Self-stabilizing algorithms for the distance-k coloring problem

Shuang Wang

Thesis for the degree of Mater of Science

Department of Informatics
University of Bergen
Norway

November 21, 2007

Contents

Acknowledgement

I would like to thank my supervisor Fredrik Manne for all his help and advice with this thesis.

I also want to thank my family and my friends for their support.

Chapter 1

Introduction

Wireless networks have significantly impacted the world since World War II. Using wireless networks, armies sent information overseas or behind enemy lines easily and quickly. From then on wireless networks have continued to develop and its uses have significantly grown. Today we use it in a number of areas in our everyday life, sometimes even without noticing. People and businesses use wireless networks to send and share data quickly whether in a small office building or across the world. One of the advantages of wireless connections is that they offer more mobility, one does not need to worry about messy cables. The cellular phones we use every day make use of this advantage and change the way people interact and communicate with each other dramatically.

A wireless sensor network is a network made of numerous small independent sensor nodes. The sensor nodes are self-contained, battery-powered units with radio links that enable the entities to communicate with each other and exchange data. Just like wireless networks, the development of wireless sensor networks was originally motivated by military applications. However, wireless sensor networks are now used in many civilian application areas, including environment, habitat monitoring, home automation, and traffic control and so on. In computer science and telecommunications, wireless sensor networks have become more and more popular over the last years.

Many network problems of practical interest can be modeled as coloring problems. For example, when you are making a phone call using a mobile phone, it is necessary to transform the voice signal into an electronic wireless signal. Each such signal requires a frequency. If two mobile phones close

to each other are using the same frequency, they could interfere with each other and disrupt the signals. If we assign colors to the different frequencies, it is easy to see that a coloring of the phones where phones close to each other receive different colors solves the frequency assignment problem. This problem can easily be generalized so that one requires that phones within a certain closeness of each other have to receive different frequencies. One such generalization is the distance-k coloring problem where we view the units as being connected in a graph and where units within distance-k, for some $k > 0$, must have different colors.

In my thesis I will investigate self-stabilizing algorithms to solve the distance-k coloring problem. The self-stabilizing approach is attractive when dealing with fault tolerance in distributed systems. It provides a way to recover from faults without the cost and inconvenience of human intervention: after a fault is diagnosed, one simply has to repair the faulty components, and the system, by itself, will return to a good global state within a relatively short amount of time. There are several existing self-stabilizing coloring algorithms in [6][8][11]. Paper [6] describes an algorithm for coloring the nodes of a planar graph with no more than six colors using a self-stabilizing approach. Paper [8] proposes two new self-stabilizing distributed algorithms for proper $\Delta+1$ (Δ is the maximum degree of a node in the graph) colorings of arbitrary system graphs. In [11], they give a new simple self-stabilizing distributed algorithm for coloring a bipartite graph. But what we are interested in is the distance-k Grundy coloring problem using the self-stabilizing method. A Grundy coloring is a sequential coloring such that the color assigned to each vertex is the smallest available color which is not taken by its neighbors. In a self-stabilizing algorithm, each node only has knowledge about itself and its neighbors, no node knows the global graph. So to solve the distance-k coloring problem, we need a mechanism so that a node can obtain distance-k knowledge. There exists a paper by Hedetniemi et al. [7] about obtaining distance-k knowledge using a self-stabilizing algorithm. It explains how a node can obtain information that is distance-k away.

In this thesis we study how to obtain distance-k knowledge, then we describe how this scheme can be used in a coloring algorithm so that a node can select a color that is different from its distance-k neighbors. We show that the running time of this algorithm is $O(n^4 n^{logk})$ where n is the number of nodes of the graph. This approach is feasible but could require excessive amounts of memory, on the order of $O(n^2)$ for the whole graph. Thus we also design a distance-2 coloring algorithm that only requires five variables stored

in each node and with a running time of $O(mn^2)$ where m is the number of edges of the graph. This distance-2 coloring algorithm can only guarantee to solve the distance-2 coloring problem, but both the number of different colors and the value of the highest color could be high. We therefore give an improved distance-2 coloring algorithm that limits the number of colors. This algorithm requires eight variables stored in each node.

The remaining thesis is organized as follows. In Chapter 2 we give an introduction to self-stabilizing algorithms and coloring problems. In Chapter 3 we present an existing self-stabilizing algorithm for providing distance-k knowledge in general networks and point out some new results. In Chapter 4 we investigate how this distance-k knowledge can be used in coloring algorithms to produce a self-stabilizing distance-k coloring algorithm **DkCA**. In Chapter 5 we design a distance-2 coloring algorithm **D2UCM**. Compared with the **DkCA** algorithm, this **D2UCM** algorithm requires less memory. In Chapter 6, we improve the **D2UCM** algorithm, to make the final configuration close to a Grundy coloring. Finally, in Chapter 7, we conclude and mention some open problems.

Chapter 2

Background

In this chapter, we present our graph model then give a short introduction to self-stabilizing algorithms and coloring problems. Also some examples are given.

2.1 Graph model

Before we introduce the problem and algorithms of this thesis, it is necessary to present our graph model for networks. Let us take the mobile phone network as an example. We regard each phone as a vertex of a graph, and for two mobile phones within a certain distance, we put an edge between these two vertices. Thus our network can be modeled as an undirected graph $G(V, E)$, where V is the node set and E is the edge set. We assume $|V| = n$ and $|E| = m$. Each node has a unique ID. For a node i, we let $N(i)$ denote the set of nodes to which i is adjacent, we call these the neighbors of i. Let $N[i] = N(i) \bigcup \{i\}$ denote the set of all the neighbors of i and i itself. We let $d_i = |N(i)|$, the number of neighbors of node i, and let $\Delta = \max\{d_i | i \in V\}$. We define the distance from a node i to another node j as the minimum distance from i to j.

2.2 Self-stabilizing algorithms

In this section, we explain what a self-stabilizing algorithm is and motivate its use. We also show an example to illustrate the use of such algorithms.

The notion of a self-stabilizing algorithm was introduced by Edsger W. Dijkstra in 1973 [2]. In a self-stabilizing algorithm, each node only has knowledge

about itself and its neighbors, no node knows the global graph. On each node, there is a set of variables that denotes the local state of the node. A self-stabilizing algorithm is given as a set of rules, where each rule is composed of a predicate $p(i)$ (i is a node) and a move M (an action which causes a change in the local state of i) on the form "**if** $p(i)$ **then** M". If $p(i)$ is true then we say the node i is privileged. Until a node becomes privileged the corresponding moves cannot be executed. If no node is privileged we say that the algorithm becomes stable and that we have reached a stable configuration.

The environment of a self-stabilizing algorithm is modeled by the notion of a daemon. This is like a supervisor that controls which node makes the next move. There are two main characteristics for the daemon: it can be either central (meaning that exactly one privileged node can be executed at a given time) or distributed (meaning that any subset of privileged nodes can be executed at a given time). In an orthogonal way, a daemon can either be fair(meaning every privileged node will be executed eventually) or adversarial(meaning the daemon always works against you, and only some of the privileged nodes will be executed eventually). In this thesis, we only consider the central and adversarial daemon.

Self-stabilization algorithm is an attractive approach for dealing with problems of fault tolerance in distributed systems. Normally, when a fault happens in a network, one needs to shut down the whole network and restart it to get a good global state again. This requires a large amount of work and is also very time consuming. But a self-stabilization algorithm provides a way to recover from faults without the cost and inconvenience of a generalized human intervention: after a fault is diagnosed, one simply has to repair the faulty components. The system will return to a good global state within a relatively short amount of time by itself. Because of this property, a self-stabilizing algorithm can reach a stable state in a finite number of moves regardless of the initial configuration of the graph. For further reading about self-stabilizing algorithms, the reader can consult the book by Dolev [3].

In the following we give an example of a self-stabilizing algorithm that computes a maximal independent set of an undirected graph. The formal proof that the algorithm is self-stabilizing can be found in [9]. In graph theory, a maximal independent set is an independent set of vertices that is not a subset of any other independent set. That is, a set S such that every edge of the graph has at least one endpoint not in S and every vertex not in S has at

least one neighbor in S. To achieve this, we assign a color to each node, for a node i its color is denoted by $c(i)$. The nodes in S cannot have the same color as their neighbors.
The algorithm is as followed:

ENTER: if $c(i) = 0 \wedge (\forall j \in N(i))(c(j) = 0)$
then $c(i) = 1$

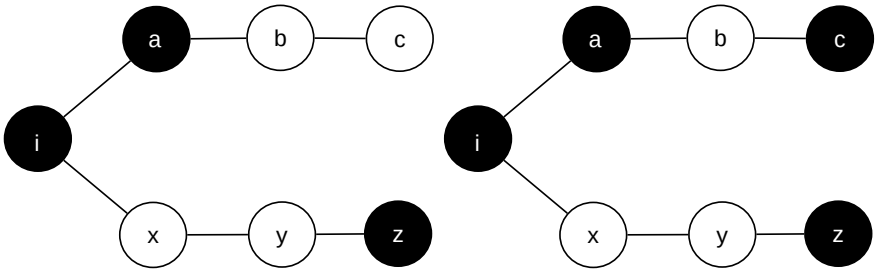
LEAVE: if $c(i) = 1 \wedge (\exists j \in N(i))(c(j) = 1)$
then $c(i) = 0$

In this algorithm, let $c(i) = 0$ mean that the node i is white and let $c(i) = 1$ mean that node i is black. The purpose of the ENTER rule is that if all the neighbors of a white node are white, this node should change into black. While the purpose of the LEAVE rule is that whenever a black node has a black neighbor, it should change its color to white. When the algorithm terminates, the maximal independent set S consists of the black nodes($c(i) = 1$). We assume no two nodes move simultaneously. In other words, there is a daemon that selects which node moves next. So whenever there is a white node and all its neighbors are white, the ENTER rule makes this node privileged. On the other hand, if a node is black and there exists at least one neighbor which is black, the LEAVE rule makes this node privileged. The daemon then decides which node among all the privileged nodes makes a move and changes its color. The algorithm becomes stable when there are no privileged nodes.

Figure 2.1 gives an example of how the algorithm could execute:

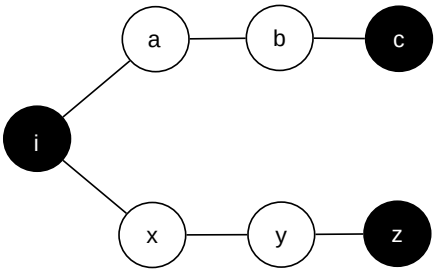
In Figure 2.1(a) nodes i, a, and z are colored black, while nodes b, c, x, and y are colored white. According to the ENTER rule, node c is privileged, while according to the LEAVE rule, nodes i and a are privileged. The daemon then decides which node makes the next move. Assume that this is c, then c will change its color to black as shown in Figure 2.1(b). After this move nodes i and a are still privileged, then the daemon could decide that node a makes the next move. After a changes its color to white, the algorithm will become stable, because no node is privileged as one can see in Figure 2.1(c). In the final solution nodes i, c, and z are colored black, and nodes a, b, x, and y are colored white.

There can be many possible results depending on the order in which the



(a)

(b)



(c)

Figure 2.1: The maximal independent set algorithm.

privileged nodes execute the moves. For the example above, if node a makes the first move, nodes b and c become privileged. Then if b changes to black, the algorithm will become stable. The graph will now end with nodes i, b, and z with black color, and nodes a, c, x, and y with white color. This example can also achieve the maximum solution if i changes its color first. This would make x privileged. After x changes its color to black, the algorithm will become stable. The graph now ends with nodes i, b, and y with white color, and nodes a, c, x, and z with black color which is the maximum independent set in this example.

2.3 The coloring problem

Many problems of practical interest can be modeled as coloring problems. The general form of these applications involves forming a graph with nodes representing items of interest. An edge connects two incompatible items. For example, if we want to schedule courses for a university, two courses given at the same time slot cannot be scheduled into the same classroom. We can construct a graph-coloring model for this problem where a graph consists of nodes representing the courses, and we use an edge to connect the courses given at the same time slot. We assign a color (represents a classroom) to each node, then we can solve the problem by giving different colors to all the adjacent nodes.

Technically a coloring of a graph is an assignment of a range of numbers $\{1, 2, \ldots, n\}$ (called colors) to its nodes such that no two adjacent nodes receive the same color. The assignment of colors is fulfilled by a function c. A node i is properly colored if for all $j \in N(i)$, $c(i) \neq c(j)$. If all the nodes are properly colored, c is a proper coloring. Finding a coloring of a general graph where each node has the smallest color number not taken by any neighbor is called a Grundy coloring [10], which can be solved in polynomial time. But to find the minimum number of colors for a given graph is an NP-hard problem [5]. Thus we do not expect to find a polynomial time algorithm to solve this NP-hard problem. In practice, however, sequential greedy coloring heuristics (Grundy coloring)[1] have been found to be quite effective.
The following describes the general Grundy coloring algorithm:

Set all nodes to uncolored
for each node $i \in V$
    set $c(i) = \min\{l \geq 1 | (\forall j \in N(i))(c(j) \neq l)\}$

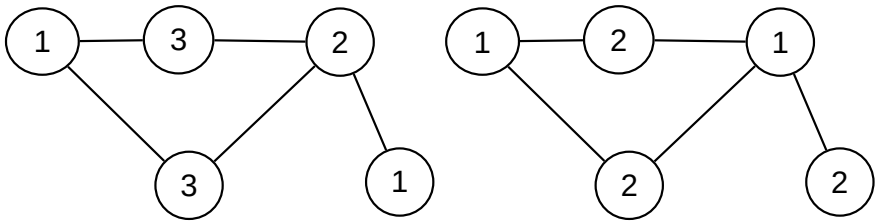


Figure 2.2: (a) Grundy coloring , (b)using the minimum number of colors.

The nodes can be colored in any order, but the result will be different depending on which order is chosen. Note that a coloring using the minimum number of colors may not be a Grundy coloring. On the other hand, a Grundy coloring may not use the minimum number of colors either. But we can apply the rules of Grundy coloring to a graph with the minimum number of colors to get a Grundy coloring with the minimum number of colors. The graphs in Figure 2.2 illustrate the difference between Grundy coloring and using the minimum number of colors. The number in each node represents its color number.

The regular coloring problem can be extended into a distance-k coloring problem which consists of assigning colors to each node i, such that it has a different color than its neighbors within distance-k. We are interested in the distance-k Grundy coloring problem, which is a distance-k coloring problem where each node must have the minimum possible color number that is not taken by its distance-k neighbors. The reason why we pay this much attention to Grundy coloring is that colors to some extend are resources. Thus to limited the number of colors is a way to save resources. In the course scheduling problem we mentioned previously, we can assign each course to a different classroom. But this solution will waste a large amount of resources. The distance-k Grundy coloring problem can be solved in polynomial time by a greedy algorithm. But to find the minimum number of colors for a distance-k coloring problem is an NP-hard problem, as it is at least as hard as the distance-1 coloring problem using the minimum number of colors.

We are particularly interested in distance-2 coloring problem as it has applications in radio broadcast networks. We can model a radio network as a graph. At any time step each node i is either transmitting (node i attempts to deliver its message to all its neighbors) or listening (node i attempts to

collect a message transmitted by one of its neighbors). If a node i is listening and some of its neighbors are transmitting simultaneously on the same channel, then i cannot hear anything because of interference. But if its neighbors are transmitting on different frequencies (we give them different colors), then there will be no collision. As we show in Figure2.3, when u and v are transmitting simultaneously, we must give them different frequencies (colors) to make sure i can hear everything from u and v.

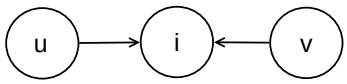


Figure 2.3: An example of radio broadcast

Chapter 3

Distance-k information

In the previous chapter, we gave an introduction to self-stabilizing algorithms. Normally, for self-stabilizing algorithms, each node only has knowledge about its immediate neighbors. But there are many problems that require more information besides the immediate neighbors. The distance-k coloring problem is one such example. These types of problems are much easier to solve when a node has access to the extended graph. A recent paper by Hedetniemi et al. [7] have described a self-stabilizing algorithm S_k for propagating this distance-k information. Since we are going to use this information, it is necessary to first outline how S_k works.

In this chapter, we expand more about distance-k knowledge in self-stabilizing algorithms. First we give some notation, then we give an outline describing how to get the distance-k information. After that, we give a small example. Finally, we show a new result about which edges a node has information about in the distance-k model.

3.1 Notation

We first define some notation that we will use in our presentation.
Let $N^s[i]$ denote the set of nodes whose distance from i is at most s, and let $N^s(i) = N^s[i] - \{i\}$. Also let $E^s(i)$ denote the set of edges between nodes $j \in N^s[i]$, and let $G^s(i)$ denote the graph composed by the nodes $j \in N^s[i]$ and the edges in $E^s(i)$. Also, let $G^s(i) - G^l(i)$ $(1 \leq l < s)$ denote the graph composed of the nodes $j \in (N^s[i] - N^l[i])$ and the set of edges between these nodes. Take Figure 3.1 as an example. The letter in each node is the ID of the node. We can see that $N^1[i] = \{i, a, x\}$, $E^1(i) = \{(i, a), (i, x)\}$

and $G^1(i) = \{N^1[i], E^1(i)\}$. Also $N^4[i] = \{i, a, b, x, y, c, d, z, w\}$, $E^4(i) = \{(i,a),(a,b),(i,x),(x,y),(b,c),(c,d),(y,z),(z,w)\}$ and $G^4(i) = \{N^4[i], E^4(i)\}$. So $G^4(i) - G^1(i) = \{\{b, y, c, d, z, w\}, \{(b,c),(c,d),(y,z),(z,w)\}\}$.
We define $dk(i)$ as the set of nodes at exactly distance k from i. For a node i, a $(ds(i), dt(i))$ edge denotes an edge between one of i's distance-s neighbors and one of its distance-t neighbors.
When we talk about the distance-k self-stabilizing model, we make the assumption that each node i can directly access all the states of nodes in $N^k[i]$.

3.2 Distance-k knowledge in self-stabilizing algorithms

In this section we explain how the algorithm S_k by Hedetniemi et al. works. Since we will use the rules and explain the details of the algorithm in Chapter 4, we do not give out all the rules here. Now we just give an outline of how a node can get information which is distance-k away.

The idea of this algorithm is based on the technique "Distance-two information in self-stabilizing algorithms" in [4]. This technique is a general mechanism that allows a node to get its distance-2 knowledge. When we use this technique recursively, we can get distance-4 knowledge based on the distance-2 model, and then distance-8 knowledge based on distance-4 model etc. Until we get the distance-k knowledge using the distance-$k/2$ model, each time the distance is doubled, thus when we investigate a distance-k algorithm, k is always a power of 2. In this way, it provides a method for transforming any distance-k algorithm into a self-stabilizing algorithm.

3.2.1 Variables

Every node has:
- A local variable f, which stores the state of the node.
- A variable σ that stores a local copy of $f(j)$ for each $j \in N^{k/2}(i)$. We say $\sigma(i)$ is correct if for all $j \in N^{k/2}(i)$, $f(j)$ in $\sigma(i)$ is correct.
- A pointer that stores the ID of a member of $N^{k/2}(i)$.

3.2.2 How the algorithm works

We now give an outline of how the algorithm S_k works and how it obtains this distance-k information.

This algorithm is used for each node i to get the correct knowledge within distance-k based on the correct σ values of the distance-$k/2$ model. In the distance-1 model, σ is empty, and node i can directly read the state of nodes $j \in N^1(i)$. When we translate a distance-1 model to distance-2, for each node i in the graph there is a $\sigma(i)$ value which stores a local copy of the state of each node $j \in N^1[i]$. Since i can read the contents of nodes $j \in N^1(i)$, which contain $\sigma(j)$, node i has indirect information about the nodes in $d2(i)$. When this distance-2 model is translated to distance-4, each node i will have a $\sigma(i)$ variable that stores a local copy of the state of each node $j \in N^2[i]$. Because node i already has a view of the nodes within distance-2, and i can get the contents of nodes $u \in d2(i)$, which contain $\sigma(u)$, then node i has indirect information about nodes in $d3(i)$ and $d4(i)$. It follows that when a distance-$k/2$ model is translated to distance-k, each node i will have a $\sigma(i)$ variable that stores a local copy of the state of each node $j \in N^{k/2}[i]$. Because node i already has a view of the nodes within distance-$k/2$, and i can get the contents of nodes $u \in dk/2(i)$, which contain $\sigma(u)$. So node i has indirect information about nodes from distance-$k/2$ to distance-k. In other words, i can have knowledge about all the nodes within distance-k.

In this algorithm, we assume there is already a distance-$k/2$ model, we show how to turn this into a distance-k model by the following rules: UPDATE-σ, ASK, RESET, and CHANGE. We use a pointer for each node to tell the system which node should make the next move. The UPDATE-σ rule will update an incorrect σ value of a node. The remaining three rules are all executed based on the correct σ value. We say that node i is S_k-alive if it is privileged for S_k. A node i that notices it is S_k-alive will try to make an ASK move. This node can make this ASK move if it and all its distance-$k/2$ neighbors are all pointing to NULL. When making an ASK move, node i sets its pointer to point to itself. The next time that the node becomes privileged, it will either change its f value (by a CHANGE move) or make a RESET move. The RESET move is made if i has a distance-$k/2$ neighbor that is pointing to itself and has a lower ID than i, this move sets the pointer of i to the distance-$k/2$ neighbor that is pointing to itself with smallest ID. The CHANGE move is made if all neighbors within distance-$k/2$ of i are pointing to i. This move then updates $f(i)$ and sets i's pointer back to NULL.

3.2.3 Example

We now give an example to illustrate how the algorithm works. In this example we skip the pointer part, and just concentrate on the σ values.
For the example in Figure 3.1, in the distance-1 model, node i can directly read the state of nodes a and x. When we translate a distance-1 model to a distance-2 model, node i can read the contents of nodes a and x, which contain $\sigma(a)$ and $\sigma(x)$ respectively. Thus node i has indirect information about f_b and f_y. When this distance-2 model is translated to distance-4 then, because node i already has a view of a, b, x, and y, and i also can read the contents of nodes b and y, which contain $\sigma(b)$ and $\sigma(y)$ respectively, node i has indirect information about f_c, f_d, f_z, and f_w.

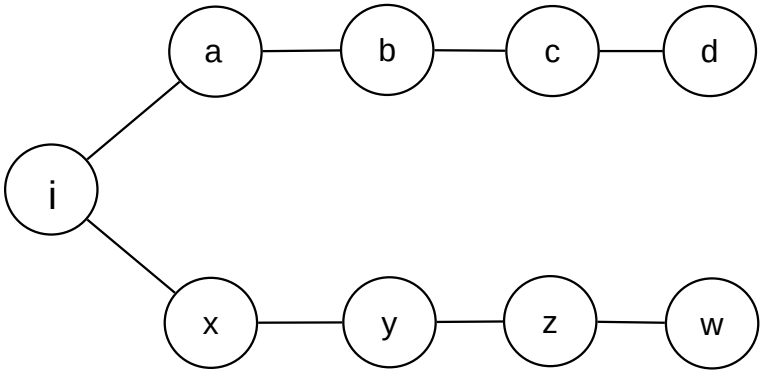


Figure 3.1: Example

3.3 The $(dk(i), dk(i))$ edges

In [7] it is shown that in a network with n nodes, a distance-k algorithm that stabilizes in A moves can be implemented in the distance-1 model by an algorithm that stabilizes in $An^{O(\log k)}$ moves. If the same distance-k algorithm uses F bits per node, then the corresponding distance-1 algorithm uses $Fn^{O(\log k)}$ bits per node. But besides the running time and memory needed there are still some details that need to be pointed out. When we say that a node i has a distance-k knowledge, intuitively, we think node i knows all the nodes within distance-k and the edges between them. But as the following shows this is not true.

Lemma 1 *For a distance-k model, a node i has knowledge about each edge in $E^k(i)$ except the $(dk(i), dk(i))$ edges.*

Proof: The proof is by induction on the k in distance k.
For $k = 1$ the distance is 2:
The node i already knows $G^1(i)$, except any $(d1(i), d1(i))$ edges. Node i can read the $\sigma(j)$ information from any node $j \in N^1(i)$, and from this i can calculate the set of nodes in $d2(i)$. It can also get all the $(d1(i), d1(i))$ and $(d1(i), d2(i))$ edges. But i cannot know if there are $(d2(i), d2(i))$ edges, because it can only read $\sigma(j)$ information from nodes in $N^1(i)$, and $\sigma(j)$ only stores the direct neighbors of nodes in $d1(i)$. This again proves that the claim is true for $k = 1$.

Assume the result is true for $k - 1$, we will show that this implies that it is also true for k.

For distance k:
From the induction hypothesis, node i already knows the structure of $G^{k/2}(i)$, except the $(dk/2(i), dk/2(i))$ edges. It also follows that, if we just consider the graph $H(i) = G^k(i) - G^{(k/2-1)}(i)$, the set of nodes in $dk/2(i)$ has knowledge about $H(i)$, except the $(dk(i), dk(i))$ edges. This follows because the distance from nodes in $dk/2(i)$ to any node in $H(i)$ is at most $k/2$. Next we consider $G^k(i)$. Because node i can read $\sigma(j)$ where $j \in dk/2(i)$, it follows that node i can know from $\sigma(j)$ if there are $(dk/2(i), dk/2(i))$ edges, and i can also get the entire structure of $H(i)$, except the $(dk(i), dk(i))$ edges, by reading what is already stored in the nodes of $dk/2(i)$. Note that there is no edge between any vertex in $G^{k/2}(i)$ and one in $H(i)$, because if there is such an edge, both of the two nodes will belong to $G^{k/2}(i)$. So in this way, the claim is also true for k.

We notice that if we want to color the nodes within distance-k from node i just by the information we can get from i, we can give the nodes within distance-$k - 1$ the correct color. But since we have no idea about the $(dk(i), dk(i))$ edges, we cannot guarantee to give the right color to the nodes in $dk(i)$. In this way, we get the corollary below.

Corollary 2 *In a distance-k model, if an algorithm requires that each node i has knowledge about its $(dk(i), dk(i))$ edges, then every node must store the local copy of $f(j)$ for each $j \in N^k(i)$ in the $\sigma(i)$ information.*

In other words, if one wants to know these $(dk(i), dk(i))$ edges, one must double the distance in which one is gathering information from $k/2$ to k.

But this also increases the amount of memory used. One way to get these $(dk(i), dk(i))$ edges information without doubling the distance is to change the algorithm in the following way. Each node not only stores its own state but also stores its neighbor list. So for each node i, instead of using $N^k[i]$ memory, we can use $N^{k/2}[i] + N(i)$ memory. We can generalize this idea into the distance-l problem, where $k < l < 2k$. Instead of doubling the distance from k to $2k$, we can make each node not only store its own state but also store its distance-$(l-k)$ neighbor list.

Chapter 4

The self-stabilizing distance-k coloring problem

In the previous chapter, we presented how to get distance-k knowledge in self-stabilizing algorithms. In this chapter we are going to use this distance-k knowledge to design a self-stabilizing algorithm for the distance-k coloring problem.

There are several existing self-stabilizing coloring algorithms in [6][8][11]. But we are interested in the distance-k Grundy coloring problem using the self-stabilizing method. We have given an introduction to Grundy coloring in Chapter 2. Here we present the self-stabilizing Grundy coloring algorithm from [8] first. Then we show our new algorithm **DkCA** which is based on Hedetniemi et al.'s algorithm [7] and the self-stabilizing Grundy coloring algorithm from [8].

4.1 Self-stabilizing Grundy coloring algorithm

In this section, we introduce the self-stabilizing Grundy coloring algorithm from [8]. Similarly to the sequential Grundy coloring algorithm in Chapter 2, in the self-stabilizing Grundy coloring algorithm, a node i also has a variable $c(i)$ to represent the color of this node. The value of $c(i)$ must be one of the numbers in the range $\{1, 2, \ldots, n\}$.
The self-stabilizing Grundy coloring algorithm is as follows:

if $c(i) \neq \min\{l \geq 1 | (\forall j \in N(i))(c(j) \neq l)\}$
  then set $c(i) = \min\{l \geq 1 | (\forall j \in N(i))(c(j) \neq l)\}$

In the algorithm above, if a node has not been assigned the smallest color not taken by any of its neighbors, then it should change into this color. It shows in [8] that the algorithm has a running time $O(n + 2m)$.

Under the distance-k model (each node i knows the state information of all nodes in $N^k[i]$), we can modify this algorithm into a distance-k self-stabilizing Grundy coloring algorithm as follows:

if $c(i) \neq \min\{l \geq 1 | (\forall j \in N^k(i))(c(j) \neq l)\}$
then set $c(i) = \min\{l \geq 1 | (\forall j \in N^k(i))(c(j) \neq l)\}$

It follows that if we want to solve a distance-k coloring problem without having the distance-k knowledge, we should use the Hedetniemi et al.'s algorithm to get the distance-k knowledge and then combine this with the distance-k Grundy coloring algorithm.

4.2 Variables and notation

We first give some variables and notations that will be used in our **DkCA** algorithm to be presented.
For a graph $G(V, E)$, every node has:
- A local variable c, which represents the color of the node.
- A variable $\sigma(i)$ that stores a local copy of $c(j)$ for each node $j \in N^{k/2}(i)$.
- A pointer that stores the ID of a member of $N^{k/2}(i)$.

Let $d_i^k =\mid N^k(i) \mid$, the number of distance-k neighbors of node i. We say that node i is **DkCA**-alive if it is privileged for **DkCA**. Define $minN^{k/2}[i]$ as the node with the smallest ID among those that point to themselves within distance-$k/2$ of i.

4.3 The algorithm

First, we will give an outline of how the algorithm works, and after this we give the details of the algorithm.

The algorithm is to get the distance-k information based on the distance-$k/2$ model, and then solve the distance-k coloring problem. For a node i, whenever its $\sigma(i)$ value is incorrect, the UPDATE-σ rule will make it correct. Based on the correct $\sigma(i)$ value of i, if i needs to change its color, an ASK

move will change the pointer of i from NULL to point to itself. Note that this can only happen when all the nodes in $N^{k/2}[i]$ are all pointing to NULL. If a node is not pointing to the node with the smallest ID in $N^{k/2}[i]$, a RESET move will make it point to this node with the smallest ID. A node i will change its color only when all the nodes within distance-$k/2$ are pointing to it. The CHANGE rule will change the color of i according to the distance-k Grundy coloring algorithm. After a node i has changed its color, the pointer of i will point to NULL again.

The following are the rules in the distance-k Grundy coloring algorithm **DkCA** based on the distance-$k/2$ model. The algorithm is designed by replacing the update $f(i)$ rule in [7] with the distance-k Grundy coloring rules :

UPDATE-σ: if $\sigma(i)$ is incorrect
then update $\sigma(i)$

ASK: if i is **DkCA**-alive $\bigwedge(\forall j \in N^{k/2}[i] : j \rightarrow NULL) \bigwedge \sigma(i)$ is correct
then $i \rightarrow i$

RESET: if $i \not\rightarrow \min N^{k/2}[i] \bigwedge \sigma(i)$ is correct
then $i \rightarrow \min N^{k/2}[i]$

CHANGE: if $\forall j \in N^{k/2}[i] : j \rightarrow i \bigwedge \sigma(i)$ is correct

$$\textbf{then} \begin{cases} \text{if } c(i) \neq \min\{l \geq 1 | (\forall j \in N^k(i))(c(j) \neq l)\}, \\ \text{then set } c(i) = \min\{l \geq 1 | (\forall j \in N^k(i))(c(j) \neq l)\} \\ i \rightarrow \text{NULL} \end{cases}$$

4.4 Correctness and running time

In this section, we first prove that a stable solution of the distance-k Grundy coloring algorithm is a correct solution. Then we look at the complexity of each rule. Finally we give the complexity of the whole algorithm.

First we define some notation that will be used in the following lemmas.
We let M_t denote the tth move. The state resulting from M_t is denoted by s_t. We define a REAL-CHANGE move as a CHANGE move in which the variable c is assigned a new value. If the CHANGE move increases the value of $c(i)$, then we say that it is an increasing move. Otherwise, we call it a

decreasing move. We say that node i makes a correct move when $\sigma(j)$ is correct for all $j \in N^{k/2}(i)$. Otherwise, we say that the move is incorrect.

Theorem 3 *Any stable solution of the distance-k Grundy coloring algorithm is a correct solution.*

Proof: We assume that the final configuration is still not a distance-k coloring when the algorithm stops. This means that there exists a node i that still has the same color as at lease one of the nodes in $N^k(i)$. We assume that the ID of i is the lowest in the whole graph among the nodes that have the same color as one of their distance-k neighbors.
There are then two cases to consider. It could be either that node i has not noticed that it needs to change its color, or that node i has already noticed but is not ready for a CHANGE move. The first case can only happen when $\sigma(i)$ is incorrect. This will force an UPDATE-σ move. For the second case, after i notices that it needs to change its color, an ASK move will change its pointer to point to itself. Note this can only happen when all the nodes in $N^{k/2}[i]$ are pointing to NULL. Otherwise, for each node j in $N^{k/2}[i]$ that is not pointing to NULL, a RESET move will force each node in $N^{k/2}(i)$ to point to the node with the smallest ID in $N^{k/2}(i)$, or to make a CHANGE move that sets its pointer to NULL. If the distance-$k/2$ neighbors of i are all pointing to i, then a CHANGE move of i should be executed. So in either case, the algorithm is not stable if the final solution is not a correct distance-k coloring. This causes a contradiction. It follows that when the algorithm stops it must be a correct distance-k coloring.

A REAL-CHANGE move of j is not always correct, it can also be incorrect meaning that $\sigma(j)$ is not correct for all $j \in N^{k/2}(i)$. But as shown in [7], an incorrect REAL-CHANGE move can only occur as a node's first CHANGE move, and any subsequent CHANGE moves will be correct. So each node can make at most one incorrect REAL-CHANGE move.

Lemma 4 *After any correct REAL-CHANGE move made by a node i, $1 \leq c(i) \leq d_i^k + 1$.*

Proof: After a correct REAL-CHANGE move, $c(i)$ is set so that $1 \leq c(i)$. Also, $c(i)$ should be equal to the minimum possible color that is not taken by any of its distance-k neighbors. Since i only has d_i^k neighbors, the result follows.

A correct REAL-CHANGE move can be a decreasing move or an increasing

move. Now we look separately at how many decreasing and increasing moves a node can make, and when the node will make these moves.

Lemma 5 *After the first correct REAL-CHANGE move, a node i can make at most $d_i^k + 1$ consecutive decreasing moves.*

Proof: This follows from Lemma 4.

Lemma 6 *A node can make a correct increasing REAL-CHANGE move M_t only if it is not properly colored in state s_{t-1}.*

Proof: Node i can only execute an increasing move when one of its distance-k neighbors has the same color $c(i)$.

Lemma 7 *After node i executes the first correct REAL-CHANGE move, it becomes properly colored and remains so.*

Proof: Node i becomes properly colored after executing a REAL-CHANGE rule, this is obvious. It will remain so because, after executing any REAL-CHANGE rule, no node can destroy the proper coloring of any of its distance-k neighbors.

Lemma 8 *Except for the initial incorrect REAL-CHANGE move, each node i can make at most one increasing move, and that can only occur before it makes any decreasing move.*

Proof: This follows from Lemma 6 and Lemma 7.

We now show the total number of correct REAL-CHANGE moves each node can make.

Lemma 9 *Each node i can make at most $d_i^k + 1$ correct REAL-CHANGE moves.*

Proof: By Lemma 5, if node i never makes an increasing move, then it will make at most $d_i^k + 1$ consecutive decreasing moves. If it makes an increasing move, then due to Lemma 8 this occurs only once before all the decreasing moves. After this first move, the remaining moves must be decreasing moves, and by Lemma 4 there are at most d_i^k such moves.

We can now look at the time complexity of the whole algorithm.

Corollary 10 *Given a graph, based on the distance-$k/2$ model, we can find a distance-k Grundy coloring in at most $O(n^4)$ moves.*

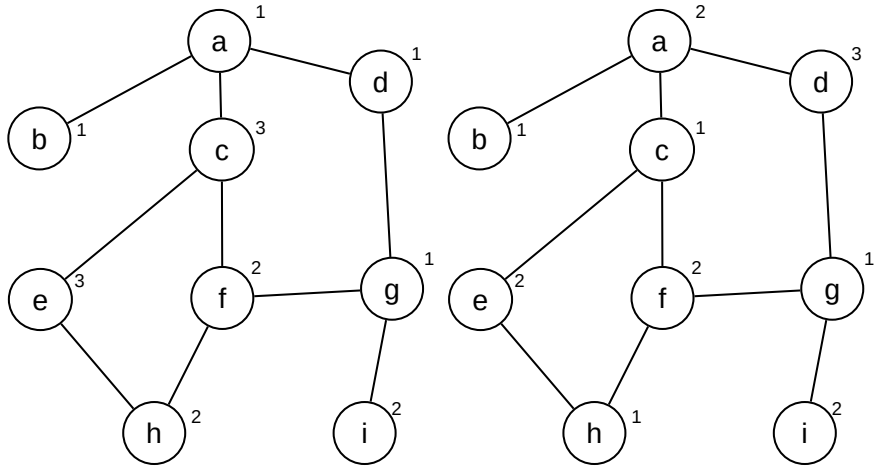
Proof: Based on Lemma 9, there are at most n^2 correct REAL-CHANGE moves. In [7] it is shown that there are at most n incorrect REAL-CHANGE moves and also there are $O(n^2)$ moves during an interval without REAL-CHANGE moves. We see that any sequence of moves takes time $O(n^4)$.

Corollary 11 *A distance-k coloring algorithm can be implemented in the distance-1 model by an algorithm that stabilizes in $O(n^4 n^{logk})$ moves.*

Proof: A distance-k Grundy coloring can be computed by at most $O(n^4)$ moves based on the distance-$k/2$ model by Corollary 10. Transferring the distance 1 model to the distance-$k/2$ model gives a slowdown of $O(n^{logk/2})$ as is shown in [7].

4.5 Example

The example in Figure 4.1 illustrates the distance-1 coloring problem, and Figure 4.2 and Figure 4.3 illustrate the distance-2 coloring problem. The number at the top right corner of a node i is the local variable $c(i)$ of i. The numbers in the bracket at the top right corner of i is $\sigma(i)$. To simplify the graph, we just list the nonrepetitive variables. This means that if there are several identical numbers in $\sigma(i)$, we only list it once. We assume that the privileged nodes execute the rules in alphabetic order according to their IDs. To simplify the graphs, for each pair of graphs we actually make several moves: Figure 4.1(b) is the final Grundy coloring starting from Figure 4.1(a). Each node chooses the minimum possible color number which is not taken by any of its neighbors. The nodes change their colors in the order a, c, d, h, e.
Between each pair of graphs in Figure 4.2 and Figure 4.3 there is one CHANGE move and all the possible UPDATE-σ moves. The dark node is the node that will make the next CHANGE move. We assume that there are no incorrect REAL-CHANGE moves.



(a) (b)

Figure 4.1: (a) the original graph , (b)For the distance-1 coloring problem, all the nodes just execute the rules in the sequential Grundy coloring algorithm.

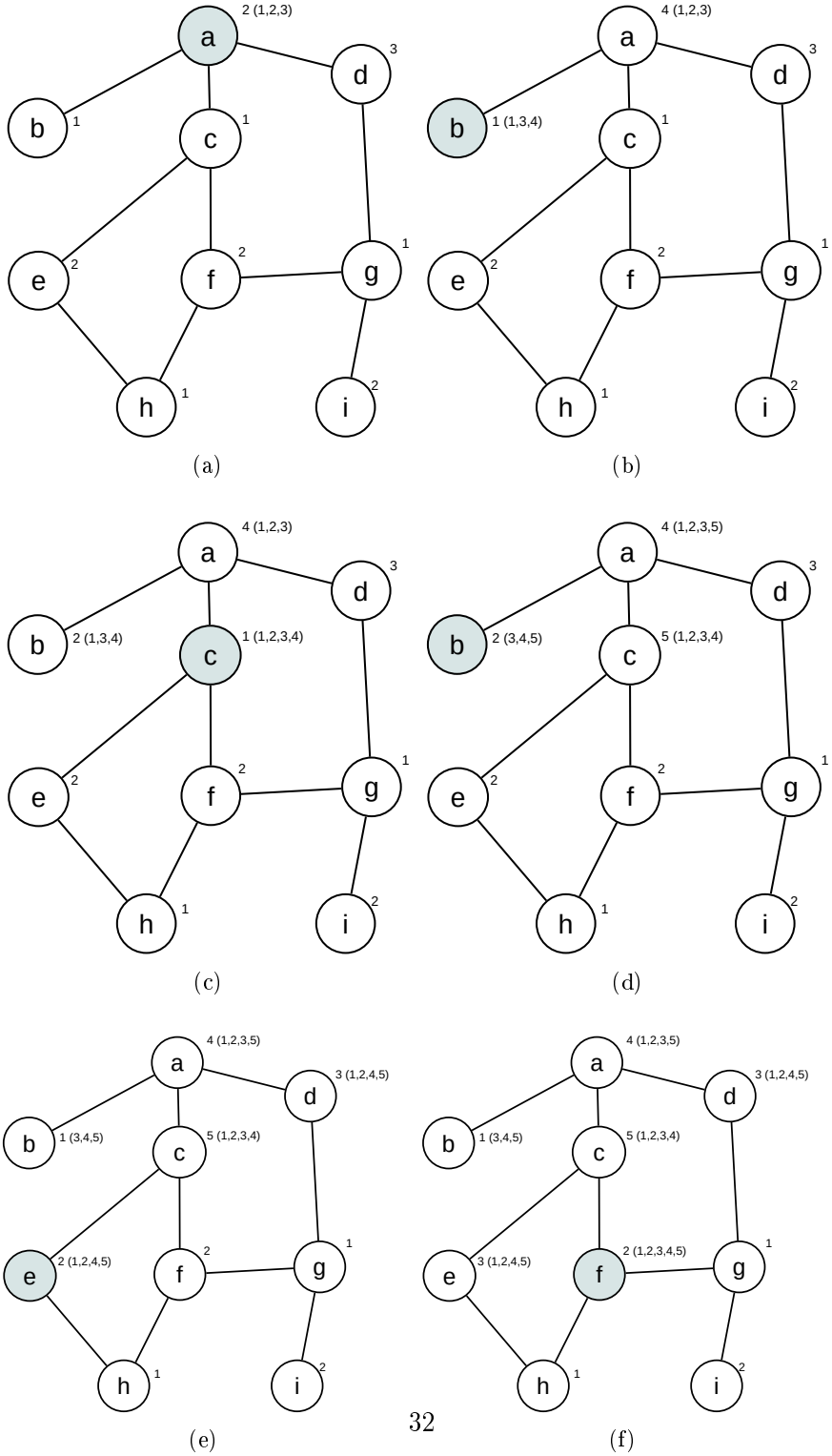


Figure 4.2: The distance-2 problem. (a) update $\sigma(a)$. (b) change $c(a)$ and update $\sigma(a)$ and $\sigma(b)$. (c) change $c(b)$ and update $\sigma(a)$, $\sigma(b)$, and $\sigma(c)$. (d) change $c(c)$ and update $\sigma(a)$ and $\sigma(b)$. (e) change $c(b)$ and update $\sigma(a)$, $\sigma(b)$, $\sigma(c)$, $\sigma(d)$, and $\sigma(e)$. (f) change $c(e)$ and update $\sigma(a)$, $\sigma(b)$, $\sigma(c)$, $\sigma(d)$, $\sigma(e)$, and $\sigma(f)$.

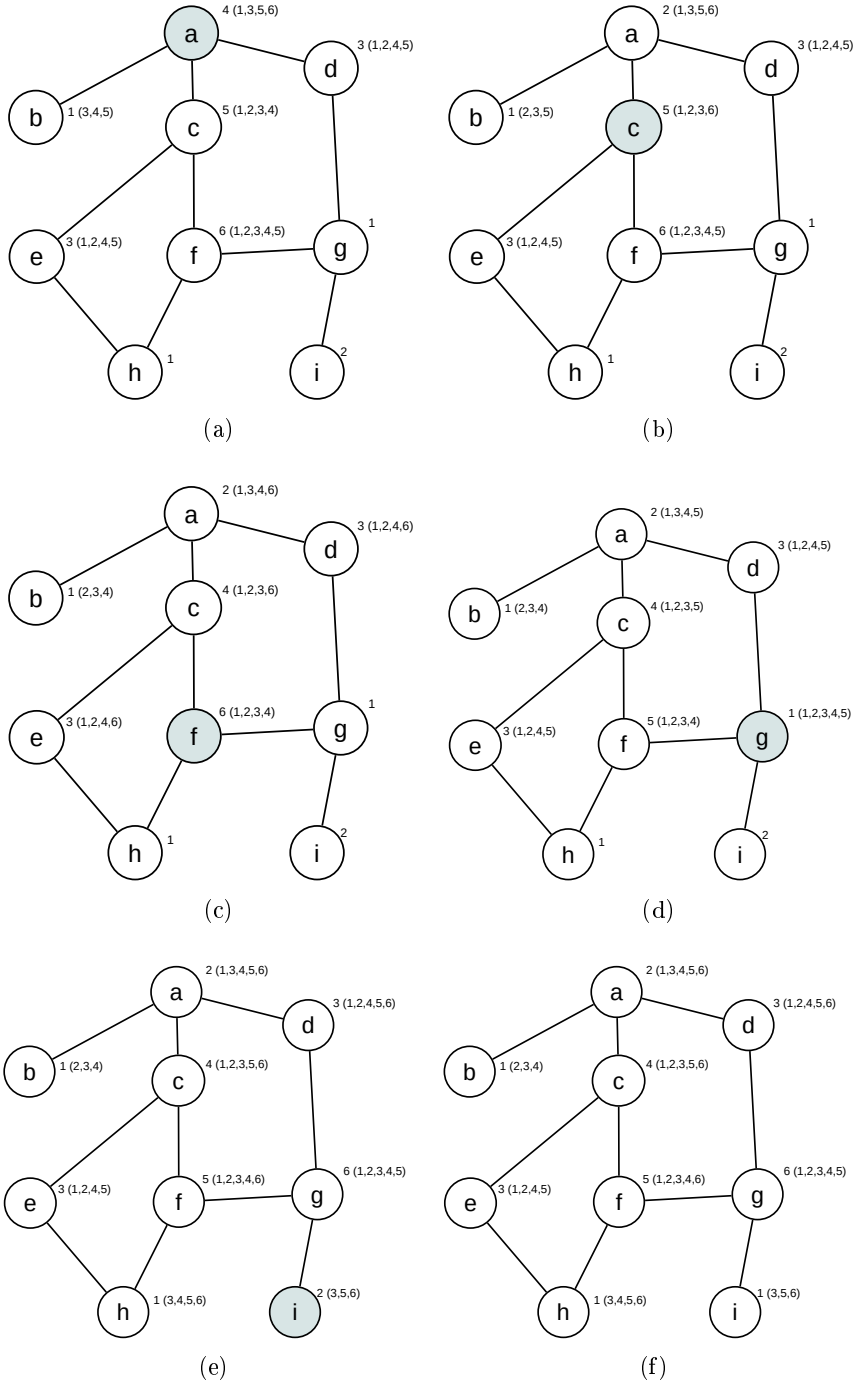


Figure 4.3: The distance-2 problem. (a) change $c(f)$ and update $\sigma(a)$. (b) change $c(a)$ and update $\sigma(a)$, $\sigma(b)$, and $\sigma(c)$. (c) change $c(c)$ and update $\sigma(a)$, $\sigma(b)$, $\sigma(c)$, $\sigma(d)$, $\sigma(e)$, and $\sigma(f)$. (d) change $c(f)$ and update $\sigma(a)$, $\sigma(b)$, $\sigma(c)$, $\sigma(d)$, $\sigma(e)$, $\sigma(f)$, and $\sigma(g)$. (e) change $c(g)$ and update $\sigma(a)$, $\sigma(b)$, $\sigma(c)$, $\sigma(d)$, $\sigma(e)$, $\sigma(f)$, $\sigma(g)$, $\sigma(h)$, and $\sigma(i)$. (f) change $c(i)$. After that, there is no privileged node, thus the algorithm is stable.

Chapter 5

A self-stabilizing distance-2 coloring algorithm

The distance-k coloring algorithm presented in Chapter 4 stabilizes in $O(n^4 n^{logk})$ moves. However, each node i needs $O(\mid N^{k/2}[i] \mid)$ memory to store the σ information. For a large value of k, this could be a significant amount of memory. Thus it would be of interest to find an algorithm that uses less memory. For any new problem, it is always easy to start with the simplest case. For this reason we start with the distance-2 coloring problem. What we are interested in is a distance-2 coloring algorithm that only uses a constant number of variables per node. When we run the distance-2 coloring algorithm using the **DkCA** algorithm in Chapter 4, the running time is $O(n^5)$ and each node needs to store $O(\mid N[i] \mid)$ variables. As we can see, even for $k = 2$, the total memory is as large as $2m$.

Before describing the algorithm in detail, we first explain the variables that the algorithm uses. We also give an outline of how the algorithm works.

5.1 Variables

For a graph $G(V, E)$, every node i has:
- A unique ID.
- A local variable $c(i)$, which represents the color of the node.
- A pointer $\rightarrow$ that either points to one of i's neighbors or to NULL. We use the notation $i \rightarrow j$ both for assigning i to point to j and for testing if i is pointing to j. On the other hand, we write $i \not\rightarrow j$ to mean that i is not pointing to j.

• A boolean $flag(i)$ variable which denotes if the node wants to change its color or not. When the value of $flag(i)$ is true, we assign it to 1, otherwise 0.
• A $c_{min}(i)$ variable will be used in the algorithm. We will explain the use of this value later in Section 5.2.

In this distance-2 algorithm only five variables are stored in each node, thus the total memory used by the algorithm is $O(n)$. This should be compared with $O(m)$ variables from the **DkCA** algorithm in Chapter 4, when **DkCA** is used to solve the distance-2 coloring problem.

5.2 The algorithm

In this section, we give this distance-2 coloring algorithm **D2UCM** that only uses five variables per node. Before outlining of the algorithm, it is necessary to investigate the distance-2 coloring problem and the purpose of the variables in Section 5.1.

A node i has a distance-2 coloring problem when it has the same color as at least one node in $N^2(i)$. If i has the same color as any node in $N(i)$, then this is easy to deal with. We can solve this problem by executing the rule of the self-stabilizing distance-1 Grundy coloring presented in Section 4.1. But if i has the same color as one of its distance-2 neighbors v, then there are more things that we need to consider. It is feasible to consider this problem from the point of view of a node j that is adjacent to both i and v. Since j can read the information in its neighbors i and v, j can discover this distance-2 coloring problem. Among the nodes in $N(j)$, there could be more node pairs than $\{i, v\}$ with the same color, we call each such pair a distance-2 coloring conflict. We let $sN(j)$ denotes the set of all the nodes in $N(j)$ that have a distance-2 coloring conflict with some node in $N(j)$, and let $min\, sN(j)$ denote the node $u \in N(j)$ with the smallest ID among the nodes in $sN(j)$. The $flag$ value of a node denotes if the node wants to change its color or not. The $flag$ value of a node is equal to 1 when this node wants to change its color. We let $fN(j)$ denote the set of all the nodes in $N(j)$ that have the $flag$ value equal to 1, and $min\, fN(j)$ denotes the node $u \in N(j)$ with the smallest ID among the nodes in $fN(j)$. The pointer of j is then used to show which node in $N(j)$ might need to change its color and to avoid that nodes with a distance-2 coloring conflict change to the same color again. The pointer of j should point to the

node u with the smallest ID among the nodes in $sN(j)$ and $fN(j)$. When $j \rightarrow u$, the variable $c_{min}(j)$ is used to tell u the smallest color number larger than u's current color that u could change into without causing a distance-2 coloring conflict. If both $sN(j)$ and $fN(j)$ are empty, j should point to NULL. A node can only change its color when all its neighbors are pointing to it. After a node changes its color, the node will update its $flag$ value to 0.

Now let us look at how the algorithm works. For a node i, if it has the same color as any node in $N(i)$, i will change its color into the minimum possible color that is different from any node in $N(i)$. A RESET move will implement this, it is the same as a distance-1 Grundy coloring. Note that i with this move could change into the same color as some nodes of its distance-2 neighbors. We say that node i is **D2UCM**-alive when $sN(i)$ or $fN(i)$ is not empty. If a node i is **D2UCM**-alive, its pointer should point to the node with the smallest ID among the nodes in $sN(i)$ and $fN(i)$. This is achieved by executing the SET MIN rule. If node i is not **D2UCM**-alive and not pointing to NULL, the SET NULL rule will make i point to NULL. For each node i, if $i \rightarrow j$ where $j = \min sN(i)$, the variable $c_{min}(i)$ stores the smallest color number that is bigger than the current color of j. Note that $c_{min}(i)$ also cannot be the same as the color of any node in $N[i]$. If $i \not\rightarrow \min sN(i)$, then $c_{min}(i)$ should be equal to 0. The UPDATE rule will update the $c_{min}(i)$ value if it is not correct. The SET FLAG rule will set the $flag$ value of i to 1 when there exists at least one neighbor j that is pointing to it and $c_{min}(j)$ is larger than $c(i)$. Whenever a node i notices that all the nodes in $N(i)$ are pointing to it, and at least one node j in $N(i)$ has a $c_{min}(j)$ value larger than $c(i)$, i will change its color and then set $flag(i)$ to 0 by executing a CHANGE move. Note that each time after a node changes its color, the color number always increases.

For a node i, we define a function CorrectValue to update its $c_{min}(i)$ value, and a function CorrectColor to change its color.

The following function will return the minimum color number that is bigger than the color of j where $j = min\, sN(i)$. Note that the value that is returned cannot be the same as the color of any node in $N[i]$.

int CorrectValue$(i, j)\{$
if $i \rightarrow j$, where $j = \min sN(i)$
    return $(min\{a, a > c(j) \bigwedge \forall j \in N[i] : a \neq c(j)\})$
else

return (0)
}

The following function is used for changing the color of a node. It returns the minimum color value that does not cause a distance-1 or distance-2 coloring conflict within $N[i]$, that is, the smallest value larger or equal to the maximum $c_{min}(j)$ value, where $j \in N(i)$. Note that the value that is returned will not be the same as the color of any node in $N[i]$.

int CorrectColor(i){
return $(min\{b, \forall j \in N(i) : b \geq c_{min}(j) \bigwedge b \neq c(j)\})$
}

The following set of rules give our self-stabilizing distance-2 coloring algorithm **D2UCM**. We assume that whenever a node i can execute the RESET rule, it will execute this before executing any of the other rules. The rest of the rules are executed in order:

RESET: if $\exists j \in N(i) : c(i) = c(j)$

then $c(i) = \min\{l \geq 1 | \forall j \in N(i) : c(j) \neq l\}$

SET MIN: if i is **D2UCM**-alive $\bigwedge i \not\rightarrow \min\{sN(i), fN(i)\}$

then $i \rightarrow \min\{sN(i), fN(i)\}$

SET NULL: if i is $\neg$ **D2UCM**-alive $\bigwedge i \not\rightarrow NULL$

then $i \rightarrow NULL$

UPDATE: if $c_{min}(i) \neq$ CorrectValue$(i, \rightarrow)$

then $c_{min}(i) =$CorrectValue$(i, \rightarrow)$

SET FLAG: if $flag(i) \neq 1 \wedge \exists j \in N(i) : (j \rightarrow i \wedge c_{min}(j) > c(i))$

then $flag(i) = 1$

CHANGE: if $(\forall k \in N(i) : k \rightarrow i) \bigwedge (\exists j \in N(i) : c_{min}(j) > c(i))$

then $c(i) = CorrectColor(i)$

$flag(i) = 0$

5.3 Example

We now give an example in Figure 5.1 that illustrates how the algorithm works. The ID of each node is represented by a letter, and nodes execute the rules in alphabetic order according to their IDs. The number inside node i is the value of $c(i)$. The arrows on the edges represent pointer values. The number on the top of each arrow is the value of c_{min}. The number on the left side of the node is its flag value. To simplify the graphs, for each pair of graphs we actually make several moves. Also, we do not show the NULL pointers and the 0 flag values.

Figure 5.1(a) is the initial configuration. Since nodes B and H are **D2UCM**-alive, they both change their pointers to point to A and update $c_{min}(B)$ and $c_{min}(H)$. Then node A changes $flag(A)$ to 1. It follows that nodes K and F also change their pointers to point to A and update $c_{min}(K)$ and $c_{min}(F)$. This is the configuration that is shown in Figure 5.1(b). After this, node A changes its color and sets $flag(A)$ to 0 and node H and F change their pointers to NULL. This causes nodes B and K to be **D2UCM**-alive, and they change their pointers to point to A and update $c_{min}(B)$ and $c_{min}(K)$. Then node A will change $flag(A)$ back to 1. It follows that node H and F also will change their pointers to point to A and update $c_{min}(H)$ and $c_{min}(F)$. This is the configuration that is shown in Figure 5.1(c). Node A then changes its color and sets $flag(A)$ to 0. Nodes B, K, and F will then change their pointers to NULL. Then node H becomes **D2UCM**-alive and changes its pointer to point to A and updates $c_{min}(H)$. This forces node A to change $flag(A)$ to 1. It follows that nodes B, K, and F will also change their pointers to point to A and update $c_{min}(B)$, $c_{min}(K)$, and $c_{min}(F)$. This is the configuration that is shown in Figure 5.1(d). After node A changes its color and sets $flag(A)$ to 0, nodes B, H, and F will also change their pointers to NULL. Then node K becomes **D2UCM**-alive and it changes its pointer to point to A and updates $c_{min}(K)$. Now node A changes $flag(A)$ to 1. It follows that nodes B, H, and F also will change their pointers to point to A and update $c_{min}(B)$, $c_{min}(H)$, and $c_{min}(F)$. This is the configuration that is shown in Figure 5.1(e). After node A has changed its color, node A changes $flag(A)$ to 0. All the pointers now point to NULL as shown in Figure 5.1(f). There is no privileged node and the algorithm is stable.

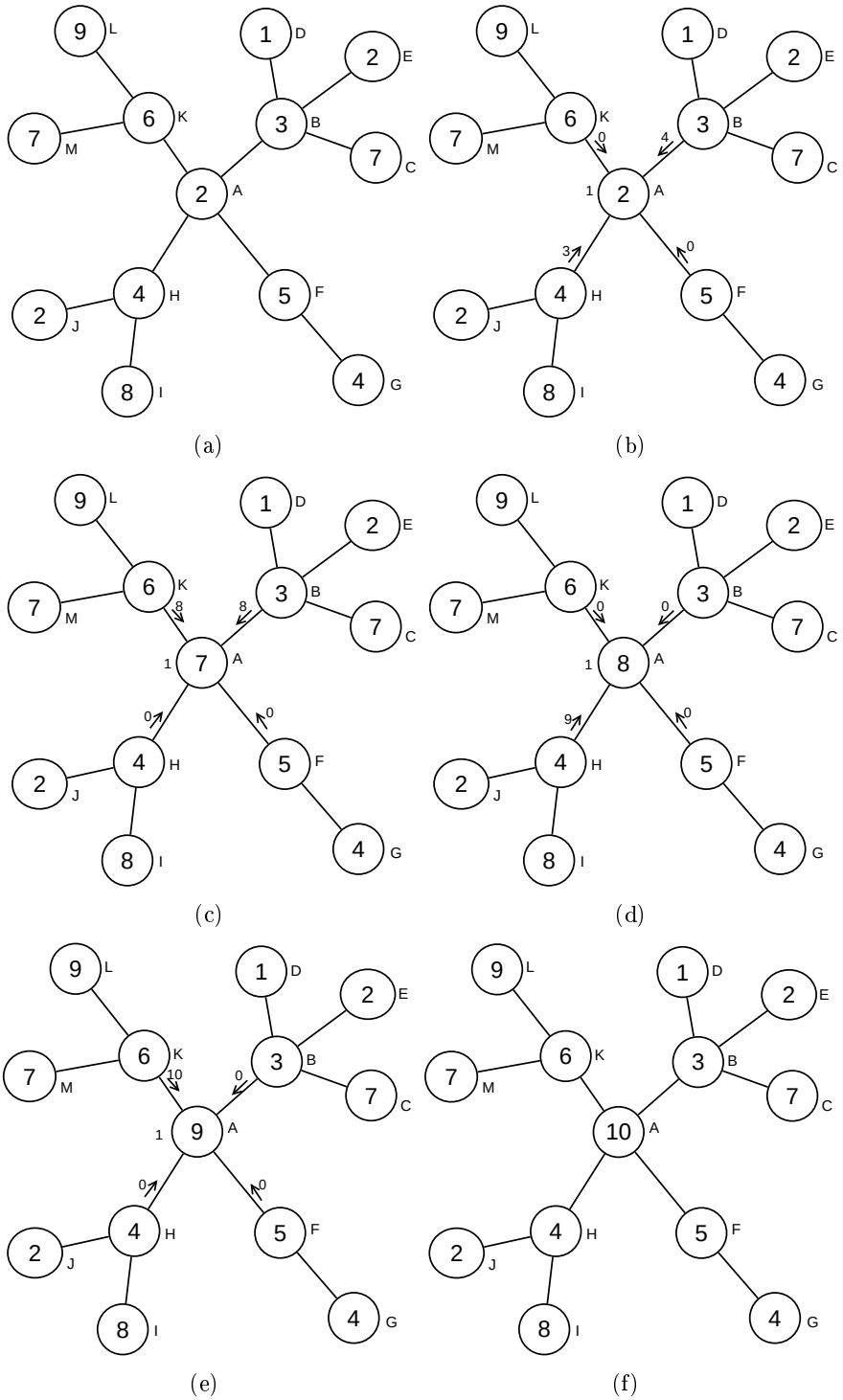


Figure 5.1: Example illustrates how the D2UCM algorithm works.

5.4 Correctness and running time

In this section, we show that **D2UCM** is a correct self-stabilizing algorithm for the distance-2 coloring problem. We do this by first proving that a stable solution is a correct solution. Then we bound the complexity of each rule before finally giving the complexity of the whole algorithm.
Before we discuss the correctness of the algorithm, we first look at a special case.

Lemma 12 *In a stable solution of **D2UCM**, a node i cannot have $i \rightarrow j$ where j does not have a distance-2 coloring conflict.*

Proof: We assume that there exists a node i in a stable solution such that $i \rightarrow j$ where j does not have a distance-2 coloring conflict. Since j does not have a distance-2 coloring conflict, it follows from the SET MIN rule that $flag(j)$ must be equal to 1. Since if $flag(j) = 0$, then i cannot point to j. From the SET FLAG rule, there must therefore exist a node $u \in N(j) - \{i\}$ such that $u \rightarrow j$ and $c_{min}(u) > c(j)$. Node u can only have $c_{min}(u) > c(j)$ when there exists a node $v \in N(u) - \{j\}$ such that $c(v) = c(j)$, which is a contradiction. Thus the result follows.

We are now going to use Lemma 12 to show that we have a legal distance-2 coloring when the algorithm is stable.

Theorem 13 *Any stable solution for **D2UCM** is a correct solution for the distance-2 coloring problem.*

Proof: We assume that the final configuration is still not a distance-2 coloring when the algorithm stops. There are then two possibilities. One is that a node i still has the same color as at least one of its neighbors j, while the other is that a node i has the same color as at least one of its distance-2 neighbors u. The final configuration can contain either or both of these cases. For the first case, a RESET move will be made to solve this problem. Thus this cannot be the case in a stable configuration. For the second case, we assume that the ID of i is the lowest among those nodes that have the same color as one of their distance-2 neighbors. Then there are three subcases. First, if the node j between i and u does not point to any node, then since j is **D2UCM**-alive, this will force j to make a SET MIN move. Second, if j is pointing to another node $v \neq i$, it follows from Lemma 12 that v also must have a distance-2 coloring conflict and since the ID of i is the lowest among all the nodes with a distance-2 conflict, j will be forced to make a SET MIN move such that $j \rightarrow i$. Third, if j is pointing

to i, then since $i = \min\{sN(j), fN(j)\}$, it follows from the CorrectValue function that $c_{min}(j)$ will be updated to a correct value larger than $c(i)$. There are now again two possibilities depending on the other neighbors of i. (1) There exists at least one neighbor x of i that is not pointing to i. This could be caused by two instances. If node i did not change its flag value to 1, then i should make a SET FLAG move. Otherwise if $flag(i)$ is already equal to 1, then x should make a SET MIN move to make its pointer also point to i. In either case, the solution is not stable. (2) If all neighbors of i are pointing to i, then i should execute a CHANGE move. So in both cases, the algorithm cannot be stable, which causes a contradiction. It follows that when the algorithm stops we must have a correct distance-2 coloring.

We are now going to discuss the complexity of each rule. We first look at the RESET rule.

Lemma 14 *Each node can make at most one RESET move.*

Proof: Note that each node executes the RESET rule before any other rules, and node i only makes a RESET move when it has the same color as at least one of its neighbors j. Thus after the RESET move, the color of i will change into the smallest possible color different from all of its neighbors. Due to the definition of the CorrectColor function, a node will never change into the same color as any node in $N(i)$ after a CHANGE move. It follows that each node can make at most one RESET move.

Since we will use the complexity of the CHANGE rule when we discuss the complexity of the rest of the rules, we now look at the CHANGE rule. We define an incorrect CHANGE move of a node i such that after the CHANGE move, the color that i changed from disappears from the distance-2 neighbors of i. A CHANGE move that is not incorrect is a correct CHANGE move. This means that after a correct CHANGE move, the color that a node is changing from still exist in at least one of its distance-2 neighbors. First, let us see how many incorrect CHANGE moves each node can make.

Lemma 15 *Each node can make $O(n)$ incorrect CHANGE moves.*

Proof: We are going to show that there are two cases when a node will make an incorrect CHANGE move. First, an incorrect CHANGE move can occur as a node i's first CHANGE move. At the start of the algorithm, it could be that all of i's neighbors are pointing to i and that there exists a neighbor j such that $c_{min}(j) > c(i)$ without $c(i)$ existing in any of i's distance-2 neighbors. Thus this could force i to make an incorrect CHANGE move where the

initial color of i disappears from the distance-2 neighbors of i. Each node i can make at most one incorrect CHANGE move in this case.
Second, as we explain in the following, an incorrect CHANGE move of i could also occur after a RESET move by i's distance-2 neighbor which initial had the same color as i. Immediately after the initial CHANGE move, $c(i)$ is larger or equal to $c_{min}(j)$ for all $j \in N(i)$. So if $c(i)$ is to change again, at least one of the nodes in $N(i)$, for instance j, should update $c_{min}(j) > c(i)$. At this time j will also point to i. The node j can only do this if there exists a node $u \in N(j)$ such that the ID of u is higher than i and $c(i) = c(u)$. As long as j is pointing to i, the value of $c(u)$ cannot change, except that u can execut a RESET move. So if u makes a RESET move before i makes the CHANGE move, then after i makes the CHANGE move, $c(i)$ might disappear from the distance-2 neighbors of i. It follows from Lemma 14 that each node can make at most one RESET move. Since i can have at most n distance-2 neighbors, i can make at most n incorrect CHANGE moves in this case. After the RESET move of u, as long as j is pointing to i, the value of $c(u)$ cannot change. So if i changes its color due to a distance-2 coloring conflict with u, the color that i is changing from still remains in the node u, and the following CHANGE move by node i is a correct CHANGE move. It follows that each node can make $O(n)$ incorrect CHANGE moves.

After considering the incorrect CHANGE moves, we now look at the correct CHANGE moves. We define $G^{2k}(i)$ as a subgraph of G. A node j is part of $G^{2k}(i)$ if there exists a sequence of nodes $i = v_1, v_2, \cdots, v_l = j$ such that for each consecutive pair v_r and v_{r+1} the following two conditions are satisfied: (1) The ID of v_{r+1} is higher than the ID of v_r. (2) v_r and v_{r+1} are distance-2 neighbors. Figure 5.2 gives an example of how $G^{2k}(i)$ is determined. The letter inside the node is the name of the node. The number outside the node is the ID of the node. Only the dark nodes are parts of $G^4(i)$. We define $d^{2k}(i)$ as the number of nodes in the subgraph $G^{2k}(i)$.

Lemma 16 *Each node i can make $O(n^2)$ correct CHANGE moves.*

Proof: After the first CHANGE move of i, it follows from Lemma 15 that as long as i's distance-2 neighbors do not make a RESET move before i makes a CHANGE move, then all further CHANGE moves of i are correct. Only a node with larger ID can force a node with smaller ID to change its color through a CHANGE move. A node i only makes a correct CHANGE move when at least one of its distance-2 neighbors with higher ID has the same color as it. After each correct CHANGE move by node i, the color α that i

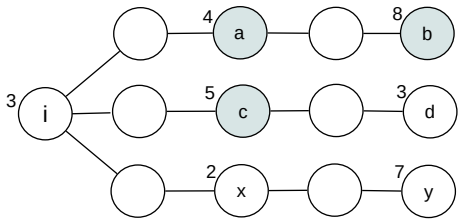


Figure 5.2: Example of how $G^{2k}(i)$ is determined

just changed from must still exist in at least one of its distance-2 neighbors v. If v later changes from α with a correct CHANGE move, then α must still be in $G^{2k}(i)$. This follows since $G^{2k}(v) \subseteq G^{2k}(i)$. The color α could disappear from the subgraph $G^{2k}(i)$ later on, but this would only be caused by some other node's incorrect CHANGE move or RESET move. Since each node can make at most n incorrect CHANGE move and one RESET move, there can at most be $d^{2k}(i)n$ incorrect CHANGE moves and $d^{2k}(i)$ RESET moves in $G^{2k}(i)$. So at most $d^{2k}(i)(n+1)$ colors can disappear from $G^{2k}(i)$. Each color that i changes away from must either remain in $G^{2k}(i)$ or disappears as a consequence of an incorrect CHANGE move in $G^{2k}(i)$. Since at most $d^{2k}(i)(n+1)n$ colors can disappear and $G^{2k}(i)$ can only store $d^{2k}(i)$ distinct colors, it follows that each node executes at most $d^{2k}(i)(n+2)$ correct CHANGE moves. Since $d^{2k}(i) < n$, the result follows.

It follows from Lemma 15 and Lemma 16 that each node can make $O(n^2)$ CHANGE moves. After having analyzed the CHANGE moves, we now look at the complexity of the remaining rules of our algorithm one by one.

Lemma 17 *Each node i can make at most $O(n^2)$ SET FLAG moves.*

Proof: On any node, SET FLAG moves and CHANGE moves must alternate. There can at most be one more CHANGE move than SET FLAG moves, this can only happen when the CHANGE move is the initial move of the node. Since each node can make at most n^2 CHANGE moves, we get that each node can make $O(n^2)$ SET FLAG moves.

Lemma 18 *Each node i can make at most $O(d_i n^2)$ SET MIN moves.*

Proof: A SET MIN move can only happen when a node i is **D2UCM**-alive and it is not pointing to the node with the smallest ID in $\{sN(i), fN(i)\}$.

We are going to show that there are two cases when a node will make a SET MIN move. First, a SET MIN move causes the pointer of i to go from NULL or to switch to some node j with higher priority(with lower ID) than the node i is currently pointing to after j updates $flag(j)$ to 1. It follows from Lemma 17 that each node can make $O(n^2)$ SET FLAG moves. Thus there could be $O(d_i n^2)$ such SET MIN moves made by node i. Second, a SET MIN move will make i point to a node v with a higher ID than j when j makes a CHANGE or RESET move. We know that each node can make $O(n^2)$ CHANGE moves, and it follows from Lemma 14 that each node can make at most one RESET move. Thus there are at most $d_i(n^2+1)$ such SET MIN moves. In total, each node i can make $O(d_i n^2)$ SET MIN moves.

Lemma 19 *Each node i can make $O(d_i n^2)$ SET NULL moves.*

Proof: On any particular node, a SET NULL move and a SET MIN move must alternate. There can at most be one more SET NULL move than SET MIN moves, this only happen when the SET NULL move occure before the first SET MIN move of the node. It follows from Lemma 18 that each node i can make $O(d_i n^2)$ SET MIN moves, and $O(d_i n^2)$ SET NULL moves.

Lemma 20 *Each node i can make at most $O(d_i n^2)$ UPDATE moves.*

Proof: Whenever a node i changes its pointer, it will make an UPDATE move. From Lemma 18 and Lemma 19, each node can change its pointer at most $O(d_i n^2)$ times. Thus each node i can make at most $O(d_i n^2)$ UPDATE moves.

We can now give the total complexity of our algorithm.

Theorem 21 *The **D2UCM** algorithm will stabilize in $O(mn^2)$ moves with a valid distance-2 coloring.*

Proof: From Lemma 15 through Lemma 20, each node makes at most $O(d_i n^2)$ moves. Thus it follows that the number of moves of algorithm can make is $\sum_{i \in V} d_i n^2 = n^2 \sum_{i \in V} d_i = 2mn^2 = O(mn^2)$. From Theorem 13, a stable solution is a correct solution and the result follows.

Chapter 6

A grundy type self-stabilizing distance-2 algorithm

The distance-2 algorithm **D2UCM** in the previous chapter can only guarantee to solve the distance-2 coloring problem, but both the number of different colors and the value of the highest color could be high. In the worst case, the algorithm could use n colors. This is the case if each node starts with a different color. Also the highest color could be any large value that can be stored in $c(i)$ and $c_{min}(i)$. Thus in this chapter we are going to give an improved algorithm **IMPROVED D2UCM** that limits the value of the highest color. What we are trying to do is to limit the color number of a node i to the number of i's distance-2 neighbors, or a little more but still near the number of its distance-2 neighbors. To achieve this, a node must know its number of distance-2 neighbors.

6.1 Variables

In this section, we will use all the notation and variables from the previous chapter. Besides this, we require some new variables that we now present.

- Let $n1n(i)$ be a variable that is set to the number of i's direct neighbors.
- $n2n[i]$ is a variable that contains the summation of $n1n(j), \forall j \in N(i)$ plus 1 for i itself.
- When a node i notices that one of its neighbors j has a coloring problem, i will change its pointer to point to j. In addition node i will also set a variable $p(i)$ to store the current color $c(j)$ that j should change away from.

Thus in this algorithm eight variables are stored in each node. This should

be compared with five variables from the previous **D2UCM** algorithm.

6.2 The algorithm

Like the former distance-2 algorithm, for a node i, we define a function CorrectValue to update its $c_{min}(i)$ value, and a function CorrectColor to change its color. The CorrectColor function is the same as for **D2UCM**, but the CorrectValue function is slightly different.

The following function will return the first value in the range $(c(j), \cdots, n2n[j], 1, 2, \cdots, c(j)-1]$ that is not equal to $c(r)$ for any $r \in N[i]$. Notice that such a value must exist if the value of this value $n2n[i]$ is correct.

int CorrectValue$(i, j)\{$
if $i \rightarrow j$, where $j = \min\ sN(i)$
    return the first value a in $(c(j), \cdots, n2n[j], 1, 2, \cdots, c(j) - 1]$:
    $\forall r \in N[i] : a \neq c(r)$
else
    return (0)
$\}$

Compared with the **D2UCM** algorithm, we have added three new rules: NUMD1, NUMD2, and LOWER. The NUMD1 rule calculates $|N(i)|$ for the node i and assigns this vale to $n1n(i)$. The NUMD2 rule gets an approximation of the number of nodes within distance-2 of i and assigns this value to $n2n[i]$. If there are no cycles of length four containing i, the $n2n[i]$ value we get is exactly the same as the number of nodes within distance-2 from i. But if there is any such cycle, $n2n[i]$ will be more than the real number, because nodes in the cycle are added more than once to $n2n[i]$. The reason why we need the $n2n[i]$ number is that we wish to limit the color of i to the approximate number of i's neighbors within distance-2. The purpose of the LOWER rule is to make the large color number lower. If the color number of a node i is larger than $n2n[i]$, we set $c(i) = 1$.

Besides these three new rules, we change slightly in the SET MIN rule, SET FLAG rule, and the CHANGE rule. We introduce a new variable $p(i)$ in these three rules. The reason why we need to use this variable $p(i)$ is as follows. Based on the CorrectValue function, when $i \rightarrow j$ where $j = \min\ sN(i)$, the correct $c_{min}(i)$ value could be less than $c(j)$. Thus if i does not update

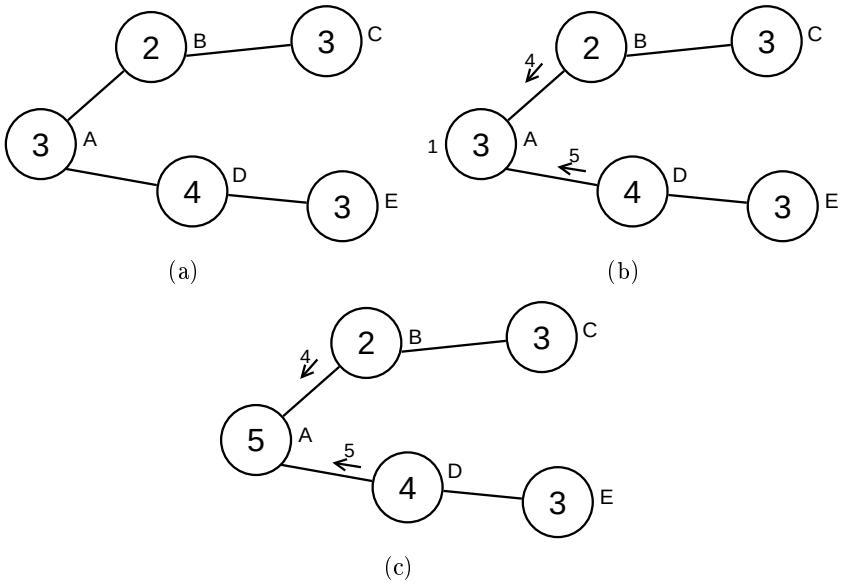


Figure 6.1: Example illustrates the necessity of the variable $p(i)$.

its pointer or $c_{min}(i)$ value after j has made a CHANGE move, j will not know if the pointer that is pointing to it and the $c_{min}(i)$ value i is providing is because of the old coloring problem that is already solved or because of a new coloring conflict. Thus j might be forced to make infinitly many wrong CHANGE moves. For the **D2UCM** algorithm, the color number a node j is changing into always increases. Thus if the node i that is pointing to j does not update its pointer or $c_{min}(i)$ value after j has made a CHANGE move, this will not force j to make unnecessary CHANGE moves.
Apart from these changes, the rules are the same as for the **D2UCM** algorithm.

We now give an example in Figure 6.1 that shows what will happen if we do not use the $p(i)$ variable. We remind the reader that the ID of each node is represented by a letter, and nodes execute the rules in alphabetic order according to their IDs. The number inside node i is the value of $c(i)$. The arrows on the edges represent pointer values. The number on the top of each arrow is the value of c_{min}. The number on the left side of the node is its flag value. To simplify the graphs, for each pair of graphs we actually make several moves. Also, we do not show the NULL pointers and the 0 flag values.

Figure 6.1(a) is the initial configuration. Since nodes B and D are **IMPROVED D2UCM**-alive, they both change their pointers to point to A and update $c_{min}(B)$ and $c_{min}(D)$. Then node A changes $flag(A)$ to 1. This is the configuration that is shown in Figure 6.1(b). After this, node A changes its color and sets $flag(A)$ to 0. But B and D do not update their pointers and $c_{min}(B)$ and $c_{min}(D)$. This is the configuration that is shown in Figure 6.1(c). Thus A might repeatly change its color between 4 and 5 an infinite number of times, since node A cannot tell if the pointers from B and D and $c_{min}(B)$ and $c_{min}(D)$ are because of a new coloring conflict or a problem that is already solved.

The following set of rules give our improved self-stabilizing distance-2 coloring algorithm. Note that whenever a node i can execute the RESET rule, it will execute this before executing any of the other rules. Also note that the value returned by the CorrectValue function is slightly changed compared to the **D2UCM** algorithm.

RESET: if $\exists j \in N(i) : c(i) = c(j)$
then $c(i) = \min\{l \geq 1 | \forall j \in N(i) : c(j) \neq l\}$

NUMD1: if $n1n(i) \neq \mid N(i) \mid$
then $n1n(i) = \mid N(i) \mid$

NUMD2: if $n2n[i] \neq 1 + \sum n1n(j), \forall j \in N(i)$
then $n2n[i] = 1 + \sum n1n(j), \forall j \in N(i)$

LOWER: if $c(i) > n2n[i]$
then $c(i) = 1$

SET MIN: if i is **IMPROVED D2UCM**-alive $\bigwedge i \not\rightarrow \min\{sN(i), fN(i)\}$
then $i \rightarrow \min\{sN(i), fN(i)\}$
$p(i) = c(\min\{sN(i), fN(i)\})$

SET NULL: if i is $\neg$ **IMPROVED D2UCM**-alive $\bigwedge i \not\rightarrow NULL$
then $i \rightarrow NULL$

UPDATE: if $c_{min}(i) \neq$ CorrectValue$(i, \rightarrow)$
then $c_{min}(i) =$CorrectValue$(i, \rightarrow)$

SET FLAG: if $flag(i) \neq 1 \wedge \exists j \in N(i) : (j \rightarrow i \wedge p(j) = c(i))$

then $flag(i) = 1$

CHANGE: if $\forall k \in N(i) : k \rightarrow i \bigwedge p(k) = c(i)$
then $c(i) = CorrectColor(i)$
$flag(i) = 0$

6.3 Example

We use the same graph as the one we used for the **D2UCM** algorithm, and show that we can color the graph with fewer colors using the new algorithm. To simplify the graphs, several moves are made between each subfigure. The number at the top right corner of each node i is $n2n[i]$. The number in the bracket near the pointer of i is $p(i)$. The other symbols are the same as for the former example.

Figure 6.2(a) is the initial configuration. Each node i updates its $n2n[i]$ value, this is the configuration that is shown in Figure 6.2(b). Each node i such that $c(i) > n2n[i]$ should change its color to 1. Thus the colors of C, G, I, L, and M are all changed to 1. This is the configuration that is shown in Figure 6.2(d). Nodes B and H are **IMPROVED D2UCM**-alive, and both B and H change their pointers to point to A and update $p(B), p(H)$, $c_{min}(B)$, and $c_{min}(H)$. Then node A changes $flag(A)$ to 1. It follows that node K and F also change their pointers to point to A and update $p(K), p(F)$, $c_{min}(K)$, and $c_{min}(F)$. This is the configuration that is shown in Figure 6.2(e). Since all the neighbors of A are pointing to A, and all the p value of them are equal to $c(A)$, node A then changes its color and $flag(A)$ changes to 0. Nodes B and K are now **IMPROVED D2UCM**-alive. B changes its pointer to point to C and K changes its pointer to point to L. Then B and K update $p(B), p(K)$, $c_{min}(B)$, and $c_{min}(K)$. Both B and K then change their flag values to 1. This is the configuration that is shown in Figure 6.2(f). In Figure 6.2(c), C and L change their colors. After this, there is no privileged node, and the algorithm is stable.

The same example, using the **D2UCM** algorithm in the previous chapter, would result in the graph being colored with 10 colors. But using the **IMPROVED D2UCM** algorithm, the graph can be colored with only 7 colors.

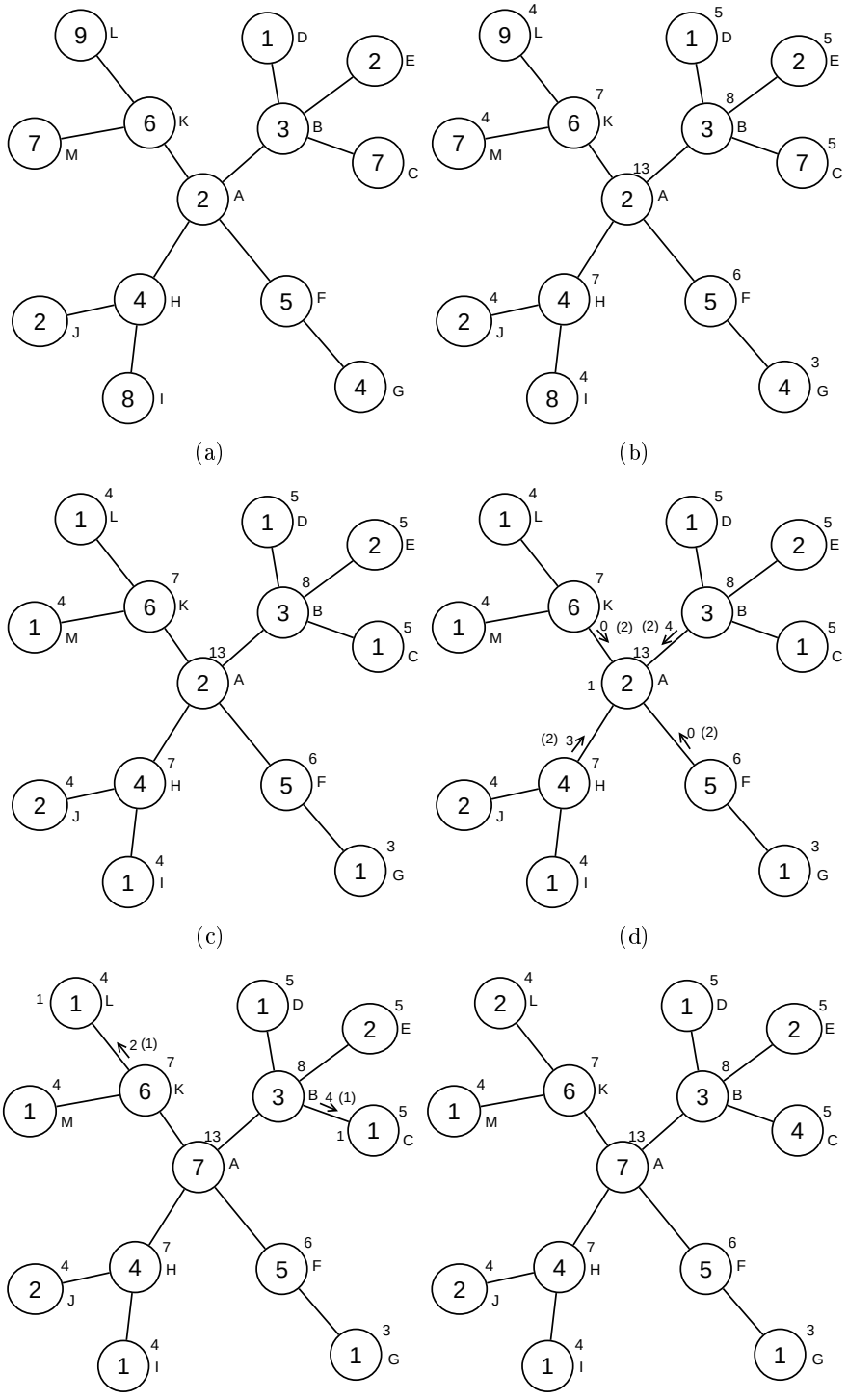


Figure 6.2: Example illustrates how the **IMPROVED D2UCM** algorithm works.

6.4 Correctness and running time

Because of time limitations, we do not prove that a stable solution of the **IMPROVED D2UCM** algorithm is a correct solution. Neither do we give the complexity of the whole algorithm. We only discuss the complexity of the new rules compare with the **D2UCM** algorithm.

Lemma 22 *Each node can make at most one NUMD1 move.*

Proof: Node i only makes a NUMD1 move when the $n1n(i)$ value is incorrect. Since i can read the information from nodes in $N(i)$, it only needs to calculate $n1n(i)$ once. Then $n1n(i)$ will never change after the NUMD1 move of i. It follows that each node can make at most one NUMD1 move.

Lemma 23 *Each node can make at most $d_i + 1$ NUMD2 moves.*

Proof: First i can make one initial NUMD2 move. Then whenever any node j in $N(i)$ updates its $n1n(j)$ value, node i will make a NUMD2 move to update $n2n[i]$. It follows from Lemma 22 that each node can make at most one NUMD1 move. Thus each node can make at most $d_i + 1$ NUMD2 moves.

Lemma 24 *Each node can make at most $d_i + 2$ LOWER moves.*

Proof: Node i only makes a LOWER move when $c(i) > n2n[i]$. After the LOWER move of i, $c(i)$ will become 1. Later $c(i)$ could change to any value in the range of $(c(i), \cdots, n2n[i], 1, 2, \cdots, c(i)-1]$, but never change to a color number that is larger than the current $n2n[i]$ again. It follows from Lemma 23 that each node can make at most $d_i + 1$ NUMD2 move. The worst case is every time after the NUMD2 move the value of $n2n[i]$ decrease, and the value $c(i)$ changed into is always larger than the new $n2n[i]$ value. Thus each node can make at most $d_i + 2$ LOWER moves.

Chapter 7

Conclusion

In this chapter we first give a summary of the results in this thesis. We then will point out some open problems.

7.1 Summary

In this thesis we have investigated the development of self-stabilizing algorithms for the distance-k coloring problem. We have presented the paper by Hedetniemi et al. [7] on how to get distance-k information using a self-stabilizing algorithm. The **DkCA** algorithm showed how to use this distance-k knowledge to solve the distance-k coloring problem to obtain a Grundy coloring. This algorithm needs to store $O(n)$ variables on each node and has a running time of $O(n^4 n^{log k})$. Since this algorithm can use quite a bit of memory, we designed the distance-2 coloring algorithm **D2UCM**. This only uses five variables on each node and has a running time of $O(mn^2)$ compared to $O(N[i])$ and $O(n^5)$ respectively in the **DkCA** algorithm when applied to distance-2. We notice that, for a distance-2 problem, the **D2UCM** algorithm is both faster and uses less memory than the **DkCA** algorithm. However the **D2UCM** algorithm can only guarantee to solve the distance-2 coloring problem, but both the number of different colors and the value of the highest color could be high. We therefore designed the **IMPROVED D2UCM** algorithm where the color numbers are limited to approximately the number of each node's distance-2 neighbors. This algorithm has to store eight variables on each node.

7.2 Open problem

Because of the time limitation on this thesis, there are still some problems of interest that we did not investigate. For the **D2UCM** algorithm, we only considered the distance-2 case. Based on the idea of this algorithm, we could have done further research to find out a way to solve the distance-k coloring problem using a constant amount of memory on each node. One should try $k = 3$ first. Also we could work on proving the correctness and bounding the running time of the **IMPROVED D2UCM** algorithm.

As we have shown, for the distance-2 coloring problem, the **D2UCM** algorithm is both faster and uses less memory than the **DkCA** algorithm. It is possible that the same idea as were used in the **D2UCM** and **IMPROVED D2UCM** algorithms could be used in designing efficient algorithms for other problems other than the coloring problem.

Bibliography

[1] T. F. Coleman and J. J More. Estimation of sparse jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal.*, 20:187 – 209, 1983.

[2] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Comm. ACM*, 17 (11):643 – 644, 1974.

[3] S. Dolev. *Self-Stabilization.* 2000.

[4] M. Gairing, W. Goddard, S. T. Hedetniemi, P. Kristiansen, and A. A. McRae. Distance-two information in self-stabilizing algorithms. *Parallel. Proc. Lett.*, pages 387 – 398, 2004.

[5] M. R. Garey and D. S. Johnson. Computers and intractability. *Freeman*, 1979.

[6] S. Ghosh and M. H. Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distrib. Comput.*, 7:55 – 59, 1993.

[7] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and V. Trevisan. Distance-k knowledge in self-stabilizing algorithms. 2006.

[8] S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Linear time self-stabilizing colorings. *Inf. Proc. Lett.*, pages 251 – 255, 2003.

[9] S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. *Comput. Math. Appl.*, 46 (5 - 6):805 – 811, 2003.

[10] T. R. Jensen and B. Toft. *Graph Coloring Problems.* 1995.

[11] S. Sur and P. K. Srimani. A self-stabilizing algorithm for coloring bipartite graphs. *Inform. Sci.*, 49:219 – 227, 1993.