the tree rooted at x takes $\Theta(x.size)$.

The rebuilt tree is 1/2-balanced. Because the two half sides of a k-element array (subarry) must have at most $\frac{k}{2}$ elements, both the left subtree and the right subtree have at most $\frac{k}{2}$ elements, which satisfies the definition of "1/2-balanced".

(b) Consider the most skewed n-node α-balanced tree. Every node x in the tree (except for the leaf nodes) has a left subtree with $\lfloor \alpha \cdot x.size \rfloor$ and a right subtree with $x.size - \lfloor \alpha \cdot x.size \rfloor - 1$. The height of the tree is at most $1 + \lg_{\frac{1}{\alpha}} n$. Since the height of an n-node α-balanced binary search tree is $1 + \lg_{\frac{1}{\alpha}} n = O(\lg n)$, performing a search in the tree takes $O(\lg n)$ worst-case time.

(c) (1) We know that a sum of nonnegative values is also nonnegative. Since $\Delta(x) \geq 0$ and the potential function of any binary tree T is a sum of $\Delta(x)$ by definition, $\Phi(T)$ is thus nonnegative regardless of the input binary search tree.

(2) Let T be a 1/2-balanced tree rooted at x. Let n_x, n_l and n_r denote $x.size$, $x.left.size$, and $x.right.size$, respectively. Let k denote $\Delta(x)$. Suppose $n_l \geq n_r$ and thus $n_l = n_r + k$ for some $k \geq 0$. Then, to meet the property of a 1/2-balanced tree, we have

$$\begin{aligned}
& \frac{n_l}{n_x} \leq \frac{1}{2} \\
\Rightarrow \quad & \frac{n_l}{n_l + n_r + 1} \leq \frac{1}{2} \\
\Rightarrow \quad & \frac{n_r + k}{(n_r + k) + n_r + 1} \leq \frac{1}{2} \\
\Rightarrow \quad & \frac{n_r + k}{2n_r + k + 1} \leq \frac{1}{2} \\
\Rightarrow \quad & 2n_r + 2k \leq 2n_r + k + 1 \\
\Rightarrow \quad & k \leq 1.
\end{aligned}$$

Similarly, if $n_r \geq n_l$, we can obtain $k \leq 1$ by the similar argument. Therefore, we have that for a 1/2-balanced tree rooted at x, $\Delta(x) \leq 1$. Since $\Delta(x) \leq 1$ for every node in T, which is a root for a 1/2-balanced tree, the potential $\Phi(T)$ is thus 0.

(d) Suppose T_{i-1} will become non-α-balanced if inserting a node into or deleting a node from it. T_i is the rebuilt 1/2-balanced tree and thus $\Phi(T_i) = 0$ by part (c). $\Phi(T_{i-1})$ is at least $c\big(\lfloor \alpha m \rfloor - (m - \lfloor \alpha m \rfloor - 1)\big)$.

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= m + 0 - \Phi(T_{i-1}) \\
&= m - \Phi(T_{i-1}) \\
&= m - c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x) \\
&\leq m - c\Big(\lfloor \alpha m \rfloor - (m - \lfloor \alpha m \rfloor - 1)\Big) \\
&= m - c(2\lfloor \alpha m \rfloor - m + 1) \\
&= m - 2c\lfloor \alpha m \rfloor + cm - c \\
&\leq m - 2c(\alpha m - 1) + cm - c \\
&= \Big(1 - c(2\alpha - 1)\Big)m + c
\end{aligned}$$

We must find a sufficiently large c that bounds $\hat{c}_i$ by a constant. Therefore, we have

$$\begin{aligned}
& 1 - c(2\alpha - 1) \leq 0 \\
\Rightarrow \quad & c \geq \frac{1}{2\alpha - 1}.
\end{aligned}$$

(e) To insert a node into or delete a node from an α-balanced tree with n nodes, we first need to perform a search to find the position where to insert or delete the node, which takes $O(\lg n)$ time (see part (b)). After insertion or deletion, the tree may become non-α-balanced and therefore needs to be rebuilt. Because one node is inserted or deleted, only its ancestors may be unbalanced. Searching the highest unbalanced node takes $O(\lg n)$ time by traversing all the ancestors of the inserted or deleted node. Furthermore, the amortized time of rebuilding the subtree rooted at the highest unbalanced node a subtree is $O(1)$ (see part (d)). Therefore, inserting a node into or deleting a node from an n-node α-balanced tree takes $O(\lg n)$ amortized time.

13. (40) DIY.