Eurographics Conference on Visualization (EuroVis) 2021
R. Borgo, G. E. Marai, and T. von Landesberger
(Guest Editors)

Volume 40 (2021), Number 3

Animated Presentation of Static Infographics with InfoMotion Supplementary Material

Yun Wang[1], Yi Gao[1,2], Ray Huang[1], Weiwei Cui[1], Haidong Zhang[1], and Dongmei Zhang[1]

[1]Microsoft Research Asia [2]Nanjing University

1. Information Structure Inference

Infographic designs can be organized into information structures consisting of visual elements. The major component of an infographic are the units that are composed with visual elements. The units usually have similar designs and are placed to certain positions to imply the relations. The connectors are usually used to connect units together. Embellishments with various design may appear at any positions in an infographic. For each element, we assign semantic tags, such as titles, descriptions, icons, et.

$$infoStructure := \{units, layout, conns, embs, semanTags\}$$

$$units := \{unit_1, unit_2, ..., unit_i\}$$

$$unit_i := \{elem_{i,1}, elem_{i,2}, ..., elem_{i,j}\}$$

$$elemGroup_j := \{unit_1.elem_{1,j}, unit_2.elem_{2,j}, ..., unit_i.elem_{i,j}\}$$

Our approach starts by (1) finding repeating (similar) elements that are used across repeating units, which are constitutive of repeating units (Line 2, Algorithm 1). Then we (2) organize those elements into repeating units to model the information structure of the infographic designs (Line 3, Algorithm 1). Finally, we (3) recognize infographic connectors (Line 4, Algorithm 1) and (4) add semantic tags (Line 5, Algorithm 1) to complete the structure inference that enables flexible animation arrangements .

Algorithm 1 Information Structure Inference

```
INPUT: $elements$                    ▷ visual elements in an infographic
OUTPUT: $infoStructure := (units, layout, conns, embs, semants)$
1: procedure EXTRACTINFOSTRUCTURE($elements$)
2:     $(clusters, n) \leftarrow$ cluster($elements$)
3:     $(units, layout, others) \leftarrow$ mergeCluster($clusters, n$)
4:     $conns \leftarrow$ detectConns($units, others$)
5:     $embs \leftarrow$ detectEmbellishments($units, others$)
6:     $semanTags \leftarrow$ detectSemants($units, others$)
   return $(units, layout, conns, embs, semanTags)$
```

2. Identifying Repeating Units

After extracting many clusters, we assign them into repeating units (Algorithm 2). we first identify the layout type of every cluster of $size(C_n) = N$ by classifying them into one of the four layout types. We take the most frequent layout type as the overall unit layout (Line 3, Algorithm 2). Then we try to merge clusters of $size(C_n) >= N$ one by one into units by layout or proximity (Line 8-12, Algorithm 2). When the clusters' layouts are identified and the same with the units' layout, and the sizes of the clusters are of N, we naturally try to merge them into the N units by layout; otherwise, we merge them by proximity.

Algorithm 2 Merge Cluster

```
1: procedure MERGECLUSTER($clusters, n$)
2:     $units \leftarrow []$
3:     $unitLayout \leftarrow$ mostFrequentLayout($clusters$)
4:     $unitCands \leftarrow$ filter($clusters, c \Rightarrow c.len \geq n$)
5:     $others \leftarrow$ filter($clusters, c \Rightarrow c.len < n$)
6:     $u \leftarrow unitCands$.pop()
7:     while $u \neq$ NULL do
8:         $layout \leftarrow$ detectLayout($u$)
9:         if $layout == unitLayout$ and $u.len == N$ then
10:            $assembleResult \leftarrow$ assembleByLayout($units, u$)
11:        else
12:            $assembleResult \leftarrow$ assembleByProximity($units, u$)
13:        if $assembleResult$ == NULL then
14:            add $u$ to $others$
15:        else
16:            add $assembleResult.selected$ to $units$
17:            if $assembleResult.left.len \geq n$ then
18:                add $assembleResult.left$ to tail of $unitCands$
19:            else
20:                add $assembleResult.left$ to tail of $others$
21:        $u \leftarrow unitCands$.pop()
    return (unitLayout, units, others)
```

When elements are merged by layout, they are sorted according to their positions and added to the units accordingly. When elements are assembled by proximity, we calculate the distance between the elements in the new cluster and existing units, and put each element in the cluster into correct units. To ensure proper merging into units, we further leverage element regularity across units by calculating standard deviations of the distances between current units and the newly added elements (Algorithm 3). We permutate all the ways of merging these elements into units and calculate standard deviations of the distances between current units and the newly added elements. We take the permutation with the minimal total distance. The standard deviation should be lower than a threshold p to avoid grouping by mistake. The default value of p is set to 0.04 based on our experiments.

Algorithm 3 Assemble Elements in a Cluster into Units

```
 1: procedure AssembleByProximity($units, candCluster$)
 2:     $minDist \leftarrow$ MAX_NUMBER_VALUE
 3:     $unitCentroids \leftarrow []$
 4:     $selected \leftarrow []$
 5:     $left \leftarrow []$
 6:     $n \leftarrow units.len$
 7:     for $i = 0 \dots n-1$ do
 8:         $unitCentroids[i] \leftarrow$ calculateCentroid($units, i$)
 9:     for $each\ perm$ in permutateElement($candCluster, n$) do
10:         for $i = 0 \dots n-1$ do
11:             $dist[i] \leftarrow$ calculateDistance($unitCentroids[i], perm[i]$)
12:         $totalDist \leftarrow$ sum($dist$)
13:         $diffFactor \leftarrow$ stddev($dist$)
14:         if $diffFactor <$ THRESHOLD and $totalDist < minDist$ then
15:             $selected \leftarrow perm$
16:             $left \leftarrow candCluster$.substract($perm$)
17:             $minDist \leftarrow totalDist$
18:     if $minDist ==$ MAX_NUMBER_VALUE then
19:         return NULL
20:     else
21:         return ($selected, left$)
```

When the clusters are of $size(C_n) > N$, we take out the elements to be merged to the units, and check whether the remaining elements are still of $size(C_n) >= N$. It is common that the remaining elements can still contribute to the units. We put them back to the tail of candidate clusters. If the remaining elements are of size less than N, we put the remaining elements to the set of *others*. Finally, the *others* elements that cannot be successfully merged into units will be further considered as connectors, embellishments, or other components in the next step.

3. Animation Effects

We explore a data-driven method to build an element-effect model. Through this way, we can further understand whether designers have similar considerations when adopting animation effects. We extract 461 visual element-animation effect pairs from our dataset. We select six common animation styles, namely, fading, floating, zooming, wiping, flying, and splitting, as general choices for the effects, covering 95.5% of the effects in the dataset.

We train a random forest model to recommend top-k ideal effects for the visual elements [rf20]. Random forests are an ensemble learning method of constructing a multitude of decision trees at training time and outputting the majority vote from these individual trees as final predictions. The parameters for each element include element width, element height, element shape, unit layout, and element position. We construct 100 decision trees in this forest at the training time. When looking for the best split, we only consider the radical number of features and we use Gini impurity to measure the quality of a split in each tree. Tree nodes are expanded until all leaves are pure. The model takes the properties of visual elements as input and retrieves top-k animation effects with highest probabilities as the output. We set k as a parameter to adjust the number of animations that we can recommend based on the requirements of applications and the accuracy of our model. While training the model, we use 10-fold cross-validation. In other words, we first split the dataset into 10 groups. Then for each group, we take the group for testing and take the remaining for training.

We calculate the accuracy of animation prediction if the animation effect in one test case is included. The average accuracy of cross validation is 86.98% (k=3), 73.69% (k=2), and 65.8% (k=1). Depending on the flexibility of animation design environments, animation design applications can recommend top-1, or top-k animation design to the users. When users need to modify the animation effect for a given component, applications can recommend a ranked list of effects. This enables users to easily modify the animation effects based on their personal preferences.

References

[rf20] sklearn.ensemble.randomforestclassifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html, 2020. 2