COS 226 Bob's Notes: The WordNet Assignment
Fall 2010
R. Tarjan, Preceptor

The second part of today's notes examines the WordNet programming assignment and in particular ideas for addressing the "shortest ancestral path" (SAP) problem that is the algorithmic heart of the assignment. Caveat: the ideas to follow are just my thoughts and notes on ideas that came up in precept. Should you decide to use any of them, you will need to verify (1) you can use them given the problem constraints (required API's and methods) and (2) they result in sufficiently fast code. Your friendly preceptor makes no guarantees about the usefulness of any of the ideas that follow.

I'll make the following assumption, although it is not explicitly stated in the problem description: the set of words and the set of synsets are disjoint. That is, no word is a synset, and no synset is a word. Given this assumption, and ignoring the glosses, the data can be thought of as two directed graphs, one, S, whose vertex set is the set of synsets, with an edge (x, y) iff y is a hypernym of x, and the other, W, whose vertex set is the union of the set of words and the set of synsets, with an edge (x, y) iff word x is in synset y. For me the best way to think about the problem is to combine these into one graph G. This graph contains two kinds of edges: those that lead from a word to a synset, and those that lead from a synset to another synset. Every word has no incoming arcs. Each synset can have incoming arcs from words and/or from other synsets; it need not have both kinds. (If it has neither, it is an empty concept, and presumably a data error.)

Given two distinct words x and y, a "shortest ancestral path" is defined in the problem specification as follows. It is a synset z, a path P in S from a synset of x to z, and a path Q in S from a synset of y to z, such that the total number of edges on P and Q is minimum. In general there can be many such paths. We can restate this definition in terms of the graph G as follows: a shortest ancestral path is a synset z, a path P' in G from x to z, and a path Q' in G from y to z, such that the total number of edges on P' and Q' is minimum; the "length" of the path is the total number of edges on P' and Q' minus two. (The edges from x and from y do not count.)

Exercise: Verify that the second definition of "shortest ancestral path" is equivalent to the first.

The most straightforward way to find a shortest ancestral path is to run two breadth-first searches in G starting from x and from y, thereby computing the distance of each synset from x and from y. Then, for each synset reachable from x and from y, sum the two distances, Finally, take the minimum of these sums and subtract two. (Or, instead of subtracting two, in each distance computation avoid counting the first edge.) An improvement is to run both searches concurrently, visiting vertices at distance one from either x or y, then vertices at distance two

from x or y, and so on. Once some vertex is reached from both x and y, the sum of its distances from x and y gives an upper bound on the distance beyond which the searches need not go. The upper bound can be reduced each time a shorter pair of paths is found. This allows early termination, especially in the case of nearby words.

If W is acyclic, an additional heuristic can be used to try to further limit the searches. (If W is acyclic, so is G: why?) Do a preprocessing step to number the vertices in topological order. When choosing whether to do a step of the BFS from x or the one from y, visit the vertex smaller in topological order. This strategy can be combined with the shortest distance strategy in various ways. The assignment requires that your algorithm work on graphs with cycles, but the topological ordering algorithm can detect if there are cycles, in which case the strategy is not valid (although a generalization that topologically numbers strongly connected components might be).

As I mentioned in class, the shortest ancestral path problem on G can be converted into a standard shortest path problem on a bigger graph G2. Let 1, 2,…, N be the vertices in G. Graph G2 contains 2N vertices. It consists of one copy of G, whose vertices are 1, 2,…, N, a separate copy G' of the reversal of G, whose vertices are 1', 2',…, N', and N additional arcs (i, i') for i = 1,2,…, N. The reversed graph G' has an edge (j', i') for each edge (i, j) in G. It is critical in this construction that G and G' have different sets of vertices; otherwise the construction fails. If x and y are vertices in G, there is an ancestral path of k edges connecting x and y if and only if there is an path of k + 1 edges from x to y' in G2. A path from x to y' in G2 contains a unique edge (z, z') and corresponds to paths in G from x to z and from y to z. Thus we can compute shortest ancestral paths in G by computing shortest paths in G2. G2 is symmetric (but not undirected): if there is a path P from x to y', there is a path Q from y to x' obtained by swapping i and i' for all i and reversing each edge.

Exercise: Verify this.

Thus we can compute shortest ancestral paths by computing ordinary shortest paths on a graph of about twice the size. I do not necessarily recommend using this approach for computing individual SAP distances, however, for two reasons: (1) It requires building a graph that has twice as many vertices as G and more than twice as many edges, consuming extra time and space; and, more importantly (2) if the problem graph has larger average in-degree than average out-degree, then searching forward is likely to explore less of the graph than searching backward. This may well be true of the WordNet graph. To use the doubled graph efficiently, one would probably want to use a bidirectional search strategy, mostly searching forward in G and backward in G', but this would be similar to the concurrent search strategy on G already discussed. Only experiments can determine what approach is best.

Another idea one might try is to precompute certain distances and to use table lookup to trade extra space and preprocessing time for faster query time. This is a very general idea; you might want to think about whether and how to use it to answer multiple SAP queries.

Finally, the outcast problem on k words requires the computation of $k(k-1)/2$ SAP distances, between each pair of distinct words. One would like to avoid doing this computation separately for each pair. It suffices to do $k-1$ one-way searches on G2. (Exercise: develop such a method.) Depending on the size of k and how far apart the words are, this may be an improvement over doing separate pairwise searches on G.