|  |  |
| --- | --- |
| 99down voteaccepted | Here is a short summary about the community detection algorithms currently implemented in igraph:   * edge.betweenness.community is a hierarchical decomposition process where edges are removed in the decreasing order of their edge betweenness scores (i.e. the number of shortest paths that pass through a given edge). This is motivated by the fact that edges connecting different groups are more likely to be contained in multiple shortest paths simply because in many cases they are the only option to go from one group to another. This method yields good results but is very slow because of the computational complexity of edge betweenness calculations and because the betweenness scores have to be re-calculated after every edge removal. Your graphs with ~700 vertices and ~3500 edges are around the upper size limit of graphs that are feasible to be analyzed with this approach. Another disadvantage is that edge.betweenness.community builds a full dendrogram and does not give you any guidance about where to cut the dendrogram to obtain the final groups, so you'll have to use some other measure to decide that (e.g., the modularity score of the partitions at each level of the dendrogram). * fastgreedy.community is another hierarchical approach, but it is bottom-up instead of top-down. It tries to optimize a quality function called modularity in a greedy manner. Initially, every vertex belongs to a separate community, and communities are merged iteratively such that each merge is locally optimal (i.e. yields the largest increase in the current value of modularity). The algorithm stops when it is not possible to increase the modularity any more, so it gives you a grouping as well as a dendrogram. The method is fast and it is the method that is usually tried as a first approximation because it has no parameters to tune. However, it is known to suffer from a resolution limit, i.e. communities below a given size threshold (depending on the number of nodes and edges if I remember correctly) will always be merged with neighboring communities. * walktrap.community is an approach based on random walks. The general idea is that if you perform random walks on the graph, then the walks are more likely to stay within the same community because there are only a few edges that lead outside a given community. Walktrap runs short random walks of 3-4-5 steps (depending on one of its parameters) and uses the results of these random walks to merge separate communities in a bottom-up manner like fastgreedy.community. Again, you can use the modularity score to select where to cut the dendrogram. It is a bit slower than the fast greedy approach but also a bit more accurate (according to the original publication). * spinglass.community is an approach from statistical physics, based on the so-called Potts model. In this model, each particle (i.e. vertex) can be in one of *c* spin states, and the interactions between the particles (i.e. the edges of the graph) specify which pairs of vertices would prefer to stay in the same spin state and which ones prefer to have different spin states. The model is then simulated for a given number of steps, and the spin states of the particles in the end define the communities. The consequences are as follows: 1) There will never be more than *c* communities in the end, although you can set *c* to as high as 200, which is likely to be enough for your purposes. 2) There may be less than *c* communities in the end as some of the spin states may become empty. 3) It is not guaranteed that nodes in completely remote (or disconencted) parts of the networks have different spin states. This is more likely to be a problem for disconnected graphs only, so I would not worry about that. The method is not particularly fast and not deterministic (because of the simulation itself), but has a tunable resolution parameter that determines the cluster sizes. A variant of the spinglass method can also take into account negative links (i.e. links whose endpoints prefer to be in different communities). * leading.eigenvector.community is a top-down hierarchical approach that optimizes the modularity function again. In each step, the graph is split into two parts in a way that the separation itself yields a significant increase in the modularity. The split is determined by evaluating the leading eigenvector of the so-called modularity matrix, and there is also a stopping condition which prevents tightly connected groups to be split further. Due to the eigenvector calculations involved, it might not work on degenerate graphs where the ARPACK eigenvector solver is unstable. On non-degenerate graphs, it is likely to yield a higher modularity score than the fast greedy method, although it is a bit slower. * label.propagation.community is a simple approach in which every node is assigned one of *k*labels. The method then proceeds iteratively and re-assigns labels to nodes in a way that each node takes the most frequent label of its neighbors in a synchronous manner. The method stops when the label of each node is one of the most frequent labels in its neighborhood. It is very fast but yields different results based on the initial configuration (which is decided randomly), therefore one should run the method a large number of times (say, 1000 times for a graph) and then build a consensus labeling, which could be tedious.   igraph 0.6 will also include the state-of-the-art Infomap community detection algorithm, which is based on information theoretic principles; it tries to build a grouping which provides the shortest description length for a random walk on the graph, where the description length is measured by the expected number of bits per vertex required to encode the path of a random walk.  Anyway, I would probably go with fastgreedy.community or walktrap.community as a first approximation and then evaluate other methods when it turns out that these two are not suitable for a particular problem for some reason. |