Graph-Based Techniques for Image Segmentation

Avi Kak

Purdue University

Thursday 5th December, 2024 10:00

Preamble

In general, a graph-based algorithm allows you to aggregate the pixels that are similar to one another in some loose sense and that, taken together, are dissimilar from the background pixels in the image.

While you can certainly apply the graph-based algorithms (more commonly referred to as the **graph-partitioning algorithms** in the context of this lecture) at the pixel level, **you are more likely to use them at a higher level of aggregation as you try to extract meaningful objects from the photo of a scene**. Aggregating pixels at the local level is relatively easy — all you have to do is to group them together based on the similarity of, say, color, texture, etc. — but, after you have formed small blobs through local aggregation, joining them into more meaningful objects is much challenging. And, that's where graph-partitioning algorithms can help.

Another way of saying the same thing as above is that if you have an algorithm that is over-segmenting images, you might want to explore using graph-partitioning methods for a higher-level aggregation of the blobs produced by the algorithm so that your final output corresponds to to meaningful objects — things like people, buildings, trees, roads, things that move on the road, etc.

Preamble (contd.)

Here is a classic example of a publication that illustrates the message presented on the previous slide — using graph-partitioning to merge together the blobs by lower-level aggregation of the pixels:

https://engineering.purdue.edu/RVL/Publications/Martinez04OnCombining.pdf

For another example of using graph-partitioning for merging the blobs generated through over-segmentation, see the problem definition in Section 6 and **the results in Section 10** of my tutorial on clustering data on manifolds:

https://engineering.purdue.edu/kak/Tutorials/ClusteringDataOnManifolds.pdf

The graph-partitioning algorithms I present in this lecture are implemented in my YOLOLogic module:

https://engineering.purdue.edu/kak/distYOLO/

The best way to learn from the graph-partitioning part of YOLOLogic is through the subdirectory **ExamplesRegionProposals** of the module.

Outline

Outline

Representing an Image with a Graph

- As mentioned in the Preamble, graph-based methods are often indispensable for merging the blobs produced by an image segmentation algorithm that tends to over-segment the images.
- Even though their most frequent application is at a level higher than than that of individual pixels, it is nevertheless easier to explain the basic idea of graph-based partitioning if we assume that we are operating at the pixel level.
- In keeping with the spirit mentioned above, let's represent an image with a graph $G = (V, E)$ in which, at the beginning, the vertices in V are the individual pixels and an edge in E between a pair of vertices is a measure of the similarity of the two pixels on the basis of a user-specified criterion.
- In the simplest cases, the similarity may depend directly on the difference between the color values at the pixels, their immediate neighborhoods, and also on how far apart the pixels are.

Representing an Image with a Graph (contd.)

- Given the representation on the previous slide, the goal of graph partitioning would be to partition an image into regions so that the total similarity weight in each region is maximized, while it is minimized for all pixel pairs involving pixels are in different regions.

- In other words, we want to partition a graph G into disjoint collections of vertices so that the vertices in each collection are maximally similar, while, at the same time, the collections are maximally dissimilar from one another.

- A central notion in graph partitioning algorithms is that of *graph Laplacian*, as you will see in this lecture. It is the eigendecomposition of the graph Laplacian that can yield a usable image segmentation. The algorithms of the sort we will talk about are also known as the *graph spectral clustering algorithms.* [**You must be puzzling over the fact that a graph, which has the flavor of a network of connections (something that, at some level of reckoning, is purely symbolic), can lend itself to a deeply numerical processing step as represented by eigendecomposition. If that's the case, hang on to your sense of wonderment. You are in for a treat! As you will see, human ingenuity has no bounds.**]

Representing an Image with a Graph (contd.)

- For some key fundamental notions related to graph partitioning, I recommend going through the tutorial "A Tutorial on Spectral Clustering" by Luxburg that is available at:

  https://arxiv.org/abs/0711.0189

- So far I have only talked about using graph partitoning for grouping pixels into regions. However, at a higher level of implementation, if you have some other process that outputs blobs of pixels, you can model each blob as a node in a graph and then use graph partitioning to merge the nodes into more meaningful constructs.

- This higher level application of graph-based algorithms will be demonstrated in the other sections of this presentation.

Outline

On Partitioning a Graph

- Let's say an image has N pixels, with each pixel represented by the index i whose values go from 0 through $N-1$. [Instead of using a pair of indices (i, j) for identifying a pixel through its location in an array, we are using a single index i as a pixel identifier. As you'll see, this makes it easier to bring in the graph-theoretic notions.]
- Let w_{ij} express the similarity of the pixel i to pixel j. The quantity w_{ij} could depend on, say, (1) the color difference at the two pixels; and (2) inversely on the distance between the two pixels.
- We could also make w_{ij} proportional to some attribute of local grayscale or color variations in the neighborhoods around the pixels i and j, as used in the Census Transform.
- As mentioned earlier, we represent an image by a graph $G = (V, E)$, where the *vertex set* V is the set of pixels, indexed 0 through $N-1$, and E the set of edges between the vertices. Given two vertices i and j, we set $E_{ij} = w_{ij}$.

On Partitioning a Graph (contd.)

- In general, in using graph-theoretic tools for image segmentation, our goal would be to partition V into subsets $\{V_1, V_2, ..., V_K\}$ for some user-specified K and we would do so subject to the following two criteria:
  - We would want to **maximize** the similarity weight between pairs of pixels within each partition; and
  - We would want to **minimize** the similarity weight associated with the links *between* the partitions.
- However, for this presentation, I'll assume that we just want to carry out "figure-ground separation" in the image. That is, we want to solve the problem of **bipartition**, meaning that we want to create two disjoint partitions A and B from the graph subject to the above optimality criteria.

On Partitioning a Graph (contd.)

- For any bipartition (A, B) of V, we can associate a value $cut(A, B)$ with the partition as follows:

$$cut(A, B) = \sum_{i \in A, j \in B} w_{ij}$$

- The edges that go from any vertex in A to any vertex in B are referred to as the **cutset** of the partition. Some folks like to use the word **cutset** only for an optimum partition in which the value of $cut(A, B)$ is the least it can be.

- For a moment, let's consider a given (A, B) to be an 'optimum' partition if, from all possible choices for such partitions, it minimizes the value of $cut(A, B)$.

- To gain insights into what we might achieve by minimizing $cut(A, B)$, consider a simple graph (that obviously does not represent an image):

```
          5               8                5
      a----------b-----------------   h-------------i
     /            \                  /               \
    /5             \5      2        /5                \5
   /                \              /                   \
  c                  d-----------g                      j
   \                /              \                   /
    \5             /5               \5                /5
     \            /                  \               /
      e----------f                    k-------------m
          5                                 5
```

To Gain Insights into What $cut(A, B)$ Stands For

- For the example on the previous slide, a solution that minimizes the cutset weight consists of using edges $\{\{b, h\}, \{d, g\}\}$ as the cut set. This optimum solution has a cutset weight of 10.

- Note that the optimum cutset is not unique in the example on the previous slide. Here is another cutset that also has the cutset weight of 10: $\{\{h, i\}, \{k, m\}\}$. There exist additional solutions also. For example, $\{\{m, j\}, \{i, j\}\}$, with one partition containing only one vertex, etc.

- For another example, consider the following graph:

```
             5              8                   5
        a-----------b-------------------h-------------i
       /             \                 /
      /5              \5              /5
     /                 \      2      /
    c                   d-----------g                  j
     \                 /             \                /
      \5              /5              \5             /5
       \             /                 \            /
        e-----------f                   k------------m
             5                                5
```

To Gain Insights into What $cut(A, B)$ Stands For (contd.)

- As with the first example, the example shown on the previous slide again has a number of cutsets, especially if we include cutsets that result in one-vertex partitions. Of all the solutions that are possible, the following four are particularly interesting:

```
Solution 1:      cutset: { {b,h}, {d,g} }      cutset weight = 10

Solution 2:      cutset: { {b,h}, {g,h} }      cutset weight = 13

Solution 3:      cutset: { {a,b}, {e,f} }      cutset weight = 10

solution 4:      cutset: { {g,k} }             cutset weight = 5
```

- The optimum solution corresponds to Solution 4 with a cutset weight of 5.

Relevance of the Min-Cut Solution to Computer Vision

- It was shown by Greig, Porteous, and Seheult in a paper "Exact Maximum A Posteriori Estimation for Binary Images" way back in 1989 that the problem of finding the best MAP solution to the restoration of binary images can be cast as a min-cut problem. The solution obtained with min-cut was superior to the one obtained with simulated annealing.

- The best way to solve a min-cut problem is with the **max-flow algorithms**. **These algorithms have low-order polynomial-time complexity**.

- As for the name "max-flow" for the algorithms, one can show that in a network of pipes for transporting, say, oil, the maximum flow capacity between any two given points is determined by the pipes that are in the min-cut of the graph that describes the pipe network.

Moving on to Normalized Cuts

- Unfortunately, the min-cut solutions do not always work for solving computer vision problems. They frequently result in highly unbalanced graph bipartitions, unbalanced to the extent that one of the partitions may consist of just a single pixel.

- Of the graph-based algorithms, **what has worked for image partitioning is the minimization of the Normalized Cut criterion**. This criterion, denoted $Ncut$, seeks a bipartition (A, B) of a graph $G = (V, E)$ that minimizes the following:

$$
\begin{aligned}
NCut(A, B) &= \frac{cut(A, B)}{assoc(A, V)} + \frac{cut(B, A)}{assoc(B, V)} \\
assoc(A, V) &= \sum_{u \in A, t \in V} w_{ut}
\end{aligned}
$$

Outline

The Shi-Malik Algorithm for Minimizing *Ncut*

- Define an **indicator vector** $\vec{x}$ of size N assuming V is of cardinality N. The i^{th} element x_i of the vector $\vec{x}$ is $+1$ if vertex i is in A. Otherwise, x_i is -1.

- Define an N element vector $\vec{1}$ as consisting of all 1s.

- We can now express $cut(A, B)$ as

$$cut(A, B) = \sum_{x_i > 0, x_j < 0} -w_{ij} x_i x_j$$

- We associate with each vertex $v_i \in V$ the degree d_i defined by

$$d_i = \sum_{j=1}^{N} w_{ij}$$

Obviously, d_i is the sum of all the similarity weights emanating from the vertex i in the graph.

The Shi-Malik Algorithm for Minimizing *Ncut* (contd.)

- We place all the node degrees, d_i's, on the diagonal of an $N \times N$ matrix D, with its all other elements set to 0. D is called the *Degree Matrix*. Along the diagonal, the i^{th} element of D is d_i.

- In terms of the indicator vector elements x_i and the degrees d_i, we can now express the formula for the normalized cut as

$$Ncut(A,B) \quad = \quad \frac{\sum_{x_i>0, x_j<0} -w_{ij}x_i x_j}{\sum_{x_i>0} d_i} + \frac{\sum_{x_i<0, x_j>0} -w_{ij}x_i x_j}{\sum_{x_i<0} d_i}$$

- The ratios on the right can be expressed more compactly as

$$Ncut(\vec{x}) \quad = \quad \frac{(\vec{1}+\vec{x})^T(D-W)(\vec{1}+\vec{x})}{k\vec{1}^T D\vec{1}} + \frac{(\vec{1}-\vec{x})^T(D-W)(\vec{1}-\vec{x})}{(1-k)\vec{1}^T D\vec{1}}$$

where W is the matrix representation of the similarity weights w_{ij}. The unit vector $\vec{1}$ was defined on the previous slide. The quantity k is given by:

$$k \quad = \quad \frac{\sum_{x_i>0} d_i}{\sum_{i=1}^{N} d_i}$$

The Shi-Malik Algorithm for Minimizing *Ncut* (contd.)

- Note that k is a normalized sum of all the similarity weights in just the partition A. Therefore, $1-k$ would be a normalized sum of all the similarity weights in just the partition B.

- The expression for $Ncut(\vec{x})$ shown on the previous slide can be further simplified to

$$\begin{aligned} Ncut(\vec{x}) &= \frac{\vec{y}^T(D-W)\vec{y}}{\vec{y}^T D\vec{y}} \\ \vec{y} &= (\vec{1}+\vec{x}) - b(\vec{1}-\vec{x}) \\ b &= \frac{k}{1-k} \end{aligned}$$

- The form $D-W$ is famous unto itself. Recall that D is a diagonal matrix whose i^{th} element is sum of all the similarity weights emanating from the vertex i in the graph.

Graph Laplacian and Its Properties

- The matrix $L = D - W$ is known as the *graph Laplacian* of a similarity matrix W. It has the following interesting properties:
  1. L is symmetric and positive semidefinite;
  2. Its smallest eigenvalue is always 0 and the corresponding eigenvector is $\vec{1}$, meaning a vector of all 1's.; and
  3. All of its eigenvalues are non-negative.

- On the next slide you will see a variant of the graph Laplacian known as the symmetric normalized graph Laplacian and given by

$$L_{sym} = D^{-\frac{1}{2}}(D - W)D^{-\frac{1}{2}}$$

- The form for $Ncut(\vec{x})$ shown on the previous slide is an example of what is known as the Raleigh Quotient that looks like

$$R(A, \vec{x}) \quad = \quad \frac{\vec{x}^T A \vec{x}}{\vec{x}^T \vec{x}}$$

Using the Raleigh Quotient for Minimization

- For a given matrix A the vector $\vec{x}$ that minimizes its Raleigh Quotient is the smallest eigenvector A.

- We get the Raleigh Quotient form exactly for *Ncut* if set

$$\vec{y} = D^{-\frac{1}{2}}\vec{z}$$

- Substituting the above in the formula for *Ncut*, we get

$$\min_{\vec{x}} Ncut(\vec{x}) = \min_{\vec{z}} \frac{\vec{z}^T L_{sym}\vec{z}}{\vec{z}^T\vec{z}}$$

  where L_{sym} was defined on the previous slide.

- It follows from the properties of the graph Laplacian as stated on the previous slide that L_{sym} is also symmetric positive semidefinite, that its smallest eigenvalue is 0, and that the corresponding eigenvector is $\vec{z} = D^{-\frac{1}{2}}\vec{1}$.

Using the Raleigh Quotient for Minimization (contd.)

- **We therefore use the next to the smallest eigenvalue and its corresponding eigenvector as the solution for $\vec{x}$ for bipartitioning a graph.**

Some Results Obtained with Ncut Minimization and Clustering

Here is a result from our own paper:

`https://engineering.purdue.edu/RVL/Publications/Martinez04OnCombining.pdf`

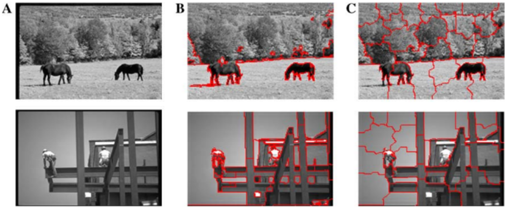

Fig. 9. For the two images shown, it is necessary to extract several segments corresponding to highly localized detail. (A) Original images. (B) Segmentations obtained using Koontz–Fukunaga clustering with $e_2 = 100$. (C) Segmentations obtained using k-means clustering with $e_2 = 100$.

Outline

The Felzenszwalb and Huttenlocher (FH) Algorithm

- The graph-based algorithm by Felzenszwalb and Huttenlocher has emerged as a strong competitor to the graph spectral clustering based method described in the previous section. You can download the FH paper from:

  https://cs.brown.edu/people/pfelzens/papers/seg-ijcv.pdf

- FH algorithm is a recursive merging algorithm in the graph $G = (V, E)$ representation of an image in which initially the vertices in V represent the individual pixels and the edges in E represent the pairwise similarity between the pixels. Subsequently, each vertex in V represents a blob obtained by previous merging steps, and each edge a measure of similarity between a pair of blobs.

- FH bases its blob merging decisions on two quantities: (1) the largest inter-pixel color difference at adjacent pixels within each blob ; and (2) the smallest inter-pixel difference for a pair of pixels that are in two different blobs.

The FH Algorithm (contd.)

- The largest value of the inter-pixel color difference at a pair of *adjacent pixels* **within a blob** is represented by $Int(u)$ for a vertex u in the graph.

- In order to account for the fact that, at the beginning, each vertex consists of only one pixel [which would not allow for the calculation of $Int(u)$], the unary property of the pixels at a vertex is extended from $Int(u)$ to $MInt(u)$ with the addition of a vertex-size dependent number equal to $k/|C|$ where k is a user-specified parameter and $|C|$ the cardinality of the set of pixels represented by the vertex u in the graph.

- As mentioned above, initially the edges in the graph representation of an image are set to the color difference between the two 8-adjacent pixels that correspond to two *different* vertices, meaning to two different blobs.

The FH Algorithm (contd.)

- That is, initially, the edge $E(u, v)$ between two vertices u and v in the graph is set to the inter-pixel color difference for two adjacent pixels. In subsequent iterations, $E(u, v)$ is set to the smallest inter-pixel color difference between the blobs represented by the nodes u and v. For this measurement, the two blobs must be adjacent, that is, at least one pixel in one blob u must be an 8-neighbor of some pixel in blob v. The value of $E(u, v)$ is smallest such inter-pixel color difference for two such adjacent pixels.

- At each iteration of the algorithm, two vertices u and v are merged provided $E(u, v)$ is less than the smaller of the $MInt(u)$ or $MInt(v)$ attributes at the two vertices. My experience is that for most images the algorithm terminates of its own accord after a small number of iterations while the vertex merging condition can be satisfied.

The FH Algorithm (contd.)

- Since the algorithm is driven by the color differences between 8-adjacent pixels — within the individual blobs and across two different blobs at their common boundary — the FH algorithm is likely to create too fine a segmentation of an image.

- The segments produced by FH can be made larger by using the logic of SS that allows blobs of pixels to merge into larger blobs provided doing so makes sense based on the inter-blob values for mean color levels, color variances, texture values, etc.

Outline

YOLOLogic's Implementation of FH

- In FH, the fundamental condition for merging the pixel blobs corresponding to two different vertices `u` and `v` is: The value of the `E[u,v]`, which keeps track of the smallest inter-pixel color difference for two pixels that are each other's 8-neighbors, with one pixel in blob `u` and the other in blob `v`, must be less than the smaller of the `Int` values for either of the two blobs.

- In order to carry out this comparison efficiently, the code you see in Slide 34 also associates another attribute with each edge, `Mint[u,v]`, that stores the minimum of the `Int` values at `u` and `v`.

- When two blobs `u` and `v` are merged, they must both be dropped from the list of blobs maintained by the system and we must also drop the edge corresponding to them.

- It stands to reason that the edges that do not satisfy the condition for merging of the blobs represented by the vertices at their two ends should be eliminated from further consideration. This is what is accomplished by the statements labeled (B), (C), and (D) in Slide 34.

YOLOLogic's Implementation of FH (contd.)

- After the step at the end of the last slide, all other edges are good candidates for the merging of the blobs represented by their vertex ends.

- From the remaining edges, we now take up one edge at a time in order of increasing value of the `Mint` value associated with the edges. This is on account of the call to `sorted()` in line (A).

- Therefore the edge whose vertices we want in line (E) is the one with the smallest value of `Mint` value, that is, with the smallest of either of the two values for `Int` attribute at the two vertices that define the edge.

- One thing that requires care in coding FH is that we want to go through all the edges in the `sorted_edges` while we are deleting the vertices that are merged and the edges that are no longer relevant because of vertex deletion.

YOLOLogic's Implementation of FH (contd.)

- You have to be careful when debugging the code in the the main `for` loop. The problem is that the sorted edge list is made from the original edge list which is modified by the code in the `for` loop.
- Let's say that the edge `(u,v)` is a good candidate for the merging of the pixel blobs corresponding to u and v. After the main `for` loop shown on the next slide has merged these two blobs corresponding to these two vertices, the u and v vertices in the graph do not exist and must be deleted. Deleting these two vertices requires that we must also delete from E all the other edges that connect with either u and v.
- Shown on the next slide is YOLOLogic's main loop that implements the FH algorithm.

YOLOLogic's Implementation of FH (contd.)

```
V,E = graph
sorted_vals_and_edges = list( sorted( (v,k) for k,v in E.items() ) )
sorted_edges = [x[1] for x in sorted_vals_and_edges]
edge_counter = 0
for edge in sorted_edges:                                                                   ## (A)
    if edge not in E: continue
    edge_counter += 1
    if E[edge] >  MInt[edge]:                                                               ## (B)
        del E[edge]                                                                         ## (C)
        del MInt[edge]                                                                      ## (D)
        continue
    ###  Let us now find the identities of the vertices of the edge whose two vertices
    ###  are the best candidates for the merging of the two pixel blobs.
    vert1,vert2 = int(edge[:edge.find(',')]), int(edge[edge.find(',')+1 :])               ## (E)

    if (vert1 not in V) or (vert2 not in V): continue
    affected_edges = []
    for edg in E:
        end1,end2 = int(edg[:edg.find(',')]), int(edg[edg.find(',')+1 :])
        if (vert1 == end1) or (vert1 == end2) or (vert2 == end1) or (vert2 == end2):
            affected_edges.append(edg)
    if self.debug:
        print("\n\n\naffected edges to be deleted: %s" % str(affected_edges))
    for edg in affected_edges:
        del E[edg]
        del MInt[edg]
    merged_blob = V[vert1] + V[vert2]
    V[index_for_new_vertex] = merged_blob
    if self.debug:
        print("\n\n\n[Iter Index: %d] index for new vertex: %d   and the merged blob: %s" % (master_iteration_index, index_for_new_vertex, str(merged_blob)))
    ###   We will now calculate the Int (Internal Difference) and MInt property to be
    ###   to be associated with the newly created vertex in the graph:
    within_blob_edge_weights = []
    for u1 in merged_blob:
        i = u1[0] * arr_width + u1[1]
        for u2 in merged_blob:
            j = u2[0] * arr_width + u2[1]
            if i > j:
                ij_key = "%d,%d" % (i,j)
                if ij_key in initial_graph_edges:
                    within_blob_edge_weights.append( initial_graph_edges[ ij_key ] )
    Int_prop[index_for_new_vertex] = max(within_blob_edge_weights)
    MInt_prop[index_for_new_vertex] = Int_prop[index_for_new_vertex] + kay / float(len(merged_blob))
    ###   Now we must calculate the new graph edges formed by the connections between the newly
    ###   formed node and all other nodes.  However, we first must delete the two nodes that
    ###   we just merged:
    del V[vert1]
    del V[vert2]
    del Int_prop[vert1]
    del Int_prop[vert2]
    del MInt_prop[vert1]
    del MInt_prop[vert2]
    for v in sorted(V):
        if v == index_for_new_vertex: continue
        ###   we need to store the edge weights for the pixel-to-pixel edges
        ###   in the initial graph with one pixel in the newly constructed
        ###   blob and other in a target blob
        pixels_in_v = V[v]
        for u_pixel in merged_blob:
            i = u_pixel[0] * arr_width + u_pixel[1]
            inter_blob_edge_weights = []
            for v_pixel in pixels_in_v:
                j = v_pixel[0] * arr_width + v_pixel[1]
                if i > j:
                    ij_key = "%d,%d" % (i,j)
                else:
                    ij_key = "%d,%d" % (j,i)
                if ij_key in initial_graph_edges:
                    inter_blob_edge_weights.append( initial_graph_edges[ij_key ] )
            if len(inter_blob_edge_weights) > 0:
                uv_key = str("%d,%d" % (index_for_new_vertex,v))
                E[uv_key] = min(inter_blob_edge_weights)
                MInt[uv_key] = min( MInt_prop[index_for_new_vertex], MInt_prop[v] )
    index_for_new_vertex = index_for_new_vertex + 1
```

Some Results Obtained with FH as Reported by the Authors

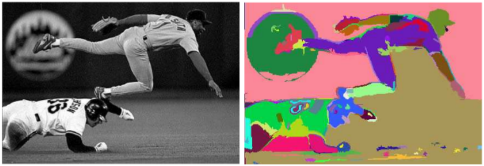

Figure 3: A baseball scene (432×294 grey image), and the segmentation results produced by our algorithm ($\sigma = 0.8$, $k = 300$).

Figure 4: An indoor scene (image 320×240, color), and the segmentation results

Outline

The Selective Search (SS) Algorithm for Region Proposals

- If for whatever reason an algorithm for generating region proposals is creating too fine a division of the image, you can use the Selective Search (SS) algorithm proposed by Uijlings, van de Sande, Gevers, and Smeulders for merging them into larger proposals. You can access their paper at:

  `http://www.huppelen.nl/publications/selectiveSearchDraft.pdf`

- In the publication mentioned above, the logic of SS sits on top of the image partitions produced by the FH algorithm and that's what I have also implemented in my Python module `YOLOLogic`.

Outline

The Selective Search (SS) Algorithm for Region Proposals

- In the YOLOLogic module, the recursive blob-merging logic of SS is based on comparing the adjacent pixel blobs on the basis of the following four properties:
  1. pairwise adjacency
  2. pairwise comparison of color homogeneity
  3. pairwise comparison of grayscale variance
  4. pairwise comparison of LBP textures

  In order to appreciate what is meant by color homogeneity and LBP (Local Binary Patterns) texture values, I'll refer the reader to the following report by me:

  `https://engineering.purdue.edu/kak/Tutorials/TextureAndColor.pdf`

- The YOLOLogic module maintains an ever-increasing integer index as the ID for each blob and associates a `pair_id` with every pair of blobs discovered in an image as shown in the code fragment on the next slide.

SS Algorithm (contd.)

- The comparative properties for each pair of blobs are calculated as shown below and subsequently stored in the dictionary `all_pairwise_similarities`:

```
for blob_id_1 in pixel_blobs:
    for blob_id_2 in pixel_blobs:
        if blob_id_1 > blob_id_2:
            pair_id = str("%d,%d" % (blob_id_1,blob_id_2))
            pairwise_adjacency[pair_id] = True if pair_id in E else False
            pairwise_gray_homogeneity_val[pair_id] = abs(gray_mean_vals[blob_id_1] - gray_mean_vals[blob_id_2])
            pairwise_color_homogeneity_val[pair_id] = [abs(color_mean_vals[blob_id_1][j]
                                      - color_mean_vals[blob_id_2][j]) for j in range(3)]
            pairwise_gray_var_comp[pair_id] = abs(gray_vars[blob_id_1] - gray_vars[blob_id_2])
            pairwise_texture_comp[pair_id] =  np.linalg.norm(texture_vals[blob_id_1] - texture_vals[blob_id_2])
all_pairwise_similarities['adjacency'] = pairwise_adjacency
all_pairwise_similarities['color_homogeneity'] = pairwise_color_homogeneity_val
all_pairwise_similarities['gray_var'] = pairwise_gray_var_comp
all_pairwise_similarities['texture'] = pairwise_texture_comp
```

- The unary properties of the blobs are calculated as follows. Note that we consider the blobs in the reverse order of their sizes — just in case we want to ignore the very smallest of the blobs. With the option `reverse=True`, `sorted` returns a list in descending order of the sorting criterion:

```
sorted_blobs = sorted(pixel_blobs, key=lambda x: len(pixel_blobs[x]), reverse=True)
for blob_id in sorted_blobs:
    pixel_blob = pixel_blobs[blob_id]
    pixel_vals_color = [im_array_color[pixel[0],pixel[1],:].tolist() for pixel in pixel_blob]
    pixel_vals_gray =  np.array([im_array_gray[pixel] for pixel in pixel_blob])
    color_mean_vals[blob_id]  = [ float(sum([pix[j] for pix in pixel_vals_color])) / float(len(pixel_vals_color)) for j in range(3) ]
    gray_mean_vals[blob_id] =  np.mean(pixel_vals_gray)
    gray_vars[blob_id] = np.var(pixel_vals_gray)
    texture_vals[blob_id] = estimate_lbp_texture(pixel_blob, im_array_gray)
```

SS Algorithm (contd.)

- Shown below is YOLOLogic's logic for merging the blobs recursively:

```
while ss_iterations < 1:
    sorted_up_blobs = sorted(merged_blobs, key=lambda x: len(merged_blobs[x]))
    sorted_down_blobs = sorted(merged_blobs, key=lambda x: len(merged_blobs[x]), reverse=True)
    for blob_id_1 in sorted_up_blobs:
        if blob_id_1 not in merged_blobs: continue
        for blob_id_2 in sorted_down_blobs[:-1]:        # the largest blob is typically background
            if blob_id_2 not in merged_blobs: continue
            if blob_id_1 not in merged_blobs: break
            if blob_id_1 > blob_id_2:
                pair_id = "%d,%d" % (blob_id_1,blob_id_2)
                if (pairwise_color_homogeneity_val[pair_id][0] < self.color_homogeneity_thresh[0])\
                   and                                                                           \
                   (pairwise_color_homogeneity_val[pair_id][1] < self.color_homogeneity_thresh[1])\
                   and                                                                           \
                   (pairwise_color_homogeneity_val[pair_id][2] < self.color_homogeneity_thresh[2])\
                   and                                                                            \
                   (pairwise_gray_var_comp[pair_id] < self.gray_var_thresh)                       \
                   and                                                                            \
                   (pairwise_texture_comp[pair_id] < self.texture_homogeneity_thresh):
                    if self.debug:
                        print("\n\n\nmerging blobs of id %d and %d" % (blob_id_1, blob_id_2))
                    new_merged_blob = merged_blobs[blob_id_1] +  merged_blobs[blob_id_2]
                    merged_blobs[next_blob_id] = new_merged_blob
                    del merged_blobs[blob_id_1]
                    del merged_blobs[blob_id_2]
                    ###  We need to estimate the unary properties of the newly created
                    ###  blob:
                    pixel_vals_color = [im_array_color[pixel[0],pixel[1],:].tolist() for pixel in
                                                                            new_merged_blob]
                    pixel_vals_gray = np.array([im_array_gray[pixel] for pixel in new_merged_blob])
                    color_mean_vals[next_blob_id]  = [float(sum([pix[j] for pix in \
                        pixel_vals_color])) / float(len(pixel_vals_color)) for j in range(3)]
                    gray_mean_vals[next_blob_id] = np.mean(pixel_vals_gray)
                    gray_vars[next_blob_id] = np.var(pixel_vals_gray)
                    texture_vals[next_blob_id] = estimate_lbp_texture(new_merged_blob, im_array_gray)
                    ###  Now that we have merged two blobs, we need to create entries
                    ###  in pairwise dictionaries for entries related to this new blob
                    for blb_id in sorted_up_blobs:
                        if blb_id not in merged_blobs: continue
                        if next_blob_id > blb_id:
                            pair_id = "%d,%d" % (next_blob_id, blb_id)
                            pairwise_adjacency[pair_id] = \
          True if are_two_blobs_adjacent(new_merged_blob, pixel_blobs[blb_id]) else False
                            pairwise_color_homogeneity_val[pair_id] = \
          [abs(color_mean_vals[next_blob_id][j]  - color_mean_vals[blb_id][j]) for j in range(3)]
                            pairwise_gray_homogeneity_val[pair_id] = \
                                 abs(gray_mean_vals[next_blob_id] - gray_mean_vals[blb_id])
                            pairwise_gray_var_comp[pair_id] = \
                                 abs(gray_vars[next_blob_id] - gray_vars[blb_id])
                            pairwise_texture_comp[pair_id] =  \
                        np.linalg.norm(texture_vals[next_blob_id] - texture_vals[blb_id])
            next_blob_id += 1
    ss_iterations += 1
```

Some Results Obtained with SS as Reported by the Authors

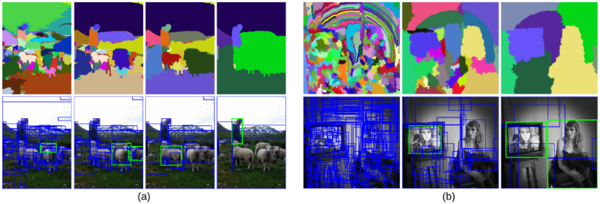

Figure 2: Two examples of our selective search showing the necessity of different scales. On the left we find many objects at different scales. On the right we necessarily find the objects at different scales as the girl is contained by the tv.