11 Cluster Discovery in Biological Networks

11.1 Introduction

Clustering is the process of grouping similar objects based on some similarity measure. The aim of any clustering method is that the objects belonging to a cluster should be more similar to each other than to the rest of the objects under consideration. Clustering is one of the most studied topics in computer science as it has numerous applications in bioinformatics, data mining, image processing, and complex networks such as social networks, biological networks, and the Web.

We will make a distinction between clustering data points commonly distributed in 2D plane which we will call *data clustering* and clustering objects which are represented as vertices of a graph in which case we will use the term *graph clustering*. Furthermore, graph clustering can be investigated as *inter-graph clustering* where a subset from a given set of graphs are clustered based on their similarity or *intra-graph clustering* in which our object is to find clusters in a given graph. We will assume the latter when we investigate clustering in biological networks in this chapter.

Intra-graph clustering or graph clustering in short, considers the neighborhood relationship of the vertices while searching for clusters. In unweighted graphs, we try to cluster nodes that have strong neighborhood connections to each other and this problem can be viewed as finding cliques of a graph in the extreme case. Our aim in edge-weighted graphs, however, is to place neighbors that are close to each other in the same cluster using some metric.

Biological networks are naturally represented as graphs as we have seen in Chap. 10, and any graph clustering algorithm can be used to detect clusters in biological networks such as the gene regulation networks, metabolic networks, and PPI networks. There are, however, important differences between a graph representing a general random network and the graph of a biological network. First of all, the size of a biological network is huge, reaching tens of thousands of vertices and hundreds of thousands of edges, necessitating the use of highly efficient clustering algorithms as well as usage of distributed algorithms for this computation-intensive task. Secondly, biological networks are scale-free with few very high-degree nodes and many low-degree nodes. Third, they exhibit small-world property having small diameters relative to their sizes. These last two observations may be exploited to design efficient clustering algorithms with low time complexities but this alone does not provide the needed performance in many cases and using distributed algorithms is becoming increasingly more attractive to solve this problem.

Our aim in this chapter is to first provide a formal background and a classification of clustering algorithms in biological networks. We then describe and review efficient sample algorithms, most of which are experimented in biological networks and have distributed versions. In cases where there are no distributed algorithms known to date, we propose distributed algorithm templates and point potential areas of further investigation which may lead to efficient algorithms.

11.2 Analysis

We can have overlapping clusters where a node may belong to two or more clusters or a node of the graph becomes a member of exactly one cluster at the end of a clustering algorithm, which is called *graph partitioning*. Also, we may specify the number of clusters k beforehand and the algorithm stops when there are exactly k clusters, or it terminates when a certain criteria is met. Another distinction is whether a node belongs fully to a cluster or with some probability. In *fuzzy clustering*, membership of a node to a cluster is specified using a value between 0 and 1 showing this probability [47].

Formally, a clustering algorithm divides a graph $G(V, E)$ into a number of possibly overlapping clusters $\mathcal{C} = C_1, \ldots, C_k$ where a vertex $v \in C_i$ is closer to all other vertices in C_i than to vertices in other clusters. This similarity can be expressed in a number of ways and a common parameter for graph clustering is based on the average density of the graph and the densities of the clusters. We will now evaluate the quality of a graph clustering method based on these parameters.

11.2.1 Quality Metrics

A basic requirement from any graph clustering algorithm is that the vertices in a cluster output from the algorithm should be connected which means there will be at least one path between every vertex pair (u, v) in a cluster C_i. Furthermore, the path between u and v should be internal to the cluster C_i meaning u is close to v [40], assuming the diameter of the cluster is much smaller than the diameter of the graph. However, a more fundamental criteria is based on evaluating the densities of the graph in a cluster and outside the cluster and comparing them. We will describe two methods to evaluate these densities next.

11.2.1.1 Cluster Density

The quality of a clustering method is closely related to the density of vertices in a cluster which can be evaluated in terms of the density of the unweighted, undirected graph as a whole. Let us first define the density of a graph. The density $\rho(G)$ of an unweighted, undirected simple graph G is the ratio of the size of its existing edges to the size of maximum possible edges in G as follows:

$$\rho(G) = \frac{2m}{n(n-1)} \tag{11.1}$$

Let us now examine the edges incident to a vertex v in a cluster C_i. Some of these edges will be connecting v to other vertices in C_i which are called *internal* edges and the rest of the edges on v that connect it to other clusters are called *external* edges. Clearly, degree of vertex v is the sum of its internal and external edges. The size of internal edges ($\delta_{\text{int}}(v)$) and external edges ($\delta_{\text{ext}}(v)$) of a vertex v gives a good indication of the appropriateness of v being in C_i. Considering the ratio $\delta_{\text{int}}(v)/\delta_{\text{ext}}(v)$, if this is small, we can conclude v may have been wrongly placed in C_i, and on the contrary, a large ratio reveals v is properly situated in C_i. We can generalize this concept to the clustering level and define the *intra-cluster density* of a cluster C_i as the ratio of all internal edges in C_i to all possible edges in C_i as follows [40].

$$\delta_{\text{int}}(C_i) = \frac{2\sum_{v \in C_i} \delta_{\text{int}}(v)}{|C_i||C_i - 1|} \tag{11.2}$$

We can then define the intra-cluster density of the whole graph as the average of all intra-cluster densities as follows:

$$\delta_{\text{int}}(G) = \frac{1}{k}\sum_{i=1}^{k} \delta_{\text{int}}(C_i) \tag{11.3}$$

where k is the number of clusters obtained. For example, the intra-cluster densities for clusters C_1, C_2, and C_3 in Fig. 11.1a are 0.66, 0.33, and 0.5 respectively and the average intra-cluster density is 0.50. We divide the same graph into different clusters in (b) with intra-cluster densities of 0.7, 0.5, and 0.6 for these clusters and the average density becomes 0.6. We can say that the clusters obtained in (b) are better as we have a higher average intra-cluster density, as can be observed visually. The *cut size* of a cluster is the size of the edges between C_i to all other clusters it is connected. The *inter-cluster density* $\delta_{\text{ext}}(G)$ is defined as the ratio of the size of inter-cluster edges to the maximum possible size of edges between all clusters as shown below [40]. In other words, we subtract the size of maximum total possible intra-cluster edges from the size of the maximum possible edges between all nodes in the graph to find the size of the maximum possible inter-cluster edges, and the inter-cluster density should be as low as possible when compared with this parameter.

$$\delta_{\text{ext}}(G) = \frac{2 \times \text{sum of inter-cluster edges}}{n(n-1) - \sum_{i=1}^{k}(|C_i||C_i - 1|)} \tag{11.4}$$

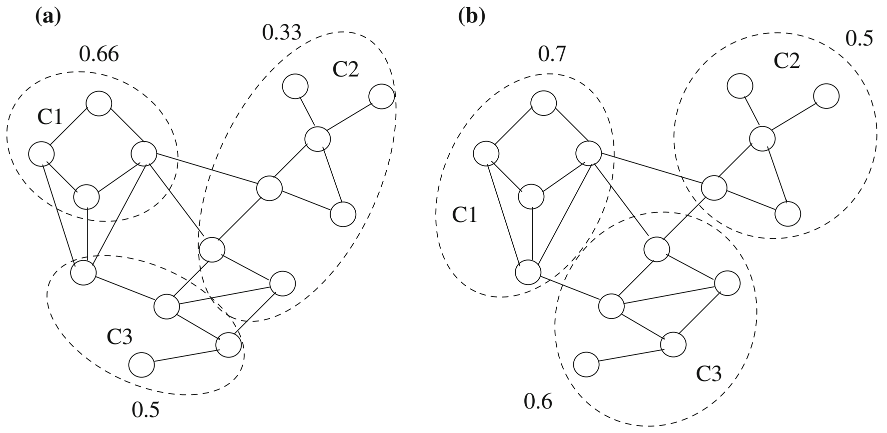

Fig. 11.1 Intra-cluster and inter-cluster densities example

The inter-cluster densities in Fig. 11.1a, b are 0.08 and 0.03 respectively, which again shows the clustering in (b) is better since we require this parameter to be as small as possible. The graph density in this example is $(2 \times 22)/(15 \times 14) = 0.21$ and based on the foregoing, we can conclude that a good clustering should provide a significantly higher intra-cluster density than the graph density, and the inter-cluster density should be significantly lower than the graph density.

When we are dealing with weighted graphs, we need to consider the total weights of edges in the cut set, as the internal and external edges, rather than the number of such edges. The density of an edge-weighted graph can be defined as the ratio of total edge weight to the maximum possible number of edges as follows:

$$\rho(G(V, E, w)) = \frac{2 \sum_{(u,v) \in E} w_{(u,v)}}{n(n-1)} \tag{11.5}$$

The intra-cluster density of a cluster C_i in such an edge-weighted graph can then be computed similarly to the unweighted graph but we sum the weights of edges inside the clusters and divide it by the maximum possible number of edges in C_i this time. The graph intra-cluster density is the average of intra-cluster densities of clusters as before and the general requirement is that this parameter should be significantly higher than the edge-weighted graph density. For inter-cluster density of an edge-weighted graph, we can compute the sum of weights of all edges between each pair of clusters and divide it by the maximum possible number of edges between clusters as in Eq. 11.4 by just replacing the number of edges with their total weight. We can then compare this value with the graph density as before and judge the quality of clustering.

11.2.1.2 Modularity

The modularity parameter proposed by Newman [35] is a more direct evaluation of the goodness of clustering than the above described procedures. Given an undirected and unweighted graph $G(V, E)$ which has a cluster set $\mathcal{C} = \{C_1, .., C_k\}$, modularity Q is defined as follows [36]:

$$Q = \sum_{i=1}^{k} (e_{ii} - a_i^2) \tag{11.6}$$

where e_{ii} is the percentages of edges in C_i, and a_i is the percentage of edges with at least one edge in C_i. We actually sum the differences of probabilities of an edge being in C_i and a random edge would exist in C_i. The maximum value of Q is 1 and a high value approaching 1 shows good clustering. For calculating Q conveniently, we can form a modularity matrix M which has an entry m_{ij} showing the percentage of edges between clusters i and j. The diagonal elements in this matrix represent the e_{ii} parameter in Eq. 11.6 and the sum of each row except the diagonal is equal to a_{ij} of the same equation. We will give a concrete example to clarify these concepts. Evaluating the modularity matrices M_1 and M_2 for Fig. 11.1a, b respectively yields:

$$M_1 = \begin{bmatrix} 0.18 & 0.09 & 0.14 \\ 0.09 & 0.32 & 0.14 \\ 0.14 & 0.14 & 0.14 \end{bmatrix} \quad M_2 = \begin{bmatrix} 0.32 & 0.05 & 0.09 \\ 0.05 & 0.23 & 0.05 \\ 0.09 & 0.05 & 0.27 \end{bmatrix}$$

For the first clustering, we can calculate the contributions to Q using M_1 from clusters C_1, C_2 and C_3 as 0.127, 0.267, and 0.060 giving a total Q value of 0.247. We can see straight away clustering structure in C_3 is worse than others as it has the lowest score. For M_2 matrix of clusters in (b), the contributions are 0.30, 0.22, and 0.25 providing a Q value of 0.77 which is significantly higher than the value obtained using M_1 and also closer to unity. Hence, we can conclude that the clustering in (b) is much more favorable than the clustering in (a). We will see in Sect. 11.4 that there is a clustering algorithm based on the modularity concept described.

11.2.2 Classification of Clustering Algorithms

There are many different ways to classify the clustering algorithms based on the method used. In our approach, we will focus on the methods used for clustering in biological networks and provide a taxonomy of clustering algorithms used for this purpose only as illustrated in Fig. 11.2. We have mostly included fundamental algorithms in each category that have distributed versions or can be distributed.

We classify the clustering algorithms in four basic categories as hierarchical, density-based, flow-based, and spectral algorithms. The hierarchical algorithms construct nested clusters at each step and they either start from each vertex being a single cluster and combine them into larger clusters at each step, or they may start from one

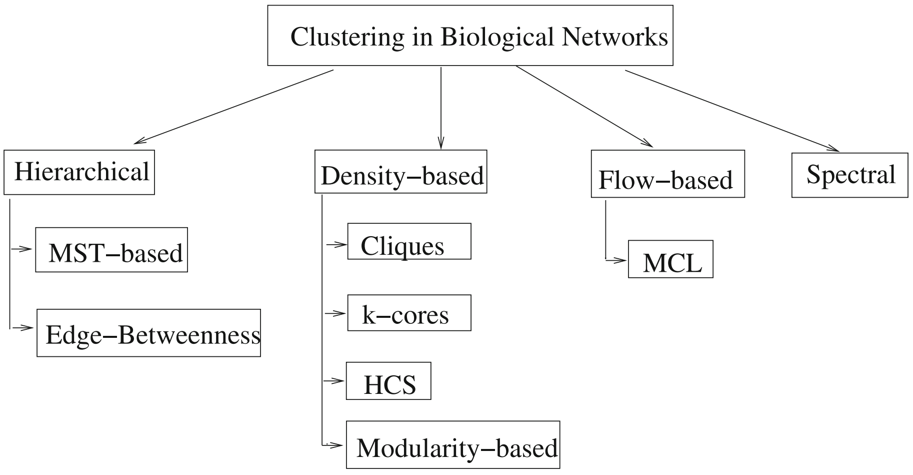

Fig. 11.2 A taxonomy of clustering algorithms in biological networks

cluster including all of the nodes and divide them into smaller clusters in each iteration [28]. The MST-based and edge-betweenness-based algorithms are examples of the latter hierarchical methods. Density-based algorithms search for the dense parts of the graph as possible clusters. Flow-based algorithms on the other hand are built on the idea that the flow between nodes in a cluster should be higher than the rest of the graph and the spectral clustering considers the spectral properties of the graph while clustering.

We search for clusters in biological networks to understand their behavior, rather than partitioning them. However, we will frequently need to partition a graph representing such a network for load balancing in a distributed memory computing system. Our aim is to send a partition of a graph to a process in such a system so that parallel processing can be achieved. The BFS-based partitioning algorithm of Sect. 7.5 can be used for this purpose. In the next sections, we will investigate sample algorithms of these methods in sequential and distributed versions in detail.

11.3 Hierarchical Clustering

We have described the basic hierarchical clustering methods in Sect. 7.3. We will now investigate two graph-based hierarchical clustering approaches to discover dense regions of biological networks.

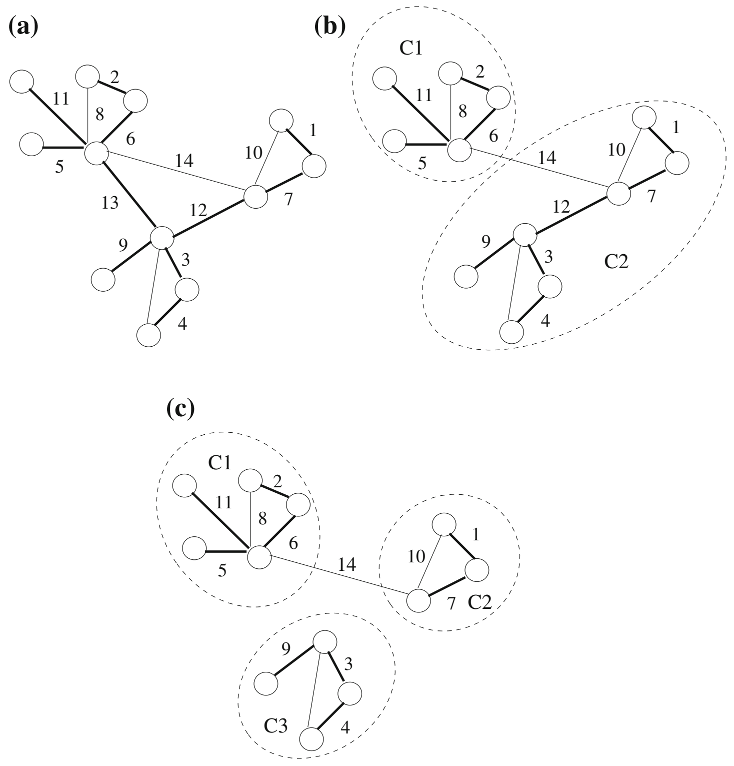

Fig. 11.3 MST-based clustering in a sample graph. MST is shown by *bold lines* and the edges are labeled with their weights. The highest weight edge in the MST has weight 13 and removed in the first step resulting in two clusters C_1 and C_2. The next iteration removes the edge with weight 12 and three clusters C_1, C_2, and C_3 are obtained

11.3.1 MST-Based Clustering

The general approach of MST-based clustering algorithms is to first construct an MST of the graph after which the heaviest weight edges from the MST are iteratively removed until the required number of clusters is obtained. The idea of this heuristic is that two nodes that are far apart should not be in the same cluster. Removing one edge in the first step will disconnect MST as MST is acyclic like any tree and will result in two clusters, hence we need to remove the heaviest $k - 1$ edges to get k clusters. Note that removing the heaviest edge may not result in a disconnected graph. Figure 11.3 displays the MST of a graph removing of two edges from which results in three clusters.

Instead of removing one edge at each iteration of the algorithm, we may start with a threshold edge weight value τ and remove all edges that have higher weights than τ in the first step which may result in a number of clusters. We can then check the quality Q of the clusters we obtain and continue if Q is lower than expected. This parameter can be the ratio of intra-cluster density to the inter-cluster density or it can simply be computed as the ratio of the total weight of intra-cluster edges in the current clusters to the total weight of inter-cluster edges. We may modify the value of τ as we proceed to refine the output clusters as a large τ value may result in many small clusters and a small value will generally give few large clusters. MST of a graph can be constructed using one of the greedy approaches as follows:

- *Prim's Algorithm*: This algorithm greedily includes an edge of minimum weight in MST among edges that are incident on the current MST vertices but not part of the current MST as we have seen in Sect. 3.6. Prim's algorithm requires $O(n^2)$ as it checks each vertex against all possible vertex connections but this time may be reduced to $O(mlogn)$ by using the binary heap data structure and to $O(m + nlogn)$ by Fibonacci heaps [13].
- *Kruskal's Algorithm*: Edges are sorted with respect to their weights and starting from the lightest weight edge, an edge is included in MST if it does not create a cycle with the existing MST edges. The time for this algorithms is dominated by the sorting of edges which is $O(m \log m)$ and if efficient algorithms such as union-find are used, it requires $O(m \log n)$ time.
- *Boruvka's Algorithm*: This algorithm is the first MST algorithm designed to construct an efficient electricity network for Moravia, dating back to 1926 [5]. It finds the lightest edges for each vertex and contracts these edges to obtain a simpler graph of components and then the process is repeated with the components of the new graph until an MST is obtained. It requires $O(m \log n)$ time to build the MST.

11.3.1.1 A Sequential Algorithm

We will now describe a simple sequential MST algorithm that does not require the number of clusters beforehand. It starts with an initial distance value τ and a cluster quality value Q and at each iteration of the algorithm, edges that have weights greater than the current value τ_i are deleted from the graph to obtain clusters. The new cluster quality Q_i is then computed and if this value is lower than the required quality of Q_{req}, another iteration is executed. Algorithm 11.1 shows the pseudocode for this algorithm [20]. *BFS_form* is a function that builds a BFS tree starting from the specified vertex and includes all vertices in this BFS tree in the specified cluster.

Algorithm 11.1 *Seq_MST_Clust*

```
1: Input : $G(V, E, w)$                                   ▷ edge-weighted graph
2:         $\tau_i \leftarrow \tau_1, Q_{req} \leftarrow Q_1$            ▷ initialize
3: Output : $C_1, ..., C_k$                                ▷ $k$ clusters
4: construct MST of $G$
5: $D[n, n] \leftarrow$ distances between points
6: int $i \leftarrow 1$
7: while $Q_i < Q_{req}$ do
8:     $j \leftarrow 1$
9:     for all $(a, b) \in D$ such that $d(a, b) < \tau_i$ do
10:        $D[a, b] \leftarrow \infty$
11:        $BFS\_form(a, C_j)$
12:        $BFS\_form(b, C_{j+1})$
13:        $j \leftarrow j + 1$
14:    end for
15:    compute $Q_i$
16:    adjust $\tau_i$ if required
17:    $i \leftarrow i + 1$
18: end while
```

11.3.1.2 Distributed Algorithms

Let us review the MST-based clustering problem; we need to first construct an MST of the graph, then we either remove the heaviest weight edge at each step or may remove a number of edges that have weights greater than a threshold value. The building of the MST dominates the time taken for the clustering algorithm and we have already reviewed ways of parallelizing this process in Sect. 7.5. We may partition the graph to processors of the distributed system and then implement Boruvka's algorithm in parallel as a first approach. The CLUMP algorithm takes a different approach by forming bipartite subgraphs and use Prim's algorithm in parallel in these graphs and combine the partial MSTs to get the final MST as described next.

CLUMP

Clustering through MST in parallel (CLUMP) is a clustering method designed to detect dense regions of biological data [38]. It is not particularly designed for biological networks, however, it uses representation of biological data as a weighted undirected graph $G(V, E)$ in which each data point is a node and an edge (u, v) connecting nodes u and v has a weight proportional to distance between these two points. This algorithm constructs an MST of the graph G and proceeds similarly to the MST-based clustering algorithms to find clusters. Since the most time-consuming part of an any MST-based clustering scheme is the construction of the MST, the following steps of CLUMP are proposed:

1. The original graph $G(V, E, w)$ is partitioned into $G_j(V_j, E_j), j = 1, \ldots, s$ where G_j is the subgraph induced by V_j, E_{ij} is the set of edges between V_i and V_j.
2. The bipartite graphs $B_{ij} = \{Vi \cup V_j, E_{ij}\}$ for all subgraphs formed in step 1 are constructed.
3. For each G_i; an MST T_{ii}, and for each B_{ij} an MST T_{ij} is constructed in parallel.
4. A new graph $G^0 = \bigcup T_{ij}$, $1 \leq i \leq j \leq s$ is constructed by merging all MSTs from step 3.
5. The MST of the graph G^0 is constructed.

The authors showed that the MST of G^0 is the MST of the original graph G. The idea of this algorithm is to provide a speedup by parallel formation of MSTs in step 3 since the formation of G^0 in the last step is relatively less time consuming due to the sparse structure of this graph. Prim's algorithm was used to construct MSTs and the algorithm was evaluated using MPI and ANSI C. The authors also provided an online CLUMP server which uses an MySQL database for registered users. CLUMP is implemented for hierarchical classification of functionally equivalent genes for prokaryotes at multi-resolution levels and also for the analysis of the *Diverse Sitager Soil Metagenome*. The performance of this parallel clustering algorithm was found highly effective and practical during these experiments.

MST-based clustering is used in various applications including biological networks [45]. A review of parallel MST construction algorithms is provided in [34] and a distributed approach using MPI is reported in [17].

11.3.2 Edge-Betweenness-Based Clustering

As we have seen in Sect. 11.5, the vertex betweenness centrality $C_B(v)$ of a vertex v is the percentage of the shortest paths that pass through v. Similarly, the edge betweenness centrality $C_B(e)$ of an edge e is the ratio of the shortest paths that pass through e to total number of shortest paths. These two metrics are shown below:

$$C_B(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}}, \qquad C_B(e) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(e)}{\sigma_{st}} \tag{11.7}$$

Girvan and Newman proposed an algorithm (GN algorithm) based on edge betweenness centrality to provide clusters of large networks which consists of the following steps [24].

1. Find edge betweenness values of all edges of the graph $G(V, E)$ representing the network.
2. Remove the edge with the highest edge betweenness value from the graph.
3. Recalculate edge betweennesses in the new graph.
4. Repeat steps 1 and 2 until a quality criteria is satisfied.

The general idea of this algorithm is that an edge e which has a higher edge betweenness value than other edges has a higher probability of joining two or more clusters as there are more shortest paths passing through it. In the extreme case, this edge could be a bridge of G in which case removing it will disconnect G. It is considered as a hierarchical divisive algorithm since it starts with a single cluster containing all vertices and iteratively divides clusters into smaller ones. The fundamental and most time consuming step in this algorithm is the computation of the edge betweenness values which can be performed using the algorithms described in Sect. 11.5.

11.3.2.1 A Distributed Edge-Betweenness Clustering Algorithm

GN algorithm has a high computational cost making it difficult to implement in large networks such as the PPI networks. Yang and Lonardi reported a parallel implementation of this algorithm (YL Algorithm) on a distributed cluster of computers and showed that a linear speedup is achieved up to 32 processors [46]. The quality of the clusters is validated using the modularity concept described in Sect. 11.2. All-pairs shortest paths for an unweighted graph G can be computed using the BFS algorithm for all nodes of G and the edge betweenness values can be obtained by summing all pair dependencies $\delta_{st}(v)$ over all traversals. The main idea of YL Algorithm is that the BFS algorithm can be executed independently in parallel on a distributed memory computer system. It first distributes a copy of the original graph G to k processors, however, each processor p_i executes BFS on its set of vertices V_i only. There is one supervisor processor p_s which controls the overall operation. Each processor p_i finds the pair dependencies for its vertices in V_i it is assigned and sends its results to p_s which in turn sums all of the partial dependencies and finds the edge e with the highest edge betweenness value. It then broadcasts the edge e to all processors which delete e from their local graph and continue with the next iteration until there are no edges left as shown in Algorithm 11.2 for the supervisor and each worker process. Modularity is used to find where to cut the output dendrogram. YL algorithm was implemented in C++ using MPI on five different PPI networks. The results showed it found clusters in these networks correctly with linear speedups up to 32 processors.

Algorithm 11.2 *YL_Alg*

```
1: Input : $G(V, E)$                                    ▷ undirected graph,
2: Output : edges $e_1, \ldots, e_m$ of $G$ in reversal removal order
3: if I am root then
4:     assign vertex sets $V_1, \ldots, V_k$ to processors $p_1, \ldots, p_k$
5: end if
6: while there are edges left on processors do
7:     if I am root then
8:         receive $\delta_{st}(v)$
9:         calculate edge betweenness values for all edges
10:        find the edge $e$ with the maximum value and broadcast $e$
11:    else
12:        for all $v \in V_i$ parallel do     ▷ worker process $p_i$ does this part in parallel with others
13:            $BFS(G_i, v)$
14:            send all pair dependencies $\delta_{st}(v_i)$ to the root
15:        end for
16:        receive the edge $e$ that has the highest betweenness
17:        remove $e$ from the local graph partition $G_i$
18:        synchronize
19:    end if
20: end while
```

11.4 Density-Based Clustering

The dense parts of an unweighted graph have more edges than average and exhibit possible cluster structures in these regions. If we can find methods to discover these dense regions, we may detect clusters. Cliques are perfect clusters and detecting a clique does not require any comparison with the density in the rest of the graph. In many cases, however, we will be interested in finding denser regions of a graph with respect to other parts of it rather than absolute clique structures. We will first describe algorithms to find cliques in a graph and then review k-cores, HCS, and modularity-based algorithms with their distributed versions in this section.

11.4.1 Clique Algorithms

A clique of a graph G is a subset of its nodes which induce a complete graph as we saw in Sect. 3.6, and finding the maximum clique of a graph is NP-hard [23]. In the extreme case, detecting clusters in a graph G can be reduced to finding cliques of G. However, a graph representing a biological network may have only few cliques due to some links not being detected or deleted from it. For this reason, we would be interested in finding *clique-like* structures in a graph rather than full cliques to discover clusters. These structures can be classified as follows [20]:

Definition 11.1 (*k-clique:*) In a k-clique subgraph G' of G, the shortest path between any two vertices in G' is at most k. Paths may consist of vertices and edges external to G'.

Definition 11.2 (*quasi-clique:*) A quasi-clique is a subgraph G' of G where G' has at least $\gamma|G'||G'|-1)/2$ edges. In other words, a quasi-clique of size m has γ fraction of the number of edges of the clique of the same size.

Definition 11.3 (*k-club:*) In a k-club subgraph G' of G, the shortest path between any two vertices which consists of vertices and edges in G', is at most k.

Definition 11.4 (*k-core:*) In a k-core subgraph G' of G, each vertex is connected to at least k other vertices in G'. A clique is a $(k-1)$ core.

Definition 11.5 (*k-plex:*) A k-plex is a subgraph G' of G, each vertex has at most k connected to at least $n-k$ other vertices in G'. A clique is a 1-plex.

11.4.1.1 Bron and Kerbosch Algorithm

Bron and Kerbosch proposed a recursive backtracking algorithm (BK algorithm) to find all cliques of a graph [9]. This algorithm works using three disjoint sets of vertices; R which is the current expanding clique, P is the set of potential vertices connected to vertices in R, and X contains all of the vertices already processed. The algorithm recursively attempts to generate extensions to R from the vertices in P which do not contain any vertices in X. It consist of the following steps [9]:

1. Select a candidate.
2. Add the selected candidate to R.
3. Create new sets P and X from the old sets by removing all vertices not connected to R.
4. Call the extension operator on the newly formed sets.
5. Upon return, remove the selected candidate from R, add it to X.

In order to have a clique, set P should be empty, as otherwise R could be extended. Also, the set X should be empty to ensure R is maximal otherwise it may have been contained in another clique. Algorithm 11.3 shows the pseudocode of this algorithm based on the above steps. The time complexity of this algorithm was evaluated to be $O(4^n)$ according to the experimental observations of the authors. Bron and Kerbosch also provided a second version of this algorithm that uses pivots with an experimental time complexity of $O(3.14^n)$ where 3^n is the theoretical limit.

Blaar et al. implemented a parallel version of Bron and Kerbosch algorithm using thread pools in Java and provided test results using 8 processors [6]. Mohseni-Zadeh et al. provided a clustering method they called Cluster-C to cluster protein sequences based on the extraction of maximal cliques [32] and Jaber et al. implemented a parallel version of this algorithm using MPI [27]. Schmidt et al. provided a scalable parallel implementation of Bron and Kerbosch algorithm on a Cray XT supercomputer [41].

Algorithm 11.3 *Bron Kerbosch Algorithm*

```
1: procedure BronKerbosch($R, P, X$)
2:     $P \leftarrow V$ includes all of the vertices and $R, X \leftarrow \emptyset$
3:     if $P = \emptyset \wedge X = \emptyset$ then
4:         return $R$ as a maximal clique
5:     else
6:         for all $v \in P$ do
7:             BronKerbosch($R \cup \{v\}, P \cap N(v), X \cap N(v)$)
8:             $P \leftarrow P \setminus \{v\}$
9:             $X \leftarrow P \cup \{v\}$
10:        end for
11:    end if
12: end procedure
```

11.4.2 *k*-core Decomposition

Given a graph $G(V, E)$, a subgraph $G_k(V', E')$ of G induced by V' is a k-core of G if and only if $\forall v \in V' : \delta(v) \geq k$ and G_k is the maximum graph with this property. *Main core* of a graph G is the core of G with maximum order and the *coreness value* of a vertex v is the highest order of a core including v. The k-class of a vertex can be defined as the set of vertices which all have a degree of k [15]. Cores of a graph may not be connected and a smaller core is the subset of a larger core. Figure 11.4 displays 3 nested cores of a sample graph.

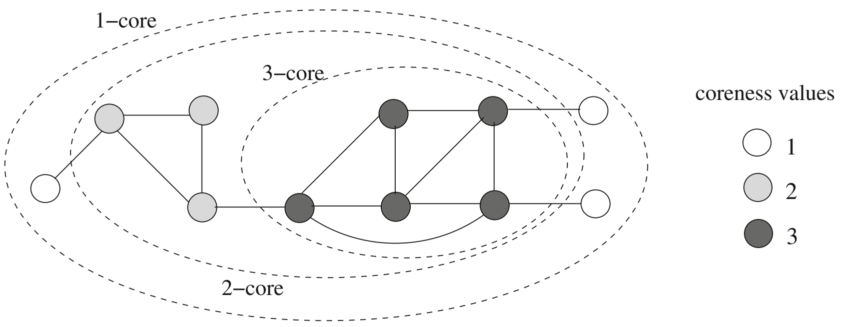

Fig. 11.4 Cores of a sample graph

The *k-core decomposition* of a graph G is to find the k-core subgraphs of G for all k which can therefore be reduced to finding coreness values of all vertices of G. Core decomposition has been used for complex network analysis [2] and to detect k-cores in PPI networks [1]. Detecting group structures such as *cliques*, *k-cliques*, *k-plexes*, and *k-clubs* are difficult and NP-hard in many cases, however, finding k-cores of a graph can be performed in polynomial time as we describe in the next section.

11.4.2.1 Batagelj and Zaversnik Algorithm

Batagelj and Zaversnik proposed a linear time algorithm (BZ algorithm) to find the core numbers of all vertices of a graph G [4] based on the property that removing all vertices of degree less than k from a graph with their incident edges recursively will result in a k-core. This algorithm first sorts the degrees of vertices in increasing order and inserts them in a queue Q. It then iteratively removes the first vertex v from the queue, and labels it with the core value which equals the current degree of v, and decrements the degree of each neighbor u of v, if u has a larger degree than v, to effectively delete the edge between them. The vertex u may not have a degree smaller than v as Q is sorted but u may have equal degree to v in which case we do not want to change its degree since u will be moved to a lower class. The pseudocode of this algorithm is shown in Algorithm 11.4 [4,20]. The algorithm ends when Q is empty and k-cores of G consist of vertices which have label values up to and including k. Batagelj and Zaversnik showed that the time complexity of this algorithm is $O(max(m, n))$, and time complexity is $O(m)$ in a connected network as $m \geq n - 1$ in such a case.

Algorithm 11.4 *BZ_Alg*

```
1: Input : $G(V, E)$
2: Output : core values of vertices
3: $Q \leftarrow$ sorted vertices of $G$ in increasing weight
4: while $Q \neq \emptyset$ do
5:     $v \leftarrow$ front of $Q$
6:     $core(v) \leftarrow \delta(v)$
7:     for all $u \in N(v)$ do
8:         if $\delta(u) > \delta(v)$ then
9:             $\delta(u) \leftarrow \delta(u) - 1$
10:        end if
11:    end for
12:    update $Q$
13: end while
```

Execution steps of this algorithm in the sample graph of Fig. 11.5 is shown in Table 11.1.

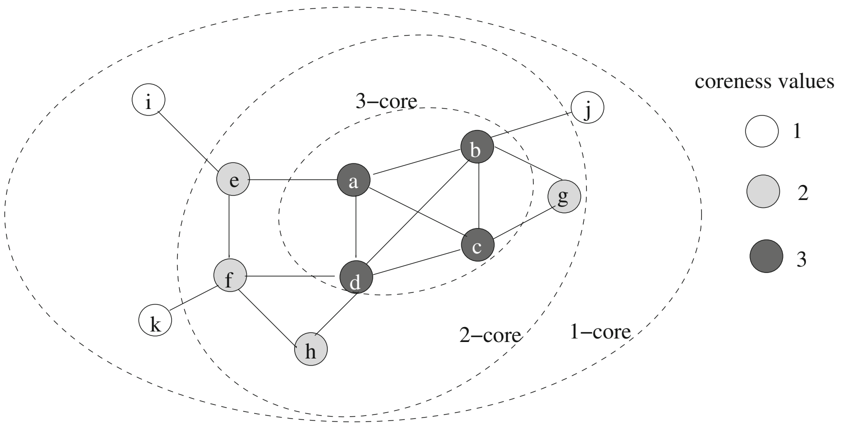

Fig. 11.5 Output of BZ algorithm on a sample graph

11.4.2.2 Molecular Complex Detection Algorithm

The Molecular Complex Detection (MCODE) Algorithm is used to discover protein complexes in large PPI networks [3]. It consists of vertex weighting, complex prediction, and optional post-processing to add or filter proteins in the output complexes. In the first step of this algorithm, the local vertex density of a vertex v is computed using the highest k-core of its neighborhood. We will repeat the definition of the clustering coefficient $cc(v)$ of a vertex v as follows:

$$cc(v) = \frac{2m_v}{n_v(n_v - 1)}, \tag{11.8}$$

where n_v is the number of neighbors of v and m_v is the existing number of edges between these neighbors. It basically shows how well connected the neighbors of v are. The MCODE algorithm defines the *core clustering coefficient* of a vertex v as the density of the highest k-core of the closed neighborhood of v. Using this parameter provides removal of many low-degree vertices seen in PPI networks due to the scale-free property, while emphasizing the high-degree nodes which we expect to see in the clusters. The weight of a vertex v, $w(v)$, is then assigned as the product of the core clustering coefficient $ccc(v)$ of vertex v and the highest k-core value $k_{\max}$ in the closed neighborhood of v as follows.

$$w(v) = ccc(v) \times k_{\max} \tag{11.9}$$

The second step of the algorithm involves selecting the highest weighted vertex v as the seed vertex and recursively adding vertices in its neighborhood if their weights are above a threshold. The threshold named vertex weight percentage (VWP) is a predetermined percentage of the weight of the seed vertex v. When there are no more vertices that can be added to the complex, this sub-step is stopped and the process is repeated with the next highest and unvisited vertex. WWP parameter effectively specifies the density of the complex obtained, with a high threshold resulting in a

Table 11.1 BZ algorithm execution

Iterations	Queue sorted in ascending degree					Coreness values		
	1	2	3	4	5	$k = 1$	$k = 2$	$k = 3$
1	i, j, k	g, h	e	a, c, f	b, d	$\{i\}$	$\{\varnothing\}$	$\{\varnothing\}$
2	j, k	g, h, e	–	a, c, f	b, d	$\{i, j\}$	$\{\varnothing\}$	$\{\varnothing\}$
3	k	g, h, e	–	a, c, f, b	d	$\{i, j, k\}$	$\{\varnothing\}$	$\{\varnothing\}$
4	–	g, h, e	f	a, c, b	d	$\{i, j, k\}$	$\{g\}$	$\{\varnothing\}$
5	–	h, e	f, b, c	a	d	$\{i, j, k\}$	$\{g, h\}$	$\{\varnothing\}$
6	–	e, f	b, c	a, d	–	$\{i, j, k\}$	$\{g, h, e\}$	$\{\varnothing\}$
7	–	f	b, c, a	d	–	$\{i, j, k\}$	$\{g, h, e, f\}$	$\{\varnothing\}$
8	–	–	b, c, a, d	–		$\{i, j, k\}$	$\{g, h, e, f\}$	$\{b\}$
9	–	–	c, a, d	–	–	$\{i, j, k\}$	$\{g, h, e, f\}$	$\{b, c\}$
10	–	–	a, d	–	–	$\{i, j, k\}$	$\{g, h, e, f\}$	$\{b, c, a\}$
11	–	–	d	–	–	$\{i, j, k\}$	$\{g, h, e, f\}$	$\{b, c, a, d\}$

smaller and a denser complex and a low value results in the contrary. The last step is used to filter and modify the complexes. Complexes that do not have at least a 2-core are removed during filtering process and the optional *fluff* operation increases the size of the complexes according to the fluff parameter which is between 0.0 and 1.0. The time complexity of this algorithm is $O(nmh^3)$ where h is the vertex size of the average vertex neighborhood in G as shown in [3].

There is not a reported parallel or distributed version of this algorithm, however, the search of dense neighbors can be performed by the BFS method and this step can be employed in parallel using a suitable parallel BFS algorithm such as in [10].

11.4.2.3 A Distributed k-core Algorithm

Although the BZ algorithm is efficient for small graphs, distributed algorithms are needed to find k-cores in large biological networks such as PPI networks. The BZ algorithm has inherently serial processing as we need to find the vertex with the smallest degree globally and hence is difficult to parallelize. One very recent effort to provide a distributed k-core decomposition method was proposed by Montresor et al. [33]. This algorithm attempts to find coreness values of vertices in a graph G which provides k-cores of G indirectly. They considered two computational models; in *one-to-one* model, each computational node is responsible for one vertex and one node handles all processing for a number of vertices in *one-to-many* model. The latter is obtained by extending the first model. The main idea of this algorithm is based on the *locality* concept in which the coreness of a node u is the greatest k value where u has at least k neighbors, each belonging to k or larger cores. Based on this concept, a node can compute its coreness value using the coreness values of its neighbors. Each node u in this algorithm forms an estimate of its coreness value and sends this value to it neighbors and uses the value received from neighbors to recompute its estimate. After a number of periodic rounds, coreness values can be determined when no new estimates are generated. The time complexity was shown to be $O(n - s + 1)$ rounds where s is the number of nodes in the graph with minimal degree. The messages exchanged during the algorithm was shown to be $O(\Delta m)$ where Δ is the maximum degree of the graph. The authors have experimented one-to-one and one-to-many versions of this algorithm with both a simulator and real large data graphs and found it is efficient.

11.4.3 Highly Connected Subgraphs Algorithm

The highly connected subgraphs (HCS) algorithm proposed by Hartuv and Shamir [26] searches dense subgraphs with high connectivity rather than cliques in undirected unweighted graphs. The general idea of this algorithm is to consider a subgraph G' of n vertices of a graph G as highly connected if G' requires a minimum of $n/2$ edges to have it disconnected. In other words, the edge connectivity of G', $k_E(G')$ should be $n/2$ to accept it as a highly connected subgraph. The algorithm shown in

Algorithm 11.5 starts by first checking if G is highly connected, otherwise uses the minimum cut of G to partition G into H and H', and recursively runs HCS procedure on H and H' to discover highly connected subgraphs.

Algorithm 11.5 *HCS_Alg*

```
1: procedure HCS(G)
2:     Input : G(V, E)
3:     Output : highly connected clusters of G
4:     (H, $\bar{H}$, C) ← MinCut(G)
5:     if G is highly connected then
6:         return(G)
7:     else
8:         HCS(H)
9:         HCS($\bar{H}$)
10:    end if
11: end procedure
```

The execution of HCS algorithm is shown in a sample graph in Fig. 11.6 after which three clusters are discovered. HCS has a time complexity of $2N \times f(n, m)$ where N is the number of clusters discovered and $f(n, m)$ is the time complexity of finding a minimum cut in a graph that has n vertices and m edges. HCS has been successfully used to discover protein complexes, and cluster identification via connecting kernel (CLICK) algorithm is an adaptation of HCS algorithm for weighted graphs [43].

11.4.4 Modularity-Based Clustering

We have seen that the modularity parameter Q provides a good indication of the quality of the clustering in Sect. 12.2. The algorithm proposed by Girvan and Newman

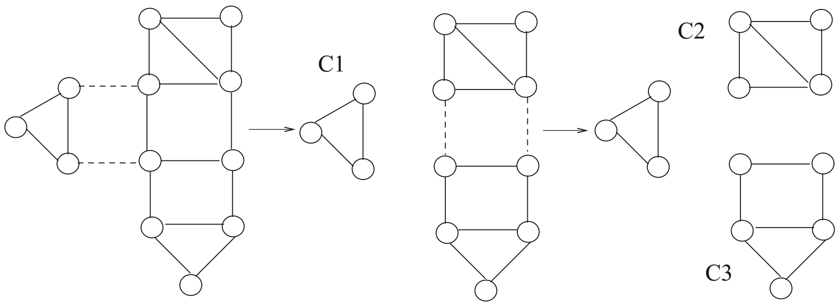

Fig. 11.6 HCS algorithm run on a sample graph. Three clusters C_1, C_2 and C_3 are discovered

(GNM algorithm) attempts to obtain clustering by increasing the value of Q as follows [35]:

1. Each node of the graph is a cluster initially.
2. Merge the two clusters that will increase the modularity Q by the largest amount.
3. If merges start reducing modularity, stop.

This algorithm can be classified as an agglomerative hierarchical clustering algorithm as it iteratively forms larger clusters and the output is a dendrogram as in such algorithms. Its time complexity is $O((m+n)n)$, or $O(n^2)$ on sparse graphs.

11.4.4.1 A Survey of Modularity-Based Algorithms

A review of modularity and methods for its maximization is presented by Fortuna [22]. Clauset et al. provided a method to find clusters using modularity in favorable time using the sparse structure of a graph [16]. They kept the clustering information using a binary tree in which every node is a cluster formed by combining its children. The time complexity of this algorithm is $O(mh \log n)$ where h is the height of the tree. The Louvain method proposed in [7] evaluates modularity by moving nodes between clusters. A new coarser graph is then formed where each node is a cluster. This greedy method optimizes modularity in two steps. The small communities are searched locally first, and these communities are then combined to form the new nodes of the network. These two steps are repeated until modularity is maximized.

Parallel and Distributed Algorithms

Parallel and distributed algorithms that find clusters using modularity are scarce. Gehweiler et al. proposed a distributed diffusive heuristic algorithm for clustering using modularity [25]. Riedy et al. proposed a massively parallel community detection algorithm for social networks based on Louvani method [39]. We will describe this algorithm in detail as it is one of the only parallel algorithms for this purpose. It consists of the following steps which are repeated until a termination condition is encountered:

1. Every edge of the graph is labeled with a score. If all edges have negative scores, exit.
2. Compute a weighted maximal matching using these scores.
3. Coarsen matched groups into a new group which are the nodes of the new graph.

In the first step, the change in optimization metric is evaluated if two adjacent clusters are merged and a score is associated with each edge. The second step involves selecting pairs of neighboring clusters merging of which will improve the quality of clustering using a greedy approximately maximum weight maximal matching and the selected clusters are contracted according to the matching in the final step. The time complexity of this algorithm is $O(mk)$ where k is the number of contraction steps. Each step of this algorithm is independent and can be performed in parallel. Reidy et al. implemented this algorithm in Cray XMT2 and Intel-based server platforms using OpenMP and obtained significant speedups with high performance and good data scalability.

LaSalle and Karypis recently provided a multithreaded modularity-based graph clustering algorithm using the multilevel paradigm [31]. Multilevel approaches for graph partitioning are popular due to the high quality partitions produced by them. These methods consist of *coarsening*, *initial clustering*, and *uncoarsening* phases. The initial graph G_0 is contracted into a series of smaller graphs $G_1, \ldots, G_s$ in the coarsening phase using some heuristics. The final graph G_s is then partitioned into a number of nonoverlapping partitions using a direct partitioning algorithm in the second phase. In the uncoarsening phase, a coarser graph is projected back to a finer graph followed by a cluster refinement procedure such as the Kernighan–Lin algorithm [29]. Maximal matching of edges is a frequently used heuristic during coarsening. The algorithm matches vertex u with its neighbor vertex v that provides maximum modularity in the coarsening phase of multilevel methods. In this study, the authors developed two modularity-based refinement methods called random boundary refinement and greedy boundary refinement which consider border vertices between clusters. This algorithm named *Nerstrand* has an overall time complexity of $O(m + n)$ which is the sum of complexities of three phases and the space complexity is shown to be also $O(m+n)$. The *Nerstrand* algorithm is implemented in shared memory multithreaded environment where the number of edges each thread works is balanced explicitly and the number of vertices for each thread is balanced implicitly. Each thread creates one or more initial clustering of its vertices and the best clustering is selected by reduction. Each thread then performs cluster projection over the vertices it is responsible. The authors implemented this algorithm in OpenMP environment and compared serial and multithreaded *Nerstrand* performances with other methods such as Louvain. They found serial *Nerstrand* performs similar or slightly better than Louvain method but parallel *Nerstand* finds clusters much faster than contemporary methods, 2.7–44.9 times faster.

11.4.4.2 A Distributed Algorithm Proposal

We now propose a distributed algorithm for modularity-based clustering. The general idea of our algorithm is to partition the modularity finding computation among k processes. Each process computes the merge that causes the maximum modularity in its partition and sends the pair that gives this value to the central process. Upon the gathering of local results in the central process, it finds the merge operation that results in maximum modularity change and broadcasts the cluster pair that has to be merged to all processes. This process continues until modularity does not change significantly anymore. We show below the main steps of this algorithm where we have k processes running on k processors and one of these processes, p_0, is designated as the root process that controls the overall execution of the algorithm as well as performing part of the algorithm. We will assume the graph $G(V, E)$ is already

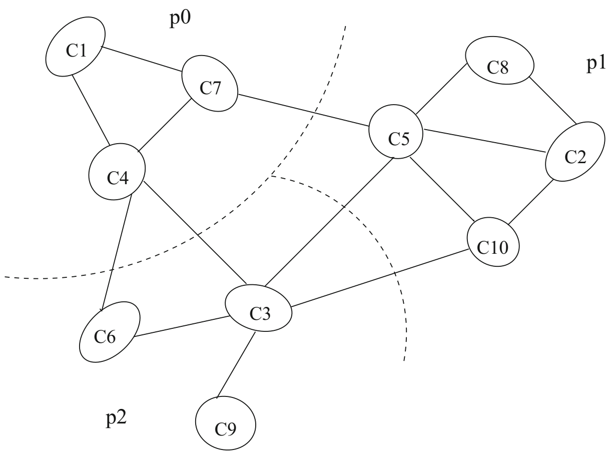

Fig. 11.7 Clusters for the distributed modularity-based clustering algorithm

partitioned into a number of clusters, however, we could have started by the original graph assuming each node is a cluster. The root performs the following steps:

1. Assume each cluster is a supernode and perform a BFS partition on the original graph to have k cluster partitions such that $\mathcal{C} = \{C_1, \ldots, C_k\}$.
2. **send** each partition to a process p_i.
3. **find** the best cluster pair in my partition.
4. **receive** best cluster pairs from each p_i.
5. **find** the pair C_x, C_y that gives the maximum modularity.
6. **broadcast** C_x, C_y to all processes.
7. **repeat** steps 3–5 until modularity starts reducing.

Figure 11.7 displays an example network that already has 10 clusters. The root process p_0 partitions this network by the BFS partitioning algorithm and sends the two cluster partitions to processes p_1 and p_2. In this example, p_1 computes the modularity values for combining operations $C_2 \cup C_8$, $C_2 \cup C_{10}$, $C_2 \cup C_5$, $C_5 \cup C_8$, $C_5 \cup C_{10}$, and $C_5 \cup C_7$, assuming for any cluster pair across the borders, the process that owns the lower identifier cluster is responsible to compute modularity. Further optimizations are possible such as in the case of local cluster operation in a process is decided to merge C_2 and C_{10} in p_1, the processes p_0 and p_2 do not need to compute their modularity values again as they are not affected.

11.5 Flow Simulation-Based Approaches

A different approach than the traditional graph clustering methods using density is considered in flow simulation-based methods. The goal in this case is to predict the regions in the graph where the flow will gather. The analogy of the graph is a water distribution network with nodes representing storages, and the edges as the pipes between them. If we pump water to such a network, flow will gather at nodes which have many pipes ending in them and hence in clusters. An effective way of simulating the flow in a network is by using random walks which is described in the next section.

11.5.1 Markov Clustering Algorithm

Markov Clustering Algorithm (MCL) is a fast clustering algorithm based on stochastic flow in graphs [18] and uses the following heuristics [20]:

1. Number of paths of length k between two nodes in a graph G is larger if they are in the same cluster and smaller if they are in different clusters.
2. A *random walk* starting from a vertex in a dense cluster of G will likely end in the same dense cluster of G.
3. Edges between clusters are likely to be incident on many shortest paths between all pairs of nodes.

The algorithm is based on random walks assuming by doing the random walks on the graph, we may be able to find where the flow gathers which shows where the clusters are. Given an undirected, unweighted graph $G(V, E)$ and its adjacency matrix A, this algorithm first forms a column-stochastic matrix M. The ith column of M shows the flows out of node v_i and the ith row contains the flows into v_i. The sum of column i of this matrix equals 1 as this is the sum of the probabilities of reaching any neighbor from vertex v_i, however, the sum of the rows may not add up to 1. The matrix M can be obtained by normalizing the columns of the adjacency matrix A as follows.

$$M(i,j) = \frac{A(i,j)}{\sum_{k=1}^{n} A(k,j)} \qquad (11.10)$$

This operation is equal to $M = AD^{-1}$ where D is the diagonal matrix of the graph G. The MCL algorithm inputs the matrix M and performs two iterative operations on M called *expansion* and *inflation* and an additional *pruning* step as follows.

- **Expansion**: This operation simply involves taking the eth power of M as below:

$$M_{\text{exp}} = M^{e}, \qquad (11.11)$$

e being a small integer, usually 2. Based on the properties of M, M_{exp} shows the distribution of a random walk of length r from each vertex.

- **Inflation**: In this step, the rth power of each entry in M is computed and this value is normalized by dividing it to the sum of the rth power of column values as below.

$$M_{\text{inf}}(i,j) = \frac{M(i,j)^r}{\sum_{k=1}^{n} M(k,j)^r} \tag{11.12}$$

  The idea here is to emphasize the flow where it is large and to decrease it where it is small. This property makes this algorithm suitable for scale-free networks such as PPI networks, as these networks have few high-degree hubs and many low-degree nodes. As clusters are formed around these hubs, emphasizing them and deemphasizing the low-degree nodes removes extra processing around the sparse regions of the graph.
- **Pruning**: The entries which have significantly smaller values than the rest of the entries in that column are removed. This step reduces the number of nonzero column entries so that memory space requirements are decreased.

Algorithm 11.6 displays the pseudocode for MCL algorithm [42].

Algorithm 11.6 *MCL_Alg*

```
1: Input : $G(V,E)$                                              ▷ undirected unweighted graph
2:         expansion parameter $e$, inflation parameter $r$
3: Output : Clusters $C_1, \ldots, C_k$ of $G$
4: $A \leftarrow$ adjacency matrix of $G$
5: $M \leftarrow AD^{-1}$                                         ▷ initialize $M$
6: repeat
7:     $M_{exp} \leftarrow M^e$                                   ▷ expand
8:     inflate $M_{inf} \leftarrow M_{exp}$ using $r$             ▷ inflate
9:     $M \leftarrow M_{exp}$
10: until $M$ converges
11: interpret the resulting matrix $M_{inf}$ to find clusters
```

After a number of iterations, there will be only one nonzero element at each column of M and the nodes that have flows to this node will be interpreted as a single cluster. The level of clustering can be modified by the parameter r, with the lower r resulting in a coarser clustering. The time complexity of this algorithm is $O(n^3)$ steps since multiplication of two $n \times n$ matrices takes n^3 time during expansion, and the inflation can be performed in $O(n^2)$ steps. The convergence of this algorithm has been shown experimentally only where the number of rounds required to converge was between 10–100 steps [3]. The MCL algorithm has been successfully implemented in biological networks in various studies [8,44], however, the scalability of MCL especially at the expansion step was questioned in [42]. Also, MCL was found to discover too many clusters in the same study and a modification to MCL by a multilevel algorithm was proposed.

11.5.2 Distributed Markov Clustering Algorithm Proposal

The MCL algorithm has two time consuming steps as expansion and inflation described above. We can see that these are matrix operations which can be performed independently on a distributed memory computing system whether a multiprocessor or totally autonomous nodes connected by a network. The expansion is basically a matrix multiplication operation in which many parallel algorithms exist. The inflation operation yields asynchronous operations and hence can be performed in a distributed system.

We will now sketch a distributed algorithm to perform MCL in parallel using m number of distributed memory processes $p_0, \ldots, p_{m-1}$. The supervisor process p_0 controls the overall flow of the algorithm and $m-1$ worker processes. The supervisor initializes the matrix M, broadcasts it to $m-1$ nodes which all perform multiplication of M by itself using row-wise 1-D partitioning and send back the partial results to the supervisor. This process now builds the M^2 matrix which can be partitioned again and sent to workers which will multiply part of it with their existing copy of M. This process is repeated for t times to conclude the expansion of the first iteration and for $t = 2$, finding M^2 will be sufficient. The supervisor can now send the expanded matrix M_{exp} by column partitioning it to $m-1$ processes each of which simply takes the eth power of each entry in columns, normalizes them and sends the resulting columns to the supervisor. The task of the supervisor now is to check whether M has converged and if this is not achieved a new iteration is started with the seed M_p. Algorithm 11.7 shows the pseudocode for the distributed MCL algorithm which can easily be implemented using MPI.

Algorithm 11.7 *DistMCL_Alg* Supervisor Process

```
1: Input : $G(V, E)$                                        ▷ undirected graph,
2:        expansion parameter $e$, inflation parameter $r$
3: Output : Clusters $C_1, \ldots, C_k$ of $G$
4: $A \leftarrow$ adjacency matrix of $G$
5: $M_p \leftarrow AD^{-1}$                                  ▷ initialize $M$
6: repeat
7:    broadcast $M_p$ to $m-1$ processes
8:    compute my partial product
9:    gather partial products from all workers              ▷ synchronize
10:   build $M_{exp}$
11:   send rows of $M_{exp}$ to $m-1$ processes
12:   compute my partial inflation
13:   gather partial products from all workers              ▷ synchronize
14:   build $M_{exp}$
15:   $M_p \leftarrow M_{exp}$
16: until $M$ converges
17: interpret the resulting matrix $M_{inf}$ to find clusters
```

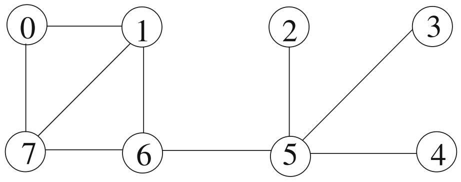

Fig. 11.8 An example graph for distributed MCL algorithm

We will show the implementation of the distributed MCI algorithm using a simple example graph of Fig. 11.8 which will also show the detailed operation of the sequential algorithm.

The M matrix row partitioned by the supervisor for parallel processing using this graph will be:

$$M = \left[\begin{array}{cccccccccl}
0 & 0.33 & 0 & 0 & 0 & 0 & 0 & 0.33 & p_0 \\
0.5 & 0 & 0 & 0 & 0 & 0 & 0.33 & 0.33 & \\
\hline
0 & 0 & 0 & 0 & 0 & 0.25 & 0 & 0 & p_1 \\
0 & 0 & 0 & 0 & 0 & 0.25 & 0 & 0 & \\
\hline
0 & 0 & 0 & 0 & 0 & 0.25 & 0 & 0 & p_2 \\
0 & 0 & 1 & 1 & 1 & 0 & 0.33 & 0 & \\
\hline
0 & 0.33 & 0 & 0 & 0 & 0.25 & 0 & 0.33 & p_3 \\
0.5 & 0.33 & 0 & 0 & 0 & 0 & 0.33 & 0 &
\end{array}\right]$$

Assuming we have 4 processors p_0, p_1, p_2 and p_3; and p_0 is the supervisor; the row partitioning of M will result in rows 2,3 to be sent to p_1; rows 4,5 to p_2 and 6,7 to p_3. When the partial products are returned to p_0, it will form M_{exp} shown below:

$$M_{exp} = \left[\begin{array}{cc|cc|cc|cc}
p_0 & & p_1 & & p_2 & & p_3 & \\
0.33 & 0.109 & 0 & 0 & 0 & 0.083 & 0.012 & 0.109 \\
0.165 & 0.383 & 0 & 0 & 0 & 0 & 0.083 & 0.274 \\
0 & 0 & 0.25 & 0.25 & 0.25 & 0 & 0.083 & 0 \\
0 & 0 & 0.25 & 0.25 & 0.25 & 0 & 0.083 & 0 \\
0 & 0 & 0.25 & 0.25 & 0.25 & 0 & 0.083 & 0 \\
0 & 0.109 & 0 & 0 & 0 & 0.159 & 0 & 0.109 \\
0.33 & 0.165 & 0.25 & 0.25 & 0.25 & 0 & 0.301 & 0.109 \\
0.165 & 0.274 & 0 & 0 & 0 & 0.083 & 0.109 & 0.383
\end{array}\right]$$

We then do a column partitioning of it and distribute it to processors p_1, p_2 and p_3 which will receive columns 2,3; 4,5; and 6,7 consecutively. After they perform inflation operation on their columns, they will return the inflated columns to p_0 which will form the M_{inf} matrix as below. Although some of the entries start diminishing as can be seen, convergence has not been detected yet, and p_0 will continue with the next iteration.

$$M_{inf} = \begin{bmatrix} 0.401 & 0.044 & 0 & 0 & 0 & 0.178 & 0.001 & 0.047 \\ 0.099 & 0.538 & 0 & 0 & 0 & 0 & 0.053 & 0.291 \\ 0 & 0 & 0.25 & 0.25 & 0.25 & 0 & 0.053 & \\ 0 & 0 & 0.25 & 0.25 & 0.25 & 0 & 0.053 & \\ 0 & 0 & 0.25 & 0.25 & 0.25 & 0 & 0.053 & \\ 0 & 0.044 & 0 & 0 & 0 & 0.643 & 0 & 0.047 \\ 0.401 & 0.099 & 0.25 & 0.25 & 0.25 & 0 & 0.695 & 0.047 \\ 0.099 & 0.275 & 0 & 0 & 0 & 0.178 & 0.092 & 0.570 \end{bmatrix}$$

As one of the few studies to provide parallel/distributed MCL algorithm, Bustamam et al. implemented it using MPI with results showing improved performance [11]. They also provided another parallel version of MCL this time using graphic cards processors (GPUs) with many cores [12].

11.6 Spectral Clustering

Spectral clustering refers to a class of algorithms that use the algebraic properties of the graph representing a network. We have noted that the Laplacian matrix of a graph G is $L = D - A$ in unnormalized form, with D being the diagonal degree matrix which has d_i as the degree of the vertex i in its diagonal and A is the adjacency matrix. In normalized form, the Laplacian matrix $L = I - D^{-1/2}AD^{-1/2}$ and L has interesting properties that can be analyzed to find the connectivity information about a graph G. First of all, the eigenvalues of L are real as L is real and symmetric. The second eigenvalue is called the *Fiedler value* and the corresponding eigenvector for this eigenvalue, the *Fiedler vector* [21] provides connectivity information about the graph G. Using the Fiedler vector, we can partition a graph G into two balanced partitions in *spectral bisection* as follows [19]. We first construct the Fiedler vector and then compare each entry of this vector with a value s, if the entry $F[i] \leq s$ then the corresponding vertex of G, v_i, is put in partition 1 and otherwise it is placed in the second partition as shown in Algorithm 11.8. The variable s could simply be 0 or the median of the Fiedler vector. Figure 11.9 displays a simple graph that is partitioned using the value of 0.

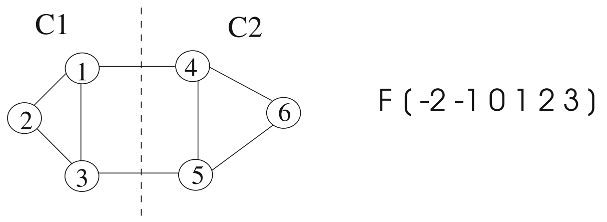

Fig. 11.9 Partitions formed using Fiedler vector. The first three elements Fiedler vector have values smaller or equal to zero and are put in the first partition and the rest are placed in the second

Algorithm 11.8 *Spect_Part*

```
1: Input : $A[n,n]$, $D[n,n]$          ▷ adjacency matrix and degree matrix of $G$
2: Output : $V_1$, $V_2$               ▷ two balanced partitions of $G$
3: $L \leftarrow D - A$
4: calculate Fiedler vector $F$ of $L$
5: for $i = 1$ to $n$ do
6:     if $F[i] \leq s$ then
7:         $V_1 \leftarrow V_1 \cup \{v_i\}$
8:     else  $V_2 \leftarrow V_2 \cup \{v_i\}$
9:     end if
10: end for
```

Newman also proposed a method based on the spectral properties of the modularity matrix Q [37]. In this method, the eigenvector corresponding to the most positive eigenvalue of the modularity matrix is first found and the network is divided into two groups according to the signs of the elements of this vector. Spectral bisection provides two partitions and can be used to find a k-way partition of a graph when executed recursively. Spectral clustering however, is more general than spectral bisection and finds the clusters in a graph directly. This method is mainly designed to cluster n data points $x_1, \ldots, x_n$ but can be adapted for graphs as it builds a similarity matrix S which have entries s_{ij} showing how similar two data points x_i and x_j are. The spectral properties of this matrix are then investigated to find clusters of data points and the normalized Laplacian matrix, $L = I - D^{-1/2}SD^{-1/2}$ can then be constructed. A spectral clustering algorithm consist of the following steps [14]:

1. Construct similarity matrix S for n data points.
2. Compute the normalized Laplacian matrix of S.
3. Compute the first k eigenvectors of L and form the matrix V with columns as these eigenvectors.
4. Compute the normalized matrix M of V.
5. Use k-means algorithm to cluster n rows of M into k partitions.

The k-means algorithm is a widely used method to cluster data points as we reviewed in Sect. 7.3. The initial centers $c_1, \ldots, c_k$ can be chosen at random initially and the distance of data points to these centers are calculated and each data point is assigned to the cluster that it is closest. The spectral clustering algorithm described requires significant computation power and memory space for matrix operations and also to run the k-means algorithm due to the sizes of matrices involved. A simple approach would involve computation of the similarity matrix S using row partitioning. Finding eigenvectors can also be parallelized using parallel eigensolvers and the final step of using the k-means algorithm can also be parallelized [30]. A distributed algorithm based on the described parallel operations was proposed by Chen et al. and they experimented this algorithm with two large data sets using MPI and concluded it is scalable and provides significant speedups [14].

11.7 Chapter Notes

Graph clustering is a well-studied topic in computer science and there are numerous algorithms for this purpose. Our focus in this chapter was the classification and revision of fundamental sequential and distributed clustering algorithms in biological networks. We have also proposed two new distributed algorithms which can be implemented conveniently using a distributed programming environment such as MPI.

Two types of hierarchical clustering algorithms, MST-based and edge betweenness-based algorithms have found applications in clustering of biological networks more than other algorithms. Finding the MST of a graph using Boruvka's algorithm can be parallelized conveniently due to its nature. A different approach is taken in the CLUMP algorithm where the graph is partitioned into a number of subgraphs and bipartite graphs are formed. The MSTs for the partitions and the bipartite graphs are formed in parallel and then merged to find the MST of the whole graph. The edge-betweenness algorithm removes the edge with the highest betweenness value from the graph at each iteration to divide it into clusters. The algorithm proposed by Yang and Lonardi partitions the graph into processors which find pair dependencies by running BFS in parallel on their local partitions.

Density-based clustering algorithms aim to discover dense regions of a graph as these areas are potential clusters. Cliques are one example of such dense regions, however, clique-like structures such as k-cliques, k-cores, k-plexes, and k-clubs are more frequently found in biological networks than full cliques due to the erroneous measurements and dynamicity in such environments. Out of these structures, only k-cores of a graph can be found in linear time and therefore our focus was on sequential and distributed algorithms for k-core decomposition of graphs. The MCODE algorithm which uses a combination of clustering coefficient parameter and the k-core concept is successfully used in various PPI networks. There is not a reported parallel/distributed version of MCODE algorithm and this may be a potential research area as there are possibilities of independent operations such as finding weights of vertices. The k-core based algorithms are widely used to discover clusters in complex networks such as the biological networks and the Internet, and we reviewed a distributed k-core algorithm. The modularity concept provides a direct evaluation of the quality of clusters obtained and has formed the basis of various clustering algorithms used in social and biological networks. We proposed as simple distributed modularity-based algorithm that can be used for any complex network including biological networks.

Flow-based algorithms consider the flows in the networks and assume flows gather in clusters. An example algorithm with favorable performance that has been experimented in PPI networks is the MCL algorithm which we reviewed in sequential form and proposed its distributed version which can be implemented easily. Lastly, we described spectral bisection and clustering based on the Laplacian matrix of a graph and showed ways to implement distributed spectral clustering.

In summary, we can state most of these algorithms perform well in biological networks. However, each have merits and demerits and complexities of a number of

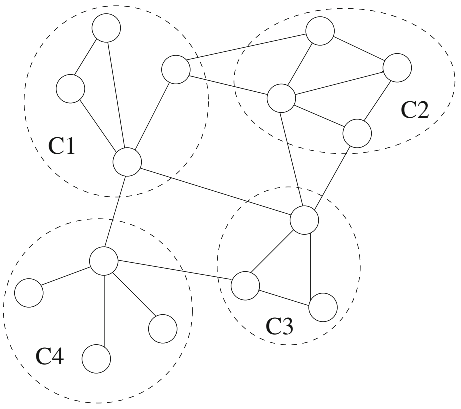

Fig. 11.10 Example graph for Exercise 1

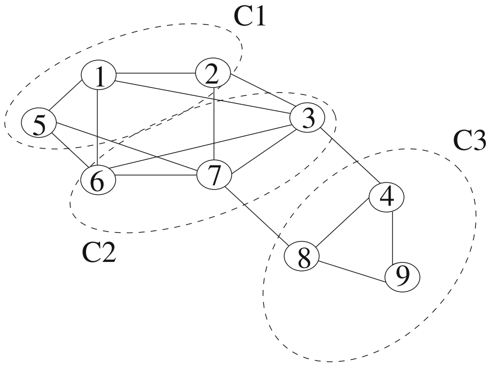

Fig. 11.11 Example graph for Exercise 2

them have only been determined experimentally. Distributed algorithms for this task are very rare and this may be a potential research area for researchers in this field.

Exercises

1. Find the intra-cluster and inter-cluster densities of the graph of Fig. 11.10. Do these values indicate good clustering? What can be done to improve this clustering?
2. Work out the modularity value in the example graph of Fig. 11.11 based on three clusters C_1, C_2, and C_3 and determine which merge operation is to be done to improve modularity. We need to check combining each cluster pair and decide on the pair that improves modularity by the largest amount.
3. Show the output of the BFS-based graph partitioning algorithm on the example graph of Fig. 11.12 in the first iteration, with the root vertex s. Then, partition the resulting graphs again to obtain four partitions and validate the partitions in terms of the number of vertices in each partition and the size of the minimum edge cut between them.

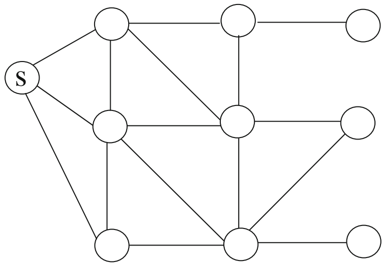
Fig. 11.12 Example graph for Exercise 3

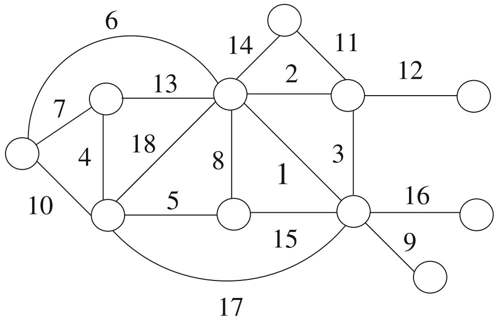
Fig. 11.13 Example graph for Exercise 4

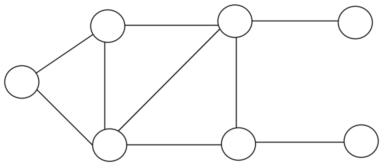
Fig. 11.14 Example graph for Exercise 5

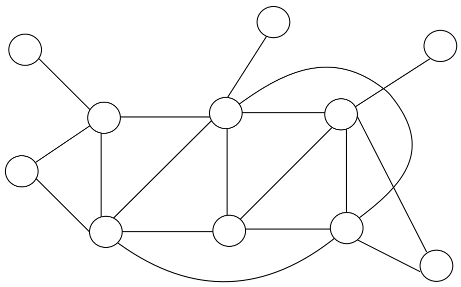
Fig. 11.15 Example graph for Exercise 6

4. Work out the MST of the weighted graph of Fig. 11.13 using Boruvka's algorithm. In the second step, partition the graph into two processor p_1 and p_2 and provide a distributed algorithm in which p_1 and p_2 will form the MSTs in parallel. Show also the implementation of the distributed Boruvka algorithm in this sample graph.
5. Find the edge betweenness values for all edges in the graph of Fig. 11.14 and partition this graph into 3 clusters using Newman's edge betweenness algorithm.

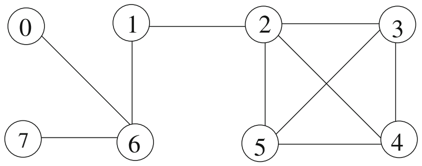

Fig. 11.16 Example graph for Exercise 7

6. For the example graph of Fig. 11.15, implement Batagelj and Zaversnik algorithm to find the coreness values of all vertices. Show all iterations of this algorithm and compose the k-cores for this graph in the final step.
7. Work out the two iterations of Markov Clustering Algorithm (MCL) in the example graph of Fig. 11.16. Is there any display of the clustering structure?

References

1. Altaf-Ul-Amin Md et al (2003) Prediction of protein functions based on kcores of protein-protein interaction networks and amino acid sequences. Genome Inf 14:498–499
2. Alvarez-Hamelin JI, DallAsta L, Barrat A, Vespignani A (2006) How the k-core decomposition helps in understanding the internet topology. In: ISMA workshop on the internet topology
3. Bader GD, Hogue CWV (2003) An automated method for finding molecular complexes in large protein interaction networks. BMC Bioinform 4(2)
4. Batagelj V, Zaversnik M (2003) An O(m) algorithm for cores decomposition of networks. CoRR (Computing Research Repository), arXiv:0310049
5. Boruvka O (1926) About a certain minimal problem. Prce mor. prrodoved. spol. v Brne III (in Czech, German summary) 3:37–58
6. Blaar H, Karnstedt M, Lange T, Winter R (2005) Possibilities to solve the clique problem by thread parallelism using task pools. In: Proceedings of the 19th IEEE international parallel and distributed processing symposium (IPDPS05)—Workshop 5—Volume 06 in Germany
7. Blondel VD, Guillaume J-L, Lambiotte R, Lefebvre E (2008) Fast unfolding of communities in large networks. J Stat Mech: Theory Exp (10):P10008
8. Brohee S, van Helden J (2006) Evaluation of clustering algorithms for protein-protein interaction networks. BMC Bioinform 7:488. doi:10.1186/1471-2105-7-488
9. Bron C, Kerbosch J (1973) Algorithm 457: finding all cliques of an undirected graph. Commun ACM 16:575–577
10. Buluc A, Madduri K (2011) Parallel breadth-first search on distributed memory systems. CoRR arXiv:1104.4518
11. Bustamam A, Sehgal MS, Hamilton N, Wong S, Ragan MA, Burrage K (2009) An efficient parallel implementation of Markov clustering algorithm for large-scale protein-protein interaction networks that uses MPI. In: Proceedings of the fifth IMT-GT international conference mathematics, statistics, and their applications (ICMSA), pp 94–101
12. Bustamam A, Burrage K, Hamilton NA (2012) Fast parallel Markov clustering in bioinformatics using massively parallel computing on GPU with CUDA and ELLPACK-R sparse format. IEEE/ACM Trans Comp Biol Bioinform 9(3):679–691
13. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms, 3rd edn. The MIT Press
14. Chen W-Y, Song Y, Bai H, Lin C-J, Chang EY (2011) Parallel spectral clustering in distributed systems. IEEE Trans Pattern Anal Mach Intell 33(3):568–586

15. Cheng J, Ke Y, Chu S, Ozsu MT (2011) Efficient core decomposition in massive networks. In: ICDE'11 proceedings of the 2011 IEEE 27th international conference data engineering, pp 51–62
16. Clauset A, Newman ME, Moore C (2004) Finding community structure in very large networks. Phys Rev E 70(6):066111
17. Du Z, Lin F (2005) A novel approach for hierarchical clustering. Parallel Comput 31(5):523–527
18. Dongen SV (2000) Graph clustering by flow simulation. PhD Thesis, University of Utrecht, The Netherlands
19. Elsnern U (1997) Graph partitioning, a survey. Technical report, Technische Universitat Chemnitz
20. Erciyes K (2014) Complex networks: an algorithmic perspective. CRC Press, Taylor and Francis. SBN-13: 978-1466571662, ISBN-10: 1466571667, Chap. 8
21. Fiedler M (1989) Laplacian of graphs and algebraic connectivity. Comb Graph Theory 25:57–70
22. Fortunato S (2010) Community detection in graphs. Phys Rep 486(3):75–174
23. Garey MR, Johnson DS (1978) Computers and intractability: a guide to the theory ofNP-completeness. Freeman
24. Girvan M, Newman MEJ (2002) Community structure in social and biological networks. Proc Natl Acad Sci USA 99:7821–7826
25. Gehweiler J, Meyerhenke H (2010) A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In: Proceedings of the 7th high-performance grid computing workshop (HGCW10) in conjunction with 24th international parallel and distributed processing symposium (IPDPS10). IEEE Computer Society
26. Hartuv E, Shamir R (2000) A clustering algorithm based on graph connectivity. Inf Process Lett 76(4):175–181
27. Jaber K, Rashid NA, Abdullah R (2009) The parallel maximal cliques algorithm for protein sequence clustering. Am J Appl Sci 6:1368–1372
28. Johnson SC (1967) Hierarchical clustering schemes. Psychometrika 2:241–254
29. Kernighan BW, Lin S (1970) An efficient heuristic procedure for partitioning graphs. Bell Syst Tech J 49(2):291–307
30. Kraj P, Sharma A, Garge N, Podolsky R, Richard A, Mcindoe RA (2008) ParaKMeans: implementation of a parallelized K-means algorithm suitable for general laboratory use. BMC Bioinform 9:200
31. LaSalle D, Karypis G (2015) Multi-threaded modularity based graph clustering using the multilevel paradigm. J Parallel Distrib Comput 76:66–80
32. Mohseni-Zadeh S, Brezelec P, Risler JL (2004) Cluster-C, an algorithm for the large-scale clustering of protein sequences based on the extraction of maximal cliques. Comput Biol Chem 28:211–218
33. Montresor A, Pellegrini FD, Miorandi D (2013) Distributed k-Core decomposition. IEEE Trans Parallel Distrib Syst 24(2):288–300
34. Murtagh F (2002) Clustering in massive data sets. Handbook of massive data sets, pp 501–543
35. Newman MEJ (2004) Fast algorithm for detecting community structure in networks. Phys Rev E 69:066133
36. Newman MEJ, Girvan M (2004) Finding and evaluating community structure in networks. Phys Rev E 69:026113
37. Newman MEJ (2006) Finding community structure in networks using the eigenvectors of matrices. Phys Rev E 74:036104
38. Olman V, Mao F, Wu H, Xu Y (2009) Parallel clustering algorithm for large data sets with applications in bioinformatics. IEEE/ACM Trans Comput Biol Bioinform 6:344–352
39. Riedy J, Bader DA, Meyerhenke H (2012) Scalable multi-threaded community detection in social networks. In: 2012 IEEE 26th international parallel and distributed processing symposium workshops and PhD forum (IPDPSW), IEEE, pp 1619–1628

40. Schaeffer SE (2007) Graph clustering. Comput Sci Rev 1:27–64
41. Schmidt MC, Samatova NF, Thomas K, Park B-H (2009) A scalable, parallel algorithm for maximal clique enumeration. J Parallel Distrib Comput 69:417–428
42. Satuluri V (2009) Scalable graph clustering using stochastic flows: applications to community discovery. In: Proceedings of the 15th ACM SIGKDD international conference on knowledge discovery and data mining, pp 737–746. Srinivasan Parthasarathy, Proceeding KDD'09
43. Sharan R, Shamir R (2000) CLICK: a clustering algorithm with applications to gene expression analysis. Proc Int Conf Intell Syst Mol Biol 8(307):16
44. Vlasblom J, Wodak SJ (200) Markov clustering versus affinity propagation for the partitioning of protein interaction graphs. BMC Bioinform 10:99
45. Xu X, Jager J, Kriegel H-P (1999) A fast parallel clustering algorithm for large spatial databases. Data Min Knowl Disc 3:263–290
46. Yang Q, Lonardi S (2007) A parallel edge-betweenness clustering tool for protein-protein interaction networks. Int J Data Min Bioinform (IJDMB) 1(3):241–247
47. Zadeh LA (1965) Fuzzy sets. Inf Control 8:338–353