Jon Berry's Challenge List for Graph Algorithms in the Language of Linear Algebra

- Example 1: finding triangles. The best way I know how to do this is the gather-scatter strategy of GraphLab, in which the crucial computational step is hopscotch or cuckoo hashing to find neighborhood intersections. This strategy seems to beat "Cohen's algorithm" (which also goes by other names). The latter relies on building buckets for vertices, placing edges in buckets, and looking at all pairs of edges per bucket. Neither algorithm seems a natural fit for linear algebra.

- Example 2: connected components: The best way I know how to do this for graphs with heavy-tailed degree distributions is to BFS from the highest degree vertex, marking the giant component, then (without explicitly inducing a subgraph) to run the "bully algorithm" in which each unmarked vertex attempts to mark its (O(log n)-sized) component in the style of the game of Risk. For example, lower-id's might dominate higher ones. I posed this as a friendly challenge to John the other day. The BFS is clearly matvec-friendly, but I don't think the rest is. See Example 6 for another unexpected twist on BFS.

- Example 3: maximum independent sets (exact): Cindy Phillips and I are experimenting with a low-exponential algorithm for small instances from the literature (SODA, but can't find the reference at the moment) in which a kernel operation is the following: find an edge (u,v) such that the neighborhood of u is a proper subset of the neighborhood of v (i.e., enumerating triangles is a subroutine). Vertex v would never be as good of a choice to join a maximum independent set as u would, so we can delete v and its incident edges, but not its neighboring vertices. Repeat that process until there are no such (u,v) remaining, then go on to an even more complicated kernelization step that involves identifying vertices that have no "anti-triangle"(independent set of size 3) in their neighborhood, deleting some of their neighbors, and replacing them with composite nodes that maintain independent set-related invariants.

- Example 4: maximal independent sets: Luby's algorithm chooses random sets of vertices, traverses edges in search of those induced by the set, then deactivates the vertex of lower degree and repeats. Perhaps this could be done with Jeremy's filters(?), but it's not natural for me to think that way.

- Example 5: sssp: A kernel responsible for doubling parallel performance (i.e., halving runtime) in Kamesh Madduri's work with David and me was to partition some of the adjacency lists individually into low and high degree vertices. Note that a full sort would kill performance. I suppose that this might reduce to a user-defined row/column permutation operator that could be used to partition the whole matrix into low and high degree. However, that would move most of the data, whereas Kamesh could partition a few adjacency lists and leave most of the others alone.

- Example 6: "special betweenness" In the subgraph isomorphism algorithm I've been working on forever, we build a DAG that has a super-source at the left, a super-sink at the right, and a multi-partite structure in the middle (think: columns of nodes), with all arcs pointing left to right. Given one of these arcs 'e', we wish to compute a "special betweenness" – namely the number of s-t paths that traverse e. Since we care about only one pair (s,t), this is a relaxation of the classical betweenness. It can be computed using two BFS-like operations: search from s, propagating blue bread crumbs down the line (if a node gets k bread crumbs, it propagates k to each downstream neighbor, etc.). Then search backwards from t, propagating red bread crumbs. Postprocess edges in parallel O(1) time, multiplying blue and red counts to obtain the special betweenness.