Fast Distance Queries in Series-Parallel Graphs using
Lowest Common Ancestor

Andrei Gönczi
Advised by Steven Lindell
May 2011

Abstract
Fast distance querying in certain types of graphs is often used in databases,
ranked keyword search, XML databases, social networks, games and for other
purposes. As graphs within these categories get larger and larger, the need for
faster query algorithms increases. But since the same graphs are used for many
different queries, it is sometimes best to find a fast way of answering a single
query at a time after a fairly quick preprocessing stage.
I will be looking at both the directed and undirected case for a specific kind of
graph, trying to look at ways of speeding up queries as much as possible. Series-
parallel graphs are not only widely used, but are an essential step into solving
general distance queries. They are the 2nd simplest kind of graphs (after trees),
and thus are essential to our understanding of how computation works.

1 Introduction

Let $G = (V, E)$ be an unweighted graph, where $N = |V|$. Given $u$ and $v$, both in
$V$, the shortest distance problem answers what the minimum number of edges is, that
have to be traversed in order to get from $u$ to $v$. Whether the graph is directed or not
does not make a difference yet, but I will look into the differences later.
There are some simple algorithms that answer the shortest path problem. The easiest
solution is just running a $BFS$ (breadth first search) from node $u$ until node $v$ is
found. This guarantees that the shortest distance will be found and does not need any
preprocessing, but the running time of this algorithm is $O(N)$ for each query $(u,v)$.
Supposing that the number of queries on the same graph is comparable in size to $N$,
the total running time of the $N$ queries will be $O(N^2)$, which is not feasible for modern
standards of databases or other applications.

A way of answering queries in constant time is to store all pairs of distances after
computing them. Running a $BFS$ from each vertex of the graph will return all $N^2$
distances in the graph, and these can be stored and retrieved in $O(1)$ during queries.
The problem with this is that it takes up \(O(N^2)\) space, which is, again, unfeasible.

For the less general case I am looking at, series-parallel graphs, there is a fast way of answering \(N\) queries in \(O(N)\) total time, if the graph is directed from the source to the sink. In the undirected case, my solution can answer \(N\) queries in a total time of \(O(N \cdot \log^2(N))\) in the worst case.

The transitive closure (\(TC\)) problem simply answers the question of whether there is a path from node \(u\) to node \(v\). That is, \(TC(u,v) = 1\) only if there is a path that starts at node \(u\) and ends at node \(v\). Otherwise, \(TC(u,v) = 0\). If the graph we are looking at is undirected, answering transitive closure queries is trivial: a \(BFS\) is run on each connected component of the graph and the all the nodes within the same component are labeled with the same number. Thus, if the labels of nodes \(u\) and \(v\) are the same, \(TC(u,v) = 1\); if not, \(TC(u,v) = 0\). This takes \(O(N)\) time preprocessing and \(O(1)\) time for each query.

If the graph we are looking at is directed, labeling will not help, since having the same label only means that the two nodes have a common ancestor. But, this problem can also be solved in \(O(N)\) preprocessing and constant time query answering.

## 2 Series-Parallel Graphs

Series-parallel graphs (\(SPG\)’s) “are graphs with two distinguished vertices called terminals, formed recursively by two simple composition operations” (www.wikipedia.org): stringing two series-parallel graphs in series and merging the sink of the upper one with the source of the lower one; or stringing two series-parallel graphs in parallel and merging the sinks and the sources of the two graphs. The smallest series-parallel graph used in the recursive formation of a graph has one edge, although a single node still classifies as a series-parallel graph.

Series-parallel graphs have a tree-width of two and are the second simplest types of graphs other than trees. The solution for fast distance queries in trees has been known for decades.

Also, every series-parallel graph has a series-parallel decomposition that is a tree that can be constructed in linear time and used to string the \(SPG\) back together. I will use this decomposition to find the transitive closure of two nodes in constant time, that will also help in finding the distance between these two nodes in the directed case. In the undirected case, we will use the SP-decomposition again to answer distance queries.

### 2.1 Directed Series-Parallel Graphs

In the directed case, we will consider that all edges are directed from the source to the sink of the graph and in any sub-graph. In this case, finding the transitive closure of two nodes in the graph in constant time becomes somewhat difficult, but can be solved using the seriesparallel decomposition. In the SP-decomposition, by finding the lowest common ancestor of the two nodes, we can tell whether or not the second node
is accessible from the first one.

3 The Ancestor Problem

Definition: In a tree, node $x$ is the ancestor of node $y$ if and only if node $x$ is on the path from the root of the tree to node $y$.

Any node $u$ is an ancestor of itself. Ancestor queries (“is $x$ an ancestor of $y$?”) can be answered in constant-time after $O(N)$ preprocessing on a tree. This can be done in many ways, but one of the simplest ways to do it is using the pre-order and post-order numbering’s of the tree. Node $x$ is $y$’s ancestor if and only if:

$$(\text{pre}.x < \text{pre}.y) \land (\text{post}.x > \text{post}.y)$$

4 Lowest Common Ancestor Retrieval

Definition: In a tree $T$, the lowest common ancestor (lca) of nodes $u$ and $v$ is the deepest node in $T$ that is an ancestor of both $u$ and $v$.

The lowest common ancestor of two nodes in a tree can be found in constant time, which will help us solve the shortest distance problem in series-parallel graphs. I will now present Gusfield’s algorithm and explain how it is used for the shortest distance problem later.

Gusfield supposes that operation on numbers with up to $\log(N)$ bits are considered to be processed in constant time. This includes all arithmetic operations, boolean operations and comparisons.

4.1 Binary trees

The simple case of finding the lowest common ancestor is when the tree is a complete binary tree. In this case we can preprocess the tree and number its nodes with numbers following an in-order traversal. The binary representation of each node represents the path bits, which describe the unique path from the root of the tree to the node. 0’s in the path number represent a move towards a left child, while 1’s represent a move towards a right child. For instance, if the path from the root to the node was right, left, right, then the path number would be 101, followed by a 1, and as many 0’s as are needed to get the total number of bits in the number up to $\log(N)$. This can be seen in Figure 1 – for instance node 5 passes the path bits 101, which is the equivalent of starting at the root, going right and then left. This can be seen to be correct in Figure 1.
Figure 1: A binary tree with 7 nodes

Definition: node $v$ in $B$ will be labeled as the string that describes the path to the node, followed by a 1 and 0's. This is equivalent to the in-order traversal numbering.

To find the $lca$ of two nodes, $x$ and $y$, in a binary tree, we need to find how many of the path bits of the two nodes are the same. This can be done by a simple xor operation and by finding the left-most 0-bit in the result of the xor. If the left-most 1-bit in the result is in position $k$ (counting from the left), then the left-most $k-1$ bits of $x$ and $y$ are the same. Thus, if $lca(x, y)$ is neither $x$ nor $y$, then their $lca$ can be found by concatenating the first $k-1$ bits of $x$ (or $y$, since they are the same), and adding a 1 and then as many 0’s as are needed to fill the rest of the $\log(N)$ bits. This way we find the deepest node that is still on the path from the root to both $x$ and $y$.

Definition: the $lca$ of nodes $x$ and $y$ in $B$ will be the string composed of the longest common prefix of $x$ and $y$ followed by a 1 and 0’s.

For instance, in Figure 1, the $lca$ of nodes 1 (001) and 3 (011) will be their longest common prefix (“0”) followed by a 1 and ending 0’s. This means their $lca$ is 010, which is node 2.

If the $lca(x, y)$ is $x$ or $y$, then the result we get by using the earlier algorithm could be wrong, but this can be easily solved. We are going to need the height of each node to solve the this problem. This can be done in constant-time for any node by finding the right-most 1-bit.

If $height(lca(x, y)) < \max(height(x), height(y))$, then the correct $lca(x, y)$ is the node among $x$ and $y$ that has the bigger height (the root has height $\log(N)$).
For instance, the lca of nodes 6 and 7 in Figure 1 would be 111, which is node 7. But node 7 has lower height then 6. Thus the actual lca is node 6. This is a constant-time way of finding the lca of two nodes in a binary tree. The next step is using this algorithm to solve lca queries in any tree. This is done by mapping the tree, $T$, to a binary tree $B$, in such a way that lca retrievals in $B$ will help solve lca queries in $T$ in constant-time.

### 4.2 General trees

General trees have to be mapped onto a binary tree, $B$, in a way that keeps enough information about the original tree so that the lca can be determined in the general case. The mapping is fairly simple, although not intuitive. The nodes in $T$ will be numbered by their depth-first numbering, and from hereon we will refer to them by this numbering.

**Definition:** Let $h(k)$, for any number $k$, be the position of the least-significant 1-bit in the binary representation of $k$.

**Definition:** For a node $v$ in $T$, let node $I(v)$ in $B$ be a node $w$ in $T$, such that $h(w)$ is maximum over all nodes in the sub-tree of $v$, including itself.

Since we are using a pre-order numbering for $T$, it can be proven that $I(v)$ is unique for any $v$ in $T$. Thus, all nodes $v$ in $T$ are mapped to nodes $I(v)$ in a complete binary tree $B$.

This mapping can be done in linear-time, using a simple depth-first search and a bottom-up algorithm to compute $I(v)$ for every $v$ in $T$. We use the depth-first search to label $h(v)$ for every $v$ in $T$. Then, we use dynamic programming to find $I(v)$ for every $v$ in $T$. For each $v$, $I(v)$ is set to $I(v')$ such that $h(I(v))$ is maximum over all children of $v$. If $h(v) > h(I(v))$ then we set $I(v)$ to $v$, otherwise we move up in the tree.

$$I(v) = \begin{cases} v, & \text{if } h(v) > h(I(u)) \text{ for all } u \text{ that are children of } v \\ I(u), & \text{if } h(I(u)) \geq h(I(w)) \text{ for all } w \text{ that are children of } v \\ & \text{and } h(I(u)) > h(v) \end{cases}$$

**Definition:** A run in $T$ is a continuous set of nodes such that for any $v$ in the run, $I(v)$ is the same.

During this mapping, two more pieces of information need to be stored about each node in $T$. $L(v)$ will be the head (or lead) of the run containing node $v$ (the highest node mapped to $I(v)$). Also, for each node $v$, in $T$, we need to create $A(v)$, a $\log(N)$ bit number, in which bit $i$ is set to 1 if and only if $v$ has an ancestor in $T$ that maps
Figure 2: A graph, $T$, with 13 nodes, with in-order numbering and functions $h$ and $I$ labeled onto each node.

Figure 3: the mapping of $T$ onto a binary tree $B$. 

The nodes in $T$ that map to that node.
to height $i$ in $B$. We will use $L(v)$ and $A(v)$ to retrieve information that was lost in the mapping from $T$ to $B$. $A(v)$ essentially encodes to what $I$-values $v$’s ancestors map to. We will use this piece of information later.

All $A(v)$ can be calculated in linear-time using dynamic programming, with a simple recurrence. For any $v$, we need the information from its parent and from itself. Thus in $A(v)$ we need to set to 1 the bits that are already set to 1 in its parent, and the bit that corresponds to the height of $I(v)$ in $B$.

$$A(v) = \begin{cases} I[v], & \text{if } v \text{ is the root of the run} \\ A(p(v)) \lor 2^{h(I(v))}, & \text{else, where } p(v) \text{ is the parent of node } v \end{cases}$$

This way, all the preprocessing we have done so far has been linear-time. After this, the actual retrieval of the $lca$ of two nodes in $T$ can be done in constant-time.

Suppose that $lca(x, y) = z$ in $T$. The first step of the process of finding $z$ is finding the $lca$ of $I(x)$ and $I(y)$ in $B$. We will call this node $b$. Node $b$ is the lowest possible $I$-value denoting a run that $z$ can be in. Thus, using $b$, we can find $I(z)$, knowing what run $z$ is in. To find $I(z)$ we use $A(x)$ and $A(y)$.

As an example, we will try to find the $lca$ of nodes 4 and 7 in Figure 2. $I(4) = 4$ and $I(7) = 7$. The $lca$ of these two nodes in $B$ is $b = 4$. $h(b) = 2$.

We look at $A(x)$ and $A(y)$ and find the smallest position $j$ that is greater than or equal to $h(b)$, such that position $j$ is a 1 in both $A(x)$ and $A(y)$. We do this in order to find the lowest height that both nodes $x$ and $y$, in $T$, map to in $B$. Let us call this bit-position $j$. $j$ is $h(I(z))$, which means we know $I(z)$. All we have to do now is find the exact node $z$.

**Definition:** $I(z)$ is the ancestor in $B$ of $lca(I(x), I(y))$ at height $j$, where $j$ is the position of the least-significant 1-bit in $A(x) \land A(y)$, such that $j \geq h(b)$.

For this step of our example, we need $A(4)$ and $A(7)$. I will generate the two $A$-values by just looking at the graph and the definition of $A$ instead of going through the entire dynamic programming. $A(4) = 1100$ because node 4 has 4 ancestors in $T$: 4, 3, 2 and 1 – which map to height 2, 2, 3 and 3 respectively in $B$. $A(7) = 1001$ because node 7 has 4 ancestors in $T$: 7, 6, 2 and 1 – which map to height 0, 3, 3 and 3 respectively in $B$.

To find $h(I(z))$ we now find the smallest $j$ such that both $A(4)$ and $A(7)$ have a 1 in that position and $j > h(b)$. In our case, $j = 3$, which means that $z$ in $T$ mapped to the ancestor of node $b$ in $B$ at height $j$. Thus, the $lca$ we are looking for was mapped to the ancestor at height 3 of node 4 in $B$. In this case, this is node $I(z) = 8$. Thus, we now know that $z$ is one of the nodes that map to node 8 in $B$: 1,2,6 or 8.
The next step is finding \( x' \), the lowest ancestor of node \( x \), such that \( I(x') = I(z) \). We need to look at two cases for this:

1. If \( I(x) = I(z) \). To check this we have to look at the position of the right-most 1-bit in \( A(x) \). Let us denote this by \( p \). If \( p = j \) then \( x \) and \( z \) are in the same run and \( x' = x \), since \( x \) is the lowest ancestor of \( x \) that is in the same run as \( z \).

2. If \( I(x) \neq I(z) \). In this case we find the left-most 1-bit that is to the right of \( j \) in \( A(x) \). This position is the height in \( B \) of run \( I(x') \), such that \( I(p(L(x'))) = I(z) \), where \( p(n) \) is the parent of node \( n \) in \( T \). Essentially, this is the run just under the run in which \( z \) is, on the path from the root to node \( x \). Now, it is easy to see that \( p(L(x')) \) is the lowest ancestor of \( x \) in the same run as \( z \). To find \( I(x') \), we copy the left-most bits in \( I(x) \) up to position \( j+1 \), then add a 1 and completing 0’s. Now we know \( I(x') \), which means we can find \( L(x') \), and thus, we set \( x = p(L(x')) \).

What we are doing in this step is essentially finding the first run that is directly under \( I(z) \) and that is on the path from \( z \) to \( x \). We then take the head of that run, and we now know that the parent of this node is in the same run as \( z \).

In our example, we have \( I(4) \neq I(z) \). Thus we find the left-mode 1-bit that is to the right of \( j \) in \( A(4) \). This is \( j' = 2 \). Now we find \( I(4) = 4 \)'s ancestor in \( B \) at height 2. This is in fact still node \( 4 \). This is the run that is directly under \( I(z) \) and on the path from \( z \) to \( x \). We now take the head of this run, which is node 3 in \( T \). Then we take the parent of this node in \( T \), which is node 2. Node 2 is now a candidate for \( z \).

We do the same for \( y \), finding \( y' \), then we compare the two and we find that:

\[
\text{If } x' \neq y', \text{ then } z = x', \text{ else } z = y'
\]

In our example, \( I(7) = 7 \). The left-mode 1-bit that is to the right of \( j \) in \( A(7) \) is at position \( j' = 0 \). Thus, our \( I(y') = 7 \). The head of this run is in fact node 7, and the parent of node 7 in \( T \) is node 6. This is our second candidate for \( z \). But since \( 2 \neq 6 \), we find that \( z = 2 \). Thus, in our example in Figure 2, we found that \( \text{lca}(4,7) = 2 \).

All of these operations are constant-time, thus, from hereon, I will suppose that given any tree \( T \), we can find the lowest common ancestor of any two nodes in constant-time.

### 5 Series-Parallel Decomposition

The series-parallel decomposition of a series-parallel graph is a mapping of a the graph onto a binary tree, such that the graph can be reconstructed from the tree. The series-parallel decomposition is essentially the binary tree constructed from the steps used to build the \( SPG \).
When recursively putting two series-parallel graphs together, we also connect them in the binary tree by putting them as the left and right children of a node that is labeled either “p” for parallel or “s” for series. While constructing this tree we also label the source and the sink of each node in the decomposition (which also represents an $SPG$). We repeat this until we have decomposed the $SPG$ into simple, one edge graphs.

The easiest way of visualizing this process is through an example. Let us take a simple series-parallel graph and decompose it.

When decomposing a series-parallel graph, the first step is looking at the source and the sink and seeing how the two are connected. In Figure 4, the source, node 1, and the sink, node 5, are connected in series, thus, the root of our tree decomposition will be an “s” node. We will also encode the nodes 1 and 5 from the $SPG$ onto this node in the decomposition. Now we have just split the original $SPG$ into two or more smaller $SPG$’s. In this case, the two $SPG$’s are in Figure 5.

We then recursively repeat the procedure and the decomposition of the top graph (the one on the left in Figure 5) will be the left sub-tree of the current node (in this case the root) and the decomposition of the lower graph (the one on the right in Figure 5) will be the right sub-tree of the node.

In our example, the root of the decomposition will be an “s” node: the left sub-tree will be the decomposition of the graph on the left in Figure 5 and the right sub-tree will be the decomposition of the graph on the right in Figure 5.

The end-result of the decomposition of the graph in Figure 4 is in Figure 6. Nodes in the decomposition that refer to edges in the original graph are not labeled at all. The numbers on the left and right of the nodes represent the sink and the source of the $SGP$ in the original graph. Thus, the root has 1 and 5 as the source and the sink, while the right sub-graph has 2 and 5 as the source and the sink (see Figure 4 and Figure 5).

The interpretation of this graph is that the left and right sub-tree of any node are two $SPG$’s that are connected accordingly to the label of the current node. We will use this graph later to find important information about our $SPG$’s.
6 Distance in $SPG$’s

6.1 Directed case

In a directed $SPG$, we can only move down in the graph. Still, the problem of finding the shortest distance from one node to another in constant-time is not trivial. In a directed $SPG$, transitive closure needs to be considered since some nodes cannot be reached from others. By first answering the transitive closure problem, we can then find the shortest distance between any two nodes in constant-time fairly easily.

Definition: a directed series-parallel graph has the nodes of any sub-$SPG$ directed from the source to the sink.

6.1.1 Transitive closure

Definition: a path from node $x$ to $y$ in a directed series-parallel graph is a sequence of nodes such that there is an edge from each of these nodes to the next one in the sequence. The first node in this sequence is $x$ and the last node is $y$. 

Figure 6: The SP-decomposition of the graph in Figure 4
The transitive closure problem is as follows: given two nodes, $x$ and $y$, can we reach node $y$ from $x$; or: is there a path from $x$ to $y$? In a directed $SPG$, this can be done if, in order to reach the node $y$ from $x$, we do not have to turn backwards in a parallel composition. Now we just have to check whether this is true for our $(x,y)$ pair. We now need to use our SP-decomposition.

What does the $lca$ of two leaves represent in our SP-decomposition? First of all, each leaf in the SP-decomposition represents one edge in the original $SPG$. Thus, the $lca$ of two leaves in the SP-decomposition represents the smallest $SPG$ in which both original edges are included. One edge is in the left sub-graph of the $lca$, and the other is in the right sub-graph of the $lca$.

Also, the label on the node in the SP-decomposition shows how the two $SPG$’s (each of which contains one of the two edges) were combined.

Thus, when checking the transitive closure of $x$ and $y$ we have to pick two edges – one that has the node $x$, and one that has the node $y$ – in order to find the two sub-graphs that include the nodes. Now, if these two sub-graphs are connected in series, then $y$ can be reached from $x$ if $x$ is in the left subgraph of the $lca(x,y)$ in the SP-decomposition; or if $x$ is in the right subgraph of the $lca(x,y)$ in the SP-decomposition, then $x$ can be reached from $y$. But if the two sub-graphs are connected in parallel, then $y$ cannot be reached from $x$ (and we cannot calculate a shortest path).

Using Figure 7, we will look at going from node 2 to node 7. We now pick an edge that contains node 2, for instance edge 1-2, and an edge that contains node 7, for instance edge 7-8. We now find the $lca$ of these two edges in the SP-decomposition (the edges represent leaves in the SP-decomposition). The $lca$ in this case is the root of the decomposition (see Figure 8). Since this is an “s”-node, we know that the $SPG$ that contains node 2 and the $SPG$ that contains node 7 are connected in series, thus, there is a path from node 2 to 7. It is also important to note that node 2 is in the left sub-tree of the $lca$, meaning that it is higher than the right sub-tree, which contains node 7. Thus, we can go from node 2 to 7, but not the other way around.

It is also important to note that we can choose any two edges that contain the nodes and we will still get the same result. There is one exception to this, when at least one of the nodes is the source or sink of a “p”-sub-graph of the original graph. In this case, we might get that their $lca$ is a “p”-node. Here we have to check whether one of the nodes is not the source or sink of the sub-graph. If it is, then there is a path between the two nodes, otherwise there is no path.

Figure 7: A slightly more complex $SPG$
An example of this in Figure 7 involves the nodes 2 and 4. There is clearly a path from 2 to 4, but if we pick the edges 2-4 and 3-4, these are connected in parallel, and the lca of these two edges in the SP-decomposition is a “p”-node, with the source 1 and sink 4. But since 4 is the sink of this “p”-sub-graph, we conclude that there is a path from 2 to 4.

6.1.2 Distance

In order to find the distance from $x$ to $y$, we need to use some sort of preprocessing that will enable us to find intermediate distances in constant-time. The way to do this is to use two breadth-first searches, one from the source of the SPG, and one from the sink of the SPG (considering the directions on the edges to be backwards). We form two BF-trees from these breadth-first searches, as seen in Figure 9 and Figure 10, and we will use these to find certain distances in constant-time. We label the distance from the root of the tree to any node onto that node.

Now, we use the SP-decomposition (Figure 8) again to find the node that is at the intersection of the sub-graphs that contain the $x$ and $y$. Without loss of generality, suppose that we can reach $y$ from $x$, otherwise we would not be making these calculations. We know that if there is a path from node $x$ to $y$ then this node, which I will call $z$, is a node that connects two different SPG’s in series. It is important to note that because this is a series-node, in order to go from $x$ to $y$, we have to pass through $z$. 
The process of finding node $z$ is fairly simple. We use the SP-decomposition: when the $lca$ we found is an “s”-node, the node $z$ will be the sink of the left-node in the SP-decomposition (which is also the source of the right-node), since this is the node in the original $SPG$ that connects the two $SPG$’s that contain $x$ and $y$. If the $lca$ we found is a “p”-node, and $x$ is the source of the sub-graph in the decomposition, then $z = x$, otherwise, if $y$ is the sink of the sub-graph in the decomposition, then $z = y$. If $x$ is not the source and $y$ is not the sink of the sub-graph, then there is no path from $x$ to $y$.

In our example let $x = 2$ and $y = 7$. Now, in order to find $z$, we need to find the $lca$ of two leaves in the SP-decomposition, one containing node 2 and one containing node 7. Using the same example as for the transitive closure, we pick two edges, such as edge 1-2 and edge 7-8, which are both leaves in the SP-decomposition ($Figure 8$). We now find the $lca$ of these two leaves in the SP-decomposition. Their $lca$ is the root of the tree which, is an “s”-node, thus we know that node 7 can be reached from node 2.

To find node $z$ in the original $SPG$ we only need to look at either one of the children.
of the lca we found. If we look at the left child, z will then be the sink of that node otherwise, it will be the source of the right child. In this case we find $z = 4$. We now know that in order to go from node 2 to node 7 we need to pass through node 4.

Since node $z$ connects two SPG’s in series, we are going to split up the distance from $x$ to $y$ into the distance from $x$ to $z$ and the distance from $z$ to $y$. Keeping in mind that node $z$ connects two SPG’s in series, we are going to find both of these distances in constant-time. I will look at the distance from $x$ to $z$, since the distance from $z$ to $y$ is symmetrical if we look at the other BF-tree.

We are actually going to find the distance from $z$ to $x$ in the BF-tree starting at the sink of the original graph (Figure 10). All we need to do to find the distance from $z$ to $x$ is subtract the distance from the root of the BF-tree to $z$ from the distance from the root of the BF-tree to $x$, which we both already have calculated and labeled. Let us call the root of the BF-tree $r$. This is the correct distance because node $z$ in the original SPG connects the largest sub-SPG containing the sink of the graph (which is the root of the BF-tree) with the largest sub-SPG containing node $x$ in series. Thus, in order to go from the sink of the original SPG to node $x$ we need to go through node $z$. Similarly, to go from the root of the BF-tree (Figure 10) to node $x = 2$, we need to go through node $z = 4$.

We know that the distance from the root of the BF-tree to node $z$ is correct, and that the distance from the root to node $x$ is correct. Now we also know that the distance from the root to node $x$ contains node $z$. Thus:

$$\text{dist}(z, x) = \text{dist}(r, x) - \text{dist}(r, z)$$

Since for any node $n$ we can get the distance $\text{dist}(r, n)$ in constant-time, we can also get the distance from $z$ to $x$ in constant-time.

In our example, the distance from node 4 to node 2 is:

$$\text{dist}(4, 2) = \text{dist}(8, 2) - \text{dist}(8, 4) = 4 - 3 = 1$$

Thus we have the upper part of the distance from $x$ to $y$ through $z$. Similarly, we can find the distance from $z$ to $y$, by using the other BF-tree that we started at the source of the original SPG (Figure 9). In our example, we find that the distance from node 7 to node 4 is:

$$\text{dist}(7, 4) = \text{dist}(1, 7) - \text{dist}(1, 4) = 4 - 2 = 2$$

Note that the distance from $x$ to $z$ might not be correct in the BF-tree that started from the source of the original graph (Figure 9), and, conversely, the distance from $z$ to $y$ might not be correct in the other BF-tree (Figure 10).

Now that we have the distances from $x$ to $z$ and from $z$ to $y$, all we have to do is add these two in order to find the shortest distance from $x$ to $y$. Thus, we have solved the problem for the directed case.
6.2 Undirected case

In the undirected case, finding the shortest path from x to y is a much harder problem and the fastest solution we could find in this paper is an $O(N \cdot \log(N))$ preprocessing with queries taking $O(\log^2 N)$ in the worst case.

6.2.1 Preprocessing

The undirected case also uses the SP-decomposition of the SPG, and in fact, we will use this tree much more in the undirected case. We will use a slightly different version of the SP-decomposition, where “s”-nodes that are only series of nodes (no parallel SPG’s in the series), will be one node in the decomposition, with a number on it, which will represent the number of nodes in that series. For each node in the original
graph, we will also note which “s”-node it can be found in, and which one it is from the top of the series. This will take $O(N)$ space.

In this case it is important to note that for “s”-nodes, the children are ordered from the left to the right as they are from the top to the bottom in the original graph; and for “p”-nodes, the children are ordered from the left to the right as they are from the left to the right in the original graph. Also, since our lca algorithm works for any trees, we can still find lca’s in our SP-decomposition.

In order to answer queries quickly, the next preprocessing step we will need is finding the distances between any two nodes in the original SPG (Figure 11) that are consecutive “p”-nodes in the SP-decomposition. This can be easily done in parallel by running a BF-search from each of these nodes. This also allows us to find the shortest path (in the original SPG) between the source and the sink of each “p”-node, which we will also use later.

There are two more preprocessing steps that will help us solve queries quickly. The first one is a BF-search from the root of the SP-decomposition, where we only count “p”-nodes as actual nodes. Essentially we want to know how far down each “p”-node is in the SP-decomposition. We will use this to do binary search on the “p”-nodes in order to find the distance from any “p”-node to any other one in the SP-decomposition. The last step of preprocessing is generating a matrix of size $N \cdot \log(N)$, with the information:

$$A[i, x] = \text{the } 2^i\text{-th “p”-node ancestor of “p”-node } x \text{ in the SP-decomposition}$$

This matrix will allow us to find the $k$-th “p”-node ancestor of any “p”-node in the SP-decomposition in $\log(H)$ time, where $H$ is the height of the SP-decomposition. This will allow us to find the shortest path from any “p”-node to any other “p”-node in $O(\log^2(H))$ using binary search. Calculating matrix $A$ can be done in $O(N \cdot \log(N))$ time using the recursion:

$$A[0, x] = p(x), \text{ where } p(x) \text{ is } x\text{’s parent}$$

$$A[i, x] = A[i - 1, A[i - 1, x]]$$

### 6.2.2 Querying

To answer queries in $\log^2(H)$ time we need to first find the lca of nodes $x$ and $y$ in our SP-decomposition. In Figure 11, node $x$ is node number 1 in the “s”-node marked as $x$, and node $y$ is the $15^{th}$ node in the “s”-node marked as $y$. The lca of these nodes is the “p”-node that has $so$ and $si$ as its source and sink (see Figure 12).

If the lca is a “p”-node, we know that the path from $x$ to $y$ has to pass either through node $so$, node $si$, or both. If the lca is an “s”-node, the algorithm is not much more difficult, there are just more cases to explore.
We will look at the case if the lca is a “p”-node. Keep in mind that we know the shortest path from node so to si (we have it from the preprocessing part). Now the problem has been reduced to finding the minimum value of the following 4 sums:

\[
\begin{align*}
&\text{dist}(so, x) + \text{dist}(so, y) \\
&\text{dist}(si, x) + \text{dist}(si, y) \\
&\text{dist}(so, x) + \text{dist}(si, so) + \text{dist}(si, y) \\
&\text{dist}(si, x) + \text{dist}(si, so) + \text{dist}(so, y)
\end{align*}
\]

If the lca had been an “s”-node, the problem would still be the same, except the nodes so and si would be different. The complexity of the problem, on the other hand, would stay the same.

Finding the distances from x to so and si and from y to so and si is the same algorithm so I will only look at one of these cases. I am going to find the shortest distance from node x to nodes so and si. The first step is to find the smallest “p”-node that contains node x. This will always be the parent of the “s”-node containing x. In this case it is the lowest “p”-node in Figure 12. To go from node x to so, we will need to pass through the source or the sink of this “p”-node, or possibly both. First we find the
distances from $x$ to the source and the sink of this “p”-node, which can be done in $O(1)$, since we know what number node $x$ is in the series. In this case, the distance to the source is 1 and the distance to the sink is 10.

Next, we are going to find the distance between any two “p”-nodes in the SP-decomposition in $\log^2(H)$ time. When finding this distance between two “p”-nodes, I am referring to finding all 4 distances: from source to source, from source to sink, from sink to source and from sink to sink.

To find these distances we are going to use binary search: if the distance between the two “p”-nodes in the SP-decomposition is 1 (they are adjacent), then we have the four distances from preprocessing; if the two “p”-nodes are not adjacent, we will divide the distance into two parts, and calculate each in parallel.

Without loss of generality, let the “p”-node that is higher be $p_1$ and the one that is lower be $p_2$.

Since we are only looking at “p”-nodes at the moment, all we have to do is find how many “p”-nodes there are between $p_1$ and $p_2$. Since we know that $p_1$ is on the path from the root to $p_2$, we can find this number by subtracting the distances from the root of the two nodes, that we preprocessed in our BFS on the SP-decomposition. Let this number be $q$, and let $r = q \text{ div } 2$.

Now we are going to find the $r^{th}$ ancestor of $p_2$, which we will call $p_3$. We are essentially doing binary search on the “p”-nodes in the SP-decomposition. This can be done in $O(\log(H))$ using the $A$-matrix which we preprocessed.

We now recursively find the distances from $p_1$ to $p_3$ and from $p_3$ to $p_2$. Finally we get that:

$$\text{dist}(p_1, p_2) = \text{dist}(p_1, p_3) + \text{dist}(p_3, p_2)$$

It is important to note that we have to call each half of the recursion on a new processor, otherwise the complexity of this part of the query is $O(N \cdot \log(N))$. But if we call each recursion on a new processor, we can calculate each of the steps in parallel (since they are completely independent), in $O(1)$. Thus, the total time is the number of steps, which is $O(\log(H))$. The total time for each query is then $O(\log^2(H))$, since we have to find the $r^{th}$ ancestor at each step at least once.

Our example is not great for showing this case, since the distance between our two “p”-nodes is one, so a more interesting example can be seen in Figure 13.

Suppose we want to find the distances from $p_1$ to $p_7$. Let us call our recursive function $\text{distr}(i, j)$, where $i$ and $j$ are the numbers of the two “p”-nodes.

We call $\text{distr}(1, 7)$. We then find the distance from $p_1$ to $p_7$ to be $q = 6$, and we find $r = 3$. We now find the 3rd ancestor of $p_7$, which is $p_4$. So far we have done $O(\log(H))$ work. We now call $\text{distr}(1, 4)$ and $\text{distr}(4, 7)$ in parallel. We continue doing so:

$$\text{distr}(1, 4) = \text{distr}(1, 2) + \text{distr}(2, 4)$$
$$\text{distr}(4, 7) = \text{distr}(4, 5) + \text{distr}(5, 7)$$
Figure 13: An example of an SP-decomposition of an even larger $SPG$
Each of these steps takes $O(\log(H))$, but since they are done in parallel, this entire step takes $O(\log(H))$. Recursively, we then call:

\[
distr(2, 4) = distr(2, 3) + distr(3, 4)
\]
\[
distr(5, 7) = distr(5, 6) + distr(6, 7)
\]

Again, this entire step takes $O(\log(H))$ in parallel. We now have all the distances, since we have preprocessed all $distr(x, x+1)$. Thus, since we have at most $\log(H)$ of these parallel steps, the total complexity of each of these queries is $O(\log^2(H))$.

It is easy to see how this way of finding distances of long series of "s"-nodes divides this distance in half each time and works from there. Intuitively, this can we the explanation as to why the number of steps necessary to find the distance between two points is proportional to the logarithm of their distance (which is, at most, $N$). It is also important to note that we are using a total of $1+2+4+...+H$ processors at most, which is approximately $2^H$ processors.

Note: There is also a different way of going about this to solve queries in $O(\log(H))$. The above solution is the idea of the author, but there is also another way of finding the distance between any two points in a linked list in $\log(H)$ time, where $H$ is the height of the list. Since we need to find the distances from the $lca$ to a node somewhere in the tree lower, we could use “pointer jumping” to further better the algorithm.

7 Overview and Complexity Analysis

7.1 Directed Case

Let $S$ be our $SPG$ and $x$ and $y$ be the nodes given in the problem. Here are the steps required to solve the directed case algorithm:

7.1.1 Preprocessing

1. Get the SP-decomposition of $S$, with the appropriate labels on. This can be done in $O(N)$.

2. Do a breadth-first traversal of $S$ from both the source and the sink, labeling the distances from the root of the BF-tree to each node. This can also be done in $O(N)$.

The total time for preprocessing the directed case is $O(N)$.

7.1.2 Querying

1. Pick any two edges in $S$, one containing $x$ and the other containing $y$. Time: $O(1)$. 

20
2. Find the \textit{lca} of the leaves representing these two edges in the SP-decomposition. Time taken: $O(1)$.

3. Check the transitive closure of $x$ and $y$. If $y$ cannot be reached from $x$, we are done, else go to step 4. Time: $O(1)$.

4. Find node $z$, which will be the node (in $T$) connecting the two sub-graphs of the \textit{lca} found at step 2. Time: $O(1)$.

5. Find the distance from $x$ to $z$ in one BF-tree and the distance from $z$ to $y$ in the other one by subtracting the distances from the root. Time: $O(1)$.

6. Add the two distances to find the distance from $x$ to $y$.

The total time for each query is $O(1)$. Thus, for the directed case, we can do preprocessing in $O(N)$ and then answer distance queries in $O(1)$.

### 7.2 Undirected Case

Again, let $S$ be our \textit{SPG} and $x$ and $y$ be the nodes given in the problem. Also, let $H$ be the height of $S$. Here are the steps required to solve the undirected case:

#### 7.2.1 Preprocessing

1. Get the SP-decomposition of $S$, with the appropriate labels on. This can be done in $O(N)$.

2. Do a breadth-first traversal of $S$ in parallel starting at each node that is the source or sink of any \textit{sub-SPG} that is in parallel. This can also be done in $O(N)$ and requires at most $2 \cdot N$ processors.

3. Label onto each “p”-node in $S$ the distance from its source to its sink. This can be done during step 2.

4. Generate the matrix $A$. This takes $O(N \cdot \log(N))$ time.

5. Do a breadth-first traversal of $S$ and label onto “p”-nodes how many “p”-nodes there are from the root of $S$ to that node. This can be done in $O(N)$.

The total time for the preprocessing for the undirected case is $O(N \cdot \log(N))$. 

21
### 7.2.2 Querying

1. Pick any two edges in \( S \), one containing \( x \) and the other containing \( y \). Time: \( O(1) \).

2. Find the \( lca \) of the leaves representing these two edges in the SP-decomposition. This is node \( z \). Time taken: \( O(1) \).

3. Find \( distr(z,x) \) and \( distr(z,y) \) recursively. Each \( distr(a,b) \) will have 4 subcategories: the distance from the source of \( a \) to the source of \( b \), the distance from the source of \( a \) to the sink of \( b \), the distance from the sink of \( a \) to the source of \( b \), the distance from the sink of \( a \) to the sink of \( b \). For each \( distr(a,b) \):
   
   - Find the mid-point between \( a \) and \( b \), called \( c \). Time: \( O(log(H)) \).
   
   - \( distr(a,b) = distr(a,c) + distr(c,b) \), calling each half on a different processor. This means we will repeat these steps \( O(log(H)) \) times. Total time: \( O(log^2(H)) \) parallel.

4. \( distr(x,y) = distr(z,x) + distr(z,y) \). Then we just look at the shortest of the four possible paths, and we have our shortest path from \( x \) to \( y \). Time: \( O(1) \).

The total time for each query is \( O(log^2(H)) \). Thus, for the undirected case, we can do preprocessing in \( O(N \cdot log(N)) \) and then answer distance queries in \( O(log^2(H)) \).

### 8 Conclusion

The goal of this paper was to find a quick algorithm that can find distances in series-parallel graphs. We looked at preprocessing algorithms that yield quick results on queries. First we looked at a fast algorithm for finding the lowest common ancestor of two nodes in a tree. We then looked at a way to use this by mapping a series-parallel to a tree, called the series-parallel decomposition.

Finally we looked at both the directed and undirected cases, and found quick solutions to both. This way, we know we can find the distances between all \( N^2 \) nodes in a directed series-parallel graph in \( O(N^2) \) time, and in \( O(N^2 \cdot log^2(N)) \) time in an undirected series-parallel graph (or \( O(N^2 \cdot log(N)) \) if we use pointer jumping).

Overall these results are encouraging, since \( SPG \)'s are the only graphs with a tree-width of 2, which is one step up from trees, and one step closer to bounded tree-width.
Bibliography


2. Fang Wei, 2010, *TEDI: Efficient Shortest Path Query Answering on Graphs*