Navigation – Analysis

Author: Emil Indzhev

Part I

Subtasks 1-4

1 Subtask 1: Cycle

The idea is to circle around the circle.

We need 2 colors: `INIT` and `DONE`. Always set the current node to `DONE` and go to an `INIT`. Otherwise, if no `INIT`s, terminate.

Code: `emil_cycle_2.cpp`

2 Subtask 2: Star

The idea here is that we can distinguish whether we are in a leaf or the base of the star. So we'll just visit all leaves and mark them and then terminate.

We need 2 colors: `INIT` and `DONE`. If in a leaf, set the current node to `DONE` and go to the base of the star. If in the base of the star, go to an `INIT` leaf. Otherwise, if no `INIT` leaves, terminate.

Code: `emil_star_2.cpp`

3 Subtask 3: Path

The idea is we go to 1 end of the path and then to the other and then terminate.

We need 3 colors: `INIT`, `LEFT` and `DONE`. First, if we see no `DONE`s, we are in phase 1, so we set current node to `LEFT` and move to an `INIT`. If no `INIT`s, switch to phase 2. In phase 2: set current node to `DONE` and move to an `INIT` or a `LEFT`. If no `INIT`s and no `LEFT`s, terminate. Notice that phase switching and termination can only happen in leaves.

Code: `emil_path_3.cpp`

4 Subtask 4: Tree

The idea is to run a DFS.

We need 3 colors: `INIT`, `DFS` and `DONE`. `INIT`s are unvisited nodes. `DONE`s are visited nodes whose full subtree has been explored. `DFS`es are nodes that are currently on the path from the start (root) to the current node (i.e. they are on the DFS stack).

The strategy is: if we see any `INIT`s, set the current node to `DFS` and go to an `INIT`. Otherwise, if we see any `DFS`es (there can be at most one), set the current node to `DONE` and go to the `DFS`. Otherwise (we must be in the root), terminate.

Code: `emil_tree_3.cpp`

Part II

Subtasks 5-7

5 The problem

We can try running this on a non-tree and the issue is when we need to backtrack but the current node has a back-edge. Then we see multiple `DFS` nodes and cannot know which way to go.

There is a "known" approach that works on general graphs, seen in some past problems (such as Robot at IOI 2023) of running a BFS using layered DFS runs that expand it by 1 layer each time. The idea is that a BFS tree has no back-edges (only edges to the previous, current or next layers). So we can record the distance from the root to the current node modulo 3 and this way we can classify each edge. Naively, such a solution might take 19 colors, but it is possible to do this with 8 colors. However, this approach cannot be used for this problem, since the number of iteration it needs is $O(N^2)$, while here we need $O(N)$ iterations.

In general, anything that is not effectively a DFS will need a superlinear number of steps. Therefore, we want to do a DFS. However, we can see that this is not possible on a general graph: In order to backtrack, we need to have a way to recover the DFS stack, which is basically a permutation. The issue is that there are $N!$ possible permutations, while there are only $N \times C^N$ possible configurations (for some constant C). Thus, we need to take advantage of the fact that the graph is a cactus.

6 Solution with $N + 1$ colors

The above argument does not hold if $C = N$. One simple solution that works even in a general graph is to run a DFS, but instead of a `DFS` color have N `DFS`-like colors, marking the depth of the node. Then when we backtrack, we just go to the deepest node with depth smaller than ours (this way we don't even need a `DONE` color). If we need to backtrack from the root (which has depth 0), we terminate. This solution actually only does $2N$ iterations. It gets 15% of the points for these subtasks.

Code: `emil_cactus_bad_n+1.cpp`

7 Subtask 5

This subtask serves a few purposes.

Firstly, it allows solutions with $O(\text{MaxDegree})$ number of colors (due to storing which node is our parent), which rely on the fact that the order of `adjColors` is consistent across multiple visits to the same node. This again, works on a general graph with bounded degree.

Secondly, it simplifies some of the cases of most of the full solution ideas and allows easier elimination of some of their colors. Notably, in this subtask, each node is part of at most one cycle, which simplifies a lot of things. Additionally, since the starting node has degree 1, in most solutions we'd only start there and finish there, so we don't need to worry about special cases due to the root.

8 Key observations

The key observation is that in a cactus back-edges are much more well behaved than ones in a general graph. This results in several concrete observations. Various solutions (and variations on these solutions) use one or both of these or further more specific observations.

1. Each node has at most 1 back-edge (this is because the parent to self edge would otherwise be in two simple cycles). Therefore, while running a DFS, the current node sees at most 2 nodes on the DFS stack.
2. If a node has a back-edge, its parent cannot have a back-edge (this is because the grandparent to parent edge would be in two simple cycles).

9 Solutions

We have 3 very distinct types of solutions:

1. Storing parent information along the DFS path/stack.
2. Storing no extra information and instead finding it when we need to backtrack.
3. Storing child information along the DFS path/stack.

Some of these end up being slightly worse than others, but all can net a high score on the problem. We also believe studying them is useful, since the techniques in them can be useful for other problems.

9.1 Storing parent information

9.1.1 Storing the parent's index

Since each node sees at most 2 `DFS`-like nodes (one of which is its parent), we can simply record which of the 2 is its actual parent. Basically, for all `DFS`-like colors, we have 2 versions, specifying which node is the current node's parent.

The colors we need are: `INIT`, `DONE`, `DFS_1/2`, `DFS_FRONT_1/2`, `DFS_NEW_1/2` (8 in total).

Each time we go to an `INIT`, we leave the current node in a `DFS_FRONT` color. Then its child can identify it as its parent and figure out the parent's index (0 or 1) among the `DFS`-like nodes it sees. Then the child is colored to a `DFS_NEW` color and we go to the parent. Now the parent returns to a `DFS` color and goes to the `DFS_NEW` color it sees (the child). After that, we proceed normally. Notice that this procedure allowed us to tell a node who its parent is and store that information in the node, while returning the parent to a regular color. Finally, when we need to backtrack, we just backtrack to the correct parent `DFS`-like node according to the index the current node stores. This solution needs $4N$ time, since the robot goes through each tree edge 4 times (and doesn't go through back-edges).

Code: `emil_cactus_par_8.cpp`

We can optimize this to 6 colors by removing the `DFS_NEW` one. We can merge it with the `DFS_FRONT` ones. Everything works normally, except that when we are in a `DFS_FRONT` node, we figure out whether it is the original `DFS_FRONT` or the original `DFS_NEW` based on whether it sees another `DFS_FRONT` node.

Code: `emil_cactus_par_6.cpp`

9.1.2 Marking the parent

The issue with the above ideas (besides needing $4N$ iterations) is that the number of `DFS`-like colors is multiplied by 2 due to storing the index of the parent in the node. A much better idea is to simply mark the parent on its own node. Then when we need to backtrack, if we have a back-edge, we just backtrack to the marked parent. The key is that we only need this marking if the current node has a back-edge. And, if that is the case, no node can have a back-edge to the current node's parent.

More concretely, we still follow a similar `DFS`/`DFS_FRONT` procedure. However, when we enter `INIT` we normally just backtrack, leaving it as `INIT` (which means we enter the `DFS_FRONT` parent, which is turned into `DFS` and we go back to the `INIT` child). This allows us to reuse `DFS_FRONT` as the parent marker too. However, if we detect that we need to mark the parent (when we enter an `INIT` node and we see 2 `DFS`-like nodes), we simply proceed without going back to the parent. Thus, the parent is naturally left in the `DFS_FRONT` state. This needs only 4 colors, but still $4N$ iterations.

Code: `emil_cactus_parimp_4.cpp`

9.2 Storing no extra information

In this solution we won't record any extra info while running the DFS. Instead, we'll figure out which node is our parent when we need to backtrack. If we only see one DFS-like node, we just go to it. If not, then we must have

a back-edge. Therefore, our parent cannot have a back-edge. Therefore, our parent would see only one DFS-like node (excluding the current node). On the other hand, any other DFS-like node would see at least 2 DFS-like nodes (its parent and its child); however, the root is an exception to this (so we can use an extra color for it – notice that this case does not matter in subtask 5).

The colors we need are: `INIT`, `DONE`, `DFS`, `DFS_ROOT`, `PARSEEK`, `PARFOUND`, `DFS_PAR`, `DFS_NOTPAR` (8 in total).

The color `DFS_ROOT` will be used only for the root. That way, if we see a `DFS` node and a `DFS_ROOT` node, we know the `DFS` one is our parent. So the only issue is if we see 2 `DFS` nodes. In that case, we mark the current node with `PARSEEK`, meaning we are looking for its parent. Then we go to a `DFS` node. Then (we know we are in this case, since we see a `PARSEEK` node), we check whether it is the parent or not (based on how many `DFS`-like nodes it sees). If yes, set it to `DFS_PAR`. Otherwise, set it to `DFS_NOTPAR`. In either case, go back to the `PARSEEK` node. There, we may need to do this again until we see a `DFS_PAR` node.

Once we see a `DFS_PAR`, we need to clean up the `DFS_NOTPAR` ones (if any), color the current node in `PARFOUND` and go to the `DFS_NOTPAR`. There, color it in `DFS` and go back to the `PARFOUND` node. Finally, set the current node to `DONE` and go to the `DFS_PAR` node. Notice that it will be overwritten with `DFS` (if exploring further), `DONE` (if backtracking) (`PARSEEK` cannot happen).

This solution does at most 6 steps per node with back-edge. In a cactus, at most half the nodes have back-edges (the maximum number of edges is $3N/2$). Therefore, in total we get $2N + 6N/2 = 5N$ iterations. Notice that it doesn't use the ordering of the adjacency list being consistent.

Code: `emil_cactus_find_8.cpp`

There are multiple possible optimizations, which can mostly be done in any order. We'll present one order of doing them, but you can also find codes for some other orders in the sample solutions.

9.2.1 Removing `PARFOUND`

First, we can remove `PARFOUND`, its only purpose is to tell the `DFS_NOTPAR` node that it needs to clear itself. However, this is the only situation in which we are ever in a `DFS_NOTPAR` node. Therefore, we can merge `PARFOUND` with `PARSEEK`.

Code: `emil_cactus_find_7_no-found.cpp`

9.2.2 Removing `DFS_ROOT`

We can also remove the `DFS_ROOT` and merge it with `DFS` with one key observation. Let's see what the issue is if we do this. Everything works, until we have a back-edge to the root. If we first try our actual parent, everything is fine. However, if we try the root, it looks just as if it is our parent (only sees one `DFS`-like node), so we'll mistakenly go to it. However, we can imagine this as something like a DFS (which keeps a stack of nodes), but at this point flipping the stack, i.e. swapping which side we're expanding from. The important invariant that all the DFS-like nodes form a single path still holds. Therefore, we'll still visit all nodes and then terminate (though this termination might just not happen from the root, which is not an issue). Notice that the new subtree of the old root cannot have back-edges to above the root (in the flipped stack). Additionally, the other end of the stack (the new root) has visited all of its other children, so it will never be a back-edge. Therefore this stack-flipping/rerooting operation will only happen once.

Code: `emil_cactus_find_6_no-found-root.cpp`

9.2.3 Removing `DFS_NOTPAR`

Next, we can also exploit the fact that we have consistent ordering in order to remove `DFS_NOTPAR` and merge it with `DFS`. This optimization will also lower the number of iterations to $3N$. When we try a node (i.e. when we see `PARSEEK`), we'll just either mark it with `DFS_PAR` or leave it as DFS and return to the `PARSEEK` node. Then when we are in `PARSEEK` (note that we are not actually in `PARSEEK` before we try the first of the 2 `DFS` nodes), we can see that either we found the parent (so we just backtrack to it) or did not (in which case we only see 2 `DFS` nodes). In that latter case we can just backtrack to the other DFS node (i.e. at the start always try the left DFS node and in this case always backtrack to the right DFS node). Besides removing one color, this also makes us only do 2 extra steps per back-edge, for a total of $3N$ steps, which also lets us get points from subtask 7.

Code: `emil_cactus_find_5_no-root-found-notpar.cpp`

9.2.4 Removing `DFS_PAR`

Finally, we can merge `PARSEEK` and `DFS_PAR` in a similar way to how we merged `DFS_FRONT` and `DFS_NEW` in the other type of solution. Now, we when the current node is in this new `PARSEEK_OR_PAR` color, we treat it as `PARSEEK` when we see 2 `DFS` nodes or see another `PARSEEK_OR_PAR` node (which we treat as `DFS_PAR`). We also treat it as `PARSEEK`, if we are in a `DFS` node and see a `PARSEEK_OR_PAR` (it is just `PARSEEK`). However, if we are in `PARSEEK_OR_PAR`, but see no other `PARSEEK_OR_PAR` and see at most 1 `DFS` node, we must be in `DFS_PAR`.

Code: `emil_cactus_find_4_no-found-root-notpar-par.cpp`

9.2.5 Variations

Next are some interesting variations not covered above, that would be interesting to understand and show useful techniques for such problems. Additionally, it is interesting to see what the best solution is if the adjacency list is unordered (e.g. if the grader shuffles it each time).

9.2.6 Clearing `DFS_NOTPAR` with `PARSEEK_OR_PAR`

Let's see whether we can remove the `DFS_PAR` color (merge it with `PARSEEK`) without removing the `DFS_NOTPAR` color (and without depending on the consistent order of the adjacent nodes).

There is an issue when we try to clear `DFS_NOTPAR` and see two `PARSEEK_OR_PAR` nodes (can happen when there is a triangle, i.e. a cycle with 3 nodes). In this situation we can simply guess which node is the `PARSEEK` one and go to it (clearing to `DFS`, as usual). Then when we are in a `PARSEEK_OR_PAR` node, but see `INIT`s, this must have happened and we must have guessed wrongly, so we just go to the adjacent `PARSEEK_OR_PAR` (while keeping that node as `PARSEEK_OR_PAR`) – this fixes the wrong guess with one extra step. Notice that in case the `DFS_PAR` had already visited all its other children, we may mistakenly think it is the real `PARSEEK`. However, this is not an issue, since this just mean we backtrack the other way around the triangle but everything is still correct after that. This ends up with $5.5N$ iterations in total.

Code: `emil_cactus_find_5_no-found-root-par.cpp`

9.2.7 Lazy `DFS_NOTPAR` clearing

In all of the above solutions that do have `DFS_NOTPAR` we can actually just not clear it. Let's look at doing this on our simplest solution. When we are in `PARSEEK` and see a `DFS_NOTPAR`, we will just backtrack to the DFS node. This would leave some `DFS_NOTPAR` nodes lying around, but no node would have them as a back-edge. Additionally, that node's child cannot have any back-edge. Therefore, we never see both `DFS` and `DFS_NOTPAR` while traversing normally. So we just act normally or backtrack to the `DFS_NOTPAR`, if it is the only `DFS`-like node we see. This solution only does $3N$ steps, which means we can pass the last subtask even without any of the other optimizations.

Code: `emil_cactus_find_7_no-found_lazy-notpar.cpp`

The above argumentation only works until we remove the `DFS_ROOT` color. This is because once we switch the root the stack is flipped. There are no back-edges from the new subtree of the old root until we backtrack above the root. Then it is possible for some node to have a back-edge to a `DFS_NOTPAR` node. However, in that case it would also have a regular DFS node and that must be its parent. Additionally, such a node cannot have another back-edge. Therefore, we can just ignore the `DFS_NOTPAR` nodes unless we see no DFS node, in which case we backtrack to the `DFS_NOTPAR` node. All of this also works if we merge `PARSEEK` and `DFS_PAR`, so this is actually a cleaner resolution to the issue from before. Thus we get a 5 color, $3N$ iterations solution to the unordered version of the problem.

Code: `emil_cactus_find_5_no-found-root-par_lazy-notpar.cpp`

9.2.8 Lazy PARSEEK clearing

In the previous solutions, we merged PARSEEK and DFS_PAR. Note that DFS_PAR is needed only for clearing PARSEEK. Another idea is to simply not clear PARSEEK immediately (if we happen to guess the right parent, but just leave it there).

Of course, to have any chance, we need to only go to PARSEEK, if it looks like we might be answering its query, i.e. if we see 2 or more DFS-like nodes. There is still an issue with this – if we later mistakenly think we are answering the PARSEEK; this can only happen if we actually are in the middle of answering another PARSEEK and we are not the parent (if we just backtracked to the current node, we would only see 1 DFS node – we cannot have a back-edge since our child did). When this happens, we might go to the wrong PARSEEK and then think we are answering it that we are the wrong parent and go to its other adjacent DFS-like node. After that, we turn out to be in the middle of the DFS path with no PARSEEK around us, so we end up losing. This actually cannot happen on subtask 5, so it is a valid solution for it.

However, we can see that the issue comes from the current node's ex-children being marked as PARSEEK – they will always be earlier in the adjacency list than the new PARSEEK. Therefore, we can fix this solution by always answering the latest/rightmost PARSEEK in our adjacency list. This needs only 4 colors and does $3N$ iterations, so it is also a full solution.

Code: emil_cactus_find_4_no-found-root-notpar-par_lazy-parseek.cpp

9.3 Storing child information

One other idea, similar to the one storing the parent's index, is to store the child's index. Namely, whenever we go down towards a child, we can see what its index (among the DFS-like nodes) will be and store it in the current node. Then when we need to backtrack from a node and it sees multiple DFS-like nodes, we can guess which one the parent is. If we choose right, we are done. If we choose wrong, we can use the stored child indices to descend down to the correct parent (without changing any colors along the way). This sounds like it might need $O(N^2)$ steps, but we can notice that we go around each cycle at most once, this this would actually be just $3N$ steps.

Now we just need to figure out how to know that we are in this descend mode. For non-root nodes without back-edges, we can tell by simply checking whether the number of seen DFS-like nodes is 2 or 1: if 2, we are descending; if 1, we are in the regular mode. For the root, for now, we can just have a special color (and we never backtrack to it unless it is the only option). Finally, we can notice that we never need to descend through nodes with back-edges (since it is impossible to have a back-edge to a parent/grandparent of theirs), so they don't need to store child information. We can just have a special color for such nodes (found by seeing 2 DFS-like nodes when in an INIT node). Lastly, we need to not backtrack to such nodes, if we see regular DFS-like nodes. This solution uses 6 colors and needs $3N$ iterations.

Code: emil_cactus_child_6.cpp

We can optimize this solution in a similar way to our main one and simply remove the color for the root. This means that we may backtrack to it (if we guess wrongly). Then the root starts exploring its other subtrees (and we never encounter back-edges to the first subtree). Finally, we start backtracking through the first subtree (and exploring any unexplored branches). In this last part, the notations of child indices of the not-yet-backtracked nodes are wrong. However, we can never see back-edges to them, since that would break the cactus property (as that whole path is part of a cycle). Therefore, this is fine and would work. This uses 5 colors (and still $3N$ iterations). It is the best solution we have for the unordered version of the problem.

Code: emil_cactus_child_5_no-root.cpp

10 Even better solutions?

We do not known whether a 3 color solution exists or whether a 4 color one exists for the unordered version of the problem. However, we have proven (using a bruteforce) that a 2 color solution does not exist even for a path (even in the ordered version).

Similarly, we do not know whether a faster than $3N$ solution exists with a constant number of colors for the ordered or the unordered version of the problem.