CSE 373: Algorithmic Techniques

Pete Morcos
University of Washington
5/22/00

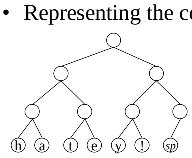
http://www.cs.washington.edu/education/courses/cse373/00sp

Character Encoding

- Consider saving the string "ha ha tee hee hey!" into a file
- How much space do we need?
- Well, there are only 7 characters, so we could just use 3 bits per character
  - h=000, a=001, t=010, e=011, y=100, !=101, spc=110
- 18 chars * 3 bits = 54 bits

Non-uniform coding

- We can improve our result by noticing that some characters are more frequent than others
  - Use shorter codes for those
- Representing the code as a tree helps make it clear

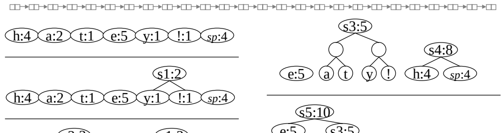


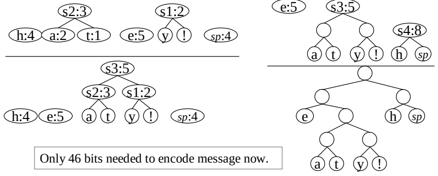
symbol	frequency	code
h	4	000
a	2	001
t	1	010
e	5	011
y	1	100
!	1	101
space	4	110

Huffman Coding

- Huffman coding is another example of a greedy algorithm
- Given symbol frequencies, it constructs an encoding tree
  - Start with N independent symbols
  - Take the two least frequent symbols
    - Merge them under a new pseudo-symbol parent
    - Parent frequency is sum of two children
  - Repeat until single tree is formed

Huffman example





Only 46 bits needed to encode message now.

symbol	code
h	10
a	0100
t	0101
e	00
y	0110
!	0111
space	11

Huffman Properties

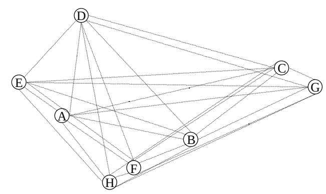
- The Huffman algorithm generates the optimal encoding tree of this type
- An important property is that no symbol's code is a prefix of another symbol's code
  - If this happened, decoding would be ambiguous
- Note that when saving the data, we must prepend the encoding somehow
  - For small messages, this can actually expand the file!
- This coding assumes all characters occur independently
  - A more sophisticated scheme might notice, for example, that 'q' is always followed by 'u', and not use any bits for the 'u'

Closest Points Problem

- Given N points in a plane, find the pair closest to each other
- Belongs to a class of problems known as *computational geometry*
- Naïve algorithm: compute distance between every possible pair
  - Obviously requires $O(N^2)$ time
  - Also only uses O(1) space
- We can improve this to O(N log N) time

Example Problem (N=8), naive solution

- There are 8(8-1)/2 = 28 distances to compute:
  - AB, AC, AD, AE, AF, AG, AH
  - BC, BD, BE, BF, BG, BH
  - CD, CE, CF, CG, CH
  - DE, DF, DG, DH
  - EF, EG, EH
  - FG, FH
  - GH



Divide and Conquer

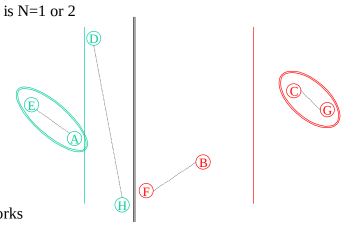
- Divide and Conquer involves breaking a problem into easier subproblems, and merging those solutions into the full answer
  - We've seen quicksort and mergesort, for example
- In this case, the insight is that if we divide the plane into two halves, the closest pair must be either:
  - Both in left half—find them using a recursive call
  - Both in right half—find them using a recursive call
  - One in left, one in right—find them some other way

Divide

- To speed things up, we will presort the set of points by their x coordinate
  - This costs O(N log N), so our final solution will be at least that expensive asymptotically
- Now it takes O(1) time to divide the set in half
  - Just pick the array index halfway between the bounds
- Our recurrence now looks like this:
  - $T(N) = 2\,T(N/2) + f(N)$
  - To get O(N log N), f(N) must be O(N); that's how much work we're allowed to do to merge the answers

Divide Example

- Obviously, base case is N=1 or 2
- Sorted by x, we get EADHFBCG
- Subcalls are:
  - EADH
    - EA
    - DH
  - FBCG
    - FB
    - CG
- Assume recursion works correctly; final result will return:
  - $D_L = 10$ (EA) and $D_R = 8$ (CG)



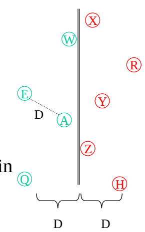
Conquer

- The recursions will find the closest pairs in the left or right halves (N/2 points in each half)
  - Call those distances D_L and D_R; let $D=\min(D_L,D_R)$
- We must test whether there is a pair of points that cross the boundary and are closer than D
- There are (N/2) * (N/2 – 1) / 2 possible crossing pairs
  - That's $O(N^2)$ work
  - Need way to limit work done at each merge to O(N)

Conquer using strips

- Key point is that the points must each be within D distance of the dividing line
  - Otherwise no way their distance is < D
- So, restrict search to points within a strip of width 2D
  - In example, A, W, Z, X, Y, H
- If points are randomly distributed, on average O(√N) points in each strip
  - So average case work is O(N)—ok
- But in worst-case could have all N points within the two strips
  - Worst-case is still $O(N^2)$—not ok

sorted by x:
EQAWZXYHR



Improving strips

- As described, strips remove from consideration all points more than 2D apart in the x direction
- We must also throw out points that are too distant in the y direction
- Suppose for a moment that we had the points in the strip sorted by y
  - (Not obvious we can do this in O(N) time)
  - For each point, only try other points that are less than D greater in y

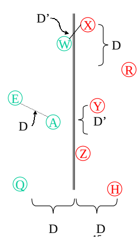
Improved strips example

- Consider X as the first point
  - W is within D below
  - X-W has shorter distance D'
    - Can just use this shorter distance in future tests
- Consider W
  - No points are vertically near
- Consider Y
  - A is within D' below
  - But Y-A are not closer than D'
- etc...A, Z, H don't have nearby points

- Final result: closest pair is W-X, D' distance

sorted by x: EQAWZXYHR

strip sorted by y: XWYAZH



Improved strips analysis

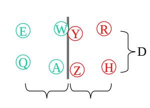
- Recall our recurrence: T(N) = 2 T(N/2) + f(N)
  - We want f(N) to be O(N)
- Work done in merge ("conquer") step:
  - Divide point set in half horizontally—O(1)
    - There are O(N) points in each half worst-case
  - Find points in strip near dividing line
    - Could use an O(log N) binary search, since points are sorted by x
  - Somehow sort strip by y coordinate [?? time ??]
  - For each point in strip [O(N) worst-case]:
    - Scan rest of strip for points within D distance vertically [?? time ??]
- For f(N) to be O(N), we must have: sorting by y is O(N), scanning is O(1)

Final details (1)

- Sorting O(N) points in strip by y
  - Seems to be O(N log N), which is too much work per step
  - Trick: start with point set sorted by y (call this YLIST)
  - At each step, do a linear scan of the sorted list
    - If point is in left half, copy to LEFT_YLIST
    - If point is in right half, copy to RIGHT_YLIST
    - Pass these sublists to recursive calls
    - For merge step, do linear scan of YLIST; remove items not in strip

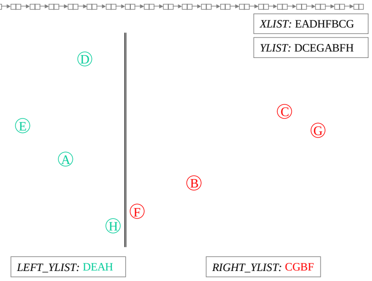
Final details (2)

- For each point in strip, we scan all lower points within D vertical distance
  - N steps, so must have O(1) points to scan per step
- For each point, we scan a 2D x D area
  - Strip is 2D wide, we scan up to D vertically
- At most 4 points on each side of dividing line
  - If there are any more, then recursions would have returned a smaller distance than D!
  - So, at most 7 other points need to be checked
  - Not a function of N, so this is O(1)



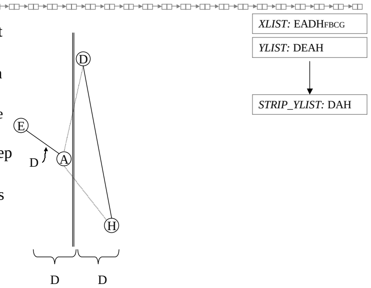
A full example

- Cut XLIST in half
- Scan YLIST, create two sublists
- Recurse...



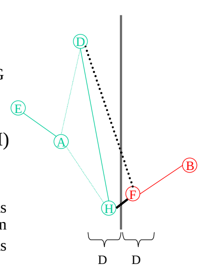
A full example (2)

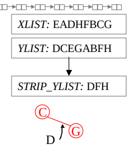
- Recursion on left half shown here
- Divide XLIST in half
  - Recursions are simple
- Scan YLIST; keep items in the strip
- No pair in strip is closer than D
- Return D



A full example (3)

- Recursions return
  - min dist is between CG
- Scan YLIST, extract items in strip (DFH)
- Do vertical scan:
  - From D: F is too far down
  - From F: H is closest





Note how few comparisons had to be made, vs. the $O(N^2)$ naive version given at the start.