A Finite State Machine Approach to Cluster Identification Using the Hoshen-Kopelman Algorithm

Matthew Aldridge

Objective

Cluster Identification
● Want to find and identify homogeneous patches in a 2D matrix, where:
  ● Cluster membership defined by adjacency
    ● No need for distance function
    ● Sequential cluster IDs not necessary
  ● Common task in analysis of geospatial data (landscape maps)

Hoshen-Kopelman Algorithm

Overview
● Assigns unique IDs to homogeneous regions in a lattice
● Handles only one target class at a time
  ● Lattice preprocessing needed to filter out unwanted classes
● Single-pass cluster identification
  ● Second pass to relabel temporary IDs, but not strictly necessary
● 2-D lattice represented as matrix herein

Data structures
● Matrix
  ● Preprocessed to replace target class with -1, everything else with 0
● Cluster ID/size array ("csize")
  ● Indexing begins at 1
  ● Index represents cluster ID
  ● Positive values indicate cluster size
    ● Proper cluster label
  ● Negative values provide ID redirection
    ● Temporary cluster label
Hoshen-Kopelman Algorithm

Csize array
- + values: cluster size
  - Cluster 2 has 8 members
- - values: ID redirection
  - Cluster 4 is the same as cluster 1, same as cluster 3
  - Cluster 4/1/3 has 5 members
- Redirection allowed for noncircular, recursive path for finite number of steps

Clustering procedure
- Matrix traversed row-wise
- If current cell nonzero
  - Search for nonzero (target class) neighbors
  - If no nonzero neighbors found ...
    - Give cell new label
  - Else ...
    - Find proper labels \( K \) of nonzero neighbor cells
    - \( \min(K) \) is the new proper label for current cell and nonzero neighbors

Nearest-Four Neighborhood
- North/East/West/South neighbors
- Used in classic HK implementations
- Of the four neighbors, only N/W have been previously labeled at any given time

Nearest-4 HK in action...
- Matrix has been preprocessed
  - Target class value(s) replaced with -1, all others with 0
Hoshen-Kopelman Algorithm

Nearest-4 HK in action...

- First row, two options:
  - Add top buffer row of zeros, OR
  - Ignore N neighbor check
### Hoshen-Kopelman Algorithm

#### Nearest-4 HK in action...

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

---

### Hoshen-Kopelman Algorithm

#### Nearest-4 HK in action...

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>0</th>
<th>3</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

---

### Hoshen-Kopelman Algorithm

#### Nearest-4 HK in action...

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>0</th>
<th>3</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

---

### Hoshen-Kopelman Algorithm

#### Nearest-4 HK in action...

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>0</th>
<th>3</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

---
Hoshen-Kopelman Algorithm

Nearest-4 HK in action...

1 0 2 0 0 3 3 0
1 0 2 0 4 3 0 -1
0 0 -1 -1 -1 -1 0 -1
-1 -1 0 -1 0 -1 0 -1
-1 0 0 0 -1 0 -1 0
-1 0 0 -1 -1 0 0 0
0 0 -1 -1 -1 -1 0 -1
0 0 -1 -1 -1 -1 0 -1

Hoshen-Kopelman Algorithm

Nearest-4 HK in action...

1 0 2 0 0 3 3 0
1 0 2 0 4 3 0 5
0 0 -1 -1 -1 -1 0 -1
-1 -1 0 -1 0 -1 0 -1
-1 0 0 0 -1 0 -1 0
-1 0 0 -1 -1 0 0 0
0 0 -1 -1 -1 -1 0 -1
0 0 -1 -1 -1 -1 0 -1

Hoshen-Kopelman Algorithm

Nearest-4 HK in action...

1 0 2 0 0 3 3 0
1 0 2 0 4 3 0 5
0 0 -1 -1 -1 -1 0 -1
-1 -1 0 -1 0 -1 0 -1
-1 0 0 0 -1 0 -1 0
-1 0 0 -1 -1 0 0 0
0 0 -1 -1 -1 -1 0 -1
0 0 -1 -1 -1 -1 0 -1

Hoshen-Kopelman Algorithm

Nearest-4 HK in action...

1 0 2 0 0 3 3 0
1 0 2 0 4 3 0 5
0 0 -1 -1 -1 -1 0 -1
-1 -1 0 -1 0 -1 0 -1
-1 0 0 0 -1 0 -1 0
-1 0 0 -1 -1 0 0 0
0 0 -1 -1 -1 -1 0 -1
0 0 -1 -1 -1 -1 0 -1
Hoshen-Kopelman Algorithm

Nearest-4 HK in action...

1 0 2 0 0 0 3 0
1 0 2 0 4 3 0 5
0 0 2 2 2 2 0 5
6 6 0 2 0 2 0 5
6 0 0 0 7 0 8 0
6 0 0 9 7 0 0 0
0 0 10 7 7 7 0 11
0 0 7 7 7 7 0 11

• Skipping ahead

Hoshen-Kopelman Algorithm

Nearest-4 HK in action...

1 0 2 0 0 2 2 0
1 0 2 0 2 2 0 5
0 0 2 2 2 2 0 5
6 6 0 2 0 2 0 5
6 0 0 7 0 8 0 0
6 0 0 7 7 0 0 0
0 0 7 7 7 7 0 11
0 0 7 7 7 7 0 11

• Optional second pass to relabel cells to their proper labels

Hoshen-Kopelman Algorithm

Nearest-Eight Neighborhood

• NW, N, NE, E, SE, S, SW, W
• When examining a cell, compare to W, NW, N, NE neighbors

Hoshen-Kopelman Algorithm

Nearest-Eight Neighborhood

• Sometimes more appropriate in landscape analysis
• Rasterization can segment continuous features if only using nearest-four neighborhood
Hoshen-Kopelman Algorithm

Nearest-4 vs. Nearest-8 Results

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>0</th>
<th>2</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>11</td>
</tr>
</tbody>
</table>

UNION-FIND Algorithm

Disjoint-Set Data Structure
- Maintains collection of non-overlapping sets of objects
- Each set identifiable by a single representative object
  - Rep. may change as set changes, but remains the same as long as set unchanged
- Disjoint-set forest is a type of D-S data structure with sets represented by rooted trees
  - Root of tree is representative

Disjoint-Set Data Structure Operations
- MAKE-SET(x)
  - Creates a new set whose only member is x
- UNION(x, y)
  - Combines the two sets containing objects x and y
- FIND-SET(x)
  - Returns the representative of the set containing object x
- An algorithm that performs these ops is known as a UNION-FIND algorithm

HK relation to UNION-FIND
- csize array may be viewed as a disjoint-set forest
**HK relation to UNION-FIND**
- Implementation of UNION-FIND operations
  - MAKE-SET: When a cell is given a new label and new cluster is formed
  - UNION: When two clusters are merged
  - FIND-SET: Also when two clusters are merged (must determine that the proper labels of the two clusters differ)

**Heuristics to improve UNION-FIND**
- Path compression
  - Used in FIND-SET to set each node's parent link to the root/representative node
  - FIND-SET becomes two-pass method
    1) Follow parent path of x to find root node
    2) Traverse back down path and set each node's parent pointer to root node

**Heuristics to improve UNION-FIND**
- Union by rank
  - Goal: When performing UNION, set root of smaller tree to point to root of larger tree
  - Size of trees not explicitly tracked; rather, a rank metric is maintained
  - Rank is upper bound on height of a node
  - MAKE-SET: Set rank of node to 0
  - UNION: Root with higher node becomes parent; in case of tie, choose arbitrarily and increase winner's rank by 1

**Applying these heuristics to HK**
- Original HK did not use either heuristic
- Previous FSM implementation (Constantin, et al.) used only path compression
- Implementation in this study uses path compression and union by cluster size
  - U by cluster size: Similar to U by rank, but considers size of cluster represented by tree, not size of tree itself
    - Reduces the number of relabeling ops in 2nd pass
Finite State Machines

Computational model composed of:
- Set of states
  - Each state stores some form of input history
- Input alphabet (set of symbols)
  - Input is read by FSM sequentially
- State transition rules
  - Next state determined by current state and current input symbol
  - Need rule for every state/input combination

Formal definition: \((S, \Sigma, \delta, q_0, F)\)
- \(S\): Set of states
- \(\Sigma\): Input alphabet
  - Input is read by FSM sequentially
- \(\delta\): State transition rules
  - \((\delta: S \times \Sigma \rightarrow S)\)
- \(q_0\): Starting state
- \(F\): Set of final states

Nearest-8 HK with FSM

Why apply FSM to Nearest-8 HK?
- Want to retain short-term knowledge on still relevant, previously examined cells
  - Helps avoid costly memory accesses
- Recall from Nearest-8 HK that the W, NW, N, NE neighbors' values are checked when examining each cell
  - *(only when the current cell is nonzero!)*

- Note that a cell and its N, NE neighbors are next cell's W, NW, N neighbors
- Encapsulate what is known about current cell and N, NE neighbors into next state
  - Number of neighbor comparisons can be reduced by up to 75%
Let's define our state space...

- Current cell value is always checked, thus always encapsulated in the next state
- Assume current cell value is nonzero
  - N, NE neighbor values are checked (along with NW, but that's irrelevant for next cell)
  - This produces four possible states when examining the next cell:

```
1 0
1
```

= cluster (nonzero)
= no cluster (zero)

And if current cell is zero?

- Neighbor values are not checked
  - But some neighbor knowledge may still be retained. Consider:

```
    Current cell nonzero, neighbors checked
```

```
    Current cell zero, NO neighbors checked
```

Even though previous cell was zero, we can retain knowledge of NW neighbor

What about multiple sequential zeros?

- Current cell nonzero, neighbors checked
- Current cell zero, NO neighbors checked
- Current cell zero, NO neighbors checked

Here we do NOT know the NW neighbor value

So, after a single zero value...

- We can still retain knowledge of NW neighbor
- This produces two more states:

```
Current cell nonzero, neighbors checked
Current cell zero, NO neighbors checked
```

= cluster = no cluster = unknown

= cluster (nonzero) = no cluster (zero) = unknown value = known value (zero or nonzero)
Nearest-8 HK with FSM

Putting it all together...

```
<table>
<thead>
<tr>
<th>s0</th>
<th>?</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>s2</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>s3</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>s4</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>s5</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>s6</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
```

= current  
= cluster  
= no cluster  
= unknown

Details...

- Previous slide is missing a final state
  - In formal definition, a terminal symbol is specified, to be located after last cell
  - From any state, encountering this symbol leads to final state
  - Implementation does not include final state explicitly
  - Bounds checking used instead

More details...

- Row transitions
  - If matrix is padded on both sides with buffer columns of all zeros, FSM will reset to $s_0$ before proceeding to next row
  - In actual implementation, no buffer columns
    - Again, explicit bounds checking performed
    - At beginning of row, FSM reset to $s_6$
    - At end of row, last cell handled as special case

In action...

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

- Start with first row clustered as before
Nearest-8 HK with FSM

In action...

1  0  2  0  0  3  3  0  0
1  0  2  0  3  -1  0  -1

s0 ? ?
s1 ? ?
s2 ?
s3 ?
s4 ?
s5 ?
s6 ?

= current
= cluster
= no cluster
= unknown

Nearest-8 HK with FSM

In action...

1  0  2  0  0  3  3  0  0
1  0  2  0  3  3  0  -1

s0 ? ?
s1 ? ?
s2 ?
s3 ?
s4 ?
s5 ?
s6 ?

= current
= cluster
= no cluster
= unknown
In action...

Alternative Implementations

- Parallel computing
  - MPI used for process communication
  - Controller/worker design, round-robin job assignment
- Matrix divided row-wise into $s$ segments
- csize also divided into $s$ segments, with mutually exclusive cluster ID spaces
- Results merged by controller node
- Minimal speedup, mostly due to staggered I/O
- May be useful for *much* larger data than used here

Alternative Implementations

- Concurrent FSMs
  - Identify multiple target classes in single pass
  - Each FSM maintains separate state
  - No longer in-place
  - Must maintain explicit state variables, rather than separate blocks of execution and implicit state

Methodology

- Tests performed on Linux workstation
  - 2.4 GHz Intel Xeon
  - 8 KB L1 cache
  - 512 KB L2 cache
- Timed over complete cluster analysis
  - First AND second pass (relabeling)
  - File I/O and data structure initialization not included
- Average time of 40 executions for each implementation and parameter set
Test Data

- One set of 5000x5000 randomly generated binary matrices
  - Target class densities: { 0.05, 0.1, 0.15, ..., 0.95 }
- Three actual land cover maps
  - 2771x2814 Fort Benning, 15 classes
  - 4300x9891 Tennessee Valley, 21 classes
  - 400x500 Yellowstone, 6 classes
**Conclusions**

- FSM clearly outperforms non-FSM for both landscape and random data
  - Sparse clusters: non-FSM still competitive
  - Dense clusters: FSM advantage increases due to retaining knowledge of neighbor values more often
- Proper merging (using union by cluster size) is key to performance
Palm PDA Performance

Why a PDA?
- Perhaps FSM can shine in high-latency memory system
- Conceivable applications include...
  - Mobile computing for field researchers
  - Cluster analysis in low-powered embedded systems

Methodology
- Tests performed on Palm IIIxe
  - 16 MHz Motorola Dragonball 68328EZ
  - 8MB RAM
  - No cache
- Only one run per implementation and parameter set
  - Single-threaded execution gives very little variation in run times (within 1/100 second observed)
  - Very small datasets

Test Data
- One set of 150x150 randomly generated binary matrices
  - Target class densities: \{ 0.05, 0.1, 0.15, \ldots, 0.95 \}
- 175x175 segment of Fort Benning map
  - 13 target classes

Random Data Results
Palm PDA Performance

Fort Benning Data Results

![Graph showing performance comparison between non-FSM and FSM implementations]

Branching in FSM vs. Non-FSM

- non-FSM
- FSM
  (dashed lines indicate state-based branching)

Conclusions

- Non-FSM implementation faster in all cases
  - FSM more competitive with higher target class densities
- Why is the FSM slower?
  - Ironically, lack of cache
  - Also, reduced program locality and execution branching
  - Adding as little as 1-2 KB of cache can reduce Palm's effective memory access time by 50% (Carroll, et al.)

In Closing

Possible Future Work

- Extension to three (or higher?) dimensions
  - Higher dimensions => more neighbors => many more states
    - Automated FSM construction would ease burden, allow non-programmers to define custom neighborhood rules
    - If effects of complex control logic/branching can be mitigated, then FSM savings should be great
- FSM adaptation for different data ordering (e.g. Z- or Morton-order)
- Implement FSM HK in hardware (FPGAs, etc.)