COL351 Assignment 2

Avval Amil - 2020CS10330
Ujjwal Mehta - 2020CS10401

September 22, 2022

1 Maximum Sum

We first look at a linear (array) representation of the disc. If $D = (d_1, d_2 \dots d_n)$ was the disc, then let the array $A = [d_1, d_2 \dots d_n]$ represent the disc. We define the 'complement' of a contiguous collection of sectors of the disc to be the set of all the sectors that are NOT present in that collection. Note that the complement of a contiguous collection is also contiguous in the disc. Also note that a contiguous collection in the disc may not correspond to a contiguous collection in the array (e.g the collection $\{d_n, d_1\}$ is contiguous in the disc but not in the array). Now we prove the following claims :

Claim 1

Given any contiguous collection in the disc, either it or its complement is contiguous in the array.

Proof
Let the contiguous collection be $\{d_i, \dots d_j\}$ (note that the index of each element in the collection is one larger than that of the previous one except the case when d_n and d_1 are two consecutive elements in the collection).
There are two cases now -
Case 1 : $i \leq j$
If this is the case then the corresponding collection in the array is $[d_i \dots d_j]$ which is contiguous due to the fact that $i \geq 1$ and $j \leq n$.
Case 2: $j < i$
This case will happen only when d_n and d_1 both are in the collection. This is because if neither of them were in the collection then as we saw earlier the index of each element is greater than the one before it and thus j cannot be less than i. So, let the collection be $\{d_i, \dots d_n, d_1, \dots d_j\}$. The complement of this collection is $\{d_{j+1}, \dots d_{i-1}\}$. If $j = i - 1$, then it means that the entire disc was the contiguous collection in which case its complement is the null set (which we can assume to be contiguous in the array as well). If $j < i - 1$ then it implies that $j + 1 \leq i - 1$ and hence the collection $\{d_{j+1}, \dots d_{i-1}\}$ is contiguous in the array.

Claim 2

The complement of a contiguous collection of maximum sum is a contiguous collection of minimum sum in the disc.

Proof
Assume for contradiction that this is not the case and there exists a contiguous collection M of maximum sum whose complement is not the contiguous collection of minimum sum. Let the sum of the elements of M be S and the total sum of all elements in the disc be T. Then the sum of the elements of the complement of M is $T - S$.

Let the minimum contiguous sum collection be m with sum of elements s. We know that $s < T - S$ as the complement of M did not have minimum sum. This implies that $T - s > S$. However, $T - s$ is precisely the sum of the complement of m. As this is also a contiguous collection, it cannot have sum more than S and hence we reach a contradiction.

From the above two claims, we can say that for the maximum sum contiguous collection, either it is also contiguous in the array and hence is the maximum sum contiguous subarray of the array (if it was not then we have a contradiction as the maximum sum contiguous subarray of the array is also contiguous in the disc), or it is not contiguous in the array but its complement is contiguous in the array and is the minimum sum contiguous subarray of the array (again by contradiction as contiguous subarrays are also contiguous in the disc). We can find both of these and take the maximum accordingly.
If we can find out both the max and min sum contiguous subarrays in the array A (with sums S and s), then using the total sum of all elements T we can find out the maximum of the values S and $T - s$, if $S > T - s$ then the max contiguous subarray is the contiguous collection, otherwise the complement of the min contiguous subarray is the contiguous collection.
Also note that finding minimum contiguous subarray is same as finding the maximum contiguous subarray (multiply each entry of the array by -1 and now the minimum contiguous subarray becomes the maximum contiguous subarray). So if we have an algorithm to find the maximum sum contiguous subarray in $\mathcal{O}(n)$ time we can apply it to the array A and the array A multiplied by -1 (element wise), take the collection corresponding to the maximum and we will be done.

Maximum Sum Subarray Algorithm (in an Array)

The algorithm iterates over all elements of the array A and at the i^{th} iteration keeps track of the maximum contiguous subarray ending at $A[i]$. Out of all these subarrays the maximum one is chosen and its start and finish indices returned. For the proof of correctness we will prove the following claims -

Algorithm 1: Maximum Sum Contiguous Subarray

```
$max \leftarrow (-\infty)$;
$curr \leftarrow 0$;
$start \leftarrow 1$;
$finish \leftarrow 1$;
$startHere \leftarrow 1$;
for $i$ = 1 to $n$ do
    if $curr + A[i] < 0$ then
        $curr \leftarrow 0$;
        $startHere \leftarrow i + 1$;
    else
        $curr \leftarrow curr + A[i]$;
    end
    if $max < curr$ then
        $max \leftarrow curr$;
        $finish \leftarrow i$;
        $start \leftarrow startHere$;
    end
end
Return $(start, finish)$;
```

Claim 3

The variable $curr$ at the end of the i^{th} iteration of the algorithm stores the value of the contiguous subarray of maximum sum ending at $A[i]$.

Proof
Proof by Induction on i.

Base Case : i=1
If $A[1]$ was positive then $curr$ would get value $A[1]$ which is the correct maximum contiguous subarray, and if $A[1]$ was negative then $curr$ would be zero corresponding to a null subarray. Hence base case is satisfied.

Induction Hypothesis : At end of $(k-1)^{th}$ iteration, $curr$ contains sum of maximum contiguous subarray ending at $A[k-1]$

Induction Step: If the maximum contiguous subarray ending at $A[k]$ is nonempty, then that means that it must be the maximum contiguous subarray ending at $A[k-1]$ appended with $A[k]$. This is because if this was not the case then we could simply have taken the maximum contiguous subarry ending at $A[k-1]$ and appended it with $A[k]$ to get subarray of larger sum. Also, if the max subarray ending at $A[k-1]$ appended with $A[k]$ has a positive sum then the max subarray ending at $A[k]$ cannot be empty as then the sum would be zero which is lesser. Similarly, if the max subarray ending at $A[k-1]$ appended with $A[k]$ has a negative sum then the max subarray ending at $A[k]$ will be empty as then the sum would be zero.
$curr$ gets value $curr + A[i]$ iff the latter is greater than or equal to zero, otherwise it gets the value of 0, this is justified due to the reasoning above as initially $curr$ contains max subarray value ending at $A[k-1]$. Hence, proved.

Claim 4

Variable max stores the sum of the max sum contiguous subarray encountered so far

Proof
The variable max is only updated when $curr$ exceeds the current value of max and hence it will contain the maximum value.

Claim 5

The $startHere$ variable at the end of the i^{th} iteration stores the starting index of the maximum subarray ending at $A[i]$. If the subarray is empty then the starting index is supposed to be the next index.

Proof
Proof by induction on i

Base Case : i=1
If $A[1]$ is positive, then $startHere$ is assigned, 1 else it is assigned 2, which is the correct answer.

Induction Hypothesis: At end of $(k-1)^{th}$ iteration, $startHere$ contains starting index of maximum contiguous subarray ending at $A[k-1]$

Induction Step: If maximum contiguous subarray ending at $A[k-1]$ was empty and $A[k]$ is positive, then $startHere$ was assigned value k in the previous iteration and wont change now which is correct. If maximum contiguous subarray ending at $A[k-1]$ was empty and $A[k]$ is negative, $startHere$ is assigned value $k+1$ which is correct by definition. If maximum contiguous subarray ending at $A[k-1]$ was not empty and after appending $A[k]$ sum becomes negative, then $startHere$ is assigned value $k+1$ which is correct and finally if maximum contiguous subarray ending at $A[k-1]$ was not empty and after appending $A[k]$ sum is still positive, then $startHere$ is not changed which is correct as only the ending index of the two subarrays differ in this case. Hence, for all cases the induction step works.

Using the above claims we can say that the variables $start$ and $finish$ contain the starting and finishing indices of the maximum contiguous sum subarray encountered so far, as they are only updated when the value of max is changed, and the $start$ variable is assigned value of $startHere$ while $finish$ is assigned the index of the current index which is correct as that is the position where the subarray ends.
Note that if all elements of the array are negative, $start$ will have value $n+1$ in the end which indicates that the maximum sum subarray is empty. This proves the correctness of the algorithm.

Time Complexity Analysis

It is seen that only one pass is required over the entire array, and in each iteration of the pass, we are doing at most a constant amount of assignments and comparisons, hence the time taken is $\mathcal{O}(n)$.

Using this maximum contiguous subarray algorithm, we can first find the maximum sum subarray in A (lets say with sum S), then find the minimum sum subarray in A (lets say with sum s) using the maximum sum subarray algorithm applied to A' which is just A with elements multiplied by negative 1.
If total sum of all elements of A is T and $S > T - s$ then the maximum contiguous subarray is the maximum contiguous collection in the disk, otherwise the complement of the minimum contiguous subarray is the maximum contiguous collection in the disk. Total time taken is $\mathcal{O}(n)$ the algorithm described above is applied 2 times, and summing over elements of the array also takes $\mathcal{O}(n)$ time.

2 Forex Trading

Part 1

Let $G = (C, E)$ be a graph with the vertex set V representing the currencies $c_1, c_2 \dots c_n$ and the edge weights corresponding to the negative logarithm of the currency exchange rate (given by $R(i, j)$ in the question). That is, if there exists an edge (i, j) in the graph, the weight of that edge will be $-log(R(i, j))$ or $wt(i, j) = -log(R(i, j))$.

This is done to ensure that a positive gain cycle in the graph given in the question (i.e a sequence of vertices $c_1, c_2, \dots c_k, c_1$ such that $R(c_1, c_2) \cdot R(c_2, c_3) \cdot \dots \cdot R(c_k, c_1) > 1$ corresponds to the same sequence in the graph G but with a negative cycle. This is because if $R(c_1, c_2) \cdot R(c_2, c_3) \cdot \dots \cdot R(c_k, c_1) > 1$ then taking the logarithm on both sides and multiplying by a negative we get - $-log(R(c_1, c_2)) - log(R(c_2, c_3)) \cdots - log(R(c_k, c_1)) < 0$ which is precisely the weight of the cycle $c_1, c_2 \dots c_k, c_1$.

Thus we need to detect a negative cycle in the graph G which is equivalent to finding a positive gain cycle (as a number if greater than one iff its logarithm is greater than zero or the negative of its logarithm is less than zero). The algorithm used will be a modification of the Bellman Ford algorithm and is given now. Now, we give the proof of correctness of this algorithm -

Algorithm 2: Negative Cycle in Directed Graph

```
$D, P$ are arrays of the vertices in $G = (C, E)$
Let $s$ be a specific vertex in $G$
for each $c \in C$ do
    $D[c] = \infty$
    $P[c] = NULL$
end
$D[s] = 0$ for $x$ = 1 to n-1 do
    for each $(c_i, c_j) \in E$ do
        if ($D[c_j] > D[c_i] + wt(c_i, c_j)$) then
            $D[c_j] > D[c_i] + wt(c_i, c_j)$
            $P[c_j] = c_i$
        end
    end
end
if ($\exists(c_i, c_j) \in E$ satisfying $D[c_j] > D[c_i] + wt(c_i, c_j)$) then
    Return 'Negative Edge Found'
end
Return $D, P$
```

Proof of Correctness

We have to prove that if the algorithm finds a vertex x such that $D[x] > D[y] + wt(y, x)$ in the last step then a negative cycle exists and the converse of this.

Part 1: If there are no negative cycles in the graph, then there will not be any vertex x such that $D[x] > D[y] + wt(y, x)$ in the last step.
Proof: If there are no negative cycles in the graph, then we know that after the $n - 1$ iterations of the loop , $D[v]$ stores the weight of the shortest path from s to v for all vertices v (proof of correctness of the Bellman Ford algorithm done in class). So, if in the n^{th} iteration there is a vertex x such that $D[x] > D[y] + wt(y, x)$ then from s there is a path to y to which when the edge (y, x) is appended gives a path from s to x which is shorter than the path found after $n - 1$ iterations. This is a contradiction as it contradicts the fact that Bellman Ford in the absence of negative cycles gives the shortest paths from the source after $n - 1$ iterations.

Part 2: If there is a negative cycle, then there will be a vertex x such that $D[x] > D[y] + wt(y, x)$ in the last step.
Proof: Assume that $v_1, v_2, \dots v_k, v_1$ is a negative cycle in the graph. Assume for the sake of contradiction that there is no vertex x for which the value of $D[x]$ can be decreased in the last step. Then for all the vertices in the negative cycle, the value of D cannot be decreases in the last step. This means that -

$$D[v_1] \leq D[v_2] + wt(v_1, v_2)$$
$$D[v_2] \leq D[v_3] + wt(v_2, v_3)$$
$$\dots$$
$$D[v_k] \leq D[v_1] + wt(v_k, v_1)$$

Adding these inequalities, we get that -

$$0 \leq \Sigma_{i=1}^{k} wt(v_i, v_i + 1)$$

This is a contradiction as we knew that these vertices formed a negative cycle. Hence, by contradiction we have proved that if there is a negative cycle, then at least one update will be present in the n^{th} iteration.

Hence, the correctness of the algorithm follows.

Time Complexity Analysis

The loop runs for $\mathcal{O}(n)$ iterations and in every iteration each edge is considered, hence the total time complexity of this algorithm is $\mathcal{O}(mn)$ or $\mathcal{O}(n^3)$ as $m \in \mathcal{O}(n^2)$.

Part 2

To do this part, we have to find a negative cycle in G and for that we use dynamic programming. Again we do choose a special vertex s and we create a matrix of size $(n+1) \times n$ where n is the number of vertices in the graph. Let the elements on the horizontal axis be indexed by the vertices of the graph and the elements on the vertical axis be indexed by numbers from 0 to n. Let the matrix be called DP. We will build the matrix such that $DP[v, i]$ will represent the min cost of an $s - v$ path using at most i edges. If no such path exists, we will put $DP[v, i] = \infty$

Using this definition, we can say that the first row of the matrix can be initialized as follows -

$DP[s, 0] = 0$ and $DP[v, 0] = \infty$ for all vertices $v \neq s$.

Also, we can split the general case of $DP[v, i]$ into two cases to build a recursion on which we can use the DP matrix -

Case 1: The min cost of an $s - v$ path using at most i edges has strictly less than i edges. In this case we can say that $DP[v, i] = DP[v, i-1]$.

Case 2: The min cost of an $s - v$ path using at most i edges has exactly i edges. In this case let's suppose that the vertex just preceding v on this $s - v$ path was w. Then the subpath of this path from s to w must be the minimum cost path from s to w containing at most $i - 1$ edges (if this was not we could find a shorter one by taking that path and appending the edge (w, v) to it). So, if we consider all the incoming neighbours of v (vertices which have an edge towards v) and take the minimum of these as well as Case 1 then we can find $DP[v, i]$ recursively.

Formally, we can write -

$$DP[v, i] = min(DP[v, i-1], min_{w \in N(v)}(DP[w, i-1] + wt(w, v)))$$

Where $N(v)$ denotes the vertices which have an edge directed towards v. We can see that to compute a particular row of the DP matrix, we just need the row above it and hence we can use dynamic programming effectively compute the entire matrix. Now, we present two claims -

Claim 1

If the graph contains no negative cycles, then $DP[v, n] = DP[v, n-1]$ for all vertices v.

Proof

As the graph has no negative cycles, the shortest path overall from s to any vertex v will contain at most $n - 1$ edges. This is because if the path contained more than these many edges, then some vertex will have to repeat on the path and hence it will contain a cycle. As this cycle will have non-negative weight, we can simply remove this from the path and get another path using lesser number of vertices and with the same or less weight. Thus we know that the shortest path overall from s to v will contain at most $n-1$ edges, and hence the shortest path from s to v containing at most n edges ($DP[v,n]$) will only have at most $n-1$ edges and hence be equal to $DP[v,n-1]$. Hence, proved.

Claim 2

If the graph has negative cycles, then $DP[v,n] \neq DP[v,n-1]$ for some vertex v.

Proof

The proof of this is similar to the one we did in part 1 of this question. Let $v_1, v_2, \dots v_k, v_1$ be a negative weight cycle. Then by the recurrence of dynamic programming we can say that -

$$DP[v_1,n] \leq DP[v_k,n-1] + wt(v_k,v_1)$$
$$DP[v_2,n] \leq D[v_1,n-1] + wt(v_1,v_2)$$
$$\dots$$
$$D[v_k,n] \leq D[v_{k-1},n-1] + wt(v_{k-1},v_k)$$

This was because by the recurrence $DP[v,n]$ is less than or equal to $DP[w,n-1] + wt(w,v)$ for all incoming neighbours of v and we have only taken a few specific ones.
Adding the above inequalities, we get that -

$$0 \leq \Sigma_{i=1}^{k} wt(v_i, v_i+1)$$

This is a contradiction as the cycle was assumed to be of negative weight. Hence, proved.

From the above two claims, we have shown that the graph contains a cycle if and only if for some vertex v in the graph $DP[v,n] \neq DP[v,n-1]$.

Using the above developed logic we can detect a negative cycle after computing the DP matrix (using the claims proved). Further, we can also find the negative cycle using the DP matrix as follows - First observe that if $DP[v,n-1] \neq DP[v,n]$ for some vertex v, then the shortest path from s to v containing at most n edges will have exactly n edges. This is because if this was not the case then $DP[v,n]$ and $DP[v,n-1]$ would have been the same. So, the first thing to do after computing the matrix is to find such a vertex v.
Now, as this path contains n edges, it must have $n+1$ vertices. By the Pigeon-Hole Principle, as there are only n vertices, at least two of the $n+1$ vertices must be the same. This implies that there is cycle in the shortest path from s to v containing at most n edges. Also, we can claim that this cycle will be a

negative cycle as otherwise the path would not be the shortest. Now, after creating the matrix DP and finding a vertex v such that $DP[v, n] \neq DP[v, n-1]$, we can start backtracking from $DP[v, n]$ to find the shortest path from s to v containing at most n edges.
Let this path be $s = v_0, v_1 \dots v_{n+1} = v$. Then we can say that for any vertex v_i, $DP[v_i, i-1] \neq DP[v_i, i]$. This is because if this was the case then the path would have less than n edges.
So, starting from $DP[v, n]$, consider all the incoming neighbours of v (call such a neighbour w) such that the quantity $DP[w, n-1] + wt(w, v)$ is minimized. This will be the vertex v_n of the path. Similarly, repeat this procedure for v_n to get v_{n-1} and repeat until we get to $s = v_0$.
After finding all the vertices, there is a guarantee that for two indices i and j, $v_i = v_j$. So, we simply return the set of vertices $\{v_i, v_{i+1}, \dots v_{j-1}, v_j = v_i\}$ which as we have shown above is a negative cycle. This is the final output of the algorithm.
The proof of correctness of this algorithm follows from all the claims we have used and proved above.

Time Complexity Analysis

The time taken to compute a row of the matrix DP is proportional to the sum of all the incoming degrees of the vertices, as we have to take the minimum over all incoming neighbours for each vertex. Thus, to compute one row, the time taken is $\mathcal{O}(\Sigma_v(deg(v)) = \mathcal{O}(m)$. So, for $\mathcal{O}(n)$ rows, the time taken would be $\mathcal{O}(mn)$. This is the time to compute the matrix.
For the backtracking path, we have to spend $\mathcal{O}(deg(v)) = \mathcal{O}(m)$ time on a single vertex of that path, and as there are at $\mathcal{O}(n)$ vertices in the path the total time taken for the backtracking is also $\mathcal{O}(mn)$.
Thus, for the overall algorithm, the time complexity is $\mathcal{O}(mn)$ or $\mathcal{O}(n^3)$ as $m \in \mathcal{O}(n^2)$.

3 Disjoint Collection of Paths

Question 3 : Let T be a binary tree, and S be a set of paths in the tree. Two paths P, Q $\in$ S are said to be disjoint if they do not have a common edge. Give an efficient algorithm which finds a maximum cardinality subset of paths such that each pair of paths in this subset is disjoint.

Solution :
We are given a binary tree T and a set of paths S in tree where S = $\{P_1, P_2, ...P_k\}$ (i.e. $|S| = k$) where P_i is a path in binary tree T and we need to find the maximum cardinality subset of paths such that each pair of paths in that set are disjoint (i.e. there is no common edge in any 2 paths). Now in order to solve this problem we will break the given problem in terms of its sub problems where each sub problem is the answer of maximum cardinality set on some sub tree of T. The following is the algorithm for the same.
Algorithm : Before writing the complete algorithm we will try to write a helper function which will help us break the problem in terms of its sub problem on some sub tree of T. Let's first describe the helper function below.

```
Function max_cardinal(T,S,children_type,memoize) {
    if((T,children_type) in memoize) {
        return memoize[(T,children_type)];
    }
    root_set = {}
    for i in range(length(S)){
        if ($P_i$ contains root(T) and Lies completely inside T with children_type)
        {
            root_set.insert($P_i$);
        }
    }
    if (root_set = empty or T is empty) return {};
    max_set = {}
    for i in range(0,length(root_set)){
        $Path_i = v_0 v_1 .... v_k$; // path in root_set
        temp_set = {$Path_i$};
        for j in range(0,k) {
            if (edge($v_j$,$v_{j+1}$) is such that $v_j$ is parent of $v_{j+1}$ in tree T and it is a left edge) {
                temp_set = temp_set ∪ max_cardinal(T',S, right,memoize);
                // Here T' is tree with root node $v_j$
            }
            else if (edge($v_j$,$v_{j+1}$) is such that $v_j$ is parent of $v_{j+1}$ in tree T and it is a right edge) {
                temp_set = temp_set ∪ max_cardinal(T',S, left,memoize);
                // Here T' is tree with root node $v_j$
            }
            else if (edge($v_j$,$v_{j+1}$) is such that $v_{j+1}$ is parent of $v_j$ in tree T and it is a left edge) {
                temp_set = temp_set ∪ max_cardinal(T',S, right,memoize);
                // Here T' is tree with root node $v_{j+1}$
            }
            else if (edge($v_j$,$v_{j+1}$) is such that $v_{j+1}$ is parent of $v_j$ in tree T and it is a right edge) {
                temp_set = temp_set ∪ max_cardinal(T',S, left,memoize);
                // Here T' is tree with root node $v_{j+1}$
            }
        }
        at end points of path do temp_set = temp_set ∪ max_cardinal(T',S,both,memoize)
;
        // Here both means that both the subtrees should be considered for the root(T') equal to end points of path
        max_set = maximum(max_set, temp_set);
    }
    max_set = maximum(max_set, max_cardinal($T_{left}$,S,both,memoize) ∪ max_cardinal($T_{right}$,S,both,memoize));
    // Here in left and right subtrees keep in mind the children_type
    memoize[(T,children_type)] = max_set;
    return max_set;
}
```

Now we will explain what the above helper function is doing and define the meaning of children_type.

Explanation of Helper Function :

The helper function max_cardinal is used to calculate and give the maximum cardinallity disjoint subset of paths in a given tree T with a specific children type(here by children type we mean that if it is "left" then that means that we will consider that portion of tree T with the same root as T along with its left sub tree and will exclude the right sub tree portion of T and similarly we can have "right" and "both" here both means that all of the tree T is taken while evaluating the maximum cardinal subset i.e. all the paths of max subset belong to that tree).

Verbal explanation of the helper function algorithm is as follow :

We first check if the answer to the given tree with the corresponding children_type is evaluated already or not (we are using memoization to store the already evaluated result). After this if we do not find the result then will iterate over all the the paths in S and check which of them lie in the corresponding T portion and also pass through its root.**(Note that for easily checking this we can do DFS on that portion of tree and get all the vertices that are inside that tree portion and then check if the given path P_i contains any vertex other then those or not)**. Once we have all such paths then we can easily iterate over all these filtered paths and try to generate a maximum cardinallity subset that contains that given path $Path_i$ and finally after iterating through all such paths (and also taking the case where no such path is there in the maximum cardinallity subset) we take the subset with maximum cardinallity and report it as our answer. In order to generate the maximum cardinallity subset of paths containing a given path $Path_i$, we evaluate the maximum disjoint cardinallity set on all the subtrees of T with root in path $Path_i$ with such a children type such that we don't have any edge common with the path $Path_i$ and take their union.

Final Algorithm :
Step 1 : memoize = {};
Step 2 : return max_cardinal(T,S,both,memoize);
We have evaluated our maximum cardinallity subset of disjoint paths in T using the help of memoization(storing the already evaluated answers for a given tree root node T' and children type). Now we will move on to the proof of correctness of the above algorithm along with the proof of correctness of helper function.

Proof of Correctness :
In order to prove that the given algorithm computes the correct required result, we will first prove the correctness of the helper function max_cardinal.

Claim 1 : In order to find the the maximum cardinallity subset of disjoint paths using max_cardinal function of a given tree T portion with a given children type which contain a path P_i present in the tree T with children type and passing through its root node, we can say that it is equal to the union of all the maximum cardinallity disjoint paths subset evaluated at each sub tree of T whose root node lies on path P_i and no edge lies on the path P_i along with the path P_i and the net maximum cardinallity disjoint subset is found after traversing over and picking the max over all such paths P_i(along with the case without taking any such P_i) .

Solution : In order to prove this claim which we will apply **induction** on the number of vertices in tree T.

Base Case : When the number of vertices in the Tree T is equal to 1 then since no path will lie on the Tree T hence the maximum cardinallity subset with disjoint paths will be the empty set. Hence the base case is true.
Induction Hypothesis : The given claim 1 is true for the number of vertices < n in tree T.
Induction : If we have the path P_i satisfying the above constraints in the claim 1 and the number of vertices in Tree T portion (with a given children type) is = n. Let's say the path P_i is given by $v_0v_1v_2.....v_l$ then we can say that if we look at all the sub trees of T with root rooted inside the path P_i (containing the portion corresponding to the given children type of T) and not including any edges in path P_i, then all such sub trees will be disjoint and won't have any common edge(this can be proved easily since if did had some common edge then in order to go from one such sub tree to the other sub tree, we would have to go via some edge in path P_i since the path connecting their parents is in P_i. Hence we can say that all the maximum cardinallity subsets of disjoint paths for all these sub trees will be disjoint (since these trees already don't have any common edge). Also since all the paths in the maximum cardinallity set of tree T which

contain path P_i will be disjoint hence they will be part of some of these sub trees. So we can say that in order to maximize the number of such paths we can take the union of all maximum cardinallity subsets of these sub trees (since these sub trees are disjoint and independent of each other) along with the path P_i. Now by the induction hypothesis we can clearly see that since the number of vertices in these sub trees would be < n so all the subset generated using above claim would be maximum cardinal disjoint subsets , hence the subset we get using this approach for the tree T will be the maximum cardinallity subset with disjoint paths containing the path P_i. Now if we traverse over all such P_i(along with the case of no such P_i taken in which we just look at the left and right sub trees) and take the maximum then we can say that since at least 1 such P_i or none of these P_i would be a part of solution set then we can say that since this is exhaustive hence the answer we get using this approach will be the maximum cardinal subset with disjoint paths. Hence proved.

Proof of Correctness of Final Algorithm :
Since in the final algorithm we are calling the max_cardinal function on the tree T with both of its children which will consider the whole tree T while evaluating the answer then clearly by claim 1 we can say that the result obtained by this algorithm is indeed the maximum cardinallity subset with disjoint paths.

Time Complexity Analysis :
Let's say that the number of paths in S is K and the

number of vertices in tree T is n. Since we are storing all the evaluated results when calling on the max_val function on the sub trees in a memoized manner hence we can say that each node is processed at max of 3 times (when children type is left or right or both).
$\Rightarrow$ Time to process a single node to filter the paths $= O(n * K)$ (as initially we iterate over all the K paths and check if path is in tree T and pass through vertex or not(using DFS))
$\Rightarrow$ Time to process a given filtered path set for a given tree T root node $= O(n * K * n)$ (as we iterate over all the vertices of a given path P_i and generate a maximum cardinallity subset (the size at max will be n-1 since if we take all the edges as paths P_j's in max set then its length will be n -1 at max) and we do this K times in worst case.
$\Rightarrow$ Since we are doing the same thing for all the nodes of tree T hence the total time complexity can be written as : $O(n * (n * K * n + n * K))$
$\Rightarrow$ Time Complexity $= O(n^3K)$ Time. Now as K is $O(n^2)$
$\Rightarrow$ **Final Time Complexity** $= O(n^5)$ which is polynomial time in n (the number of vertices in tree T).