D More String Matching

D.1 Redundant Comparisons

Let's go back to the character-by-character method for string matching. Suppose we are looking for the pattern 'ABRACADABRA' in some longer text using the (almost) brute force algorithm described in the previous lecture. Suppose also that when $s = 11$, the substring comparison fails at the fifth position; the corresponding character in the text (just after the vertical line below) is not a C. At this point, our algorithm would increment s and start the substring comparison from scratch.

HOCUSPOCUSABRABRACADABRA...
ABRA~~CADABRA~~
ABRACADABRA

If we look carefully at the text and the pattern, however, we should notice right away that there's no point in looking at $s = 12$. We already know that the next character is a B — after all, it matched $P[2]$ during the previous comparison — so why bother even looking there? Likewise, we already know that the next two shifts $s = 13$ and $s = 14$ will also fail, so why bother looking there?

HOCUSPOCUSABRABRACADABRA...
ABRA~~CADABRA~~
~~ABRACADABRA~~
~~ABRACADABRA~~
ABRACADABRA

Finally, when we get to $s = 15$, we can't immediately rule out a match based on earlier comparisons. However, for precisely the same reason, we shouldn't start the substring comparison over from scratch — we already know that $T[15] = P[4] = \texttt{A}$. Instead, we should start the substring comparison at the *second* character of the pattern, since we don't yet know whether or not it matches the corresponding text character.

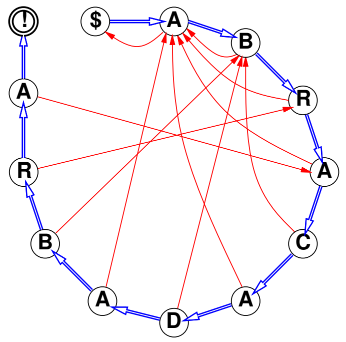
If you play with this idea long enough, you'll notice that the character comparisons should always advance through the text. **Once we've found a match for a text character, we never need to do another comparison with that character again.** In other words, we should be able to optimize the brute-force algorithm so that it always *advances* through the text.

You'll also eventually notice a good rule for finding the next 'reasonable' shift s. A *prefix* of a string is a substring that includes the first character; a *suffix* is a substring that includes the last character. A prefix or suffix is *proper* if it is not the entire string. Suppose we have just discovered that $T[i] \neq P[j]$. **The next reasonable shift is the smallest value of s such that $T[s\,..\,i-1]$, which is a suffix of the previously-read text, is also a proper prefix of the pattern.**

In this lecture, we'll describe a string matching algorithm, published by Donald Knuth, James Morris, and Vaughn Pratt in 1977, that implements both of these ideas.

D.2 Finite State Machines

If we have a string matching algorithm that follows our first observation (that we always advance through the text), we can interpret it as feeding the text through a special type of *finite-state machine*. A finite state machine is a directed graph. Each node in the graph, or *state*, is labeled with a character from the pattern, except for two special nodes labeled $\$$ and $!$. Each node has two outgoing edges, a *success* edge and a *failure* edge. The success edges define a path through the characters of the pattern in order, starting at $ and ending at !. Failure edges always point to earlier characters in the pattern.



A finite state machine for the string 'ABRADACABRA'.
Thick arrows are the success edges; thin arrows are the failure edges.

We use the finite state machine to search for the pattern as follows. At all times, we have a current text character $T[i]$ and a current node in the graph, which is usually labeled by some pattern character $P[j]$. We iterate the following rules:

- If $T[i] = P[j]$, or if the current label is $, follow the success edge to the next node and increment i. (So there is no failure edge from the start node $.)
- If $T[i] \neq P[j]$, follow the failure edge back to an earlier node, but do not change i.

For the moment, let's simply assume that the failure edges are defined correctly—we'll come back to this later. If we ever reach the node labeled !, then we've found an instance of the pattern in the text, and if we run out of text characters ($i > n$) before we reach !, then there is no match.

The finite state machine is really just a (very!) convenient metaphor. In a real implementation, we would not construct the entire graph. Since the success edges always go through the pattern characters in order, we only have to remember where the failure edges go. We can encode this *failure function* in an array $fail[1..n]$, so that for each j there is a failure edge from node j to node $fail[j]$. Following a failure edge back to an earlier state exactly corresponds, in our earlier formulation, to shifting the pattern forward. The failure function $fail[j]$ tells us how far to shift after a character mismatch $T[i] \neq P[j]$.

Here's what the actual algorithm looks like:

```
KnuthMorrisPratt(T[1..n], P[1..m]):
  j ← 1
  for i ← 1 to n
      while j > 0 and T[i] ≠ P[j]
          j ← fail[j]
      if j = m        ⟨⟨Found it!⟩⟩
          return i − m + 1
      j ← j + 1
  return 'none'
```

Before we discuss computing the failure function, let's analyze the running time of Knuth-MorrisPratt under the assumption that a correct failure function is already known. At each character comparison, either we increase i and j by one, or we decrease j and leave i alone.

We can increment i at most $n-1$ times before we run out of text, so there are at most $n-1$ successful comparisons. Similarly, there can be at most $n-1$ failed comparisons, since the number of times we decrease j cannot exceed the number of times we increment j. In other words, we can amortize character mismatches against earlier character matches. Thus, the total number of character comparisons performed by KNUTHMORRISPRATT in the worst case is $O(n)$.

D.3 Computing the Failure Function

We can now rephrase our second intuitive rule about how to choose a reasonable shift after a character mismatch $T[i] \neq P[j]$:

$$\boxed{P[1\,..\,fail[j]-1] \text{ is the longest proper prefix of } P[1\,..\,j-1] \text{ that is also a suffix of } T[1\,..\,i-1].}$$

Notice, however, that if we are comparing $T[i]$ against $P[j]$, then we must have already matched the first $j-1$ characters of the pattern. In other words, we already know that $P[1\,..\,j-1]$ is a suffix of $T[1\,..\,i-1]$. Thus, we can rephrase the prefix-suffix rule as follows:

$$\boxed{P[1\,..\,fail[j]-1] \text{ is the longest proper prefix of } P[1\,..\,j-1] \text{ that is also a suffix of } P[1\,..\,j-1].}$$

This is the definition of the Knuth-Morris-Pratt failure function $fail[j]$ for all $j > 1$.[1] By convention we set $fail[1] = 0$; this tells the KMP algorithm that if the first pattern character doesn't match, it should just give up and try the next text character.

$P[i]$	A	B	R	A	C	A	D	A	B	R	A
$fail[i]$	0	1	1	1	2	1	2	1	2	3	4

Failure function for the string 'ABRACADABRA'
(Compare with the finite state machine on the previous page.)

We could easily compute the failure function in $O(m^3)$ time by checking, for each j, whether every prefix of $P[1\,..\,j-1]$ is also a suffix of $P[1\,..\,j-1]$, but this is not the fastest method. The following algorithm essentially uses the KMP search algorithm to look for the pattern inside itself!

```
COMPUTEFAILURE(P[1..m]):
  j ← 0
  for i ← 1 to m
      fail[i] ← j        (∗)
      while j > 0 and P[i] ≠ P[j]
          j ← fail[j]
      j ← j + 1
```

Here's an example of this algorithm in action. In each line, the current values of i and j are indicated by superscripts; \$ represents the beginning of the string. (You should imagine pointing at $P[j]$ with your left hand and pointing at $P[i]$ with your right hand, and moving your fingers according to the algorithm's directions.)

[1]CLR defines a similar *prefix function*, denoted $\pi[j]$, as follows:

$$\boxed{P[1\,..\,\pi[j]] \text{ is the longest proper prefix of } P[1\,..\,j] \text{ that is also a suffix of } P[1\,..\,j].}$$

These two functions are not the same, but they are related by the simple equation $\pi[j] = fail[j+1] - 1$. The off-by-one difference between the two functions adds a few extra +1s to CLR's version of the algorithm.

$j \leftarrow 0$, $i \leftarrow 1$	$\$^j$	A^i	B	R	A	C	A	D	A	B	R	X ...
$fail[i] \leftarrow j$		**0**										...
$j \leftarrow j+1$, $i \leftarrow i+1$	\$	A^j	B^i	R	A	C	A	D	A	B	R	X ...
$fail[i] \leftarrow j$		0	**1**									...
$j \leftarrow fail[j]$	$\$^j$	A	B^i	R	A	C	A	D	A	B	R	X ...
$j \leftarrow j+1$, $i \leftarrow i+1$	\$	A^j	B	R^i	A	C	A	D	A	B	R	X ...
$fail[i] \leftarrow j$		0	1	**1**								...
$j \leftarrow fail[j]$	$\$^j$	A	B	R^i	A	C	A	D	A	B	R	X ...
$j \leftarrow j+1$, $i \leftarrow i+1$	\$	A^j	B	R	A^i	C	A	D	A	B	R	X ...
$fail[i] \leftarrow j$		0	1	1	**1**							...
$j \leftarrow j+1$, $i \leftarrow i+1$	\$	A	B^j	R	A	C^i	A	D	A	B	R	X ...
$fail[i] \leftarrow j$		0	1	1	1	**2**						...
$j \leftarrow fail[j]$	\$	A^j	B	R	A	C^i	A	D	A	B	R	X ...
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C^i	A	D	A	B	R	X ...
$j \leftarrow j+1$, $i \leftarrow i+1$	\$	A^j	B	R	A	C	A^i	D	A	B	R	X ...
$fail[i] \leftarrow j$		0	1	1	1	2	**1**					...
$j \leftarrow j+1$, $i \leftarrow i+1$	\$	A	B^j	R	A	C	A	D^i	A	B	R	X ...
$fail[i] \leftarrow j$		0	1	1	1	2	1	**2**				...
$j \leftarrow fail[j]$	\$	A^j	B	R	A	C	A	D^i	A	B	R	X ...
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C	A	D^i	A	B	R	X ...
$j \leftarrow j+1$, $i \leftarrow i+1$	\$	A^j	B	R	A	C	A	D	A^i	B	R	X ...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	**1**			...
$j \leftarrow j+1$, $i \leftarrow i+1$	\$	A	B^j	R	A	C	A	D	A	B^i	R	X ...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	**2**		...
$j \leftarrow j+1$, $i \leftarrow i+1$	\$	A	B	R^j	A	C	A	D	A	B	R^i	X ...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	**3**	...
$j \leftarrow j+1$, $i \leftarrow i+1$	\$	A	B	R	A^j	C	A	D	A	B	R	X^i...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	3	**4** ...
$j \leftarrow fail[j]$	\$	A^j	B	R	A	C	A	D	A	B	R	X^i...
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C	A	D	A	B	R	X^i...

ComputeFailure in action. Do this yourself by hand.

Just as we did for KnuthMorrisPratt, we can analyze ComputeFailure by amortizing character mismatches against earlier character matches. Since there are at most m character matches, ComputeFailure runs in $O(m)$ time.

Let's prove (by induction, of course) that ComputeFailure correctly computes the failure function. The base case $fail[1] = 0$ is obvious. Assuming inductively that we correctly computed $fail[1]$ through $fail[i]$ in line $(*)$, we need to show that $fail[i+1]$ is also correct. Just after the ith iteration of line $(*)$, we have $j = fail[i]$, so $P[1 \mathinner{..} j-1]$ is the longest proper prefix of $P[1 \mathinner{..} i-1]$ that is also a suffix.

Let's define the iterated failure functions $fail^c[j]$ inductively as follows: $fail^0[j] = j$, and

$$fail^c[j] = fail[fail^{c-1}[j]] = \overbrace{fail[fail[\cdots[fail}^{c}[j]]\cdots]].$$

In particular, if $fail^{c-1}[j] = 0$, then $fail^c[j]$ is undefined. We can easily show by induction (see [CLR, p.872]) that every string of the form $P[1 \mathinner{..} fail^c[j]-1]$ is both a proper prefix and a proper suffix of $P[1 \mathinner{..} i-1]$, and in fact, these are the only examples. Thus, the longest proper prefix/suffix of $P[1 \mathinner{..} i]$ must be the longest string of the form $P[1 \mathinner{..} fail^c[j]]$ — *i.e.*, the one with smallest c — such that $P[fail^c[j]] = P[i]$. This is exactly what the while loop in ComputeFailure computes; the $(c+1)$th iteration compares $P[fail^c[j]] = P[fail^{c+1}[i]]$ against $P[i]$. ComputeFailure is actually a *dynamic programming* implementation of the following recursive definition of $fail[i]$:

$$fail[i] = \begin{cases} 0 & \text{if } i = 0, \\ \max\limits_{c \geq 1} \left\{ fail^c[i-1]+1 \;\middle|\; P[i-1] = P[fail^c[i-1]] \right\} & \text{otherwise.} \end{cases}$$

D.4 Optimizing the Failure Function

We can speed up KnuthMorrisPratt slightly by making one small change to the failure function. Recall that after comparing $T[i]$ against $P[j]$ and finding a mismatch, the algorithm compares $T[i]$ against $P[fail[j]]$. With the current definition, however, it is possible that $P[j]$ and $P[fail[j]]$ are actually the same character, in which case the next character comparison will automatically fail. So why do the comparison at all?

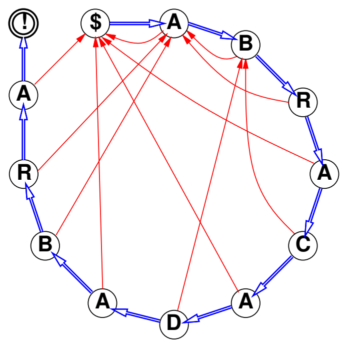
We can optimize the failure function by 'short-circuiting' these redundant comparisons with some simple post-processing:

```
OptimizeFailure(P[1..m], fail[1..m]):
  for i ← 2 to m
      if P[i] = P[fail[i]]
          fail[i] ← fail[fail[i]]
```

We can also compute the optimized failure function directly by adding three new lines (in bold) to the ComputeFailure function.

```
ComputeOptFailure(P[1..m]):
  j ← 0
  for i ← 1 to m
      if P[i] = P[j]
          fail[i] ← fail[j]
      else
          fail[i] ← j
      while j > 0 and P[i] ≠ P[j]
          j ← fail[j]
      j ← j + 1
```

This optimization slows down the preprocessing slightly, but it may significantly decrease the number of comparisons at each text character. The worst-case running time is still $O(n)$; however, the constant is about half as big as for the unoptimized version, so this could be a significant improvement in practice.



Optimized finite state machine for the string 'ABRADACABRA'

$P[i]$	A	B	R	A	C	A	D	A	B	R	A
$fail[i]$	0	1	1	**0**	2	**0**	2	**0**	**1**	**1**	**0**

Optimized failure function for 'ABRACADABRA', with changes in bold.

Here are the unoptimized and optimized failure functions for a few more patterns:

$P[i]$	A	N	A	N	A	B	A	N	A	N	A	N	A
unoptimized $fail[i]$	0	1	1	2	3	4	1	2	3	4	5	6	5
optimized $fail[i]$	0	1	0	1	0	4	0	1	0	1	0	6	0

Failure functions for 'ANANABANANANA'.

$P[i]$	A	B	A	B	C	A	B	A	B	C	A	B	C
unoptimized $fail[i]$	0	1	1	2	3	1	2	3	4	5	6	7	8
optimized $fail[i]$	0	1	0	1	3	0	1	0	1	3	0	1	8

Failure functions for 'ABABCABABCABC'.

$P[i]$	A	B	B	A	B	B	A	B	A	B	B	A	B
unoptimized $fail[i]$	0	1	1	1	2	3	4	5	6	2	3	4	5
optimized $fail[i]$	0	1	1	0	1	1	0	1	6	1	1	0	1

Failure functions for 'ABBABBABABBAB'.

$P[i]$	A	A	A	A	A	A	A	A	A	A	A	A	B
unoptimized $fail[i]$	0	1	2	3	4	5	6	7	8	9	10	11	12
optimized $fail[i]$	0	0	0	0	0	0	0	0	0	0	0	0	12

Failure functions for 'AAAAAAAAAAAAB'.