**COL 351**

# TUTORIAL SHEET 12

1. [**KT-Chapter8**] Consider a set $A = \{a_1, ..., a_n\}$ and a collection $B_1, B_2, ..., B_m$ of subsets of $A$. (That is, $B_i \subseteq A$ for each $i$.) We say that a set $H \subseteq A$ is a hitting set for the collection $B_1, B_2, ..., B_m$ if $H$ contains at least one element from each $B_i$ ? that is, if $H \cap B_i$ is not empty for each $i$. We now define the Hitting Set problem as follows. We are given a set $A = \{a_1, ..., a_n\}$, a collection $B_1, B_2, ..., B_m$ of subsets of $A$, and a number $k$. We are asked: is there a hitting set $H \subseteq A$ for $B_1, B_2, ..., B_m$ so that the size of $H$ is at most $k$? Prove that Hitting Set is NP-complete.

   **Solution:** It is easy to show that Hitting Set is in NP. A solution just needs to exhibit the set $H$ – one can easily verify in polynomial time whether $H$ is of size $k$ and intersects each of the sets $B_1, \ldots, B_m$.

   We reduce from vertex cover. Consider an instance of the vertex cover problem – graph $G = (V, E)$ and a positive integer $k$. We map it to an instance of the hitting set problem as follows. The set $A$ is the set of vertices $V$. For every edge $e \in E$, we have a set $S_e$ consisting of the two end-points of $e$. It is easy to see that a set of vertices $S$ is a vertex cover of $G$ iff the corresponding elements form a hitting set in the hitting set instance.

2. [**KT-Chapter9**] Consider the Hitting Set Problem described above. Furthermore, suppose that each set $B_i$ has at most $c$ elements, for a constant $c$. Give an algorithm that solves this problem with a running time of the form $O(f(c, k)p(n, m))$, where $f$ is an arbitrary function of $c$ and $k$, and $p$ is a polynomial on $n$ and $m$.

   **Solution:** The algorithm is similar to the one done in class for vertex cover. It is a divide -and-conquer algorithm. Let $\mathcal{A}$ be the algorithm which given a hitting set instance, either outputs a hitting set of size $k$ or outputs NO. In the base case, when $k = 1$, the algorithm can just check if there is any element which belongs to all the sets. So assume $k > 1$ and $\mathcal{I}$ be such an instance of hitting set. Let $A$ denote the set of elements in $\mathcal{I}$, and $B_1, \ldots, B_m$ be the subsets. The algorithm $\mathcal{A}$ picks a subset $B_1$ (say) – let the elements in $B_1$ be $e_1, \ldots, e_c$ (recall that each set has at most $c$ elements). Any solution must pick at least one element from $B_1$. For an element $e$, let $\mathcal{I} - e$ denote the instance where we remove all sets which contain $e$ from $\mathcal{I}$ (in other words, if we pick $e$, then $\mathcal{I} - e$ denotes the remaining sets which need to be hit). The algorithm $\mathcal{A}$ is as follows: for each element $e_i \in B_1$, recursively call $\mathcal{A}$ on $\mathcal{I} - e_i$ with size $k - 1$. Note that $\mathcal{A}$ makes $c$ recursive calls. If any of these recursive calls returns a set $S$ (say the one with $\mathcal{I} - e_i$ does), then $\mathcal{A}$ returns $S \cup \{e_i\}$. If all the recursive calls return NO, $\mathcal{A}$ returns NO.

   The running time: total number of recursive calls is $c^k$. In each recursive call, it spends $O(n + m)$ time (in creating new instances, etc.). Thus running time is $O(c^k(n + m))$.

3. **[KT-Chapter9]** Give an algorithm for the Hamiltonian path problem in a directed graph whose running time is $O(2^n p(n))$, where $p(n)$ is a polynomial in $n$ (here, $n$ denotes the number of vertices in the graph).

   **Solution:** This is done by dynamic programming. For every subset $S$ of vertices, let $G[S]$ denote the subgraph of $G$ induced by $S$, i.e., those edges of $G$ which have both end-points in $S$. For every set of vertices $S$ and vertices $v \in S$, we have a table entry $T[S, v]$ which is supposed to store a Hamiltonian path in $G[S]$ which starts with $v$ (if there exists such a path, otherwise it stores NO). Now the recursive definition of T. Let $N_v$ be the set of neighbors of $v$ in the set $S$. Then $T[S, v]$ is false if $T[S - v, w]$ is false for all $w \in N_v$. Otherwise suppose $T[S - v, w]$ is a path $P$. Then $T[S, v]$ stores the path $v$ followed by $P$. It is easy to check that the time taken by the algorithm is $O^(2^n n^2)$.

4. **[KT-Chapter10]** Suppose we use the following heuristic for the Interval Selection Problem – given a set of intervals, we repeatedly pick the shortest interval $I$, delete all the other intervals $I?$ that intersect $I$, and iterate. Show that this is not an optimal algorithm. However, it turns out to have the following interesting approximation guarantee. If $s^\star$ is the maximum size of a set of non-overlapping intervals, and $s$ is the size of the set produced by the Shortest-First algorithm, then $s \geq s^\star/2$. (That is, Shortest-First is a 2-approximation.) Prove this fact.

   **Solution:** Let $S^\star$ denote the optimal set of intervals. As our algorithm picks intervals in a set $S$, we remove the intervals in $S^\star$ which overlap with $S$. Thus $S^\star$ denotes the optimal set of intervals which do not overlap with any of the intervals the greedy algorithm has picked so far. Now when the greedy algorithm picks an interval, it will overlap at most 2 intervals from $S^\star$ – if it overlaps 3 intervals, then the middle one is smaller than the interval just picked, a contradiction. Thus, in each iteration of greedy, we will remove at most 2 intervals from $S^\star$. Thus, the greedy algorithm will pick at least $s^\star/2$ intervals.

5. Consider the greedy algorithm for vertex cover. Show that there exist bipartite graphs on $n$ vertices for which the size of the vertex cover produced by the greedy algorithm can be at least $\Omega(\log_2 n)$ times the size of the optimal vertex cover.

   **Solution:** Let $n$ be a power of 2. We will construct a bipartite graph: $L$ denotes the set of vertices on the left and $R$ denote the set of vertices on right. $L$ will have $n$ vertices and $R$ will have $n \log_2 n/2$ vertices. The optimal solution should pick all of $L$, but greedy will end up picking all of $R$. The graph is as follows: we divide the vertices in $R$ into blocks of size $n/2$ each (so there are $\log_2 n$ blocks) – $B_1, B_2, \ldots, B_k$, where $k = \log_2 n$. Each of the vertices in $B_1$ is joined by an edge to every vertex in $L$. For $B_2$, we divide the vertices in it into 2 halves (each of size $n/4$) – call these $B_2'$ and $B_2''$. Similarly, we divide all the vertices in $L$ into two halves $L'$ and $L''$ – we join every vertex in $B_2'$ to $L'$ and every vertex in $B_2''$ to every vertex in $L''$. For $B_3$, we divide into 4 parts, and so on.