



Theory of Distributed Computing I

Algorithms and Lower Bounds

(Part 2: Message Passing)

Dr. Stefan Schmid

Co-lecturer (shared memory): Dr. Petr Kuznetsov

Thanks to Prof. Dr. Roger Wattenhofer for basis of manuscript!

Spring 2011

Introduction

In the second part of this course, we turn our attention to distributed computing in *networks*, i.e., where computers (or processors) communicate via message passing along the links of the communication network. Typical “old school” examples are parallel computers, or the Internet whose architecture often follows the design principle of decentralization to avoid bottlenecks and single points of failures. More recent application examples of distributed systems include peer-to-peer systems, sensor networks, and multi-core architectures.

These applications have in common that, like in the first part of the course, many processors or entities (often called nodes) are active in the system at any moment. The nodes have certain degrees of freedom: they may have their own hardware, their own code, and sometimes their own independent task. Nevertheless, the nodes may share common resources and information, and, in order to solve a problem that concerns several—or maybe even all—nodes, coordination is necessary. Despite these commonalities, a peer-to-peer system, for example, is quite different from a multi-core architecture. In some systems the nodes operate synchronously, in other systems they operate asynchronously. There are simple homogeneous systems, and heterogeneous systems where different types of nodes, potentially with different capabilities, objectives etc., need to interact. There are different communication techniques. Sometimes the communication infrastructure is tailor-made for an application, sometimes one has to work with any given infrastructure. The nodes in a system sometimes work together to solve a global task, occasionally the nodes are autonomous agents that have their own agenda and compete for common resources.

This second part of the course continues to study basic principles of distributed computing. In particular, we are interested in the following challenges in distributed systems with message passing:

1. **Network Design:** What is a “good” communication network for message passing?
2. **Communication:** What is the cost of communication? (Often communication cost dominates the cost of local processing or storage. Sometimes we may even assume that everything but communication is free!)
3. **Locality/Scalability:** Networks keep growing. Luckily, global information is not always needed to solve a task, often it is sufficient if nodes talk to their neighbors. We will address the fundamental question in distributed computing whether a local solution is possible for a wide range of problems.

Unfortunately, a complete coverage of this active and exciting field is impossible, and we focus on highlighting common themes and techniques. We will set off with some evergreen problems such as LEADER ELECTION that give a feeling for the main setting and the tradeoffs. We will also study classic problems from theoretical computer science, such as MAXIMAL INDEPENDENT SET, from the new perspective of distributed computing. A highlight of the course might be the VERTEX COLORING algorithms: vertex coloring is the problem of how to assign colors to nodes in a network, such that no two adjacent nodes have the same color. Colorings are for example important in wireless networks: nodes which are close to each other should use different frequency bands (i.e., colors) in order to avoid interference and minimize retransmissions. As we will see, the coloring problem and the independent set problem are closely related, and solutions for one problem can be used for the other.

Finally, if time permits, we take a look at *social networks*. Interestingly, today we still do not know much about the structure and dynamics of these networks, and research is very active in this field. Moreover, due to the nice properties of these networks, researchers have proposed to build computer networks similarly to social networks.

Have fun!

Chapter 1

Communication Networks

1.1 Example: Peer-to-Peer

The term *peer-to-peer* (P2P) is ambiguous and used in a variety of different contexts, such as:

- In popular media coverage, P2P is often synonymous to software or protocols that allow users to “share” files, often of dubious origin. In the early days, P2P users mostly shared music, pictures, and software; nowadays books, movies or tv shows have caught on.
- In academia, the term P2P is used mostly in two ways. A narrow view essentially defines P2P as the “theory behind file sharing protocols”. In other words, how do Internet hosts need to be organized in order to deliver a search engine to find (file sharing) content efficiently? A popular term is “distributed hash table” (DHT), a distributed data structure that implements such a content search engine. A DHT should support at least a search (for a key) and an insert (key, object) operation. A DHT has many applications beyond file sharing, e.g., the Internet domain name system (DNS).
- A broader view generalizes P2P beyond file sharing: Indeed, there is a growing number of applications operating outside the juridical gray area, e.g., P2P Internet telephony à la Skype, P2P mass player games on video consoles connected to the Internet, P2P live video streaming as in Zattoo or StreamForge, or P2P social storage such as Wuala. So, again, what is P2P?! Still not an easy question... Trying to account for the new applications beyond file sharing, one might define P2P as a large-scale distributed system that operates without a central server bottleneck. However, with this definition almost everything we learn in this course is P2P! Moreover, according to this definition early-day file sharing applications such as Napster (1999) that essentially made the term P2P popular would not be P2P! On the other hand, the plain old telephone system or the world wide web do fit the P2P definition...
- From a different viewpoint, the term P2P may also be synonymous for privacy protection, as various P2P systems such as Freenet allow publishers of information to remain anonymous and uncensored. (Studies show

that these freedom-of-speech P2P networks do not feature a lot of content against oppressive governments; indeed the majority of text documents seem to be about illicit drugs, not to speak about the type of content in audio or video files.)

So we cannot hope for a single well-fitting definition of P2P, as some of them even contradict. In the following we mostly employ the academic viewpoints (second and third definition above). In this context, it is generally believed that P2P will have an influence on the future of the Internet. The P2P paradigm promises to give better scalability, availability, reliability, fairness, incentives, privacy, and security, just about everything researchers expect from a future Internet architecture. As such it is not surprising that new “clean slate” Internet architecture proposals often revolve around P2P concepts.

One might naively assume that for instance scalability is not an issue in today’s Internet, as even most popular web pages are generally highly available. However, this is not really because of our well-designed Internet architecture, but rather due to the help of so-called overlay networks: The Google website for instance manages to respond so reliably and quickly because Google maintains a large distributed infrastructure, essentially a P2P system. Similarly companies like Akamai sell “P2P functionality” to their customers to make today’s user experience possible in the first place. Quite possibly today’s P2P applications are just testbeds for tomorrow’s Internet architecture.

1.2 P2P Architecture Variants

Several P2P architectures are known:

- Client/Server goes P2P: Even though Napster is known to be the first P2P system (1999), by today’s standards its architecture would not deserve the label P2P anymore. Napster clients accessed a central server that managed all the information of the shared files, i.e., which file was to be found on which client. Only the downloading process itself was between clients (“peers”) directly, hence peer-to-peer. In the early days of Napster the load of the server was relatively small, so the simple Napster architecture made a lot of sense. Later on, it became clear that the server would eventually be a bottleneck, and more so an attractive target for an attack. Indeed, eventually a judge ruled the server to be shut down, in other words, he conducted a juridical denial of service attack.
- Unstructured P2P: The Gnutella protocol is the anti-thesis of Napster, as it is a fully decentralized system, with no single entity having a global picture. Instead each peer would connect to a random sample of other peers, constantly changing the neighbors of this virtual overlay network by exchanging neighbors with neighbors of neighbors. (In such a system it is part of the challenge to find a decentralized way to even discover a first neighbor; this is known as the bootstrap problem. To solve it, usually some random peers of a list of well-known peers are contacted first.) When searching for a file, the request was being flooded in the network. Indeed, since users often turn off their client once they downloaded their content there usually is a lot of *churn* (peers joining and leaving at high rates) in

a P2P system, so selecting the right “random” neighbors is an interesting research problem by itself. However, unstructured P2P architectures such as Gnutella have a major disadvantage, namely that each search will cost m messages, m being the number of virtual edges in the architecture. In other words, such an unstructured P2P architecture will not scale.

- **Hybrid P2P:** The synthesis of client/server architectures such as Napster and unstructured architectures such as Gnutella are hybrid architectures. Some powerful peers are promoted to so-called superpeers (or, similarly, trackers). The set of superpeers may change over time, and taking down a fraction of superpeers will not harm the system. Search requests are handled on the superpeer level, resulting in much less messages than in flat/homogeneous unstructured systems. Essentially the superpeers together provide a more fault-tolerant version of the Napster server, all regular peers connect to a superpeer. As of today, many popular P2P systems have such a hybrid architecture, carefully trading off reliability and efficiency, but essentially not using any fancy algorithms and techniques.
- **Structured P2P:** Inspired by the early success of Napster, the academic world started to look into the question of efficient file sharing. Indeed, even earlier, in 1997, Plaxton, Rajaraman, and Richa proposed a hypercubic architecture for P2P systems. This was a blueprint for many so-called structured P2P architecture proposals, such as Chord, CAN, Pastry, Tapestry, Viceroy, Kademia, Koorde, SkipGraph, SkipNet, etc. In practice structured P2P architectures are not so popular yet, apart from the Kad (from Kademia) architecture which comes for free with the eMule client, or the distributed trackers used in BitTorrent (with millions of peers constituting the network which also resembles Kademia). Indeed, also the Plaxton et al. paper was standing on the shoulders of giants. Some of its eminent precursors are:
 - Research on linear and consistent hashing, e.g., the paper “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web” by Karger et al. (co-authored also by the late Daniel Lewin from Akamai), 1997.
 - Research on locating shared objects, e.g., the papers “Sparse Partitions” or “Concurrent Online Tracking of Mobile Users” by Awerbuch and Peleg, 1990 and 1991.
 - Work on so-called compact routing: The idea is to construct routing tables such that there is a trade-off between memory (size of routing tables) and stretch (quality of routes), e.g., “A trade-off between space and efficiency for routing tables” by Peleg and Upfal, 1988.
 - ...and even earlier: hypercubic networks, see next section!

1.3 Hypercubic Networks

This section reviews some popular families of network topologies. These topologies are used in countless application domains, e.g., in classic parallel computers or telecommunication networks, or more recently (as said above) in P2P computing. In the following, let us assume an all-to-all communication model, i.e.,

each node can set up direct communication links to arbitrary other nodes. Such a virtual network is called an *overlay network*, or in this context, P2P architecture. In this section we present a few overlay topologies of general interest.

The most basic network topologies used in practice are *trees*, *rings*, *grids* or *tori*. Many other suggested networks are simply combinations or derivatives of these. The advantage of trees is that the routing is very easy: for every source-destination pair there is only one possible simple path. However, since the root of a tree is usually a severe bottleneck, so-called *fat trees* have been used. These trees have the property that every edge connecting a node v to its parent u has a capacity that is equal to all leaves of the subtree rooted at v . See Figure 1.1 for an example.

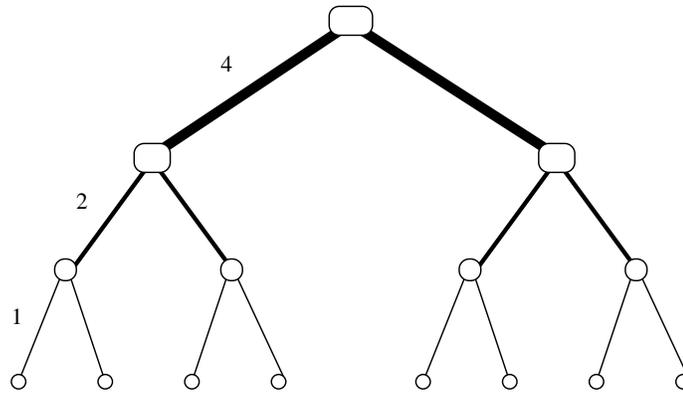


Figure 1.1: The structure of a fat tree.

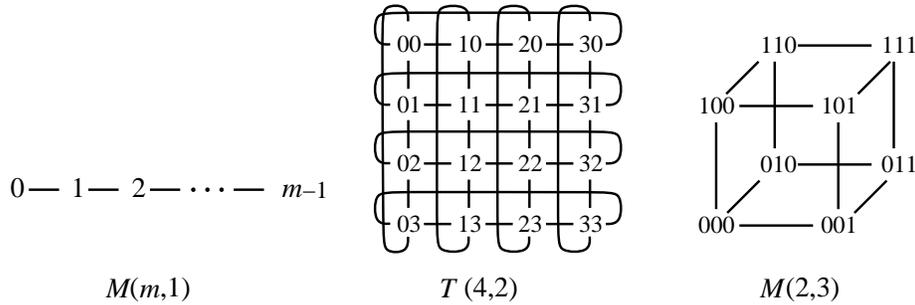
Remarks:

- Fat trees belong to a family of networks that require edges of non-uniform capacity to be efficient. Easier to build are networks with edges of uniform capacity. This is usually the case for grids and tori. Unless explicitly mentioned, we will henceforth treat all edges to be of capacity 1. In the following, $[x]$ means the set $\{0, \dots, x-1\}$.

Definition 1.1 (Torus, Mesh). *Let $m, d \in \mathbb{N}$. The (m, d) -mesh $M(m, d)$ is a graph with node set $V = [m]^d$ and edge set*

$$E = \left\{ \{(a_1, \dots, a_d), (b_1, \dots, b_d)\} \mid a_i, b_i \in [m], \sum_{i=1}^d |a_i - b_i| = 1 \right\}.$$

The (m, d) -torus $T(m, d)$ is a graph that consists of an (m, d) -mesh and additionally wrap-around edges from nodes $(a_1, \dots, a_{i-1}, m, a_{i+1}, \dots, a_d)$ to nodes $(a_1, \dots, a_{i-1}, 1, a_{i+1}, \dots, a_d)$ for all $i \in \{1, \dots, d\}$ and all $a_j \in [m]$ with $j \neq i$. In other words, we take the expression $a_i - b_i$ in the sum modulo m prior to computing the absolute value. $M(m, 1)$ is also called a line, $T(m, 1)$ a cycle, and $M(2, d) = T(2, d)$ a d -dimensional hypercube. Figure 1.2 presents a linear array, a torus, and a hypercube.

Figure 1.2: The structure of $M(m, 1)$, $T(4, 2)$, and $M(2, 3)$.**Remarks:**

- Routing on mesh, torus, and hypercube is trivial. On a d -dimensional hypercube, to get from a source bitstring s to a target bitstring d one only needs to fix each “wrong” bit, one at a time; in other words, if the source and the target differ by k bits, there are $k!$ routes with k hops.
- The hypercube can be used for a structured P2P architecture (a distributed hash table (DHT)): We have n nodes, n for simplicity being a power of 2, i.e., $n = 2^d$. As in the hypercube, each node gets a unique d -bit ID, and each node connects to d other nodes, i.e., the nodes that have IDs differing in exactly one bit. Now we use a globally known hash function f , mapping file names to long bit strings; SHA-1 is popular in practice, providing 160 bits. Let f_d denote the first d bits (prefix) of the bitstring produced by f . If a node is searching for file name X , it routes a request message $f(X)$ to node $f_d(X)$. Clearly, node $f_d(X)$ can only answer this request if all files with hash prefix $f_d(X)$ have been previously registered at node $f_d(X)$.
- There are a few issues which need to be addressed before our DHT works, in particular churn (nodes joining and leaving without notice). To deal with churn the system needs some level of replication, i.e., a number of nodes which are responsible for each prefix such that failure of some nodes will not compromise the system. In addition there are other issues (e.g., security, efficiency) which can be addressed to improve the system. Delay efficiency for instance is already considered in the seminal paper by Plaxton et al. These issues are beyond the scope of this lecture.
- The hypercube has many derivatives, the so-called *hypercubic networks*. Among these are the butterfly, cube-connected-cycles, shuffle-exchange, and de Bruijn graph. We start with the butterfly, which is basically a “rolled out” hypercube (hence directly providing replication!).

Definition 1.2 (Butterfly). *Let $d \in \mathbb{N}$. The d -dimensional butterfly $BF(d)$ is a graph with node set $V = [d + 1] \times [2]^d$ and an edge set $E = E_1 \cup E_2$ with*

$$E_1 = \{(i, \alpha), (i + 1, \alpha)\} \mid i \in [d], \alpha \in [2]^d\}$$

and

$$E_2 = \{ \{ (i, \alpha), (i+1, \beta) \} \mid i \in [d], \alpha, \beta \in [2]^d, \alpha \text{ and } \beta \text{ differ only at the } i^{\text{th}} \text{ position} \} .$$

A node set $\{(i, \alpha) \mid \alpha \in [2]^d\}$ is said to form level i of the butterfly. The d -dimensional wrap-around butterfly $W\text{-}BF(d)$ is defined by taking the $BF(d)$ and identifying level d with level 0 .

Remarks:

- Figure 1.3 shows the 3-dimensional butterfly $BF(3)$. The $BF(d)$ has $(d+1)2^d$ nodes, $2d \cdot 2^d$ edges and degree 4. It is not difficult to check that combining the node sets $\{(i, \alpha) \mid i \in [d]\}$ into a single node results in the hypercube.
- Butterflies have the advantage of a constant node degree over hypercubes, whereas hypercubes feature more fault-tolerant routing.
- Although butterflies are used in the P2P context (e.g. Viceroy), they have been used decades earlier for communication switches. The well-known Benes network is nothing but two back-to-back butterflies. And indeed, butterflies (and other hypercubic networks) are even older than that; students familiar with fast fourier transform (FFT) may recognize the structure. Every year there is a new application for which a hypercubic network is the perfect solution!
- Indeed, hypercubic networks are related. Since all structured P2P architectures are based on hypercubic networks, they in turn are all related.
- Next we define the cube-connected-cycles network. It only has a degree of 3 and it results from the hypercube by replacing the corners by cycles.

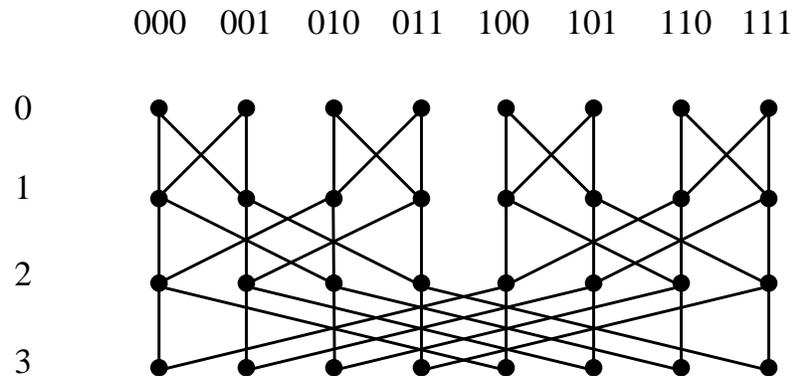


Figure 1.3: The structure of $BF(3)$.

Definition 1.3 (Cube-Connected-Cycles). *Let $d \in \mathbb{N}$. The cube-connected-cycles network $CCC(d)$ is a graph with node set $V = \{(a, p) \mid a \in [2]^d, p \in [d]\}$ and edge set*

$$E = \{ \{(a, p), (a, (p + 1) \bmod d)\} \mid a \in [2]^d, p \in [d] \} \\ \cup \{ \{(a, p), (b, p)\} \mid a, b \in [2]^d, p \in [d], a = b \text{ except for } a_p \} .$$

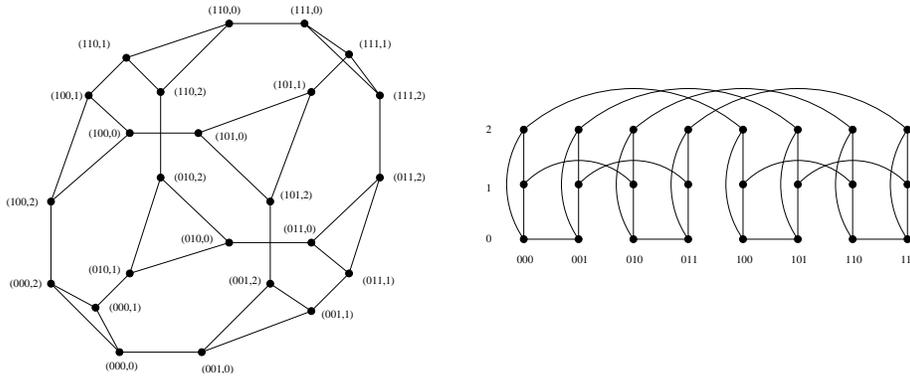


Figure 1.4: The structure of $CCC(3)$.

Remarks:

- Two possible representations of a CCC can be found in Figure 1.4.
- The shuffle-exchange is yet another way of transforming the hypercubic interconnection structure into a constant degree network.

Definition 1.4 (Shuffle-Exchange). *Let $d \in \mathbb{N}$. The d -dimensional shuffle-exchange $SE(d)$ is defined as an undirected graph with node set $V = [2]^d$ and an edge set $E = E_1 \cup E_2$ with*

$$E_1 = \{ \{(a_1, \dots, a_d), (a_1, \dots, \bar{a}_d)\} \mid (a_1, \dots, a_d) \in [2]^d, \bar{a}_d = 1 - a_d \}$$

and

$$E_2 = \{ \{(a_1, \dots, a_d), (a_d, a_1, \dots, a_{d-1})\} \mid (a_1, \dots, a_d) \in [2]^d \} .$$

Figure 1.5 shows the 3- and 4-dimensional shuffle-exchange graph.

Definition 1.5 (DeBruijn). *The b -ary DeBruijn graph of dimension d $DB(b, d)$ is an undirected graph $G = (V, E)$ with node set $V = \{v \in [b]^d\}$ and edge set E that contains all edges $\{v, w\}$ with the property that $w \in \{(x, v_1, \dots, v_{d-1}) : x \in [b]\}$, where $v = (v_1, \dots, v_d)$.*

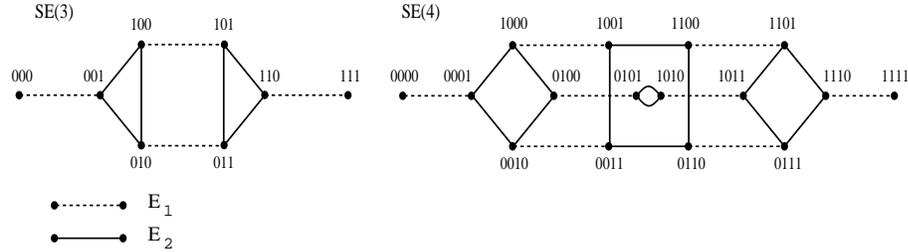
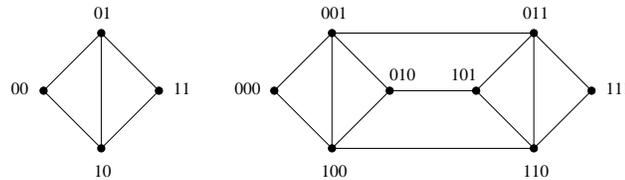


Figure 1.5: The structure of SE(3) and SE(4).

Figure 1.6: The structure of $DB(2, 2)$ and $DB(2, 3)$.**Remarks:**

- Two examples of a DeBruijn graph can be found in Figure 1.6. The DeBruijn graph is the basis of the Koorde P2P architecture.
- There are some data structures which also qualify as hypercubic networks. An obvious example is the Chord P2P architecture, which uses a slightly different hypercubic topology. A less obvious (and therefore good) example is the skip list, the balanced binary search tree for the lazy programmer:

Definition 1.6 (Skip List). *The skip list is an ordinary ordered linked list of objects, augmented with additional forward links. The ordinary linked list is the level 0 of the skip list. In addition, every object is promoted to level 1 with probability $1/2$. As for level 0, all level 1 objects are connected by a linked list. In general, every object on level i is promoted to the next level with probability $1/2$. A special start-object points to the smallest/first object on each level.*

Remarks:

- Search, insert, and delete can be implemented in $O(\log n)$ expected time in a skip list, simply by jumping from higher levels to lower ones when overshooting the searched position. Also, the amortized memory cost of each object is constant, as on average an object only has two forward pointers.
- The randomization can easily be discarded, by deterministically promoting a constant fraction of objects of level i to level $i + 1$, for all i . When inserting or deleting, object o simply checks whether its left and right

level i neighbors are being promoted to level $i + 1$. If none of them is, promote object o itself. Essentially we establish a maximal independent set on each level, hence at least every third and at most every second object is promoted.

- There are obvious variants of the skip list, e.g., the skip graph. Instead of promoting only half of the nodes to the next level, we always promote all the nodes, similarly to a balanced binary tree: All nodes are part of the root level of the binary tree. Half the nodes are promoted left, and half the nodes are promoted right, on each level. Hence on level i we have 2^i lists (or, more symmetrically: rings) of about $n/2^i$ objects. This is pretty much what we need for a nice hypercubic P2P architecture.
- One important goal in choosing a topology for a network is that it has a small diameter. The following theorem presents a lower bound for this.

Theorem 1.7. *Every graph of maximum degree $d > 2$ and size n must have a diameter of at least $\lceil (\log n)/(\log(d - 1)) \rceil - 2$.*

Proof. Suppose we have a graph $G = (V, E)$ of maximum degree d and size n . Start from any node $v \in V$. In a first step at most d other nodes can be reached. In two steps at most $d \cdot (d - 1)$ additional nodes can be reached. Thus, in general, in at most k steps at most

$$1 + \sum_{i=0}^{k-1} d \cdot (d - 1)^i = 1 + d \cdot \frac{(d - 1)^k - 1}{(d - 1) - 1} \leq \frac{d \cdot (d - 1)^k}{d - 2}$$

nodes (including v) can be reached. This has to be at least n to ensure that v can reach all other nodes in V within k steps. Hence,

$$(d - 1)^k \geq \frac{(d - 2) \cdot n}{d} \quad \Leftrightarrow \quad k \geq \log_{d-1}((d - 2) \cdot n/d).$$

Since $\log_{d-1}((d - 2)/d) > -2$ for all $d > 2$, this is true only if $k \geq \lceil (\log n)/(\log(d - 1)) \rceil - 2$. \square

Remarks:

- In other words, constant-degree hypercubic networks feature an asymptotically optimal diameter.
- There are a few other interesting graph classes, e.g., expander graphs (an expander graph is a sparse graph which has high connectivity properties, that is, from every not too large subset of nodes you are connected to a larger set of nodes), or small-world graphs (popular representations of social networks). At first sight hypercubic networks seem to be related to expanders and small-world graphs, but they are not.

1.4 DHT & Churn

As written earlier, a DHT essentially is a hypercubic structure with nodes having identifiers such that they span the ID space of the objects to be stored. We

described the straightforward way how the ID space is mapped onto the peers for the hypercube. Other hypercubic structures may be more complicated: The butterfly network, for instance, may directly use the $d + 1$ layers for replication, i.e., all the $d + 1$ nodes with the same ID are responsible for the same hash prefix. For other hypercubic networks, e.g., the pancake graph, assigning the object space to peer nodes may be more difficult.

In general a DHT has to withstand churn. Usually, peers are under control of individual users who turn their machines on or off at any time. Such peers join and leave the P2P system at high rates (“churn”), a problem that is not existent in orthodox distributed systems, hence P2P systems fundamentally differ from old-school distributed systems where it is assumed that the nodes in the system are relatively stable. In traditional distributed systems a single unavailable node is a minor disaster: all the other nodes have to get a consistent view of the system again, essentially they have to reach consensus which nodes are available. In a P2P system there is usually so much churn that it is impossible to have a consistent view at any time.

Most P2P systems in the literature are analyzed against an adversary that can crash a fraction of random peers. After crashing a few peers the system is given sufficient time to recover again. However, this seems unrealistic. The scheme sketched in this section significantly differs from this in two major aspects. First, we assume that joins and leaves occur in a worst-case manner. We think of an adversary that can remove and add a bounded number of peers; it can choose which peers to crash and how peers join. We assume that a joining peer knows a peer which already belongs to the system. Second, the adversary does not have to wait until the system is recovered before it crashes the next batch of peers. Instead, the adversary can constantly crash peers, while the system is trying to stay alive. Indeed, the system is *never fully repaired* but *always fully functional*. In particular, the system is resilient against an adversary that continuously attacks the “weakest part” of the system. The adversary could for example insert a crawler into the P2P system, learn the topology of the system, and then repeatedly crash selected peers, in an attempt to partition the P2P network. The system counters such an adversary by continuously moving the remaining or newly joining peers towards the sparse areas.

Clearly, we cannot allow the adversary to have unbounded capabilities. In particular, in any constant time interval, the adversary can at most add and/or remove $O(\log n)$ peers, n being the total number of peers currently in the system. This model covers an adversary which repeatedly takes down machines by a distributed denial of service attack, however only a logarithmic number of machines at each point in time. The algorithm relies on messages being delivered timely, in at most constant time between any pair of operational peers, i.e., the synchronous model. Using the trivial synchronizer this is not a problem. We only need bounded message delays in order to have a notion of time which is needed for the adversarial model. The duration of a round is then proportional to the propagation delay of the slowest message.

In the remainder of this section, we give a sketch of the system: For simplicity, the basic structure of the P2P system is a hypercube. Each peer is part of a distinct hypercube node; each hypercube node consists of $\Theta(\log n)$ peers. Peers have connections to other peers of their hypercube node and to peers of

the neighboring hypercube nodes.¹ Because of churn, some of the peers have to change to another hypercube node such that up to constant factors, all hypercube nodes own the same number of peers at all times. If the total number of peers grows or shrinks above or below a certain threshold, the dimension of the hypercube is increased or decreased by one, respectively.

The balancing of peers among the hypercube nodes can be seen as a dynamic token distribution problem on the hypercube. Each node of the hypercube has a certain number of tokens, the goal is to distribute the tokens along the edges of the graph such that all nodes end up with the same or almost the same number of tokens. While tokens are moved around, an adversary constantly inserts and deletes tokens. See also Figure 1.7.

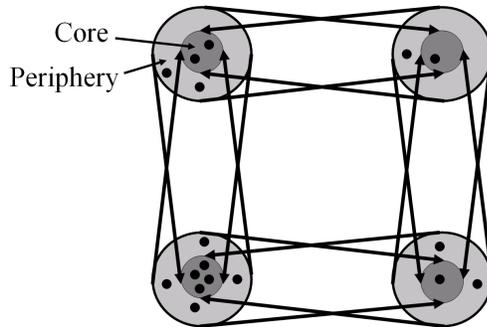


Figure 1.7: A simulated 2-dimensional hypercube with four nodes, each consisting of several peers. Also, all the peers are either in the core or in the periphery of a node. All peers within the same node are completely connected to each other, and additionally, all peers of a node are connected to the core peers of the neighboring nodes. Only the core peers store data items, while the peripheral peers move between the nodes to balance biased adversarial changes.

In summary, the P2P system builds on two basic components: i) an algorithm which performs the described dynamic token distribution and ii) an information aggregation algorithm which is used to estimate the number of peers in the system and to adapt the dimension of the hypercube accordingly:

Theorem 1.8 (DHT with Churn). *We have a fully scalable, efficient P2P system which tolerates $O(\log n)$ worst-case joins and/or crashes per constant time interval. As in other P2P systems, peers have $O(\log n)$ neighbors, and the usual operations (e.g., search, insert) take time $O(\log n)$.*

Remarks:

- Indeed, handling churn is only a minimal requirement to make a P2P system work. Later studies proposed more elaborate architectures which can also handle other security issues, e.g., privacy or Byzantine attacks.

¹Having a logarithmic number of hypercube neighbor nodes, each with a logarithmic number of peers, means that each peers has $\Theta(\log^2 n)$ neighbor peers. However, with some additional bells and whistles one can achieve $\Theta(\log n)$ neighbor peers.

- It is surprising that unstructured (in fact, hybrid) P2P systems dominate structured P2P systems in the real world. One would think that structured P2P systems have advantages, in particular their efficient logarithmic data lookup. On the other hand, unstructured P2P networks are simpler, in particular in light of non-exact queries.

Chapter 2

Leader Election

2.1 Distributed Algorithms and Complexity

In the second part of this course we will often model the distributed system as a network or graph, and study protocols in which nodes (i.e., the processors) can only communicate with their neighbors to perform certain tasks. We are often interested in the following synchronous model or algorithm.

Definition 2.1 (Synchronous Distributed Algorithm). *In a synchronous algorithm, nodes operate in synchronous rounds. In each round, each processor executes the following steps:*

1. *Do some local computation (of reasonable “local complexity”).*
2. *Send messages to neighbors in graph (of reasonable size).*
3. *Receive messages (that were sent by neighbors in step 2 of the same round).*

Remarks:

- Any other step ordering is fine.

The other cornerstone model is the asynchronous algorithm.

Definition 2.2 (Asynchronous Distributed Algorithm). *In the asynchronous model, algorithms are event driven (“upon receiving message . . . , do . . .”). Processors cannot access a global clock. A message sent from one processor to another will arrive in finite but unbounded time.*

Remarks:

- The asynchronous model and the synchronous model (Definition 2.1) are the cornerstone models in distributed computing. As they do not necessarily reflect reality there are several models in between synchronous and asynchronous. However, from a theoretical point of view the synchronous and the asynchronous model are the most interesting ones (because every other model is in between these extremes).
- Note that in the asynchronous model, messages that take a longer path may arrive earlier.

In order to evaluate an algorithm, apart from the local complexity mentioned above, we consider the following metrics.

Definition 2.3 (Time Complexity). *For synchronous algorithms (as defined in 2.1) the time complexity is the number of rounds until the algorithm terminates.*

Definition 2.4 (Time Complexity). *For asynchronous algorithms (as defined in 2.1) the time complexity is the number of time units from the start of the execution to its completion in the worst case (every legal input, every execution scenario), assuming that each message has a delay of at most one time unit.*

Remarks:

- You cannot use the maximum delay in the algorithm design. In other words, the algorithm has to be correct even if there is no such delay upper bound.

Definition 2.5 (Message Complexity). *The message complexity of a synchronous and asynchronous algorithm is determined by the number of messages exchanged (again every legal input, every execution scenario).*

2.2 Anonymous Leader Election

Some algorithms (e.g., for medium access) ask for a special node, a so-called “leader”. Computing a leader is a most simple form of symmetry breaking. Algorithms based on leaders do generally not exhibit a high degree of parallelism, and therefore often suffer from poor (parallel) time complexity. However, sometimes it is still useful to have a leader to make critical decisions in an easy (though non-distributed!) way.

The process of choosing a leader is known as *leader election*. Although leader election is a simple form of symmetry breaking, there are some remarkable issues that allow us to introduce notable computational models.

In this chapter we concentrate on the ring topology. The ring is the “dro-sophila” of distributed computing as many interesting challenges already reveal the root of the problem in the special case of the ring. Paying special attention to the ring also makes sense from a practical point of view as some real world systems are based on a ring topology, e.g., the token ring standard for local area networks.

Problem 2.6 (Leader Election). *Each node eventually decides whether it is a leader or not, subject to the constraint that there is exactly one leader.*

Remarks:

- More formally, nodes are in one of three states: *undecided*, *leader*, *not leader*. Initially every node is in the undecided state. When leaving the undecided state, a node goes into a *final state* (leader or not leader).

Definition 2.7 (Anonymous). *A system is anonymous if nodes do not have unique identifiers.*

Definition 2.8 (Uniform). *An algorithm is called uniform if the number of nodes n is not known to the algorithm (to the nodes, if you wish). If n is known, the algorithm is called non-uniform.*

Whether a leader can be elected in an anonymous system depends on whether the network is symmetric (ring, complete graph, complete bipartite graph, etc.) or asymmetric (star, single node with highest degree, etc.). Simplifying slightly, in this context a symmetric graph is a graph in which the extended neighborhood of each node has the same structure. We will now show that non-uniform anonymous leader election for synchronous rings is impossible. The idea is that in a ring, symmetry can always be maintained.

Lemma 2.9. *After round k of any deterministic algorithm on an anonymous ring, each node is in the same state s_k .*

Proof by induction: All nodes start in the same state. A round in a synchronous algorithm consists of the three steps sending, receiving, local computation (see Definition 2.1). All nodes send the same message(s), receive the same message(s), do the same local computation, and therefore end up in the same state.

Theorem 2.10 (Anonymous Leader Election). *Deterministic leader election in an anonymous ring is impossible.*

Proof (with Lemma 2.9): If one node ever decides to become a leader (or a non-leader), then every other node does so as well, contradicting the problem specification 2.6 for $n > 1$. This holds for non-uniform algorithms, and therefore also for uniform algorithms. Furthermore, it holds for synchronous algorithms, and therefore also for asynchronous algorithms.

Remarks:

- Sense of direction is the ability of nodes to distinguish neighbor nodes in an anonymous setting. In a ring, for example, a node can distinguish the clockwise and the counterclockwise neighbor. Sense of direction does not help in anonymous leader election.
- Theorem 2.10 also holds for other symmetric network topologies (e.g., complete graphs, complete bipartite graphs, ...).
- Note that Theorem 2.10 does not hold for randomized algorithms; if nodes are allowed to toss a coin, symmetries can be broken.

2.3 Asynchronous Ring

We first concentrate on the asynchronous model from Definition 2.2. Throughout this section we assume non-anonymity; each node has a unique identifier as proposed in Assumption 4.2:

Assumption 2.11 (Node Identifiers). *Each node has a unique identifier, e.g., its IP address. We usually assume that each identifier consists of only $\log n$ bits if the system has n nodes.*

Having ID's seems to lead to a trivial leader election algorithm, as we can simply elect the node with, e.g., the highest ID.

Algorithm 1 Clockwise

-
- 1: **Each node** v executes the following code:
 - 2: v sends a message with its identifier (for simplicity also v) to its clockwise neighbor. {If node v already received a message w with $w > v$, then node v can skip this step; if node v receives its first message w with $w < v$, then node v will immediately send v .}
 - 3: **if** v receives a message w with $w > v$ **then**
 - 4: v forwards w to its clockwise neighbor
 - 5: v decides not to be the leader, if it has not done so already.
 - 6: **else if** v receives its own identifier v **then**
 - 7: v decides to be the leader
 - 8: **end if**
-

Theorem 2.12 (Analysis of Algorithm 1). *Algorithm 1 is correct. The time complexity is $O(n)$. The message complexity is $O(n^2)$.*

Proof: Let node z be the node with the maximum identifier. Node z sends its identifier in clockwise direction, and since no other node can swallow it, eventually a message will arrive at z containing it. Then z declares itself to be the leader. Every other node will declare non-leader at the latest when forwarding message z . Since there are n identifiers in the system, each node will at most forward n messages, giving a message complexity of at most n^2 . We start measuring the time when the first node that “wakes up” sends its identifier. For asynchronous time complexity (Definition 2.4) we assume that each message takes at most one time unit to arrive at its destination. After at most $n - 1$ time units the message therefore arrives at node z , waking z up. Routing the message z around the ring takes at most n time units. Therefore node z decides no later than at time $2n - 1$. Every other node decides before node z .

Remarks:

- Note that in Algorithm 1 nodes need to distinguish between clockwise and counterclockwise neighbors. In fact they do not: It is okay to simply send your own identifier to any neighbor, and forward a message m to the neighbor you did not receive the message m from. So nodes only need to be able to distinguish their two neighbors.
- Can we improve this algorithm?

Theorem 2.13 (Analysis of Algorithm 2). *Algorithm 2 is correct. The time complexity is $O(n)$. The message complexity is $O(n \log n)$.*

Proof: Correctness is as in Theorem 2.12. The time complexity is $O(n)$ since the node with maximum identifier z sends messages with round-trip times $2, 4, 8, 16, \dots, 2 \cdot 2^k$ with $k \leq \log(n + 1)$. (Even if we include the additional wake-up overhead, the time complexity stays linear.) Proving the message complexity is slightly harder: if a node v manages to survive round r , no other node in distance 2^r (or less) survives round r . That is, node v is the only node in its 2^r -neighborhood that remains active in round $r + 1$. Since this is the same for every node, less than $n/2^r$ nodes are active in round $r + 1$. Being active in round

Algorithm 2 Radius Growth (For readability we provide pseudo-code only; for a formal version please consult [Attiya/Welch Alg. 3.1])

- 1: **Each node** v does the following:
 - 2: Initially all nodes are *active*. {all nodes may still become leaders}
 - 3: Whenever a node v sees a message w with $w > v$, then v decides to not be a leader and becomes *passive*.
 - 4: Active nodes search in an exponentially growing neighborhood (clockwise and counterclockwise) for nodes with higher identifiers, by sending out *probe* messages. A probe message includes the ID of the original sender, a bit whether the sender can still become a leader, and a time-to-live number (*TTL*). The first probe message sent by node v includes a TTL of 1.
 - 5: Nodes (active or passive) receiving a probe message decrement the TTL and forward the message to the next neighbor; if their ID is larger than the one in the message, they set the leader bit to zero, as the probing node does not have the maximum ID. If the TTL is zero, probe messages are returned to the sender using a *reply* message. The reply message contains the ID of the receiver (the original sender of the probe message) and the leader-bit. Reply messages are forwarded by all nodes until they reach the receiver.
 - 6: Upon receiving the reply message: If there was no node with higher ID in the search area (indicated by the bit in the reply message), the TTL is doubled and two new probe messages are sent (again to the two neighbors). If there was a better candidate in the search area, then the node becomes passive.
 - 7: If a node v receives its own probe message (not a reply) v decides to be the leader.
-

r costs $2 \cdot 2 \cdot 2^r$ messages. Therefore, round r costs at most $2 \cdot 2 \cdot 2^r \cdot \frac{n}{2^{r-1}} = 8n$ messages. Since there are only logarithmic many possible rounds, the message complexity follows immediately.

Remarks:

- This algorithm is asynchronous and uniform as well.
- The question may arise whether one can design an algorithm with an even lower message complexity. We answer this question in the next section.

2.4 Lower Bounds

Lower bounds in distributed computing are often easier than in the standard centralized (random access machine, RAM) model because one can argue about messages that need to be exchanged. In this section we present a first lower bound. We show that Algorithm 2 is asymptotically optimal.

Definition 2.14 (Execution). *An execution of a distributed algorithm is a list of events, sorted by time. An event is a record (time, node, type, message), where type is “send” or “receive”.*

Remarks:

- We assume throughout this course that no two events happen at exactly the same time (or one can break ties arbitrarily).
- An execution of an asynchronous algorithm is generally not only determined by the algorithm but also by a “god-like” scheduler. If more than one message is in transit, the scheduler can choose which one arrives first.
- If two messages are transmitted over the same directed edge, then it is sometimes required that the message first transmitted will also be received first (“FIFO”).

For our lower bound, we assume the following model:

- We are given an asynchronous ring, where nodes may wake up at arbitrary times (but at the latest when receiving the first message).
- We only accept uniform algorithms where the node with the maximum identifier can be the leader. Additionally, every node that is not the leader must know the identity of the leader. These two requirements can be dropped when using a more complicated proof; however, this is beyond the scope of this course.
- During the proof we will “play god” and specify which message in transmission arrives next in the execution. We respect the FIFO conditions for links.

Definition 2.15 (Open Schedule). *A schedule is an execution chosen by the scheduler. A schedule for a ring is open if there is an open edge in the ring. An open (undirected) edge is an edge where no message traversing the edge has been received so far.*

The proof of the lower bound is by induction. First we show the base case:

Lemma 2.16. *Given a ring R with two nodes, we can construct an open schedule in which at least one message is received. The nodes cannot distinguish this schedule from one on a larger ring with all other nodes being where the open edge is.*

Proof: Let the two nodes be u and v with $u < v$. Node u must learn the identity of node v , thus receive at least one message. We stop the execution of the algorithm as soon as the first message is received. (If the first message is received by v , bad luck for the algorithm!) Then the other edge in the ring (on which the received message was not transmitted) is open. Since the algorithm needs to be uniform, maybe the open edge is not really an edge at all, nobody can tell. We could use this to glue two rings together, by breaking up this imaginary open edge and connect two rings by two edges.

Lemma 2.17. *By gluing together two rings of size $n/2$ for which we have open schedules, we can construct an open schedule on a ring of size n . If $M(n/2)$ denotes the number of messages already received in each of these schedules, at least $2M(n/2) + n/4$ messages have to be exchanged in order to solve leader election.*

Proof by induction: We divide the ring into two sub-rings R_1 and R_2 of size $n/2$. These subrings cannot be distinguished from rings with $n/2$ nodes if no messages are received from “outsiders”. We can ensure this by not scheduling such messages until we want to. Note that executing both given open schedules on R_1 and R_2 “in parallel” is possible because we control not only the scheduling of the messages, but also when nodes wake up. By doing so, we make sure that $2M(n/2)$ messages are sent before the nodes in R_1 and R_2 learn anything of each other!

Without loss of generality, R_1 contains the maximum identifier. Hence, each node in R_2 must learn the identity of the maximum identifier, thus at least $n/2$ additional messages must be received. The only problem is that we cannot connect the two sub-rings with both edges since the new ring needs to remain open. Thus, only messages over one of the edges can be received. We “play god” and look into the future: we check what happens when we close only one of these connecting edges. With the argument that $n/2$ new messages must be received, we know that there is at least one edge that will produce at least $n/4$ additional messages when being scheduled. (These messages may not be sent over the closed link, but they are *caused* by a message over this link. They cannot involve any message along the other (open) edge at distance $n/2$.) We schedule this edge and the resulting $n/4$ messages, and leave the other open.

Lemma 2.18. *Any uniform leader election algorithm for asynchronous rings has at least message complexity $M(n) \geq \frac{n}{4}(\log n + 1)$.*

Proof by induction: For simplicity we assume n being a power of 2. The base case $n = 2$ works because of Lemma 2.16 which implies that $M(2) \geq 1 = \frac{2}{4}(\log 2 + 1)$. For the induction step, using Lemma 2.17 and the induction hypothesis we have

$$\begin{aligned} M(n) &= 2 \cdot M\left(\frac{n}{2}\right) + \frac{n}{4} \\ &\geq 2 \cdot \left(\frac{n}{8} \left(\log \frac{n}{2} + 1\right)\right) + \frac{n}{4} \\ &= \frac{n}{4} \log n + \frac{n}{4} = \frac{n}{4} (\log n + 1). \end{aligned}$$

□

Remarks:

- To hide the ugly constants we use the “big Omega” notation, the lower bound equivalent of $O()$. A function f is in $\Omega(g)$ if there are constants x_0 and $c > 0$ such that $|f(x)| \geq c|g(x)|$ for all $x \geq x_0$. Again we refer to standard text books for a formal definition. Rewriting Lemma 2.18 we get:

Theorem 2.19 (Asynchronous Leader Election Lower Bound). *Any uniform leader election algorithm for asynchronous rings has $\Omega(n \log n)$ message complexity.*

2.5 Synchronous Ring

The lower bound relied on delaying messages for a very long time. Since this is impossible in the synchronous model, we might get a better message complexity

in this case. The basic idea is very simple: In the synchronous model, *not* receiving a message is information as well! First we make some additional assumptions:

- We assume that the algorithm is non-uniform (i.e., the ring size n is known).
- We assume that every node starts at the same time.
- The node with the minimum identifier becomes the leader; identifiers are integers.

Algorithm 3 Synchronous Leader Election

- 1: **Each node** v concurrently executes the following code:
 - 2: The algorithm operates in synchronous phases. Each phase consists of n time steps. Node v counts phases, starting with 0.
 - 3: **if** phase = v **and** v did not yet receive a message **then**
 - 4: v decides to be the leader
 - 5: v sends the message “ v is leader” around the ring
 - 6: **end if**
-

Remarks:

- Message complexity is indeed n .
- But the time complexity is huge! If m is the minimum identifier it is $m \cdot n$.
- The synchronous start and the non-uniformity assumptions can be dropped by using a wake-up technique (upon receiving a wake-up message, wake up your clockwise neighbors) and by letting messages travel slowly.
- There are several lower bounds for the synchronous model: comparison-based algorithms or algorithms where the time complexity cannot be a function of the identifiers have message complexity $\Omega(n \log n)$ as well.
- In general graphs efficient leader election may be tricky. While time-optimal leader election can be done by parallel flooding-echo (see next chapter), bounding the message complexity is generally more difficult.

Chapter 3

Tree Algorithms

In this chapter we learn a few basic algorithms on trees, and how to construct trees in the first place so that we can run these (and other) algorithms. The good news is that these algorithms have many applications, the bad news is that this chapter is a bit on the simple side. But maybe that's not really bad news?!

3.1 Broadcast

Definition 3.1 (Broadcast). *A broadcast operation is initiated by a single processor, the source. The source wants to send a message to all other nodes in the system.*

Definition 3.2 (Distance, Radius, Diameter). *The distance between two nodes u and v in an undirected graph G is the number of hops of a minimum path between u and v . The radius of a node u is the maximum distance between u and any other node in the graph. The radius of a graph is the minimum radius of any node in the graph. The diameter of a graph is the maximum distance between two arbitrary nodes.*

Remarks:

- Clearly there is a close relation between the radius R and the diameter D of a graph, such as $R \leq D \leq 2R$.
- The world is often fascinated by graphs with a small radius. For example, movie fanatics study the who-acted-with-whom-in-the-same-movie graph. For this graph it has long been believed that the actor Kevin Bacon has a particularly small radius. The number of hops from Bacon even got a name, the Bacon Number. In the meantime, however, it has been shown that there are “better” centers in the Hollywood universe, such as Sean Connery, Christopher Lee, Rod Steiger, Gene Hackman, or Michael Caine. The center of other social networks has also been explored, Paul Erdős for instance is well known in the math community.

Theorem 3.3 (Broadcast Lower Bound). *The message complexity of broadcast is at least $n - 1$. The source's radius is a lower bound for the time complexity.*

Proof: Every node must receive the message.

Remarks:

- You can use a pre-computed spanning tree to do broadcast with tight message complexity. If the spanning tree is a breadth-first search spanning tree (for a given source), then the time complexity is tight as well.

Definition 3.4 (Clean). *A graph (network) is clean if the nodes do not know the topology of the graph.*

Theorem 3.5 (Clean Broadcast Lower Bound). *For a clean network, the number of edges is a lower bound for the broadcast message complexity.*

Proof: If you do not try every edge, you might miss a whole part of the graph behind it.

Remarks:

- This lower bound proof directly brings us to the well known *flooding* algorithm.

Algorithm 4 Flooding

- 1: The source (root) sends the message to all neighbors.
 - 2: **Each other node** v upon receiving the message the first time forwards the message to all (other) neighbors.
 - 3: Upon later receiving the message again (over other edges), a node can discard the message.
-

Remarks:

- If node v receives the message first from node u , then node v calls node u *parent*. This parent relation defines a spanning tree T . If the flooding algorithm is executed in a synchronous system, then T is a breadth-first search spanning tree (with respect to the root).
- More interestingly, also in asynchronous systems the flooding algorithm terminates after R time units, R being the radius of the source. However, the constructed spanning tree may not be a breadth-first search spanning tree.

3.2 Convergecast

Convergecast is the same as broadcast, just reversed: Instead of a root sending a message to all other nodes, all other nodes send information to a root. The simplest convergecast algorithm is the echo algorithm:

Algorithm 5 Echo

Require: This algorithm is initiated at the leaves.

- 1: A leaf sends a message to its parent.
 - 2: If an inner node has received a message from each child, it sends a message to the parent.
-

Remarks:

- Usually the echo algorithm is paired with the flooding algorithm, which is used to let the leaves know that they should start the echo process; this is known as flooding/echo.
- One can use convergecast for termination detection, for example. If a root wants to know whether all nodes in the system have finished some task, it initiates a flooding/echo; the message in the echo algorithm then means “This subtree has finished the task.”
- Message complexity of the echo algorithm is $n - 1$, but together with flooding it is $O(m)$, where $m = |E|$ is the number of edges in the graph.
- The time complexity of the echo algorithm is determined by the depth of the spanning tree (i.e., the radius of the root within the tree) generated by the flooding algorithm.
- The flooding/echo algorithm can do much more than collecting acknowledgements from subtrees. One can for instance use it to compute the number of nodes in the system, or the maximum ID (for leader election), or the sum of all values stored in the system, or a route-disjoint matching.
- Moreover, by combining results one can compute even fancier aggregations, e.g., with the number of nodes and the sum one can compute the average. With the average one can compute the standard deviation. And so on . . .

3.3 BFS Tree Construction

In synchronous systems the flooding algorithm is a simple yet efficient method to construct a breadth-first search (BFS) spanning tree. However, in asynchronous systems the spanning tree constructed by the flooding algorithm may be far from BFS. In this section, we implement two classic BFS constructions—Dijkstra and Bellman-Ford—as asynchronous algorithms.

We start with the Dijkstra algorithm. The basic idea is to always add the “closest” node to the existing part of the BFS tree. We need to parallelize this idea by developing the BFS tree layer by layer:

Theorem 3.6 (Analysis of Algorithm 6). *The time complexity of Algorithm 6 is $O(D^2)$, the message complexity is $O(m + nD)$, where D is the diameter of the graph, n the number of nodes, and m the number of edges.*

Proof: A broadcast/echo algorithm in T_p needs at most time $2D$. Finding new neighbors at the leaves costs 2 time units. Since the BFS tree height is bounded

Algorithm 6 Dijkstra BFS

-
- 1: The algorithm proceeds in phases. In phase p the nodes with distance p to the root are detected. Let T_p be the tree in phase p . We start with T_1 which is the root plus all direct neighbors of the root. We start with phase $p = 1$:
 - 2: **repeat**
 - 3: The root starts phase p by broadcasting “start p ” within T_p .
 - 4: When receiving “start p ” a leaf node u of T_p (that is, a node that was newly discovered in the last phase) sends a “join $p + 1$ ” message to all quiet neighbors. (A neighbor v is quiet if u has not yet “talked” to v .)
 - 5: A node v receiving the first “join $p+1$ ” message replies with “ACK” and becomes a leaf of the tree T_{p+1} .
 - 6: A node v receiving any further “join” message replies with “NACK”.
 - 7: The leaves of T_p collect all the answers of their neighbors; then the leaves start an echo algorithm back to the root.
 - 8: When the echo process terminates at the root, the root increments the phase
 - 9: **until** there was no new node detected
-

by the diameter, we have D phases, giving a total time complexity of $O(D^2)$. Each node participating in broadcast/echo only receives (broadcasts) at most 1 message and sends (echoes) at most once. Since there are D phases, the cost is bounded by $O(nD)$. On each edge there are at most 2 “join” messages. Replies to a “join” request are answered by 1 “ACK” or “NACK”, which means that we have at most 4 additional messages per edge. Therefore the message complexity is $O(m + nD)$.

Remarks:

- The time complexity is not very exciting, so let’s try Bellman-Ford!

The basic idea of Bellman-Ford is even simpler, and heavily used in the Internet, as it is a basic version of the omnipresent border gateway protocol (BGP). The idea is to simply keep the distance to the root accurate. If a neighbor has found a better route to the root, a node might also need to update its distance.

Algorithm 7 Bellman-Ford BFS

-
- 1: Each node u stores an integer d_u which corresponds to the distance from u to the root. Initially $d_{\text{root}} = 0$, and $d_u = \infty$ for every other node u .
 - 2: The root starts the algorithm by sending “1” to all neighbors.
 - 3: **if** a node u receives a message “ y ” with $y < d_u$ from a neighbor v **then**
 - 4: node u sets $d_u := y$
 - 5: node u sends “ $y + 1$ ” to all neighbors (except v)
 - 6: **end if**
-

Theorem 3.7 (Analysis of Algorithm 7). *The time complexity of Algorithm 7 is $O(D)$, the message complexity is $O(nm)$, where D, n, m are defined as in Theorem 3.6.*

Proof: We can prove the time complexity by induction. We claim that a node at distance d from the root has received a message “ d ” by time d . The root

knows by time 0 that it is the root. A node v at distance d has a neighbor u at distance $d - 1$. Node u by induction sends a message “ d ” to v at time $d - 1$ or before, which is then received by v at time d or before. Message complexity is easier: A node can reduce its distance at most $n - 1$ times; each of these times it sends a message to all its neighbors. If all nodes do this we have $O(nm)$ messages.

Remarks:

- Algorithm 6 has the better message complexity and Algorithm 7 has the better time complexity. The currently best algorithm (optimizing both) needs $O(m + n \log^3 n)$ messages and $O(D \log^3 n)$ time. This “trade-off” algorithm is beyond the scope of this course.

3.4 MST Construction

There are several types of spanning trees, each serving a different purpose. A particularly interesting spanning tree is the minimum spanning tree (MST). The MST only makes sense on weighted graphs, hence in this section we assume that each edge e is assigned a weight ω_e .

Definition 3.8 (MST). *Given a weighted graph $G = (V, E, \omega)$, the MST of G is a spanning tree T minimizing $\omega(T)$, where $\omega(G') = \sum_{e \in G'} \omega_e$ for any subgraph $G' \subseteq G$.*

Remarks:

- In the following we assume that no two edges of the graph have the same weight. This simplifies the problem as it makes the MST unique; however, this simplification is not essential as one can always break ties by adding the IDs of adjacent vertices to the weight.
- Obviously we are interested in computing the MST in a distributed way. For this we use a well-known lemma:

Definition 3.9 (Blue Edges). *Let T be a spanning tree of the weighted graph G and $T' \subseteq T$ a subgraph of T (also called a fragment). Edge $e = (u, v)$ is an outgoing edge of T' if $u \in T'$ and $v \notin T'$ (or vice versa). The minimum weight outgoing edge $b(T')$ is the so-called blue edge of T' .*

Lemma 3.10. *For a given weighted graph G (such that no two weights are the same), let T denote the MST, and T' be a fragment of T . Then the blue edge of T' is also part of T , i.e., $T' \cup b(T') \subseteq T$.*

Proof: For the sake of contradiction, suppose that in the MST T there is edge $e \neq b(T')$ connecting T' with the remainder of T . Adding the blue edge $b(T')$ to the MST T we get a cycle including both e and $b(T')$. If we remove e from this cycle we still have a spanning tree, and since by the definition of the blue edge $\omega_e > \omega_{b(T')}$, the weight of that new spanning tree is less than the weight of T . We have a contradiction.

Remarks:

- In other words, the blue edges seem to be the key to a distributed algorithm for the MST problem. Since every node itself is a fragment of the MST, every node directly has a blue edge! All we need to do is to grow these fragments! Essentially this is a distributed version of Kruskal's sequential algorithm.
- At any given time the nodes of the graph are partitioned into fragments (rooted subtrees of the MST). Each fragment has a root, the ID of the fragment is the ID of its root. Each node knows its parent and its children in the fragment. The algorithm operates in phases. At the beginning of a phase, nodes know the IDs of the fragments of their neighbor nodes.

Algorithm 8 GHS (Gallager–Humblet–Spira)

- 1: Initially each node is the root of its own fragment. We proceed in phases:
 - 2: **repeat**
 - 3: All nodes learn the fragment IDs of their neighbors.
 - 4: The root of each fragment uses flooding/echo in its fragment to determine the blue edge $b = (u, v)$ of the fragment.
 - 5: The root sends a message to node u ; while forwarding the message on the path from the root to node u all parent-child relations are inverted {such that u is the new temporary root of the fragment}
 - 6: node u sends a merge request over the blue edge $b = (u, v)$.
 - 7: **if** node v also sent a merge request over the same blue edge $b = (v, u)$
 then
 - 8: either u or v (whichever has the smaller ID) is the new fragment root
 - 9: the blue edge b is directed accordingly
 - 10: **else**
 - 11: node v is the new parent of node u
 - 12: **end if**
 - 13: the newly elected root node informs all nodes in its fragment (again using flooding/echo) about its identity
 - 14: **until** all nodes are in the same fragment (i.e., there is no outgoing edge)
-

Remarks:

- Algorithm 8 was stated in pseudo-code, with a few details not really explained. For instance, it may be that some fragments are much larger than others, and because of that some nodes may need to wait for others, e.g., if node u needs to find out whether neighbor v also wants to merge over the blue edge $b = (u, v)$. The good news is that all these details can be solved. We can for instance bound the asynchronicity by guaranteeing that nodes only start the new phase after the last phase is done, similarly to the phase-technique of Algorithm 6.

Theorem 3.11 (Analysis of Algorithm 8). *The time complexity of Algorithm 8 is $O(n \log n)$, the message complexity is $O(m \log n)$.*

Proof: Each phase mainly consists of two flooding/echo processes. In general, the cost of flooding/echo on a tree is $O(D)$ time and $O(n)$ messages. However,

the diameter D of the fragments may turn out to be not related to the diameter of the graph because the MST may meander, hence it really is $O(n)$ time. In addition, in the first step of each phase, nodes need to learn the fragment ID of their neighbors; this can be done in 2 steps but costs $O(m)$ messages. There are a few more steps, but they are cheap. Altogether a phase costs $O(n)$ time and $O(m)$ messages. So we only have to figure out the number of phases: Initially all fragments are single nodes and hence have size 1. In a later phase, each fragment merges with at least one other fragment, that is, the size of the smallest fragment at least doubles. In other words, we have at most $\log n$ phases. The theorem follows directly.

Remarks:

- Algorithm 8 is called “GHS” after Gallager, Humblet, and Spira, three pioneers in distributed computing. Despite being quite simple the algorithm won the prestigious Edsger W. Dijkstra Prize in Distributed Computing in 2004, among other reasons because it was one of the first (1983) non-trivial asynchronous distributed algorithms. As such it can be seen as one of the seeds of this research area.
- We presented a simplified version of GHS. The original paper by Gallager et al. featured an improved message complexity of $O(m + n \log n)$.
- In 1987, Awerbuch managed to further improve the GHS algorithm to get $O(n)$ time and $O(m + n \log n)$ message complexity, both asymptotically optimal.
- The GHS algorithm can be applied in different ways. GHS for instance directly solves leader election in general graphs: The leader is simply the last surviving root!

Chapter 4

Vertex Coloring

4.1 Introduction

Vertex coloring is an infamous graph theory problem. Vertex coloring does have quite a few practical applications, for example in the area of wireless networks where coloring is the foundation of so-called TDMA MAC protocols. Generally speaking, vertex coloring is used as a means to break symmetries, one of the main themes in distributed computing. In this chapter we will not really talk about vertex coloring applications but treat the problem abstractly. At the end of the class you probably learned the fastest (but not constant!) algorithm ever! Let us start with some simple definitions and observations.

Problem 4.1 (Vertex Coloring). *Given an undirected graph $G = (V, E)$, assign a color c_u to each vertex $u \in V$ such that the following holds: $e = (v, w) \in E \Rightarrow c_v \neq c_w$.*

Remarks:

- Throughout this course, we use the terms *vertex* and *node* interchangeably.
- The application often asks us to use few colors! In a TDMA MAC protocol, for example, less colors immediately imply higher throughput. However, in distributed computing we are often happy with a solution which is sub-optimal. There is a tradeoff between the optimality of a solution (efficacy), and the work/time needed to compute the solution (efficiency).

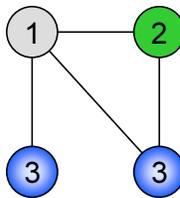


Figure 4.1: 3-colorable graph with a valid coloring.

Assumption 4.2 (Node Identifiers). *Each node has a unique identifier, e.g., its IP address. We usually assume that each identifier consists of only $\log n$ bits if the system has n nodes.*

Remarks:

- Sometimes we might even assume that the nodes exactly have identifiers $1, \dots, n$.
- It is easy to see that node identifiers (as defined in Assumption 4.2) solve the coloring problem 4.1, but not very well (essentially requiring n colors). How many colors are needed at least is a well-studied problem.

Definition 4.3 (Chromatic Number). *Given an undirected Graph $G = (V, E)$, the chromatic number $\chi(G)$ is the minimum number of colors to solve Problem 4.1.*

To get a better understanding of the vertex coloring problem, let us first look at a simple non-distributed (“centralized”) vertex coloring algorithm:

Algorithm 9 Greedy Sequential

- 1: **while** \exists uncolored vertex v **do**
 - 2: color v with the minimal color (number) that does not conflict with the already colored neighbors
 - 3: **end while**
-

Definition 4.4 (Degree). *The number of neighbors of a vertex v , denoted by $\delta(v)$, is called the degree of v . The maximum degree vertex in a graph G defines the graph degree $\Delta(G) = \Delta$.*

Theorem 4.5 (Analysis of Algorithm 9). *The algorithm is correct and terminates in n “steps”. The algorithm uses $\Delta + 1$ colors.*

Proof: Correctness and termination are straightforward. Since each node has at most Δ neighbors, there is always at least one color free in the range $\{1, \dots, \Delta + 1\}$.

Remarks:

- For many graphs coloring can be done with much less than $\Delta + 1$ colors.
- This algorithm is not distributed at all; only one processor is active at a time. Still, maybe we can use the simple idea of Algorithm 9 to define a distributed coloring subroutine that may come in handy later.

Now we are ready to study distributed algorithms for this problem. The following procedure can be executed by every vertex v in a distributed coloring algorithm. The goal of this subroutine is to improve a given initial coloring.

Procedure 10 First Free

Require: Node Coloring {e.g., node IDs as defined in Assumption 4.2}Give v the smallest admissible color {i.e., the smallest node color not used by any neighbor}

Algorithm 11 Reduce

- 1: Assume that initially all nodes have ID's (Assumption 4.2)
- 2: **Each node** v executes the following code
- 3: node v sends its ID to all neighbors
- 4: node v receives IDs of neighbors
- 5: **while** node v has an uncolored neighbor with higher ID **do**
- 6: node v sends "undecided" to all neighbors
- 7: node v receives new decisions from neighbors
- 8: **end while**
- 9: node v chooses a free color using subroutine **First Free** (Procedure 10)
- 10: node v informs all its neighbors about its choice

Remarks:

- With this subroutine we have to make sure that two adjacent vertices are not colored at the same time. Otherwise, the neighbors may at the same time conclude that some small color c is still available in their neighborhood, and then at the same time decide to choose this color c .

Theorem 4.6 (Analysis of Algorithm 11). *Algorithm 11 is correct and has time complexity n . The algorithm uses $\Delta + 1$ colors.*

Remarks:

- Quite trivial, but also quite slow.
- However, it seems difficult to come up with a fast algorithm.
- Maybe it's better to first study a simple special case, a tree, and then go from there.

4.2 Coloring Trees

Lemma 4.7. $\chi(\text{Tree}) \leq 2$

Constructive Proof: If the distance of a node to the root is odd (even), color it 1 (0). An odd node has only even neighbors and vice versa. If we assume that each node knows its parent (root has no parent) and children in a tree, this constructive proof gives a very simple algorithm:

Remarks:

- With the proof of Lemma 4.7, Algorithm 12 is correct.
- How can we determine a root in a tree if it is not already given? We will figure that out later.

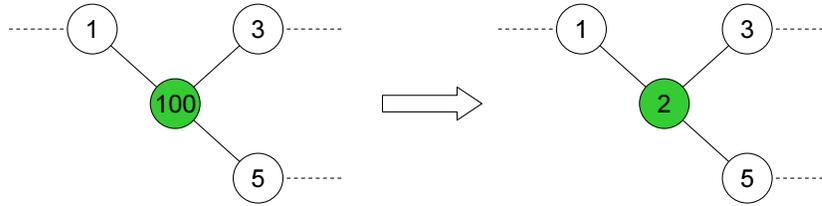


Figure 4.2: Vertex 100 receives the lowest possible color.

Algorithm 12 Slow Tree Coloring

-
- 1: Color the root 0, root sends 0 to its children
 - 2: **Each node** v concurrently executes the following code:
 - 3: **if** node v receives a message x (from parent) **then**
 - 4: node v chooses color $c_v = 1 - x$
 - 5: node v sends c_v to its children (all neighbors except parent)
 - 6: **end if**
-

- The time complexity of the algorithm is the height of the tree.
- If the root was chosen unfortunately, and the tree has a degenerated topology, the time complexity may be up to n , the number of nodes.
- Also, this algorithm does not need to be synchronous ...!

Theorem 4.8 (Analysis of Algorithm 12). *Algorithm 12 is correct. If each node knows its parent and its children, the (asynchronous) time complexity is the tree height which is bounded by the diameter of the tree; the message complexity is $n - 1$ in a tree with n nodes.*

Remarks:

- In this case the asynchronous time complexity is the same as the synchronous time complexity.
- Nice trees, e.g. balanced binary trees, have logarithmic height, that is we have a logarithmic time complexity.
- This algorithm is not very exciting. Can we do better than logarithmic?!?

The following algorithm terminates in $\log^* n$ time. Log-Star?! That's the number of logarithms (to the base 2) you need to take to get down to at least 2, starting with n :

Definition 4.9 (Log-Star).

$$\forall x \leq 2 : \log^* x := 1 \quad \forall x > 2 : \log^* x := 1 + \log^*(\log x)$$

Remarks:

- Log-star is an amazingly slowly growing function. Log-star of all the atoms in the observable universe (estimated to be 10^{80}) is 5! There are functions which grow even more slowly, such as the inverse Ackermann function, however, the inverse Ackermann function of all the atoms is 4. So log-star increases indeed very slowly!

Here is the idea of the algorithm: We start with color labels that have $\log n$ bits. In each synchronous round we compute a new label with exponentially smaller size than the previous label, still guaranteeing to have a valid vertex coloring! But how are we going to do that?

Algorithm 13 “6-Color”

```

1: Assume that initially the vertices are legally colored. Using Assumption 4.2
   each label only has  $\log n$  bits
2: The root assigns itself the label 0.
3: Each other node  $v$  executes the following code (synchronously in parallel)
4: send  $c_v$  to all children
5: repeat
6:   receive  $c_p$  from parent
7:   interpret  $c_v$  and  $c_p$  as little-endian bit-strings:  $c(k), \dots, c(1), c(0)$ 
8:   let  $i$  be the smallest index where  $c_v$  and  $c_p$  differ
9:   the new label is  $i$  (as bitstring) followed by the bit  $c_v(i)$  itself
10:  send  $c_v$  to all children
11: until  $c_w \in \{0, \dots, 5\}$  for all nodes  $w$ 

```

Example:

Algorithm 13 executed on the following part of a tree:

Grand-parent	0010110000	→	10010	→	...
Parent	1010010000	→	01010	→	111
Child	0110010000	→	10001	→	001

Theorem 4.10 (Analysis of Algorithm 13). *Algorithm 13 terminates in $\log^* n$ time.*

Proof: A detailed proof is, e.g., in [Peleg 7.3]. In class we do a sketch of the proof.

Remarks:

- Colors $11*$ (in binary notation, i.e., 6 or 7 in decimal notation) will not be chosen, because the node will then do another round. This gives a total of 6 colors (i.e., colors $0, \dots, 5$).
- Can one reduce the number of colors in only constant steps? Note that algorithm 11 does not work (since the degree of a node can be much higher than 6)! For fewer colors we need to have siblings monochromatic!
- Before we explore this problem we should probably have a second look at the end game of the algorithm, the UNTIL statement. Is this algorithm truly local?! Let's discuss!

Algorithm 14 Shift Down

- 1: Root chooses a new (different) color from $\{0, 1, 2\}$
 - 2: **Each** other **node** v concurrently executes the following code:
 - 3: Recolor v with the color of parent
-

Lemma 4.11 (Analysis of Algorithm 14). *Algorithm 14 preserves coloring legality; also siblings are monochromatic.*

Now Algorithm 11 (Reduce) can be used to reduce the number of used colors from six to three.

Algorithm 15 Six-2-Three

- 1: **Each node** v concurrently executes the following code:
 - 2: Run Algorithm 13 for $\log^* n$ rounds.
 - 3: **for** $x = 5, 4, 3$ **do**
 - 4: Perform subroutine Shift down (Algorithm 14)
 - 5: **if** $c_v = x$ **then**
 - 6: choose new color $c_v \in \{0, 1, 2\}$ using subroutine **First Free** (Algorithm 10)
 - 7: **end if**
 - 8: **end for**
-

Theorem 4.12 (Analysis of Algorithm 15). *Algorithm 15 colors a tree with three colors in time $O(\log^* n)$.*

Remarks:

- The term $O()$ used in Theorem 4.10 is called “big O” and is often used in distributed computing. Roughly speaking, $O(f)$ means “in the order of f , ignoring constant factors and smaller additive terms.” More formally, for two functions f and g , it holds that $f \in O(g)$ if there are constants x_0 and c so that $|f(x)| \leq c|g(x)|$ for all $x \geq x_0$. For an elaborate discussion on the big O notation we refer to other introductory math or computer science classes.
- As one can easily prove, a fast tree-coloring with only 2 colors is more than exponentially more expensive than coloring with 3 colors. In a tree degenerated to a list, nodes far away need to figure out whether they are an even or odd number of hops away from each other in order to get a 2-coloring. To do that one has to send a message to these nodes. This costs time linear in the number of nodes.
- Also other lower bounds have been proved, e.g., any algorithm for 2-coloring the d -regular tree of radius r which runs in time at most $2r/3$ requires at least $\Omega(\sqrt{d})$ colors.
- The idea of this algorithm can be generalized, e.g., to a ring topology. Also a general graph with constant degree Δ can be colored with $\Delta + 1$ colors in $O(\log^* n)$ time. The idea is as follows: In each step, a node compares its label to each of its neighbors, constructing a logarithmic difference-tag

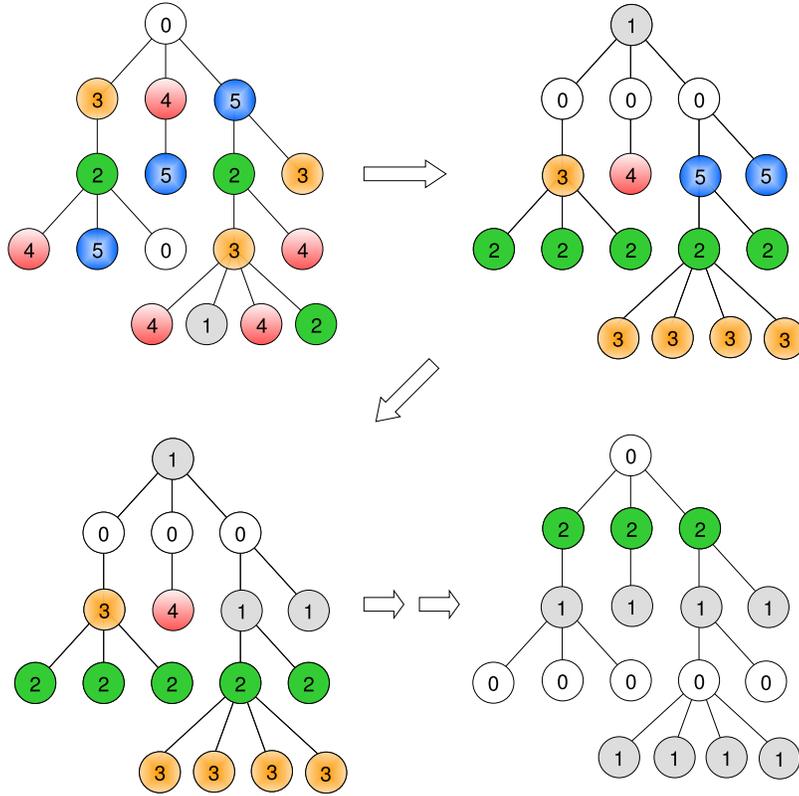


Figure 4.3: Possible execution of Algorithm 15.

as in 6-color (Algorithm 13). Then the new label is the concatenation of all the difference-tags. For constant degree Δ , this gives a 3Δ -label in $O(\log^* n)$ steps. Algorithm 11 then reduces the number of colors to $\Delta + 1$ in $2^{3\Delta}$ (this is still a constant for constant Δ) steps.

- Recently, researchers have proposed other methods to break down long ID's for log-star algorithms. With these new techniques, one is able to solve other problems, e.g., a maximal independent set in bounded growth graphs in $O(\log^* n)$ time. These techniques go beyond the scope of this course.
- Unfortunately, coloring a general graph is not yet possible with this technique. We will see another technique for that in Chapter 5. With this technique it is possible to color a general graph with $\Delta + 1$ colors in $O(\log n)$ time.
- A lower bound by Linial shows that many of these log-star algorithms are asymptotically (up to constant factors) optimal. This lower bound uses an interesting technique. However, because of the one-topic-per-class policy we cannot look at it today.

Chapter 5

Maximal Independent Set

In this chapter we present a highlight of this course, a fast maximal independent set (MIS) algorithm. The algorithm is the first randomized algorithm that we study in this class. In distributed computing, randomization is a powerful and therefore omnipresent concept, as it allows for relatively simple yet efficient algorithms. As such the studied algorithm is archetypal.

A MIS is a basic building block in distributed computing, some other problems pretty much follow directly from the MIS problem. At the end of this chapter, we will give two examples: matching and vertex coloring (see Chapter 4).

5.1 MIS

Definition 5.1 (Independent Set). *Given an undirected Graph $G = (V, E)$ an independent set is a subset of nodes $U \subseteq V$, such that no two nodes in U are adjacent. An independent set is maximal if no node can be added without violating independence. An independent set of maximum cardinality is called maximum.*

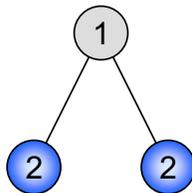


Figure 5.1: Example graph with 1) a maximal independent set (MIS) and 2) a maximum independent set (MaxIS).

Remarks:

- Computing a maximum independent set (MaxIS) is a notoriously difficult problem. It is equivalent to maximum clique on the complementary graph. Both problems are NP-hard, in fact not approximable within $n^{\frac{1}{2}-\epsilon}$.
- In this course we concentrate on the maximal independent set (MIS) problem. Please note that MIS and MaxIS can be quite different, indeed e.g. on a star graph the MIS is $\Theta(n)$ smaller than the MaxIS (cf. Figure 5.1).
- Computing a MIS sequentially is trivial: Scan the nodes in arbitrary order. If a node u does not violate independence, add u to the MIS. If u violates independence, discard u . So the only question is how to compute a MIS in a distributed way.

Algorithm 16 Slow MIS

Require: Node IDs**Every node** v executes the following code:

- 1: **if** all neighbors of v with larger identifiers have decided not to join the MIS
 - then**
 - 2: v decides to join the MIS
 - 3: **end if**
-

Remarks:

- Not surprisingly the slow algorithm is not better than the sequential algorithm in the worst case, because there might be one single point of activity at any time. Formally:

Theorem 5.2 (Analysis of Algorithm 16). *Algorithm 16 features a time complexity of $O(n)$ and a message complexity of $O(m)$.*

Remarks:

- This is not very exciting.
- There is a relation between independent sets and node coloring (Chapter 4), since each color class is an independent set, however, not necessarily a MIS. Still, starting with a coloring, one can easily derive a MIS algorithm: We first choose all nodes of the first color. Then, for each additional color we add “in parallel” (without conflict) as many nodes as possible. Thus the following corollary holds:

Corollary 5.3. *Given a coloring algorithm that needs C colors and runs in time T , we can construct a MIS in time $C + T$.*

Remarks:

- Using Theorem 4.12 and Corollary 5.3 we get a distributed deterministic MIS algorithm for trees (and for bounded degree graphs) with time complexity $O(\log^* n)$.

- With a lower bound argument one can show that this deterministic MIS algorithm for rings is asymptotically optimal.
- There have been attempts to extend Algorithm 13 to more general graphs, however, so far without much success. Below we present a radically different approach that uses randomization. Please note that the algorithm and the analysis below is not identical with the algorithm in Peleg's book.

5.2 Fast MIS from 1986

Algorithm 17 Fast MIS

The algorithm operates in synchronous rounds, grouped into phases.

A single phase is as follows:

- 1) Each node v marks itself with probability $\frac{1}{2d(v)}$, where $d(v)$ is the current degree of v .
 - 2) If no higher degree neighbor of v is also marked, node v joins the MIS. If a higher degree neighbor of v is marked, node v unmarks itself again. (If the neighbors have the same degree, ties are broken arbitrarily, e.g., by identifier).
 - 3) Delete all nodes that joined the MIS and their neighbors, as they cannot join the MIS anymore.
-

Remarks:

- Correctness in the sense that the algorithm produces an independent set is relatively simple: Steps 1 and 2 make sure that if a node v joins the MIS, then v 's neighbors do not join the MIS at the same time. Step 3 makes sure that v 's neighbors will never join the MIS.
- Likewise the algorithm eventually produces a MIS, because the node with the highest degree will mark itself at some point in Step 1.
- So the only remaining question is how fast the algorithm terminates. To understand this, we need to dig a bit deeper.

Lemma 5.4 (Joining MIS). *A node v joins the MIS in step 2 with probability $p \geq \frac{1}{4d(v)}$.*

Proof: Let M be the set of marked nodes in step 1. Let $H(v)$ be the set of neighbors of v with higher degree, or same degree and higher identifier. Using independence of the random choices of v and nodes in $H(v)$ in Step 1 we get

$$\begin{aligned}
 P[v \notin \text{MIS} | v \in M] &= P[\exists w \in H(v), w \in M | v \in M] \\
 &= P[\exists w \in H(v), w \in M] \\
 &\leq \sum_{w \in H(v)} P[w \in M] = \sum_{w \in H(v)} \frac{1}{2d(w)} \\
 &\leq \sum_{w \in H(v)} \frac{1}{2d(v)} \leq \frac{d(v)}{2d(v)} = \frac{1}{2}.
 \end{aligned}$$

Then

$$P[v \in \text{MIS}] = P[v \in \text{MIS} | v \in M] \cdot P[v \in M] \geq \frac{1}{2} \cdot \frac{1}{2d(v)}.$$

□

Lemma 5.5 (Good Nodes). *A node v is called good if*

$$\sum_{w \in N(v)} \frac{1}{2d(w)} \geq \frac{1}{6}.$$

Otherwise we call v a bad node. A good node will be removed in Step 3 with probability $p \geq \frac{1}{36}$.

Proof: Let node v be good. Intuitively, good nodes have lots of low-degree neighbors, thus chances are high that one of them goes into the independent set, in which case v will be removed in step 3 of the algorithm.

If there is a neighbor $w \in N(v)$ with degree at most 2 we are done: With Lemma 5.4 the probability that node w joins the MIS is at least $\frac{1}{8}$, and our good node will be removed in Step 3.

So all we need to worry about is that all neighbors have at least degree 3: For any neighbor w of v we have $\frac{1}{2d(w)} \leq \frac{1}{6}$. Since $\sum_{w \in N(v)} \frac{1}{2d(w)} \geq \frac{1}{6}$ there is a subset of neighbors $S \subseteq N(v)$ such that $\frac{1}{6} \leq \sum_{w \in S} \frac{1}{2d(w)} \leq \frac{1}{3}$

We can now bound the probability that node v will be removed. Let therefore R be the event of v being removed. Again, if a neighbor of v joins the MIS in Step 2, node v will be removed in Step 3. We have

$$\begin{aligned} P[R] &\geq P[\exists u \in S, u \in \text{MIS}] \\ &\geq \sum_{u \in S} P[u \in \text{MIS}] - \sum_{u, w \in S; u \neq w} P[u \in \text{MIS} \text{ and } w \in \text{MIS}]. \end{aligned}$$

For the last inequality we used the inclusion-exclusion principle truncated after the second order terms. Let M again be the set of marked nodes after Step 1. Using $P[u \in M] \geq P[u \in \text{MIS}]$ we get

$$\begin{aligned} P[R] &\geq \sum_{u \in S} P[u \in \text{MIS}] - \sum_{u, w \in S; u \neq w} P[u \in M \text{ and } w \in M] \\ &\geq \sum_{u \in S} P[u \in \text{MIS}] - \sum_{u \in S} \sum_{w \in S} P[u \in M] \cdot P[w \in M] \\ &\geq \sum_{u \in S} \frac{1}{4d(u)} - \sum_{u \in S} \sum_{w \in S} \frac{1}{2d(u)} \frac{1}{2d(w)} \\ &\geq \sum_{u \in S} \frac{1}{2d(u)} \left(\frac{1}{2} - \sum_{w \in S} \frac{1}{2d(w)} \right) \geq \frac{1}{6} \left(\frac{1}{2} - \frac{1}{3} \right) = \frac{1}{36}. \end{aligned}$$

□

Remarks:

- We would be almost finished if we could prove that many nodes are good in each phase. Unfortunately this is not the case: In a star-graph, for instance, only a single node is good! We need to find a work-around.

Lemma 5.6 (Good Edges). *An edge $e = (u, v)$ is called bad if both u and v are bad; else the edge is called good. The following holds: At any time at least half of the edges are good.*

Proof: For the proof we construct a directed auxiliary graph: Direct each edge towards the higher degree node (if both nodes have the same degree direct it towards the higher identifier). Now we need a little helper lemma before we can continue with the proof.

Lemma 5.7. *A bad node has outdegree at least twice its indegree.*

Proof: For the sake of contradiction, assume that a bad node v does not have outdegree at least twice its indegree. In other words, at least one third of the neighbor nodes (let's call them S) have degree at most $d(v)$. But then

$$\sum_{w \in N(v)} \frac{1}{2d(w)} \geq \sum_{w \in S} \frac{1}{2d(w)} \geq \sum_{w \in S} \frac{1}{2d(v)} \geq \frac{d(v)}{3} \frac{1}{2d(v)} = \frac{1}{6}$$

which means v is good, a contradiction. \square

Continuing the proof of Lemma 5.6: According to Lemma 5.7 the number of edges directed into bad nodes is at most half the number of edges directed out of bad nodes. Thus, the number of edges directed into bad nodes is at most half the number of edges. Thus, at least half of the edges are directed into good nodes. Since these edges are not bad, they must be good.

Theorem 5.8 (Analysis of Algorithm 17). *Algorithm 17 terminates in expected time $O(\log n)$.*

Proof: With Lemma 5.5 a good node (and therefore a good edge!) will be deleted with constant probability. Since at least half of the edges are good (Lemma 5.6) a constant fraction of edges will be deleted in each phase.

More formally: With Lemmas 5.5 and 5.6 we know that at least half of the edges will be removed with probability at least $1/36$. Let R be the number of edges to be removed. Using linearity of expectation we know that $\mathbb{E}[R] \geq m/72$, m being the total number of edges at the start of a phase. Now let $p := P[R \leq \mathbb{E}[R]/2]$. Bounding the expectation yields

$$\mathbb{E}[R] = \sum_r P[R = r] \cdot r \leq p \cdot \mathbb{E}[R]/2 + (1 - p) \cdot m.$$

Solving for p we get

$$p \leq \frac{m - \mathbb{E}[R]}{m - \mathbb{E}[R]/2} < \frac{m - \mathbb{E}[R]/2}{m} \leq 1 - 1/144.$$

In other words, with probability at least $1/144$ at least $m/144$ edges are removed in a phase. After expected $O(\log m)$ phases all edges are deleted. Since $m \leq n^2$ and thus $O(\log m) = O(\log n)$ the Theorem follows. \square

Remarks:

- With a bit of more math one can even show that Algorithm 17 terminates in time $O(\log n)$ “with high probability”.
- The presented algorithm is a simplified version of an algorithm by Michael Luby, published 1986 in the SIAM Journal of Computing. Around the same time there have been a number of other papers dealing with the same or related problems, for instance by Alon, Babai, and Itai, or by Israeli and Itai. The analysis presented here takes elements of all these papers, and from other papers on distributed weighted matching. The analysis in the book by David Peleg is different, and only achieves $O(\log^2 n)$ time.
- Though not as incredibly fast as the \log^* -coloring algorithm for trees, this algorithm is very general. It works on any graph, needs no identifiers, and can easily be made asynchronous.
- Surprisingly, much later, there have been half a dozen more papers published, with much worse results!! In 2002, for instance, there was a paper with linear running time, improving on a 1994 paper with cubic running time, restricted to trees!
- In 2009, Métivier, Robson, Saheb-Djahromi and Zemmari found a slightly different (and simpler) way to compute a MIS in the same logarithmic time:

5.3 Fast MIS from 2009

Algorithm 18 Fast MIS 2

The algorithm operates in synchronous rounds, grouped into phases.

A single phase is as follows:

- 1) Each node v chooses a random value $r(v) \in [0, 1]$ and sends it to its neighbors.
 - 2) If $r(v) < r(w)$ for all neighbors $w \in N(v)$, node v enters the MIS and informs its neighbors.
 - 3) If v or a neighbor of v entered the MIS, v terminates (v and all edges adjacent to v are removed from the graph), otherwise v enters the next phase.
-

Remarks:

- Correctness in the sense that the algorithm produces an independent set is simple: Steps 1 and 2 make sure that if a node v joins the MIS, then v 's neighbors do not join the MIS at the same time. Step 3 makes sure that v 's neighbors will never join the MIS.
- Likewise the algorithm eventually produces a MIS, because the node with the globally smallest value will always join the MIS, hence there is progress.
- So the only remaining question is how fast the algorithm terminates. To understand this, we need to dig a bit deeper.

- Our proof will rest on a simple, yet powerful observation about expected values of random variables that *may not be independent*:

Theorem 5.9 (Linearity of Expectation). *Let X_i , $i = 1, \dots, k$ denote random variables, then*

$$\mathbb{E} \left[\sum_i X_i \right] = \sum_i \mathbb{E} [X_i].$$

Proof. It is sufficient to prove $\mathbb{E} [X + Y] = \mathbb{E} [X] + \mathbb{E} [Y]$ for two random variables X and Y , because then the statement follows by induction. Since

$$\begin{aligned} P[(X, Y) = (x, y)] &= P[X = x] \cdot P[Y = y | X = x] \\ &= P[Y = y] \cdot P[X = x | Y = y] \end{aligned}$$

we get that

$$\begin{aligned} \mathbb{E} [X + Y] &= \sum_{(X, Y) = (x, y)} P[(X, Y) = (x, y)] \cdot (x + y) \\ &= \sum_{X=x} \sum_{Y=y} P[X = x] \cdot P[Y = y | X = x] \cdot x \\ &\quad + \sum_{Y=y} \sum_{X=x} P[Y = y] \cdot P[X = x | Y = y] \cdot y \\ &= \sum_{X=x} P[X = x] \cdot x + \sum_{Y=y} P[Y = y] \cdot y \\ &= \mathbb{E} [X] + \mathbb{E} [Y]. \end{aligned}$$

Remarks:

- How can we prove that the algorithm only needs $O(\log n)$ phases in expectation? It would be great if this algorithm managed to remove a constant fraction of nodes in each phase. Unfortunately, it does not.
- Instead we will prove that the number of *edges* decreases quickly. Again, it would be great if any single edge was removed with constant probability in Step 3. But again, unfortunately, this is not the case.
- Maybe we can argue about the expected number of edges to be removed in one single phase? Let's see: A node v enters the MIS with probability $1/(d(v) + 1)$, where $d(v)$ is the degree of node v . By doing so, not only are v 's edges removed, but indeed all the edges of v 's neighbors as well – generally these are much more than $d(v)$ edges. So there is hope, but we need to be careful: If we do this the most naive way, we will count the same edge many times.
- How can we fix this? The nice observation is that it is enough to count just some of the removed edges. Given a new MIS node v and a neighbor $w \in N(v)$, we count the edges only if $r(v) < r(x)$ for all $x \in N(w)$. This looks promising. In a star graph, for instance, only the smallest random value can be accounted for removing all the edges of the star.

Lemma 5.10 (Edge Removal). *In a single phase, we remove at least half of the edges in expectation.*

Proof: To simplify the notation, at the start of our phase, the graph is simply $G = (V, E)$. Suppose that a node v joins the MIS in this phase, i.e., $r(v) < r(w)$ for all neighbors $w \in N(v)$. If in addition we have $r(v) < r(x)$ for all neighbors x of a neighbor w of v , we call this event $(v \rightarrow w)$. The probability of event $(v \rightarrow w)$ is at least $1/(d(v) + d(w))$, since $d(v) + d(w)$ is the maximum number of nodes adjacent to v or w (or both). As v joins the MIS, all edges (w, x) will be removed; there are $d(w)$ of these edges.

In order to count the removed edges, we need to weigh events properly.

Whether we remove the edges adjacent to w because of event $(v \rightarrow w)$ is a random variable $X_{(v \rightarrow w)}$. If event $(v \rightarrow w)$ occurs, $X_{(v \rightarrow w)}$ has the value $d(w)$, if not it has the value 0. For each edge $\{v, w\}$ we have two such variables, the event $X_{(v \rightarrow w)}$ and $X_{(w \rightarrow v)}$. Due to Theorem 5.9, the expected value of the sum X of all these random variables is at least

$$\begin{aligned} \mathbb{E}[X] &= \sum_{\{v,w\} \in E} \mathbb{E}[X_{(v \rightarrow w)}] + \mathbb{E}[X_{(w \rightarrow v)}] \\ &= \sum_{\{v,w\} \in E} P[\text{Event } (v \rightarrow w)] \cdot d(w) + P[\text{Event } (w \rightarrow v)] \cdot d(v) \\ &\geq \sum_{\{v,w\} \in E} \frac{d(w)}{d(v) + d(w)} + \frac{d(v)}{d(w) + d(v)} \\ &= \sum_{\{v,w\} \in E} 1 = |E|. \end{aligned}$$

In other words, in expectation all edges are removed in a single phase!?! Probably not. This means that we still counted some edges more than once. Indeed, for an edge $\{v, w\} \in E$ our random variable X includes the edge if the event $(u \rightarrow v)$ happens, but X also includes the edge if the event $(x \rightarrow w)$ happens. So we may have counted the edge $\{v, w\}$ twice. Fortunately however, not more than twice, because at most one event $(\cdot \rightarrow v)$ and at most one event $(\cdot \rightarrow w)$ can happen. If $(u \rightarrow v)$ happens, we know that $r(u) < r(w)$ for all $w \in N(v)$; hence another $(u' \rightarrow v)$ cannot happen because $r(u') > r(u) \in N(v)$. Therefore the random variable X must be divided by 2. In other words, in expectation at least half of the edges are removed.

Remarks:

- This enables us to follow a bound on the expected running time of Algorithm 18 quite easily.

Theorem 5.11 (Expected running time of Algorithm 18). *Algorithm 18 terminates after at most $3 \log_{4/3} m + 1 \in O(\log n)$ phases in expectation.*

Proof: The probability that in a single phase at least a quarter of all edges are removed is at least $1/3$. For the sake of contradiction, assume not. Then with probability less than $1/3$ we may be lucky and many (potentially all) edges are removed. With probability more than $2/3$ less than $1/4$ of the edges are removed. Hence the expected fraction of removed edges is strictly less than $1/3 \cdot 1 + 2/3 \cdot 1/4 = 1/2$. This contradicts Lemma 5.10.

Hence, at least every third phase is “good” and removes at least a quarter of the edges. To get rid of all but two edges we need $\log_{4/3} m$ good phases in expectation. The last two edges will certainly be removed in the next phase. Hence a total of $3 \log_{4/3} m + 1$ phases are enough in expectation.

Remarks:

- Sometimes one expects a bit more of an algorithm: Not only should the expected time to terminate be good, but the algorithm should *always* terminate quickly. As this is impossible in randomized algorithms (after all, the random choices may be “unlucky” all the time!), researchers often settle for a compromise, and just demand that the probability that the algorithm does not terminate in the specified time can be made absurdly small. For our algorithm, this can be deduced from Lemma 5.10 and another standard tool, namely Chernoff’s Bound.

Definition 5.12 (W.h.p.). *We say that an algorithm terminates w.h.p. (with high probability) within $O(t)$ time if it does so with probability at least $1 - 1/n^c$ for any choice of $c \geq 1$. Here c may affect the constants in the Big- O notation because it is considered a “tunable constant” and usually kept small.*

Definition 5.13 (Chernoff’s Bound). *Let $X = \sum_{i=1}^k X_i$ be the sum of k independent $0 - 1$ random variables. Then Chernoff’s bound states that w.h.p.*

$$|X - \mathbb{E}[X]| \in O\left(\log n + \sqrt{\mathbb{E}[X] \log n}\right).$$

Corollary 5.14 (Running Time of Algorithm 18). *Algorithm 18 terminates w.h.p. in $O(\log n)$ time.*

Proof: In Theorem 5.11 we used that *independently* of everything that happened before, in each phase we have a constant probability p that a quarter of the edges are removed. Call such a phase *good*. For some constants C_1 and C_2 , let us check after $C_1 \log n + C_2 \in O(\log n)$ phases, in how many phases at least a quarter of the edges have been removed. In expectation, these are at least $p(C_1 \log n + C_2)$ many. Now we look at the random variable $X = \sum_{i=1}^{C_1 \log n + C_2} X_i$, where the X_i are independent $0 - 1$ variables being one with exactly probability p . Certainly, if X is at least x with some probability, then the probability that we have x good phases can only be larger (if no edges are left, certainly “all” of the remaining edges are removed). To X we can apply Chernoff’s bound. If C_1 and C_2 are chosen large enough, they will overcome the constants in the Big- O from Chernoff’s bound, i.e., w.h.p. it holds that $|X - \mathbb{E}[X]| \leq \mathbb{E}[X]/2$, implying $X \geq \mathbb{E}[X]/2$. Choosing C_1 large enough, we will have w.h.p. sufficiently many good phases, i.e., the algorithm terminates w.h.p. in $O(\log n)$ phases.

Remarks:

- The algorithm can be improved a bit more even. Drawing random real numbers in each phase for instance is not necessary. One can achieve the same by sending only a total of $O(\log n)$ random (and as many non-random) bits over each edge.
- One of the main open problems in distributed computing is whether one can beat this logarithmic time, or at least achieve it with a deterministic algorithm.

- Let's turn our attention to applications of MIS next.

5.4 Applications

Definition 5.15 (Matching). *Given a graph $G = (V, E)$ a matching is a subset of edges $M \subseteq E$, such that no two edges in M are adjacent (i.e., where no node is adjacent to two edges in the matching). A matching is maximal if no edge can be added without violating the above constraint. A matching of maximum cardinality is called maximum. A matching is called perfect if each node is adjacent to an edge in the matching.*

Remarks:

- In contrast to MaxIS, a maximum matching can be found in polynomial time (Blossom algorithm by Jack Edmonds), and is also easy to approximate (in fact, already any maximal matching is a 2-approximation).
- An independent set algorithm is also a matching algorithm: Let $G = (V, E)$ be the graph for which we want to construct the matching. The auxiliary graph G' is defined as follows: for every edge in G there is a node in G' ; two nodes in G' are connected by an edge if their respective edges in G are adjacent. A (maximal) independent set in G' is a (maximal) matching in G , and vice versa. Using Algorithm 18 directly produces a $O(\log n)$ bound for maximal matching.
- More importantly, our MIS algorithm can also be used for vertex coloring (Problem 4.1):

Algorithm 19 General Graph Coloring

- 1: Given a graph $G = (V, E)$ we virtually build a graph $G' = (V', E')$ as follows:
 - 2: Every node $v \in V$ clones itself $d(v) + 1$ times ($v_0, \dots, v_{d(v)} \in V'$), $d(v)$ being the degree of v in G .
 - 3: The edge set E' of G' is as follows:
 - 4: First all clones are in a clique: $(v_i, v_j) \in E'$, for all $v \in V$ and all $0 \leq i < j \leq d(v)$
 - 5: Second all i^{th} clones of neighbors in the original graph G are connected: $(u_i, v_i) \in E'$, for all $(u, v) \in E$ and all $0 \leq i \leq \min(d(u), d(v))$.
 - 6: Now we simply run (simulate) the fast MIS Algorithm 18 on G' .
 - 7: If node v_i is in the MIS in G' , then node v gets color i .
-

Theorem 5.16 (Analysis of Algorithm 19). *Algorithm 19 $(\Delta + 1)$ -colors an arbitrary graph in $O(\log n)$ time, with high probability, Δ being the largest degree in the graph.*

Proof: Thanks to the clique among the clones at most one clone is in the MIS. And because of the $d(v) + 1$ clones of node v every node will get a free color! The running time remains logarithmic since G' has $O(n^2)$ nodes and the exponent becomes a constant factor when applying the logarithm.

Remarks:

- This solves our open problem from Chapter 4.1!
- Together with Corollary 5.3 we get quite close ties between $(\Delta+1)$ -coloring and the MIS problem.
- However, in general Algorithm 19 is not the best distributed algorithm for $O(\Delta)$ -coloring. For fast distributed vertex coloring please check Kothapalli, Onus, Scheideler, Schindelhauer, IPDPS 2006. This algorithm is based on a $O(\log \log n)$ time *edge* coloring algorithm by Grable and Panconesi, 1997.
- Computing a MIS also solves another graph problem on graphs of bounded independence.

Definition 5.17 (Bounded Independence). $G = (V, E)$ is of bounded independence, if each neighborhood contains at most a constant number of independent (i.e., mutually non-adjacent) nodes.

Definition 5.18 ((Minimum) Dominating Sets). A dominating set is a subset of the nodes such that each node is in the set or adjacent to a node in the set. A minimum dominating set is a dominating set containing the least possible number of nodes.

Remarks:

- In general, finding a dominating set less than factor $\log n$ larger than a minimum dominating set is NP-hard.
- Any MIS is a dominating set: if a node was not covered, it could join the independent set.
- In general a MIS and a minimum dominating sets have not much in common (think of a star). For graphs of bounded independence, this is different.

Corollary 5.19. *On graphs of bounded independence, a constant-factor approximation to a minimum dominating set can be found in time $O(\log n)$ w.h.p.*

Proof: Denote by M a minimum dominating set and by I a MIS. Since M is a dominating set, each node from I is in M or adjacent to a node in M . Since the graph is of bounded independence, no node in M is adjacent to more than constantly many nodes from I . Thus, $|I| \in O(|M|)$. Therefore, we can compute a MIS with Algorithm 18 and output it as the dominating set, which takes $O(\log n)$ rounds w.h.p.

Chapter 6

Locality Lower Bounds

In Chapter 4, we looked at distributed algorithms for coloring. In particular, we saw that rings and rooted trees can be colored with 3 colors in $\log^* n + O(1)$ rounds. In this chapter, we will reconsider the distributed coloring problem. We will look at a classic lower bound by Nathan Linial that shows that the result of Chapter 4 is tight: Coloring rings (and rooted trees) indeed requires $\Omega(\log^* n)$ rounds. In particular, we will prove a lower bound for coloring in the following setting:

- We consider deterministic, synchronous algorithms.
- Message size and local computations are unbounded.
- We assume that the network is a directed ring with n nodes.
- Nodes have unique labels (identifiers) from 1 to n .

Remarks:

- A generalization of the lower bound to randomized algorithms is possible. Unfortunately, we will however not have time to discuss this.
- Except for restricting to deterministic algorithms, all the conditions above make a lower bound stronger. Any lower bound for synchronous algorithms certainly also holds for asynchronous ones. A lower bound that is true if message size and local computations are not restricted is clearly also valid if we require a bound on the maximal message size or the amount of local computations. Similarly also assuming that the ring is directed and that node labels are from 1 to n (instead of choosing IDs from a more general domain) strengthen the lower bound.
- Instead of directly proving that 3-coloring a ring needs $\Omega(\log^* n)$ rounds, we will prove a slightly more general statement. We will consider deterministic algorithms with time complexity r (for arbitrary r) and derive a lower bound on the number of colors that are needed if we want to properly color an n -node ring with an r -round algorithm. A 3-coloring lower bound can then be derived by taking the smallest r for which an r -round algorithm needs 3 or fewer colors.

Algorithm 20 Synchronous Algorithm: Canonical Form

-
- 1: In r rounds: **send** complete initial state to nodes at distance at most r
 - 2: // do all the communication first
 - 3: Compute output based on complete information about r -neighborhood
 - 4: // do all the computation in the end
-

6.1 Locality

Let us for a moment look at distributed algorithms more generally (i.e., not only at coloring and not only at rings). Assume that initially, all nodes only know their own label (identifier) and potentially some additional input. As information needs at least r rounds to travel r hops, after r rounds, a node v can only learn about other nodes at distance at most r . If message size and local computations are not restricted, it is in fact not hard to see, that in r rounds, a node v can exactly learn all the node labels and inputs up to distance r . As shown by the following lemma, this allows to transform every deterministic r -round synchronous algorithm into a simple canonical form.

Lemma 6.1. *If message size and local computations are not bounded, every deterministic, synchronous r -round algorithm can be transformed into an algorithm of the form given by Algorithm 20 (i.e., it is possible to first communicate for r rounds and then do all the computations in the end).*

Proof. Consider some r -round algorithm \mathcal{A} . We want to show that \mathcal{A} can be brought to the canonical form given by Algorithm 20. First, we let the nodes communicate for r rounds. Assume that in every round, every node sends its complete state to all of its neighbors (remember that there is no restriction on the maximal message size). By induction, after r rounds, every node knows the initial state of all other nodes at distance at most i . Hence, after r rounds, a node v has the combined initial knowledge of all the nodes in its r -neighborhood. We want to show that this suffices to locally (at node v) simulate enough of Algorithm \mathcal{A} to compute all the messages that v receives in the r communication rounds of a regular execution of Algorithm \mathcal{A} .

Concretely, we prove the following statement by induction on i . For all nodes at distance at most $r - i + 1$ from v , node v can compute all messages of the first i rounds of a regular execution of \mathcal{A} . Note that this implies that v can compute all the messages it receives from its neighbors during all r rounds. Because v knows the initial state of all nodes in the r -neighborhood, v can clearly compute all messages of the first round (i.e., the statement is true for $i = 1$). Let us now consider the induction step from i to $i + 1$. By the induction hypothesis, v can compute the messages of the first i rounds of all nodes in its $(r - i + 1)$ -neighborhood. It can therefore compute all messages that are received by nodes in the $(r - i)$ -neighborhood in the first i rounds. This is of course exactly what is needed to compute the messages of round $i + 1$ of nodes in the $(r - i)$ -neighborhood. \square

Remarks:

- It is straightforward to generalize the canonical form to randomized algorithms: Every node first computes all the random bits it needs throughout the algorithm. The random bits are then part of the initial state of a node.

Definition 6.2 (*r*-hop view). *We call the collection of the initial states of all nodes in the *r*-neighborhood of a node *v*, the *r*-hop view of *v*.*

Remarks:

- Assume that initially, every node knows its degree, its label (identifier) and potentially some additional input. The *r*-hop view of a node *v* then includes the complete topology of the *r*-neighborhood (excluding edges between nodes at distance *r*) and the labels and additional inputs of all nodes in the *r*-neighborhood.

Based on the definition of an *r*-hop view, we can state the following corollary of Lemma 6.1.

Corollary 6.3. *A deterministic *r*-round algorithm \mathcal{A} is a function that maps every possible *r*-hop view to the set of possible outputs.*

Proof. By Lemma 6.1, we know that we can transform Algorithm \mathcal{A} to the canonical form given by Algorithm 20. After *r* communication rounds, every node *v* knows exactly its *r*-hop view. This information suffices to compute the output of node *v*. \square

Remarks:

- Note that the above corollary implies that two nodes with equal *r*-hop views have to compute the same output in every *r*-round algorithm.
- For coloring algorithms, the only input of a node *v* is its label. The *r*-hop view of a node therefore is its labeled *r*-neighborhood.
- Since we only consider rings, *r*-hop neighborhoods are particularly simple. The labeled *r*-neighborhood of a node *v* (and hence its *r*-hop view) in a directed ring is simply a $(2r + 1)$ -tuple $(\ell_{-r}, \ell_{-r+1}, \dots, \ell_0, \dots, \ell_r)$ of distinct node labels where ℓ_0 is the label of *v*. Assume that for $i > 0$, ℓ_i is the label of the i^{th} clockwise neighbor of *v* and ℓ_{-i} is the label of the i^{th} counterclockwise neighbor of *v*. A deterministic coloring algorithm for directed rings therefore is a function that maps $(2r + 1)$ -tuples of node labels to colors.
- Consider two *r*-hop views $\mathcal{V}_r = (\ell_{-r}, \dots, \ell_r)$ and $\mathcal{V}'_r = (\ell'_{-r}, \dots, \ell'_r)$. If $\ell'_i = \ell_{i+1}$ for $-r \leq i \leq r - 1$ and if $\ell'_r \neq \ell_r$, the *r*-hop view \mathcal{V}'_r can be the *r*-hop view of a clockwise neighbor of a node with *r*-hop view \mathcal{V}_r . Therefore, every algorithm \mathcal{A} that computes a valid coloring needs to assign different colors to \mathcal{V}_r and \mathcal{V}'_r . Otherwise, there is a ring labeling for which \mathcal{A} assigns the same color to two adjacent nodes.

6.2 The Neighborhood Graph

We will now make the above observations concerning colorings of rings a bit more formal. Instead of thinking of an r -round coloring algorithm as a function from all possible r -hop views to colors, we will use a slightly different perspective. Interestingly, the problem of understanding distributed coloring algorithms can itself be seen as a classical graph coloring problem.

Definition 6.4 (Neighborhood Graph). *For a given family of network graphs \mathcal{G} , the r -neighborhood graph $\mathcal{N}_r(\mathcal{G})$ is defined as follows. The node set of $\mathcal{N}_r(\mathcal{G})$ is the set of all possible labeled r -neighborhoods (i.e., all possible r -hop views). There is an edge between two labeled r -neighborhoods \mathcal{V}_r and \mathcal{V}'_r if \mathcal{V}_r and \mathcal{V}'_r can be the r -hop views of two adjacent nodes.*

Lemma 6.5. *For a given family of network graphs \mathcal{G} , there is an r -round algorithm that colors graphs of \mathcal{G} with c colors iff the chromatic number of the neighborhood graph is $\chi(\mathcal{N}_r(\mathcal{G})) \leq c$.*

Proof. We have seen that a coloring algorithm is a function that maps every possible r -hop view to a color. Hence, a coloring algorithm assigns a color to every node of the neighborhood graph $\mathcal{N}_r(\mathcal{G})$. If two r -hop views \mathcal{V}_r and \mathcal{V}'_r can be the r -hop views of two adjacent nodes u and v (for some labeled graph in \mathcal{G}), every correct coloring algorithm must assign different colors to \mathcal{V}_r and \mathcal{V}'_r . Thus, specifying an r -round coloring algorithm for a family of network graphs \mathcal{G} is equivalent to coloring the respective neighborhood graph $\mathcal{N}_r(\mathcal{G})$. \square

Remarks:

- If an algorithm is non-uniform, i.e., the nodes know n , we can see this as having different neighborhood graphs for different values of n (as opposed to a disconnected neighborhood graph).
- This does not make much of a difference for coloring algorithms on the ring, as we are interested in neighborhoods that are much smaller than n .

Instead of directly defining the neighborhood graph for directed rings, we define directed graphs $\mathcal{B}_{k,n}$ that are closely related to the neighborhood graph. Let k and n be two positive integers and assume that $n \geq k$. The node set of $\mathcal{B}_{k,n}$ contains all k -tuples of increasing node labels ($[n] = \{1, \dots, n\}$):

$$V[\mathcal{B}_{k,n}] = \{(\alpha_1, \dots, \alpha_k) : \alpha_i \in [n], i < j \rightarrow \alpha_i < \alpha_j\} \quad (6.1)$$

For $\underline{\alpha} = (\alpha_1, \dots, \alpha_k)$ and $\underline{\beta} = (\beta_1, \dots, \beta_k)$ there is a directed edge from $\underline{\alpha}$ to $\underline{\beta}$ iff

$$\forall i \in \{1, \dots, k-1\} : \beta_i = \alpha_{i+1}. \quad (6.2)$$

Lemma 6.6. *Viewed as an undirected graph, the graph $\mathcal{B}_{2r+1,n}$ is a subgraph of the r -neighborhood graph of directed n -node rings with node labels from $[n]$.*

Proof. The claim follows directly from the observations regarding r -hop views of nodes in a directed ring from Section 6.1. The set of k -tuples of increasing node labels is a subset of the set of k -tuples of distinct node labels. Two nodes of $\mathcal{B}_{2r+1,n}$ are connected by a directed edge iff the two corresponding r -hop views are connected by a directed edge in the neighborhood graph. Note that if there

is an edge between $\underline{\alpha}$ and $\underline{\beta}$ in $\mathcal{B}_{k,n}$, $\alpha_1 \neq \beta_k$ because the node labels in $\underline{\alpha}$ and $\underline{\beta}$ are increasing. \square

To determine a lower bound on the number of colors an r -round algorithm needs for directed n -node rings, it therefore suffices to determine a lower bound on the chromatic number of $\mathcal{B}_{2r+1,n}$. To obtain such a lower bound, we need the following definition.

Definition 6.7 (Diline Graph). *The directed line graph (diline graph) $\mathcal{DL}(G)$ of a directed graph $G = (V, E)$ is defined as follows. The node set of $\mathcal{DL}(G)$ is $V[\mathcal{DL}(G)] = E$. There is a directed edge $((w, x), (y, z))$ between $(w, x) \in E$ and $(y, z) \in E$ iff $x = y$, i.e., if the first edge ends where the second one starts.*

Lemma 6.8. *If $n > k$, the graph $\mathcal{B}_{k+1,n}$ can be defined recursively as follows:*

$$\mathcal{B}_{k+1,n} = \mathcal{DL}(\mathcal{B}_{k,n}).$$

Proof. The edges of $\mathcal{B}_{k,n}$ are pairs of k -tuples $\underline{\alpha} = (\alpha_1, \dots, \alpha_k)$ and $\underline{\beta} = (\beta_1, \dots, \beta_k)$ that satisfy Conditions (6.1) and (6.2). Because the last $k-1$ labels in $\underline{\alpha}$ are equal to the first $k-1$ labels in $\underline{\beta}$, the pair $(\underline{\alpha}, \underline{\beta})$ can be represented by a $(k+1)$ -tuple $\underline{\gamma} = (\gamma_1, \dots, \gamma_{k+1})$ with $\gamma_1 = \alpha_1$, $\gamma_i = \beta_{i-1} = \alpha_i$ for $2 \leq i \leq k$, and $\gamma_{k+1} = \beta_k$. Because the labels in $\underline{\alpha}$ and the labels in $\underline{\beta}$ are increasing, the labels in $\underline{\gamma}$ are increasing as well. The two graphs $\mathcal{B}_{k+1,n}$ and $\mathcal{DL}(\mathcal{B}_{k,n})$ therefore have the same node sets. There is an edge between two nodes $(\underline{\alpha}_1, \underline{\beta}_1)$ and $(\underline{\alpha}_2, \underline{\beta}_2)$ of $\mathcal{DL}(\mathcal{B}_{k,n})$ if $\underline{\beta}_1 = \underline{\alpha}_2$. This is equivalent to requiring that the two corresponding $(k+1)$ -tuples $\underline{\gamma}_1$ and $\underline{\gamma}_2$ are neighbors in $\mathcal{B}_{k+1,n}$, i.e., that the last k labels of $\underline{\gamma}_1$ are equal to the first k labels of $\underline{\gamma}_2$. \square

The following lemma establishes a useful connection between the chromatic numbers of a directed graph G and its diline graph $\mathcal{DL}(G)$.

Lemma 6.9. *For the chromatic numbers $\chi(G)$ and $\chi(\mathcal{DL}(G))$ of a directed graph G and its diline graph, it holds that*

$$\chi(\mathcal{DL}(G)) \geq \log_2(\chi(G)).$$

Proof. Given a c -coloring of $\mathcal{DL}(G)$, we show how to construct a 2^c coloring of G . The claim of the lemma then follows because this implies that $\chi(G) \leq 2^{\chi(\mathcal{DL}(G))}$.

Assume that we are given a c -coloring of $\mathcal{DL}(G)$. A c -coloring of the diline graph $\mathcal{DL}(G)$ can be seen as a coloring of the edges of G such that no two adjacent edges have the same color. For a node v of G , let S_v be the set of colors of its outgoing edges. Let u and v be two nodes such that G contains a directed edge (u, v) from u to v and let x be the color of (u, v) . Clearly, $x \in S_u$ because (u, v) is an outgoing edge of u . Because adjacent edges have different colors, no outgoing edge (v, w) of v can have color x . Therefore $x \notin S_v$. This implies that $S_u \neq S_v$. We can therefore use these color sets to obtain a vertex coloring of G , i.e., the color of u is S_u and the color of v is S_v . Because the number of possible subsets of $[c]$ is 2^c , this yields a 2^c -coloring of G . \square

Let $\log^{(i)} x$ be the i -fold application of the base-2 logarithm to x :

$$\log^{(1)} x = \log_2 x, \quad \log^{(i+1)} x = \log_2(\log^{(i)} x).$$

Remember from Chapter 4 that

$$\log^* x = 1 \text{ if } x \leq 2, \quad \log^* x = 1 + \min\{i : \log^{(i)} x \leq 2\}.$$

For the chromatic number of $\mathcal{B}_{k,n}$, we obtain

Lemma 6.10. *For all $n \geq 1$, $\chi(\mathcal{B}_{1,n}) = n$. Further, for $n \geq k \geq 2$, $\chi(\mathcal{B}_{k,n}) \geq \log^{(k-1)} n$.*

Proof. For $k = 1$, $\mathcal{B}_{k,n}$ is the complete graph on n nodes with a directed edge from node i to node j iff $i < j$. Therefore, $\chi(\mathcal{B}_{1,n}) = n$. For $k > 2$, the claim follows by induction and Lemmas 6.8 and 6.9. \square

This finally allows us to state a lower bound on the number of rounds needed to color a directed ring with 3 colors.

Theorem 6.11. *Every deterministic, distributed algorithm to color a directed ring with 3 or less colors needs at least $(\log^* n)/2 - 1$ rounds.*

Proof. Using the connection between $\mathcal{B}_{k,n}$ and the neighborhood graph for directed rings, it suffices to show that $\chi(\mathcal{B}_{2r+1,n}) > 3$ for all $r < (\log^* n)/2 - 1$. From Lemma 6.10, we know that $\chi(\mathcal{B}_{2r+1,n}) \geq \log^{(2r)} n$. To obtain $\log^{(2r)} n \leq 2$, we need $r \geq (\log^* n)/2 - 1$. Because $\log_2 3 < 2$, we therefore have $\log^{(2r)} n > 3$ if $r < \log^* n/2 - 1$. \square

Corollary 6.12. *Every deterministic, distributed algorithm to compute an MIS of a directed ring needs at least $\log^* n/2 - O(1)$ rounds.*

Remarks:

- It is straightforward to see that also for a constant $c > 3$, the number of rounds needed to color a ring with c or less colors is $\log^* n/2 - O(1)$.
- There basically (up to additive constants) is a gap of a factor of 2 between the $\log^* n + O(1)$ upper bound of Chapter 4 and the $\log^* n/2 - O(1)$ lower bound of this chapter. It is possible to show that the lower bound is tight, even for undirected rings (for directed rings, this will be part of the exercises).
- The presented lower bound is due to Nathan Linial. The lower bound is also true for randomized algorithms. The generalization for randomized algorithms was done by Moni Naor.
- Alternatively, the lower bound can also be presented as an application of Ramsey's theory. Ramsey's theory is best introduced with an example: Assume you host a party, and you want to invite people such that there are no three people who mutually know each other, and no three people which are mutual strangers. How many people can you invite? This is an example of Ramsey's theorem, which says that for any given integer c , and any given integers n_1, \dots, n_c , there is a Ramsey number $R(n_1, \dots, n_c)$, such that if the edges of a complete graph with $R(n_1, \dots, n_c)$ nodes are colored with c different colors, then for some color i the graph contains some complete subgraph of color i of size n_i . The special case in the party example is looking for $R(3, 3)$.

- Ramsey theory is more general, as it deals with hyperedges. A normal edge is essentially a subset of two nodes; a hyperedge is a subset of k nodes. The party example can be explained in this context: We have (hyper)edges of the form $\{i, j\}$, with $1 \leq i, j \leq n$. Choosing n sufficiently large, coloring the edges with two colors must exhibit a set S of 3 edges $\{i, j\} \subset \{v_1, v_2, v_3\}$, such that all edges in S have the same color. To prove our coloring lower bound using Ramsey theory, we form all hyperedges of size $k = 2r + 1$, and color them with 3 colors. Choosing n sufficiently large, there must be a set $S = \{v_1, \dots, v_{k+1}\}$ of $k + 1$ identifiers, such that all $k + 1$ hyperedges consisting of k nodes from S have the same color. Note that both $\{v_1, \dots, v_k\}$ and $\{v_2, \dots, v_{k+1}\}$ are in the set S , hence there will be two neighboring views with the same color. Ramsey theory shows that in this case n will grow as a power tower (tetration) in k . Thus, if n is so large that k is smaller than some function growing like $\log^* n$, the coloring algorithm cannot be correct.
- The neighborhood graph concept can be used more generally to study distributed graph coloring. It can for instance be used to show that with a single round (every node sends its identifier to all neighbors) it is possible to color a graph with $(1 + o(1))\Delta^2 \ln n$ colors, and that every one-round algorithm needs at least $\Omega(\Delta^2 / \log^2 \Delta + \log \log n)$ colors.
- One may also extend the proof to other problems, for instance one may show that a constant approximation of the minimum dominating set problem on unit disk graphs costs at least log-star time.
- Using r -hop views and the fact that nodes with equal r -hop views have to make the same decisions is the basic principle behind almost all locality lower bounds (in fact, we are not aware of a locality lower bound that does not use this principle). Using this basic technique (but a completely different proof otherwise), it is for instance possible to show that computing an MIS (and many other problems) in a general graph requires at least $\Omega(\sqrt{\log n / \log \log n})$ and $\Omega(\log \Delta / \log \log \Delta)$ rounds.

Chapter 7

Social Networks

Distributed computing is applicable in various contexts. This lecture exemplarily studies one of these contexts, social networks, an area of study whose origins date back a century. To give you a first impression, consider Figure 7.1.

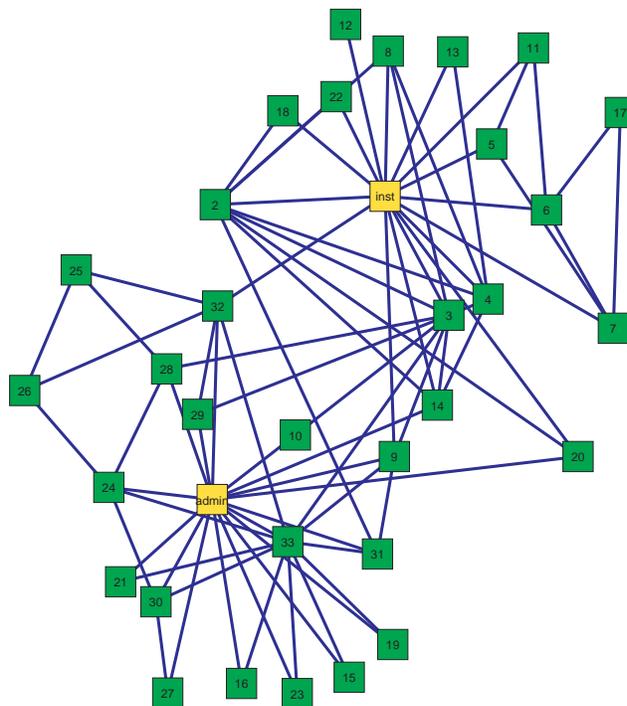


Figure 7.1: This graph shows the social relations between the members of a karate club, studied by anthropologist Wayne Zachary in the 1970s. Two people (nodes) stand out, the instructor and the administrator of the club, both happen to have many friends among club members. At some point, a dispute caused the club to split into two. Can you predict how the club partitioned? (If not, just search the Internet for Zachary and Karate.)

7.1 Small-World Networks

Back in 1929, Frigyes Karinthy published a volume of short stories that postulated that the world was “shrinking” because human beings were connected more and more. Some claim that he was inspired by radio network pioneer Guglielmo Marconi’s 1909 Nobel Prize speech. Despite physical distance, the growing density of human “networks” renders the actual social distance smaller and smaller. As a result, it is believed that any two individuals can be connected through at most five (or so) acquaintances, i.e., within six hops.

- The topic was hot in the 1960s. For instance, in 1964, Marshall McLuhan coined the metaphor “Global Village”. He wrote: “As electrically contracted, the globe is no more than a village”. He argues that due to the almost instantaneous reaction times of new (“electric”) technologies, each individual inevitably feels the consequences of his actions and thus automatically deeply participates in the global society. McLuhan understood what we now can directly observe – real and virtual world are moving together. He realized that the transmission medium, rather than the transmitted information is at the core of change, as expressed by his famous phrase “the medium is the message”.
- This idea has been followed ardently in the 1960s by several sociologists, first by Michael Gurevich, later by Stanley Milgram. Milgram wanted to know the average path length between two “random” humans, by using various experiments, generally using randomly chosen individuals from the US Midwest as starting points, and a stockbroker living in a suburb of Boston as target. The starting points were given name, address, occupation, plus some personal information about the target. They were asked to send a letter to the target. However, they were not allowed to *directly* send the letter, rather, they had to pass it to somebody they knew on first-name basis and that they thought to have a higher probability to know the target person. This process was repeated, until somebody knew the target person, and could deliver the letter. Shortly after starting the experiment, letters have been received. Most letters were lost during the process, but if they arrived, the average path length was about 5.5. The observation that the entire population is connected by short acquaintance chains got later popularized by the terms “six degrees of separation” and “small world”.
- Statisticians tried to explain Milgram’s experiments, by essentially giving network models that allowed for short diameters, i.e., each node is connected to each other node by only a few hops. Until today there is a thriving research community in statistical physics that tries to understand network properties that allow for “small world” effects.
- One of the keywords in this area are power-law graphs, networks where node degrees are distributed according to a power-law distribution, i.e. the number of nodes with degree δ is proportional to $\delta^{-\alpha}$, for some $\alpha > 1$. Such power-law graphs have been witnessed in many application areas, apart from social networks also in the web, or in Biology or Physics.

- Obviously, two power-law graphs might look and behave completely differently, even if α and the number of edges is exactly the same.

One well-known model towards this end is the Watts-Strogatz model. Watts and Strogatz argued that social networks should be modeled by a combination of two networks: As the basis we take a network that has a large cluster coefficient ...

Definition 7.1. *The cluster coefficient of a network is defined by the probability that two friends of a node are likely to be friends as well, summing up over all the nodes.*

..., then we augment such a graph with random links, every node for instance points to a constant number of other nodes, chosen uniformly at random. This augmentation represents acquaintances that connect nodes to parts of the network that would otherwise be far away.

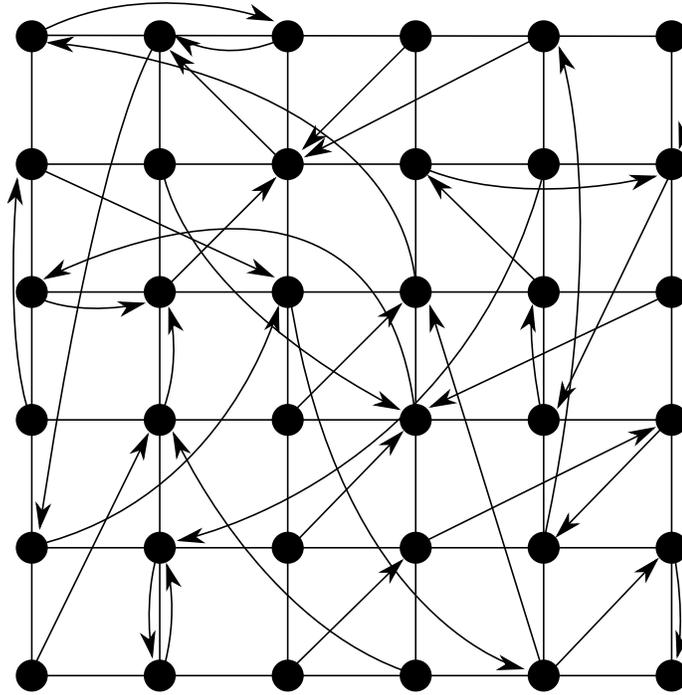
Remarks:

- Without further information, knowing the cluster coefficient is of questionable value: Assume we arrange the nodes in a grid. Technically, if we connect each node to its four closest neighbors, the graph has cluster coefficient 0, since there are no triangles; if we instead connect each node with its eight closest neighbors, the cluster coefficient is $3/7$. The cluster coefficient is quite different, even though both networks have similar characteristics.

This is interesting, but not enough to really understand what is going on. For Milgram's experiments to work, it is not sufficient to connect the nodes in a certain way. In addition, the nodes *themselves* need to know how to forward a message to one of their neighbors, even though they cannot know whether that neighbor is really closer to the target. In other words, nodes are not just following physical laws, but they make decisions themselves. In contrast to those mathematicians that worked on the problem earlier, Jon Kleinberg understood that Milgram's experiment essentially shows that social networks are "navigable", and that one can only explain it in terms of a *greedy routing*.

In particular, Kleinberg set up an artificial network with nodes on a grid topology, plus some additional random links per node. In a quantitative study he showed that the random links need a specific distance distribution to allow for efficient greedy routing. This distribution marks the sweet spot for any navigable network.

Definition 7.2 (Augmented Grid). *We take $n = m^2$ nodes $(i, j) \in V = \{1, \dots, m\}^2$ that are identified with the lattice points on an $m \times m$ grid. We define the distance between two nodes (i, j) and (k, ℓ) as $d((i, j), (k, \ell)) = |k - i| + |\ell - j|$ as the distance between them on the $m \times m$ lattice. The network is modeled using a parameter $\alpha \geq 0$. Each node u has a directed edge to every lattice neighbor. These are the local contacts of a node. In addition, each node also has an additional random link (the long-range contact). For all u and v , the long-range contact of u points to node v with probability proportional to $d(u, v)^{-\alpha}$, i.e., with probability $d(u, v)^{-\alpha} / \sum_{w \in V \setminus \{u\}} d(u, w)^{-\alpha}$. Figure 7.2 illustrates the model.*

Figure 7.2: Augmented grid with $m = 6$ **Remarks:**

- The network model has the following geographic interpretation: nodes (individuals) live on a grid and know their neighbors on the grid. Further, each node has some additional acquaintances throughout the network.
- The parameter α controls how the additional neighbors are distributed across the grid. If $\alpha = 0$, long-range contacts are chosen uniformly at random (as in the Watts-Strogatz model). As α increases, long-range contacts become shorter on average. In the extreme case, if $\alpha \rightarrow \infty$, all long-range contacts are to immediate neighbors on the grid.
- It can be shown that as long as $\alpha \leq 2$, the diameter of the resulting graph is polylogarithmic in n (polynomial in $\log n$) with high probability. In particular, if the long-range contacts are chosen uniformly at random ($\alpha = 0$), the diameter is $O(\log n)$.

Since the augmented grid contains random links, we do not know anything for sure about how the random links are distributed. In theory, all links could point to the same node! However, this is almost certainly not the case. Formally this is captured by the term *with high probability*.

Definition 7.3 (With High Probability). *Some probabilistic event is said to occur with high probability (w.h.p.), if it happens with a probability $p \geq 1 - 1/n^c$, where c is a constant. The constant c may be chosen arbitrarily, but it is considered constant with respect to Big-O notation.*

Remarks:

- For instance, a running time bound of $c \log n$ or $e^{c^1} \log n + 5000c$ with probability at least $1 - 1/n^c$ would be $O(\log n)$ w.h.p., but a running time of n^c would not be $O(n)$ w.h.p. since c might also be 50.
- This definition is very powerful, as any polynomial (in n) number of statements that hold w.h.p. also holds w.h.p. at the same time, regardless of any dependencies between random variables!

Theorem 7.4. *The diameter of the augmented grid with $\alpha = 0$ is $O(\log n)$ with high probability.*

Proof Sketch. For simplicity, we will only show that we can reach a node w starting from some node v . However, it can be shown that (essentially) each of the intermediate claims holds with high probability, which then by means of the union bound yields that *all* of the claims hold simultaneously with high probability for *all* pairs of nodes.

Let N_g be the $\lceil \log n \rceil$ -hop neighborhood of v on the grid, containing $\Omega(\log^2 n)$ nodes. Each of the nodes in N_g has a random link, probably leading to distant parts of the graph. As long as we have reached only $o(n)$ nodes, any new random link will with probability $1 - o(1)$ lead to a node for which none of its grid neighbors has been visited yet. Thus, in expectation we find almost $|N_g|$ new nodes whose neighbors are “fresh”. Using their grid links, we will reach $(4 - o(1))|N_g|$ more nodes within one more hop. If bad luck strikes, it could still happen that many of these links lead to a few nodes, already visited nodes, or nodes that are very close to each other. But that is very unlikely, as we have lots of random choices! Indeed, it can be shown that not only in expectation, but with high probability $(5 - o(1))|N_g|$ many nodes are reached this way.

Because all these shiny new nodes have (so far unused) random links, we can repeat this reasoning inductively, implying that the number of nodes grows by (at least) a constant factor for every two hops. Thus, after $O(\log n)$ hops, we will have reached $n/\log n$ nodes (which is still small compared to n). Finally, consider the expected number of links from these nodes that enter the $(\log n)$ -neighborhood of some target node w with respect to the grid. Since this neighborhood consists of $\Omega(\log^2 n)$ nodes, in expectation $\Omega(\log n)$ links come close enough to w . This is large enough to almost guarantee that this happens. Summing everything up, we still used merely $O(\log n)$ hops in total to get from v to w . □

This shows that for $\alpha = 0$ (and in fact for all $\alpha \leq 2$), the resulting network has a small diameter. Recall however that we also wanted the network to be navigable. For this, we consider a simple greedy routing strategy (Algorithm 21).

Algorithm 21 Greedy Routing

- 1: **while** not at destination **do**
 - 2: go to a neighbor which is closest to destination (considering grid distance only)
 - 3: **end while**
-

Lemma 7.5. *In the augmented grid, Algorithm 21 finds a routing path of length at most $2(m-1) \in O(\sqrt{n})$.*

Proof. Because of the grid links, there is always a neighbor which is closer to the destination. Since with each hop we reduce the distance to the target at least by one in one of the two grid dimensions, we will reach the destination within $2(m-1)$ steps. \square

This is not really what Milgram's experiment promises. We want to know how much the additional random links speed up the process. To this end, we first need to understand how likely it is that two nodes u and v are connected by a random link in terms of n and their distance $d(u, v)$.

Lemma 7.6. *Node u 's random link leads to a node v with probability*

- $\Theta(1/(d(u, v)^\alpha m^{2-\alpha}))$ if $\alpha < 2$.
- $\Theta(1/(d(u, v)^2 \log n))$ if $\alpha = 2$,
- $\Theta(1/d(u, v)^\alpha)$ if $\alpha > 2$.

Moreover, if $\alpha > 2$, the probability to see a link of length at least d is in $\Theta(1/d^{\alpha-2})$.

Proof. For $\alpha \neq 2$, we have that

$$\sum_{w \in V \setminus \{u\}} \frac{1}{d(u, w)^\alpha} \in \sum_{r=1}^m \frac{\Theta(r)}{r^\alpha} = \Theta \left(\int_{r=1}^m \frac{1}{r^{\alpha-1}} dr \right) = \Theta \left(\left[\frac{r^{2-\alpha}}{2-\alpha} \right]_1^m \right).$$

If $\alpha < 2$, this gives $\Theta(m^{2-\alpha})$, if $\alpha > 2$, it is in $\Theta(1)$. If $\alpha = 2$, we get

$$\sum_{w \in V \setminus \{u\}} \frac{1}{d(u, w)^\alpha} \in \sum_{r=1}^m \frac{\Theta(r)}{r^2} = \Theta(1) \cdot \sum_{r=1}^m \frac{1}{r} = \Theta(\log m) = \Theta(\log n).$$

Multiplying with $d(u, v)^\alpha$ yields the first three bounds.

For the last statement, compute

$$\sum_{\substack{w \in V \\ d(u, v) \geq d}} \Theta(1/d(u, v)^\alpha) = \Theta \left(\int_{r=d}^m \frac{r}{r^\alpha} dr \right) = \Theta \left(\left[\frac{r^{2-\alpha}}{2-\alpha} \right]_d^m \right) = \Theta(1/d^{\alpha-2}).$$

\square

Remarks:

- For $\alpha \neq 2$, this is bad news for the greedy routing algorithm, as it will take $n^{\Omega(1)} = m^{\Omega(1)}$ expected steps to reach the destination. This is disappointing, we were hoping for something polylogarithmic.
- If $\alpha < 2$, in distance $m^{(2-\alpha)/3}$ to the target are $m^{2(2-\alpha)/3}$ many nodes. Thus it takes $\Theta(m^{(2-\alpha)/3})$ links in expectation to find a link that comes that close to the destination. Without finding such a link, we have to go at least this far using grid links only.

- If $\alpha > 2$, it takes $\Theta(m^{(\alpha-2)/(\alpha-1)})$ steps until we see a link of length at least $m^{1/(\alpha-1)}$ in expectation. Without such links, it takes at least $m/m^{1/(\alpha-1)} = m^{(\alpha-2)/(\alpha-1)}$ steps to travel a distance of m .
- Any algorithm that uses only the information on long-range contacts that it can collect at the so far visited nodes cannot be faster.
- However, the case $\alpha = 2$ looks more promising.

Definition 7.7 (Phase). *Consider routing from a node u to a node v and assume that we are at some intermediate node w . We say that we are in phase j at node w if the lattice distance $d(w, v)$ to the target node v is between $2^j < d(w, v) \leq 2^{j+1}$.*

Remarks:

- Enumerating the phases in decreasing order is useful, as notation becomes less cumbersome.
- There are $\lceil \log m \rceil \in O(\log n)$ phases.

Lemma 7.8. *Assume that we are in phase j at node w when routing from u to v . The probability for getting to phase $j - 1$ in one step is at least $\Omega(1/\log n)$.*

Proof. Let B_j be the set of nodes x with $d(x, v) \leq 2^j$. We get from phase j to phase $j - 1$ if the long-range contact of node w points to some node in B_j . Note that we always make progress while following the greedy routing path. Therefore, we have not seen node w before and the long-range contact of w points to a random node that is independent of anything seen on the path from u to w .

For all nodes $x \in B_j$, we have $d(w, x) \leq d(w, v) + d(x, v) \leq 2^{j+1} + 2^j < 2^{j+2}$. Hence, for each node $x \in B_j$, the probability that the long-range contact of w points to x is $\Omega(1/2^{2j+4} \log n)$. Further, the number of nodes in B_j is at least $(2^j)^2/2 = 2^{2j-1}$. Hence, the probability that some node in B_j is the long range contact of w is at least

$$\Omega\left(|B_j| \cdot \frac{1}{2^{2j+4} \log n}\right) = \Omega\left(\frac{2^{2j-1}}{2^{2j+4} \log n}\right) = \Omega\left(\frac{1}{\log n}\right). \quad \square$$

Theorem 7.9. *Consider the greedy routing path from a node u to a node v on an augmented grid with parameter $\alpha = 2$. The expected length of the path is $O(\log^2 n)$.*

Proof. We already observed that the total number of phases is $O(\log n)$ (the distance to the target is halved when we go from phase j to phase $j - 1$). At each point during the routing process, the probability of proceeding to the next phase is at least $\Omega(1/\log n)$. Let X_j be the number of steps in phase j . Because the probability for ending the phase is $\Omega(1/\log n)$ in each step, in expectation we need $O(\log n)$ steps to proceed to the next phase, i.e., $\mathbb{E}[X_j] \in O(\log n)$. Let $X = \sum_j X_j$ be the total number of steps of the routing process. By linearity of expectation, we have

$$\mathbb{E}[X] = \sum_j \mathbb{E}[X_j] \in O(\log^2 n). \quad \square$$

7.2 Propagation Studies

In networks, nodes may influence each other's behavior and decisions. There are many applications where nodes influence their neighbors, e.g. they may impact their opinions, or they may bias what products they buy, or they may pass on a disease.

On a beach (modeled as a line segment), it is best to place an ice cream stand right in the middle of the segment, because you will be able to "control" the beach most easily. What about the second stand, where should it settle? The answer generally depends on the model, but assuming that people will buy ice cream from the stand that is closer, it should go right next to the first stand.

Rumors can spread astoundingly fast through social networks. Traditionally this happens by word of mouth, but with the emergence of the Internet and its possibilities new ways of rumor propagation are available. People write email, use instant messengers or publish their thoughts in a blog. Many factors influence the dissemination of rumors. It is especially important where in a network a rumor is initiated and how convincing it is. Furthermore the underlying network structure decides how fast the information can spread and how many people are reached. More generally, we can speak of diffusion of information in networks. The analysis of these diffusion processes can be useful for viral marketing, e.g. to target a few influential people to initiate marketing campaigns. A company may wish to distribute the rumor of a new product via the most influential individuals in popular social networks such as Facebook. A second company might want to introduce a competing product and has hence to select where to seed the information to be disseminated. Rumor spreading is quite similar to our ice cream stand problem.

More formally, we may study propagation problems in graphs. Given a graph, and two players. Let the first player choose a seed node u_1 ; afterwards let the second player choose a seed node u_2 , with $u_2 \neq u_1$. The goal of the game is to maximize the number of nodes that are closer to one's own seed node.

In many graphs it is an advantage to choose first. In a star graph for instance the first player can choose the center node of the star, controlling all but one node. In some other graphs, the second player can at least score even. But is there a graph where the second player has an advantage?

Theorem 7.10. *In a two player rumor game where both players select one node to initiate their rumor in the graph, the first player does not always win.*

Proof. See Figure 7.3 for an example where the second player will always win, regardless of the decision the first player. If the first player chooses the node x_0 in the center, the second player can select x_1 . Choice x_1 will be outwitted by x_2 , and x_2 itself can be answered by z_1 . All other strategies are either symmetric, or even less promising for the first player. \square

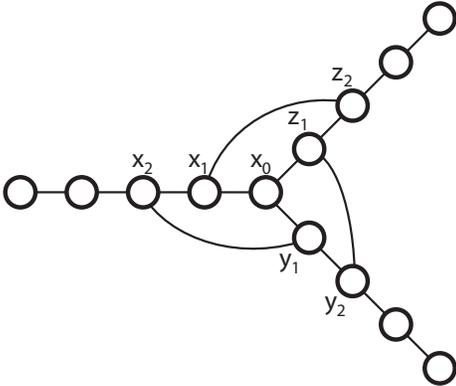


Figure 7.3: Counter example.