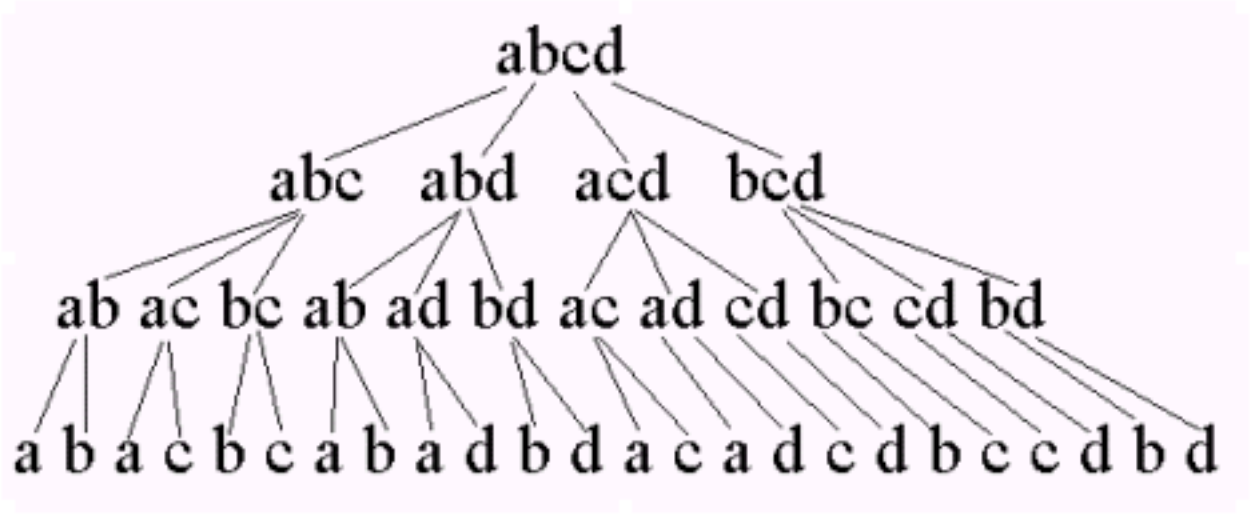
Longest Common Subsequence

student **David Morano**
advisor **Professor David Kaeli**

NUCAR seminar 99/10/29



Problem Definition

Given :

A finite alphabet **A**, finite set **R** of strings from **A***, and a positive integer **K**

Is There :

A string **w** element of **A*** with **| w |** >= **K** such that **w** is a subsequence of each **x** element of **R** ?

Complexity :

- The problem remains NP-complete for all **A (R)** >= 2
- Analogous Longest Common Substring problem is trivially solvable in polynomial time

Example Instance

Given as input, the strings :

DEABC

ABCD

FGHABD

Common Subsequences Are :

A, B, D, AB

The Longest Common Subsequences Are :

AB

Some Applications

- String - to - String Correction (spell checking & correction)
- Distance Between Strings (database keys)
- Minimum Cost Edit Operation Between Strings
- Amino Acid Sequences in Molecular Biology
- Data Compression Techniques

Benchmarks

No standard benchmarks were found !

We created our own benchmarks for comparison purposes

All generated randomly

15 “easy” instances (short strings)

15 “hard” instances (long strings)

CPLEX Solver

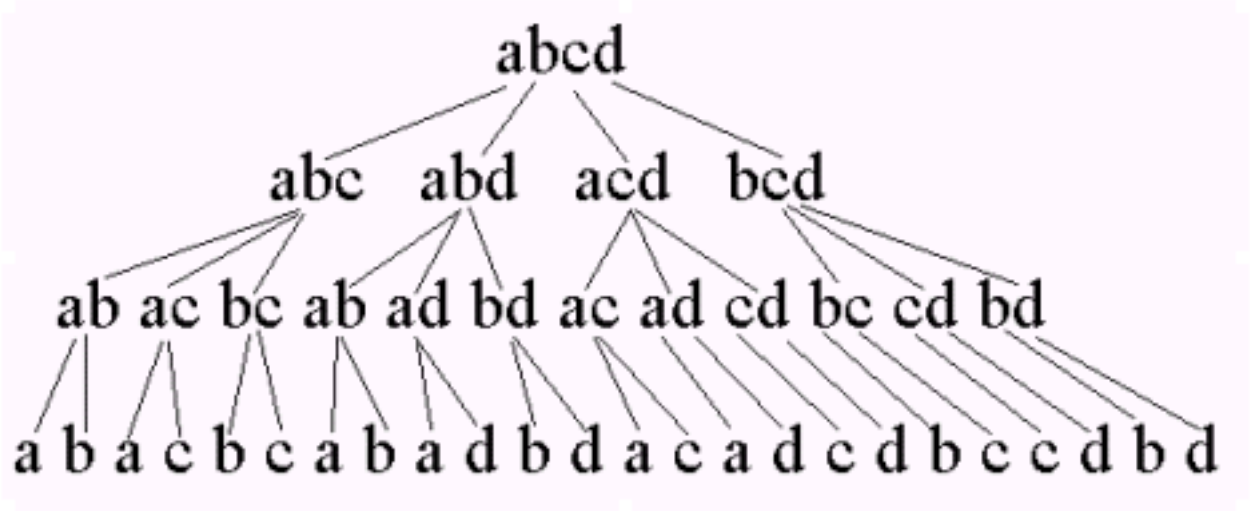
- difficult to express in AMPL
- no easy way to parametize the AMPL model
- exceedingly long execution times for even simple problem instances !
- all but useless for real problem solving needs !
- is its running time just a :
  - constant times the exponential bound
  - a polynominal function times the bound
  - or the product of two or more expontentials ?

Exhaustive Algorithms

- Exhaustive #1

given input strings :DEABC, ABCD, FGHABD

expand them all into subset subsequences as :



take the intersections of all subsets between the expansions of all strings

Exhaustive Algorithms

- Exhaustive # 1 (continued)
  - take expansion of each string in the order (depth first) :

    abcd, abc, ab, a, b, ac, a, c, bc, b, c, ...
  - check, in turn, each of these subsequences against all other subset subsequences of the other input strings
  - common subsequences found are :

    ab, a, b
  - the LCS is : ab

Exhaustive Algorithms

❑ Exhaustive #2

- similar to Exhaustive #1 but now we only expand out the shortest string in the instance
- check each of these subsequences directly in the other strings

❑ Exhaustive #3

- similar to Exhaustive #2 but now we check the expanded subsequences from the longest to the shortest and we stop when we start to check a subsequence shorter than the incumbent LCS found so far (our Branch & Bound was like this but took subproblems depth first)

Exhaustive Algorithms

- Exhaustive #4
  - similar to Exhaustive #3, we stop as soon as we consider a subsequence that is the same size or smaller than the incumbent LCS found so far (only one LCS is found)

Exhaustive Algorithms

- Exhaustive #5
  - expands out the shortest input string according to a binary counting order (counting down from all ones)
  - characters set in the counter are considered, characters not set are ignored
  - to speed it up, no possible subsequence is considered that is shorter than the incumbent LCS found so far

1111...
0111...
1011...
0011...

Greedy Algorithm

- expands out the shortest input string looking for a single character that is common to all input strings
- tries to lengthen the single character LCS by searching all remaining characters in the “expanding string” for matches in the other strings (in order after previous characters)
- this last step is what we call an “add-on” local search and is used extensively in almost all other algorithms we devised
- this algorithm, although not the best, finds remarkably good LCSes compared with the other non-exhaustive algorithms

Local Search Algorithms

- Greedies with “add-on”

  - find starting LCSes by running the first part of Greedy on all input strings
  - runs “add-on” local search on all LCSes found
    - —uses two types of bounds to save time
      - – remaining length of expanding string
      - – remaining length of searched string
  - finds almost optimal LCSes for most instances
  - runs exceedingly fast in O(input_strings * string_lengths)

Local Search Algorithms

❑ Greedies with “back-track”

- same as with “add-on” only but uses a Tabu back-tracking technique to get out of the local optimum found from expanding each input string
  - —backtracks through the entire expanding string (back to front) stopping at each previously found, greedily added, subsequence character
  - —at each stop we then greedily find a possibly better LCS from that point through the end of the expanding string
- finds almost optimal LCSes for most instances (usually better than previous)
- runs exceedingly fast in O(input_strings * string_lengths)

Local Search Algorithms

- Random with “add-on”

  - find starting LCSes by finding random binary selections of the characters of each of the input strings
    —there are many ways to randomly select characters
  - runs “add-on” local search on all LCSes found
  - finds almost optimal LCSes for most instances
  - runs exceedingly fast

Local Search Algorithms

❑ Random with “back-track”

- same but uses a back-tracking technique to get out of the local optimum found from expanding each case
- finds almost optimal LCSes for most instances
- runs exceedingly fast for most instances (even long ones)

Local Search Algorithms

- Exhaustive-like with “add-on”

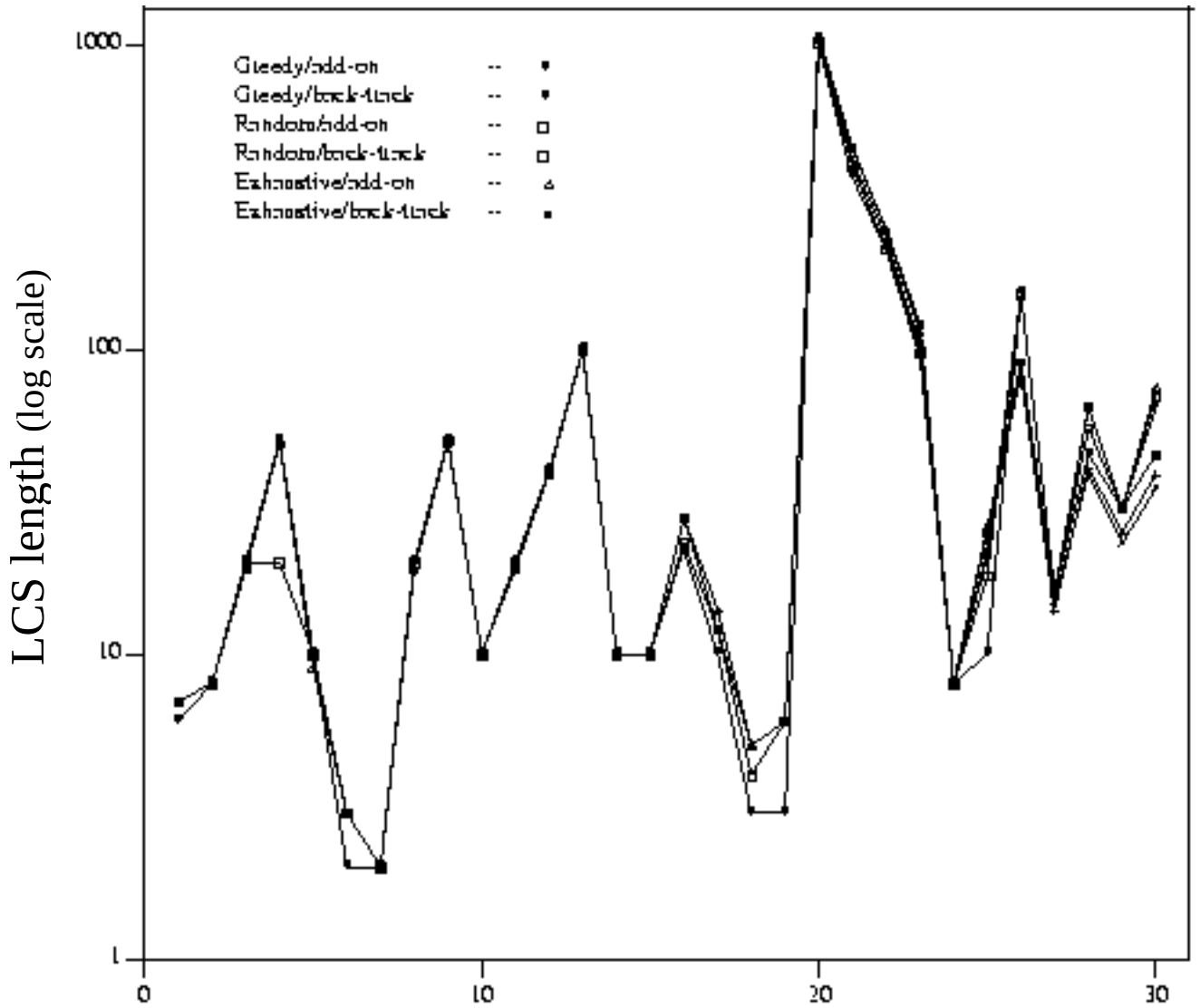
  - find starting LCSes by finding a subset of all binary selections of the characters of each of the input strings
  - runs “add-on” local search on all LCSes found
  - finds almost optimal LCSes for most instances
  - usually finds the optimal LCSes for “shorter” instances
  - runs moderately fast for small to medium, longer for large

Local Search Algorithms

- Exhaustive-like with “back-track”

  - same but uses a back-tracking technique to get out of the local optimum found from expanding each case
  - similar quality to above but different result characteristics
  - similar run time to above

Results


1000
100
10
1
0
10
20
30
LCS length (log scale)
instances
Greedy/add-on --
Greedy/back-track --
Random/add-on --
Random/back-track --
Exhaustive/add-on --
Exhaustive/back-track --

Some Observations

- fast executions tend to occur when at least one string is short (about 16 characters or less in size) or when the optimum LCS is close in size to the shortest input string
- “back-tracking” and “exhaustively started local searches” tend to give more LCSes per solution (not necessarily better LCSes)
- exhaustively started local search is good for shorter instances but randonly started local searches tend to be better for longer instances
- sometimes, greedy related algorithms give some of the best LCSes of all non-exhaustive searches
- for hard instances, the quality of the solutions from all algorithms seems to converge to simply how much search space is covered in a given time
- no single algorithm (except for an exhaustive search) is always better than all of the others

Other Things Tried

- use each problem string as the expanding string for a while
- some variations on random character selection
- several combinations of all of above

What Could Still be Tried

- dynamic programming
- optimal exhaustive expansion finding all LCSes
  - enumerate the expanding string from longest subsequences first
  - upper bound is still O(e**N) but probably much less typically
  - search through all possible LCSes of the same length before stopping
- use varying thresholds for selecting random characters from the expanding string
- use variable random probability density functions for selecting random characters from the expanding string for possible LCSes
- et cetera !