CPSC 320 Sample Solution, Clustering

October 11, 2017

You're working on software to manage people's photos. Your algorithm receives as input:

- a bunch of uncategorized photos,
- the number of categories to group them into (i.e., how many categories to use),
- and a *similarity measure* for each pair of photos.

(A 0 similarity indicates two photos are nothing like each other; a 1 indicates two photos are exactly the same. All other similarities are in between.)

Your algorithm's job is to create a "categorization": the requested number of categories, where a category is just a (non-empty) set of the photos contained in that category. Every photo belongs to some category, and no photo belongs to more than one category. (I.e., a categorization is a "partition".)

Note: we assume that to your algorithm the photos are just nodes with no special content. The similarities encode everything it needs to know about them. So, for example, your algorithm cannot "find the prominence of a person in the photo" because that relies on the photos' contents.

Sketches (drawings) will be incredibly important for understanding this problem!

1 Trivial and Small Instances

1. Write down all the **trivial** instances of the categorization problem.

   SOLUTION: An empty graph (with zero categories, the only legitimate number to ask for) is clearly trivial.

   Let's let n be the number of photos and c be the number of categories requested. Then, any instance with $c = 1$ besides the empty graph is also trivial because every photo must go in exactly one category, and there **is** only one category!

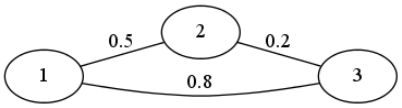
   Conversely, what if there are just enough photos to make the categories non-empty: $c = n$. In that case, each photo goes in its own category, and the instance is still trivial.

   For the rest of the problem, let's assume the empty graph and zero categories are disallowed, since that instance doesn't fit well with the other trivial instances.

   You might also find instances with all similarities equal or other interesting cases to be trivial. We'll leave our list at what we've given so far.

2. Write down another **small** instance of the categorization problem of different sizes. (Try to find the smallest non-trivial instance you can.)

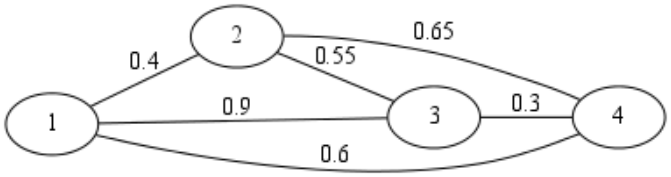
   SOLUTION: The smallest instance where $1 < c < n$ is when $n = 3$ and $c = 2$: three photos and two categories. Here's one such, where we describe the graph as a list of edge tuples (v_1, v_2, w) with two vertices and the similarity weight between them: $(\{(1, 2, 0.5), (1, 3, 0.8), (2, 3, 0.2)\}, 2)$. Drawn, this looks like:



   It seems clear that any reasonable metric should group 1 and 3 here and leave 2 alone.

3. Draw a reasonable graph for this set of numbered pictures. (You need to choose similarities between each pair of photos; just quickly choose some that you like! Note that while you're doing this step, your algorithm does **not** do it! It receives the similarities as input.)

   SOLUTION: There are many reasonable solutions. Here's one. (We simply chose plausible similarities.)



   Along with $c = 2$, this is an instance of the problem.

   It's hard to eyeball a solution to this with no metric to guide us, but it does seem likely that images 1 and 3 should be together given their strong similarity. Once we decide that, we might put images 2 and 4 together as well.

4. The graph you've drawn is **not** an instance of this problem yet because something is missing. **Read the specification at the start of this worksheet carefully**, figure out what's missing, and add that missing element. Use the value 2.

   SOLUTION: See above.

5. Give solutions to your trivial/small instances by hand if you haven't already done so. In your small instances, briefly justify why your solution is the best one.

6. Hold this space for another instance, in case we need more.

2 Represent the Problem

1. We represent an unweighted, undirected graph as $G = (V, E)$, where V is a set of nodes, E is a set of edges, and each edge is a tuple (u, v) that we consider to be unordered, where $u, v \in V$.

   That's not quite right for our problem. Modify the representation to describe the input graph in this problem. Also describe the additional parameter (the desired number of categories).

   SOLUTION: Again, we've wrestled with this above. We might say the input is (G, c), where $G = (V, E)$, V is a set of photos of size n, E is a set of weighted edges of the form (v_i, v_j, s), where $v_i, v_j \in V$ and $0 \leq s \leq 1$. c is just the desired number of categories (a natural number where $1 \leq c \leq |V|$ or $c = 0$ if G is empty).

   Note that we also want the graph to be complete—to contain every possible edge (except self-loops).

2. Go back up and rewrite one trivial and one small instance using these names.

   SOLUTION: See above.

3. Our sketched representation of weighted graphs remains great! Sketch all your instances if you haven't already.

4. Our input graph has some constraints. We'll skip expressing "no self-loops" and "no duplicate edges", since we've done that before. Similarities must also be between 0 and 1, and **every** pair of photos has a similarity. Further, only certain numbers of categories make sense. Express these constraints.

   SOLUTION: We specified $0 \leq s \leq 1$ above. We can also say that every pair of nodes has an edge: there is an edge $(v_i, v_j, s_{(i,j)})$ for all $v_i, v_j \in V$ where $v_i \neq v_j$. Finally, we said above that barring the empty graph and zero categories, we require $1 \leq c \leq n$. (A category may not be empty, and the most it can contain is all the nodes!)

3 Represent the Solution

1. What are the quantities that matter in the solution to the problem? Give them short, usable names.

   SOLUTION: The solution will be a set of categories $\{C_1, C_2, \ldots, C_c\}$, where a category C_i is a set of nodes from V.

2. Describe using these quantities what makes a solution **valid** and **good**.

   SOLUTION: A useful line at which to draw **valid** is that a categorization must be a partition. That is, every photo must belong to one and only one category. Depending on your definition of "partition", you may need to further constrain that all categories C_i be non-empty.

   It's much harder to say what a **good** solution is. Clearly we want to reward having nodes with high similarity in the same category and penalize having nodes with low similarity in the same category. If we divide edges into intra-category (between nodes in the same category) and inter-category (between nodes in different categories), then we might choose a metric like "sum of the intra-category similarities minus sum of the inter-category similarities". However, this will push us toward large categories. (Since the number of intra-category edges scales as $O(|C|^2)$, there are more intra-category edges in a categorization with one big category and many small ones than with all even-sized categories (and the same total number of categories).)

   We could instead try adding the **average** similarity of each category together. We have to decide then what to do with categories of a single node, since their average is undefined. Rate them zero?

   (Note that there's no "right" choice. We just have to decide what does a good job modeling what we care about and how efficient a solution we get. There's something fundamental about the fact that we will soon pick a metric that's totally reasonable... and which we really chose because it is "good enough" and admits a highly efficient solution. How much of our lives are now ruled by metrics that are easy to compute rather than "best"?)

3. Does your metric for the "goodness" of a categorization give the same result for these two categorizations of a four-node instance into two categories: $\{1,3\},\{2,4\}$ and $\{4,2\},\{1,3\}$? Should it? Why or why not?

   SOLUTION: All metrics described above rate these two categorizations the same, which is good. Since we have only a number of categories to create (and not a description of what the categories are), there's no way for us to assign value to exactly which category a node goes into, only to where the node sits with respect to other nodes. (Similarly, we don't care where in a category a node goes.)

   (Adjust your categorization as needed!)

4. Go back up to your trivial and small instances and write out one or more solutions to each using these names.

   SOLUTION: Left to the reader, since it's not much different than our solutions "in English".

5. Go back up to your drawn representations of instances and draw at least one solution.

   SOLUTION: We'd just circle nodes that go together, but we'll leave that to the reader!

6. From here on, we'll all use the same "goodness" measure.

   First, we define the similarity between two categories C_1 and C_2 to be the maximum similarity between any pair of photos p_1, p_2 such that $p_1 \in C_1$ and $p_2 \in C_2$.

   Then, the "goodness" of a categorization is the maximum similarity between any two of its categories, and the best "goodness" is 0. (Note that we are indeed "minimizing a maximum". The "goodness"—or maybe it would be better to call this the "badness"—of a single solution is the maximum of its categories' similarities. We don't **want** categories to be similar. So, the best solution is the one among all valid solutions that has the lowest value for this measure.)

   Go back to the questions on the previous page and re-answer them with this "goodness" measure.

   SOLUTION: This goodness measure is also insensitive to reordering of the categories, which is good. Furthermore, it corresponds to our solutions above. (So, the 4-node, 2-cluster problem's solution has a goodness of 0.6, defined by the edge between 1 and 4. The 3-node, 2-cluster problem's solution has a goodnes of 0.5 (between nodes 1 and 2).

4 Similar Problems

You're starting to learn more problems and algorithms. Spend 3 minutes trying to brainstorm at least one similar problem. (**Any** problem is fine, not just ones from a textbook or Wikipedia page; your problem could come from our quizzes, lectures, or assignments.)

SOLUTION: There are certainly similarities here to shortest path, minimum spanning tree, stable marriage, and maximal matching. As we'll see, perhaps the most promising similarity is to MST!

5 Brute Force?

1. The set of all valid solutions is the set of partitions of V into c subsets (where c is the requested number of categories). That turns out to be tricky to produce directly (challenge problem!).

   Instead, write pseudocode to produce all the "labellings" of the nodes where each node is given a label from $\{1, 2, \ldots, c\}$. (This will produce invalid solutions where some categories go unused, which we'll have to filter out. It will also produce duplicates where the same set of categories has simply been renamed. As always, **give a name** to your algorithm and its parameters, *especially* if your algorithm is recursive.)

   SOLUTION: We said a valid solution is one that forms a partition with c parts. It's not obvious how to generate these via brute-force, but generating all "labellings" with c labels is a bit easier. Imagine our categories were named, so that, e.g., category A with nodes 1 and 2 and B with 3 and 4 is different from A with 3 and 4 and B with 1 and 2. Then, we are assigning one of c category labels to each of n elements.

   It turns out to be surprisingly easy to form all the labellings. For each node, we have c choices of labels, and that choice is independent of all the other choices. We can use an algorithm like:

```
all_labellings(n, c):
  if n = 0:
    yield {}   // yield the single solution that is the empty labelling
  else:
    for soln in all_labellings(n-1, c):
      for i in c:
        yield {(n, i)} + soln   // add n labelled with i to soln
```

   How many labellings does this produce? n independent choices, with c options each. That's $\underbrace{c \times c \times \ldots \times c}_{n \text{ times}} = c^n$ labellings. If we model the code above with a function representing the number of solutions it produces $T_a(n, c)$, we get $T_a(n, c) = 1$ if $n = 0$ and otherwise $T_a(n, c) = T_a(n-1, c) * c$. This produces the same product of c multiplied together n times.

   We can then lump nodes with the same labels into a category and eliminate results that have empty categories. That will produce categorizations many times (by a factor of $c!$, in fact).

   We can cleverly avoid both problems if we take the labels as parameters as two sets—non-empty categories and empty ones:

```
// NOTE: the initial call would be all_categorizations(n, {}, {1, ..., c})
// PREREQ: 0 <= |unused_labels| <= n
// PREREQ: unused_labels and used_labels are disjoint
all_categorizations(n, used_labels, unused_labels):
  if n = 0:
    yield {}
  // ensure every label gets used
  else if n = size(unused_labels):
    // assign one unusued label to each number 1, ..., n
    // what permutation we choose is irrelevant
    yield {(1, unused_labels[1]), (2, unused_labels[2]),
           ..., (n, unused_labels[n])}
  else:
    // n could go in any of the already-started categories..
```

```
    for i in used_labels:
      for soln in all_categorizations(n-1, used_labels, unused_labels):
        yield {(n, i)} + soln
    // or it can start a new category of its own, in which case
    // the new category's specific label is arbitrary
    if unused_labels is non-empty:
      let u be an arbitrary label from unused_labels
      for soln in all_categorizations(n-1, used_labels + {u}, unused_labels - {u}):
        yield {(n, u)} + soln
```

That's rather tricky and, as discussed below, still takes exponential time or worse! So, maybe the first one is the better brute force approach just for being easier to think of/implement.

2. Choose an appropriate variable (or variables!) to represent the "size" of an instance.

   SOLUTION: c and n seem like the key variables.

3. Exactly or asymptotically, how many candidate solutions (counting invalid ones and duplicates) does this brute force approach produce?

   SOLUTION: For the brute force approach, this is just c^n.

   For the actual number of valid solutions, this is quite a tricky question!

   But the bottom line is that there are a **lot** of solutions. After all, with $c = 2$, the labellings algorithm above produces all subsets of a set of size n, of which there are 2^n. The categorization solution produces one less (cutting the case with an empty category) than half (cutting repeats where the two categories are swapped) that many: $\frac{2^n}{2} - 1 \in \Theta(2^n)$. This is already exponential, and as long as c is smallish compared to n, this only gets worse as c grows.

4. In this problem, you'll likely keep track of the best candidate solution you've found so far as you work through brute force. What will characterize how good a possible solution is?

   SOLUTION: Based on our goodness metric, it's the highest similarity of any inter-category edge.

5. Given a possible solution, how can you determine how good it is? Asymptotically, how long will this take?

   SOLUTION: A brute force approach would take $O(n^2)$ (which, for these complete graphs, is $O(m)$) time. For each pair of nodes, check if they're in the same category (which if we have labels attached to the nodes takes constant time). If they're not, check if their similarity exceeds the highest we've found so far, updating if necessary.

6. Will this brute force approach be sufficient for this problem for the domain we're interested in?

   SOLUTION: No way! There are at least an exponential number of solutions to any interesting problem.

6 Promising Approach

There is a **much** better approach.

1. Find the edge in each of your instances with the highest similarity. Should the two photos incident on that edge go in the same category? **Prove** your result.

   SOLUTION: Yes the two photos incident on the edge with highest similarity belong in the same category. Why?

   Let's revisit our metric. It ignores intra-category edges: any edge that's entirely contained within a single category (i.e., both nodes incident on the edge are in the same category). Its goal is to minimize the maximum *inter*-category edge.

   If we put the two photos incident on the highest similarity edge in different categories, that edge becomes an inter-category edge, and no other edge can possibly "beat it". That means a solution in which those two photos are in different categories is (tied for) **the worst possible solution**.

   Thus, those two photos clearly belong in the same category. What we're really saying is "start with every node being its own category, then you're best off merging the two categories connected by the highest-similarity edge".

   Once we decide that, we can actually simplify our problem and reach a sketch of a solution algorithm. Here's two ways to think about the simplification.

   Version 1: That reasoning actually applies not just to the highest-similarity edge. It applies to all the k highest-similarity edges until we've "lumped together" enough nodes that we're down to our desired number of categories. That is, no categorization can possibly do better than to "hide" the next highest-similarity edge inside a category by putting the edge's two nodes in the same category until the next edge it attempts to hide would reduce the number of categories below that requested by the instance. (Notice that not every edge actually merges two categories; some edges may already be in the same category.)

   Version 2: Once we've decided to lump together the two nodes incident on the highest-similarity edge, we can actually reduce this instance to a smaller instance of the photo categorization problem using an edge contraction. Imagine we remove nodes u and v by contracting their edge (u, v, s) (where s is whatever similarity that edge had). We make a new graph with $n-1$ nodes that lacks the two nodes we just lumped together. Instead, we introduce a combined node $n_{u,v}$. We remove the edge (u, v) from the graph. Each of u and v also had an edge to every other node, but the new graph needs just one edge from the combined node to each other node. So, for every two edges (u, n, s_u) and (v, n, s_v), we make a single new edge $(n_{u,v}, n, \max(s_u, s_v))$. We keep the maximum because it's the one that will "cause trouble" if it ends up being an inter-category edge. (Note that after this point, either both edges will be intra-category or both will be inter-category. We cannot put u and v in the same category and then put n in the u's category but not v's!) Once we get a catogerization that solves the sub-problem, we turn it into a categorization that solves the original problem by removing $n_{u,v}$ from whatever category it ended up in and replacing it there with u and v.

   The cool thing about that reduction is that we can then make an inductive argument that repeatedly picking the highest-similarity edge and merging the two incident nodes into a single category is guaranteed to produce the best categorization!

2. Based on this insight, propose an efficient algorithm to create a categorization.

   SOLUTION: We have the sketch of an algorithm already. We just need to flesh out how we (1) find the highest-similarity edge, (2) merge the two nodes incident on that edge into a single category, and (3) produce the categories at the end.

   Here's one algorithm:

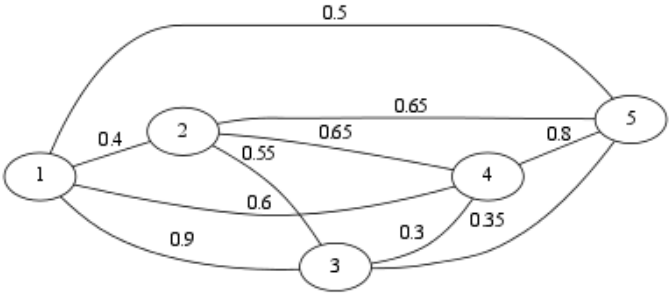
```
// Assume: P is a list of photos, sim is a function that takes two
//         photos and produces their similarity (between 0 and 1,
//         where 0 is minimally similar and 1 maximally), and c
//         is the desired number of categories such that 1 <= c <= |P|.
// Postcondition: produces a set where each element is a category
//                (i.e., is itself a set of photos categorized together)
categorize(P, sim, c):
  // First, build a priority queue of edges ordered by decreasing similarity: O(m)
  let H be a new empty array of (key, value) tuples (suitable as the array for a heap)
  for i from 1 to |P|:
    for j from i+1 to |P|:
      append (sim(P[i], P[j]), (i, j)) onto H
  let h be the max-heap resulting from calling build_max_heap on H

  // Place the photos into categories, w/appropriate data structures: O(m lg m).
  // Critically, each deleteMax takes lg(m) time, and there are O((n-c)^2) of them.
  // We'll treat this as O(m \lg m) = O(n^2 \lg n). (Recall that the graph is complete.)
  // Technically, for c nearly as large as n, this can run faster, but since
  // that is, practically speaking, an unimportant case, we use the simpler bound.
  let cats be a new union-find data structure containing 1, ..., |P|
  let num_cats = |P|
  while num_cats > c:
    // Find the highest-similarity edge remaining
    let (s, i, j) = findMax(h)
    deleteMax(h) // lg(m) time

    // Merge photos i and j, noting whether
    // the number of categoris has gone down.
    if find(cats, i) != find(cats, j):
      num_cats--
      union(cats, i, j)

  // Extract the categorization, w/appropriate data structures: O(n)
  let A be an array of (initially empty) sets of photos
  for i in range 1 to |P|:
    insert P[i] into A[find(i)]
  let S be an empty set (of sets of photos)
  for i in range 1 to |P|:
    if A[i] is non-empty:
      insert A[i] into S

  return S
```

Looking at the (rather scanty!) annotations I included above, we get a total runtime of $O(m+n^2 \lg n+n) = O(n^2 \lg n)$. In other words, we can build our categorization in time that's only slightly worse (a log-factor worse) than the amount of time to simply look at every edge. That's pretty great!

7 Challenge Your Approach

1. **Carefully** run your algorithm on your instances above. (Don't skip steps or make assumptions; you're debugging!) Analyse its correctness and performance on these instances:

   SOLUTION: Let's try it on this graph with $c = 1$. (This is silly, since we know what the answer should be with only one category, but it gives a better sense for how the algorithm works!)

   First, we create a max-heap that would produce the following edges in order if we call find/delete over and over: $(1,3), (4,5), (2,4), (2,5), (2,3), (1,5), \ldots$. (Note: $(2,4)$ and $(2,5)$ are tied. It turns out to be fine according to our metric to break these ties arbitrarily.)

   Next, we iterate through this list until we get down to one category. **WARNING:** this part will make no sense if you haven't read section 4.6. But you're reading the textbook or equivalent resources, right? Learn about the union-find data structure!

   Let's look at the edge at the heart of each iteration:

   $(1,3)$ Now 1 and 3 are in the same set, say 1. (If c were 4, we'd stop here.)

   $(4,5)$ Now 4 and 5 are in the same set, say 4. (If c were 3, we'd stop here.)

   $(2,4)$ Now 2 and 4 are in the same set. Since 4's set is larger, 2 goes into 4's set, which is named 4. (If c were 2, we'd stop here.)

   $(2,5)$ 2 and 5 are **already** in the same set; when we call `find` on them, we get back 4 for both. So, the sets don't change (nor does `num_cats`).

   $(2,3)$ Now 2 and 3 are in the same set. Since the set containing 2 is larger, we renumber 3's set to use 2's set's name (which is 4). At this point, everything is in set 4, and we stop (since `num_cats` = c=).

2. Design an instance that specifically challenges the correctness (or performance) of your algorithm:

   SOLUTION: We've sketched a proof that this is correct. So, we won't try to challenge its correctness!