On Vertex Cover Problems in Distributed Systems

Can Umut Ileri [1], Cemil Aybars Ural [1], Orhan Dagdeviren [1,*], Vedat Kavalci [1,2]

[1] Ege University
International Computer Institute
Izmir, Turkey

[2] Izmir University
School for Vocational Higher Education
Izmir, Turkey

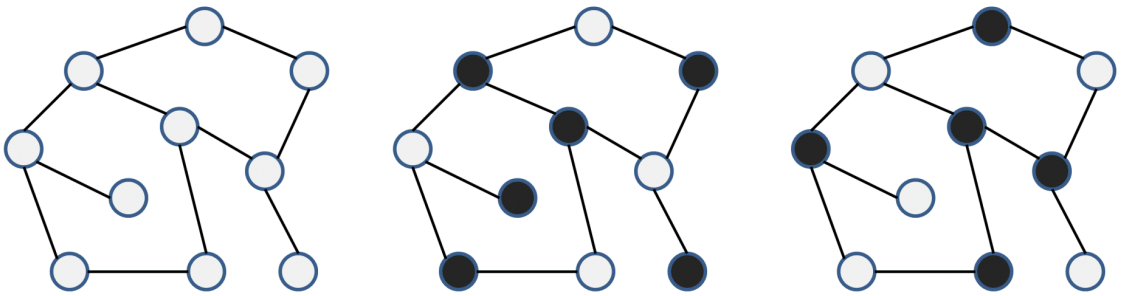
*Corresponding Author Contact: orhandagdeviren@gmail.com

Abstract An undirected graph can be represented by *G*(*V*,*E*) where V is the set of vertices and E is the set of edges connecting vertices. The problem of finding a vertex cover (*VC*) is to identify a set of vertices *VC* such that at least one endpoint of every edge in E is incident to a vertex *V* in *VC*. Vertex cover is a very important graph theoretical structure for various types of communication networks such as wireless sensor networks, since *VC* can be used for link monitoring, clustering, backbone formation and data aggregation management. In this chapter, we will define vertex cover and related problems with their applications on communication networks and we will survey some important distributed algorithms on this research area.

Keywords Vertex Cover, Weighted Vertex Cover, Connected Vertex Cover, Distributed Systems, Greedy Algorithm, Distributed Algorithm, Self-Stabilization, Wireless Sensor Networks, Link Monitoring, Clustering, Backbone Formation.

1 The Vertex Cover Problems

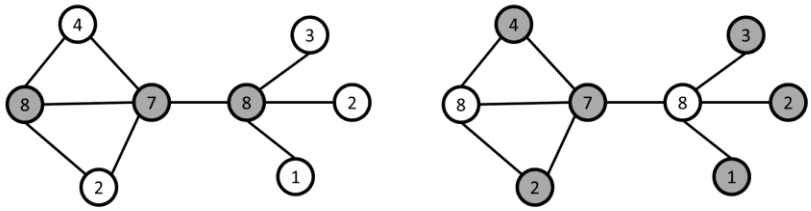
Given a graph $G(V, E)$ where V the set of vertices and E is the set of edges between vertices, the problem to find a set of vertices $VC \subseteq V$ such that for any edge $\{u, v\} \in E$, at least one of u and v is in VC is called vertex cover problem. V itself is a vertex cover and it may have numerous subsets satisfying the vertex cover conditions. Among all possible vertex covers of a given graph, the one(s) that have the minimum cardinality are called the *minimum vertex cover*. A *minimal vertex cover* is a vertex cover VC whose cardinality cannot be decreased, in other words, excluding any vertex from VC would break the vertex cover conditions. Note that a minimal vertex cover is not required to have the minimum possible cardinality. Figure 1 illustrates such a case.



(a) A given graph G(V,E) (b) A *minimal* vertex cover (c) The *minimum* vertex cover

Figure 1: Minimal and Minimum Vertex Covers: Dark vertices are the ones in the VC solution set. Removal of any vertex from the VC solution in (b) breaks the VC conditions, then it is minimal. However its cardinality is 6, while there is a possible solution having less number of vertices (5) as shown in (c).

Several objective functions are used for vertex cover problem. In *minimum cardinality vertex cover problem*, the optimum vertex cover VC^* is the one (or possible ones) that has (or have) the minimum cardinality among all possible vertex covers. In a weighted graph where a weight w_v is assigned to each vertex $v \in V$, the objective could be to find a vertex cover VC^* in which the total weights of the edges are the minimum among all possible vertex covers of the graph. This problem is called minimum weighted vertex cover problem. Note that the vertex cover having the minimum weight may not be the one having the minimum cardinality. Figure 2 illustrates such a case.



(a) Minimum cardinality vertex cover (b) Minimum weighted vertex cover

Figure 2: Minimal cardinality vs. Minimum Weight: Dark vertices are the ones in the solution set. Numbers on the vertices are the weights. In (a), solution size is 3 (the minimum) and total weight is 23; while in (b), total weight is only 19 (minimum), although the cardinality is doubled.

Another variation of the problem is the *connected vertex cover problem* in which the graph induced by the vertex cover is connected. Vertex cover can be used for link monitoring to provide secure communication in wireless ad hoc networks. To achieve this operation, nodes in VC set are trusted and can monitor the transmissions between nodes. Since every communication link will be under the coverage of one or more nodes. Other important applications are data aggregation management, backbone and cluster formations, hub or router location designation and traffic control on the information flow.

2 Algorithms for Distributed Vertex Cover Problems

2.1 Distributed Minimum Cardinality Vertex Cover Problem

2.1.1 Distributed Greedy Minimum Cardinality Vertex Cover Algorithm (MinCVC-Greedy)

This algorithm aims to attain a vertex cover set on a network with minimum cardinality. The idea is to find the vertices having the highest degree within their neighborhood and place them in the VC. This algorithm is inspired from the one by Parnas and Ron (2007). All vertices compare their degree with that of their neighbors, and if it is higher, they add themselves to the VC and delete themselves from the candidates' group. Coordination and collaboration is conveyed by means of a graph-wide round looping. This goes on until the termination condition is satisfied. Briefly, it depends on the following assumptions:

- Each node is aware of the maximum degree (Δ) of the graph.
- Each node is aware of its neighbors.
- Each node has a guaranteed communication with its neighbors via messaging and by acknowledgements.
- A global (graph-wide) round count is known by each node through messaging.

Pseudocode of the Distributed Greedy Minimum Cardinality Vertex Algorithm is given in Algorithm 1. Figure 3 shows an example VC produced by this algorithm.

Algorithm 1: Distributed Greedy Minimum Cardinality Vertex Cover Algorithm

```
/* At each node */
Message types: DROP_ME
R←0 (rounds' counter is set to zero)
At the start of a round
  R ← R+1
  if R > log2(Δ) then terminate
  if degree > (Δ / (2R) then
    enter the VC
    send DROP_ME to all neighbors

When a message is received:
  if DROP_ME message is received,
    delete the appropriate node from the adjacency list
    degree ← degree-1
```

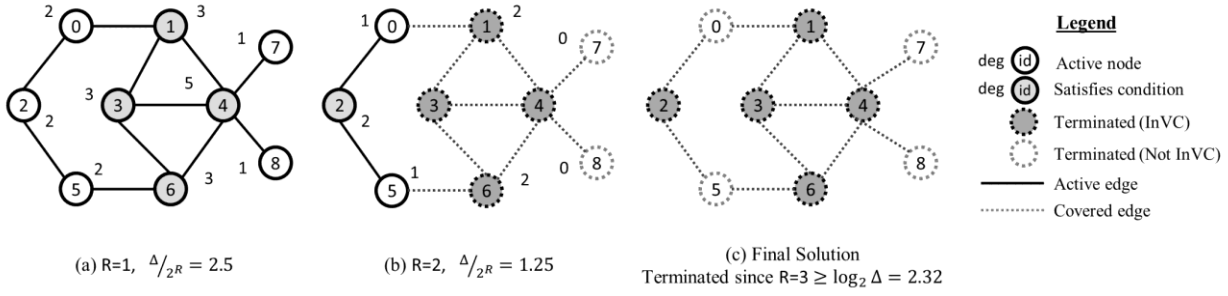


Figure 3: VC Produced by the Distributed Greedy Minimum Cardinality Vertex Cover Algorithm

Theorem 1. *The time complexity of the Distributed Greedy Minimum Cardinality Vertex Cover Algorithm is O($log_2(\Delta)$).*

Proof. The worst case happens when at each round only one node announces itself to be in the VC. In such a case, the maximum number of rounds is reached. Since each node which enters the VC sends a drop message only to its neighbors, the messaging time is *O(1)*. The maximum number of rounds possible for this algorithm is $R=log_2(\Delta)$, where Δ is the maximum degree of the graph. Hence the time complexity becomes $O(log_2(\Delta))$. □

Theorem 2. *The bit complexity of the Distributed Greedy Minimum Cardinality Vertex Cover Algorithm is* $O(n\Delta\, log_2(n))$.

Proof. At the worst case, a node can send a message to each of its neighbors. Hence, in the worst case message complexity is $O(\Delta)$ per node. Since each message is made up of $O(log_2(n))$ bits, the bit complexity becomes $O(n^2\Delta\, log_2(n))$. □

Theorem 3. *The space complexity of the Distributed Greedy Minimum Cardinality Vertex Cover Algorithm is* $O(\Delta log_2(n)))$ *per node.*

Proof. Each node has to use memory for keeping its own neighbor list and the maximum degree Δ present in the graph. Thus, the space complexity of this algorithm becomes $O(\Delta log_2(n))$.□

The VC approximation ratio of this algorithm is $(2.log_2(\Delta) +1)$ (Parnas & Ron, 2007). Therefore the approximation ratio in the worst case is of $O(log_2(\Delta))$.

2.1.2 Distributed Vertex Cover Algorithm via Greedy Matching (MinCVC-Match)

This algorithm utilizes graph matching to construct a VC. On a given a graph of $G(V, E)$, finding a set of edges $M \in E$ such that no two edges in M are incident to the same vertex is called the matching problem. The distributed greedy weighted matching protocol by Hoepman(2004) greedily and asynchronously selects the edges with the locally highest weights and the end vertices of those edges (matched vertices) drops themselves from the graph. This algorithm is adapted to the vertex cover problem by adding all of the matched vertices to the vertex cover set, unless they are matched in the very first round, dropping all adjacent edges to the matched vertices from the graph, and then repeating the procedure until all edges are covered. Edge weights are neglected in order to yield a distributed vertex cover algorithm to be applied on unweighted graphs. This goes on until the termination condition is satisfied. This algorithm depends on the same assumptions with the Distributed Greedy Minimum Cardinality Vertex Cover Algorithm's assumptions except each node may not needed to aware of Δ. Pseudocode and an example VC produced by this algorithm are given in Algorithm 2 and Figure 4 respectively.

Algorithm 2: Distributed Vertex Cover Algorithm via Greedy Matching Algorithm

```
/* At each node */
Message types: PROPOSE, DROPME
cand ← null, inVC ← false, status ← ACTIVE
//Algorithm starts
Odd-numbered round
  //Select candidate
  learn degrees of neighbors and set d(j)'s accordingly
  C ← select neighbors with the lowest degree
  cand ← select the node having the highest id in C
  //Matching
  send PROPOSE to cand
  do until the end of the round
    receive PROPOSE message from j
    if j = cand // j and me proposed to each other
      if round = 1
        if (d(me) > d(j)) OR {d(me) = d(j) AND (id(me) > id(j))}
          inVC ← true
```

```
          else
             inVC ← false
        else
          inVC ← true
        status ← INACTIVE
Even-numbered round
   if status = INACTIVE
      send DROPME message to all neighbors
      terminate
   do until the end of the round
      receive DROPME message from j
      remove j from neighborhood
      if there is no active neighbor
        status ← INACTIVE
        terminate
```

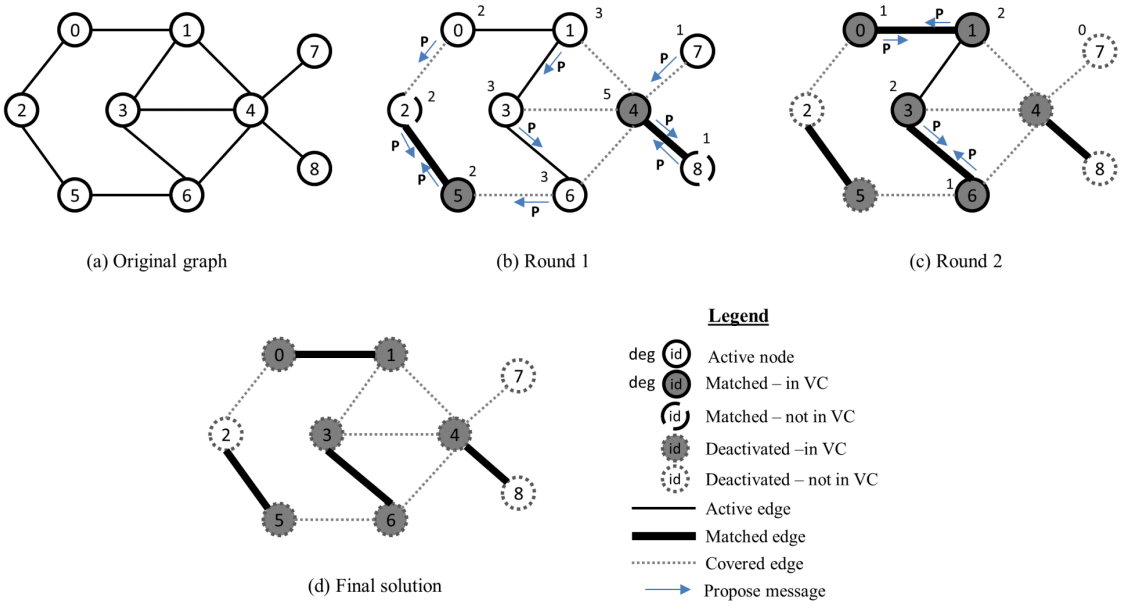


Figure 4: VC Produced by the Distributed Vertex Cover Algorithm via Greedy Matching

Theorem 4. *The time complexity of the MinCVC-Match is O(n).*

Proof. At each two rounds, at least one edge is guaranteed to match, which means that 2 nodes are deactivated. Thus, algorithm is expected to terminate in *n* rounds. □

Theorem 5. *The bit complexity of the MinCVC-Match is O(|E| log₂(n)).*

Proof. Each node sends at most one message having size of $log_2(n)$ in each round. Network of *n* nodes in *n* rounds send $O(n^2)$ messages. The bit complexity of the algorithm is $O(n^2 log_2(n))$. □

Theorem 6. *The space complexity of the MinCVC-Match is O(Δlog₂(n)) per node.*

Proof. Each node has to use memory for keeping its neighbors' original degrees and changing degrees of them within the rounds. This means a memory space two times the message count. And a fixed size

memory space c is used for keeping the ID of the matched node and another memory space is used for storing the round info. Thus, space complexity is $O(\Delta log_2(n))$. □

The VC approximation ratio of this algorithm varies between 1 and 1.6 according to the empirical results by Kavalci et. al., (2014). The matching algorithm used as a tool in Distributed Vertex Cover Algorithm via Greedy Matching is stated to have a VC approximation ratio of 2 (Hoepman, 2004).

2.1.3 Distributed Vertex Cover Algorithm via Bipartite Matching (MinCVC-BipMM)

As presented in the previous section, once a maximal matching is given, it is easy to obtain a vertex cover solution, and more importantly, the time complexity of the vertex cover problem is determined by the maximal matching algorithm.

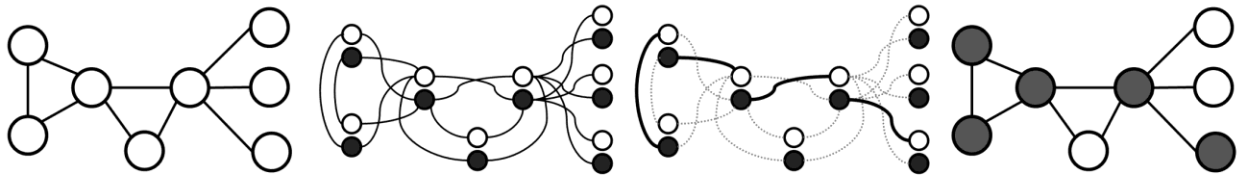
Besides, it is important to mention the fact that finding a maximal matching is relatively easier in 2-colored graphs (i.e. the graph is bipartite and each node is aware of its group) than it is in general graphs. If we let one group of nodes to make requests and let the other group only to respond to this requests, we can find a maximal matching in at most Δ steps, where Δ is the highest degree in the network (marriage problem).

Hanckowiak et. al. (1998) proposed a distributed algorithm to exploit this fact in order to find a maximal matching in general graphs in $O(\Delta)$ time. Polishchuk & Suomela (2009) used this approach in a different manner to find a maximal matching in general graphs in $O(\Delta)$ time and then to use this matching to obtain a vertex cover solution which is at most 3 times the optimum VC solution. Their algorithm, which is presented in this section, implements this reduction in the following way:

Given a general graph $G(V, E)$, one can obtain a bipartite graph $\tilde{G}(\widetilde{V_1}, \widetilde{V_2}, \tilde{E})$ such that:

- for each node $v \in V$, there will be $\tilde{v}_1 \in \tilde{V}_1$ and $\tilde{v}_2 \in \tilde{V}_2$, and
- for each edge $(u, v) \in E$, there will be two edges: $(\widetilde{u_1}, \widetilde{v_2}) \in \tilde{E}$ and $(\widetilde{u_2}, \widetilde{v_1}) \in \tilde{E}$

In other words, for each node in the general graph, we will have two copies (namely black copy and white copy) such that if two nodes u and v are connected in the general graph, then white copy of u will be connected to the black copy of v, and vice versa. An illustration of this reduction is in Figure 5.



(a) A general graph G (b) Bigraph $\tilde{G}$ obtained from G (c) A maximal matching on $\tilde{G}$ (d) Corresponding vertex cover on G

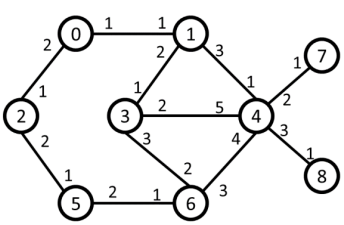
Figure 5: Example illustration of reduction from a general graph to a bipartite graph and finding a vertex cover solution by solving maximal matching problem on the bipartite graph.

After finding a maximal matching on the bipartite graph, vertex cover set can be obtained by a single decision: If any of two copies of a node in general graph is in the matching set of the bipartite graph, the node is included in the vertex set.

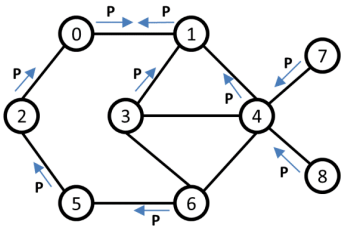
Procedure is shown formally in Algorithm 3. A node with degree d is assumed to have ports numbered from 1 to d. Each node has two pointers (black pointer and white pointer) which can point to a neighbor of the node. If a black pointer of a node u points to node v, it means in a sense that the black copy of u is matched with the white copy of v.

Algorithm 3: Distributed Vertex Cover Algorithm via Bipartite Matching

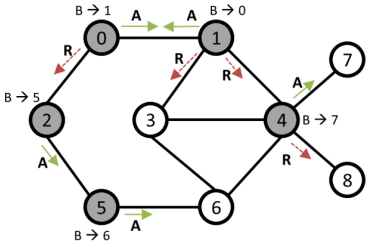
```
/* At each node */
Message types: PROPOSE, ACCEPT, REJECT
white ← null, black ← null, i = 0, inVC ← false
Odd-numbered round //Setting white pointers
  if (white is null AND 1 ≤ i ≤ d ) // d is the degree of the node
    receive message from port I   // not blocking receive
    if message is ACCEPT
      white ← i
      inVC ← true
  if (white is null AND i ≤ d)
    i = i + 1
    send PROPOSE to the port i


Even-numbered round //Setting black pointers
  //not blocking receive
  receive PROPOSE from any neighbors and add senders to the set P
  if black is null
    select p ∈ P incoming from the lowest-numbered port
    send ACCEPT to p
    send REJECT to all in P − p
    inVC ← true
  else
    send REJECT to all in P
```



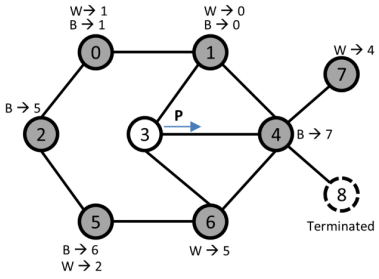
(a) Original graph and the assigned port numbers



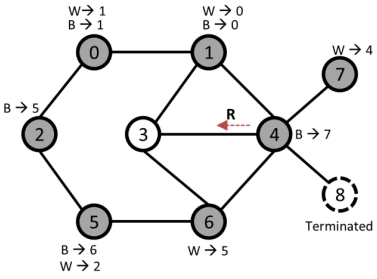
(b) Round 1: Propose messages



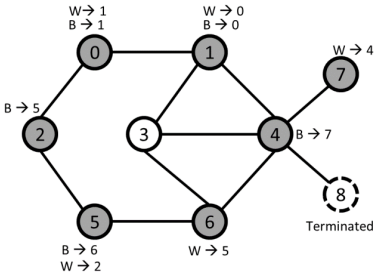
(c) Round 2: Accept and Reject messages. Black pointers are set.



(d) Round 3: Propose messages. White pointers are set.



(e) Round 4: Reject message.



(f) Round 7: Final Solution

Figure 6: Example execution of MinCVC-BipMM Algorithm

An example execution of MinCVC-BipMM is illustrated in Figure 6. Original graph is shown in (a). Numbers at on the outgoing edge of each node are the related port numbers. In the first round (b), each node sends PROPOSE message through their ports numbered 1. In the second round, each node having received a PROPOSE message selects the incoming from the lowest numbered port, sets their black pointers accordingly, enters the solution set and sends as ACCEPT message as a reply. All other proposers are responded with a REJECT message. In round 3 (in (d)), the nodes, which have received an ACCEPT message, set their white pointers accordingly and enter the solution set. Final solution is shown in (e).

Theorem 7. *MinCVC-BipMM Algorithm finds a vertex cover solution in* $2\Delta + 1$ *rounds.*

Proof. Each node, in order to set its white pointer, sends a PROPOSE message through the first port at the first round and gets its response at the next odd-numbered round. Thus, at the worst case, the node having the highest degree Δ will decide the value of its white pointer at the $(2\Delta + 1)^{th}$ round. Also, at the worst case, the node having the highest degree will decide the value of its black pointer in the $(2\Delta)^{th}$ round. □

Theorem 8. *During the execution of MinCVC-BipMM Algorithm, the transmitted bit count through the links is* $O(|E|log_2(n))$.

Proof. Each node may send at most one message through and receive at most one message from each of its port. In other words each port deals with at most two messages. Total number of ports on the graph is $2|E|$. Total number of messages is $4|E|$. The messages can be at most $log_2(n)$ bits. Thus, bit complexity of MinCVC-BipMM Algorithm is $O(|E|log_2(n))$. □

Theorem 9. *Each node requires* $O(\Delta log_2(n))$ *bits of memory space in order to execute MinCVC-BipMM Algorithm.*

Proof. Each node is assumed to have the set of ids of its neighbors, which require at most $\Delta log_2(n)$ bits. □

2.1.4 Distributed Vertex Cover Algorithm via Breadth First Search Tree (MinCVC-BFST)

This algorithm builds a Breadth First Search Tree (BFST) and constructs a VC by using this BFST as the infrastructure. BFST formation is very important for wireless ad hoc networks since BFST provides a routing backbone. In a BFST, each vertex is associated with a level value which is its level-wise distance from the root (or sink) node (Kavalci et. al., 2014).

In the following algorithm each vertex advertises its level information to its neighbors. Afterwards, each of them decides whether it should participate within VC or not. If the level value of a vertex is even, then it will be directly in the set. If it is odd and that vertex has a neighbor which has also an odd level value, then the vertex which has the larger ID number in-between these two decides whether it is in the VC or not. This algorithm depends on the same assumptions with the Distributed Vertex Cover Algorithm via Greedy Matching and a root node should be selected before the execution of the algorithm.

Pseudocode and an example VC produced by this algorithm and are given in Algorithm 4 and Figure 7.

Algorithm 4: Distributed Vertex Cover Algorithm via Breadth First Search Tree Algorithm

```
run a Distributed BFST algorithm and find the level
if level is even then
   inVC ← true
else if level is odd and the node has an odd level neighbor with a
smaller id
   inVC ← true
```

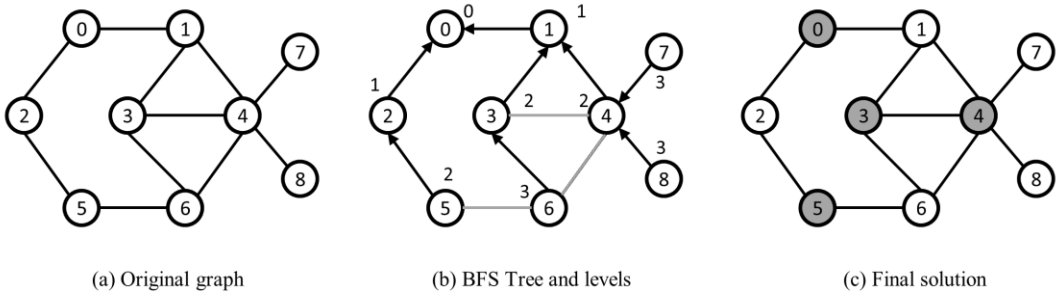


Figure 7: VC Produced by the Distributed Vertex Cover Algorithm via Breadth First Search Tree

Theorem 10. *The time complexity of the MinCVC-BFST Algorithm is O(D), where D is the diameter of graph.*

Proof. Construction of a BST takes *O(D)* time. So that the time complexity of the algorithm is *O(D)*. □

Theorem 11. *The bit complexity of the MinCVC-BFST Algorithm is* $O(n^3 log_2(n))$.

Proof. At the worst case $O(n^3)$ messages are transmitted to construct a BFST. The length of each message is $O(log_2(n))$ bits. Thus the bit complexity is $O(n^3 log_2(n))$ bits. □

Theorem 12. *The space complexity of the MinCVC-BFST Algorithm is* $O(\Delta log_2(n)))$ *per node.*

Proof. In BFST construction, each node has to keep its parent's information, hence use a constant memory space c_1. And, during the operation, every node has to keep the tree level information of its neighbor nodes. Also, it has to use a constant memory space c_2 for saving its own tree level information. Hence, $O(\Delta log_2(n)+c_1+c_2)$ memory is required per node. Overall space complexity is $O(\Delta log_2(n))$. □

The VC approximation ratio of this algorithm varies from 1 to 2, in most cases not exceeding 1.5, depending on the empirical data gathered from various trials on various graph topologies (Kavalci et. al., 2014).

2.2 Distributed Minimum Weighted Vertex Cover Problem

In this section, we present distributed algorithms aiming to approximate the minimum weighted vertex cover in a graph. We introduce four algorithms, the first of which is the basic greedy MinWVC algorithm. Next two algorithms are the distributed versions of two well-known sequential algorithms in the field: Clarkson's Algorithm (1983) and Bar-Yehuda&Even's Algorithm (1981). Finally, we present a distributed algorithm due to Grandoni et. al. (2008) which is based on the fact that a 2-approximation solution to the vertex cover problem can be computed via a solution of maximal matching.

2.2.1 Greedy Distributed Minimum Weighted Vertex Cover Algorithm (GreedyWVC)

This algorithm is similar to the greedy distributed minimum cardinality vertex cover algorithm. The only difference is that this algorithm favors the nodes having the minimum weight instead of the ones having the maximum degree.

Assuming all the nodes have the knowledge of the weights of their neighbors (this can be obtained in a single round), each node can easily determine if it is the locally lightest node. Once a node knows it is the locally lightest, it marks itself as being in the solution set VC, informs its neighbors by a COVER message and terminates. When a node receives a COVER message, it marks the sender as covered. Nodes

remove their neighbors from which they have received a COVER message. This procedure repeats until all neighbors are inactive as given in Algorithm 5.

Algorithm 5: Greedy Distributed Minimum Weighted Vertex Cover Algorithm

```
/* At each node */
Message types: COVER
// Initialization
send myweight to all my neighbors
receive their weights from all my neighbors
// Covering
while the node is not in the solution set OR it has at least one
uncovered edge, perform rounds as follows:
  if my weight is the lightest among my active neighbors
    inVC = TRUE // The node is in the solution set
    send COVER message to all active neighbors
  else
    do until the end of the round
      receive COVER message
      // edge between me and the sender is covered
      mark the sender as inactive
      remove the sender from my active neighborhood
```

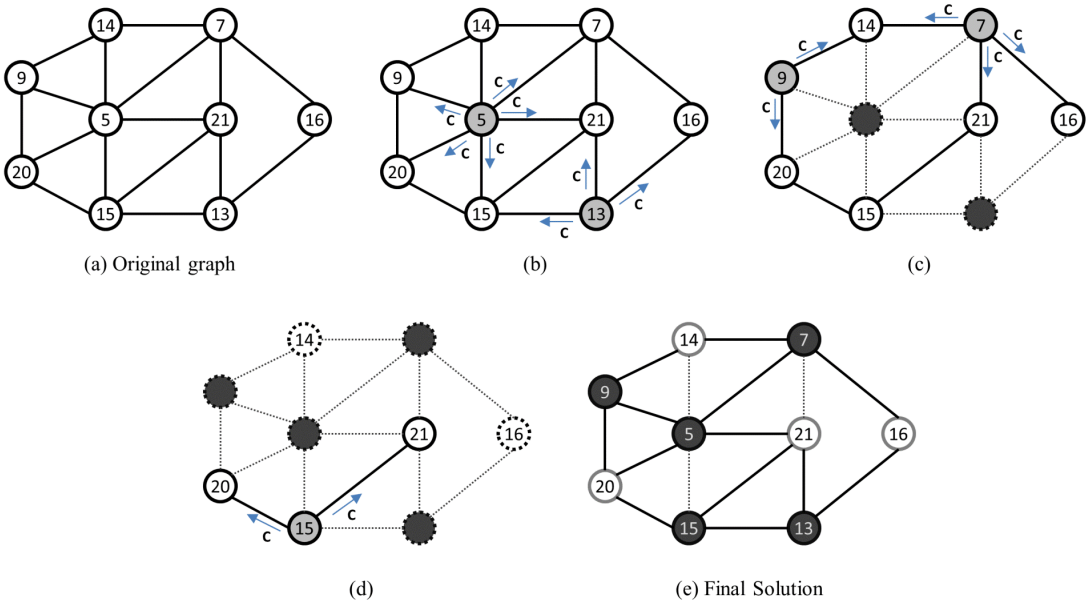
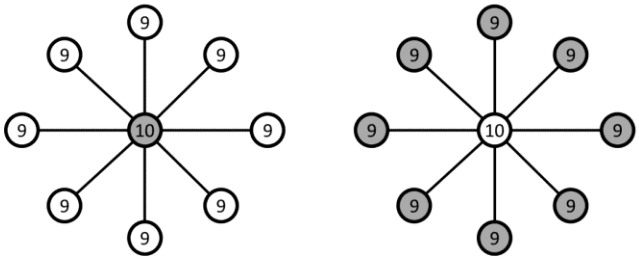


Figure 8: Sample execution of *GreedyWVC* Algorithm

An execution of GreedyWVC Algorithm on a graph with 9 nodes is shown in Figure 8. There are two locally-lowest weighted nodes, which are 5 and 13 (in (b)). They are included in VC and send COVER message to their neighbors. Deactivations of these nodes enable other nodes to be local leaders. Final solution to the problem is shown in Figure 8(e).

In this example, the greedy algorithm computes the minimum weighted vertex cover. However this algorithm does not guarantee a constant approximation to the optimal solution. For example, consider the star graph in Figure 9. Greedy algorithm for this example resulted with a solution which is almost *n* times the optimum.



(a) Optimum VC solution (b) Greedy Solution

Figure 9: Illustration of a sample case where the greedy algorithm misses trivial solution

Theorem 13. *$GreedyWVC$ Algorithm terminates in at most $O(n)$ rounds.*

Proof. In any round, the graph has a globally lightest active node and it is guaranteed to be included in the solution set in that round. □

Theorem 14. *Total number of bits transmitted through the network during the execution of $GreedyWVC$ is $O(n^2 log_2(n))$.*

Proof. In each round, each node sends exactly one message having size of $O(log_2(n))$ bits. In a single round, $O(nlog_2(n))$ bits are transmitted on the network. Assuming $O(n)$ rounds at the worst case, the bit complexity of the algorithm is $O(n^2 log_2(n))$. □

Theorem 15. *$GreedyWVC$ requires each node to have $O(\Delta(log_2(n) + log_2(w)))$ memory space and w is the maximum weight.*

Proof. Each node stores the identifier and weight of each of its neighbors, i.e. $2(log_2(n) + log_2(w))$ bits per neighbor. A node may have at most Δ neighbors. Total number of required space is $O(\Delta(log_2(n) + log_2(w)))$ bits per node. □

2.2.2 Distributed Version of Clarkson's Vertex Cover Algorithm

As we have seen above, the biggest reason why the greedy algorithm failed to perform well is that it did not take the degree of any vertex into consideration. Figure 9 shows a sample case where the greedy approach that ignores the degrees results in a "heavy" solution.

Clarkson's sequential 2-approximation weighted vertex cover algorithm (Clarkson, 1983) favors the nodes having higher degrees in forming the VC set, simply by using the weights in proportion to the degree of the node. It basically calculates, for each vertex *v*, the value $p(v) = \frac{W(v)}{D(v)}$ where W is the weight function and D is the degree function. The vertex *v* such that $p(v) \geq p(u)$ for all $u \in V$ and $(u, v) \in E$ is added to the solution set VC. For the next step of the algorithm, all of the edges incident to v are removed from the graph and weights of the neighbors of v are updated as follows, and the procedure repeats until all edges are covered:

$$\overline{W(u)} = W(u) - p(v) \text{ for all } u \in \Gamma(v)$$

This algorithm can be implemented in distributed settings. Each vertex can calculate its p value, since it knows its own weight and degree. All the nodes can get the knowledge of the p value of their neighbors in a single communication round. In Algorithm 6, we realize this information sharing with UPDATE messages. Then, as in the greedy distributed weighted vertex cover algorithm, each vertex can decide whether it has the locally lowest p value. If it does, it marks itself as being "in VC" and informs all their

neighbors about it by means of a COVER message. When a vertex receives a COVER message, it decreases its weight by the p value of the sender, decreases its degree by one and adjusts its own p value accordingly.

This procedure can be implemented as a synchronous distributed algorithm having odd- and even-numbered rounds, where the nodes share their p values in odd numbered rounds and make decisions on covering in the even-numbered rounds. This synchronous algorithm is shown in Algorithm 6.

Algorithm 6: Distributed Version of Clarkson's Weighted Vertex Cover Algorithm (MinWVC-Cla)

```
/* At each node */
Message types: COVER, UPDATE
while the node is not in the solution set OR it has at least one
uncovered edge, perform rounds as follows:
  In an odd numbered round: //Information Sharing
  P ← W/D
  send P to all my neighbors
  do until UPDATE (j, Pj) message is received from all active
neighbors
    receive UPDATE (j,Pj)
    update Pj
  In an even numbered round: // Covering
  if my P value is the lowest among my active neighbors
    inVC ← TRUE // I am in the solution set, i.e. I cover my edges
    send COVER message to all active neighbors
  do until the end of the round
    receive COVER message
    // edge between me and the sender is covered by the sender
    mark the sender as inactive
    W ← W - Psender
    D ← D - 1
    remove the sender from my active neighborhood
```

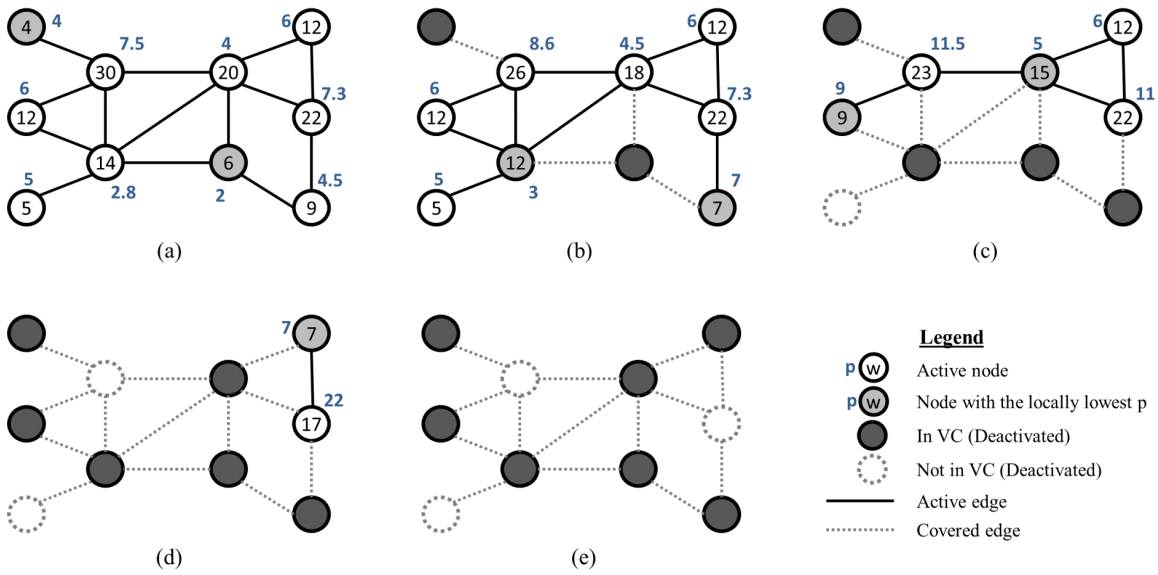


Figure 10: Sample execution of *MinWVC-Cla* Algorithm

On the graph given in Figure 10, each node firstly calculates its p values (blue numbers next to nodes). There are two nodes having the locally-lowest p values, namely 2 and 4. They are included in the solution set and inform their neighborhood. Observe that the neighbors of the locally-lowest nodes update their weights at the end of each round. When a node does not have any active neighbor, it terminates the algorithm (see the node at the bottom left in Figure 10(c)). Resulting vertex cover is shown in Figure 10(a).

Theorem 16. *$MinWVC\_Cla$ Algorithm terminates in at most $O(n)$ rounds.*

Proof. There is at least one node having the locally-lowest p value at any round, and it is guaranteed to be included in the solution set at the next round. Thus, at the worst case, the algorithm will terminate at the $(2n)^{th}$ round, where n is the node count. □

Theorem 17. *Total number of bits transmitted through the network during the execution of MinWVC-Cla is $O(n^2 log_2(n))$.*

Proof. In both odd and even numbered rounds, each node transmits a single message having at most $logn$ bits. Since there are most $2n$ rounds, a single node transmits $O(nlog_2(n))$ bits. Thus, n nodes transmit $O(n^2 log_2(n))$ bits. □

Theorem 18. *$MinWVC\_Cla$ requires each node to have $O(\Delta(log_2(n) + log_2(p)))$ memory space.*

Proof. Each node stores the identifier and p value of each of its neighbors, i.e. $2(log_2(n) + log_2(p))$ bits per neighbor. A node may have at most Δ neighbors. Total number of required space is $O(\Delta(log_2(n) + log_2(p)))$ bits. □

2.2.3 Distributed Version of Bar-Yehuda and Even's Algorithm

Bar-Yehuda and Even's sequential algorithm (Bar-Yehuda & Even, 1981) arbitrarily selects an edge (u, v) where $w_u \leq w_v$, and includes its incident node having the smaller weight *(u)* into the vertex cover solution set *V*. It then updates the weights of the incident node having the greater weight *(v)* by the following formula: $w_v \leftarrow w_v - w_u$. Node u and all edges incident to u are removed from the graph and the procedure iterates until all of the edges are covered.

This algorithm can be implemented as a synchronous distributed algorithm. As the sequential algorithm chooses an arbitrary edge at each step, it is required to decide which edges to consider first in the distributed algorithm. One way could be to rely on ids and to start with the nodes having the locally lowest or highest ids. In the implementation presented in this section, the weights of the nodes are used instead.

Algorithm 7 shows the implementation. The nodes having the locally heaviest weight make decisions. (Local identifiers can be used for symmetry breaking in case of equal weights.) They decide which edge to consider first (randomly or deterministically) and send a REQ message to the node at the other end of this edge. On the receiver side, a node may receive more than one REQ messages, since it may be connected to more than one locally-heaviest nodes. In order to wait for all request messages, a round is devoted for requests.

At the subsequent round, there are three types of nodes: (1) the ones having received one or more REQ messages, (2) the ones having sent a REQ message (locally-lowest nodes), and (3) the remaining nodes. The first group of nodes is included in VC since they are the lower-weighted part of the matching. Then, they set their own weights to zero, they choose a node k from the requesters, send an ACK message to k, NAK messages to other requesters and COVER messages to all other nodes. NAK and COVER mean that the new weight of the node is zero.

Nodes at the second group wait for a response for their REQ message. If the response is an ACK, they reduce their weight by the matched node's weight and they remove the matched node from the active neighborhood. If the response is NAK, even though the proposal is rejected, it is known that the requested node has accepted a proposal of another node, thus it is removed from the neighborhood. In any case, they send UPDATE messages to their active neighborhood in order to inform their new weights. All the other nodes wait for UPDATE or COVER messages and perform the related operations. Once a node is included in VC or all of its neighbors are deactivated, algorithm terminates. The sum of weights of nodes in the resulting solution set VC is at most twice the optimum WVC solution (Bar-Yehuda & Even, 1981).

Algorithm 7: Distributed Version of Bar-Yehuda and Even's WVC Algorithm (*MinWVC- BarEven*)

```
/* At each node */
Message types: REQ, ACK, NAK, COVER, UPDATE
while the node is not in VC or it has at least one active neighbor:
  Odd-numbered round: // Decision
    if my weight W is the highest among my active neighbors
      set cand ← j ∈ Γ(me) // select one of my neighbors as candidate
      send REQ to cand
    else
      till the end of the round
        receive REQ message from j
        add j to R // R is the set of requesters
  Even-numbered round: // Covering and Synchronization
    if R is not empty // I have received at least one REQ at the prev.
round
      W ← 0 //Update my weight
      inVC ← TRUE
      choose one k from R
      send ACK to k
      send NAK to R − {k} //reject the other requestors
      send COVER(W(me)) to Γ(me) − R // Inform neighbors
    else if cand is not null // I sent a REQ at the previous round
      do until the end of the round
        receive message from j
```

```
        if received message is an ACK
          W(me) ← W(me) – W(j)
          remove j from the active neighborhood
          send UPDATE(W(me)) to all active neighboors
        else if received message is a NAK or COVER

          remove j from the active neighborhood
    else
      do until the end of the round
        receive message //either COVER or UPDATE message
        if message is UPDATE
          update W(j) ← message.w
        else
          remove j from active neighborhood
```

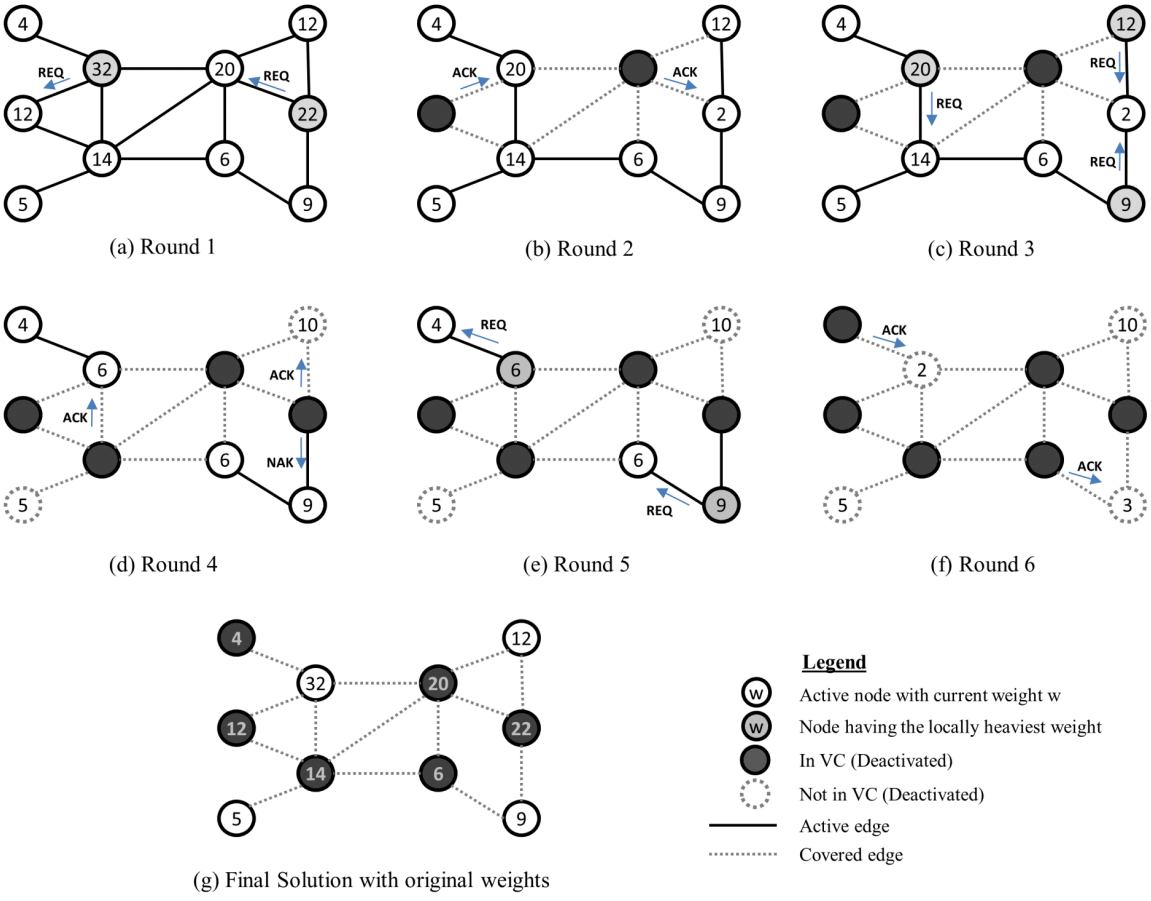


Figure 11: Sample execution of MinWVC-BarEven Algorithm

Figure 11 illustrates a sample execution of MinWVC-BarEven Algorithm. Given the graph in (a), nodes having weights 32 and 22 are locally-heaviest and make their requests for the nodes having weights 12 and 2, respectively. At second round, the requests are acknowledged. Lower-weighted nodes are deactivated and the weights of their neighbors are updated. Observe that this update caused one of the nodes to lose its "locally-heaviest" status. In round 3 (in (c)), there are three deciders, two of which make request to the same node. This node arbitrarily selects one of the requesters and accepts its proposal. In round 4, two more nodes are included in VC and two nodes are deactivated due to not having an active neighbor (d). Algorithm terminates in the 6th round when all the nodes are deactivated. Final solution is shown in (g).

Theorem 19. *$MinWVC\_BarEven$ Algorithm terminates in at most $O(n)$ rounds.*

Proof. There is at least one heaviest node in an odd numbered round. Then at least two nodes will match in an even-numbered node and one of them terminates. Thus, at the worst case, *n* nodes will be deactivated as of the $(2n)^{th}$ round. □

Theorem 20. *Total number of bits transmitted through the network during the execution of $MinWVC\_BarEven$ is $O(n^2 log_2(n))$.*

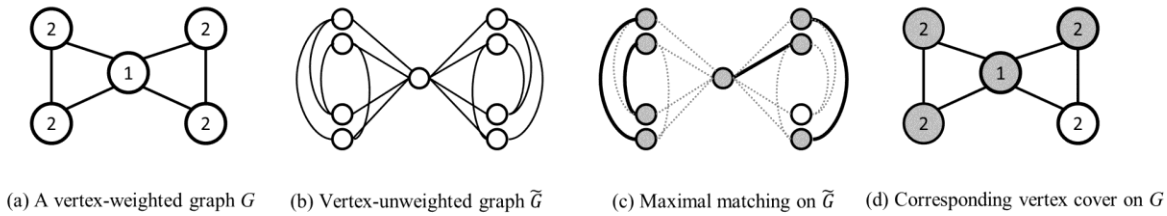
Proof. In an odd-numbered round, there can be at most *n*/2 locally heaviest nodes. Assuming each message carries the integer identifier, $O(nlog_2(n))$ bits are transmitted. In an even numbered round, each node transmits exactly one message in size of $log_2(n)$ bits. Since there are most 2*n* rounds, $O(n^2 log_2(n))$ bits are transmitted on the network. □

Theorem 21. *$MinWVC\_BarEven$ requires that each node to have $O(\Delta log_2(n))$ memory space.*

Proof. See proof of Theorem 18.

2.2.4 Distributed Weighted Vertex Cover Algorithm via Maximal Matching (MinWVC-MM)

Consider a vertex-weighted graph $G(V, E)$ with each node v having an integer weight w_v. Using G, a vertex-unweighted graph $\tilde{G}(\tilde{V}, \tilde{E})$ can be obtained in such a way that $\tilde{G}$ will include w_v *micro-nodes* ($\widetilde{v_1}, \widetilde{v_2} \dots \widetilde{v_{w_v}}$) for each $v \in V$, and there will be an edge $(u_i, v_j) \in \tilde{E}$ if and only if there is an edge $(u, v) \in E$. Figure 12 (a to b) shows such a reduction.



(a) A vertex-weighted graph G (b) Vertex-unweighted graph $\tilde{G}$ (c) Maximal matching on $\tilde{G}$ (d) Corresponding vertex cover on G

Figure 12: Example illustration of obtaining a vertex cover on vertex-weighted graphs via maximal matchings on vertex-unweighted graph.

If a maximal matching M is found on $\tilde{G}$, a corresponding vertex cover set VC can be obtained in G by including a node $v \in V$ into VC if all of the micro-nodes of v are matched in $\tilde{G}$. Figure 12(c) and 12(d) shows an example correspondence. MinWVC-MM Algorithm (Grandoni et al., 2008) is a distributed

algorithm based on this notion. Though they simplified and improved it, let us first describe the basic idea the algorithm is based on.

Each node on the distributed network is assumed to have as many micro-nodes as its weight. At the beginning of each phase of the algorithm, a micro-node becomes either a *sender* or a *receiver* with equal probability ½. (This is not permanent; each micro-node may have different status at each phase). Sender micro-nodes send a matching request to all of their neighbors. At the subsequent round, the receiver nodes having received at least one matching request select one of the senders and respond positively, and they match. Then, they are removed from the active list of their neighbors. Once a micro-node does not have an active neighbor or it is matched it is deactivated. Once all of the micro-nodes of a node are deactivated, the algorithm terminates. If all of the micro-nodes of a node are matched, the node is considered to be in VC.

It is easy to see that the number of messages a node sends or receives in a single round of the algorithm is at least as much as its weight w, namely $\Omega(W)$. So, for the networks having large weights, this algorithm is not appropriate. However, a basic simplification overcomes this problem. Since all the micro-nodes of a node have the same degree and share the same neighborhood, it is not required to hold a variable for each micro-node. Instead, one can keep the number of micro-nodes performing the same task. For example, when it is required for each micro-node to determine whether it is a sender or a receiver, instead of making decisions one by one for each micro-node, the number of senders can be decided by the node in a single operation. In a similar way, in the case of sending proposals from a node v to a neighbor u, instead of passing one message per each (v_i, u_j) micro-node pair, a single message (carrying information of the number of senders) can be sent from v to u.

Simplified and improved algorithm works as follows: Initially at each phase of the algorithm, each node decides on the number of senders (s_i) (so-called *sender micro-nodes*) and receivers (r_i) (so-called *receiver micro-nodes*) such that the total number will equal to the weight ($s_i + r_i = w_i$). It then sends a proposal to any neighbors $j \in \Gamma(v)$ with a p_{ij} value such that the sum of p_{ij}'s will be equal to the number of senders ($\sum_{j\in\Gamma(v)} p_{ij} = s_i$).

When a node i receives a proposal p_{ji} from j, it means there are p_{ji} micro-nodes of j proposing to be matched with the micro-nodes of v. Since the number of receiver micro-nodes of v is limited to r_i, it sends back an acknowledgement with value $\max(p_j, r_i)$. Then, it reduces its actual weight by the total number of acknowledged incoming proposals and acknowledged outgoing proposals. The residual weight of a node, in a sense, is the number of unmatched micro-nodes which will try to be matched at the next phase, if there will be active neighbors. This procedure is more formally demonstrated in Algorithm 8.

Algorithm 8: Distributed Weighted Vertex Cover Algorithm via Maximal Matching (*MinWVC-MM*)

```
/* At each node */
Message types: PROPOSE, RESPONSE, COVER, UPDATE
States: ACTIVE, INVC, OUTVC
while status is ACTIVE, perform Phases as follows:
  //Phase starts
  //Synchronization round
  update my knowledge of neighborhood via sending/receiving UPDATE
messages
  if I do not have an active neighbor
    set status ← OUTVC
  else
    decide number of senders s and receivers r
    distribute s among neighbors //(determine p_ij's)
    // Proposing Round
    send PROPOSE to j if p_ij > 0
```

```
    till the end of the Propose round
      receive PROPOSE from j
      set p_ji
    // Acknowledgement Round
    assign c_ij for each j where p_ji > 0 //Decide which neigh will
receive what
    for each j where p_ji > 0
      send RESPONSE(c_ij) to j
      W ← W - c_ij
    till the end of the Propose round
      receive message from j
      if message is RESPONSE(c_ji)
        set W ← W - c_ji
        if W = 0 //I have all my micro-nodes matched
          send COVER to all active neighbors
          status ← INVC
      else if message is COVER
        remove j from active neighborhood
```

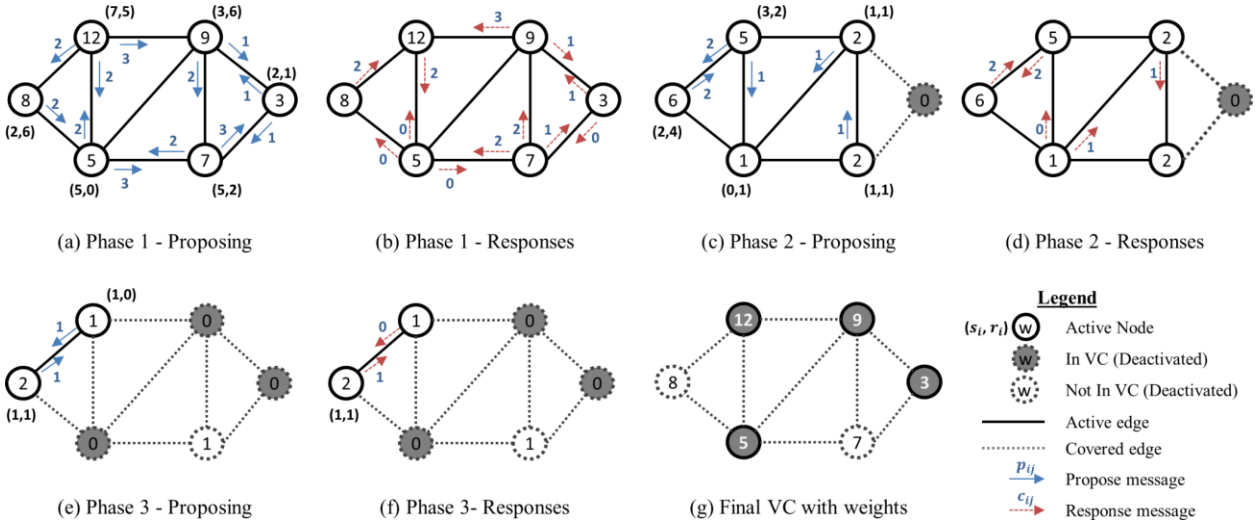


Figure 13: Sample execution of MinWVC-MM Algorithm

A sample execution of MinWVC-MM Algorithm is illustrated in Figure 13. At the proposing round of the first phase of the algorithm (a), each node decides number of senders and receivers, which are shown in parentheses in respective order. For example, the leftmost node having weight 8 decided to have 2 senders and 6 receivers, while the node assigned all of its micro-nodes as sender. Each node, then, distributes their senders among their neighbors. For example, the node having weight 12 decided that 3 of its senders will make matching proposals to the node having weight 9, 2 of its senders will propose to the leftmost node and its remaining senders will make proposal to the node having 5. All proposals are shown with a solid blue arrow.

The node having received proposals, decides on their responses in the second round of the first phase (b). For example, the node having weight 9 decided to make a positive response to all 3 proposes of node having weight 12. Note that the number of positive responses cannot exceed the number of receivers. Observe that the bottom-left node responded none of the 3 proposes positively, because it does not have a receiver micro-node. All responses are shown with a dashed red arrow.

At the end of each phase, each node reduces its weight by the number of matched micro-nodes. See in Figure 13(b) that the top-right node has 3 of its receivers matched with senders of node 12, 1 of its receivers with senders of node 3, 1 of its senders with receivers of node 3 and 2 of its senders with receivers of node 7. Thus, it has 2 unmatched micro-nodes; in other words, a residual weight of 2. (Figure 13(c)). Since the right-most node has a residual weight of 0, it is included in VC and deactivated.

The sequence of assigning sender-receivers, making proposes and making responses is repeated until all nodes are deactivated. Final solution is a vertex cover with a total weight at most twice the optimum WVC solution.

Theorem 22. MinWVC-MM Algorithm halts in $O(log_2(n) + log_2(W))$ rounds, where W is weight of the heaviest node on the network.

Proof. The residual weight of the heaviest node is expected to decrease by at least a positive constant in each phase. Then, at the worst case, a node is expected to have a residual weight 0 in $O(log_2(n) + log_2(W))$ rounds. The formal proof is beyond the scope of this book. Interested readers are referred to (Grandoni et. al., 2008).

Theorem 23. The bit complexity of MinWVC-MM Algorithm is $O(\Delta n(log_2(n) + log_2(W)))$.

Proof. In each phase, a node sends exactly one UPDATE, at most one PROPOSE and at most one RESPONSE message to each of its neighbors. Hence, it may send at most 3Δ messages in a single phase. Accordingly, total number of messages in each round is at most $3\Delta n$. Since the algorithm takes $O(log_2(n) + log_2(W))$ rounds, bit complexity is $O(\Delta n(log_2(n) + log_2(W)))$. □

Theorem 24. Each node running MinWVC-MM Algorithm requires $O(\Delta(log_2(n) + log_2(W)))$ bits of memory space.

Proof. Each node stores its id and weight of its neighbors which take $O(\Delta(log_2(n) + log_2(W)))$, assuming integer identifiers. □

2.3 Self-Stabilizing Vertex Cover Algorithms

2.3.1 Kiniwa's Self-Stabilizing Vertex Cover Algorithm (SS-VC)

Recall that the simple greedy approach to the minimum cardinality vertex cover problem presented in in Section 2.1.1, which simply chooses the node having the maximum degree, removes it from the graph and iterates until all edges are covered, approximates the optimum solution by a factor of $O(log_2(n))$ in the worst case. On the other hand, recall also that the algorithm that finds the vertex cover via a maximal matching (described in Section 2.1.2) has a bounded approximation ratio and guarantees to find a solution which is at most twice the optimum solution.

Combining these two ideas, Kiniwa proposed both a sequential algorithm and its self-stabilizing distributed version. This algorithm constructs a maximal matching by favoring the edges connecting heaviest nodes with the lightest nodes on the graph and then covers the nodes on the basis of this matching.

The algorithm works under a distributed daemon, i.e., each node independently decides when it will change its state. Every node $v \in V$ has a pointer p_v that points to the matching candidate, an integer variable $degree_v$ holding the degree of *v*, another integer variable $color_v$ holding degree of the pointed node, and a binary variable $inVC_v$ that is true when the node is in VC. Two nodes are considered *matched* if they point to each other. There are six rules determining the next state of the node.

For any node $v \in V$,

- *Rule 1*: If v is not matched, its color must be equal to its degree.
- *Rule 2*: If v is matched with a node $k \in \Gamma(v)$, its color should be the greatest of $degree_v$ and $degree_k$.
- *Rule 3*: If there are higher colored nodes pointing to the node v, then node v must point back to the one having the highest degree among them, namely k, it must set its color to the color of k, and removes itself from the solution set by setting $inVC_v$ to *false*.
- *Rule 4*: If v points to a higher colored node which does not point back to v, then v must free the pointer, reset the color to its own degree and set $inVC_v$ to *false*.
- *Rule 5*: If none of the higher colored neighbors points to v and there is at least one lower-colored node that does not point to v, v points to the one having the lowest degree among them, and sets $inVC_v$ to *true*.
- *Rule 6*: If all of the neighbors of i point to other nodes, then update the state of i as follows: if i has the minimum degree then set $cover_i$ as *false*; and set it to *true*, otherwise.

Figure 14 illustrates a sample execution of SS-VC Algorithm. Assuming starting from the empty configuration where the nodes only knows their degrees and their pointers are set to null, firstly, each node activates Rule 1 and sets its color to its degree. Then, nodes 2, 3, 5 and 6 enable Rule 5 since they are not pointed to and have a lower colored neighbor. They choose their neighbors having the lowest degree. 3 and 6 both point to 7; 2 and 5 point to 1 and 4, respectively. The configuration at this stage is shown in Figure 14(a). Then, nodes 1, 4 and 7 enable Rule 3 and point back to their responders and match. Node 7 chooses 3 instead of 6, because the node 3 has a higher degree. Thick edges show the matching. Since node 6 has not been pointed back by 7, it enables Rule 4, frees its pointer and leaves the solution set (Figure 14(b)). Node 6 enables Rule 6 and re-enters to the solution set because it is not the node with the lowest degree in its neighborhood. Stabilized solution is shown in Figure 14(c).

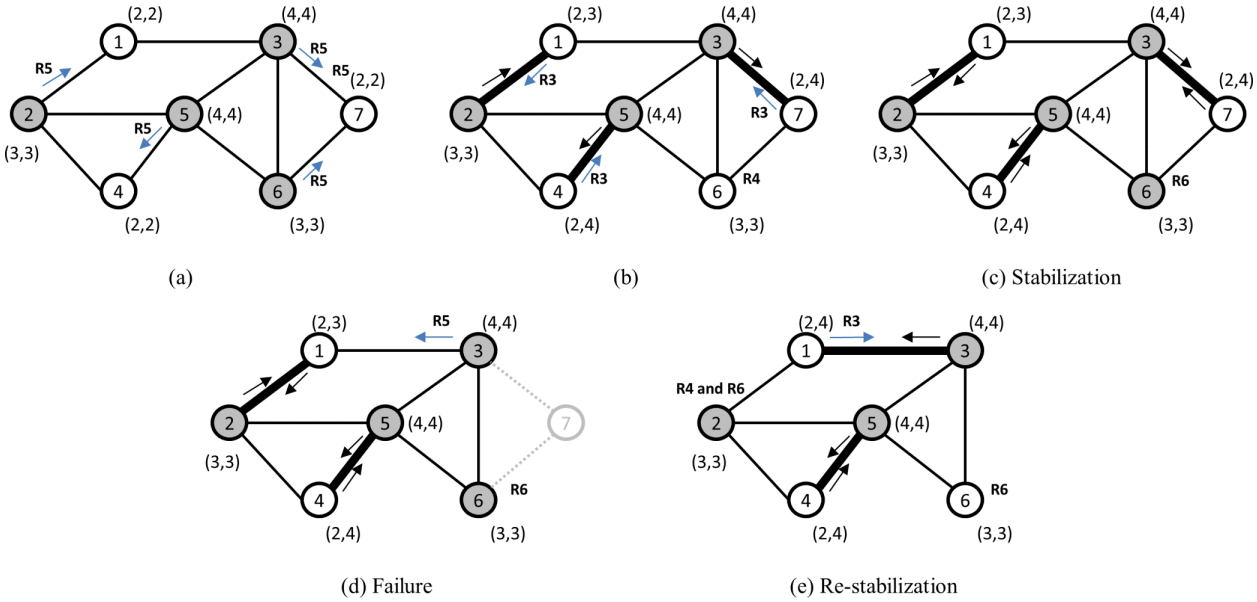


Figure 14: Sample execution of SS-VC Algorithm

In order to see the behavior of SS-VC Algorithm in case of a failure in the network, assume that node 7 is down due to any reason. In this case, node 3 starts to point 1 by means of Rule 5 (Figure 14(d)). Then node 1 breaks its matching and starts to point back the node 3 due to Rule 3. node 2 consecutively enables Rules 4 and 6 and frees its pointer. It does not leave the solution because it does not have the lowest degree. However, node 6 also enables Rule 6 and leaves the solution set due to having lowest degree.

Though in case a failure of a node, the remaining nodes are still a vertex cover, this example showed that SS-VC is able to lower the size of the VC set.

Theorem 25. SS-VC stabilizes in $O(n)$ rounds.

Proof. At the worst case, the network has only one highest degree in any round. Then, it is guaranteed that, in each round, at least one node will make a matching proposal and its proposal will be accepted at the next round. At round t, at least $t-1$ nodes will be matched. A maximal matching may include n nodes, where n is the node count. Therefore, n nodes will be matched in at most $n+1$ rounds, i.e. $O(n)$.

Approximation ratio of SS-VC is $2 - 1/\Delta$. Interested readers are referred to (Kiniwa, 2005).

2.3.2 Bipartite Matching Based Self-Stabilizing Vertex Cover Algorithm (SS-VC-Bip)

The algorithm to be presented in this section is the self-stabilizing version of Polishchuk and Suomela's distributed approximation Algorithm described in Section 2.1.3. Recall that this algorithm firstly treats the given graph as bipartite graph simply by assigning two pointers for each node, finds a maximal matching on that bipartite graph and adding the matched nodes into the solution set.

Turau (2009) implemented this idea as a self-stabilizing algorithm, in other words, it guarantees to obtain a vertex cover even in case of starting from an arbitrary state. The algorithm runs under a distributed daemon. Each node v has two pointers ($black_v$ and $white_v$) and two binary variables ($blackMatched$ and $whiteMatched$). There are two rules to determine which node or nodes the pointers point to.

- *Rule 1* (Deals with the value of white pointer): If the white pointer of the node *i* points to a node *k* but the black pointer of *k* does not point back to *i*, the white pointer of *i* is set to null. If the white pointer of *i* does not point to any node and there is at least one node whose black pointer points to *i*, then set the white pointer to one of these nodes.
- *Rule 2* (Deal with the value of black pointer): If the black pointer of node *i* points to a node k but the white pointer of k points to another node m where $m \neq i$, then free the black pointer. If the black pointer is free and there is a node $k \in \Gamma(i)$ whose white pointer is free, then set the black pointer to k.

The behavior of these rules is illustrated in Figure 15.

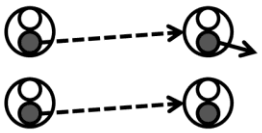
	Guard			Action	
Rule 1	White pointer is free but there is a neighbor whose black pointer points to me		→	White pointer points to that node	
	White pointer points to a vertex whose black pointer points to another vertex		→	Free white pointer	
Rule 2	Black pointer points to a vertex whose white pointer points to another vertex		→	Free the pointer	
	Black pointer is free and there is a neighbor with a free white pointer		→ →	Black pointer points to that node	

Figure 15: Possible states and the behavior of *SS-VC-Bip*

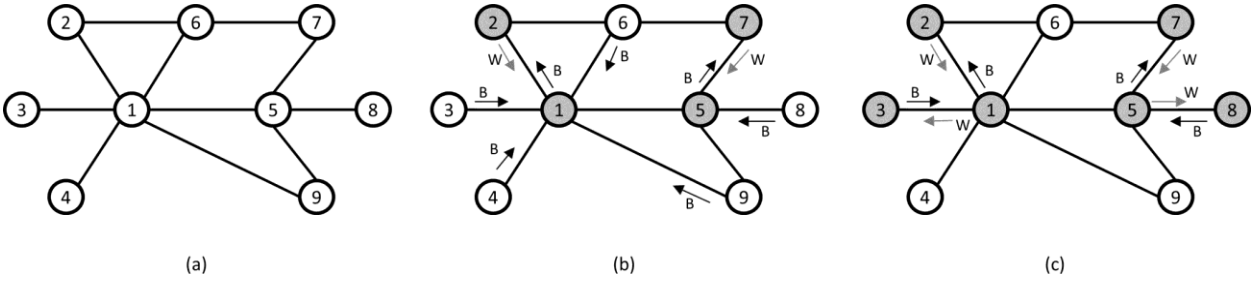


Figure 16: Sample Execution of SS-VC-Bip

A sample execution of the algorithm is illustrated in Figure 16. Though the algorithm works under distributed daemon, it is assumed for simplicity that the nodes take decision one by one in ascending order with respect to their unique identifiers. Also, each node is assumed to have an ordered list of their neighbors and favors the one having the lower id when two or more candidates are available for pointing. The time period between the activation of the first node and the last node is called a round. Figure 16(b) shows the end of the first round. Pointers are illustrated as arrows. B is the black pointer and W is the white pointer. When a node has a null pointer, it is not shown on the graph. At the first iteration, black pointer of Node 1 was set to 2 according to Rule 2. Node 2 activated Rule 1 and responded to node 1 by setting its white pointer to 1. Similarly, nodes 5 and 7 activated Rule 2 and Rule 1, respectively and became matched. Nodes 3, 4, 6, 8 and 9 activated Rule 2 and pointed their neighbor having the lowest identifier among those which have a free white pointer. Matched nodes are shown in dark color. At the second round, node 1 executed Rule 1 and chose the node 3 among the set {3,4,6,9}, which is the set of nodes having black pointer that points to 1. It pointed node 3 via its white pointer. Since its white pointer is no longer null, nodes 4 and 6 has freed their black pointer according to Rule 2. Similarly, node 5 pointed to node 8 with its white pointer, and finally, node 9 freed its black pointer since none of its neighbors has a null white pointer. The resulting configuration illustrated in Figure 16(c) is stable. As the algorithm suggests, all of the nodes are either matched or have both their pointers set to null. Matched nodes are in the solution set.

Just as Polishchuk and Suomela's Algorithm, SS-VC-Bip is a 3-approximation algorithm. Turau provides an improvement to this self-stabilizing algorithm by introducing 3 more rules which are used to eliminate the so called loose nodes on the solution set. Loose nodes are the nodes that are matched with only one node and all of their neighbors are matched. For example, the nodes 3 and 8 in Figure 16(c) are loose nodes and can be removed from the solution set without breaking the vertex cover condition. Though this approach does not improve the theoretical approximation ratio on regular graphs, it decreases the size of the solution set. Interested readers are referred to (Turau, 2009).

Theorem 26. *The algorithm SS-VC-Bip stabilizes in O(m+n) moves, assuming a distributed scheduler.*

Proof. A node running SS-VC-Bip points to each of its neighbors with its black pointer at most once. Then, at the worst case, a node i will change the value of its black pointer $2d(i)$ times, where d is the degree of i. Therefore total number of execution of Rule 2 is $4m$, since $\sum_{i \in V} d(i) = 2m$. Also, a node i can activate Rule 1 at most two times: when it starts pointing to a node which points to i with its black pointer, and when it frees its white pointer. Then, at the worst case, total number of execution of Rule 1 is $2n$. Therefore, total number of moves of the algorithm is bounded by $4m + 2n$, i.e. $O(m + n)$.

2.3.3 Self-Stabilizing Connected Vertex Cover Algorithm

The only algorithm dealing with the self-stabilizing connected vertex cover problem was introduced by Delbot et al. (2014). The main idea of the algorithm is to firstly decompose a given graph $G(V, E)$ into a set of cliques of maximal size, while securing that all cliques having a size greater than one are connected. Once this *connected minimal clique partition problem* is solved, one can obtain a connected vertex cover solution by putting the member nodes of the cliques having size at least one into the solution set.

First phase of the algorithm, which is a self-stabilizing connected minimal clique partition algorithm (*SS-MConCliq*), requires a root node $r \in V$. Also, all the other nodes should have the knowledge of their distances from the root node, namely $dist_v$. (A self-stabilizing BFS algorithm can be used for determining those values.) This distance, together with the identifier of the node, determines the clique construction priority of a node. Root node r has the highest priority. Thus, r constructs its own maximal clique and becomes the *local leader* of the clique. Other nodes in the clique know their leader by a variable $local_v$. Then, among the nodes having not been included to any clique, the ones with the locally lexicographically-lowest $(dist_v, v)$ pair become local leaders and construct their own *candidate* maximal cliques. If the nodes in this candidate clique acknowledge being in this clique, then clique is considered a *finalized* clique. Also, in order for a node to construct its clique, it should have the knowledge of the neighbors or its neighbor, in other word, its 2-hops neighborhood ($N_v \subseteq V$).

Informal descriptions of the 4 rules of SS-MConCliq algorithm are listed below:

- *N-action*: If a node v does not have the correct information of its 2-hops neighborhood, let it have. If it does not have the correct information of $dist_v$, update it.
- *C1-action*: If a node v has not been selected by a leader yet and it is not a local leader, let v be a local leader and build its candidate clique by selecting candidate neighbors.
- *C2-action:* If a node v has been selected as a candidate by a local leader and v is not the lexicographically smallest in its neighborhood, then assign the lexicographically smallest node in the clique as the $leader_v$, free the clique of v if it had one previously.
- *C3-action:* If a node v has not been selected by a leader and v has already built a candidate clique, make the candidate clique a finalized clique.

A sample execution of the algorithm is illustrated in Figure 17. It is assumed that a self-stabilizing BFS algorithm has already run and stabilized and SS-MConCliq Algorithm has not been initiated yet. When algorithm runs, initially all the nodes activate the N-action and store the correct information about their distances from the root. Within the parentheses next to each node in Figure 17(a) is the pair $(dist_v, v)$ of each node. Note that at this stage none of the nodes but only the root can activate an action. r activates C1-action and selects the nodes 2 and 4 for its candidate clique. Once 2 and 4 are selected, they are enabled to execute C2-action. They assign r as their leader. r activates C3-action and finalize the clique. Figure 17(b) shows the clique. Leader of the clique is marked with a dashed outline. As the clique is finalized, nodes 1, 3 and 5 become local leaders, execute C1-action and build their candidate cliques. 1 and 3 construct trivial cliques ($|C| = 1$). 8 executes C2-action and 5 executes C3-action (Figure 17(c)). Other cliques are built similarly (Figure 17(d)).

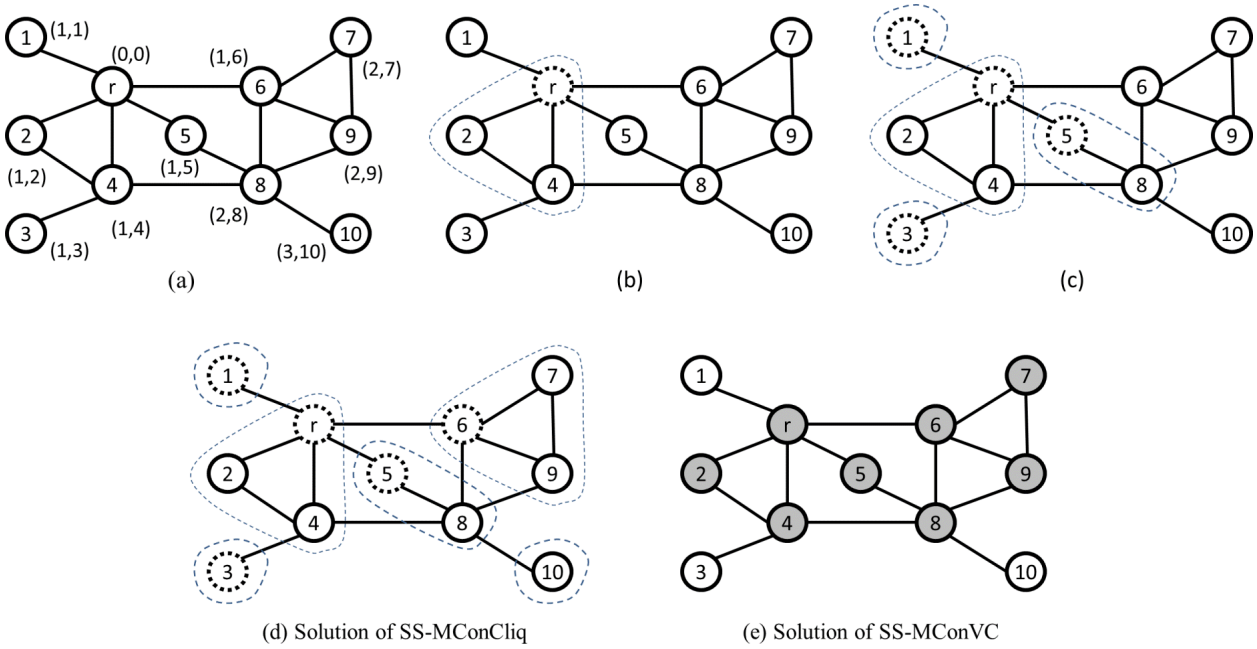


Figure 17: Sample Execution of SS-MConCliq and SS-MConVC

In order to find a vertex cover solution given the solution of SS-MConCliq, an additional binary variable $inVC_v$ and the following rule are used:

- *VC-action:* If the node v is not a leader of any clique or it is the leader of a clique having cardinality greater than 1, then set the value of $inVC_v$ to *true*, and otherwise, to *false*.

In other words, VC-action guarantees that only the nodes which are leaders of a trivial clique (|*C*|=1) is removed from VC solution set, and all other nodes are put in VC. Vertices 1, 3 and 10 in Figure 17(d) are leaders of trivial cliques and they are not included in VC (Figure 17(e)).

Theorem 27. *Assuming each node knows its distance from the root, the algorithm SS-MConCliq stabilizes in* $O(\min(n_c \times Diam, n))$ *rounds, where* n_c *is the maximum number of cliques in G,* $Diam$ *is the diameter of* G.

Proof. Since r is the only local leader at the initial step, it can construct its clique in $O(1)$ rounds, executing only C1-action and C3-action. Once the root constructs its clique, there is exactly one local leader k having the lexicographically lowest pair $(dist_v, v)$ at distance 1 and it can build its clique in $O(1)$ rounds, too. By induction, it is proved that in at most $O(t)$ rounds, a clique is built at every local leader at a distance t from r. Since the distance of a local leader from r can be at most $Diam$, all cliques are built in $O(n_c \times Diam)$ rounds. Moreover, the maximum number of cliques a graph can have is n. Therefore, the bound can be written as $O(\min(n_c \times Diam, n))$.

Theorem 28. *Assuming a stabilized configuration of SS-MConCliq Algorithm, it takes O(1) rounds for SS-MConVC to stabilize.*

Proof. Once each node is assigned to a maximal clique and knows its leader, it updates its $inVC_v$ variable in a single instruction.

References

Bar-Yehuda, R., & Even, S. (1981). A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms, 2*(2), 198-203.

Clarkson, K. L. (1983). A modification of the greedy algorithm for vertex cover.*Information Processing Letters*, *16*(1), 23-25.

Delbot, F., Laforest, C., & Rovedakis, S. (2014). Self-stabilizing algorithms for Connected Vertex Cover and Clique decomposition problems. *Principles of Distributed Systems*, 307-322.

Erciyes, K. (2013). Distributed graph algorithms for computer networks. London: Springer.

Grandoni, F., Könemann, J., & Panconesi, A. (2005). Distributed weighted vertex cover via maximal matchings. *Computing and Combinatorics*, 839-848.

Hanckowiak, M., Karonski, M., & Panconesi, A. (2001). On the distributed complexity of computing maximal matchings. *SIAM Journal on Discrete Mathematics*, *15*(1), 41-57.

Hoepman, J. H. (2004). Simple distributed weighted matchings. *arXiv preprint cs/0410047*.

Kavalci, V., Ural, A., & Dagdeviren, O. (2014). Distributed Vertex Cover Algorithms For Wireless Sensor Networks. *arXiv preprint arXiv:1402.2140*.

Kiniwa, J. (2005). Approximation of self-stabilizing vertex cover less than 2. In*Self-Stabilizing Systems* (pp. 171-182). Springer Berlin Heidelberg.

Parnas, M., & Ron, D. (2007). Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms. *Theoretical Computer Science*, *381*(1), 183-196.

Polishchuk, V., & Suomela, J. (2009). A simple local 3-approximation algorithm for vertex cover. *Information Processing Letters*, *109*(12), 642-645.

Turau, V., & Hauck, B. (2009). A self-stabilizing approximation algorithm for vertex cover in anonymous networks. *Stabilization, Safety, and Security of Distributed Systems*, 341-353.