CENG 567 Design and Analysis of Algorithms Homework 2

Ardan Yılmaz 2172195

November 8, 2022

1 Visiting all edges

For a network of two-way streets
a. Show that you can drive along these streets so that you visit all streets and you drive along each side of every street exactly once.
Visiting each edge exactly once and returning to the starting point means a Eulerian circuit exists. For Eulerian circuit to exist, the graph must be connected and all vertices must have even-degrees. The given two-street network is connected and has no odd-degree node, hence it has a Eulerian circuit. [1]

b. Suppose you drive in such a way that at each intersection, you do not leave by the street you used to enter that intersection unless you have previously left via all other streets from that intersection. Does this solve the problem in part a?

Claim: This is a valid algorithm
Pf. By Counter Example

The Eulerian circuit cannot be traversed for the following graph (that Deniz Germen came up with so cleverly [4]) using the given algorithm. In order to traverse a Eulerian circuit, one needs to start from one of the triangles, on top/bottom then finish its traversal before proceeding with the rest of the network. However, this cannot be done with the given algorithm, as it does not allow to take a u-turn unless it is the only option.

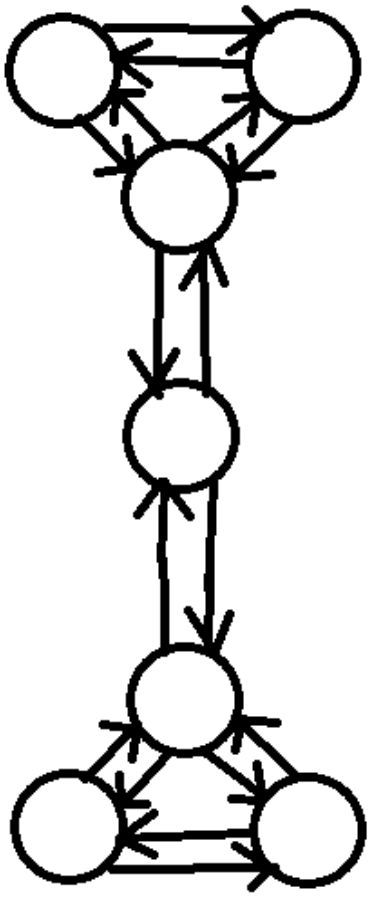

Figure 1: Such a network where the given algorithm fails to traverse a Eulerian circuit.

We have exchanged ideas with Yalgınay Yaltırık, Bilal Yağız Gündeger, and Deniz Germen regarding this question. [4]

2 Connectivity

2.1 Given a DAG, G, form a graph H from each SCC of G. If there is an edge between any pair of nodes in SCCs, put a corresponding edge in H. Show H is a DAG.

Directed Acyclic Graphs are graphs that are directed and do not contain cycles, as their name suggests. H is obviously a directed graph, so we need to prove it is acyclic, ie, there cannot be a

cycle among SCCs.
Say C and C' are SCCs of G, and there are two edges from C to C' and from C' to C in H. That is, for at least a pair of vertices such that $(v, v') \in CxC'$ there is a path from v to v' and one from v' to v. And as C and C' are SCCs and there is a cyclic path between C and C', each node from C is reachable from C' and vice versa. Set of nodes in C and C' are combined a SCC, which violates the property of a SCC, being maximal. Thus, we reach a contradiction that there cannot be a cycle between two SCCs. That is, H is acyclic, and hence a DAG.

2.2 If G is a forest (acyclic graph) with n nodes, k edges, and c components, prove that $c = n - k$.

A forest is a disjoint collection of components, and each component is a tree.
A tree with n nodes has $n - 1$ edges by definition.
Say n_i is the number of nodes of the i^{th} component in G.
$\sum_{i=1}^{c} n_i = n$ // total number of nodes is n
$\sum_{i=1}^{c}(n_i - 1) = k$ // Each component has $n_i - 1$ edges, adding up to k in total
$\sum_{i=1}^{c}(n_i - 1) = \sum_{i=1}^{c}(n_i) - c = k$
The above equation is $n - c = k$. That is, $c = n - k$

3 Job scheduling

a.
Consider the following case:
Skills of employees, E = $\{50, 54, 55\}$
Difficulty of jobs, D = $\{45, 45, 53\}$
The assignment of the given algorithm is as follows: $\{50-53, 54-45, 55-45\}$ with total difference: $3 + 9 + 10 = 22$
However, the following assignment $\{50 - 45, 54 - 45, 55 - 53\}$ yields a total cost $5 + 9 + 2 = 16$, which is better than the given algorithm. Hence the given algorithm is not optimal.

b.
A greedy heuristic to find the optimal solution would be to sort the employees and jobs based on their skill levels and difficulties, respectively. Then, assign the first employee, ie, least skilful, with the easiest job. Remove the assigned job and employee from the lists and repeat. The alternative solution described in part a uses this method.

The following is the greedy heuristic with $O(nlog(n))$ complexity.

Algorithm 1 Job Scheduling

Require: E, D //E and D keep skills of employees and difficulties of jobs
1: sort(E) // Use heap sort: $O(nlog(n))$ operation (Assuming in-place sorting)
2: sort(D) // Use heap sort: $O(nlog(n))$ operation (Assuming in-place sorting)
3: total_diff = 0
4: **for each** e, j in E x D **do** //get employee and job $O(n)$ operation
5: match(e, j)
6: total_diff += $|e - j|$
7: **end for**
8: **return** total_diff / n

A working example of this algorithm finding the optimal solution can be found in part a.

The complexity of the given algorithm is $O(nlogn + nlogn + n) = O(nlogn)$

Proof of Optimality (By Induction on the number of jobs/employees (n))

Say given employees and and jobs are sorted in ascending order based on their skill levels and difficulties, respectively as follows:
$S_e = S_1 \leq S_2 \leq ... \leq S_n$: sorted list of skills of employees and
$D = D_1 \leq D_2 \leq ... \leq D_n$: sorted list of difficulties of jobs.
Let $M = \{(S_i, D_i)\}_{i=1}^{n}$ be the minimal matching, ie, claiming the following.

Claim: $\sum_{i=1}^{n} |s_i - d_i|$ is the minimum sum of differences among all possible $\{(S_i, D_j)\}_{i=1,j=1}^{n}$ matches.

Base Case: n = 2
Claim: $|S_1 - D_1| + |S_2 - D_2| \leq |S_1 - D_2| + |S_2 - D_1|$
There are 6 possible cases for such inequality under the absolute value conditions.

1. $S_1 \geq D_1$, $S_2 \geq D_2$, $S_1 \geq D_2$, and $S_2 \geq D_1$
   $S_1 - D_1 + S_2 - D_2 \leq S_1 - D_2 + S_2 - D_1$
   This results in $0 \leq 0$, which is obviously true.
2. $S_1 \geq D_1$, $S_2 \geq D_2$, $S_1 < D_2$, and $S_2 \geq D_1$
   $S_1 - D_1 + S_2 - D_2 \leq D_2 - S_1 + S_2 - D_1$
   This results in $S_1 \leq D_2$, which is true under this assumption.
3. $S_1 < D_1$, $S_2 \geq D_2$, $S_1 < D_2$, and $S_2 \geq D_1$
   $D_1 - S_1 + S_2 - D_2 \leq D_2 - S_1 + S_2 - D_1$
   This results in $D_1 \leq D_2$, which is true as per the initial sorting.
4. $S_1 < D_1$, $S_2 < D_2$, $S_1 < D_2$, and $S_2 \geq D_1$
   $D_1 - S_1 + D_2 - S_2 \leq D_2 - S_1 + S_2 - D_1$
   This results in $D_1 \leq S_2$, which is true under this assumption.
5. $S_1 \geq D_1$, $S_2 < D_2$, $S_1 < D_2$, and $S_2 \geq D_1$
   $S_1 - D_1 + D_2 - S_2 \leq D_2 - S_1 + S_2 - D_1$
   This results in $S_1 \leq S_2$, which is true as per the initial sorting.

6. $S_1 < D_1$, $S_2 < D_2$, $S_1 < D_2$, and $S_2 < D_1$
   $D_1 - S_1 + D_2 - S_2 \leq D_2 - S_1 + D_2 - S_1$
   This results in $0 \leq 0$, which is obviously true.

Inductive Hypothesis: Assume that for k jobs and k employees the following holds: $\sum_{i=1}^{k} |s_i - d_i|$ is the minimum sum of differences among all possible $S_i - D_i$ matches

Inductive Step: Show this also holds for k+1 jobs/employees
Proof of Inductive Step by Contradiction:
Claim: Say another matching M' with a smaller sum of differences exists including at least two such pairs: (S_i, D_j) and $(S_j, D_i) \in M'$ where $i \neq j$.
Say $i < j$ ($S_i \leq D_i$) and all other matches in M' are the same as in M without loss of generality. For the claim to be true for M', the following should hold:
$\sum_{k \neq j}^{n} |S_k - D_k| + |S_i - D_j| + |S_j - D_i| \leq \sum_{k \neq j}^{n} |S_k - D_k| + |S_i - D_i| + |S_j - D_j|$
$|S_i - D_j| + |S_j - D_i| \leq |S_i - D_i| + |S_j - D_j|$, which is not possible as proven in the base case. Hence, we arrive at a contradiction.

M is the matching with minimum difference.

4 Spanning Trees and DFS

Given a connected undirected graph G, a spanning tree T of G, and a vertex v, design an algorithm to find if T is a valid DFS rooted at v.

The given spanning tree T cannot be found by DFS if G contains a cross-edge, ie, an edge from a node in one of the sub-trees of T to a node in another. Checking all sub-trees of T recursively, if there is an adjacent node in G that is neither a parent nor a child of the current root in T, there is a cross-edge.
This algorithm is developed based on the answer to the second question in [3].

Algorithm 2 isDFStree(G,T,v)

```
Require: G, T, v //G: graph, T: spanning tree, v: start vertex
mark(v)
for each edge (v, w) incident to v in T do
    if w is unmarked: isDFStree(G,T,w)
end for
//Now check for the adjacent nodes to that vertex in G
// If there is an unmarked node in G that is adjacent to v, there is a cross edge
for each edge (v, w) incident to v in G do
    if w is unmarked: return False // Cross-edge detected
end for
return True
```

1. **A worked example**

Say the given G is the graph with black edges from figure on the left and T, spanning tree of G, is indicated with red splashes on edges in Figure 1. The algorithm runs as follows:

1. Mark node 1
2. Traverse to node 2, call isDFStree(G,T,2)
3. Mark node 2
4. Traverse to node 4, call isDFStree(G,T,4)
5. Mark node 4
6. No unmarked adjacent node from node 4, go back to node 2 (returning to the recursive call)
7. Traverse to node 5 from node 2
8. No unmarked adjacent node from node 5, go back to node 2 (returning to the recursive call)
9. All adjacent nodes to node 2 in T is traversed, proceed with the following for loop
10. Checking all adjacent nodes to node 2 in G, 4 and 5 are marked, however, node 3 is unmarked. Hence, a cross-edge is detected. Figure 3 shows possible DFS trees, neither of which is the given T.

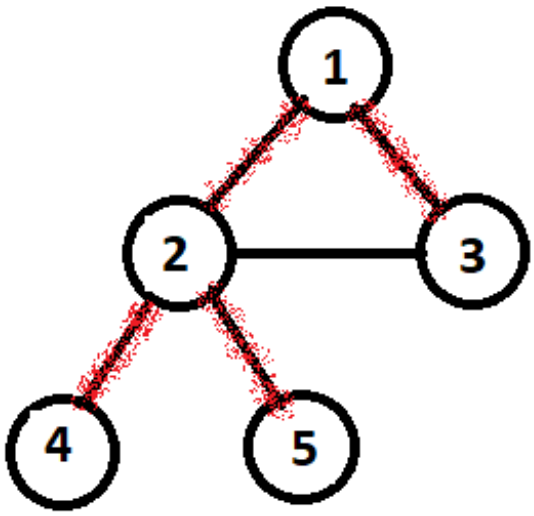

Figure 2: Given graph G and T

2. **Run-time Analysis**

The algorithm first runs through all edges in T, which is an $O(|E|)$ operation in the worst case. This is the number of recursive calls. The second for loop only runs if all ancestor and child nodes of a node is marked. The second for loop checks all adjacent nodes, $= O(|V|)$

operation. That is, after the return from recursive call of each node, $O(|V|)$ operation is run. Hence, this makes a $O(|E| + |V|)$ algorithm.

3. **Proof of Correctness**

Claim 1: Given T can be found if there is a cross edge from a subtree in T to another in G

Pf 1: (By contradictory example)

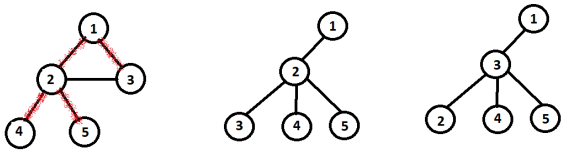

Figure 3: G, S, and T

Say the given G is the graph with black edges from figure on the left and T, spanning tree of G, is indicated with red splashes on edges. The two trees on the right of G are the possible DFS trees starting from node 1, neither of which is T. That is, in the existence of a cross-edge, T cannot be found. Hence, we arrive at a contradiction.

Claim 2: If there is no cross edge, DFS can find T.

Pf 2: (By structure)

The below figure shows a graph where there is no cross edge between Sub-tree 1 and Sub-tree 2. Provided that given T has this structure, ie, has these sub-trees, DFS can find the same tree, as indicated with red splashed edges. And, as a tree has to have a similar sub-structe in its sub-trees, as per its definition, DFS can find the these structures as long as there is no edge between different sub-trees.

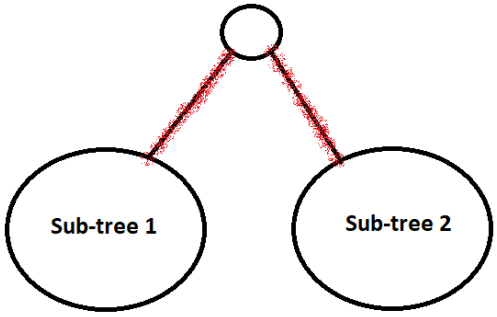

Figure 4: Graph with subtrees

Claim 3: If there is a cross edge, it is found by the given algorithm, isDFStree(G,T,v).

Pf 3: (By Contradiction) The algorithm first marks ancestors and children of a node in T, recursively. Then, checks if there is an unmarked node that is adjacent to it in the given graph. This indicates a cross-edge is detected, as there is an adjacent node from a different sub-tree, hence a cross-edge.

Say there exists an edge (u,w) in G, that is a cross-edge for T and the given algorithm failed to detect it. For the given algorithm not to detect a cross-edge, either (1) that edge is not present in G or (2) a node, w, from another sub-tree than that of u, is marked. The first case is clearly impossible. For the second case to be true, the algorithm needs to mark a node from a sub-tree that is not traversed yet. This is impossible, as the algorithm runs recursively, ie, processes nodes in the same order as DFS, hence traverses sub-tree by sub-tree. As either of the case is not possible, we arrive at a contradiction.

This proves correctness.

5 REFERENCES

1 https://www2.math.upenn.edu/ mlazar/math170/notes05-6.pdf

2 https://medium.com/@abhilashjn85/assignment-problem-minimizing-the-worker-cost-while-covering-all-the-jobs-254bc79187e7

3 http://im.ntu.edu.tw/ tsay/dokuwiki/lib/exe/fetch.php?media=courses:alg2011:hw8s.pdf

4 Yalgınay Yaltırık, Deniz Germen, Bilal Yağız Gündeger