Chapter 9

Strings

9.1 Pattern Matching

9.1.1 KMP

9.1.2 Z-func

9.1.3 Multi-pattern Matching/Aho-Corasick

9.2 Finding Minimum Rotations

9.2.1 Kevin's fancy rotation algorithm

9.2.2 Lyndon Decomposition/Duval's Algorithm

9.3 Palindromic Substrings: Manacher's Algorithm

9.4 Regexes

9.5 Suffix Trees

A suffix tree is a structure where every path from the root to a leaf encodes some suffix of an input string, and all suffixes are encoded along some such path. It can be used to solve several problems, some of which are covered earlier in the chapter, including:

1. Checking if a pattern exists in a string
2. Finding the location of all instances of a pattern within a string
3. Finding the lonest repeated substring within a string

4. Finding the lonest common substring (LCS) between two input strings
5. Finding the longest palindromic substring within an input string
6. Finding the minimum rotation of an input string
7. Constructing the suffix array

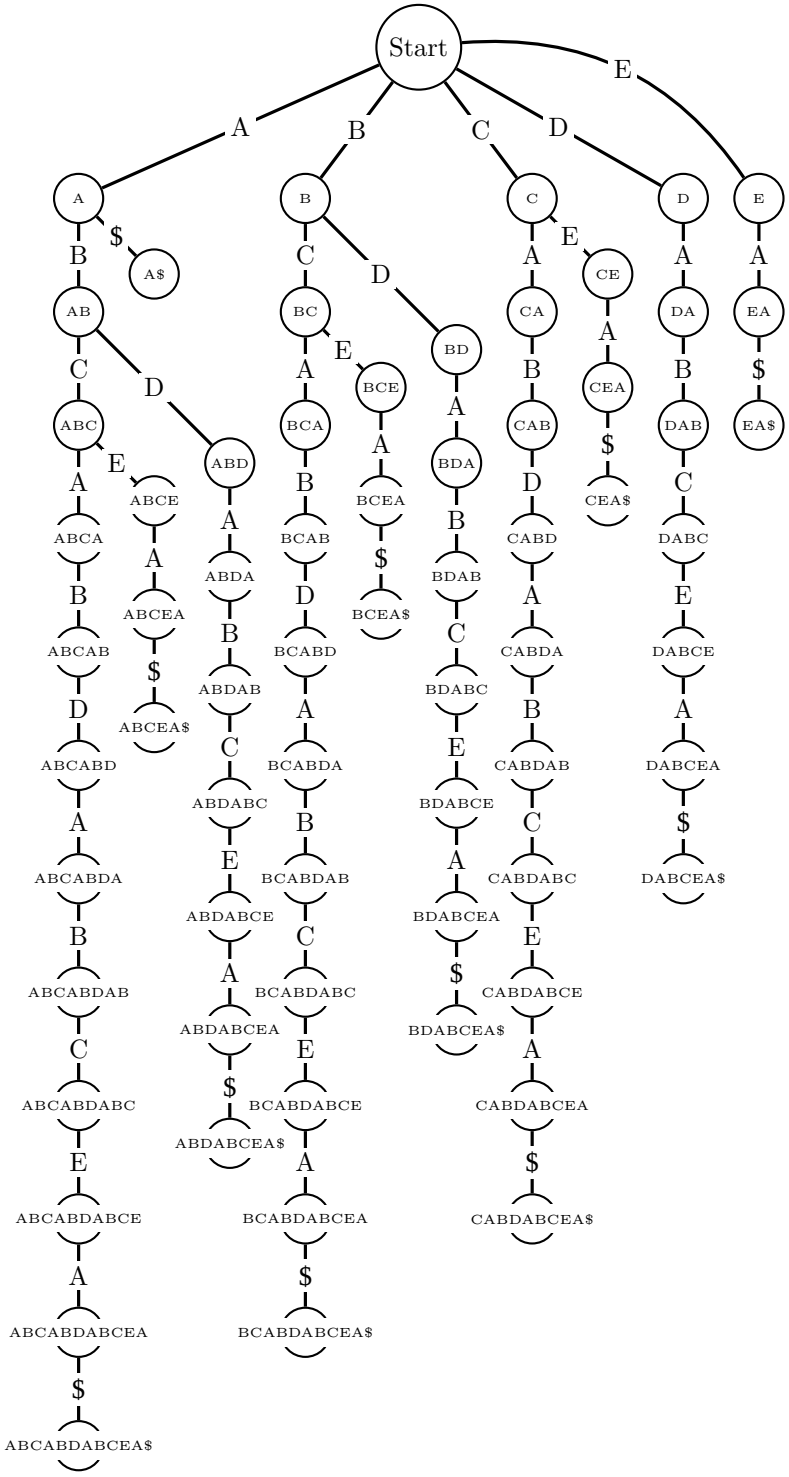
9.5.1 Building Suffix Trees

In this section, we will present a naive suffix tree, demonstrate how, through a series of optimizations, we can operate on, and then build it, in linear time. Following sections will demonstrate how to use it to solve the above listed problems. Throughout, we will use the following string in our demonstrations:

ABCABDABCEA

The Naive Tree

In the naive tree, each edge emenating from a node is labeled by a letter, which identifies the next node along the path. The tree for the above string looks as follows:


Start
A
B
C
D
E
A
$
A$
B
AB
C
D
ABC
E
ABD
A
ABCA
ABCE
ABDA
B
ABCAB
A
ABCEA
B
ABDAB
D
$
C
ABCABD
ABCEA$
ABDABC
A
E
ABCABDA
ABDABCE
B
A
ABCABDAB
ABDABCEA
C
$
ABCABDABC
ABDABCEA$
E
ABCABDABCE
A
ABCABDABCEA
$
ABCABDABCEA$
B
C
BC
E
BCE
D
BD
A
BCA
A
BDA
B
BCAB
BCEA
BDAB
D
$
C
BCABD
BCEA$
BDABC
A
E
BCABDA
BDABCE
B
A
BCABDAB
BDABCEA
C
$
BCABDABC
BDABCEA$
E
BCABDABCE
A
BCABDABCEA
$
BCABDABCEA$
C
A
CA
E
CE
B
CAB
A
CEA
D
$
CABD
CEA$
A
CABDA
B
CABDAB
C
CABDABC
E
CABDABCE
A
CABDABCEA
$
CABDABCEA$
D
A
DA
B
DAB
C
DABC
E
DABCE
A
DABCEA
$
DABCEA$
E
A
EA
$
EA$

Several notes about the tree:

- Any prefix can be found by starting at the root and traversing to a leaf node following the edges labeled by the next character in the string.
- Nodes are labeled for convenience, giving the path followed to arrive at that node from the root.
- The terminal charachter **$** must be included at the end of the string. Otherwise, we would not know for sure whether a suffix completes at a given node. Consider the node **A**. Without the terminal character, we would not know that there is a suffix which ends there. Instead, we see a path labeled with **$**, and can know conclusively. The exact character used doesn't matter all that much, so long as we can guarantee it is a character not contained in the input string.
- As a consequence of the previous bullet, for a string length N, there are exactly N leaf nodes.

The tree can be constructed by looping over each suffix of the input string and traversing the existing tree. If, when at a node, there is no path leaving the node with the next letter of the string, we create a new node, and add the edge before traversing to it. As this is not a binary tree, instead of having left and right children, we have potentially one edge for each letter of the alphabet and the terminal character. We can store the node at the other end of these edges in either a map, or an array, indexed using the ASCII character.

Even though this tree is correct and could be used to solve the prior listed problems, it will never be able to do so, nor even be constructed, in linear time. There are simply too many nodes. Note the long branchless chains in the tree which are repeated multiple times. It turns out, worst case, the number of nodes in this tree is $O(n^2)$. This can be seen by constructing the tree for a string with no repeated letters (say, ABCDEFGH).

Optimization 1: Compression

The first optimization involves creating fewer nodes. Namely, we observe that instead of creating a node for each letter in the suffix, we might create nodes at only places where there is a split in the tree. In such a situation, an edge may contain multiple letters, instead of just a single one, even if we still index it by only the first letter. Lets see how the creation of such a tree works.

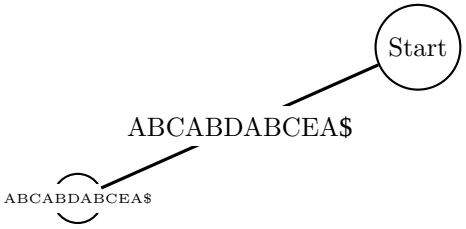


Figure 9.1: Step 1: We add the first suffix to the tree. There will necessarily be no splits in the tree, so there will be a single edge with the entire suffix leading to the only leaf node.

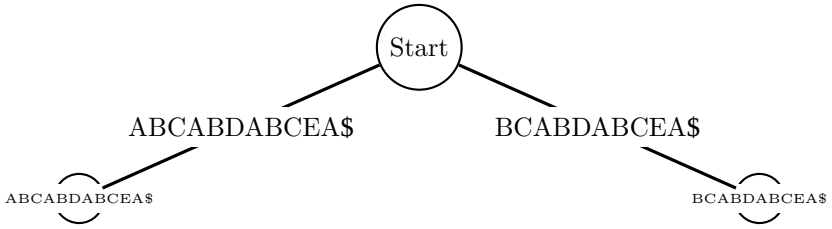


Figure 9.2: Step 2: We add the second suffix to the tree. There is no overlap with the previous edges, so it ends up juts being a single edge.

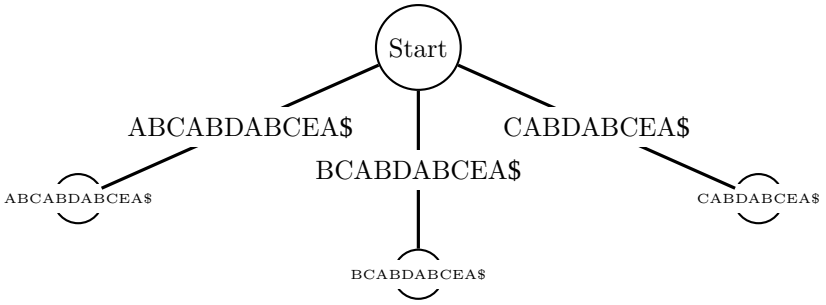


Figure 9.3: Step 3: The same process as step 2.

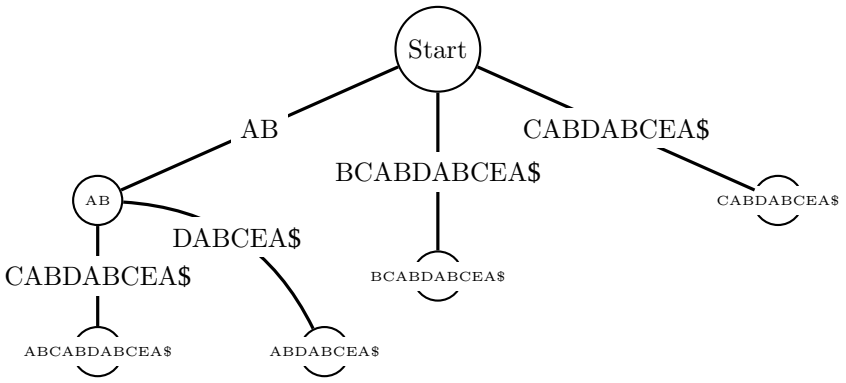


Figure 9.4: Step 4: When adding the suffix ABDABCEA$, we see that an edge beginning with A already exists at the root. So we compare the letters in that edge with our suffix until we find a difference. When we compare the C in the edge with the D in the new suffix, we find they differ. We create a new node at the point where they differ, and create two new edges. Note that if we happen to come across an intermediate node while in the process of comparing, we simply procress down the appropriate edge from that node based on the next character, as will be the case when we insert ABCEA later. We will compare A and B, arrive at a node, then progress down the CABD... edge to compare the remaining letters.

As we've now seen how to add new suffixes, we'll skip to the completed, compressed tree.

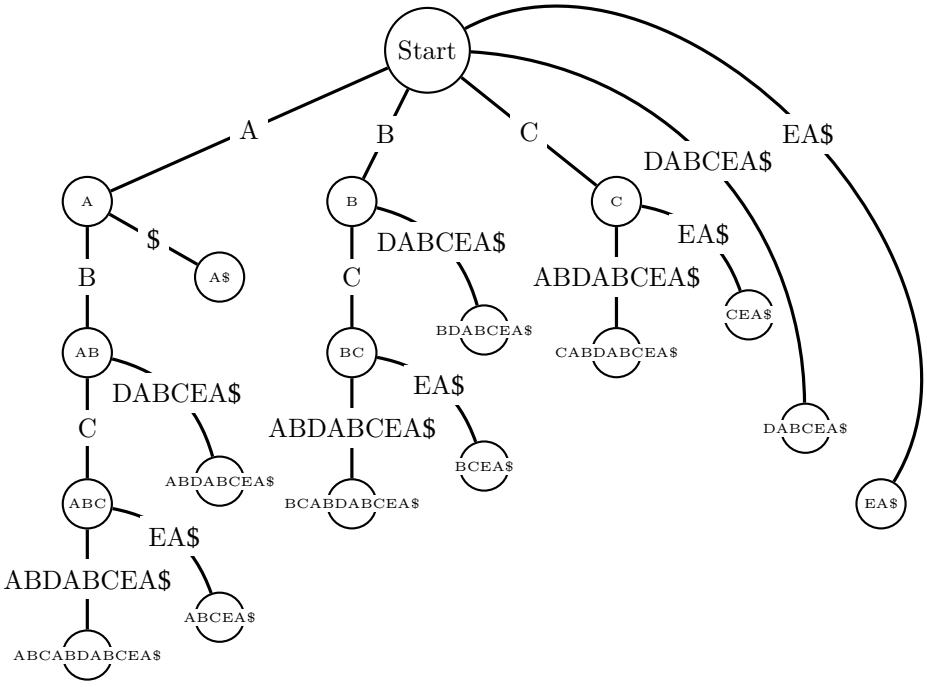


Figure 9.5: The completed, compressed suffix tree. Note that it contains all the same suffixes as the previous suffix tree, but with far fewer nodes.

After the creation of this compressed form, we note that there are far fewer nodes, but exaclt how many fewer? As noted before, we have exactly N leaf nodes. We note here, though, that every internal node, we create a subtree which contains at least two leaf nodes which were not previously in the same subtree. This limits the total number of interior nodes to a maximum of $N-1$, limiting the total number of nodes in the tree to $2 * N - 1$, which is $O(N)$: a reduction by a factor of N from the uncompressed form, even in the worst case.

Despite this, the construction still may be too slow. Consider, again, the worst-case tree with all unique letters. In that case, we are still left with $O(N)$ branches, which each must store $O(N)$ characters, making even the storing the of the edge data super-linear.

Optimization 2: Indexing

The next optimization involves avoiding the cost of copying and storing the multitude of characters on each edge. We notice that the string on each edge comprises of some substring of our input. Given that, instead of storing the substring itself, we can simply store a pair of indices pointing to the input

string. These indices indicate the start and end of the substring which represents that edge. If multiple such substrings exist, any may be used. We reprise the compressed tree, now including such indices. The substring itself is noted only for clarity, it is not copied or stored with the edge.

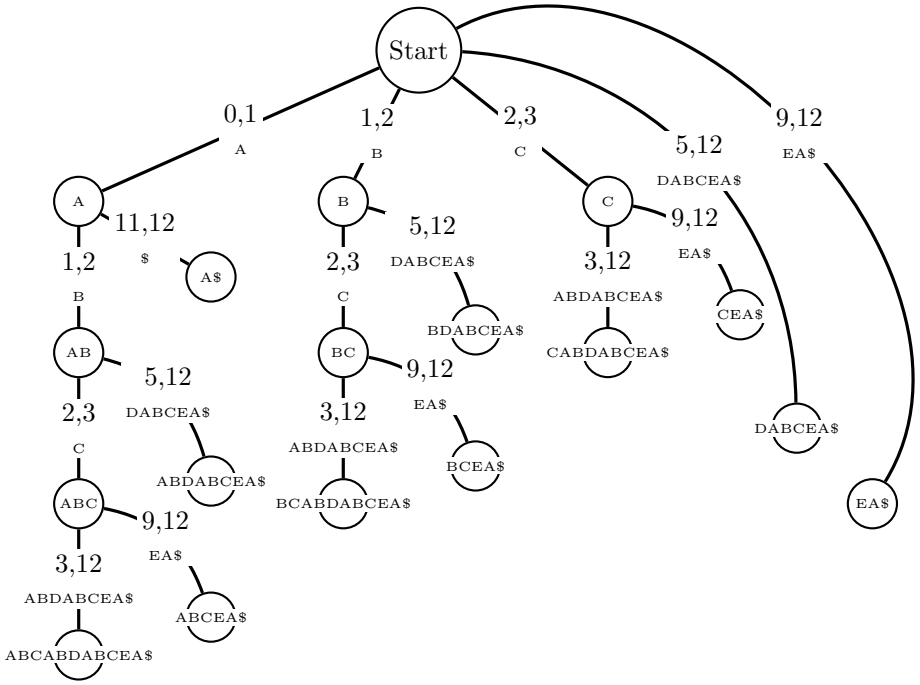


Figure 9.6: Same as before, but with the stored indices. The substrings those indices resolve to, as well as the substrings representing each node, are only presented for clarity.

Optimization 3: Per character, not per suffix

Up until now, we have dodged a fundamental issue with our methods in constructing the tree. By iteratively adding each distinct suffix of our input to a tree, we run into a wall. Fundamentally, there are $O(N)$ suffixes, and in the worst case, each one may require $O(N)$ comparisons (consider a highly repetitive string such as AAAAAAAAAAB to see why this is the case). In order to construct a tree in linear time, we must find a way to do so in a small number of, if not single pass over the input string.[1]

[1] There are algorithms that use a suffix-first approach, but they are much more conceptually difficult. See: Weiner's algorithm.

The key intuition is that instead of iterating over each suffix and adding it to the tree, we can build the suffix tree one character at a time. We will create a complete suffix tree for the substring encompassing the first character, then we will extend the input string one character at a time, and attempt to updatte the tree to accommodate it. Then, once we have reached the $ character, we will have successfully build the suffix tree for the complete input.

We will present a few iterations of building a tree with this optimization, but will not construct the full tree, as we'll see there are a few other "problems" we must solve in order to form a complete algorithm.

Here's how it works in practice.

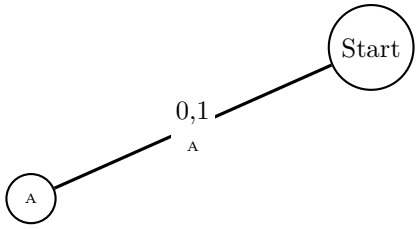


Figure 9.7: Step 1: After processing the first A, we have the single substring that ends at that character encoded in the tree. [A]

In order to add the second character, B, we must know at what nodes terminated a substring for the previous character. After the first step, this is only node A, which is noted in square brackets in the caption. We'll refer to these as *open* substrings. While we represent these based on their label here, this could feasibly use any pointer to the node, such as an index.

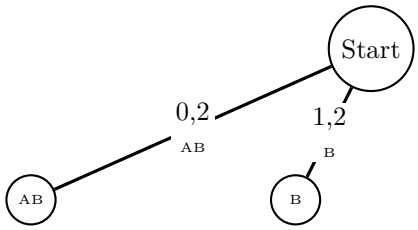


Figure 9.8: Step 2: In order to process the first B, we add a B to the open substring from the previous step by changing the label from (0,1) to (0,2) to indicate the extrac character, and then add B from the root. We now have two open substrings. [AB,B]

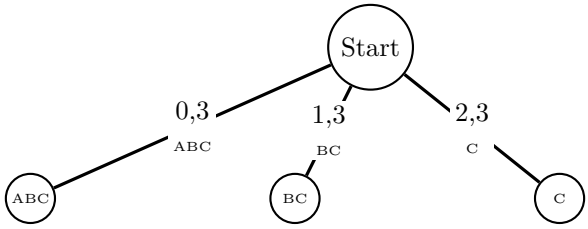


Figure 9.9: Step 3: Similar to step 2, we update (0,2) to (0,3), and then update (1,2) to (1,3), and add the new open substring to the root. [ABC,BC,C]

We'll pause here.

While we've demonstrated creating the tree by character, instead of by suffix, we also see that the construction still has a flaw that will ultimately force us into super-linear time. As we add each character, we increment the number of open strings, and for each new character, we have to update the number of open strings. We note that this most commonly results in incrementing the "end" value of each of the applicable edge labels. Without finding a way to eliminate these updates, we will be unable to reach linear time.[2]

Optimization 4: Open Ended Substrings

As we were updating the open substrings, we note that the value we were updating the labels to always ended up being to the same index, representing a substring ending at the character we just added. Instead of using a fixed value for the end index of these open substrings which we must increment each step, we can instead use a floating value which simply represents *a substring ending at whatever the previous character we added to the tree.* We will use a ? to represent this floating index.

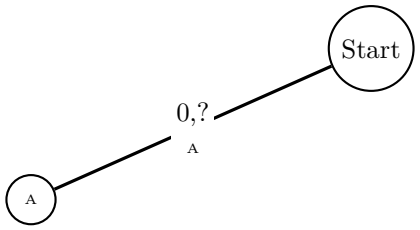


Figure 9.10: Step 1: After processing the first A, we have the single substring that ends at that character encoded in the tree. Note that we indicate its end with the floating index to indicate this edge is still open.

[2]If you're following the formal description of Ukkonen's algorithm, this is Rule 1.

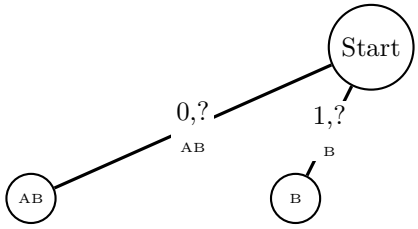


Figure 9.11: Step 2: To add the B, we only added a new open substring to the root. Note that the labels on the edge (0,?) did not change at all, but the value represented by that edge did, since the index which ? represents incremented by one. We have implicitly updated 2 edges worth of substrings while only physically updating one of them.

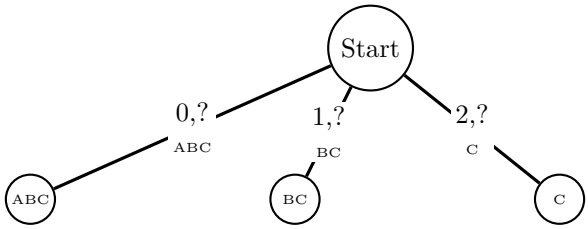


Figure 9.12: Step 3: Similar to step 2, we have only physically added a single edge to the tree, but the remaining open edges have been implicitly updated due to the floating endpoint.

We now would run into an issue when we attempt to add the second A. The A edge already exists, and without "cheating" we won't know if we are supposed to split the edge there or not, as it ends up depending on future characters. In order to allow us to efficiently cope with this, we use a non-commital suffix.

Optimization 5: Non-Committal Suffixes

The fundamental problem when we inserted the second A into the suffix tree was that we did not know if the insertion of the A would cause the tree to split. If the next character is the same, we do not want to split, but if it is not, we must. However, we cannot look arbitrarily ahead in the word without violating our per-character paradigm. We must come up with a way to encode that we have an open substring which may or may not result in a split in the edge while also not looking ahead in the string.[3]

This is accomplished by creating a special open suffix, a pointer into the tree indicating the point on a specific edge we will attempt to insert the next

[3]Splitting an edge in this case is rule 2 in the canonical description of Ukkonen's algorithm.

character. This is called the *active point*[4], and is represented using 3 values:

- the parent node of the edge containing the active point
- the character representing the specific edge from the parent node which contains the active point
- the specific character on that edge which is the active point, represented by the number of characters along that edge the active point occurs

When we reach a character we are attempting to insert into the tree, we first check if the character succeeding the active point matches. If it does, we move the active point, otherwise, we split the edge in question. The reset point is initialized to point at the root node, covering the cases when we have not seen repeat letters.

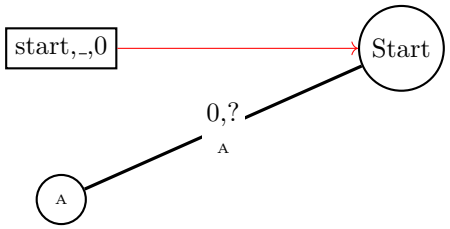


Figure 9.13: Step 1: Inserting the A is identical to before, however we note that we now have an active pointer pointing to the start node.

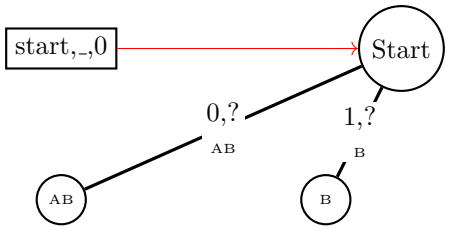


Figure 9.14: Step 2: Inserting the B is also identical to before.

[4]The active point allows us to encode rule 3 from the canonical description of Ukkonen's algorithm

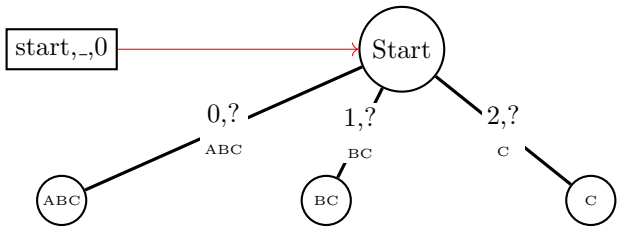


Figure 9.15: Step 3: Inserting the C is also identical to before.

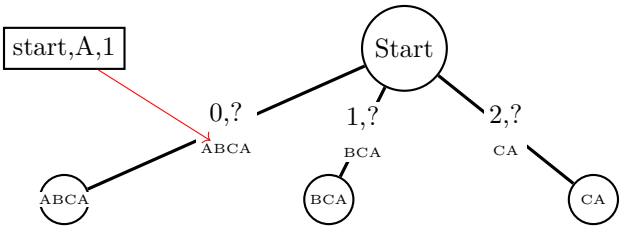


Figure 9.16: Step 4: When we go to insert the second A, we note that we already have a path for A. We don't know, however, whethether we will have to split the edge (without cheating by looking ahead). We modify the active point to the A on the appropriate edge, which is the first character on the A branch coming from the start node.

So far so good. All 4 of the suffixes are either represented by open edges, or the active point.

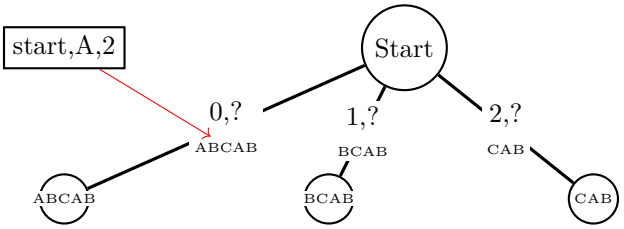


Figure 9.17: Step 5: When we go to insert the second B, we see that the next character after the active point is already a B, so we simply move the active point, in this case, by incrementing the number to 2 (to indicate the second character on the A edge from start).

Despite seemingly going swimmingly, we find we have a problem. Unlike after step 4, we no longer have all the necessary suffixes in the tree. We have, of course, the 3 open edges and the active point, but we note that none of these constructs encode the suffix B. The middle edge has BCAB, but no provision for the single character. We could add a second active point, but we can see

how that may become unweildy. Instead, we will simply cache that we haven't added it, represented by "how many suffixes haven't we added yet".[5]

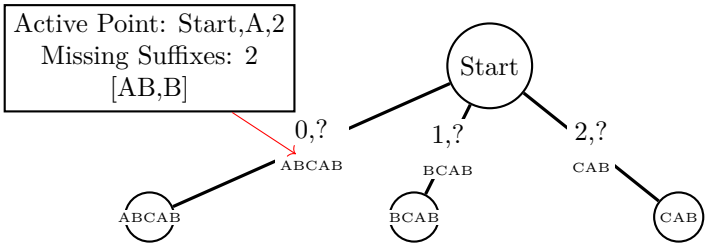


Figure 9.18: Step 5 (take 2): We now have successfully included all suffixes, either on our open edges, or the missing suffixes count. The active point lets us know where the first missing suffix would go. Note that we do not actually save the list of suffixes, only the count.

When we insert the D, we will find a mismatch at the next character of the active point. Remember that the purpose of the active point was to delay splitting an edge until we knew we needed to. So, now that we know we will need to split the edge, we can proceed to do so, and the active point resets back to the start. Now that the active point is not being used, we can proceed to insert the missing suffixes, which have expanded to include the D.

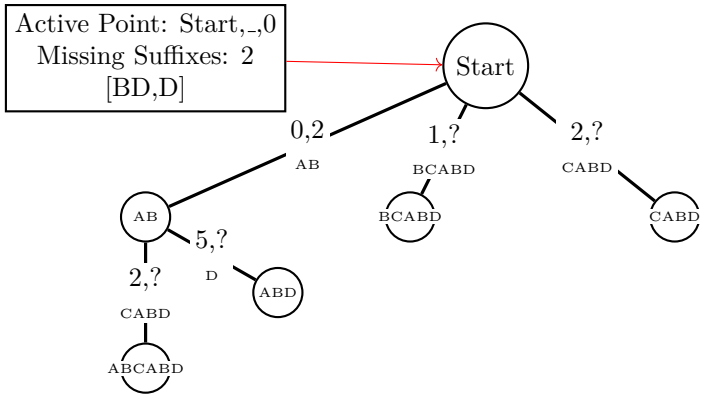


Figure 9.19: Step 6 part 1: We've split the edge, reset the active point, and noted the missing suffixes. Note that we simply increase the missing suffix count, and the last two suffixes are BD and D; we don't actually store those two suffixes, however, so there is no cost to update them. Note that all suffixes are accounted for with the four open edges, and 2 missing suffixes.

We now proceed to add the missing suffixes, as the active point is no longer

[5] Also notice that all the "missing" suffixes are actually in the tree already, they just don't have a leaf node. These are known as *implicit*.

in use. Starting with BD:

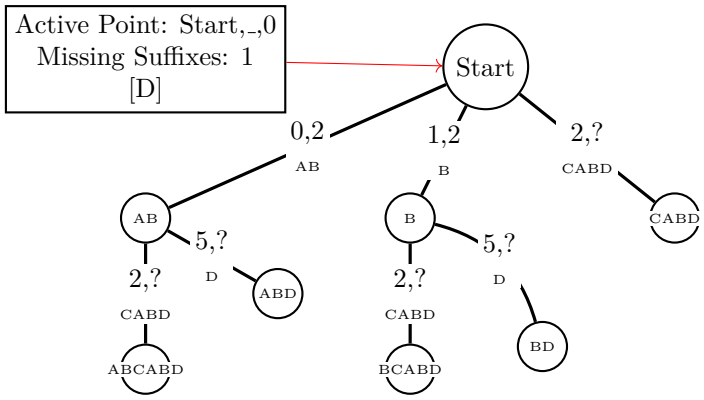


Figure 9.20: Step 6 part 2: We traversed down the tree to insert BD, and found a place to split an edge, so did so.

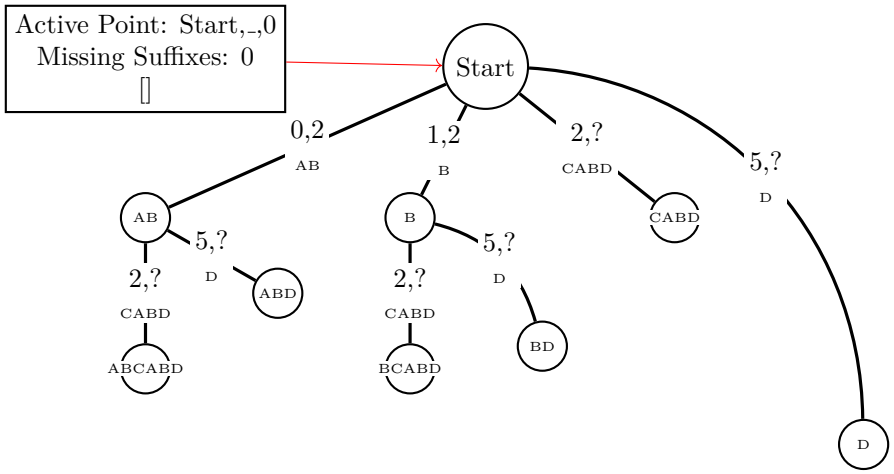


Figure 9.21: Step 6 part 3: We traverse down the tree to insert the next missing suffix, D, but find we already don't have an outgoing edge from the root, so insert it. As there are no longer any missing suffixes, we proceed to step 7.

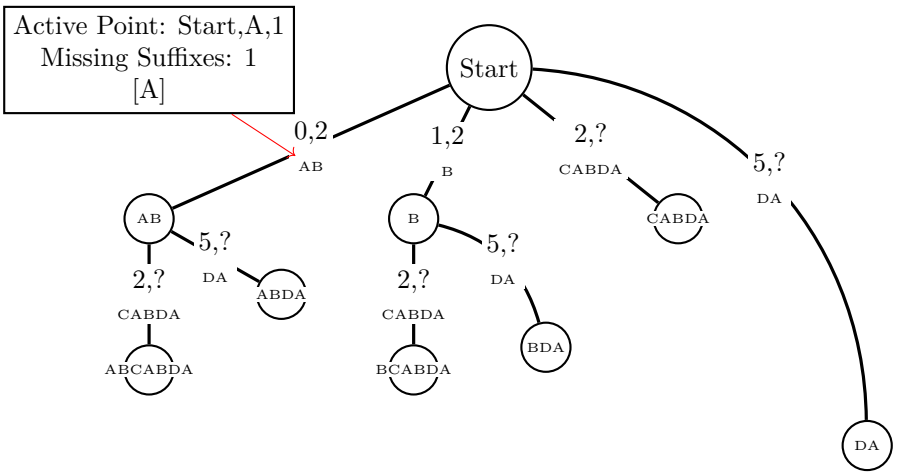


Figure 9.22: Step 7: We are inserting an A, and it already exists, so we simply move the active point.

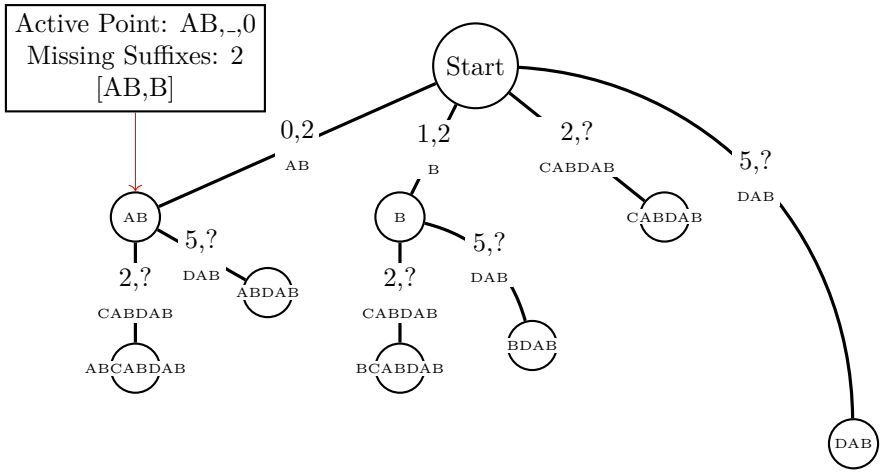


Figure 9.23: Step 8: We are inserting the B, and it already exists, but the active point reaches the end of an edge, so we update it to point to the node at the end of that edge. Note that we have also recorded that we are missing a suffix again.

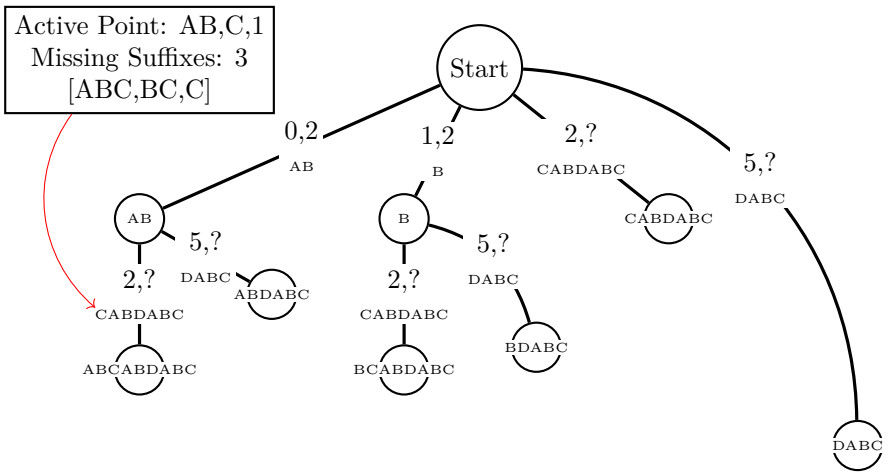


Figure 9.24: Step 9: We are inserting the C, and we can accommodate this by simply moving the active point and noting the extra missing suffixes.

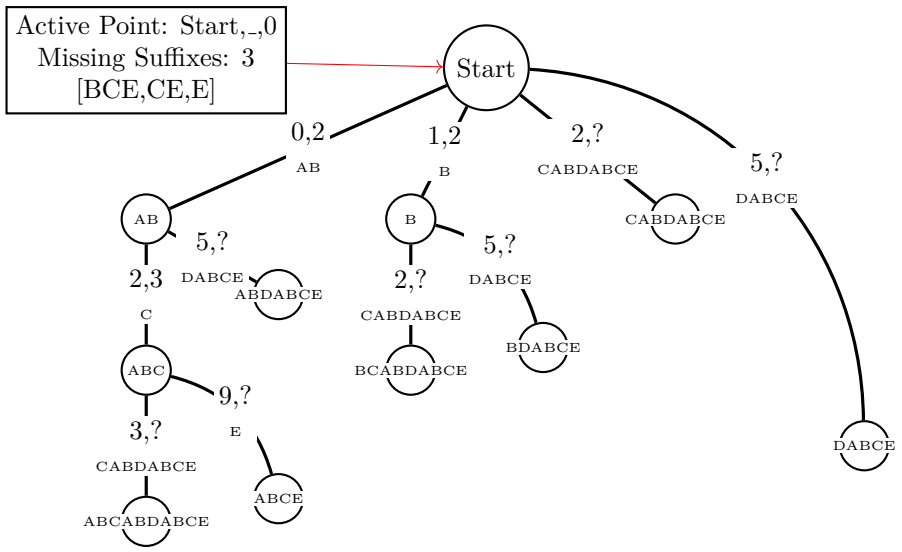


Figure 9.25: Step 10 part 1: We are inserting the E, which does not match the character succeeding the active point, so we must split the edge, return the active point back to the root, then work on inserting the missing suffixes.

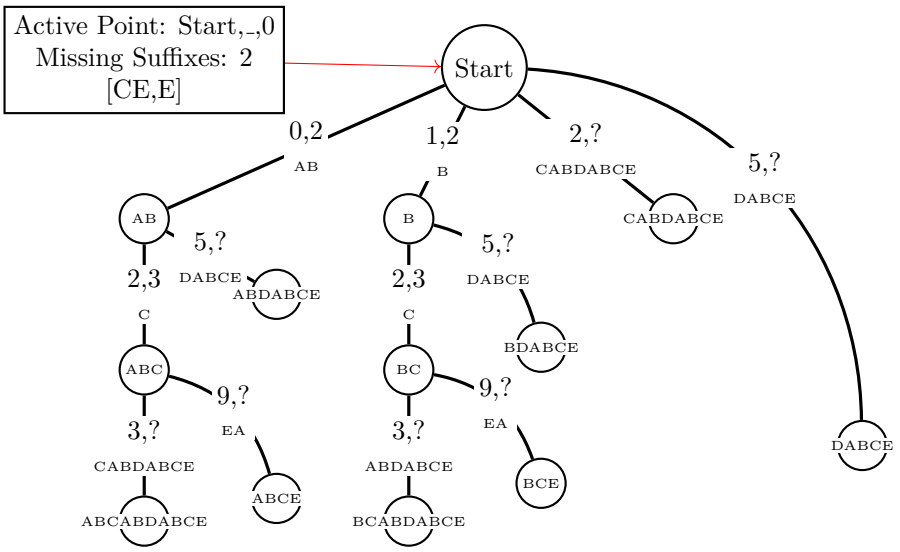


Figure 9.26: Step 10 part 2: We traverse down the tree and insert suffix BCE

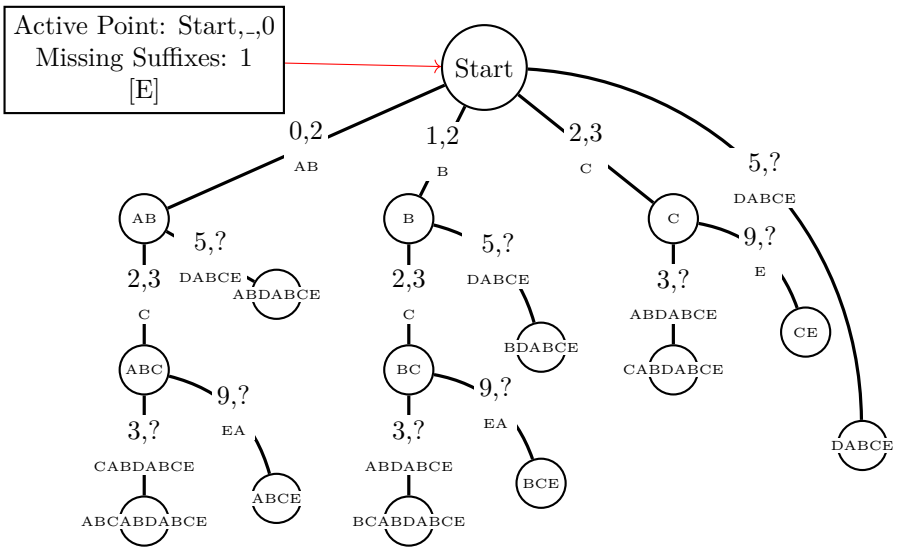


Figure 9.27: Step 10 part 3: We traverse down the tree and insert suffix CE.

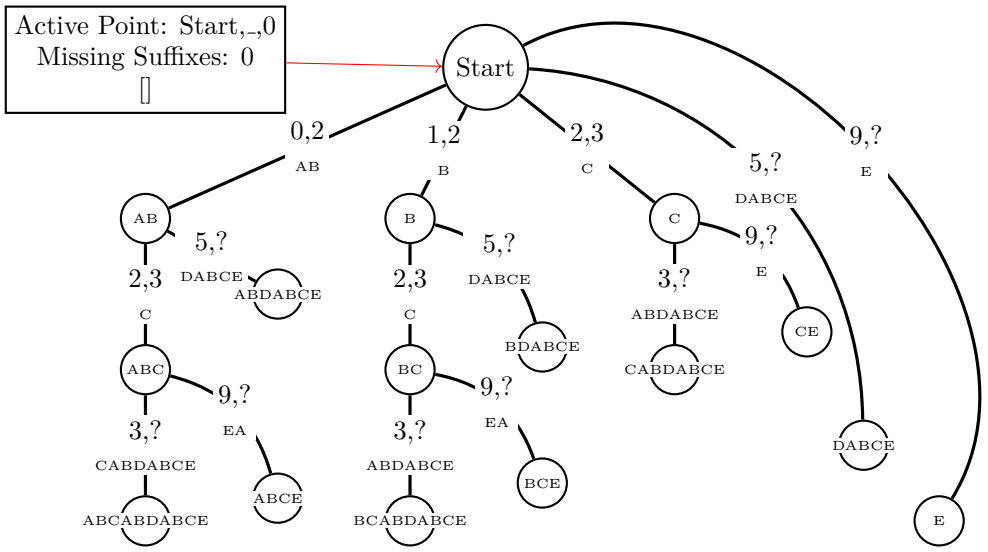


Figure 9.28: Step 10 part 4: We traverse down and insert the final suffix E, and can proceed to step 11.

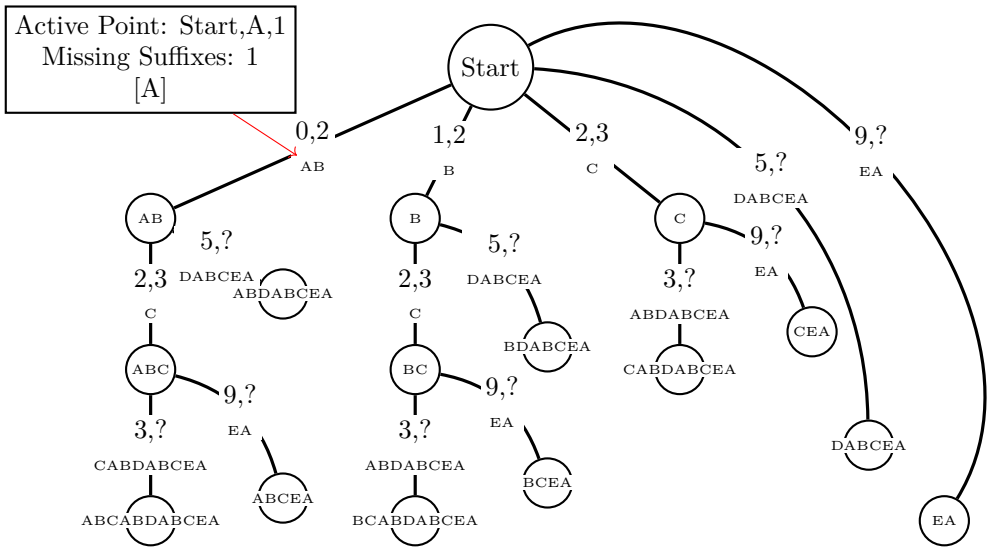


Figure 9.29: Step 11: Inserting the final A is simply a move of the active point.

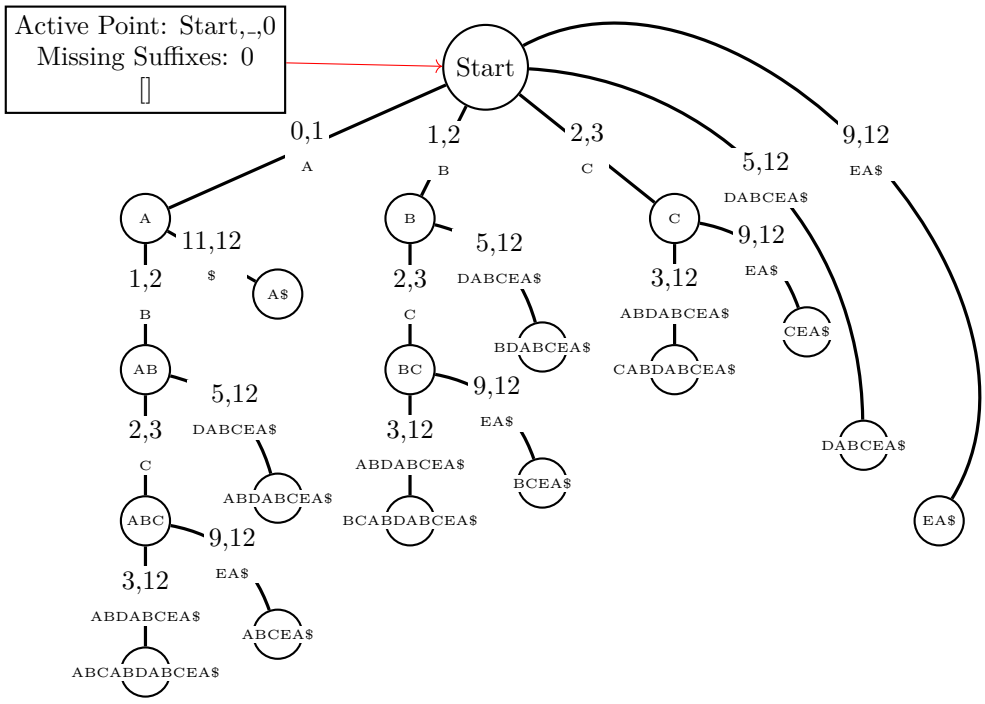


Figure 9.30: Step 12: Inserting the final $, that terminal character does not match the character folling the active point, so we split the edge and reset the active point to the start node. The tree is now complete. Note: If we so chose, we could insert a $ node coming off the start node. We convert all ? to a "real" value for completenes.

The algorithm is correct and seems to be efficient in how it builds the tree. There is one issue, however, which prevents linear time. In the stages where we inserted the missing suffixes, we had to traverse down the tree to insert each suffix individually. As there are $O(N)$ potential missing suffixes to insert, and each may take $O(N)$ comparisons, our runtime is still quadratic in the worst case.

Optimization 6: Suffix Links

In order to address the problem of inserting the missing suffixes, we first take note of some significant similarities in different regions of the tree.

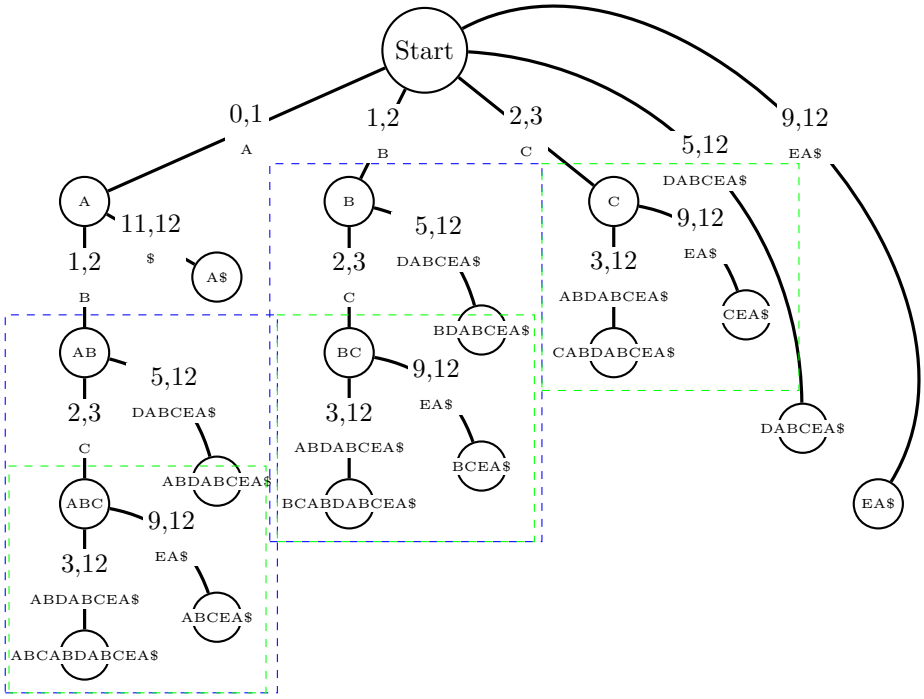


Figure 9.31: There is a "blue" subtree of 5 nodes which contains identical edges occurring 2 times, and a "green" subtree of 3 nodes which occurs 3 times.

In the previous subsection, we note that whatever operation we ended up taking in the AB subtree, we later ended up taking in the B subtree. In step 6, we split the ABCAB edge, then immediately made the same split in the B subtree. In step 10, we split the ABC subtree, then immediately split the equialent node in the BC subtree and C subtree.

By defining the proper relationship between these similar subtrees, we can avoid the previous issue of having to traverse the entire tree while inserting missing suffixes. These relationships are known as *suffix links*. When we split an edge to create a new leaf node, we remember the internal node created at the split, and if we create another leaf node during this step, we draw a suffix link from the "remembered" internal node to the equivalent one. Then, in subsequent steps, if an edge we split comes from a node which has a suffix link, instead of resetting the active point to the root, we instead follow the suffix link and start our traversal there. Suffix links connect the similar subtrees we observe, and avoid us having to traverse from the root down a particular subtree more than once when inserting missing suffixes.

A bit more formally, lets consider a suffix which creates a split on an edge,

and thus new internal and leaf nodes: $\alpha\beta$X, where α is some character, β is the substring after α until the newly created internal node, and X is the character forcing the split. Assuming β is non-empty, then there will end up being a suffix link from node $\alpha\beta$ to node β. Note that as we are inserting $\alpha\beta$X, then, since we are iterating character by character, then $\alpha\beta$ and β will already be existing paths in the tree. The node β will come from one of two sources:

1. Created in the next "part". This is the common case, and we draw the suffix link when we create the node β, to split off the leaf node to βX
2. Already existing, and we draw the suffix link when we reach β in our traversal to insert βX

Lets see it in action, starting at step 6, part 2. Remember that in step 6, part 1, we had created an internal node AB, in order to add suffix ABD. We know at this point that we will end up with a suffix link from node AB to node B in the next part.

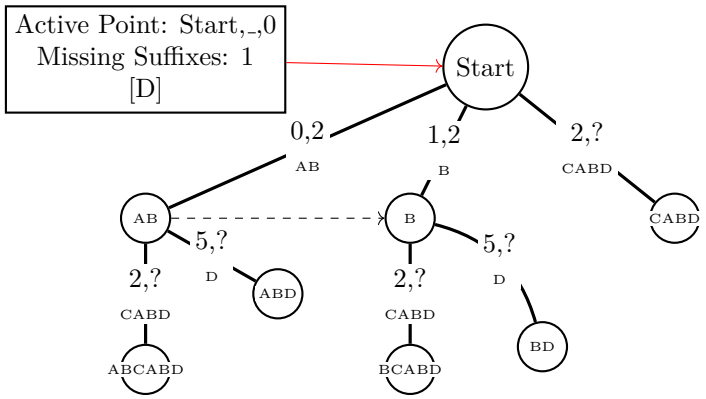


Figure 9.32: Step 6 part 2: We traversed down the tree to insert BD, and found a place to split an edge, so did so, creating node B. As noted above, we know we will need a suffix link from AB to B, so we create it.

The suffix link we drew in the previous picture we node ends up being node at the root of the two "blue" subtrees from the previous figure. The implication is that moving forward, every change which occurs in the AB subtree will also end up occurring in the B subtree. This must be true as there are is no such suffix ABX which does not also imply a suffix BX. Note that the inverse is not true, as there may be a suffix BX which was not immediately preceeded by an A.[6] This explains why the suffix link is directed. We see the utility of the link in step 10, and will demonstrate construction of the tree starting at that point.

[6]We will see an example of this shortly.

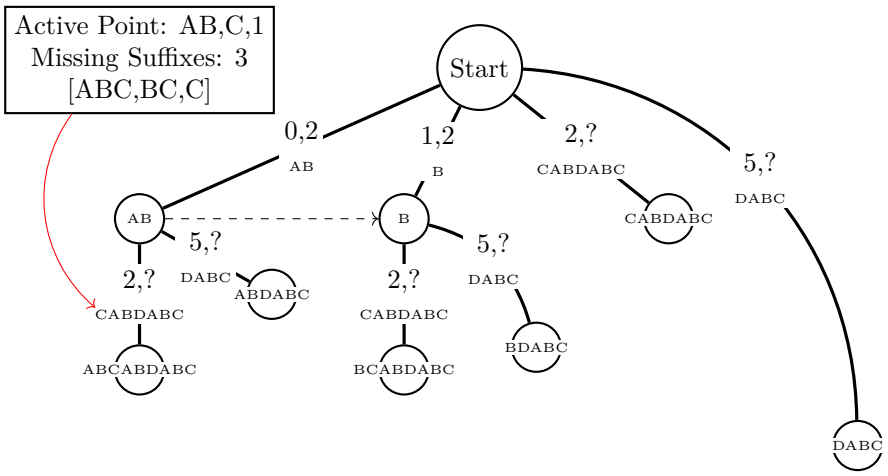


Figure 9.33: The tree after step 9.

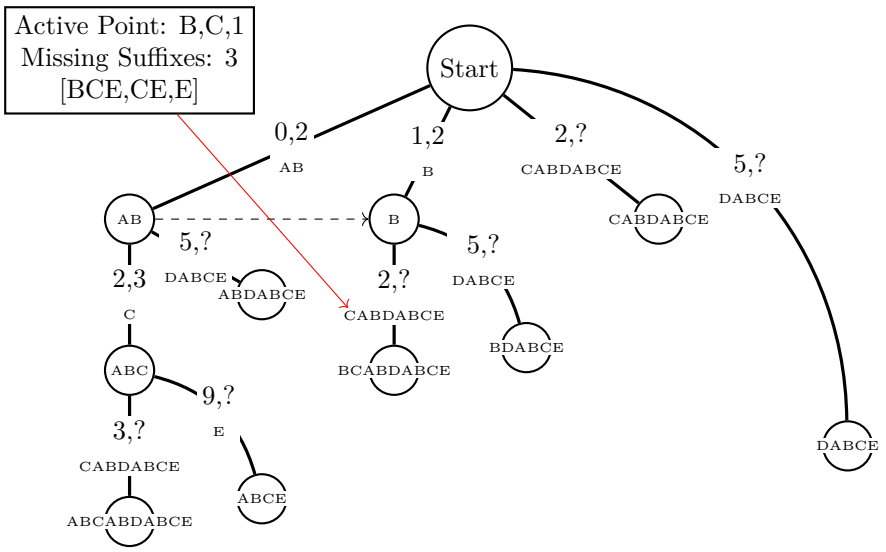


Figure 9.34: Step 10 part 1: As with before, we split the edge to insert the E. We note that as the newly created node is ABC, we will end up with a suffix link to a node BC. Instead of resetting the active point to the suffix, however, we note that the node the active point was using as a base has a suffix link, and cause the active point to "hop" to the location in the new subtree, where as previously discussed, we will end up making the same edit.

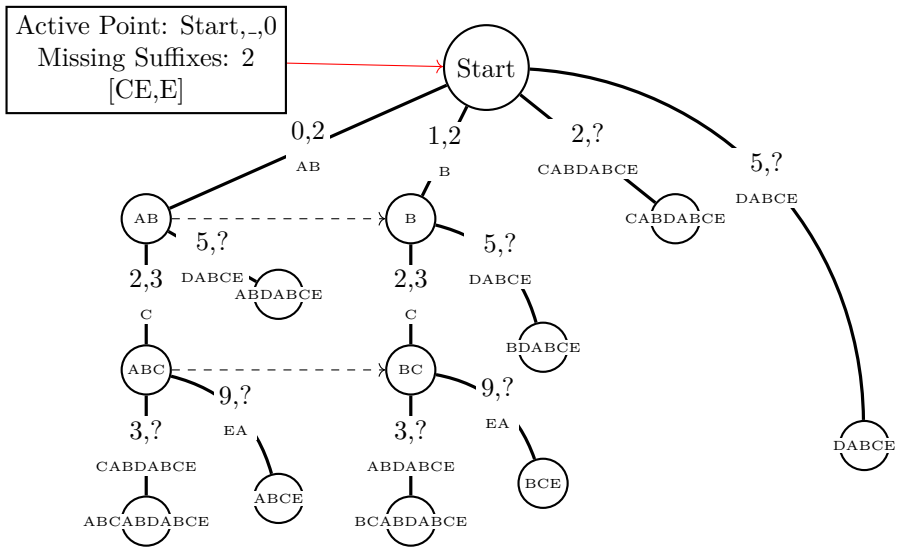


Figure 9.35: Step 10 part 2: Several things happen in this part. As before, we split the edge, since the E does not match the character following the active point. We've now created node BC, which is the endpoint of the newly needed suffix link. We also note that BC will end up needing a suffix link to a node C. Lastly, as there is no suffix link coming out of the B node, we revert the active point to the start. Note that the ndoes ABC and BC which now have a suffix links are roots of two of the green subtrees.

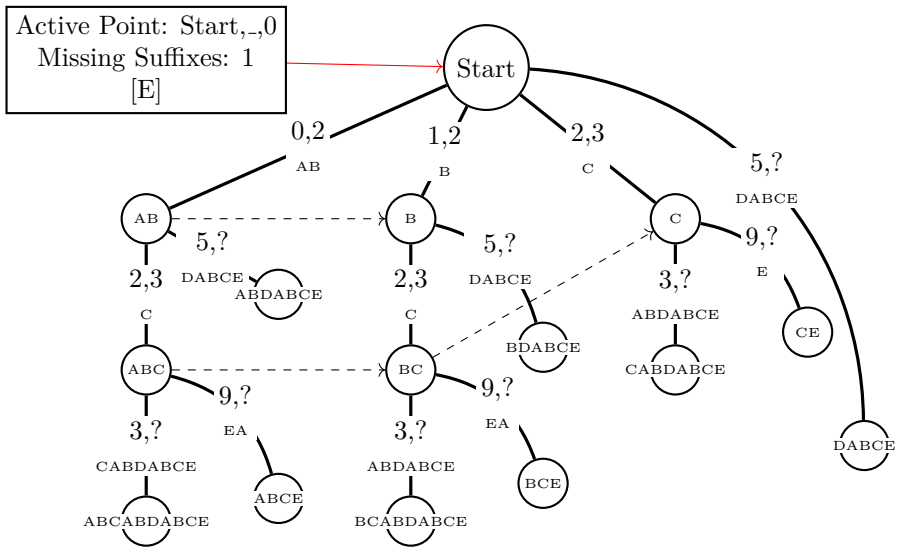


Figure 9.36: Step 10 part 3: We traverse down the tree and insert suffix CE. As with the previous part, the newly created node is the endpoint required for the suffix link we needed, so we create it. Note that this has now linked all three green subtrees.

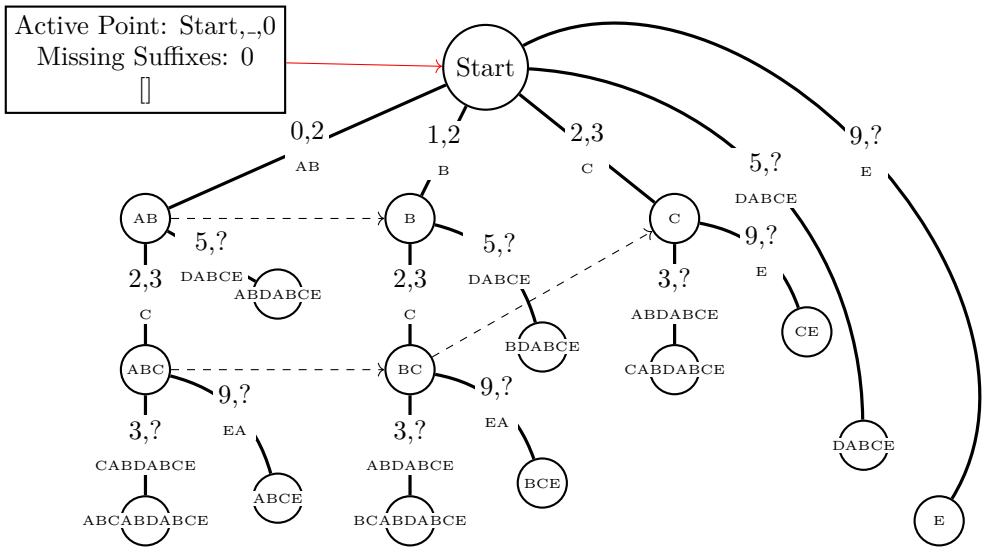


Figure 9.37: Step 10 part 4: We traverse down and insert the final suffix E, and can proceed to step 11.

Steps 11 and 12 are identical to before.

Optimization 7: Fast Traversal

Again, while the algorithm seems efficient, there is one step which is still problematic: Traversing down the tree to find a split point. In the worst case, we have to traverse $O(N$ characters down from the root to find a split point. The final optimization helps us do this more efficiently.

Critically, as noted before, we can notice that while we are manipulating the active point, all the missing suffixes do exist in the tree, but do not yet have associated leaf nodes. Further, any split which occurs does so just before the final letter of the suffix we are inserting. Using these two facts, we can traverse any edge in constant time. If we are at the root (or any node), trying to insert a given string, and there is an edge that comes from that node which matches the first character of the string, since we know that the string will be in the tree, we can simply "skip" to the next node without having to do a direct comparison. The cases here are:

1. Edge is shorter than or same size as string: Move on to the next node at the end of the edge, use edge labels to determine which character to look at next in the string.[7]

[7]The edge labels are indices, so computing the length is simply substraction.

2. String is shorter than edge: We have found the location where we will need to insert a split on this edge without having to explicitly perform comparison

This traversal is used in both places we perform such a traversal: after we have followed a suffix link, or after we mave moved the active point to the root.

9.5.2 Rough Proof of Linearity

We have a linear number of steps, and most elements during a step are constant time. The only hang up is parts where we create a node, which may happen multiple times during a step, and may involve a traversal, even if it is fast. The total number of nodes in the tree is linear, so the actual creation is not ultimately problematic, but ensuring we can find the location to create the node might be. With all above optimizations, it can be shown that we only traverse a linear total number of nodes for the entirety of the algorithm, so assuming we traverse each edge in constant time with fast traversal, gets us linear time overall. The number of traversals can be shown to be linear based on the fact that suffix links (or "resets to the root") all change the tree depth of the current active point by moving up at most one level towards the root. As such movements only happen a linear number of times, the maximum number of nodes we can "climb" the active point is linear, and the heigh of the tree is linear, so the number of downward traversals must also be bound linearly.[8]

9.5.3 Implementation

The implementation of Ukkonen's algorithm is frought with corner cases. Some notes:

- The maximum number of nodes in the tree is $2 * n$. Therefore, we can store each node in an array, indexed by the order the nodes are created.
- Each node must store outgoing edges indexed by the first character of that edge. This can either be an array, if the alphabet size is small, or if necessary, a map.
- Each outgoing edge must store two pieces of information, the start and end index of the characters representing that edge.
- There are a significant number of corner cases, so any implementation should be exceptionally well tested.[9]
- It is highly convenient to leverage the active point to store certain details as we complete the algorithm The rules we will use are as follows:

[8] For the formal proof, check out Dan Gusfield's book *Algorithms on Strings, Trees, and Sequences.*

[9] And many free online resources on the subject mis critical corner cases

1. After a split, if the active node has a suffix link, simply change the the active node to the destination node of the suffix link, leaving the edge and length the same. If the new active edge is shorter than the active length, immediately perform a fast traversal and update the active point if necessary.[10]
2. After a split, if the active node has no suffix link, the active node becomes the root, the active edge is the first character of the next missing suffix we intend to insert, and the active length is the length of the necessary traversal (i.e. one fewer than the length of the suffix itself we are inserting next). If the new active edge exists and is shorter than the active length, immediately perform a fast traversal and update the active point if necessary. Note that this should only occur if the active node represents a string which is length 1.
3. The above two rules still apply if the location where we insert the leaf node was preexisting (i.e. the active length was 0), and we then follow the suffix link, if any.

- We will have to keep track of the most-previously created node representing a string of length greater than 1, as it will necessarily be source node of the next suffix link we create. This value should be empty by the end of every step.
- If there is a most-previously created node saved, the destination of its suffix link will always be the parent of the next leaf node created.
- In many implementations, the above two rules are generalized further, and suffix links are drawn from even length-1 internal nodes back to the root. In such a case, all interior nodes have a suffix link, eliminating some corner cases.
- While we label each node in our figuers with the string it represents, the nodes themselves should not, and do not need to store this information explicitly. Nothing in the algorithm strictly depends on it, and doing could force quadratic behavior.
- It is often customary to increase the missing suffixes count at the start of a step, and account for that properly as the step goes on. This is an implementation detail.
- It is also often customary at the start of a step to set the active edge to the next character, if it is null.
- After constructing the tree, it is common to store some additional metadata, including length of the substring necessary to reach each node, index where the suffix ending at each leaf node starts, or other postprocessing information.

[10] This immediate update is known as *canonicalization*.

9.5.4 Complete Example

Now that all optimizations have been given, we will perform one more complete example to demonstrate the algorithm. We will use the string BCBDABCABD$, and will attempt to represent most of the implementation notes. We will not draw explicit suffix links to the root, however.

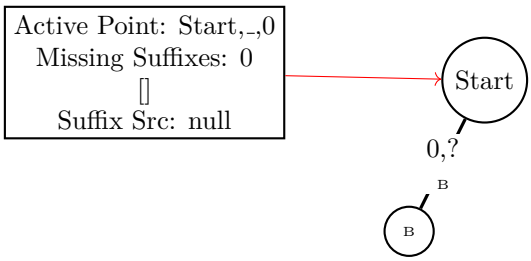


Figure 9.38: Step 1: Root does not have an edge B, so create it.

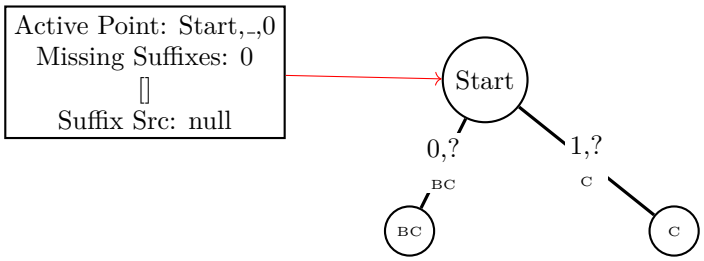


Figure 9.39: Step 2: Root does not have an edge C, so create it.

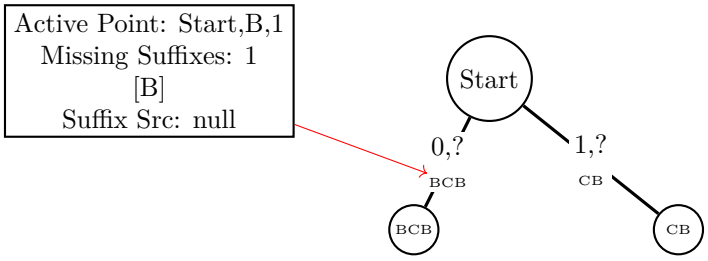


Figure 9.40: Step 3: Root does have an edge B, so move the active point

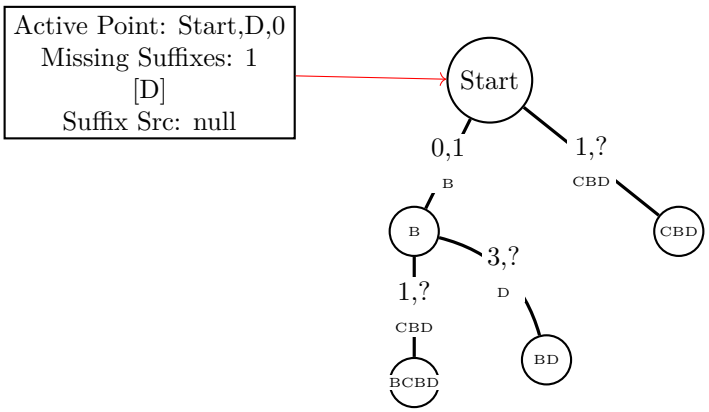


Figure 9.41: Step 4 part 1: The D mismatches, create a new edge. As there is no suffix link, the active point reverts to the root, where we adjust the active edge to D, and subtract 1 from the acttive length. The new node B represents a string of length 1, so we do not set it as the suffix src. Note: we would do so if we intended to create suffix links to the root.

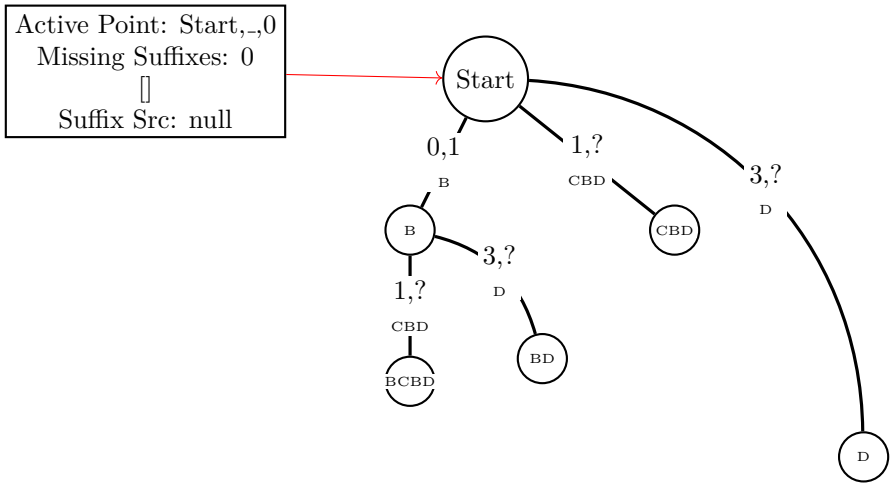


Figure 9.42: Step 4 part 2: Root does not have active edge D, so create it.

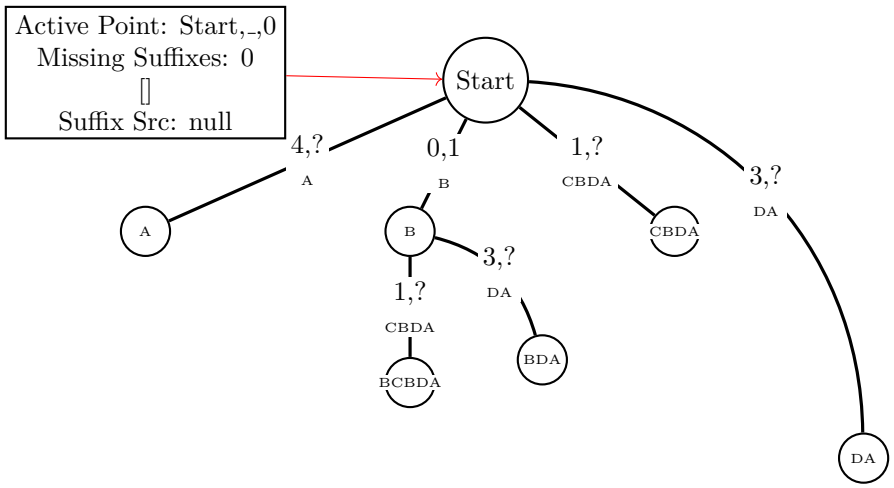


Figure 9.43: Step 5: Root does not have an edge A, so create it.

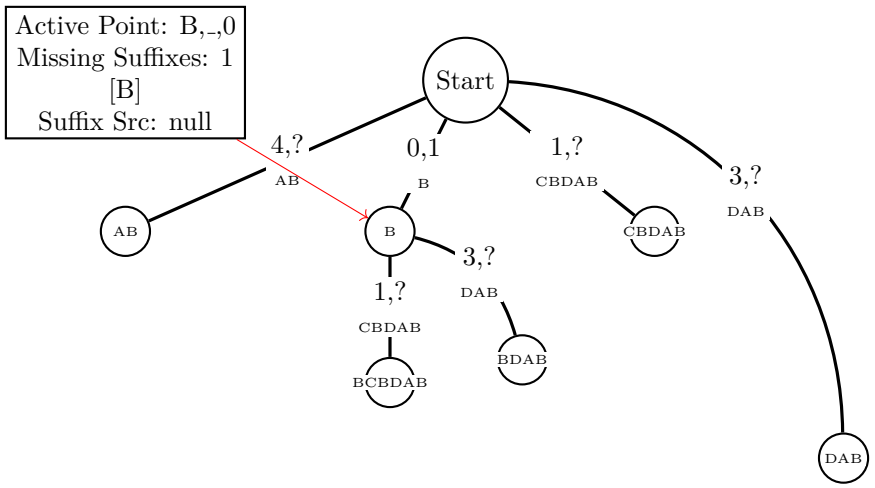


Figure 9.44: Step 6: Root does have an edge B, so move the active point on edge B, length 1. The length of the active edge matches the length of the edge it points to, so move the active point to the next node, and adjust the edge and length accordingly.

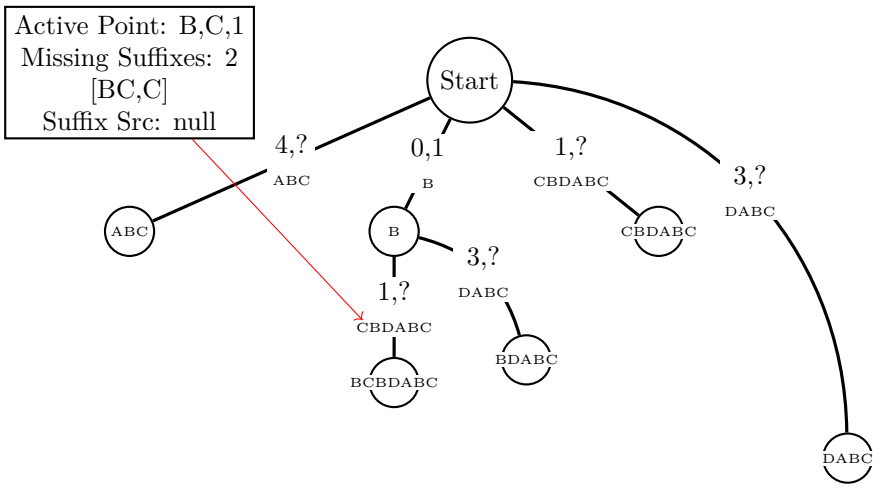


Figure 9.45: Step 7: The active node does have an edge C, so move the active point.

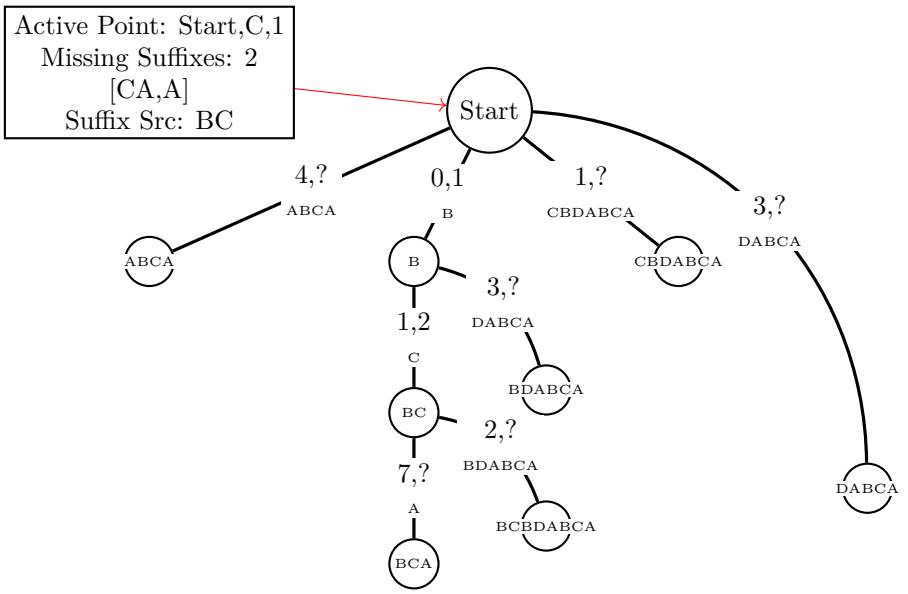


Figure 9.46: Step 8 part 1: The A does not match the next character after the active point. We split the edge at node BC. As BC is length greater than 1, we set it as the suffix src. We know it will end up with a suffix link to a node C. As the active node, B, has no outgoing suffix link, the active point reverts to the root. The next suffix is CA, so the active edge is C, and the length is 1 (i.e. we have to traverse to C (a string of length 1) to find where to insert leaf A).

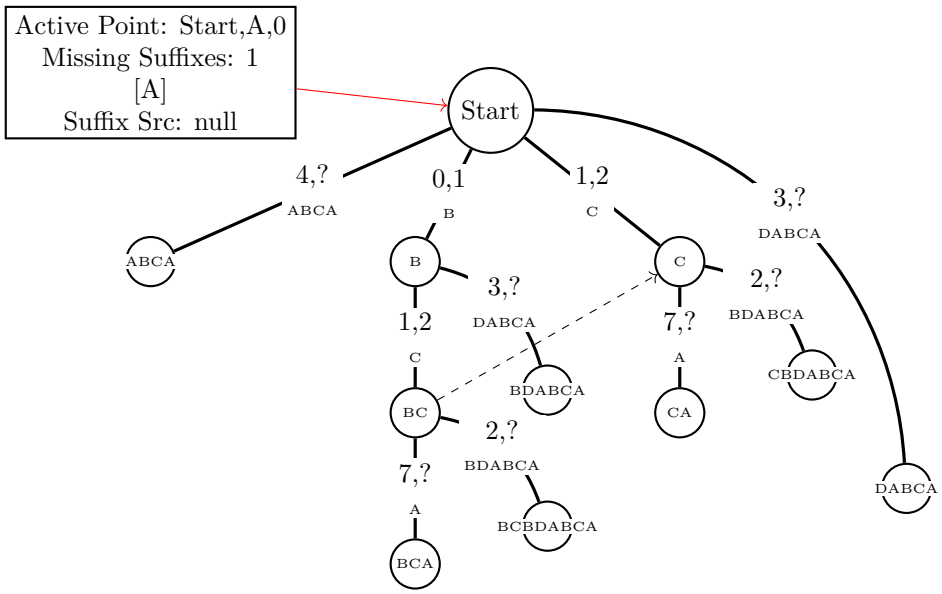


Figure 9.47: Step 8 part 2: We see that the active edge does exist from the active node, and the length of that edge (7) is less than the active length (1). We therefore have found our split location. We split the edge. We see that there was a suffix src, so we draw the suffix link from BC to B. This new node represents a string of length 1, so it is not set as the suffix src. The active node remains at the root, and we update it for the next suffix to insert.

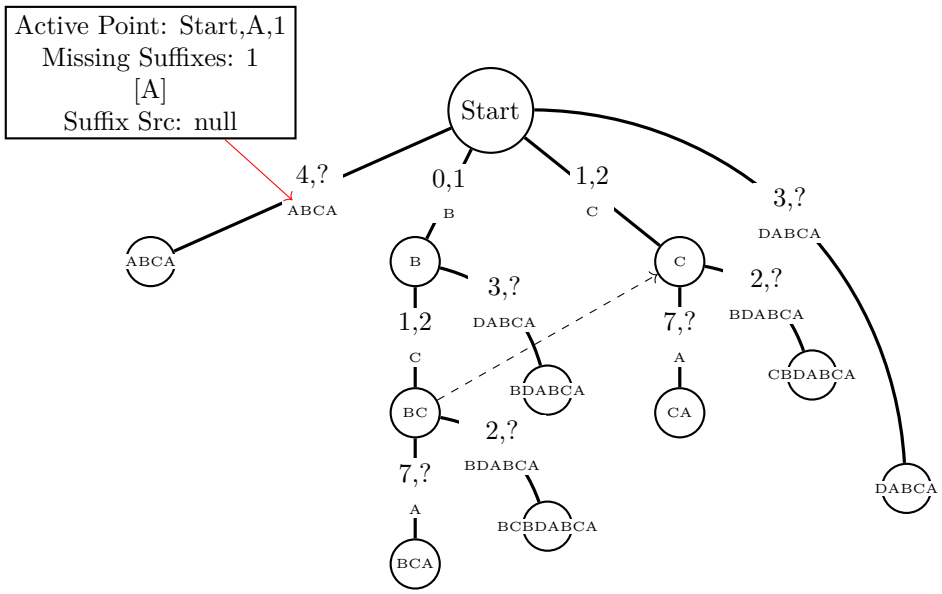


Figure 9.48: Step 8 part 3: There is only the A remaining, which means there is no traversal to do before inserting it (represented by the active length having been 0). We therefore simply note that the activenode already an edge A, so we simply move the active point.

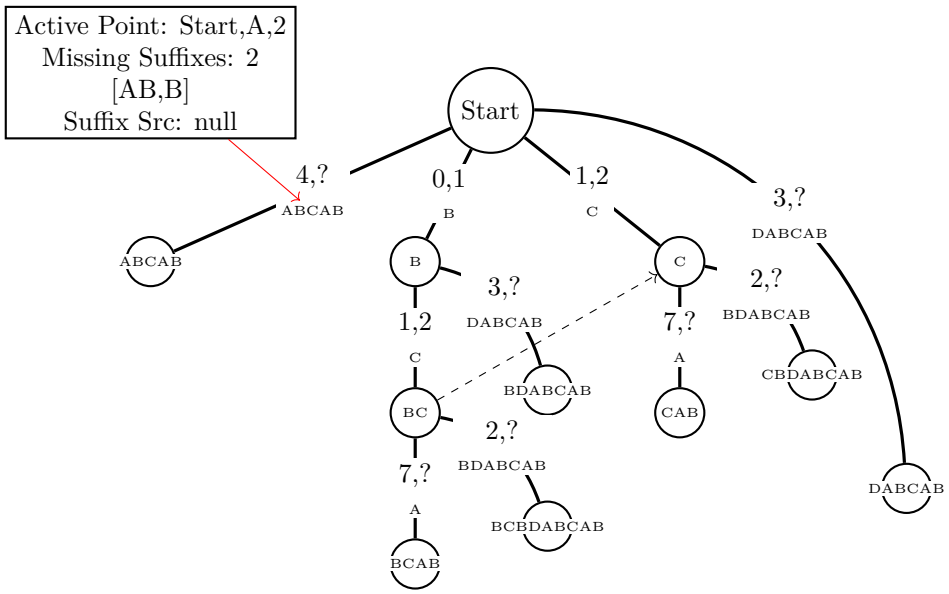


Figure 9.49: Step 9: The next character, B, matches the character after the active point, so we simply move it.

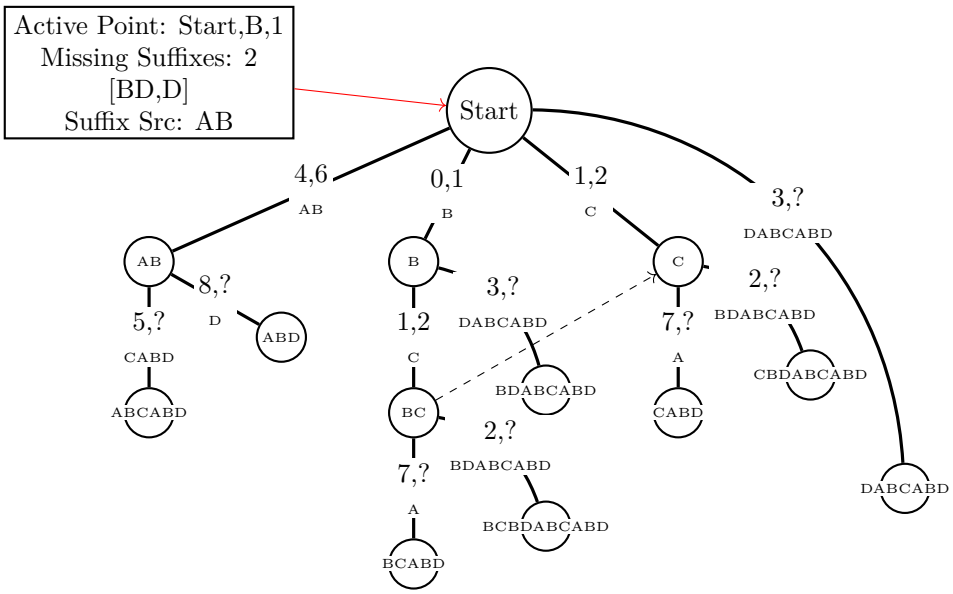


Figure 9.50: Step 10 part 1: The next character, D, does not match the character after the active point. We split this edge. We note that AB as the suffix src, as it will end up with a suffix link to B, and we update the active point to the root, since we don't have an outgoing suffix link to follow.

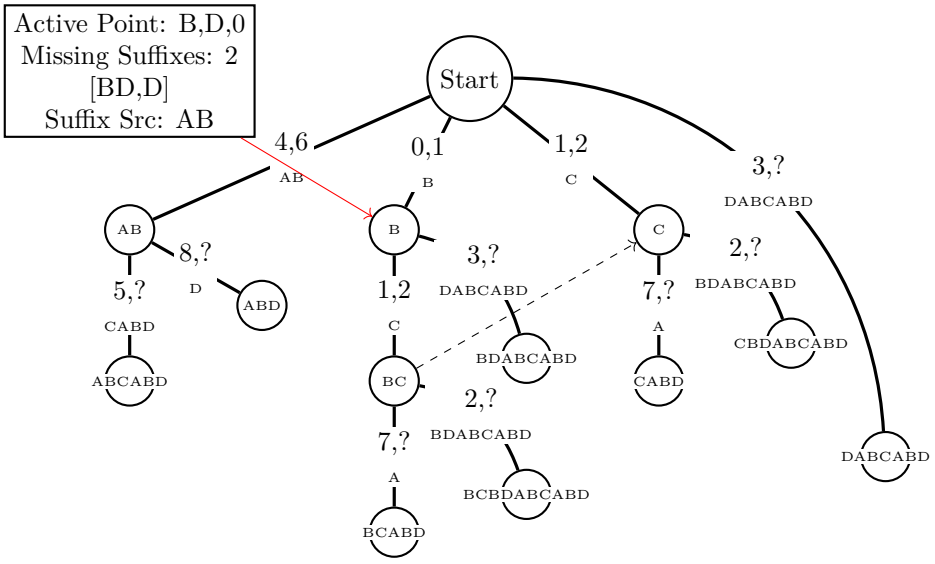


Figure 9.51: Step 10 part 2: The active edge has a length shorter or equal to the active length. We therefore move the active point to that node.

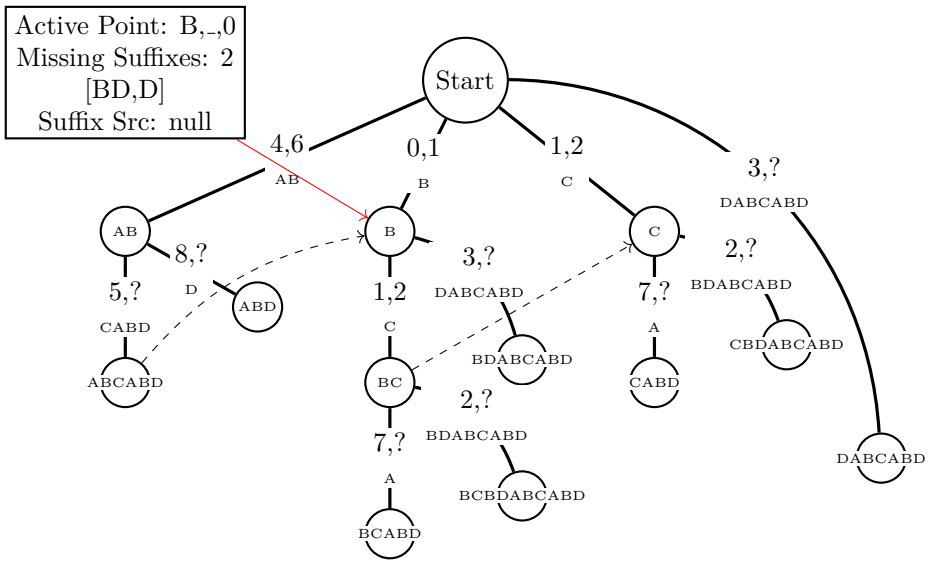


Figure 9.52: Step 10 part 3: We have found the location (B) in the tree where we would add BD. We know this since our active length is 0. A node already exists here, however, so there is no need to perform a split. We will draw our suffix link from the suffix src, however.

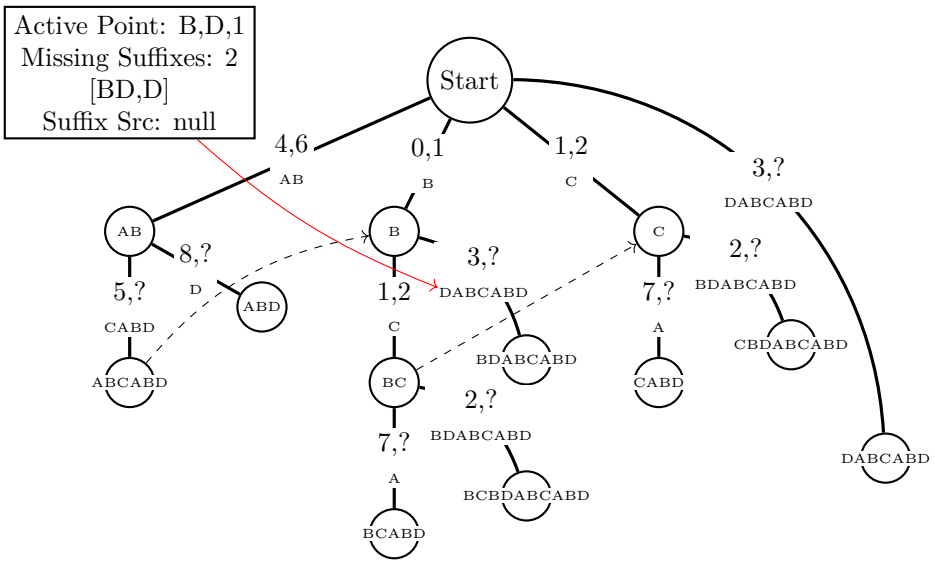


Figure 9.53: Step 10 part 4: When we go to finally create the leaf, we find node B already has an edge for D, so we simply move the active point, and are complete with this step. It may seem unusual that after a split we do not end up back at the root, but as with the original justification for non-committal suffixes, we do not know if we will have to make a split or not without looking ahead. As all suffixes are still properly tracked in either leaves or the missing suffix list, we are all set.

Step 11 is the insertion of the terminal character \$. As this section is already too long with diagrams, the parts of step 11 are described here.

1. \$ does not match the next character after the active point, so we split the edge, creating node BD. We set BD as the suffix src. As there is no outgoing suffix link from the ative node, the active point reverts to the root with edge D and length 1.
2. There is an outgoing edge D from the root, and it is longer than the active length 1. We split this edge to form node D, create the leaf node for \$, and draw the suffix link from our suffix src BD.

9.5.5 Usage

Now that we have created a suffix tree in linear time, we can discuss how to solve the previously-listed problems.

Pattern Existence

The suffix tree does not simply contain all suffixes. A suffix only occurs if we end a traversal at a leaf node. As any substring of an input must be the prefix

of the suffix starting at the same location, it must be found in the traversal of the suffix tree towards the leaf node representing that suffix. Consider the substring DAB. This substring is a prefix of DABCEA, which is a suffix of ABCABDABCEA, and therefore is found in the trafersal towards that suffix's leaf node in the suffix tree.

So check if a pattern exists in a string, we can traverse the tree using the pattern. If we are still traversing when we consume all the characters of the pattern (i.e. we have not found a mismatch), the pattern must exist in the string.

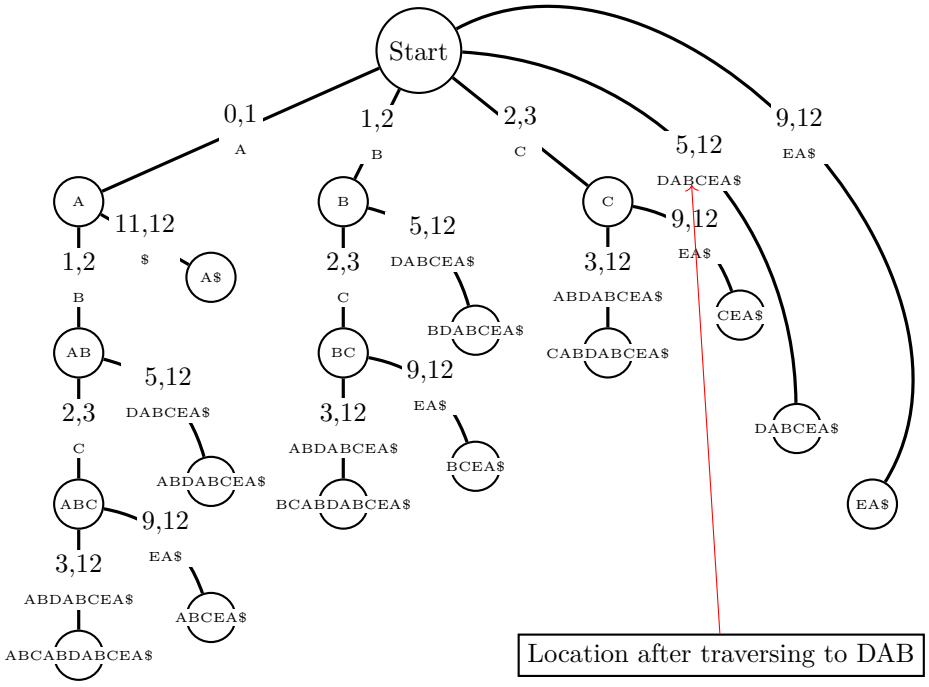


Figure 9.54: We traverse to DAB and find it is represented on some edge in the tree. Therefore we can declare it exists in the original string.

Finding All Instance of a Pattern

Every leaf node in the tree represents a suffix, and every suffix starts at a different location. The number of times a pattern occurs in a string is equal to the number of suffixes which start with that pattern. Therefore, the number of times a pattern occurs in a string is equal to the number of leaf nodes below the traversal to the pattern. For instance, AB occuurs three times in the input string, and we see it has 3 leaf nodes below its traversal.

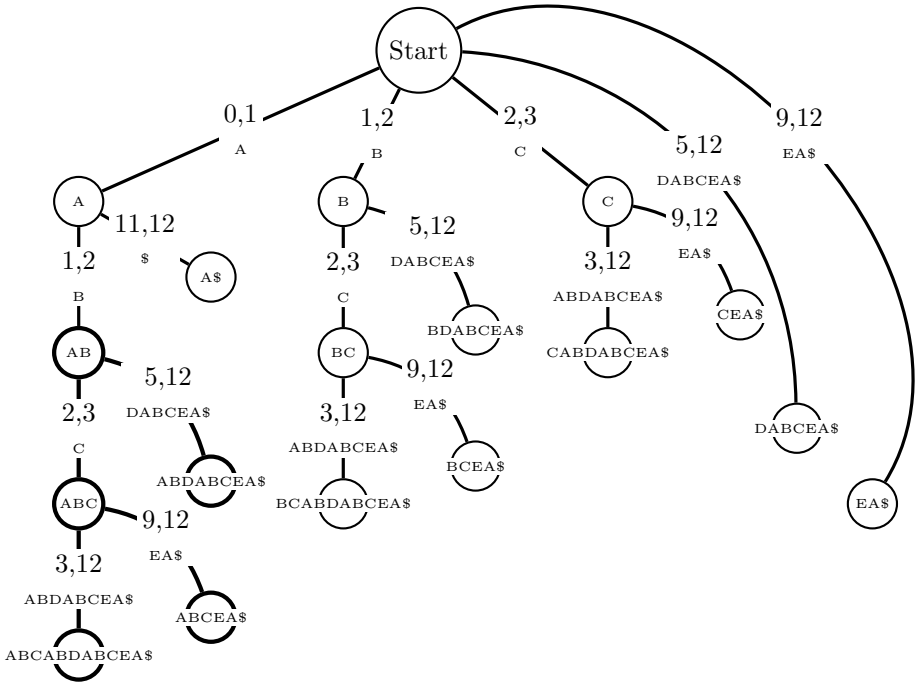


Figure 9.55: The subtree below AB is highlighted. Note there are 3 leaf nodes, representing the 3 locations of AB in the input string.

We can identify the leaf nodes with a simple BFS from the location of the pattern in the tree. Note that there at most $2 * N$ nodes, so this is sufficiently efficient. If multiple lookups are required, the number of leaf nodes could be precomputed for all nodes in the tree.

If we must recover the actual locations of the pattern, we can note the total length of the suffix of each leaf node we reach in our BFS traversal, which tells us where that suffix, and thus the searched pattern, starts. This is unnecessary if we have previously labeled all leaf nodes with their start index.

Longest Repeated Substring

Using similar rules to above, we can determine the longest repeated substring. Note again that the number of leaf nodes below a given point in the tree represent the number of times that substring occurs. Therefore, the longest repeated substring is the deepest node in the tree which has at least two leaf nodes (i.e. the deepest internal node). This can be done in a single BFS of the tree.

Longest Common Substring

The longest common substring between two input strings is equivalent to the longest repeated substring in the concatenation of those two strings which does not cross the boundary between them, and where the two instances are on opposite sides of the boundary. Example: The longest common substring of ABCDAB and CDEABCDE is ABCD. The longest repeated substring in the concatenation of those, ABCDABCDEABCDE, is ABCDE, but this requires us to cross the boundary between the two strings, which is not allowed. ABCD is the longest which does not cross that boundary.

We know how to compute the longest repeated substring, but how do we ensure it adheres to the other conditions? This is solved by leaving a terminal character in place between the two substrings. We can extend the idea of the $ character, and concatenate the two strings, but enusre they each have a unique terminal: ABCDAB$CDEABCDE&. Now, the longest repeated substring is guaranteed not to cross the boundary, as the $ terminal character is guaranteed to only appear once in the concatenated string. If we were to build a suffix tree of this and search, we would find a longest repeated substring, but how do we also guarantee the two substrings occur on opposite sides of the boundary? When comparing the two leaf nodes, we can simply check that start location of thet two suffixes place them on either side of the boundary.

The high level algorithm is as follows:

1. DFS through the tree
2. Note the distance to the root as we perform the DFS
3. When returning from the recursion, note whether there are leaf nodes at or below this node which are on the left side of the boundary, right side of the boundary, or both.
4. Store the maximum depth node which was found which has "both"

This concept of storing multiple strings with different terminal characters together is known as a *generalized suffix tree*. Note that we can make its usage slighlty more convenient by noting that every suffix starting in the first substring will contain the entire second substring. We can truncate these extra characters so that the leaf node terminates at the proper terminal character (either $ or & in our example).

Longest Common Extension

Consider the following question: Given indices i and j in an input string, how many consecutive characters following i and j match eachother? This is known as the longest common extension (LCE) problem. Assuming we have identified the leaf nodes corresponding to the suffixes starting at the two indices (which we could do in a single pass of the tree if necessary), the LCE is the longest common prefix of those two suffixes, which can be found trivially by walking the tree from the root towards the two leaves until the path splits.

A single query of LCE is not particularly interesting (and could be done without the suffix tree), however, the useful case is when we have many queries, and for which a linear lookup may not be sufficient. To accommodate this, we notice that the point where the path to the two leaves splits is the lowest common ancestor of the two leaf nodes. We can utilize any standard LCA algorithm, however the linear time processing with constant time lookup algorithm is most appropriate here.

Longest Palindromic Substring

The longest palindromic substring is the longest substring within an input which is a palindrome. This is equivalent to the longest common substring between an input string and its reverse with an additional condition that the substring occurs in a matching locations in the string and its reverse. To see why this condition is necessary, consider the example ABADXYDABAZ. The longest palindromic substring is ABA, however, the longest common substring between this string, and its mirror (ZABADYXDABA), is ABAD, which is not a palindrome.

The issue is that ABAD and its mirror BOTH occur in the input string. This causes us improperly match those two strings. When looking at the string and its mirror, we note two things:

1. Each character has a "pair" in the mirrored string. For a character at index i, this is found at $N - i - 1$ in the mirror.

2. The common substrings in each the input and its mirror comprising the candidate palindrome must contain the same characters.

3. the candidate ABAD contains characters 0,1,2,3 in the input, which should be characers 10,9,8,7 in the mirror, but instead is found at 1,2,3,4, and therefore is invalid.

Therefore the complete location condition is that if index i is the start of the candidate of length l in the first string, then the start of the candidate in the mirror must occur at index $N - i - l$. We see that candidate ABA, at index 0, length 3, properly occurs at $11 - 0 - 3 = 8$ in the mirrored string, and is therefore a solution. Using longest common substring, we can easily identify the incorrect ABAD, but how can we impose the location condition to ensure we only find ABA? One possible solution is to cache on each internal node the start index of the leaf nodes you can reach from there for both the forward and reverse string. Then, as we later traverse the tree, we can see if any pair of those suffixes adhere to the location condition. The problem with this scenario is that it is super-linear. Even storing the leaf node "cache" on each node is quadratic overall.

Instead of walking the tree, we can consider all possible mid-points of a longest palindrome. There are exactly $2 * N - 1$: N odd centers, and $N - 1$ even. From each midpoint, we wish to determine the widest palindrome with

that midpoint. This is equivalent to finding the LCE of the substring going to the right of the midpoint and the one going to the left. Assuming we have constructed a suffix tree as described above (with the forward, and mirrored string), then this is simply the LCE of the substring going to the right in the forward string, and the substring going to the left in the mirrored string. We can identify those two leaves, and execute the above LCE algorithm to determine the widest palindrome centered at that point in constant time. Looping over all possible centers gives us the necessary lienar time.

Minimum Rotation

The minimum rotation of a string is the least lexicographic substring of length N which occurs if we allow said substring to "wrap around" to the beginning if we reach the end of the input. This is equivalent to the least such substring if the input is concatenated to itself. Example: the minimum rotation of BACAA is AABAC, which is also the longest length 5 substring in BACAABACAA.

Rotations	Lexicographic Order
BACAA	4
ACAAB	3
CAABA	5
AABAC	1
ABACA	2

Therefore, if we construct a suffix tree of such a concatenation, we can follow a path from the root, along the least lexicographic edge which comes from each node, until we have found a substring which is N characters long. Note that we are guaranteed not to get "stuck" in a branch with not enough characters, as every suffix of fewer than N characters also has a "partner" with N more characters, and therefore there must be somewhere to traverse. (e.g. suffix AA has a partner AABACAA. Suffix CAA has a partner CAABACAA).

9.5.6 Suffix and LCP Array