Lecture 1: Dynamic Programming

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.stonybrook.edu/~skiena

Topic: Course Organization

- Course Organization
- Edit Distance
- Advanced Edit Distance
- Exact Pattern Matching
- Suffix Trees and Arrays

Who Am I?

- Professor of Computer Science, Stony Brook University.
- Former coach of our ICPC team: reached World Finals twice.
- Research interests in algorithms, data science, and NLP.

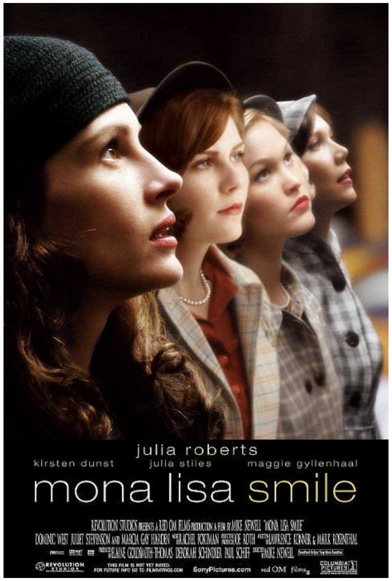
The Algorithm Design…	Programming Challenges	The Data Science…	Who's Bigger?	Computational Discrete…	Calculated Bets
1997	2003	2017	2013	2003	2001

Who are You?

- From what countries?
- From what school years?
- World finalists?
- How many robot judge problems have you solved?

Teaching Philosophy

How do you teach students who know everything?



My Vision

You are all largely self-trained in algorithms to get to the proficiency you have.
But there is a way to see the big picture of algorithm design that I think is valuable.
My plan is to teach you how I think of algorithmic problem solving in five distinct areas, and my hope is you will like it.

Lectures

- Dynamic programming and sequences
- Network flows and matchings
- Combinatorial objects and counting
- (Computational) Geometry
- Reductions and NP-completeness

Pair of Ears Needed

I do not speak Portuguese.

I need a volunteer who can relay questions from the room loudly and clearly enough for me to hear.

Thanks for your patience.

I do encourage questions and discussion in class, so please ask.

Contests

Set up by my loyal graduate student Tanzir Pial.

They are loosely coupled to the lecture topic of the day.

Do we want to discuss the previous contests at the beginning or end of lecture?

Should you do the contest as teams or individuals?

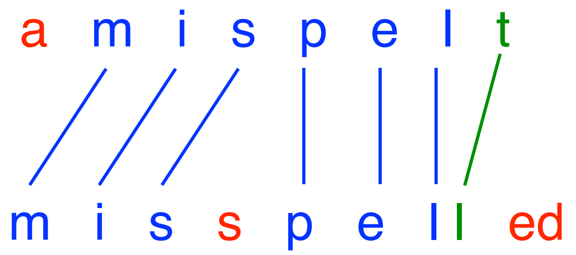
Questions?

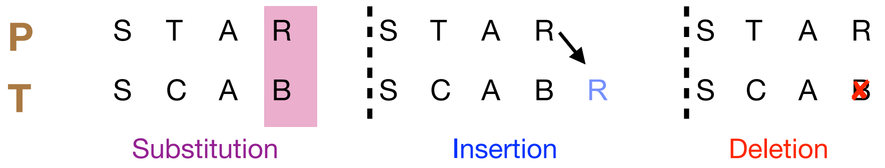
Topic: Edit Distance

- Course Organization
- Edit Distance
- Advanced Edit Distance
- Exact Pattern Matching
- Suffix Trees and Arrays

Edit Distance

amispelt ? misplaced
misspelled
misprinted





Dynamic Programming

The entire state of the recursive call is governed by the index positions into the strings. Thus there are only $|S| \times |T|$ different calls.

By storing the answers in a table and looking them up instead of recomputing, the algorithm takes quadratic time.

Dynamic programming is the algorithmic technique of efficiently computing recurrence relations by storing partial results.

It is very powerful on any *ordered* structures, like character strings, permutations, and rooted trees.

Table Structure

The table data structure keeps track of the cost of reaching this position plus the last move which took us to this cell.

```
typedef struct {
    int cost;                   /* cost of reaching this cell */
    int parent;                 /* parent cell */
} cell;


cell m[MAXLEN+1][MAXLEN+1];     /* dynamic programming table */
```

Edit Distance via Dynamic Programming

```
int string_compare(char *s, char *t, cell m[MAXLEN+1][MAXLEN+1]) {
    int i, j, k;        /* counters */
    int opt[3];         /* cost of the three options */

    for (i = 0; i <= MAXLEN; i++) {
        row_init(i, m);
        column_init(i, m);
    }


    for (i = 1; i < strlen(s); i++) {
        for (j = 1; j < strlen(t); j++) {
            opt[MATCH] = m[i-1][j-1].cost + match(s[i], t[j]);
            opt[INSERT] = m[i][j-1].cost + indel(t[j]);
            opt[DELETE] = m[i-1][j].cost + indel(s[i]);

            m[i][j].cost = opt[MATCH];
            m[i][j].parent = MATCH;
            for (k = INSERT; k <= DELETE; k++) {
                if (opt[k] < m[i][j].cost) {
                    m[i][j].cost = opt[k];
                    m[i][j].parent = k;
                }
            }
        }
    }

    goal_cell(s, t, &i, &j);
    return(m[i][j].cost);
}
```

To determine the value of cell (i, j), we need the the cells $(i-1, j-1)$, $(i, j-1)$, and $(i-1, j)$. Any evaluation order with this property will do, including the row-major order we used.

Standard String Edit Distance

The function string_compare is very general, and must be customized to a particular application.
It uses problem-specific subroutines match and indel to return the costs of character pair transitions:

```
void row_init(int i, cell m[MAXLEN+1][MAXLEN+1]) {
    m[0][i].cost = i;
    if (i > 0) {
        m[0][i].parent =  INSERT;
    } else {
        m[0][i].parent = -1;
    }
}
```

The functions row_init and column_init to initialize the boundary conditions.

```c
int match(char c, char d) {
    if (c == d) {
        return(0);
    }
    return(1);
}
```

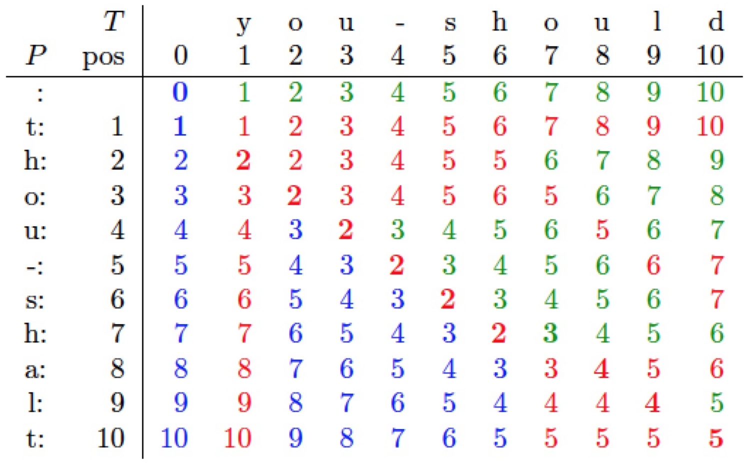
The function `goal_cell` returns the desired final cell of interest in the matrix.

```c
void goal_cell(char *s, char *t, int *i, int *j) {
    *i = strlen(s) - 1;
    *j = strlen(t) - 1;
}
```

Changing these functions lets us do substring matching, longest common subsequence, and maximum monotone subsequence as special cases.

String Matching Example: Cost Matrix

The cost matrix in converting *thou shalt not* to *you should not*:



	T		y	o	u	-	s	h	o	u	l	d
P	pos	0	1	2	3	4	5	6	7	8	9	10
:		0	1	2	3	4	5	6	7	8	9	10
t:	1	1	1	2	3	4	5	6	7	8	9	10
h:	2	2	2	2	3	4	5	5	6	7	8	9
o:	3	3	3	2	3	4	5	6	5	6	7	8
u:	4	4	4	3	2	3	4	5	6	5	6	7
-:	5	5	5	4	3	2	3	4	5	6	6	7
s:	6	6	6	5	4	3	2	3	4	5	6	7
h:	7	7	7	6	5	4	3	2	3	4	5	6
a:	8	8	8	7	6	5	4	3	3	4	5	6
l:	9	9	9	8	7	6	5	4	4	4	4	5
t:	10	10	10	9	8	7	6	5	5	5	5	5

String Matching Example: Parent Matrix

	T		y	o	u	-	s	h	o	u	l	d
P	pos	0	1	2	3	4	5	6	7	8	9	10
	0	**-1**	1	1	1	1	1	1	1	1	1	1
t:	1	**2**	0	0	0	0	0	0	0	0	0	0
h:	2	2	**0**	0	0	0	0	0	1	1	1	1
o:	3	2	0	**0**	0	0	0	0	0	1	1	1
u:	4	2	0	2	**0**	1	1	1	1	0	1	1
-:	5	2	0	2	2	**0**	1	1	1	1	0	0
s:	6	2	0	2	2	2	**0**	1	1	1	1	0
h:	7	2	0	2	2	2	2	**0**	**1**	1	1	1
a:	8	2	0	2	2	2	2	2	0	**0**	0	0
l:	9	2	0	2	2	2	2	2	0	0	**0**	1
t:	10	2	0	2	2	2	2	2	0	0	0	**0**

Reconstructing the Alignment

We must walk backwards to reconstruct the alignment either using explicit back pointers, or we can recalculate how we got to the critical cells starting from the back.

```
void reconstruct_path(char *s, char *t, int i, int j,
                      cell m[MAXLEN+1][MAXLEN+1]) {
    if (m[i][j].parent == -1) {
        return;
    }

    if (m[i][j].parent == MATCH) {
        reconstruct_path(s, t, i-1, j-1, m);
        match_out(s, t, i, j);
        return;
    }

    if (m[i][j].parent == INSERT) {
        reconstruct_path(s, t, i, j-1, m);
        insert_out(t, j);
        return;
    }
```

```
        if (m[i][j].parent == DELETE) {
            reconstruct_path(s, t, i-1, j, m);
            delete_out(s, i);
            return;
        }
    }
```

The actions we take on traceback are governed by `match_out`, `insert_out`, and `delete_out`:
The edit sequence from “thou-shalt-not” to “you-should-not” is `DSMMMMMISMSMMMM` – meaning delete the first ‘t’, replace the ‘h’ with ‘y’, match the next five characters before inserting an ‘o’, replace ‘a’ with ’u’, replace the ‘t’ with a ‘d’.

Questions?

Topic: Advanced Edit Distance

- Course Organization
- Edit Distance
- Advanced Edit Distance
- Exact Pattern Matching
- Suffix Trees and Arrays

Space Required for Edit Distance

How much space is required to compute the edit distance between strings of length n and m, where $n \geq m$?

Edit distance only looks at neighboring rows/columns to make its decisions.

Computing the *highest cell score* in the matrix does not require keeping more than then last column and the best value to date, for a total of $O(n)$ space, where $n \leq m$.

But what if we want to find the optimal alignment, instead of just its cost?

Space-Efficient Dynamic Programming

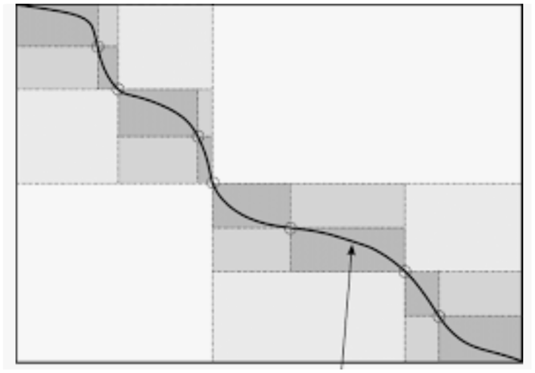
Quadratic space will kill you faster than quadratic time.

Reconstructing the optimal *alignment* seems to require keeping the entire matrix.

But Hirshberg found a clever way to reconstruct the alignment in $O(nm)$ time using only $O(n)$ space, by recomputing the appropriate portions of the matrix.

Hirchsberg's Method

For each cell, we drag along the row number where the optimal path to in crossed the middle ($m/2$nd) column.



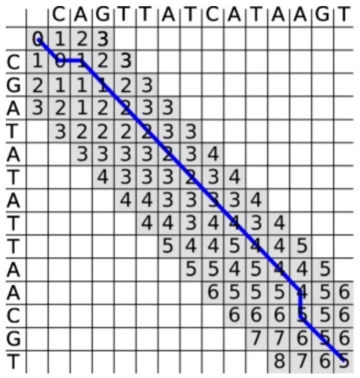
Knowing the crossing point k of the $(m/2)$nd column implies that the optimal alignment lies in submatrices A from (1,1) to $(m/2, k)$, and B from $(m/2, k)$ to (m, n).

Timing Analysis

The number of cells in these submatrices A and B total only half of the original mn cells.
Because dynamic programming algorithms are linear in the number of cells they compute, we can recur on A and B, where the the total amount of recomputation done is $\Sigma_{i=0}^{\lg m} mn/2^i = 2mn$ so the total work remains $O(mn)$.

Edit Distance between Similar Strings

When two strings are very similar, the optimal alignment cannot get too far from the main diagonal.
If the edit distance is k, it can never get more than k units from the main diagonal.



Banded Alignment

Edit distance can be done in $O(kn)$ even without knowing k in advance.

- Key idea 1: one sided binary search: try $k = 1, 2, 4, 8, \ldots$.
- Key idea 2: convergence of geometric series:

$$\sum_{i=0}^{x} 2^i < 2^{x+1}$$

  so it is order of the biggest term.

By repeatedly doubling the width of the band until we get a cost less than the width, we do $O(\log k)$ iterations with total cost $O(kn)$.

Gap Penalties

In many applications (e.g. shortening a novel) the length of the gap is relatively unimportant—“just deleting one chunk of text”

Gaps can be modeled as repeated single-base deletions, where the cost for a gap is linear in its length.

But a more general model charges a fixed penalty for the existence of each gap, plus another penalty depending upon the length of the gap.

Affine gap penalties are $A + Bt$ for gaps of length t, while *logarithmic* gap penalties of $A + B \lg t$ often useful.

Arbitrary Gap Weights

Suppose gap cost is a completely general function of length where we cannot assume monotonicity or any other property. Then we must explicitly try every possible length deletion/insertion at every possible position, i.e. $V(i,j)$ takes the best of the following options:

$$G(i,j) = V(i-1,j-1) + match(i,j)$$

$$E(i,j) = \min_{k=0}^{j-1} V(i,k) + indel(j-k)$$

$$F(i,j) = \min_{k=0}^{i-1} V(k,j) + indel(i-k)$$

Analysis: Arbitrary Gap Penalties

Because we are doing a linear amount of work for each cell, the time complexity goes to $O(n^2m + nm^2)$ or $O(n^3)$ if $n \geq m$.

This algorithm is often called Needleman-Wunsch.

Affine Gap Weights

By being clever we can avoid the extra linear cost of looking for the start of the gap for *affine* gap penalties, i.e. penalties of the form $A + Bt$ for gaps of length t

We will use the insertion and deletion recurrences E and F to encode the cost of being in gap mode, meaning we have already paid the cost of initiating the gap.

$$V(i,j) = \min(E(i,j), F(i,j), G(i,j))$$

$$G(i,j) = V(i-1,j-1) + match(i,j)$$

$$E(i,j) = \min(E(i,j-1), V(i,j-1) + A) + B$$

$$F(i,j) = \min(F(i-1,j), V(i-1,j) + A) + B$$

Analysis: Affine Gap Penalties

With constant amount of work per cell, this algorithm takes $O(mn)$ time, same as without gap costs.
The special case of *convex* penalty functions (including logarithmic costs) can be solved in $O(nm\log(mn))$ time with a more complicated algorithm.

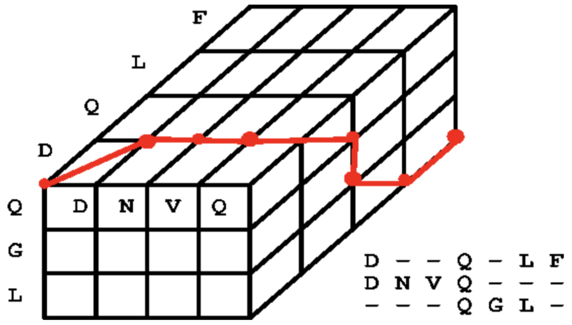
Multiple Sequence Alignment

A H A A I G D D E T N W O R T D T S	A H A A I G D D E T N W O R T D T S
H G I T G D D E A N W T O S R D S G	H G I T G D D E A N W T O S R D S G
G T A S H I D D E N W O R D T G A G	G T A S H I D D E N W O R D T G A G
H I G S D D E G N W O R G A D S T A	H I G S D D E G N W O R G A D S T A
H A I D A G D E N S W O R S A D T S	H A I D A G D E N S W O R S A D T S

The state here for k strings of length n is n^k.
Order states D_x by dominance (lesser or equal on all dimensions)
A dominance order is any topological sort of the dominance DAG.
If D_x all match, then $M[D_x] = 1 + \max(M[D]\text{for}D < D_x)$

Running Time for MSA

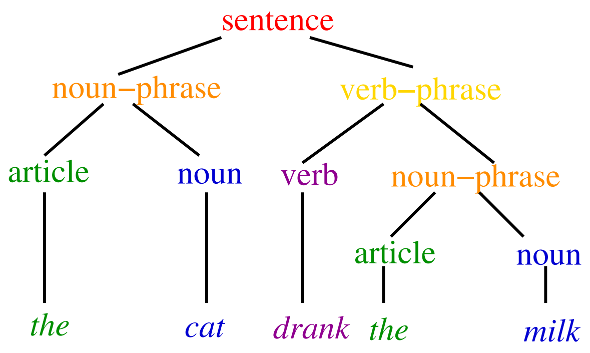
Muliple sequence alignment is exponential for large k, because there are 2^k predecessor states ($\delta = 0, -1$ for all k dimensions), so $O(2^k n^k)$ time.



Context Free Grammars

The complete set of syntactically correct programs in any programming language is defined by a context-free grammar:

sentence ::= noun–phrase
verb–phrase
noun–phrase ::= article noun
verb–phrase ::= verb noun–phrase
article ::= *the, a*
noun ::= *cat, milk*
verb ::= *drank*



Grammars are made up of rules built on collections of terminal symbols (the, a, cat, milk, drank) and *non-terminal symbols* or variable (sentence, noun-phrase, verb-phrase, article, noun).

Parsing Context-Free Grammars

The act of recognizing a syntactically correct program is called *parsing*.
Parsing builds a tree of rule applications (a parse tree) in the course of recognition.
We assume the program is length n, but the grammar is a fixed, constant size, so the complexity of parsing just depends on n.

Manipulating Grammars

Every context-free grammar can be translated into Chomsky Normal Form, where each rule is either (a) exactly two non-terminals: $X \rightarrow YZ$ or (b) exactly one terminal symbol: $X \rightarrow \alpha$.

Certainly a longer rule can be made shorter by adding a non-terminal symbol and associated rules: $X \rightarrow WYZ$ turned into $X_1 = WY$ and $X \rightarrow X_1 Z$

How can we parse strings using grammars in CNF?

Parsing using Dynamic Programming

Every node in a parse tree has a non-terminal label X, and represents a continguous substring of the input string s.

Let $M[i, j, X] =$ true iff there is a non-terminal X which has a rule which recognizes the substring $s_i \ldots s_j$.

This means there is a rule $X \rightarrow YZ$ such that there is a $i \leq k < j$ such that Y recognizes $s_i \ldots s_k$ and Z recognizes $s_{k+1} \ldots s_j$

Recurrence Relation for Parsing

$$M[i,j,X] = \bigvee_{(X \to YZ) \in G} \left(\bigvee_{k=i}^{j-1} M[i,k,Y] \wedge M[k+1,j,Z] \right)$$

where $\vee$ denotes the logical *or* over all productions and split positions, and $\wedge$ denotes the logical *and* of two Boolean values.

$M[i,i,X]$ is true iff there exists a production $X \to \alpha$ such that $S_i = \alpha$.

Running time is $O(n^3)$ assuming the grammar is of constant size.

Making Programs Syntactically Correct

Programs often contain trivial syntax errors that prevent them from compiling.

The LLMs that generate programs, like ChatGPT/Copilot may well produce syntactically incorrect code.

Problem: Given a context-free grammar G and input sequence S, find the smallest number of character substitutions you must make to S so that the resulting sequence is accepted by G.

Solution: Edit Distance with Parsing

Define $M'[i,j,X]$ to be an *integer* function that reports the minimum number of changes to subsequence $S_i \cdots S_j$ so it can be generated by nonterminal X.

$$M'[i,j,X] = \min_{(X \rightarrow YZ) \in G} \left(\min_{k=i}^{j-1} M'[i,k,Y] + M'[k+1,j,Z] \right)$$

This is still only $O(n^3)$ if you assume the grammar is constant-sized.

Boundary conditions: If there exists a production $X \rightarrow \alpha$, the cost of matching at position i depends on the contents of S_i. If $S_i = \alpha$, $M'[i,i,X] = 0$.

Questions?

Topic: Exact String Matching

- Course Organization
- Edit Distance
- Advanced Edit Distance
- Exact Pattern Matching
- Suffix Trees and Arrays

Taxonomy of String Matching Problems

Different string matching problems arise, depending on whether preprocessing techniques are appropriate:

- *Variable texts, Variable patterns* – Use an $O(n + m)$ algorithm like Knuth-Morris-Pratt or Boyer-Moore or Rabin-Karp.
- *Fixed texts, variable patterns* – e.g. search the human genome or the Bible. Suffix trees/arrays efficiently support repeated queries on fixed strings.
- *Variable texts, fixed patterns* – e.g. search a news feed for dirty words. Such applications justify preprocessing the set of patterns so as to speed search.

Randomized String Matching

The *Rabin-Karp* algorithm computes an appropriate hash function on all m-length strings of the text, and does a brute force comparison only if the hash value is the same for the text window and the pattern.
An appropriate hash function is

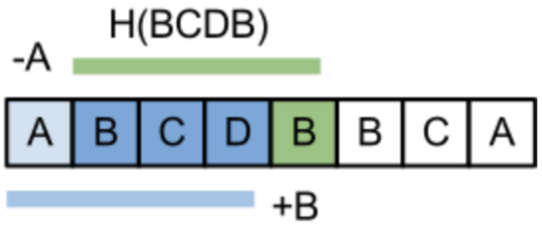
$$H(S) = \sum_{i=1}^{m-1} d^i S_i \bmod q$$

which treats each string as an m-digit base-d number, mod q.

Incremental Computation

This hash function can be computed *incrementally* in constant time as we slide the window from left to right since

$$H(S_{j+1}) = dH(S_j) + S_{j+1} - d^m S_{j-m}$$



Further, if q is a random prime the expected number of false positives is small enough to yield a *randomized* linear algorithm.

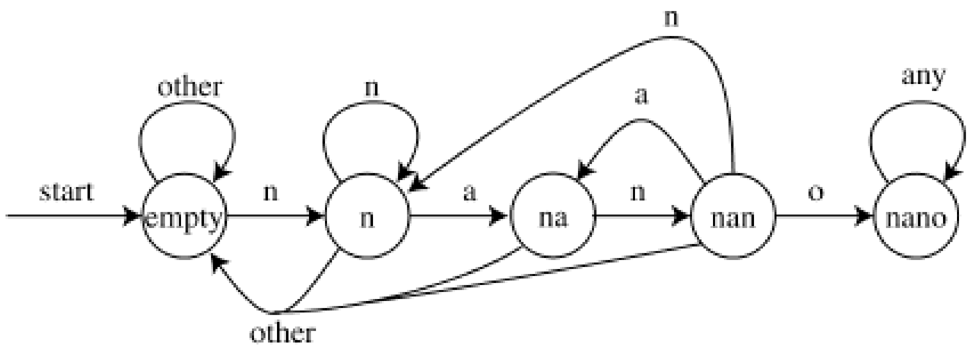
Multiple Exact Patterns

Many applications require searching a text for occurrences of any one of many patterns, e.g. searching text for dirty words or searching a genome for any one of a set of known motifs.

Pattern matching with *wild card* characters ($ACG?T$) is an important special case of multiple patterns.

Techniques from *automata theory* come into play, since any finite set of patterns can be modeled by *regular expressions*, and many interesting infinite sets (e.g. $G(AT)^*C$) as well.

The standard UNIX tool *grep* stands for "general regular expression pattern matcher".



The Aho-Corasick algorithm builds a DFA from the set of patterns and then walks through the text in linear time, taking action when reaching any accepting state.

Questions?

Topic: Suffix Trees and Arrays

- Course Organization
- Edit Distance
- Advanced Edit Distance
- Exact Pattern Matching
- Suffix Trees and Arrays

Tries and Trees

There are several interesting data structures for speeding up exact pattern matches in strings.

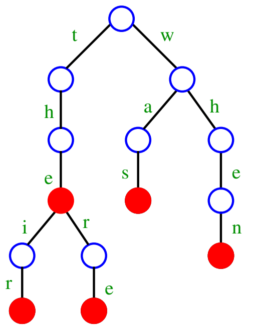
A *trie* is a data structure which permits access to any string s in an n word dictionary in $O(|s|)$ time for constant-sized alphabets.

This is optimal and independent of the dictionary size!

Note that binary search of an n word dictionary would take $O(\log n|s|)$ time.

Tries

A trie has a node for each character position, with prefixes shared:

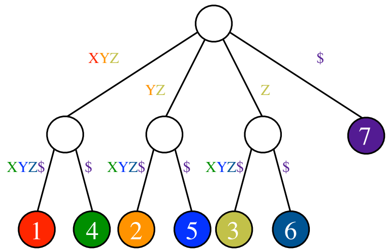


Searching in a trie is easy: just match the character and traverse down the correct path.
Building the trie is also easy: insert a new string by matching until you fail, then split the last node.

Suffix Trees

A special set of patterns are all *suffixes* of a string.

XYZXYZ$
YZXYZ$
ZXYZ$
XYZ$
YZ$
Z$
$



With such a tree, we can perform *substring* searches efficiently, since every substring is the prefix of some suffix. Further, the set of all instances of a given substring t are the leaves of the subtree rooted at t, and can be found by DFS.

Linear-Size Suffix Trees

A suffix tree can be stored in linear space, by collapsing degree-1 nodes into paths, and paths into references to the original string.

The incremental insertion algorithm to build a suffix tree might take $O(n^2)$ time to build the tree, because finding the split-point for each insertion might require $O(n)$ time in matching.

However, there are more sophisticated algorithms (Weiner's, Ukkonen's, McCreight's) which can build the *entire tree* in linear time.

Exploiting Suffix Trees: Longest Common Substring

Given two strings s_1 and s_2, what is the longest *contiguous* substrings they have in common.

Example: *live*stock and sea*live*r

The naive $O(nm)$ algorithm fully tests each alignment of s_1 against s_2.

In 1970, Knuth conjectured that a linear-time algorithm was impossible. Can you prove him wrong?

Longest Common Substring Algorithm

Build a suffix tree of the length $n+m+1$ concatenated string $s' = s_1 \# s_2$, where $\#$ does not occur in either string.

Label each leaf node of the suffix tree with the name of the string it is contained in. Label each internal node with the union of the labels of its descendents.

By doing a DFS on the $O(n+m+1)$ node tree, we can find the deepest node which has both an s_1 and s_2 descendant. This defines the longest common substring!

Exploiting Suffix Trees: Palindromes

A *palindrome* reads the same forwards and backwards: *A man, a plan, a canal – Panama* or *Madam I'm Adam*.

How can we find the longest palindrome within a string?

MAMAB MADAMIMADAMABA

Finding Palindromes

How can we find the longest palindrome within a string? Use the longest common substring algorithm with s_1 as the input sequence and s_2 as its reverse sequence!

This does not guarantee that the lcs of these strings starts/ends in the same place, so does not necessarily find a palindrome.

However, after we augment the suffix tree so as to answer *lowest common ancestor* queries in constant time…

…we can find the longest sub-palindrome in linear time by asking the 'length' of the LCA of $S[i]$ and $S[i + n + 1]$ for each $1 \leq i \leq n$.

Exploiting Suffix Trees: Circular String Linearization

To look a circular molecule up in a database, we must find a *canonical* place to break it to leave a linear string.

ABAACA
AABAAC
CAABAA
ACAABA
AACAAB
BAACAA

The most obvious place to break it uniquely is so as to always leave the *lexicographically* smallest string, i.e. the string which appears first in sorted order.
Building and sorting all n such strings takes $O(n^2)$ time. Can you do better?

Linear Time Linearization

Break the string arbitrarily to create a linear string L.
Now build the suffix tree for string $S = LL\#$. This is linear in the size of the input.
Example: $L = gcttcaat$ so $S = gcttcaatgcttcaat\#$.
Do a traversal down from the root, always picking the lexicographically smallest character. Assume that $\#$ is at the end of the alphabet.

Suffix Arrays

The *suffix array* is an amazing data structure for efficiently searching whether S is a substring of string T.
For a given string T, we construct the lexicographically sorted array of all its *suffixes*.
For $T = mississippi$, the suffix array is:

```
11 : i
8 : ippi
5 : issippi
2 : ississippi
1 : mississippi
10 : pi
9 : ppi
7 : sippi
4 : sissippi
6 : ssippi
3 : ssissippi
```

Searching in a Suffix Array

Since every substring is the prefix of some suffix, Substring search now reduces to binary search in this array. Example: is “sip” a substring of T?

Binary search in a suffix array takes $O(m \lg n)$, where n is the length of T and m the length of the matched substring.

With auxilliary data, this can be improved to $O(\lg n + m)$.

Note that we can just as easily find *all* the occurrences of a given string S in T by binary searching just before/after S.

Building and Storing Suffix Arrays

Amazingly we need only store the original string and the sorted start positions to do the search! The jth character of the ith prefix is at $T[start[i] + j - 1]$.

But how fast can be built the suffix array of an n character string?

Radix sorting n strings of n characters can be done in $O(n^2)$, linear in the size of the input.

Building Suffix Arrays Efficiently

But what is really amazing is that suffix arrays can be built in both linear time and space!

First, build a *suffix tree* in $O(n)$ time.

Performing a lexicographic depth first search of a suffix tree yields a suffix array.

Suffix arrays use many times less space than suffix trees (say $3n$ vs. $17n$ bytes), which is often the dominating factor in large text search problems.

Questions?