Jason Less
404-640-158
CS180

CS180 Homework 2

Question 1

(a)

The algorithm below can be used to show that an undirected connected graph G = (V, E) s.t. all the vertices have even degrees contains an Eulerian cycle. An example will also be used to confirm the algorithm works for such a graph.

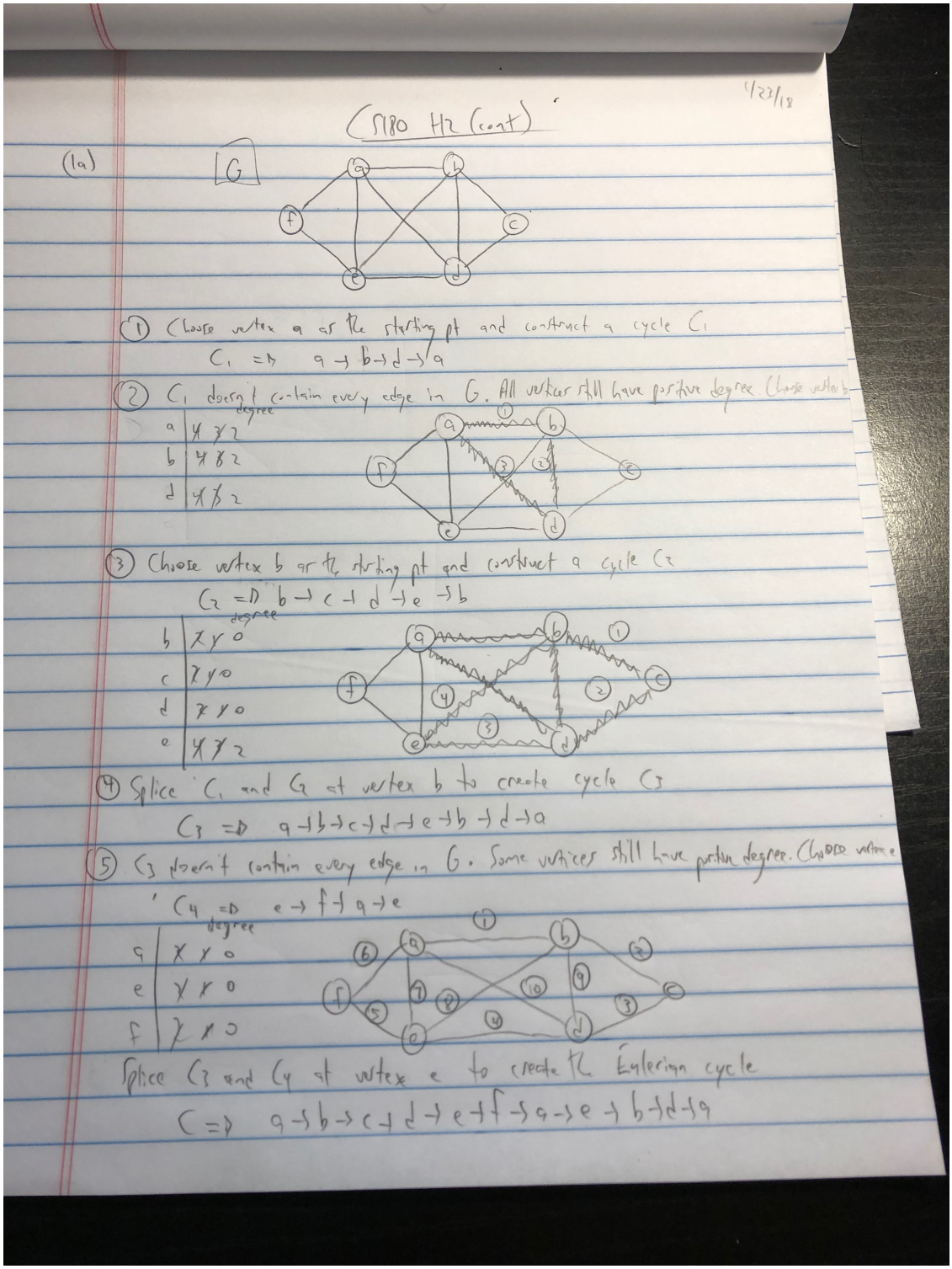
Algorithm

1. Choose a random vertex u in the graph and take a path on any of u's outgoing edges
    a. Continue along that path and discard all edges that are crossed
    b. Stop only when returning to vertex u (i.e. construct a cycle C_1 beginning and ending at vertex u)
2. If the cycle C_1 contains every edge in the graph, then done. Else, since the graph is connected (and not all edges have been crossed), there still exists at least one vertex v from the previous cycle C_1 with a positive degree
3. Choose vertex v and repeat step 1 using vertex v
    a. Construct another cycle C_2 beginning and ending at vertex v
    b. At this point, there are two edge-disjoint cycles C_1 and C_2, in which they share a common vertex v between them
4. Splice the two cycles into one cycle at the common vertex v to create a cycle C_3
5. Repeat step 2 using cycle C_3

The algorithm described above is an O(|E|) time algorithm as it visits each edge exactly once in edge-disjoint cycles and then splices them together.

Jason Less
404-640-158
CS180

Question 1
(a) (cont.) Example illustrating the algorithm on the previous page



Jason Less
404-640-158
CS180

Question 1

(b)

If G is a strongly, connected, directed graph in which in-degree equals out-degree for all vertices in G, then it can be shown that an Eulerian cycle C exists in G by breaking down Eulerian cycle C into simple cycles. By definition, a simple cycle is a cycle in a graph that doesn't repeat vertices. The Eulerian cycle C that we are assuming exists in graph G can be constructed from a set of simple cycles.

Based on the above assumption, graph G has an Eulerian cycle C, in which C is either a simple cycle or C an be constructed from a set of simple cycles. If C is a simple cycle, then each vertex has in-degree = out-degree = 1, so we are done. If C is not a simple cycle, then C can be constructed from a set of simple cycles $C_0, C_1, \dots, C_n$. We can show that an Eulerian cycle C exists by finding these simple cycles and removing them from graph G and from Eulerian cycle C. If we remove a simple cycle C_i from G and C (i.e. removing the edges associated with C_i), then C is still an Eulerian cycle for the remaining graph G. Additionally, each time a simple cycle C_i is removed from, the in- and out-degree of the vertices in the cycle are decremented by exactly one. Repeatedly removing simple cycles from graph G and Eulerian cycle C will remove all the edges, leaving all vertices with an in- and out-degree equal to zero. Thus, all vertices have in-degree equal to their out-degree.

Using the same algorithm in question 1a, the above claim can be carried out:

- Pick a random vertex u and construct a cycle C starting and ending with vertex u
    - Delete edges along this cycle C from graph G
    - Note that before visiting a vertex, the degree is even, while visiting the vertex, the degree is odd, and after visiting the vertex, the degree is back to even. Thus, the vertex still has in-degree equal to out-degree when traveling along this cycle C
- Pick a random vertex v from cycle C that still has a positive degree, and repeat the previous step
- Splice the two cycles together until we have crossed all edges, else continue

By executing the above algorithm, it is apparent that we can keep constructing cycles and splicing them together to complete an Eulerian cycle.

Algorithm:

- The algorithm used in question 1a can also be used for 1b, so to avoid rewriting the same thing, refer to question 1a. As described in 1a, the time algorithm is O(|E|).
- To add a little more detail to the algorithm used in question 1a:
    - An array can be used to keep track of visited vertices
    - An adjacency list can be used to keep track of out-going edges for each vertex
        - This is used so that edge deletion can be done in constant time (as you don't have to search for an edge each time we arrive at a new vertex)
    - The cycles starting and ending at the same vertex can be constructed using DFS

Jason Less
404-640-158
CS180

Question 2

Algorithm

- An iterative algorithm to solve the celebrity problem can be carried out in two phases:
  1. An elimination phase where we eliminate all but one candidate from being the celebrity
  2. The verification phase where we check to see if the candidate found in the last phase is indeed the celebrity
- Note that if the problem guarantees that a celebrity exists, then the verification phase is unnecessary. As this problem states to output none if no one is, I will assume the verification phase is indeed necessary

1. Elimination phase
   a. Maintain a list of all vertices (i.e. possible celebrities)
   b. Iterate through the adjacency matrix, and in each iteration we can delete one person from the list of possible celebrities:
      i. Each iteration consists of asking a simple question: Does person a know person b?
         1. If true (i.e. person a knows person b), then person a cannot be the celebrity
         2. If false (i.e. person a doesn't know person b), then person b cannot be the celebrity
   c. Repeat this process until there is only one person left (i.e. the celebrity candidate), which I will refer to as person c
2. Verification phase
   a. Iterate through the adjacency matrix again, this time asking two questions between person i (who is not c) and person c:
      i. Does person i know person c?
         1. This should be true for all i
      ii. Does person c know person i?
         1. This should be false for all i
   b. If question i and ii are always true, then person c is in fact the celebrity

This algorithm is O(N) as it consists of two steps: the elimination phase, which iterates through all n vertices of the adjacency matrix (O(N)), and the verification phase, which again iterates through the n vertices of graph G (O(N)). O(N + N) = O(N).

Jason Less
404-640-158
CS180

Question 3

Before providing the algorithm, I want to provide a little rationale to verify the algorithm will always return the diameter of an undirected tree. The longest path (i.e. the diameter) will always occur between two leaf nodes (i.e. nodes with only edge linked). This can be proved by contradiction: if one or both of the nodes in the assumed longest path are not leaves, then they can be extended to become leaf nodes, which would result in a longer path.

Therefore, we can use BFS from a leaf node s in the graph and stop once we find the farthest node possible from this node u to get the diameter. Then from node u, we can do another BFS to get the diameter. However, we don't need to start the first BFS from a leaf node; and instead, can just choose a random node in the undirected tree T. The reason we can do this is because our first BFS stops at the farthest node possible from the start node. As an undirected tree is a connected acyclic graph, the first BFS starting at any random node will always stop at a leaf (and not just any leaf but the leaf farthest away from the random node). Then from the leaf node discovered from the first BFS, perform another BFS to get the diameter of the tree.

Algorithm:

1. Run BFS on any node s in the graph
    a. Remember the last node u discovered
2. Run BFS from node u
    a. Remember the last node v discovered
    b. The diameter will consist of all edges in the path from u to v

The time complexity of this algorithm is O(N). The algorithm consists of two steps in which each step performs BFS on all nodes n in the graph (O(N)). O(N + N) = O(N).

Jason Less
404-640-158
CS180

Question 4

Using the hint, I was able to construct a recursive algorithm to partition the graph K_n into a set of n/2 edge-disjoint subsets s.t. each is a spanning tree of K_n. The base case is a complete 2-node graph that consists of one spanning tree. The inductive hypothesis for this problem is that if we assume that for any complete graph consisting of n nodes, there are n/2 edge-disjoint spanning trees.

Algorithm:

1. If the input is a complete 2-node graph, then we are done
2. Else:
    a. Remove two nodes a and b and their incident edges from the graph
    b. Recursively call the function with the remaining subgraph (which is still a complete graph)
        i. Note that from the inductive hypothesis the remaining subgraph consists of n/2 edge-disjoint spanning trees, which I will refer to as $T_1, T_2, \dots, T_{n/2}$. Also each vertex in the remaining subgraph will be referred to as $v_1, v_2, \dots, v_n$.
    c. Add back nodes a and b and their incident edges to the remaining subgraph
        i. At this point the remaining subgraph consists of n/2 edge-disjoint spanning trees $T_1, T_2, \dots, T_{n/2}$ as described in b. We need to extend each of these trees to include vertices a and b for them to be spanning trees
        ii. Since the graph is complete, both a and b have edges to each v_i (and an edge from a to b)
        iii. To each tree T_i, add edges (a, v_i) and (b, $v_{\frac{n}{2}+i}$). Since each T_i was edge-disjoint from the others, and all the edges just added were different for each T_i, the resulting spanning trees are all edge-disjoint
        iv. Construct one more spanning tree using the unused edges (and edge from a to b). Use edges (a, $v_{\frac{n}{2}+1}$) through (a, v_n) and edges (b, v_1)j through (b, $v_{n/2}$) and (a, b) for this last spanning tree.

To illustrate the above algorithm which might be somewhat confusing, I will provide an example on the subsequent page.

Jason Less
404-640-158
CS180

Question 4 (cont.)