Theoretical Computer Science 412 (2011) 5515–5526

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

A self-stabilizing $\frac{2}{3}$-approximation algorithm for the maximum matching problem

Fredrik Manne [a,*], Morten Mjelde [a], Laurence Pilard [b], Sébastien Tixeuil [c]

[a] University of Bergen, Norway

[b] University of Franche-Comté, France

[c] LIP6 & INRIA Grand Large, Université Pierre et Marie Curie - Paris 6, France

ARTICLE INFO

Keywords:
Self-stabilizing algorithm
$\frac{2}{3}$-approximation
Maximum matching

ABSTRACT

The matching problem asks for a large set of disjoint edges in a graph. It is a problem that has received considerable attention in both the sequential and the self-stabilizing literature. Previous work has resulted in self-stabilizing algorithms for computing a maximal ($\frac{1}{2}$-approximation) matching in a general graph, as well as computing a $\frac{2}{3}$-approximation on more specific graph types. In this paper, we present the first self-stabilizing algorithm for finding a $\frac{2}{3}$-approximation to the maximum matching problem in a general graph. We show that our new algorithm, when run under a distributed adversarial daemon, stabilizes after at most $O(n^2)$ rounds. However, it might still use an exponential number of time steps.

1. Introduction

A *matching* in a graph $G = (V, E)$ is a subset M of E such that no pair of edges in M have common endpoints. We say that two nodes v and w are matched if the edge (v, w) is in M. A matching M is *maximal* if no proper superset of M is also a matching. A matching M is *maximum* if there does not exist any matching with cardinality larger than $|M|$. While there exist sequential algorithms for computing a maximum matching in polynomial time, the complexity of such algorithms renders them impractical in many settings when applied to large graphs. Thus, approximation algorithms are often used to rapidly provide matchings that are within an acceptable margin of error. A maximal matching can be computed in linear time over the size of the graph, and it is well known that this results in a $\frac{1}{2}$-approximation to the maximum matching. In order to compute matchings with approximation ratios better than $\frac{1}{2}$, *augmenting paths* are often used. An augmenting path is a path in the graph, starting and ending in an unmatched node, and where every other edge is either unmatched or matched; i.e., for each consecutive pair of edges, exactly one of them must belong to the matching. Once an augmenting path p has been identified, one can increase the size of M by performing an augmenting step. This consists of removing each matched edge of p from M and including every unmatched edge of p in M. This way the cardinality of the matching is increased by one. From [12], one can easily derive that, given a graph $G = (V, E)$ and a matching $M \subseteq E$, if there does not exist an augmenting path of length $2t - 1$ or less in G, then M is a $\frac{t}{t+1}$-approximation to the maximum matching. Thus a maximal matching where no two unmatched nodes are neighbors is a $\frac{1}{2}$-approximation, and if in addition the distance between every pair of unmatched vertices is at least four then the matching is a $\frac{2}{3}$-approximation.

The matching problem is often used to model several real-world situations. Examples include the problem of assigning tasks to workers or creating pairs of entities. The latter lends itself well to a distributed network, since processes in the network may need to choose exactly one neighbor to communicate with.

* Corresponding author. Tel.: +47 40224437.
E-mail addresses: fredrikm@ii.uib.no (F. Manne), morten.mjelde@gmail.com (M. Mjelde), laurence.pilard@iut-bm.univ-fcomte.fr (L. Pilard), tixeuil@lri.fr (S. Tixeuil).

0304-3975/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.05.019

In this paper, we use augmenting paths and present a self-stabilizing algorithm that computes a $\frac{2}{3}$-approximation to the maximum matching problem in a general, unweighted graph. Our algorithm is based on using an existing maximal matching, and then identifying augmenting paths of length three. These are then used to improve the cardinality of the matching.

1.1. Self-stabilizing algorithms

Self-stabilizing algorithms [3,4] are distributed algorithms that permit forward failure recovery by means of an attractive property: starting from any arbitrary initial state, the system autonomously resumes correct behavior within finite time. Self-stabilization allows failure detection to be bypassed, yet does not make any assumptions about the nature or the span of those failures. Central to the theory of self-stabilization is the notion of a *daemon*, an abstraction for the scheduling of nodes in the system to execute their local code. A daemon is often viewed as an adversary to the algorithm that tries to prevent stabilization by scheduling the worst possible nodes for execution. The weakest possible requirement is that the daemon is *proper*, i.e. only nodes whose scheduling would change the system state are actually scheduled (these nodes are *privileged*). We only consider proper daemons in this paper. Variants of daemons can be defined along two axes: (i) a daemon may be *sequential* (meaning that no two privileged nodes may be selected by the daemon simultaneously) or *distributed* (in which case any number of privileged nodes may be selected at the same time), and (ii) a daemon may also be *fair* (which ensures that every privileged node will be allowed to move eventually) or *adversarial* (meaning that a privileged node may have to wait indefinitely, yet always scheduling *some* privileged node for execution). Intuitively, being distributed is a more general property than being sequential, and being adversarial is a more general property than being fair. Thus among these daemons, the most general is the distributed adversarial daemon, and the least general is the sequential fair daemon. As a result, an algorithm that tolerates the most general adversary also tolerates the least general one, but the converse is not necessarily true. In this paper, we only consider a distributed adversarial daemon, i.e. the most general one.

There are different ways of measuring time complexity for self-stabilizing algorithms. One common way is to measure time as the number of *rounds* executed by the algorithm. A round is the smallest subsequence of an execution in which every node privileged for at least one move at the start of a round has either executed one of these moves during the round, or has become ineligible to do so. From a practical point of view, this is perhaps the most realistic way of measuring time complexity. If we assume an asynchronous network in which each node will execute a move as soon as it becomes privileged to do so, then the number of rounds gives the number of moves executed by the slowest node.

It is also possible to count the number of single node moves or the number of *time steps* before the algorithm reaches a stable configuration. Here, a time step is one step in the execution during which at least one privileged node executes one move. Both the number of single node moves and the number of time steps can be much larger than the number of rounds. This follows since in the worst case some node might postpone execution until the rest of the graph has stabilized, even though this might take an exponentially long time. Such a sequence of moves would still only be counted as one round.

For a sequential daemon, the number of single node moves and time steps would be the same, but for a distributed daemon the number of time steps would be a measure of the total time, while the total number of moves can be interpreted as the total amount of "energy" consumed by the system. Note that the number of time steps used by a distributed daemon can be larger than the number of single node moves used by a sequential daemon. The reason for this is that if two neighboring nodes execute moves during the same time step then one of the moves may undo the effect of the other. We will give the analysis of our algorithm in terms of both the number of rounds and the number of time steps.

When no node in the graph is privileged, we say that the algorithm is *stable*, or has reached a *stable configuration*. It then follows that, in order to prove correctness, an algorithm must be shown to converge toward a stable state regardless of the initial configuration, and that this state is a solution to the problem in question.

1.2. Related work

The first self-stabilizing algorithm for computing a maximal matching was given by Hsu and Huang [13]. The authors showed a stabilization time of $O(n^3)$ moves under a sequential adversarial daemon, where n is the number of nodes in the graph. The analysis of this algorithm was later improved to $O(n^2)$ by Tel [15] and to $O(m)$ by Hedetniemi et al. [11], where m is the number of edges in the graph. The algorithm assumes an anonymous graph, and a sequential daemon is used to break symmetry. By means of randomization, Gradinariu and Johnen [9] gave a method for assigning identifiers that are unique within distance two. This was then used to transform the algorithm by Hsu and Huang so that it stabilizes under a distributed adversarial daemon, albeit only with a finite stabilization time.

Goddard et al. [6] gave a synchronous variant of Hsu and Huang's algorithm ,and showed that it stabilizes in $O(n)$ rounds. While not explicitly proved in the paper, it can be shown that this algorithm stabilizes in $\Theta(n^2)$ time steps under an adversarial distributed daemon. Gradinariu and Tixeuil [10] provide a general scheme to transform an algorithm written for a sequential adversarial daemon into an algorithm that can cope with a distributed adversarial daemon. Using this scheme on the Hsu and Huang algorithm yields a time step complexity of $O(\Delta \cdot m)$, where Δ denotes the maximum degree of the graph. Manne et al. [14] later gave an algorithm for computing a maximal matching that stabilizes in $O(n)$ rounds and in $O(m)$ time steps under a distributed adversarial daemon. The aforementioned protocols of [6,10,14] assume that the nodes are provided with unique identifiers (either globally, or within a certain distance). This is a necessary requirement, as deterministic protocols for the matching problem require symmetry breaking to deal with an adversarial daemon [14].