Spell Checking

Thomas Graf

Stony Brook University
lin220@thomasgraf.net

LIN 220, Spring 2019
Lecture 3

Where we are so far...

- n-grams for word completions
- use frequencies to pick most likely completion
- prefix trees as efficient storage and search
- It's not linguistically perfect,
  but it does well enough.

One big problem

Where we are so far...

- n-grams for word completions
- use frequencies to pick most likely completion
- prefix trees as efficient storage and search
- It's not linguistically perfect,
  but it does well enough.

One big problem

What if the user made a typo?

Spell checking: A naive solution

- word list (e.g. stored as prefix tree)
- spell checking = lookup in word list
  - word found ⇒ spelled correctly
  - word not found ⇒ spelled incorrectly
- But this simple model is **not good enough**.

Open issues

- How do we determine the correct spelling of a mistyped word?
- Not all misspellings are easy to detect.
- What if a correctly spelled word is not in the dictionary?
  specialized terminology, proper names, neologisms, slang, loan words, ...

Spell checking: A naive solution

- word list (e.g. stored as prefix tree)
- spell checking = lookup in word list
  - word found ⇒ spelled correctly
  - word not found ⇒ spelled incorrectly
- But this simple model is **not good enough**.

Open issues

- How do we determine the correct spelling of a mistyped word?
- Not all misspellings are easy to detect.
- What if a correctly spelled word is not in the dictionary?
  specialized terminology, proper names, neologisms, slang, loan words, ...

Assessing the problem

- Never type a single line of code
  before you understand the problem!
- Think about the **parameters of the problem**.
- Solving the wrong problem is pointless.

Parameters of the spell checking problem

- How is the spell checker to be used?
  automatic/auto correction VS interactive/suggestions to user
- What types of misspellings are there?
- Is it feasible to detect all of them?
- Once we know what the tool should handle,
  what is the simplest solution?

Assessing the problem

- Never type a single line of code
  before you understand the problem!
- Think about the **parameters of the problem**.
- Solving the wrong problem is pointless.

Parameters of the spell checking problem

- How is the spell checker to be used?
  automatic/auto correction VS interactive/suggestions to user
- What types of misspellings are there?
- Is it feasible to detect all of them?
- Once we know what the tool should handle,
  what is the simplest solution?

A typology of spelling mistakes

- **Cause**
  accidental typo ⇔ unawareness of correct spelling
- **Number**
  single-error ⇔ multi-error
- **Error type**

split	illicit space
	quin tuplets, the set up, atoll way
run-on	missing space
	nightvision, boothbabe, atoll way
non-word	typed word does not exist
	warte or tawer for water
real-word	misspelling yields existing word(s) of English
	car toon, it's VS its, their VS there, book for brook, atoll way

A difficulty hierarchy of tasks

- Both typos and spelling confusion can be very hard.
  - *labelled* for *labeled*: easy
  - *awter* for *water*: easy
  - *nitch* for *niche*: tricky
  - *awre* for *water*: tricky
- Single-error $<$ multi-error
- Non-word errors $<$ real-word errors
- Difficulty of real-word errors scales with complexity of context:

1. local syntactic configuration
2. non-local syntactic configuration
3. word meaning
4. discourse/cross-sentence
5. world knowledge

Examples of increasing context complexity

- **Local syntactic configuration**
  Their are some biscuits on the counter.
- **Non-local syntactic configuration**
  The man sitting at the bar seem to be enjoying the atmosphere.
- **Word meaning**
  We still have to pay off the mortgage on our mouse.
- **Discourse**
  It's like that time they canceled Futurama. I was so bad.
- **World knowledge**
  This course is taught at Stony Book University.

Proper names are impossibly hard

Proper names are impossibly hard

Xexyz

Proper names are impossibly hard

Xexyz

Mister Mxyzptlk

How much can we handle efficiently?

- Always remember:
  meaning is hard, world knowledge nigh impossible.
- Even non-local syntactic configurations are difficult.
- So we consider only models that handle at most
  local syntactic configurations.

n-Grams handle local context

- n-gram models can easily detect local real-word errors.

Example

- English sentences rarely contain the bigram *their are*.
- Hence instances of *their are* are misspellings.

Problems:

- False positive: incorrectly flags correct words if they're not in our word list
- Sparse data problem all over again.
- How do we go from detecting likely errors to finding likely corrections?
- For automatic spell checker, what about cases like *the man are*?

  change *man* to *men* VS change *are* to *is*

n-Grams handle local context

- n-gram models can easily detect local real-word errors.

Example

- English sentences rarely contain the bigram *their are*.
- Hence instances of *their are* are misspellings.

Problems:

- False positive: incorrectly flags correct words if they're not in our word list
- Sparse data problem all over again.
- How do we go from detecting likely errors to finding likely corrections?
- For automatic spell checker, what about cases like *the man are*?

  change *man* to *men* VS change *are* to *is*

Unlisted words are inevitable

- No word list can ever contain all words of English.
- It is also **undesirable**:
  - No speaker knows or uses all words of English.
  - Suppose John knows only 10% of the words in our list.
    Then John will make non-word errors that are real-word errors for the model.
  - **Bottom line**
    A big word list **makes finding misspellings harder**.

Example

- Computer scientists use the special term *memoize*.
- But for most people *memoize* is a misspelling of *memorize*.
- If the dictionary contains *memoize*,
  then the model will perform **worse** for the majority of users.

Unlisted words are inevitable

- No word list can ever contain all words of English.
- It is also **undesirable**:
  - No speaker knows or uses all words of English.
  - Suppose John knows only 10% of the words in our list.
    Then John will make non-word errors that are real-word errors for the model.
  - **Bottom line**
    A big word list **makes finding misspellings harder**.

Example

- Computer scientists use the special term *memoize*.
- But for most people *memoize* is a misspelling of *memorize*.
- If the dictionary contains *memoize*,
  then the model will perform **worse** for the majority of users.

What does "hard" mean?

- A problem is hard if it is difficult to design a model that performs well on the task.
- But what does it mean to perform well? Detecting both **positives** and **negatives**!

		Model says:	
		bad	**good**
Spelling is:	**bad**	true positive	false negative
	good	false positive	true negative

- Like in medical tests, **positive** does not mean "good".
- For spellchecking: **positive** = is a misspelled word

Precision and Recall

There are two measures of model performance:

Precision How many posited positives are actual positives?

Recall How many of the actual positives are recognized as positives?

Formal definition

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}} = \frac{\text{true positives}}{\text{all posited positives}}$$

$$\text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} = \frac{\text{true positives}}{\text{all actual positives}}$$

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	**good**
bad	25	10
good	30	35

1. True positives:
2. True negatives:
3. False positives:
4. False negatives:

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	good
bad	25	10
good	30	35

1. True positives: 25
2. True negatives:
3. False positives:
4. False negatives:

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	**good**
bad	25	10
good	30	35

1. True positives: 25
2. True negatives: 35
3. False positives:
4. False negatives:

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	good
bad	25	10
good	30	35

1. True positives: 25
2. True negatives: 35
3. False positives: 30
4. False negatives:

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	**good**
bad	25	10
good	30	35

1. True positives: 25
2. True negatives: 35
3. False positives: 30
4. False negatives: 10

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	good
bad	25	10
good	30	35

1. True positives: 25
2. True negatives: 35
3. False positives: 30
4. False negatives: 10

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	good
bad	25	10
good	30	35

1. True positives: 25
2. True negatives: 35
3. False positives: 30
4. False negatives: 10

$$\text{Precision} = \frac{25}{\text{true positives} + \text{false positives}}$$

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	**good**
bad	25	10
good	30	35

1. True positives: 25
2. True negatives: 35
3. False positives: 30
4. False negatives: 10

$$\text{Precision} = \frac{25}{25 + \text{false positives}}$$

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	**good**
bad	25	10
good	30	35

1. True positives: 25
2. True negatives: 35
3. False positives: 30
4. False negatives: 10

$$\text{Precision} = \frac{25}{25 + 30}$$

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	**good**
bad	25	10
good	30	35

1. True positives: 25
2. True negatives: 35
3. False positives: 30
4. False negatives: 10

$$\text{Precision} = \frac{25}{25 + 30} = \frac{25}{55} = 0.45 = 45\%$$

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	**good**
bad	25	10
good	30	35

1. True positives: 25
2. True negatives: 35
3. False positives: 30
4. False negatives: 10

$$\text{Precision} = \frac{25}{25 + 30} = \frac{25}{55} = 0.45 = 45\%$$

$$\text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	**good**
bad	25	10
good	30	35

1. True positives: 25
2. True negatives: 35
3. False positives: 30
4. False negatives: 10

$$\text{Precision} = \frac{25}{25 + 30} = \frac{25}{55} = 0.45 = 45\%$$

$$\text{Recall} = \frac{25}{\text{true positives} + \text{false negatives}}$$

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	**good**
bad	25	10
good	30	35

1. True positives: 25
2. True negatives: 35
3. False positives: 30
4. False negatives: 10

$$\text{Precision} = \frac{25}{25 + 30} = \frac{25}{55} = 0.45 = 45\%$$

$$\text{Recall} = \frac{25}{25 + \text{false negatives}}$$

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	good
bad	25	10
good	30	35

1. True positives: 25
2. True negatives: 35
3. False positives: 30
4. False negatives: 10

$$\text{Precision} = \frac{25}{25 + 30} = \frac{25}{55} = 0.45 = 45\%$$

$$\text{Recall} = \frac{25}{25 + 10}$$

A calculation example

Our spellchecker performs as follows over 100 words:

	bad	good
bad	25	10
good	30	35

1. True positives: 25
2. True negatives: 35
3. False positives: 30
4. False negatives: 10

$$\text{Precision} = \frac{25}{25 + 30} = \frac{25}{55} = 0.45 = 45\%$$

$$\text{Recall} = \frac{25}{25 + 10} = \frac{25}{35} = 0.71 = 71\%$$

The abstract principle

Example

A large word list

precision:

recall:

Precision and recall are quantitative counterparts to soundness and completeness:

sound	If the model says X is a positive, then X is a positive.
complete	If X is a positive, then the model says X is a positive.

The abstract principle

Example

A large word list

- increases precision:

  recall:

Precision and recall are quantitative counterparts to soundness and completeness:

sound If the model says X is a positive, then X is a positive.

complete If X is a positive, then the model says X is a positive.

The abstract principle

Example

A large word list

- increases precision: correct spellings are less likely to be flagged as incorrect

  recall:

Precision and recall are quantitative counterparts to soundness and completeness:

sound If the model says X is a positive, then X is a positive.

complete If X is a positive, then the model says X is a positive.

The abstract principle

Example

A large word list

- increases precision: correct spellings are less likely to be flagged as incorrect
- decreases recall:

Precision and recall are quantitative counterparts to soundness and completeness:

sound	If the model says X is a positive, then X is a positive.
complete	If X is a positive, then the model says X is a positive.

The abstract principle

Example

A large word list

- increases precision: correct spellings are less likely to be flagged as incorrect
- decreases recall: non-word errors by the user are incorrectly treated as correct spellings

Precision and recall are quantitative counterparts to soundness and completeness:

sound If the model says X is a positive, then X is a positive.

complete If X is a positive, then the model says X is a positive.

“I still don’t get it!”

Here’s the simplistic version:

low precision
: many actual negatives are misclassified as positives; the model is too eager to find positives

low recall
: many actual positives are misclassified as negatives; the model misses too many positives

Interim summary

Precision and recall

- "Performing well" is too vague a notion.
- In order to evaluate models, we need more rigorous metrics.
- Precision and recall allow us to quantify performance along two important axes.

Spelling

- We want to handle at least non-word errors.
- We want to handle at most local syntactic real-word errors.
- We want to be able to detect and suggest corrections.
- A word list of all English words is impossible.
- It is undesirable because it greatly lowers precision.

A cool idea that is hard to realize

- One conceivable solution is to **stratify** the dictionary into
  - a base vocabulary used by all English speakers, and
  - optional extensions for specific genres, styles, etc.
- Extensions could be loaded if the text so far
  fits certain criteria.

  high number of field-specific terms, loan words, etc.
- To the best of my knowledge,
  nobody has ever tried anything like this.
- The payoff probably isn't worth the effort.
  So what's the **alternative?**

Another Solution for Unlisted Words

- Humans can easily distinguish possible words of English from impossible ones.

Example

possible	**impossible**
blick	bnick
wrexel	rwexel
lakoo	ooakl
orcalate	orclte

- Only some sequences of characters can occur in English words.
- You guessed it: **n-grams again!**

Character n-grams for non-word detection

Non-word detection algorithm

1. Compile list of character bigrams that occur in words in the word list.
2. A word is a non-word if it
   1. is not in the dictionary, and
   2. contains an illicit character bigram

Example

- Word list: bee, bored, doom
- Character bigrams: be, ee, bo, or, re, ed, do, oo, om

Word	In list?	Illicit bigram?	Verdict?
bee	yes	—	good
boredom	no	no	good
bnick	no	yes	bad
beeeereed	no	no	good

Which impossible words are detected?

- We have seen several times by now that bigrams are insufficient in certain applications.
- We can increase the value of n (e.g. 3, 4, 5).

Example

Impossible Word	**Character Bigrams**	**Illicit Bigrams**
bnick	bn, ni, ic, ck	bn
rwexel	rw, we, ex, xe, el	**none!**
akklaim	ak, kk, kl, la, ai, im	**none!**

Improving the n-gram model

1. **Size**
   - trigrams much better than bigrams
   - *Example*
     kk and *kl* occur in a few English words, but not *kkl*
2. **End Markers**
   - beginning and end of English words are special
   - *Examples*
     nk is very common, but impossible at beginning of word
     cl is very common, but impossible at end of word
   - special character \$ for word edges
     \$nk impossible, *nk* and *nk\$* allowed
3. **Probabilities**
   - determine frequency of character n-grams
   - treat non-word probability as product of character n-grams
   - everything below a certain threshold is a non-word

Finding the correct word

- We have word n-grams for finding some instances of real-word errors.
- We have word lists and character n-grams for finding non-word errors.
- **But:** still need mechanism for **spelling suggestions**.
- **Three Common Approaches**
  - rule-based
  - similarity key
  - minimum edit distance

Option 1: Ruled-based approach

- custom rules provide candidate lists for specific misspellings

Example

- teh $\Rightarrow$ the
- refering $\Rightarrow$ referring
- fyre $\Rightarrow$ fire, fry

- can be collected by hand or automatically

Advantages	**Disadvantages**
conceptually simple	labor intense
	inflexible

Option 1: Ruled-based approach

- custom rules provide candidate lists for specific misspellings

Example

- teh $\Rightarrow$ the
- refering $\Rightarrow$ referring
- fyre $\Rightarrow$ fire, fry

- can be collected by hand or automatically

Advantages	**Disadvantages**
conceptually simple	labor intense
	inflexible

Option 2: Similarity keys

- classify strings of characters according to similarity
- words in same similarity class are offered as suggestions

Example: US Census Soundex Algorithm

1. Always retain first letter.
2. Drop all (other) occurrences of *a*, *e*, *i*, *o*, *u*, *y*, *h*, *w*.
3. Replace all (other) consonants by digits:
   - *b*, *f*, *p*, *v* ⇒ 1
   - *c*, *g*, *j*, *k*, *s*, *x*, *z* ⇒ 2
   - *d*, *t* ⇒ 3
   - *l* ⇒ 4
   - *m*, *n* ⇒ 5
   - *r* ⇒ 6
4. Remove consecutive copies of same digit.
5. Shorten/lengthen to 4 characters.

bearded	$\xrightarrow{1+2}$
brdd	$\xrightarrow{3}$
b633	$\xrightarrow{4}$
b63	$\xrightarrow{5}$
b630	$\xleftarrow{5}$
b63	$\xleftarrow{4}$
b663	$\xleftarrow{3}$
brrd	$\xleftarrow{1+2}$
borrowed	

(Dis)Advantages of similarity keys

Advantages

- easy to implement
- easy to compute

Disadvantages

- similarity key must be carefully designed for application
  Soundex is not a good choice for spell checking
- may need distinct key for distinct languages

Option 3: Minimum edit distance

How many steps does it take to transform one word into another?

Levenshtein Distance

The Levenshtein Distance between x and y is n
if the shortest sequence of

- single character deletions, and/or
- single character insertions, and/or
- single character substitutions

that turns x into y takes n steps.

Example: Transforming *meat* into *bats*

String	**Operation**
meat	
beat	substitute *b* for *m*
bat	delete *e*
bats	insert *s*

Computing Levenshtein distances

- The Levensthein distance is determined by the **shortest sequence** of operations.
- How do we know that there isn't a shorter solution?

Naive Solution

- Try all possible 1-step sequences.
- If desired word among outputs, Levenshtein distance is 1.
- Otherwise, try all 2-step sequences.
- If desired word among outputs, Levenshtein distance is 2.
- Otherwise, . . .

Evaluating the naive solution

- The naive solution is guaranteed to terminate (= it won't run forever).
- **Reason:** The Levenshtein distance between two words is at most the length of the longer word.
- **But:** The **combinatorial explosion** is enormous.

Example

4 character word, 26 letter alphabet

Steps	Possible Operations	Computed Strings
1	4 del, 5×26 ins, 4×26 sub	238
2	10 del, 15×26 ins, 10×26 sub	660

Improving efficiency

- **Problem 1:** "generate and test" is too undirected
- **Problem 2:** n-step calculation repeats computations from $(n-1)$-step calculation

Dynamic Programming

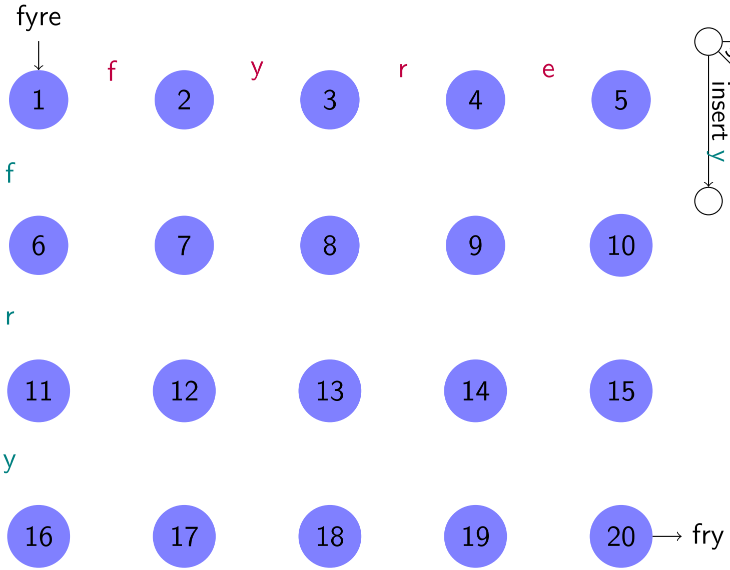
1. Decompose big problems into small problems.
2. Solve the small problems and **save the solution**.
3. Look up stored solutions rather than recomputing them.

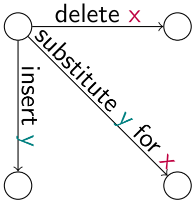
Intuition: Write down intermediate results, just like humans do.

Dynamic programming for Levenshtein distance

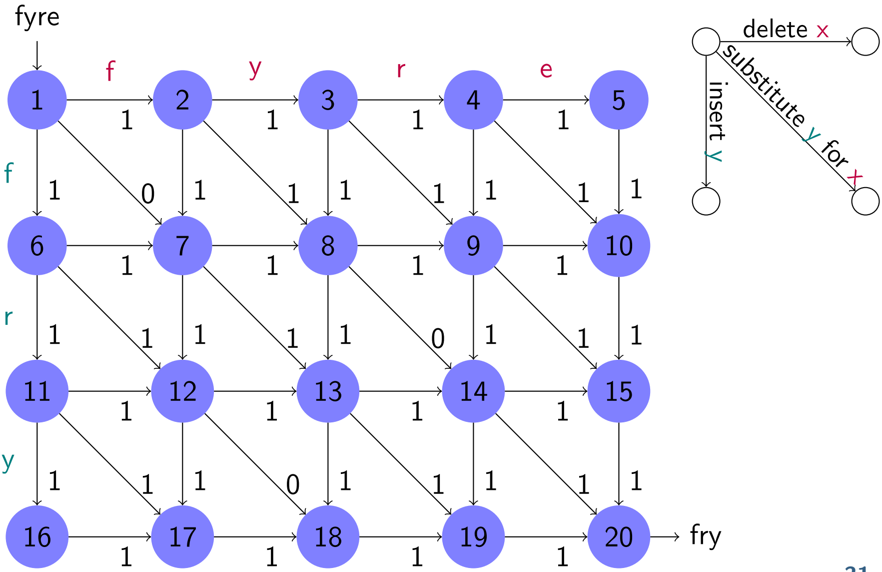
- Draw a graph that represents the possible edit sequences from x to y.
- We want the least costly path through that graph.
- Dynamic programming solution:
  1. For each node,
     what is the least costly path to it from adjacent nodes?
  2. Throw away all other paths.
  3. Go backwards from target to source to find correct path.

Example of dynamic programming

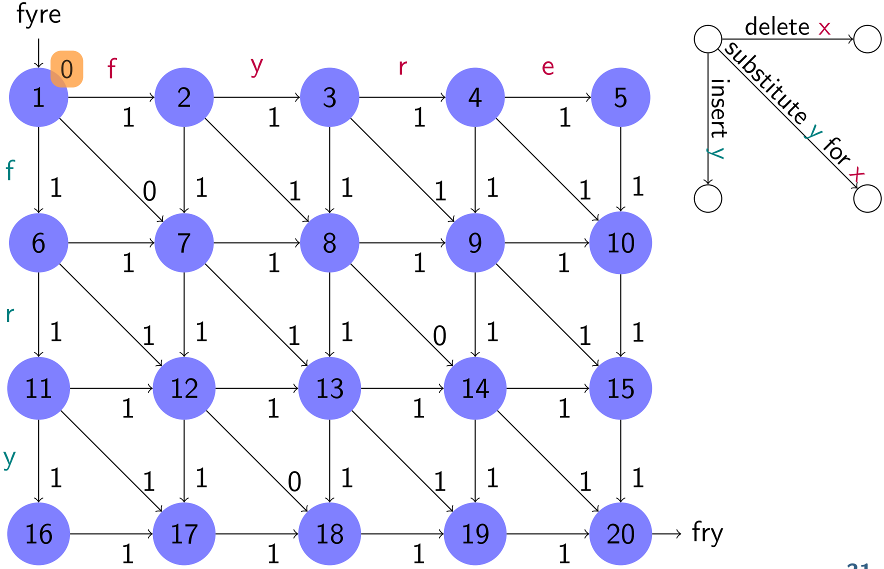




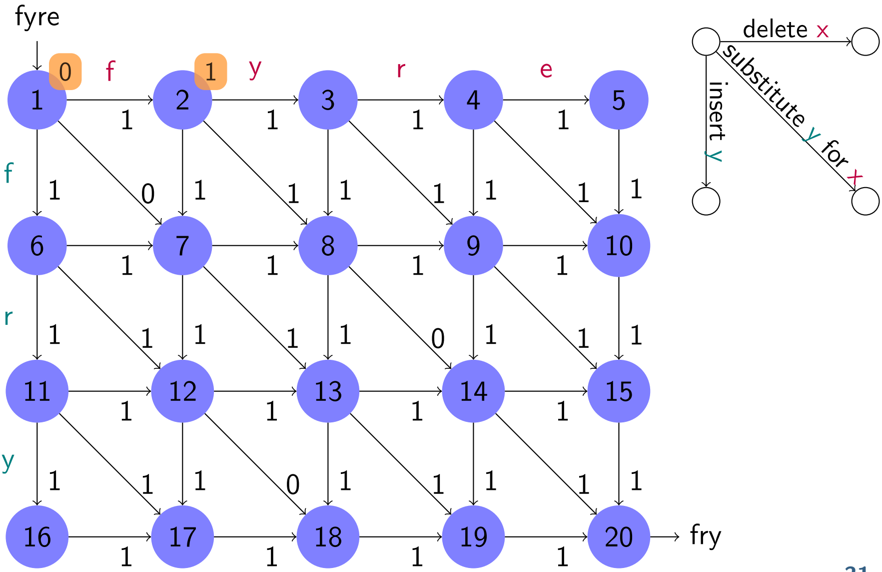
Example of dynamic programming



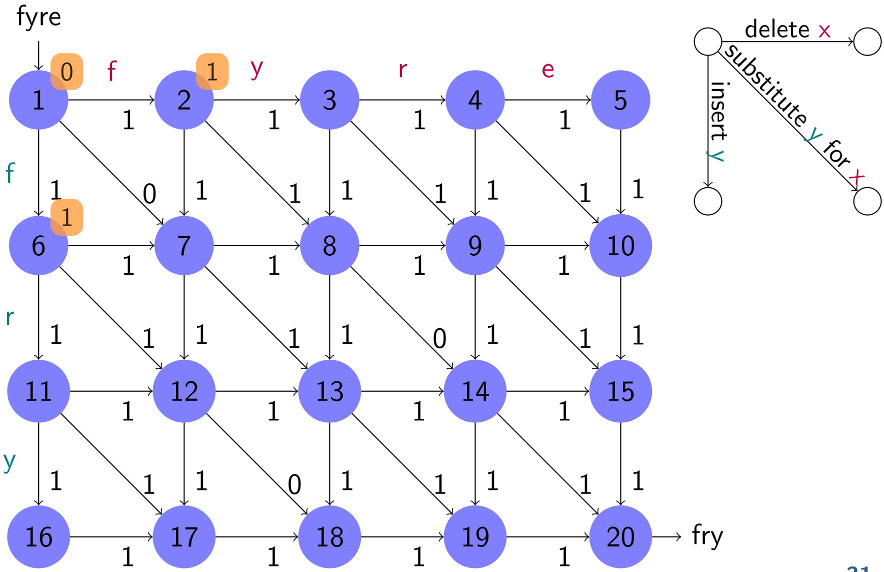
Example of dynamic programming



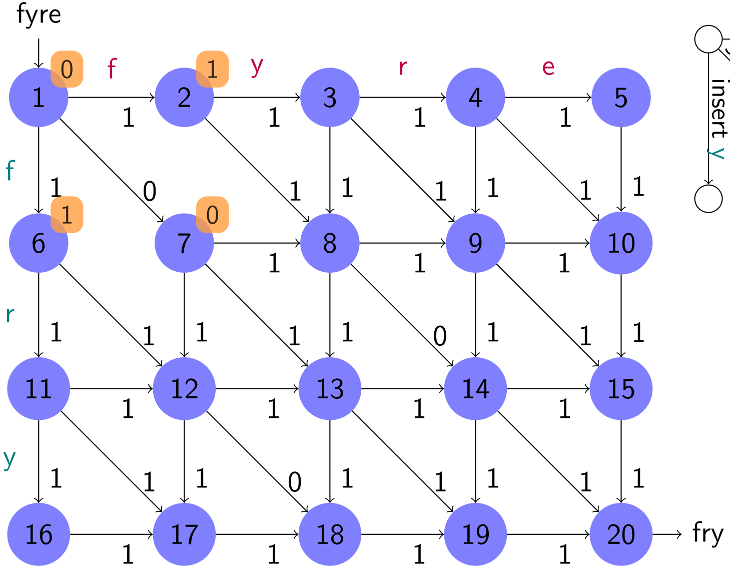
Example of dynamic programming

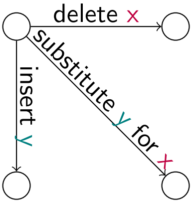


Example of dynamic programming

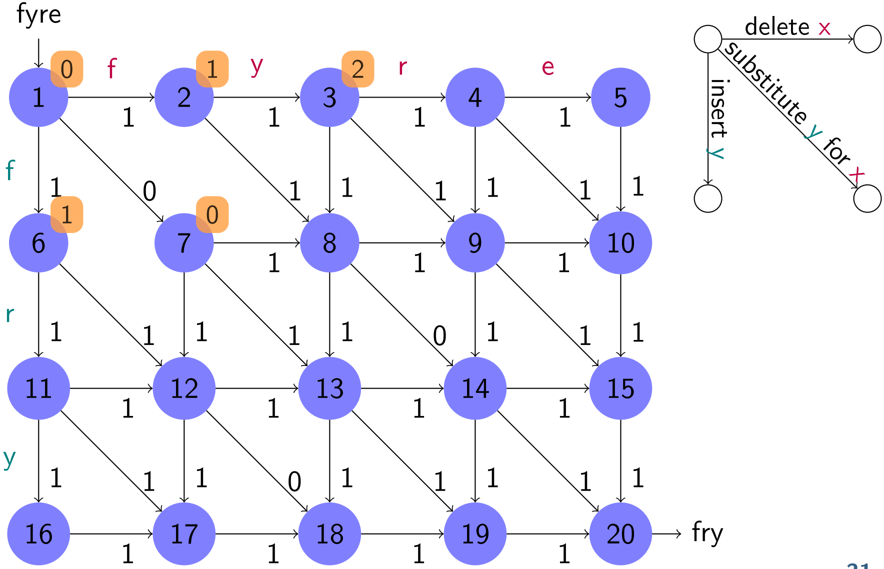


Example of dynamic programming

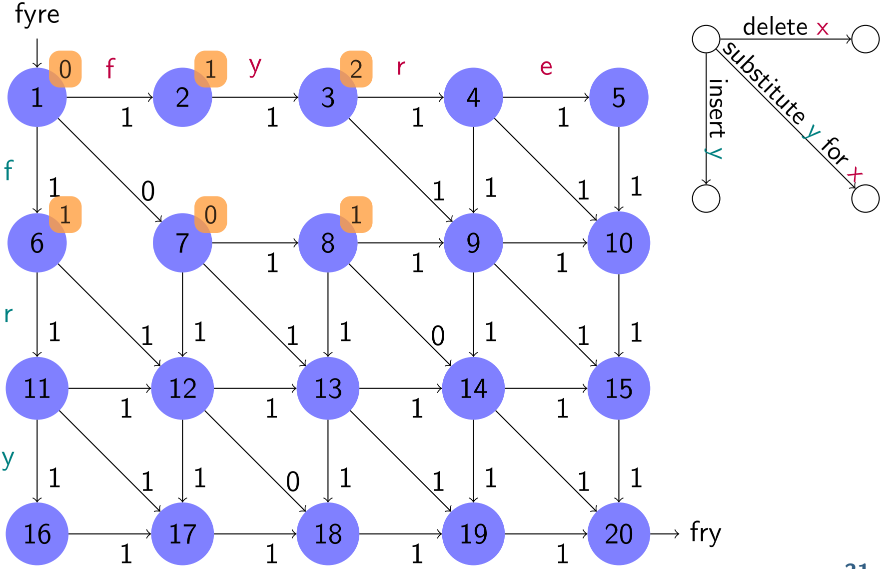




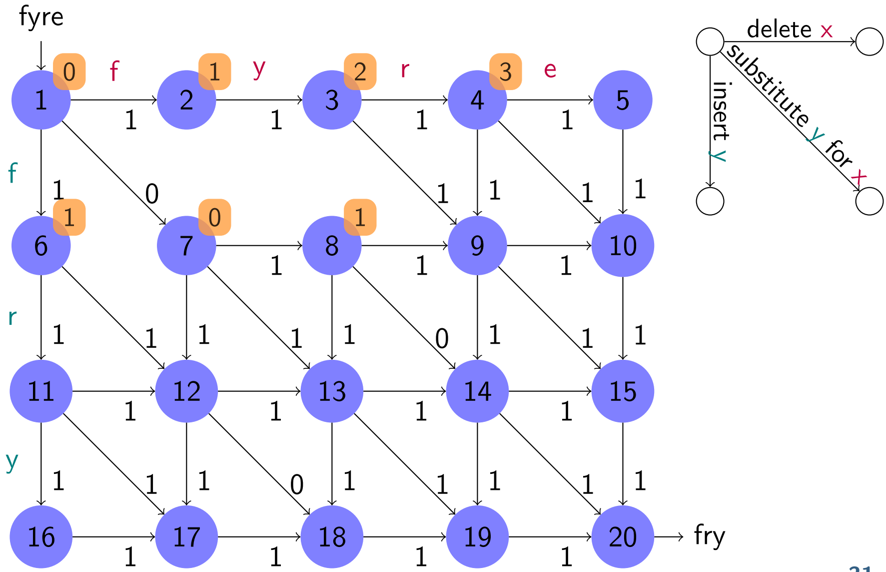
Example of dynamic programming



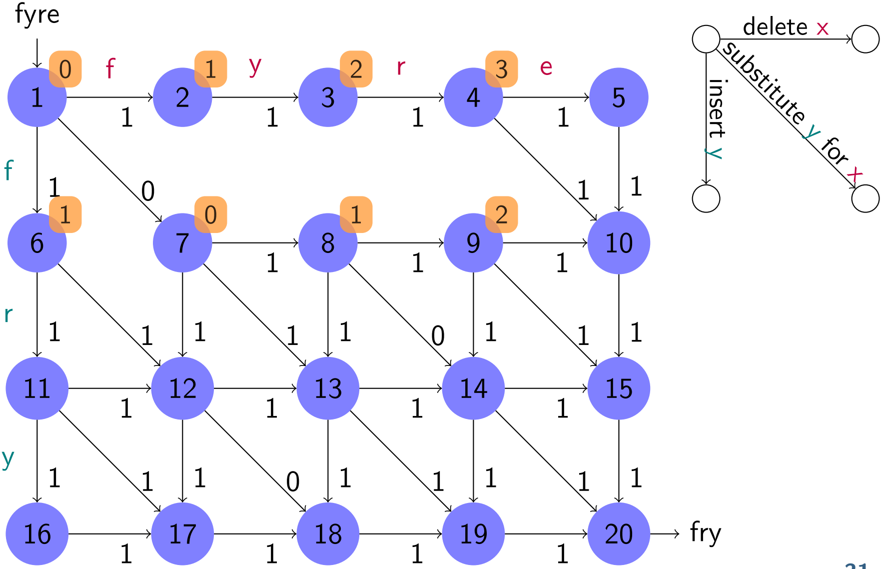
Example of dynamic programming



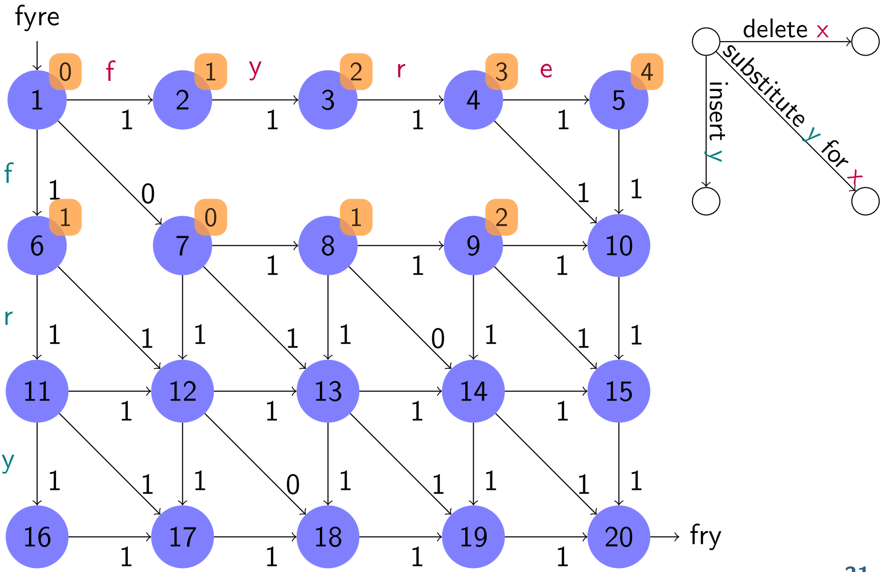
Example of dynamic programming



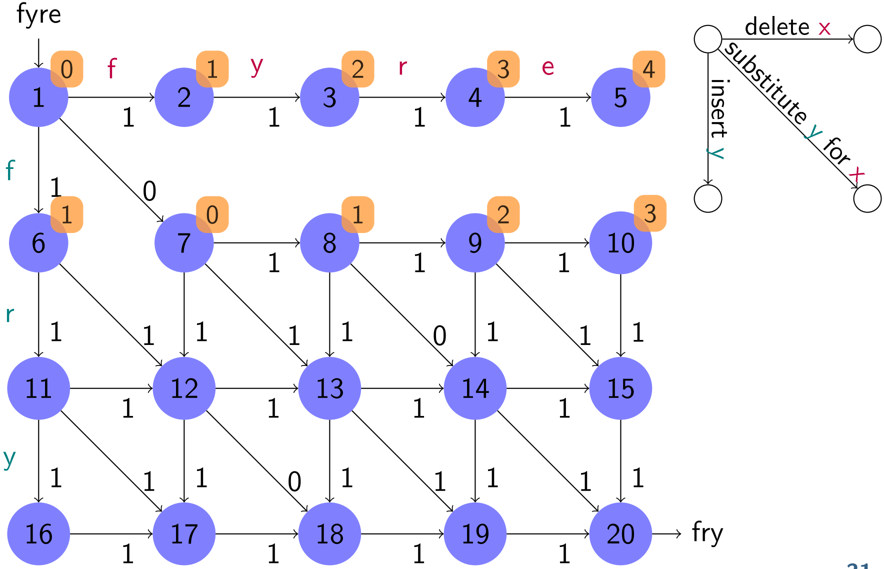
Example of dynamic programming



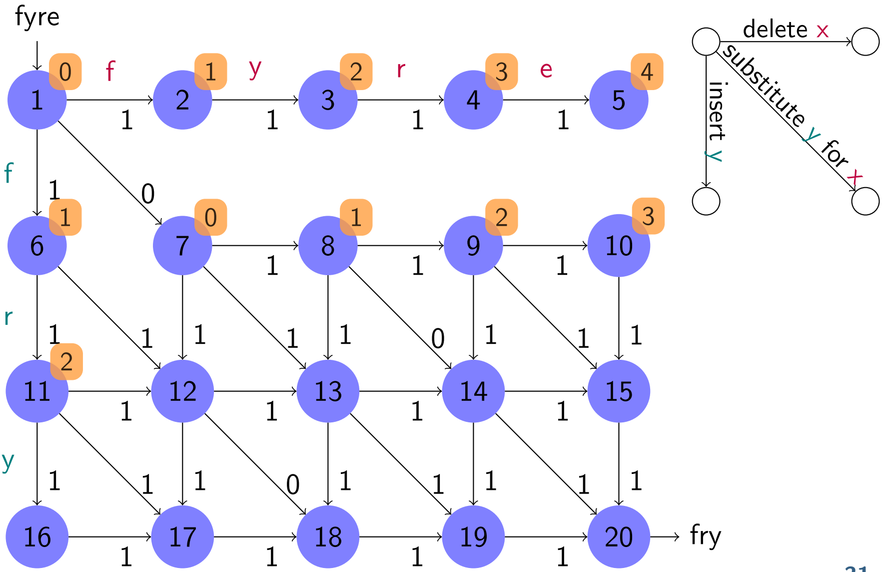
Example of dynamic programming



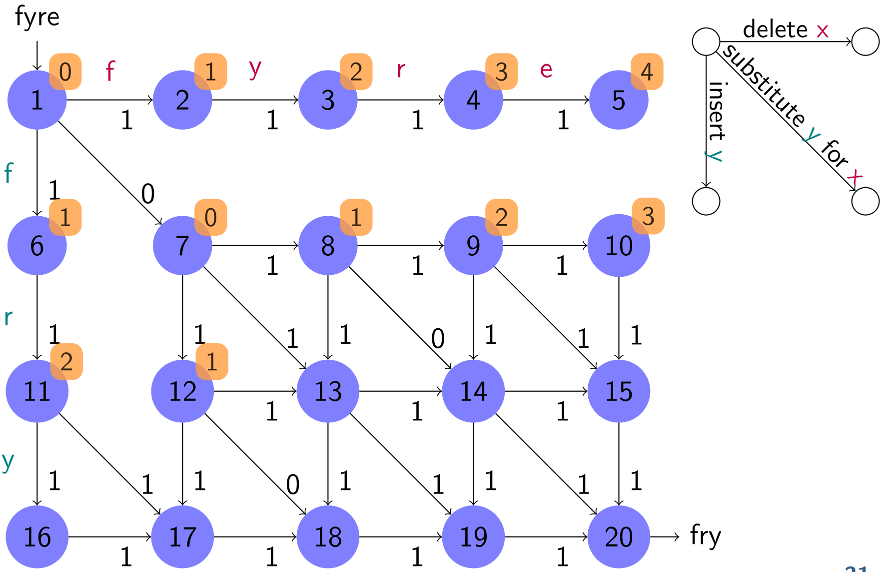
Example of dynamic programming



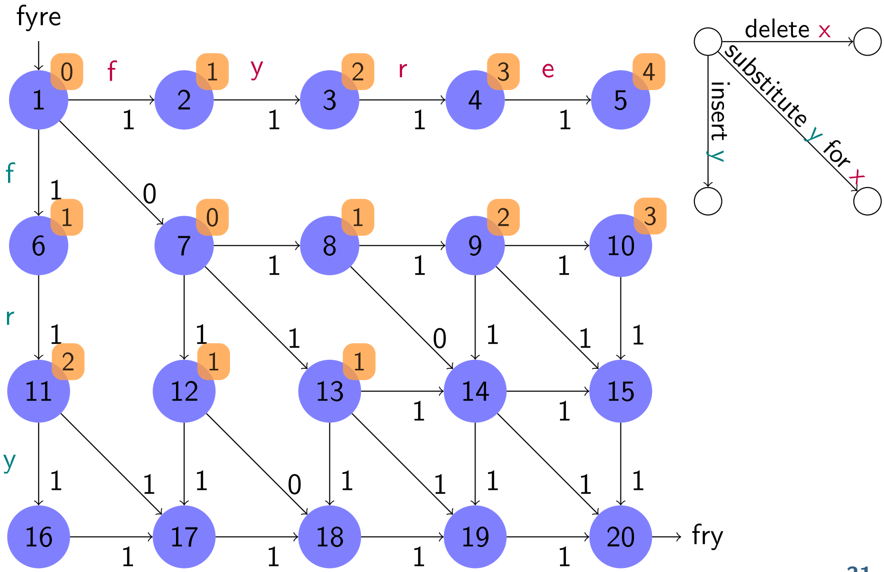
Example of dynamic programming



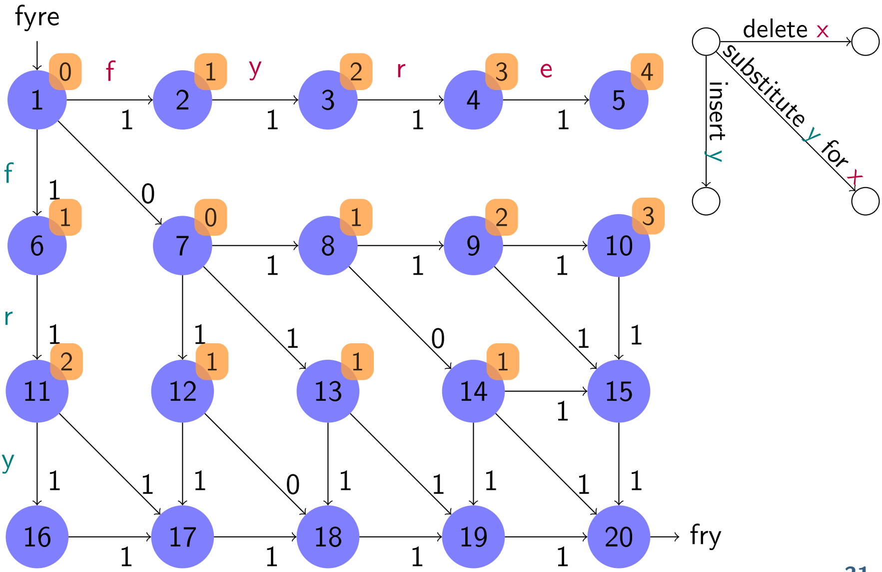
Example of dynamic programming



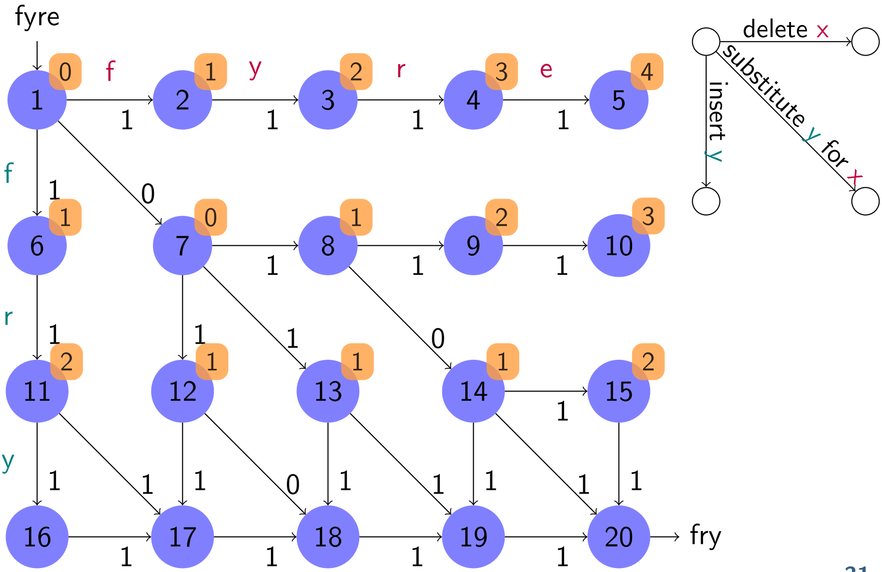
Example of dynamic programming



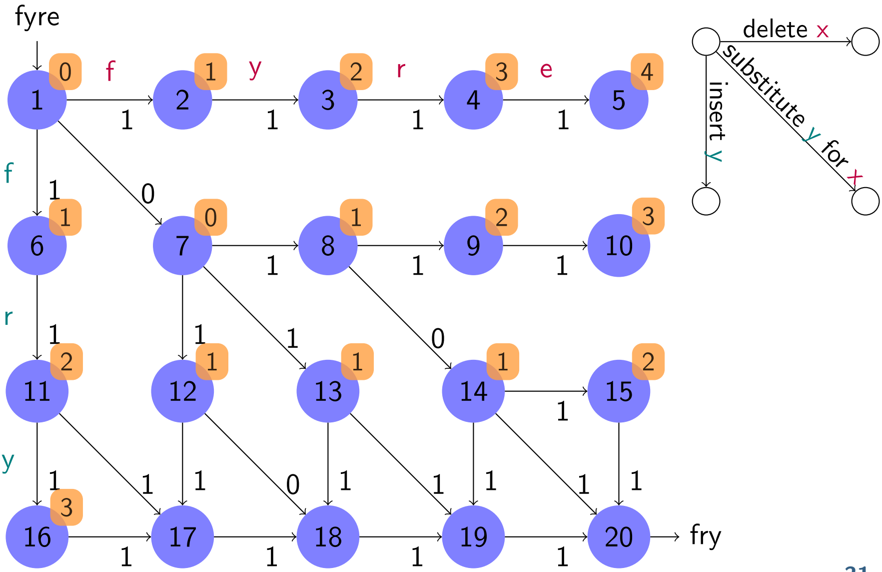
Example of dynamic programming



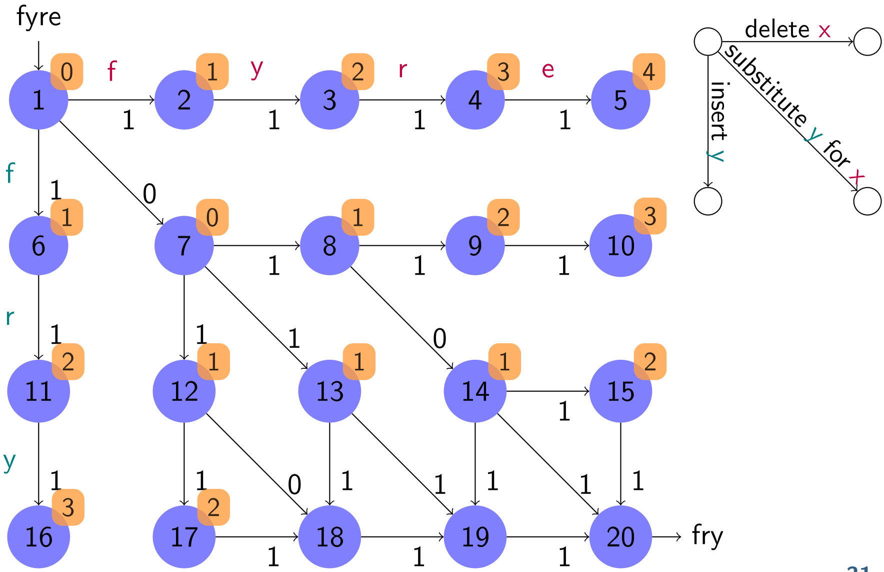
Example of dynamic programming



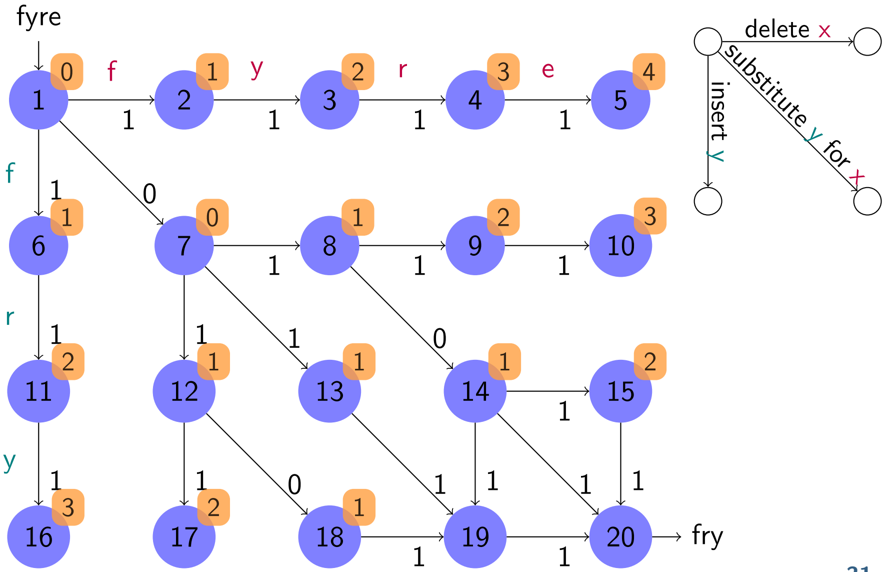
Example of dynamic programming



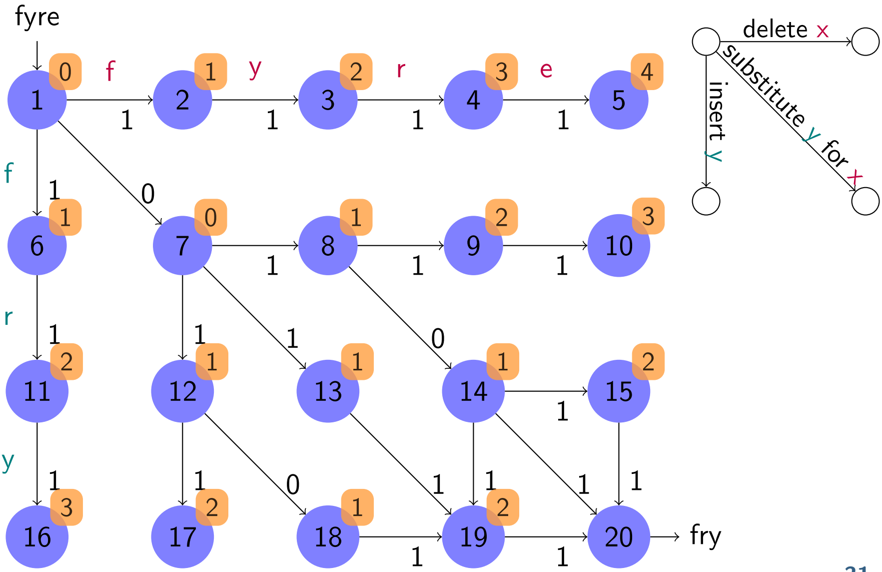
Example of dynamic programming



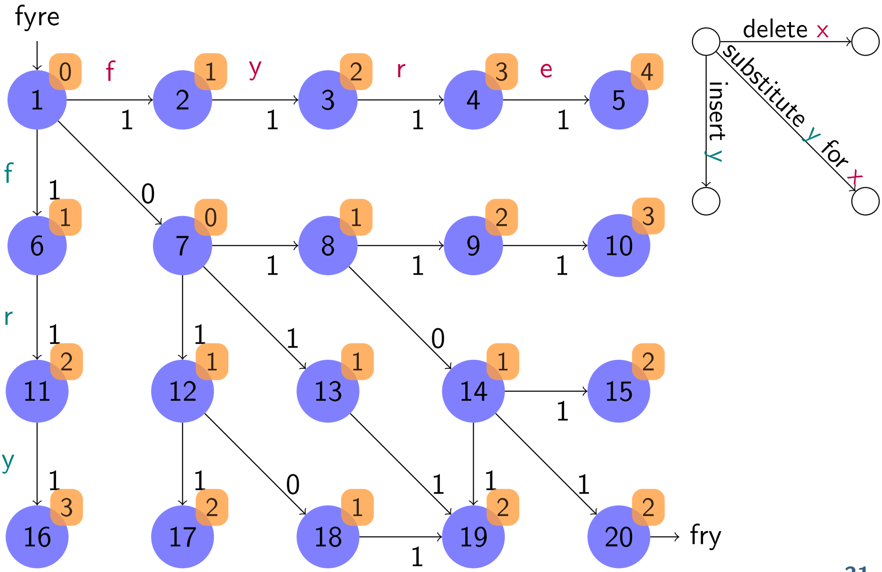
Example of dynamic programming



Example of dynamic programming



Example of dynamic programming



Evaluation of Levenstein distance

- fully automatic
- more easily applied across languages
  (given a character-based orthography)
- computationally demanding, but
  - dynamic programming tames the beast
  - majority of misspellings have distance ≤ 3

Comparsion of Edit Distance Metrics

Metric	Operations
Damerau-Levenshtein	insert, delete, substitute, transpose
Levenshtein	insert, delete, substitute
longest common subsequence	insert, delete
Hamming	substitute

Evaluation of Levenstein distance

- fully automatic
- more easily applied across languages
  (given a character-based orthography)
- computationally demanding, but
  - dynamic programming tames the beast
  - majority of misspellings have distance ≤ 3

Comparsion of Edit Distance Metrics

Metric	**Operations**
Damerau-Levenshtein	insert, delete, substitute, transpose
Levenshtein	insert, delete, substitute
longest common subsequence	insert, delete
Hamming	substitute

Adding probabilities. . . again

- Once again probabilities improve accuracy.
- **Confusion probabilities** measure how likely one word is to be typed as realization of another.
- Difficult to compute, affected by many parameters
  keyboard layout, pronunciation, optical similarity, context, . . .

Putting it all together

Step 1: Detecting Spelling Errors

- *Non-Word Errors*
  1. Is the word in the dictionary?
  2. If no, does the word contain illicit character n-grams?
  3. If no, is the word an unlikely combination of licit character n-grams?
- *Real-Word Errors*
  1. Does the word appear in an illicit n-gram?
  2. If no, does the word appear in an unlikely n-gram?

Putting it All Together [cont.]

Step 2: Computing and Ranking Possible Corrections

1. *Calculate Set of All Possible Corrections*
   - just use dictionary
   - use naive edit distance algorithm to compute set of possible corrections up to some distance, then remove all words not in dictionary
2. *Ranking Possible Corrections*
   combined heuristic based on
   - Levenshtein distance
   - confusion probability
   - word probability
   - maximizing sentence probability