Advanced Algorithm Assignment

Arghya Bandyopadhyay 20CS4103

28th December 2020

1. Coding for Christofides Approximation Algorithm.

```
def christofidetsp(G):
    print("Graph: ", G)

    # build a minimum spanning tree
    MSTree = minimum_spanning_tree(G)
    print("MSTree: ", MSTree)

    # find odd vertexes
    odd_vertexes = find_odd_vertexes(MSTree)
    print("Odd vertexes in MSTree: ", odd_vertexes)

    # add minimum weight matching edges to MST
    minimum_weight_matching(MSTree, G, odd_vertexes)
    print("Minimum weight matching: ", MSTree)

    # find an eulerian tour
    eulerian_tour = find_eulerian_tour(MSTree, G)

    print("Eulerian tour: ", eulerian_tour)

    current = eulerian_tour[0]
    path = [current]
    visited = [False] * len(eulerian_tour)
    visited[0] = True

    length = 0

    for v in eulerian_tour[1:]:
        if not visited[v]:
            path.append(v)
            visited[v] = True

            length += G[current][v]
            current = v

    path.append(path[0])

    print("Result path: ", path)
    print("Result length of the path: ", length)
    return length, path


class UnionFind:
    def __init__(self):
        self.weights = {}
        self.parents = {}

    def __getitem__(self, object):
        if object not in self.parents:
            self.parents[object] = object
            self.weights[object] = 1
            return object

        # find path of objects leading to the root
        path = [object]
        root = self.parents[object]
        while root != path[-1]:
            path.append(root)
            root = self.parents[root]

        # compress the path and return
        for ancestor in path:
            self.parents[ancestor] = root
        return root

    def __iter__(self):
        return iter(self.parents)

    def union(self, *objects):
        roots = [self[x] for x in objects]
        heaviest = max([(self.weights[r], r) for r in roots])[1]
        for r in roots:
            if r != heaviest:
                self.weights[heaviest] += self.weights[r]
                self.parents[r] = heaviest


def minimum_spanning_tree(G):
    tree = []
    subtrees = UnionFind()
    for W, u, v in sorted((G[u][v], u, v) for u in G for v in G[u]):
        if subtrees[u] != subtrees[v]:
            tree.append((u, v, W))
            subtrees.union(u, v)

    return tree


def find_odd_vertexes(MST):
    tmp_g = {}
    vertexes = []
    for edge in MST:
        if edge[0] not in tmp_g:
            tmp_g[edge[0]] = 0

        if edge[1] not in tmp_g:
            tmp_g[edge[1]] = 0

        tmp_g[edge[0]] += 1
        tmp_g[edge[1]] += 1

    for vertex in tmp_g:
        if tmp_g[vertex] % 2 == 1:
            vertexes.append(vertex)

    return vertexes


def minimum_weight_matching(MST, G, odd_vert):
    import random
    random.shuffle(odd_vert)
    odd_vert = [1, 2, 3, 4]
    while odd_vert:
        v = odd_vert.pop()
        length = float("inf")
        u = 1
        closest = 0
        for u in odd_vert:
            if v != u and G[v][u] < length:
                length = G[v][u]
                closest = u

        MST.append((v, closest, length))
        odd_vert.remove(closest)


def find_eulerian_tour(MatchedMSTree, G):
    # find neigbours
    neighbours = {}
    for edge in MatchedMSTree:
        if edge[0] not in neighbours:
            neighbours[edge[0]] = []

        if edge[1] not in neighbours:
            neighbours[edge[1]] = []

        neighbours[edge[0]].append(edge[1])
        neighbours[edge[1]].append(edge[0])

    print("Neighbours: ", neighbours)

    # finds the hamiltonian circuit
    start_vertex = MatchedMSTree[0][0]
    EP = [neighbours[start_vertex][0]]

    while len(MatchedMSTree) > 0:
        for i, v in enumerate(EP):
            if len(neighbours[v]) > 0:
                break

        while len(neighbours[v]) > 0:
            w = neighbours[v][0]

            remove_edge_from_matchedMST(MatchedMSTree, v, w)

            del neighbours[v][(neighbours[v].index(w))]
            del neighbours[w][(neighbours[w].index(v))]

            i += 1
            EP.insert(i, w)

            v = w

    return EP
```

```
def remove_edge_from_matchedMST(MatchedMST, v1, v2):
    for i, item in enumerate(MatchedMST):
        if (item[0] == v2 and item[1] == v1) or (item[0] == v1 and item[1] ==
            del MatchedMST[i]

    return MatchedMST


if __name__ == '__main__':
    graph = {0: {}, 1: {}, 2: {}, 3: {}, 4: {}}
    #the weights are been assigned here between a and b
    #graph[a][b]=graph[b][a]=weight


    graph[0][1] = graph[1][0] = 3
    graph[0][2] = graph[2][0] = 10
    graph[0][3] = graph[3][0] = 5
    graph[0][4] = graph[4][0] = 1
    graph[1][2] = graph[2][1] = 7
    graph[1][3] = graph[3][1] = 8
    graph[1][4] = graph[4][1] = 2
    graph[2][3] = graph[3][2] = 9
    graph[2][4] = graph[4][2] = 6
    graph[3][4] = graph[4][3] = 5

    christofidetsp(graph)
```

2. Use indicator random variable to compute the expected value of the sum of n dices.

Let $X1, X2, \ldots, X6$ be the random varibles which count number of times faces $1, 2, \ldots, 6$ come up. Let the X be the random variable corresponding to sum of n dice rolls.

$$E[X] = 1E[X_1] + 2E[X_2] + \cdots + 6E[X_6]$$

Expected value E[X_i], is n/6.

$$E[X] = \sum_{i=1}^{6} iE[X_i]$$

$$= \sum_{i=1}^{6} i(n/6)$$

$$= (n/6)\sum_{i=1}^{6} i$$

$$= (21/6)n$$
$$= 3.5n$$

3. Let A[1..n] be an array of n distinct numbers. If i < j and A[i] > A[j], then the pair (i, j) is called an inversion of A. Suppose that the elements of A form a uniform random permutation of ⟨1, 2, ..., n⟩. Use indicator random variables to compute the expected number of inversions.

Let us define exactly $(\frac{n}{2})$ indicator random variables

$$X_{ij}, i < j, i, j, \in 1 \ldots n$$

with

$$X_{ij} = \begin{cases} 1, & \text{if A[i]}>A[j], \\ 0, & \text{if A[i]}<A[j], \end{cases}$$

Therefore, X_{ij} are indicator random variables of inversion on i, j.
Calculate the probability of inversion:

$$P(X_{ij} = 1) = \frac{1}{2}$$

because events A[i] >A[j] and A[i] <A[j] are equally likely. Therefore,

$$E[X_{ij}] = 1 \times P(X_{ij} = 1) + 0 \times P(X_{ij} = 0) = \frac{1}{2}$$

The total number of inversions T is of course:

$$T = \sum_{i<j} X_{ij}$$

where the sum has $(\frac{n}{2})$ terms.
Therefore, the expected number of inversions is:

$$E[T] = \sum_{i<j} E[X_{ij}] = \sum_{i<j} \frac{1}{2} = \frac{n}{2} \times \frac{1}{2} = \frac{n(n-1)}{4}$$

4. Suppose that we toss balls into b bins until some bin contains two balls. Each toss is independent, and each ball is equally likely to end up in any bin. What is the expected number of ball tosses?

There are b bins, and we are interested in the expected number of balls required that any of the bins obtain 2 balls.
First analyse the possible values for the random variable of the first time when some bin contains 2 balls:

- It can't be on the first ball, because all bins are empty
- It can't be after *(b+1)*th ball, because after b balls, if you throw another ball, it is guaranteed that some bin contains 2 balls

Therefore, the possible values are *[2. . . b+1]*
Now, to calculate the expected value of this variable, we need to calculate for every k (the first time some bin has 2 balls) the appropriate probability.
If k is indeed the first such time, then the first *k —1* balls all hit empty bins (for example, the second ball hits the empty bin with probability *(b —1)/b* because only one bin is populated, and so fourth), and then on the k-th try we hit one of those *k —1* populated bins (with the probability of *(k — 1)/b*). The total probability is then, after multiplying:

$$\frac{(k-1)}{b} \times \frac{(b-k+2)}{b} \times \cdots \times \frac{(b-1)}{b}$$

The first and the last term of this product are already explained above. The second term is the probability that *k — 1*-th ball hits one of the empty *b — (k— 2) = b — k + 2* bins. Other terms are generated analogously. Finally, by the definition of expectation, the result reads as:

$$\sum_{k=2}^{b+1} k \times \frac{(k-1)}{b} \times \frac{(b-k+2)}{b} \times \cdots \times \frac{(b-1)}{b}$$