CS5300 S12 Undirected Graph Components in MapReduce

April 5, 2012

1 Introduction:

These notes discuss how to compute the connected components of a (restricted) class of undirected planar graphs efficiently in MapReduce.

For motivation, think of a digitized satellite image of a forest. We will treat the image as a grid, in which each grid cell either contains a tree or not. During a forest fire, we assume flames can leap from a tree in one cell to a tree in an adjacent cell, but no farther. Thus, the fire can spread from position p to position q if and only if there is a "path" of trees in adjacent cells that allows flames starting at p to leap from tree to adjacent tree until they reach q.

We can represent this as an undirected graph, in which each vertex represents a tree (with a "position") and there is an undirected edge connecting any pair of vertices whose positions are adjacent. A fire can spread between two trees iff there is an undirected path between the trees, which is true iff the trees are in the same *connected component* of the graph.

This is illustrated in Figure 1.

Below we formalize this and develop an efficient "divide-and-conquer" algorithm.

2 Preliminaries

To represent a graph, we use an $m \times m$ Boolean (here 0/1 rather than **true**/**false**) matrix

$$\mathbf{A} = \begin{pmatrix} a_{0,m-1} & a_{1,m-1} & \cdots & a_{m-1,m-1} \\ \vdots & \vdots & & \vdots \\ a_{0,1} & a_{1,1} & \cdots & a_{m-1,1} \\ a_{0,0} & a_{1,0} & \cdots & a_{m-1,0} \end{pmatrix}$$

Note we display this matrix unconventionally – the (i, j) entry is in the i^{th} column and j^{th} row – and the coordinates are 0-origin rather than 1-origin.

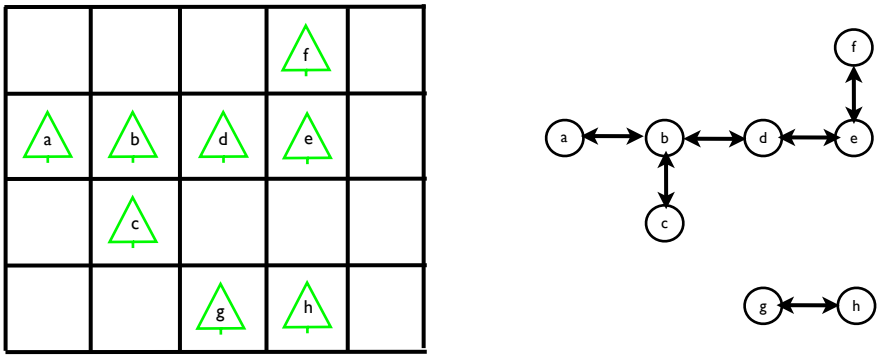

Figure 1: Satellite image of forest and corresponding graph

We want to make it easier to think of (i, j) as spatial coordinates (x, y). The 1-entries in this matrix are the positions of vertices in an $m \times m$ grid.

We use a letter among $p, q, \ldots$ to denote a *position* in a linearized representation of $\mathbf{A}$. The positions are just natural numbers, using column-major order, so the position corresponding to $\langle x, y \rangle$ is $mx + y$. You can verify that distinct (x, y) pairs map to distinct positions, provided x and y are strictly less than m. Obviously, positions are totally ordered by $<$. Further, it is simple to decide whether two positions are (horizontally or vertically) adjacent: Roughly, p and q are adjacent vertically if they differ by ± 1, and are adjacent horizontally if they differ by $\pm m$. (That is not *exactly* right – you need to be careful with positions near the boundaries of $\mathbf{A}$ – but that is not too difficult). And of course, p and q are adjacent if and only if q and p are adjacent.

Henceforth, we represent a vertex by its position number in $\mathbf{A}$. and we represent a path as a sequence adjacent positions:

$$p_0, p_1, \ldots, p_k$$

where for all $i < k$

$$p_{i+1} \text{ is adjacent to } p_i$$

We also write

$$p \rightsquigarrow q \quad \text{or} \quad p \leftrightsquigarrow q$$

to mean there is a path between p and q, and

$$p \rightsquigarrow_{\mathbf{S}} q \quad \text{or} \quad p \leftrightsquigarrow_{\mathbf{S}} q$$

(where **S** is a set of positions) to mean there is a path between p and q that stays entirely in **S**.

Suppose you wish to compute the connected components of the graph represented by **A**, using Hadoop MapReduce. In lecture we discussed computation of undirected connected components using the UNION-FIND algorithm. But that algorithm requires all the points to be in memory at once. For a large graph, it is not feasible to store all of **A** in an Amazon EC2 small instance – particularly using Java data structures. Thus, we need to partition **A** into blocks that will fit into a single reducer task, and devise some way to combine the results of the reducer tasks into a complete solution.

3 Column Groups

Choose an integer g that divides m; this will be the *column group width.* We will represent **A** as the union of column groups $G_0, G_1, \ldots, G_{m/g-1}$ where

$$G_i = \begin{pmatrix} a_{ig,m-1} & a_{ig+1,m-1} & \cdots & a_{ig+g,m-1} \\ \vdots & \vdots & & \vdots \\ a_{ig,1} & a_{ig+1,1} & \cdots & a_{ig+g,1} \\ a_{ig,0} & a_{ig+1,0} & \cdots & a_{ig+g,0} \end{pmatrix}$$

Note that the column groups are not disjoint: a column whose x index is a multiple of g is shared by two adjacent column groups. We call these shared columns *boundary columns.*

The position numbers of a column group are an interval (a contiguous subrange) of the natural numbers. The i^{th} column group corresponds to positions

$$igm, igm+1, igm+2, \ldots, (i+2)gm-2, (i+2)gm-1$$

The smallest of these position numbers represents the "lower left corner" of the Column group, the largest represents the "upper right corner," and the order of traversal looks like

$$G_i = \begin{pmatrix} \uparrow & & \uparrow & & \cdots & & \uparrow \\ \vdots & \searrow & \vdots & \searrow & & \searrow & \vdots \\ \uparrow & & \uparrow & & \cdots & & \uparrow \end{pmatrix}$$

The number of vertices in a column group is the number of 1 entries in the corresponding columns of **A**, which is at most $O(mg)$.

4 Algorithm

Our "efficient" MapReduce algorithm will be based on the following observation: If there is a path

$$p \rightsquigarrow q$$

then there exist column groups $G_{i_1}, \ldots, G_{i_k}$ and boundary positions $r_1, \ldots, r_{k-1}$ such that

$$p \rightsquigarrow_{G_{i_1}} r_1 \rightsquigarrow_{G_{i_2}} r_2 \ldots \rightsquigarrow_{G_{i_{k-1}}} r_{k-1} \rightsquigarrow_{G_{i_k}} q$$

That is, any path can be broken into subpaths, with each subpath entirely contained in a single column group, and the subpaths connected together at boundary positions. So our approach will be first to compute connected components of each column group, each in an independent reduce task, and then to combine the results of the column groups. The algorithm proceeds as follows:

1. Compute connected components of the column groups independently, each in its own reduce task, using UNION-FIND or any other connected component algorithm. The reducer for column group G will receive tuples of the form

   $\langle G, p \rangle$ where p is a position of a vertex in G

   Note positions in boundary columns are in two column groups, and thus must be sent to two reducers.

   For column group G the result will be the set $\mathbf{C}_G$ of tuples

   $\langle G, p, q \rangle$ where $q = \min\{r | p \leftrightsquigarrow_G r\}$

   That is, we "name" a connected component by the lowest position number it contains – informally, its "southwest corner." Note this corner position may or may not be in a boundary column itself.

   The space required by the reducer to compute this is just the size of G, which is $O(gm)$. We project this onto the boundary columns, that is $\mathbf{C}_G^b$ will be the set of tuples

   $\langle G, p, q \rangle \in \mathbf{C}_G$ where p is a boundary column

   Note $\mathbf{C}_G^b$ can be represented in $O(m)$ space.

2. Now another MapReduce pass using a single reducer can compute $\mathbf{C}^b$, the set of tuples

   $\langle p, q \rangle$ where $q = \min\{r | p \leftrightsquigarrow r\}$ and p is in a boundary column

   from the union of all the $\mathbf{C}_G^b$ sets for all column groups G. Again this can be done by UNION-FIND or any other connected component algorithm. Each of these sets requires $O(m)$ space, and there are m/g such sets, so the space required by this reducer is $O(m^2/g)$. This is also the size of $\mathbf{C}^b$.

3. Another MapReduce pass enables us to compute the connected components. There will be a reducer for each column group G. Its input will be the union of two sets:

   $\langle G, p, q \rangle \in \mathbf{C}^b$ where p is in G

and

$\langle G, p, p \rangle$ where p is the position of a vertex in G

(Position p is replicated in the second set of tuples so both sets will have the same arity). Again using UNION-FIND, each reducer can compute a portion of the set C of connected components. The reducer for column group G_i computes

$\langle p, q \rangle$ where $q = \min\{r | p \leftrightsquigarrow r\}$ and $igm \leq p < (i+1)gm$

You need to be a bit careful here to avoid double-counting the boundary columns, hence the arithmetic inequality on p rather than simply saying "$p \in G_i$." The union of these outputs gives the connected components of the graph.

5 Choosing g

There are two different resource restrictions presented above. For step (1) or (3), each reducer performs a UNION-FIND computation on points in some column group G; this requires $O(mg)$ space. For step (2), the reducer performs a UNION-FIND computation on the boundary positions. This requires $O(m^2/g)$ space. Our nodes must satisfy the maximum of these two memory requirements. In an asymptotic bound, max is equivalent to +, so we get

$$\text{reducer space} \quad = \quad O(mg + m^2/g)$$

Minimizing this wrt g, we get

$$0 \quad = \quad \frac{d}{dg}\left(mg + m^2/g\right) \quad = \quad \left(m - m^2/g^2\right)$$

so

$$g^2 \;=\; m$$

$$\text{reducer space is} \;\; O(m^{3/2})$$

Without any additional tricks on our part, this scales to reasonably large size problems. For example, consider an image grid with 256,000 positions on each side. This yields

$$m = 2^{18} \;\; \text{so} \;\; m^{3/2} = 2^{27} \approx 128 \text{ million}$$

A UNION-FIND data structure containing that many points requires a few gigabytes: which might be too big for an EC2 standard node, but fits comfortably in a large node.

The total number of reduce tasks is easily seen to be

$$\text{number of reducer tasks} \quad = \quad O(m/g) \quad = \quad O(m^{1/2})$$

which in this example is 512, also perfectly manageable.

6 Extending to Even Bigger Problems

Suppose you needed to perform this computation on a larger area – say all of Canada, for which m is something like 2^{22} or 2^{23}, so $m^{3/2}$ points will no longer fit in a single node?

Here is the outline of an approach.

1. Choose the largest value of g such that a column group of width g will fit in a node. Compute $\mathbf{C}_G^b$ for each column group using UNION-FIND as above. Note there will be *lots* of reduce tasks, but this is a big problem instance, so what do you expect?

2. Now you have the output of step (1), which takes up $O(m^2/g)$ space, and you need to compute $\mathbf{C}^b$, which again requires $O(m^2/g)$ space. This problem is too big to solve in a single reduce task. *But it is another instance of the undirected connected components problem, and it is a factor of* g *smaller than the problem we started with.* So solve this problem by recursive application of this technique.

3. With the output of steps (1) and (2), compute the answer as before, again with a space requirement of only $O(mg)$ for the reduce steps.

Note this algorithm does not scale indefinitely: Suppose m is so large that a single column of the input image won't fit in a reducer node?

The same approach works – you just need to partition the input image into rectangular blocks that are smaller than a column. The (messy) details are left for the interested reader.