# CS 170 Homework 4

Due **9/27/2023, at 10:00 pm (grace period until 11:59pm)**

## 1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write "none".

**Solution:** None in particular. I worked on most of these problems in homework party and received some guidance with problems 2 and 5 from TAs, but other than that the remainder is my own work.

## 2 Distant Descendants

You are given a tree $T = (V, E)$ with a designated root node $r$ and a positive integer $K$. For each vertex $v$, let $s[v]$ be the number of descendants of $v$ that are a distance of at least $K$ from $v$. Describe an $O(|V|)$ algorithm to output $s[v]$ for every $v$. Give a 3-part solution.

**Solution:** I'll start with an algorithm description, then proof of correctness, and finally a runtime analysis. For the correctness proof, I first wrote it in the form that I understand, then I follow it with an inductive proof just to make sure I'm being rigorous.

**Algorithm Description:** I did this in bullet points and not pseudocode because I think it's easier to follow:

1. Initialize a list $s$ to store $s[v]$, set it to all zeros.
2. Traverse this tree with DFS (using standard tiebreaking), and use a stack to keep track of the path between the root node and the node we're currently on.
3. Once a new node has been visited, increment the $s[v]$ value for the $K$-th node in the stack by 1. The $K$-th node is counted from the top down. If the stack has size less than $K$, do nothing. In practice, this might have to be the $K + 1$-th node, depending on when we add the current node to the stack.
4. Repeat this process until DFS is complete
5. For every node in the tree, add their current $s[v]$ value with all the $s[v]$ values of their children, starting from the leaves.

**Proof of Correctness:** Firstly, DFS is guaranteed to visit every node once, so if we assume that the algorithm outputs the correct values, we know that it will output the correct $s[v]$. As for the third step, we know that we need to increment the $s[v]$ value for the $K$-th node in the stack since from the perspective of the ancestor, the current node is a distance of $K$ away, hence we need to count it in its $s[v]$ value. Once this is completed, each $s[v]$ only tells us how many descendants of $v$ are *exactly* a distance of $K$ from $v$.

The crucial step in this algorithm is the final one, where we add the current $s[v]$ value with all the $s[v]$ values of their children, starting with the leaves. To prove that this is correct, consider a vertex $v$ and its children $c$, where each of $s[c]$ counts the number of descendants that are at least a distance $K$ from $c$. Given this fact, we then know that the nodes $s[c]$ counts will be at least $K + 1$ from $v$, so they should be added to $s[v]$, which currently stores the number of nodes exactly $K$ from $v$. Note that the number of nodes exactly $K$ from $v$ are not counted in $c$, since they'd be a distance of $K - 1$ from $c$, so we guarantee that we aren't overcounting any nodes.

If this process is done for all the children of $v$, then we can guarantee that $s[v]$ indeed counts all the vertices at least $K$ from $v$. We can then recursively apply this process up the tree until we've hit all vertices, at which point we are done.

Another way to prove this is to use induction on the number of layers in the tree, starting with the base case where we have a single node. In this case, unless $K = 0$, which would mean $s[v] = 1$, then we should have $s[v] = 0$, which our algorithm does catch successfully (see step 3).

Now let our inductive hypothesis be the fact that we get correct values for $s[v]$ on a tree with $n-1$ layers. We now prove that this works for a tree with $n$ layers.

We can construct this new tree by adding a node, then connecting it to an arbitrary number of trees with $n-1$ layers (this generates an $n$-layer tree). By our inductive hypothesis, running our algorithm on each individual $n-1$-layer tree generates the correct value for $s[v]$. This means that for the root node, our algorithm will take the sum over all the $s[v]$ values of its children (which are nodes that are at least $K+1$ away from the root), and add to it the nodes which are $K$ from the root. This ensures that we are catching all nodes that are at least $K$ from the root, hence we've proven the algorithm's correctness.

**Runtime Analysis:** The big steps that will contribute to the runtime is step 2 and 5. Step 2 executes DFS, which takes $O(|V|+|E|)$ time. Since this is a tree, then we know that there are at most $|V|-1$ edges, so this simplifies the runtime to $O(2|V|) = O(|V|)$.

For step 5, this step also requires some type of tree traversal, and since we're considering all the children of a given vertex at a time, BFS can be used here. Its runtime is the same as DFS, so it's also $O(|V|)$.

Finally, for step 3, the arithmetic on the vertices individually is assumed to take constant time, and is done $|V|$ times since we increment once per vertex, so this also takes $O(|V|)$ time. Step 1 also takes at most $O(|V|)$ times, since we need to allocate $|V|$ entries.

Thus overall, our runtime is:

$$O(|V|) + O(|V|) + O(|V|) = 3O(|V|) = O(|V|)$$

which satisfies the runtime constraint.

# 3 Where's the Graph?

Each of the following problems can be solved with techniques taught in lecture. Construct a simple directed graph, write an algorithm for each problem by black-boxing algorithms taught in lecture, and analyze its runtime. You do not need to provide proofs of correctness.

(a) The CS 170 course staff is helping build a roadway system for PNPenguin's hometown in Antarctica. Since we have skill issues, we are only able to build the system using one-way roads between igloo homes. Before we begin construction, PNPenguin wants to evaluate the reliability of our design. To do this, they want to determine the number of *reliable* igloos; an igloo is reliable if you are able to leave the igloo along some road and then return to it along a path of other roads. However, PNPenguin isn't very good at algorithms, and they need your help.

Given our proposed roadway layout, design an efficient algorithm that determines the number of reliable igloos.

**Solution:** We can run Kosaraju's algorithm on the graph to find the number of SCCs present in the graph. Then, all SCCs that have more than 1 node are immediately part of a cycle, and hence count toward the number of reliable igloos. Therefore, all we have to do after having identified all the SCCs is to add up the number of vertices in SCCs that have more than 2 vertices.

(b) There are $p$ different species of Pokemon, all descended from the original Mew. For any species of Pokemon, Professor Juniper knows all of the species *directly* descended from it. She wants to write a program that answers queries about these Pokemon. The program would have two inputs: $a$ and $b$, which represent two different species of Pokemon. Her program would then output one of three options in constant time (the time complexity cannot rely on $p$):

(1) $a$ is descended from $b$.

(2) $b$ is descended from $a$.

(3) $a$ and $b$ share a common ancestor, but neither are descended from each other.

Unfortunately, Professor Juniper's laptop is very old and its SSD drive only has enough space to store up to $O(p)$ pieces of data for the program. Give an algorithm that Professor Juniper's program could use to solve the problem above given the constraints.

*Hint: Professor Juniper can run some algorithm on her data before all of her queries and store the outputs of the algorithm for her program; time taken for this precomputation is not considered in the query run time.*

**Solution:** We can run DFS on this ancestral tree, and in every species "node," we can store its preorder and postorder values so that we can query them later. This can be stored in a dictionary (since every species is unique by assumption), then when we're given $a$ and $b$, we can query from the dictionary and compare the preorder and postorder values. Specifically:

- If $\text{pre}(a) < \text{pre}(b) < \text{post}(b) < \text{post}(a)$, then $b$ is descended from $a$.
- If $\text{pre}(b) < \text{pre}(a) < \text{post}(a) < \text{post}(b)$ then $a$ descended from $b$.
- If $\text{pre}(a) < \text{post}(a) < \text{pre}(b) < \text{post}(b)$ then $a$ and $b$ share a common ancestor but are not descendants.

Since we need to store values for each pokemon, then there are total $4p$ values we need to store, which is $O(p)$ in storage complexity. In terms of time complexity, we are performing four dictionary queries with $O(1)$ time complexity and comparing them, so the total time complexity of the algorithm is $O(1)$, as desired.

(c) Bob has $n$ different boxes. He wants to send the famous "Blue Roses' Unicorn" figurine from his glass menagerie to his crush. To protect it, he will put it in a sequence of boxes. Each box has a weight $w$ and size $s$; with advances in technology, some boxes have negative weight. A box $a$ inside a box $b$ cannot be more than 15% smaller than the size of box $b$; otherwise, the box will move, and the precious figurine will shatter. The figurine needs to be placed in the smallest box $x$ of Bob's box collection.

Bob (and Bob's computer) can ask his digital home assistant Falexa to give him a list of all boxes less than 15% smaller than a given box $c$ (i.e. all boxes that have size between 0.85 to 1 times that of $c$). Bob will need to pay postage for each unit of weight (for negative weights, the post office will pay Bob!). Find an algorithm that will find the lightest sequence of boxes that can fit in each other in linear time (in terms of the graph).

*Hint: How can we create a graph knowing that no larger box can fit into a smaller box, and what property does this graph have?*

**Solution:** We know that we need to place the figurine in box $x$, so create a graph with the following condition: if box $a$ fits into box $b$ (that is, $a$ is within 0.85 to 1 times $b$'s size), then $a \longrightarrow b$ share a directed edge. Furthermore, let the weight of this edge be determined by the weight of box $b$. With this construction, this graph must be acyclic (and hence a DAG), since having a cycle would imply that box $a$ fits within itself, which is impossible.

With this construction, finding the lightest sequence of boxes essentially asks us to find the path with the minimum edge weight sum, starting from $x$. Since this directed graph has negative edge weights, we need to use the Bellman-Ford algorithm to determine distances from $x$. Once this is computed, we simply have to find the node that has the minimum distance, then return the path from $x$ to that node.

# 4 Semiconnected DAG

A directed acyclic graph $G$ is *semiconnected* if for any two vertices $A$ and $B$, there is a path from $A$ to $B$ or a path from $B$ to $A$. Show that $G$ is semiconnected if and only if there is a directed path that visits all of the vertices of $G$. Make sure to prove both sides of the "if and only if" condition.

*Hint: Is there a specific arrangement of the vertices that can help us solve this problem?*

**Solution:** We first prove that if there is a directed path that visits all of $G$'s vertices, then $G$ is semiconnected. Since this directed path visits all vertices of $G$, then we can rethink of $G$ as a string of vertices $S = \{v_1, v_2, \dots, v_n\}$, where every successive vertex is the next vertex in the directed path. This means that we can think of the vertices as being somewhere along the line $v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_n$.

Given this construction, now let's look at two vertices $v_i$ and $v_j$. We know that since these vertices are in $G$, then they lie somewhere within $S$. Since they lie somewhere within the string vertices, then we know that if $v_i$ comes before $v_j$ in the string, then there is a path from $v_i \longrightarrow v_j$. Otherwise, there must be a path from $v_j \longrightarrow v_i$, since the string goes through all vertices.

Now we proceed to prove the alternate direction, that if $G$ is semiconnected, then there exists a path that visits all vertices of $G$. Since $G$ is a DAG, then we know that it must have a source node, otherwise we'd have a cycle in $G$. Now, we can generate our cycle using the following construction:

Start at the source node of $G$, call it $v$, Since $G$ is semi-connected and $G$ is a source node, then we know that $v$ has an outgoing edge to every other vertex in $G$. Now, consider the graph without $v$. Since $v$ was a source node, then the remaining graph must also be a DAG, so it must also have a source node $u$. Walk along the path from $v$ to $u$, and repeat this process until all vertices have been visited. Further, based on the fact of $G$ is semiconnected, then $u$ must have an outgoing edge to all vertices except for $v$, since $u$ is also a source node in $G \setminus v$; this proves that our recursive construction can indeed be completed.

By the end of this construction, we will have found a sequence of vertices starting from the source of $G$ that traverses all the vertices of $G$, as desired.

Alternatively, I have also heard in Homework Parties that this is equivalent to topologically sorting the DAG and traversing it in sorted order, which is also essentially what I'm doing here.

# 5 2-SAT

In the 2SAT problem, you are given a set of clauses, where each clause is the disjunction (OR) of two literals (a literal is a Boolean variable or the negation of a Boolean variable). You are looking for a way to assign a value true or false to each of the variables so that all clauses are satisfied – that is, there is at least one true literal in each clause. For example, here's an instance of 2SAT:

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee x_4)$$

Recall that $\vee$ is the logical-OR operator and $\wedge$ is the logical-AND operator and $\overline{x}$ denotes the negation of the variable $x$. This instance has a satisfying assignment: set $x_1$, $x_2$, $x_3$, and $x_4$ to `true, false, false, and true`, respectively.

The purpose of this problem is to lead you to a way of solving 2SAT efficiently by reducing it to the problem of finding the strongly connected components of a directed graph. Given an instance $I$ of 2SAT with $n$ variables and $m$ clauses, construct a directed graph $G_I = (V, E)$ as follows.

- $G_I$ has $2n$ nodes: one for each variable and its negation.
- $G_I$ has $2m$ edges: for each clause $(\alpha \vee \beta)$ of $I$ (where $\alpha$, $\beta$ are literals), $G_I$ has an edge from from $\overline{\alpha}$ to $\beta$, and one from the $\overline{\beta}$ to $\alpha$.

Note that the clause $(\alpha \vee \beta)$ is equivalent to each of the implications $\overline{\alpha} \implies \beta$ and $\overline{\beta} \implies \alpha$. In this sense, $G_I$ records all implications in $I$.

(a) Show that if $G_I$ has a strongly connected component containing both $x$ and $\overline{x}$ for some variable $x$, then $I$ has no satisfying assignment.

**Solution:** If $x$ and $\overline{x}$ are part of a strongly connected component, then there is a sequence of implications that is constructible (via a path from $x$ to $\overline{x}$) that resolves to $x \implies \overline{x}$, and also $\overline{x} \implies x$ due to the properties of SCC. This is impossible, since this pair of implications is equivalent to $x \iff \overline{x}$, which has no satsifying assignment.

(b) Now show the converse of (a): namely, that if none of $G_I$'s strongly connected components contain both a literal and its negation, then the instance $I$ must be satisfiable.

*Hint: Pick a sink SCC of $G_I$. Assign variable values so that all literals in the sink are True. Why are we allowed to do this, and why doesn't it break any implications?*

**Solution:** We follow the hint: consider $G_I$, and let $G_{\text{sink}}$ be the set of sink nodes in this graph. We will set all the nodes in $G_{\text{sink}}$ to be true. This doesn't break any implications since all nodes $x$ in the sink are part of an implication of the form $v \implies x$, and setting $v$ to true or false is allowable as long as $x$ is true.

Before we go on, we need to first prove that this assignment does not generate a case where some literal $a \implies x$ in the sink and also $a \implies \overline{x}$; had this existed, then we'd have no satisfying assignment. To do this, we use the property that if $a \implies x$, then $\overline{x} \implies a$ for any literal $a$, meaning that the node $\overline{x}$ must be part of a *source* – this ensures that nothing can imply $\overline{x}$, so we don't have any contradictions. To put this into graph language, it's equivalent to saying that once we've identified $G_{\text{sink}}$, then the negation of all literals in $G_{\text{sink}}$ are found in another set $G_{\text{source}}$ which are all source nodes.

Now, note if a node is not part of the sink/source, they can (for now) be assigned to anything and it wouldn't affect the sink and source nodes. Therefore, we can consider the graph without $G_{\text{sink}}$ and $G_{\text{source}}$, and we can recursively apply this same logic of identifying *new* sink and source nodes until we've covered all nodes. Since at every step we're not generating all logical implications, this construction proves that indeed $G_I$ has a satisfying assignment.

(c) Conclude that there is a linear-time algorithm for solving 2SAT. Provide the algorithm description and runtime analysis; proof of correctness is not required.

**Solution:** Based on parts (a) and (b), we can see that we have a satisfying assignment if and only if $G_I$ does not contain a strongly connected component that contains both $x$ and $\overline{x}$. Therefore, we can run Kosaraju's algorithm, then for each SCC that this algorithm identifies, check whether the SCC contains literals $x$ and $\overline{x}$. If they do, then return false, else return true.

As for runtime analysis, we know that Kosaraju's algorithm runs in $O(|V| + |E|)$ time,

# 6 [Coding] DFS and Kosaraju's Algorithm

For this week's coding questions, we'll implement DFS and apply it to find the strongly connected components within a graph via Kosaraju's algorithm. There are two ways that you can access the notebook and complete the problems:

1. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   `https://github.com/Berkeley-CS170/cs170-fa23-coding`

   and navigate to the `hw04` folder. Refer to the `README.md` for local setup instructions.

2. **On Datahub**: Click here and navigate to the `hw04` folder if you prefer to complete this question on Berkeley DataHub.

Notes:

- *Submission Instructions:* Please download your completed submission `.zip` file and submit it to the gradescope assignment titled "Homework 4 Coding Portion".
- *OH/HWP Instructions:* Designated coding course staff will provide conceptual and debugging help during office hours and homework parties.
- *Academic Honesty Guideline:* We realize that code for some of the algorithms we ask you to implement may be readily available online, but we strongly encourage you to not directly copy code from these sources. Instead, try to refer to the resources mentioned in the notebook and come up with code yourself. That being said, we **do acknowledge** that there may not be many different ways to code up particular algorithms and that your solution may be similar to other solutions available online.