# Design and Implementation of Speech Recognition Systems

*Spring 2013*

Class 4: Dynamic Time Warping
4 Feb 2013

# English or German?

The European Commission has just announced that English, and not German, will be the official language of the European Union.

But, as part of the negotiations, the British Government conceded that English spelling had some room for improvement and has accepted a 5- year phase-in plan that would become known as "Euro-English".

# English or German?

In the first year, "s" will replace the soft "c“. The hard "c" will be dropped in favour of "k". This should klear up konfusion, and keyboards kan have one less letter.

In the sekond year the troublesome "ph" will be replaced with "f". This will make words like fotograf 20% shorter.

In the 3rd year Governments will enkourage the removal of double letters which have always ben a deterent to akurate speling. Also, the horibl mes of the silent "e" will go away.

By the 4th yer people wil be reseptiv to steps such as replasing "th" with "z" and "w"with "v".

During ze fifz yer, ze unesesary "o" kan be dropd from vords kontaining "ou" and after ziz fifz yer, ve vil hav a reil sensi bl riten styl. Zer vil be no mor trubl or difikultis and evrivun vil find it ezi tu understand ech oza. Ze drem of a united urop vil finali kum tru.

Und efter ze fifz yer, ve vil al be speking German like zey vunted in ze ferst plas!

# Why is Garbled Text Recgonizable?

- E.g.:
  - Also, al wil agre that the horibl mes of the silent "e" in the languag is disgrasful and it should go away.

- Why do we think ***horibl*** should be ***horrible*** and not ***broccoli*** or ***quixotic***?

- May sound like a silly question, but one of the keys to speech recognition

# Why is Garbled Text Recgonizable?

- Possible reasons:
  - Words "look" recognizable, barring spelling errors
    - E.g. ***publik***
  - Words "sound" recognizable when sounded out
    - E.g. ***urop***
  - Context provides additional clues
    - E.g. ***oza*** in " … each oza."

- Of these, which is the most rudimentary? Most complex?

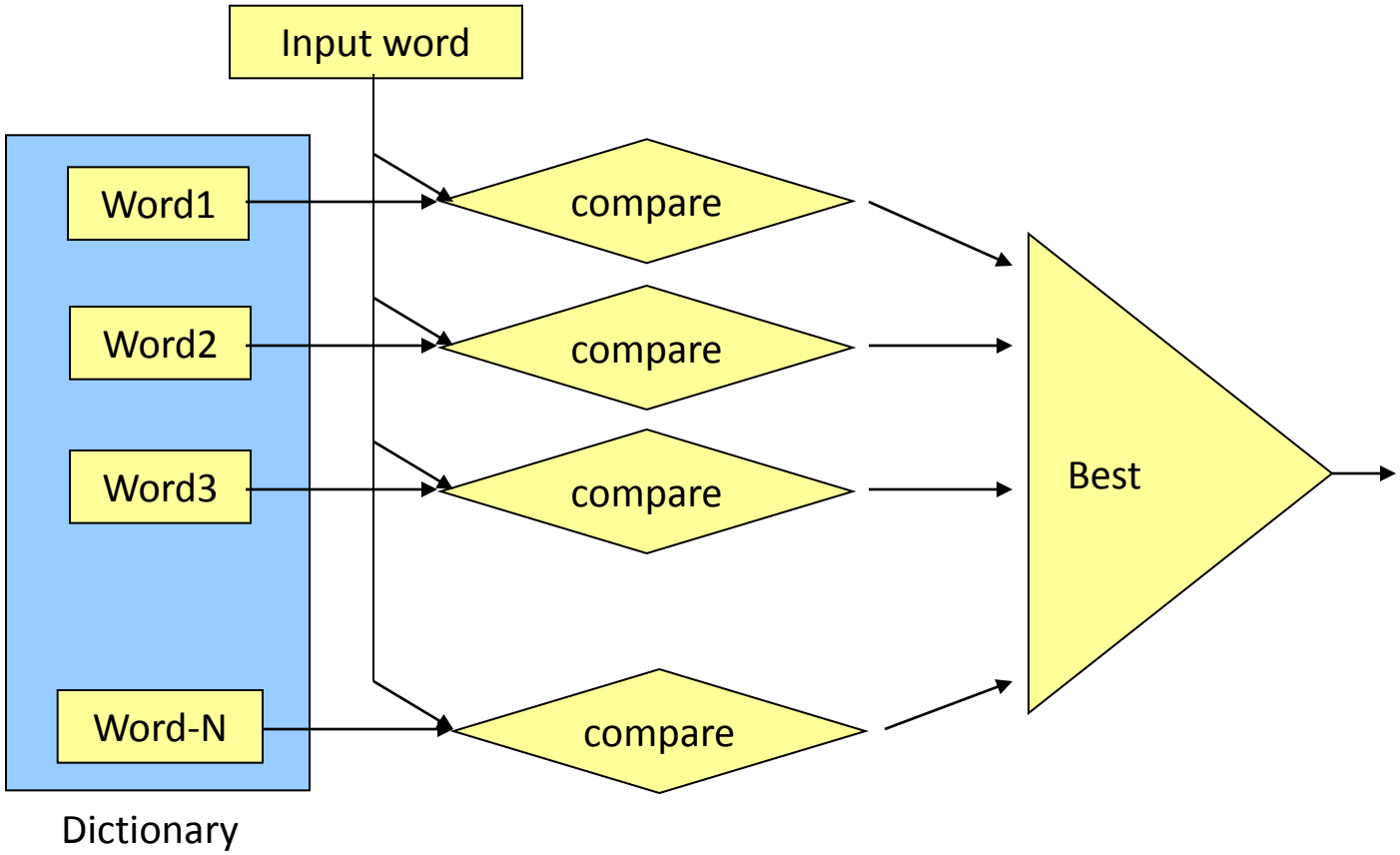
# How to Automate German -> English?

# How to Automate German -> English?

- Start with simple problem:
  - Treat each word in isolation
  - Handle spelling errors only (surface feature)

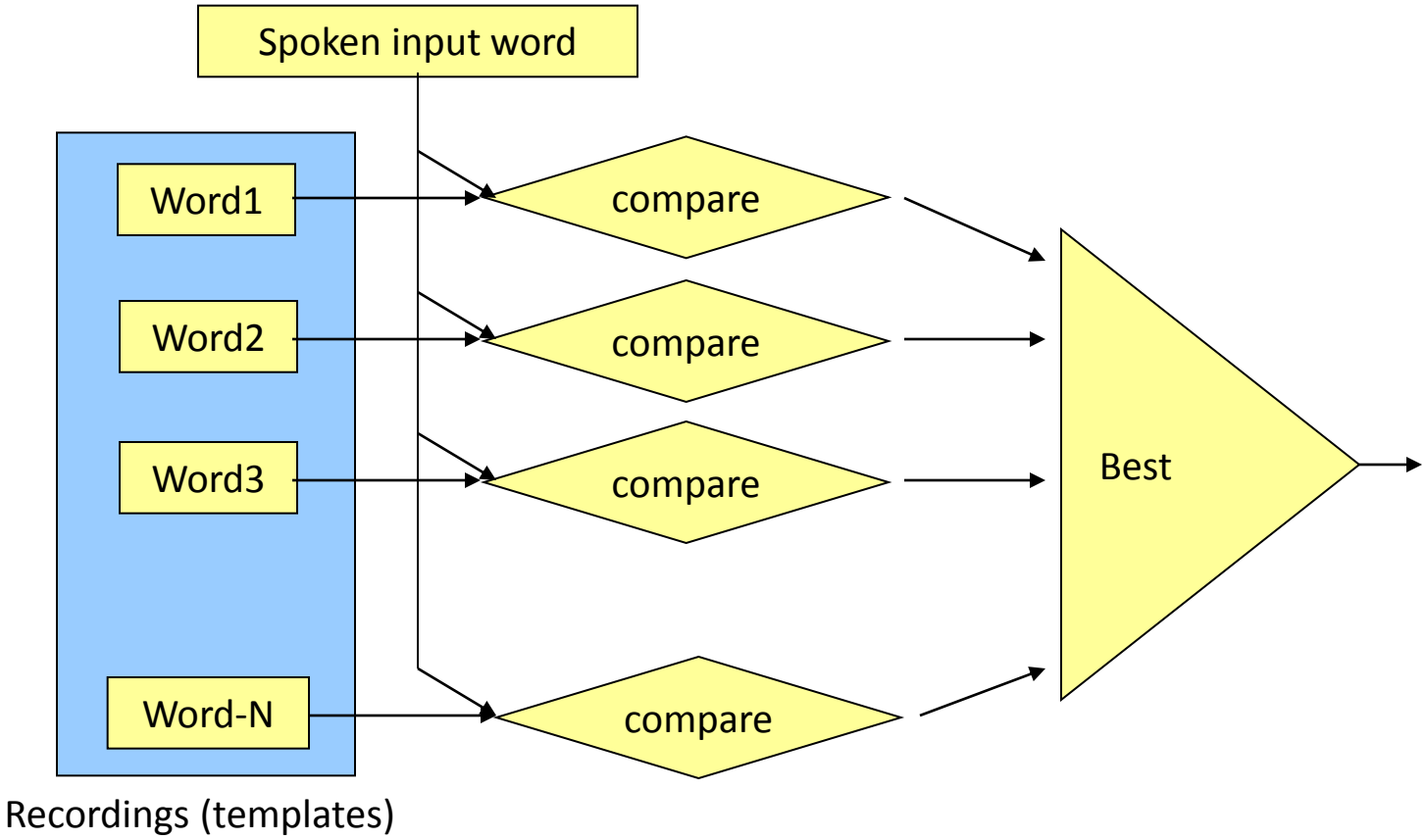
- In other words:
  - Ignore “sounding like” and “context” aspects

# How to Automate German -> English?




- Only unknown: The ***compare*** box
  - Exactly what is the comparison algorithm?

# Relation to Speech Recognition?




- Isolated word recognition scenario

# Problems in Comparing Strings?

# String Comparison

- If the only spelling mistakes involve ***substitution*** errors, comparison is easy:
  - Line up the words, letter-for-letter, and count the number of corresponding letters that differ:

    ***P U B L I K***
    ***P U B L I C***

    ***P U B L I K***
    ***P E O P L E***

  - But what about words like ***agre*** (agree)? How do we "line up" the letters?

# String Comparison

- In general, we consider three types of string errors:
  - *Substitution*: a template letter has been changed in the input
  - *Insertion*: a spurious letter is introduced in the input
  - *Deletion*: a template letter is missing from the input

- These errors are known as ***editing operations***

***P U B L I K***
***P U B L I C***

***P O T A T O E***
***P O T A T O***

***A G R  E***
***A G R E E***

# String Comparison

- Why did we pick the above ***alignments***? Why not some other alignment of the letters:

***P U B L I K***
***P U B L I C***

***A G R E***
***A G R E E***

# String Comparison

- Why did we pick the above ***alignments***? Why not some other alignment:

***P U B L I K***
***P U B L I C***

***A G R E***
***A G R E E***

- Because these alignments exhibit a ***greater*** edit distance than the "correct" alignment

# String Comparison Problem

- Given two arbitrary strings, find the ***minimum edit distance*** between the two:
  - Edit distance = the *minimum number of editing operations* needed to convert one into the other
  - Editing operations: substitutions, insertions, deletions
  - Often, the distance is also called ***cost***
- This minimum distance is a measure of the ***dissimilarity*** between the two strings
- Also called the ***Levenshtein distance***

# String Comparison Problem

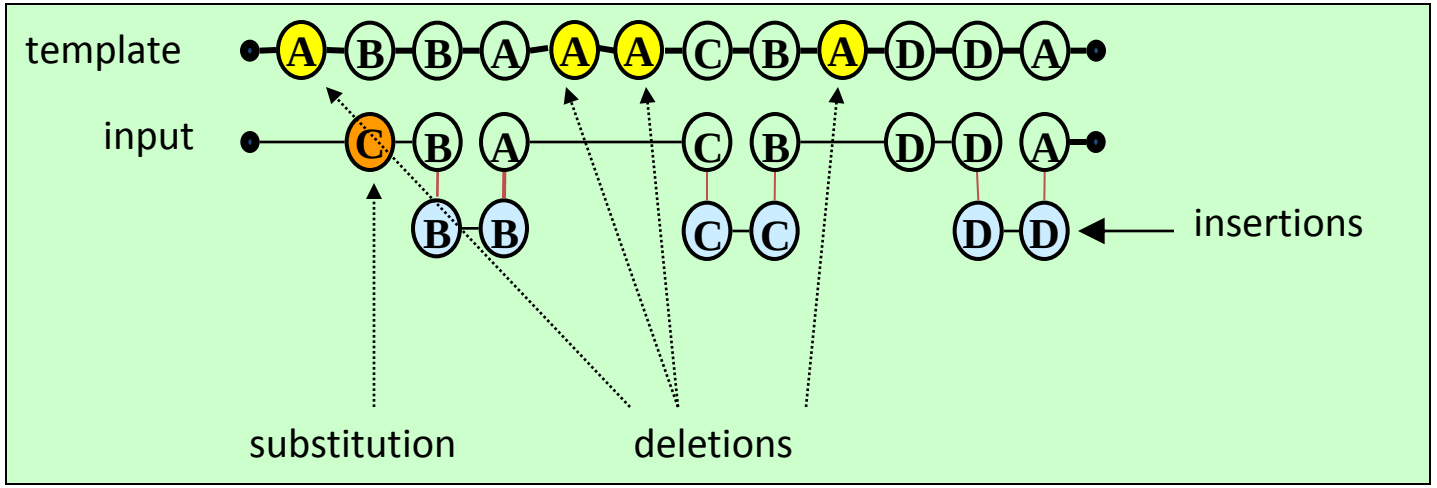
- How do we compute this minimum edit distance?
- With words like ***agre*** and ***publik***, we could eyeball and "guess" the correct alignment

- Such words are familiar to us
- But we cannot "eyeball and guess" with unfamiliar words

- Corollary: ***ALL*** words are unfamiliar to computers!
  - We need an algorithm

# String Comparison Example

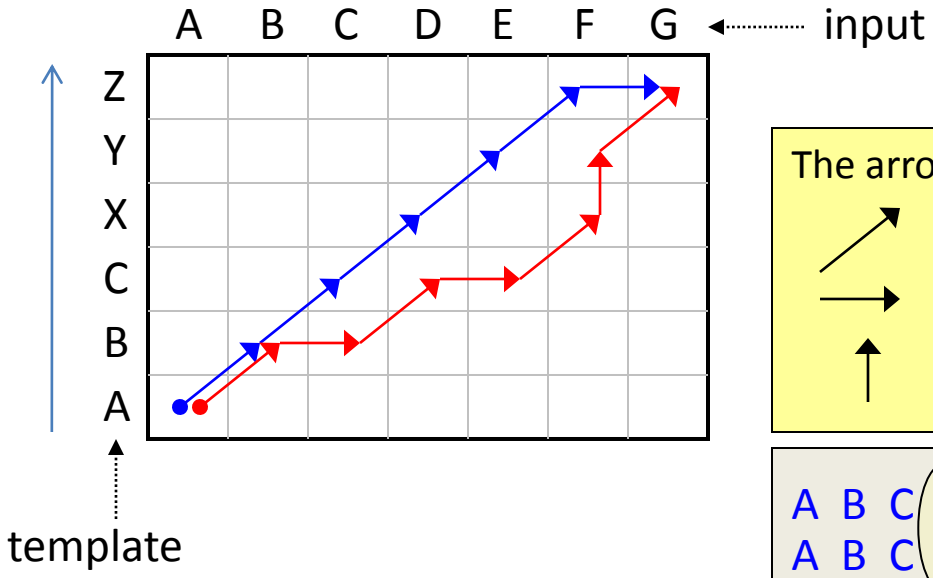
- Hypothetical example of unfamiliar word:
  - Template: ABBAAACBADDA
  - Input: CBBBACCCBDDDDA




- Other alignments are possible
- Which is the "correct" (minimum distance) alignment?
- Need an algorithm to compute this!

# String Edit Distance Computation

- Measuring edit distance is best visualized as a 2-D diagram of the template being ***aligned*** or ***warped*** to best match the input
  - Two possible alignments of template to input are shown, in blue and red

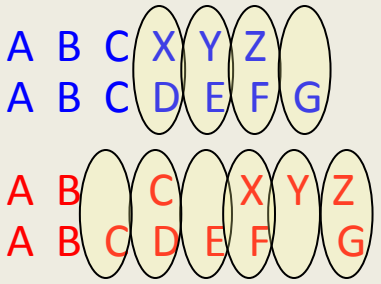



The arrow directions have specific meaning:

= Correct or substituted

= Input character inserted

= Template character deleted



# Minimum String Edit Distance

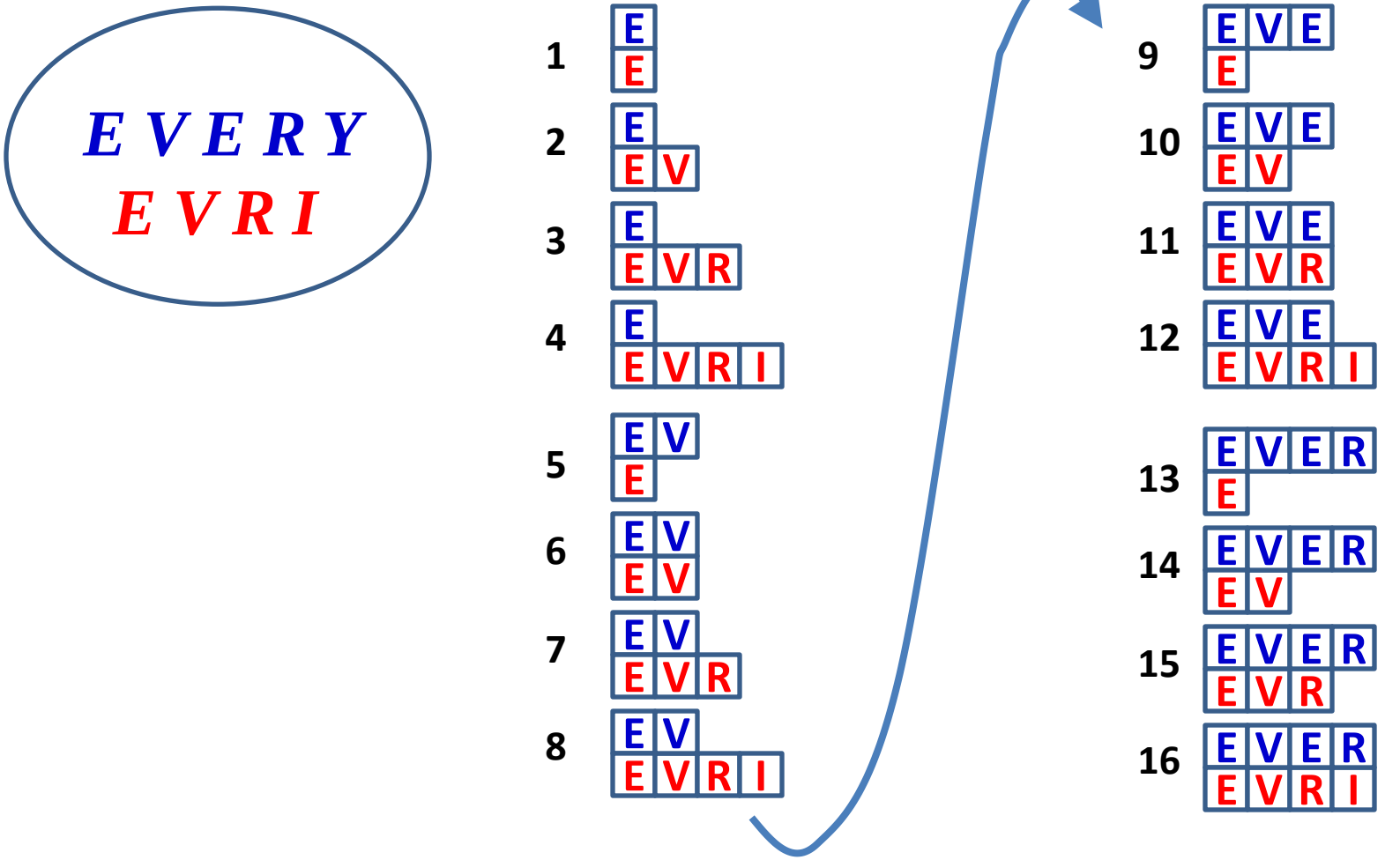
- This is an example of a ***search*** problem, since we need to search among all possible paths for the best one

- First possibility: Brute force search
  - Exhaustive search through all possible paths from bottom-left to top-right, and choose path with minimum cost
  - But, computationally intractable; exponentially many paths!
    - (*Exercise*: Exactly how many different paths are there?)
    - (A path is a connected sequence made up of the three types of arrows: diagonal, vertical and horizontal steps)

- Solution**:** ***Dynamic Programming*** (DP)
  - Find optimal (minimum cost) path by utilizing (re-using) optimal sub-paths
  - ***Central to virtually all major speech recognition systems***

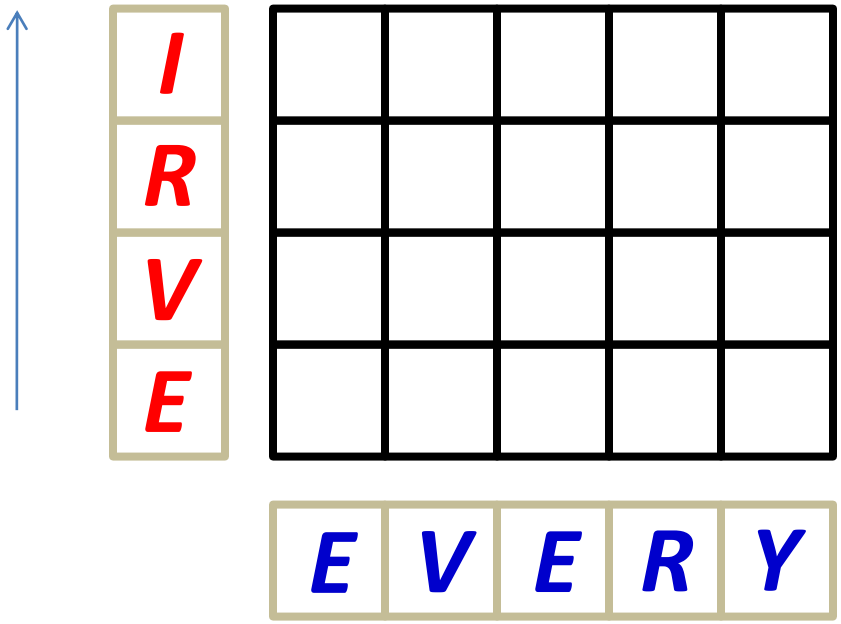
# Minimum String Edit Distance: DP

- *Central idea*: Compute the edit distance by incrementally comparing substrings of increasing length:



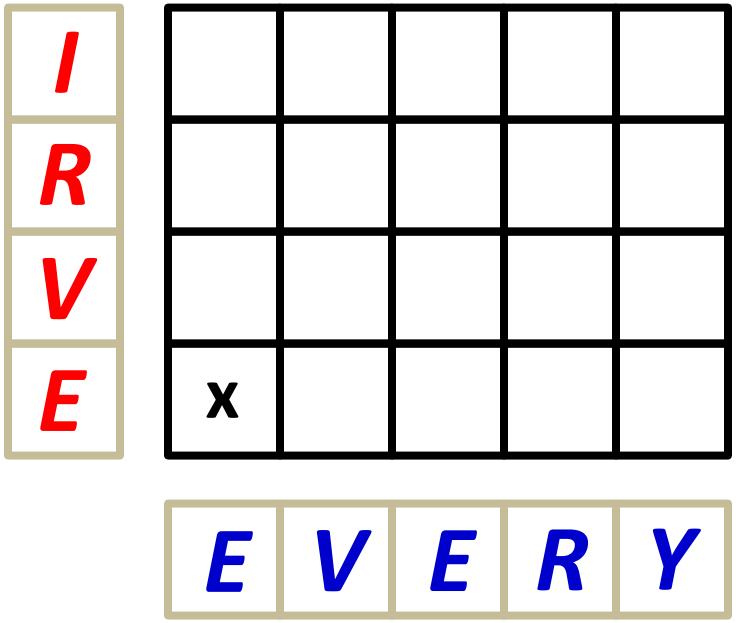

Continues till ***EVERY*** is compared to ***EVRI***

# *EVERY* vs. *EVRI*



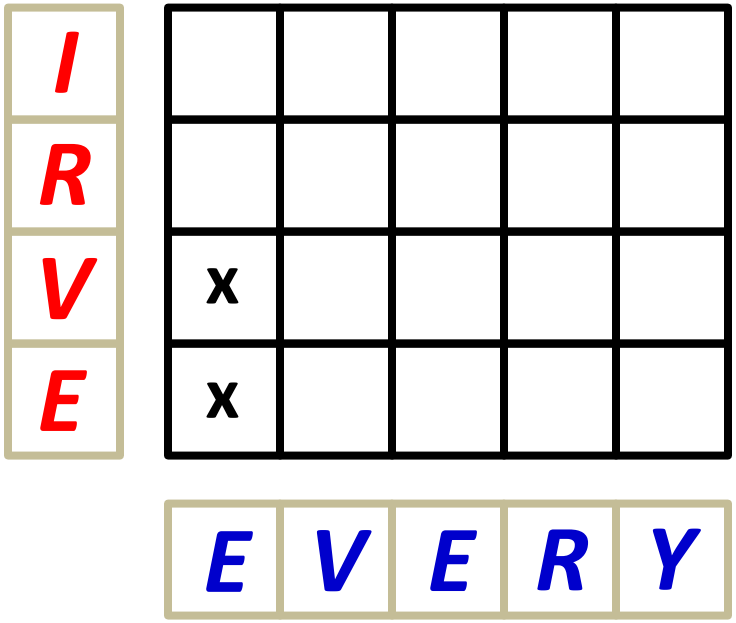

- Incrementally compare substrings
- Each substring-substring comparison depends only on the results of the previous substring comparisons

# *EVERY* vs. *EVRI*



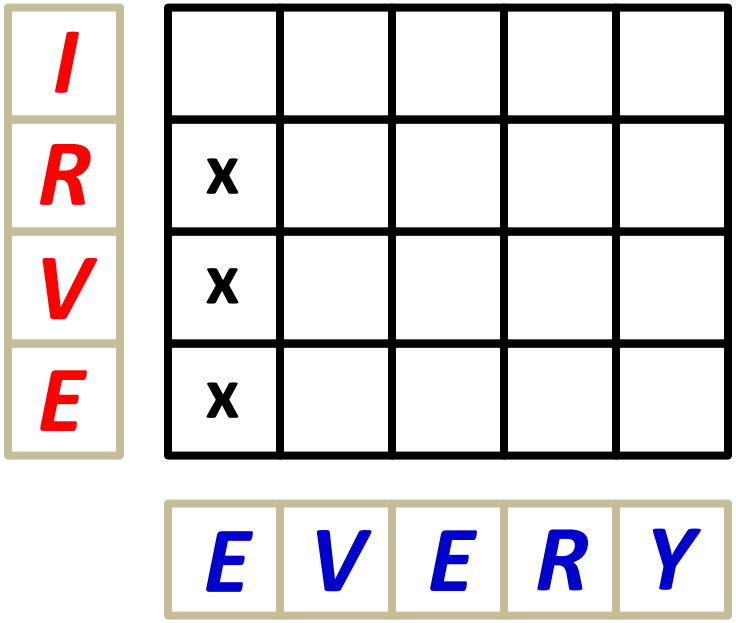

- Incrementally compare substrings
- Each substring-substring comparison depends only on the results of the previous substring comparisons

# *EVERY* vs. *EVRI*



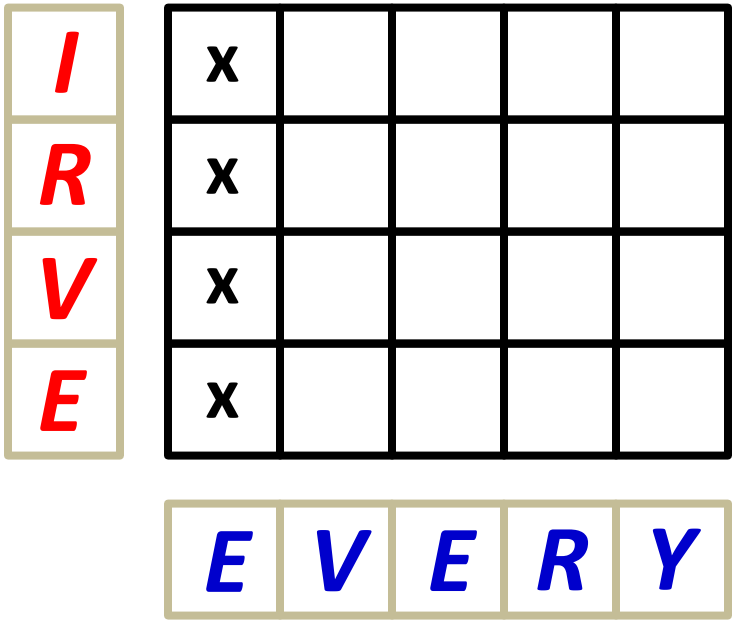

- Incrementally compare substrings
- Each substring-substring comparison depends only on the results of the previous substring comparisons

# *EVERY* vs. *EVRI*



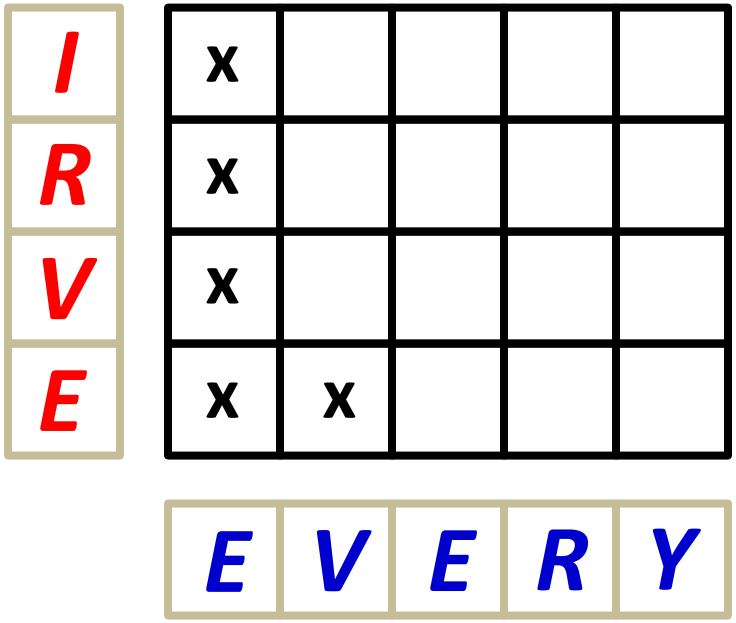

- Incrementally compare substrings
- Each substring-substring comparison depends only on the results of the previous substring comparisons

# *EVERY* vs. *EVRI*



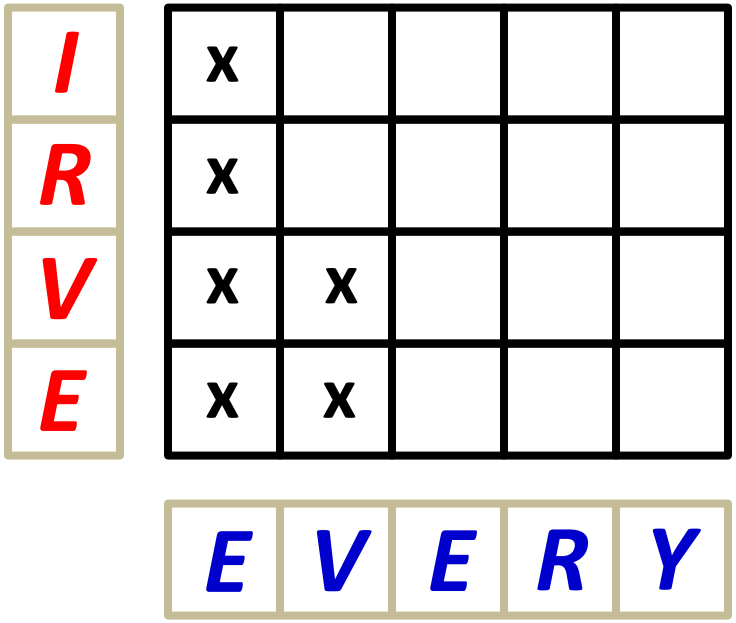

- Incrementally compare substrings
- Each substring-substring comparison depends only on the results of the previous substring comparisons

# *EVERY* vs. *EVRI*



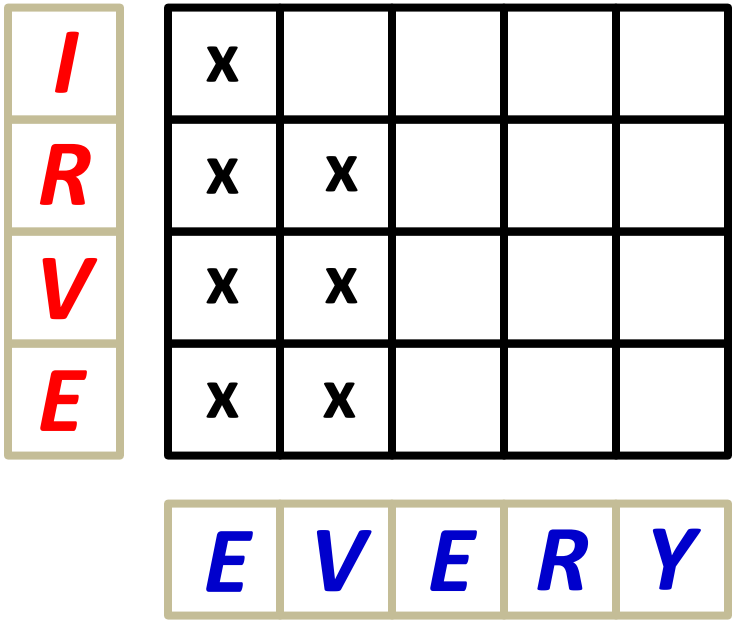

- Incrementally compare substrings
- Each substring-substring comparison depends only on the results of the previous substring comparisons

# *EVERY* vs. *EVRI*



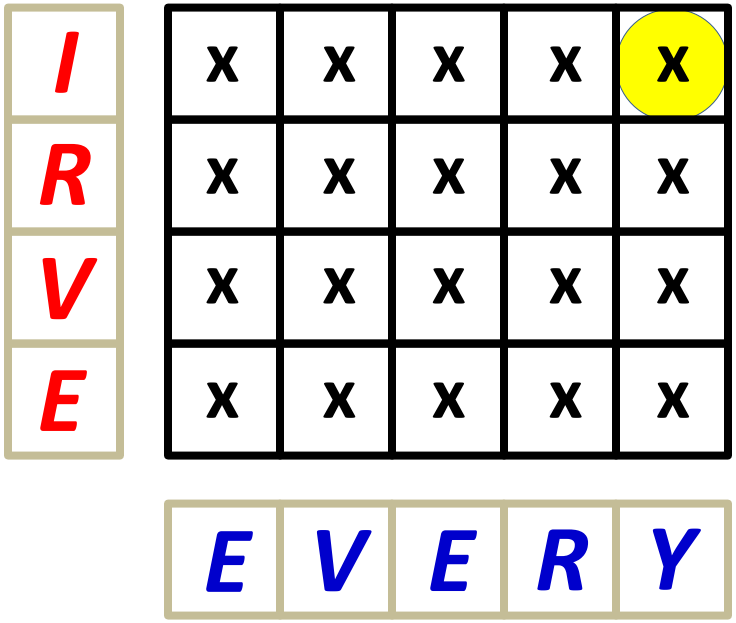

- Incrementally compare substrings
- Each substring-substring comparison depends only on the results of the previous substring comparisons

# *EVERY* vs. *EVRI*




- Incrementally compare substrings
- Each substring-substring comparison depends only on the results of the previous substring comparisons

# *EVERY* vs. *EVRI*

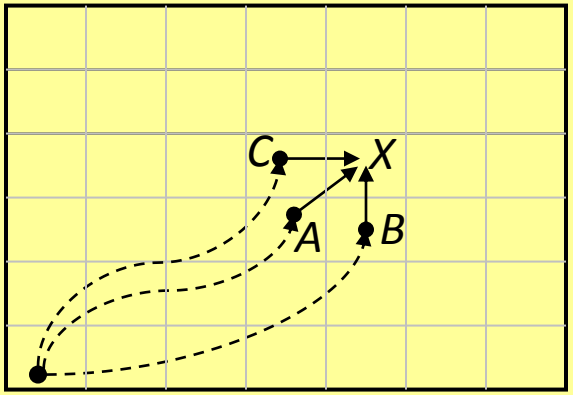



- Incrementally compare substrings
- Each substring-substring comparison depends only on the results of the previous substring comparisons
- **The total computation is < $O(M^2N)$**
  - **M and N are the lengths of the string along the Y and X axes**
  - **In reality, it is closer to O(MN)**

# Minimum String Edit Distance: DP

- In algorithmic terms formulate optimal path to any intermediate point *X* in the matrix *in terms of optimal paths of all its immediate predecessors*
  - Let $M_X$ = Min. path cost from origin to any pt. *X* in matrix
  - Say, *A*, *B* and *C* are all the predecessors of *X*
  - Assume $M_A$, $M_B$ and $M_C$ are known (shown by dotted lines)

  - Then, $M_X$ = min ($M_A$+*AX*, $M_B$+*BX*, $M_C$+*CX*)
    - *AX* = edit distance for diagonal transition
      = 0 if the aligned letters are same, 1 if not)
    - *BX* = edit distance for vertical transition
      = 1 (deletion)
    - *CX* = edit distance for horizontal transition
      = 1 (insertion)



# Minimum String Edit Distance: DP

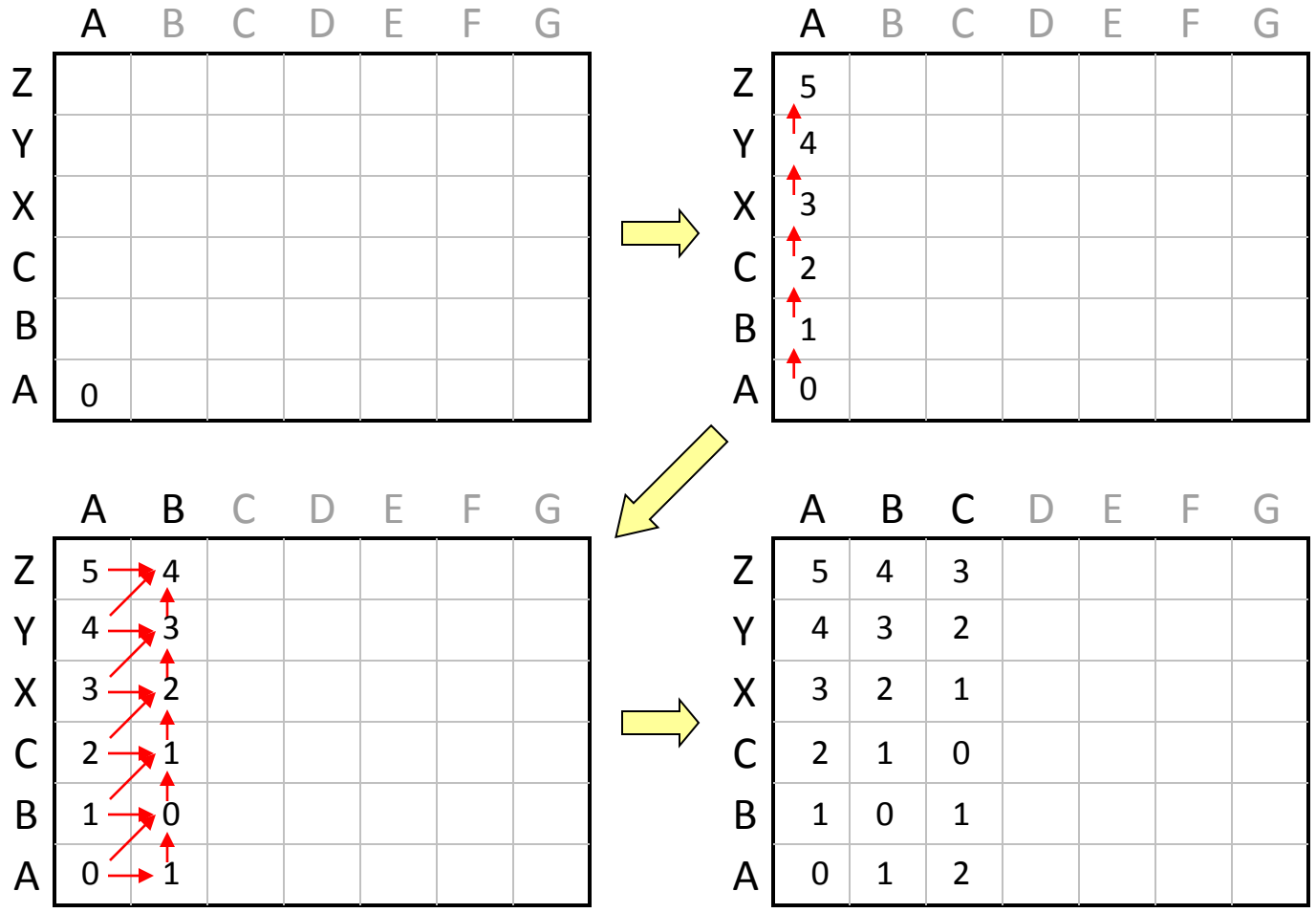
- Hence, start from the origin, and compute min. path cost for every matrix entry, proceeding from bottom-left to top-right corner

- Proceed methodically, one column (*i.e.* one input character) at a time:
  - Consider each input character, one at a time
  - Fill out min. edit distance for that entire column before moving on to next input character
  - Forces us to examine every unit of input (in this case, every character) one at a time
  - Allows each input character to be processed *as it becomes available* ("online" operation possible)

- Min. edit distance = value at top right corner

# DP Example

- First, initialize top left corner, aligning the first letters



# DP Example (contd.)

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Z | 5 | 4 | 3 | 3 | | | |
| Y | 4 | 3 | 2 | 2 | | | |
| X | 3 | 2 | 1 | 1 | | | |
| C | 2 | 1 | 0 | 1 | | | |
| B | 1 | 0 | 1 | 2 | | | |
| A | 0 | 1 | 2 | 3 | | | |

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Z | 5 | 4 | 3 | 3 | 3 | | |
| Y | 4 | 3 | 2 | 2 | 2 | | |
| X | 3 | 2 | 1 | 1 | 2 | | |
| C | 2 | 1 | 0 | 1 | 2 | | |
| B | 1 | 0 | 1 | 2 | 3 | | |
| A | 0 | 1 | 2 | 3 | 4 | | |

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Z | 5 | 4 | 3 | 3 | 3 | 3 | |
| Y | 4 | 3 | 2 | 2 | 2 | 2 | |
| X | 3 | 2 | 1 | 1 | 2 | 2 | |
| C | 2 | 1 | 0 | 1 | 2 | 3 | |
| B | 1 | 0 | 1 | 2 | 3 | 4 | |
| A | 0 | 1 | 2 | 3 | 4 | 5 | |

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Z | 5 | 4 | 3 | 3 | 3 | 3 | 4 |
| Y | 4 | 3 | 2 | 2 | 2 | 2 | 3 |
| X | 3 | 2 | 1 | 1 | 2 | 2 | 3 |
| C | 2 | 1 | 0 | 1 | 2 | 3 | 4 |
| B | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| A | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

- Min. edit distance (ABCXYZ, ABCDEFG) = 4
  - One possible min. distance alignment is shown in blue

# A Little Diversion: Algorithm Bug

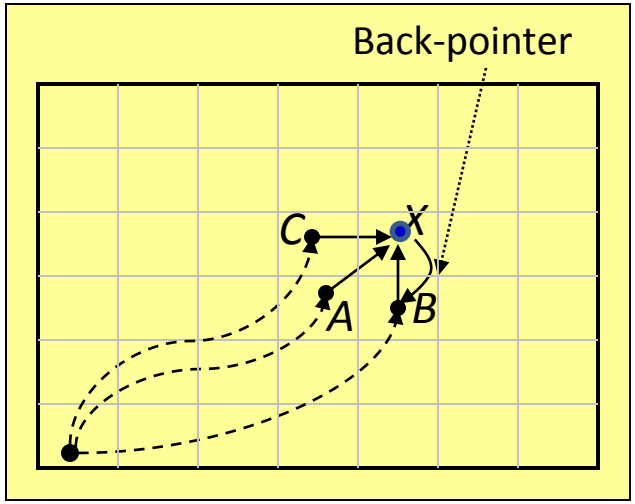
- The above description and example has a small bug. What is it?
- *Hint:* Consider input and template: ***urop*** and ***europe***
  - What is their correct minimum edit distance? (Eyeball and guess!)
  - What does the above algorithm produce?

- *Exercise:* How can the algorithm be modified to fix the bug?

# DP: Finding the Best Alignment

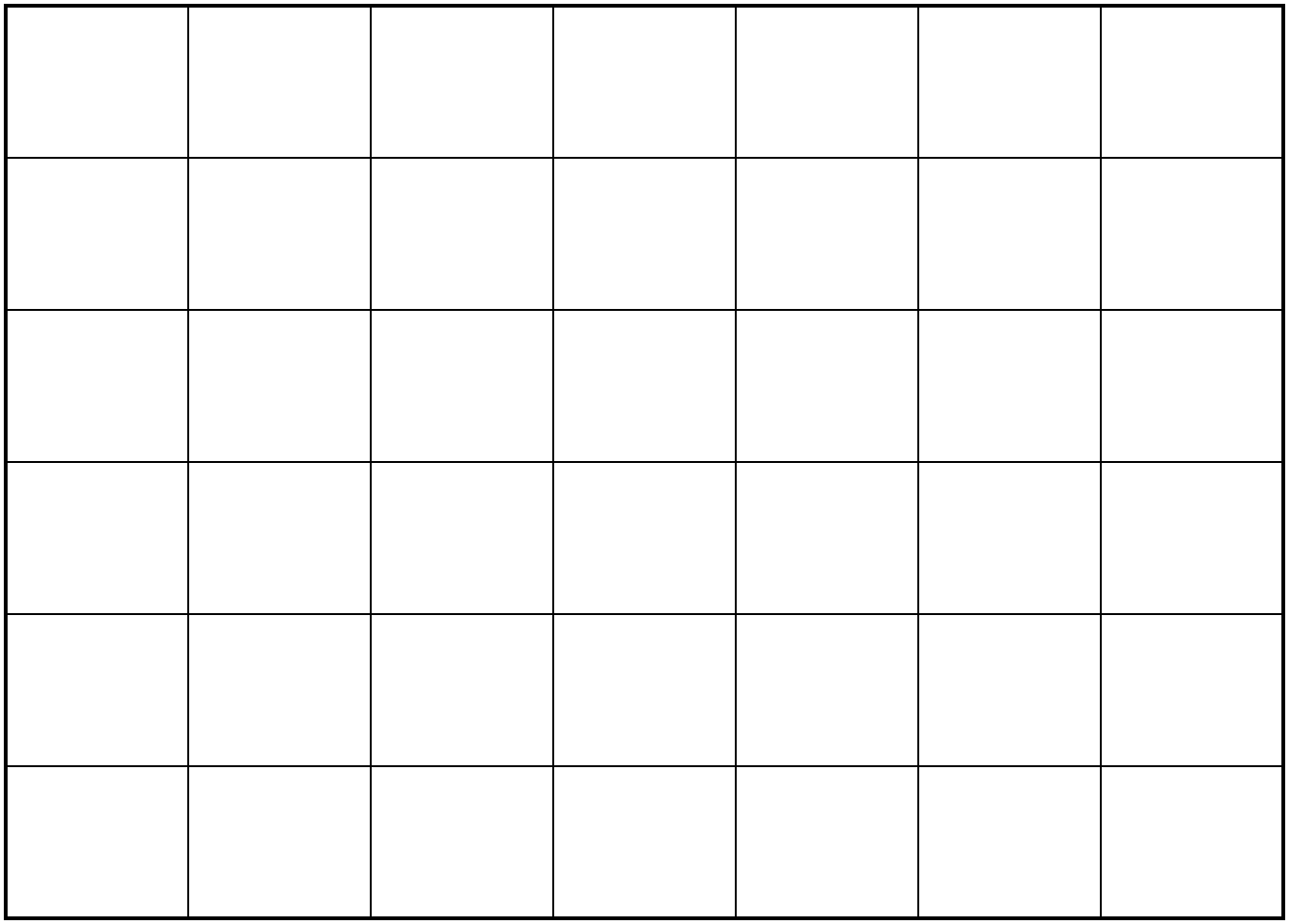
- The algorithm so far only finds the *cost*, not the alignment itself
- How do we find the actual path that minimizes edit distance?
  - There may be multiple such paths, any one path will suffice

- To determine the alignment, we modify the algorithm as follows
- Whenever a cell *X* is filled in, we maintain a ***back-pointer*** from *X* to its predecessor cell that led to the best score for *X*
  - Recall $M_X = \min(M_A+AX, M_B+BX, M_C+CX)$
  - So, if $M_B+BX$ happens to be the minimum we create a back-pointer *X*->*B*
  - If there are ties, break them arbitrarily
- Thus, every cell has a single back-pointer



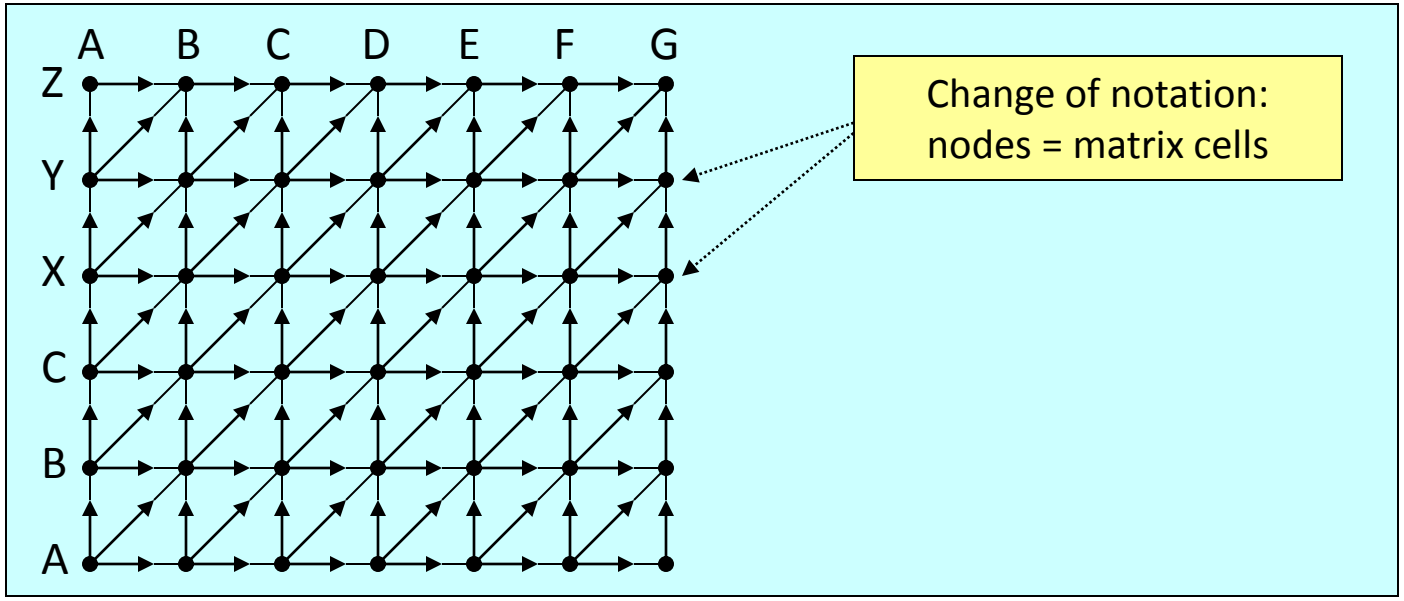

- At the end, we ***trace back*** from the final cell to the origin, using the back-pointers, to obtain the best alignment

# Finding the Best Alignment: Example



# DP Trellis

- The 2-D matrix, with all possible transitions filled in, is called the ***search trellis***
  - Horizontal axis: time. Each step deals with the next input unit (in this case, a text character)
  - Vertical axis: Template (or ***model***)
- Search trellis for the previous example:



# DP Trellis

- DP does *not* require that transitions be limited to the three types used in the example

- The primary requirement is that the optimal path be computable recursively, based on a node's predecessors' optimal sub-paths

# DP Trellis (contd.)

- *The search trellis is arguably one of the most crucial concepts in modern day speech recognizers*!
  - We will encounter this again and again
- Just about any decoding problem is usually cast in terms of such a trellis
- It is then a matter of searching through the trellis for the best path

# Computational Complexity of DP

- Computational cost ~

  No. of nodes x No. of edges entering each node

- For string matching, this is:

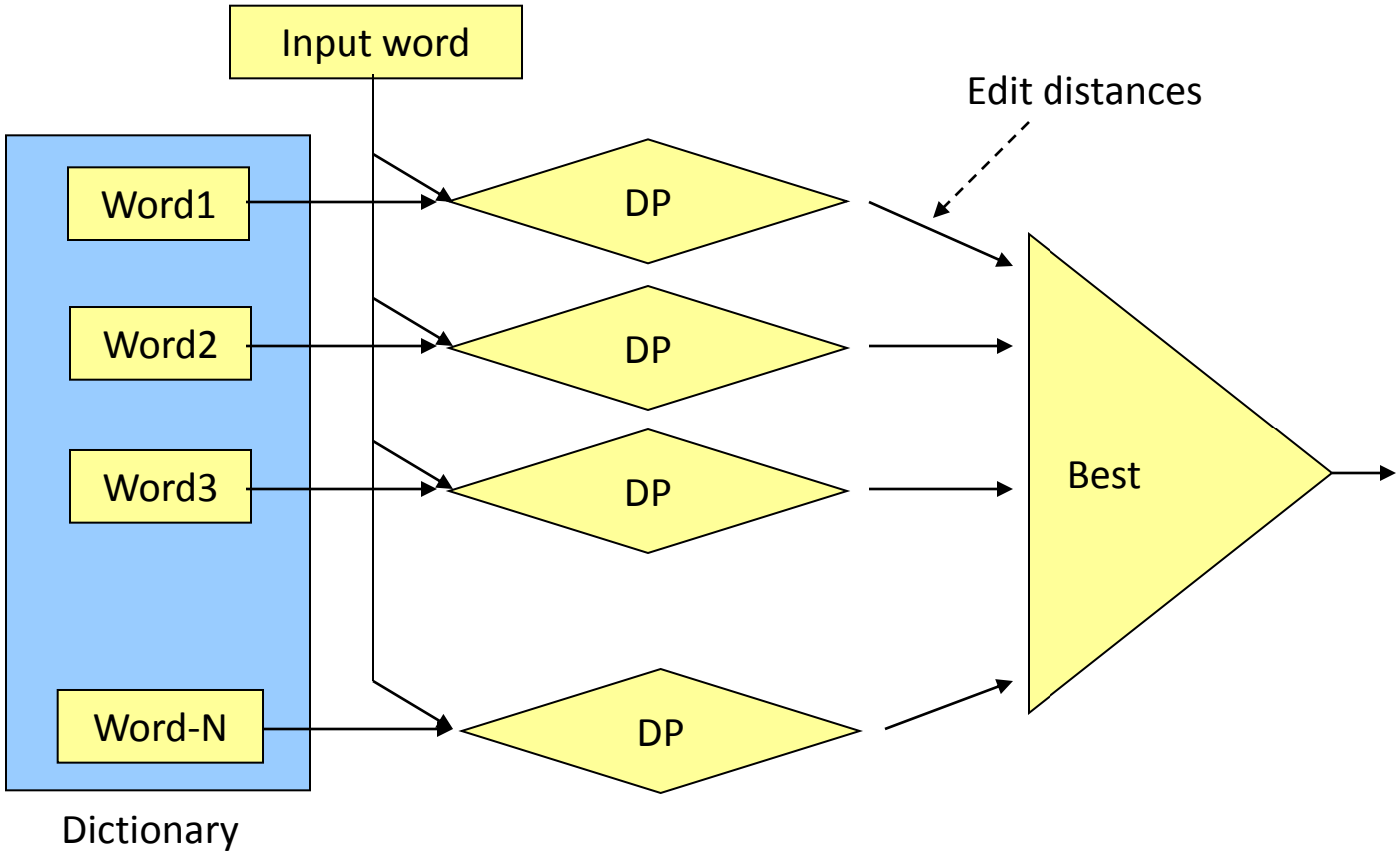
  String-length(template) x String-length(input) x 3

  – (Compare to exponential cost of brute force search!)

- Memory cost for string matching?
  – No of nodes (String-length(template) x String-length(input))?
  – Actually, we don't need to store the entire trellis if all we want is the min. edit distance (*i.e.* not the alignment; no back pointers)
  – Since each column depends only on the previous, we only need storage for 2 columns of the matrix
    - The current column being computed and the previous column

  – Actually, in most cases a column can be updated *in-place*
    - Memory requirement is reduced to just one column

# Back to German -> English




- ***Compare*** box = DP computation of minimum edit distance
- Select the word with the minimum edit distance
  - The "closest" word

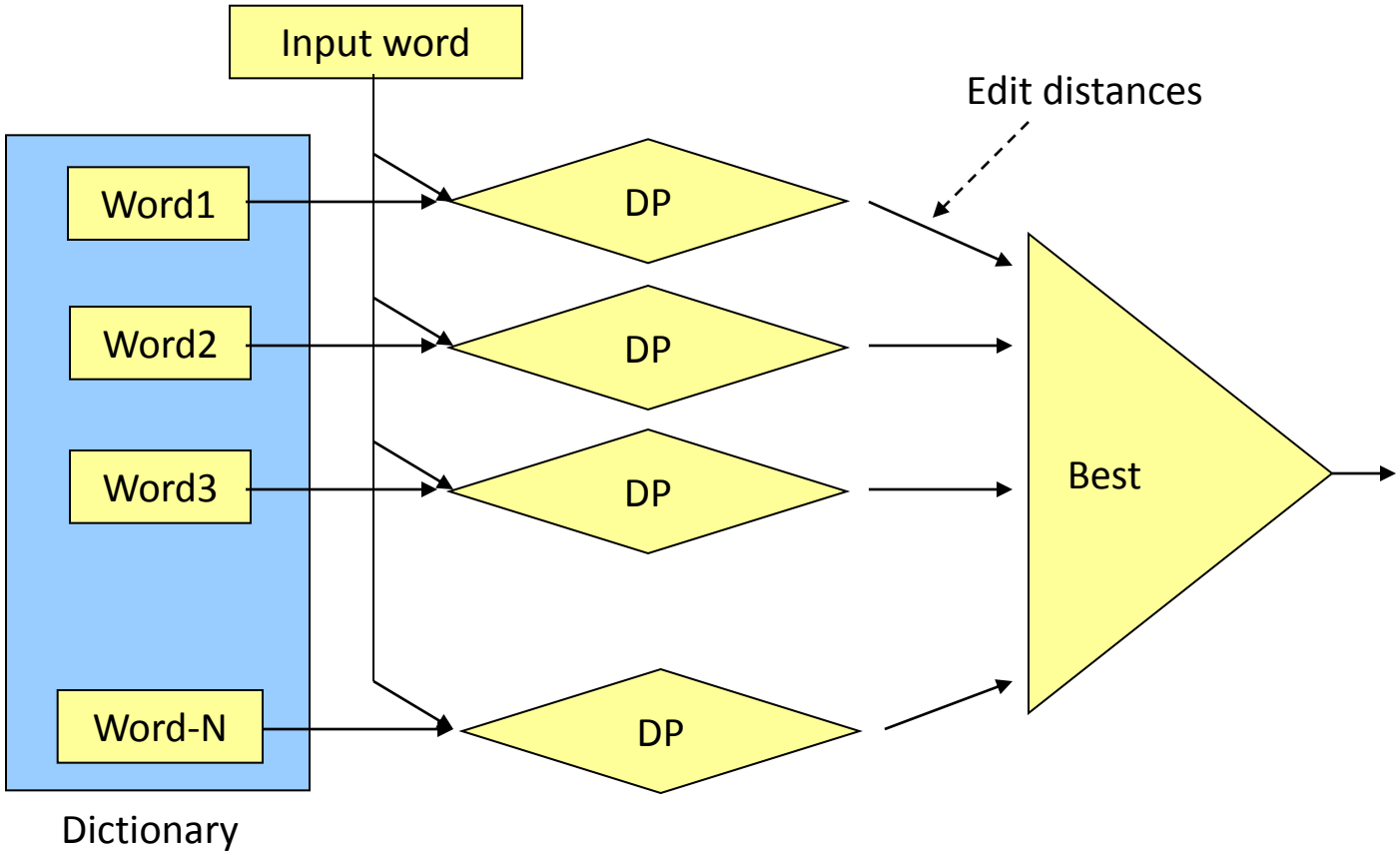
# Rejection

- Spell checker: How to determine if the word is not in the dictionary at all?
  - What we have is a "distance" from an input word to each word in a dictionary
  - We choose the word with the minimum distance
  - How can we be sure that this word is really a match for the misspelled word?

- *Rejection!*
  - If the distance from the closest word is too large, the word is probably not in the dictionary
    - $\text{Min}_{\text{dictionary words}}$(DPScore(input string, Dictionary word)) > Threshold == reject
    - A "rejection threhsold"
  - How do we define "too large"?
    - The maximum acceptable number of errors in a misspelling of "Floccinaucinihilipilification" cannot be compared to the number of errors in "and"
  - The rejection threshold must depend on the length of the word
    - Dictionary word or misspelled input word?

- We return to this in the next class

# Back to German -> English



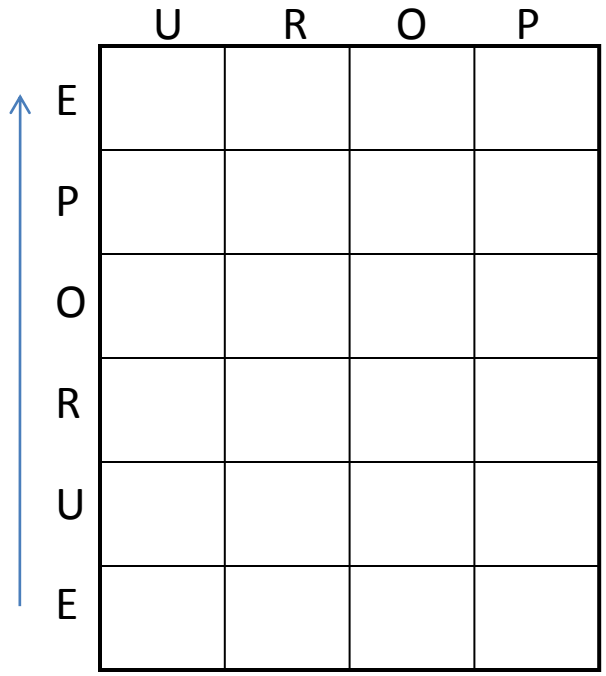

- ***Compare*** box = DP computation of minimum edit distance
- A separate DP trellis for each dictionary word (?)

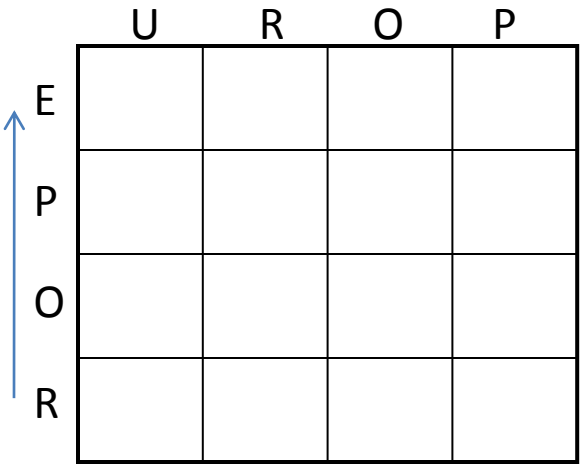
# Computing in Parallel

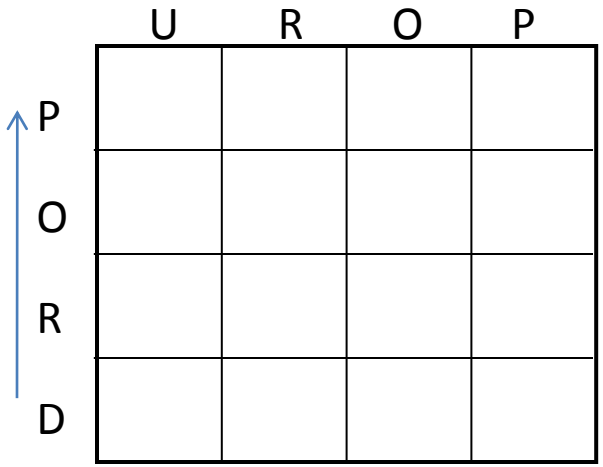
- Spell checker:  Each input word is compared to *every* word in the dictionary
- Create and evaluate V trellises
  - V = size of dictionary
- Select the word with the lowest cost

# Serial Computation

- Example: Comparing “UROP” to “EUROPE”, “ROPE” and “DROP”
  - To determine the closest fit

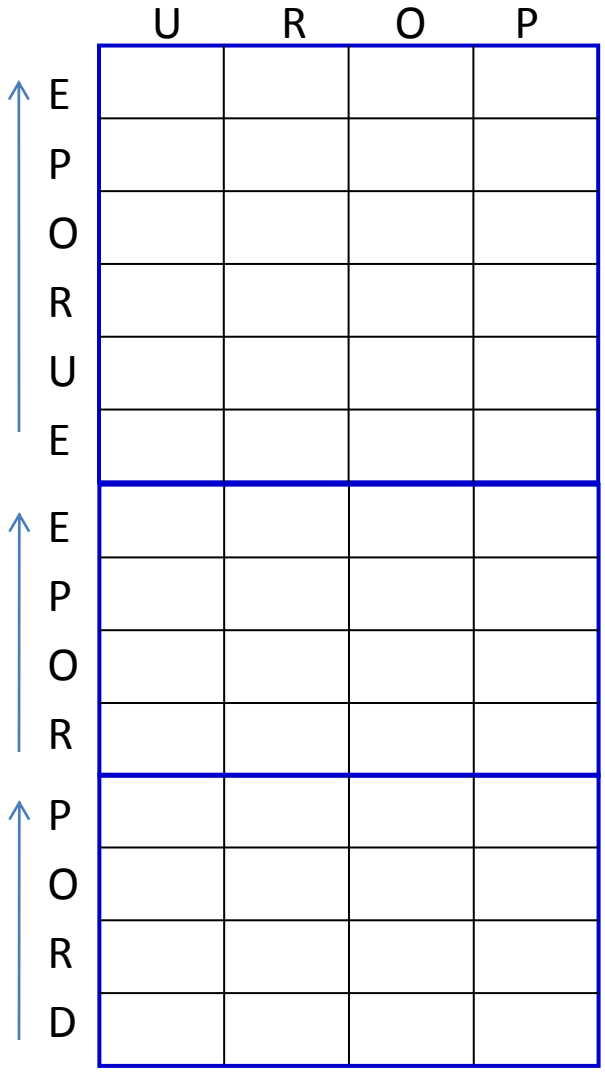



- Three separate trellises computed serially
- Not amenable to online computation

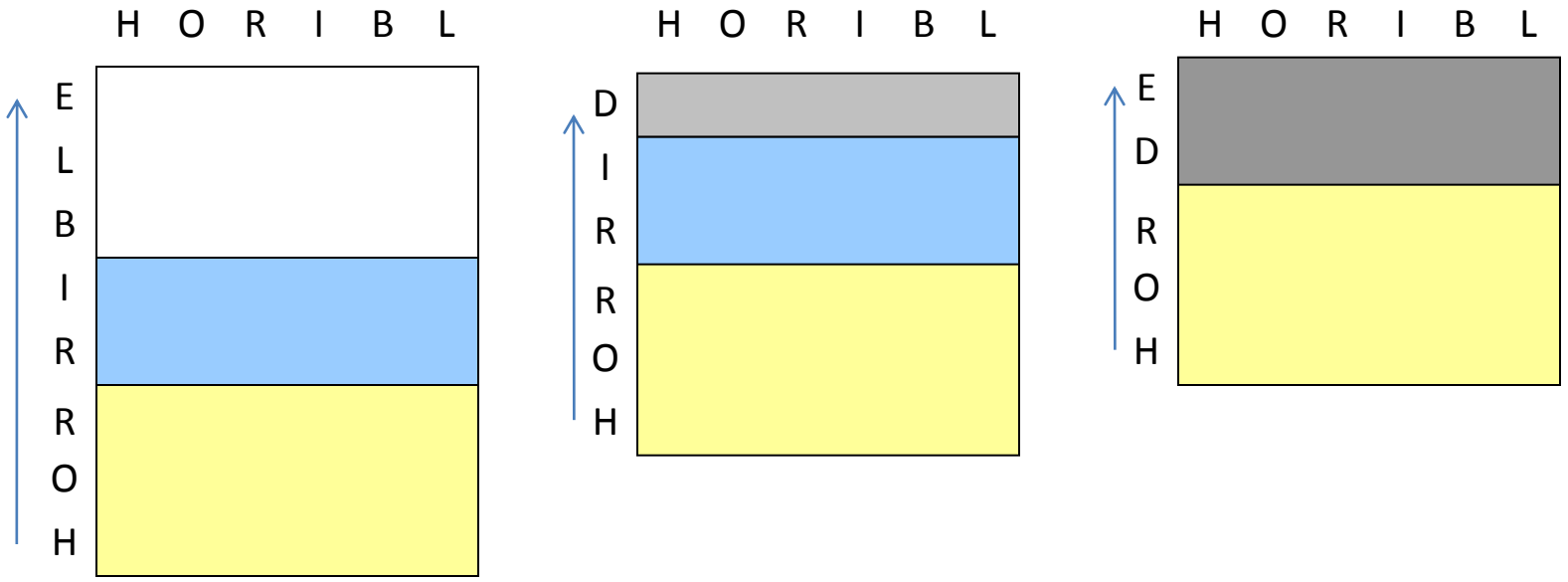
# Parallel Computation




- Compute all Trellises concurrently
  - Each input character is compared to the corresponding column for all templates
- Enables *online* processing
  - The time taken to type in a character is often sufficient to fill in the column for a large dictionary of template words
- Also enables additional processing tricks..

# Optimization: Trellis Sharing

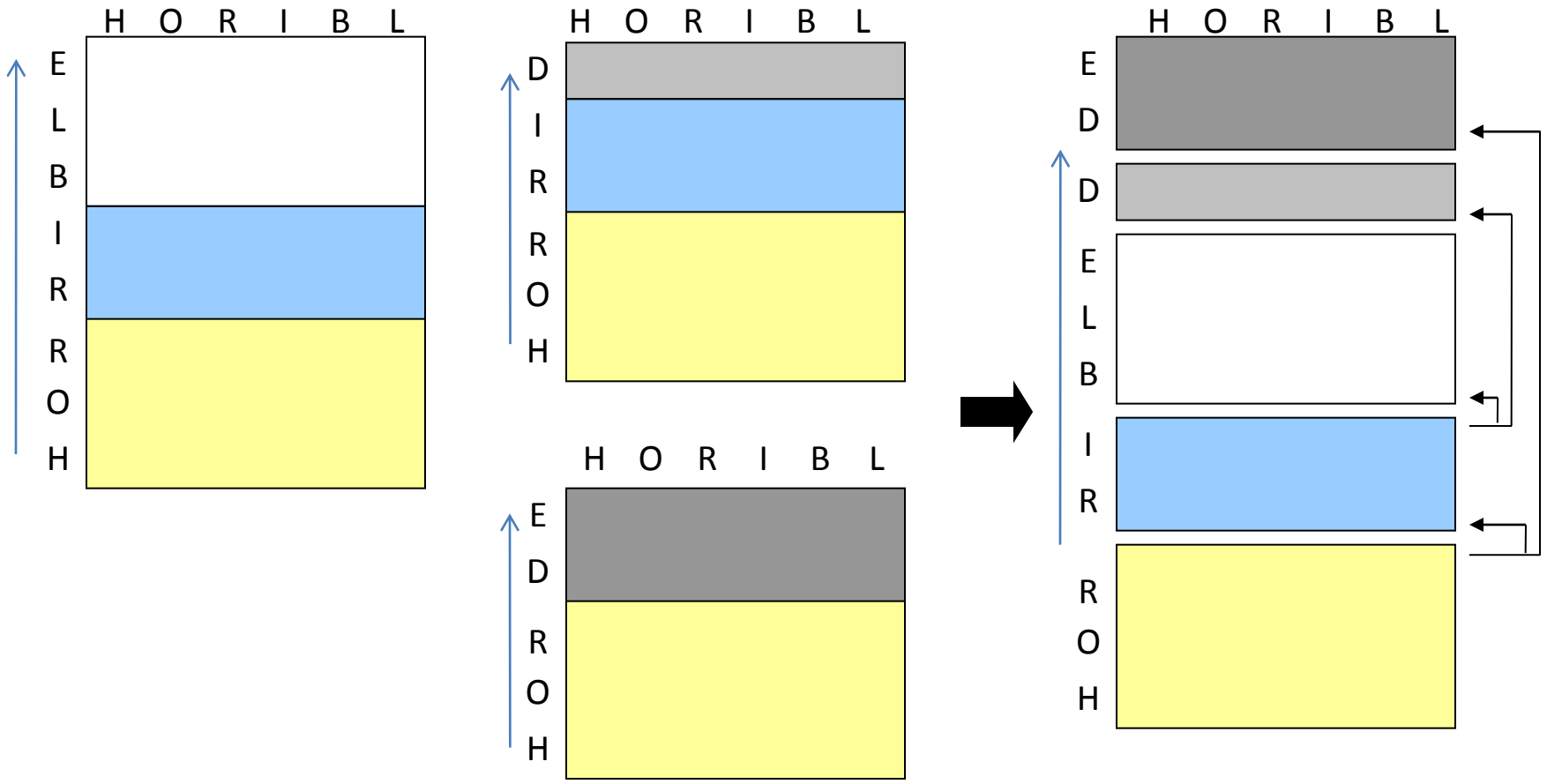
- Consider templates ***horrible***, ***horrid***, ***horde*** being matched with input word ***horibl***




- Trellises shown above
  - Colors indicate identical contents of trellis
- How can we avoid this duplication of computation?

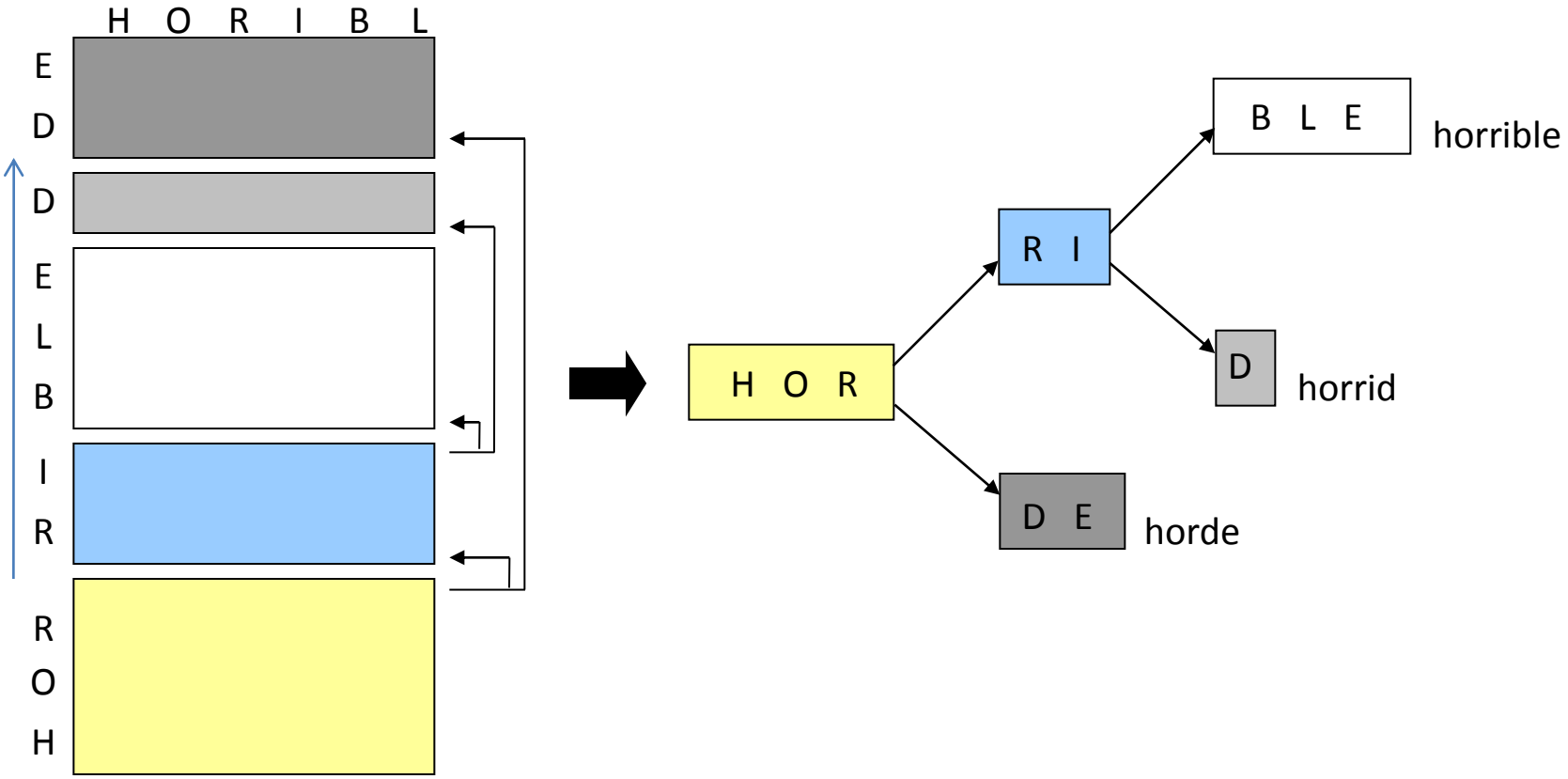
# Optimization: Trellis Sharing

- Compute only the unique subsets (sub-trellises)
- Allow multiple successors from a given sub-trellis



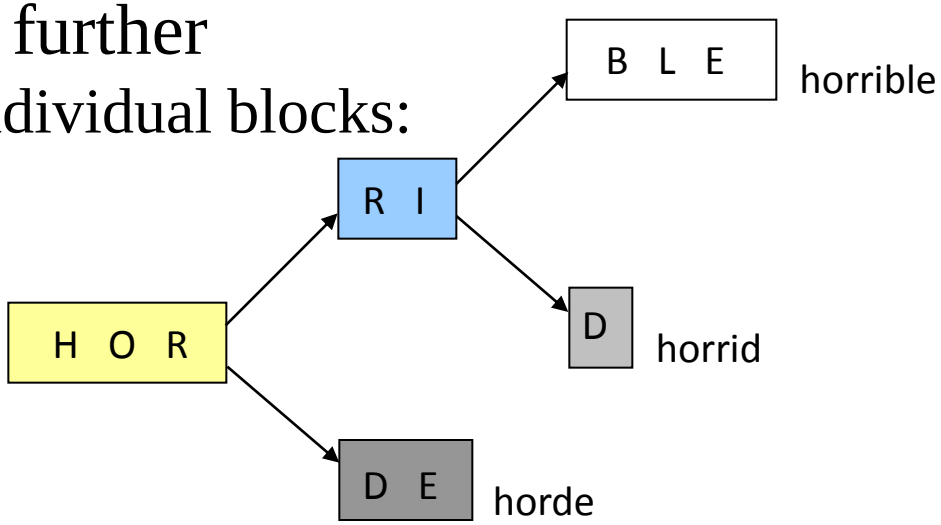
# Trellis Sharing => Template Sharing

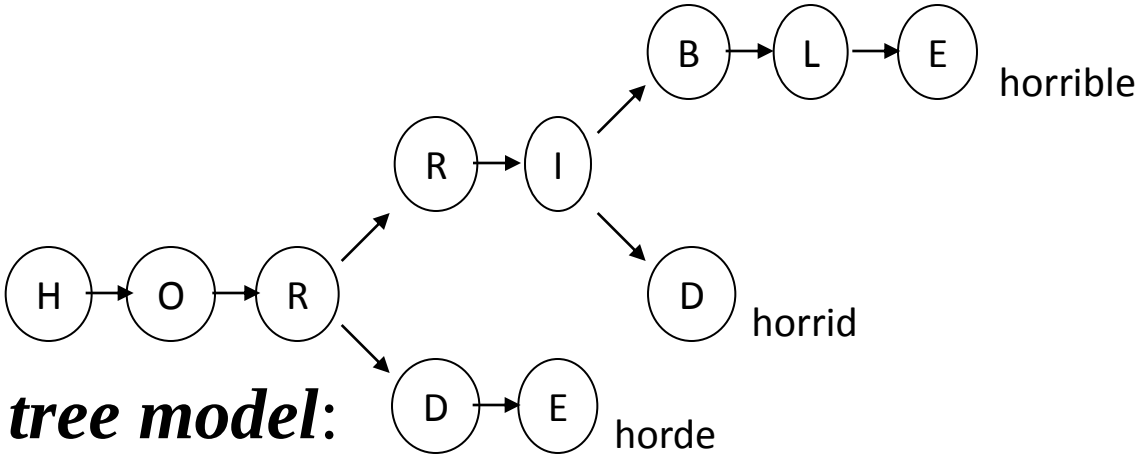
- Notice that templates have become fragmented!
- Derive new template network to facilitate trellis sharing:



# Template Sharing -> Lexical Trees

- Take it one step further
  - Break down individual blocks:

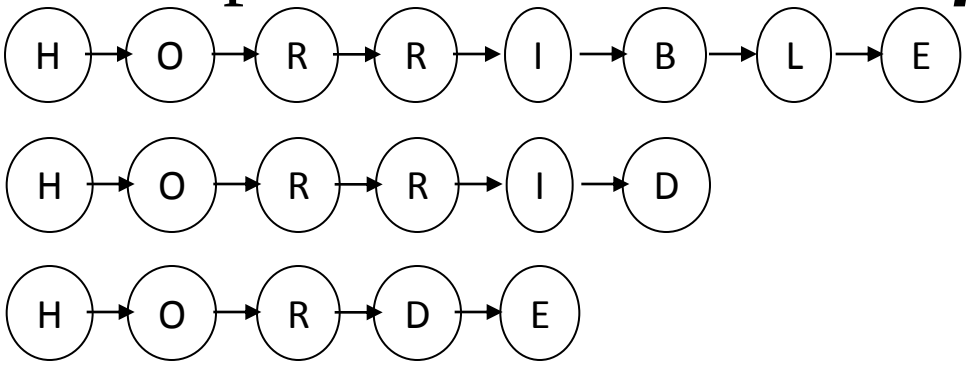



- We get: ***Lexical tree model***:

# Building Lexical Trees

- Original templates were ***linear*** or ***flat*** models:




- *Exercise:* How can we convert this collection to a lexical tree?

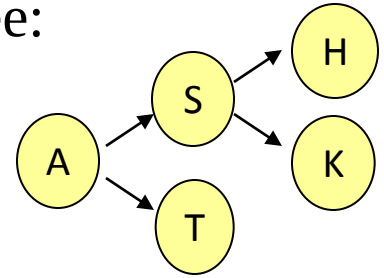
# Trellises for Lexical Trees

- We saw that it is desirable to share sub-trellises, to reduce computation
- We saw the connection between trellis sharing and structuring the templates as lexical trees
- You now (hopefully!) know how to construct lexical trees

- *Q*: Given a lexical tree representing a group of words, what does its search trellis look like?
- *A*:
  - Horizontal axis: time (input characters), as before
  - Vertical axis: nodes in the model (lexical tree nodes)
  - Trellis transitions: nothing but the transitions in the lexical tree, *unrolled over time*
    - We are stepping the model one input unit at a time, and looking at its state at each step

# Trellises for Lexical Trees: Example

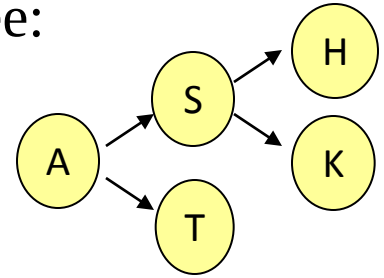
- Simple example of templates: ***at***, ***ash***, ***ask***
  - Lextree:



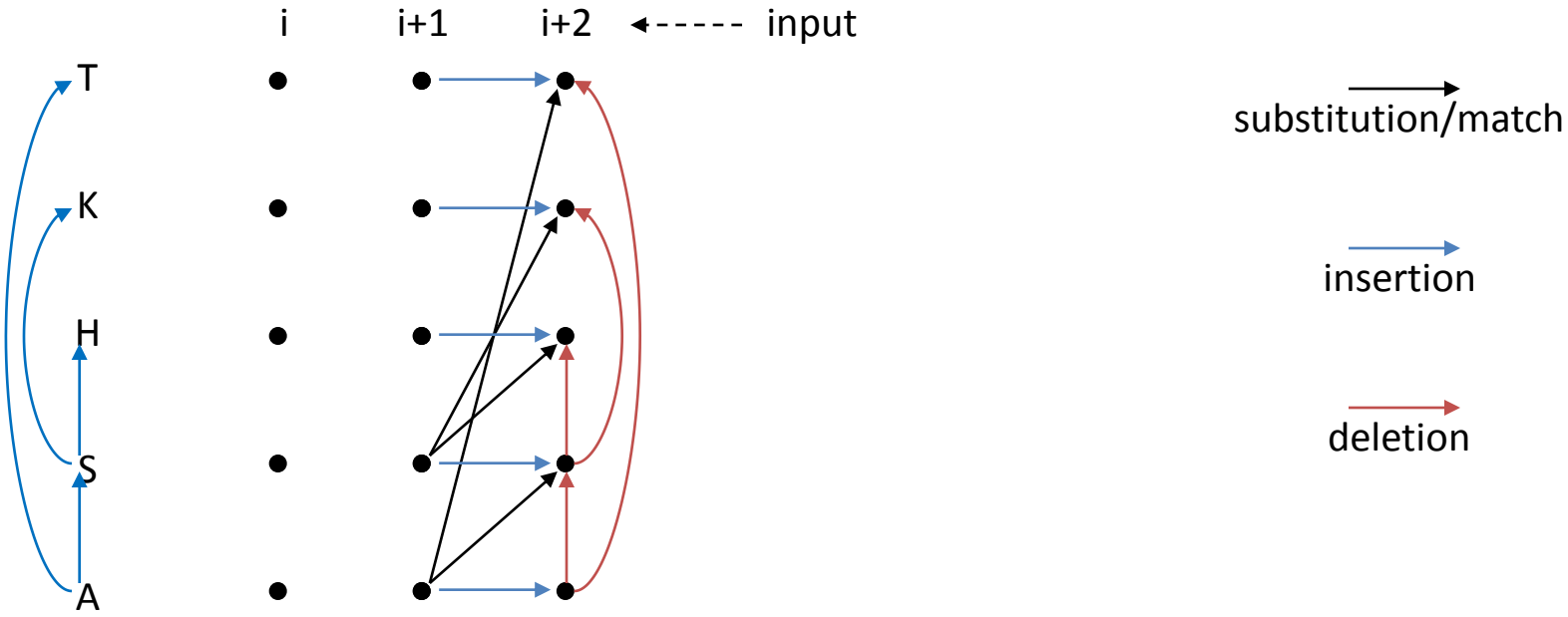

  - Trellis:

# Trellises for Lexical Trees: Example

- Simple example of templates: ***at***, ***ash***, ***ask***
  - Lextree:

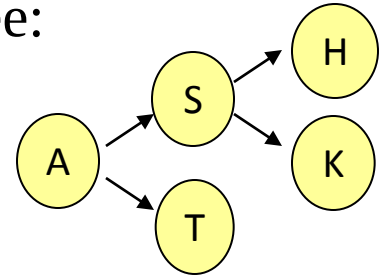



  - Trellis:

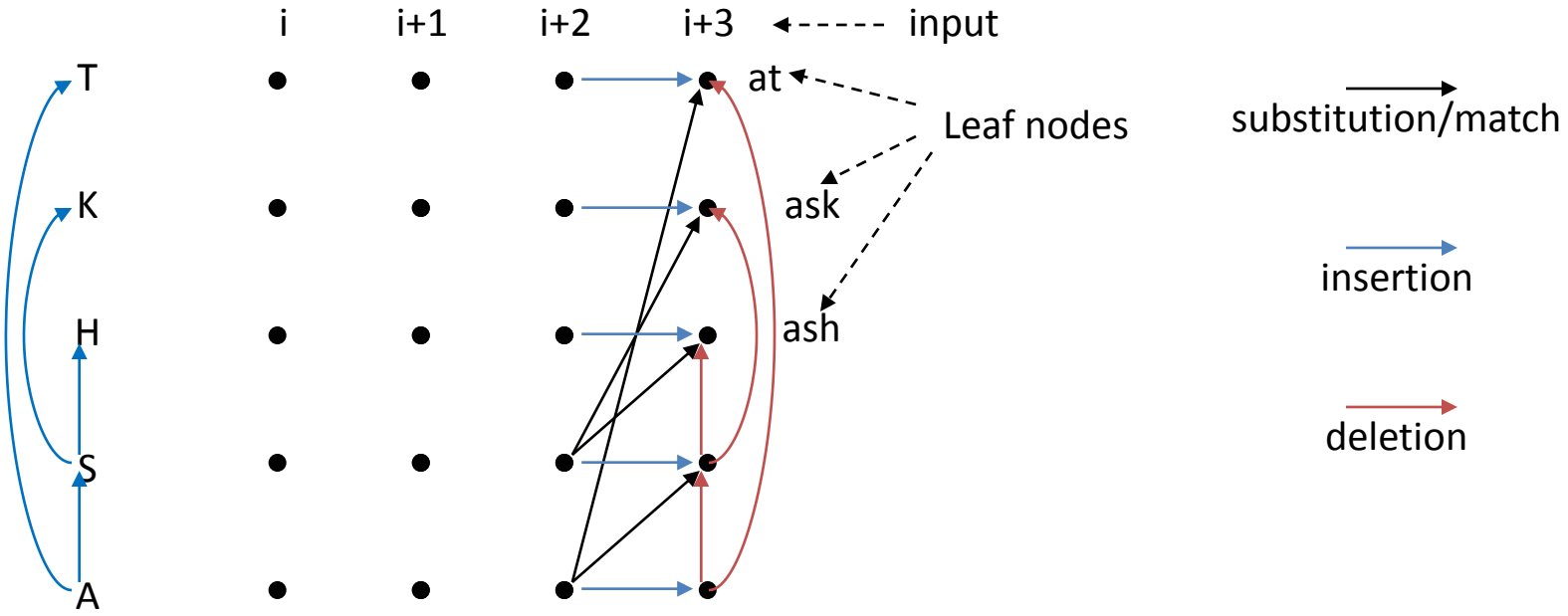


# Trellises for Lexical Trees: Example

- Simple example of templates: ***at***, ***ash***, ***ask***
  - Lextree:



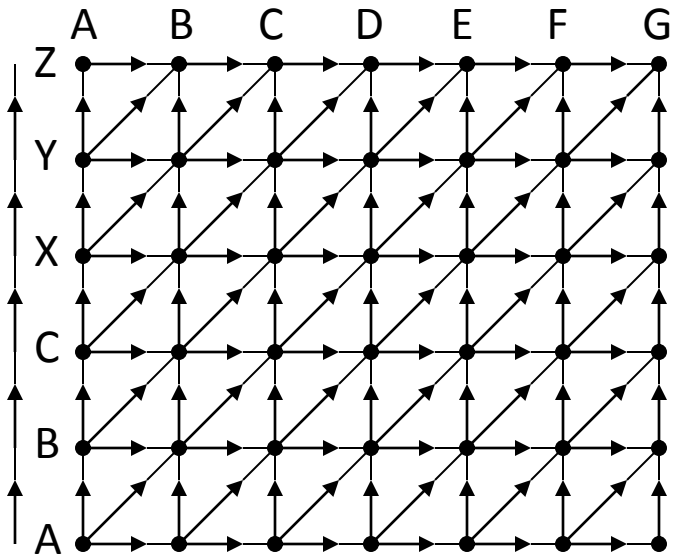

  - Trellis:



# Search Trellis for Graphical Models

- The scheme for constructing trellises from lextree models applies to any graphical model

- Note that the simple trellis shown at the beginning follows directly from this scheme, where the model is a degenerate, linear structure:



# Summary: Elements of the Search Trellis

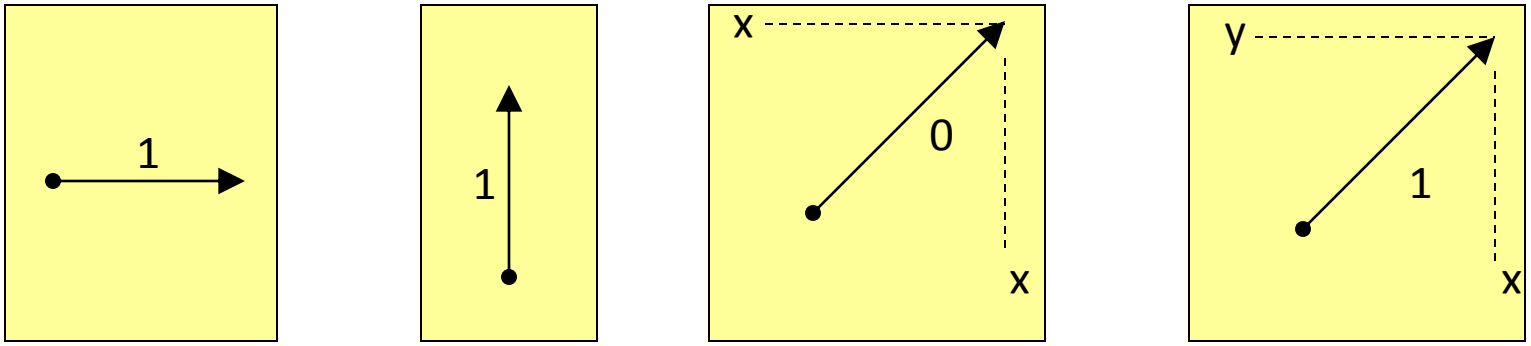
- Nodes represent the cells of the DP matrix
- Edges are the allowed transitions according to some ***model*** of the problem
  - In string matching we allow substitutions, insertions, and deletions
- Every edge optionally has an ***edge cost*** for taking that edge
- Every node optionally has a ***local node cost*** for aligning the particular input entry to the particular template entry
  - The node and edge costs depend on the application and model

- The DP algorithm, at every node, maintains a ***path cost*** for the *best path* from the origin to that node
  - In string matching, this cost is the *substring* minimum edit distance
  - Path costs are computed by accumulating local node and edge costs according to the recursive formulation already seen (minimizing cost)

- One may also use a *similarity* measure, instead of *dissimilarity*
  - In this case DP algorithm should try to *maximize* the total path score

# Edge and Node Costs for String Match

- Edge costs:




insertion deletion correct substitution

- Local node costs: None

# Reducing Search Cost: Pruning

- Reducing search cost implies reducing the size of the trellis/lattice that has to be evaluated

- There are several ways to accomplish this
  - Reducing the complexity and size of the models (templates)
    - *E.g.* using lextrees (and thereby sharing trellis computation)
    - We have already seen this above
  - Eliminating parts of the lattice from consideration altogether
    - This approach is called ***search pruning***, or just ***pruning***

- Basic consideration in pruning: *As long as the best cost path is not eliminated by pruning, we obtain the same result*

# Pruning

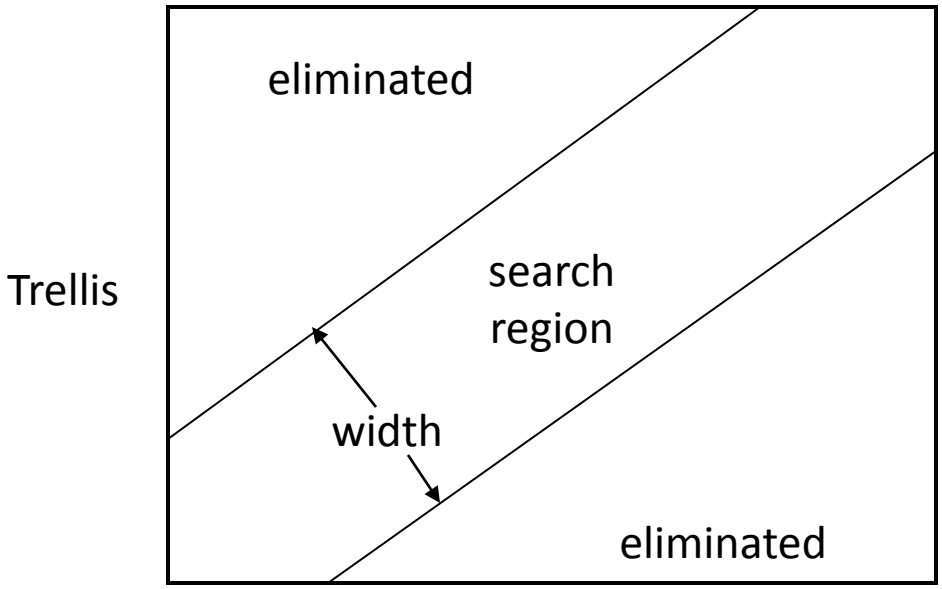
- Pruning is a ***heuristic***: typically, there is a ***threshold*** on some measured quantity, and anything above or below the threshold is eliminated

- It is all about choosing the right measure, and the right threshold

- Let us see two different pruning methods:
  - Based on deviation from the diagonal path in the trellis
  - Based on path costs

# Pruning by Limiting Search Paths

- Assume that the the input and the *best matching* template do not differ significantly from each other
  - The best path matching the two will lie close to the "diagonal"
- Thus, we need not search far off the diagonal. If the search-space "width" is kept constant, cost of search is linear in utterance length instead of quadratic



# Pruning by Limiting Search Paths

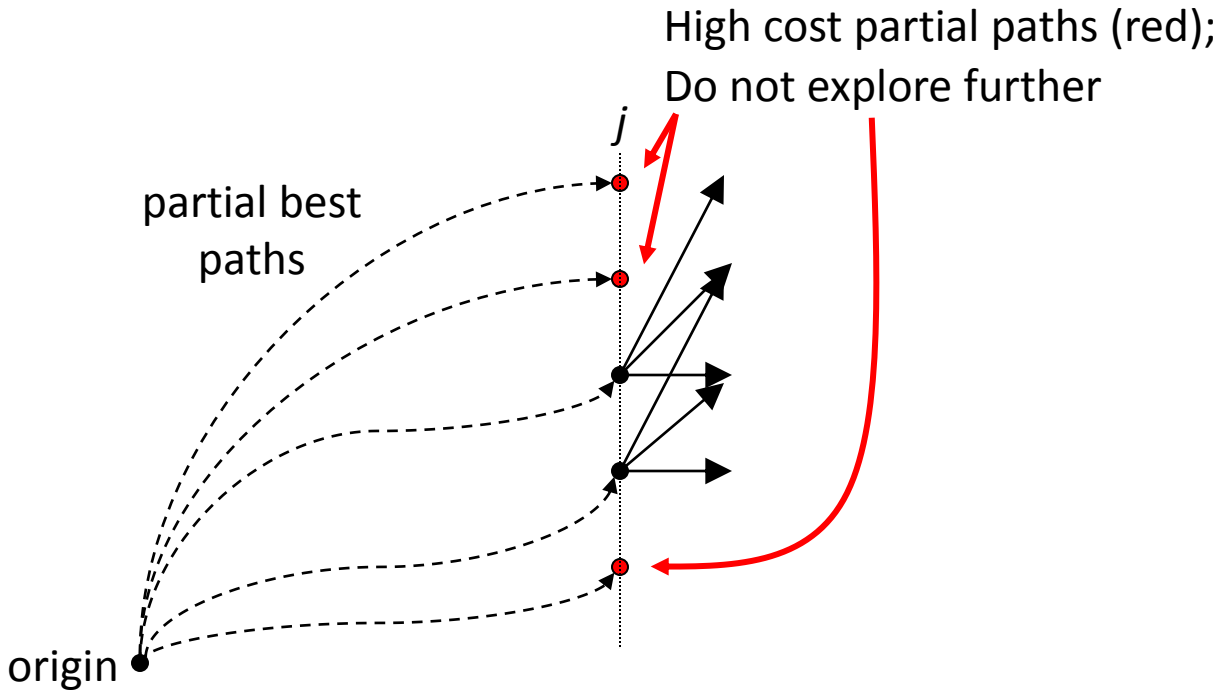
- What are problems with this approach?

# Pruning by Limiting Search Paths

- What are problems with this approach?
  - With lexical tree models, the notion of "diagonal" becomes difficult

# Option 2: Pruning by Limiting Path Cost

- *Observation*: Partial paths that have "very high" costs will rarely recover to win
- Hence, poor partial paths can be eliminated from the search:
  - For each column *j*, after computing all the trellis nodes path costs, determine which nodes have too high costs
  - Eliminate them from further exploration
- *Q*: How do we define "high cost"?



# Pruning by Limiting Path Cost

- One *could* define high path cost as a value worse than some fixed threshold

- Will this work?

# Pruning by Limiting Path Cost

- One *could* define high path cost as a value worse than some fixed threshold

- Problem: Absolute path cost increases monotonically with input length!
  - Thresholds have to be loose enough to allow for the longest inputs
  - But such thresholds will be too permissive at shorter lengths, and not constrain computation effectively

- Solution: Look at *relative* path cost instead of *absolute* path cost

# Pruning: Beam Search

- *Solution*: At each time step *j*, set the pruning threshold by a fixed amount *T relative to the best cost* at that column (input symbol)
  - *I.e.* if the best partial path cost achieved at column *t* is *X*, prune away all nodes with partial path cost > $X+T$ before moving to time $t+1$
- Advantages:
  - Unreliability of absolute path costs is eliminated
  - Monotonic growth of path costs with time is also irrelevant
- Search that uses such pruning is called ***beam search***
  - This is the most widely used search optimization strategy
- The relative threshold *T* is usually called ***beam width*** or just ***beam***

# Determining the Optimal Beam Width

- Determining the *optimal* beam width to use is crucial
  - Too *narrow* or *tight* a beam (too low *T*) can prune the best path
    - And result in too high a match cost, and errors
  - Too large (wide) a beam results in unnecessary computation
    - From searching unlikely paths

- *Unfortunately, there is no mathematical solution to determining an optimal beam width*
- Common method: Try a wide range of beams on some test data until the desired operating point is found
  - The operating point may be determined by some combination of matching /recognition accuracy and computational efficiency

# Conclusion

- Minimum string edit distance
- Dynamic programming search to compute minimum edit distance
- Lextree construction for compact templates
- Graphical representations of models
- Search trellis construction for given graphical models
- Search pruning

- Next up: Application to speech:
  - All concepts learned with strings apply to speech recognition!

# Assignment 2

- On website..