Social Network Graph Analysis (for dogs)

Justin Watt

5/12/16

Project Domain

In the world of Animal Rescue there is an idea that is being popularized known as 'Playing for Life'. It's a system that uses the interactions between animals to normalize their behavior and get them socialized and ready to be adopted. Using what is referred to as a 'base dog' or base group, new dogs are introduced to the group under supervision. Dogs that get on well together will then spend time and the newer dogs can learn from the base group and improve their behavior over time.

What is difficult about this system is that in a volunteer run organization it can be hard keep track of what animals get along with each other. When new volunteers don't know the animals very well it can be useful to track the relational data from the dog interactions and present it to the user.

Plan

This problem can be represented as a graph with the edges representing the strength of the relationship between two dogs. Edges with a negative weight indicate dogs that have had negative interactions and should not be put into play groups together. An ideal solution would have two parts, a play group with only positive relations between the dogs and a 'suggest a dog' transitive relationship where

$$\text{Dog A} \rightarrow \text{Dog B} \vee \text{Dog B} \rightarrow \text{Dog C} \implies \textit{DogA} \rightarrow \text{Dog C}.$$

Data Structure

```
{:Fran   {:Henry 3, :NewGuy 3, :BT 2,  :Ellie 5, :Shaw 3, :Rosy -1},
 :Henry  {:Fran 3,  :NewGuy 2, :BT -1, :Ellie 1, :Shaw 5,:Rosy -1,:Rogue 2,:Kevin -1},
 :NewGuy {:Fran 3,  :Henry 2,  :BT -1, :Ellie 5, :Rosy 2, :Rogue 2},
 :BT     {:Fran 2,  :Henry -1, :NewGuy -1, :Ellie 2, :Rosy -1},
 :Ellie  {:Fran 5,  :Henry 1,  :NewGuy 5, :BT 2, :Shaw -1, :Rogue 3, :Kevin 4}}
```

The graph's clojure representation is a nested dictionary structure. The key represents the node while the value is another dictionary containing all of the edges for that node.

Algorithm

The idea is to construct a suggestion algorithm that considers the nodes in common between those in a set and then filling in the gaps with suggestions of nodes that the base set may be interested in.

```
(defn suggestion-list [graph nodes]
  "Generates a list of suggestions given a base set of nodes"
  (let [neighbors (mapcat (partial find-neighbors graph) nodes)
        no-fly    (incompatible neighbors)]
    (->> neighbors ;; 1
         (filter #(> (second %) 0)) ;; 2
         (map (comp filter-negative (partial find-neighbors graph) first)) ;; 3
         (reduce #(merge-with + %1 %2) {}) ;; 4
         (filter #((complement contains?) (s/union no-fly nodes) (first %))) ;; 5
         (sort-by second) ;; 6
         reverse
         vec)))
```

1. This is a macro that threads the value neighbor as the last argument of the next function. The return value from that function is then threaded as the last argument of the next function and so on.
2. Filter out all of the negative neighbors of the base set.
3. Find transitive relationships of the neighbors.
4. Merge duplicates and add their weights.

5. Filter any nodes that are already in the no-fly list or the base set of nodes

6. Sort by value, descending and return as a list.