Introduction to Theoretical CS

Two fundamental questions.
- What can a computer do?
- What can a computer do with limited resources?

General approach.
- Don’t talk about specific machines or problems.
- Consider minimal abstract machines.
- Consider general classes of problems.

Why Learn Theory

In theory . . .
- Deeper understanding of what is a computer and computing.
- Foundation of all modern computers.
- Pure science.
- Philosophical implications.

In practice . . .
- Web search: theory of pattern matching.
- Sequential circuit: theory of finite state automata.
- Compilers: theory of context free grammar.
- Cryptography: theory of complexity.
- Data compression: theory of information.

Finite State Automaton

Simple machine with N states.
Finite State Automata

Simple machine with N states.
- Start in state 0.
- Read an input bit.
- Move to new state
  - depends on input bit and current state
- Stop when last bit read.
  - ‘yes’ if end in accept state(s)
  - ‘no’ otherwise

'Yes' also called accepted or recognized inputs from a language.

A Second Example

Consider the following two state FSA.

What bit strings does it accept?
- Yes: 0, 11110, 00000, 100100111011
- No: 1, 1111, 00, 1011100111011

C Code for FSA

```c
#include <stdio.h>
define STATES 4
#define START_STATE 0
#define ACCEPT_STATE 3

int main(void) {
    int i, state = START_STATE;
    int transition[STATES][2] =
        { {2, 1}, {3, 2}, {2, 2}, {2, 1} };
    while (scanf("%1d", &i) != EOF)
        state = transition[state][i];
    if (state == ACCEPT_STATE)
        printf("Yes.\n");
    else
        printf("No.\n");
    return 0;
}
```

An Application: Bounce Filter

Bounce filter: remove isolated b's and g's in input.
- Input: b b g b b b g g b b b b
- Output: w b b b b b b g g g g g g g b b b b
  (one-step delay)

Transition Table

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Transition Table

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

An Application: Bounce Filter

Bounce filter: remove isolated b's and g's in input.
- Input: b b g b b b g g b b b b
- Output: w b b b b b b g g g g g g g b b b b
  (one-step delay)
**An Application: Bounce Filter**

Bounce filter: remove isolated b’s and g’s in input.
- Input: b b g b b g b g g g b b b b
- Output: w b b b b b b g g g g g g g b b b b (one-step delay)

State interpretations.
- W: start
- BB: at least two consecutive b’s.
- G: sequence of b’s followed by g.
- GG: at least two consecutive g’s.
- B: sequence of g’s followed by b.

**Text Searching**

Build an FSA that accepts all strings that contain ‘acat’ as a substring.
- tatgacatg
- acacatg

Start building:

State name represents largest prefix of “acat” that input currently matches.

**Text Searching**

Finish building:

State name represents largest prefix of “acat” that input currently matches.

**Web Search Application**

Web search engines build FSAs.

Standard Web search for: cos 126 pattern matching

Search engines have different methods for specifying patterns.
- Which one is most powerful?
- Theory of computation helps us address such issues.
Unix Pattern Matching Tool: egrep

General regular expressions pattern matching.
- Acts as filter.
- Sends lines from stdin to stdout that "match" argument string.

<table>
<thead>
<tr>
<th>Elementary Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>%egrep 'beth' classlist</td>
</tr>
<tr>
<td>03/Smythe/Elizabeth/6/esmythe</td>
</tr>
<tr>
<td>03/Bethke/Kristen/3/kbethke</td>
</tr>
<tr>
<td>%egrep '/3/' classlist</td>
</tr>
<tr>
<td>03/Marin/Anthony/3/amarin</td>
</tr>
<tr>
<td>03/Arellano/Belen/3/arellano</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>03/Weiss/Jacob/3/weiss</td>
</tr>
<tr>
<td>%egrep 'zeuglodon' mobydick.txt</td>
</tr>
<tr>
<td>rechristened the monster zeuglodon and in his</td>
</tr>
<tr>
<td>%egrep 'acat' human.data</td>
</tr>
<tr>
<td>gcaacgacacacaacatgcatttt</td>
</tr>
</tbody>
</table>

Crossword Puzzle or Scrabble Too Hard?

/usr/dict/words is a list of (25,143) words in dictionary.
/u/cs126/files/textfiles/wordlist.txt is a list of 234,936 words.

<table>
<thead>
<tr>
<th>More Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>% egrep 'hh' /usr/dict/words</td>
</tr>
<tr>
<td>beachhead</td>
</tr>
<tr>
<td>highhanded</td>
</tr>
<tr>
<td>withheld</td>
</tr>
<tr>
<td>withhold</td>
</tr>
<tr>
<td>% egrep 'u.u.u' /usr/dict/words</td>
</tr>
<tr>
<td>cumulus</td>
</tr>
<tr>
<td>% egrep '..oo..oo' /usr/dict/words</td>
</tr>
<tr>
<td>bloodroot</td>
</tr>
<tr>
<td>nincompoophood</td>
</tr>
<tr>
<td>schoolbook</td>
</tr>
<tr>
<td>schoolroom</td>
</tr>
</tbody>
</table>

Egrep Pattern Conventions

Conventions for egrep:
- c  any non-special character matches itself
- .  any single character
- r* zero or more occurrences of r
- r+ one or more occurrences of r
- r? zero or one occurrences of r
- (r) grouping
- r1|r2 logical OR
- [aeiou] any vowel
- [^aeiou] any non-vowel
- ^ beginning of line
- $ end of line

Flags for egrep:
- egrep -v match all lines except those specified by pattern

Still More Examples

<table>
<thead>
<tr>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td>% egrep 'n(ie</td>
</tr>
<tr>
<td>neither</td>
</tr>
<tr>
<td>% egrep 'actg(atac)*gcta' human.data</td>
</tr>
<tr>
<td>ggtactggctaggac</td>
</tr>
<tr>
<td>% egrep 'actg(atac)*gcta' student.data</td>
</tr>
<tr>
<td>tataactgatacatactgctattac</td>
</tr>
<tr>
<td>% egrep '.*y.(.)*y$' /usr/dict/words</td>
</tr>
<tr>
<td>yesterday</td>
</tr>
<tr>
<td>% egrep -v '[aeiou]' /usr/dict/words</td>
</tr>
<tr>
<td>rhythm</td>
</tr>
<tr>
<td>syzygy</td>
</tr>
</tbody>
</table>

Do spell checking by specifying what you know.

Starts and ends with y, odd number of characters.

Find all words with no vowels and 6 or more letters.
Specifying "pattern" for grep can be complex.

Which aspects are essential?
- Unix grep regular expressions are useful.
- But more complex than theoretical minimum.

Regular expressions.
- Match WHOLE string.
  (to convert to equivalent `grep` pattern, surround by `^` and `$)
  - regular expression: \(0(0|1)^1\)
  - `grep` pattern: `^0(0|1)^1$`
  c                any non-special character matches itself
  r*               zero or more occurrence of r
  (r)              grouping
  r1|r2            logical OR
  r1 \cdot r2      concatenate (usually suppress \cdot symbol)
  .                any single character
  [aeiou]          any vowel
  [^ aeiou]        any non-vowel
  r+                one or more occurrences of r
  r?                zero or one occurrence of r
  -v                match all patterns except...

What kinds of patterns can be specified by regular expressions?
(All but one of following)

<table>
<thead>
<tr>
<th>All bit strings that:</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin with 0 and end with 1.</td>
<td>0001011011</td>
</tr>
<tr>
<td>Equal number of 0’s and 1’s.</td>
<td>0111100010</td>
</tr>
<tr>
<td>Have no consecutive 1’s.</td>
<td>0100101001</td>
</tr>
<tr>
<td>Has an odd number of 0’s.</td>
<td>0100101011</td>
</tr>
<tr>
<td>Has 011010 as a substring.</td>
<td>0001101000</td>
</tr>
</tbody>
</table>

What kinds of patterns can be specified by regular expressions?
(All but one of following)

<table>
<thead>
<tr>
<th>All bit strings that:</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin with 0 and end with 1.</td>
<td>0(0</td>
</tr>
<tr>
<td>Equal number of 0’s and 1’s.</td>
<td>not possible</td>
</tr>
<tr>
<td>Have no consecutive 1’s.</td>
<td>((0 \mid 10)^* (1 \mid 0^*))</td>
</tr>
<tr>
<td>Has and odd number of 0’s.</td>
<td>((1<em>01^<em>01^</em>)^</em> (1<em>01^</em>))</td>
</tr>
<tr>
<td>Has 011010 as a substring.</td>
<td>(0(0</td>
</tr>
</tbody>
</table>
Formal Languages

An ALPHABET is a finite set of symbols.
- Binary alphabet \( = \{0, 1\} \)
- Lower-case alphabet \( = \{a, b, c, d, \ldots, y, z\} \)
- Genetic alphabet \( = \{a, c, t, g\} \)

A STRING is a finite sequence of symbols in the alphabet.
- '0111011011' is a string in the binary alphabet.
- 'tigers' is a string in the lower-case alphabet.
- 'acctgaacta' is a string in the genetic alphabet.

A FORMAL LANGUAGE is an (unordered) set of strings in an alphabet.
- Can have infinitely many strings.
- Examples:
  \( \{0, 010, 0110, 01110, 011110, \ldots\} \)
  \( \{11, 1111, 111111, 11111111, \ldots\} \)

Duality Between FSA’s and RE’s

Observation: for each FSA we create, we can find a regular expression that matches the same strings that the FSA accepts.

Is this always the case?

What about the OTHER way around?

Stay tuned: see Lecture T2.

Limitations of FSA

FSA are simple machines.
- N states \( \Rightarrow \) can’t "remember" more than N things.
- Some languages require "remembering" more than N things.

No FSA can recognize the language of all bit strings with an equal number of 0’s and 1’s.

A warmup exercise:

If \( 01xyz \) accepted then so is \( 00001xyz \)
Limitations of FSA

No FSA can recognize the language of all bit strings with an equal number of 0’s and 1’s.

- Suppose an N-state FSA can recognize this language.
- Consider following input: 000000011111111

  N+1 0’s   N+1 1’s

- FSA must accept this string.
- Some state x is revisited during first N+1 0’s since only N states.
  000000011111111

  x   x

- Machine would accept same string without intervening 0’s.
  000011111111

- This string doesn’t have an equal number of 0’s and 1’s.

Looking Ahead

Today.

- Defined a simple abstract machine = FSA.
- Capable of pattern matching.
- Incapable of “counting.”
- Need to consider more powerful machines.

Future lectures.

- Define an abstract machine.
- Understand how it works and what it can do.
- Find things it can’t do.
- Define a more powerful machine.
- Repeat until we run out of problems or machines.

Hmm. Which will we run out of first?

Lecture T1: Supplemental Notes

C Code for FSA

```c
#include <stdio.h>

int main(void) {
    int c, state = 0;
    while ((c = getchar()) != EOF) {
        if (state == 0 && c == '0') state = 2;
        else if (state == 0 && c == '1') state = 1;
        else if (state == 1 && c == '0') state = 3;
        else if (state == 1 && c == '1') state = 1;
        else if (state == 2 && c == '0') state = 2;
        else if (state == 2 && c == '1') state = 2;
        else if (state == 3 && c == '0') state = 2;
        else if (state == 3 && c == '1') state = 1;
    }
    if (state == 3)
        printf("Yes.\n");
    else
        printf("No.\n");
    return 0;
}
```

straightforward to convert FSA’s into C program or to build with hardware
A Fourth Example

FSA to decide if integer (represented in binary) is divisible by 3?

What bit strings does it accept?
• Yes: 11 (3\textsubscript{10}), 110 (6\textsubscript{10}), 1001 (9\textsubscript{10}), 1100 (12\textsubscript{10}), 1111 (15\textsubscript{10}), 10011 (18\textsubscript{10}), integers divisible by 3.
• No: 1 (1\textsubscript{10}), 10 (2\textsubscript{10}), 100 (4\textsubscript{10}), 101 (5\textsubscript{10}), 111 (7\textsubscript{10}), integers not divisible by 3.

A Fourth Example

FSA to decide if input (convert binary to decimal) is divisible by 3?

How does it work?
• State 0: input so far is divisible by 3.
• State 1: input has remainder 1 upon division by 3.
• State 2: input has remainder 2 upon division by 3.
• Transition example.
  – Input 1100 (12\textsubscript{10}) ends in state 0.
  – If next bit is 0 then stay in state 0: 11000 (24\textsubscript{10}).
  – Adding 0 to last bit is same as multiplying number by 2. Remains divisible by 3.

Regular Expressions

Rules for creating regular expressions (RE’s):
- 0 or 1 or \(\varepsilon\) symbols
- (a) grouping
- ab concatenation
- a + b logical OR
- a* closure (0 or more replications)

where a and b are regular expressions.

Examples:
- \((10)^*\) \(\varepsilon, 10, 1010, 101010, \ldots\)
- \(0(0 + 1)^*0\) \(00, 000, 010, 0000, 0110, \ldots\)
- \((1*01*01*01^*)^*\) \(\varepsilon, 000, 00000, 11101110101111, \ldots\)