CSCE-629 Analysis of Algorithms

Fall 2019

Instructor: Dr. Jianer Chen
Office: HRBB 338C
Phone: 845-4259
Email: `chen@cse.tamu.edu`
Office Hours: T,Th 10:50 am–12:30 pm

Teaching Assistant: Qin Huang
Office: HRBB 309D
Phone: (979) 402-6216
Email: `huangqin@email.tamu.edu`
Office Hours: MWF 3:30 pm–4:30 pm

Solutions to Assignment # 5
(Prepared with TA Qin Huang)

1. A vertex v in an undirected graph G is an *odd cycle transversal* if every cycle of odd length in G contains the vertex v. Develop a linear-time algorithm for the following problem: given a graph G and a vertex v in G, decide if v is an odd cycle transversal.

Solutions. It is easy to prove that a graph is bipartite if and only if it contains no odd cycles. Using DFS, we can decide in linear-time whether a given graph is bipartite (see Question 4).

To solve the problem given in the question on a graph G, we first decide in linear time whether G is bipartite. If G is bipartite, then v is not an odd cycle transversal because G does not contain any odd cycles. Otherwise, remove v from the graph G to obtain a graph G', which can be done in linear time. Now decide in linear-time whether the graph G' is bipartite. If G' is bipartite, then the vertex v is an odd cycle transversal in the graph G; otherwise, v is not because there are odd cycles in the graph G' that does not contain the vertex v. Combining the above steps, we have a linear-time algorithm that solves the problem given in the question.

2. Suppose that each class C_i has an enrollment r_i while each classroom R_j has a capacity c_j. A classroom R_j is "feasible" for a class C_i if $c_j/2 \leq r_i \leq c_j$. Develop an efficient algorithm that, on a set of classes (with enrollments given) and a set of classrooms (with capacities given), make a feasible assignment of the classes to the classrooms such that as many classes as possible can get held starting at 9am on Monday.

Solutions. Let $C = \{C_1, C_2, \ldots, C_m\}$ be the given set of m classes, where each class C_i is associated with an enrollment r_i, and let $R = \{R_1, R_2, \ldots, R_n\}$ be the given set of n classrooms, where each classroom R_j is associated with a capacity c_j.

First, we construct a bipartite graph $G = (U, V, E)$ as follows: let $U = \{u_1, u_2, \ldots, u_m\}$ and $V = \{v_1, v_2, \ldots, v_n\}$, where for each i, u_i corresponds to the class C_i, and for each j, v_j corresponds to the classroom R_j. There is an edge between u_i and v_j if $c_j/2 \leq r_i \leq c_j$. Thus, the graph G has at most mn edges, and constructing the graph G takes time $O(mn)$.

Now use the algorithm discussed in class to construct the maximum matching M in the graph G. Obviously, M is the desired optimal feasible assignment.

The maximum matching algorithm runs in time $O(n_1 m_1)$ on a graph of n_1 vertices and m_1 edges. Since the graph G has $n_1 = |U| + |V| = n + m$ vertices and $m_1 \leq mn$ edges, the above algorithm solving the problem runs in time $O(n_1 m_1) = O(n^2 m + nm^2)$.

3. Suppose that in addition to edge capacities, a flow network also has *vertex capacities*, i.e., each vertex v has a limit $c(v)$ on how much flow can pass through v. Show how to transform a flow network $G = (V, E)$ with vertex capacities into a flow network $G' = (V', E')$ without vertex capacities, such that a maximum flow in G' has the same value as a maximum flow in G.

Solutions. Let $G = (V, E)$ be a flow network with vertex capacities (in addition to edge capacities). Let s and t be the source and sink in the flow network G, respectively. We construct an "equivalent" flow network $G' = (V', E')$ without vertex capacities such that a maximum flow in G' has the same value as that of a maximum flow in G.

The flow network G' is constructed from the flow network G, as follows. For every vertex v with vertex capacity $c(v)$ in G, "split" v into two vertices v_1 and v_2, and add an edge $[v_1, v_2]$ with edge capacity $c'(v_1, v_2) = c(v)$. For every edge $[u, v]$ in G, where the vertex u is split into $[u_1, u_2]$ and v is split into $[v_1, v_2]$, replace $[u, v]$ with the edge $[u_2, v_1]$ with capacity $c'(u_2, v_1) = c(u, v)$. This gives the flow network $G' = (V', E')$ without vertex capacities. The source of G' is s_1 while the sink of G' is t_2. It is easy to see that $|E'| = |V| + |E|$ and $|V'| = 2|V|$.

Let f be a flow from s to t in the flow network G. We create a flow f' in the flow network G' as follows: for each edge $[u, v]$ in G, $f'(u_2, v_1) = f(u, v)$, and for each vertex $v \neq s$ in G, let $f'(v_1, v_2) = \sum_{[u,v]\in E} f(u, v)$. Moreover, let $f'(s_1, s_2) = \sum_{[s,u]\in E} f(s, u)$. It is easy to verify that f' is a valid flow in G' and its value is equal to that of the flow f in G.

Conversely, given a flow f' in the flow network G', we can construct a flow f in the flow network G as follows: for each edge $[u, v]$ in G, let $f(u, v) = f'(u_2, v_1)$. Since f' satisfies the capacity constraint, in particular the capacity constraints on the edges $[v_1, v_2]$ where v_1 and v_2 correspond to a vertex v in G, f satisfies both edge capacity constraints and the vertex capacity constraints in G. Thus, f is a valid flow in the flow network G with the same value as f'.

Therefore, there is a one-to-one correspondence between flows in G and flows in G' where the corresponding flows have the same value. Thus, the maximum flow value in G equals the maximum flow value in G'.

4. Develop a linear-time algorithm that tests if a given graph is bipartite.

Solutions. See the algorithm given in the next page.

Let G be the input graph. We use an array $side[1..n]$ to label the vertices of G so that if $side[v] = L$ then the vertex v is on the left side and if $side[v] = R$, then the vertex v is on the right side. If we can successfully place the vertices in the two sides such that there is no edge connecting two vertices in the same side, then the graph G is bipartite. Otherwise, the graph G is not bipartite. The labeling process can be implemented using depth-first search, as given in the algorithm in the next page.

To see the correctness of the algorithm, first note that the depth-first search process examines every edge in the graph. Thus, step 5 of Algorithm 2 checks for each edge if its both ends are in the same side. As a result, if the algorithm does not stop at step 5 in Algorithm 2, then no edge in the graph has it both ends in the same side, so the graph is bipartite, and the algorithm will return correctly at step 3 in Algorithm 1. On the other hand, if the algorithm stops at step 5 in Algorithm 2, then we get an edge $[v, w]$ with $side[w] = sied[v]$ and $color[w] \neq white$. The condition $color[w] \neq white$ indicates that the vertex w is an ancestor of the vertex v in the DFS tree. Moreover, the condition $side[w] = sied[v]$ shows that w cannot be the parent of v (because a vertex always assigns a different side value to its children). Since the vertices on the path in the DFS from vertex w to vertex v have been assigned side values alternatively, the condition $side[w] = sied[v]$ shows that the graph G has an odd cycle, which is formed by the path in the

DFS tree from w to v plus the edge $[v, w]$. As a consequence, the graph G has an odd cycle so is not bipartite, and the algorithm reports correctly at step 5 in Algorithm 2.

Since the algorith is basically the depth-first search, it takes time $O(n + m)$.

Algorithm 1 Main Algorithm for Problem 4

Input: $G = (V, E)$

1: **for** (each vertex v in V) **do** { $\mathit{color}[v] = \mathit{white}$; $\mathit{side}[v] =$ ‘ ’ };

2: **for** (each vertex u in V) **do if** ($\mathit{color}[v] == \mathit{white}$) **then** DFS$(v, L)$;

3: Return(’bipartite’).

Algorithm 2 Function DFS for Problem 4

DFS(v, c) \\ c is the side value of vertex v

1: $\mathit{color}[v] = \mathit{gray}$; $\mathit{side}[v] = c$;

2: **if** ($c == L$) **then** $c' = R$ **else** $c' = L$;

3: **for** (each edge $[v, w]$ in E) **do**

4:     **if** ($\mathit{color}[w] == \mathit{white}$) **then** DFS$(w, c')$

5:     **else if** ($\mathit{side}[w] == \mathit{side}[v]$) **then** STOP(’not bipartite”);

6: $\mathit{color}[v] = \mathit{black}$;
