Chapter 9

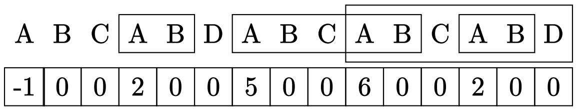
Strings

9.1 Pattern Matching

9.1.1 Z Function

The Z *function*[1] of a given index i in a string is the longest substring starting at i which is a prefix of the string itself. More simply, if $x = Z(i)$, then the first x characters starting at i are identical to beginning of the string, but no more.[2]

Let's consider the Z function computed over the string *ABCABDABCABCABD*. Boxes are drawn arond the substrings matching the Z value, which all match the beginning of the string exactly. These are known as Z *boxes*.



9.1.1.1 Applications

While the utility of such a function may seem esoteric, it is useful both for a few specific applications, some fundamental concepts it exposes, and the simplicity of its implementation, which we will see shortly.

- Finding the longest common extension of any index with the start of the string. This is the definition of the Z function.
- Finding the longest prefix of a second string starting at any index in a given string. This is accomplished by concatenating the two strings with a terminal character which appears in neither of the other strings. More clearly, if we wish to find the longest prefix of a string T which occurs

[1]As described in Dan Gusfield's book *Algorithms on Strings, Trees, and Sequences* as *fundamental preprocessing* of a string.

[2]This is exactly the longest common extension of 0 and i in the string which is discussed more generally later in this chapter.

at any index in another string S, we could create a new string $T\$S$, and compute the Z function over that new string. Note that the \$ character ensures no Z value exceeds the length of T, since no character in S can match the \$.

- Finding all occurrences of a string T in S. Similar to the above, we concatenate with a separator \$ and look for any Z value which is exactly the length of T.

9.1.1.2 Implementation

Trivially, we could compute the Z values in $O(n^2)$ time using a brute force approach. However, with some clever rules, we can reduce this. From a high level, we will compute the Z values in order, and remember some infromation about previously found Z boxes that enable us to quickly compute future Z values using the following rules.

1. Rule 1: If the current index is not found in any known Z box, compute the Z value directly. If we find a Z box, note its left and right endpoints. This rule would cause us to find the Z boxes as indices 3 and 6 in the earlier string.

2. Rule 2: If we are in a Z box, we know that the Z box is duplicated exactly at the start of the string. Look up the Z value at the corresponding index in the beginning of the string.

   (a) Rule 2.1: If the Z value at this corresponding index would keep us within our current Z box, take it as the Z value for the current index. We see this rule apply at index 12.

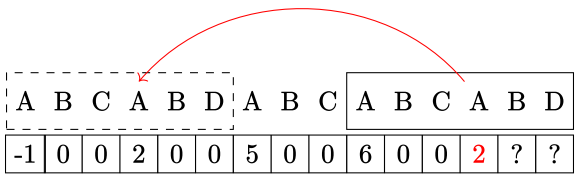


Figure 9.1: The Z value at the corresponding A is 2, so we remain inside the Z box. This means the later A also has a Z value of 2. Note that this must be true, as we know the solid and dashed boxes must be exact copies of eachother, so we could not have a larger Z value in the later Z box than we had in the beginning, so long as the Z value doesn't extend past the bounds of that box.

   (b) Rule 2.2: If the Z value at this corresponding index extends to or past the bounds of the Z box, we cannot rely on 2.1, since more characters might match. We use brute force to examine further characters past the current Z box to find the first mismatch. We then update the current Z box to the one further to the right, since that encapsulates the most current information we have. We see this rule apply at index 9.

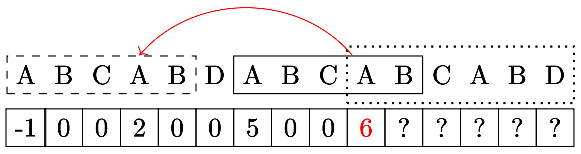


Figure 9.2: The Z value at the corresponding A is 2, but this aligns with the end of the current Z box, so there could be further characters which match. We perform this comparison at the C following the solid Z box, and the corresponding character at the beginning of the string, the C at index 2. We find we match an additional 4 characters, making our total Z value $2 + 4 = 6$. We record the new value, and move to the newer, dotted, Z box for all future lookups.

We note that by using these rules, we always track the Z box with the rightmost endpoint that we know about so far. More critically, we note that we only perform a comparsion within a Z box a single time, when we are creating that Z box, and a comparison against a given letter in the string either is part of the creation of a Z box (so that it is never compared against again), or it causes us to complete the computation of the Z value for a given index. This guarantees us $O(n)$ overall time to compute the Z array.

A compact implementation could be as follows.

Listing 9.1: C++

```cpp
vector<int> zfunc(string s){
    vector<int> z(s.length());
    int l=-1,r=-1; // current z-box
    for(int i=1;i<s.length();i++){
        if(r==-1){ // Rule 1
            while(s[z[i]]==s[i+z[i]])z[i]++; // abuse z[i] as our pointer.
            l=i;
            r=l+z[i];
        }else{
            z[i]=z[i-l]; // Rule 2
            if(z[i]>=r-i) {// Special handling for Rule 2.2
                while(s[z[i]]==s[i+z[i]])z[i]++;
                l=i;
                r=l+z[i];
            }
        }
    }
    return z;
}
```

9.1.2 Knuth-Morris-Pratt

Knuth-Morris-Pratt (KMP) is one of the more well-known algorithms for identifying instances of a fixed pattern in a string. Most languages have a built-in, highly-performant way to perform this operation. Further, other algorithms are simpler (such as the above Z-func) or more performant (such as Boyer-Moore). So why is KMP presented here? The method used for KMP is very powerful, and while string matching cna be done in other ways, the method backing KMP can be extended to solve more difficult problems (such as matching multiple patterns). For this reason, it is essential to understand KMP so that the more complex algorithms can be understood.

9.1.2.1 Intuition

First, observe the naive way in which we might match a pattern to a string. We consider the pattern ABABC, and an text ABCABABABC. We'll attempt to align the pattern and the text at every position, and check if there is a match:

Listing 9.2: C++

```cpp
void naive_match(string t,string p){
  for(int i=0;i<=t.length()-p.length();i++){ //all alignments of p in t
    bool match=true;
    for(int j=0;j<p.length();j++){ //check all characters
      if(p[j]!=t[i+j]){
        match=false;
        break;
      }
    }
    if(match)cout<<i;
  }
}
```

We can observe how the algorithm progresses. The failed comparison in each loop is highlighted.

$$\begin{array}{cccccccccc} A & B & \boxed{C} & A & B & A & B & A & B & C \\ A & B & \boxed{A} & B & C & & & & & \end{array}$$

Figure 9.3: `i==0`

$$\begin{array}{cccccccccc} A & \boxed{B} & C & A & B & A & B & A & B & C \\ & \boxed{A} & B & A & B & C & & & & \end{array}$$

Figure 9.4: `i==1`

$$\begin{array}{cccccccccc} A & B & \boxed{C} & A & B & A & B & A & B & C \\ & & \boxed{A} & B & A & B & C & & & \end{array}$$

Figure 9.5: `i==2`

$$\begin{array}{cccccccccc} A & B & C & A & B & A & B & \boxed{A} & B & C \\ & & & A & B & A & B & \boxed{C} & & \end{array}$$

Figure 9.6: `i==3`

$$\begin{array}{cccccccccc} A & B & C & A & \boxed{B} & A & B & A & B & C \\ & & & & \boxed{A} & B & A & B & C & \end{array}$$

Figure 9.7: `i==4`

$$\begin{array}{cccccccccc} A & B & C & A & B & A & B & A & B & C \\ & & & & & A & B & A & B & C \end{array}$$

Figure 9.8: `i==5`, a match is found

Examining alignment 0, we successfully compared two characters before we failed and mvoed to alignment 1 and performing a further comparison. The second character compared at alignment 0, or in any alignment where we match at least two characters, is B. More importantly, that second character is **not** the first character in the pattern, A. Therefore we can conclude if we ever match the first two characters in an alignment, the pattern will always fail in the subsequent alignment. We see this when `i=0`, implying we cannot match at alignment 1, and we also see this when `i=3`, implying we cannot match at alignment 4.

The fundamental intuition behind KMP is that we can leverage information from comparisons in previous alignments to "skip" future alignments. By skipping enough alignments and efficiently computing which alignments can be skipped, we can achieve linear overall time.

The algorithm occurs in two large steps:

- Pre-process the pattern, producing a table mapping the location of a comparison failure to the next possible alignment which could potentially match. For instance, this table would map a failure on the second character in the pattern to the alignment two ahead.[3]

[3]Note early on that a failure on the k-th character does not imply we will always move k ahead. This is demonstrated by a failure on the last character of the example pattern, which only allows us to skip two ahead.

- Align the pattern with the first position in the target text and perform a character by character comparison. When that comparison is complete,[4] use the table to determine the next alignment and where in the alignment to resume comparison.

9.1.2.2 Preprocessing: Defining the Overlap Function

The preprocessing table indicates how far we can shift the pattern depending on how far into the pattern a miscompare occurs. We want this shift to be as far as possible (to limit comparisons) while also ensuring we do not miss any potential matches. Our goal will be to ensure once we have successfully compared a character in the text, to never compare that character again.[5]

In order to meet the last criterion, the following must be true. Consider only the part of the pattern which matched. After the shift, the overlapping parts of the pattern must match. If there is a chance they do not match, then we would have to compare characters with our text which we had previously successfully compared.[6]

Consider the pattern ABCDABCE, and a text ABCDABCXXX. We show the first alignment, and the next possible alignment, showing that the overlapping parts of the pattern must match. Note that such an overlap does not guarantee the pattern will match at that alignment; it is a necessary but not sufficient criterion.

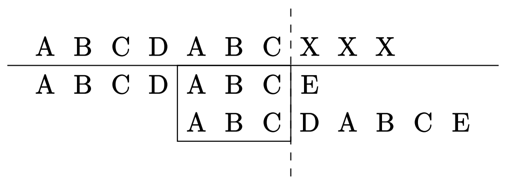


Figure 9.9: We successfully compared the first 7 letters of the pattern and text with a dashed line indicating the end of that succsessful comparison. In order to ensure we never have to compare any of those 7 letters of the text again, we must guarantee the overlapping parts of any future alignment (up to the comparison failure) are equal. We show the next future alignment with a box around that overlap. Any lesser shift results in miscompares to the left of the dashed line. Note that even though the last character of the pattern also overlaps after the shift, it is not considered as it is past the dashed line of furthest successful comparison, and thus does not need to match in order to fulfil the criterion.

We can state the overlap-match more formally:

> Let l be the maximum match of the patern at some index, and $P_{0,l}$ be the sub-pattern representing that match. If $P_{0,l}$ is shifted by

[4]either with a successful match or a miscompare

[5]This will ultimately enable us to prove linear bound.

[6]violating the criterion we are seeking to meet

k and compared against $P_{0,l}$ itself, then the overlap matches only if $P_{0,l-k}$ is the same as $P_{k,l}$. The suffix of the sub-pattern must be a prefix.

In order to ensure we do not accidentally skip a potential match, we will seek to find the minimum shift, therefore the maximum amount of overlap, and ultimately the longest suffix of the sub-pattern which is also a prefix.[7] The size of this longest suffix which is also a prefix will be known as the *overlap* function. This overlap function will be used to compute the minimum shift such that all characters of the text we have previously compared still match, and that no matches have been skipped.

We can see the values for two examples.

ABCDABCE		
Matched Substring	Overlap	Explanation
$\emptyset$	$\emptyset$ (0)	If no characters match, by defition we will shift by 1.
A	$\emptyset$ (0)	There is no suffix of A which is a prefix. We can skip all overlapping shifts.
AB	$\emptyset$ (0)	There is no suffix of AB which is a prefix. We can skip all overlapping shifts.
ABC	$\emptyset$ (0)	There is no suffix of ABC which is a prefix. We can skip all overlapping shifts.
ABCD	$\emptyset$ (0)	There is no suffix of ABCD which is a prefix. We can skip all overlapping shifts.
ABCDA	A (1)	The longest suffix which is also a prefix is A. We can shift by 4 to reach that matching overlap.
ABCDAB	AB (2)	The longest suffix which is also a prefix is AB. We can shift by 4 to reach that matching overlap.
ABCDABC	ABC (3)	The longest suffix which is also a prefix is ABC We can shift by 4 to reach that matching overlap.
ABCDABCE	$\emptyset$ (8)	There is no suffix which is a prefix. We can skip all overlapping shifts.

[7]but shorter than the entire sub-pattern,which would result in no shift!

ABCABABABC		
Matched Substring	Overlap	Explanation
$\emptyset$	$\emptyset$ (0)	If no characters match, by defition we will shift by 1.
A	$\emptyset$ (0)	There is no suffix of A which is a prefix. We can skip all overlapping shifts.
AB	$\emptyset$ (0)	There is no suffix of AB which is a prefix. We can skip all overlapping shifts.
ABC	$\emptyset$ (0)	There is no suffix of ABC which is a prefix. We can skip all overlapping shifts.
ABCA	A (1)	The longest suffix which is also a prefix is A. We can shift by 3 to reach that matching overlap.
ABCAB	AB (2)	The longest suffix which is also a prefix is AB. We can shift by 3 to reach that matching overlap.
ABCABA	A (1)	The longest suffix which is also a prefix is A. We can shift by 5 to reach that matching overlap.
ABCABAB	AB (2)	The longest suffix which is also a prefix is AB We can shift by 5 to reach that matching overlap.
ABCABABA	A (1)	The longest suffix which is also a prefix is A We can shift by 7 to reach that matching overlap.
ABCABABAB	AB (2)	The longest suffix which is also a prefix is AB We can shift by 7 to reach that matching overlap.
ABCABABABC	ABC (3)	The longest suffix which is also a prefix is ABC We can shift by 7 to reach that matching overlap.

9.1.2.3 Preprocessing: Computing the Overlap Function

As the goal of the algorithm is linear time, we must be able to compute the overlap function in linear time as opposed to the naive quadratic implementation. While it should be apparent that one can compute the overlap from the Z-func,[8] our goal is not simplicity but comprehension. The alternative[9] method shown here better extends to further algorithms such as Aho-Corasick and therefore its presentation is a pre-requisite for understanding such.

[8]The overlap is equivalent to the Z-box with the left most endpoint which covers a given letter. We can extract all the necessary values by caterpillaring through the pattern, alternatively iterating until the right end of the current Z-box is found, and then iterating until the next Z-box left endpoint is found.

[9]actually the original

From a high level, we will compute the overlaps from left to right. To compute $Olap(i+1)$, we will examine all suffixes which end at i which have a corresponding overlappling prefix in decreasing order of length, and identify the first[10] one which has a matching character following the two instances.

We make this far clearer with a picture where we attempt to compute $Olap(i+1)$ from $Olap(i)$ and all other lesser values.

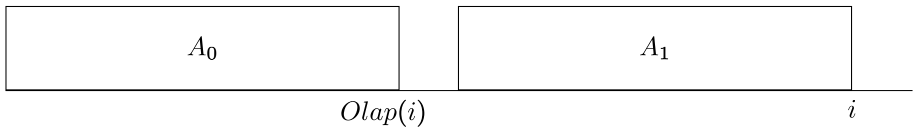


Figure 9.10: Consider $Olap(i)$. We see the overlap itself A appears in totality at both the beginning and end of the substring.

We will consider A as our first candidate and compare succeeding letters.

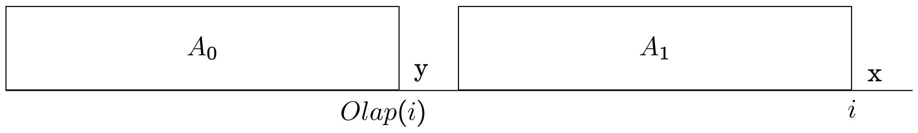


Figure 9.11: The letters after A_0 and A_1 do not match, and therefore we cannot simply extend A by 1. If those two letters had matched, we would simply set $Olap(i+1) = Olap(i) + 1$ and increment to solve for the next i.

As our current candidate failed, we must find the next longest candidate overlap.

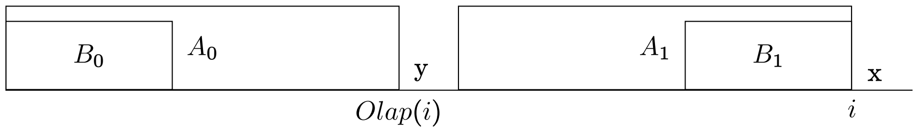


Figure 9.12: The substring B represents the next longest overlap which ends at i and occurs at the beginning of the string.

We know we have to identify B, but how can we do so efficiently? Incrementing through each suffix in A and checking whether that substring is also a prefix is too slow. The key lies in the fact that as we know that the two A boxes are exact copies of eachother, there are actually two other instances of B.

[10]and therefore longest

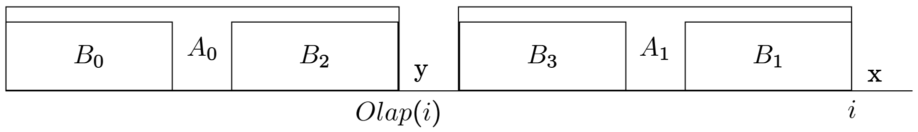


Figure 9.13: Since the two A boxes are identical, there must be at least 4 copies of B in the string: at the bounds of each A box. Note: The subscripts on the label for each box do not carry semantic meaning, and are simply for identification.

How can we use this information to quickly find B? If we look strictly at A_0, we note that there is a copy of B at both the start and end. Is there a way to look up the longest string which is both a suffix and prefix of A? Of course! It is exactly the $Olap$ function we have computed at the right endpoint of A. As the right endpoint of A is $Olap(i)$, the size of B must be $Olap(Olap(i))$ or $Olap^2(i)$.

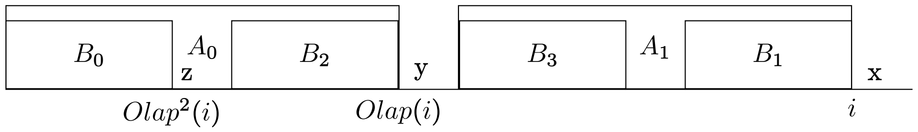


Figure 9.14: Iteratively applying the $Olap$ function enabled us to find the length of the next-longest suffix ending at i which is also a prefix. We again see that there is no match.

The fun doesn't stop there. Our next-longest overlap which is shorter than B occurs at least 8 times in the string, twice in each B-box. For clarity, we only draw the relevant instance.

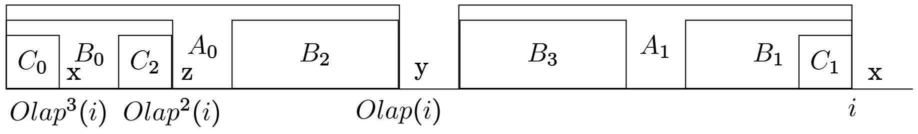


Figure 9.15: We iteratively apply $Olap$ to find the next longest suffix ending at i which is also a prefix has length $Olap^3(i)$. Examining the succeeding characteres of C_0 and C_1 shows they are both the same. We can therefore set $Olap(i+1) = Olap^3(i) + 1$.

This leaves the following overall algorithm to compute $Olap(i + 1)$ from all lesser values of i:

1. Chose l as the longest suffix ending at i which is also a prefix by looking up $Olap(i)$.
2. Examine if the character at $i + 1$ is the same as the character at $l + 1$.

    - If so, set $Olap(i+1) = Olap(i) + 1$ and return.

3. Find the next longest suffix ending at i which is also a prefix by applying $l = Olap(l)$.
4. Loop to step 2, or exit with a base case if there are no more suffixes to examine.

Implementation

Listing 9.3: C++

```cpp
//assume a pattern, p.
//olap is the maximum overlap of a substring of length i.
int olap[p.size()+1]={};

// - start with box from previous index
// - recurse until the next char matches or we hit a base case
// - save box if we found one which matched
for(int i=2;i<=p.size();i++){
   int box=olap[i-1];
   while(box&&p[i-1]!=p[box])box=olap[box];
   olap[i]=(p[box]==p[i-1])?box+1:0;
}
```

Rough Proof of Runtime We iterate through all values of i, which is at best linear time. It would seem that the recursives step in the inner loop would yield a runtime which is superlinear. We can bound the number of times we perform this inner loop as follows:

- Consider the value of $Olap(i)$ and how it changes as we iterate through i. As its value increase by at most 1 from one value of i to the next, the total amount it increases over the course of the function is at most n.
- Each call to $Olap$ for a given value of i decreases the value of the overlap function for the subsequent i. Specifically, each call decreases the maximum possible value of the subsequent $Olap$ by at least 1. If 3 calls are made, the value decreases by at least 1.[11]
- As the value of $Olap$ only increases by n and is never negative, the decreases can also not exceed n. Any more than 2 calls to $Olap$ for a given i incrementally decreases the value. Therefore the total number of calls to $Olap$ cannot exceed $3n$, which is $O(n)$.

[11] If $Olap$ is called once for a given value of i, then $Olap$ increaeses by 1. If it is called twice, the $Olap$ increases by at most 0. If it is called 3 times, $Olap$ decreaes by at least 1.

9.1.2.4 Alignment

Once the overlap function is known for all i, we can use it to compute all matches. This is largely the before described process which we formalize here.

1. Align the pattern at index 0.
2. Iterate through the pattern, starting at the index of the last successful comparison, comparing character to character
   - If we reach the end of the pattern, indicate a match
3. Based on the last successfully compared character in the pattern, i, shift to the next alignment, which is $i - Olap(i)$, and begin comparison after the last successfully compared character in the text.

We can use our original naive match as an example of the improved method.[12]

$$\begin{array}{cc|c|ccccccc}
\cline{3-3}
\mathrm{A} & \mathrm{B} & \mathrm{C} & \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{C} \\
\mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{C} & & & & & \\
\cline{3-3}
\end{array}$$

Figure 9.16: `i==0`. Comparison fails after 2 characters in p. $Olap(2) = 0$, so we shift by $2 - Olap(2) = 2$ and begin comparison at the first unsuccessfully compared character, the C.

$$\begin{array}{cc|c|ccccccc}
\cline{3-3}
\mathrm{A} & \mathrm{B} & \mathrm{C} & \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{C} \\
 & & \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{C} & & & \\
\cline{3-3}
\end{array}$$

Figure 9.17: `i==2`. Comparison fails on the first character. By definition, we will shift by 1 and must begin comparison on the next character.

$$\begin{array}{ccccccc|c|cc}
\cline{8-8}
\mathrm{A} & \mathrm{B} & \mathrm{C} & \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{C} \\
 & & & \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{C} & & \\
\cline{8-8}
\end{array}$$

Figure 9.18: `i==3`. We successfully compared the first 4 characters of p before reaching a failure. $Olap(4) = 2$, so we shift by $4 - Olap(4) = 2$, and begin comparison at the first unsuccessfully compared character, the A. Note that we will never compare any of the 4 successfully compared characters of the text again.

[12]While a reader should be able to compute it, the values for $Olap$ in this pattern are 0,0,0,1,2,0.

```
A B C A B A B A B C
          A B A B C
```

Figure 9.19: `i==5`. We began comparison at the last A, and matched the remaining ABC, finding a match. Note that we only had to compare 3 total characters in this step, as we previously compared the first two when i was 3. If the text were longer, we would shift by $5 - Olap(5)$ before continuing comparison at the character after the C.

Rough Proof of Linearity Each character in the text is only compared successfully once, for a total of $O(n)$ successful comparisons. On a failed comparison, the pattern is shifted to a new alignment. As the number of alignments is $O(n-p)$, this limits the total number of failed comparisons to $O(n)$ as well. The overlap function is also computed in linear time, leading to an overall runtime of $O(n)$.

9.1.2.5 Implementation

The implementation, while straightforward, is finicky due to the potential for off by one errors and other corner cases. For simplicitly, instead of explicitly "shifting" the pattern and aligning to the text, we track the current pointer into both the text and the pattern, and adjust those pointers as necessary.

Listing 9.4: C++

```cpp
//assume a text, t, and a pattern, p.

//preprocess
int olap[p.size()+1]={};
for(int i=2;i<=p.size();i++){
   int box=olap[i-1];
   while(box&&p[i-1]!=p[box])box=olap[box];
   olap[i]=(p[box]==p[i-1])?box+1:0;
}

//align
//the outer loop moves i through t one char at a time
//inside the loop, we shift p until it matches at i, or hit a base case
//if we have gotten j all the way through p, it indicates a match, so
//we have to shift to set up for next match
for(int i=0,j=0;i<t.size();i++){
   while(j&&t[i]!=p[j])j=olap[j]; //shift
   if(t[i]==p[j])j++; //advance in p
   if(j==p.size()){ //match found!
      cout<<i+1-p.size()<<"\n";
      j=olap[j];
   }
}
```

9.1.3 Multi-pattern Matching/Aho-Corasick

With KMP, we were able to optimize matching by iteratively aligning a pattern with a text, and then optimizing by only checking certain alignments. *Aho-Corasick* allows us to search for multiple patterns at once using similar intuition.

Naively, if we were to run KMP on each of P patterns, we would find we take $O(nP)$ time, as we need to interate through the text independently for each pattern. We will see how this time can be reduced in the following high level stages.

1. Match all patterns in a single pass of the text
2. Optimize the matching to yield a runtime proportional to $O(n+p)$, where n is the length of the text, and p is the total length of all patterns
3. Address the corner case where one pattern is a substring contained in another

9.1.3.1 Single-Pass Matching

When we first introduced KMP, we saw in section **??** that we could naively attempt to align a pattern at each index in the text. We can modify this implementation to cope with multiple patterns in a single pass of the text using a trie, and aligning each index in the text with the root of the trie. Once there aligned, we will walk the trie to see if we reach the end node of some pattern.

We will walk through an example of this using the following words:[13]

- booboo
- booster
- oboe

The trie containing these three words appears as follows:

[13]Note that none of these are substrings of any other, a point which will be addressed in section **??**

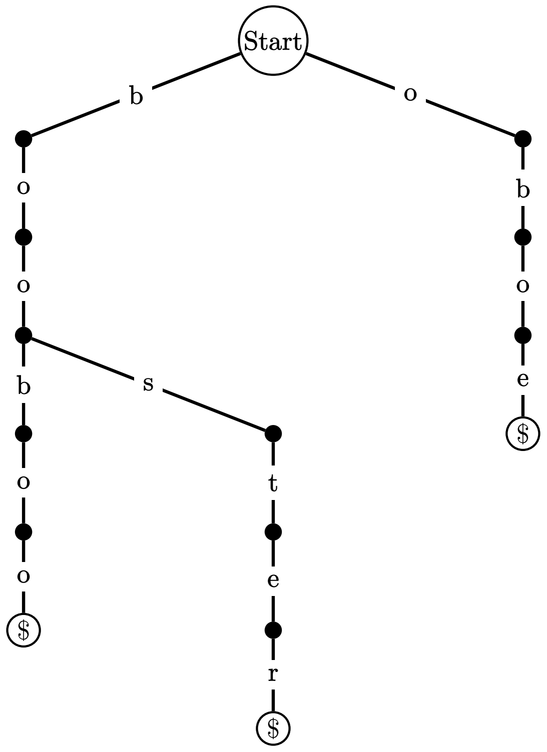


Figure 9.20: Nodes where a pattern terminates are marked with a '$'.

Consider matching these words against the text "obeobooboe". We attempt to walk the trie starting at each index into the text.

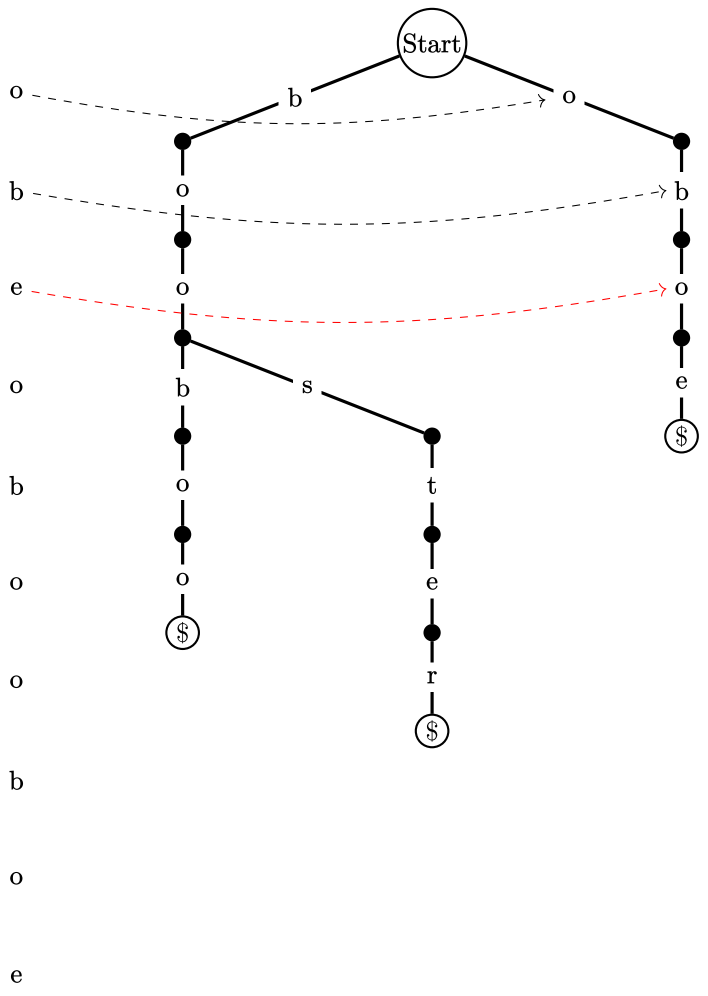


Figure 9.21: We walk the trie from the root. Upon comparing the third character, we find there is no outgoing edge where we can continue to match against. This is indicated by the red arrow, where the only outgoing edge from our current node is 'o', but the text is 'e'. There is no pattern that matches at this alignment.

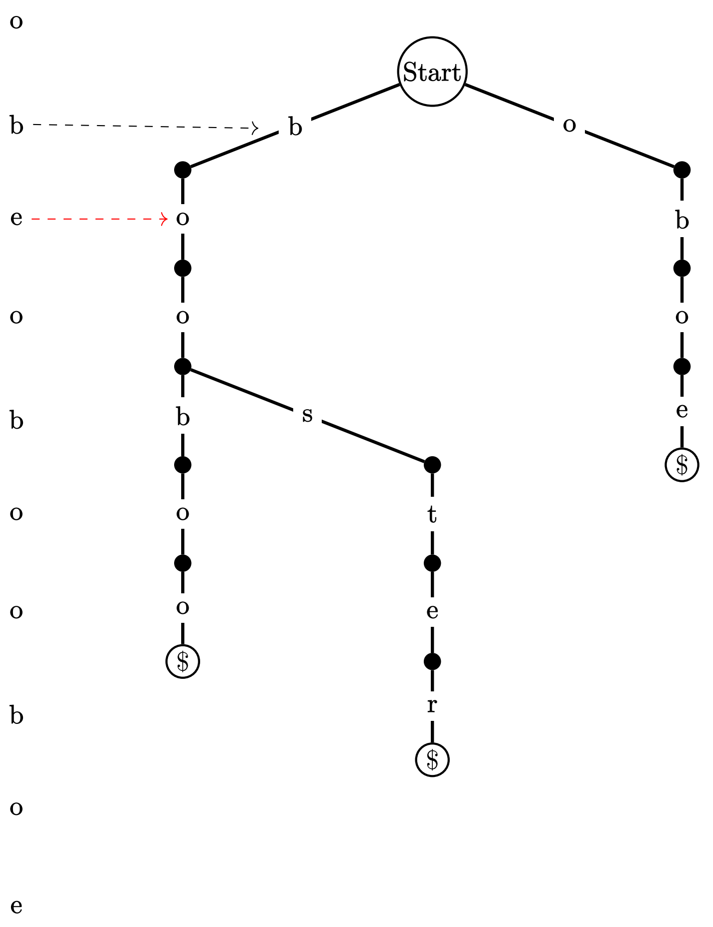


Figure 9.22: At the next alignment, we again fail to find a match.

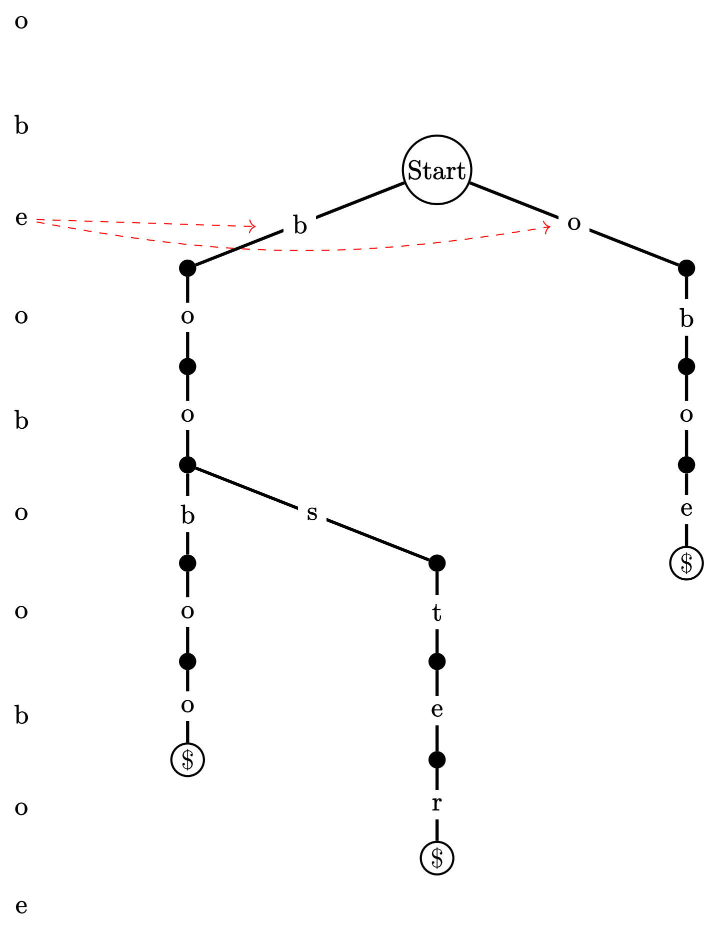


Figure 9.23: Checking the next alignment doesn't get far. This character matches none of the edges emenating from the current node.

We can continue checking alignments, and see that all of them fail until we reach one at the end:

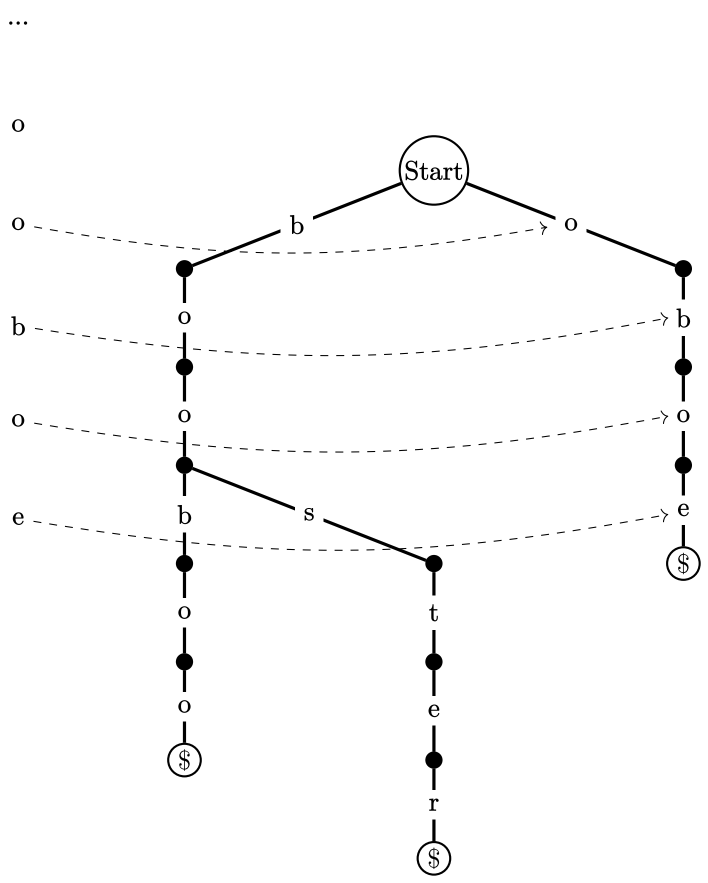


Figure 9.24: At the final alignment, we reach the end node of some pattern, indicating the text matches the pattern at this alignment.

After trying all possible alignments against the tree, we have found one alignment which match some pattern in the trie. These constitute all matches of the three patterns to the text. As we have tried each of n alignments, and each alignment requires $O(n)$ time to walk the tree, our runtime is $O(n^2)$.[14]

9.1.3.2 Preprocessing: Defining Overlap Edges

If we examine the alignments starting at index 1 (*be...*) and 2 (*eo...*), we notice we can optimize as we did with KMP. The failed character (e) does not appear at the start of any pattern, and therefore the next alignment (which starts with e) cannot match any pattern, and can be skipped. To achieve this with KMP, we defined the *Olap* function which provides the longest suffix of the pattern where our comparison last succeeded which is also a prefix of the pattern.[15]

[14]It is also $O(np)$.

[15]See the KMP section for further details on the intuition behind this.

After shifting by $Olap$, we were able to resume comparison at the same position in the text.

We can apply similar logic to our trie in the following manner:

1. Compute the longest prefix of any pattern which is a suffix of the pattern we were matching when comparison failed

2. Identify where in the trie to resume comparison to ensure we don't successfully compare a character of the text more than onces[16]

With KMP, the second condition was met trivially. When we shift by the $Olap$, we know where to begin comparison again by simple subtraction.[17]. This is more difficult with a trie, as after the shift, we don't necessarily know which pattern or patterns we should continue matching against, or put another way, which branch of the trie we are in. To solve this purpose, we will extend the $Olap$ function to return instead of a length, a node in the trie. From each node in the trie, if a comparison fails, instead of subtracting the $Olap$, we will traverse this special $Olap$ edge and resume comparison.[18]

The edge is formally defined as follows.

> Consider the sequence of letters required to traverse from the root to some node as that nodes label. Consider all suffixes of this label, and further, such suffixes which themselves are the label of some other node in the trie. The Olap edge is the edge from the former node to the node which is labeled by the longest such suffix.

Each node has an Olap edge (the degenerate case points back to the root), and it is unique.[19] If suffix edges were drawn on a trie which only had a single pattern, the edges would traverse exactly from a node representing character i to the node representing character $Olap(i)$, as defined by KMP. In KMP, we resume comparison with the character immediately after that defined by the $Olap$, and here, we resume comparison with the characters immediately after the node defined by the Olap edge.

Our trie is redrawn here with all Olap edges shown.

[16]As with KMP, achieving linear time requires such a strict condition

[17]See the Alignment section

[18]This edge is called various things in various sources, such as "Suffix edge" or "failure links". We choose our terminology to demonstrate the relationship to KMP.

[19]Every string has at most one node representing it in the trie, and all considered suffixes have distinct length.

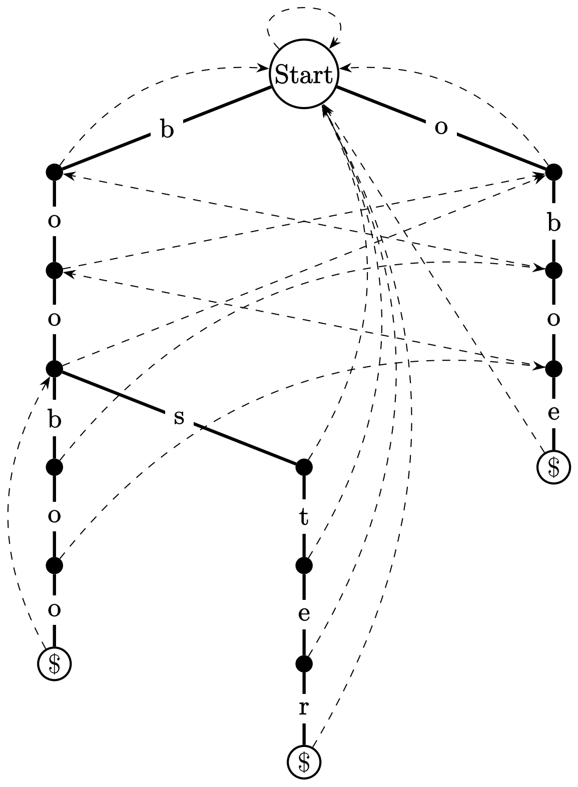


Figure 9.25: Nodes with their Olap edges drawn in dashed lines. Each node has an edge to the node such that the overlap of the destination nodes label and the suffix of the label of the source node is maximal.

The visualization of the Olap edges is very cluttered, so let's view a description of each such edge in table form:

Label	Overlap Label	Explanation
$\emptyset$	$\emptyset$	If no characters matched, and therefore we are still at the start node, we will shift by one and continue comparison at the root.
b	$\emptyset$	There is no suffix of b in the trie. We resume comparison at the root.
bo	o	The longest suffix of bo which is the prefix of some pattern is o. We resume comparison matching oboe.
boo	o	The longest suffix of boo which is the prefix of some pattern is o. We resume comparison matching oboe.
boob	ob	The longest suffix of boob which is the prefix of some pattern is ob. We resume comparison matching oboe.
boobo	obo	The longest suffix of boobo which is the prefix of some pattern is obo. We resume comparison matching oboe.
booboo	boo	The longest suffix of booboo which is the prefix of some pattern is boo. We resume comparison matching booboo or booster.
boos	$\emptyset$	There is no suffix of boos in the trie. We skip all overlapping shifts and resume at the root.
boost	$\emptyset$	There is no suffix of boost in the trie. We skip all overlapping shifts and resume at the root.
booste	$\emptyset$	There is no suffix of booste in the trie. We skip all overlapping shifts and resume at the root.
booster	$\emptyset$	There is no suffix of booster in the trie. We skip all overlapping shifts and resume at the root.
o	$\emptyset$	There is no suffix of o in the trie. We resume comparison at the root.
ob	b	The longest suffix of ob which is the prefix of some pattern is b. We resume comparison matching booboo or booster.
obo	bo	The longest suffix of obo which is the prefix of some pattern is bo. We resume comparison matching booboo or booster.
oboe	$\emptyset$	There is no suffix of oboe in the trie. We skip all overlapping shifts and resume at the root.

The table descriptions are naturally very similar to those presented for KMP. The major difference is instead of a raw length, we produce a node itself. The number of alignments which end up ”skipped” is implied by the difference in length between the label and overlap label.

9.1.3.3 Preprocessing: Computing Overlap Edges

While it may seem difficult to compute these edges, the logic by which we compute the *Olap* function for KMP applies almost directly.

The following algorithm computes $Olap_{edge}(n_c)$ for some node n_c, representing a node n's child, following an edge labeled with character c, assuming $Olap_{edge}$ has been computed for n and all its direct ancestors.

1. Choose p as the node whose label contains the longest suffix of n by looking up $Olap_{edge}(n)$.
2. Examine if p has an outgoing edge c, pointing at some p_c.
    - If so, set $Olap_{edge}(n_c)$ to p_c and return.
3. Find the node whose label contains the next longest suffix of n by applying $p = Olap_{edge}(p)$.
4. Loop to step 2, or exit with a base case if there are no more suffixes to examine.

The proof of correctness is nearly identical to the justification of the KMP algorithm. At each step, the label of p is the longest suffix which terminates at n, and therefore the longest suffix which terminates at p and exists in the trie must be exactly the next longest such suffix of n.[20]

Similarly, the proof of linearlity follows from that of KMP.

1. Consider the depth of the node pointed to by $Olap_{edge}$ as we progress down a branch of the trie. It increases by 1 from one node to the next. Therefore the total amount it increases over the progression from the root to some leaf node is at most the depth of that leaf.
2. Each recursive call to $Olap_{edge}$ for a given value of n_c reduces the depth of the candidate p_c by at least 1.
3. As depth cannot be negative, and the total increase for any branch is linear, the total number of calls to $Olap_{edge}$ to compute that function for the entire branch is also linear. Therefore, the total number of calls to $Olap_{edge}$ for the entire tree must be $O(p)$.

[20]The reader is encouraged to review the computation of the *Olap* function for KMP if this is not fully understood. The boxes there-presented apply equally to branches of the trie as they do to a single pattern.

9.1.3.4 Alignment

Once the overlap edge is known for all nodes, we can use it to compute all matches. This is done simply by comparing each character of the text in turn with our current location in the trie. If an edge exists, we follow it, otherwise we follow the $Olap_{edge}$. The alignment is implicitly defined by the depth in the trie and the charatcter we are currently comparing.[21]

We return to our example.

[21] Recall that our goal is to only successfully compare each character once, so we can maintain an index of that character to imply the alignment without actually storing it.

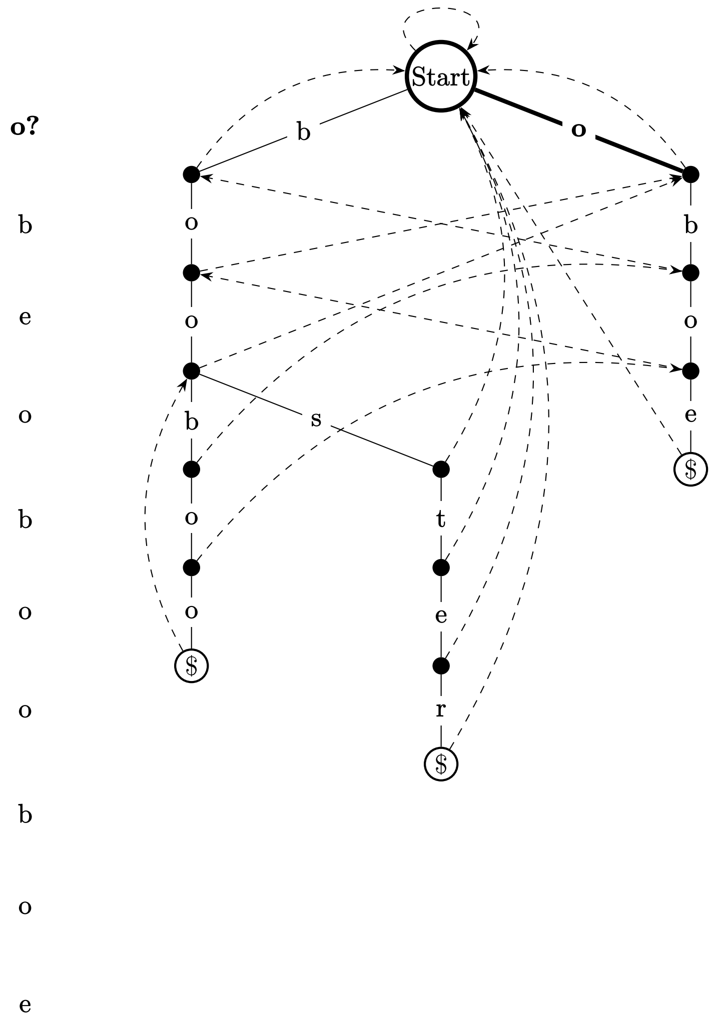


Figure 9.26: In the first alignment, and starting at the root, we match an outgoing edge o.

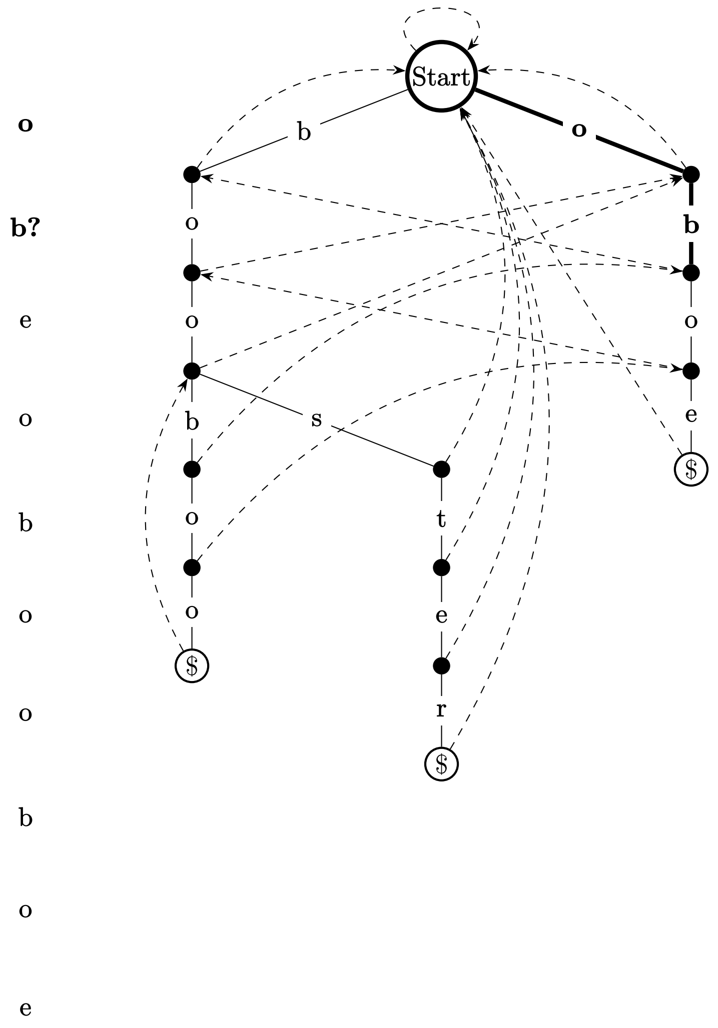


Figure 9.27: Continuing from the o node, we match an outgoing edge b.

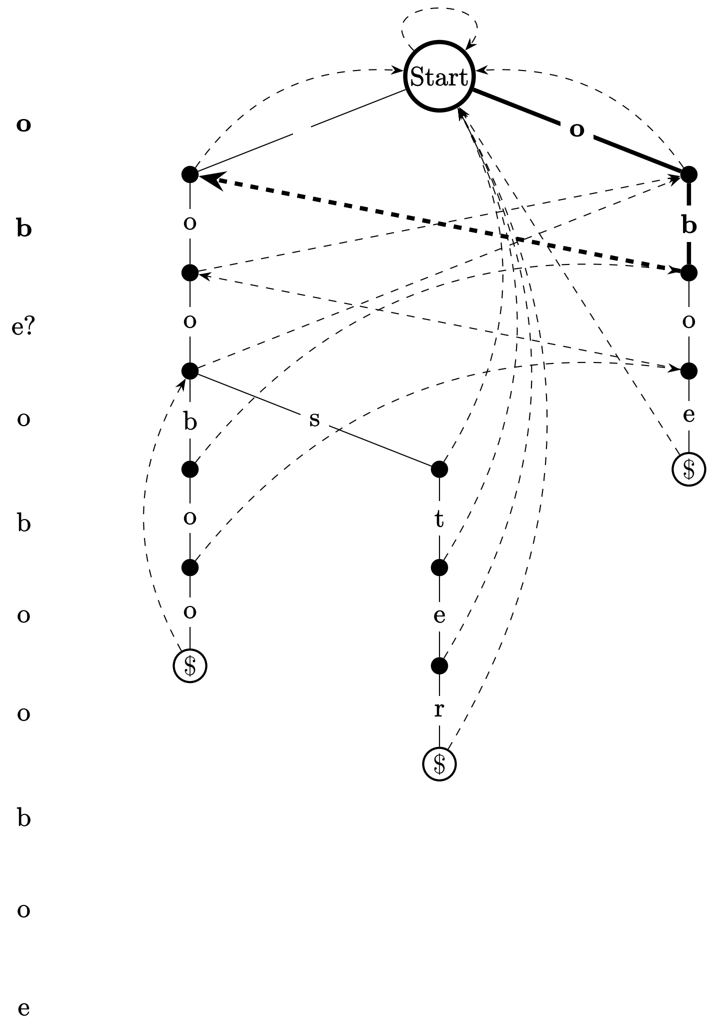


Figure 9.28: Continuing from the ob node, we do not find a match for e, so we follow the overlap edge.

Note that as we have followed an overlap edge, the depth of the node we are currently at has decreased. As we are still comparing the same character in the text, as indicated by the question mart, the alignment of the text against the trie shifts. This is seen in the following image, where instead of the 'o' being the first character matched to some edge, the 'b' is.

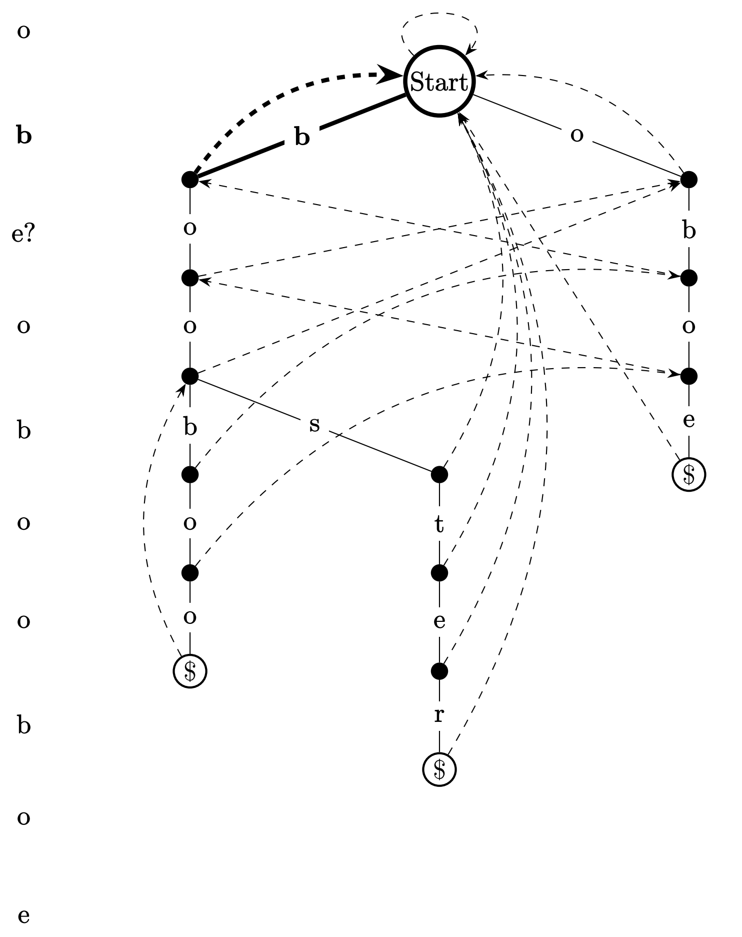


Figure 9.29: Continuing from the b node, we do not find a match for e, so we follow the overlap edge back to the root.

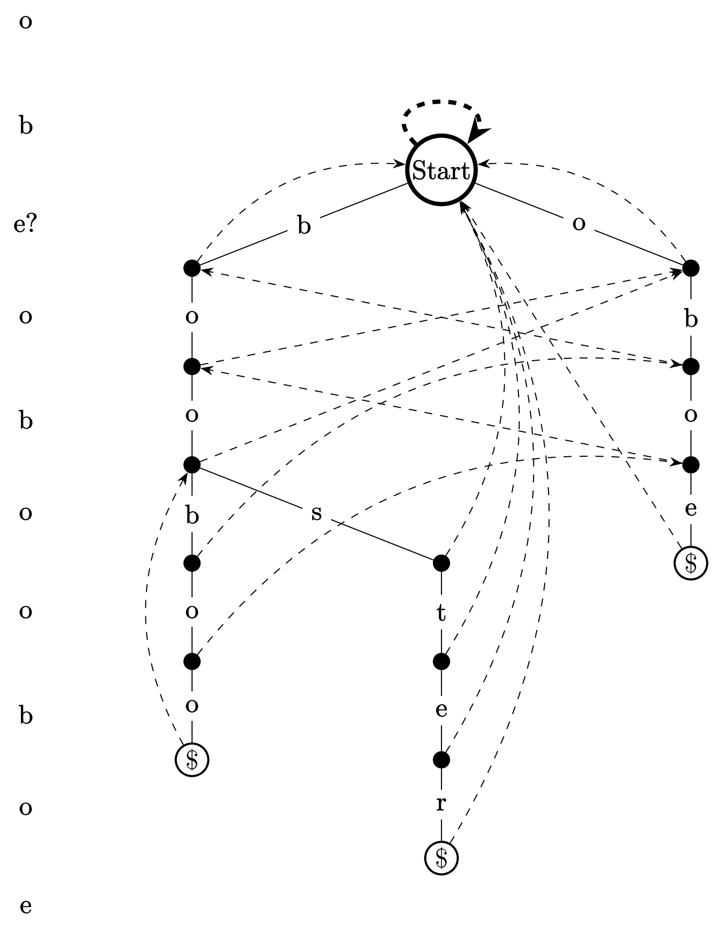


Figure 9.30: Continuing from the root node, we do not find a match for e, so we follow the overlap edge back to the root.

As we are traversing the root node to itself, we know that no pattern can match at this alignment in the text. We explicitly increment the alignment of the text and the trie. This is the only time in the algorithm this happens explicitly. It is regulaly implied by the decrease in depth of the current node we are comparing to relative to the text.

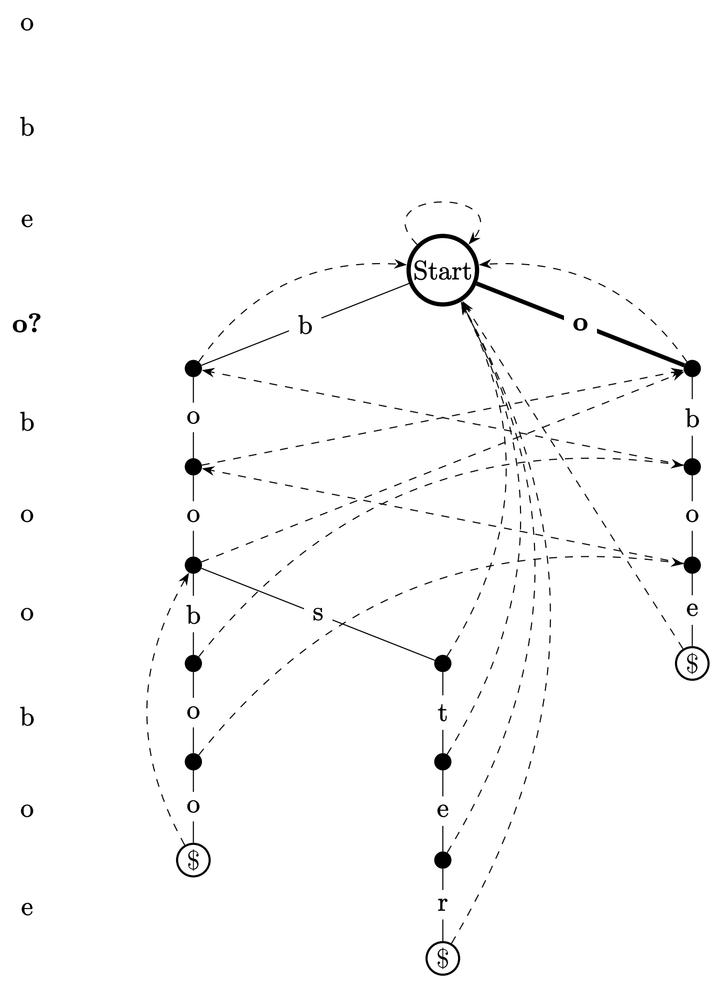


Figure 9.31: Starting at the root, we match the outgoing edge o.

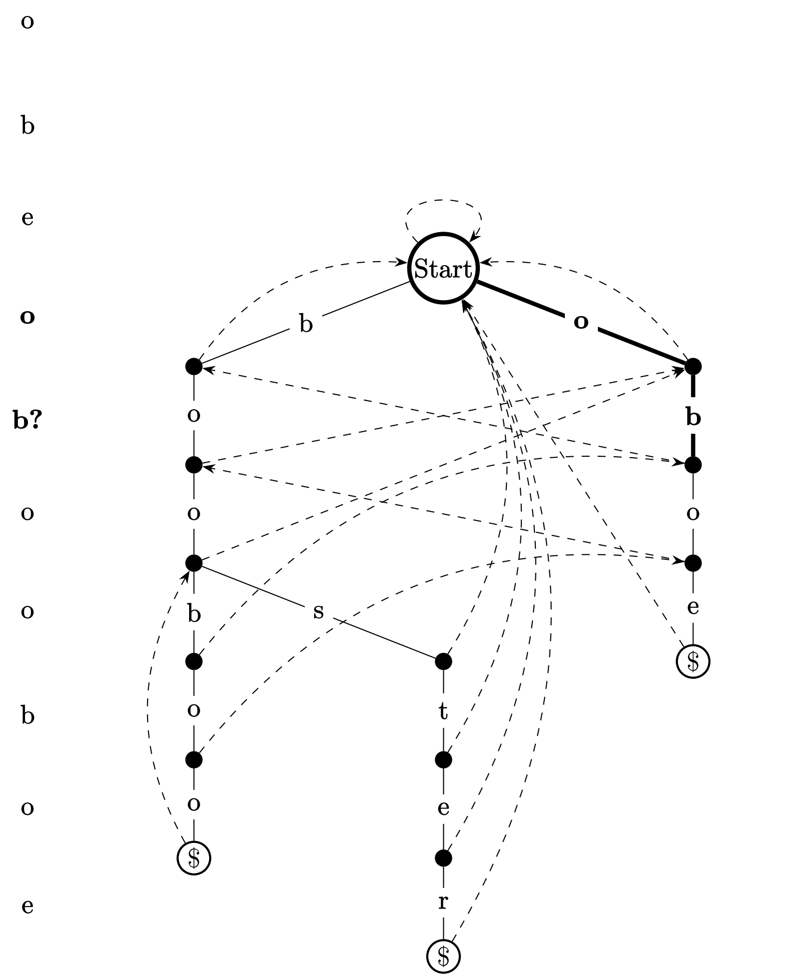


Figure 9.32: Continuing from the o node, we match an outgoing edge b.

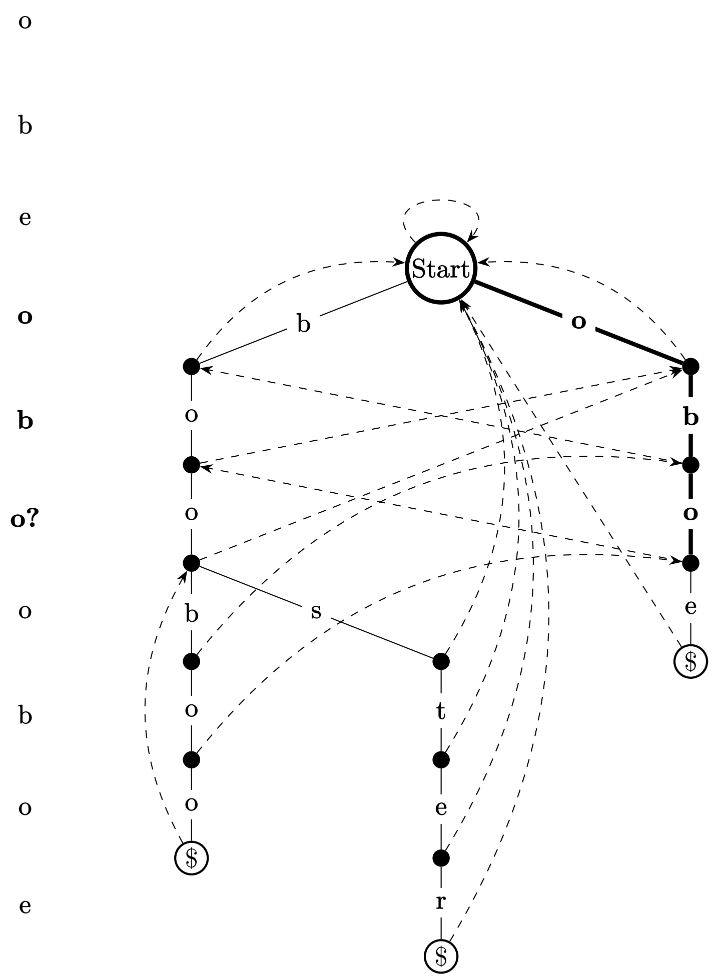


Figure 9.33: Continuing from the ob node, we match an outgoing edge o.

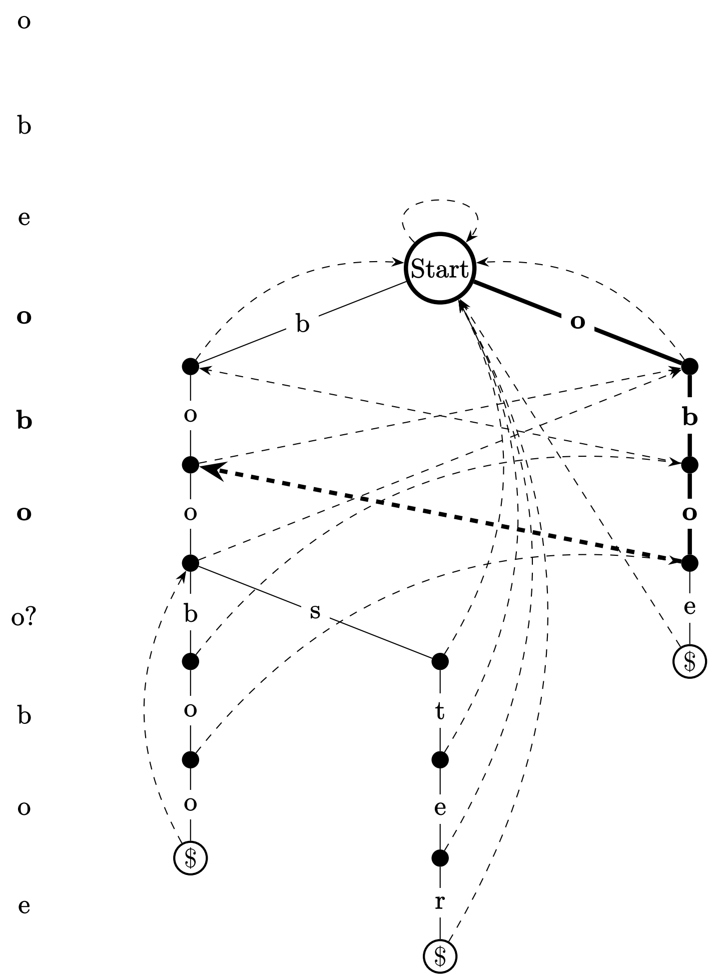


Figure 9.34: We do not find an outgoing edge for the o, so we follow the overlap edge.

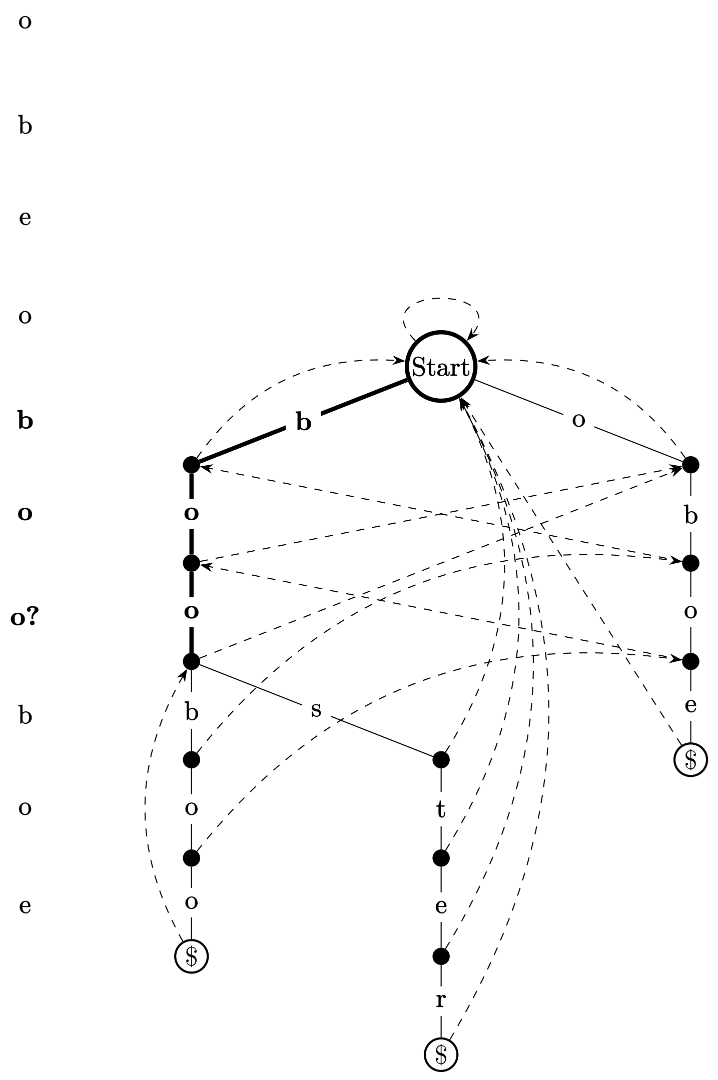


Figure 9.35: After implicitly following the overlap edge, we now find a match for the o.

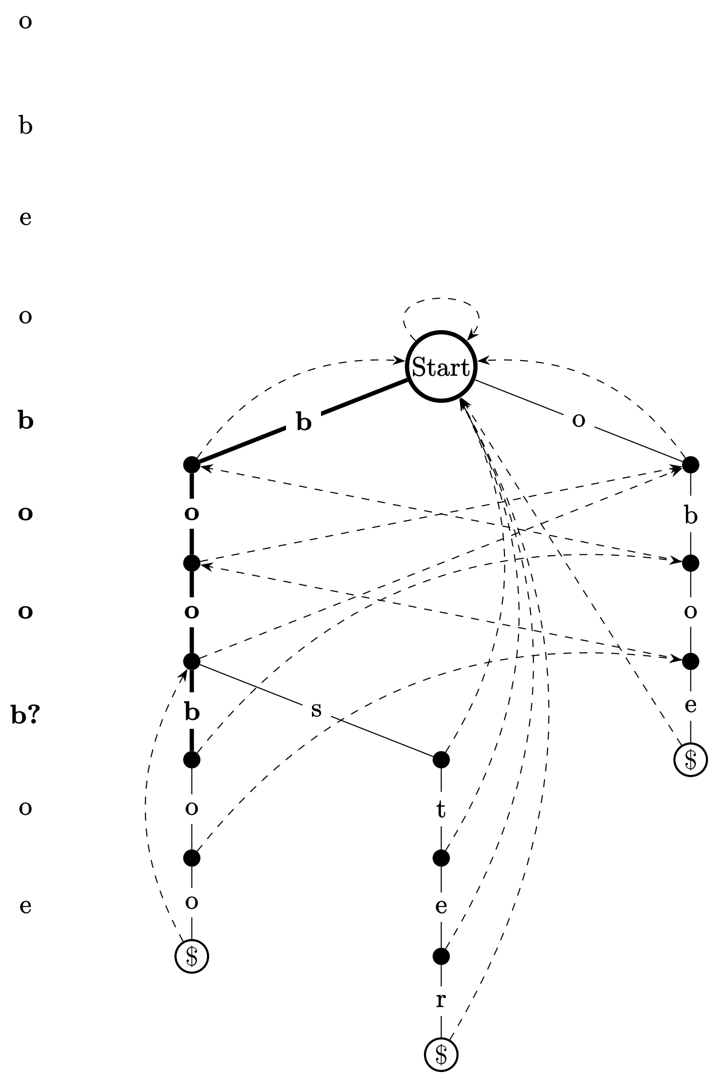


Figure 9.36: We find a match for the b.

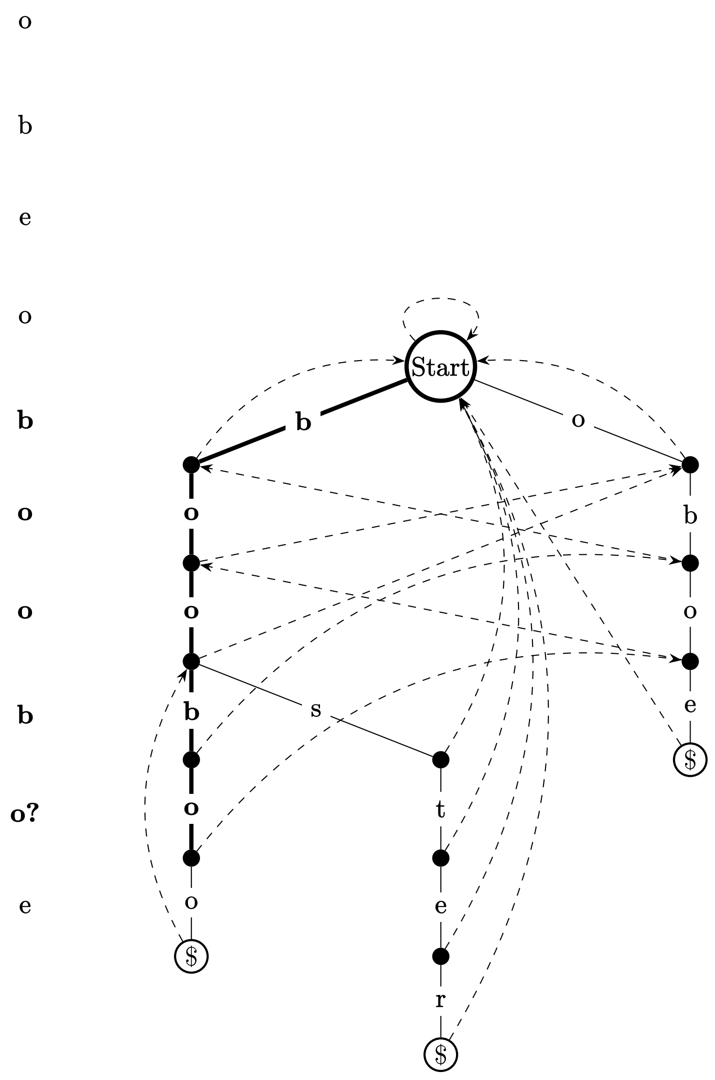


Figure 9.37: We find a match for the o.

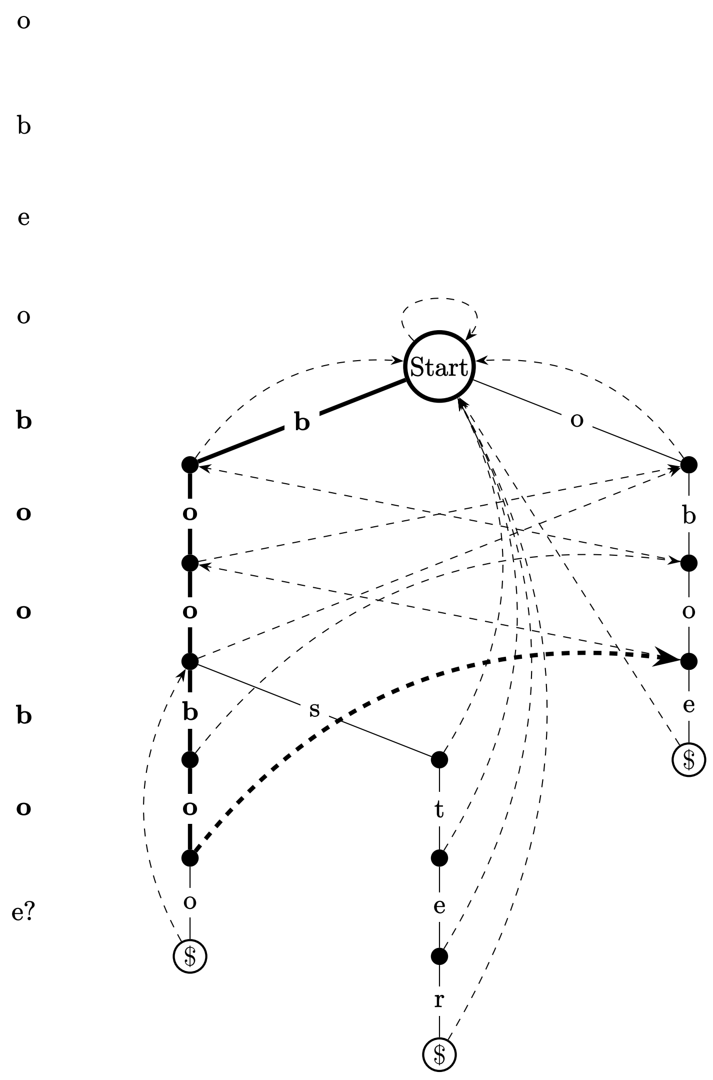


Figure 9.38: We do not find a match for the e, so follow the overlap edge.

After following the overlap edge in this stage, note that as the depth of the node we are on decreased by 2, we have effectively skipped one alignment of the text relative to the tree, saving valuable computation.

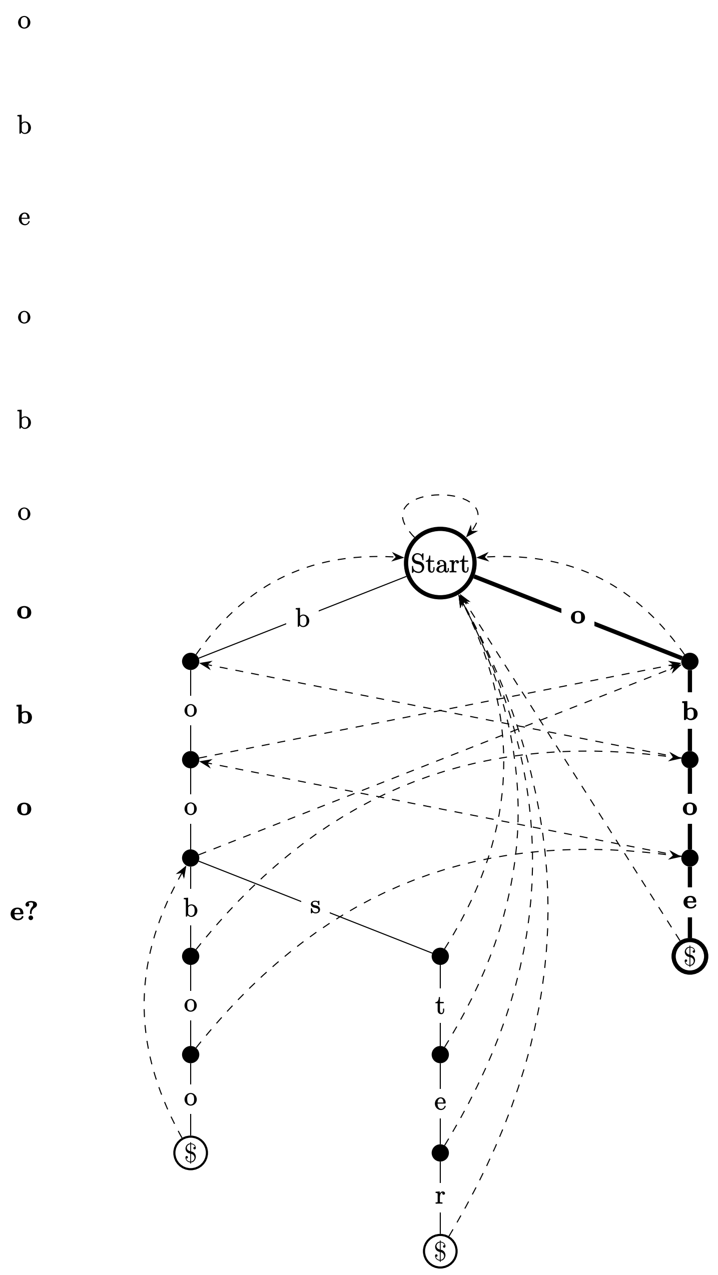


Figure 9.39: We find a match for the e.

At this stage in the algorithm, we have arived at the terminal node of some pattern. We should at this point record the match, including which pattern we found, and at which index we found it. While our text ends here, were it to continue, we would note there are no outgoing edges from that node, and follow the overlap edge, as usual.

Proof of Linearity Each comparison of a character either results to progressing to the next character in the text, or following an overlap edge. As the overlap edge always decreases the depth of the current node by at least 1, the alignment of the text to the trie is also shifted by at least one. As there are only $O(n)$ alignments, the total runtime of this step is also $O(n)$. Combined with the time to compute the overlap edges, we have performed the algorithm in $O(n + p)$ time.

9.1.3.5 Corner Case: Patterns Containing Other Patterns

Recall we made the assumption that no pattern is a substring of some other pattern. Our previous example was precisely constructed to avoid this possibility. If it were the case, executing the algorithm as described above may miss matches. We can see this trivially by adding the pattern "ob" to our search. The modified trie appears as follows:

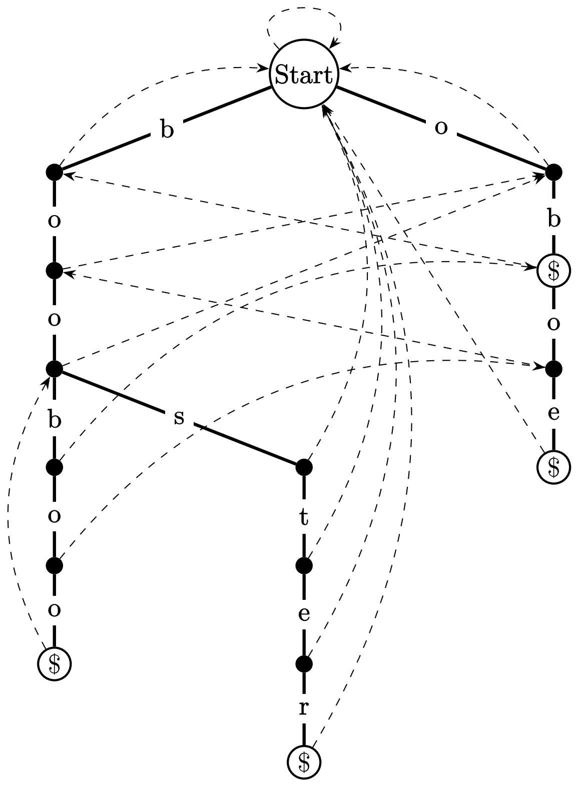


Figure 9.40: The ob node now indicates a pattern terminates there.

It is easy to see this pattern may be missed. Consider matching simply the word "booboo." Executing the algorithm as described will discover the match of booboo itself,, as well as the match of the entire word, but will miss the the occurrence of 'ob'.

To resolve this issue, we need a way to indicate while traversing distal parts of the tree, that we may have traversed a match of some other contained pattern. In this case, for every instance of ob in the trie, we must be able to know that ob is a match in itself. More generally, if the suffix ending at a given node is a complete match elsewhere in the trie, we must indicate so. To accomplish this, we will link all such nodes with a third type of edge we'll call a pattern edge.[22]

A pattern edge will be drawn from a node to a terminal node whose label is both a suffix of the source node, and for which the length of that match is the longest of any other matching suffix in the trie. By this definition, we can glean two things:

1. Each node has at most one pattern edge.[23]
2. By recursively following pattern edges from a given node, we can identify all patterns which are suffixes of the label of the that node.[24]

The trie amended with the pattern edge appears as follows:

[22] Also known as a dictionary link or output link in other texts. There is no good justification for not choosing one of these terminologies here, but we haven't used precendented terminology yet, so why start now?

[23] There is at most one pattern of a given length which is wholly contained and terminates at any particular index in any individual pattern.

[24] If two patterns of different lengths terminate at the same index in some third pattern, the shorter of those two patterns must be a suffix of the longer, and thus have a pattern edge, or by induction, a chain of pattern edges to reach from the longer to the shorter.

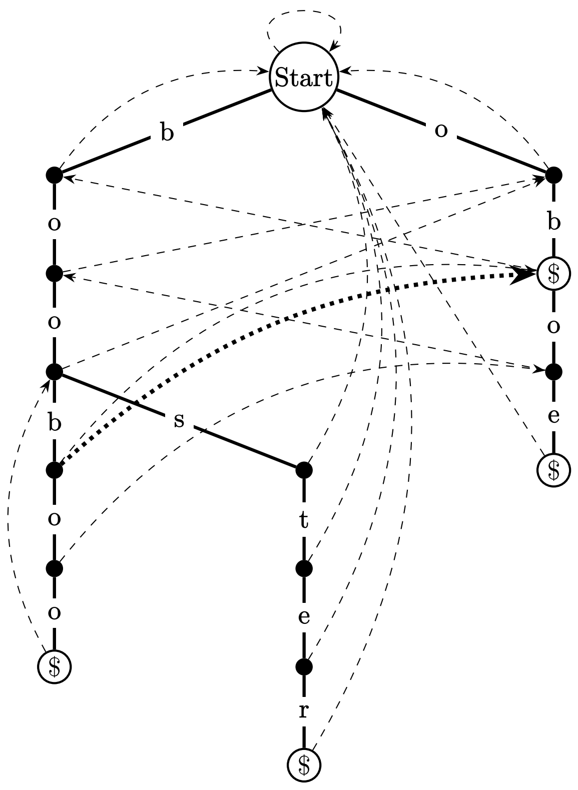


Figure 9.41: The dotted edge indicates there is a remote node in the tree which has a suffix "ob". Traversals of that node must indicate that a match was found.

If we knew all such pattern edges, we could amend the algorithm as follows:

> When traversing a node, if there is an outgoing pattern edge, recursively follow it and record all matches. Once complete, resume matching at the original node.

In the case of our example matching "booboo," we traverse the node "boob" and detect the pattern edge. We note the match represented by the destination "ob" node, as well as there are no pattern edges originating from that node. We then proceed matching from the original "boob" node. Care must be taken when following an overlap edge after following a pattern edge to ensure that the pattern is not matched a second time, nor are subsequent edges duplicately followed, as might be the case if matching booboe.[25]

Upon adding this modification, we add another term to the runtime, k, the number of matches. As we may have to follow a pattern edge to form a match

[25]Note that while in this case the pattern and overlap edges are the same, that is not always the case. We can guarantee, however, the overlap edge is at least as deep as the pattern edge, and that if they are not the same, then the source and destination of the overlap edge both contain exactly the same pattern edge.

without advancing either the current character in the text or the alignment of the text to the trie, it must be added, leading to a runtime of at least $O(n+p+k)$.

Computing Pattern Edges The question then remains how to efficiently compute the edges. As the pattern edges are a subset of all suffixes of a given node which exist in the trie, we can do this while computing overlap edges using the following rule:

1. If $Olap_{edge}(n)$ is the terminal node of a pattern, draw a pattern edge. It is, by definition, the longest suffix which exists in the trie, and thus there can be no longer pattern which is a suffix.
2. If $Olap_{edge}(n)$ is not a terminal, but has a pattern edge, draw a pattern edge fron n to the destination of that preexisting pattern edge.[26]
3. Otherwise, do not draw a pattern edge.[27]

As this process adds only a constant time evaluation at each node, it does not impact the overall runtime of what is now a complete Aho-Corasick algorithm.

9.1.3.6 Implementation

Despite being substantially similar in idea,[28] the implementation of aho-corasick is far trickier, as it involves the construction of a tree[29] rather than a simple integer valued function.

[26]Suppose this node is not the longest complete pattern which is a suffix of n. Either it is longer than the destination of the overlap edge(in which case the definition of the overlap edge is contradicted as this suffix is longer), it is the same length as the destination of the overlap edge (violating the assertion that node is not a terminal), or it is shorter than the length of the destination of the overlap edge (in which case it also must be the pattern edge of $Olap_{edge}(n)$).

[27]The proof that this finds all terminal edges is a direct corollary to the proof of case 2. If any pattern existed, it must be findable via the overlap edge.

[28]formally, KMP is a specific case of Aho-Corasick

[29]with multiple types of edges