An application of snap-stabilization: matching in bipartite graphs *

Rachid Hadid
Department of Computer Science
Mazoon College
P.O. Box 101, PC 133 Sultanate of Oman

Mehmet Hakan Karaata
Department of Computer Eng
Kuwait University
P.O. Box 5969, Safat 13060 Kuwait

Abstract: A *matching* $\mathcal{M}$ of graph $G = (V, E)$ is a subset of the edges E, such that no vertex in V is incident to more than one edge in $\mathcal{M}$. The matching $\mathcal{M}$ is maximum if there is no matching in G with size strictly larger than the size of $\mathcal{M}$. In this paper, we present a distributed stabilizing algorithm for finding maximum matching in bipartite graphs based on the stabilizing PIF algorithm of [8]. Since our algorithm is stabilizing, it does not require initialization and withstands transient faults. The complexity of the proposed algorithm is $O(d \times n)$ rounds, where d is the diameter of the communication network and n is the number of nodes in the network. The space complexity is $O((\Delta \times d)^2)$, where Δ is the largest degree of all the nodes in the communication network. The proposed algorithm can easily be adapted to devise a linear time optimal algorithm.

Keywords: distributed systems, fault-tolerance, propagation of information with feedback and cleaning, self-stabilization, maximum matching.

1 Introduction

A *matching* $\mathcal{M}$ of graph $G = (V, E)$ is a subset of the edges E, such that no vertex in V is incident to more than one edge in $\mathcal{M}$. The matching $\mathcal{M}$ is maximum if there is no matching in G with size strictly larger than the size of $\mathcal{M}$. Given a matching $\mathcal{M}$, vertex $v \in V$ is said to be *saturated*, if there is an edge in $\mathcal{M}$ incident on v. If there is no edge in $\mathcal{M}$ incident on v, vertex v said to be *unsaturated*. An *alternating path* in G, with respect to $\mathcal{M}$, is a path whose edges are alternately in $\mathcal{M}$ and in $(E$ - $\mathcal{M})$. An *augmenting path* is an alternating path connecting two unsaturated vertices.

Berge [3] shows that a matching is maximum if and only if it does not contain an augmenting path. This well known theorem gives an interesting approach to find maximum matching: First, augmenting paths in G are found, and then the current matching along each such path is augmented, i.e., each matched edge on the path is made unmatched and vice versa. If no such path is found, then the matching is optimal. Given this result, a procedure is needed to find augmenting paths. Finding augmenting paths is a well studied problem in graph theory, for which several efficient sequential algorithms exist [12]. *Edmonds* [11] was the first to propose a polynomial algorithm for finding augmenting paths. This algorithm is based on the construction of augmenting paths by using a breadth first search. This method computes a maximum matching in $O(mn)$ time for a bipartite graph. Hopcroft and Karp [18] proposed an algorithm computing the maximum matching in bipartite graphs in $O(\sqrt{n}m)$ time. This method combines the breadth first and the depth first search to find the set of the shortest augmenting paths, and then augment the current matching along these paths. Micali and Vazirani extend this algorithm to general

*Contact Author: Mehmet Hakan Karaata, Email: karaata@eng.kuniv.edu.kw, Phone: (965) 498 5842, Fax: (965) 483 9461

graphs [25]. The maximum matching problem has important applications to problems such as job assignment and task scheduling [16].

A desirable property of the proposed algorithm is the property of self-stabilization. A self-stabilizing system guarantees that, regardless of the current configuration, the system reaches a legal state in a bounded number of steps and the system state remains legal thereafter. Since the introduction of self-stabilization Dijkstra [10], self-stabilizing algorithms for many fundamental problems in distributed systems have been proposed. For example, self-stabilizing mutual exclusion algorithms for a variety of network classes have been presented [4, 6, 10]. Self-stabilizing model transformers appear in [26, 20]. Self-stabilizing algorithms for a variety of graph theoretic problems are presented in [19, 9, 27, 5, 22, 21]. General techniques for constructing self-stabilizing algorithms are dealt with in [1, 2, 24]. Self-Stabilizing algorithms that contain the effects of a single fault are presented in [13, 15, 17, 14]. Self-stabilizing algorithms are able to withstand *transient failures*. We view a fault that perturbs the state of the system but not the program as a transient fault. Due to these features, devising stabilizing distributed, sensor and mobile network protocols is desirable.

Surprisingly, few papers dealt with this problem in distributed systems and they are stabilizing [23, 7]. A stabilizing algorithm for finding a maximum matching using $O(n^4)$ moves is presented in [23]. Another stabilizing algorithm was presented in [7] for bipartite graphs, where each process knows its bipartition and some upper bound N on the number of processes in the network. This solution uses a breadth first search method to find augmenting paths. In addition, processes in each bipartition have a different algorithm than the processes in the other bipartition. The time complexity of this algorithm is $O(n^2)$ rounds.

In this paper, we propose a stabilizing matching algorithm in bipartite graphs. Our motivation is not to present yet another matching algorithm in bipartite graphs, but to present a new approach to deal with this and similar problems. Our solution uses the well-known propagation of information with feedback (PIF) concept. Specifically, we use a variation of the PIF algorithm, called *Propagation of Information with Feedback and Cleaning* algorithm introduced in [8]. The space requirement of our algorithm is $O((\Delta \times d)^2)$ per process and the time complexity is $O(dn)$ rounds, where d is the diameter of the graph. Due to its simplicity, the proposed methodology can easily be adapted to devise a linear time optimal algorithm or a stabilizing solution to the problem for arbitrary graphs. Since it is obtained by making some simple modification and additions to the stabilizing PIF algorithm of Cournier et. al [8], it is simpler and more understandable (after understanding the mechanism of the stabilizing PIF algorithm) than that of Chattopadnyay et. al [7]. In addition, the algorithm's complexity is improved over the algorithm of Chattopadnyay et. al [7]. Unlike [7], only unsaturated processes need to know their bipartition and no process needs to know an upperbound on the size of the communication network in the proposed algorithm.

The rest of the paper is organized as follows. In Section 2, we describe the distributed system in consideration and define the model of computation. The proposed stabilizing matching algorithm is presented in Section 3. We then prove the correctness of the algorithm in Section 4. Finally, we included some concluding remarks and discussions on future work in Section 5.

2 Distributed System and Program

The distributed system under consideration is represented by a *bipartite graph* $G = (V, E)$ with vertex (node) set V and edge set E. A bipartite graph G is a graph such that the vertices of G can be partitioned in two non-empty sets $\mathcal{U}$ and $\mathcal{V}$ in such a way that every edge in

G joins a vertex in $\mathcal{U}$ to a vertex in $\mathcal{V}$. Let $\mathcal{U}_{un}$ and $\mathcal{U}_s$ be the set of unsaturated and saturated processes in bipartition $\mathcal{U}$, respectively. Also let $\mathcal{V}_{un}$ and $\mathcal{V}_s$ be the set of unsaturated and saturated processes in bipartition $\mathcal{V}$, respectively. In our solution, only unsaturated processes need to know their bipartition.

Nodes of G represent *processes* and edges represent *bidirectional communications links*. A communication link (i, j) exists iff i and j are neighbors. We assume that each vertex of G is a process with a unique identity. For each process i, N_i denotes the set of its neighbors.

The distributed program of any process consists of a set locally *shared variables* (henceforth referred to as variables) and a finite set of guarded actions. We assume the local shared memory model for interprocess communication, where a process can write only to its own variables, and read its own variables and those of the neighbors. Each action is of the following form: $< label >:< guard > \rightarrow < statement >$. The *guard* of an action in the program of i is a boolean predicate over the variables of i and its neighbors. The statement of an action of i updates one or more variables of i. If a guard is **true**, then the corresponding action is said to be *enabled*. A process is called *enabled* if it has at least one action enabled. We assume that the actions are atomically executed, meaning that the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in an atomic step. The atomic execution of an action of i is called a *step* of i. The *state* of a process is defined by the values of its variables. The state of the system (*configuration*) is a cartesian product of the states of all processes. An *execution* (*computation*) is a maximal sequence of states $e = \gamma_0, \gamma_1, ...$ such that for $i \geq 0$, $\gamma_i \mapsto \gamma_{i+1}$ (a single computation step) by executing some action. We assume a *weakly fair distributed daemon*. In a computation step, if one or more processes are enabled, the distributed daemon chooses one or more of these processes to execute their actions. When a process is chosen for execution, one of its enabled guards of the process is selected nondeterministically and the corresponding action is executed. The weakly fair daemon ensures that if a process is continuously enabled, then it will be eventually chosen by the daemon. The concept of *rounds* is used as a time complexity measure, and it is related to the slowest process in any execution. Given a computation e, the first round of e, e' is the minimal prefix of e containing one atomic step of every continuously enabled process from the first configuration. Let e'' be the suffix of e, i.e., $e = e'e''$. Then the *second round* of e is the first round of e'', and so on.

3 The Maximum Matching Algorithm.

In this section, we first describe the (normal) behavior of the algorithm starting in a legal configuration. Then, we explain the method of dealing with arbitrary initialization (error correction).

3.1 Basis of the algorithm.

Starting in a state where some arbitrary set of edges are already matched, if the matching is not maximum, a forest referred to as *matching forest* is constructed. The matching forest consists of a set of disjoint trees, referred to as *matching trees* such that each vertex or edge of the graph belongs at most to one matching tree. Each matching tree is rooted at an unsaturated process in $\mathcal{U}_{un}$, called the root $r \in \mathcal{U}_{un}$ of the tree, and contains all maximal alternating paths with the root as their origin. For a matching tree rooted at process r, the processes and the edges of a matching tree are the processes and the edges, respectively, of these maximal alternating paths originating at r. The terminus (leaf processes) of these paths are either unsaturated processes in $\mathcal{V}_{un}$ or saturated processes in $\mathcal{U}_s$ or $\mathcal{V}_s$.

After the matching forest is constructed, in every matching tree rooted at a process

$r \in \mathcal{U}_{un}$, root process r initiates the *augmentation process* along one of the augmenting paths originating at process r. The augmentation process causes each matched edge to be unmatched and vice versa in an augmenting path. Through repeatedly constructing matching trees and augmenting one of the augmenting paths in each tree, the maximum matching is obtained by the proposed algorithm. Note that when possible, the proposed algorithm concurrently and independently builds disjoint matching trees and augment the discovered augmenting paths.

We use the *Propagation of information with Feedback and Cleaning ($\mathcal{PFC}$) Algorithm* of Cournier et. al [8] to build the matching trees. The $\mathcal{PFC}$ algorithm can be informally described as follows: starting in an initial configuration where no process has participated in the broadcast phase, root r initiates the broadcast phase by entering the broadcasting state. Upon discovering a neighboring process p in the broadcast state, each process q participates in this phase by assuming process p as its parent and entering the broadcast state. As a result, a spanning tree $T(r)$ rooted at r is gradually built during the broadcast phase. The processes which are not able to become the parent of another process become leaf processes in $T(r)$. Once the broadcast phase reaches the leaf processes of $T(r)$, it notifies its parent in $T(r)$ of the termination of the broadcast phase by entering the feedback state. Notice that after the broadcast reaches all the leaves, the construction of $T(r)$ is completed. Then, upon discovering that all its immediate descendants in $T(r)$ have participated in the feedback phase, each internal process also participates in the feedback phase by entering the feedback state. In this manner, the feedback phase continues towards the root process. After itself and its parent (if any) enter the feedback state, each node enters the cleaning state marking the completion of the PIF with respect to the process.

Each process p in the system maintains a variable S_p. This variable can be in three different states, *cleaning state* C, *broadcast state* B, and *feedback state* F. We now describe the meaning of a process p to be in each one of these states. If a system process p is in state C, then process p is ready to participate in the next $\mathcal{PFC}$ cycle. Process p in state B indicates that process p has participated in the broadcast phase by entering the broadcast state and if $(p \neq r)$, p has a neighbor in the broadcast state. If process p is in state F, then process p and all its ancestors have participated in the broadcast phase, and all the children (if any) of process p in $T(r)$ and process p itself have participated in the feedback phase.

Algorithm 3.1 Algorithm for an unsaturated process i ($i \in \{\mathcal{U}_{un} \cup \mathcal{V}_{un}\}$)

For process i in Partition $\mathcal{U}_{un}$ (Root):

Variables
$S_i \in \{B, F, C\}$, $P_i : \{1, ..., \Delta_i, \perp\}$, $L_i :\leq d$
$M_i : \{1, ..., \Delta_i, \perp\}$, $AugP_i, MP_i : \{\textbf{true}, \textbf{false}\}$
Predicates
$\textbf{\textit{Broadcast}}(i) \equiv S_i = C \wedge \forall_{j \in N_i}(S_j \neq C \Rightarrow P_j \neq i))$

$\textbf{\textit{Feedback}}(i) \equiv S_i = B \wedge Normal(i) \wedge$
$\forall_{j \in N_i}(P_j = i \Rightarrow S_j = F)$

$\textbf{\textit{Cleaning}}(i) \equiv S_i = F \wedge (P_i \neq \perp \Rightarrow S_{P_i} = C) \wedge$
$\forall_{j \in N_i}(P_j = i \Rightarrow S_j \in \{F, C\})$

$\textbf{\textit{Normal}}(i) \equiv (S_i = B \Rightarrow (MP_i \wedge P_i = \perp \wedge L_i = 0)) \wedge$
$(S_i = F \Rightarrow \forall_{j \in N_i}(P_j = i \Rightarrow S_j = F))$

Actions
(a_1) [] $Broadcast(i) \rightarrow$
$S_i := B$; $L_i := 0$; $P_i := \perp$;
$M_i := \perp$; $MP_i :=$ **true**;

(a_2) [] $Cleaning(i) \rightarrow$
$S_i := C$;
$M_i := min\{j \in N_i :: (P_j = i) \wedge AugP_j\}$;

(a_3) [] $Feedback(i) \rightarrow$
$S_i := F$;

(a_4) [] $\neg Normal(i) \rightarrow$
$S_i := C$;

For process i

Variables
$S_i \in \{F(B = F)$
$M_i : \{1, ..., \Delta_i, \perp$
Predicates
$\textbf{\textit{Feedback}}(i) \equiv S$

$\textbf{\textit{Cleaning}}(i) \equiv S$

$\textbf{\textit{GoodLevel}}(i) \equiv$

$\textbf{\textit{Augmenting}}(i)$

$\textbf{\textit{Normal}}(i) \equiv S_i$

Macro
$\textbf{\textit{PotentialP}}_i =$

Actions
(a_5) [] $Feedback$
$S_i := F$;
$L_i := L_P$

(a_6) [] $Cleaning$
$S_i := C$;
if $M_{P_i} =$

(a_7) [] $\neg Normal$
$S_i := C$;

In addition, each process p maintains two additional variables P_p and L_p. Variable P_p denotes the parent of process p and variable L_p denotes the length of the path followed by the broadcast phase from r to p.

Each unsaturated process $r \in \mathcal{U}_{un}$ initiates the construction of its matching tree by initi-

ating the *broadcast* phase (Action a_1). When a saturated process p (such that $S_p = C$) finds one of its neighbors q in the broadcast state (state B), process p enters the broadcast state by setting variable S_p to B, makes process q its parent by assigning q to variable P_p and decides it level by assigning $L_q + 1$ to variable L_p (Action a_8).

Algorithm 3.2 Algorithm for a saturated process i ($i \in V/\{\mathcal{U}_{un} \cup \mathcal{V}_{un}\}$)

Variables
$S_i \in \{B, F, C\}$, $P_i : \{0, 1, ..., \Delta_i - 1\}$, $L_i :\leq d$
$M_i : \{0, 1, ..., \Delta_i\}$, $AugP_i, MP_i : \{\mathbf{true}, \mathbf{false}\}$

Predicates
$\boldsymbol{Broadcast}(i) \equiv S_i = C \wedge PotentialP_i \neq \emptyset \wedge \forall_{j \in N_i}(S_j \neq C \Rightarrow P_j \neq i)$

$\boldsymbol{Cleaning}(i) \equiv S_i = F \wedge Normal(i) \wedge S_{P_i} = C \wedge \forall_{j \in N_i}(P_j = i \Rightarrow S_j = \{F, C\})$

$\boldsymbol{Feedback}(i) \equiv S_i = B \wedge Normal(i) \wedge \forall_{j \in N_i}(P_j = i \Rightarrow S_j = F)$

$\boldsymbol{GoodPif}(i) \equiv (S_i = B \Rightarrow S_{P_i} = B) \wedge (S_i = F \Rightarrow \forall_{j \in N_i}(P_j = i \Rightarrow S_j = F))$

$\boldsymbol{GoodLevel}(i) \equiv L_i = L_{P_i} + 1$

$\boldsymbol{Alternating}(i, j) \equiv m(i, j) \vee (\neg m(i, j) \Rightarrow MP_j)$

$\boldsymbol{Augmenting}(i) \equiv S_i = F \Rightarrow (AugP_i = \exists_{j \in N_i}(P_j = i) \wedge (\bigvee_{j \in N_i :: (P_j = i)}(AugP_j)))$

$\boldsymbol{Normal}(i) \equiv S_i \neq C \Rightarrow (\boldsymbol{GoodPif}(i) \wedge \boldsymbol{GoodLevel}(i) \wedge \boldsymbol{Alternating}(i, P_i) \wedge \boldsymbol{Augmenting}(i))$

Macro
$\boldsymbol{PotentialP}_i = \{j \in N_i :: (S_j = B) \wedge Alternating(i, j) \wedge P_j \neq i \wedge L_j < d\}$

Actions
(a_8) [] $Broa$
$S_i :=$
$L_i :=$
Aug

(a_9) [] $Clear$
$S_i :=$
if M_i

(a_{10}) [] $Feed$
$S_i :=$
Aug

(a_{11}) [] $\neg N$
$S_i :=$

Now we describe the mechanism employed by the construction of the matching trees. During the tree construction, we ensure that each path of the tree originating form r is an alternating path. We implement this constraint as follows: A process p is permitted to join the tree, by pointing to its parent q, if the edge connecting them is matched or this edge is unmatched however the edge connecting q to its parent is matched. To implement this, we introduce another variable called MP at each process. Each root sets its MP variable to **true** upon initiating the broadcast phase (Action a_1). A non-root process sets its MP variable to **true** when it joins the tree, if the edge connecting it to its parent is matched. Otherwise; it is set to **false** (Action a_8).

Gradually, each alternating path of each matching tree grows from its root $r \in \mathcal{U}_{un}$ until it reaches a leaf process which is either an unsaturated process $p \in \mathcal{V}_{un}$ or a saturated process with all its neighbors already participated in the broadcast phase. We denote this tree rooted at process r by $MTree(r)$. Once the broadcast phase reaches a leaf process in $MTree(r)$, it notifies its parent in $MTree(r)$ of the termination of the broadcast phase by entering the feedback phase (Actions a_5 and a_{10}). Upon entering the feedback phase, each leaf process p also informs its parent about whether or not it is saturated or unsaturated (indicating whether or not it is the terminus of an augmenting path). Consecutively, each internal process in $MTree(r)$ informs its parent whether or not it has a descendent in $MTree(r)$ that is the terminus of an augmenting path. This is implemented by using the boolean variable $AugP$ maintained by each system process. When an unsaturated leaf process

$p \in \mathcal{V}_{un}$ enters the feedback phase, it sets its $AugP_p$ variable to **true** indicating that it is the terminus of an augmenting path (Action a_5). Analogously, when a saturated leaf process $p \in \mathcal{U}_s \cup \mathcal{V}_s$ enters the feedback phase, it sets its $AugP$ variable to **false** indicating that it is the terminus of an alternating path. Then, when a saturated internal process $p \in \mathcal{U}_s \cup \mathcal{V}_s$ participates in the feedback phase, it sets its $AugP$ variable to **true** if at least one of its children has its $AugP$ variable equal to **true**. Otherwise; it is set to **false** (Action a_{10}). Eventually, all children of r enter the feedback phase by assigning F to their S variables. Consecutively, r sets its S variable to F (Action a_3). In such a configuration, an augmenting path (if any) is identified as a path from root r to a leaf process such that all the processes on the path have **true** in their $AugP$ variables.

We now describe the mechanism employed by the process of augmentation. We first describe the maintenance of matched edges. Each process p maintains a single pointer variable, denoted by M, which points to one of its neighbors. Let $\mathcal{M} = \{(p,q) \in E \mid M_p = q \text{ and } M_q = p\}$ define the set of matched edges, where the size of $\mathcal{M}$ is given by $|\mathcal{M}|$.

Upon completion of the feedback phase, the root process initiates the cleaning phase. In this phase, an augmenting path originating at root r and marked by processes whose $AugP$ variables are **true** is identified and augmented. Note that if all the children of r have their $AugP$ variable equal to **false**, since no augmenting path is detected in $MTree(r)$, root r will not initiate the augmentation process. In this case, the cleaning phase proceeds in a top-down manner by assigning C to S variables of the processes without carrying out the augmentation (Action a_9). If the root detects an augmenting path originating at r, the root process initiates this phase by assigning C to its S variable and pointing its M variable to one of its children whose $AugP_p$ variable is equal to **true** (Action a_2). Then, upon discovering that its parent has C in its S variable, each process p assigns C to its S variable. In addition, if its parent has pointed to p with its M variable, it points also its parent with its M variable (Action a_9). As a result, edge (p, P_p) is matched. Then, a child q of p becomes unsaturated and now execute the algorithm as a root process. Then an edge connecting q to one of its children is matched in the same manner as edge (r, p) is matched (Actions a_2 and a_9). In this manner, an augmenting path is destroyed and as a result matching tree $MTree(r)$ is destroyed.

The algorithm is shown in Algorithms 3.1 and 3.2 for the unsaturated and saturated processes, respectively.

3.2 Error Correction

In order for the proposed algorithm to exhibit the aforementioned behavior, referred to as the *normal behavior*, all system processes must maintain some properties based on the value of their variables and those of their parents.

For each process p not in $\mathcal{U}_{un}$ (non-root process), the following properties need to be maintained.

1. If p is in a broadcast phase, then its parent is also in the broadcast phase (Implemented by Predicate *GoodPif*).

2. If p is involved in the PIF cycle, then its level L_p is one plus that of its parent (Predicate *GoodLevel*).

3. If p participates in the broadcast phase, then p and its parent must belong to an alternating path (Predicate *Alternating*).

4. If p is in the feedback phase, then its $AugP$ is **true** if $p \in \mathcal{V}_{un}$ or $p \notin \mathcal{V}_{un}$ and at least one of its children has **true** in its $AugP$ variable (Predicate *Augmenting*).

For each root process $r \in \mathcal{U}_{un}$ involved in a new PIF cycle, $L = 0$, $P = \perp$, and $MP =$ **true** hold.

A process conforming to the above conditions is said to be in a *normal state* (Predicate *Normal*). Otherwise, it is said to be in an *abnormal state*. For satisfying these properties, the correction actions in both Algorithms 3.1 and 3.2 (Actions a_4, a_7, and a_{11}) are used.

Unfortunately, removing processes in abnormal states may not be enough to bring the system to a *normal behavior*. There may be some paths (called *abnormal paths*, defined in Section **??**) in the network which do not follow the normal behavior of the algorithm. Such paths need to be removed to make sure that the augmenting paths are detected and eventually destroyed.

4 Proof of Correctness

The proof of correctness is not included in this version of the paper.

5 Conclusion

We presented a distributed stabilizing algorithm for maximum matching problem. We introduced, for the first time, the PIF scheme for the design of maximum matching protocol. This tool allows to design a stabilizing maximum matching protocol which needs $O(MaxR \times d)$ rounds to achieve the maximum matching, where $MaxR \leq n$. The space requirement of our algorithm is $O((\Delta \times d)^2)$ per process. In this algorithm, only unsaturated processes need to know their bipartition. So, each saturated process in the bipartite graph runs the same algorithm.

The drawback of our algorithm is that if $MaxR \approx n$ then it achieves the maximum matching in polynomial time. We can remedy this problem as follows: we duplicate the state of each unsaturated process $p \in \mathcal{V}_s$ (Leaf) (variables S_p and P_p) as much as the number of its neighbors. This allows p to become leaf of multiple matching trees at the same time. Then each process $r \in \mathcal{U}_{un}$ will be able to select and then destroy an augmenting path within at most $5 \times d + 4$ rounds. So, after $O(d)$ rounds each process $r \in \mathcal{U}_{un}$ becomes saturated. Hence, all augmenting paths are destroyed concurrently. The proposed algorithm works only for bipartite graphs. In general graphs, the presence of odd cycles makes it difficult to find augmenting paths. We intend to solve this problem in general graphs using a stabilizing PIF algorithm.

References

[1] A. Arora and M. G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994.

[2] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[3] C. Berge. Two theorems in graph theory. In *Proceedings of the Natl. Acad. Sci.*, volume 43, pages 842–844, 1957.

[4] G. M. Brown, M. G. Gouda, and C. L. Wu. Token systems that self-stabilize. *IEEE Trans. Comp.*, 38(6):844–852, 1989.

[5] Steven C. Bruell, Sukumar Ghosh, Mehmet Hakan Karaata, and Sriram V. Pemmaraju. Self-stabilizing algorithms for finding centers and medians of trees. *SIAM Journal on Computing*, 29(2):600–614, 1999.

[6] J. E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11:330–344, 1989.

[7] Subhendu Chattopadhyay, Lisa Higham, and Karen Seyffarth. Dynamic and self-stabilizing distributed matching. In *PODC*, pages 290–297, 2002.

[8] A Cournier, AK Datta, F Petit, and V Villain. Self-stabilizing PIF algorithm in arbitrary rooted networks. pages 91–98, 2001.

[9] AK Datta, C Johnen, F Petit, and V Villain. Self-stabilizing depth-first token circulation in arbitrary rooted net-

works. *Distributed Computing*, 13(4):207–218, 2000.

[10] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. In *EWD 391, In Selected Writings on Computing: A Personal Perspective*, pages 41–46, 1973.

[11] J. Edmonds. Paths, trees, and flowers. *Canad. J. Math*, 17:449–467, 1965.

[12] Z. Galil. Efficient algorithms for finding maximum matchings in graphs. *ACM Computing Surveys*, 18(1):171–175, 1986.

[13] S Ghosh, A Gupta, and SV Pemmaraju. A fault-containing self-stabilizing algorithm for spanning trees. *Journal of Computing and Information*, 2:322–338, 1996.

[14] S Ghosh and X He. Fault-containing self-stabilization using priority scheduling. *Information Processing Letters*, 73(3-4):145–151, 2000.

[15] S Ghosh and SV Pemmaraju. Tradeoffs in fault-containing self-stabilization. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 157–169. Carleton University Press, 1997.

[16] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, Cambridge, 1985.

[17] T Herman and S Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Information Processing Letters*, 73(1-2):41–46, 2000.

[18] Hopcroft and Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2, 1973.

[19] Shing-Tsaan Huang and Nian-Shing Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7:61–66, 1993.

[20] Mehmet Hakan Karaata. Self-stabilizing strong fairness under weak fairness. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):337–345, 2001.

[21] Mehmet Hakan Karaata. A stabilizing algorithm for finding biconnected components. *Journal of Parallel and Distributed Computing*, 62(5):982–999, 2002.

[22] Mehmet Hakan Karaata and Pranay Chaudhuri. A self-stabilizing algorithm for bridge finding. *Distributed Computing*, 2:47–53, 1999.

[23] MH Karaata and KA Saleh. A distributed self-stabilizing algorithm for finding maximum matching. *Computer Systems Science and Engineering*, 15(3):175–180, 2000.

[24] S Katz and KJ Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.

[25] Silvio Micali and Vijay V. Vazirani. An O(sqrt($|v|$) $|e|$) algorithm for finding maximum matching in general graphs. In *FOCS*, pages 17–27. IEEE, 1980.

[26] M Nesterenko and A Arora. Stabilization-preserving atomicity refinement. In *DISC99 Distributed Computing 13th International Symposium, Springer-Verlag*, pages 254–268. Springer-Verlag, 1999.

[27] F Petit and V Villain. A space-efficient and self-stabilizing depth-first token circulation protocol for asynchronous message-passing systems. In *Euro-par'97 Parallel Processing, Proceedings LNCS:1300*, pages 476–479. Springer-Verlag, 1997.