EEE2048: Computer Algorithms and Architecture
2019/20 Programming Assignment

Quadtree Encoding of a Binary Image

Deadline for Submission to SurreyLearn: **4:00pm, Tuesday 10th December 2019**

Introduction

Silhouettes contain much of the shape information necessary to recognise simple objects. For this reason they have been used widely in machine vision systems that check the shape of industrial parts or provide grasping information to robots. The digital version of a silhouette is a binary image i.e. an array of 1's and 0's. Figure 1 shows a digital image. In a binary image version of this, 0 represents a dark area and 1 represents a light area. The positions of pixels are defined with respect to the coordinate system shown in the figure.

A significant problem of image processing is the large amount of storage needed for image data and the consequent computational load of performing calculations on large numbers of pixels. There has therefore been much effort in developing data representations or codings which are more compact but still capture the relevant features of the data. One promising data structure is the *quadtree*. The quadtree encoding of the binary image shown in Figure 1 is given in Figure 2.

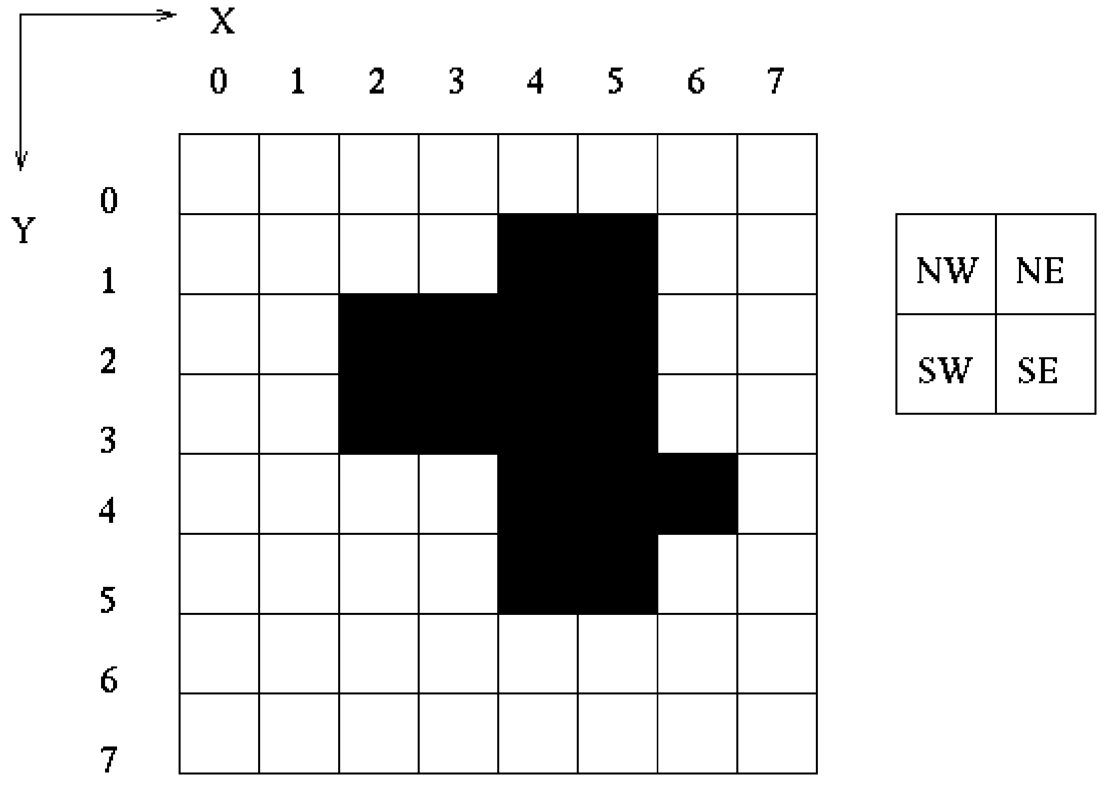

Figure 1: Binary image array

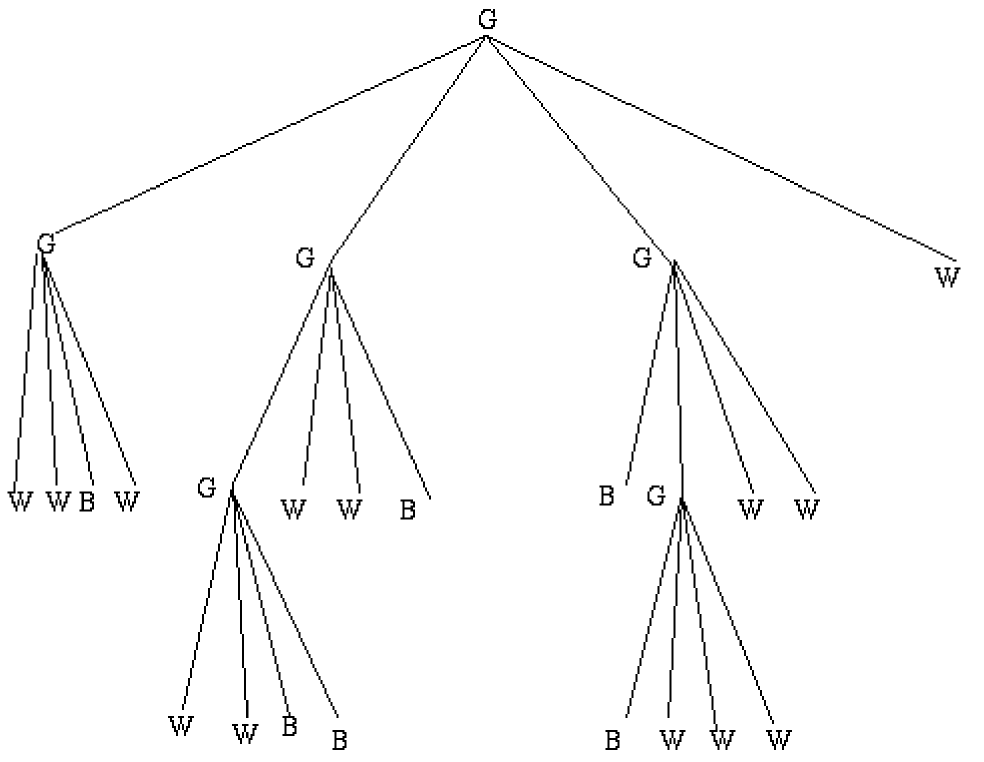

Figure 2: Quadtree corresponding to binary image array

Quadtrees are created from square images whose widths are integer powers of 2. Each node of the quadtree corresponds to a square area in the binary image. The node at level 0 of the tree corresponds to the whole image area. Child nodes correspond to the areas given by dividing the image into 4 non-overlapping quadrants. The quadrants are denoted NW (north-west), NE (north-east), SE (south-east) and SW (south-west) respectively, as shown in the inset part of Figure 1. Nodes at lower levels of the tree again correspond to splitting of parent areas into smaller quadrants. Each node is labelled to indicate the type of pixels contained within its area. Three labels are distinguished: W for white (all pixels in area are of value 1), B for black (all pixels in area are of value 0) and G for grey (area contains both pixels of value 1 and pixels of value 0). Splitting of areas into sub-quadrants terminates when nodes contain only one type of pixel i.e. when nodes are labelled B or W. The resultant data structure generally contains far fewer nodes than the number of pixels in the original image. Nodes near the root of the tree correspond to large blocks of the image and if the binary image contains objects which are large and convex then they will be represented naturally by few nodes.

Programming Task

The programming assignment consists of writing procedures which will

1. Read a square binary image from a text file into an array structure,
2. Generate a pointer based quadtree from the binary image,
3. Traverse through the pointer based quadtree and print out for each black terminal node the position of the top left pixel of the quadrant and the width of the quadrant.

The program must read the image data from a text file which contains coded information on the width of the square image and the number and position of black pixels in the image. The image file contains integers and the file has the following format:

image width in pixels

number of black pixels

x y location of each black pixel

For the example of the image in Figure 1, the file would contain the data shown in Figure 3.

```
8
15
4 1
5 1
2 2
3 2
4 2
5 2
2 3
3 3
4 3
5 3
4 4
5 4
6 4
4 5
5 5
```

Figure 3: Input file example

Use an integer array as an intermediate data structure to help in generating the pointer based quadtree. Producing the pointer based quadtree involves taking the array and generating a node to represent it. The array can then be tested for colour uniformity. If it is not of a uniform colour then the array must be divided into four quadrants and nodes created for these. Each quadrant must then be tested and the same process recursively performed until all terminal nodes of the tree represent uniformly coloured areas. Internally your program should represent each node of the quadtree as a `struct` datatype containing 8 fields. The first four pieces of information are pointers to the four children of the node (NW, NE, SE, SW), the next piece of information is the colour of the node (i.e. B, W or G), and the final three pieces of information are the width and the location of the block of pixels that the node represents (the location is represented by the x and y coordinates of the top left pixel in that quadrant).

Input parameters

Your program should read data from a file with path specified using a command line parameter. This parameters should be the only parameter expected by the program, and should be required to be specified immediately after the executable name at run-time e.g. `quadtree my_image.txt`

The input file will contain the image data in a format as illustrated in Figure 3 and in the example file `input.txt` provided on SurreyLearn.

Output data

Your program should print out on the screen the list of all terminal black nodes. The output should indicate for each black terminal node the position of the top left pixel of the quadrant and the width of the quadrant. For example, the output of processing the example file `input.txt` provided on SurreyLearn is shown in Figure 4 (also available on SurreyLearn in file `output.txt`). It should be noted that there is no specific requirement in terms of the order in which the black terminal nodes are printed out. Therefore any output where the lines shown in Figure 4 are permuted would be equally valid.

```
Black terminal node at position (2,2) with width 2
Black terminal node at position (4,1) with width 1
Black terminal node at position (5,1) with width 1
Black terminal node at position (4,2) with width 2
Black terminal node at position (4,4) with width 2
Black terminal node at position (6,4) with width 1
```

Figure 4: Example screen output when processing input file shown in Figure 3

Coding standards

All programs should be clearly structured with good indentation to indicate the structure. Aim to make all code largely self-documenting. Comments should indicate the purpose and means of implementation of each program section. Procedures should be kept self-contained with well-defined interfaces to other procedures. Each procedure should contain a commented header explaining:

- The purpose of the procedure or function,
- The description of all input and output parameters.

Deliverables

When trying to solve a problem using a computer, the major effort should be directed at the design of the algorithm and choice of the best data structures. Actual coding is only a small part of the exercise. The deliverables for the project will be:

1) ***A short written report***

The report should include:

- A section giving an overview of the program design. This should include a description of the purpose and functionality of each function. It should give a brief structural description of the program with a simple top-down structured diagram which indicates the inter-relationship of functions (calling hierarchy).
- A testing section presenting example results of the run of the program. This should demonstrate the correct operation of the program on a range of pertinently chosen input test images.
- A section analysing empirically the computational complexity of the algorithm for generating a pointer-based quadtree representation from a binary image (assuming the image has already been read from the file and ignoring the printing of the black terminal nodes). This should include a plot of the measured execution time as a function of the input image size (its total number of pixels) in the average case.

The report should not exceed 4 pages in length (excluding the cover page and table of contents). Any pages beyond 4 will be ignored for assessment purposes. The report should be submitted to SurreyLearn in pdf format.

2) ***C source code programs***

These should be submitted to SurreyLearn. Submitted files should include quadtree.c and any necessary header files included from the C code that are not standard C header files.

The code must be capable of being compiled using the Linux machines in the Otter lab. If students develop code elsewhere then they will need to ensure it meets this requirement prior to submitting it.

Assessment Criteria

Criteria for the assessment of the coursework are: clarity of program description, inclusion of suitable results, appropriate analysis of results, and presentation.

Assessment of the code will involve a mix of automated test procedures and human assessment. Checking will test for compilation warnings and errors, run-time errors and correctness of information. Human assessment will consider structure, design and clarity of the solution. Aspects which the human marker will pay attention to include: modularity of solution, internal re-use of code, good commenting of code, sensible and clear layout, avoidance of global declarations, appropriate use of data-types, good error checking, easily understandable output and efficiency of implementation. You should aim to make the code easy to use and modify. Accuracy of output will be checked manually, therefore you need not worry about formatting the output to exactly match the example provided above (for example, white space or carriage return characters), as long as your program clearly returns the expected information.

You must work individually and independently to develop the code and produce the report. Source code will be compared using TurnItIn to check that it is original code and has not been written in collaboration with other students.

Marking Scheme

Criteria	Maximum marks	Explanation
Tests: Compilation	5	Code compiles cleanly without errors or warnings.
Tests: Execution	5	Code executes cleanly for a variety of test cases, including “no data” and “bad-data”.
Tests: Accuracy of output results	25	Results are output correctly for a variety of test cases, including all the information specified (5 tests will be performed with a maximum of 4 marks allocated to each test).
Code: Modularisation	5	Code is broken down into sensible functions.
Code: Variables	5	Appropriate data types used, global variables avoided where possible, any dynamically allocated memory is freed.
Code: Error checking	5	Anticipated errors such as incorrect data, file access problems or failure for calls to malloc are handled so that the program exits cleanly.
Code: Control flow	5	Flow through the program is correct, there are no unnecessary loops or blocks of code.
Code: Code commenting	5	Comments are thorough and explain all key aspects of processing.
Code: Code compliance	5	Code complies to given specifications (node representation, functionalities, output formatting).
Code: Code layout	5	Code is indented appropriately so that code in a given block is aligned and indented.
Report: description of program design	10	Description of the program and of the purpose and functionality of each function including diagram indicating the inter-relationship of functions (calling hierarchy).
Report: program testing	10	Example results illustrating the operation of the program on a range of pertinently chosen input test images.
Report: computational complexity analysis	10	Plot showing the measured execution time for the quadtree construction as a function of the input image size (i.e. its total number of pixels) in the average case. Discussion of the empirically obtained run-time behaviour.

Tips

* Remember the steps in problem solving using computers: defining the problem, outlining a solution, developing an algorithm, programming the algorithm and testing the program.

* Feel free to discuss the problem with your fellow students or in a tutorial. However, exchange of ideas is very different from copying of code and the latter is strongly discouraged and will be dealt with via the University procedures for academic misconduct.

* Do the bookwork first - make sure that your algorithm is correct, and familiarise yourself with its behaviour, before you start to code it.

* Design your program on paper before starting to work on the computer. Only with considerable experience is it possible to hand-craft code directly into the editor, and even then the probability of success is considerably diminished.

* In the early stages of testing include extra diagnostic statements in your code so that you can confirm in the initial stages that your algorithm is working as expected.