The Structured Flow Tree

A structured, high-performance model for marketing automation workflows

Abstract

Workflow automation systems are commonly modeled using linear sequences or directed acyclic graphs (DAGs), both of which present tradeoffs in expressiveness, performance, and usability. Linear models lack the ability to represent branching logic, while DAG-based systems introduce structural and computational complexity, particularly for traversal, mutation, and visualization.

This paper introduces the **Structured Flow Tree (SFT)**, a constrained, tree-based representation of workflow execution that encodes branching and merging behavior without explicit edge management. The SFT models workflows using a hierarchy of triggers, actions, logic nodes, and structural branches, enabling deterministic construction and intuitive manipulation while preserving the expressive power of a restricted class of series-parallel graphs.

We present a method for flattening the SFT into an ordered representation using metadata coordinates (branch, order, level), allowing efficient storage and reconstruction without loss of structural information. Building on this representation, we define a traversal model that enables reachability checks in O(1) time for most cases and O(D) in the worst case, where D is the depth of the tree—significantly improving upon traditional graph traversal approaches.

The SFT is designed with practical constraints in mind, including forward-only execution, automatic node edge inference, and minimal-impact mutation operations such as node insertion, movement, and deletion. These constraints enable a simpler and more predictable user experience while maintaining computational efficiency at scale.

We demonstrate that by constraining workflow structure and encoding relationships implicitly through hierarchy and metadata, the Structured Flow Tree provides a viable alternative to graph-based workflow models, offering improved performance, simpler implementation, and better alignment with real-world automation use cases.

Introduction

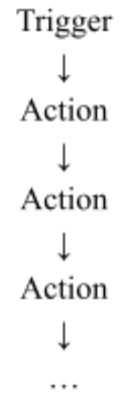
Groundhogg's original flow editor was **linear,** composed of only trigger and action nodes (no logic nodes yet) in sequence. With linear flows, there were limited logic capabilities and no "branching".

An example flow would look like an ordered list of Trigger and Action nodes.

```
[ Trigger, Action, Action, Action… ]
```

Visually represented vertically in a waterfall, as in *figure 1.* These nodes would be stored flat in a table in the database with order (index+1) preserved so we can rebuild the flow visually.

Figure 1.



Traversing the flow node by node meant simply doing `getNodeByOrder(current.order + 1)`. Testing whether traversability between two nodes P and Q meant testing `P.order < Q.order` thus the time and space complexity of `hasPath(P,Q)` is O(1) in this case.

Adding branches and logic nodes to this existing structure is not obvious. Ideally we'd like to…

- Make minimal changes to the existing database structure, so the nodes can be parsed and presented as a flat array
- Keep development complexity to a minimum
- Preserve structure of existing flows in use in the wild
- Keep time and space complexity small

From an end user editing experience, linear flows are also quite straightforward because moving nodes in the flow is drag & drop and moving a node does not impact the nodes before or after it. Adding nodes is also easy because you can simply insert a node into the array of nodes at any desired position.

So we also need to preserve…

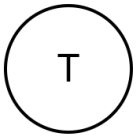
- Ease of moving and reordering nodes within a flow
- Ease of adding and deleting nodes in a flow
- Connection between nodes is automatic depending on adjacency

Establishing constraints for flows and traversal

Before we look at possible models we should establish some constraints about how we expect flows and traversal to work.

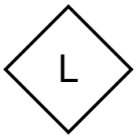
Types of nodes

We must establish the types of nodes we'll be working with and our constraints for them. Our goal is to create/determine the best model to represent these nodes in a flat structure while persevering their shape (visual representation in 2D space).



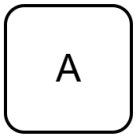
Triggers

A trigger node is an entry or jump point in a flow and can have a single descendent connection to another non-trigger node. Adjacent triggers operate as an OR condition.



Logic

A logic node can branch out to an arbitrary number of descendent nodes.



Actions

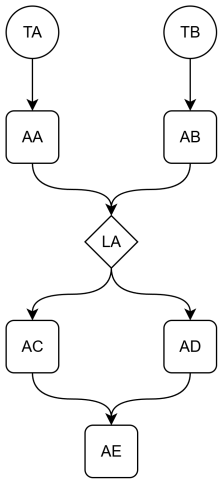
An action node can be connected to one other descendent node.

You can only travel “forward” in a flow

Taking from our existing structure, you can only travel from any node P to a node Q if Q is “after” P. This establishes that we can only “move forward” in a flow.

Also, if traveling from nodes P → Q then every node in the path of P → Q should also be “after” P and “before” Q, and every node N should be “before” N+1 and should be “after” N-1.

Figure 2.



Example

For example, in this flow there is **no valid path** AA → AB, as the path would be AA → LA → AB and LA is before AB.

AC → AD would also be invalid, since it would require going through LA, being before AC, or AE, which is after AD.

AA → AD is valid, as is AB → AC, and TB → AE, as each node in between is “between” P and Q.

This gives us the need to come up with functions `isAfter(P, Q)` and `isBefore(P, Q)`.

Adding nodes

Adding a node from A → B in the flow should be intuitive and make sense given the previous constraints for nodes.

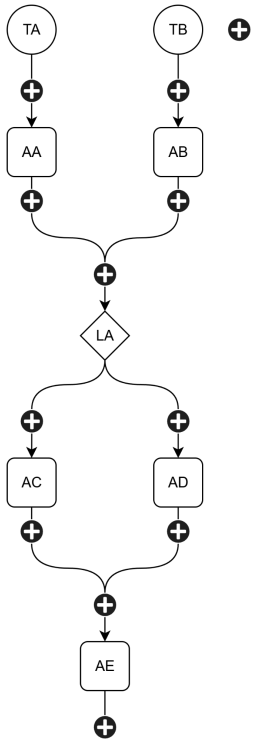
Looking at *figure 2,* we can add nodes **before** AA, AB, LA, AC, AD, and AE. We can add nodes **after** TA, TB, AA, AB, AC, AD, AE. Where there is an overlap of insertion points after a node and before the next node, we can say adding a node *between* those two nodes.

You can also add a Trigger node *next* to TB, for the purpose of adding an additional entry point, and if you were to add TC next to TB it would be connected to LA such that TC → LA.

When adding a node at an insertion point, the valid connections between adjacent nodes should be automatically taken care of and not require additional user input.

See an example of valid insertion points in *figure 3.* All insertion points meet the restrictions of the node types surrounding them.

Figure 3.



Moving Nodes

Moving a node from position A → B in the flow should be intuitive for the end user in a drag and drop interface, and ideally intuitive for the model as well.

You should be able to move an existing node to any valid insertion point where you would otherwise add a node.

Figure 4.

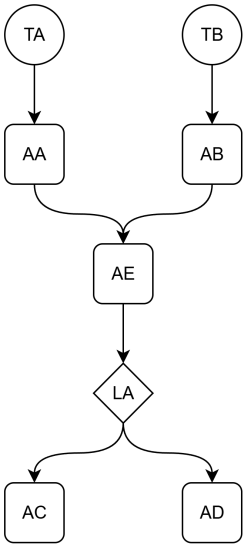
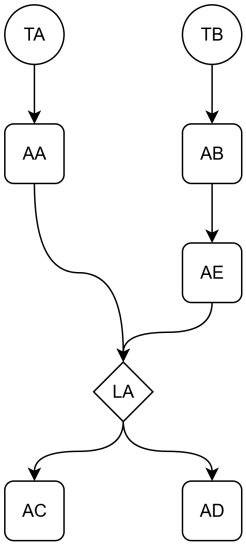


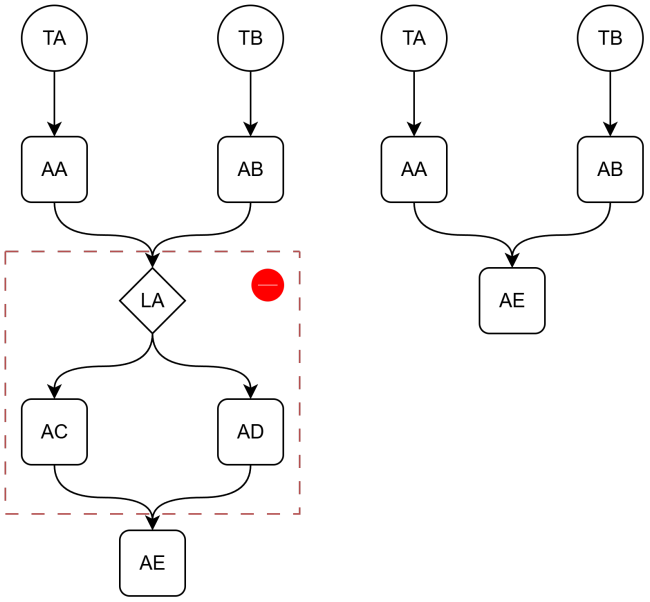
Figure 5.



Again looking at *figure 3,* if we want to move LA to after AE, such that there is a connection of LA → AE. If we move *just* LA, nodes AC and AD don't have any obvious valid connections. Thus when moving LA, it should also move AC and AD (its "sub-flow") with it. The resulting flow will appear as in *figure 4.*

If we want to move AE to after AB, the connections of AC → AE and AD → AE must also be removed and AB → LA becomes AB → AE → LA as shown in *figure 5*.

Figure 6.



Deleting nodes

Deleting a node should do *minimal damage* to other nodes, meaning preserve the most possible other nodes.

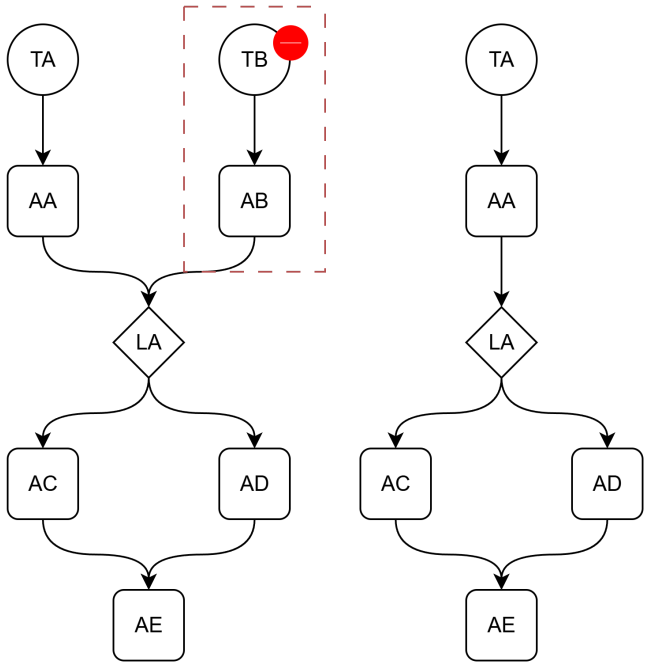
Looking at *figure 3* again, if we delete the node LA from the flow, doing the least damage would mean deleting any other nodes that would no longer have valid connections, which happens to be LA's

sub-flow (AC and AD). But, isn't AE a child of AC and AD? Yes, but if LA, AC and AD are deleted there is a valid connection of AA and AB to AE. See *figure 6.*

So the idea is to delete the minimum number of nodes that would not have a valid connection if the parent node is deleted.

Deleting TB would delete AB also, but not any other nodes, as shown in *figure 7.*

Figure 7.



Moving and **deleting** nodes necessitates the need to determine any node's minimum "sub-flow".

Investigations

We looked into a couple different implementation options.

Linked list implementation

Linked lists were deemed insufficient because they only really know about the immediate previous node (singular) and the immediate next node (singular) so branching isn't actually possible since a linked list is a more complex ordered list.

Directed graph implementation

Our flows look **a lot** like a directed graph, and on the surface it seems like a reasonable solution to our problem. The data structure of a directed graph has each node aware of its immediate parent node(s) and its immediate child node(s), such that a flat array of nodes in a directed graph would appear as…

Figure 8.

```
[
  { id: TA, parents: [], children: [AA] },
  { id: AA, parents: [TA], children: [LA] },
  { id: TB, parents: [], children: [AB] ,
  { id: AB, parents: [TB], children: [LA]},
  { id: LA, parents: [AA, AB], children: [AC, AD] },
  …
]
```

However there are severe complexities to overcome when using this data structure.

Directed graph theory is already quite robust, a quick Google shows that to determine if there exists a valid path P → Q (based on our constraints) is of time complexity O(V+E) where V is the number of vertices (nodes) and

E the number of edges (connections to other nodes). That's **slow**. We expect end users will have significantly large datasets, thus this time complexity essentially eliminates directed graphs as an option.

There are other challenges as well.

Adding a node into a directed graph structure is simple enough, but what about deleting a node or moving nodes around? To find the minimum sub-flow of a node when deleting requires DFS/BFS and has time complexity $O(k(V + E))$ where k is the number of edges from the node being deleted. Again, **slow.**

You also require an algorithm to visually arrange the nodes in a visually appealing way in 2D space. There are many libraries that provide such a solution on GitHub, but after investigating they simply weren't sufficient to provide an interactive editing experience.

Altogether we spent way too much time going down the directed graph rabbit hole.

Searching the web

Searching things like "best data structure/model for workflow automation" did not reveal any guides or libraries that provided an explicit answer to our questions. Additionally, no one appears to have formally written about or open-sourced a solution to the set of constraints that we have.

Competitor analysis

Looking at existing "automation flow builders" in the wild we see that many of them opted to use a directed graph and design complex optimizations around it, which *we could do* but felt that there must be a better way.

Competitors often opt to have customers arrange nodes and connections **manually** in a canvas using a coordinate system to place nodes visually. We want to avoid this to keep things simple.

The DOM Tree

Ultimately the editor and visual display of flows will be in HTML in our application, so whatever data structure we choose should lend itself to being rendered out in HTML. Consider for a moment the HTML DOM tree, it is a rooted (under `<document>`), **ordered**, tree data structure that is n-ary meaning a node can have any number of children. It also has well defined node types, as do we!

One of the great things about the DOM tree is if you delete a node, it deletes its sub-tree without impacting the rest of the tree. If you move a node, it takes its sub-tree with it. And adding to the DOM tree is a simple operation, this meets a lot of our constraints.

Thinking about XML...

Running with the HTML DOM for a moment, let's abstract away HTML and use an XML like structure for a moment which is also a rooted, ordered, n-ary tree structure. Let's call it a "**Flow Tree"**.

Consider for a moment how a **linear** flow (as from *figure 1*) would look like in this format. It would look something like…

Figure 9.

```
<Flow>
  <Trigger/>
  <Action/>
  <Action/>
  <Action/>
  <Action/>
</Flow>
```

So how do we go from linear to the directed graph-like representation of flows like in *figure 2*? Let's again think about the HTML DOM tree and its node types for a moment.

Consider the `<select>` node type. It can only contain `<option>` nodes, and `<input/>` nodes can have no children. `<p>` can only have other text type nodes like `<span>`, `<em>`, `<strong>`, etc…

Thus we should define the constraints for Trigger, Action, and Logic nodes in an XML like tree structure in terms of if they can have children, and if they do, limitations on the types of nodes that can be children.

Option a) Any node's next node(s) is its child(ren).

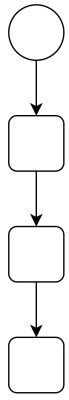
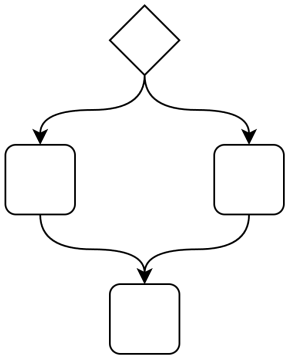


Figure 10.

```
<Flow>
  <Trigger>
    <Action>
      <Action>
        <Action/>
      </Action>
    </Action>
  </Trigger>
</Flow>
```

You might think that *nesting* a flow tree in a way where each node's next node is its child makes sense. A simple flow of Trigger → Action → Action → Action would look something like in *figure 10.*

Figure 11.



Traversal would be simple enough to, `nextNode(P)` would simply be `P.children[i]`. But consider another flow that has branching to multiple nodes, and then merges back into another node, such as in *figure 11.* How could you represent that in a nested tree? The last node would need to be a child of the two nodes that came before it, duplicating itself in the Flow Tree.

Over a large enough flow, you could have dozens of duplicate node paths every time there's a merge in the flow.

What we've done here accidentally is created another Directed graph, only worse.

How would you even represent this in HTML (which we ultimately need to)? We'd need a complex algorithm to merge duplicates into an HTML friendly tree structure… So why don't we figure out what *that* structure looks like instead?

Option b) Any node's next node is its sibling

Let's look back at *figure 9* where `nextNode(P)` is simply `P.nextSibling`. We're technically doing this in our current product, so it makes sense for us to continue doing it. But what happens with a logic node with 2 children like in *figure 11*? If the two `<Action>` nodes are siblings of the `<Logic>` node, how do we know which one to traverse? See *figure 12.* Obviously this is insufficient.

Figure 12.

```
<Flow>
  <Logic/>
  <Action/>
  <Action/>
  <Action/>
</Flow>
```

Option c) Any node's next node(s) is its child(ren) **or** sibling

What happens if we combine siblings **and** children together? We must set up constraints for our node types in order to successfully define the shape of our flow tree.

Intuitively we can see that whenever a node **branches** we need a way to know what the possible next nodes are, and we need a way to identify wherever we **merge** (two or more nodes pointing to the same node) so that we don't get duplicate paths in the tree.

So we can say that a node should contain all its descendent nodes **up until there is a merge.** If we set that constraint *figure 12* becomes *figure 13.*

Figure 13.

```
<Flow>
  <Logic>
    <Action/>
    <Action/>
  </Logic>
  <Action/>
</Flow>
```

We're onto something!

Defining connections (edges) between nodes

With our basic rule, "a node should contain all its descendent nodes **up until there is a merge,**" we can start to define how two nodes become *connected*. One of our original conditions for a solution was that connections between nodes are automatic based on the.

Looking at *figure 11* we can see that the logic node has connections to its two children, and those children are connected to the last node. Now looking at *figure 13*, we can say that a node P with children has connections (edges) to its children, and P's children have connections to P's next sibling.

If P does **not** have any children, then it is connected to its next sibling.

Let's have a look at a more complex example in *figure 14.* Based on our definitions so far, *figure 14's* tree would appear as in *figure 14a.*

Figure 14.

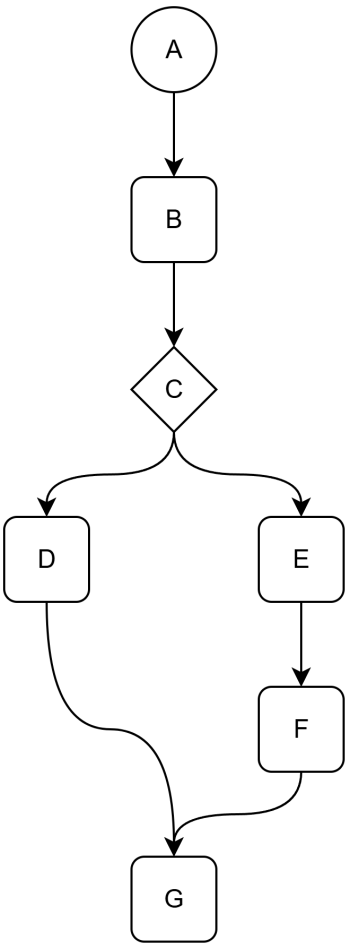


Figure 14a.

```
<Flow>
  <Trigger id="A"/>
  <Action id="B"/>
  <Logic id="C">
    <Action id="D"/>
    <Action id="E"/>
    <Action id="F"/>
  </Logic>
  <Action id="G"/>
</Flow>
```

But that would imply C has **direct** connections to D, E, and F, which in *figure 14* it clearly does not. It would also imply that E has a connection to G, which again, it does not. So our definition that "a node P with children has connections (edges) to its children, and P's children have connections to P's next sibling," is insufficient to describe the shape of a flow.

We need a way to describe in our flow tree that C has connections to D and E, but not F, and that D and F and connections to G, but not E.

Examining *figure 14* we can see that if E and F were a single node, "EF," *that* would satisfy our definition. So let's describe a type of pseudo node that would create that behaviour.

Image a new node type called **Branch.** A Branch's children can be any number of Action, Logic, and Trigger nodes (but not other Branch nodes). A Branch node must also be the child of a Logic or Trigger node, like an `<option>` can't exist outside of a `<select>`.

Triggers can have up to one branch, and Logic nodes can have up to N branches.

With the addition of the `<Branch>` node our flow tree is organised as in *figure 14b*, with D in a branch and E and F in a branch.

We must now update our definition to, "a node P has a connection to the first node of each child branch. The last child of a branch belonging to P has a connection to P's next sibling."

So far our working definition of a connection between two nodes is (examples referencing *figure 14b*)…

Figure 14b.

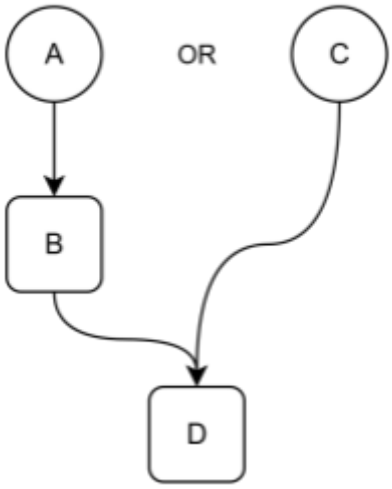
```
<Flow>
  <Trigger id="A"/>
  <Action id="B"/>
  <Logic id="C">
    <Branch>
      <Action id="D"/>
    </Branch>
    <Branch>
      <Action id="E"/>
      <Action id="F"/>
    </Branch>
  </Logic>
  <Action id="G"/>
</Flow>
```

1. If a node has no branches, it is connected to its next sibling (if one exists)
   Ex: A → B, E → F, B → C
2. If a node has branches, it is connected to the first node of each branch
   Ex: C → D, C → E
3. If the node has no next sibling, and is in a branch, it is connected to the branch parent's next sibling
   Ex: D → G, F → G

Caveats for trigger nodes

Triggers have some special caveats that must be addressed. Triggers *adjacent* to each other must operate as an OR condition, so that means you **can't** have Triggers that are connected to each other.

Figure 15.



In *figure 15* we have two adjacent triggers A and B which function as an OR, offering different entry points into the flow. In a flow tree it would look like…

Figure 15a.

```
<Flow>
  <Trigger id="A">
    <Branch>
      <Action id="B"/>
    </Branch>
  </Trigger>
  <Trigger id="C"/>
  <Action id="D"/>
</Flow>
```

Based on definition 3, B would connect to C since C is A's next sibling. So that doesn't work. There are two solutions to this problem.

Option a) Use <TriggerGroup>

Define another node type called `<TriggerGroup>` which can only contain triggers. A TriggerGroup node can only be placed in a `<Branch>` or the root `<Flow>` node, and we must modify our definition of `<Trigger>` that a <Trigger> can only be placed in a `<TriggerGroup>`. You can also not have a `<TriggerGroup>` adjacent to another `<TriggerGroup>`. If adding a Trigger into a flow, it's added to the closest adjacent TriggerGroup and if one does not exist, one is created and it's added to that one.

Then we modify our definitions slightly with additional caveats for TriggerGroups.

Figure 15b.

```
<Flow>
  <TriggerGroup>
    <Trigger id="A">
      <Branch>
        <Action id="B"/>
      </Branch>
    </Trigger>
    <Trigger id="C"/>
  </TriggerGroup>
  <Action id="D"/>
</Flow>
```

1. If a node has no branches, it is connected to its next sibling (if one exists)
    a. Unless the node is a Trigger, in which case it is connected to its parent TriggerGroup's next sibling
2. If a node has branches, it is connected to the first node of each branch
3. If the node has no next sibling, and is in a branch, it is connected to the branch parent's next sibling
    a. Unless the branch parent is a Trigger, in which case it is connected to the TriggerGroup's next sibling.
4. If the next sibling is a TriggerGroup, then the node is connected to each of the Triggers in the TriggerGroup

This is *fine* and the definition is exhaustive*,* but a tad verbose. It would be much *cleaner* if we could avoid an additional node type.

Option b) Ignore adjacent triggers

Instead of using another node type, we could instead update our definitions to ignore adjacent triggers.

1. If a node has no branches, it is connected to its next sibling (if one exists)
    a. Unless the node is a Trigger, in which case it is connected to the first available non-trigger sibling.
2. If a node has branches, it is connected to the first node of each branch
3. If the node has no next sibling, and is in a branch, it is connected to the branch parent's next sibling
    a. Unless the branch parent is a Trigger, in which case it is connected to the Trigger's first available non-trigger sibling.
4. If the next sibling is a Trigger, then the node is connected to any adjacent Triggers as well

This option is cleaner, and is the one we implemented in our project.

> *When going from an SFT to HTML, we do end up wrapping adjacent Triggers in a TriggerGroup anyway since it's better for display and interactivity, but we unwrap them for storage and backend processing.*

This defines the **structure** of a flow tree, giving the model the name **Structured Flow Tree.**

Manipulating an SFT

Earlier we established ideal behaviour for adding, moving, and deleting nodes, the SFT satisfies those ideals beautifully.

Adding nodes

We showed in *figure 3* where we would like to see insertion points. Using a SFT we can establish rules for where insertion points should be.

1. Before any `<Action>` or `<Logic>` node
2. Before the **end/closing** of a `<Branch>` node (`</Branch>`). Including the root `<Flow>` node.

3. Before any group of adjacent `<Trigger>` nodes (or before a `<TriggerGroup>`), if and only if the group of triggers are not the first nodes in an SFT (order 1).
4. After any group of adjacent `<Trigger>` nodes (or at the end/close of a `<TriggerGroup>`)

After adding a node to any of the defined insertion points all connections will be automatically updated by virtue of their updated position relative to siblings within the tree.

Moving nodes

When moving nodes, if the node contains branches, it brings the whole sub-tree with it, and you can move a node to any insertion point. Connections are based on sibling adjacency, so connections are updated automatically without need for complex algorithms.

Deleting nodes

When deleting a node, it also deletes any nodes in its sub-tree, not affecting any sibling nodes and also automatically updating connections.

Flattening an SFT

Being able to describe the shape of a flow in a tree-like structure is incredibly useful, but it's not *that* useful when it comes to processing in real-world applications. In most cases applications will desire to store nodes as individual rows in a database table, basically a flat array structure. So how do we go from a tree to an array?

Well, arrays are ordered, as is our existing structure, so we need to assign an **order** to each node. We have two options to assign order, Depth First or Breadth First. **Depth first** makes the most sense because that's how nodes appear in the tree reading from top to bottom.

Imagine a 2D plane. There is an X axis and a Y axis, and a point on the plane is described by coordinates (x, y). The SFT is kind of like a 2D plane (in that it exists in 2D space), where order describes a coordinate $\boldsymbol{y}$, but what do we use to describe $\boldsymbol{x}$?

Let's describe the data structure we actually want to end up with. Remember that the solution should require minimal changes to our existing structure, which is that each node of a flow (only triggers and actions right now) has its own table row and order.

What's new is logic nodes, branches, and triggers containing up to one branch, so we need to somehow describe this information in the node metadata.

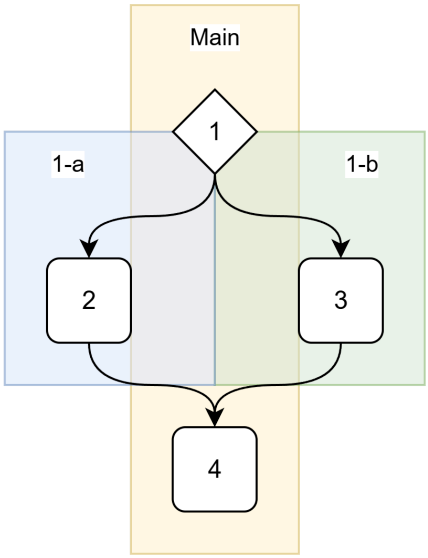
Flattening branches

When we flatten an SFT do we need to preserve `<Branch>` nodes in the end result as separate items, such that the result will be [Logic, Branch, Action, Branch, Action…]? Given that branches are not *really* functional but

are simply describing shape, we should find a way that our end result is just an array of functional nodes (Trigger, Action, Logic).

Notice how every functional node in an SFT is either under the root `<Flow>` node or under a `<Branch>` node. We could say that the root `<Flow>` node is really just a root `<Branch>` node. Thus, every functional node is in a `<Branch>` node.

Figure 16.



We can give every branch a unique identifier (like 1, 2, 3, 4… on an X axis) such that we can identify the container branch of a node P by checking P's metadata, like `P.branch`. We can be *especially clever* by prefixing the branch ID with the ID of the node the branch is descending from, that way we can always identify the functional parent node of a branch from a functional node within a descendent branch. We'll use *main* for the root branch.

In *figure 16* the logic node with ID 1 has two branches, which we'll simply call a and b. The ID of those branches will thus be $1\text{-}a$ and $1\text{-}b$ respectively. If our node P is node 2, we can get the container branch of node 2 by `getNodeById(2).branch` which would return $1\text{-}a$.

Likewise, `getNodeById(1).branch` would return *main*.

We can delegate any branching nodes (mainly logic nodes) to be responsible for keeping track of its branches in its metadata, `P.branches` would be an array of IDs, $[\ 1\text{-a}, 1\text{-b}, 1\text{-c...}]$. Since triggers can only have one branch, the ID of that branch can simply be the ID of the trigger itself.

Thus the branch in which any functional node exists is our x coordinate. Placement in an SFT is determined by coordinates $(branch, order)$.

If order is determined by DFS and every node stores the ID of the branch that contains it, a flattened tree of *figure 16* would appear as it does in *figure 16a.*

Figure 16a.

```
[
  { id: 1, order: 1, branch: 'main' },
  { id: 2, order: 2, branch: '1-a' },
  { id: 3, order: 3, branch: '1-b' },
  { id: 4, order: 4, branch: 'main' }
]
```

The algorithm to produce a flattened tree would look something like the following.

> *The following algorithms and function definitions are pseudo code and not fully complete. Some functions are referenced without provided definitions, like getNodeById(id). In those cases it is expected that you fill in the blanks with whatever storage API you plan on using for your project.*
>
> *The time complexity that we provide for each algorithm is based on the time complexity of getting any node by its ID or order within a flow is $O(1)$ constant time.*
>
> *If using a system like MySQL, you should index the order column so that direct DB queries are basically $O(1)$.*

```
/**
 * Given a root node or branch, return a flattened array representation of the tree
 *
 * @param node $branch the branch node
 *
 * @return node[] an array of nodes
 */
function flattenSFT( branch, order = 1 ){
  let nodeList = []
  let branchId = branch.id // 'main' or prefixed ID like `1-a`
  branch.children.forEach( node => {
    // convert our node into some kind of flattened representation of the node
    let flattenedNode = flattenNode( node ) // { id: foo, ... }
    flattenedNode.order = order // set the node order according to DFS
    flattenedNode.branch = branchId // set the branch
    // increment the order
    order++
    // add the flattened node to our array
    nodeList.append( flattenedNode )
    // if the node has children, they must be branches according to our shape definitions
    if ( node.children ){
      node.children.forEach( branch => {
        // flatten the branch, passing order to preserve DFS ordering
        let flattenedTree = flattenSFT( branch, order )
        // set the order to the order of the last node +1
        order = flattenedTree[flattenedTree.length-1].order + 1
        // merge the flattened branch tree into our main node list
        nodeList.append(...flattenedTree)
      } )
    }
  } )
  return nodeList
}

flattenSFT( FlowRoot ) // where FlowRoot is the root node of our flow tree
```

The time complexity of this is O(N), but we're not really worried about it since we only flatten the tree occasionally while editing.

To rebuild a tree from a flattened array is just as straightforward. The following algorithm is likely not as optimized as it could be, but works fine. Again, not super worried about time complexity since we do not intend to rebuild the tree frequently.

```
/**
 * Rebuild a flow tree from a flattened array
 *
 * @param array  $nodes  a flattened flow tree
 * @param string $branch the branch to start building the tree from
 */
function unflattenSFT( nodes, branch ){
  // make sure the node list is sorted by order ASC
```

```
  nodes.sort((a, b) => a.order - b.order)
  // filter nodes for the current branch from the node list ar
  let branchNodes = nodes.filter( node => node.branch === branch.id )
  // iterate over found nodes
  branchNodes.forEach( node => {
    // add the node to the branch
    branch.append( node )
    // if the node has branches
    if ( node.branches ){
      // iterate over the branch IDs
      node.branches.forEach( branchId => {
        // create a new branch node
        let branchNode = createBranchNode( branchId )
        // add that branch to the node
        node.append( branchNode)
        // recursively build the tree
        unflattenSFT( nodes, branchNode )
      } )
    }
  } )
}

// nodeList is our flattened tree
unflattenSFT( nodeList, createFlowRoot( 'main' ) )
```

The time complexity of building a tree with this function is O(DxN) where D is the depth of an SFT (maximum number of nested branches). The best way to optimize this further is a `filterSplice()` function that filters out branch nodes while removing them from the larger nodeList, which would reduce the time complexity to O(N) time. Not that it matters much as rebuilding an SFT happens infrequently, only during editing.

Traversing a flattened SFT

Traversing a workflow using `node.nextSibling` and `branch.children` is convenient when working with an in-memory tree during editing, but it is less practical for most backend applications. Backend systems typically store workflows as flat records in a database, process them in stateless jobs, and require fast random access to node relationships without first rebuilding the full tree in memory. Repeated recursive traversal and tree reconstruction introduce unnecessary overhead at execution time. For backend processing, it is more practical to traverse a flattened representation whose structural relationships can be inferred directly from node metadata.

getNextNode(P)

Earlier we established how nodes are connected earlier. They are…

1. If a node has no branches, it is connected to its next sibling (if one exists)
    a. Unless the node is a Trigger, in which case it is connected to the first available non-trigger sibling.

2. If a node has branches, it is connected to the first node of each branch
3. If the node has no next sibling, and is in a branch, it is connected to the branch parent's next sibling
    a. Unless the branch parent is a Trigger, in which case it is connected to the Trigger's first available non-trigger sibling.
4. If the next sibling is a Trigger, then the node is connected to any adjacent Triggers as well

We also established that you can only move "forward" in a flow, and that node position is described by coordinates $(branch, order)$. So a sibling node is any node within the same branch, and the next sibling node specifically is the first available node in the same branch of higher order.

Thus given any starting node and a flattened nodeList we can get the next node by with the following…

```
/**
 * Get the next node based on our traversal rules
 */
function getNextNode( curr, nodeList ){

  // might have been the last node
  if ( ! nodeList.length ){
    return null
  }

  // ensure node list is sorted by order ASC
  nodeList.sort((a,b) => a.order - b.order )

  // if a node has branches (Trigger or Logic) we go to the first child of one of those branches
  if ( curr.branches ){
    // some logic function to select a specific branch
    let targetBranch = pickBranch( curr.branches )
    // get the list of nodes within the target branch
    let branchNodes = getNodesInBranch( targetBranch, nodeList )
    // if there's at least one node
    if ( branchNodes.length ){
      // return the first node
      return branchNodes[0]
    }
  }
  // try to go to the next sibling
  let nextSiblingNode = getNextSiblingNode( curr, nodeList )
  if ( nextSiblingNode ){
    return nextSiblingNode
  }
  // there's no viable next sibling, thus we move up to the tree
  while ( curr.branch !== 'main' ){
    // we prefix branches with the ID of the parent node
    let parentNodeId = curr.branch.split( '-' )[0]
    // set curr to the parent node for the sake of the while loop
    curr = getNodeById( parentNodeId, nodeList )
    // get the sibling node of the parent
    nextSiblingNode = getNextSiblingNode( curr, nodeList )
    // one exists, return it
```

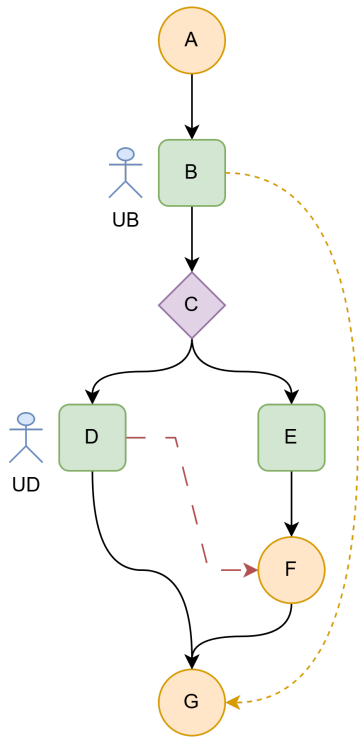
```
    if ( nextSiblingNode ){
      return nextSiblingNode
    }
  }

  // no viable next node was found
  return null
}

/**
 * Get the next sibling of a node
 */
function getNextSiblingNode( curr, nodeList ){
  // we're assuming here that nodeList is sorted by order ASC here
  if ( curr.type === 'TRIGGER' ){
    // trigger has special case, find the first non trigger sibling of higher order in the same
branch
    return nodeList.find( node => node.order > curr.order && node.branch === curr.branch &&
node.type !== 'TRIGGER' )
  }
  // next sibling is first available node of higher order in the same branch
  return nodeList.find( node => node.order > curr.order && node.branch === curr.branch )
}
```

isParallel(P, Q)

Figure 17.

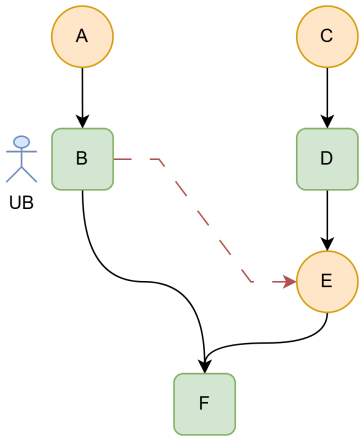


One of the unique functions of trigger nodes is that they act as entry **and** jump points within a flow. Look at *figure 17.* If user UB is at node B and does some interaction or user input that matches the trigger conditions for G, they will skip any nodes in between B and G.

But what about user UD? What if they match the trigger conditions for trigger F? Can UD travel D → F? No, because F is not "after" D, F is *parallel* to D.

We'll define two nodes being *parallel* as any two nodes which are in different branches of a common ancestor. In *figure 17* [D] (being in branch C-a), and [E,F] (being in branch C-b) are *parallel* because they are in different branches of C.

Figure 17a.



Two nodes might also be parallel if they exist in the branches of adjacent trigger nodes, as in *figure 17a.* UB can't travel from B → E because there is no valid path between them.

So *parallel* nodes are defined as any two nodes which are in different branches of a common ancestor node OR in different branches of adjacent trigger nodes. We should also specify that adjacent trigger nodes are also considered *parallel*, so A → C or A → E would be equally invalid.

Because of the shape of an SFT, we can use the ancestry of a node up to the root branch *main* to determine the *parallelism* of two nodes as in the following example.

```
/**
 * Returns the ancestry path up to the root of a flow tree, deepest first
 *
 * @param $node node the starting node
 * @param $nodeList node[] the flattened tree
 *
 * @return [ node[], string[] ] the ancestry
 */
function getAncestry( node, nodeList ){

  let ancestors = []
  let branch_path = []

  let curr = node

  while (curr){
    branch_path.push( curr.branch )
    curr = getParentNode( curr, nodeList )
    if ( curr ){
      ancestors.push(curr)
    }
  }

  // this allows us to quickly see trigger adjacency
  if ( isTrigger(node) ){
    branch_path.unshift(node.branches[0])
  }

  return [ ancestors, branch_path ]
}

/**
 * Determine if two nodes in the tree are parallel
 *
 * @param $p        node
 * @param $q        node
 * @param $nodeList node[]
 *
 * @return bool true if parallel, false otherwise
 */
function isParallel( p, q, nodeList ){

  let [ p_ancestors, p_branch_path] = getAncestry( p, nodeList )
  let [ q_ancestors, q_branch_path] = getAncestry( q, nodeList )

  // check trigger adjacency special case
  let q_diffed = diff( q_branch_path, p_branch_path ).reverse()
  let p_diffed = diff( p_branch_path, q_branch_path ).reverse()
```

```
  // due to tree shape, adjacent triggers will always appear at the same index in the branch
path after being diffed
  if ( q_diffed.length && p_diffed.length && isTriggerBranch( q_diffed[0] ) && isTriggerBranch(
p_diffed[0] ) && isAdjacentTriggers( getNodeByBranch( q_diffed[0] ), getNodeByBranch(
p_diffed[0] ), nodeList ) ){
    return true
  }

  // otherwise, check common case
  let commonAncestors = intersect( p_ancestors, q_ancestors )
  let commonBranches  = intersect( p_branch_path, q_branch_path ) // will always be at least [
'main' ]

  // if there is a common ancestor
  if ( commonAncestors.length ){
    let lowestCommonAncestor = commonAncestors[0]
    let lowestCommonBranch   = commonBranches[0]

    // if the lowest common ancestor is the parent of the lowest common branch, then not
parallel.
    // if the lowest common ancestor is NOT the parent of common branch, that means they're
parallel
    return ! isSameNode( lowestCommonAncestor, getNodeByBranch( lowestCommonBranch, nodeList ) )
  }

  return false;
}
```

This time complexity of `isParallel(P,Q)` is O(D) where D is the depth of the tree from the tree root. It does not require us to rebuild the tree which would be O(N) time.

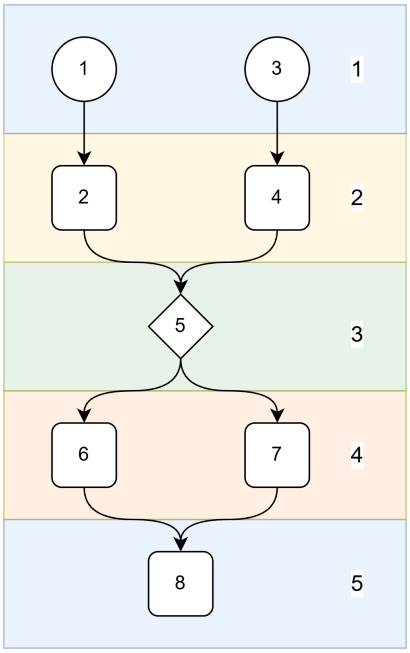
> *`isAdjacentTriggers(P,Q)` will be O(1) time given some pre-computation that will be explained later.*

Potential optimizations for isParallel()

You could optimize further by pre-computing the branch path and ancestors during flattening and storing them in the node's metadata, limiting the need to perform possibly expensive queries.

You could also pre-check possible exclusive scenarios in O(1) time so that in *most cases* `isParallel()` is O(1) time. For example you could check if P and Q are in the same branch to begin with, in which case they might only be parallel if they are both triggers and adjacent. You could also check if both nodes have the same parent but are in different branches in O(1) time. This would make checking nodes *near* each other much cheaper.

Figure 18.



Notice also that because *main* is our root branch, all paths "come from" or "merge into" *main* at some point. So given any P and Q if either P or Q is in the *main* branch and they are not adjacent trigger nodes, then there **must** exist a valid path P → Q.

You could further say if either P or Q are in the lowest common ancestor branch of P and Q then there must also exist a valid path from P → Q, given they aren't adjacent triggers.

Optimize using level

P.order < Q.order is a cheap check to make sure we are moving "forward" in a flow. Order is assigned DFS, and we can see in *figure 18* that 2 → 4 is not possible even though 2 < 4, so we'd have to use our O(D) time `isParallel()` function even though the nodes are near each other.

We can see visually that 2 and 4 appear in the same "row." Every possible forward move the next node is of a greater level than the previous node. Thus, we can say that any valid path should have P.level < Q.level as well.

Levels can be calculated during flattening and stored in the node's metadata so that a quick level check is always O(1) time. Here's a modified version of `flattenSFT()` that also adds level to the node metadata.

```
/**
 * Given a root node or branch, return a flattened array representation of the tree
 *
 * @param node $branch the branch node
 *
 * @return node[] an array of nodes
 */
function flattenSFT( branch, order=1, level=1 ){

  let nodeList = []
  let branchId = branch.id // 'main' or prefixed ID like `1-a`

  branch.children.forEach( node => {
    // convert our node into some kind of flattened representation of the node
    let flattenedNode = flattenNode( node ) // { id: foo, ... }
    flattenedNode.order = order // set the node order according do DFS
    flattenedNode.level = level // set the node level
    flattenedNode.branch = branchId // set the branch
    // increment the order
    order++
    // add the flattened node to our array
    nodeList.push( flattenedNode )
    // if the node has children, they must be branches according to our shape definitions
    if ( node.children ){
      node.children.forEach( branch => {
        // flatten the branch, passing order to preserve DFS ordering
        let flattenedTree = flattenSFT( branch, order, level+1 )
        // set the order to the order of the last node +1
        order = flattenedTree[flattenedTree.length-1].order + 1
```

```
        // merge the flattened branch tree into our main node list
        nodeList.push(...flattenedTree)
      } )
    }
    // adjacent benchmarks have the same level
    if (node.type !== 'trigger') {
      current_level++
    }
  } )

  return nodeList
}

flattenSFT( FlowRoot ) // where FlowRoot is the root node of our flow tree
```

In a previous code example we used `isAdjacentTriggers()` without a definition, now we can use node level to provide one. Adjacent triggers **must have the same level**.

```
function isAdjacentTriggers( p, q ){
  return p.branch === q.branch && p.level === q.level
}
```

hasPath(P, Q)

Finally, we can write a highly optimized function `hasPath(P,Q)` which will be $O(1)$ in most cases and worst case $O(D)$. Either way, significantly better than $O(k(V + E))$ speed of traversing a directed graph.

The $O(1)$ checks we can do to verify an **invalid** path are…

1. `P.order >= Q.order`
2. `P.level >= Q.level`
3. `P.branch != Q.branch AND P.parent = Q.parent`

The $O(1)$ checks we can do to verify a **valid** path are…

1. `P.branch = Q.branch AND P.level < Q.level`
2. `If either P.branch or Q.branch is` *`main`* `AND P.level < Q.level`

Otherwise, we use `!isParallel(P, Q)` to check, because there must exist a valid path between two non-parallel nodes.

```
/**
* Determine if there is a valid path from P to Q
*
* @param p node
* @param q node
* @param nodeList node[]
*/
```

```
function hasPath( p, q, nodeList ){

 // Optimized O(1) valid checks
 if (( p.branch === q.branch || p.branch === 'main' || q.branch === 'main' ) && p.level < q.level ){
   return true
 }

 // Optimized O(1) invalid checks
 if ( q.order < p.order
   || q.level < p.level
   || ( p.branch !== q.branch && isSameNode( getParentNode( p, nodeList ), getParentNode( q, nodeList ) ) )
   || isAdjacentTriggers( getParentNode( p, nodeList ), getParentNode( q, nodeList ) )
 ){
   return false
 }

 // O(D) check
 return ! isParallel( p, q, nodeList )
}
```

Relationship to Series-Parallel Graphs

The structure described in this paper shares similarities with **series-parallel graphs (SP graphs)**, a well-studied class of directed acyclic graphs constructed through recursive series and parallel composition.

In a series-parallel graph:

- **Series composition** represents sequential execution (A → B)
- **Parallel composition** represents branching and merging paths

These concepts map closely to the structure of an SFT.

- Sequential nodes (Trigger → Action → Action) correspond to **series composition**
- Logic nodes introducing multiple branches correspond to **parallel composition**
- Merging behavior after branches aligns with the recombination step in SP graphs

However, an SFT is **not a direct implementation of a series-parallel graph**, but rather a constrained and specialized representation designed for workflow automation systems.

Key differences

The SFT diverges from traditional SP graph representations in several important ways:

Tree-based representation

SP graphs are typically represented as adjacency structures (nodes with parent/child references), requiring traversal algorithms such as DFS or BFS.

The SFT instead:

- Uses a **strict hierarchical (tree-like) structure**
- Introduces **Branch nodes** to encode parallelism without duplicating nodes
- Avoids explicit edge storage entirely

This enables a deterministic and easily serializable structure compatible with UI rendering (e.g., HTML-like trees).

Flattened representation using coordinates

Rather than storing edges, SFTs are flattened into an ordered list of nodes with metadata:

- `order` → DFS position (vertical axis)
- `branch` → structural grouping (horizontal axis)
- `level` → visual hierarchy constraint

This creates a coordinate-like system, any node's position is defined by `(branch, order)`. This encoding allows reconstruction of relationships without explicit graph traversal.

Optimized traversal without graph search

In SP graphs, determining reachability between two nodes generally requires traversal with time complexity:

$$O(V + E)$$

In contrast, the Flow Tree introduces a set of constraints and precomputed metadata that allow:

- $O(1)$ path validation in most cases
- $O(D)$ worst-case checks (where D is tree depth)

This is achieved using:

- ordering constraints (`order`)
- structural constraints (`branch`)
- hierarchy constraints (`level`)
- parallel detection via ancestry comparison

UX-constrained structure

Unlike SP graphs, which are purely mathematical constructs, the SFT is designed around **user interaction constraints**, including:

- Predictable drag-and-drop behavior
- Automatic connection inference
- Minimal-impact deletion ("sub-flow" removal)
- No manual edge management

These constraints significantly restrict the shape of valid graphs, enabling simpler algorithms.

By restricting the generality of SP graphs and encoding structure into node metadata, the SFT achieves improved performance and usability for workflow automation systems.

Conclusion

Traditional workflow systems often rely on directed acyclic graphs to model execution. While flexible, these approaches introduce unnecessary complexity for many real-world automation use cases, particularly in traversal, mutation, and visualization. The need to explicitly manage edges and perform graph-wide computations can make these systems difficult to reason about and inefficient to operate at scale.

In this paper, we introduced the **Structured Flow Tree (SFT)**, a constrained, tree-based model for representing workflow execution. By restricting the shape of valid workflows and encoding relationships implicitly through hierarchy and metadata, the SFT eliminates the need for explicit edge management while preserving the expressive power required for branching and merging logic.

The combination of a hierarchical structure and a flattened representation using `(branch, order, level)` enables efficient storage, deterministic reconstruction, and highly optimized traversal. In practice, this allows most reachability checks and execution decisions to be performed in constant time, with minimal reliance on full structural analysis.

Beyond performance, the SFT aligns closely with the needs of interactive systems. Operations such as adding, moving, and deleting nodes are simplified by the inherent properties of the tree, ensuring predictable behavior and minimal unintended side effects. This makes the model particularly well-suited for visual workflow builders, where usability and correctness are equally important.

While the SFT shares structural similarities with series-parallel graphs, its primary contribution lies in its constrained design, tree-based encoding, and metadata-driven execution model. By prioritizing practical constraints over theoretical generality, the SFT provides a simpler and more efficient alternative to traditional graph-based workflow representations.

Ultimately, the Structured Flow Tree demonstrates that by carefully limiting the problem space and embedding structural information directly into the representation, it is possible to achieve both improved performance and a better developer and user experience in workflow automation systems.

Future work

There are several opportunities to extend and optimize the Structured Flow Tree (SFT) model.

One area of interest is further reducing the worst-case time complexity of reachability `hasPath()` checks. While the current approach achieves $O(1)$ performance in most cases and $O(D)$ in the worst case, additional metadata—such as precomputed ancestry paths or branch encodings—could enable strictly constant-time evaluation for all node pairs.

Another direction is the introduction of a compilation phase, where flows are analyzed and optimized prior to execution. This could include validation of unreachable or redundant nodes, simplification of branch structures, and precomputation of execution paths to reduce runtime overhead.

The SFT may also be extended to support parallel execution semantics, allowing multiple branches to be evaluated concurrently with defined synchronization and merge behavior. This would enable more advanced orchestration use cases while preserving the structural constraints of the model.

Additional work could explore indexing and query capabilities over the flattened representation, enabling efficient retrieval of reachable nodes, branch-local operations, and structural analysis without reconstructing the full tree.

Finally, while the SFT is intentionally constrained and cannot represent arbitrary directed acyclic graphs, it may be possible to introduce controlled extensions that support limited cross-branch connections without sacrificing performance or usability. Investigating the boundaries of this tradeoff presents an interesting avenue for future research.