C Programming: Graph traversals: DFS and BFS

Yipeng Huang

Rutgers University

February 17, 2022
Table of contents

Announcements
  Quizzes and programming assignments
  Reading and class session plan

Graphs: representations, DFS, and BFS
  The solveMaze problem
  Using graphutils.h
  A DFS approach for solving isTree (using recursion)
  A BFS approach for solving solveMaze (using a queue)

Bits and bytes
  Why binary
  Decimal, binary, octal, and hexadecimal
  Bitwise operations
  Representing characters

Integers and basic arithmetic
  Representing negative and signed integers
Quizzes and programming assignments

Short quiz 3

▶ Now out, due tonight Thursday February 17, 11:59 pm.
▶ No weekly short quiz next week. Focus on PA2 instead

Programming assignment 2

▶ Has been out, due Thursday February 24, 11:59 pm.
▶ More review of CS 111 and 112 concepts in the C language.
▶ Stacks. Queues. BSTs. Graph algorithms.
Reading and class session plan

Reading: CS:APP Chapter 2

- Chapter 2: Representing and manipulating information
- First focus on Chapters 2.1-2.3, integers

Class session plan

1. Today: Discussion of PA2. Representing integers in computer architecture.
Table of contents

Announcements
   Quizzes and programming assignments
   Reading and class session plan

Graphs: representations, DFS, and BFS
   The solveMaze problem
   Using graphutils.h
   A DFS approach for solving isTree (using recursion)
   A BFS approach for solving solveMaze (using a queue)

Bits and bytes
   Why binary
   Decimal, binary, octal, and hexadecimal
   Bitwise operations
   Representing characters

Integers and basic arithmetic
   Representing negative and signed integers
Graphs: representations, DFS, and BFS

Programming Assignment 2 parts

1. balanced: needs a stack, demoed on Tuesday
2. bstLevelOrder: needs a queue (available in pa2/queue, will discuss today)
3. edgelist: will discuss today
4. isTree: needs either DFS (stack) or BFS (queue)
5. mst: a greedy algorithm
6. findCycle: needs either DFS (stack) or BFS (queue)
The solveMaze problem

- The adjacency matrix representation
- The query
Using `graphutils.h`

- The adjacency list representation
- The edgelist representation
- The query
A DFS approach for solving isTree (using recursion)

- Solution using DFS
- Using recursion
- The visited array of Booleans indicating if a node already visited
- Careful not to backtrack
- Where is the stack data structure??
A BFS approach for solving solveMaze (using a queue)

- Solution using BFS
- The parents array
- The queue
- Advanced topic: How to refactor code to make a generic queue using void*
Table of contents

Announcements
   Quizzes and programming assignments
   Reading and class session plan

Graphs: representations, DFS, and BFS
   The solveMaze problem
   Using graphutils.h
   A DFS approach for solving isTree (using recursion)
   A BFS approach for solving solveMaze (using a queue)

Bits and bytes
   Why binary
   Decimal, binary, octal, and hexadecimal
   Bitwise operations
   Representing characters

Integers and basic arithmetic
   Representing negative and signed integers
Why binary

Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - ... and represent and manipulate numbers, sets, strings, etc...
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires

Why binary

Figure: Rahul Sarpeshkar. Analog Versus Digital: Extrapolating from Electronics to Neurobiology. 1998.
Why binary

Digital encodings
Each doubling of either precision or range only needs one additional bit.

Analog encodings
Each doubling of either precision or range needs doubling of either area or power.
### Decimal, binary, octal, and hexadecimal

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Octal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0b0000</td>
<td>0o0</td>
<td>0x0</td>
</tr>
<tr>
<td>1</td>
<td>0b0001</td>
<td>0o1</td>
<td>0x1</td>
</tr>
<tr>
<td>2</td>
<td>0b0010</td>
<td>0o2</td>
<td>0x2</td>
</tr>
<tr>
<td>3</td>
<td>0b0011</td>
<td>0o3</td>
<td>0x3</td>
</tr>
<tr>
<td>4</td>
<td>0b0100</td>
<td>0o4</td>
<td>0x4</td>
</tr>
<tr>
<td>5</td>
<td>0b0101</td>
<td>0o5</td>
<td>0x5</td>
</tr>
<tr>
<td>6</td>
<td>0b0110</td>
<td>0o6</td>
<td>0x6</td>
</tr>
<tr>
<td>7</td>
<td>0b0111</td>
<td>0o7</td>
<td>0x7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Octal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0b1000</td>
<td>0o10</td>
<td>0x8</td>
</tr>
<tr>
<td>9</td>
<td>0b1001</td>
<td>0o11</td>
<td>0x9</td>
</tr>
<tr>
<td>10</td>
<td>0b1010</td>
<td>0o12</td>
<td>0xA</td>
</tr>
<tr>
<td>11</td>
<td>0b1011</td>
<td>0o13</td>
<td>0xB</td>
</tr>
<tr>
<td>12</td>
<td>0b1100</td>
<td>0o14</td>
<td>0xC</td>
</tr>
<tr>
<td>13</td>
<td>0b1101</td>
<td>0o15</td>
<td>0xD</td>
</tr>
<tr>
<td>14</td>
<td>0b1110</td>
<td>0o16</td>
<td>0xE</td>
</tr>
<tr>
<td>15</td>
<td>0b1111</td>
<td>0o17</td>
<td>0xF</td>
</tr>
</tbody>
</table>

In C, format specifiers for `printf()` and `fscanf()`:

1. decimal: `%d`
2. binary: none
3. octal: `%o`
4. hexadecimal: `%x`
Decimal, binary, octal, and hexadecimal

How to represent the range of unsigned char in each?
Unsigned char is one byte, 8 bits.

1. decimal: 0 to 255
2. binary: 0b0 to 0b11111111
3. octal: 0 to 0o377 (group by 3 bits)
4. hexadecimal: 0x00 to 0xFF (group by 4 bits)
Why are bitwise operations important?

- Network and UNIX settings using bit masks (e.g., umask)
- Hardware and microcontroller programming (e.g., Arduinos)
- Instruction set architecture encodings (e.g., ARM, x86)
Bitwise operations

\( \sim \): bitwise NOT

unsigned char \( a = 128 \)

\[
\begin{array}{c}
a = 0b1000\_0000 \\
\sim a = \sim 0b1000\_0000 \\
= 0b0111\_1111 \\
= 127
\end{array}
\]
Bitwise operations

&: bitwise AND

\[ 3 \& 1 = 0b11 \& 0b01 = 0b01 = 1 \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>( a &amp; b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Bitwise operations

|: bitwise OR

\[
3 | 1 = 0b11 | 0b01 \\
\quad = 0b11 \\
\quad = 3
\]

\[
2 | 1 = 0b10 | 0b01 \\
\quad = 0b11 \\
\quad = 3
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a | b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Bitwise operations

^: bitwise XOR

\[
3 \land 1 = 0b11 \land 0b01 = 0b10 = 2
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a ^ b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Don’t confuse bitwise operators with logical operators

**Bitwise operators**

- ~
- &
- |
- ^

**Logical operators**

- !
- &&
- ||
- != (for bool type)
Representing characters

Figure: ASCII character set. Image credit Wikimedia
Table of contents

Announcements
  Quizzes and programming assignments
  Reading and class session plan

Graphs: representations, DFS, and BFS
  The solveMaze problem
  Using graphutils.h
  A DFS approach for solving isTree (using recursion)
  A BFS approach for solving solveMaze (using a queue)

Bits and bytes
  Why binary
  Decimal, binary, octal, and hexadecimal
  Bitwise operations
  Representing characters

Integers and basic arithmetic
  Representing negative and signed integers
Representing negative and signed integers

Ways to represent negative numbers

1. Sign magnitude
2. 1’s complement
3. 2’s complement
Representing negative and signed integers

Sign magnitude
Flip leading bit.
Representing negative and signed integers

1’s complement

- Flip all bits
- Addition in 1’s complement is sound
- In this encoding there are 2 encodings for 0
- -0: 0b1111
- +0: 0b0000
Representing negative and signed integers

2’s complement

<table>
<thead>
<tr>
<th>signed char</th>
<th>weight in decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>00000100</td>
<td>4</td>
</tr>
<tr>
<td>00001000</td>
<td>8</td>
</tr>
<tr>
<td>00010000</td>
<td>16</td>
</tr>
<tr>
<td>00100000</td>
<td>32</td>
</tr>
<tr>
<td>01000000</td>
<td>64</td>
</tr>
<tr>
<td>10000000</td>
<td>-128</td>
</tr>
</tbody>
</table>

Table: Weight of each bit in a signed char type

- what is the most positive value you can represent? 127
- what is the most negative value you can represent? -128
- how to represent -1? 11111111
- how to represent -2? 11111110
Representing negative and signed integers

2’s complement

<table>
<thead>
<tr>
<th>signed char</th>
<th>weight in decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>00000100</td>
<td>4</td>
</tr>
<tr>
<td>00001000</td>
<td>8</td>
</tr>
<tr>
<td>00010000</td>
<td>16</td>
</tr>
<tr>
<td>00100000</td>
<td>32</td>
</tr>
<tr>
<td>01000000</td>
<td>64</td>
</tr>
<tr>
<td>10000000</td>
<td>-128</td>
</tr>
</tbody>
</table>

Table: Weight of each bit in a signed char type

- MSB: 1 for negative
- Take the 1’s complement number + 1
- Most important; good properties for digital logic
Importance of paying attention to limits of encoding

Figure: Image credit: CS:APP
Importance of paying attention to limits of encoding

Figure: Image credit: CS:APP
Importance of paying attention to limits of encoding

Figure: Image credit: CS:APP