Chapter 1

Jaccard Coefficients

Contributed by Neil Butcher

1.1 Introduction

Jaccard Coeffcients is a proposed High Performance Computing (HPC) metric that is used in a wide variety of real world applications. The Jaccard metric is used as a way to define similarity between the neighborhood of two nodes. The definition of a neighborhood of a node is the adjacent vertices to a specific node. The Jaccard metric was originally introduced as a way to detect communities in botanical species [3]. This idea has been further expanded to other community detection algorithms[5] [1] [5], and for other purposes. The Jaccard coefficient has been used by Wikipedia [2] to determine the relationship between web-pages based upon common authors of pages. Jaccard is an interesting problem because it replicates the complexity of real world graph problems without being overly complex.

The example in figure 1.1 shows a problem that is best solved by using Jaccard coefficients. Often an insurance company will attempt to determine reliability of a person before offering them insurance. There are many ways that the reliability of a person can be determined, for instance people they have shared residence with. This can be represented as a graph problem, that can in fact be solved by Jaccard! This is one of many examples of problems that can be solved using the Jaccard coefficient. [4]

1.2 Basic Definition

The Jaccard coefficient gives a value that represents the similarity of the neighborhoods of two vertices. Given a pair of vertices U and V we represent Jaccard as the intersection of the neighborhoods of U and V, divided by the union of the neighborhoods of the two vertices. From a computational standpoint most work comes from computing the intersection of U and V. This is because to compute the union you can simply take the size of both the neighborhoods, add them together and then subtract by the size of the intersection, so that the shared vertices will only be counted once.

For examples look at 1.2. The figure shows a simple graph in which it is easy to demonstrate Jaccard computations. For example we demonstrate how to compute the Jaccard for vertex A and vertex D. The intersection of their neighborhoods is the single node, vertex B. The size of the union is two, nodes C and B. Note that despite that both A and D are neighbors of B, we only count B as one node in the union. This makes the Jaccard value 1/2. This simple example demonstrates the computation needed to compute a Jaccard value.

1.3 How to Compute Jaccard

There are many different ways that a Jaccard can be computed. Assuming that we are interested in only computing a single Jaccard value gives us a couple of options depending on how the data is presented to us. Given neighborhoods that are sorted based on vertex id greatly simplifies computing the intersection. To do this we iterate through each neighborhood simultaneously, checking for collisions. We only iterate on the list that currently has the smaller vertex id. Then once we have counted all the intersections we can compute the union in constant time given that we know the size of the neighborhoods. This results in a complexity of O(M), where M is the size of the neighborhoods. In the case that the neighborhoods are not sorted by vertex id we have two choices. We can sort the neighborhoods first O($Mlog(M)$), then we use our previous sorted neighborhoods algorithm. The other option is the just do a brute force comparison of the neighborhoods determine the intersection, which has a complexity of O(M^2). This may be more efficient then sorting, given that it may be a rather large constant to perform the sort.

Algorithm 1 JaccardPair

```
1: Given a pair of vertices U, V with sorted neighbor lists
2: intesect starts at 0
3: Nv is the current neighbor of V, starting with the first value in the list
4: Nu is the current neighbor of U, starting with the first value in the list
5: While Nv or Nu can still iterate
6: if Nv == Nu then
7:     intersect++
8: if Nv greater then Nu then
9:     Nu++
10: if Nu greater then Nv then
11:     Nv++
12: End While
```

Often real applications that want to utilize Jaccard coefficients require a Jaccard value for all pairs of vertices. The obvious solution is to just perform a brute force comparison and compute on all nodes. This gives a complexity of O(N^2), where N is the number of vertices in the graph. This can be further improved if we attempt to find a way to 'ignore' the '0' value Jaccard coefficients. This essentially comes down to ignoring vertices that do not share a two-hop path. If they do not share a two-hop path there it is impossible that they have any intersecting vertices. The pseudocode that performs a search of all two hop paths can be seen below. A summary of the concept is that starting from each vertex check all two hop paths, if you haven't computed the Jaccard yet on any two hop path you find, then compute it. Since this algorithm revolves around finding two hop paths the most benefit will be seen in the cases in which a graph is fairly sparse. We start by going through each of the verticies which is O(N). Then for each vertex we go through each of it's neighbor's neighborhoods. O(M^2), where M is the average size of a neighborhood. Then of course we have to accumulate for each unique two hop path we find.

Jaccard can also be computed by using the Graph Basic Linear Algebra Subroutines (GraphBLAS). The GraphBLAS library is a simple C interface that represents graphs as matrices and

Algorithm 2 Jaccard

```
1: for each Vertex V do
2:     for each Neighbor N of V do
3:         for each Neighbor J of N do
4:             intersect[V][J]++
```

performs matrix operations to perform graph algorithms. Jaccard can easily be transformed to a matrix operation, the intersection of a all pairs of nodes can be done by a matrix multiply operation. The values in the results matrix will end up representing the intersection of the corresponding nodes. This implementation is straightforward. There are many different parallel implementations of GraphBLAS that currently exist. This is a scalable and simple way to implement Jaccard.

1.4 Parallizing Jaccard

There are a wide variety of problems the can utilize the structure of a Jaccard coefficient. Since each Jaccard coefficient can be computed independently of the other, parallelizing the computation is fairly straightforward. There however can obviously be multiple caveats to computing the Jaccard coefficient. One simple and effective way to parallelize the computation is the use Hadoop Map-Reduce algorithms.

1.5 Real World Data Sets

In conducting experiments to observe the performance characteristics of different Jaccard algorithms there are a wide variety of data sets to choose from. The simplest choice is the RMAT graphs. These graphs are a dramatic simplification of real world problem, but are easy to demonstrate strong scaling behaviour. The RMAT graphs are artificial graphs produced for benchmarks, most notably Graph500. This makes RMAT graphs an interesting data point because one of the goals of developing Jaccard codes is to implement in as a benchmark alongside the BFS benchmark in Graph500.

Jaccard was initially developed as a community detection metric. This makes it an obvious candidate for graphs with clear and noticable communities. Previous work computes the similarity between Wikipedia pages. These data sets are available and make an obvious comparision point for any new work that is performed. There are also many other SNAP datasets that exist that have many clearly defined networks and make a useful point of comparison.

1.6 Next Steps

There are a wide variety of Jaccard algorithms that exist, many of them are quite clever. The goal of coming up with a new way of improving how parallel the computation is of great interest. The paper shows a great deal of work in computing triangle counting while keeping the working sets minimal. We think it would be interesting to adapt their algorithm towards Jaccard and tailor it towards to utilize the multi-channel DRAM (MCDRAM) found in Intel's Knight's Landing chip as well as perhaps GPU algorithms. We think that adapting this triangle counting algorithm will require careful consideration but will provide benefit when implemented correctly.

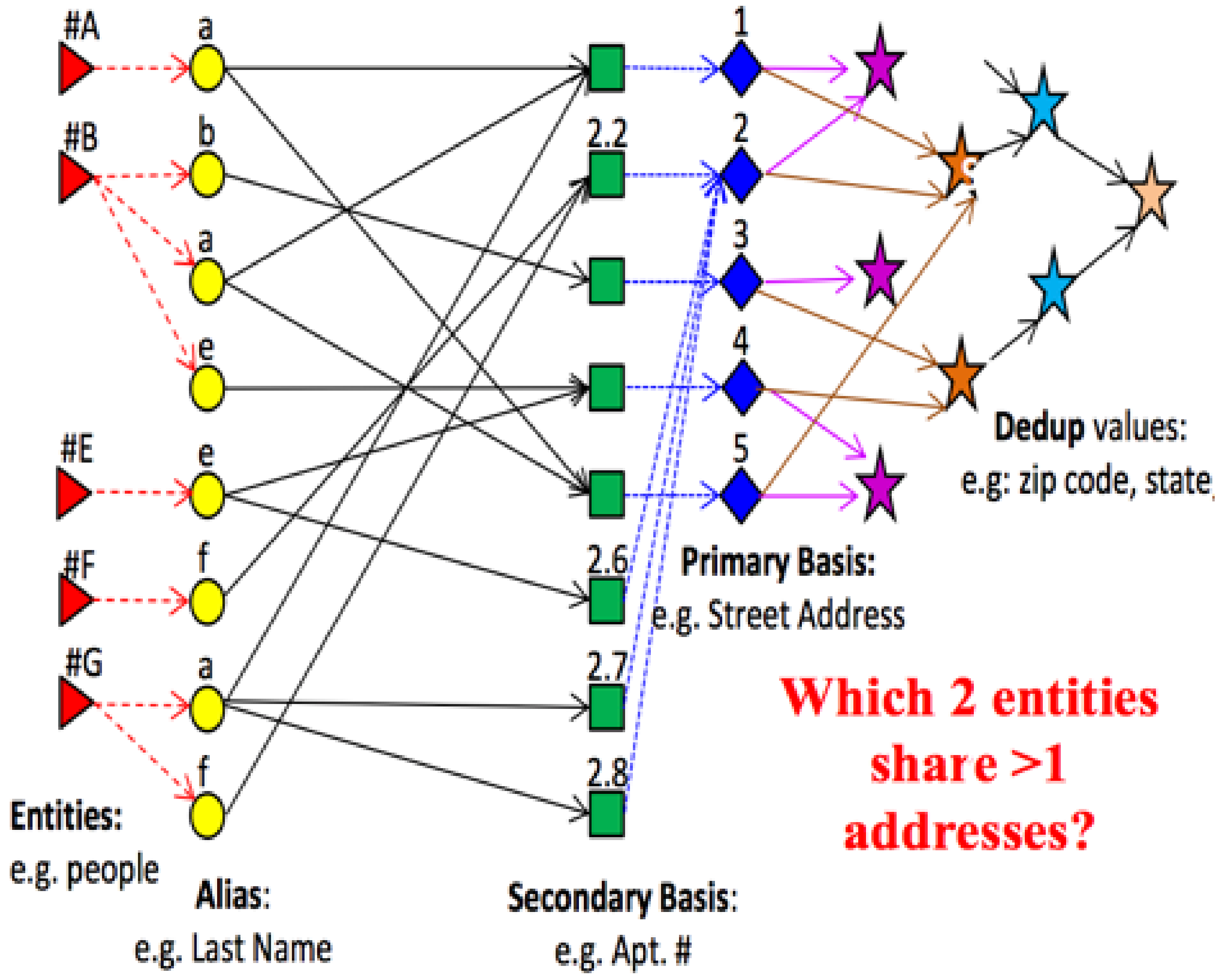

Figure 1.1: Example in which Jaccard Computations a relevant

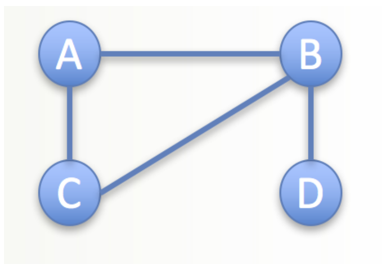

Figure 1.2: Small graph with simple Jaccard computations

Bibliography

[1] Brian Ball, Brian Karrer, and Mark EJ Newman. Efficient and principled method for detecting communities in networks. *Physical Review E*, 84(3):036103, 2011.

[2] Jacob Bank and Benjamin Cole. Calculating the jaccard similarity coefficient with map reduce for entity pairs in wikipedia. *Wikipedia Similarity Team*, pages 1–18, 2008.

[3] Paul Jaccard. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50, 1912.

[4] Peter M Kogge. Jaccard coefficients as a potential graph benchmark. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 921–928. IEEE, 2016.

[5] Chayant Tantipathananandh, Tanya Berger-Wolf, and David Kempe. A framework for community identification in dynamic social networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 717–726, New York, NY, USA, 2007. ACM.