Overview of the project

This project solves Insight's challenges of finding out whether two users are within distance *k* in a graph constructed from their transactions, for various values of *k* (namely, 1, 2 and 4). This code is general and can efficiently scale to higher values of k (more on this in the Implementation Details section).

Implementation details

The problem requires to choose a data structure able to deal with a large list of connections frequently updated and to provide an API that allows queries based on the distance of two elements. The data can be represented well conceptually by an undirected graph, on which one can run breadth-first search to get the answers we are looking for.

The graph was implemented as a dictionary of sets. Each key in the dictionary is a _user*id* and the set associated with it contains all of its neighbors (i.e. the nodes to which it is directly connected). This allows to store the graph as a sparse representation and save a ton of memory, as opposed to a matrix representation.

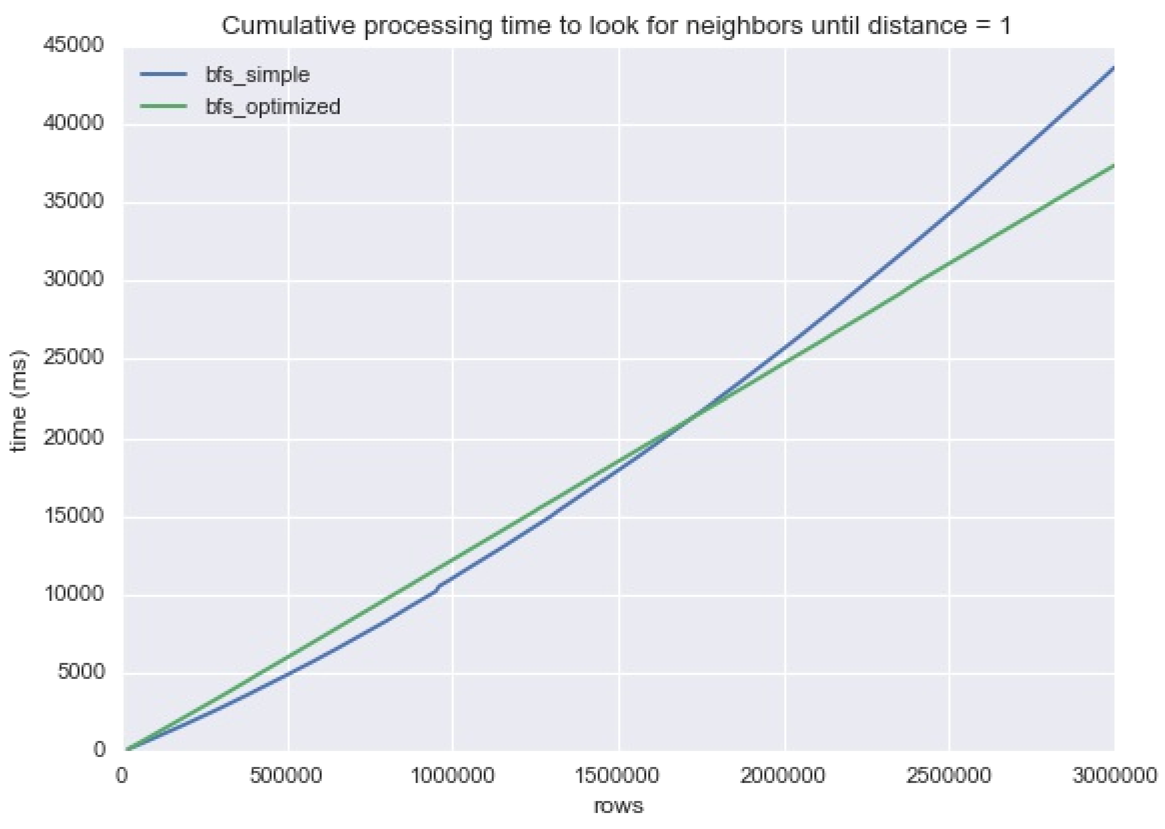
BFS is an exploration algorithm that starts from a source node, guarantees to touch each node only once and to get to a target in the shortest path possible (if edges are not weighted). It works by looking at a node at a time while keeping a FIFO queue of its neighbors that have not yet been examined. The FIFO queue guarantees that BFS examines all nodes at distance *i* from the source node before moving to those at distance *i+1*. This guarantees that BFS always finds the shortest path between two nodes. By simply adding a control on the distance from the source this algorithm can be easily modified to stop when a target node is reached within a specific range. BFS is not an expensive algorithm, running on O(|V| + |E|) (i.e. linear in the number of nodes and edges in the graph). However, it may need a lot of memory if the maximum distance *d* is large. For example, let's consider a graph in which each node has 4 neighbors and in which we want to check if node *v* and *w* (whose distance is 5) are connected: BFS would start from s, then check its 4 neighbors, then check *their* neighbors (so 4 * 4), then *these nodes'* neighbors and so on, for a total of 4^5 nodes examined.

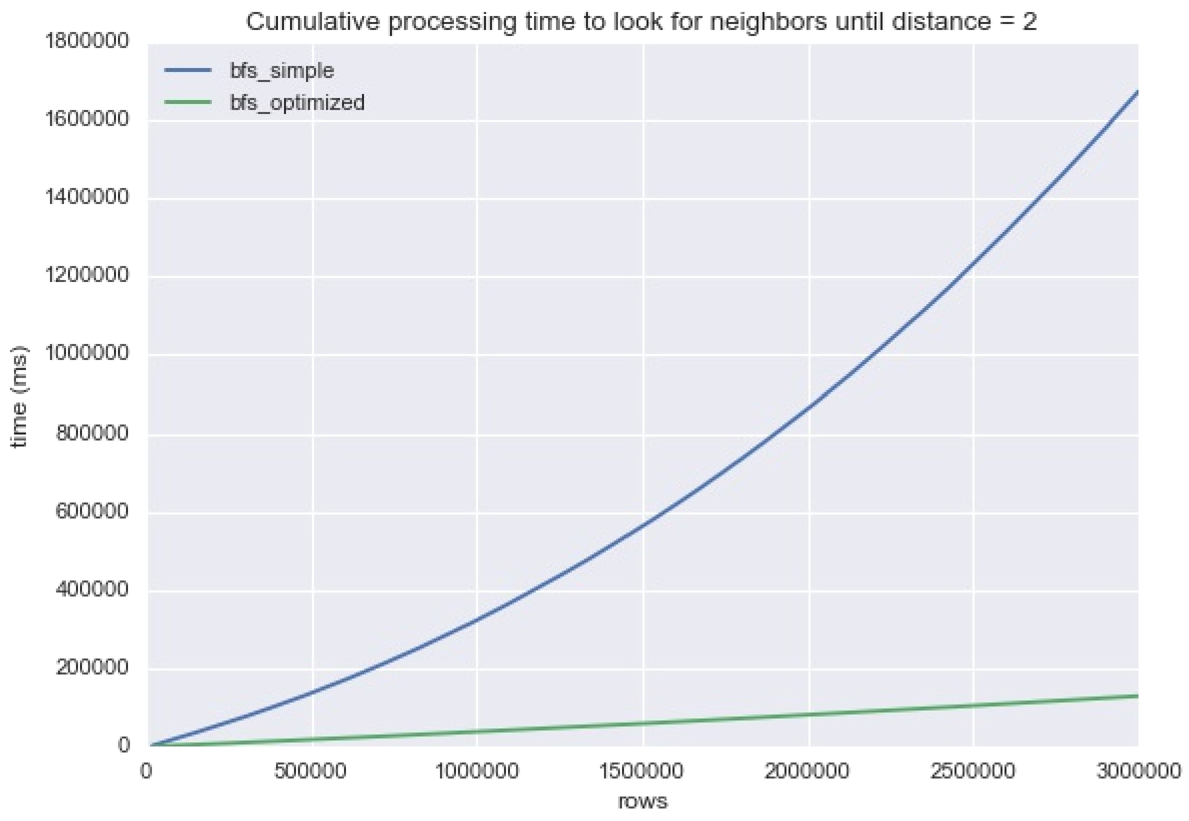
One nice improvement that boosts BFS' performance is to implement bi-directional BFS. The logic is the same as BFS but instead of starting from *v* and moving towards *w*, it moves from each of the nodes towards their middle point. If the two streams cross within *ceil(limit/2)* iterations then they must be connected. It is important to notice that *ceil(limit/2)* produces the same result both for *limit=2n/2* and *limit=(2n/2)-1*: in order to take this fact into account it is necessary to control that the original limit is greater than minimum sum of the distances relative to start and target on a node found from the intersection of the two streams. In the previous example, using bi-directional BFS would mean examining 2 * 4^3 nodes, which is a huge performance boost (this visits 128 nodes, while normal BFS would need to visit 1024).

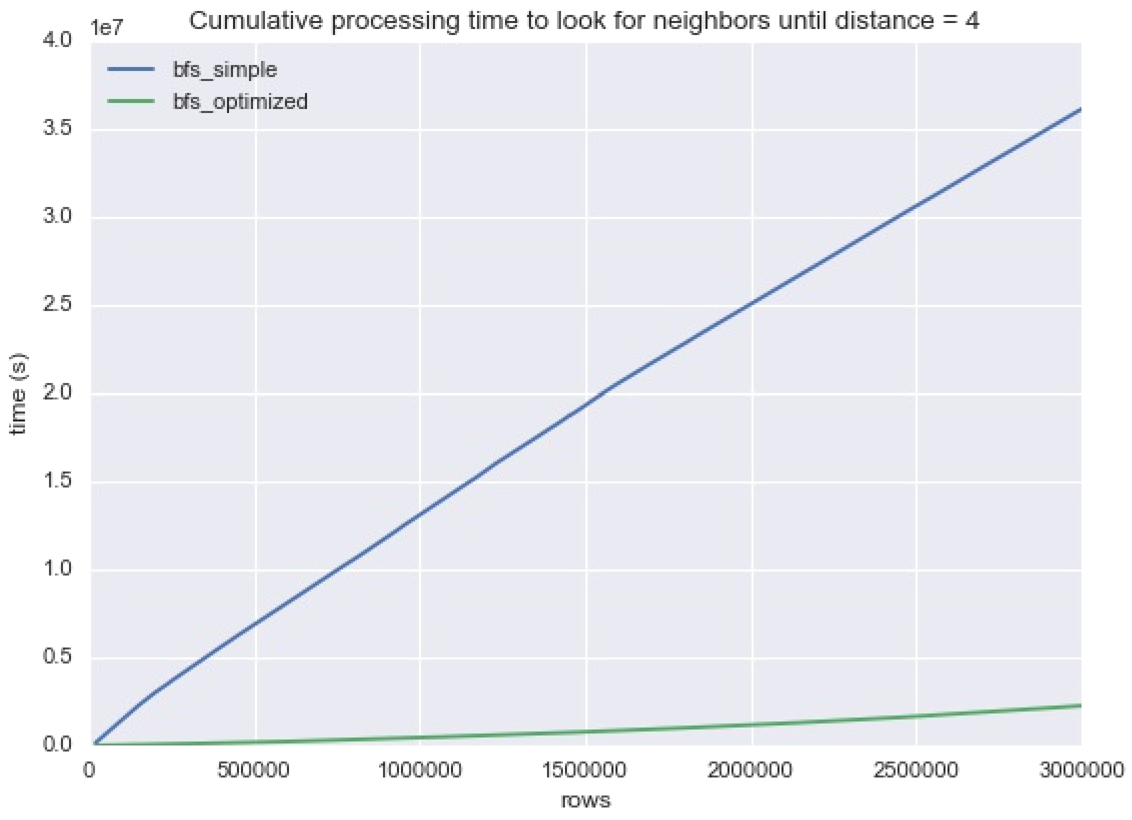
This is not the only optimization that can be done. In order to further speed up the algorithm I considered two edge cases: when *v* and *w* are at distance 1 and when they are not connected at all. For the first case the algorithm checks if *w* belongs to the set under the key of *v* in the dictionary. Given that the adjacency list for each node is stored in a set, this check is done in O(1). For the second case, I used a Union-Find data structure (which I have not implemented: the file contains the relevant attribution). Every time an edge is added between two vertices the connected groups are also Unioned. When a query asks if *v* and *w* are connected within a range, we first check if they are at all connected in the Union-Find before investing time in the BFS. If they are not connected, we can simply mark them as Unverified. This works faster than BFS because of the extreme efficiency of the Union-Find data structure, which can retrieve connectivity information in O(log(log(N))) [for all practical purposes, this is not unlike O(1)].

Performance

The average time of execution of one query is 0.1ms, 0.4ms and 7ms when distance is 1, 2 and 4 respectively. The overall execution time of "streaming_input.csv" (300k rows) for the same distances is roughly 1m, 3m and 45m. The following graphs show the cumulative time needed to process the whole file for both the simple BFS and the optimized one for all 3 features.







Extra features

Thanks to the Union-Find data structure, we can efficiently check whether two users are at all connected (regardless of the distance) very, very fast. This feature may be a good compromise between security and performance in case the service needed to scale to a very large amount of users (quasi-constant vs linear). Usually, decisions like these are taken by a classifier, that can learn what factors are relevant in these decisions directly from the data. Given that the infrastructure would allow a limited budget in ms per user query, it is often better to allocate more of this budget to a classifier, thus the savings we'd get by relaxing the _max*distance* constraint to a simple general connectivity problem would allow for a more complex classifier and, very likely, more classification accuracy.