DATA STRUCTURE CONVERSION

Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: hjs@umiacs.umd.edu

Copyright © 2000 Hanan Samet

These notes may not be reproduced by any means (mechanical or electronic or any other) without the express written permission of Hanan Samet
QUADTREES FROM BINARY ARRAYS

MacDraw figure
QUADTREES FROM BINARY ARRAYS

• Traverse pixels in Morton order

• No need to merge four leaf nodes of the same color as only create maximal leaf nodes

• Execution time proportional to number of pixels

• Ex:
QUADTREES FROM BINARY ARRAYS

• Traverse pixels in Morton order

• No need to merge four leaf nodes of the same color as only create maximal leaf nodes

• Execution time proportional to number of pixels

• Ex:
QUADTREES FROM BINARY ARRAYS

- Traverse pixels in Morton order
- No need to merge four leaf nodes of the same color as only create maximal leaf nodes
- Execution time proportional to number of pixels
- Ex:
QUADTREES FROM BINARY ARRAYS

- Traverse pixels in Morton order
- No need to merge four leaf nodes of the same color as only create maximal leaf nodes
- Execution time proportional to number of pixels
- Ex:

```
1  2  5  6  17 18 21 22
3  4  7  8  19 20 23 24
9 10 13 14 25 26 29 30
11 12 15 16 27 28 31 32
33 34 37 38 49 50 53 54
35 36 39 40 51 52 55 56
41 42 45 46 57 58 61 62
43 44 47 48 59 60 63 64
```
QUADTREES FROM BINARY ARRAYS

- Traverse pixels in Morton order
- No need to merge four leaf nodes of the same color as only create maximal leaf nodes
- Execution time proportional to number of pixels

Ex:

```
1 2 5 6 17 18 21 22
3 4 7 8 19 20 23 24
9 10 13 14 25 26 29 30
11 12 15 16 27 28 31 32
33 34 37 38 49 50 53 54
35 36 39 40 51 52 55 56
41 42 45 46 57 58 61 62
43 44 47 48 59 60 63 64
```
QUADTREES FROM BINARY ARRAYS

- Traverse pixels in Morton order
- No need to merge four leaf nodes of the same color as only create maximal leaf nodes
- Execution time proportional to number of pixels
- Ex:

```
1 2 5 6 17 18 21 22
3 4 7 8 19 20 23 24
9 10 13 14 25 26 29 30
11 12 15 16 27 28 31 32
33 34 37 38 49 50 53 54
35 36 39 40 51 52 55 56
41 42 45 46 57 58 61 62
43 44 47 48 59 60 63 64
```
QUADTREES FROM BINARY ARRAYS

• Traverse pixels in Morton order
• No need to merge four leaf nodes of the same color as only create maximal leaf nodes
• Execution time proportional to number of pixels
• Ex:
BUILDING QUADTREES FROM RASTERS

• Using a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern and southern directions and adding them if they are not present

Algorithm: process array row by row

• Odd row — add nodes to tree
• Even row — add nodes to tree; attempt to merge
• Execution time is $O$(number of pixels)

• Ex:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
</tr>
<tr>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
</tr>
</tbody>
</table>
BUILDING QUADTREES FROM RASTERS

- Using a variant of `EQUAL_LATERAL_NEIGHBOR` to find neighbors in the eastern and southern directions and adding them if they are not present.

Algorithm: process array row by row

- Odd row — add nodes to tree
- Even row — add nodes to tree; attempt to merge
- Execution time is $O(\text{number of pixels})$

Ex:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
</tr>
<tr>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
</tr>
</tbody>
</table>

Result of processing the first row
BUILDING QUADTREES FROM RASTERS

- Using a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern and southern directions and adding them if they are not present

Algorithm: process array row by row

- Odd row — add nodes to tree
- Even row — add nodes to tree; attempt to merge
- Execution time is $O(\text{number of pixels})$

Ex:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
</tr>
<tr>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
</tr>
</tbody>
</table>

- Result of processing the first row
- Result just prior to merging nodes corresponding to pixels 1, 2, 9, and 10
BUILDING QUADTREES FROM RASTERS

- Using a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern and southern directions and adding them if they are not present

Algorithm: process array row by row

- Odd row — add nodes to tree
- Even row — add nodes to tree; attempt to merge
- Execution time is $O(\text{number of pixels})$

Ex:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
</tr>
<tr>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
</tr>
</tbody>
</table>

- Result of processing the first row
- Result just prior to merging nodes corresponding to pixels 1, 2, 9, and 10
- Result of processing the second row
BUILDING QUADTREES FROM RASTERS

- Using a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern and southern directions and adding them if they are not present

Algorithm: process array row by row

- Odd row — add nodes to tree
- Even row — add nodes to tree; attempt to merge
- Execution time is $O(\text{number of pixels})$

Ex:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
</tr>
<tr>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
</tr>
</tbody>
</table>

- Result of processing the first row
- Result just prior to merging nodes corresponding to pixels 1, 2, 9, and 10
- Result of processing the second row
- Final quadtree

Copyright © 2002 by Hanan Samet
INITIAL RASTER TO QUADTREE BUILDING SEQUENCE

- Ex: build quadtree for first four pixels in the first row

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>
INITIAL RASTER TO QUADTREE BUILDING SEQUENCE

• Ex: build quadtree for first four pixels in the first row

• Use a variant of **EQUAL_LATERAL_NEIGHBOR** to find neighbors in the eastern direction and add them if they are not present
INITIAL RASTER TO QUADTREE BUILDING SEQUENCE

• Ex: build quadtree for first four pixels in the first row

• Use a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern direction and add them if they are not present
OPTIMAL QUADTREE BUILDING

$I = \text{cost of a block (i.e., node) insertion operation}$
$c = \text{cost of examining a pixel}$

$2^n \times 2^n \text{ image}$

$N = \text{number of blocks in the output quadtree}$

Naive algorithm:
- Examine each pixel and insert it into the quadtree
- Cost $= 2^{2n} \cdot (c + I)$

Optimal algorithm:
- Examine each pixel but only insert largest block for which current pixel is the first (i.e., upper leftmost) pixel
- Avoids the need for merging
- Cost $= c \cdot 2^{2n} + I \cdot N$
- Speedup is obvious because $c << I$ and $N << 2^{2n}$

<table>
<thead>
<tr>
<th>Map Name</th>
<th>Num of Blocks</th>
<th>Num of inserts</th>
<th>Time (secs)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Naive</td>
<td>Optimal</td>
<td>Naive</td>
</tr>
<tr>
<td>Floodplain</td>
<td>5266</td>
<td>180000</td>
<td>2352</td>
<td>413.2</td>
</tr>
<tr>
<td>Topography</td>
<td>24859</td>
<td>180000</td>
<td>12400</td>
<td>429.8</td>
</tr>
<tr>
<td>Landuse</td>
<td>28447</td>
<td>180000</td>
<td>14675</td>
<td>436.7</td>
</tr>
<tr>
<td>Center</td>
<td>4687</td>
<td>262144</td>
<td>2121</td>
<td>603.8</td>
</tr>
<tr>
<td>Pebble</td>
<td>44950</td>
<td>262144</td>
<td>20770</td>
<td>630.1</td>
</tr>
<tr>
<td>Stone</td>
<td>31969</td>
<td>262144</td>
<td>14612</td>
<td>629.5</td>
</tr>
</tbody>
</table>

Storage requirements:
- Higher for optimal algorithm since it must keep track of active blocks
- Size of intermediate quadtree is smaller as there is no merging
OPTIMAL ALGORITHM

• When building a quadtree there is a processed and unprocessed part of the image; blocks covering both parts have been assigned

Def: a block is *active* if at least one pixel, but not all of the pixels, covered by the block has been processed and it differs in color from a block that contains it

Ex: blocks A and B are both active after processing pixel 5

Algorithm: traverse image in raster scan order; for each pixel:
OPTIMAL ALGORITHM

• When building a quadtree there is a processed and unprocessed part of the image; blocks covering both parts have been assigned

Def: a block is *active* if at least one pixel, but not all of the pixels, covered by the block has been processed and it differs in color from a block that contains it

Ex: blocks A and B are both active after processing pixel 5

Algorithm: traverse image in raster scan order; for each pixel:

1. if the pixel is the same color as the appropriate active block, then do nothing
   Ex: pixel 2
OPTIMAL ALGORITHM

• When building a quadtree there is a processed and unprocessed part of the image; blocks covering both parts have been assigned

Def: a block is *active* if at least one pixel, but not all of the pixels, covered by the block has been processed and it differs in color from a block that contains it

Ex: blocks A and B are both active after processing pixel 5

Algorithm: traverse image in raster scan order; for each pixel:

1. if the pixel is the same color as the appropriate active block, then do nothing
   Ex: pixel 2

2. else insert the largest possible block for which this is the first (i.e., upper leftmost) pixel and (if it is not a $1 \times 1$ block) add it to the list of active blocks
   Ex: pixel 3
OPTIMAL ALGORITHM

- When building a quadtree there is a processed and unprocessed part of the image; blocks covering both parts have been assigned.

Def: a block is *active* if at least one pixel, but not all of the pixels, covered by the block has been processed and it differs in color from a block that contains it.

Ex: blocks A and B are both active after processing pixel 5.

Algorithm: traverse image in raster scan order; for each pixel:

1. if the pixel is the same color as the appropriate active block, then do nothing.
   Ex: pixel 2.

2. else insert the largest possible block for which this is the first (i.e., upper leftmost) pixel and (if it is not a 1×1 block) add it to the list of active blocks.
   Ex: pixel 3.

3. remove any active blocks for which this is the last (i.e., lower rightmost) pixel.
   Ex: pixel 12.
IMPLEMENTATION

- The algorithm must keep track of all the active blocks.
- For a $2^n \times 2^n$ image the number of active blocks is $\leq 2^{n-1}$.
- Use a data structure called TABLE to keep a list of the active blocks organized by levels:
  1. one row for each level in quadtree (except for level 0)
  2. row $i$ has $2^{n-i}$ entries
  3. for a pixel in column $j$ of image, the color of the active block (if there is one) at level $i$ is in entry $\lfloor j / 2^i \rfloor$ of row $i$

- Problem: must search for the smallest active block covering a pixel:
  1. start at lowest level and stop at first non-empty entry
  2. slow — could require $n$ steps
- Solution: use an access array LIST:
  1. LIST[$\lfloor j / 2 \rfloor$] indicates the row of TABLE corresponding to the smallest active block that contains the pixel at column $j$
  2. initially, each entry of LIST points to $n$ — i.e., the row of TABLE corresponding to the root of a $2^n \times 2^n$ image
  3. as active blocks are inserted and completed (i.e., deleted from the active block data structure), TABLE and LIST are updated.

- Ex: blocks A, E, H, and I are all active after processing pixel X, while blocks A, E, and I all cover this pixel.

![Diagram showing active blocks A, E, H, and I covering pixel X.](image)
EXAMPLE OF OPTIMAL QUADTREE BUILDING

(\(X,Y\)) \(2^3 \times 2^3 \) image \(X\)

\[
\begin{array}{|c|}
\hline
A \\
\hline
\end{array}
\]

<table>
<thead>
<tr>
<th>Pixel</th>
<th>Action</th>
<th>Size</th>
<th>Active Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>insert WHITE block A</td>
<td>8x8</td>
<td>3:A</td>
</tr>
</tbody>
</table>

Copyright © 2002 by Hanan Samet
# Example of Optimal Quadtree Building

A quadtree is a tree data structure in which each internal node has exactly four children. It is commonly used to partition a two-dimensional space into Rectangular regions for efficient querying of spatial data. The example below illustrates how a quadtree is built from a 2D image.

![2D image](image.png)

### 2D Image

- **Image Size:** $2^3 \times 2^3$ pixels
- **Origin:** $(0,0)$

### Quadtree Representation

<table>
<thead>
<tr>
<th>Pixel</th>
<th>Action</th>
<th>Size</th>
<th>Active Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>insert WHITE block A</td>
<td>8x8</td>
<td>3:A</td>
</tr>
<tr>
<td>(2,4)</td>
<td>insert BLACK block B</td>
<td>2x2</td>
<td>3:A 1:B</td>
</tr>
</tbody>
</table>

The table above lists the pixels and their corresponding actions, sizes, and active blocks in the quadtree.

---

Copyright © 2002 by Hanan Samet
EXAMPLE OF OPTIMAL QUADTREE BUILDING

Pixel | Action | Size | Active Blocks
--- | --- | --- | ---
(0,0) | insert WHITE block A | 8×8 | 3:A
(2,4) | insert BLACK block B | 2×2 | 3:A 1:B
(2,6) | insert BLACK block C | 2×2 | 3:A 1:B,C
EXAMPLE OF OPTIMAL QUADTREE BUILDING

(X,Y) → 2³×2³ image → X

Y

<table>
<thead>
<tr>
<th>Pixel</th>
<th>Action</th>
<th>Size</th>
<th>Active Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>insert WHITE block A</td>
<td>8×8</td>
<td>3:A</td>
</tr>
<tr>
<td>(2,4)</td>
<td>insert BLACK block B</td>
<td>2×2</td>
<td>3:A 1:B</td>
</tr>
<tr>
<td>(2,6)</td>
<td>insert BLACK block C</td>
<td>2×2</td>
<td>3:A 1:B,C</td>
</tr>
<tr>
<td>(3,5)</td>
<td>remove B from active</td>
<td></td>
<td>3:A 1:C</td>
</tr>
</tbody>
</table>

Copyright © 2002 by Hanan Samet
EXAMPLE OF OPTIMAL QUADTREE BUILDING

\[(X,Y) \xrightarrow{2^3 \times 2^3 \text{ image}} X\]

<table>
<thead>
<tr>
<th>Pixel</th>
<th>Action</th>
<th>Size</th>
<th>Active Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>insert WHITE block A</td>
<td>8x8</td>
<td>3:A</td>
</tr>
<tr>
<td>(2,4)</td>
<td>insert BLACK block B</td>
<td>2x2</td>
<td>3:A 1:B</td>
</tr>
<tr>
<td>(2,6)</td>
<td>insert BLACK block C</td>
<td>2x2</td>
<td>3:A 1:B,C</td>
</tr>
<tr>
<td>(3,5)</td>
<td>remove B from active</td>
<td></td>
<td>3:A 1:C</td>
</tr>
<tr>
<td>(3,7)</td>
<td>remove C from active</td>
<td></td>
<td>3:A</td>
</tr>
</tbody>
</table>
EXAMPLE OF OPTIMAL QUADTREE BUILDING

<table>
<thead>
<tr>
<th>Pixel</th>
<th>Action</th>
<th>Size</th>
<th>Active Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>insert WHITE block A</td>
<td>8x8</td>
<td>3:A</td>
</tr>
<tr>
<td>(2,4)</td>
<td>insert BLACK block B</td>
<td>2x2</td>
<td>3:A 1:B</td>
</tr>
<tr>
<td>(2,6)</td>
<td>insert BLACK block C</td>
<td>2x2</td>
<td>3:A 1:B,C</td>
</tr>
<tr>
<td>(3,5)</td>
<td>remove B from active</td>
<td></td>
<td>3:A 1:C</td>
</tr>
<tr>
<td>(3,7)</td>
<td>remove C from active</td>
<td></td>
<td>3:A</td>
</tr>
<tr>
<td>(4,3)</td>
<td>insert BLACK block D</td>
<td></td>
<td>3:A</td>
</tr>
</tbody>
</table>
**EXAMPLE OF OPTIMAL QUADTREE BUILDING**

![23x23 image](image.png)

<table>
<thead>
<tr>
<th>Pixel</th>
<th>Action</th>
<th>Size</th>
<th>Active Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>insert WHITE block A</td>
<td>8x8</td>
<td>3:A</td>
</tr>
<tr>
<td>(2,4)</td>
<td>insert BLACK block B</td>
<td>2x2</td>
<td>3:A 1:B</td>
</tr>
<tr>
<td>(2,6)</td>
<td>insert BLACK block C</td>
<td>2x2</td>
<td>3:A 1:B,C</td>
</tr>
<tr>
<td>(3,5)</td>
<td>remove B from active</td>
<td>2x2</td>
<td>3:A 1:C</td>
</tr>
<tr>
<td>(3,7)</td>
<td>remove C from active</td>
<td></td>
<td>3:A</td>
</tr>
<tr>
<td>(4,3)</td>
<td>insert BLACK block D</td>
<td></td>
<td>3:A</td>
</tr>
<tr>
<td>(4,4)</td>
<td>insert BLACK block E</td>
<td>4x4</td>
<td>3:A 2:E</td>
</tr>
</tbody>
</table>
EXAMPLE OF OPTIMAL QUADTREE BUILDING

Pixel | Action | Size | Active Blocks
--- | --- | --- | ---
(0,0) | insert WHITE block A | 8×8 | 3:A
(2,4) | insert BLACK block B | 2×2 | 3:A 1:B
(2,6) | insert BLACK block C | 2×2 | 3:A 1:B,C
(3,5) | remove B from active | | 3:A 1:C
(3,7) | remove C from active | | 3:A
(4,3) | insert BLACK block D | | 3:A
(4,4) | insert BLACK block E | 4×4 | 3:A 2:E
(5,2) (5,3) | insert BLACK blocks F, G | | 3:A 2:E
### Example of Optimal Quadtree Building

A 2^3 × 2^3 image is shown with a quadtree representation on the right. The quadtree is built by inserting black and white blocks and removing blocks from the active set.

<table>
<thead>
<tr>
<th>Pixel</th>
<th>Action</th>
<th>Size</th>
<th>Active Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>insert WHITE block A</td>
<td>8x8</td>
<td>3:A</td>
</tr>
<tr>
<td>(2,4)</td>
<td>insert BLACK block B</td>
<td>2x2</td>
<td>3:A 1:B</td>
</tr>
<tr>
<td>(2,6)</td>
<td>insert BLACK block C</td>
<td>2x2</td>
<td>3:A 1:B,C</td>
</tr>
<tr>
<td>(3,5)</td>
<td>remove B from active</td>
<td>2x2</td>
<td>3:A 1:C</td>
</tr>
<tr>
<td>(3,7)</td>
<td>remove C from active</td>
<td></td>
<td>3:A</td>
</tr>
<tr>
<td>(4,3)</td>
<td>insert BLACK block D</td>
<td>3x3</td>
<td>3:A</td>
</tr>
<tr>
<td>(4,4)</td>
<td>insert BLACK block E</td>
<td>4x4</td>
<td>3:A 2:E</td>
</tr>
<tr>
<td>(5,2) (5,3)</td>
<td>insert BLACK blocks F, G</td>
<td></td>
<td>3:A 2:E</td>
</tr>
<tr>
<td>(6,2)</td>
<td>insert BLACK block H</td>
<td>2x2</td>
<td>3:A 2:E 1:H</td>
</tr>
</tbody>
</table>

Copyright © 2002 by Hanan Samet
EXAMPLE OF OPTIMAL QUADTREE BUILDING

<table>
<thead>
<tr>
<th>Pixel</th>
<th>Action</th>
<th>Size</th>
<th>Active Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>insert WHITE block A</td>
<td>8x8</td>
<td>3:A</td>
</tr>
<tr>
<td>(2,4)</td>
<td>insert BLACK block B</td>
<td>2x2</td>
<td>3:A 1:B</td>
</tr>
<tr>
<td>(2,6)</td>
<td>insert BLACK block C</td>
<td>2x2</td>
<td>3:A 1:B,C</td>
</tr>
<tr>
<td>(3,5)</td>
<td>remove B from active</td>
<td></td>
<td>3:A 1:C</td>
</tr>
<tr>
<td>(3,7)</td>
<td>remove C from active</td>
<td></td>
<td>3:A</td>
</tr>
<tr>
<td>(4,3)</td>
<td>insert BLACK block D</td>
<td></td>
<td>3:A</td>
</tr>
<tr>
<td>(4,4)</td>
<td>insert BLACK block E</td>
<td>4x4</td>
<td>3:A 2:E</td>
</tr>
<tr>
<td>(5,2) (5,3)</td>
<td>insert BLACK blocks F, G</td>
<td></td>
<td>3:A 2:E</td>
</tr>
<tr>
<td>(6,2)</td>
<td>insert BLACK block H</td>
<td>2x2</td>
<td>3:A 2:E 1:H</td>
</tr>
<tr>
<td>(6,6)</td>
<td>insert WHITE block I</td>
<td>2x2</td>
<td>3:A 2:E 1:H,I</td>
</tr>
</tbody>
</table>
EXAMPLE OF OPTIMAL QUADTREE BUILDING

<table>
<thead>
<tr>
<th>Pixel</th>
<th>Action</th>
<th>Size</th>
<th>Active Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>insert WHITE block A</td>
<td>8×8</td>
<td>3:A</td>
</tr>
<tr>
<td>(2,4)</td>
<td>insert BLACK block B</td>
<td>2×2</td>
<td>3:A 1:B</td>
</tr>
<tr>
<td>(2,6)</td>
<td>insert BLACK block C</td>
<td>2×2</td>
<td>3:A 1:B,C</td>
</tr>
<tr>
<td>(3,5)</td>
<td>remove B from active</td>
<td></td>
<td>3:A 1:C</td>
</tr>
<tr>
<td>(3,7)</td>
<td>remove C from active</td>
<td></td>
<td>3:A</td>
</tr>
<tr>
<td>(4,3)</td>
<td>insert BLACK block D</td>
<td>3×3</td>
<td>3:A</td>
</tr>
<tr>
<td>(4,4)</td>
<td>insert BLACK block E</td>
<td>4×4</td>
<td>3:A 2:E</td>
</tr>
<tr>
<td>(5,2)</td>
<td>insert BLACK blocks F, G</td>
<td></td>
<td>3:A 2:E</td>
</tr>
<tr>
<td>(5,3)</td>
<td>insert BLACK blocks F, G</td>
<td></td>
<td>3:A 2:E</td>
</tr>
<tr>
<td>(6,2)</td>
<td>insert BLACK block H</td>
<td>2×2</td>
<td>3:A 2:E 1:H</td>
</tr>
<tr>
<td>(6,6)</td>
<td>insert WHITE block I</td>
<td>2×2</td>
<td>3:A 2:E 1:H,I</td>
</tr>
<tr>
<td>(7,3)</td>
<td>remove H from active</td>
<td></td>
<td>3:A 2:E 1:I</td>
</tr>
</tbody>
</table>
EXAMPLE OF OPTIMAL QUADTREE BUILDING

<table>
<thead>
<tr>
<th>Pixel</th>
<th>Action</th>
<th>Size</th>
<th>Active Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>insert WHITE block A</td>
<td>8x8</td>
<td>3:A</td>
</tr>
<tr>
<td>(2,4)</td>
<td>insert BLACK block B</td>
<td>2x2</td>
<td>3:A 1:B</td>
</tr>
<tr>
<td>(2,6)</td>
<td>insert BLACK block C</td>
<td>2x2</td>
<td>3:A 1:B,C</td>
</tr>
<tr>
<td>(3,5)</td>
<td>remove B from active</td>
<td></td>
<td>3:A 1:C</td>
</tr>
<tr>
<td>(3,7)</td>
<td>remove C from active</td>
<td></td>
<td>3:A</td>
</tr>
<tr>
<td>(4,3)</td>
<td>insert BLACK block D</td>
<td></td>
<td>3:A</td>
</tr>
<tr>
<td>(4,4)</td>
<td>insert BLACK block E</td>
<td>4x4</td>
<td>3:A 2:E</td>
</tr>
<tr>
<td>(5,2)</td>
<td>insert BLACK blocks F, G</td>
<td></td>
<td>3:A 2:E</td>
</tr>
<tr>
<td>(6,2)</td>
<td>insert BLACK block H</td>
<td>2x2</td>
<td>3:A 2:E 1:H</td>
</tr>
<tr>
<td>(6,6)</td>
<td>insert WHITE block I</td>
<td>2x2</td>
<td>3:A 2:E 1:H, I</td>
</tr>
<tr>
<td>(7,3)</td>
<td>remove H from active</td>
<td></td>
<td>3:A 2:E 1:I</td>
</tr>
<tr>
<td>(7,5)</td>
<td>insert WHITE block J</td>
<td></td>
<td>3:A 2:E 1:I</td>
</tr>
</tbody>
</table>
### Example of Optimal Quadtree Building

#### Diagram:
- **Image Size:** $2^3 \times 2^3$
- **Axes:** $(X, Y)$

#### Table:
<table>
<thead>
<tr>
<th>Pixel</th>
<th>Action</th>
<th>Size</th>
<th>Active Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>insert WHITE block A</td>
<td>8x8</td>
<td>3:A</td>
</tr>
<tr>
<td>(2,4)</td>
<td>insert BLACK block B</td>
<td>2x2</td>
<td>3:A 1:B</td>
</tr>
<tr>
<td>(2,6)</td>
<td>insert BLACK block C</td>
<td>2x2</td>
<td>3:A 1:B,C</td>
</tr>
<tr>
<td>(3,5)</td>
<td>remove B from active</td>
<td></td>
<td>3:A 1:C</td>
</tr>
<tr>
<td>(3,7)</td>
<td>remove C from active</td>
<td></td>
<td>3:A</td>
</tr>
<tr>
<td>(4,3)</td>
<td>insert BLACK block D</td>
<td>3x3</td>
<td>3:A</td>
</tr>
<tr>
<td>(4,4)</td>
<td>insert BLACK block E</td>
<td>4x4</td>
<td>3:A 2:E</td>
</tr>
<tr>
<td>(5,2)</td>
<td>insert BLACK blocks F, G</td>
<td></td>
<td>3:A 2:E</td>
</tr>
<tr>
<td>(5,3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6,2)</td>
<td>insert BLACK block H</td>
<td>2x2</td>
<td>3:A 2:E 1:H</td>
</tr>
<tr>
<td>(6,6)</td>
<td>insert WHITE block I</td>
<td>2x2</td>
<td>3:A 2:E 1:H,I</td>
</tr>
<tr>
<td>(7,3)</td>
<td>remove H from active</td>
<td></td>
<td>3:A 2:E 1:I</td>
</tr>
<tr>
<td>(7,5)</td>
<td>insert WHITE block J</td>
<td></td>
<td>3:A 2:E 1:I</td>
</tr>
<tr>
<td>(7,7)</td>
<td>remove I, E, A from active</td>
<td></td>
<td>—</td>
</tr>
</tbody>
</table>
CONVERTING QUADTREES TO RASTERS

- Useful when outputting an image

- Generate row-by-row by visiting each node once for every row that intersects it — $2^k$ visits to a $2^k \times 2^k$ block

- Preferable to generating the entire array at once and then outputting as it takes too much memory to store the array

- Two approaches
  1. top-down: starts at the root each time it visits a node in the row
  2. bottom-up: use neighbor-finding techniques (variant of \texttt{EQUAL\_LATERAL\_NEIGHBOR}) to visit adjacent nodes in the row

- Ex:
CONVERTING QUADTREES TO RASTERS

• Useful when outputting an image

• Generate row-by-row by visiting each node once for every row that intersects it — $2^k$ visits to a $2^k \times 2^k$ block

• Preferable to generating the entire array at once and then outputting as it takes too much memory to store the array

• Two approaches
  1. top-down: starts at the root each time it visits a node in the row
  2. bottom-up: use neighbor-finding techniques (variant of EQUAL_LATERAL_NEIGHBOR) to visit adjacent nodes in the row

• Ex:
COMPLEXITY OF CONVERTING QUADTREES TO RASTERS

- Assume a $2^n \times 2^n$ image

- Top-down algorithm visits $(n-i+1) \cdot 2^i$ nodes when it outputs a node of size $2^i \times 2^i$

- Bottom-up algorithm:
  1. number of nodes visited < 4 times the size of the sides of the blocks in the image
  2. alternative complexity measure
     - let $b_i =$ number of blocks of size $2^i \times 2^i$
     - visits $2^{2n+2} - 2^{n+2} \sum_{i=1}^{n} b_i \cdot (2^{2i+2} - 2^{i+2})$ nodes

- Bottom-up is preferable for images larger than $2^4 \times 2^4$ as neighbor finding cost is bounded by 4 and maximum depth is greater than 4 which is a factor in the cost of the bottom-up method

- Amount of work is identical for different images with the same block configuration

Ex:
COMPLEXITY OF CONVERTING QUADTREES TO RASTERS

• Assume a $2^n \times 2^n$ image

• Top-down algorithm visits $(n-i+1) \cdot 2^i$ nodes when it outputs a node of size $2^i \times 2^i$

• Bottom-up algorithm:
  1. number of nodes visited < 4 times the size of the sides of the blocks in the image
  2. alternative complexity measure
     • let $b_i =$ number of blocks of size $2^i \times 2^i$
     • visits $2^{2n+2} - 2^{n+2} \sum_{i=1}^{n} b_i \cdot (2^{2i+2} - 2^{i+2})$ nodes

• Bottom-up is preferable for images larger than $2^4 \times 2^4$ as neighbor finding cost is bounded by 4 and maximum depth is greater than 4 which is a factor in the cost of the bottom-up method

• Amount of work is identical for different images with the same block configuration

Ex:
CHAIN CODE

- Records relative position of boundaries of similarly-valued adjacent grid squares
- Four directions
- Usually assume the image is to the right
- Assume 4-connected, meaning that A and B are not in same region unless 8-connected
- Aggregation similar to runlength representation
  1. based on direction of boundary, not value of location
  2. no 2 consecutive elements on the aggregated boundary have the same direction
  3. implement with a number to indicate length, OR

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

starting at lower left corner yields

```
1 3 0 1 1 1 0 1 1 2 0 4
3 4 2 2 3 1 2 1 3 1 2 3
```

- can use +1 and –1 to indicate a change of direction
- more compact than absolute directions 0,1,2,3

4. just store the locations where the directions change

```
code = direction /90
```

- Comparison
  1. 3 is relative while 4 is absolute
  2. 3 is more compact than 4
  3. 4 is more robust as an error means that only one element is corrupted
CONVERTING FROM QUADTREES TO CHAIN CODES

- Trace the boundary of a region in the clockwise direction once an appropriate starting point has been determined.
- Starting point is an adjacency between a **black** block $P$ and a **white** block $Q$.
- Assume $P$ is to the north of $Q$.
- Possible overlap relationships between $P$ and $Q$. 
CONVERTING FROM QUADTREES TO CHAIN CODES

• Trace the boundary of a region in the clockwise direction once an appropriate starting point has been determined.

• Starting point is an adjacency between a **BLACK** block \( P \) and a **WHITE** block \( Q \).

• Assume \( P \) is to the north of \( Q \).

• Possible overlap relationships between \( P \) and \( Q \).

1. \( P \) extends past \( Q \).
CONVERTING FROM QUADTREES TO CHAIN CODES

- Trace the boundary of a region in the clockwise direction once an appropriate starting point has been determined.
- Starting point is an adjacency between a BLACK block $P$ and a WHITE block $Q$.
- Assume $P$ is to the north of $Q$.
- Possible overlap relationships between $P$ and $Q$.

1. $P$ extends past $Q$
2. $Q$ extends past $P$
CONVERTING FROM QUADTREES TO CHAIN CODES

- Trace the boundary of a region in the clockwise direction once an appropriate starting point has been determined.
- Starting point is an adjacency between a BLACK block $P$ and a WHITE block $Q$.
- Assume $P$ is to the north of $Q$.
- Possible overlap relationships between $P$ and $Q$.

1. $P$ extends past $Q$
2. $Q$ extends past $P$
3. $P$ and $Q$ meet at the same point.
ALGORITHM TO CONVERT FROM QUADTREES TO CHAIN CODES

• Possible block configurations

• Algorithm
ALGORITHM TO CONVERT FROM QUADTREES TO CHAIN CODES

• Possible block configurations

```
P Q
```


• Algorithm

1. Output the links of the chain code associated with the part of BLACK block P’s border that is adjacent to WHITE block Q
   • length of chain = minimum of sides of P and Q
ALGORITHM TO CONVERT FROM QUADTREES TO CHAIN CODES

• Possible block configurations

1. Output the links of the chain code associated with the part of BLACK block P’s border that is adjacent to WHITE block Q
   • length of chain = minimum of sides of P and Q

2. Determine the BLACK/WHITE block pair to be visited next
   • may need to inspect some adjacent blocks X and Y
   • use neighbor finding (variant of EQUAL_LATERAL_NEIGHBOR)
ALGORITHM TO CONVERT FROM QUADTREES TO CHAIN CODES

• Possible block configurations

- Algorithm

1. Output the links of the chain code associated with the part of BLACK block P’s border that is adjacent to WHITE block Q
   - length of chain = minimum of sides of P and Q

2. Determine the BLACK/WHITE block pair to be visited next
   - may need to inspect some adjacent blocks X and Y
   - use neighbor finding (variant of EQUAL_LATERAL_NEIGHBOR)
   - different result if X is BLACK
ALGORITHM TO CONVERT FROM QUADTREES TO CHAIN CODES

- Possible block configurations

- Algorithm
  1. Output the links of the chain code associated with the part of BLACK block P’s border that is adjacent to WHITE block Q
     - Length of chain = minimum of sides of P and Q
  2. Determine the BLACK/WHITE block pair to be visited next
     - May need to inspect some adjacent blocks X and Y
     - Use neighbor finding (variant of EQUAL_LATERAL_NEIGHBOR)
     - Different result if X is BLACK
     - Another result if Y is BLACK given X is BLACK
     - Don’t care about Y if X is WHITE
ALGORITHM TO CONVERT FROM QUADTREES TO CHAIN CODES

• Possible block configurations

• Algorithm

1. Output the links of the chain code associated with the part of BLACK block P’s border that is adjacent to WHITE block Q
   • length of chain = minimum of sides of P and Q

2. Determine the BLACK/WHITE block pair to be visited next
   • may need to inspect some adjacent blocks X and Y
   • use neighbor finding (variant of EQUAL_LATERAL_NEIGHBOR)
     • different result if X is BLACK
     • another result if Y is BLACK given X is BLACK
     • don’t care about Y if X is WHITE

• O(perimeter) execution time assuming a constant cost for neighbor finding
CONVERTING FROM CHAIN CODES TO QUADTREES

Algorithm:

1. traverse boundary in clockwise order and build a quadtree with BLACK nodes of unit size (i.e., $1 \times 1$) adjacent to the boundary
   • remaining nodes are left uncolored

2. assign the appropriate color to the uncolored nodes
STEP 1 OF CHAIN CODE TO QUADTREE CONVERSION ALGORITHM

- Construct the tree by creating pixel-sized BLACK nodes adjacent to the links in the chain code using the relationship between successive links in the chain code.

Nodes are added by use of neighbor finding (variant of EQUAL_LATERAL_NEIGHBOR).

As links in the chain code are processed, some nodes will be encountered more than once indicating that they are adjacent to the boundary on more than one side.

Use a code to keep track of the part of the boundary to which the node is adjacent on its right:
0: none of the sides of the node are adjacent to the boundary (such a node is said to be uncolored) 8
1: to the right of the northern boundary
2: to the right of the eastern boundary
4: to the right of the southern boundary
8: to the right of the western boundary

Codes are additive.

Ex: sample region

result of step 1
STEP 1 OF CHAIN CODE TO QUADTREE CONVERSION ALGORITHM

• Construct the tree by creating pixel-sized BLACK nodes adjacent to the links in the chain code using the relationship between successive links in the chain code.

• Nodes are added by use of neighbor finding (variant of EQUAL_LATERAL_NEIGHBOR).

• As links in the chain code are processed, some nodes will be encountered more than once indicating that they are adjacent to the boundary on more than one side.

• Use a code to keep track of the part of the boundary to which the node is adjacent on its right:
  0: none of the sides of the node are adjacent to the boundary (such a node is said to be uncolored)
  8: to the right of the northern boundary
  1: to the right of the eastern boundary
  2: to the right of the southern boundary
  4: to the right of the western boundary

• Codes are additive.

• Ex: sample region

result of step 1

Copyright © 2002 by Hanan Samet
EXAMPLE OF STEP 1 OF CONVERTING FROM CHAIN CODES TO QUADTREES

- Ex:
- Use a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern direction and add them if they are not present.
EXAMPLE OF STEP 1 OF CONVERTING FROM CHAIN CODES TO QUADTREES

- Ex:
- Use a variant of EQUAL_LATERAL_NEIGHBOR to find neighbors in the eastern direction and add them if they are not present.
EXAMPLE OF STEP 2 OF CONVERTING FROM CHAIN CODES TO QUADTREES

• Need to determine which nodes are inside the region and which ones are outside — i.e., assign colors to uncolored nodes and merge if necessary

• Could use seed filling algorithm or point-in-polygon tests but cumbersome

• Instead, use the codes in the result of step 1

• Key observation is that if any uncolored sons are determined to be BLACK (WHITE), then all remaining uncolored brothers must also be BLACK (WHITE)

• Algorithm traverses tree in preorder looking for uncolored nodes that are adjacent to colored ones (along the border at deepest level of the tree) and merging whenever possible

Ex:
EXAMPLE OF STEP 2 OF CONVERTING FROM CHAIN CODES TO QUADTREES

• Need to determine which nodes are inside the region and which ones are outside — i.e., assign colors to uncolored nodes and merge if necessary

• Could use seed filling algorithm or point-in-polygon tests but cumbersome

• Instead, use the codes in the result of step 1

• Key observation is that if any uncolored sons are determined to be BLACK (WHITE), then all remaining uncolored brothers must also be BLACK (WHITE)

• Algorithm traverses tree in preorder looking for uncolored nodes that are adjacent to colored ones (along the border at deepest level of the tree) and merging whenever possible

Ex:

• Boundary elements
EXAMPLE OF STEP 2 OF CONVERTING FROM CHAIN CODES TO QUADTREES

• Need to determine which nodes are inside the region and which ones are outside — i.e., assign colors to uncolored nodes and merge if necessary

• Could use seed filling algorithm or point-in-polygon tests but cumbersome

• Instead, use the codes in the result of step 1

• Key observation is that if any uncolored sons are determined to be BLACK (WHITE), then all remaining uncolored brothers must also be BLACK (WHITE)

• Algorithm traverses tree in preorder looking for uncolored nodes that are adjacent to colored ones (along the border at deepest level of the tree) and merging whenever possible

Ex:

• Boundary elements
• Output of step 1
EXAMPLE OF STEP 2 OF CONVERTING FROM CHAIN CODES TO QUADTREES

• Need to determine which nodes are inside the region and which ones are outside — i.e., assign colors to uncolored nodes and merge if necessary

• Could use seed filling algorithm or point-in-polygon tests but cumbersome

• Instead, use the codes in the result of step 1

• Key observation is that if any uncolored sons are determined to be BLACK (WHITE), then all remaining uncolored brothers must also be BLACK (WHITE)

• Algorithm traverses tree in preorder looking for uncolored nodes that are adjacent to colored ones (along the border at deepest level of the tree) and merging whenever possible

Ex:

• Boundary elements
• Output of step 1

• Uncolored node C must be BLACK because it is adjacent to a GRAY node containing nodes A and B which are BLACK, and hence C’s uncolored brothers D and E must also be BLACK
EXAMPLE OF STEP 2 OF CONVERTING FROM CHAIN CODES TO QUADTREES

• Need to determine which nodes are inside the region and which ones are outside — i.e., assign colors to uncolored nodes and merge if necessary

• Could use seed filling algorithm or point-in-polygon tests but cumbersome

• Instead, use the codes in the result of step 1

• Key observation is that if any uncolored sons are determined to be BLACK (WHITE), then all remaining uncolored brothers must also be BLACK (WHITE)

• Algorithm traverses tree in preorder looking for uncolored nodes that are adjacent to colored ones (along the border at deepest level of the tree) and merging whenever possible

Ex:

![Quadtree Diagram]

• Boundary elements

• Output of step 1

• Uncolored node C must be BLACK because it is adjacent to a GRAY node containing nodes A and B which are BLACK, and hence C’s uncolored brothers D and E must also be BLACK

• Average execution time \( \approx \) the region’s perimeter