Answer the questions below. For the questions involving both programming and analysis, please submit your code through Udacity and your analysis through T-square.

1. In a list of distinct numbers $a_1, \ldots a_n$, we say that the elements a_i and a_j are *inverted* if $i < j$ but $a_i > a_j$. Suppose that all orderings of $a_1, \ldots a_n$ are equally probable under some probability distribution (In other words we shuffled a set of numbers perfectly to obtain the order $a_1 \ldots a_n$.) What is the expected number of inversions?

Solution

Define X_{ij} be the random variable that is 1 if a_i and a_j are inverted and 0 otherwise. Then $E[X_{ij}] = 1/2$ because for every pair of positions for a_i and a_j it is equally probable that the two positions are reversed. The total number of inversions is $\sum_{i<j} X_{ij}$, and by the linearity of expectations, we have that $E[\sum_{i<j} X_{ij}] = \sum_{i<j} E[X_{ij}] = \frac{1}{2} \cdot \frac{n(n-1)}{2} = \frac{n(n-1)}{4}$.

2. Consider the following greedy algorithm for finding a matching.
   - Initialize M to an empty set of edges
   - While there is an edge where *both* vertices are unmatched by M
     - Add the above edge to M
   - Return M

   a. What is the running time of this algorithm?
   b. Give an example of a graph where the algorithm does not find the maximum matching.
   c. Show that the matching found by the algorithm always has at least half as many edges as a maximum matching.
   d. (Bonus) Now consider a the analogous algorithm for finding a maximum weight matching (assume all weights positive) in an edge-weighted graph: greedily add the heaviest edge possible to the current matching; stop when no further edge can be added. Show that this algorithm finds a matching whose weight is at least half the optimum.

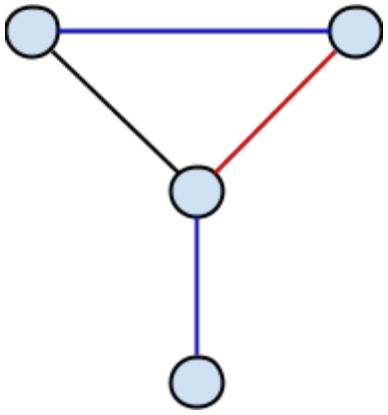
      For notation, let $w(e)$ denote the weight of the edge e, let $w(S) = \sum_{e \in S} w(e)$ for any subset of the edges S, let M^* be an optimum matching and let M be one returned by the algorithm. You are asked to show that $2w(M) \geq w(M^*)$.

Solution

a. With the right data structures, the algorithm can be implemented in $O(E)$ time. This can be achieved by in an adjacency list representation and by having the edge type

store the two relevant list pointers and having the lists themselves contain references to the edge.

b. Classic examples include an odd-length path or a “flower”-like structure pictured below. The blue edges below represent a *maximum* matching, the red a *maximal* one that might be found by the algorithm.



c. For a matching M, we define $E(M)$ to be the set of edges in the matching and $V(M)$ to be the set of vertices in the matching. Let M^* be an optimum matching and let M be one found by the algorithm. Note that by construction M is maximal, since no more edges can be included in M without subtracting an edge from M first.

For every edge $(u,v) \in E(M^*)$, either u or v must be in $V(M)$. Otherwise, M would no longer be maximal. Since neither u nor v can be in another edge in $E(M)$, we have that $|E(M^*)| \leq |V(M)| = 2|E(M)|$.

Alternative 1

Let M and M^* denote the set of matched edges in a maximal and a maximum matching respectively. Then,

$$2|M| = \sum_{e \in M} 2 \geq \sum_{e \in M} \sum_{e' \in M^*} |e \cap e'| = \sum_{e' \in M^*} \sum_{e \in M} |e \cap e'| \geq \sum_{e' \in M^*} 1 = |M^*|.$$

where the second inequality follows from the fact that M is maximal.

Alternative 2

Let M and M^* be defined as above. Then $M^* \oplus M$ will have connected components of alternating paths and cycles. Because, both M^* and M are maximal, however, there will be no isolated edges. Hence, the ratio between edges in M^* and those in M is at most 2:1 in $M^* \oplus M$, the result follows.

d. Let M^* be an optimum matching and let M be one produced by a greedy algorithm. Note that because every edge has positive weight, M is maximal. As in part c, every edge $e \in M^*$ must share a vertex with an edge in M. Otherwise, e can be added to M to form a larger matching making M non-maximal. Therefore, for every $e \in M^* - M$, there must be a neighboring edge $g(e) \in M - M^*$ such that $w(e) \leq w(g(e))$. (There may be two such edges; choose one arbitrarily in constructing the function g). Summing the weights of all the edges in M^* gives $w(M^*) \leq \sum_{e \in M^*} w(g(e))$. Each edge of M appears at most twice in the second term, so $w(M^*) \leq 2w(M)$ as desired.

[This also represents another variation of the proof for part c.]

3. Consider the following factor 2 approximation algorithm for the minimum cardinality vertex cover problem. Find a depth first search tree in the given graph, G, and output the set, say S, of all the non-leaf vertices of this tree. Show that S is indeed a vertex cover for G and $|S| \leq 2 \cdot OPT$.

Hint: Use the tree to construct a matching that includes all the vertices in S.

Solution:

First we argue that S is a vertex cover. Suppose not. Then there is a edge (u, v) where $u \notin S$ and $v \notin S$. If u is discovered first by the depth-first search, then v is a child of u in the depth-first tree, implying that u is not a leaf and therefore $u \in S$. The same holds if v is discovered first. In either case, we arrive at a contradiction. Therefore, S is a vertex cover.

Let M be the matching constructed in the manner of problem 2 where edges are considered in the order they are traversed in the depth-first search that produced S. In this way, a vertex in S is always matched with one of its children if it has not been already matched with its parent. Thus, S is a subset of the vertices matched in M, so $|S| \leq 2|M|$.

The minimum vertex cover must include at least one vertex from each edge in M, and since these edges in M are vertex disjoint we have that $|M| \leq OPT$. Combining the inequalities gives the desired result $|S| \leq 2|M| \leq 2 \cdot OPT$.

4. The following is known as the maximum acyclic subgraph problem. Given a directed graph $G = (V, E)$, pick a maximum cardinality set of edges from E so that the resulting subgraph is acyclic. Implement an algorithm that gives a $1/2$ factor approximation here.

https://www.udacity.com/course/viewer#!/c-ud557/l-1209378918/m-3519618629

Hint: Rather than thinking about picking $1/2$ the optimum, think about picking $1/2$ of the edges.

Solution

Consider two following partition of the edges: those going from a lower index to a larger one and those going from larger index to a lower one. Both sets of edges form acyclic subgraphs, so it suffices simply to choose the larger set.

```
def acyclicsubgraph(G):
    """ Input: adjacency matrix of a directed graph G as numpy array
        Output: the adjacency matrix of an acyclic subgraph of G that
                has at least half the edges of G."""

    n = G.shape[0]

    forward_count = 0
    for i in range(n):
        for j in range(i+1,n):
            forward_count += G[i][j]

    back_count = 0
    for i in range(n):
        for j in range(0,i):
            back_count += G[i][j]

    ans = np.zeros((n,n),dtype=int)

    if forward_count > back_count:
        for i in range(n):
            for j in range(i+1,n):
                ans[i][j] = G[i][j]
    else:
        for i in range(n):
            for j in range(0,i):
                ans[i][j] = G[i][j]
    return ans
```

5. In the lesson, we gave a randomized algorithm for finding a minimum cut set in graph. Now, we consider the problem of finding a *maximum* cut set (all weights on edges are all one).
    a. Consider an algorithm that partitions the vertices into two sets A and B by placing each vertex uniformly and independently at random into one of the two sets. What is the expected value of $C(A,B)$, i.e. the number of edges crossing the cut?
    b. Use the method of conditional expectation to derandomize the above algorithm so that it always achieves this expected value. Implement your algorithm here.

    https://www.udacity.com/course/viewer#!/c-ud557/l-1209378918/m-3481888754

    c. Explain why your algorithm is correct. Use the notation $x_k = A$ to describe the event that vertex k is assigned to the set A in the partition. Use Y to denote the number of cut edges, and let $P : \{1,\ldots n\} \rightarrow \{A,B\}$ represent the partition returned by the derandomized algorithm.

Solution

a. The probability that the two vertices in an edge are placed in different halves of the partition is $1/2$. From the linearity of expectation it follows that the expected number of edges crossing the the cut is $|E|/2$.

b.

```
def approxmaxcut(G):
        """ Input: adjacency matrix of an undirected graph G as numpy array
            Output: a numpy array partitioning the vertices of the graph
                    so that entry equal to 1 are in one set and those equal
                    to -1 are in the other."""
        n = G.shape[0]

        ans = np.zeros(n, dtype=int)

        for i in range(n):
            ans[i] = 1 if np.dot(G[i,:],ans) < 0 else -1

        return ans
```

c. As argued in part a), in the randomized algorithm $E[Y] = |E|/2$. Also, since the randomized algorithm chooses the partition for each vertex uniformly at random, the conditional expectation

$$E[Y|x_1 = P(1),\ x_2 = P(2), \ldots, x_k = P(k)] \ =$$
$$E[Y|x_1 = P(1),\ x_2 = P(2), \ldots, x_k = P(k),\ x_{k+1} = A]/2 \ +$$
$$E[Y|x_1 = P(1),\ x_2 = P(2), \ldots, x_k = P(k),\ x_{k+1} = B]/2.$$

One of the conditional expectations on the right hand side is at least the one on the left, and this represents the one chosen by the deterministic algorithm from part b. Thus,

$$|E|/2 \leq E[Y|x_1 = P(1)] \leq \ldots \leq E[Y|x_1 = P(1),\ x_2 = P(2), \ldots, x_n = P(n)]$$

the last being the number of edges cut by the partition.