Data Structures II (CS-201) Project Report
Dynamic Trees for optimized Network Flow algorithms

Musab - 07811, Sara Baloch - , Meesum Abbas - 08056, Rameez Wasif - 08479

April 30, 2024

Contents

1 Abstract

My project was about ...
I developed a system to ...
We did some experiments to find out ...
The main results were ...

2 Intro

For our project, we decided to implement a data structure and visualise its operations based off the paper *"A Data Structure for Dynamic Trees"* [2] by Dr Sleator and Dr Tarjan. This data structure provides a solution for the Dynamic Trees problem, different from the obvious solution, in order to optimize some operations, at the cost of slowing down some others. While it may seem like a harsh tradeoff, this optimization allows the implementation of a max-flow algorithm with time-complexity $O(mn \log n)$ as opposed to a time-bound of $O(n^2 m)$ of Dinit's original algorithm and the previously best $O(nm(\log n)^2)$ for sparse graphs by Galil and Naamad. The max-flow problem is crucial in network routing, transportation and logistics. Therefore, this data structure is incredibly valuable for real-world problems where the order of data is large and these asymptotic improvements become equivalent to days of computing power.
In our report, in section 3 we explore the Data Structure, sharing our own key insights. In section 4, we review the functionality of our implementation and share visualisations of the tree and some of its operations. In section 5, we take a look at Network Flow algorithms. In section 6, we test our data structure and carry out analysis on its computation time and accuracy. In section 7, we offer our review on the data structure and the project overall.

3 Understanding the Data Structure

First, we aim to comprehend the paper's essence. In simple terms, our goal is to represent a tree and efficiently update it - efficiency a key requirement. While representing the tree in its raw form might suffice for certain tasks, offering a time complexity $O(1)$ for some operations, for others, it could incur a time complexity of $O(n)$, which is undesirable. Is there a middle ground, a workaround?

Operations on a tree are dependent on its height. So the simple solution is to balance the tree. But the problem is, you actually want the original tree - with its root and its connections, so you can't balance it. So the data structure has an external representation and an internal representation, each with their own functions. Putting all of this together gives us around 4 levels of abstraction.

We begin by assuming all nodes are unconnected and define the following eight operations of the Dynamic tree, i.e. the upper most interface (pseudocode available in the paper):

1. `parent(v)`: Obtain the parent of node v (return null if no parent).
2. `root(v)`: Retrieve the root of the tree.
3. `cost(v)`: Access the cost of an edge between a node and its parent (v must not be a tree root).
4. `mincost(v)`: Identify the vertex w between v and the $root(v)$ with the minimum cost. In case of a tie, select the vertex closest to the root (v must not be a tree root).
5. `update(v, x)`: Increment the cost of all edges in the path from v to $root(v)$ by x.
6. `link(v, w, x)`: Establish an edge (v, w)—with w becoming the parent—having a cost of x, where v is a tree root. This operaton combines their respective trees.
7. `cut(v)`: Sever the tree containing v by breaking the edge between v and its parent (v must not be a tree root).
8. `evert(v)`: Modify the tree containing v such that v becomes the root.

The paper explores an option to represent the tree using an alternative data structure. Specifically, it separates the tree into two parts. Some edges are grouped together to form different paths, while the remaining edges are stored in a map for efficient access. Consequently, when modifications to multiple edges within a tree are required, operations are applied on the overarching path containing those edges, which proves quicker than altering each edge individually. The remaining edges are modified as normal.

The edges in the map are represented as dashed edges in the diagrams, while the edges in paths are depicted as bold edges. Additionally, it's important to note that a vertex can only belong to one path. Moreover, a path always ascends, with its head being the lowest node in the path and its tail being the highest node.

Descending one layer of abstraction, we encounter the `splice` and `expose` functions (pseudocode provided in the paper).

The `splice(v)` function extends a bold path upwards by permitting a bridge over one dashed edge, making it bold. However, in making this connection bold, the previous connection of the node to its other children becomes dashed.

The `expose(v)` function connects a node all the way to the root via a bold path and makes and downward connections from a node dashed. Essentially, it applies splice over and over again until we reach the root. It also makes my children dashed.

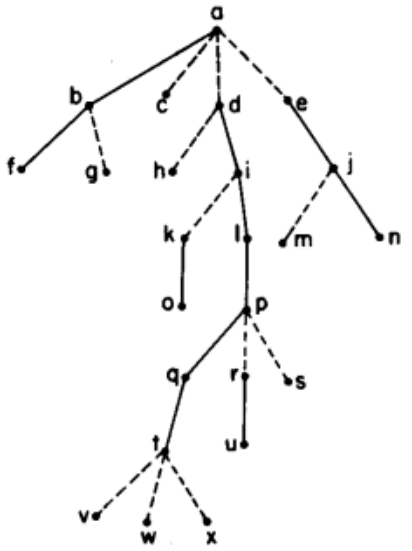


Figure 1: An example of a dynamic tree. The bold edges represent paths while the dashed edges are stored in maps. (source: paper)

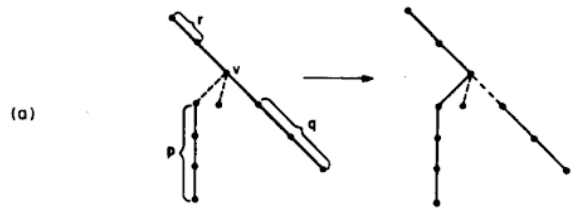


Figure 2: The effect of a splice operation. The connection of v to p becomes bold while it's other downward connection (with q becomes dashed) (source: paper)

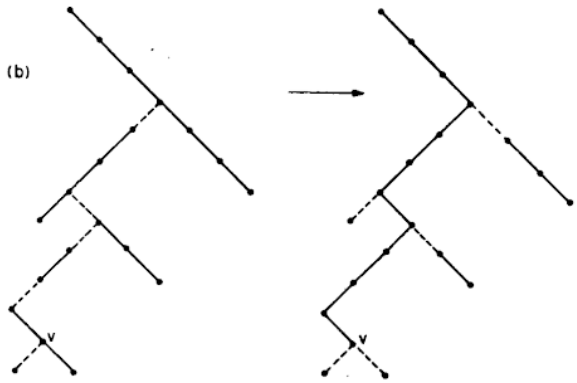


Figure 3: The effect of an expose operation. (source: paper)

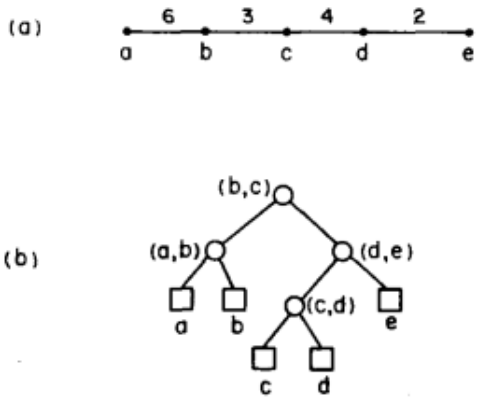


Figure 4: A path and its internal representation. (source: paper)

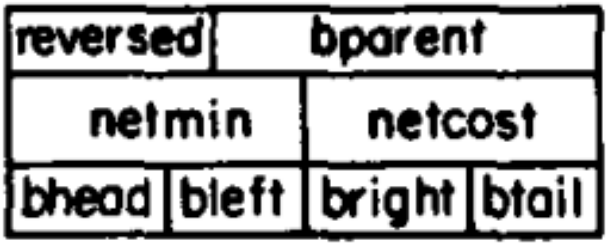


Figure 5: An internal node's contents. (source: paper)

Moving down another layer of abstraction, let's now look at the path. It is stored as a full tree, where each of the non-leaf nodes (internal + root) represent an edge. Moreover, its subtree represents a subpath. The leaves/external nodes represent the vertices in the path. Figure 4. is handy.

Moreover, if we do an inorder traversal of the path, that gives us a path from head to tail, or if talking about our original tree, it's a path moving upwards.

Before we look at each of these nodes, let's clarify two important terms related to costs of edges:

- `Grosscost`: Cost of the edge represented by a node.
- `Grossmin`: Least cost of all the edges in the path represented by the subtree of a node.

Each node has many aspects. Let's take a look at each of these:

- `Reversed`: Denotes the reversal state of the path represented by the node's subtree. If a node is reversed, its left child is its right child and vice versa + its head is its tail and vice versa.
  Note: The actual reversal state of a node is determined by counting the number of reversal states from the node to root. If odd then the node/path is reversed, if even, then all the reversals cancel out and the node/path is not reversed.
- `Bparent`: Node's parent in the tree.
- `Netmin`: The difference between the mincost edge in the path represented by the node's subtree and the mincost edge of the path represented by the node's parent's subtree → `grossmin(v)` if v is root else `grossmin(v) - grossmin(parent(v))`.
  Note: The grossmin of a node can be found by summing up the netmin from the node to the root of the path. In a sense, it's similar to prefix sums.

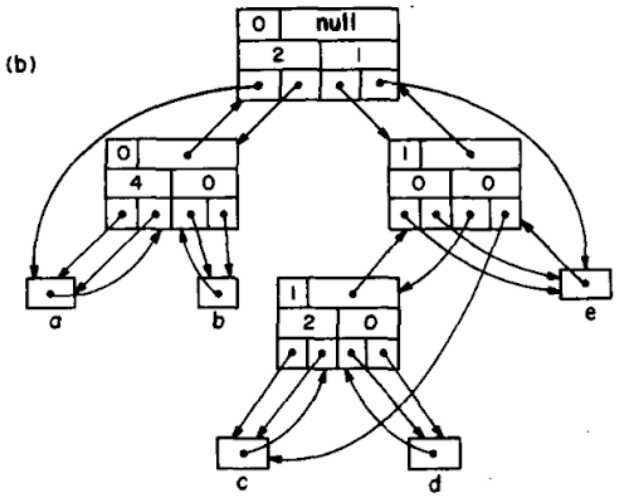


Figure 6: An internal tree for the path in Figure 4. with each node's attributes shown. (source: paper)

- `Netcost`: How much greater is the cost of the edge represented by this node as compared to the mincost of the subtree represented by this node → `grosscost(v)` - `grossmin(v)`.
  Note: If we know the grossmin of a node, we can get its cost i.e. the cost of the edge represented by this node, by summing up grossmin and netcost.
- `Bhead`: The head of the path represented by the node's subtree.
- `Bleft`: The left child of the node in the binary tree.
- `Bright`: The right child of the node in the binary tree.
- `Btail`: The tail of the path represented by the node's subtree.

Note that some of these properties are not relevant for a leaf node, e.g. `netmin` and `netcost`.

We use 11 operations to work on these paths (explanations for implementation for the first 9 operations are provided in the paper) - note that `v` refers to a vertex and `p` to a path:

1. `path(v)` - Find the node identifying the path of a vertex, i.e., the root of the tree (simply traverse upwards from the vertex node until unable to proceed).
2. `head(p)` - Find the first node in the path, considering reversal states (access the `head` of the path if the path isn't reversed, otherwise access the `tail`).
3. `tail(p)` - Symmetric to `head`.
4. `before(v)` - Find the vertex just before v in the bold path (ascend the path until the current node is a right child of some parent node, then access this parent node's left child and obtain its `tail`).
5. `after(v)` - Symmetric to `before`.
6. `pcost(v)` - Find the cost of the edge $(v, \texttt{after}(v))$ (traverse upwards until the current node is a left child of some parent node, similar to `after`. This node represents the edge between

$(v, \texttt{after}(v))$. Determine its cost by computing `grossmin` (method detailed earlier) and adding it to `netcost`).

7. `pmincost(p)` - Find the cost of the smallest edge in the path. In case of a tie, choose the one closest to the tail of the path.
   Note: the description of this has a slight error in the paper, a complete revised description is provided in the Appendix note 1.

8. `pupdate(p, x)` - Add x to the cost of all edges in the path (add x to `netmin` of p).

9. `reverse(p)` - Reverse the path (change reverse bit of p).

10. `concatenate(p, q, x)` - Join two paths using an edge of cost x.

11. `split(v)` - Break the path into the path before v and the path after v.

Note that for each of these functions, we have to be mindful of reversal states - for example, in the stopping conditions for `before`, where we are looking for a node where v is a right child. If the node is reversed, then we will check if v is its left child.

`Concatenate` and `Split` are special because they actually modify the path, while the others simply traverse it and access/modify some node data. Since they manipulate the structure of the path, we need a way to optimize how these changes occur, ensuring future operations can continue to be performed efficiently.

However, another barrier we face is that such an important function doesn't actually have much description in the paper... at least not this one.

Initially, we started with a rather simplistic implementation:

- For `concatenate`, simply create a new node linked to the root of both paths.
- For `split`, note that the parent in the path tree for a vertex is an edge connected to it. Rotate this edge upwards until it's at the top of the tree. Once it's at the top, delete this node and update its immediate children if necessary. Repeat this process a second time if needed, as a node can have two edges connected to it (in and out).

This implementation, while functional, is not optimal. For a random sequence of operations, this is just fine and actually faster than what's targeted in the paper. However, there exists some sequence of m operations where our time complexity approaches $O(mn)$, where m is the number of nodes in the tree. Particularly, this happens in the sequence of steps that links nodes in the order $1, 2, 3, \ldots, n$ and then cuts it as $1, 2, 3, \ldots, n$. This isn't ideal. Luckily, there is an optimization we can still do to cut this down to $O(m \log n)$.

To understand how these optimizations take place, we have to refer to Sleator, Tarjan and Bent's paper on *"Biased Search Trees."* [1] Here, we basically try to keep more frequently required nodes higher up in the tree, while less frequently accessed nodes are kept lower down. Through some analysis, Sleator and Tarjan highlight that this setup actually amortizes the cost to $O(\log n)$, which is exactly what we need to speed up the data structure.

`Concatenate` and `Split` are mentioned in this other paper, along with one of their helper functions - `tilt` (left and right).

However, these operations are built on top of a 4th level of abstraction (partly referenced in our description of `concatenate` and `split`). This includes the functions:

- `construct(u, v, x)` - Given the roots v, w of two trees and a real value x, combine the trees by creating a new node with left child v and right child w and `grosscost` x.

- `destroy(u)` - Given a root u of a non-trivial tree, delete u, returning $[v, w, x]$ where v is its left child, w is its right child, and x is the cost of the edge represented by u. Note that since our binary tree is always full, both children will exist for a non-trivial tree.
- `rotateleft(v)` - Perform a single left rotation at node v. Node v must have an internal right child.
- `rotateright(v)` - Perform a single right rotation at node v. Node v must have an internal left child.

For the last two operations, we opted for a single option `rotate(v)`, which takes the child of the node being rotated and determines the type of rotation based on whether it is a left child or a right child.

Generally, these operations are standard tree operations, but in our case, we have some messy attributes to be careful with—namely cost and reversal state. We will go over solutions to these operation by operation.

1. `construct(u, v, x)`:
   - **Cost**: The `netmin` of the new node is supposed to represent the combined `grossmin` of u, v, and x, so we just consider the minimum of x, `netmin(u)`, `netmin(v)` since u and v have no parents and therefore their `netmin` is their `grossmin`. x is the `grosscost` of the new node so we can determine the `netcost` by subtracting `grossmin` from x. Also note that the children of this new node now have parents so their `netmin` will be updated, particularly, their `netmin` -= the new node's `netmin`. Their `netcost` is not updated as `netcost` depends upon `grosscost` and `grossmin` which only refers to their own subtrees, thus adding a node above has no effect.
   - **Reversal**: Set the reversed state of new node to 0.
2. `destroy(u)`:
   - **Cost**: Since u's children will no longer have parents, we must add back u's `grossmin` (same as u's `netmin`) to their `netmin`.
   - **Reversal**: If u is reversed, then flip the reversal state of both it's children before deleting.
3. `rotate(v)`:
   - **Cost**: Let u be the parent of v, s be the sibling of v, c be the child of v that stays v's child, and x be the child of v that becomes u's child.
     - **Case 1**: The `grossmin` of u was in v's subtree:
       - v's `netmin` becomes u's previous `netmin` since it has u's previous `grossmin` and u's previous parent. v's `netcost` is unchanged since its `grossmin` is unchanged.
       - c is unchanged since its parent's `grossmin` is unchanged and its subtree is unchanged.
       - u's `netmin` becomes the least out of its previous `netcost` and its new children's (s, x) previous `netmin`, because its children's previous `netmin` were with respect to the `grossmin` of u which is now the `grossmin` of v. And its `netcost` was how much more it was from its previous `grossmin` which is now the `grossmin` of v. u's `netcost` is reduced by its `netmin` as u's `grossmin` has increased by `netmin`.
       - For s/x, `netmin` is reduced by u's new `netmin` as u now acts like an intermediate between s/x and the actual smallest edge—keep in mind the prefix sum type property. `Netcost` unchanged as subtree unchanged.

  - **Case 2**: The `grossmin` of u was not in v's subtree:
    * v's `netmin` becomes u's previous `netmin` since it has u's previous `grossmin` (due to having u's previous subtree) and u's previous parent. v's `netcost` increases by `netmin` since its `grossmin` has decreased by as much.
    * c's `netmin` is increased by v's new `netmin` as v's `grossmin` has decreased by as much. Its `netcost` is unchanged since its subtree is unchanged.
    * u's `netmin` becomes the least out of its previous `netcost` and its new children's (s, x) previous `netmin`, because its children's previous `netmin` were with respect to the `grossmin` of u which is now the `grossmin` of v. And its `netcost` remains unchanged as it has the smallest edge and so its `grossmin` is unchanged.
    * For s, `netmin` is reduced by u's new `netmin` as u now acts like an intermediate between s and the actual smallest edge—keep in mind the prefix sum type property. `Netcost` unchanged as subtree unchanged.
    * For x, `netmin` increases by the original `netmin` of v as v used to act like an intermediate between x and the actual smallest edge—keep in mind the prefix sum type property. `Netcost` unchanged as subtree unchanged.

- **Reversal**: If the reversal state of v is 1, we should resolve this before rotating, i.e., swap v's left and right children as well as itst head and tail and flip its reversal state. If v's parent is reversed, that means the path in its subtree is reversed. However, after v is rotated, its subtree will have all the same nodes it's former parent's subtree had. So v just takes its former parent's reversal state and v's former parent takes a reversal state of 0.

There's one last thing to look at before we're done with our data structure. And that is ranks and weights for optimization.

The paper states that the rank of a node should be the logarithm (base 2) of its weight.

Moreover, it describes weight for leaf nodes to be:

$$\mathrm{wt}(v) = \begin{cases} \mathrm{size}(v) & \text{if no solid edge enters } v; \\ \mathrm{size}(v) - \mathrm{size}(w) & \text{if the solid edge } (w, v) \text{ enters } v \end{cases}$$

where size(v) refers to the number of nodes in the subtree of v.

Moreover, the weight of non-leaf nodes becomes the sum of the weight of its immediate children, which is also the same as the sum of the weight of all leaf nodes in its subtree.

This leads to changes in `construct`, `rotate`, `splice`, and `expose`, and it also requires that we initialize our nodes with a weight of 1 as all nodes are initially unconnected.

The changes in `construct` and `rotate` are trivial—the weight of a new node is the sum of the weights of its children. For weight after rotating a node v, we can determine its original parent u's weight by summing up the weight of its children. After doing this, we can update the weight of v by summing up the weight of its children.

The changes in `splice` and `expose` are detailed in the paper (pseudocode); however, it misses one key detail in `expose`, highlighted in Appendix note 2.

With these changes implemented, our data structure is complete and optimized.

4 Testing our Implementation

Let us now walk through some examples of operations on the data structure and our visualizations of both the internal state and the actual tree.

For the actual tree, the labels in the node indicate the vertex, while the edges represent the cost of edges.

In the internal tree, the leaves are round and their labels indicate the vertex they represent. The non-leaf nodes are box-shaped, and their label represents the edge they refer to, e.g., (2, 1) indicates the edge (2, 1). Moreover, a * on a node indicates that it is reversed, so (2, 4)* means this node is reversed. The bold edges represent left and right children pointers, labelled as `l` and `r` respectively, while dashed edges are used for head and tail pointers, labelled as `h` and `t` respectively. The value at the top-left of a node represents its rank, which is used in balancing the internal tree.

For the first example, assume that we start with 3 nodes with no connections as in Figure 7. We perform the operation `link(2, 1, 5)` making 1 the parent of 2 through an edge of cost 5 (Figure 8). Now we perform the operation `link(3, 2, 4)`. Note that in Figure 9, the `l` label is on an edge going to the right, while the `r` label is on the edge going left. This is a limitation in our visualization. However, if we just focus on the label names, we can see that if we carry out an inorder traversal on the nodes, we get the bold path of the actual tree from bottom to top.

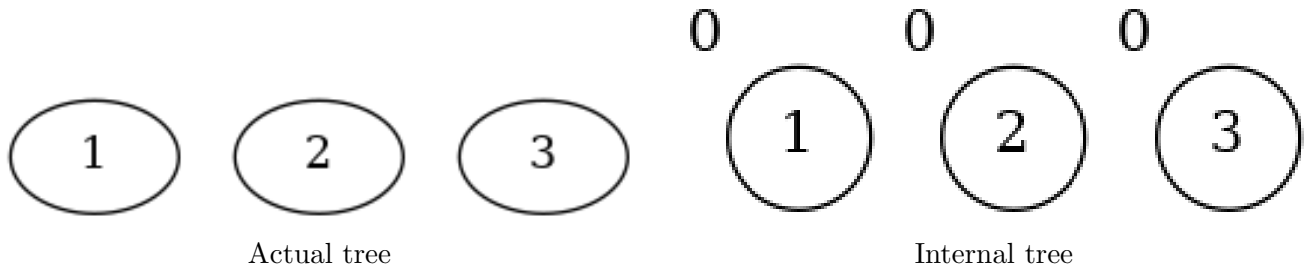


Figure 7: Initial tree of 3 nodes

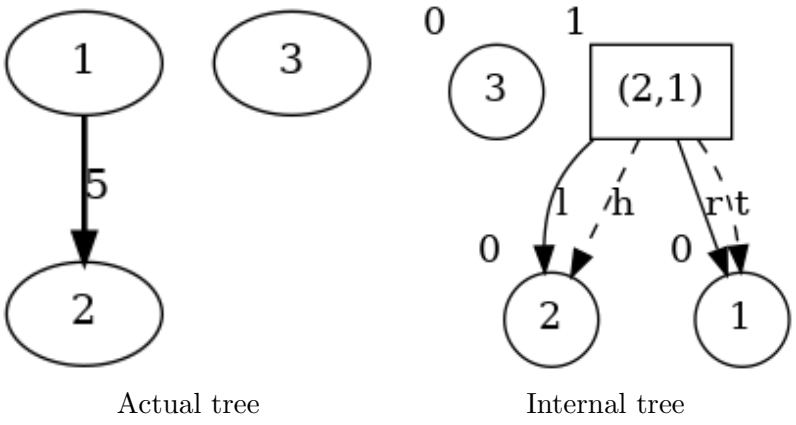


Figure 8: After `link(2, 1, 5)`

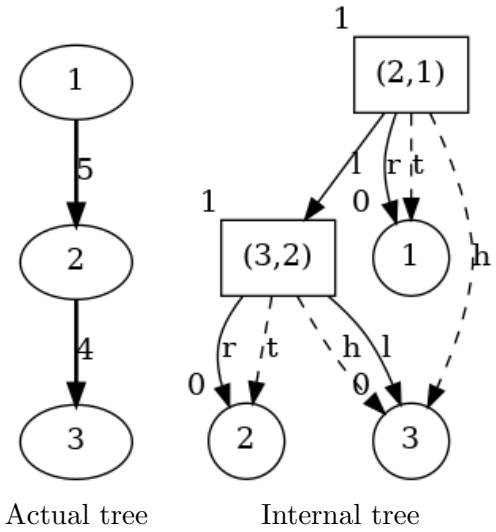


Figure 9: After `link(3, 2, 4)`

Let us now consider a larger example now. Consider the tree in Figure 10 as our initial tree. We consider separately the following operations on this tree:

- `evert(4)` (Figure 11)
- `cut(2)` (Figure 12)
- `update(10, 2)` (Figure 13)

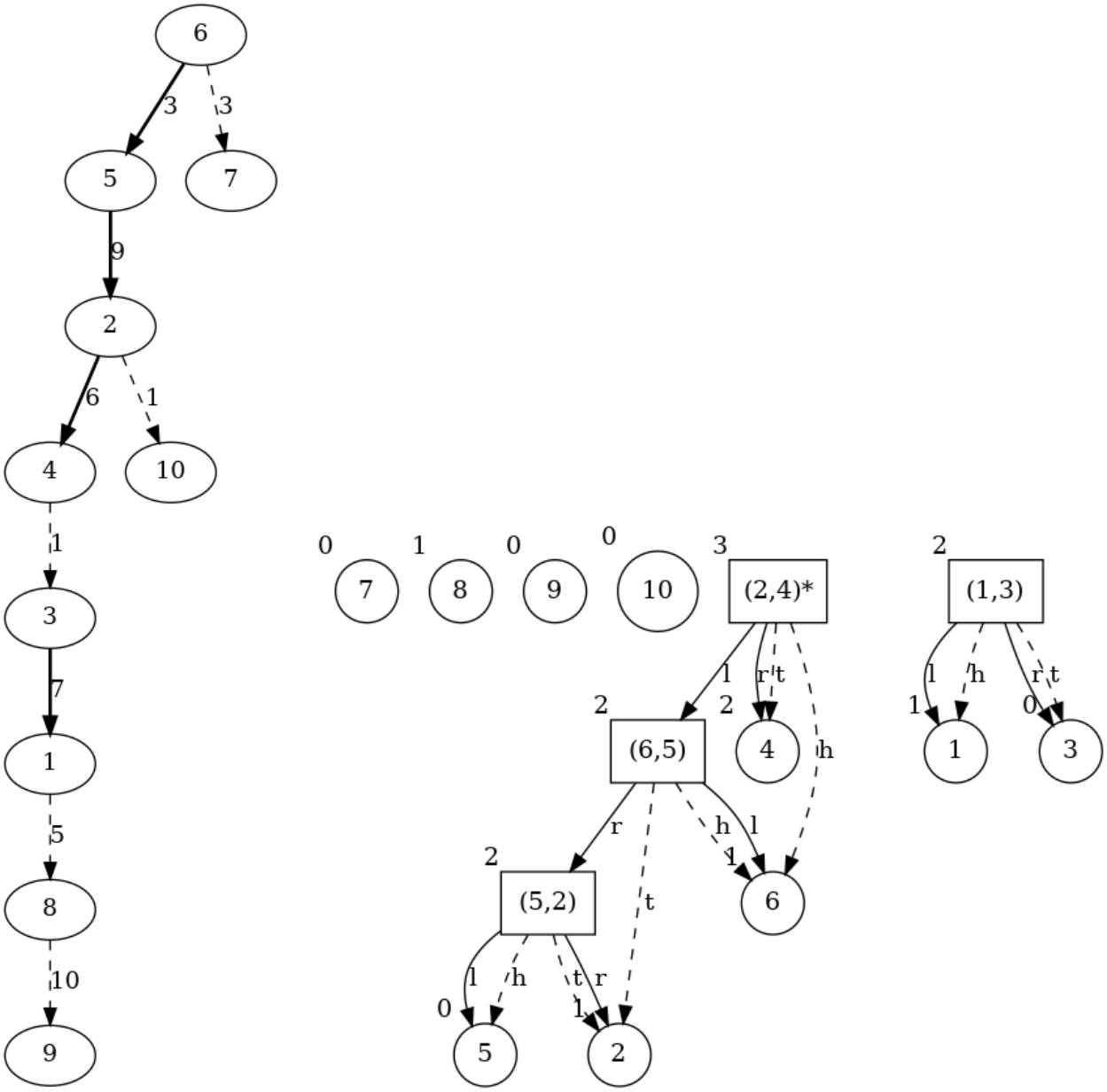
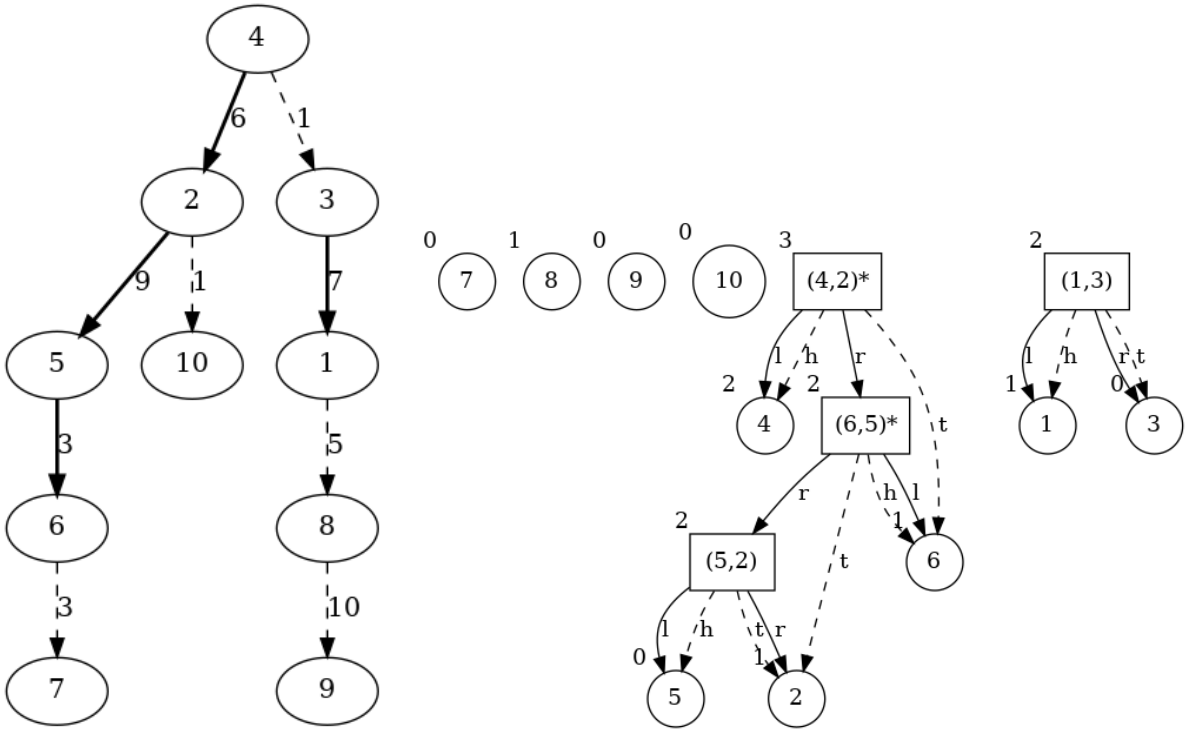
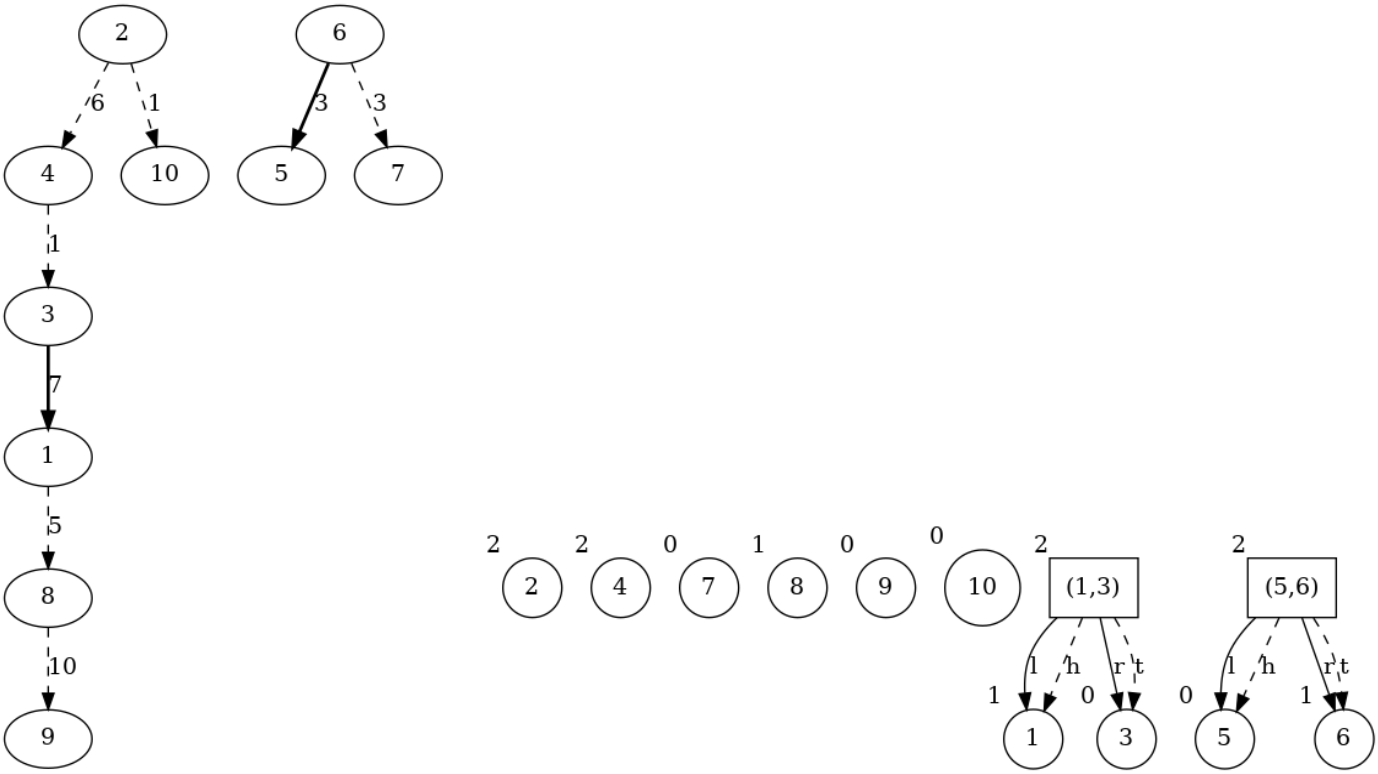


Figure 10: Sample Tree



Actual tree Internal tree

Figure 11: After `evert(4)` on tree in Figure 10



Actual tree Internal tree

Figure 12: After `cut(2)` on tree in Figure 10

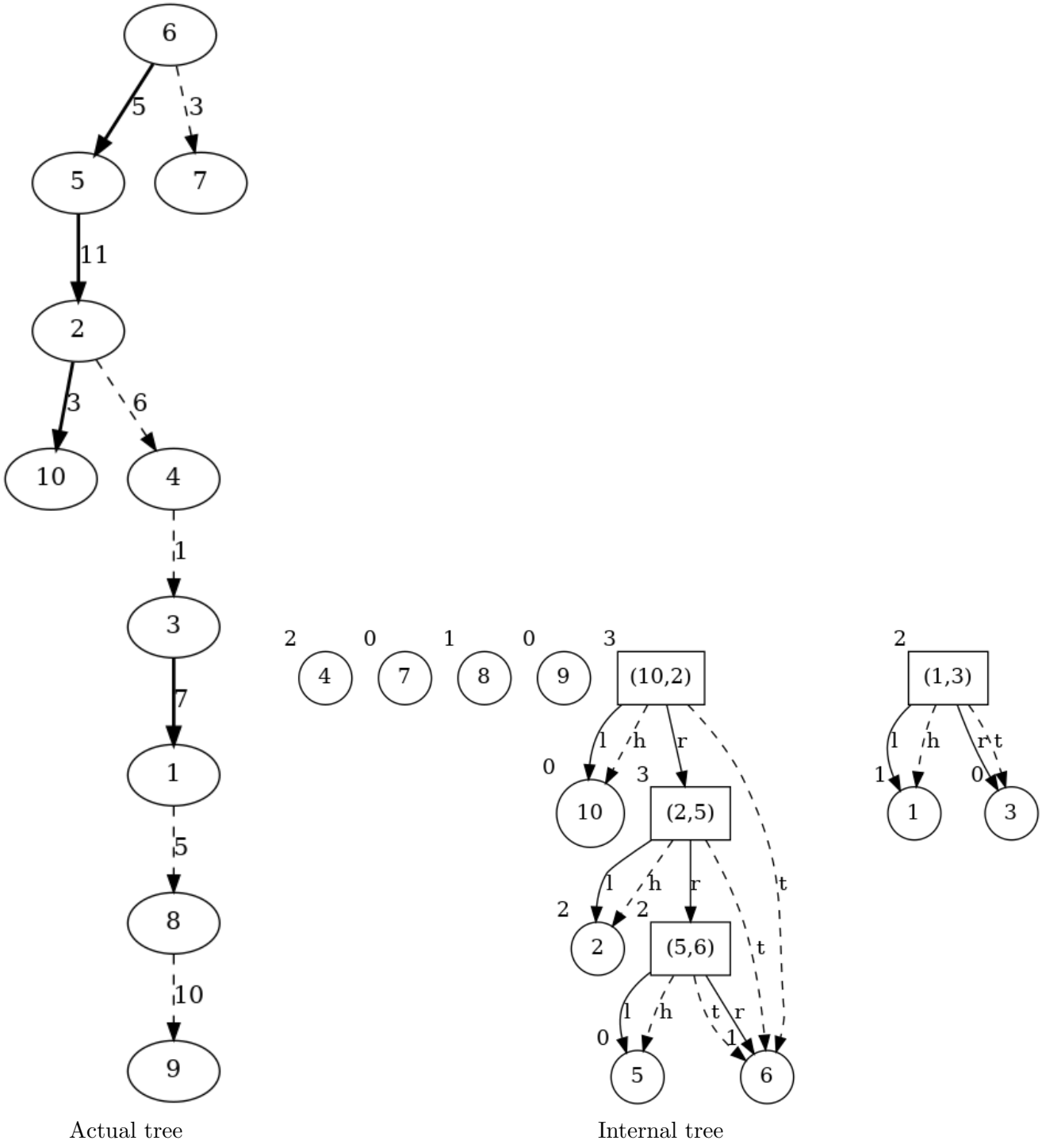


Figure 13: After `update(10, 2)` on tree in Figure 10

We also note that some of the remaining functions: `parent`, `root`, `cost`, `mincost`; do potentially modify the bold and dashed paths. We leave the reader to predict and test these changes.

Moreover, we provide the resulting value of the following operations on the tree in Figure 10:

- `parent(4)` = node 2
- `root(4)` = node 6

- `cost(4)` = 6
- `mincost(4)` = node 5

5 Understanding Network Flow

The maximum flow of a network, i.e., given the capacities of various connections, is the maximum amount of flow that can be sent from a source to a sink without exceeding the capacity of any connections.

Dinit's algorithm generates a level graph which only keeps edges that take us closer to the sink. If no path to a sink is found in generating a level graph, we stop. With the level graph made, we now search for paths to the sink and send flow across it. When considering the paths, we can also traverse an edge in reverse if there is already flow going through it in the forward direction. We keep searching for paths and sending flow until no more paths can be found. This is a blocking flow. Once we reach this state, we generate a new level graph and repeat the process.

Dinit's algorithm generates at most n blocking flows. A normal approach takes $O(nm)$ time to evaluate each of these blocking flows, thus giving an overall time complexity of $O(n^2m)$ for the max flow problem. However, using dynamic trees, we can reduce the time to $O(nm \log n)$. (Algorithm steps provided in the paper)

6 Analysis

We start our analysis with a comparison of our unoptimized Dynamic Tree (with a naive implementation of `construct` and `split`) and the optimized Dynamic Tree (making use of locally biased binary trees).

We consider the following sequence of $2n - 1$ operations on n nodes:

```
link(1, 2, 1), link(2, 3, 1), ..., link(n-1, n, 1)

cut(1), cut(2), ..., cut(n)
```

When we compare the performance of the two data structures, we see in Figure 14 that the execution time for the unoptimized data structure rises rapidly. We conjecture its performance to be $O(n^2)$. Whereas, the theoretical complexity of the optimized Dynamic Tree is $O(n \log n)$. Our results seem to confirm this.

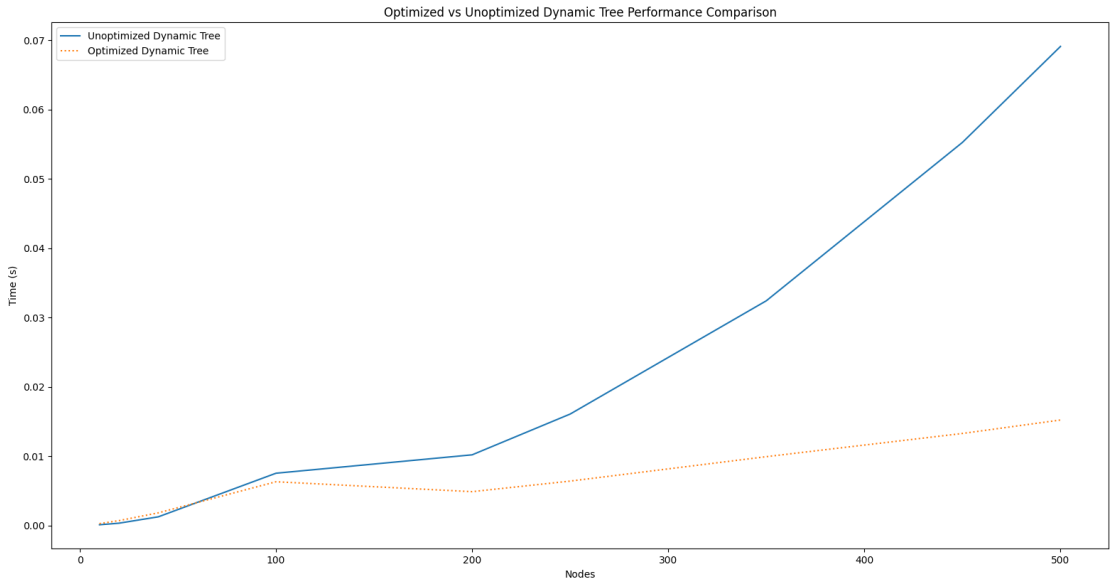


Figure 14: The effect of an expose operation.

Now we come to the evaluation of Dynamic Trees for the max flow problem. We generate graphs of n nodes with random edges, i.e., each possible edge has a specified probability of being present in our graph (in our examples 2% (Fig. 15) and 30% (Fig. 16)) along with a random capacity between 1 and n.

We run the following max flow algorithms:

- Ford Fulkerson - $O(m^2 n)$ - taken from GeeksforGeeks
- Dinit's max flow algorithm - $O(n^2 m)$ - taken from GeeksforGeeks
- Dinit's max flow algorithm (using unoptimized Dynamic Trees) - $O(n^2 m)$ in the worst case
- Dinit's max flow algorithm (using optimized Dynamic Trees) - $O(nm \log n)$

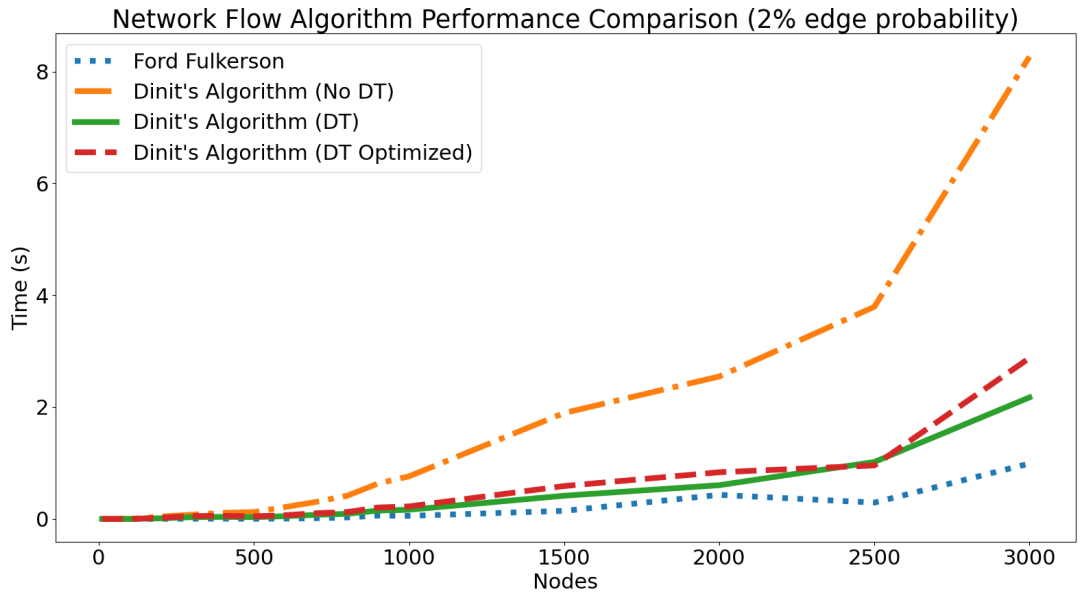


Figure 15: Network flow algorithm comparison for edge density 2%.

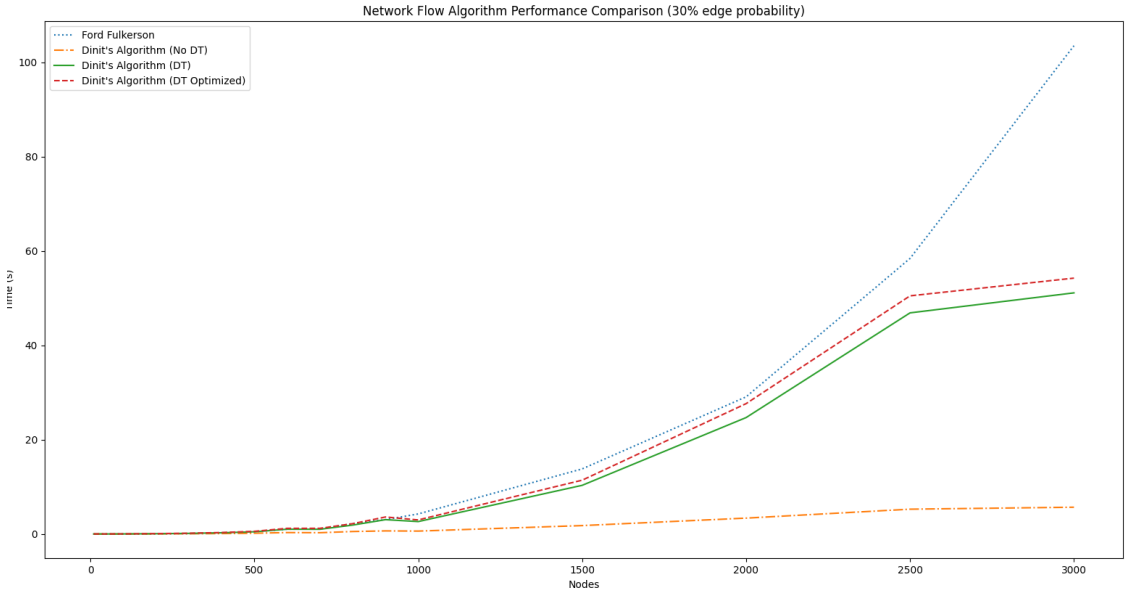


Figure 16: Network flow algorithm comparison for edge density 30%.

Our results have three key insights:

1. The two variants of the Dynamic Trees have minimal difference in execution. In fact, the unoptimized data structure often outperforms the optimized one. We believe this to be due to the randomization in the generation of the input graph.

2. For fewer numbers of edges (Fig. 15), where the impact on execution is largely from the number of nodes, Dinit's max flow using Dynamic Trees notably outperforms the general Dinit's max flow algorithm. Ford Fulkerson performs best as it has the smallest dependency on the number of nodes for execution.

3. For a larger number of edges (Figure 16), where the impact on execution is largely due to the number of edges, Dinit's max flow using Dynamic Trees struggles to compete with the general Dinit's max flow algorithm. Perhaps this is due to large constant costs in the Dynamic Tree. Moreover, we see that our Dynamic Tree implementation does outperform Ford Fulkerson, thus confirming that our implementation isn't just a worse version of Ford Fulkerson.

The discrepancy in the last result in the Dynamic Tree implementation versus the general Dinit's algorithm may be because the number of edges is not fixed but rather is a function of the number of vertices. We note that the number of edges is approximately n^2 times the probability of edge.

Regardless, we conclude that the Dynamic Trees do seem to be implementing a version of Dinit's flow, as can be seen by their ability to outperform the Ford-Fulkerson algorithm for a large number of edges. Moreover, it also seems to be optimizing on Dinit's as it outperforms it when edge density is less.

7 Conclusion

8 Appendix

1. ***Revised explanation for `pmincost(p)` (errors in bold—comments in [ ])***:
   Starting from p, which is the root of a binary tree, proceed downward in the tree, keeping track of reversal states, until finding the node u last in symmetric order such that `grosscost(u)` = **grossmin(p)** [because that means the edge represented by u has the least cost in the path]. This can be done by initializing u to be p and repeating the following step until u has `netcost` zero and its right child is either external or has positive `netmin`: if the right child of u is internal and has **netmin** zero [as `netmin` allows me to check if the least cost edge is in this subtree or not], replace u by its right child; otherwise if u has positive `netcost`, replace u by its left child.

2. ***Revised pseudocode for `expose` (errors in bold)***:

```
function expose(v):
    path p, q, r; real x, y;
    [q,r,x,y] := split(v)
    if q != null ->
        dparent(tail(q)), dcost(tail(q)) := v, x;
        wt(v) := wt(v) + wt(q)
    fi;
    rank(v) = log(wt(v));
    if r = null -> p := path(v)
     | r != null -> p := concatenate(path(v), r, y)
    fi;
    do dparent(tail(p)) != null -> p := splice(p) od;
    return p
end expose;
```

3. ***GitHub Repository***: DynamicTree_NF

References

[1] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM Journal on Computing*, 1985.

[2] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3), 1983.