Change Identifiers

Max Dupenois

1 Introduction

This is a listing of the various possible change identifiers and the processes used to caclulate them.

2 Form

All change identifiers should be of the form:
Input: The current dot pattern D.
Input: The previous value of this identifer ci_{-1}.
Input: The current footprint F.
Output: A value x indicating the change amount.

3 Normalisation

For the values to be useful they will need to be normalised to stop any inherent prioritisation. AS such we should be able to get all values between 0 and 1, hopefully.

4 Change Identifiers

4.1 Change in centroid scaled by the bounding box

Requires: D and ci_{-1}

Preferred Data Structure: Binary search tree such that extreme x and y dots are swiftly located.

Process:
The centroid is found by $C = \frac{d_0+d_1+\cdots+d_{n-1}}{n}$, where n is the number of dots and $d_0 \cdots d_{n-1}$ is a dot from the set. However we need to scale by the bounding box so that we can know whether or not the change is small. Fig. 1 demonstrates this scaling effect.

Pseudo-code: see *Algorithm 1*

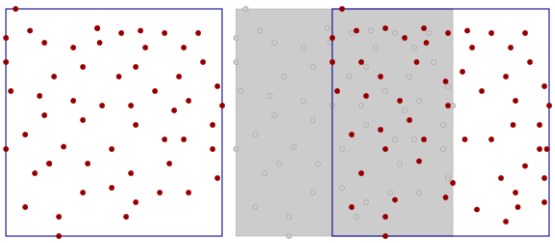
Complexity: $O(n)$

Normalisation: To scale this value we need a maximum possible change.. not sure yet

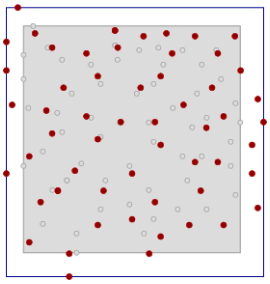
Algorithm 1 Pseudo-code for Scaled Centroid

Input: The current dot pattern D.
Input: A coordinate ci_{-1}: the previous scaled centroid
Output: A coordinate ci for this identifier
Output: A value c representing the change

1: $x = 0$
2: $y = 0$
3: $count = 0$
4: **for all** D **as** d **do**
5: $\quad x{+}{=}d_x$
6: $\quad y{+}{=}d_y$
7: $\quad count{+}{+}$
8: **end for**
9: $ci_x = x/count$
10: $ci_y = y/count$
11: $d_{max(x)}, d_{min(x)}, d_{max(y)}, d_{min(y)}$ as the coordinates which represent the coordinate with the maximum x value, the minimum x, maximum y and minimum y respectively.
12: Use the intersections of these lines to find the bounding box key coordinates (bottom left corner B_1 and top right corner B_2)
13: $s = |B_1 - B_2|$ where s is our scaling value
14: $ci_x = i_x/s$
15: $ci_y = i_y/s$
16: $c = |ci - ci_{-1}|$
17: **return** ci, c



(a) Original (b) Centroid Moved



(c) Centroid Moved 2

Figure 1: Effect of scale on centroid measurement

4.2 Change in standard deviation from the bounding box

Requires: D and ci_{-1}

Preferred Data Structure: Binary search tree such that extreme x and y dots are swiftly located.

Process:

The standard deviation is:

$$\sqrt{\frac{\sum_{i=0}^{n} |d_i - C|^2}{n}} \tag{1}$$

Where C is the centroid, n is the number of dots and d_i is a dot from the set. So the centroid will need to be found first. To save processing if this is coupled with *Indentifier 4.1* we should store the value, to this end a library of information should be built up to save re-performing calculations.

Pseudo-code: see *Algorithm 2*

Algorithm 2 Pseudo-code for Standard Deviation from Centroid

Input: The current dot pattern D.
Input: A value ci_{-1}: the previous standard deviation
Output: A value ci for this identifier
Output: A value c representing the change

1: $x = 0$
2: $y = 0$
3: $count = 0$
4: **for all** D **as** d **do**
5: $\quad x{+}{=}d_x$
6: $\quad y{+}{=}d_y$
7: $\quad count{+}{+}$
8: **end for**
9: $ce_x = x/count$ where ce is the centroid
10: $ce_y = y/count$
11: $ci = 0$
12: **for all** D **as** d **do**
13: $\quad ci = (|d - C|)^2$
14: **end for**
15: $ci = ci/count$
16: $ci = \sqrt{ci}$
17: $c = ci - ci_{-1}$
18: **return** ci, c

Complexity: $O(2n)$

Normalisation: To normalise this we need a maximum possible deviation ... hmmm

4.3 Change in distances between the medians on the axes

Requires: D and ci_{-1}

Preferred Data Structure: We need an structure that allows us to have both the x and y coordinates ordered. A 2-dimensional tree structure should allow this.

Process:

The median on either access can be easily located as long as the dots are ordered. Once the median is found we take the total difference between the dots coordinate values on both axes and take this as our value.

Pseudo-code: see *Algorithm 3*

Algorithm 3 Pseudo-code for Distances between the medians

Input: The current dot pattern D.
Input: A value ci_{-1}: the previous median values
Output: A value ci for this identifier
Output: A value c representing the change

1: L_x: an empty list
2: L_y: an empty list
3: $count = 0$
4: [This looping may not be necessary, we may have the size already and the tree structure will give us a way of locating the middle value.]
5: **for all** D_x **as** x **do**
6: add x to L_x
7: $count$++
8: **end for**
9: **for all** D_y **as** y **do**
10: add y to L_y
11: **end for**
12: ci_x
13: ci_y
14: **if** $count \mod 2 == 0$ **then**
15: $i = \text{ceil}(count/2)$
16: $ci_x = L_x(i_1)$
17: $ci_y = L_y(i_1)$
18: **else**
19: $i_1 = \text{ceil}(count/2)$
20: $i_2 = i_1 + 1$
21: $ci_x = (L_x(i_1) + L_x(i_2))/2$
22: $ci_y = (L_y(i_1) + L_y(i_2))/2$
23: **end if**
24: $c = ci_x - ci_{-1x}$
25: c+=$ci_y - ci_{-1y}$
26: **return** c, ci

Complexity: $O(2n)$

Normalisation: To normalise this value we need a maximum possible change.. not sure

4.4 Proportion of points outside the boundary of the previous footprint

Requires: D and F

Preferred Data Structure: Irrelevant as long as it can be easily looped through

Process:

Using the ray tracing method we count the number of dots external to the footprint and then return this as a proportion of the total number of dots. Currently we will count dots on the boundary to be within the point set.

Pseudo-code: see *Algorithm 4*

Algorithm 4 Pseudo-code for Proportion of points outside the boundary of the previous footprint

Input: The current dot pattern D.
Input: The current footprint F.
Output: A value ci for this identifier

```
1:  externalCount = 0
2:  count = 0
3:  for all D as d do
4:      intersectionCount = 0
5:      for all F as f do
6:          s start vertex of edge f
7:          e end vertex of edge f
8:          line equation = y = d_y
9:          Find i intersection of edge and line.
10:         if iϵf && i_x ≥ d_x then
11:             intersectionCount++
12:         end if
13:     end for
14:     if intersectionCount mod 2 == 0 then
15:         externalCount++
16:     end if
17:     count++
18: end for
19: ci = (externalCount/count)
20: return ci
```

Complexity: $O(nm)$ where m is the number of edges.

Normalisation: Already Normalised

4.5 Change in number of dots

Requires: D and ci_{-1}

Preferred Data Structure: Irrelevant as long as it can be counted

Process:
Simple one, just count the number of dots and see if there's a change.

Pseudo-code: see *Algorithm 5*

Algorithm 5 Pseudo-code for Number of dots

Input: The current dot pattern D.
Input: A value ci_{-1}: the previous count
Output: A value ci for this identifier
Output: A value c representing the change

```
1: ci = 0
2: for all D as d do
3:     ci++
4: end for
5: c = ci − ci_{−1}
6: return c, ci
```

Complexity: $O(n)$

Normalisation: To normalise this value we need a maximum possible change.. not sure yet

4.6 Difference in area of bounding box

Requires: D and ci_{-1}

Preferred Data Structure: Binary search tree such that extreme x and y dots are swiftly located.

Process:
Once the bounding box is found then we can just compare their areas which is $h * w$.

Pseudo-code: see *Algorithm 6*

Complexity: $O(\log n)$ assuming a log search for extreme values

Normalisation: To normalise this value we need a maximum possible change.. not sure yet

Algorithm 6 Pseudo-code for Bounding Box Area

Input: The current dot pattern D.
Input: A value ci_{-1}: the previous count
Output: A value ci for this identifier
Output: A value c representing the change

1: $d_{max(x)}, d_{min(x)}, d_{max(y)}, d_{min(y)}$ as the coordinates which represent the coordinate with the maximum x value, the minimum x, maximum y and minimum y respectively.
2: Use the intersections of these lines to find the bounding box key coordinates (bottom left corner B_1 and top right corner B_2)
3: $ci = (B_{2_x} - B_{1_x}) * (B_{2_y} - B_{1_y})$
4: $c = ci - ci_{-1}$
5: **return** c, ci