Algorithms Over Large Data in LB-Datalog

Yannis Smaragdakis
yannis.smaragdakis@logicblox.com

Abstract

We discuss common techniques for writing highly efficient algorithms in LB-Datalog. Although one can write a variety of algorithms in the language, the real challenge is dealing with huge but sparse datasets. In that case, an efficient LB-Datalog program has to avoid taking some variant of the transitive closure of the data relations, or the sparse data will become dense data and too enormous to handle. We show efficient graph algorithms ideas and present their practical applicability. In general, we add some very powerful techniques to the arsenal of the sophisticated LB-Datalog programmer. All algorithms discussed are in a specific subclass of parallel algorithms on sparse graphs.

1. Introduction: Should You Read This Document?

The "hello world" program for Datalog is the reachable/transitive closure computation:

```
TransitiveEdge(?n1,?n2) <- Edge(?n1,?n2).
TransitiveEdge(?n1,?n2) <-
  Edge(?n1,?n3), TransitiveEdge(?n3,?n2).
```

Figure 1. Bread and butter of Datalog computations.

This simple kind of recursion forms the basis for many useful computations in LB-Datalog. Even if you are not computing transitive closure directly, you are likely to be computing some combinations of subsets of the transitive closure of a relation. Notice that the whole point is that the starting relation is sparse: the number of `Edge` facts is comparable to the number of distinct entities appearing in the `Edge` relation (e.g., nodes). The transitive closure, however, is a dense relation: its size is comparable to the *square* of the number of nodes. *The only way to write scalable algorithms over large datasets is to exploit the sparseness of the dataset.* These concepts can all be viewed in terms of sparse graphs, but don't get scared by the "graph" terminology. Every binary (two-argument) relation can be viewed as a graph, so when we are taking about graphs here, you can be thinking of plain old relations.

I am running the examples of this paper on a sample graph of about 19000 nodes (4300 nodes in the main connected body) and 14000 edges. The transitive closure of the edge relation on that graph (i.e., the `TransitiveEdge` relation, above) contains more than 7 million facts! These are the kinds of numbers you should have in mind when we talk about sparse and dense. The same graph, represented as direct edges (i.e., the `Edge` relation) has 14000 facts, represented as transitive edges (`TransitiveEdge` relation) it has 7 million facts. It stands to reason that we want to do as much of our computation as possible in the sparse representation. In fact, the data may be so big that you cannot even explicitly compute the transitive closure. If the `Edge` relation had data in the millions or billions, good luck squaring that and computing the transitive closure with a 2-rule Datalog program.

This is exactly what this paper is about. There are two equally important topics. First, we want to see how to efficiently compute dense relations derived from a sparse input. Second, we want to see techniques that let us avoid computing dense relations over our sparse input data. This bypassing of the dense computation is particularly important when the end result we want to produce is sparse itself. E.g., if we have n input facts, and the output is also of size $\Theta(n)$, we want to find algorithms that avoid computing something of size $\Theta(n^2)$ in intermediate steps.

As our first example, let's say I rewrite our `TransitiveEdge` computation as follows:

```
TransitiveEdge(?n1,?n2) <- Edge(?n1,?n2).
TransitiveEdge(?n1,?n2) <-
  TransitiveEdge(?n1,?n3), TransitiveEdge(?n3,?n2).
```

Figure 2. TransitiveEdge computation, with aggressive recursion. *Never do this for sparse data.*

The idea is that it may be faster to join large relations, rather than doing many steps of recursion before reaching a fixpoint. Indeed, we are reaching a fixpoint in fewer recursive steps: just 17 instead of 37 rule evaluations on my sample graph. However, if you really look at the time taken for these rule evaluations you will find out that the performance is catastrophic. On my Xeon X5650, 2.66GHz machine running LB 3.7.3, the program of Figure 1 takes less than 19 sec to compute the 7 million edges (using 37 joins, as mentioned). The program of Figure 2 takes 35 minutes (to do just 17 joins). That's more than 100 times slower!

You may think you know the reason for this slowdown: there is no good way to optimize the second rule. Under the current indexing scheme for LB-Datalog relations (last variable of a relation is the major index for the B-tree storing the relation), the engine needs to iterate over the entire contents of `TransitiveEdge(?n3,?n2)` every time it performs a join with a small delta of new facts, `+TransitiveEdge(?n1,?n3)`. Knowing `?n3` does not help with indexing inside `TransitiveEdge(?n3,?n2)`: the major index of the latter is `?n2`.

If you think this indexing issue has anything to do with the slowdown, you are wrong. The problem is not indexing, because convergence of these rules takes extremely few steps. Thus, the delta relation `+TransitiveEdge(?n1,?n3)` is not that much smaller than the full relation `TransitiveEdge(?n3,?n2)`. Therefore, the engine will always choose a join order where `TransitiveEdge(?n3,?n2)` is exhaustively traversed and the value of `?n3` is used to efficiently index into `+TransitiveEdge(?n1,?n3)`. So, indexing is just not the issue. Instead the issue is the central one in this paper: we are trying to compute the join of two huge relations. (In fact, it's the same relation joined with itself, but this makes no difference.) So, even if the indexing is perfectly efficient, trying 7-million-cases-times-another-7-million-cases is insanely inefficient. Intuitively, what's happening when we join a transitive closure with itself (instead of joining it with its base relation) is that we compute many different combinations that all collapse down to the same output fact. This is bad, and should be avoided.

To confirm that indexing is a red herring, let's incur some cost for computing and using a second relation just to ensure that our indexing is optimal:

```
InverseTransitiveEdge(?n1,?n2) <- TransitiveEdge(?n2,?n1).

TransitiveEdge(?n1,?n2) <- Edge(?n1,?n2).
TransitiveEdge(?n1,?n2) <-
  TransitiveEdge(?n1,?n3), InverseTransitiveEdge(?n2,?n3).
```

Figure 3. TransitiveEdge computation, with aggressive recursion and optimal indexing.

The result? We incur a *further slowdown*, from 35min to 48min (and nearly double our memory consumption). This shows quite clearly that optimizing the indexing is not the solution. In fact, indexing is not the (major) solution in any of the problems discussed in this document. If we manage to remain in the sparse domain (i.e., never compute a full transitive closure or other dense combination of sparse facts) we can get good performance even with not-so-great indexing. Since indexing is also well-covered elsewhere (e.g., in my "LogicBlox Architecture Principles" document) and since the indexing optimization that the engine can perform automatically may change in future versions, I mostly ignore indexing in this document. My code snippets may occasionally show bad indexing, although I have profiled all of them and should have noticed major inefficiencies.

The performance contrast of Figure 1 and Figure 2 leads to our first big lesson:

LESSON 1 (Sparse is the way). *Never use a dense relation for a computation that can be performed with a sparse one.*

If all you want to do is compute one large dense relation which is output or used in its entirety and if the dense relation is small enough to fit in memory, then you don't need to read further in this document. We already saw how to compute a pretty big `TransitiveEdge` in under 19 secs, after all. The problems begin when the dense relation is just a convenience step in the middle of a long computation, or when it just cannot fit in memory but you only need to check whether that relation holds for a small subset of its domain. The next few techniques offer ideas for such cases.

2. Strongly Connected Components

An important thing to remember throughout this document is that "your mileage may vary". We will be discussing techniques that may or may not benefit performance, depending on the properties of your data. Since we are presenting everything in terms of graphs, the shape of your graph determines what techniques will work well for it. There is no substitute for profiling your code and checking which rules take a long time to execute.

Having said this, the most obvious decomposition technique for a directed graph is what's called a "strongly connected components" (SCC) decomposition. Strongly connected components are subgraphs of the graph such that every node can reach every other node. The SCC decomposition splits the original graph into maximal strongly connected components. The result is a DAG (directed acyclic graph) of SCCs: there can be no cycle or the SCCs participating in the cycle would be one maximal SCC. Typically when people say "SCC" they mean a maximal SCC, although we will relax this restriction a bit here.

If we have computed the `TransitiveEdge` relation on a graph, then we can easily compute whether two nodes are in the same SCC:

```
InSameSCC(?n1,?n2) <-
  TransitiveEdge(?n1,?n2), TransitiveEdge(?n2,?n1).
InSameSCC(?n,?n) <- Node(?n).
```

Figure 4. Are two nodes in the same SCC? Computation based on transitive closure of entire edge relation.

(Note that we make the `InSameSCC` relation be reflexive (i.e., `InSameSCC(?n,?n)` for every `?n`). This is for convenience in our definitions. Generally all relations we will compute from this point on will assume that the reflexive case is uninteresting: Either it is always the case that `Relation(?n,?n)` or this never holds. This is a standard convention. It should be fairly easy to adapt the code to handle the reflexive case correctly for the relation you have in mind, it special requirements arise.)

For our 14000-edge graph, the size of `InSameSCC` is 2.8 million facts. This is quite high but still much smaller than the 7+ million pairs in the transitive closure of the edge relation. This gives rise to the primary use of SCCs: instead of doing computations on the full graph, do them on the reduced graph resulting from the SCC decomposition. This graph has no cycles, and everything we infer (reachability-wise) about it can be translated to the original graph: all the nodes inside an SCC can reach all others, so if we also know which SCCs can reach which other SCCs, we know everything about reachability.

In our case, we can replace all uses of the `TransitiveEdge` relation with uses of `InSameSCC`, as well as a second relation `TransitiveEdgeBetweenSCCs`. The former is the transitive closure of intra-SCC edges, the latter the transitive closure of inter-SCC edges. We can compute it simply by first electing a "representative" node inside each SCC and then pretending that all inter-SCC edges are edges between SCC representatives. We will pick the node with the minimum index (i.e., id inside the engine) as the representative of an SCC:[1]

```
SCCRepresentativeOf[?n1] =
  m <- agg<<m = min(?n2)>>(InSameSCC(?n1,?n2)).

EdgeBetweenSCCs(?from,?to) <-
  SCCRepresentativeOf[?n1] = ?from,
  SCCRepresentativeOf[?n2] = ?to,
  ?from != ?to,
  Edge(?n1,?n2).

TransitiveEdgeBetweenSCCs(?from,?to) <-
  EdgeBetweenSCCs(?from,?to).

TransitiveEdgeBetweenSCCs(?from,?to) <-
  EdgeBetweenSCCs(?from,?interm),
  ?from != ?to,
  TransitiveEdgeBetweenSCCs(?interm,?to).
```

Figure 5. Transitive closure of inter-SCC edges.

(Note that for this relation we have eliminated the reflexive case, i.e., we consider any transitive edge between an SCC and itself to be an intra-SCC edge and not an inter-SCC edge. This is again a fairly standard convention.)

With these two relations defined, we no longer need to use `TransitiveEdge`. Every use of `TransitiveEdge(?n1,?n2)`[2] can be equivalently replaced by the union/disjunction of two queries:

```
SCCRepresentativeOf[?n1] = SCCRepresentativeOf[?n2] ; //or
TransitiveEdgeBetweenSCCs(SCCRepresentativeOf[?n1],
                          SCCRepresentativeOf[?n2])
```

[1] We could also have used a skolem relation to create a new entity for SCCs.

[2] Again, with the exception of reflexive queries (`TransitiveEdge(?n,?n)`), which our code ignores.

(Forward pointer: You may wonder why I didn't just replace the first of the above queries by just `InSameSCC(?n1,?n2)`. For the code we have seen so far this would be equivalent, but using representative nodes is more robust: We will later keep partial information about SCCs, so our `InSameSCC` relation may not be transitively closed. In that case, using representatives works well because even partially connected parts of the same real SCC are likely to pick the same representative, thus making evident that they are in the same SCC without computing an intra-SCC transitive closure.)

So what do we gain by computing reachability based on SCCs? For my 14000-edge graph, I get 575K inter-SCC edges and 2.8M intra-SCC edges, as opposed to over 7M of transitive edges. It takes 26 sec to compute the SCC relations (19 of which were just computing the transitive closure of the `Edge` relation). And this is just a small-ish graph. I will call it my "small" graph from now on. I'm also using a "large" graph, with 52K nodes, 277K edges. For that one, the transitive closure of the `Edge` relation contains 258M edges, while the numbers of intra-SCC and inter-SCC edges are 20M and 61M, respectively. The large graph barely fits in memory in dense form.

As you can see, using SCCs results in a much smaller representation. Any further rules that need to join with the `TransitiveEdge` relation can be themselves expressed in terms of SCC concepts, resulting in smaller costs. The gains depend on the size and the shape of the graph, but typically SCC decomposition shrinks the graph down quite a bit.

LESSON 2 (SCCs are your friend). *Decompose your graph into strongly connected components and work on this representation when transitive reachability properties are needed.*

"But hold on!", I hear you say. We used `TransitiveEdge` to compute SCCs in the first place. So, we suffered the cost of a full transitive closure. Wasn't the whole point to avoid dense computations altogether? As it turns out, we can do very well by approximating SCCs instead of fully computing them, and we can do this approximation without computing the transitive closure of the edge relation at all. Even more interestingly, the approximation techniques that we will use are of general applicability. They are equally valid in a variety of settings where computing a full result is not possible or desirable.

3. Approximate SCCs Using Finite Neighborhoods

The important observation about SCCs is that we don't need to compute them fully. We can get perfectly good results, as far as reachability/transitive closure computations are concerned, with under-approximate SCCs. More precisely, all we need to do is:

- Compute for each node a representative node, such that the node and its representative are strongly connected (i.e., there is a path from either one to the other) in the original graph.

- Reduce the graph into a graph of components, each of which corresponds to a representative node. All nodes with the same representative are collapsed into the same component.
- Compute the reachability/transitive closure relation on the reduced graph. Use the same logic as for full SCCs to answer reachability queries: node a can reach node b iff they have the same representatives or the representative of a can reach (transitively) the representative of b.

The above requirements are pretty weak: we can pick a node's representative pretty much at will, as long as it is strongly connected with the node. It is not required that two nodes in the same SCC of the original graph have the same representative. It is even possible that a representative node has a different node as its own representative! (The representative may itself belong in a different partial SCC.) In fact, the representatives are a red herring: we just use them as "id"s, not as "representatives of subgraphs". All we are doing is partitioning the original graph into disjoint components that are certain to be strongly connected internally. That is, we are underapproximating the maximal SCCs: we are computing partitions that all belong in the same maximal SCC.

A simple way to underapproximate SCCs is to concentrate on finding cycles in the graph within a finite neighborhood. Breaking up a graph (or a linear sequence, or a tree, or any data structure) into finite neighborhoods and links among them is a general technique that can be adapted to virtually all circumstances of dealing with sparse data. In our case we can start with computing a transitive closure of length up to 8, and the corresponding cycles of length up to 16:

```
TransitiveEdgeUpTo2(?n1,?n2) <- Edge(?n1,?n2).
TransitiveEdgeUpTo2(?n1,?n2) <-
  Edge(?n1,?n3), Edge(?n3,?n2).
TransitiveEdgeUpTo4(?n1,?n2) <-
  TransitiveEdgeUpTo2(?n1,?n2).
TransitiveEdgeUpTo4(?n1,?n2) <-
  TransitiveEdgeUpTo2(?n1,?n3),
  TransitiveEdgeUpTo2(?n3,?n2).
TransitiveEdgeUpTo8(?n1,?n2) <-
  TransitiveEdgeUpTo4(?n1,?n2).
TransitiveEdgeUpTo8(?n1,?n2) <-
  TransitiveEdgeUpTo4(?n1,?n3),
  TransitiveEdgeUpTo4(?n3,?n2).
InSameSCC(?n,?n) <- Node(?n).
InSameSCC(?n1,?n2) <-
  TransitiveEdgeUpTo8(?n1,?n2),
  TransitiveEdgeUpTo8(?n2,?n1).
```

Figure 6. SCC underapproximation with cycles up to 16 edges long.

This works pretty well. ... Or does it now? Didn't I just violate lesson 1? I'm joining together two "denser" relations, when I could do the same with sparser ones. Indeed, we can do better:

```
TransitiveEdge2(?n1,?n2) <-
  Edge(?n1,?n3), Edge(?n3,?n2).
TransitiveEdge3(?n1,?n2) <-
  Edge(?n1,?n3), TransitiveEdge2(?n3,?n2).
TransitiveEdge4(?n1,?n2) <-
  Edge(?n1,?n3), TransitiveEdge3(?n3,?n2).
TransitiveEdge5(?n1,?n2) <-
  Edge(?n1,?n3), TransitiveEdge4(?n3,?n2).
TransitiveEdge6(?n1,?n2) <-
  Edge(?n1,?n3), TransitiveEdge5(?n3,?n2).
TransitiveEdge7(?n1,?n2) <-
  Edge(?n1,?n3), TransitiveEdge6(?n3,?n2).
TransitiveEdge8(?n1,?n2) <-
  Edge(?n1,?n3), TransitiveEdge7(?n3,?n2).
TransitiveEdgeUpTo8(?n1,?n2) <-
   Edge(?n1,?n2) ; TransitiveEdge2(?n1,?n2) ;
   TransitiveEdge3(?n1,?n2) ; TransitiveEdge4(?n1,?n2) ;
   TransitiveEdge5(?n1,?n2) ; TransitiveEdge6(?n1,?n2) ;
   TransitiveEdge7(?n1,?n2) ; TransitiveEdge8(?n1,?n2).

InSameSCC(?n,?n) <- Node(?n).
InSameSCC(?n1,?n2) <-
  TransitiveEdgeUpTo8(?n1,?n2),
  TransitiveEdgeUpTo8(?n2,?n1).
```

Figure 7. Faster SCC underapproximation with cycles up to 16 edges long.

The rest of the code stays the same as in Figure 5. Our underapproximate SCCs are a straightforward replacement for full SCCs.

On my "small" graph (the one with 7+ million transitive edges) the underapproximate SCC decomposition yields some 328K intra-SCC edges, 1.4M inter-SCC edges and 2M edges for the transitive closure up-to-8. All these relations together are computed in 11.7 sec. (Recall that it took 19sec to compute the most efficient edge transitive closure we could define on this graph.) Roughly speaking, we saved half the memory space and almost 40% of the time, with very little effort. We can now use our two usual queries wherever we would normally need to ask if one node can transitively reach another:

```
SCCRepresentativeOf[?n1] = SCCRepresentativeOf[?n2] ; //or
TransitiveEdgeBetweenSCCs(SCCRepresentativeOf[?n1],
                          SCCRepresentativeOf[?n2])
```

But didn't we already use `TransitiveEdge` for something? Let me see ... what was that? It was computing the SCCs themselves, wasn't it? So, if underapproximate SCCs can help us quickly compute the edge transitive closure implicitly, and if the edge transitive closure can get us fully precise SCCs, then why can't we use underapproximate SCCs to compute the fully precise ones? "Eat our own dogfood", so to speak. Figure 8 will certainly please dogfood lovers.

The whole computation (including the 11.7 sec for underapproximate SCCs) takes just 14.7 sec for my "small" graph. This experiment leads us to our next lesson.

```
InSameRealSCC(?n1,?n2) <-
  SCCRepresentativeOf[?n1] = SCCRepresentativeOf[?n2].
InSameRealSCC(?n1,?n2) <-
  SCCRepresentativeOf[?n1] = ?rep1,
  SCCRepresentativeOf[?n2] = ?rep2,
  TransitiveEdgeBetweenSCCs(?rep1,?rep2),
  TransitiveEdgeBetweenSCCs(?rep2,?rep1).
```

Figure 8. Using underapproximate SCCs to compute real SCCs.

LESSON 3 (SCCs are easy to underapproximate). *It is almost as useful to compute underapproximate SCCs as full SCCs. An easy way is to just limit the search to a finite neighborhood.*

This approach was trivial to implement but will not always work well. There is no substitute for knowing your graph and its shape (e.g., whether there are many tight cycles, many isolated components, etc.). If we apply the above code to my "large" graph, the result will take longer to compute than doing the full transitive closure of the edge relation. (Although it will still take less than doing the transitive closure *and* computing full SCCs based on it.) The reason is that my "large" graph is almost *designed* to defeat finite-neighborhood heuristics. My "small" graph is the callgraph of the Java JRE 1.4 libraries under a given points-to analysis (1-object-sensitive, if you must know). My "large" graph is the *context-sensitive* version of the "small" graph. This means that any cycle that exists in the "large" graph has to exist in the "small" one. This limits the size of the SCC components in the "large" graph, and also means that it contains many distinct, unconnected copies of the SCC components of the "small" graph. An SCC-decomposition into components of a small, finite size will not help nearly as much in the "large" graph but it will take much longer to compute. What we need to do is use more powerful and deep SCC-underapproximation techniques. These also introduce some general and quite neat graph decomposition concepts that work well in Datalog. But before this, take our next lesson to heart:

LESSON 4 (Know your graph). *The applicability of heuristic decomposition techniques depends on the shape of the graph. Good understanding of the shape of your sparsely-represented graph can translate into large performance and space gains.*

4. Approximate SCCs Using Spanning Structures

How can we discover SCC underapproximations in a large graph? All we need is heuristics for finding cycles, preferably long cycles. Then we can define underapproximate SCCs using all nodes in the cycle, merge with other cycles for even larger SCCs, etc.

The easiest idea for traversing a graph in a Datalog-friendly way is to just have every node pick a successor node. This defines a *spanning walk* of the graph: a set of linear paths that have common suffixes and may end in a partial loop (i.e., the path has an initial segment and then a loop), such that each node in the original graph belongs in some path (i.e., the walk "spans" the graph). That is, if we were to reduce the final loop of each path down to a single node, the paths would form a tree. This construction is trivial to write as a parallel algorithm, and therefore quite straightforward to implement in Datalog. Consider, for instance, two such spanning walks:

```
MinValueWalk[?n] = m <-
  agg<<m = min(?n2)>>(Edge(?n,?n2)).
MinValueBackWalk[?n] = m <-
  agg<<m = min(?n2)>>(Edge(?n2,?n)).
```

Figure 9. Two simple spanning walks.

These walks are essentially random: each node picks a successor or a predecessor (for the first and second walk, respectively) with the lowest internal identifier. If these internal identifiers are truly chosen at random (i.e., have no correlation with the shape of the graph) then we indeed have two truly random spanning walks.

It is also quite easy to produce a spanning forest, i.e., to make sure that every component is a tree, or, equivalently, that the loop is always a self-loop at its very end. All we need to do is to also include the node itself as a potential "next" node in a walk and to have some criterion for picking "next" that is well-founded (always shrinks).[3] For instance, we could have:

```
SelfOrNeighbor(?n,?n) <- Node(?n).
SelfOrNeighbor(?n,?nNeigh) <- Edge(?n,?nNeigh).
UpSpanningTreeNeighbor[?n1] = m <-
  agg<<m = min(?n2)>>(SelfOrNeighbor(?n1,?n2)).
```

Figure 10. Simple spanning forest.

In this way, the numerical "id"s of nodes strictly decrease along each path (since the current id is included in the computation of the minimum). The paths become shorter, always ending in a self-loop.

Of course, we can influence our walk based on the shape of the graph. For instance, Figure 11 shows a walk where each node picks an outgoing edge to a node that has the highest number of outgoing edges.

And certainly there is no reason to limit our attention to walks. For instance, relation ChildrenWithMaxChildren in Figure 11 defines a subgraph of the original graph, where each node can reach its maximum-outgoing-edges neighbors.

[3] Of course, we could also generate a spanning forest where every node is a separate tree, but that is uninteresting. Our goal is to create helpful spanning structures, not just any spanning structure.

```
// How many nodes does ?n point to?
NumberOfChildren[?n] = i <-
  agg<<i = count()>>(Edge(?n,_)).
// How many nodes do nodes that ?n points to point to?
ChildrenForSomeChild(?n,?i) <-
  Edge(?n,?nChi), NumberOfChildren[?nChi] = ?i.
// What's the maximum number of children of ?n's children?
MaxChildrenForSomeChild[?n] = m <-
  agg<<m = max(?iChi)>>(ChildrenForSomeChild(?n,?iChi)).
// Which nodes (may be more than one) that ?n points to
// have a maximum number of children?
ChildrenWithMaxChildren(?n,?nChi) <-
  Edge(?n,?nChi),
  NumberOfChildren[?nChi] = MaxChildrenForSomeChild[?n].
// Pick an arbitrary (min node value) child node with max
// number of children.
SomeChildWithMaxChildren[?n] = m <-
  agg<<m = min(?nChi)>>(ChildrenWithMaxChildren(?n,?nChi)).
```

Figure 11. A walk where each node picks its neighbor with the highest number of further neighbors.

Defining such subgraphs of the original graph lets us quickly explore cycles, by computing the transitive closure of an edge relation that is much smaller than the original. It is particularly good to combine walks. For instance, if we take the graph defined as the union of the two spanning walks of Figure 9, we end up with at least one out-edge and one in-edge for every node (as long as the node had in-edges and out-edges in the original graph, of course). This ensures that the graph is fairly well-connected, even though it is likely much smaller than the original. If we then compute the transitive closure of the edge relation on that graph, we are likely to find cycles, which are also cycles in the original graph and mean that all participating nodes are in the same maximal SCC. The code of Figure 12 computes underapproximate SCCs using a variety of the heuristics we discussed. It first defines two subgraphs of the original graph, each consisting of the union of two spanning walks. The two subgraphs are defined by relations `Edge2` and `Edge3`. Relation `Edge2` is the union of the two walks in Figure 9. Relation `Edge3` is the union of two analogous walks, each picking the maximum-id (instead of minimum-id for `Edge2`) neighbor as the next/previous node. We then compute the transitive closure of the `Edge2` and `Edge3` relations. Relations `InSameSCC2` and `InSameSCC3` are defined by simple cycle detection on this transitive closure (also using the original `Edge` relation). Note the use of two separate rules, which ensure that not just the endpoints but also all internal nodes of the cycle are in the SCC. Finally, we compute the symmetric closure of these relations, put them together, and compute the transitive closure of their union, which gives us several large underapproximate SCCs in our usual relation `InSameSCC`. The rest of the code again remains as in Figure 5

On my "large" graph, these heuristics compute underapproximate SCCs in 466sec. The SCC decomposition has fewer than 160M inter-SCC edges, and 4.5M intra-SCC edges. As before, the SCC relations can be used to perform

```
Edge2(?n1,?n2) <- MinValueWalk[?n1] = ?n2.
Edge2(?n1,?n2) <- MinValueBackWalk[?n2] = ?n1.

Edge3(?n1,?n2) <- MaxValueWalk[?n1] = ?n2.
Edge3(?n1,?n2) <- MaxValueBackWalk[?n2] = ?n1.

TransitiveEdge2(?from,?to) <- Edge2(?from,?to).
TransitiveEdge2(?from,?to) <-
  Edge2(?from,?interm), TransitiveEdge2(?interm,?to).

TransitiveEdge3(?from,?to) <- Edge3(?from,?to).
TransitiveEdge3(?from,?to) <-
  Edge3(?from,?interm), TransitiveEdge3(?interm,?to).

InSameSCC2(?n1,?n2) <-
  TransitiveEdge2(?n1,?n2), Edge(?n2,?n1).
InSameSCC2(?n1,?n2) <-
  TransitiveEdge2(?n1,?n2), Edge(?n2,?n3),
  TransitiveEdge2(?n3,?n1).

InSameSCC3(?n1,?n2) <-
  TransitiveEdge3(?n1,?n2), Edge(?n2,?n1).
InSameSCC3(?n1,?n2) <-
  TransitiveEdge3(?n1,?n2), Edge(?n2,?n3),
  TransitiveEdge3(?n3,?n1).

InSameSCC2(?n1,?n2) <- InSameSCC2(?n2,?n1).
InSameSCC3(?n1,?n2) <- InSameSCC3(?n2,?n1).

InSameSCC(?n1,?n1) <- Node(?n1).
InSameSCC(?n1,?n2) <- InSameSCC2(?n1,?n2).
InSameSCC(?n1,?n2) <- InSameSCC3(?n1,?n2).
InSameSCC(?n1,?n2) <-
  InSameSCC2(?n1,?nInterm), InSameSCC(?nInterm,?n2).
InSameSCC(?n1,?n2) <-
  InSameSCC3(?n1,?nInterm), InSameSCC(?nInterm,?n2).
```

Figure 12. Applying several heuristics in order to find underapproximate SCCs.

all transitive reachability computations. Computing full transitive reachability on the "large" graph results in over 258M edges and takes 656sec, therefore the SCC approach is again both faster and space-saving.

LESSON 5 (Random spanning subgraphs are great). *Computing "random" subgraphs of a graph is easy and can help a lot in discovering shape properties of the graph, including cycles for SCCs.*

5. Full SCCs Using a Sequential DFS Algorithm

Can probably do, but needs hacks, possibly better engine support, will be discussed in future work, after the issues are understood a little more. Wouldn't help much compared to previous section anyway, as my measurements suggest.