

Boyer-Moore Algorithm

Alexander Duong

July 2016

1 Boyer-Moore Algorithm

The idea of the boyer-moore pattern matching algorithm is this: If we search for the pattern P in the text T , and compared backwards then we can get a larger shift of P when we have a mismatch which reduces the number of guesses we have to make.

We use two heuristics with Boyer-Moore to calculate the next guess after a mismatch: bad character and good suffix. (i.e. we have two arrays we use to calculate where we can jump to in the text after a mismatch).

Last-Occurrence Function

The Last-Occurrence function deals with the bad character heuristic. When we're matching backwards with Boyer-Moore, what should we do when we fail? The bad character heuristic gives us one way to deal with the problem. If the character is in our pattern, we can shift our pattern to match the last occurrence of the character in the pattern. If the character is not in the pattern then we can shift our pattern past the current section of text we're looking at since we'll never have a match.

The key idea is when a mismatch occurs at $T[i] = c$

- If the pattern P contains the character c , we can shift P to align the last occurrence of c in P with $T[i]$. of c and align it with $T[i]$.
- If c was not in the pattern then we can shift the beginning of the pattern to match up with $T[i+1]$.

To put this into practice, we build the last-occurrence function $L(c)$. We consider all the characters c in both the pattern and in the text. $L(c)$ is then defined as:

- Set $L(c)$ equal to the largest index i such that $P[i] = c$ if i exists
- Set $L(c)$ equal to -1 if the character is not in the pattern

For example, if we have pattern $P = \text{abacab}$ and our text is 'abadacab' then we get the last occurrence function below.

c	a	b	c	d
$L(c)$	4	5	3	-1

Good Suffix Array

The good suffix array deals with the good suffix heuristic (who would've thought?). This is a difficult array to understand at first, but if you really look at what we're doing then it's not that hard. Now consider this situation: we're matching backwards and we fail a match. But, I notice that there is a substring in my text (which I know because I matched a suffix in my pattern to it) that I have in my pattern... Why don't I just shift my pattern to match the substring in it to the matching one in the text? That's exactly what the good suffix array wants to do.

Now mathematically, what we're doing is this: our good suffix array S of size m : for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i + 1..m - 1] = P[j + 1..j + m - 1 - i]$ and $P[j] \neq P[i]$. If we have negative indices then it makes the condition true.

Sadly, the mathematical definition doesn't really help you intuitively understand what is happening. Let's go through building a suffix array to figure out what we're actually doing.

i	0	1	2	3	4	5	6	7
$P[i]$	b	o	n	o	b	o	b	o
$S[i]$								

To build $S[i]$ we have to remember what we're actually doing in Boyer-Moore. Remember, we're comparing backwards. What the suffix array is doing is considering what happens when we fail at a certain point in our pattern. Lets go through an example and then generalize the process.

We start at $i = 7$. If I fail at $P[7]$ it means I've matched nothing (i.e. the empty character) and I've failed at 'o'. Now, if you remember, we're trying to match substrings that we've matched in the text to our pattern. The important part is, we want to look for the largest index such that I have the same pattern, but it cannot be prefixed by the current letter we're on. Now think about why this is. For the suffix array, we're looking at what happens when we fail on a character. If I shift my pattern so that the substring I've matched in the text and pattern are the same but I have the same character prefixing it, it will just fail again because we know that we failed on the character in our first attempt to match.

So where is the next time I have matched nothing, but it is not prefixed with an o? Well, since nothing is matched everywhere, the next time it matches is

at $i=6$. $P[6] = b$ so it satisfies both conditions. Thus, my suffix array is now:

i	0	1	2	3	4	5	6	7
P[i]	b	o	n	o	b	o	b	o
S[i]								6

How do we generalize this? Let's consider this, the first part of the definition for the suffix array means we're looking for a pattern [pattern] that appears in both the text and the original pattern P. The second part is saying that we will fail at a character [c], so we're looking for the largest index where we find [pattern] but prefixed by something other than [c].

Let's try this with the next index $i = 6$. At this point, I have failed at character 'b'. So my [c] in this case is 'b'. This means that I have matched 'o' so my [pattern] = 'o'. To get the index for the suffix array I must find the largest index where I can find [pattern] such that it is not prefixed by [c] (i.e. [something that is not c][pattern] i-where this means concatenation). At $i = 4$, we have [something] concatenated to [pattern] but [something] = [c]. So if we shifted our pattern here, it would just fail immediately because I know that it failed earlier with [c]. I move on to $i = 2$. now i have [something] concatenated to [pattern] where [something] != [c]. So I can put 2 into S[6].

i	0	1	2	3	4	5	6	7
P[i]	b	o	n	o	b	o	b	o
S[i]							2	6

Let's keep going. [pattern] = 'bo' now and [c] = 'o'. Where is the largest index such that I have [something][pattern] where [something] != [c]. At index $i = 3$, I have [something] concatenated to 'bo' but [something] = [c] which means it wouldn't work. The next time I see pattern is starting at index 0 and 1. This means that the index of [something] (and the value of our suffix array) is $i = -1$. According to our definition earlier, negative indices automatically satisfy our condition. Thus, $S[5] = -1$. I will get into more about how to deal with the negatives when we get to bigger negatives.

i	0	1	2	3	4	5	6	7
P[i]	b	o	n	o	b	o	b	o
S[i]						-1	2	6

For $i = 4$, [pattern] = 'obo' and [c] = 'b'. Where is the next time I find [pattern] prefixed with something other than [c]? At index $i = 2$, we have [something][pattern] where [something] = 'n' != [c]. Thus $S[4] = 2$.

i	0	1	2	3	4	5	6	7
P[i]	b	o	n	o	b	o	b	o
S[i]					2	-1	2	6

Woooo! Starting to look easy am I right? For $i = 3$, $[\text{pattern}] = \text{'bobo'}$ and $[c] = \text{'o'}$. If I try to look for the $[\text{pattern}] = \text{'bobo'}$ in my pattern I can't actually find it again... What do I do? Well, first of all do not try my other steps of 1. Give up, 2. Cry, 3. Become a McDonald's worker. We can figure this out! What we want to do is this: We will get the negative indices by extending our pattern $P[i]$ to negative indices i by concatenating $[P][P]$. BUT!!!! We will overlap the largest prefix that is a suffix in the pattern. In our case, this is 'bo' so I will have the pattern 'bonobobonobobo'. So we start our original pattern at $i = 0$, and everything before that we just count down into the negatives. BUT!!!! we're going to look at two different concatenations. One will be $[P][P]$ let's call this and the other will be $[P][P]$ EXCEPT we will overlap the largest suffix that is a prefix. In this case, this is 'bo' so I will have the pattern 'bonobobonobobo'. So we start our original pattern at $i = 0$, and everything before that we just count down into the negatives. Now, to build the suffix array, we will look at both of these new concatenated patterns and pick the one that gives use the largest index.

i	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
P[i]	b	o	n	o	b	o	b	o	b	o	n	o	b	o	b	o

i	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
P[i]	b	o	n	o	b	o	b	o	n	o	b	o	b	o

Now using these two new concatenated patterns, where can I find $[\text{pattern}] = \text{'bobo'}$ prefixed with $[\text{something}] \neq [c]$? Let's consider $[P][P]$ WITHOUT the overlap. Well at index $i = -3$, we have $[\text{something}][\text{pattern}]$ but $[\text{something}] = [c]$! Oh no D: Well remember, for negative indices the condition is automatically true! So $s[i] = -3$. Let's check the concatenated patterns with overlapping suffix and prefix. At index $i = -3$ (again), we have the same result as before! Since both of them give us $i = -3$, we can say that $S[3] = -3$.

i	0	1	2	3	4	5	6	7
P[i]	b	o	n	o	b	o	b	o
S[i]				-3	2	-1	2	6

Nice! We're almost done. We have all the tools to finish it now. For $i = 2$, we are now looking for $[\text{pattern}] = \text{'obobo'}$ not prefixed with $[c] = \text{'n'}$. This occurs at $i = -4$ using both of our extended patterns. Thus, $S[i] = -4$.

i	0	1	2	3	4	5	6	7
P[i]	b	o	n	o	b	o	b	o
S[i]			-4	-3	2	-1	2	6

For $i = 1$, we're looking for the $[\text{pattern}] = \text{'nobobo'}$ not prefixed with $[c] = \text{'o'}$. For the extended pattern $[P][P]$ with NO overlap, this occurs at $i = -7$. Whereas for the extended pattern with overlap, this occurs at $i = -5$, so we choose $S[i] =$

-5. And finally, for $i = 0$, we're looking for the $[\text{pattern}] = \text{'onobobo'}$ not prefixed with $[c] = \text{'b'}$. This occurs at $i = -8$ for the extended pattern $[\text{pattern}][\text{pattern}]$ with no overlap and this occurs at $i = -6$ for the extended pattern with overlap, thus $S[i] = -6$.

i	0	1	2	3	4	5	6	7
P[i]	b	o	n	o	b	o	b	o
S[i]	-6	-5	-4	-3	2	-1	2	6

Honestly, I know that it is confusing. Let's try doing actual pattern matching with boyer-moore, and I'll have an extra example for the suffix array below this that I'll go over.

Pattern Matching with Boyer-Moore

Let's consider a new pattern $P[i] = \text{bonobo}$. We have $T[i] = \text{bonoaob-nobonobo}$ so our alphabet is a, b, o, n. I'll leave it as an exercise to build the last occurrence function and good suffix array.

The last occurrence function is:

c	a	b	n	o
L(c)	-1	4	2	5

The good suffix array is:

i	0	1	2	3	4	5
P[i]	b	o	n	o	b	o
S[i]	-4	-3	-2	-1	2	4

Now to do pattern matching with Boyer-Moore, we consider two indices. i for the text and j for the pattern. m will represent the length of the pattern. We start index i and j both at $m-1$. So $i = m-1$ and $j = m-1$. We do this because we're going to match backwards. So we compare $T[i]$ and $P[j]$. If they match, we decrement i and j by 1, and compare again. We do this until we fail. An example of this is seen below where the top row is the index i , and the second row is $T[i]$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
b	o	n	o	a	o	b	n	o	b	o	n	o	b	o
				h	o									

So we've matched 'o' successfully, but then we fail at 'b' since $T[i] = 'a' \neq P[i] = 'b'$. Now what do we do? We have to use our heuristics to determine where to shift our pattern in our text. We do this by setting $i = i + m - 1 - \min(L(T[i]), S[j])$. So we're shifting our pattern to start one pattern's length away from where we are, then we adjust it using the bad character array or good suffix array depending on which one makes the biggest jump. Note, we subtract it so the smaller the value of the functions, the larger the jump. In our case, $i = 4$ where we failed and $T[i] = 'a'$. So $L(T[i]) = L('a') = -1$ and $S[j] = S[4] = 2$. Since $L(T[i])$ is smaller, we choose it to jump. so we set $i = i + m - 1 - L(T[i])$. So $i = 4 + 6 - 1 - (-1) = 10$. We then set $j = m-1$ (to start matching from the back of the pattern) and we repeat the above.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
b	o	n	o	a	o	b	n	o	b	o	n	o	b	o
				y	o									
						ϕ	n	o	b	o				

So why did we shift there? What is our heuristic telling us? We shifted our pattern to $i = 10$, because we know that at $i = 4$ when we failed we saw the character 'a'. Since a is not in our pattern, then that index will never ever match. So we can shift our pattern past it. This time we failed at index $i = 6$. $T[i] = 'b'$ so $L(T[i]) = L('b') = 4$ and $S[j] = S[1] = -3$. Thus, $S[1]$ is the minimum so we can shift our pattern to $i = 6 + 6 - 1 - (-3) = 14$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
b	o	n	o	a	o	b	n	o	b	o	n	o	b	o
				y	o									
						ϕ	n	o	b	o				
									b	o	n	o	b	0

And we've matched! Huzzah! But again, why did we shift here? Normally, for the good suffix array, we are looking for a substring in my text that I can match to a substring in my pattern. But, when we hit negatives, we're looking at characters we might not have checked yet. We do however know that we've matched 'bo' because it is a suffix that is also a prefix. So, we can shift our pattern to match that 'bo'.

Extra Good Suffix Array Example

Let's try with the pattern $P = \text{ingstating}$. (NOTE: if you were in my review session then you know that I failed miserably at this and shall now correct my mistake :D)

i	0	1	2	3	4	5	6	7	8	9
P[i]	i	n	g	s	t	a	t	i	n	g
S[i]										

Once again, I'll explain this thoroughly to make sure you understand how this works. First let's look at $i = 9$. At this index, if we're looking for $S[9]$ then we know that we have currently matched nothing in the text and have failed at the character 'g'. So, now I have to find the largest index such that I match the empty character (which conveniently matches at every index) BUT does not have a 'g' in front of it. So essentially, you're looking for the largest index that is not the letter 'g'. This occurs at $i = 8$, so $S[9] = 8$.

i	0	1	2	3	4	5	6	7	8	9
P[i]	i	n	g	s	t	a	t	i	n	g
S[i]										8

Now for $i = 8$, what we're really doing is matching through boyer-moore. So imagine we have a text, and 'g' at $i = 9$ successfully matches. Then, we try 'n' but it fails miserably. What I want to do is find the next time I see the pattern 'g' in my pattern P, BUT the caveat is that n can't be in front of it. To understand why that is, remember that the Good Suffix Array is a way to shift the pattern to match a suffix that has already been matched. If I've failed at the character n, then shift my pattern to match 'ng' again then I'll just shift to a fail. That is a nono. So I see that 'g' appears at index 2, huzzah! But it has an 'n' in front of it. That doesn't work. This means that we'll have to go into our negative case. So, I will create two new patterns which are the pattern concatenated with itself. One, if you recall, will overlap the largest prefix that is a suffix. The other one will not.

i	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9
P[i]	i	n	g	s	t	a	t	i	n	g	i	n	g	s	t	a	t	i	n	g

i	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9
P[i]	i	n	g	s	t	a	t	i	n	g	s	t	a	t	i	n	g

Now we're going to look for the [pattern] = 'g' but without an 'n' in front of it. There doesn't seem to be one... Remember though, that if we have an index that is negative, we can automatically match it! So for the extended pattern without overlap, we get $i = -2$. Since there is a 'g' at $i = -1$, and [something] at $i = -2$. For the extended pattern with overlap, we get $i = -6$. Since $i = -2$ is the larger index, $S[8] = -2$.

i	0	1	2	3	4	5	6	7	8	9
P[i]	i	n	g	s	t	a	t	i	n	g
S[i]									-2	8

For index $i = 7$, we have a similar situation as above. Where can I find the $[\text{pattern}] = \text{'ng'}$ but without an 'i' in front. Well, that doesn't occur anywhere so we have to use the extended patterns. For the extended pattern without overlap, we find $[\text{pattern}] = \text{'ng'}$ at index $i = -3$. For the extended pattern with overlap, we find it at $i = -7$. Once again, since $i = -3$ is larger, we choose $S[7] = -3$.

i	0	1	2	3	4	5	6	7	8	9
P[i]	i	n	g	s	t	a	t	i	n	g
S[i]								-3	-2	8

Woot! We're almost there. For $i = 6$, we are looking for $[\text{pattern}] = \text{'ing'}$ without 't' in front of it. Once again, we can't find it in the actual pattern so we use the extended patterns. For the extended pattern with no overlap we find $[\text{pattern}] = \text{'ing'}$ at index $i = -4$. For the pattern with overlap, we find $[\text{pattern}] = \text{'ing'}$ at index $i = -1$. Thus, we choose $S[6] = -1$.

i	0	1	2	3	4	5	6	7	8	9
P[i]	i	n	g	s	t	a	t	i	n	g
S[i]							-1	-3	-2	8

So you see that in some cases with the good suffix array, we'll have to jump back and forth between the two extended patterns.

Let's keep going. For $i = 5$, we have matched $[\text{pattern}] = \text{'ting'}$ and we failed at an 'a'. So we're looking for 'ting' in our pattern, without an 'a' in front of it. Once again, it is not in our original pattern so we look at the extended patterns. For the extended pattern with no overlap, we see 'ting' at $i = -5$. And for the extended pattern with overlap, we see 'ting' at $i = -2$. Thus we choose $s[5] = -2$.

i	0	1	2	3	4	5	6	7	8	9
P[i]	i	n	g	s	t	a	t	i	n	g
S[i]						-2	-1	-3	-2	8

Continuing on with the process, for $i = 4$ we have matched $[\text{pattern}] = \text{'ating'}$ and failed at 't'. We look at the extended patterns which give $i = -6$ for the one without overlap and $i = -3$ for the one with overlap. Thus $S[4] = -3$.

i	0	1	2	3	4	5	6	7	8	9
P[i]	i	n	g	s	t	a	t	i	n	g
S[i]					-3	-2	-1	-3	-2	8

Finishing it off, for $i = 3$, we have matched $[\text{pattern}] = \text{'tating'}$ and failed at 's'. So we look for the pattern 'tating' without an 's' in front of it. We look at the extended patterns which give $i = -7$ for the one without overlap and $i = -4$ for the one with overlap. Thus we choose $s[3] = -4$.

For $i = 2$, we have matched $[\text{pattern}] = \text{'stating'}$ failing at 'g'. The extended patterns give $i = -8$ and $i = -5$ respectively (in the same order as the above paragraph) , so $S[2] = -5$.

For $i = 1$, we have matched $[\text{pattern}] = \text{'gstating'}$ failing at 'n'. The extended patterns give $i = -9$, and $i = -6$ so $S[1] = -6$.

Finally, for $i = 0$, we have matched $[\text{pattern}] = \text{'ngstating'}$ failing at 'i'. The extended patterns give $i = -10$, and $i = -7$ so $S[0] = -7$. This gives us the final result for the suffix array as:

i	0	1	2	3	4	5	6	7	8	9
P[i]	i	n	g	s	t	a	t	i	n	g
S[i]	-7	-6	-5	-4	-3	-2	-1	-3	-2	8

Hopefully this document has helped you somewhat in understanding how boyer-moore works!

Alex