CSCI 2270

Data Structures & Algorithms

Gabe Johnson

Lecture 23 Mar 11, 2013

Text Processing

Upcoming Homework Assignment

HW #7 **Due: Friday, Mar 15**

Huffman Encoding

We're writing a codec to turn plain text into a (slightly silly) "bit string" format and back. This is one of the earliest compression algorithms but it it is similar in spirit to many more modern ones, and gives insight into how greedy algorithms and clustering work. I'll talk about some of the funky things included in the header file.

Lecture Goals

1. Huffman Assignment
2. Text Compression
3. Regular Expressions
4. PL Lexing/Parsing
5. Natural Lang Processing

Huffman Assignment

‘Typedef’ can be used to alias a complicated or confusign data type to an simpler or more meaningful one:

```
typedef complicated_thing easy_thing;
```

This is what I did in the header file:

```
typedef
priority_queue<freq_info*,
               vector<freq_info*>,
               freq_info>
tree_queue;
```

Typedef'ed pq

This is what I did in the header file:

```
typedef
priority_queue<freq_info*,
                         vector<freq_info*>,
                         freq_info>
tree_queue;
```

This means that whenever I type 'tree_queue', it is as though I actually typed 'priority_queue <freq_info*, vector<freq_info*>, freq_info>.

Eh?

```
priority_queue<freq_info*,
                           vector<freq_info*>,
                           freq_info>
```

This declares a C++ STL priority queue that:

- Contains freq_info* elements
- Is backed by a vector of freq_info* elements
- Uses the freq_info definition to sort the elements

Weird ( ) Operator in Struct

```
struct freq_info {
  ...
  bool operator() (freq_info* a, freq_info* b) {
    return a->freq > b->freq;
  }
};
```

This is the thing that lets us sort, and it is why the last param to the `priority_queue` was `freq_info`. This declares an anonymous function that returns a `bool` and takes two `freq_info*` parameters.

Rest of huffman.cpp

You will notice that the huffman.cpp file has quite a bit implemented already. This is because I thought it would be overly cruel to make you have to do the text processing. The first steps that involve reading from a file and forming the initial priority queue are done for you. You should look for the functions marked with *implement me* in the cpp file, and finish them. Also if you got the unit test before Sunday 6:00pm, you should **git pull** again to get the current version.

Text Compression

The **LZW algorithm** (the basis of the *zip* format) is based on building lookup tables. It is based on the insight that we often have sequences of characters (like in English, “ing” is common) that appear frequently, and that we can encode those sequences with a single bit string.

Another less common but often more effective approach is the **Burrows-Wheeler Transform** (the basis of the *bzip2* format).

Regular Expressions

A regular expression (often / usually called a *regex)* is a powerful way to programmatically process text.

Take a pattern that looks like this:

```
\d{3}-\d{4}
```

And apply it to arbitrary text to see if it matches. It matches “867-5309” but not “hello”.

Regex Fun

You can play with regular expressions here:

http://regexpal.com/

It is a very convenient way to play and debug your regular expressions.

When To Use A RegEx

Regexes are helpful when you need to determine what the nature of some text is. They are supremely better than using if/else statements. For example, is

“0122 5555 5555#222”

a properly formatted UK mobile phone number? A regex to match this is:

```
^((\(?0\d{4}\)?\s?\d{3}\s?\d{3})|(\(?
0\d{3}\)?\s?\d{3}\s?\d{4})|(\(?0\d{2}\)?\s?
\d{4}\s?\d{4}))(\s?\#(\d{4}|\d{3}))?$
```

Semi-random aside

```
⍝ John Conway's "Game of Life".
⍝ Expression for next generation.
life←{
    ↑1 ω∨.^3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂ω
}
```

This is APL, an actual programming language. It is often referred to as a “write-only programming language”, because you can write things in it, but *nobody—not even the author—can understand what the code means afterwards.*

RegEx: a write-only syntax

Below, ‘*a*’ and ‘*b*’ represent integers; x is a sub-pattern.

`\`*a*	`backref`	`(x)`	`group for backref`
`\d`	`digit`	`x*`	`0 or more`
`\D`	`not digit`	`x+`	`1 or more`
`\s`	`whitespace`	`x?`	`0 or 1`
`\S`	`not W.S.`	`x{`*a*`}`	*a* `repetitions`
`\w`	`word`	`x{a,}`	`a ≤ n reps.`
`\W`	`not word`	`x{a,b}`	`a ≤ n ≤ b reps.`

There are lots more rules and special symbols.
Google for “ECMAScript RegEx Syntax”.

RegEx Example 1

```
(\d{3})-\1\d
```

Match three digits in a row, a dash, and then the original three digits with one more arbitrary digit.

\d{3} means any three digits. They are captured with the parens, and their value is recalled with the \1.

RegEx Example 2

```
\w+[\w-\.]*\@\w+((-\w+)|(\w*))\.[a-z]
{2,3}$|^([0-9a-zA-Z'\.]{3,40})\*|
([0-9a-zA-Z'\.]+)@([0-9a-zA-Z']+)\.
([0-9a-zA-Z']+)$|([0-9a-zA-Z'\.]
+)@([0-9a-zA-Z']+)\*+$|^$
```

Validates an email address. (Taken from regexlib.com, author: Julio de la Yncera).

See what I mean about write-only syntax?

RegEx in C++

RegExes are available in C++ using the new C++11 standard. To use, `#include <regex>`.

More documentation is on cplusplus.com.

Prog. Lang. Processing

Regular expressions only work on strings that have predictable formats. They can’t match recursively nested structures like the text of a C++ program.

To process complex (but still grammatically precise) text, we can use fancier matching strategies, like *deterministic finite automata*, or **DFA**.

PL Lexing

Lexing is the process of taking some input stream and identifying the lexical components called *tokens*. This is based on rules, which are often represented as regular expressions (though they can be something else).

Tokens are the smallest operating component of a source code file. They include symbols like { } ( ) + % # and compound things like keywords (bool, int) and literals like “Hello World!”

PL Lexing Example

In C++, lexing a ‘hello world’ program might give us the tokens: comment, pre_proc_dir, using, namespace, std, semicolon, int, main, left_paren, right_paren, left_curl, cout, left_angle, left_angle, str_literal, semicolon, right_curl.

```
// my first program in C++

#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World!";
}
```

PL Parsing

Parsing a source code file is a different beast. Here we have rules that specify how a compound thing can be composed out of tokens or other compound things.

For example, a function definition in a my language called “Slippy” (Google for “SlippyParser.g” if you want to see it) has a full grammar defined for ANTLR v3.

PL Parsing 2

Here are some rules (often called *productions*) that start to define the grammar for a program structure. Programs are composed of one or more statements. Statements are either definition or non-definition statements. Definition statements are either function or class definitions. A real programming language has dozerns or hundreds of these rules.

```
program :
      statement+;

statement:
      definitionStatement |
      nondefStatement;

definitionStatement:
      functionDefStatement |
      classDefStatement;

nondefStatement:
      flowControlStatement |
      expression;
```

PL Parsing 3

The parsing process operates *entirely* on the tokens produced by a lexer. Many parser productions are high level and only involve lower-level productions, but ultimately, everything in the parser uses tokens.

If you want to know more about how programming language lexing/parsing works I might be convinced to do a full lecture on it. Just email me and I'll do it if there's enough interest.

Natural Language Processing

Programming languages must conform to some syntax and grammar in order to be used.

But in natural language, syntax and grammar are more like friendly guidelines. We can understand each other even if we don’t speak in the Queen’s English. This is easy for humans because our brains have evolved to handle it. But we haven’t yet been able to make computers understand human language to the same level of understanding.

NLP: Where to Start?

There's *spoken* language processing, which deals with converting sound into text,

and there's *machine translation*, which involves converting text into meaning.

It is tempting to just turn the spoken language processing part into a black box and assume that someone, somewhere, will figure it out, and we can just tackle the machine translation part.

Words vs Meaning

Another problem in NLP is in the distinction between determining lexically what was said (the words) and the semantics of what was said (the meaning).

Like with spoken language and machine translation, we can use knowledge in one to improve accuracy in the other.

Use Grammar

Say we have several possible interpretations of some voice input that have reasonable lexical breakdowns:

"The mall is on the other side of the park."
"Them all is on the other side of the park."

We can often, **but not always**, use a natural language's grammar to disambiguate statements. We can't do this if a statement has poor grammar.

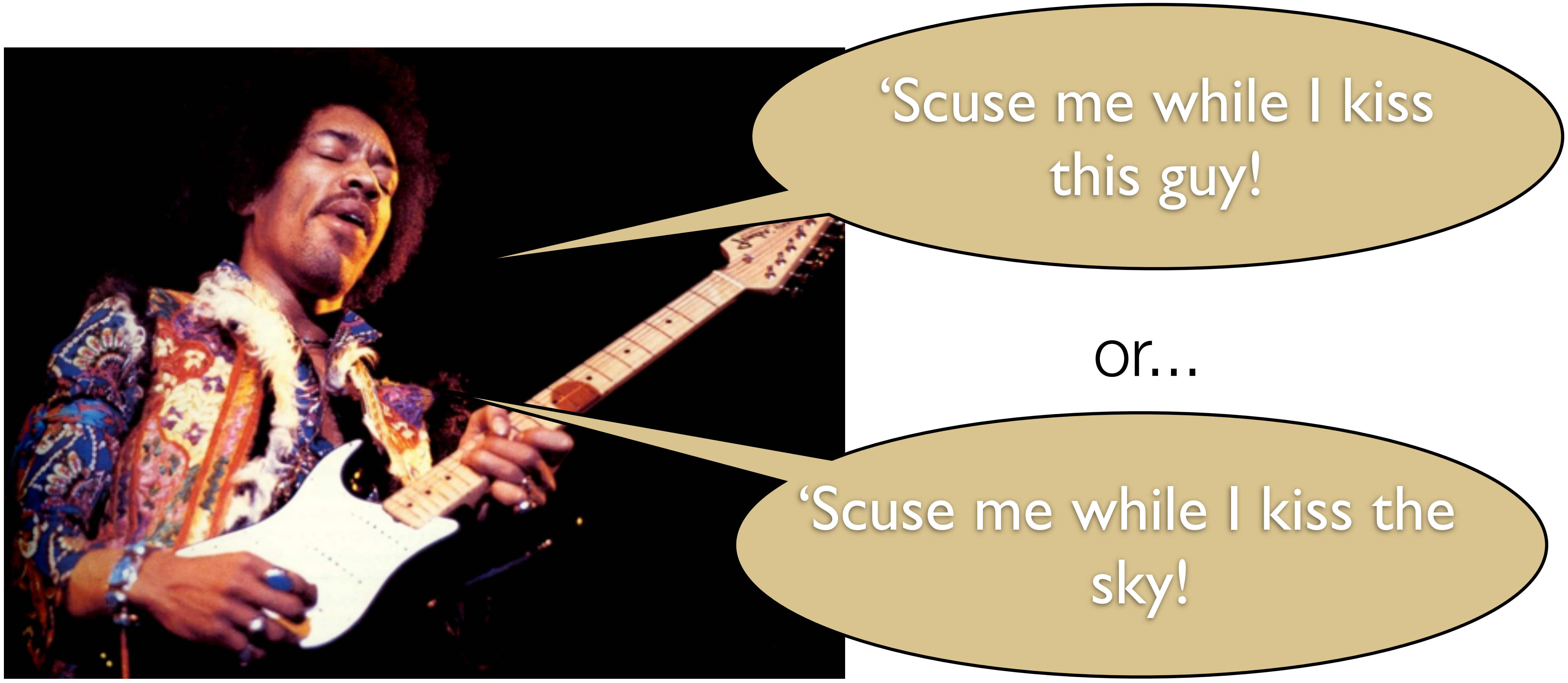
Use Context

“Nice Shot!”

Does ‘shot’ refer to:

- medicine from a needle?
- playing basketball?
- archery?
- photography?
  (if so: the exposure, or the field of view?)
- an insult?
- an attempt?

Use Common Sense


‘Scuse me while I kiss this guy!
or...
‘Scuse me while I kiss the sky!

Actually this might not be the best example, because neither really makes sense, unless you can dig trippy ideas like kissing the sky. Worked for Jimi.

Vague Pronoun

What does ‘they’ refer to in this sentence?

“I fed the bananas to the monkeys because they were ______.”

If the word in the blank is “hungry”, *they* probably refers to the monkeys.

If it is “ripe” *they* probably refers to the bananas.

Use Probability

Say we have a partial translation and would like to fill in the blank:

“Four score and __________ our fathers brought forth…”

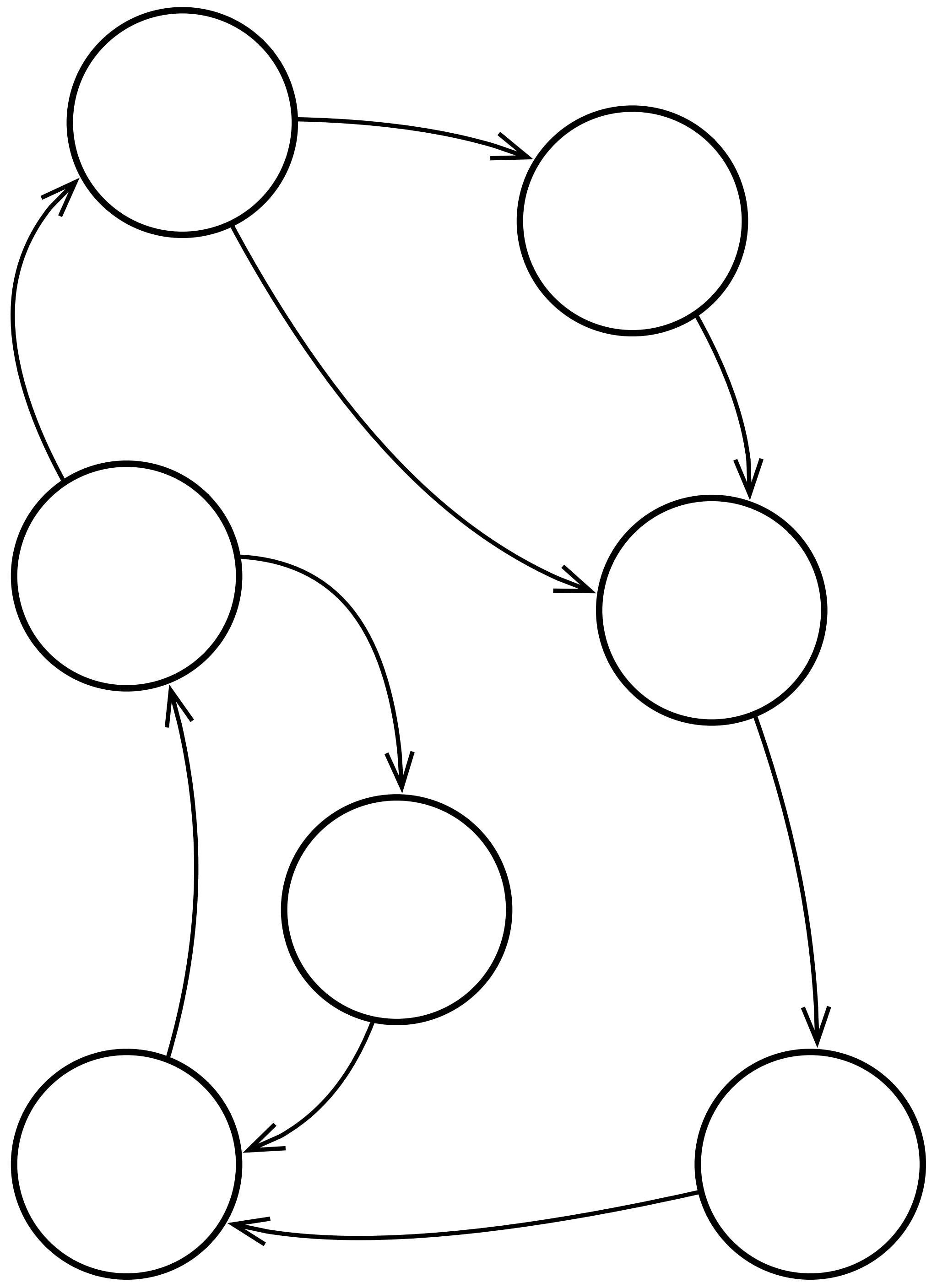
Humans have history, and we have literature, and we have commonly used sayings. I’m willing to bet most of the Americans in the room (at least) know exactly what goes in that blank.

Probability and Graphs

Now back to data structures and algorithms.

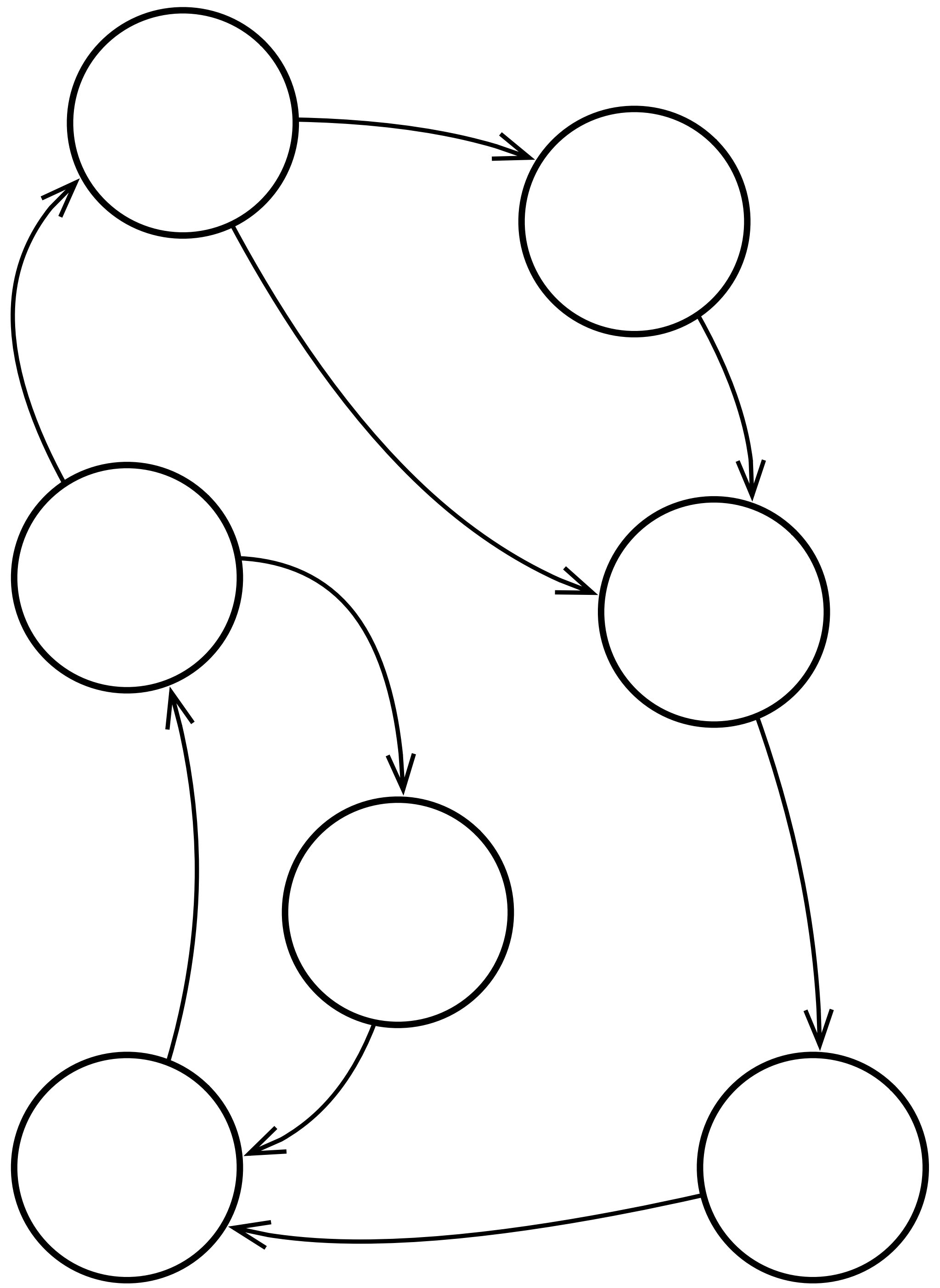
All of the natural language processing we’ve talked about today involves some educated guesswork. We’re pretty sure Hendrix was talking about *the sky*, and we know that Abe Lincoln said “four score *and seven years ago”.* But are we absolutely sure? No, but we’re sure enough to bet money on it.

Tangled Webs of Words



In NLP and lots and lots of other AI topics, not to mention basically everything Facebook and Google does, can be modeled using graphs. A graph has nodes (the circles) and edges.

Graphs in language



Nodes may refer to words (the lexical items present), concepts (global ideas), context (locally present ideas). The edges (arrows) may refer to transitions between words, statistical correlation among concepts or topics, or passage of time.

Graphs: After Break

I’ll start getting into Graphs on Friday. They won’t be on the exam, though, so this is basically “after spring break” territory.

They have so many uses in modern computation that they really should have the same status as linked lists and binary trees. All the homework we do after break will involve graphs in some way.