COMP 40 Laboratory: Problem-Solving and Black Edges

September 23, 2011

(*Digression on compiling*: If you don't already have a compile script for this assignment, please copy your previous compile script and edit the two `case` statements that begin "`case $link in`".)

In this lab, please work with your programming partner. The staff are on hand primarily to answer your questions; raise hands early and often.

NOTE: As part of the lab, you will *submit three test cases*.

Understanding the problem

The problem is to "fix" a scanned image by removing *black edge pixels*. What is a black edge pixel? Your homework gives a precise inductive definition.

1. *Think of example inputs.*

   Begin with

   - Inputs that are as simple as possible (e.g., all-black image)
   - Other simple inputs (please come up with your own examples)
   - Pathological inputs (e.g., the scanner malfunctions and puts a black bar all the way across a scanned input—please think of additional examples)
   - Expected inputs

   Please think of several examples, and *write down what you think should happen*. This information will find its way into a design checklist.

2. *Do the problem by hand for simple inputs*.

   In addition to very simple inputs you think of, do some small random inputs. You can view or print such an input using the shell script in Figure 1:

   ```
   random-bitmap | pnmscale 40 | display -               # view
   random-bitmap | pnmscale 50 | pnmtops | lpr -Php118   # print
   random-bitmap | pnmtoplainpnm                         # view ASCII
   ```

   (The script is installed in the `/comp/40/bin` directory and should be made visible by `use comp40`.)

   Not all bitmaps are square:

   ```
   random-bitmap 16 9 | pnmscale 40 | display -
   ```

```
#!/bin/bash

function usage {
  echo "Usage: $(basename $0) [width [height [percent-black]]]" 1>&2
  exit 1
}

case $# in
  0|1|2|3) ;; # OK; do nothing
  *) usage ;;
esac

case $1,$2,$3 in
  *-*) usage ;;
esac

w="$1" # width
h="$2" # height
p="$3" # percent black

### use defaults for values not given on the command line

[ -z "$w" ] && w=10   # default width is 10
[ -z "$h" ] && h="$w" # default height makes a square bitmap
[ -z "$p" ] && p=50   # default bitmap is 50% black

# emit pbm(5) in the "plain" format

echo "P1"
echo "# random bitmap generated by: $(basename $0) $w $h $p"
echo "$w $h"

for ((i=0; i<w; i++)); do
  for ((j=0; j<h; j++)); do
    n=$(($RANDOM % 100))
    echo $((n < p))
  done
done
```

Figure 1: Shell script for generating random bitmaps

3. *Construct and submit test cases.*

   Before the lab is over, please use the `submit40-lab-unblack` program to submit three test cases and their solutions. We expect these files:

   ```
   test1.pbm   answer1.pbm
   test2.pbm   answer2.pbm
   test3.pbm   answer3.pbm
   ```

   Be sure you check these files with `display` to make sure they are formatted correctly!

   If you like, you may also include a file `TESTS` describing your tests.

   - An easy way to create a test case by hand is to use `pnmtoplainpnm` on a random bitmap, then edit the results in a text editor. You can display edited bitmaps using `pnmscale` and `display`.
   - *If you are not sure your test cases are right, ask a member of the course staff.*
   - Your test cases will be used to evaluate your solutions as well as everyone else's solutions.
   - If you develop a test case that exposes a fault in someone else's program, we will be impressed. Feel free to make your test cases as challenging as you like.

Stepwise refinement

4. *Can the problem be broken down into a sequence of simpler steps?*

   For the problem of the black edge pixels, I know two (very similar) solutions that involve breaking the problem into two steps.[1] (It's possible that you may have already broken the problem into steps mentally, so much so that you can't see the steps. If so, call for a member of the course staff.)

Identifying data in the problem

5. *What data do you see in the problem? Do different data structures go with different steps?*

   In this problem, understanding the data in the problem may be the most difficult part.

   Here are a couple of tricks:

   - *List the key words in the problem statement*
     Do these words suggest any additional data structures? Any algorithms?
   - *Would it be helpful to put any of the key items into set? A sequence? A finite map?*
     For example, the problem repeatedly mentions pixels. Probably there are one or more useful data structures involving pixels.

Revisit your examples, algorithmically

6. *Play computer*.

   Revisit your work from step 2. Can you identify an *algorithm* for working these examples? Try to "play computer" and see if your algorithmic ideas work on the examples.

7. *Identify invariant properties*.

Algorithms and data structures

8. *What algorithms will you use to help solve the problem?*

9. *What abstractions (black boxes) could help you solve the problem?*

Architecture of your solution

The word for "how black boxes and functions work together to solve a problem" is the *architecture* of that solution.

10. *What are the major components in your system and what are their* ***interfaces***?

    Functions have simple interfaces (a prototype and a contract). Abstractions (black boxes) have more complex interfaces.

    Simpler interfaces are better. In particular, it's better to have a few simple interfaces than to have one complex interface. *Each interface should correspond to a single subproblem.* Alternatively, "every module hides a secret."

    For example, my program for finding fingerprint groups has these components:

    - A separate, reusable module exporting one function for reading input lines of arbitrary length:

      ```
      extern char *getline40 (FILE *fp); // Read a line, no matter how long.
      ```

    - A single function which takes a string representing a line, plus pointers to containers for name and fingerprint. It either parses the line successfully, putting the two elements in their containers, or it returns `false`:

[1] A friend came up with a solution that works in a single step, using a very simple auxiliary data structure, but it exploits a programming technique you haven't learned yet.

```
static bool splitline(const char *line, const char ** fp, const char **name);
```

- Hanson's `List` and `Table` interfaces.
- A `struct` definition and apply function for use with `Table_map`, which prints a list of names as appropriate, then frees the list.
- A `main` function which orchestrates the work.

11. *How do the components interact?*

    For example, in my program for finding fingerprint groups, the `main` function depends on every other component, and almost all the components are independent. The exception is that the code for printing lists of names shares a secret with the code in the `main` function (which creates those lists of names). The shared secret is that both components know the invariants of the `Table` structure.