Due 23:59 Dec 6 (Tuesday). There are 100 points in this assignment.
Submit your answers (**must be typed**) in pdf file to CourSys
`https://coursys.sfu.ca/2022fa-cmpt-705-x1/`. Submissions received after 23:59 of Dec 6 will get penalty of reducing points: 20 and 50 points deductions for submissions received at $[00:00, 00:10]$ and $(00:10, 00:30]$ of Dec 6, respectively; no points will be given to submissions after 00 : 30 of Dec 6.

1. (Chapter 10 Problem 1 of the text book) 20 points

   The Hitting Set problem is defined as follows: Let $A = \{a_1, .., a_n\}$ and $B_1, .., B_m$ be a collection of subsets of A, a subset H of A is a hitting set for $B_1, .., B_m$ is $H \cap B_i \neq \emptyset$ for every $1 \leq i \leq m$. The Hitting Set problem is that given A and $B_1, .., B_m$, and integer $k > 0$, whether there is a hitting set of size k or not. The problem is NP-complete. Assume that $|B_i| \leq c$ for some constant $c > 0$. Give an algorithm that solves the problem with a running time of $O(f(c, k) \cdot p(n, m))$, where $p(n, m)$ is a polynomial in n and m, and $f(c, k)$ is an arbitrary function depending only on c and k, not on n or m.

   Answer We can prove that Hitting Set problem is NP complete by reducing another problem, Vertex Cover, to be solvable within polynomial time. We pose the problem as follows: a is an element in A ($a \in A$), which can be deleted from the set A and then deleting all sets from the other set B_i which contain a. This can easily be proven to be under polynomial time. We see that if we have a set B_i which is any of the sets in the Hitting sets ($B_i = \{x_1, x_2, ..., x_c\} \subseteq A$). With this given, we can safely say that we have at least one of these belong to the Hitting set. With this stated, we can say that B_i can be hit by k length H set if and only if for some value of i (iterator/subscript $i = 1, 2, ...c$) the instance reduced by x_i has a Hitting set H of length $k - 1$.

   We can model our algorithm to pick any set $B_i = \{x_1, x_2, ...x_i, ..., x_c\}$ and for each x_i we recursively check if the instance reduced by x_i has a Hitting set of $k - 1$ elements. If any of these recursive calls result in $True$, we return $True$. On worst case we will run the recursion a total of c times and each takes running complexity of $O(c)$ which makes it $O(c^{k+1})$, We need to do this a total of k times as we are checking if the set H is of size k or not. So the total complexity ends up being $O(k \times c^{k+1})$. This is the contribution of a function $f(c, k)$. The size of set A is n and there are a total of m collection of subsets of A, namely Bs. The function $p(n, m)$ is also a polynomial function. So to conclude, the running time is within the asked range.

   ---

2. (Chapter 10 Problem 3 of the text book) 15 points

   Give a digraph G of $V(G) = \{v_1, .., v_n\}$, we want to decide if G has a Hamiltonian path from v_1 to v_n. Give an $O(2^n p(n))$ time algorithm to solve the Hamiltonian path problem in G, where $p(n)$ is a polynomial in n.

Answer

In a Hamiltonian Path we have to remember the walk of the tree, but not the order of the walk. For this we have to construct a table (T) with indices $2^n \times n$. If we have a path in a subset of the graph G, lets call it $G[S]$. Say, we have two vertices $v_i \& v_j$ which are in this subset. The existence of paths in the set S can be denote by $T(S, v_i, v_j)$, essentially we are looking for the answer to $T(V, 1, n)$ (Path to all vertices). To determine if there is a path between any vertices $v_i \& v_j$, we have to look at all the nodes which have an edge (v_i, v_k) to another vertex v_k which then looks for a path to v_j which is only $True$ if there exists a path between v_k and v_j. This will take a running time of $O(n)$. To fill out the table T, we look for the smallest sets which have a Hamiltonian Path and then work our way up to bigger and bigger sets, till we reach to the answer to $T(V, 1, n)$. As we have n vertices, we will have to check for a path between v_i and v_j, n times. In the end we have to pass through the whole table T, which already takes $2^n \times n$ steps. The total running time for this algorithm is $O(2^n \times n^3)$, which is what was required.

We can pose this problem differently, where we find path which flows from vertices not more than once. The algorithm runs as follows:

Initialize value of T(v1, v) to 1 and the rest of them to be 0.
for i in 2 to n:
    for all S (subsets of V) such that $len(S) = i - 1$ for all $v_j \in V$
    if $T(S, v_j) = 1$
        for all $v_k \notin S$ such that (v_j, v_k) are edges
            $T(S \cup v_k, v_j) = 1$
if $T(n, v_i) = 1$ for $v_i \in V$
    return True
return False

---

3. (Chapter 10 Problem 5 of the text book) 15 points

   A dominating set D of a graph G is a subset of $V(G)$ such that for every node u of G, either $u \in D$ or u is adjacent to a node $v \in D$. The minimum dominating set problem in G is to find a minimum dominating set D of G (the problem is NP-complete). Give a polynomial time algorithm for the minimum dominating set problem in a tree.

Answer In the question 5 (chapter 10) of the book, we are given Minimum-Cost Dominating Set problem, where the graph G is a tree. This problem can be dealt with the use of Dynamic Programming, where at each step we decide if a particular node needs to be included in the set. We can start from any node, u, inside the tree and have two sub trees, say $T_{u1}\&T_{u2}$. We now have two sub problems given this preface which are: (i) the cost of independent set of T_u given that node u is a part of the independent set: $M_{included}$, (ii) the cost of independent set of T_u where u is not a part of the independent set $M_{excluded}$. We can take the minimum value out of these two to know which value is optimal. This calculated value is the maximum cost of independent set of T_u that does not dominate u. So this can be written in algorithmic form as follows:

def minWeightDomSet(Tree T):
    Select any node
    for all nodes u in T:
        if u is $T[0]$ (root):
            $M_{included}[u]$ = u.cost
            $M_{excluded}[u] = \infty$
            $M_{min}[u] = 0$
        else:
            $M_{included}[u]$ = u.cost + $[sum(M_{min}[v]) for v in u.children]$
            $M_{excluded}[u] = [min(M_{included}[v], M_{excluded}[v])$ for v in u.children] +
                $[min(M_{included}[w], M_{excluded}[w])$ for w in v.children]
            $M_{min}[u] = [sum(min(M_{included}[v], M_{excluded}[v]))$ for v in u.children]

The running time of this algorithm is $O(n)$ as we have 3 problems per node, we have a total of $3n$ problems for all the nodes in the tree.

---

4. (15 points)

Below is an algorithm to create a random permutation of n elements in an array $A[1..n]$.
RANDOM-PERMUTE(A, n)
for $i = 1$ **to** n
select j uniformly at random from $\{i, i+1, .., n\}$ and exchange $A[i]$ with $A[j]$;

Prove that the algorithm creates every permutation of $A[1..n]$ with probability$1/n!$.

Answer

The function gives out a random permutation which is a reordering of elements from 1, ..., n. There are a total of $n!$ permutations of n elements. If we look at a special sorting algorithm which has all elements referring to a permutation, $A[i] = p[i]$. We can denote a sorted form of this array to be A', where the contents are sorted so $A'[i] = i$. If the random permutation algorithm is to sort the array in increasing order, it will pick the smallest number from $A[i..n]$. This would suggest that the algorithm doesn't look at the contents of array. We can see that the combinations, given any value of j, are equal to $n \times (n-1) \times ... \times 1 = n!$. There are $n!$ permutations of n elements, and the algorithm can do $n!$ distinct permutations (for values of j). So now for all permutations we see that we get $1/n!$ choices, i.e., $1/n \times 1/(n-1) \times ... \times 1/1 = 1/n!$. This means that each permutation has a probability of $1/n!$ using this algorithm, which is what was required.

---

5. (20 points)

Given a graph G, we consider the following problem: color the nodes of G using k colors; an edge $e = \{u, v\}$ is called satisfied if u and v are colored by different colors; and we want to color the nodes to satisfy as many edges as possible.

(a) (14 points) A randomized algorithm RA for the maximization problem is as follows: for every node v of G, select one of the k colors independently with probability $1/k$ and assign the color to v. Prove that the expected number of edges satisfied by RA is $(1 - 1/k)m$, where m is the number of edges in G. Assume $(1 - 1/k)m$ is an integer, prove that the probability that RA satisfies at least $(1 - 1/k)m$ edges is at least $1/m$.

Answer

For this part, as we are given that we can have a possible of k colors for every node. For each edge we have two scenarios: (i) where the edge has a valid combination of colors, (ii) where the edge does not have a valid combination of colors. This is a maximization problem hence we have an upper bound of $c^* \leq m$. For an edge there are a possible of k^2 ways we can color the two nodes, and there are a total of k ways when the coloring is invalid (if $k = 4$, there are a total of 16 color combinations and 4 combinations are invalid; same color). The expected value where the coloring for two nodes is not satisfied are $\frac{k}{k^2}$ or $\frac{1}{k}$. Since we are dealing with probabilities here, we can:

$$P[edge\ e\ is\ not\ satisfied] = \frac{k}{k^2} = \frac{1}{k}$$

Since the probability of both cases is equal to 1. The probability of getting the valid case will be $1 - P[edge\ e\ is\ not\ satisfied]$

$$P[edge\ e\ is\ satisfied] = 1 - \frac{1}{k} = E[X]$$

Since we have a total of m edges, this formula becomes:

$$P[All\ edges\ are\ satisfied] = (1 - \frac{1}{k})m = E[Y]$$

Now we have linearity of expectations in play, we get:

$$P[Satisfying\ 1 - \frac{1}{k}\ edges] = \frac{1 - \frac{1}{k}}{(1 - \frac{1}{k})m} = \frac{1}{m}$$

(b) (6 points) Assume $(1 - 1/k)m$ is an integer. Give a Monte Carlo algorithm which satisfies at least $(1 - 1/k)m$ edges with probability at least $1 - 1/m$. Give a Las Vegas algorithm which satisfies at least $(1 - 1/k)m$ edges.

6. (Chapter 13 Problem 11 of the text book) 15 points

   There are k machines and k jobs. Each job is assigned to one of the k machines independently at random (with each machine equally likely).

   (a) Let $N(k)$ be the expected number of machines that do not receive any jobs, so $N(k)/k$ is the expected fraction of machines with no job. What is the limit $\lim_{k\to\infty} N(k)/k$? Give a proof of your answer.

Answer

As the jobs are assigned based on a random uniform distribution. If we have a machine, which does not have any job at all. The probability of job not getting assigned to that machine is $1 - 1/k$. if we have a total of k jobs, and the expected number of machines to have no jobs would be $\Pi^k 1 - 1/k = (1 - 1/k)^k$. So, the expected number of machines not having any jobs is $N(k) = k(1 - 1/k)^k$. The expected fraction of machines not having any jobs is:

$$N(k)/k = (1 - 1/k)^k$$

The fraction $N(k)/k$ goes to $1/e$ as the limit goes to infinity. So the limit of this $N(k)/k$ to infinity would be k/e.

(b) Suppose that machines are not able to queue up excess jobs, so if the random assignment of jobs to machines sends more than one job to a machine M, then M will do the first of the jobs it receives and reject the rest. Let $R(k)$ be the expected number of rejected jobs; so $R(k)/k$ is the expected fraction of rejected jobs. What is $\lim_{k\to\infty} R(k)/k$? Give a proof of your answer.

Answer

In this part, we have the number of rejected jobs equal to $N(J_{rejected})$. We have a total of k jobs, so the number of rejected jobs is $N(J_{rejected}) = k - N(J_{accepted})$. Since all of the machines do exactly one job, so the number of jobs accepted is total number of jobs take away machines with no jobs (as number of machines and number of jobs are the same k), i.e., $N(J_{rejected}) = k - (k - N(J_{nojob})) = N(J_{0jobs})$. So, as the expected fraction of rejected jobs goes to $1/e$ the limit goes to infinity. So the limit of $R(k)/k$ would be k/e.

(c) Now assume that machines have slightly larger buffers; each machine M will do the first two jobs it receives, and reject any additional jobs. Let $R_2(k)$ denote the expected number of rejected jobs under this rule. What is $\lim_{k\to\infty} R_2(k)/k$? Give a proof of your answer.

Answer

We have the number of machines with no jobs to be k/e. We then get the number of machines with exactly one job and then look where the second job goes. The probability of one job getting assigned to the machine is $(1-1/k)^{k-1}$ (This comes from probability of job being assigned to a machine is $1/k$ and the probability of rest of the jobs not being assigned to the machine is $(1-1/k)^{k-1}$, and there are a total of k jobs which gets canceled out by $1/k$). Now the rest of the machines accepting two jobs is $k - \frac{2k}{e}$, This is what the fraction of $R_2(k)/k$ goes to as the limit goes to infinity.
