12.1 Computing the Failure Function

The last set of notes left off on how to create a pattern matching machine $M_p$ with backtracking arrows that represent the failure function. Now we will show how to construct the failure function explicitly for any given string $p$.

The failure function $f$ can be computed iteratively in much the same manner in which $M_p$ operates. By definition, $f(1) = 0$. Suppose that at some point we have so far computed $f(1), ..., f(j)$. Let us set $f(j) = i$. To compute $f(j + 1)$ we look at $p_{j+1}$ and $p_{i+1}$. From here:

- If $p_{j+1} = p_{i+1}$ then $f(j + 1) = f(j) + 1$. This is because we have now found that $p_1p_2...p_ip_{i+1} = p_j...p_{i+1}p_{i+2}...p_{j+1}$.
- If $p_{j+1} \neq p_{i+1}$, we then find the smallest $m$ for which either:
  - Case 1: $f^m(j) = u$ and $p_{j+1} = p_{u+1}$. Here we set $f(j + 1) = u + 1$
  - Case 2: $f^m(j) = 0$ and $p_{j+1} \neq p_1$. Here we set $f(j + 1) = 0$.

12.2 The Failure Function in Code

We will now take the logic above and present it in the form of code. Note the sublime beauty and elegance of this amazing procedure. Recall that it allows us to perform linear-time string search - truly a gift!!
Algorithm 1 Failure Function

Input: $p = p_1p_2\ldots p_L$
Output: $f$, the failure function

procedure FAILUREFUNCTION($p$)
  $f(1) = 0$
  $i = 0$
  for $j ← 2, 3, \ldots, L$ do
    $i = f(j - 1)$
    while $p_j \neq p_{i+1}$ and $i > 0$ do
      $i = f(i)$;
    end while
    if $p_j \neq p_{i+1}$ and $i = 0$ then
      $f(j) = 0$
    else
      $f(j) = i + 1$
    end if
  end for
return $f$
end procedure

12.3 Correctness and Runtime

Theorem 12.1. The Failure Function algorithm above computes $f$ correctly.

Proof. We shall prove by induction on $j$ that $f(j)$ is the longest integer $i$ less than $j$ such that $p_1p_2\ldots p_i = p_{j-i+1}p_{j-i+2}\ldots p_j$. If no such $i$ exists, $f(j) = 0$. By definition $f(1) = 0$. Suppose the inductive hypothesis is true for all $f(k)$ such that $k < j$. In computing $f(j)$ the algorithm compares $p_j$ with $p_{f(j-1)+1}$ in the conditional statement of the while loop. Now we have two cases:

Case 1: Suppose $p_j = p_{f(j-1)+1}$ since $f(j - 1)$. is the largest $i$ such that $p_1\ldots p_j = p_{j-i}\ldots p_{j-1}$ it follows that $f(j) = i + 1$ is correct. Thus by the if statement in the algorithm presented, $f(j)$ is computed correctly.

Case 2: Suppose $p_j \neq p_{f(j-1)+1}$. Then we must find the largest value of $i$ such that:

- $p_1\ldots p_i = p_{j-i}\ldots p_{j-1}$ AND
- $p_{i+1} = p_j$ if such $i$ exists

If no such $i$ exists then clearly $f(j) = 0$ and $f(j)$ is correctly computed by the time the if statement is reached.

Now let $i_1, i_2, \ldots$ be the largest, second largest, etc. values of $i$ such that $p_1p_2\ldots p_i = p_{j-i}\ldots p_{j-1}$. By simple deduction we see that:

$i_1 = f(j - 1)$
$i_2 = f(i_1) = f^2(j - 1)$
\ldots
$i_s = f(i_{s-1}) = f^s(j - 1)$

Thus by the time the while statement is reached, the algorithm considers $i_1, i_2, \ldots$ in turn until a value of $i$ exists. On termination of the execution of the while statement, $i = i_m$ is such that $i_m$ exists and thus $f(j)$ correctly computes $f(j)$ before the if statement is reached. Thus $f(j)$ is correctly computed for all $j$. 
Theorem 12.2. The Failure Function Algorithm computes $f$ in $O(L)$ steps, where $L = |p|$.

Proof. First, note that the first two lines of the algorithm incur constant cost in the size of the input. From there, the procedure enters a for loop, which iterates a linear number of times in the size of the input. Inside this for loop all of the steps are of constant cost except for the internal while loop. The cost of the while statement is proportional to the number of times $i$ is decreased by the statement $i = f(i)$ following the do (remember that by definition $f(i) < i$, so this step always involves decrementing $i$). The only way $i$ is incremented is by assigning $f(j) = i + 1$ in the else clause, and then setting $i = f(j - 1)$ in the next iteration of the for loop. Since $i = 0$ initially and the else case is executed no more than $L - 1$ times, we conclude that the while statement cannot be executed more than $L$ times. Thus the total cost of executing the while too is linear ($O(L)$ time). The remainder of the algorithm cost is clearly $O(L)$, and thus the whole algorithm takes linear time.

We can show by an argument like in Theorem 12.1 that the pattern-matching machine $M_p$ will be in state $i$ after reading $t_1 t_2 \ldots t_k$ if and only if $p_1 p_2 \ldots p_i$ is the longest prefix of $p$ that is a suffix of $t_1 t_2 \ldots t_k$. Thus $M_p$ correctly finds the leftmost occurrence of $p$ in the text $t = t_1 \ldots t_n$. By the argument of Theorem 12.2 we can show that $M_p$ will execute at most $2 \cdot |t|$ state transitions in processing input text $t$. Thus we can determine whether $p$ is a substring of $t$ by tracing out the state transitions of $M_p$ on input $t$. To do this all we need is the Failure Function $f$ for $p$. This can be constructed in $O(|p|)$ time. Thus we can can determine whether $p$ is a substring of $t$ in $O(|p| + |t|)$ time independent of the alphabet size.