CS 170 Homework 4

Allen Guo
with Melissa Huang, Courtney Vu, and Peter Xu

September 25, 2015

Let $\mathbb{N}$ *represent the set of all positive integers.*

The remainder of this page is intentionally left blank.

1 DFS Warm-up

Part a

DFS visit order: $A, B, D, E, G, F, C, H, I$.
Pre- and post-numbers:

- A: $1, 12$
- B: $2, 11$
- D: $3, 6$
- E: $4, 5$
- G: $7, 10$
- F: $8, 9$
- C: $13, 18$
- H: $14, 17$
- I: $15, 16$

Edges:

- (A, B): Tree
- (B, G): Tree
- (B, D): Tree
- (G, D): Cross
- (D, E): Tree
- (E, D): Back
- (G, F): Tree
- (C, H): Tree
- (H, I): Tree
- (C, I): Forward
- (A, E): Forward

Part b

There are eight strongly connected components.

2 Directed Paths

Main Idea

After linearizing G to find a source, s, we recursively find the length of the longest directed path starting from s and check if it matches $|G|$. If it does, we know there exists a directed path that goes through every vertex in G.

Pseudocode

```
procedure PATH_EXISTS(G):
    input: G, a DAG
    output: whether there exists a directed path that goes through every vertex in G

    let G' be a topological sorting of G
    let S be the first vertex in G'
    let N be the number of vertices in G

    return if HELPER(S) equals N

procedure HELPER(S):
    input: S, a DAG vertex
    output: the number of vertices in the longest directed path starting from S

    if the result of calling HELPER(S) has been memoized:
        return the memoized result

    if S has no children:
        return 1

    call HELPER on each of the children of S
        store the results to a list R
    let NUM be max(R) + 1

    memoize NUM as the result of calling HELPER(S)
    return NUM
```

Analysis

Linearization takes $O(|E| + |V|)$ time, as described in the textbook.

`HELPER` behaves like DFS: for each node, we do constant work, then iterate over all outgoing edges. In total, we iterate over each edge once. Therefore, computing `HELPER(S)` also takes $O(|E| + |V|)$ time.

All other steps are expected to be constant-time—for instance, we can use a hash table to memoize calls to `HELPER`.

As a result, the entire algorithm runs in $O(|E| + |V|)$ time.

Proof of Correctness

If there is a directed path that visits every vertex in G, then G must contain a single source. (Why? Suppose toward a contradiction that there were multiple sources. Then no directed path would include all of the sources, since by definition they have indegrees of 0.) We first linearize G in order to find this source, s.

We then find the length of the longest directed path that starts from s. We do this recursively: Our base case occurs for a sink, in which case the longest path length is trivially 0. The length of the longest path starting from s is one plus the greatest of the lengths of the paths starting from children of s. Why? Suppose there was a longer path, of length l. This would mean that a child has a longest path length of length $l - 1$. But this is greater than the longest path lengths of all of the other children of s, which leads to a contradiction.

If there exists a directed path that visits every vertex in G, it must start from the source s and it must have length $|G|$. Therefore, we can check if such a path exists by checking if the length of the longest directed path starting from s matches $|G|$.

3 A Game of Choosing Edges in a DAG

Main Idea

We first introduce a definition: a node n is *dominant* iff a player starting from n has a winning strategy independent of the opponent's moves. The opposite is a *non-dominant* node. (This terminology is vaguely game theoretic sounding and is less confusing than denoting nodes as either "winning" or "losing".)

Our goal therefore is to determine if s is a dominant node.

We find this recursively: Any sink is a non-dominant node. A non-sink node is dominant iff any of its children are non-dominant.

Pseudocode

```
procedure DOMINANT(S):
    input: S, a node in a DAG
    output: whether or not S is dominant per the given definition

    if S is a sink:
        return FALSE

    compute DOMINANT for each child of S
    return TRUE if any of the results are FALSE
```

In the interest of clarity and concision, we simply state that calls to `DOMINANT` should be memoized, as above in Question 2. If a memoized value is needed, it is not recomputed.

Analysis

As with DFS, `DOMINANT` does constant work for each node, then loops over outgoing edges. Overall, each edge is encountered once. Therefore, our algorithm runs in $O(|E| + |V|)$ time.

Proof of Correctness

Our base case is, as all good base cases should be, trivial: any sink is a non-dominant node because any player starting from a sink will lose immediately according to the rules of the game.

We now prove the statement behind the recursive case: If a non-sink node n is dominant, then at least one of its children are non-dominant; the converse is also true.

- Proof of contrapositive: Suppose all of the children of n are dominant. Then regardless of where a player P_1 starting from n moves, the opponent P_2 will have their next turn starting from a dominant node. This means P_1 does *not* have a winning strategy independent of the opponent's moves, so n must be *non-dominant*.
- Proof of converse: Suppose there is at least one child of n that is non-dominant. The winning strategy for a player P_1 at n is to move to that child. Then, in the next turn, their opponent P_2 will *not* have a winning strategy independent of the opponent's moves. This means that P_1 can, by choosing the right moves, force a victory: in other words, n is a dominant node.

It immediately follows that our algorithm is correct.

4 Counting Paths

Part a

We seek to inductively prove that $(A^k)_{s,t}$ is equal to the number of paths from s to t of length k.

Our base case occurs for $k = 1$: Any path of length 1 must consist of a single directed edge. Because A^1 is the adjacency matrix for G, $A_{s,t}$ is by definition the number of paths from s to t of length 1.

Now suppose the proposition above holds for $k = m$. We must now show that it holds for $k = m + 1$.

We know that $A^{m+1} = A^m \times A$. Because of how multiplication is defined for matrices,

$$(A^{m+1})_{s,t} = \sum_{w=1}^{n} \left[(A^m)_{s,w} \times A_{w,t} \right]$$

Consider the product in square brackets on the RHS. This is product of the number of ways s can reach w with a path of length m (by the inductive step) multiplied by the number of ways w can reach t with a path of length 1 (by the base case). This product is then (by the multiplicative counting principle) the number of ways s can reach t with a path of length $m + 1$ (by passing through w). We sum over all valid w, yielding the total number of paths from s to t of length $m + 1$.

Our inductive proof is hereby complete.

Part b

Main Idea

We can visualize G^k as a series of $k + 1$ columns/sets of vertices $V_0 \dots V_k$, with edges lying between neighboring sets. Every set of edges between two sets V_i and V_{i+1} is essentially a copy of those in E. This means our graph G^k is essentially just k copies of the original graph G that have been laid side-by-side and "glued together" using the metaphorical "glue" that is E.

We know from Part a that we can determine the number of paths of a given length between two nodes using matrix multiplication.

Suppose the matrix A is an adjacency matrix such that $A_{i,j} = 1$ if there is an edge between $i \in V_0$ and $j + n \in V_1$. Then to find the number of paths from s (in V_0) to t (in V_k), we simply calculate $(A^k)_{s,t'}$, where $t' = t \mod n$. (t' is the "mirror image" of t in V_1.)

Pseudocode

```
procedure NUM_PATHS(G, n, s, t):
    inputs: G, a directed graph represented as an adjacency matrix
            n, the row/column size of G
            s, the start node
            t, the destination node
    output: the number of paths from s to t of length k

    find G^k using exponentiation by squaring
        as described in lecture

    let t' be t mod n
    return the element of G^k at row s and column t'
```

Analysis

Exponentiation by squaring requires $O(\log k)$ steps, as shown in lecture. Each such step runs in $O(n^3)$ time. Therefore, the algorithm as a whole runs in $O(n^3 \log k)$.

Proof of Correctness

We have already proven the relationship between adjacency matrix exponentiation and counting paths. In this section, we'll formalize and expand upon the "glue" concept described in the "Main Idea" section above.

Consider G^1, which consists of vertices $V_0 \cup V_1$ and edges E. From the given definitions, we know that

$$V_0 = \{1, \ldots, n\}$$

and

$$V_1 = n + V_0 = n + \{1, \ldots, n\} = \{n+1, \ldots, 2n\}$$

The edges are given by E, which (due to the bipartiteness mentioned in the problem) only go from V_0 to V_1. So essentially we have shown that $G^1 = G$. We also see that G^1 is one copy of G glued to, well, nothing—there's just one copy!

But now suppose G^z is z copies of G glued together for some integer $z \geq 1$. We seek to prove that G^{z+1} is $z+1$ copies of G glued together.

G^{z+1} is different from G^z in that there is one additional set of vertices, V_{z+1}. This is true because the given definition implies that

$$V_{z+1} = (z+1)n + V_0$$

Similarly, there is also one additional set of edges, E_n, which point from V_n to V_{n+1} (due to bipartiteness) and are essentially copies of those in E. This is true because the given definition implies that

$$E_{z+1} = (zn, zn) + E_1$$

We have thus shown inductively that a path from s to t essentially involves moving through k copies of G. The rest of the algorithm follows immediately from this fact and the logic laid out in the "Main Idea" section.

5 Fixing a Basketball Cup

Main Idea

We form a directed graph G where each of the n teams is a node and E is the set of edges. (So an edge exists from team i to team j iff i would beat j.)

To find W, we detect strongly connected components (SCCs), then find the metagraph, G'. Then:

- If G' has a single node, $W = G$.
- If G' has multiple sources, W is the empty set.
- Otherwise, W is the set of all nodes in G corresponding to the source node of the metagraph G'.

Pseudocode

```
procedure FIND_W(G):
    input:  G, a directed graph where the vertices V are the n teams and
               the edges are given by E
    output: W, a set containing every team for which
               there exists a possible cup schedule where victory is guaranteed

    find G', the metagraph of G
        using the SCC algorithm

    if G' contains one node:
        return V

    find ss, a list of all source nodes in G'
        using the DFS algorithm: explore the entire graph, keeping all nodes with indegree = 0

    if ss contains more than one element:
        return the empty set
    return all nodes in G corresponding to ss[0] in G'
```

The SCC algorithm is described in the textbook on page 94; DFS is on page 85.

Analysis

The bottlenecks here are the SCC and DFS algorithms. As stated in the textbook, both run in $O(|E| + |V|)$ time. So our algorithm will also run in $O(|E| + |V|) = O(|E| + n)$ time.

Proof of Correctness

Suppose we have a graph G consisting of a single SCC; that is to say, G is strongly connected. Then W must equal G. We can prove this inductively:

Our base case occurs when G contains a single node, in which case $W = G$ trivially because that node is a winner without even trying, that lucky duck.

Suppose we know that, given a strongly connected graph G_n with n nodes, that $W = G_n$. We seek to show that a strongly connected graph G_{n+1} with $n + 1$ nodes has $W = G_{n+1}$.

WLOG, let us select a node s from G_{n+1}. Let us call the remaining nodes (that are not s) T.

- If T is strongly connected, any node in T can be its winner (from the inductive hypothesis). Suppose we specifically choose one of the children of s to be the winner of the nodes in T. Now behold: s must be able to defeat that child—after all, we constructed our edges such that an edge exists from team i to team j iff i would beat j. So we have found a possible cup schedule where s is guaranteed victory, so s is definitely in W.
- If T is not strongly connected, there must be a direct path from a child of s to a parent of s. (Why? Suppose there was no direct path from any child of s to any parent of s. Then G_{n+1} would not be strongly connected. Contradiction.) Then that child of s is clearly able to win among the nodes in T, in which case we can arrange s to defeat its child, as above. Therefore, s is in W.

We chose s without loss of generality, so we could repeat this for any s in G_{n+1}. Therefore, W must contain all nodes in G_{n+1}.

This completes our inductive proof. We have shown that a SCC allows any of its nodes to be a guaranteed winner.

As a result, coming back to our original problem, dividing G into SCCs helps us better understand the relative strengths of the teams. Each node in G' (the metagraph) is a set of teams that essentially forms an equivalence class—if one of its members can be defeated by a team s then *any* of its members can be defeated by s, using the logic presented above.

Hence, the source in the metagraph is the set of teams that are capable of defeating teams from all other metagraph nodes. At the same time, any of them can win among each other. This is why the source in the metagraph constitutes W.

What if the metagraph has multiple sources? Since nothing is known about how teams in different metagraph source nodes would do against each other, we can yield no conclusions, so W is the empty set.

6 Course Prerequisites

Main Idea

We begin by linearizing G. If we find that G cannot be linearized, we stop immediately, since that means G contains cycles and it is impossible to graduate.

Afterwards, we simply find the longest path in the linearized graph.

Pseudocode

```
procedure SEMESTERS_NEEDED(G):
    input:  G, a directed graph of size n such that every node is a course
               and there is an edge from course v to course w iff v is a
               prereq for w
    output: the minimum number of semesters for graduation, or -1 if
            graduation is possible

    find G', a valid linearization of G
        if G cannot be linearized, return -1

    return the length of the longest path in G'
```

Linearization is described in the textbook. The "longest path in a DAG" algorithm is given in the "Russian Boxes" problem from discussion.

Analysis

Linearizing G to find G' takes linear time, as described in the textbook.

Finding the longest path in G' takes linear time also, since it involves recursively finding the longest path starting from each vertex, which means each vertex is processed once and each edge is examined once.

Therefore, the algorithm as a whole runs in linear time.

Proof of Correctness

A cycle indicates that graduation is impossible, since it would imply that a class is its own prerequisite.

A directed graph can be linearized iff it is acyclic, so if we are unable to linearized G, we immediately know that graduation is impossible.

Suppose the length of the longest path in G is x. Why is x the minimum number of semesters required for completion? Suppose it was possible to complete the curriculum faster, say in y semesters, where $y < x$. But then the student could have taken at most y of the classes in the longest path, since each class in the longest path is a prereq for the one after. And since $y < x$, the student must not have finished yet. Contradiction.