Problem Set 5 Solutions

Problem 1.  [Edge-cover]

Solution

(a) Let us consider a matching \( M \): \( M \) covers \( 2|M| \) vertices. In order to construct an edge cover it is sufficient to add to \( M \) at most \( |V| - 2|M| \) other edges, so as to cover the remaining \( |V| - 2|M| \) edges. The resulting edge cover will have at most

\[
(|V| - 2|M|) + |M| = |V| - |M|
\]

edges.

(b) Starting from an edge-cover \( E' \), choose a subset \( M \subseteq E' \) in the following way: fix an order, say the lexicographic order, between the edges in \( E' \); for each edge \((u, v) \in E'\), put \((u, v) \in M\) iff there is no edge in \( E' \) that comes before \((u, v)\) in the lexicographic order and that shares a vertex with \((u, v)\).

This rule ensures that \( M \) is a matching. Suppose by contradiction that there were two edges in \( M \) sharing the same vertex, say \((u, v)\) and \((u, w)\), where \((u, v)\) comes before \((u, w)\) in lexicographic order. Then we would have violated our selection rule when selecting \((u, w)\).

The rule also ensure that for every edge \((u, v)\) in \( E' - M \) there is an edge in \( M \) that shares an edge with \((u, v)\).

The edges in \( M \) cover \( 2M \) vertices, and each edge in \( E' - M \) can cover (due to the above observation) at most one more vertex. Since all vertices are covered, we must have

\[
n \leq 2|M| + (|E'| - |M|)
\]

which is equivalent to \( M \geq n - |E'| \).

(c) Let \( M^{*} \) be a maximum matching. Following the solution to part (a), we note that we can add at most \( n - 2|M^{*}| \) edges to \( M^{*} \) and so construct a set of edges \( E^{*} \) that is an edge cover and such that \( E^{*} \leq n - |M^{*}| \). From the solution to part (b), it follows that \( E^{*} \) is the smallest possible edge cover, because if there were a smaller edge cover \( E' \), then \( |E'| \leq n - |M^{*}| - 1 \); but then, using part (b), there would be a matching \( M' \) such that \( |M'| \geq n - |E'| \geq |M^{*}| + 1 \) and this would contradict the optimality of \( M^{*} \).
Problem 2.  \hspace{0.2cm} [A Scheduling Problem]

Solution

The dynamic programming solution works as follows. We consider a matrix $D[\cdot, \cdot]$, having dimension $(n + 1) \times n(M + 1)/2$ having values in \{0, \ldots, n(M + 1)/2\}.

The intended meaning of $D[i, a]$, for $i \in \{0, \ldots, n\}$ and $a \in \{0, \ldots, n(M + 1)/2\}$ is that $D[i, a]$ is the minimum, over all possible ways of scheduling processes $t_1, \ldots, t_i$ between $A$ and $B$, of the total time of processes scheduled on $B$, subject to the constraint that the processes scheduled on $A$ take total time $\leq a$.

The recursive rule is

\[
D[0, 0] = 0 \\
D[i, a] = \min\{D[i - 1, a + b_i, D[i - 1, a - a_i]\}
\]

Without loss of generality assume that $D[i, a] = \infty$ if $a < 0$. Note that to construct the matrix the time required is $O(n^2 M)$.

However we are not done yet. The above matrix only tells us the cost of an optimum solution. In order to actually construct the optimal partition we need one more matrix. We define a Boolean matrix $C[\cdot, \cdot]$ having the same size of $D[\cdot, \cdot]$ and such that

\[
C[i, a] = 1 \text{ iff } D[i, a] = D[i - 1, a] + b_i
\]

that is, $C[i, a] = 1$ when it is possible to partition the first $i$ tasks among $A$ and $B$ in such a way that the total time on $A$ is at most $a$, the total time on $B$ is at most $D[i, a]$, and the process $t_i$ goes on machine $B$. Similarly, $C[i, a] = 0$ when it is possible to partition the first $i$ tasks among $A$ and $B$ in such a way that the total time on $A$ is at most $a$, the total time on $B$ is at most $D[i, a]$, and the process $t_i$ goes on machine $A$.

Note that $C$ can be constructed together with $D$, so the time needed to construct $C$ and $D$ remains $O(n^2 M)$.

Now, once the two matrices are constructed we can construct the optimal solution as follows. The total running time in the optimal solution is indeed given by

\[
\min\{t \leq (n(M + 1)/2) : D[n, t] \leq t\}
\]

Once we find such a minimum cost $t$, we can use the matrix $C$ to construct the actual partition. We first look at $C[n, t]$ and, depending on its value we consider the following cases:

1) if $C[n, t] = 1$ then the $n^{th}$ element of the optimal partition has to be executed on machine $B$. Then we have to schedule the remaining $n - 1$ elements so that the total running time on $A$ is $\leq t$ and the total running time on $B$ is $D[n - 1, t - b_n]$

2) if $C[n, t] = 0$ then the $n^{th}$ element of the optimal partition has to be executed on machine $A$. Then we have to schedule the remaining $n - 1$ elements so that the total running time on $A$ is $\leq t - a_n$ and the total running time on $B$ is $D[n - 1, t]$

At the next step we will look at either $B[n - 1, t]$ or $B[n - 1, t - a_n] = 1$ and so on.

In $n$ steps we will have reconstructed the optimal solution.
Problem 3. [Hamiltonian paths]

Solution

The basic idea of the algorithm is to use topological sort and then dynamic programming to determine the longest possible path that starts from the given vertex \( s \). If such a path has length \(|V| - 1\) then there is an Hamiltonian path, otherwise no Hamiltonian path can be found.

Note that since the given graph is directed and acyclic is possible to use the topological sort algorithm from CLR to produce a linear ordering of the vertices in \( V \).

Assume that \( G = (V, E) \) and that the vertices of \( V \) are \( u_1, \ldots, u_n \), already in topological order. Then we define a matrix \( D[] \) with \( n = |V| \) entries, where \( D[u] \) is the length of the longest path that starts at \( u \). We also define an auxiliary matrix \( P[] \), also of size \( n \), where \( P[u] \) is the vertex that follows \( u \) in the longest path that starts at \( u \).

The recursive rules for \( D[] \) are

\[
D[u_n] = 1 \\
D[u_i] = \max_{j \geq i, (u_i, u_j) \in E} \{ D[u_j] + 1 \}
\]

where we set \( D[u_i] = 1 \) if there is no out-going edge from \( u_i \).

The recursive rules for \( P[] \) are

\[
P[u_n] = \emptyset \\
P[u_i] = u_j \text{ where } u_j \text{ maximizes } \max_{j \geq i, (u_i, u_j) \in E} \{ D[u_j] + 1 \}
\]

where we set \( P[u_i] = \emptyset \) if there is no out-going edge from \( u_i \).

The matrices can be computed in \( O(|V| + |E|) \) time. The length of the longest path starting at a vertex \( u \) is \( D[u] \), and the actual path is \( u, P[u], P[P[u]], \ldots \) where we reconstruct the path by sequential applications of \( P[] \) until we find the \( \emptyset \) value.