Winter Internship : Exploring Methods to Perform Multi-Source BFS

> **NOTE***: If you are a prospective contributor, please scroll down to the bottom of the page*

Single source BFS, i.e BFS where we start with only one source node, has an established algorithm for the hadoop framework. The algorithm and the implementation is discussed here. Though this algorithm does use the parallel processing power of Haoop and MapReduce, it does seem that increasing the number of initial source nodes should give us a better load balancing in the initial exploration stages.

The major problem with such an approach is combining the different BFS trees formed from the different trees. We step down to solving the relatively simpler problem of finding the shortest path in an undirected graph using the BFS approach. Finding the shortest path between `s` and `t` is easy using the regular BFS method, the only thing we have to do is report the final distance of `t` from `s`. Extending it to multiple source nodes posses a challenge.

With this in mind we extended the BFS algorithm to incorporate two sources first. That is, now we are exploring the graph using two BFS Trees instead of one and using some analysis to combine the information. We choose the trees to originate from `s` and `t` and analyze the meeting points of the trees to calculate the shortest path between the two. The two source node BFS can be achieved by keeping two additional fields in the data per vertex used in the original BFS algorithm, holding the distance of a vertex from both the sources.

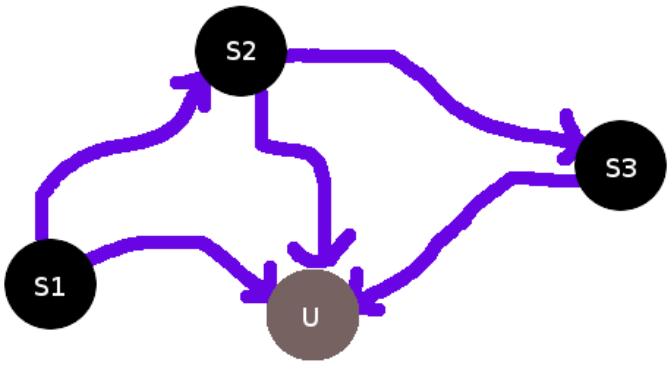
A more detailed explanation can be found in my previous report from the summer internship.

Extending to N Nodes

Extending to a general N node source presented a major implementation challenge. But before we try to understand it, lets recap the problem statement itself.

> We are trying to find an algorithm to calculate st connectivity and shortest path between two nodes in a large graph using the MapReduce framework. Our approach is to work on improving the existing BFS algorithm and finding out ways to start the BFS process from multiple sources so that the load is balanced to a greater degree.

Lets first start a BFS from three source nodes `s1, s2 and s3`. We need to find the shortest path information between say, `s1 and u` as shown in the figure that follows. Each data entry in the input in this approach will hold three additional entries - the distance of the node from each of the source nodes.



For now let us assume that the following case is encountered, i.e the following paths exist. Also assume that we know the shortest path distances between `s1, s2 and s3`. To find the shortest path in such a scenario, we have to find the shortest node among all the paths involving `s1 and u`. These paths are

- `s1-u`
- `s1-s2-u`

- `s1-s2-s3-u`

We have to find the minimum of these three combinations. As it can be clearly seen, the number of such paths will blow up as the number of source nodes increase. There seems to be no general way of solving this.

Another problem is termination condition. To terminate optimally for a three node case, we have to manually consider these 6 (**3!**) cases :

- `s-t-u` : trivial
- `u-t-s` : trivial
- `u-s-t` : trivial
- `t-s-u` : trivial
- `s-u-t` : if (`s-u + u-t < i`) break;
- `t-u-s` : if (`t-u + u-s < i`) break;

Where `s-t-u` indicates that `s` was discovered before `t` , which in turn was discovered before `u` . Further, here we require the shortest path between `s` and `t` . `i` here represents the level of iteration in the BFS tree.

An Alternative Approach 1

The above challenges meant that the approach used for two or even 3 sources cannot be scaled arbitrarily and we need a new approach. The following approach, which I proposed during the summer time was what I worked on during the winter is what I'm going to try and explain in the rest of the document.

Consider the following algorithm :

```
1) Run simultaneous BFS with N nodes as roots make sure s and t are among them.
2) Update the connectivity matrix.
3) Run shortest path algorithm on the matrix to find the shortest path.
4) if( shortest Path < 2*N) // Premature termination
        break
5) else
        goto 1
```

Implementation

Step 1 requires you to run simultaneous BFS from N nodes, i.e. running N parallel BFS trees. As we did with the two and three source BFS algorithms, this can be done by appending extra data columns to the input data indicating the distance or depth of the current node from each source. This is similar to how distance was measured in the PEP based BFS algorithm as outlined in the Wiki. This will, after each pass give you information regarding the discovery of nodes from each source.

Step 2 is the harder part. Keeping track of the N nodes will require us to indicate the updates to the context and the global runner. This requires the use of counters causing the implementation problems. The approach I followed was to use a counter for each pair of source points. For example, for a node with ID `node_id1` and `node_id2` , I created a counter by the name `node_id1-node_id2` and used it to indicate the connection of `id1` and `id2` .

These counters were then used to update the global NxN matrix of connectivity.

I never got to finding a decent implementation for step 2 and this still remains the part of the algorithm yet to be implemented in a clean an acceptable manner.

An alternative to the above approach is to check if the current node is a part of all the source nodes we started of with and if yes, reflect this on the global table. This is using the fact that one the shortest path between two nodes are found, it is never updated(since all future paths will be at least the length of the shortest). But this would require at least one of the following.

- A global method to pass all the source node such that it is accessible at all stages by all the nodes.
- The names/ids of all the source nodes saved with each data entry.

The first method was not available on the Hadoop implementation and the second seemed a lot of redundant space usage. This method is a possible improvement on the container based approach.

Step 3. At this point we have the information regarding connectivity of some of the source vertices. We can consider the NxN matrix as an adjacency matrix representation of some graph (namely, the connectivity graph of source vertices) and run an APSP algorithm on it to see if `s and t` are connected and find the shortest path between them. You need to note two things here:

1. The APSP algorithm and the NxN matrix is designed to be contained in the RAM and a RAM model algorithm like *Floyd-Warshall* can do the trick here.
2. The shortest path found is just the shortest among all the connectivity information we have till this iteration and this might be incomplete.

This means that at each iteration the shortest path can get updated.

Step 4 and 5 defines the termination conditions which have been discussed in the following section.

The Premature Termination Condition:

The premature termination condition is based on the following observations.

Let us assume we are at iteration `i` . This means that all the BFS trees have explored to a depth of at least i. Lets further assume that a new path between `s` and `t` were formed at this point. Consider the following deductions :

If the path included no other source nodes, it will be of form

```
s----t
```

The minimum length of such a path is 1 and the maximum is 2N. The minimum case should be obvious from the fact that this could be a single node path. The maximum is derived from the fact that two nodes newly discovered at the iteration `i` can be at a separation of at max `2i` and the maximum of this value is `2N` , when `i=N`

Further, if `s and t` had `1` other source nodes in between, the path will be of form

```
s --- u --- t
```

with the minimum becoming `2N + 1` and the maximum becoming `4N + 1` . Minimum when `s---u` and `u---t` are at minimum distance to each other (`N`) and maximum when they are at maximum distance (`2N`).

Continuing on this pattern, if there were `p` source nodes in between, the path would have a lower bound on the distance at `2N+p` and an upper bound at `2N*(p+1)` .

This leads us to the following conclusion:

> Lets the shortest path distance we have from a previous iteration is d and the current iteration is i, then if d <= 2i, no further iteration will lead to a shortest path and we can safely terminate.

Pseudo Code

```
SHORTEST_PATH(G, Source[T])
    1. Run MAP_REDUCE BFS with all node n in Source[] as root nodes.
    2. Update the connectivity matrix X
    3. Run SHORTEST_PATH_ADJ_LIST to find the shortest path between s and t in the matrix.
    4. if( shortest_path <= 2N)
    5.          break;
    5. else {
    6.          N++;
    7.          GOTO 1;
    8. }
```

View the code base here

Possible Advantages of the Algorithm

While working on the above idea I noticed the following advantages of the method:

1. The termination condition depends only on `N`, the sample size of source vertices and not the size of the graph. That is once the shortest path has been discovered, we can verify it in time proportional to the size of the sample of source virtices we took initially.
2. The choice of the `N` vertices can be arbitrary. The only restriction is that they should include our `s and t`. This can be used, with some modifications to find multiple SSSP quickly by selecting the appropriate source vertices. After all the `NxN` matrix we maintain is nothing but a connectivity graph of selected vertices.
3. The step 3, the *APSP* step is currently being run at each iteration. This has a complexity of at best **O(v^3)**. We can perform optimizations in this step, such as running *APSP* on odd passes or so.

An Alternative Approach 2

An alternative approach, proposed by Dr. Bera focuses on determining the diameter of the graph in a preprocessing step and the using this information to terminate the BFS exploration from a point where-in we know that no shorter path can be further found. I never got around to exploring this though. Here is his exact words.

> ...(try a method of) obtaining diameter, all-pairs shortest path on dense undirected graphs? Sample 100 nodes (100 is just a parameter) and form a subgraph G1. Do this for 100 subgraphs G2 ... G100, in parallel. Run the diameter finding algorithm on them in parallel -- they will give an upper bound on the diameter -- let this be d. Now run BFS, upto level d, from each node in parallel.

Work Flow for Future Contributor

> *NOTE : The links mentioned may become dead, please use the alt text to search for similar sites. NOTE : The code will most certainly become outdated because of the fast development cycle of Hadoop. Please user it only as a reference/starting point.*

There is a lot of scope in continuing the work done here and here is a guideline for someone starting from scratch on this project.

1. Read and understand the fundamentals of MapReduce framework and the non Ram based model we are working with. Give this part a little time.
2. Implement a few MapReduce examples and then implement the BFS algorithm as outlined here. You can refer to the wiki for more information.
3. Implement the modified versions of BFS outlined in this paper to understand the problem we are trying to solve.
4. Look into some MapReduce based clustering techniques such as this to understand various techniques which can be used to find subgraphs and understand the motivation behind the above algorithm.
5. Read the summer report I submitted outlining the above method.
6. Read through this document and the codebase here.

@Don Dennis (metastableB)