CSE 584A Class 21

Jeremy Buhler

April 11, 2018

1 A Bit-Twiddling Approach to Dynamic Programming

- Given pattern P and text T
- Find all occurrences of P in T with at most k differences
- (differences can be substitution or indel)
- Suppose P is short (say, at most 128 characters)
- An algorithm called `bitap` solves this problem. See Wu and Manber, *CACM* 35(10):83-91, 1992.
- part of a tool called **agrep** ("approximate grep")

Today, we'll look at basics of how it works.

2 Basic Algorithm

bitap is a good example of *bit-parallel* approximate matching algorithms. (Alternative methods exist by, e.g., Baeza-Yates and Navarro, and by Gene Myers).

- Let P and T be pattern and text
- **Defn**: for $0 \leq i \leq |P|$,

$$R_j[i] = \begin{cases} 1 & \text{if } P[1..i] = T[j-i+1..j] \\ 0 & \text{otherwise} \end{cases}$$

- In words, $R_j[i]$ is 1 iff ith prefix of pattern matches a suffix of $T[1..j]$.
- (By convention, $R_j[0] = 1$, since empty suffix always matches.)
- **Example**:

- If $R_j[|P|] = 1$, then P occurs in T starting at position $j - |P| + 1$.

So far, so good. Let's give rule for incrementally computing R.

- **Lemma**: let $i > 0$. Then $R_j[i] = 1$ iff both $R_{j-1}[i-1] = 1$ and $P[i] = T[j]$.
- **Proof**: follows immediately from definition of R.
- $R_j[i] = 1$ iff $P[1..i] = T[j-i+1..j]$.
- **picture**:

- But $P[1..i] = T[j-i+1..j]$ iff $P[i] = T[j]$ and $P[1..i-1] = T[j-i+1..j-1]$.
- Latter condition is exactly defn of $R_{j-1}[i-1] = 1$. QED
- Let $\delta(a, b) = 1$ if $a = b$, or 0 otherwise.
- Then we can write, in reasonably pure arithmetic,

$$R_j[i] = R_{j-1}[i-1] \otimes \delta(P[i], T[j]).$$

- (Remember that in base case, $R_{j-1}[0] = 1$.)

A simple $\Theta(|P||T|)$ algorithm for exact matching would be as follows:

MATCH(P, T)
  $R_0[0] \leftarrow 1$
  for $1 \leq i \leq |P|$ **do**
    $R_0[i] \leftarrow 0$
  for $j = 1..|T| - |P| + 1$ **do**
    $R_j[0] \leftarrow 1$
    for $i = 1..|P|$ **do**
      $R_j[i] \leftarrow R_{j-1}[i-1] \otimes \delta(P[i], T[j])$
    if $R_j[|P|] = 1$
      report match at $T[j - |P| + 1]$

3 Where Are the Bits?

Why do we call the simple matching algorithm "bit-parallel"?

- Each pass through inner loop over i sets one bit of $R[j]$.
- Note that inner loop can be executed in parallel for $1 \leq i \leq |P|$.

- We will treat $R_j[1..|P|]$ as a $|P|$-bit word and set all its bits in parallel!
- Define S_a to be an $|P|$-bit vector such that $S_a[i] = \delta(P[i], a)$, $1 \le i \le |P|$.
- (Can precompute S_a for P and each character $a \in \Sigma$.)
- We can rewrite the inner loop computation as follows:

$$\begin{aligned} R_j[i] &= R_{j-1}[i-1] \otimes \delta(P[i], T[j]) \\ &= R_{j-1}[i-1] \otimes S_{T[j]}[i] \\ &= ((R_{j-1} \ll 1) \oplus 1)[i] \otimes S_{T[j]}[i] \end{aligned}$$

- (the OR with 1 is needed because $\ll$ shifts in a zero)
- Removing the i's, we have reduced the inner loop to three bit ops!

$$R_j = ((R_{j-1} \ll 1) \oplus 1) \otimes S_{T[j]}.$$

- (We need not represent $R_j[0]$ explicitly!)
- **Example**:

If $|P|$ is small enough that we can perform a bit op on R_j in constant time, then we can find all copies of P in T in time $\Theta(|T|)$.

- A 32-bit machine can handle $|P| = 32$
- A 64-bit machine can handle $|P| = 64$
- An x86 with AVX can handle $|P| = 128$ (or 256 or 512, depending on generation).
- (Needs some fiddling to do scalar shift by arbitrary # of bits, vs whole # of bytes, using SSE or AVX)
- *Note*: need a small alphabet so that all S_a values can be cached simultaneously! Isn't DNA great?

4 Handling Wildcards

So far, we have a spiffy bit-parallel exact matching algorithm.

- *First extension*: classes and wildcards
- Each position of pattern may be arbitrary *subset* of alphabet.
- *Example*: ARGNCGWT (R is A or G, W is A or T, N is wildcard)
- How can we extend basic algo to handle classes?
- Extend matching function: $\delta(C, a)$ is one if character a is member of class C, zero otherwise
- Using this extended definition, basic recurrence does the right thing:

$$R_j[i] = R_{j-1}[i-1] \otimes \delta(P[i], T[j]).$$

- Can build vectors S_a as before, using extended version of δ:

$$S_a[i] = \delta(P[i], a).$$

- bit-parallel matching algorithm is unchanged!
- (note: this extension is especially nice for matching regulatory motifs in DNA)

Why don't wildcards cause us trouble? We're not using transitivity at all, just bog-stupid $\Theta(|P||T|)$ computation plus parallelism.

5 Handling Errors

We will show how to find matches with at most *one* error (substitution or indel), then extend to at most k errors.

- For consistency later, define $R_j^0 \equiv R_j$.
- Define $R_j^1[i]$ to be 1 if $P[1..i]$ matches $T[...j]$ with at most one error, 0 otherwise.
- How many ways can $R_j^1[i] = 1$? Four cases:
  - *Case 1*: $P[1..i-1]$ matches $T[...j-1]$ with at most one error, and $P[i] = T[j]$.
  - This case requires that $R_{j-1}^1[i-1] \otimes \delta(P[i], T[j]) = 1$.
  - *Case 2*: $P[1..i-1]$ matches $T[...j-1]$ perfectly, but $P[i] \neq T[j]$.
  - This case requires only that $R_{j-1}^0[i-1] = 1$.
  - *Case 3*: $P[1..i-1]$ matches $T[...j]$ perfectly, and $P[i]$ is unaligned.
  - This case requires only that $R_j^0[i-1] = 1$.
  - *Case 4*: $P[1..i]$ matches $T[...j-1]$ perfectly, and $T[j]$ is unaligned.
  - This case requires only that $R_{j-1}^0[i] = 1$.

- As a mathematical expression, we have

$$R_j^1[i] = (R_{j-1}^1[i-1] \otimes \delta(P[i], T[j])) \oplus R_{j-1}^0[i-1] \oplus R_j^0[i-1] \oplus R_{j-1}^0[i].$$

- Can generalize this recurrence to compute $R_j^k[i]$, which is 1 iff $P[1..i]$ matches $T[1..j]$ with at most k errors.
- (Replace 1 by k, 0 by $k-1$.)
- Initial conditions? For $0 \leq i \leq k$, $R_0^k[i] = 1$, since we can leave all of $P[1..k]$ unaligned and still have a "match" to empty string with only k errors.

OK, now how about that bit parallelism?

- It's only slightly horrible to write down.
- As before, we compute S_a's with $S_a[i] = \delta(P[i], a)$
- We compute

$$\begin{aligned} R_j^k &= \Big[((R_{j-1}^k \ll 1) \oplus 1) \otimes S_{T[j]}\Big] \oplus \\ & ((R_{j-1}^{k-1} \ll 1) \oplus 1) \oplus \\ & ((R_j^{k-1} \ll 1) \oplus 1) \oplus \\ & R_{j-1}^{k-1} \end{aligned}$$

- Can simplify slightly by merging middle two terms:

$$\begin{aligned} R_j^k &= \Big[((R_{j-1}^k \ll 1) \oplus 1) \otimes S_{T[j]}\Big] \oplus \\ & (((R_{j-1}^{k-1} \oplus R_j^{k-1}) \ll 1) \oplus 1) \oplus \\ & R_{j-1}^{k-1} \end{aligned}$$

- We need to perform $k+1$ of the above operations per character of T, one for each # of errors between 0 and k.
- We check $R_j^k[|P|]$ to find out if there is a match at the current position.
- **Example** ($d = 1$):

One more fancy trick – allowing variable-length wildcards

- *Idea*: allow pattern to contain "#" character that can match *any number* of arbitrary characters.
- (Basically, a variable-length wildcard)
- Suppose we insert a "#" after position i of the pattern.
- If $R^k_{j'}[i] = 1$, then we have already matched $P[1..i]$ to $T[...j']$ with at most k errors.
- For every $j > j'$, we *also* have $R^k_j[i] = 1$, since all characters in between j' and j could be skipped.
- Hence, we have that, *only for bit i*,

$$R^k_j[i] = R^k_{j-1}[i] \oplus \dots$$

- How to implement with bit parallelism?
- Create bitmask $S^{\#}$ with a 1 at position i, 0's elsewhere
- For *every* i, we can now say that

$$R^k_j[i] = R^k_{j-1}[i] \otimes S^{\#}[i] \oplus \dots$$

- Same idea works for #'s after positions $i_1, i_2, \dots i_q$ – set corresponding bits of $S^{\#}$.

6 Even Fancier Stuff

I'm not going to talk about all the clever things agrep can do.

- Note, however, that "grep" stands for *general regular expression search*
- You can in fact give agrep an arbitrary regular expression E as its pattern.
- It will find all matches to E in the text with at most k errors!
- (Method described in paper)