ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Principles of

Distributed Computing

Roger Wattenhofer
wattenhofer@tik.ee.ethz.ch

Assistant: Christoph Lenzen (lenzen@tik.ee.ethz.ch)

Spring 2010

Introduction

What is Distributed Computing?

In the last few decades, we have experienced an unprecedented growth in the area of distributed systems and networks. Distributed computing now encompasses many of the activities occurring in today's computer and communications world. Indeed, distributed computing appears in quite diverse application areas: Typical "old school" examples are parallel computers or the Internet. More recent application examples of distributed systems include peer-to-peer systems, sensor networks, or multi-core architectures.

These applications have in common that 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. Due to such differences, many different models and parameters are studied in the area of distributed computing. In some systems, the nodes operate synchronously, and 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: nodes may communicate by exchanging messages, or by means of shared memory. 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. Sometimes the nodes can be assumed to work correctly, at times they may exhibit failures. In contrast to a single-node system, distributed systems may still function correctly despite failures as other nodes can take over the work of the failed nodes. There are different kinds of failures that can be considered: nodes may just crash, or they might exhibit an arbitrary, erroneous behavior, maybe even to a degree where it cannot be distinguished from malicious (also known as Byzantine) behavior. It is also possible that the nodes do follow the rules, however they tweak the parameters to get the most out of the system; in other words, the nodes act selfishly.

Apparently, there are many models (and even more combinations of models) that can be studied. We will not discuss them in greater detail now, but simply define them when we use them. Towards the end of the course a general picture should emerge. Hopefully!

This course introduces the basic principles of distributed computing, highlighting common themes and techniques. In particular, we study some of the fundamental issues underlying the design of distributed systems:

- Communication: Communication does not come for free; often communication cost dominates the cost of local processing or storage. Sometimes we even assume that everything but communication is free.

- Coordination: How can you coordinate a distributed system so that it performs some task efficiently?

- Fault-tolerance: As mentioned above, one major advantage of a distributed system is that even in the presence of failures the system as a whole may survive.

- Locality: Networks keep growing. Luckily, global information is not always needed to solve a task, often it is sufficient if nodes talk to their neighbors. In this course, we will address the fundamental question in distributed computing whether a local solution is possible for a wide range of problems.

- Parallelism: How fast can you solve a task if you increase your computational power, e.g., by increasing the number of nodes that can share the workload? How much parallelism is possible for a given problem?

- Symmetry breaking: Sometimes some nodes need to be selected to orchestrate the computation (and the communication). This is typically achieved by a technique called *symmetry breaking*.

- Synchronization: How can you implement a synchronous algorithm in an asynchronous system?

- Uncertainty: If we need to agree on a single term that fittingly describes this course, it is probably "uncertainty". As the whole system is distributed, the nodes cannot know what other nodes are doing at this exact moment, and the nodes are required to solve the tasks at hand despite the lack of global knowledge.

Finally, there are also a few areas that we will not cover in this course, mostly because these topics have become so important that they deserve and have their own courses. Examples for such topics are distributed programming, software engineering, and also security and cryptography.

In summary, in this class we explore essential algorithmic ideas and lower bound techniques, basically the "pearls" of distributed computing and network algorithms. We will cover a fresh topic every week.

Have fun!

Chapter 1

Vertex Coloring

1.1 Introduction

Vertex coloring is an infamous graph theory problem. It is also a useful toy example to see the style of this course already in the first lecture. 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 1.1 (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 suboptimal. There is a tradeoff between the optimality of a solution (efficacy), and the work/time needed to compute the solution (efficiency).

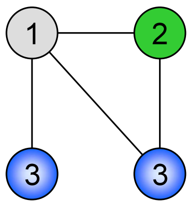


Figure 1.1: 3-colorable graph with a valid coloring.

Assumption 1.2 (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 $1, \ldots, n$.
- It is easy to see that node identifiers (as defined in Assumption 1.2) solve the coloring problem 1.1, but not very well (essentially requiring n colors). How many colors are needed at least is a well-studied problem.

Definition 1.3 (Chromatic Number)**.** *Given an undirected Graph* $G = (V, E)$*, the* chromatic number $\chi(G)$ *is the minimum number of colors to solve Problem 1.1.*

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

Algorithm 1 Greedy Sequential

```
1: while ∃ 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 1.4 (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 1.5 (Analysis of Algorithm 1)**.** *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 $\{1, \ldots, \Delta + 1\}$.

Remarks:

- In Definition 1.7 we will see what is meant by "step".
- 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 1 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 2 First Free

```
Require: Node Coloring {e.g., node IDs as defined in Assumption 1.2}
  Give $v$ the smallest admissible color {i.e., the smallest node color not used by
  any neighbor}
```

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.

Definition 1.6 (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 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.

Algorithm 3 Reduce

```
Assume that initially all nodes have ID's (Assumption 1.2)
Each node $v$ executes the following code
node $v$ sends its ID to all neighbors
node $v$ receives IDs of neighbors
while node $v$ has an uncolored neighbor with higher ID do
  node $v$ sends "undecided" to all neighbors
  node $v$ receives new decisions from neighbors
end while
node $v$ chooses a free color using subroutine First Free (Procedure 2)
node $v$ informs all its neighbors about its choice
```

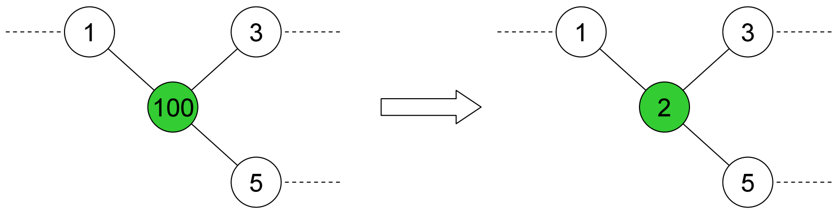


Figure 1.2: Vertex 100 receives the lowest possible color.

Definition 1.7 (Time Complexity)**.** *For synchronous algorithms (as defined in 1.6) the* time complexity *is the number of rounds until the algorithm terminates.*

Remarks:

- The algorithm terminates when the last processor has decided to terminate.
- To guarantee correctness the procedure requires a legal input (i.e., pairwise different node IDs).

Theorem 1.8 (Analysis of Algorithm 3)**.** *Algorithm 3 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.

1.2 Coloring Trees

Lemma 1.9. $\chi(Tree) \leq 2$

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:

Algorithm 4 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**
```

Remarks:

- With the proof of Lemma 1.9, Algorithm 4 is correct.
- How can we determine a root in a tree if it is not already given? We will figure that out later.
- 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 ...!

Definition 1.10 (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 1.6) 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.

Definition 1.11 (Time Complexity)**.** *For asynchronous algorithms (as defined in 1.6) 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 1.12 (Message Complexity)**.** *The* message complexity *of a synchronous or asynchronous algorithm is determined by the number of messages exchanged (again every legal input, every execution scenario).*

Theorem 1.13 (Analysis of Algorithm 4)**.** *Algorithm 4 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 1.14 (Log-Star)**.**
$\forall x \leq 2 : \ \log^* x := 1 \qquad \forall x > 2 : \ \log^* x := 1 + \log^*(\log x)$

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 5 "6-Color"

```
1: Assume that initially the vertices are legally colored. Using Assumption 1.2
   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 5 executed on the following part of a tree:

Grand-parent	0010110000	$\rightarrow$	10010	$\rightarrow$	...
Parent	1010010000	$\rightarrow$	01010	$\rightarrow$	111
Child	0110010000	$\rightarrow$	10001	$\rightarrow$	001

Theorem 1.15 (Analysis of Algorithm 5). *Algorithm 5 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,..., 5).

- Can one reduce the number of colors in only constant steps? Note that algorithm 3 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 6 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 1.16 (Analysis of Algorithm 6)**.** *Algorithm 6 preserves coloring legality; also siblings are monochromatic.*

Now Algorithm 3 (Reduce) can be used to reduce the number of used colors from six to three.

Algorithm 7 Six-2-Three

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

Theorem 1.17 (Analysis of Algorithm 7)**.** *Algorithm 7 colors a tree with three colors in time $O(\log^* n)$.*

Remarks:

- The term $O()$ used in Theorem 1.15 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

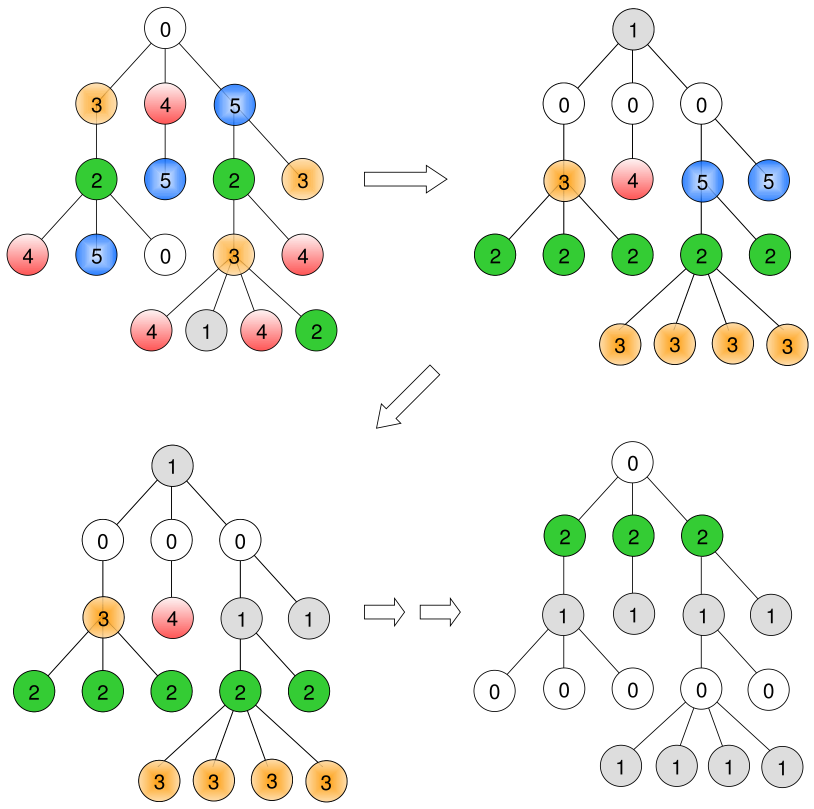


Figure 1.3: Possible execution of Algorithm 7.

as in 6-color (Algorithm 5). 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 3 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 10. 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 2

Leader Election

2.1 Anonymous Leader Election

Some algorithms (e.g. the slow tree coloring algorithm 4) 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 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 "drosophila" 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 2.1 (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 2.2 (Anonymous)**.** *A system is anonymous if nodes do not have unique identifiers.*

Definition 2.3 (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 2.4. *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 1.6). 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 2.5 (Anonymous Leader Election)**.** *Deterministic leader election in an anonymous ring is impossible.*

Proof (with Lemma 2.4): 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 2.1 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 2.5 also holds for other symmetric network topologies (e.g., complete graphs, complete bipartite graphs, ...).
- Note that Theorem 2.5 does not hold for randomized algorithms; if nodes are allowed to toss a coin, symmetries can be broken.

2.2 Asynchronous Ring

We first concentrate on the asynchronous model from Definition 1.10. Throughout this section we assume non-anonymity; each node has a unique identifier as proposed in Assumption 1.2. 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.

Theorem 2.6 (Analysis of Algorithm 8)**.** *Algorithm 8 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 1.11) we assume that

Algorithm 8 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
```

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 8 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 2.7 (Analysis of Algorithm 9)**.** *Algorithm 9 is correct. The time complexity is $O(n)$. The message complexity is $O(n \log n)$.*

Proof: Correctness is as in Theorem 2.6. The time complexity is $O(n)$ since the node with maximum identifier z sends messages with round-trip times $2, 4, 8, 16, \ldots, 2 \cdot 2^k$ 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 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.

Algorithm 9 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.
```

2.3 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 9 is asymptotically optimal.

Definition 2.8 (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 2.9 (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 2.10. *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 2.11. *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 $n/4$ more messages when being scheduled. This need not to be sent over the closed link, but because they are *caused* by a message over this link, they cannot involve any message along the other open link in distance $n/2$. We schedule this edge and the resulting $n/4$ messages, and leave the other open.

Lemma 2.12. *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 2.10 which implies that $M(2) \geq 1 = \frac{2}{4}(\log 2 + 1)$. For the induction step, using Lemma 2.11 and the induction hypothesis we have

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

□

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 2.12 we get:

Theorem 2.13 (Asynchronous Leader Election Lower Bound)**.** *Any uniform leader election algorithm for asynchronous rings has $\Omega(n \log n)$ message complexity.*

2.4 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 10 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 Erdös 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 11 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 dis-
   card 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 12 Echo

```
Require: This algorithm is initiated at the leaves.
1: A leave 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 13)**.** *The time complexity of Algorithm 13 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 13 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 14 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 14)**.** *The time complexity of Algorithm 14 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 13 has the better message complexity and Algorithm 14 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 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 15 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 15 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 13.

Theorem 3.11 (Analysis of Algorithm 15)**.** *The time complexity of Algorithm 15 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 15 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

Distributed Sorting

"Indeed, I believe that virtually *every* important aspect of programming arises somewhere in the context of sorting [and searching]!"

– Donald E. Knuth, The Art of Computer Programming

In this chapter we study a classic problem in computer science—sorting—from a distributed computing perspective. In contrast to an orthodox single-processor sorting algorithm, no node has access to all data, instead the to-be-sorted values are *distributed*. Distributed sorting then boils down to:

Definition 4.1 (Sorting)**.** *We choose a graph with n nodes $v_1, \dots, v_n$. Initially each node stores a value. After applying a sorting algorithm, node v_k stores the k^{th} smallest value.*

Remarks:

- What if we route all values to the same central node v, let v sort the values locally, and then route them to the correct destinations?! According to the message passing model studied in the first few chapters this is perfectly legal. With a star topology sorting finishes in $O(1)$ time!
- Indeed, if we allow the All-to-All model of Chapter **??** we can even sort n values in a single round! So we need to make sure that we restrict our model appropriately:

Definition 4.2 (Node Contention)**.** *In each step of a synchronous algorithm, each node can only send and receive $O(1)$ messages containing $O(1)$ values, no matter how many neighbors the node has.*

Remarks:

- Using Definition 4.2 sorting on a star graph takes linear time.

4.1 Array & Mesh

To get a better intuitive understanding of distributed sorting, we start with two simple topologies, the array and the mesh. Let us begin with the array:

Algorithm 16 Odd/Even Sort

```
1: Given an array of $n$ nodes $(v_1, \dots, v_n)$, each storing a value (not sorted).
2: repeat
3:    Compare and exchange the values at nodes $i$ and $i+1$, $i$ odd
4:    Compare and exchange the values at nodes $i$ and $i+1$, $i$ even
5: until done
```

Remarks:

- The compare and exchange primitive in Algorithm 16 is defined as follows: Let the value stored at node i be v_i. After the compare and exchange node i stores value $\min(v_i, v_{i+1})$ and node $i+1$ stores value $\max(v_i, v_{i+1})$.
- How fast is the algorithm, and how can we prove correctness/efficiency?
- The most interesting proof uses the so-called 0-1 Sorting Lemma. It allows us to restrict our attention to an input of 0's and 1's only, and works for any "oblivious comparison-exchange" algorithm. (Oblivious means: Whether you exchange two values must only depend on the relative order of the two values, and not on anything else.)

Lemma 4.3 (0-1 Sorting Lemma)**.** *If an oblivious comparison-exchange algorithm sorts all inputs of 0's and 1's, then it sorts arbitrary inputs.*

Proof. We prove the opposite direction (does not sort arbitrary inputs $\Rightarrow$ does not sort 0's and 1's). Assume that there is an input $x = x_1, \dots, x_n$ that is not sorted correctly. Then there is a smallest value k such that the value at node v_k after running the sorting algorithm is strictly larger than the k^{th} smallest value $x(k)$. Define an input $x_i^* = 0 \Leftrightarrow x_i \leq x(k)$, $x_i^* = 1$ else. Whenever the algorithm compares a pairs of 1's or 0's, it is not important whether it exchanges the values or not, so we may simply assume that it does the same as on the input x. On the other hand, whenever the algorithm compares some values $x_i^* = 0$ and $x_j^* = 1$, this means that $x_i \leq x(k) < x_j$. Therefore, in this case the respective compare-exchange operation will do the same on both inputs. We conclude that the algorithm will order x^* the same way as x, i.e., the output with only 0's and 1's will also not be correct. $\square$

Theorem 4.4. *Algorithm 16 sorts correctly in n steps.*

Proof. Thanks to Lemma 4.3 we only need to consider an array with 0's and 1's. Let j_1 be the node with the rightmost (highest index) 1. If j_1 is odd (even) it will move in the first (second) step. In any case it will move right in every following step until it reaches the rightmost node v_n. Let j_k be the node with the k^{th} rightmost 1. We show by induction that j_k is not "blocked" anymore (constantly moves until it reaches destination!) after step k. We have already anchored the induction at $k = 1$. Since j_{k-1} moves after step $k-1$, j_k gets a right 0-neighbor for each step after step k. (For matters of presentation we omitted a couple of simple details.) $\square$

Algorithm 17 Shearsort

```
1: We are given a mesh with $m$ rows and $m$ columns, $m$ even, $n = m^2$.
2: The sorting algorithm operates in phases, and uses the odd/even sort algorithm on rows or columns.
3: repeat
4:    In the odd phases $1, 3, \ldots$ we sort all the rows, in the even phases $2, 4, \ldots$ we sort all the columns, such that:
5:    Columns are sorted such that the small values move up.
6:    Odd rows $(1, 3, \ldots, m-1)$ are sorted such that small values move left.
7:    Even rows $(2, 4, \ldots, m)$ are sorted such that small values move right.
8: until done
```

Remarks:

- Linear time is not very exciting, maybe we can do better by using a different topology? Let's try a mesh (a.k.a. grid) topology first.

Theorem 4.5. *Algorithm 17 sorts n values in $\sqrt{n}(\log n + 1)$ time in snake-like order.*

Proof. Since the algorithm is oblivious, we can use Lemma 4.3. We show that after a row and a column phase, half of the previously unsorted rows will be sorted. More formally, let us call a row with only 0's (or only 1's) *clean*, a row with 0's *and* 1's is *dirty*. At any stage, the rows of the mesh can be divided into three regions. In the north we have a region of all-0 rows, in the south all-1 rows, in the middle a region of dirty rows. Initially all rows can be dirty. Since neither row nor column sort will touch already clean rows, we can concentrate on the dirty rows.

First we run an odd phase. Then, in the even phase, we run a peculiar column sorter: We group two consecutive dirty rows into pairs. Since odd and even rows are sorted in opposite directions, two consecutive dirty rows look as follows:

$$00000 \ldots 11111$$

$$11111 \ldots 00000$$

Such a pair can be in one of three states. Either we have more 0's than 1's, or more 1's than 0's, or an equal number of 0's and 1's. Column-sorting each pair will give us at least one clean row (and two clean rows if "$|0| = |1|$"). Then move the cleaned rows north/south and we will be left with half the dirty rows.

At first glance it appears that we need such a peculiar column sorter. However, any column sorter sorts the columns in exactly the same way (we are very grateful to have Lemma 4.3!).

All in all we need $2 \log m = \log n$ phases to remain only with 1 dirty row in the middle which will be sorted (not cleaned) with the last row-sort. □

Remarks:

- There are algorithms that sort in $3m + o(m)$ time on an m by m mesh (by diving the mesh into smaller blocks). This is asymptotically optimal, since a value might need to move $2m$ times.
- Such a $\sqrt{n}$-sorter is cute, but we are more ambitious. There are non-distributed sorting algorithms such as quicksort, heapsort, or mergesort that sort n values in (expected) $O(n \log n)$ time. Using our n-fold parallelism effectively we might therefore hope for a distributed sorting algorithm that sorts in time $O(\log n)$!

4.2 Sorting Networks

In this section we construct a graph topology which is carefully manufactured for sorting. This is a deviation to previous chapters where we always had to work with the topology that was given to us. In many application areas (e.g. peer-to-peer networks, communication switches, systolic hardware) it is indeed possible (in fact, crucial!) that an engineer can build the topology best suited for her application.

Definition 4.6 (Sorting Networks)**.** *A* comparator *is a device with two inputs x, y and two outputs x', y' such that $x' = min(x, y)$ and $y' = max(x, y)$. We construct so-called* comparison networks *that consist of wires that connect comparators (the output port of a comparator is sent to an input port of another comparator). Some wires are not connected to output comparators, and some are not connected to input comparators. The first are called input wires of the comparison network, the second output wires. Given n values on the input wires, a* sorting network *ensures that the values are sorted on the output wires.*

Remarks:

- The odd/even sorter explained in Algorithm 16 can be described as a sorting network.
- Often we will draw all the wires on n horizontal lines (n being the “width” of the network). Comparators are then vertically connecting two of these lines.
- Note that a sorting network is an oblivious comparison-exchange network. Consequently we can apply Lemma 4.3 throughout this section. An example sorting network is depicted in Figure 4.1.

Definition 4.7 (Depth)**.** *The depth of an input wire is* 0*. The depth of a comparator is the maximum depth of its input wires plus one. The depth of an output wire of a comparator is the depth of the comparator. The depth of a comparison network is the maximum depth (of an output wire).*

Definition 4.8 (Bitonic Sequence)**.** *A* bitonic sequence *is a sequence of numbers that first monotonically increases, and then monotonically decreases, or vice versa.*

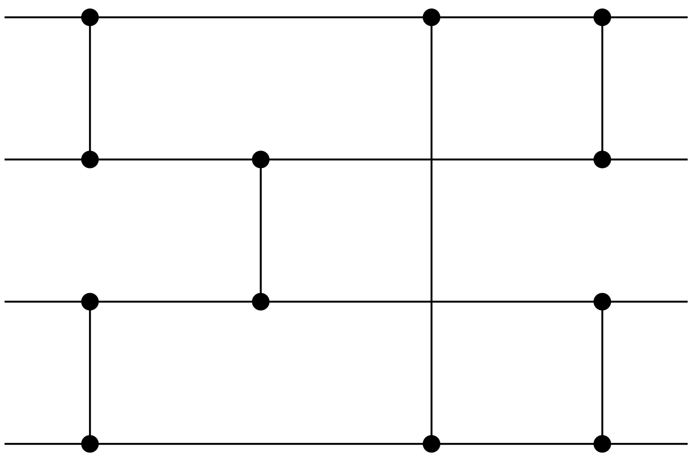


Figure 4.1: A sorting network.

Remarks:

- $< 1, 4, 6, 8, 3, 2 >$ or $< 5, 3, 2, 1, 4, 8 >$ are bitonic sequences.
- $< 9, 6, 2, 3, 5, 4 >$ or $< 7, 4, 2, 5, 9, 8 >$ are not bitonic.
- Since we restrict ourselves to 0's and 1's (Lemma 4.3), bitonic sequences have the form $0^i 1^j 0^k$ or $1^i 0^j 1^k$ for $i, j, k \geq 0$.

Algorithm 18 Half Cleaner

```
1: A half cleaner is a comparison network of depth 1, where we compare wire
   $i$ with wire $i + n/2$ for $i = 1, \ldots, n/2$ (we assume $n$ to be even).
```

Lemma 4.9. *Feeding a bitonic sequence into a half cleaner (Algorithm 18), the half cleaner cleans (makes all-0 or all-1) either the upper or the lower half of the n wires. The other half is bitonic.*

Proof. Assume that the input is of the form $0^i 1^j 0^k$ for $i, j, k \geq 0$. If the midpoint falls into the 0's, the input is already clean/bitonic and will stay so. If the midpoint falls into the 1's the half cleaner acts as Shearsort with two adjacent rows, exactly as in the proof of Theorem 4.5. The case $1^i 0^j 1^k$ is symmetric. □

Algorithm 19 Bitonic Sequence Sorter

```
1: A bitonic sequence sorter of width $n$ ($n$ being a power of 2) consists of a
   half cleaner of width $n$, and then two bitonic sequence sorters of width $n/2$
   each.
2: A bitonic sequence sorter of width 1 is empty.
```

Lemma 4.10. *A bitonic sequence sorter (Algorithm 19) of width n sorts bitonic sequences. It has depth $\log n$.*

Proof. The proof follows directly from the Algorithm 19 and Lemma 4.9. □

Remarks:

- Clearly we want to sort arbitrary and not only bitonic sequences! To do this we need one more concept, merging networks.

Algorithm 20 Merging Network

```
1: A merging network of width $n$ is a merger followed by two bitonic sequence
   sorters of width $n/2$. A merger is a depth-one network where we compare
   wire $i$ with wire $n - i + 1$, for $i = 1, \dots, n/2$.
```

Remarks:

- Note that a merging network is a bitonic sequence sorter where we replace the (first) half-cleaner by a merger.

Lemma 4.11. *A merging network (Algorithm 20) merges two sorted input sequences into one.*

Proof. We have two sorted input sequences. Essentially, a merger does to two sorted sequences what a half cleaner does to a bitonic sequence, since the lower part of the input is reversed. In other words, we can use same argument as in Theorem 4.5 and Lemma 4.9: Again, after the merger step either the upper or the lower half is clean, the other is bitonic. The bitonic sequence sorters complete sorting. □

Remarks:

- How do you sort n values when you are able to merge two sorted sequences of size $n/2$? Piece of cake, just apply the merger recursively.

Algorithm 21 Batcher's "Bitonic" Sorting Network

```
1: A batcher sorting network of width $n$ consists of two batcher sorting net-
   works of width $n/2$ followed by a merging network of width $n$. (See Figure
   4.2.)
2: A batcher sorting network of width 1 is empty.
```

Theorem 4.12. *A sorting network (Algorithm 21) sorts an arbitrary sequence of n values. It has depth $O(\log^2 n)$.*

Proof. Correctness is immediate: at recursive stage k ($k = 2, 4, 8, \dots, n$) we merge $n/(2k)$ sorted sequences into n/k sorted sequences. The depth $d(n)$ of the sorter of level n is the depth of a sorter of level $n/2$ plus the depth $m(n)$ of a merger with width n. The depth of a sorter of level 1 is 0 since the sorter is empty. Since a merger of width n has the same depth as a bitonic sequence sorter of width n, we know by Lemma 4.10 that $m(n) = \log n$. This gives a recursive formula for $d(n)$ which solves to $d(n) = \frac{1}{2}\log^2 n + \frac{1}{2}\log n$. □

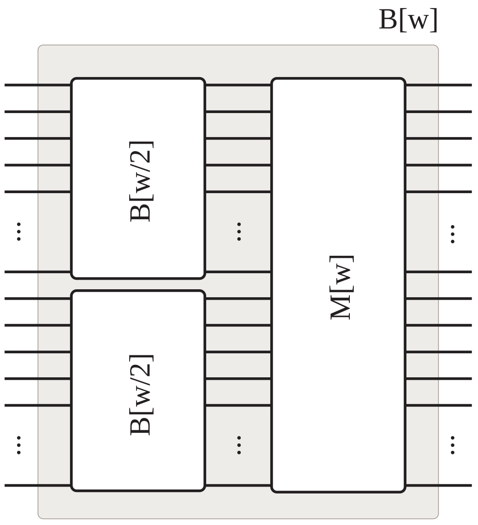


Figure 4.2: A batcher sorting network

Remarks:

- Simulating Batcher's sorting network on an ordinary sequential computer takes time $O(n \log^2 n)$. As said, there are sequential sorting algorithms that sort in asymptotically optimal time $O(n \log n)$. So a natural question is whether there is a sorting network with depth $O(\log n)$. Such a network would have some remarkable advantages over sequential asymptotically optimal sorting algorithms such as heapsort. Apart from being highly parallel, it would be completely oblivious, and as such perfectly suited for a fast hardware solution. In 1983, Ajtai, Komlos, and Szemeredi presented a celebrated $O(\log n)$ depth sorting network. (Unlike Batcher's sorting network the constant hidden in the big-O of the "AKS" sorting network is too large to be practical, however.)
- It can be shown that Batcher's sorting network and similarly others can be simulated by a Butterfly network and other hypercubic networks, see next Chapter.
- What if a sorting network is asynchronous?!? Clearly, using a synchronizer we can still sort, but it is also possible to use it for something else. Check out the next section!

4.3 Counting Networks

In this section we address distributed counting, a distributed service which can for instance be used for load balancing.

Definition 4.13 (Distributed Counting)**.** *A* distributed counter *is a variable that is common to all processors in a system and that supports an atomic* test-and-increment *operation. The operation delivers the system's counter value to the requesting processor and increments it.*

Remarks:

- A naive distributed counter stores the system's counter value with a distinguished central node. When other nodes initiate the test-and-increment operation, they send a request message to the central node and in turn receive a reply message with the current counter value. However, with a large number of nodes operating on the distributed counter, the central processor will become a bottleneck. There will be a congestion of request messages at the central processor, in other words, the system will not scale.
- Is a scalable implementation (without any kind of bottleneck) of such a distributed counter possible, or is distributed counting a problem which is inherently centralized?!?
- Distributed counting could for instance be used to implement a load balancing infrastructure, i.e. by sending the job with counter value i (modulo n) to server i (out of n possible servers).

Definition 4.14 (Balancer)**.** *A balancer is an asynchronous flip-flop which forwards messages that arrive on the left side to the wires on the right, the first to the upper, the second to the lower, the third to the upper, and so on.*

Algorithm 22 Bitonic Counting Network.

```
1: Take Batcher's bitonic sorting network of width $w$ and replace all the comparators with balancers.
2: When a node wants to count, it sends a message to an arbitrary input wire.
3: The message is then routed through the network, following the rules of the asynchronous balancers.
4: Each output wire is completed with a "mini-counter."
5: The mini-counter of wire $k$ replies the value "$k + i \cdot w$" to the initiator of the $i^{th}$ message it receives.
```

Definition 4.15 (Step Property)**.** *A sequence $y_0, y_1, \ldots, y_{w-1}$ is said to have the* step property*, if $0 \leq y_i - y_j \leq 1$, for any $i < j$.*

Remarks:

- If the output wires have the step property, then with r requests, exactly the values $1, \ldots, r$ will be assigned by the mini-counters. All we need to show is that the counting network has the step property. For that we need some additional facts...

Facts 4.16. *For a balancer, we denote the number of consumed messages on the i^{th} input wire with x_i, $i = 0, 1$. Similarly, we denote the number of sent messages on the i^{th} output wire with y_i, $i = 0, 1$. A balancer has these properties:*

(1) A balancer does not generate output-messages; that is, $x_0 + x_1 \geq y_0 + y_1$ in any state.

(2) Every incoming message is eventually forwarded. In other words, if we are in a quiescent state (no message in transit), then $x_0 + x_1 = y_0 + y_1$.

(3) The number of messages sent to the upper output wire is at most one higher than the number of messages sent to the lower output wire: in any state $y_0 = \lceil (y_0 + y_1)/2 \rceil$ *(thus* $y_1 = \lfloor (y_0 + y_1)/2 \rfloor$*).*

Facts 4.17. *If a sequence* $y_0, y_1, \ldots, y_{w-1}$ *has the step property,*

(1) then all its subsequences have the step property.

(2) then its even and odd subsequences satisfy

$$\sum_{i=0}^{w/2-1} y_{2i} = \left\lceil \frac{1}{2} \sum_{i=0}^{w-1} y_i \right\rceil \text{ and } \sum_{i=0}^{w/2-1} y_{2i+1} = \left\lfloor \frac{1}{2} \sum_{i=0}^{w-1} y_i \right\rfloor .$$

Facts 4.18. *If two sequences* $x_0, x_1, \ldots, x_{w-1}$ *and* $y_0, y_1, \ldots, y_{w-1}$ *have the step property,*

(1) and $\sum_{i=0}^{w-1} x_i = \sum_{i=0}^{w-1} y_i$*, then* $x_i = y_i$ *for* $i = 0, \ldots, w-1$*.*

(2) and $\sum_{i=0}^{w-1} x_i = \sum_{i=0}^{w-1} y_i + 1$*, then there exists a unique* j *(*$j = 0, 1, \ldots, w-1$*) such that* $x_j = y_j + 1$*, and* $x_i = y_i$ *for* $i = 0, \ldots, w-1$*,* $i \neq j$*.*

Remarks:

- That's enough to prove that a Merger preserves the step property.

Lemma 4.19. *Let* $M[w]$ *be a Merger of width* w*. In a quiescent state (no message in transit), if the inputs* $x_0, x_1, \ldots, x_{w/2-1}$ *resp.* $x_{w/2}, x_{w/2+1}, \ldots, x_{w-1}$ *have the step property, then the output* $y_0, y_1, \ldots, y_{w-1}$ *has the step property.*

Proof. By induction on the width w.

For $w = 2$: $M[2]$ is a balancer and a balancer's output has the step property (Fact 4.16.3).

For $w > 2$: Let $z_0, \ldots, z_{w/2-1}$ resp. $z'_0, \ldots, z'_{w/2-1}$ be the output of the upper respectively lower $M[w/2]$ subnetwork. Since $x_0, x_1, \ldots, x_{w/2-1}$ and $x_{w/2}, x_{w/2+1}, \ldots, x_{w-1}$ both have the step property by assumption, their even and odd subsequences also have the step property (Fact 4.17.1). By induction hypothesis, the output of both $M[w/2]$ subnetworks have the step property. Let $Z := \sum_{i=0}^{w/2-1} z_i$ and $Z' := \sum_{i=0}^{w/2-1} z'_i$. From Fact 4.17.2 we conclude that $Z = \lceil \frac{1}{2} \sum_{i=0}^{w/2-1} x_i \rceil + \lfloor \frac{1}{2} \sum_{i=w/2}^{w-1} x_i \rfloor$ and $Z' = \lfloor \frac{1}{2} \sum_{i=0}^{w/2-1} x_i \rfloor + \lceil \frac{1}{2} \sum_{i=w/2}^{w-1} x_i \rceil$. Since $\lceil a \rceil + \lfloor b \rfloor$ and $\lfloor a \rfloor + \lceil b \rceil$ differ by at most 1 we know that Z and Z' differ by at most 1.

If $Z = Z'$, Fact 4.18.1 implies that $z_i = z'_i$ for $i = 0, \ldots, w/2 - 1$. Therefore, the output of $M[w]$ is $y_i = z_{\lfloor i/2 \rfloor}$ for $i = 0, \ldots, w-1$. Since $z_0, \ldots, z_{w/2-1}$ has the step property, so does the output of $M[w]$ and the Lemma follows.

If Z and Z' differ by 1, Fact 4.18.2 implies that $z_i = z'_i$ for $i = 0, \ldots, w/2-1$, except a unique j such that z_j and z'_j differ by only 1, for $j = 0, \ldots, w/2 - 1$. Let $l := min(z_j, z'_j)$. Then, the output y_i (with $i < 2j$) is $l + 1$. The output y_i (with $i > 2j + 1$) is l. The output y_{2j} and y_{2j+1} are balanced by the final balancer resulting in $y_{2j} = l + 1$ and $y_{2j+1} = l$. Therefore $M[w]$ preserves the step property. □

A bitonic counting network is constructed to fulfill Lemma 4.19, i.e., the final output comes from a Merger whose upper and lower inputs are recursively merged. Therefore, the following Theorem follows immediately.

Theorem 4.20 (Correctness)**.** *In a quiescent state, the w output wires of a bitonic counting network of width w have the step property.*

Remarks:

- Is every sorting networks also a counting network? No. But surprisingly, the other direction is true!

Theorem 4.21 (Counting vs. Sorting)**.** *If a network is a counting network then it is also a sorting network, but not vice versa.*

Proof. There are sorting networks that are not counting networks (e.g. odd/even sort, or insertion sort). For the other direction, let C be a counting network and $I(C)$ be the isomorphic network, where every balancer is replaced by a comparator. Let $I(C)$ have an arbitrary input of 0's and 1's; that is, some of the input wires have a 0, all others have a 1. There is a message at C's i^{th} input wire if and only if $I(C)$'s i input wire is 0. Since C is a counting network, all messages are routed to the upper output wires. $I(C)$ is isomorphic to C, therefore a comparator in $I(C)$ will receive a 0 on its upper (lower) wire if and only if the corresponding balancer receives a message on its upper (lower) wire. Using an inductive argument, the 0's and 1's will be routed through $I(C)$ such that all 0's exit the network on the upper wires whereas all 1's exit the network on the lower wires. Applying Lemma 4.3 shows that $I(C)$ is a sorting network. □

Remarks:

- We claimed that the counting network is correct. However, it is only correct in a quiescent state.

Definition 4.22 (Linearizable)**.** *A system is* linearizable *if the order of the values assigned reflects the real-time order in which they were requested. More formally, if there is a pair of operations o_1, o_2, where operation o_1 terminates before operation o_2 starts, and the logical order is "o_2 before o_1", then a distributed system is not linearizable.*

Lemma 4.23 (Linearizability)**.** *The bitonic counting network is not linearizable.*

Proof. Consider the bitonic counting network with width 4 in Figure 4.3: Assume that two *inc* operations were initiated and the corresponding messages entered the network on wire 0 and 2 (both in light gray color). After having passed the second resp. the first balancer, these traversing messages "fall asleep"; In other words, both messages take unusually long time before they are received by the next balancer. Since we are in an asynchronous setting, this may be the case.

In the meantime, another *inc* operation (medium gray) is initiated and enters the network on the bottom wire. The message leaves the network on wire 2, and the *inc* operation is completed.

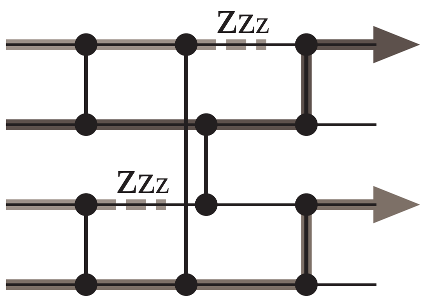


Figure 4.3: Linearizability Counter Example.

Strictly afterwards, another *inc* operation (dark gray) is initiated and enters the network on wire 1. After having passed all balancers, the message will leave the network wire 0. Finally (and not depicted in figure 4.3), the two light gray messages reach the next balancer and will eventually leave the network on wires 1 resp. 3. Because the dark gray and the medium gray operation do conflict with Definition 4.22, the bitonic counting network is not linearizable. □

Remarks:

- Note that the example in Figure 4.3 behaves correctly in the quiescent state: Finally, exactly the values $0, 1, 2, 3$ are allotted.
- It was shown that linearizability comes at a high price (the depth grows linearly with the width).

Chapter 5

Shared Memory

5.1 Introduction

In distributed computing, various different models exist. So far, the focus of the course was on loosely-coupled distributed systems such as the Internet, where nodes asynchronously communicate by exchanging messages. The “opposite” model is a tightly-coupled parallel computer where nodes access a common memory totally synchronously—in distributed computing such a system is called a Parallel Random Access Machine (PRAM).

A third major model is somehow between these two extremes, the *shared memory* model. In a shared memory system, asynchronous processes (or processors) communicate via a common memory area of shared variables or registers:

Definition 5.1 (Shared Memory)**.** *A shared memory system is a system that consists of asynchronous processes that access a common (shared) memory. A process can atomically access a register in the shared memory through a set of predefined operations. Apart from this shared memory, processes can also have some local (private) memory.*

Remarks:

- Various shared memory systems exist. A main difference is how they allow processes to access the shared memory. All systems can atomically read or write a shared register R. Most systems do allow for advanced *atomic* read-modify-write (RMW) operations, for example:
  - test-and-set(R): $t := R$; $R := 1$; return t
  - fetch-and-add(R, x): $t := R$; $R := R + x$; return t
  - compare-and-swap(R, x, y): if $R = x$ then $R := y$; return **true**; else return **false**; endif;
  - load-link/store-conditional: Load-link returns the current value of the specified register. A subsequent store-conditional to the same register will store a new value (and return **true**) only if no updates have occurred to that register since the load-link. If any updates have occurred, the store-conditional is guaranteed to fail (and return **false**), even if the value read by the load-link has since been restored.

- Maurice Herlihy suggested that the power of RMW operations can be measured with the so-called *consensus-number*: The consensus-number of a RMW operation defines whether one can solve consensus for k processes. Test-and-set for instance has consensus-number 2 (one can solve consensus with 2 processes, but not 3), whereas the consensus-number of compare-and-swap is infinite. In his 1991 paper, Maurice Herlihy proved the "universality of consensus", i.e., the power of a shared memory system is determined by the consensus-number. This insight had a remarkable theoretical and practical impact. In practice for instance, hardware designers stopped developing shared memory systems supporting weak RMW operations. Consequently, Maurice Herlihy was awarded the Dijkstra Prize in Distributed Computing in 2003.

- Many of the results derived in the message passing model have an equivalent in the shared memory model. Consensus for instance is traditionally studied in the shared memory model.

- Whereas programming a message passing system is rather tricky (in particular if fault-tolerance has to be integrated), programming a shared memory system is generally considered easier, as programmers are given access to global variables directly and do not need to worry about exchanging messages correctly. Because of this, even distributed systems which physically communicate by exchanging messages can often be programmed through a shared memory middleware, making the programmer's life easier.

- We will most likely find the general spirit of shared memory systems in upcoming multi-core architectures. As for programming style, the multi-core community seems to favor an accelerated version of shared memory, *transactional memory*.

- From a message passing perspective, the shared memory model is like a bipartite graph: One one side you have the processes (the nodes) which pretty much behave like nodes in the message passing model (asynchronous, maybe failures). On the other side you have the shared registers, which just work perfectly (no failures, no delay).

5.2 Mutual Exclusion

A classic problem in shared memory systems is mutual exclusion. We are given a number of processes which occasionally need to access the same resource. The resource may be a shared variable, or a more general object such as a data structure or a shared printer. The catch is that only one process at the time is allowed to access the resource. More formally:

Definition 5.2 (Mutual Exclusion)**.** *We are given a number of processes, each executing the following code sections:*
<Entry> → <Critical Section> → <Exit> → <Remaining Code>
A mutual exclusion algorithm consists of code for entry and exit sections, such that the following holds

- *Mutual Exclusion: At all times* at most one *process is in the critical section.*
- *No deadlock: If some process manages to get to the entry section, later* some *(possibly different) process will get to the critical section.*

Sometimes we in addition ask for

- *No lockout: If some process manages to get to the entry section, later* the same *process will get to the critical section.*
- *Unobstructed exit: No process can get stuck in the exit section.*

Using RMW primitives one can build mutual exclusion algorithms quite easily. Algorithm 23 shows an example with the test-and-set primitive.

Algorithm 23 Mutual Exclusion: Test-and-Set

```
Input: Shared register $R := 0$
<Entry>
1: repeat
2:     $r :=$ test-and-set($R$)
3: until $r = 0$
<Critical Section>
4: ...
<Exit>
5: $R := 0$
<Remainder Code>
6: ...
```

Theorem 5.3. *Algorithm 23 solves the mutual exclusion problem as in Definition 5.2.*

Proof. Mutual exclusion follows directly from the test-and-set definition: Initially R is 0. Let p_i be the i^{th} process to successfully execute the test-and-set, where successfully means that the result of the test-and-set is 0. This happens at time t_i. At time t'_i process p_i resets the shared register R to 0. Between t_i and t'_i no other process can successfully test-and-set, hence no other process can enter the critical section concurrently.

Proving no deadlock works similar: One of the processes loitering in the entry section will successfully test-and-set as soon as the process in the critical section exited.

Since the exit section only consists of a single instruction (no potential infinite loops) we have unobstructed exit. □

Remarks:

- No lockout, on the other hand, is not given by this algorithm. Even with only two processes there are asynchronous executions where always the same process wins the test-and-set.
- Algorithm 23 can be adapted to guarantee fairness (no lockout), essentially by ordering the processes in the entry section in a queue.

- A natural question is whether one can achieve mutual exclusion with only reads and writes, that is without advanced RMW operations. The answer is yes!

Our read/write mutual exclusion algorithm is for two processes p_0 and p_1 only. In the remarks we discuss how it can be extended. The general idea is that process p_i has to mark its desire to enter the critical section in a "want" register W_i by setting $W_i := 1$. Only if the other process is not interested ($W_{1-i} = 0$) access is granted. This however is too simple since we may run into a deadlock. This deadlock (and at the same time also lockout) is resolved by adding a priority variable Π. See Algorithm 24.

Algorithm 24 Mutual Exclusion: Peterson's Algorithm

```
Initialization: Shared registers $W_0, W_1, \Pi$, all initially 0.
Code for process $p_i$ , $i = \{0, 1\}$
<Entry>
1: $W_i := 1$
2: $\Pi := 1 - i$
3: repeat until $\Pi = i$ or $W_{1-i} = 0$
<Critical Section>
4: ...
<Exit>
5: $W_i := 0$
<Remainder Code>
6: ...
```

Remarks:

- Note that line 3 in Algorithm 24 represents a "spinlock" or "busy-wait", similarly to the lines 1-3 in Algorithm 23.

Theorem 5.4. *Algorithm 24 solves the mutual exclusion problem as in Definition 5.2.*

Proof. The shared variable Π elegantly grants priority to the process that passes line 2 first. If both processes are competing, only process p_Π can access the critical section because of Π. The other process $p_{1-\Pi}$ cannot access the critical section because $W_\Pi = 1$ (and $\Pi \neq 1 - \Pi$). The only other reason to access the critical section is because the other process is in the remainder code (that is, not interested). This proves mutual exclusion!

No deadlock comes directly with Π: Process p_Π gets direct access to the critical section, no matter what the other process does.

Since the exit section only consists of a single instruction (no potential infinite loops) we have unobstructed exit.

Thanks to the shared variable Π also no lockout (fairness) is achieved: If a process p_i loses against its competitor p_{1-i} in line 2, it will have to wait until the competitor resets $W_{1-i} := 0$ in the exit section. If process p_i is unlucky it will not check $W_{1-i} = 0$ early enough before process p_{1-i} sets $W_{1-i} := 1$ again in line 1. However, as soon as p_{1-i} hits line 2, process p_i gets the priority due to Π, and can enter the critical section. □

Remarks:

- Extending Peterson's Algorithm to more than 2 processes can be done by a tournament tree, like in tennis. With n processes every process needs to win $\log n$ matches before it can enter the critical section. More precisely, each process starts at the bottom level of a binary tree, and proceeds to the parent level if winning. Once winning the root of the tree it can enter the critical section. Thanks to the priority variables Π at each node of the binary tree, we inherit all the properties of Definition 5.2.

5.3 Store & Collect

5.3.1 Problem Definition

In this section, we will look at a second shared memory problem that has an elegant solution. Informally, the problem can be stated as follows. There are n processes $p_1, \ldots, p_n$. Every process p_i has a read/write register R_i in the shared memory where it can *store* some information that is destined for the other processes. Further, there is an operation by which a process can *collect* (i.e., read) the values of all the processes that stored some value in their register.

We say that an operation *op1* precedes an operation *op2* iff *op1* terminates before *op2* starts. An operation *op2* follows an operation *op1* iff *op1* precedes *op2*.

Definition 5.5 (Collect). *There are two operations: A* STORE(*val*) *by process p_i sets val to be the latest value of its register R_i. A* COLLECT *operation returns a* view*, a partial function V from the set of processes to a set of values, where $V(p_i)$ is the latest value stored by p_i, for each process p_i. For a* COLLECT *operation* cop*, the following validity properties must hold for every process p_i:*

- *If $V(p_i) = \perp$, then no* STORE *operation by p_i precedes* cop.
- *If $V(p_i) = v \neq \perp$, then v is the value of a* STORE *operation* sop *of p_i that does not follow* cop*, and there is no* STORE *operation by p_i that follows* sop *and precedes* cop.

Hence, a COLLECT operation *cop* should not read from the future or miss a preceding STORE operation *sop*.

We assume that the read/write register R_i of every process p_i is initialized to $\perp$. We define the step complexity of an operation *op* to be the number of accesses to registers in the shared memory. There is a trivial solution to the *collect* problem as shown by Algorithm 25.

Algorithm 25 Collect: Simple (Non-Adaptive) Solution

```
Operation STORE($val$) (by process $p_i$) :
1: $R_i := val$
Operation COLLECT:
2: for $i := 1$ to $n$ do
3:     $V(p_i) := R_i$                         // read register $R_i$
4: end for
```

Remarks:

- Algorithm 25 clearly works. The step complexity of every STORE operation is 1, the step complexity of a COLLECT operation is n.
- At first sight, the step complexities of Algorithm 25 seem optimal. Because there are n processes, there clearly are cases in which a COLLECT operation needs to read all n registers. However, there are also scenarios in which the step complexity of the COLLECT operation seems very costly. Assume that there are only two processes p_i and p_j that have stored a value in their registers R_i and R_j. In this case, a COLLECT in principle only needs to read the registers R_i and R_j and can ignore all the other registers.
- Assume that up to a certain time t, $k \leq n$ processes have finished or started at least one operation. We call an operation *op* at time t *adaptive* to contention if the step complexity of *op* only depends on k and is independent of n.
- In the following, we will see how to implement adaptive versions of STORE and COLLECT.

5.3.2 Splitters

Algorithm 26 Splitter Code

```
Shared Registers: $X : \{\perp\} \cup \{1, \ldots, n\}$; $Y$ : boolean
Initialization: $X := \perp$; $Y :=$ false

Splitter access by process $p_i$:
$X := i$;
if $Y$ then
   return right
else
   $Y :=$ true
   if $X = i$ then
      return stop
   else
      return left
   end if
end if
```

To obtain adaptive collect algorithms, we need a synchronization primitive, called a *splitter*.

Definition 5.6 (Splitter)**.** *A splitter is a synchronization primitive with the following characteristic. A process entering a splitter exits with either* ***stop****,* ***left****, or* ***right****. If k processes enter a splitter, at most one process exits with* ***stop*** *and at most $k-1$ processes exit with* ***left*** *and* ***right****, respectively.*

Hence, it is guaranteed that if a single process enters the splitter, then it obtains **stop**, and if two or more processes enter the splitter, then there is at most one process obtaining **stop** and there are two processes that obtain

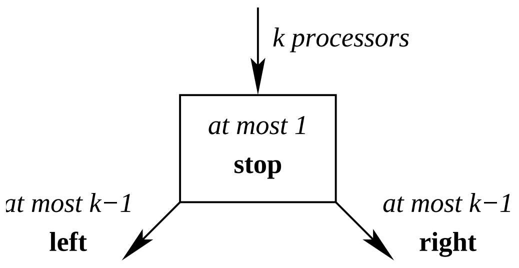


Figure 5.1: A Splitter

different values (i.e., either there is exactly one **stop** or there is at least one **left** and at least one **right**). For an illustration, see Figure 5.1. The code implementing a splitter is given by Algorithm 26.

Lemma 5.7. *Algorithm 26 correctly implements a splitter.*

Proof. Assume that k processes enter the splitter. Because the first process that checks whether Y = **true** in line 2 will find that Y = **false**, not all processes return **right**. Next, assume that i is the last process that sets $X := i$. If i does not return **right**, it will find $X = i$ in line 6 and therefore return **stop**. Hence, there is always a process that does not return **left**. It remains to show that at most 1 process returns **stop**. For the sake of contradiction, assume p_i and p_j are two processes that return **stop** and assume that p_i sets $X := i$ before p_j sets $X := j$. Both processes need to check whether Y is **true** before one of them sets $Y :=$ **true**. Hence, they both complete the assignment in line 1 before the first one of them checks the value of X in line 6. Hence, by the time p_i arrives at line 6, $X \neq i$ (p_j and maybe some other processes have overwritten X by then). Therefore, p_i does not return **stop** and we get a contradiction to the assumption that both p_i and p_j return **stop**. □

5.3.3 Binary Splitter Tree

Assume that we are given $2^n - 1$ splitters and that for every splitter S, there is an additional shared variable $Z_S : \{\perp\} \cup \{1, \dots, n\}$ that is initialized to $\perp$ and an additional shared variable M_S : **boolean** that is initialized to **false**. We call a splitter S marked if M_S = **true**. The $2^n - 1$ splitters are arranged in a complete binary tree of height $n - 1$. Let $S(v)$ be the splitter associated with a node v of the binary tree. The STORE and COLLECT operations are given by Algorithm 27.

Theorem 5.8. *Algorithm 27 correctly implements* STORE *and* COLLECT. *Let k be the number of participating processes. The step complexity of the first* STORE *of a process p_i is $O(k)$, the step complexity of every additional* STORE *of p_i is $O(1)$, and the step complexity of* COLLECT *is $O(k)$.*

Proof. Because at most one process can stop at a splitter, it is sufficient to show that every process stops at some splitter at depth at most $k - 1 \leq n - 1$ when invoking the first STORE operation to prove correctness. We prove that at most $k - i$ processes enter a subtree at depth i (i.e., a subtree where the root has distance i to the root of the whole tree). By definition of k, the number of

Algorithm 27 Adaptive Collect: Binary Tree Algorithm

```
Operation STORE($val$) (by process $p_i$) :
1: $R_i := val$
2: if first STORE operation by $p_i$ then
3:   $v :=$ root node of binary tree
4:   $\alpha :=$ result of entering splitter $S(v)$;
5:   $M_{S(v)} :=$ true
6:   while $\alpha \neq$ stop do
7:     if $\alpha =$ left then
8:       $v :=$ left child of $v$
9:     else
10:      $v :=$ right child of $v$
11:    end if
12:    $\alpha :=$ result of entering splitter $S(v)$;
13:    $M_{S(v)} :=$ true
14:  end while
15:  $Z_{S(v)} := i$
16: end if

Operation COLLECT:
Traverse marked part of binary tree:
17: for all marked splitters $S$ do
18:   if $Z_S \neq \perp$ then
19:     $i := Z_S$; $V(p_i) := R_i$                // read value of process $p_i$
20:   end if
21: end for                              // $V(p_i) = \perp$ for all other processes
```

processes entering the splitter at depth 0 (i.e., at the root of the binary tree) is k. For $i > 1$, the claim follows by induction because of the at most $k - i$ processes entering the splitter at the root of a depth i subtree, at most $k - i - 1$ obtain **left** and **right**, respectively. Hence, at the latest when reaching depth $k - 1$, a process is the only process entering a splitter and thus obtains **stop**. It thus also follows that the step complexity of the first invocation of STORE is $O(k)$.

To show that the step complexity of COLLECT is $O(k)$, we first observe that the marked nodes of the binary tree are connected, and therefore can be traversed by only reading the variables M_S associated to them and their neighbors. Hence, showing that at most $2k - 1$ nodes of the binary tree are marked is sufficient. Let x_k be the maximum number of marked nodes in a tree, where k processes access the root. We claim that $x_k \leq 2k - 1$, which is true for $k = 1$ because a single process entering a splitter will always compute **stop**. Now assume the inequality holds for $1, \ldots, k - 1$. Not all k processes may exit the splitter with **left** (or **right**), i.e., $k_l \leq k - 1$ processes will turn left and $k_r \leq \min\{k - k_l, k - 1\}$ turn right. The left and right children of the root are the roots of their subtrees, hence the induction hypothesis yields

$$x_k \leq x_{k_l} + x_{k_r} + 1 \leq (2k_l - 1) + (2k_r - 1) + 1 \leq 2k - 1,$$

concluding induction and proof. □

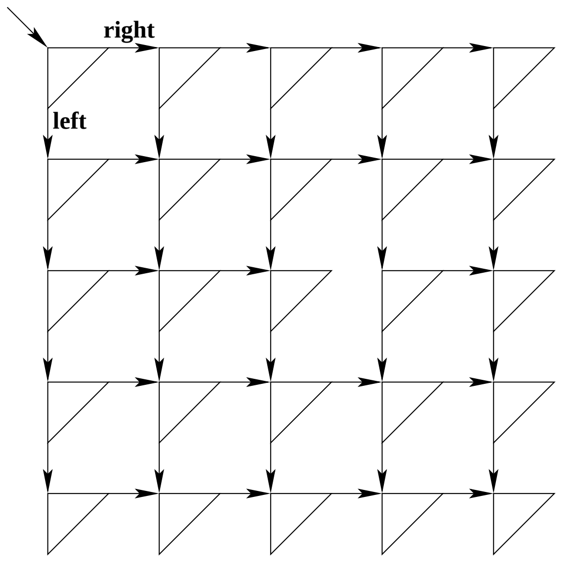


Figure 5.2: 5×5 Splitter Matrix

Remarks:

- The step complexities of Algorithm 27 are very good. Clearly, the step complexity of the COLLECT operation is asymptotically optimal. In order for the algorithm to work, we however need to allocate the memory for the complete binary tree of depth $n-1$. The space complexity of Algorithm 27 therefore is exponential in n. We will next see how to obtain a polynomial space complexity at the cost of a worse COLLECT step complexity.

5.3.4 Splitter Matrix

Instead of arranging splitters in a binary tree, we arrange n^2 splitters in an $n \times n$ matrix as shown in Figure 5.2. The algorithm is analogous to Algorithm 27. The matrix is entered at the top left. If a process receives **left**, it next visits the splitter in the next row of the same column. If a process receives **right**, it next visits the splitter in the next column of the same row. Clearly, the space complexity of this algorithm is $O(n^2)$. The following theorem gives bounds on the step complexities of STORE and COLLECT.

Theorem 5.9. *Let k be the number of participating processes. The step complexity of the first* STORE *of a process p_i is $O(k)$, the step complexity of every additional* STORE *of p_i is $O(1)$, and the step complexity of* COLLECT *is $O(k^2)$.*

Proof. Let the top row be row 0 and the left-most column be column 0. Let x_i be the number of processes entering a splitter in row i. By induction on i, we show that $x_i \leq k - i$. Clearly, $x_0 \leq k$. Let us therefore consider the case $i > 0$. Let j be the largest column such that at least one process visits the splitter in row $i-1$ and column j. By the properties of splitters, not all processes entering the splitter in row $i-1$ and column j obtain **left**. Therefore, not all processes entering a splitter in row $i-1$ move on to row i. Because at least one processes stays in every row, we get that $x_i \leq k - i$. Similarly, the number of processes entering column j is at most $k - j$. Hence, every process stops at the latest in row $k - 1$ and column $k - 1$ and the number of marked splitters is at most k^2. Thus, the step complexity of COLLECT is at most $O(k^2)$. Because the longest path in the splitter matrix is $2k$, the step complexity of STORE is $O(k)$. $\square$

Remarks:

- With a slightly more complicated argument, it is possible to show that the number of processes entering the splitter in row i and column j is at most $k - i - j$. Hence, it suffices to only allocate the upper left half (including the diagonal) of the $n \times n$ matrix of splitters.
- The binary tree algorithm can be made space efficient by using a randomized version of a splitter. Whenever returning left or right, a randomized splitter returns left or right with probability $1/2$. With high probability, it then suffices to allocate a binary tree of depth $O(\log n)$.
- Recently, it has been shown that with a considerably more complicated deterministic algorithm, it is possible to achieve $O(k)$ step complexity and $O(n^2)$ space complexity.

Chapter 6

Shared Objects

6.1 Introduction

Assume that there is a common resource (e.g. a common variable or data structure), which different nodes in a network need to access from time to time. If the nodes are allowed to change the common object when accessing it, we need to guarantee that no two nodes have access to the object at the same time. In order to achieve this mutual exclusion, we need protocols that allow the nodes of a network to store and manage access to such a shared object. A simple and obvious solution is to store the shared object at a central location (see Algorithm 28).

Algorithm 28 Shared Object: Centralized Solution

```
Initialization: Shared object stored at root node $r$ of a spanning tree of the
    network graph (i.e., each node knows its parent in the spanning tree).
Accessing Object: (by node $v$)
1: $v$ sends request up the tree
2: request processed by root $r$ (atomically)
3: result sent down the tree to node $v$
```

Remarks:

- Algorithm 28 works. Instead of a spanning tree, one can use routing.
- It is however not very efficient. Assume that the object is accessed by a single node v repeatedly. Then we get a high message/time complexity. Instead v could store the object, or at least cache it. But then, in case another node w accesses the object, we might run into consistency problems.
- Alternative idea: The accessing node should become the new master of the object. The shared object then becomes mobile. There exist several variants of this idea. The simplest version is a home-based solution like in Mobile IP (see Algorithm 29).

```
Algorithm 29 Shared Object: Home-Based Solution
Initialization: An object has a home base (a node) that is known to every
    node. All requests (accesses to the shared object) are routed through the
    home base.
Accessing Object: (by node $v$)
 1: $v$ acquires a lock at the home base, receives object.
```

Remarks:

- Home-based solutions suffer from the triangular routing problem. If two close-by nodes access the object on a rotating basis, all the traffic is routed through the potentially far away home-base.

6.2 Arrow and Friends

We will now look at a protocol (called the Arrow algorithm) that always moves the shared object to the node currently accessing it without creating the triangular routing problem of home-based solutions. The protocol runs on a precomputed spanning tree. Assume that the spanning tree is rooted at the current position of the shared object. When a node u wants to access the shared object, it sends out a *find* request towards the current position of the object. While searching for the object, the edges of the spanning tree are redirected such that in the end, the spanning tree is rooted at u (i.e., the new holder of the object). The details of the algorithm are given by Algorithm 30. For simplicity, we assume that a node u only starts a find request if u is not currently the holder of the shared object and if u has finished all previous find requests (i.e., it is not currently waiting to receive the object).

Remarks:

- The parent pointers in Algorithm 30 are only needed for the find operation. Sending the variable to u in line 13 or to w.successor in line 23 is done using routing (on the spanning tree or on the underlying network).
- When we draw the parent pointers as arrows, in a quiescent moment (where no "find" is in motion), the arrows all point towards the node currently holding the variable (i.e., the tree is rooted at the node holding the variable)
- What is really great about the Arrow algorithm is that it works in a completely asynchronous and concurrent setting (i.e., there can be many find requests at the same time).

Theorem 6.1. *(Arrow, Analysis) In an asynchronous, steady-state, and concurrent setting, a "find" operation terminates with message and time complexity D, where D is the diameter of the spanning tree.*

Algorithm 30 Shared Object: Arrow Algorithm

Initialization: As for Algorithm 28, we are given a rooted spanning tree. Each node has a pointer to its parent, the root r is its own parent. The variable is initially stored at r. For all nodes v, v.successor := **null**, v.wait := **false**.

Start Find Request at Node u:

```
do atomically
  $u$ sends “find by $u$” message to parent node
  $u$.parent := $u$
  $u$.wait := true
end do
```

Upon w Receiving “Find by u” Message from Node v:

```
do atomically
  if $w$.parent $\neq$ $w$ then
    $w$ sends “find by $u$” message to parent
    $w$.parent := $v$
  else
    $w$.parent := $v$
    if not $w$.wait then
      send variable to $u$      // w holds var. but does not need it any more
    else
      $w$.successor := $u$      // w will send variable to u a.s.a.p.
    end if
  end if
end do
```

Upon w Receiving Shared Object:

```
perform operation on shared object
do atomically
  $w$.wait := false
  if $w$.successor $\neq$ null then
    send variable to $w$.successor
    $w$.successor := null
  end if
end do
```

Before proving Theorem 6.1, we prove the following lemma.

Lemma 6.2. *An edge $\{u, v\}$ of the spanning tree is in one of four states:*

1.) Pointer from u to v (no message on the edge, no pointer from v to u)
2.) Message on the move from u to v (no pointer along the edge)
3.) Pointer from v to u (no message on the edge, no pointer from u to v)
4.) Message on the move from v to u (no pointer along the edge)

Proof. W.l.o.g., assume that initially the edge $\{u, v\}$ is in state 1. With a message arrival at u (or if u starts a "find by u" request, the edge goes to state 2. When the message is received at v, v directs its pointer to u and we are therefore in state 3. A new message at v (or a new request initiated by v) then brings the edge back to state 1. □

Proof of Theorem 6.1. Since the "find" message will only travel on a static tree, it suffices to show that it will not traverse an edge twice. Suppose for the sake of contradiction that there is a first "find" message f that traverses an edge $e = \{u, v\}$ for the second time and assume that e is the first edge that is traversed twice by f. The first time, f traverses e. Assume that e is first traversed from u to v. Since we are on a tree, the second time, e must be traversed from v to u. Because e is the first edge to be traversed twice, f must re-visit e before visiting any other edges. Right before f reaches v, the edge e is in state 2 (f is on the move) and in state 3 (it will immediately return with the pointer from v to u). This is a contradiction to Lemma 6.2. □

Remarks:

- Finding a good tree is an interesting problem. We would like to have a tree with low stretch, low diameter, low degree, etc.
- It seems that the Arrow algorithm works especially well when lots of "find" operations are initiated concurrently. Most of them will find a "close-by" node, thus having low message/time complexity. For the sake of simplicity we analyze a synchronous system.

Theorem 6.3. *(Arrow, Concurrent Analysis) Let the system be synchronous. Initially, the system is in a quiescent state. At time* 0, *a set* S *of nodes initiates a "find" operation. The message complexity of all "find" operations is* $O(\log |S| \cdot m^*)$ *where* m^* *is the message complexity of an optimal (with global knowledge) algorithm on the tree.*

Proof Sketch. Let d be the minimum distance of any node in S to the root. There will be a node u_1 at distance d from the root that reaches the root in d time steps, turning all the arrows on the path to the root towards u_1. A node u_2 that finds (is queued behind) u_1 cannot distinguish the system from a system where there was no request u_1, and instead the root was initially located at u_1. The message cost of u_2 is consequentially the distance between u_1 and u_2 on the spanning tree. By induction the total message complexity is exactly as if a collector starts at the root and then "greedily" collects tokens located at the nodes in S (greedily in the sense that the collector always goes towards the closest token). Greedy collecting the tokens is not a good strategy in general because it will traverse the same edge more than twice in the worst case. An asymptotically optimal algorithm can also be translated into a depth-first-search collecting paradigm, traversing each edge at most twice. In another area of computer science, we would call the Arrow algorithm a nearest-neighbor TSP heuristic (without returning to the start/root though), and the optimal algorithm TSP-optimal. It was shown that nearest-neighbor has a logarithmic overhead, which concludes the proof. □

Remarks:

- An average request set S on a not-too-bad tree gives usually a much better bound. However, there is an almost tight $\log |S| / \log \log |S|$ worst-case example.
- It was recently shown that Arrow can do as good in a dynamic setting (where nodes are allowed to initiate requests at any time). In particular the message complexity of the dynamic analysis can be shown to have a $\log D$ overhead only, where D is the diameter of the spanning tree (note that for logarithmic trees, the overhead becomes $\log \log n$).
- What if the spanning tree is a star? Then with Theorem 6.1, each find will terminate in 2 steps! Since also an optimal algorithm has message cost 1, the algorithm is 2-competitive. . . ? Yes, but because of its high degree the star center experiences contention. . . It can be shown that the contention overhead is at most proportional to the largest degree Δ of the spanning tree.
- Thought experiment: Assume a balanced binary spanning tree—by Theorem 6.1, the message complexity per operation is $\log n$. Because a binary tree has maximum degree 3, the time per operation therefore is at most $3 \log n$.
- There are better and worse choices for the spanning tree. The stretch of an edge $\{u, v\}$ is defined as distance between u and v in a spanning tree. The maximum stretch of a spanning tree is the maximum stretch over all edges. A few years ago, it was shown how to construct spanning trees that are $O(\log n)$-stretch-competitive.

What if most nodes just want to read the shared object? Then it does not make sense to acquire a lock every time. Instead we can use caching (see Algorithm 31).

Theorem 6.4. *Algorithm 31 is correct. More surprisingly, the message complexity is 3-competitive (at most a factor 3 worse than the optimum).*

Proof. Since the accesses do not overlap by definition, it suffices to show that between two writes, we are 3-competitive. The sequence of accessing nodes is w_0, r_1, r_2, $\ldots$, r_k, w_1. After w_0, the object is stored at w_0 and not cached anywhere else. All reads cost twice the smallest subtree T spanning the write w_0 and all the reads since each read only goes to the first copy. The write w_1 costs T plus the path P from w_1 to T. Since any data management scheme must use an edge in T and P at least once, and our algorithm uses edges in T at most 3 times (and in P at most once), the theorem follows. □

Algorithm 31 Shared Object: Read/Write Caching

- Nodes can either read or write the shared object. For simplicity we first assume that reads or writes do not overlap in time (access to the object is sequential).
- Nodes store three items: a parent pointer pointing to one of the neighbors (as with Arrow), and a cache bit for each edge, plus (potentially) a copy of the object.
- Initially the object is stored at a single node u; all the parent pointers point towards u, all the cache bits are false.
- When initiating a read, a message follows the arrows (this time: without inverting them!) until it reaches a cached version of the object. Then a copy of the object is cached along the path back to the initiating node, and the cache bits on the visited edges are set to true.
- A write at u writes the new value locally (at node u), then searches (follow the parent pointers and reverse them towards u) a first node with a copy. Delete the copy and follow (in parallel, by flooding) all edge that have the cache flag set. Point the parent pointer towards u, and remove the cache flags.

Remarks:

- Concurrent reads are not a problem, also multiple concurrent reads and one write work just fine.
- What about concurrent writes? To achieve consistency writes need to invalidate the caches before writing their value. It is claimed that the strategy then becomes 4-competitive.
- Is the algorithm also time competitive? Well, not really: The optimal algorithm that we compare to is usually offline. This means it knows the whole access sequence in advance. It can then cache the object before the request even appears!
- Algorithms on trees are often simpler, but have the disadvantage that they introduce the extra stretch factor. In a ring, for example, any tree has stretch $n - 1$; so there is always a bad request pattern.

Algorithm 32 Shared Object: Pointer Forwarding

```
Initialization: Object is stored at root $r$ of a precomputed spanning tree $T$ (as
    in the Arrow algorithm, each node has a parent pointer pointing towards
    the object).
Accessing Object: (by node $u$)
1: follow parent pointers to current root $r$ of $T$
2: send object from $r$ to $u$
3: $r$.parent := $u$; $u$.parent := $u$;                // $u$ is the new root
```

Algorithm 33 Shared Object: Ivy

```
Initialization: Object is stored at root $r$ of a precomputed spanning tree $T$
    (as before, each node has a parent pointer pointing towards the object). For
    simplicity, we assume that accesses to the object are sequential.
Start Find Request at Node $u$:
1: $u$ sends "find by $u$" message to parent node
2: $u$.parent := $u$
Upon $v$ receiving "Find by $u$" Message:
3: if $v$.parent = $v$ then
4:     send object to $u$
5: else
6:     send "find by $u$" message to $v$.parent
7: end if
8: $v$.parent := $u$                          // $u$ will become the new root
```

6.3 Ivy and Friends

In the following we study algorithms that do not restrict communication to a tree. Of particular interest is the special case of a complete graph (clique). A simple solution for this case is given by Algorithm 32.

Remarks:

- If the graph is not complete, routing can be used to find the root.
- Assume that the nodes line up in a linked list. If we always choose the first node of the linked list to acquire the object, we have message/time complexity n. The new topology is again a linearly linked list. Pointer forwarding is therefore bad in a worst-case.
- If edges are not FIFO, it can even happen that the number of steps is unbounded for a node having bad luck. An algorithm with such a property is named "not fair," or "not wait-free." (Example: Initially we have the list $4 \rightarrow 3 \rightarrow 2 \rightarrow 1$; 4 starts a find; when the message of 4 passes 3, 3 itself starts a find. The message of 3 may arrive at 2 and then 1 earlier, thus the new end of the list is $2 \rightarrow 1 \rightarrow 3$; once the message of 4 passes 2, the game re-starts.)

There seems to be a natural improvement of the pointer forwarding idea. Instead of simply redirecting the parent pointer from the old root to the new root, we can redirect all the parent pointers of the nodes on the path visited

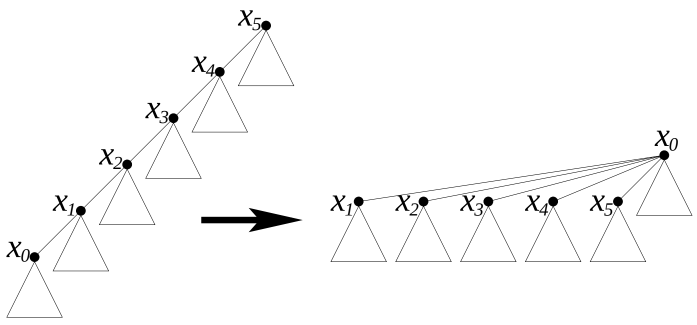


Figure 6.1: Reversal of the path $x_0, x_1, x_2, x_3, x_4, x_5$.

during a find message to the new root. The details are given by Algorithm 33. Figure 6.1 shows how the pointer redirecting affects a given tree (the right tree results from a find request started at node x_0 on the left tree).

Remarks:

- Also with Algorithm 33, we might have a bad linked list situation. However, if the start of the list acquires the object, the linked list turns into a star. As the following theorem will show, the search paths are not long on average. Since paths sometimes can be bad, we will need amortized analysis.

Theorem 6.5. *If the initial tree is a star, a find request of Algorithm 33 needs at most* $\log n$ *steps on average, where* n *is the number of processors.*

Proof. All logarithms in the following proof are to base 2. We assume that accesses to the shared object are sequential. We use a potential function argument. Let $s(u)$ be the size of the subtree rooted at node u (the number of nodes in the subtree including u itself). We define the potential Φ of the whole tree T as (V is the set of all nodes)

$$\Phi(T) = \sum_{u \in V} \frac{\log s(u)}{2}.$$

Assume that the path traversed by the i^{th} operation has length k_i, i.e., the i^{th} operation redirects k_i pointers to the new root. Clearly, the number of steps of the i^{th} operation is proportional to k_i. We are interested in the cost of m consecutive operations, $\sum_{i=1}^{m} k_i$.

Let T_0 be the initial tree and let T_i be the tree after the i^{th} operation. Further, let $a_i = k_i - \Phi(T_{i-1}) + \Phi(T_i)$ be the *amortized cost* of the i^{th} operation. We have

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} \big(k_i - \Phi(T_{i-1}) + \Phi(T_i)\big) = \sum_{i=1}^{m} k_i - \Phi(T_0) + \Phi(T_m).$$

For any tree T, we have $\Phi(T) \geq \log(n)/2$. Because we assume that T_0 is a star, we also have $\Phi(T_0) = \log(n)/2$. We therefore get that

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} k_i.$$

Hence, it suffices to upper bound the amortized cost of every operation. We thus analyze the amortized cost a_i of the i^{th} operation. Let $x_0, x_1, x_2, \ldots, x_{k_i}$ be the path that is reversed by the operation. Further for $0 \leq j \leq k_i$, let s_j be the size of the subtree rooted at x_j before the reversal. The size of the subtree rooted at x_0 after the reversal is s_{k_i} and the size of the one rooted at x_j after the reversal, for $1 \leq j \leq k_i$, is $s_j - s_{j-1}$ (see Figure 6.1). For all other nodes, the sizes of their subtrees are the same, therefore the corresponding terms cancel out in the ammortized cost a_i. We can thus write a_i as

$$\begin{aligned} a_i &= k_i - \left(\sum_{j=0}^{k_i} \frac{1}{2} \log s_j\right) + \left(\frac{1}{2} \log s_{k_i} + \sum_{j=1}^{k_i} \frac{1}{2} \log(s_j - s_{j-1})\right) \\ &= k_i + \frac{1}{2} \cdot \sum_{j=0}^{k_i - 1} \big(\log(s_{j+1} - s_j) - \log s_j\big) \\ &= k_i + \frac{1}{2} \cdot \sum_{j=0}^{k_i - 1} \log\left(\frac{s_{j+1} - s_j}{s_j}\right). \end{aligned}$$

For $0 \leq j \leq k_i - 1$, let $\alpha_j = s_{j+1}/s_j$. Note that $s_{j+1} > s_j$ and thus that $\alpha_j > 1$. Further note, that $(s_{j+1} - s_j)/s_j = \alpha_j - 1$. We therefore have that

$$\begin{aligned} a_i &= k_i + \frac{1}{2} \cdot \sum_{j=0}^{k_i - 1} \log(\alpha_j - 1) \\ &= \sum_{j=0}^{k_i - 1} \left(1 + \frac{1}{2} \log(\alpha_j - 1)\right). \end{aligned}$$

For $\alpha > 1$, it can be shown that $1 + \log(\alpha - 1)/2 \leq \log \alpha$ (see Lemma 6.6). From this inequality, we obtain

$$\begin{aligned} a_i &\leq \sum_{j=0}^{k_i - 1} \log \alpha_j = \sum_{j=0}^{k_i - 1} \log \frac{s_{j+1}}{s_j} = \sum_{j=0}^{k_i - 1} (\log s_{j+1} - \log s_j) \\ &= \log s_{k_i} - \log s_0 \leq \log n, \end{aligned}$$

because $s_{k_i} = n$ and $s_0 \geq 1$. This concludes the proof. □

Lemma 6.6. *For $\alpha > 1$, $1 + \log(\alpha - 1)/2 \leq \log \alpha$.*

Proof. The claim can be verified by the following chain of reasoning:

$$\begin{aligned} 0 &\leq (\alpha - 2)^2 \\ 0 &\leq \alpha^2 - 4\alpha + 4 \\ 4(\alpha - 1) &\leq \alpha^2 \\ \log_2\big(4(\alpha - 1)\big) &\leq \log_2\left(\alpha^2\right) \\ 2 + \log_2(\alpha - 1) &\leq 2 \log_2 \alpha \\ 1 + \frac{1}{2} \log_2(\alpha - 1) &\leq \log_2 \alpha. \end{aligned}$$

□

Remarks:

- Systems guys (the algorithm is called Ivy because it was used in a system with the same name) have some fancy heuristics to improve performance even more: For example, the root every now and then broadcasts its name such that paths will be shortened.

- What about concurrent requests? It works with the same argument as in Arrow. Also for Ivy an argument including congestion is missing (and more pressing, since the dynamic topology of a tree cannot be chosen to have low degree and thus low congestion as in Arrow).

- Sometimes the type of accesses allows that several accesses can be combined into one to reduce congestion higher up the tree. Let the tree in Algorithm 28 be a balanced binary tree. If the access to a shared variable for example is "add value x to the shared variable," two or more accesses that accidentally meet at a node can be combined into one. Clearly accidental meeting is rare in an asynchronous model. We might be able to use synchronizers (or maybe some other timing tricks) to help meeting a little bit.

Chapter 7

Dynamic Networks

Many large-scale distributed systems and networks are dynamic. In some networks, e.g., peer-to-peer, nodes participate only for a short period of time, and the topology can change at a high rate. In wireless ad-hoc networks, nodes are mobile and move around. In this chapter, we will study how to solve some basic tasks if the network is dynamic. Under what conditions is it possible to compute an accurate estimate of the size or some other property of the system? How efficiently can information be disseminated reliably in the network? To what extent does stability in the communication graph help solve these problems?

There are various reasons why networks can change over time and as a consequence, there also is a wide range of possible models for dynamic networks. Nodes might join or leave a distributed system. Some components or communication links may fail in different ways. Especially if the network devices are mobile, the connectivity between them can change. Dynamic changes can occur constantly or they might be infrequent enough so that the system can adapt to each change individually.

We will look at a synchronous dynamic network model in which the graph can change from round to round in a worst-case manner. To simplify things (and to make the problems we study well-defined), we assume that the set of nodes in the network is fixed and does not change. However, we will make almost no assumptions how the set of edges changes over time. We require some guarantees about the connectivity, apart from this, in each round, the communication graph is chosen in a worst-case manner by an adversary.

7.1 Synchronous Edge-Dynamic Networks

We model a synchronous dynamic network by a dynamic graph $G = (V, E)$, where V is a static set of nodes, and $E : \mathbb{N}_0 \to \binom{V}{2}$ is a function mapping a round number $r \in \mathbb{N}_0$ to a set of undirected edges $E(r)$. Here $\binom{V}{2} := \{\{u, v\} \mid u, v \in V\}$ is the set of all possible undirected edges over V.

Definition 7.1 (T-Interval Connectivity)**.** *A dynamic graph $G = (V, E)$ is said to be* T-interval connected *for $T \in \mathbb{N}$ if for all $r \in \mathbb{N}$, the static graph $G_{r,T} := \left(V, \bigcap_{i=r}^{r+T-1} E(r)\right)$ is connected. If G is 1-interval connected we say that G is* always connected.

For simplicity, we restrict to deterministic algorithms. Nodes communicate with each other using *anonymous broadcast*. At the beginning of round r, each node u decides what message to broadcast based on its internal state; at the same time (and independently), the adversary chooses a set $E(r)$ of edges for the round. As in standard synchronous message passing, all nodes v for which $\{u, v\} \in E(r)$ receive the message broadcast by node u in round r and each node can perform arbitrary local computations upon receiving the messages from its neighbors. We assume that all nodes in the network have a unique identifier (ID). In most cases, we will assume that messages are restricted to $O(\log n)$ bits. In these cases, we assume that node IDs can be represented using $O(\log n)$ bits, so that a constant number of node IDs and some additional information can be transmitted in a single message. We refer to the special case where all nodes are woken up at once as *synchronous start* and to the general case as *asynchronous start*.

We assume that each node in the network starts an execution of the protocol in an initial state which contains its own ID and its input. Additionally, nodes know nothing about the network, and initially cannot distinguish it from any other network.

7.2 Problem Definitions

In the context of this chapter, we study the following problems.

Counting. An algorithm is said to solve the counting problem if whenever it is executed in a dynamic graph comprising n nodes, all nodes eventually terminate and output n.

***k*-verification.** Closely related to counting, the k-verification problem requires nodes to determine whether or not $n \leq k$. All nodes begin with k as their input, and must eventually terminate and output "yes" or "no". Nodes must output "yes" if and only if there are at most k nodes in the network.

***k*-token dissemination.** An instance of k-token dissemination is a pair (V, I), where $I : V \to \mathcal{P}(\mathcal{T})$ assigns a set of tokens from some domain $\mathcal{T}$ to each node, and $|\bigcup_{u \in V} I(v)| = k$. An algorithm solves k-token dissemination if for all instances (V, I), when the algorithm is executed in any dynamic graph $G = (V, E)$, all nodes eventually terminate and output $\bigcup_{u \in V} I(u)$. We assume that each token in the nodes' input is represented using $O(\log n)$ bits. Nodes may or may not know k, depending on the context. Of particular interest is *all-to-all token dissemination*, a special case where $k = n$ and each node initially knows exactly one token, i.e., $|I(u)| = 1$ for all nodes u.

***k*-committee election.** As an useful step towards solving counting and token dissemination, we consider a problem called k-committee election. In this problem, nodes must partition themselves into sets, called *committees*, such that

a) the size of each committee is at most k and

b) if $k \geq n$, then there is just one committee containing all nodes.

Each committee has a unique committee ID, and the goal is for all nodes to eventually terminate and output a committee ID such that the two conditions are satisfied.

7.3 Basic Information Dissemination

To start, let us study how a single piece of information is propagated through a dynamic network. We assume that we have a dynamic network graph G with n nodes such that G is always connected (G is 1-interval connected as defined in Definition 7.1). Further assume that there is a single piece of information (token), which is initially known by a single node.

Theorem 7.2. *Assume that there is a single token in the network. Further assume that at time* 0 *at least one node knows the token and that once they know the token, all nodes broadcast it in every round. In a* 1*-interval connected graph* $G = (V, E)$ *with* n *nodes, after* $r \leq n-1$ *rounds, at least* $r+1$ *nodes know the token. Hence, in particular after* $n-1$ *rounds, all nodes know the token.*

Proof. We can proof the theorem by induction on r. Let $T(r)$ be the set of nodes that know the token after r rounds. We need to show that for all $r \geq 0$, $|T(r)| \geq \min\{r+1, n\}$. Because we assume that at time 0 at least one node knows the token, clearly, $|T(0)| \geq 1$. For the induction step, assume that after r rounds, $|T(r)| \geq \min\{r+1, n\}$. If $T(r) = V$, we have $|T(r+1)| \geq |T(r)| = n$ and we are done. Otherwise, we have $V \setminus T(r) \neq \emptyset$. Therefore, by the 1-interval connectivity assumption, there must be two nodes $u \in T(r)$ and $v \in V \setminus T(r)$ such that $\{u, v\} \in E(r+1)$. Hence, in round $r+1$, node v gets the token an therefore $|T(r+1)| \geq |T(r)| + 1 \geq \min\{r+2, n\}$. □

Remarks:

- Note that Theorem 7.2 only shows that after $n-1$ rounds all nodes know the token. If the nodes do not know n or an upper bound on n, they do not know if all nodes know the token.
- We can apply the above techniques also if there is more than one token in the network, provided that tokens form a totally-ordered set and nodes forward the smallest (or biggest) token they know. It is then guaranteed that the smallest (resp. biggest) token in the network will be known by all nodes after at most $n-1$ rounds. Note, however, that in this case nodes do not *know* when they know the smallest or biggest token.

The next theorem shows that essentially, for the general asynchronous start case, 1-interval connectivity does not suffice to obtain anything better than what is stated by the above theorem. If nodes do not know n or an upper bound on n initially, they cannot find n.

Theorem 7.3. *Counting is impossible in 1-interval connected graphs with asynchronous start.*

Proof. Suppose by way of contradiction that $\mathcal{A}$ is a protocol for counting which requires at most $t(n)$ rounds in 1-interval connected graphs of size n. Let $n' =$

$\max\{t(n)+1, n+1\}$. We will show that the protocol cannot distinguish a static line of length n from a dynamically changing line of length n'.

Given a sequence $A = a_1 \circ \ldots \circ a_m$, let $\text{shift}(A, r)$ denote the cyclic left-shift of A in which the first r symbols ($r \geq 0$) are removed from the beginning of the sequence and appended to the end. Consider an execution in a dynamic line of length n', where the line in round r is composed of two adjacent sections $A \circ B_r$, where $A = 0 \circ \ldots \circ (n-1)$ remains static throughout the execution, and $B(r) = \text{shift}(n \circ \ldots \circ (n'-1), r)$ is left-shifted by one in every round. The computation is initiated by node 0 and all other nodes are initially asleep. We claim that the execution of the protocol in the dynamic graph $G = A \circ B(r)$ is indistinguishable in the eyes of nodes $0, \ldots, n-1$ from an execution of the protocol in the static line of length n (that is, the network comprising section A alone). This is proven by induction on the round number, using the fact that throughout rounds $0, \ldots, t(n)-1$ none of the nodes in section A ever receives a message from a node in section B: although one node in section B is awakened in every round, this node is immediately removed and attached at the end of section B, where it cannot communicate with the nodes in section A. Thus, the protocol cannot distinguish the dynamic graph A from the dynamic graph $A \circ B(r)$, and it produces the wrong output in one of the two graphs. □

Remark:

- The above impossibility result extends to all problems introduced in Section 7.2 as long as we do not assume that the nodes know n or an upper bound on n.

In light of the impossibility result of Theorem 7.3, let us now first consider the synchronous start case where all nodes start the protocol at time 0 (with round 1). We first look at the case where there is no bound on the message size and describe a simple linear-time protocol for counting (and token dissemination). The protocol is extremely simple, but it demonstrates some of the ideas used in some of the later algorithms, where we eliminate the large messages using a stability assumption (T-interval connectivity) which allows nodes to communicate with at least one of their neighbors for at least T rounds.

In the simple protocol, all nodes maintain a set A containing all the IDs they have collected so far. In every round, each node broadcasts A and adds any IDs it receives. Nodes terminate when they first reach a round r in which $|A| \leq r$.

```
$A \leftarrow \{self\}$;
for $r = 1, 2, \ldots$ do
    broadcast $A$;
    receive $B_1, \ldots, B_s$ from neighbors;
    $A \leftarrow A \cup B_1 \cup \ldots \cup B_s$;
    if $|A| \leq r$ then terminate and output $|A|$;
end
```

Algorithm 1: Counting in linear time using large messages

Before analyzing Algorithm 1, let us fix some notation that will help to argue about the algorithms we will study. If x is a variable of an algorithm, let $x_u(r)$ be the value of the variable x at node u after r rounds (immediately before the broadcast operation of round $r+1$). For instance in Algorithm 1, $A_u(r)$ denotes the set of IDs of node u at the end of the r^{th} iteration of the for-loop.

Lemma 7.4. *Assume that we are given an* 1*-interval connected graph* $G = (V, E)$ *and that all nodes in* V *execute Algorithm 1. If all nodes together start at time* 0*, we have* $|A_u(r)| \geq r + 1$ *for all* $u \in V$ *and* $r < n$*.*

Proof. We prove the lemma by induction on r. We clearly have $|A_u(0)| = 1$ for all u because initially each node includes its own ID in A. Hence, the lemma is true for $r = 0$.

For the induction step, assume that the claim of the lemma is true for some given $r < n-1$ for all dynamic graphs G. Let $A'_u(r+1)$ be the set of identifiers known by node u if all nodes start the protocol at time 1 (instead of 0) and run it for r rounds. By the induction hypothesis, we have $|A'_u(r+1)| \geq r+1$. If the algorithm is started at time 0 instead of time 1, the set of identifiers in $A_u(r+1)$ is exactly the union of all the identifiers known by the nodes in $A'_u(r+1)$ after the first round (at time 1). This includes all the nodes in $A'_u(r+1)$ as well as their neighbors in the first round. If $|A'_u(r+1)| \geq r+2$, we also have $|A_u(r+1)| \geq r+2$ and we are done. Otherwise, by 1-interval connectivity, there must at least be one node $v \in V \setminus A'_u(r+1)$ for which there is an edge to a node in $A'_u(r+1)$ in round 1. We therefore have $|A_u(r+1)| \geq |A'_u(r+1)| + 1 \geq r+2$. □

Theorem 7.5. *In an* 1*-interval connected graph* G*, Algorithm 1 terminates at all nodes after* n *rounds and output* n*.*

Proof. Follows directly from Lemma 7.4. For all nodes u, $|A_u(r)| \geq r + 1 > r$ for all $r < n$ and $|A_u(n)| = |A_u(n-1)| = n$. □

Lemma 7.6. *Assume that we are given a* 2*-interval connected graph* $G = (V, E)$ *and that all nodes in* V *execute Algorithm 1. If node* u *is waken up and starts the algorithm at time* t*, it holds that have* $|A_u(t+2r)| \geq r+1$ *for all* $0 \leq r < n$*.*

Proof. The proof follows along the same lines as the proof of Lemma 7.4 (see exercises). □

Remarks:

- Because we did not bound the maximal message size and because every node receives information (an identifier) from each other node, Algorithm 1 can be used to solve all the problems defined in Section 7.2. For the token dissemination problem, the nodes also need to attach a list of all known tokens to all messages

- As a consequence of Theorem 7.3, 1-interval connectivity does not suffice to compute the number of nodes n in a dynamic network if nodes start asynchronously. It turns out that in this case, we need a slightly stronger connectivity assumption. If the network is 2-interval connected instead of 1-interval connected, up to a constant factor in the time complexity, the above results can also be obtained in the asynchronous start case (see exercises).

- For the remainder of the chapter, we will only consider the simpler synchronous start case. For $T \geq 2$, all discussed results that hold for T-interval connected networks with synchronous start also hold for asynchronous start with the same asymptotic bounds.

7.4 Small Messages

We now switch to the more interesting (and more realistic) case where in each round, each node can only broadcast a message of $O(\log n)$ bits. We will first show how to use k-committee election to solve counting. We first describe how to obtain a good upper bound on n. We will then see that the same algorithm can also be used to find n exactly and to solve token dissemination.

7.4.1 k-Verification

The counting algorithm works by successive doubling: at each point the nodes have a guess k for the size of the network, and attempt to verify whether or not $k \geq n$. If it is discovered that $k < n$, the nodes double k and repeat; if $k \geq n$, the nodes halt and output the count.

Suppose that nodes start out in a state that represents a solution to k-committee election: each node has a committee ID, such that no more than k nodes have the same ID, and if $k \geq n$ then all nodes have the same committee ID. The problem of checking whether $k \geq n$ is then equivalent to checking whether there is more than one committee: if $k \geq n$ there must be one committee only, and if $k < n$ there must be more than one. Nodes can therefore check if $k \geq n$ by executing a simple k-round protocol that checks if there is more than one committee in the graph.

The k-verification protocol Each node has a local variable x, which is initially set to 1. While $x_u = 1$, node u broadcasts its committee ID. If it hears from some neighbor a different committee ID from its own, or the special value $\perp$, it sets $x_u \leftarrow 0$ and broadcasts $\perp$ in all subsequent rounds. After k rounds, all nodes output the value of their x variable.

Lemma 7.7. *If the initial state of the execution represents a solution to k-committee election, at the end of the k-verification protocol each node outputs 1 iff $k \geq n$.*

Proof. First suppose that $k \geq n$. In this case there is only one committee in the graph; no node ever hears a committee ID different from its own. After k rounds all nodes still have $x = 1$, and all output 1.

Next, suppose $k < n$. We can show that after the ith round of the protocol, at least i nodes in each committee have $x = 0$. In any round of the protocol, consider a cut between the nodes that belong to a particular committee and still have $x = 1$, and the rest of the nodes, which either belong to a different committee or have $x = 0$. From 1-interval connectivity, there is an edge in the cut, and some node u in the committee that still has $x_u = 1$ hears either a different committee ID or $\perp$. Node u then sets $x_u \leftarrow 0$, and the number of nodes in the committee that still have $x = 1$ decreases by at least one. Since each committee initially contains at most k nodes, after k rounds all nodes in all committees have $x = 0$, and all output 0. □

7.4.2 k-Committee Election

We can solve k-committee in $O(k^2)$ rounds as follows. Each node u stores two local variables, $committee_u$ and $leader_u$. A node that has not yet joined a

```
$leader \leftarrow self$;
$committee \leftarrow \perp$;
for $i = 0, \ldots, k$ do
    // Polling phase
    if $committee = \perp$ then
        $min\_active \leftarrow self$ ; // The node nominates itself for selection
    else
        $min\_active \leftarrow \perp$;
    end
    for $j = 0, \ldots, k-1$ do
        broadcast $min\_active$;
        receive $x_1, \ldots, x_s$ from neighbors;
        $min\_active \leftarrow \min\{min\_active, x_1, \ldots, x_s\}$;
    end
    // Update leader
    $leader \leftarrow \min\{leader, min\_active\}$;
    // Selection phase
    if $leader = self$ then
        // Leaders invite the smallest ID they heard
        $invitation \leftarrow (self, min\_active)$;
    else
        // Non-leaders do not invite anybody
        $invitation \leftarrow \perp$
    end
    for $j = 0, \ldots, k-1$ do
        broadcast $invitation$;
        receive $y_1, \ldots, y_s$ from neighbors;
        $invitation \leftarrow \min\{invitation, y_1, \ldots, y_s\}$ ; // (in lexicographic
        order)
    end
    // Join the leader's committee, if invited
    if $invitation = (leader, self)$ then
        $committee = leader$;
    end
end
if $committee = \perp$ then
    $committee \leftarrow self$;
end
```

Algorithm 2: k-committee in always-connected graphs

committee is called *active*, and a node that has joined a committee is *inactive*. Once nodes have joined a committee they do not change their choice.

Initially all nodes consider themselves leaders, but throughout the protocol, any node that hears an ID smaller than its own adopts that ID as its leader. The protocol proceeds in k cycles, each consisting of two phases, *polling* and *selection*.

1. Polling phase: for $k-1$ rounds, all nodes propagate the ID of the smallest active node of which they are aware.

2. Selection phase: in this phase, each node that considers itself a leader selects the smallest ID it heard in the previous phase and invites that node to join its committee. An invitation is represented as a pair (x, y), where x is the ID of the leader that issued the invitation, and y is the ID of the invited node. All nodes propagate the smallest invitation of which they are aware for $k-1$ (invitations are sorted in lexicographic order, so the invitations issued by the smallest node in the network will win out over other invitations. It turns out, though, that this is not necessary for correctness; it is sufficient for each node to forward an arbitrary invitation from among those it received).

   At the end of the selection phase, a node that receives an invitation to join its leader's committee does so and becomes inactive. (Invitations issued by nodes that are not the current leader can be accepted or ignored; this, again, does not affect correctness.)

At the end of the k cycles, any node u that has not been invited to join a committee outputs $committee_u = u$. The details are given in Algorithm 2.

Lemma 7.8. *Algorithm 2 solves the k-committee problem in $O(k^2)$ rounds in 1-interval connected networks.*

Proof. The time complexity is immediate. To prove correctness, we show that after the protocol ends, the values of the local $committee_u$ variables constitute a valid solution to k-committee.

1. In each cycle, each node invites at most one node to join its committee. After k cycles at most k nodes have joined any committee. Note that the first node invited by a leader u to join u's committee is always u itself. Thus, if after k cycles node u has not been invited to join a committee, it follows that u did not invite any other node to join its committee; when it forms its own committee in the last line of the algorithm, the committee's size is 1.

2. Suppose that $k \geq n$, and let u be the node with the smallest ID in the network. Following the polling phase of the first cycle, all nodes v have $leader_v = u$ for the remainder of the protocol. Thus, throughout the execution, only node u issues invitations, and all nodes propagate u's invitations. Since $k \geq n$ rounds are sufficient for u to hear the ID of the minimal active node in the network, in every cycle node u successfully identifies this node and invites it to join u's committee. After k cycles, all nodes will have joined.

□

Remark:

- The protocol can be modified easily to solve all-to-all token dissemination if $k \geq n$. Let t_u be the token node u received in its input (or $\perp$ if node u did not receive a token). Nodes attach their tokens to their IDs, and send pairs of the form (u, t_u) instead of just u. Likewise, invitations now contain the token of the invited node, and have the structure $(\mathit{leader}, (u, t_u))$. The min operation disregards the token and applies only to the ID. At the end of each selection phase, nodes extract the token of the invited node, and add it to their collection. By the end of the protocol every node has been invited to join the committee, and thus all nodes have seen all tokens.

7.5 More Stable Graphs

```
$S \leftarrow \emptyset$;
for $i = 0, \ldots, \lceil k/T \rceil - 1$ do
    for $r = 0, \ldots, 2T - 1$ do
        if $S \neq A$ then
            $t \leftarrow \min(A \setminus S)$;
            broadcast $t$;
            $S \leftarrow S \cup \{t\}$
        end
        receive $t_1, \ldots, t_s$ from neighbors;
        $A \leftarrow A \cup \{t_1, \ldots, t_s\}$
    end
    $S \leftarrow \emptyset$
end
return $A$
```

Procedure `disseminate`(A, T, k)

In this section we show that in T-interval connected graphs the computation can be sped up by a factor of T. To do this we employ a neat pipelining effect, using the temporarily stable subgraphs that T-interval connectivity guarantees; this allows us to disseminate information more quickly. Basically, because we are guaranteed that some edges and paths persist for T rounds, it suffices to send a particular ID or token only once in T rounds to guarantee progress. Other rounds can then be used for different tokens. For convenience we assume that the graph is $2T$-interval connected for some $T \geq 1$.

Procedure `disseminate` gives an algorithm for exchanging at least T pieces of information in n rounds when the dynamic graph is $2T$-interval connected. The procedure takes three arguments: a set of tokens A, the parameter T, and a guess k for the size of the graph. If $k \geq n$, each node is guaranteed to learn the T smallest tokens that appeared in the input to all the nodes.

The execution of procedure `disseminate` is divided into $\lceil k/T \rceil$ phases, each consisting of $2T$ rounds. During each phase, each node maintains the set A of tokens it has already learned and a set S of tokens it has already broadcast in the current phase (initially empty). In each round of the phase, the node broadcasts the smallest token it has not yet broadcast in the current phase, then adds that token to S.

We refer to each iteration of the inner loop as a *phase*. Since a phase lasts $2T$ rounds and the graph is $2T$-interval connected, there is some connected subgraph that exists throughout the phase. Let G'_i be a connected subgraph that exists throughout phase i, for $i = 0, \ldots, \lceil k/T \rceil - 1$. We use $\mathrm{dist}_i(u, v)$ to denote the distance between nodes $u, v \in V$ in G'_i.

Let $K_t(r)$ denote the set of nodes that know token t by the beginning of round r, that is, $K_t(r) = \{u \in V \mid t \in A_u(r)\}$. In addition, let I be the set of T smallest tokens in $\bigcup_{u \in V} A_u(0)$. Our goal is to show that when the protocol terminates we have $K_t(r) = V$ for all $t \in I$.

For a node $u \in V$, a token $t \in P$, and a phase i, we define $\mathrm{tdist}_i(u, t)$ to be the distance of u from the nearest node in G'_i that knows t at the beginning of phase i:

$$\mathrm{tdist}(u, t) := \min \{\mathrm{dist}_i(u, v) \mid v \in K_t(2T \cdot i)\}.$$

Here and in the sequel, we use the convention that $\min \emptyset := \infty$. For convenience, we use $S_u^i(r) := S_u(2T \cdot i + r)$ to denote the value of S_u in round r of phase i. Similarly we denote $A_u^i(r) := A_u(2T \cdot i + r)$ and $K_t^i(r) := K_t(2T \cdot i + r)$. Correctness hinges on the following property.

Lemma 7.9. *For any node $u \in V$, token $t \in \bigcup_{v \in V} A_v(0)$, and round r such that $\mathrm{tdist}_i(u, t) \leq r \leq 2T$, either $t \in S_u^i(r + 1)$ or $S_u(r + 1)$ includes at least $(r - \mathrm{tdist}_i(u, t))$ tokens that are smaller than t.*

Proof. By induction on r. For $r = 0$ the claim is immediate.

Suppose the claim holds for round $r - 1$ of phase i, and consider round $r \geq \mathrm{tdist}_i(u, t)$. If $r = \mathrm{tdist}_i(u, t)$, then $r - \mathrm{tdist}_i(u, t) = 0$ and the claim holds trivially. Thus, suppose that $r > \mathrm{tdist}_i(u, t)$. Hence, $r - 1 \geq \mathrm{tdist}_i(u, t)$, and the induction hypothesis applies: either $t \in S_u^i(r)$ or $S_u^i(r)$ includes at least $(r - 1 - \mathrm{tdist}_i(u, t))$ tokens that are smaller than t. In the first case we are done, since $S_u^i(r) \subseteq S_u^i(r+1)$; thus, assume that $t \notin S_u^i(r)$, and $S_u^i(r)$ includes at least $(r - 1 - \mathrm{tdist}_i(u, t))$ tokens smaller than t. However, if $S_u^i(r)$ includes at least $(r - \mathrm{tdist}_i(u, t))$ tokens smaller than t, then so does $S_u^i(r + 1)$, and the claim is again satisfied; thus we assume that $S_u^i(r)$ includes *exactly* $(r - 1 - \mathrm{tdist}_i(u, t))$ tokens smaller than t.

It is sufficient to prove that $\min \left(A_u^i(r) \setminus S_u^i(r)\right) \leq t$: if this holds, then in round r node u broadcasts $\min \left(A_u^i(r) \setminus S_u^i(r)\right)$, which is either t or a token smaller than t; thus, either $t \in S_u^i(r + 1)$ or $S_u^i(r + 1)$ includes at least $(r - \mathrm{tdist}_i(u, t))$ tokens smaller than t, and the claim holds.

First we handle the case where $\mathrm{tdist}_i(u, t) = 0$. In this case, $t \in A_u^i(0) \subseteq A_u^i(r)$. Since we assumed that $t \notin S_u^i(r)$ we have $t \in A_u^i(r) \setminus S_u^i(r)$, which implies that $\min \left(A_u^i(r) \setminus S_u^i(r)\right) \leq t$.

Next suppose that $\mathrm{tdist}_i(u, t) > 0$. Let $x \in K_t^i(0)$ be a node such that $\mathrm{dist}_i(u, x) = \mathrm{tdist}(u, t)$ (such a node must exist from the definition of $\mathrm{tdist}_i(u, t)$), and let v be a neighbor of u along the path from u to x in G_i, such that $\mathrm{dist}_i(v, x) = \mathrm{dist}_i(u, x) - 1 < r$. From the induction hypothesis, either $t \in S_v^i(r)$ or $S_v^i(r)$ includes at least $(r - 1 - \mathrm{tdist}_i(v, t)) = (r - \mathrm{tdist}_i(u, t))$ tokens that are smaller than t. Since the edge between u and v exists throughout phase i, node u receives everything v sends in phase i, and hence $S_v^i(r) \subseteq A_u^i(r)$. Finally, because we assumed that $S_u^i(r)$ contains exactly $(r - 1 - \mathrm{tdist}_i(u, t))$ tokens smaller than t, and does not include t itself, we have $\min \left(A_u^i(r) \setminus S_u^i(r)\right) \leq t$, as desired. $\square$

Using Lemma 7.9 we can show: correct.

Lemma 7.10. *If $k \geq n$, at the end of procedure* `disseminate` *the set A_u of each node u contains the T smallest tokens.*

Proof. Let $N_i^d(t) := \{u \in V \mid \text{tdist}_i(u,t) \leq d\}$ denote the set of nodes at distance at most d from some node that knows t at the beginning of phase i, and let t be one of the T smallest tokens.

From Lemma 7.9, for each node $u \in N_i^T(t)$, either $t \in S_u^i(2T+1)$ or $S_u^i(2T+1)$ contains at least $2T - T = T$ tokens that are smaller than t. But t is one of the T smallest tokens, so the second case is impossible. Therefore all nodes in $\mathbb{N}_i^T(t)$ know token t at the end of phase i. Because G_i is connected we have $|N_i^T(t)| \geq \min\{n - |K_i(t)|, T\}$; that is, in each phase T new nodes learn t, until all the nodes know t. Since there are no more than k nodes and we have $\lceil k/T \rceil$ phases, at the end of the last phase all nodes know t. $\square$

To solve counting and token dissemination with up to n tokens, we use Procedure `disseminate` to speed up the k-committee election protocol from Algorithm 2. Instead of inviting one node in each cycle, we can use `disseminate` to have the leader learn the IDs of the T smallest nodes in the polling phase, and use procedure `disseminate` again to extend invitations to all T smallest nodes in the selection phase. Thus, in $O(k+T)$ rounds we can increase the size of the committee by T.

Theorem 7.11. *It is possible to solve k-committee election in $O(k + k^2/T)$ rounds in T-interval connected graphs. When used in conjunction with the k-verification protocol, this approach yields $O(n+n^2/T)$-round protocols for counting all-to-all token dissemination.*

Remarks:

- The same result can also be achieved for the asynchronous start case, as long as $T \geq 2$.
- The described algorithm is based on the assumptions that all nodes know T (or that they have a common lower bound on T). At the cost of a log-factor, it is possible to drop this assumption and adapt to the actual interval-connectivity T.
- It is not known whether the bound of Theorem 7.11 is tight. It can be shown that it is tight for a restricted class of protocols (see exercises).
- If we make additional assumptions about the stable subgraphs that are guaranteed for intervals of length T, the bound in Theorem 7.11 can be improved. E.g., if intervals of length T induce a stable k-vertex connected subgraph, the complexity can be improved to $O(n + n^2/(kT))$.

Chapter 8

Peer-to-Peer Computing

"Indeed, I believe that virtually *every* important aspect of programming arises somewhere in the context of [sorting and] searching!"

– Donald E. Knuth, The Art of Computer Programming

8.1 Introduction

Unfortunately, the term *peer-to-peer* (P2P) is ambiguous, 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. P2P file sharing is immensely popular, currently at least half of the total Internet traffic is due to P2P!

- 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.)

In other words, 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.

8.2 Architecture Variants

Several P2P architectures are known:

- Client/Server goes P2P: Even though Napster is known to the be 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 (Algorithm 11 in Chapter 3). 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, almost all 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, Kademlia, Koorde, SkipGraph, SkipNet, etc. In practice structured P2P architectures are not yet popular, apart from the Kad (from Kademlia) architecture which comes for free with the eMule client. 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!

8.3 Hypercubic Networks

(Thanks to Christian Scheideler, TUM, for the pictures in this section.)

In this section we will introduce 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. Similarly to Chapter 4 we employ 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 routed at v. See Figure 8.1 for an example.

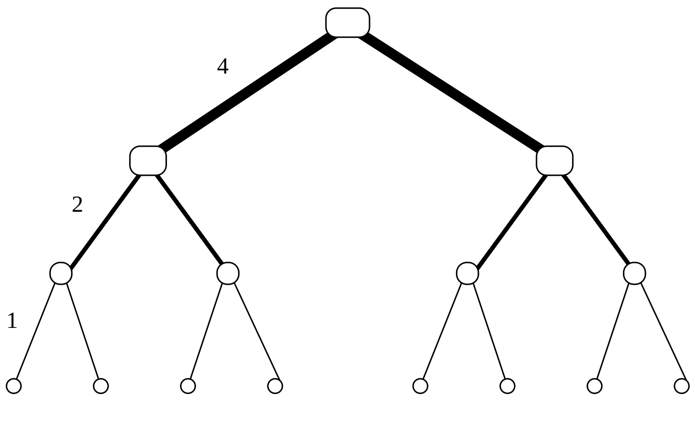


Figure 8.1: The structure of a fat tree.

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 treat all edges in the following to be of capacity 1. In the following, $[x]$ means the set $\{0, \ldots, x-1\}$.

Definition 8.1 (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 8.2 presents a linear array, a torus, and a hypercube.*

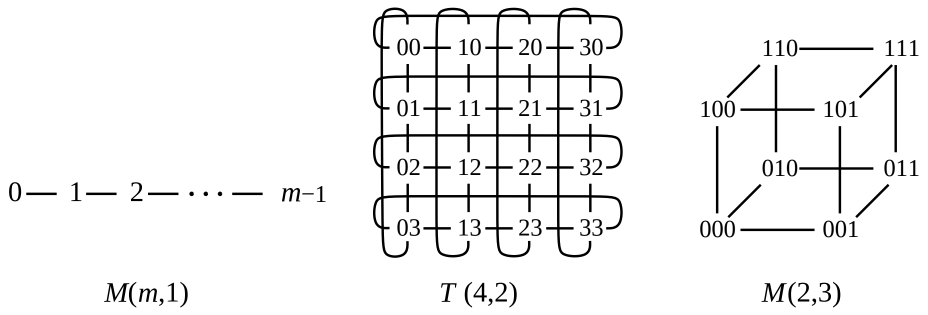


Figure 8.2: The structure of $M(m, 1)$, $T(4, 2)$, and $M(2, 3)$.

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 directly be used for a structured P2P architecture. It is trivial to construct 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. We give some more details in Section 8.4. 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 8.2 (Butterfly)**.** *Let $d \in N$. 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

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

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 8.3 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.

- The structure of a butterfly might remind you of sorting networks from Chapter 4. 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) will recognize the structure without doubt. 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.

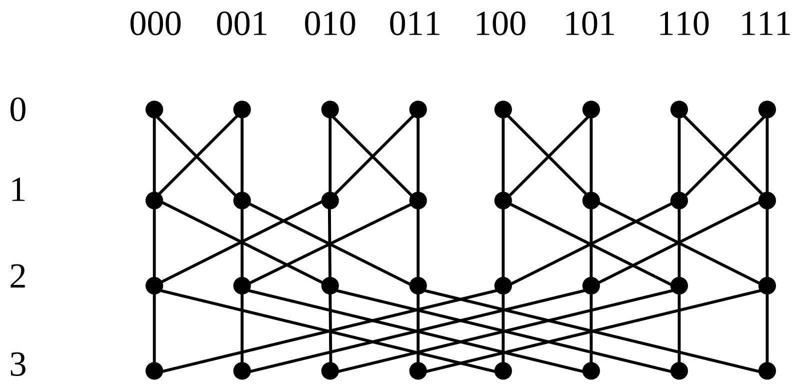
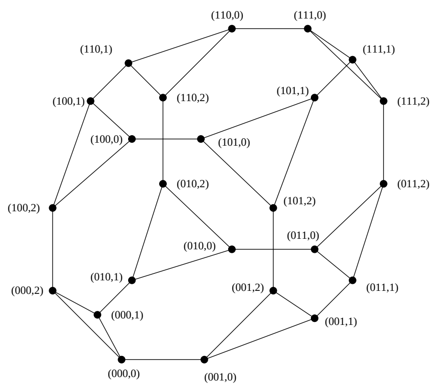


Figure 8.3: The structure of BF(3).



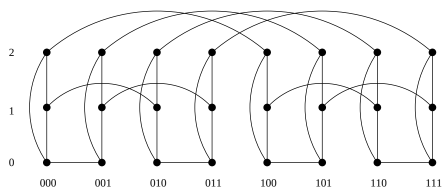


Figure 8.4: The structure of CCC(3).

Definition 8.3 (Cube-Connected-Cycles). *Let $d \in N$. 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*

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

Remarks:

- Two possible representations of a CCC can be found in Figure 8.4.
- The shuffle-exchange is yet another way of transforming the hypercubic interconnection structure into a constant degree network.

Definition 8.4 (Shuffle-Exchange). *Let $d \in N$. 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*

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

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 8.5 shows the 3- and 4-dimensional shuffle-exchange graph.

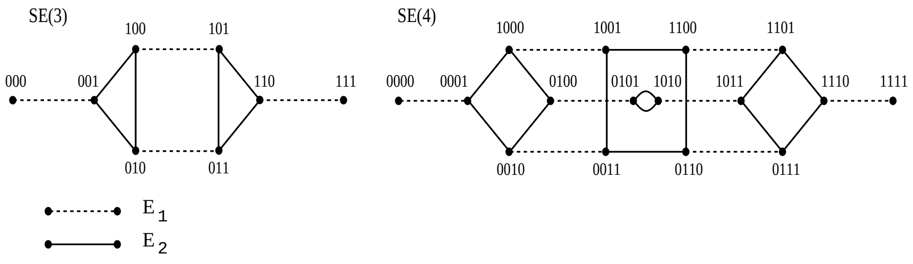


Figure 8.5: The structure of SE(3) and SE(4).

Definition 8.5 (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* $v=(v_1,\dots,v_d)$*.*

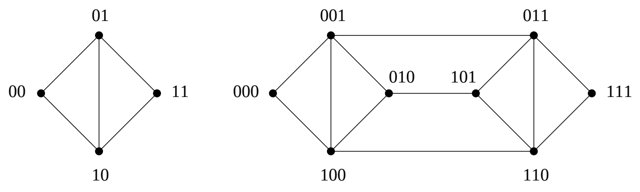


Figure 8.6: The structure of $DB(2,2)$ and $DB(2,3)$.

Remarks:

- Two examples of a DeBruijn graph can be found in Figure 8.6. 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 8.6 (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 MIS 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 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 8.7. *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

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

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,

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

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

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.

8.4 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 (see exercises), 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 8.7.

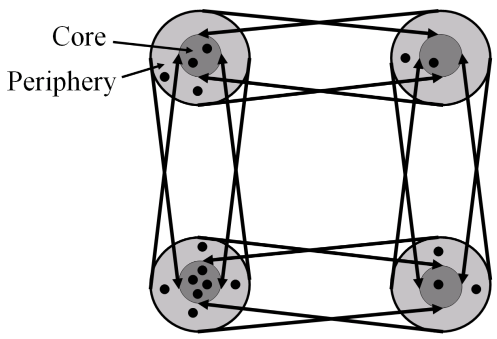


Figure 8.7: A simulated 2-dimensional hypercube with four nodes, each consisting of several peers. Also, all the peers are either in the core or in the periphery of a node. All peers within the same node are completely connected to each other, and additionally, all peers of a node are connected to the 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 8.8 (DHT with Churn). *We have a fully scalable, efficient P2P*

[1]Having 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.

system which tolerates $\mathrm{O}(\log n)$ *worst-case joins and/or crashes per constant time interval. As in other P2P systems, peers have* $\mathrm{O}(\log n)$ *neighbors, and the usual operations (e.g., search, insert) take time* $\mathrm{O}(\log n)$*.*

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.
- 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 9

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 9.1.

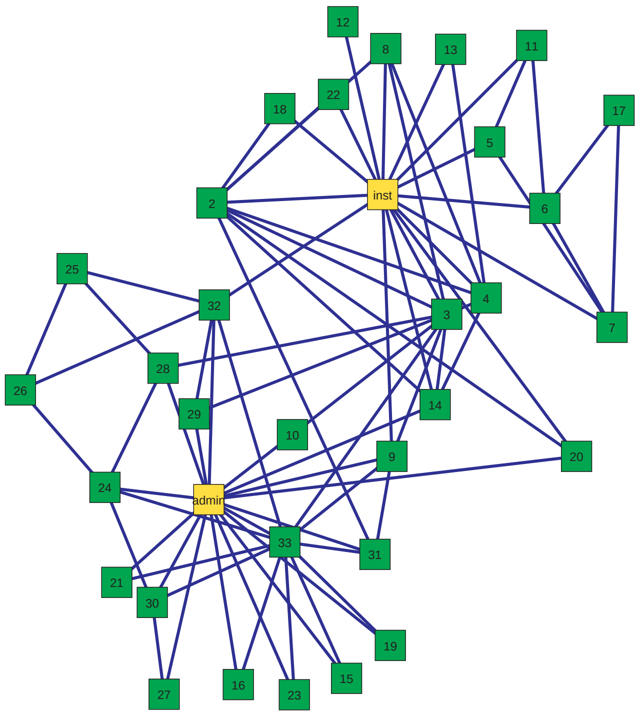


Figure 9.1: This graph shows the social relations between the members of a karate club, studied by anthropologist Wayne Zachary in the 1970s. Two people (nodes) stand out, the instructor and the administrator of the club, both happen to have many friends among club members. At some point, a dispute caused the club to split into two. Can you predict how the club partitioned? (If not, just search the Internet for Zachary and Karate.)

9.1 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 were 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 9.1. *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 9.2 (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 $d\big((i,j),(k,\ell)\big) = |k-i| + |\ell - j|$ 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 9.2 illustrates the model.*

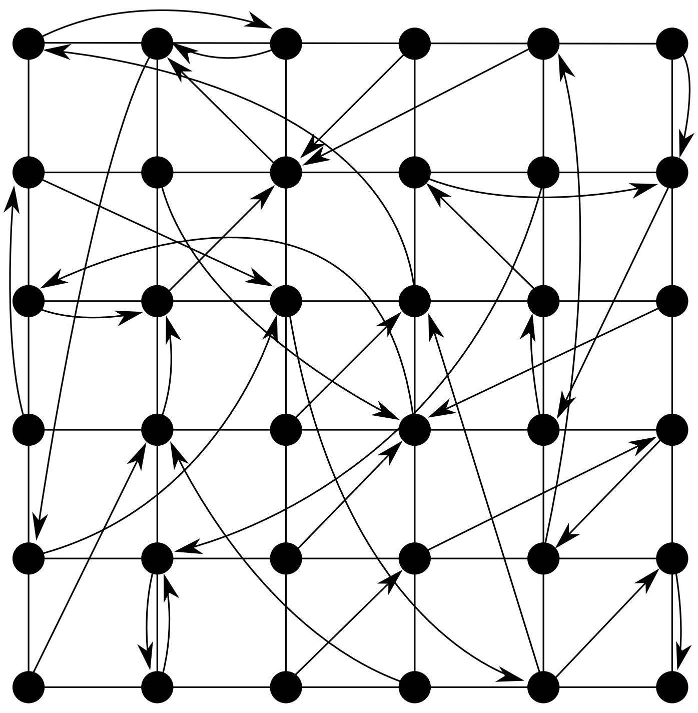


Figure 9.2: Augmented grid with $m = 6$

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 $\alpha \to \infty$, 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 9.3 (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 $e^{c!} \log n + 5000c$ 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 9.4. *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 (see exercises).

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 (see exercises).

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 (see exercises). 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 34.

Algorithm 34 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 9.5. *In the augmented grid, Algorithm 34 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. □

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 9.6. *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

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

□

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 9.7 (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 9.8. *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

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

Theorem 9.9. *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

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

9.2 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 9.10. *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 9.3 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. □

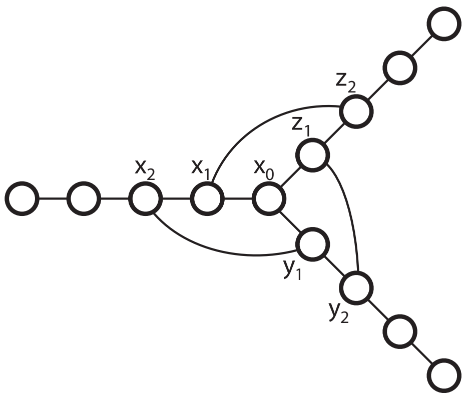


Figure 9.3: Counter example.

Chapter 10

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 1).

10.1 MIS

Definition 10.1 (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.*

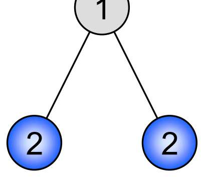


Figure 10.1: Example graph with 1) a maximal independent set (MIS) and 2) a maximum independent set (MaxIS).

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 10.1).
- 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 35 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 10.2 (Analysis of Algorithm 35)**.** *Algorithm 35 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 1), 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 10.3. *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 1.17 and Corollary 10.3 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 5 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.

10.2 Fast MIS from 1986

Algorithm 36 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 10.4 (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

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

Then

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

□

Lemma 10.5 (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 10.4 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 $\sum\limits_{w \in N(v)} \frac{1}{2d(w)} \geq \frac{1}{6}$ there is a subset of neighbors $S \subseteq N(v)$ such that $\frac{1}{6} \leq \sum\limits_{w \in S} \frac{1}{2d(w)} \leq \frac{1}{3}$

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

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

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 $P\left[u \in M\right] \geq P\left[u \in \text{MIS}\right]$ we get

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

□

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 10.6 (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 10.7. *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. □

Continuing the proof of Lemma 10.6: According to Lemma 10.7 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 10.8 (Analysis of Algorithm 36)**.** *Algorithm 36 terminates in expected time $O(\log n)$.*

Proof: With Lemma 10.5 a good node (and therefore a good edge!) will be deleted with constant probability. Since at least half of the edges are good (Lemma 10.6) a constant fraction of edges will be deleted in each phase.

More formally: With Lemmas 10.5 and 10.6 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. □

Remarks:

- With a bit of more math one can even show that Algorithm 36 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:

10.3 Fast MIS from 2009

Algorithm 37 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 10.9 (Linearity of Expectation)**.** *Let X_i, $i = 1, \ldots, k$ denote random variables, then*

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

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

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

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 10.10 (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 $(v \to w)$. The probability of event $(v \to w)$ 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 weight events properly.

Whether we remove the edges adjacent to w because of event $(v \to w)$ is a random variable $X_{(v\to w)}$. If event $(v \to w)$ occurs, $X_{(v\to w)}$ has the value $d(w)$, if not it has the value 0. For each edge $\{v, w\}$ we have two such variables, the event $X_{(v\to w)}$ and $X_{(w\to v)}$. Due to Theorem 10.9, the expected value of the sum X of all these random variables is at least

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

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 $(u \to v)$ happens, but X also includes the edge if the event $(x \to w)$ happens. So we may have counted the edge $\{v, w\}$ twice. Fortunately however, not more than twice, because at most one event $(\cdot \to v)$ and at most one event $(\cdot \to w)$ can happen. If $(u \to v)$ happens, we know that $r(u) < r(w)$ for all $w \in N(v)$; hence another $(u' \to v)$ 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 37 quite easily.

Theorem 10.11 (Expected running time of Algorithm 37)**.** *Algorithm 37 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 10.10.

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 10.10 and another standard tool, namely Chernoff's Bound.

Definition 10.12 (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 10.13 (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 10.14 (Running Time of Algorithm 37)**.** *Algorithm 37 terminates w.h.p. in $O(\log n)$ time.*

Proof: In Theorem 10.11 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.

10.4 Applications

Definition 10.15 (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 37 directly produces a $O(\log n)$ bound for maximal matching.
- More importantly, our MIS algorithm can also be used for vertex coloring (Problem 1.1):

Algorithm 38 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 37 on $G'$.
7: If node $v_i$ is in the MIS in $G'$, then node $v$ gets color $i$.
```

Theorem 10.16 (Analysis of Algorithm 38). *Algorithm 38 $(\Delta + 1)$-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 $O\left(n^2\right)$ nodes and the exponent becomes a constant factor when applying the logarithm.

Remarks:

- This solves our open problem from Chapter 1.1!
- Together with Corollary 10.3 we get quite close ties between $(\Delta + 1)$-coloring and the MIS problem.
- However, in general Algorithm 38 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 10.17 (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 10.18 ((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 an 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 10.19. *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 37 and output it as the dominating set, which takes $O(\log n)$ rounds w.h.p.

Chapter 11

Locality Lower Bounds

In Chapter 1, 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 1 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 39 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
```

11.1 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 11.1. *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 39 (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 39. 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. □

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 11.2 (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 11.1.

Corollary 11.3. *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 11.1, we know that we can transform Algorithm $\mathcal{A}$ to the canonical form given by Algorithm 39. After r communication rounds, every node v knows exactly its r-hop view. This information suffices to compute the output of node v. □

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 $(\ell_{-r}, \ell_{-r+1}, \ldots, \ell_0, \ldots, \ell_r)$ 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 $\mathcal{V}_r = (\ell_{-r}, \ldots, \ell_r)$ and $\mathcal{V}'_r = (\ell'_{-r}, \ldots, \ell'_r)$. If $\ell'_i = \ell_{i+1}$ for $-r \le i \le r-1$ and if $\ell'_r \neq \ell_i$ for $-r \le i \le r$, 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.

11.2 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 11.4 (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 11.5. *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 $\mathcal{N}_r(\mathcal{G})$. □

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 ($[n] = \{1, \ldots, n\}$):

$$V[\mathcal{B}_{k,n}] \;=\; \big\{(\alpha_1, \ldots, \alpha_k) : \alpha_i \in [n], i < j \rightarrow \alpha_i < \alpha_j\big\} \qquad (11.1)$$

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

$$\forall i \in \{1, \ldots, k-1\} : \beta_i = \alpha_{i+1}. \qquad (11.2)$$

Lemma 11.6. *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 11.1. 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. □

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 11.7 (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 $\big((w,x),(y,z)\big)$ 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 11.8. *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 (11.1) and (11.2). 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 $\underline{\gamma}_2$. □

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

Lemma 11.9. *For the chromatic numbers $\chi(G)$ and $\chi(\mathcal{DL}(G))$ of a directed graph G and its diline graph, it holds that*

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

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. □

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 1 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 11.10. *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 11.8 and 11.9. □

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

Theorem 11.11. *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 11.10, 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 $r < \log^* n/2 - 1$. □

Corollary 11.12. *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 1 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 $n_1, \ldots, n_c$, there is a Ramsey number $R(n_1, \ldots, n_c)$, such that if the edges of a complete graph with $R(n_1, \ldots, n_c)$ 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 $S = \{v_1, \ldots, v_{k+1}\}$ of $k + 1$ identifiers, such that all $k + 1$ hyperedges consisting of k nodes from S have the same color. Note that both $\{v_1, \ldots, v_k\}$ and $\{v_2, \ldots, v_{k+1}\}$ 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 12

Synchronizers

So far, we have mainly studied synchronous algorithms because generally, asynchronous algorithms are often more difficult to obtain and it is substantially harder to reason about asynchronous algorithms than about synchronous ones. For instance, computing a BFS tree (cf. Chapter 3) efficiently requires much more work in an asynchronous system. However, many real systems are not synchronous and we therefore have to design asynchronous algorithms. In this chapter, we will look at general simulation techniques, called *synchronizers*, that allow to run a synchronous algorithm in an asynchronous environment.

12.1 Basics

A synchronizer generates sequences of *clock pulses* at each node of the network satisfying the condition given by the following definition.

Definition 12.1 (valid clock pulse)**.** *We call a clock pulse generated at a node v* valid *if it is generated after v received all the messages of the synchronous algorithm sent to v by its neighbors in the previous pulses.*

Given a mechanism that generates the clock pulses, a synchronous algorithm is turned into an asynchronous algorithm in an obvious way: As soon as the i^{th} clock pulse is generated at node v, v performs all the actions (local computations and sending of messages) of round i of the synchronous algorithm.

Theorem 12.2. *If all generated clock pulses are valid according to Definition 12.1, the above method provides an asynchronous algorithm that behaves exactly the same way as the given synchronous algorithm.*

Proof. When the i^{th} pulse is generated at a node v, v has sent and received exactly the same messages and performed the same local computations as in the first $i - 1$ rounds of the synchronous algorithm. □

The main problem when generating the clock pulses at a node v is that v cannot know what messages its neighbors are sending to it in a given synchronous round. Because there are no bounds on link delays, v cannot simply wait "long enough" before generating the next pulse. In order satisfy Definition 12.1, nodes have to send additional messages for the purpose of synchronization. The total complexity of the resulting asynchronous algorithm depends on the overhead introduced by the synchronizer. For a synchronizer $\mathcal{S}$, let $T(\mathcal{S})$ and $M(\mathcal{S})$ be the time and message complexities of $\mathcal{S}$ for each generated clock pulse. As we will see, some of the synchronizers need an initialization phase. We denote the time and message complexities of the initialization by $T_{\text{init}}(\mathcal{S})$ and $M_{\text{init}}(\mathcal{S})$, respectively. If $T(\mathcal{A})$ and $M(\mathcal{A})$ are the time and message complexities of the given synchronous algorithm $\mathcal{A}$, the total time and message complexities T_{tot} and M_{tot} of the resulting asynchronous algorithm then become

$$T_{tot} = T_{\text{init}}(\mathcal{S})+T(\mathcal{A})\cdot(1+T(\mathcal{S})) \text{ and } M_{tot} = M_{\text{init}}(\mathcal{S})+M(\mathcal{A})+T(\mathcal{A})\cdot M(\mathcal{S}),$$

respectively.

Remarks:

- Because the initialization only needs to be done once for each network, we will mostly be interested in the overheads $T(\mathcal{S})$ and $M(\mathcal{S})$ per round of the synchronous algorithm.

Definition 12.3 (Safe Node)**.** *A node v is* safe *with respect to a certain clock pulse if all messages of the synchronous algorithm sent by v in that pulse have already arrived at their destinations.*

Lemma 12.4. *If all neighbors of a node v are safe with respect to the current clock pulse of v, the next pulse can be generated for v.*

Proof. If all neighbors of v are safe with respect to a certain pulse, v has received all messages of the given pulse. Node v therefore satisfies the condition of Definition 12.1 for generating a valid next pulse. □

Remarks:

- In order to detect safety, we require that all algorithms send acknowledgements for all received messages. As soon as a node v has received an acknowledgement for each message that it has sent in a certain pulse, it knows that it is safe with respect to that pulse. Note that sending acknowledgements does not increase the asymptotic time and message complexities.

12.2 Synchronizer α

Algorithm 40 Synchronizer α (at node v)

```
1: wait until $v$ is safe
2: send SAFE to all neighbors
3: wait until $v$ receives SAFE messages from all neighbors
4: start new pulse
```

Synchronizer α is very simple. It does not need an initialization. Using acknowledgements, each node eventually detects that it is safe. It then reports this fact directly to all its neighbors. Whenever a node learns that all its neighbors are safe, a new pulse is generated. Algorithm 40 formally describes synchronizer α.

Theorem 12.5. *The time and message complexities of synchronizer α per synchronous round are*

$$T(\alpha) \;=\; O(1) \quad \textit{and} \quad M(\alpha) \;=\; O(m).$$

Proof. Communication is only between neighbors. As soon as all neighbors of a node v become safe, v knows of this fact after one additional time unit. For every clock pulse, synchronizer α sends at most four additional messages over every edge: Each of the nodes may have to acknowledge a message and reports safety. □

Remarks:

- Synchronizer α was presented in a framework, mostly set up to have a common standard to discuss different synchronizers. Without the framework, synchronizer α can be explained more easily:
    1. Send message to all neighbors, include round information i and actual data of round i (if any).
    2. Wait for message of round i from all neighbors, and go to next round.
- Although synchronizer α allows for simple and fast synchronization, it produces awfully many messages. Can we do better? Yes.

12.3 Synchronizer β

Algorithm 41 Synchronizer β (at node v)

```
1: wait until $v$ is safe
2: wait until $v$ receives SAFE messages from all its children in $T$
3: if $v \neq \ell$ then
4:     send SAFE message to parent in $T$
5:     wait until PULSE message received from parent in $T$
6: end if
7: send PULSE message to children in $T$
8: start new pulse
```

Synchronizer β needs an initialization that computes a leader node ℓ and a spanning tree T that is rooted at ℓ. As soon as all nodes are safe, this information is propagated to ℓ by means of a convergecast. The leader then broadcasts this information to all nodes. The details of synchronizer β are given in Algorithm 41.

Theorem 12.6. *The time and message complexities of synchronizer β per synchronous round are*

$$T(\beta) \;=\; O(\text{diameter}(T)) \;\leq\; O(n) \quad \textit{and} \quad M(\beta) \;=\; O(n).$$

The time and message complexities for the initialization are

$$T_{\text{init}}(\beta) \;=\; O(n) \quad \textit{and} \quad M_{\text{init}}(\beta) \;=\; O(m + n\log n).$$

Proof. Because the diameter of T is at most $n-1$, the convergecast and the broadcast together take at most $2n-2$ time units. Per clock pulse, the synchronizer sends at most $2n-2$ synchronization messages (one in each direction over each edge of T).

With an improvement (due to Awerbuch) of the GHS algorithm (Algorithm 15) you saw in Chapter 3, it is possible to construct an MST in time $O(n)$ with $O(m+n\log n)$ messages in an asynchronous environment. Once the tree is computed, the tree can be made rooted in time $O(n)$ with $O(n)$ messages. □

Remarks:

- We now got a time-efficient synchronizer (α) and a message-efficient synchronizer (β), it is only natural to ask whether we can have the best of both worlds. And, indeed, we can. How is that synchronizer called? Quite obviously: γ .

12.4 Synchronizer γ

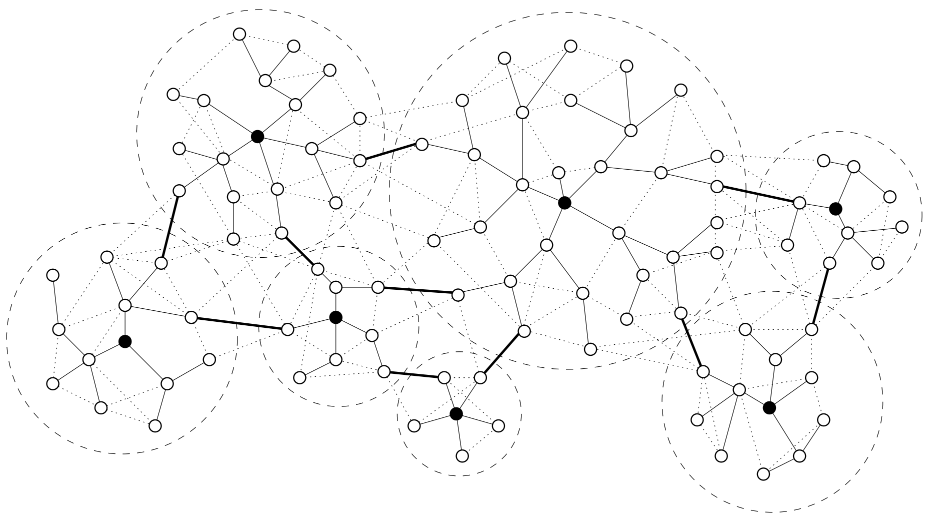


Figure 12.1: A cluster partition of a network: The dashed cycles specify the clusters, cluster leaders are black, the solid edges are the edges of the intracluster trees, and the bold solid edges are the intercluster edges

Synchronizer γ can be seen as a combination of synchronizers α and β. In the initialization phase, the network is partitioned into clusters of small diameter. In each cluster, a leader node is chosen and a BFS tree rooted at this leader node is computed. These trees are called the *intracluster trees.* Two clusters C_1 and C_2 are called neighboring if there are nodes $u \in C_1$ and $v \in C_2$ for which $(u, v) \in E$. For every two neighboring clusters, an *intercluster edge* is chosen, which will serve for communication between these clusters. Figure 12.1 illustrates this partitioning into clusters. We will discuss the details of how to construct such a partition in the next section. We say that a cluster is safe if all its nodes are safe.

Synchronizer γ works in two phases. In a first phase, synchronizer β is applied separately in each cluster by using the intracluster trees. Whenever the leader of a cluster learns that its cluster is safe, it reports this fact to all the nodes in the clusters as well as to the leaders of the neighboring clusters. Now, the nodes of the cluster enter the second phase where they wait until all the neighboring clusters are known to be safe and then generate the next pulse. Hence, we essentially apply synchronizer α between clusters. A detailed description is given by Algorithm 42.

Algorithm 42 Synchronizer γ (at node v)

```
1: wait until $v$ is safe
2: wait until $v$ receives SAFE messages from all children in intracluster tree
3: if $v$ is not cluster leader then
4:    send SAFE message to parent in intracluster tree
5:    wait until CLUSTERSAFE message received from parent
6: end if
7: send CLUSTERSAFE message to all children in intracluster tree
8: send NEIGHBORSAFE message over all intercluster edges of $v$
9: wait until $v$ receives NEIGHBORSAFE messages from all adjacent inter-
   cluster edges and all children in intracluster tree
10: if $v$ is not cluster leader then
11:    send NEIGHBORSAFE message to parent in intracluster tree
12:    wait until PULSE message received from parent
13: end if
14: send PULSE message to children in intracluster tree
15: start new pulse
```

Theorem 12.7. *Let m_C be the number of intercluster edges and let k be the maximum cluster radius (i.e., the maximum distance of a leaf to its cluster leader). The time and message complexities of synchronizer γ are*

$$T(\gamma) \;=\; O(k) \quad \text{and} \quad M(\gamma) \;=\; O(n+m_C).$$

Proof. We ignore acknowledgements, as they do not affect the asymptotic complexities. Let us first look at the number of messages. Over every intracluster tree edge, exactly one SAFE message, one CLUSTERSAFE message, one NEIGHBORSAFE message, and one PULSE message is sent. Further, one NEIGHBORSAFE message is sent over every intercluster edge. Because there are less than n intracluster tree edges, the total message complexity therefore is at most $4n + 2m_C = O(n + m_C)$.

For the time complexity, note that the depth of each intracluster tree is at most k. On each intracluster tree, two convergecasts (the SAFE and NEIGHBORSAFE messages) and two broadcasts (the CLUSTERSAFE and PULSE messages) are performed. The time complexity for this is at most $4k$. There is one more time unit needed to send the NEIGHBORSAFE messages over the intercluster edges. The total time complexity therefore is at most $4k + 1 = O(k)$. $\square$

12.5 Network Partition

We will now look at the initialization phase of synchronizer γ. Algorithm 43 describes how to construct a partition into clusters that can be used for synchronizer γ. In Algorithm 43, $B(v,r)$ denotes the ball of radius r around v, i.e., $B(v,r) = \{u \in V : d(u,v) \le r\}$ where $d(u,v)$ is the distance between u and v. The algorithm has a parameter $\rho > 1$. The clusters are constructed sequentially. Each cluster is started at an arbitrary node that has not been included in a cluster. Then the cluster radius is grown as long as the cluster grows by a factor more than ρ.

Algorithm 43 Cluster construction

```
1: while unprocessed nodes do
2:   select an arbitrary unprocessed node $v$;
3:   $r := 0$;
4:   while $|B(v,r+1)| > \rho|B(v,r)|$ do
5:     $r := r+1$
6:   end while
7:   makeCluster($B(v,r)$)          //all nodes in $B(v,r)$ are now processed
8: end while
```

Remarks:

- The algorithm allows a trade-off between the cluster diameter k (and thus the time complexity) and the number of intercluster edges m_C (and thus the message complexity). We will quantify the possibilities in the next section.
- Two very simple partitions would be to make a cluster out of every single node or to make one big cluster that contains the whole graph. We then get synchronizers α and β as special cases of synchronizer γ.

Theorem 12.8. *Algorithm 43 computes a partition of the network graph into clusters of radius at most* $\log_\rho n$. *The number of intercluster edges is at most* $(\rho - 1) \cdot n$.

Proof. The radius of a cluster is initially 0 and does only grow as long as it grows by a factor larger than ρ. Since there are only n nodes in the graph, this can happen at most $\log_\rho n$ times.

To count the number of intercluster edges, observe that an edge can only become an intercluster edge if it connects a node at the boundary of a cluster with a node outside a cluster. Consider a cluster C of size $|C|$. We know that $C = B(v,r)$ for some $v \in V$ and $r \ge 0$. Further, we know that $|B(v,r+1)| \le \rho \cdot |B(v,r)|$. The number of nodes adjacent to cluster C is therefore at most $|B(v,r+1) \setminus B(v,r)| \le \rho \cdot |C| - |C|$. Hence, the number of intercluster edges adjacent to C is at most $(\rho - 1) \cdot |C|$. Summing over all clusters, we get that the total number of intercluster edges is at most $(\rho - 1) \cdot n$. □

Corollary 12.9. *Using* $\rho = 2$, *Algorithm 43 computes a clustering with cluster radius at most* $\log_2 n$ *and with at most* n *intercluster edges.*

Corollary 12.10. *Using $\rho = n^{1/k}$, Algorithm 43 computes a clustering with cluster radius at most k and at most $O(n^{1+1/k})$ intercluster edges.*

Remarks:

- Algorithm 43 describes a centralized construction of the partitioning of the graph. For $\rho \geq 2$, the clustering can be computed by an asynchronous distributed algorithm in time $O(n)$ with $O(m + n \log n)$ (reasonably sized) messages (showing this will be part of the exercises).
- It can be shown that the trade-off between cluster radius and number of intercluster edges of Algorithm 43 is asymptotically optimal. There are graphs for which every clustering into clusters of radius at most k requires $n^{1+c/k}$ intercluster edges for some constant c.

The above remarks lead to a complete characterization of the complexity of synchronizer γ.

Corollary 12.11. *The time and message complexities of synchronizer γ per synchronous round are*

$$T(\gamma) \;=\; O(k) \quad \textit{and} \quad M(\gamma) \;=\; O(n^{1+1/k}).$$

The time and message complexities for the initialization are

$$T_{\text{init}}(\gamma) \;=\; O(n) \quad \textit{and} \quad M_{\text{init}}(\gamma) \;=\; O(m + n \log n).$$

Remarks:

- The synchronizer idea and the synchronizers discussed in this chapter are due to Baruch Awerbuch.
- In Chapter 3, you have seen that by using flooding, there is a very simple synchronous algorithm to compute a BFS tree in time $O(D)$ with message complexity $O(m)$. If we use synchronizer γ to make this algorithm asynchronous, we get an algorithm with time complexity $O(n + D \log n)$ and message complexity $O(m + n \log n + D \cdot n)$ (including the initialization phase).
- The synchronizers α, β, and γ achieve global synchronization, i.e., every node generates every clock pulse. The disadvantage of this is that nodes that do not participate in a computation also have to participate in the synchronization. In many computations (e.g. in a BFS construction), many nodes only participate for a few synchronous rounds. An improved synchronizer due to Awerbuch and Peleg can exploit such a scenario and achieves time and message complexity $O(\log^3 n)$ per synchronous round (without initialization).
- It can be shown that if all nodes in the network need to generate all pulses, the trade-off of synchronizer γ is asymptotically optimal.

- Partitions of networks into clusters of small diameter and coverings of networks with clusters of small diameters come in many variations and have various applications in distributed computations. In particular, apart from synchronizers, algorithms for routing, the construction of sparse spanning subgraphs, distributed data structures, and even computations of local structures such as a MIS or a dominating set are based on some kind of network partitions or covers.

Chapter 13

Stabilization

A large branch of research in distributed computing deals with fault-tolerance. Being able to tolerate a considerable fraction of failing or even malisciously behaving ("Byzantine") nodes while trying to reach *consensus* (on e.g. the output of a function) among the nodes that work properly is crucial for building reliable systems. However, consensus protocols require that a majority of the nodes remains non-faulty all the time.

Can we design a distributed system that survives transient (short-lived) failures, even if *all* nodes are temporarily failing? In other words, can we build a distributed system that *repairs itself*?

13.1 Self-Stabilization

Definition 13.1 (Self-Stabilization)**.** *A distributed system is* self-stabilizing *if, starting from an arbitrary state, it is guaranteed to converge to a legitimate state. If the system is in a legitimate state, it is guaranteed to remain there, provided that no further faults happen. A state is* legitimate *if the state satisfies the specifications of the distributed system.*

Remarks:

- What kind of transient failures can we tolerate? An adversary can crash nodes, or make nodes behave Byzantine. Indeed, temporarily an adversary can do harm in even worse ways, e.g. by corrupting the volatile memory of a node (without the node noticing), or by corrupting messages on the fly (without anybody noticing). However, as all failures are transient, eventually all nodes must work correctly again, that is, crashed nodes get resurrected, Byzantine nodes stop being malicious, messages are being delivered reliably, and the memory of the nodes is secure.

- Clearly, the read only memory (ROM) must be taboo at all times for the adversary. No system can repair itself if the program code itself or constants are corrupted. The adversary can only corrupt the variables in the volatile random access memory (RAM).

Definition 13.2 (Time Complexity)**.** *The time complexity of a self-stabilizing system is the time that passed after the last (transient) failure until the system has converged to a legitimate state again, staying legitimate.*

Remarks:

- Self-stabilization enables a distributed system to recover from a transient fault regardless of its nature. A self-stabilizing system does not have to be initialized as it eventually (after convergence) will behave correctly.
- Self-stabilization was introduced in a paper by Edsger W. Dijkstra in 1974, in the context of a token ring network. A token ring is an early form of local area network where nodes are arranged in a ring, communicating by a token. The system is correct if there is exactly one token in the ring.
- Let's have a look at one of Dijkstra's simple solutions. Given an oriented ring, we simply call the clockwise neighbor parent (p), and the counterclockwise neighbor child (c). Also, there is a leader node v_0. Every node v is in a state $S(v) \in \{0, 1, \ldots, n\}$, perpetually informing its child about its state. The token is implicitly passed on by nodes switching state. Upon noticing a change of the parent state $S(p)$, node v executes the following code:

Algorithm 44 Self-stabilizing Token Ring

```
1: if $v = v_0$ then
2:    if $S(v) = S(p)$ then
3:       $S(v) := S(v) + 1 \pmod{n}$
4:    end if
5: else
6:    $S(v) := S(p)$
7: end if
```

Theorem 13.3. *Algorithm 44 stabilizes correctly.*

Proof: As long as some nodes or edges are faulty, anything can happen. In self-stabilization, we only consider the system after it is correct (at time t_0, however starting in an arbitrary state).

Every node apart from leader v_0 will always attain the state of its parent. It may happen that one node after the other will learn the current state of the leader. In this case the system stabilizes after the leader increases its state at most n time units after time t_0. It may however be that the leader increases its state even if the system is not stable, e.g. because its parent or parent's parent accidentally had the same state at time t_0.

The leader will increase its state possibly multiple times without reaching stability, however, at some point the leader will reach state s, a state that no other node had at time t_0. (Since there are n nodes and n states, this will eventually happen.) At this point the system must stabilize because the leader cannot push for $s + 1 \pmod{n}$ until every node (including its parent) has s.

After stabilization, there will always be only one node changing its state, i.e., the system remains in a legitimate state.

□

Remarks:

- For his work Dijkstra received the 2002 ACM PODC Influential Paper Award. Dijkstra passed away shortly after receiving the award. With Dijkstra being such an eminent person in distributed computing (e.g. concurrency, semaphores, mutual exclusion, deadlock, finding shortest paths in graphs, fault-tolerance, self-stabilization), the award was renamed Edsger W. Dijkstra Prize in Distributed Computing.

- Although one might think the time complexity of the algorithm is quite bad, it is asymptotically optimal.

- It can be a lot of fun designing self-stabilizing algorithms. Let us try to build a system, where the nodes organize themselves as a maximal independent set (MIS, Chapter 10):

Algorithm 45 Self-stabilizing MIS

```
Require: Node IDs
   Every node $v$ executes the following code:
1: do atomically
2:    Join MIS if no neighbor with larger ID joins MIS
3:    Send (node ID, MIS or not MIS) to all neighbors
4: end do
```

Remarks:

- Note that the main idea of Algorithm 45 is from Algorithm 35, Chapter 10.

- As long as some nodes are faulty, anything can happen: Faulty nodes may for instance decide to join the MIS, but report to their neighbors that they did not join the MIS. Similarly messages may be corrupted during transport. As soon as the system (nodes, messages) is correct, however, the system will converge to a MIS. (The arguments are the same as in Chapter 10).

- Self-stabilizing algorithms always run in an infinite loop, because transient failures can hit the system at any time. Without the infinite loop, an adversary can always corrupt the solution "after" the algorithm terminated.

- The problem of Algorithm 45 is its time complexity, which may be linear in the number of nodes. This is not very exciting. We need something better! Since Algorithm 45 was just the self-stabilizing variant of the slow MIS Algorithm 35, maybe we can hope to "self-stabilize" some of our fast algorithms from Chapter 10?

- Yes, we can! Indeed there is a general transformation that takes any local algorithm (efficient but not fault-tolerant) and turns it into a self-stabilizing algorithm, keeping the same level of efficiency and efficacy. We present the general transformation below.

Theorem 13.4 (Transformation)**.** *We are given a deterministic local algorithm $\mathcal{A}$ that computes a solution of a given problem in k synchronous communication rounds. Using our transformation, we get a self-stabilizing system with time complexity k. In other words, if the adversary does not corrupt the system for k time units, the solution is stable. In addition, if the adversary does not corrupt any node or message closer than distance k from a node u, node u will be stable.*

Proof: In the proof, we present the transformation. First, however, we need to be more formal about the deterministic local algorithm $\mathcal{A}$. In $\mathcal{A}$, each node of the network computes its decision in k phases. In phase i, node u computes its local variables according to its local variables and received messages of the earlier phases. Then node u sends its messages of phase i to its neighbors. Finally node u receives the messages of phase i from its neighbors. The set of local variables of node u in phase i is given by L_u^i. (In the very first phase, node u initializes its local variables with L_u^1.) The message sent from node u to node v in phase i is denoted by $m_{u,v}^i$. Since the algorithm $\mathcal{A}$ is deterministic, node u can compute its local variables L_u^i and messages $m_{u,*}^i$ of phase i from its state of earlier phases, by simply applying functions f_L and f_m. In particular,

$$\begin{aligned} L_u^i &= f_L(u, L_u^{i-1}, m_{*,u}^{i-1}), \text{ for } i > 1, \text{ and} \qquad (13.1)\\ m_{u,v}^i &= f_m(u, v, L_u^i), \text{ for } i \geq 1. \qquad (13.2) \end{aligned}$$

The self-stabilizing algorithm needs to simulate all the k phases of the local algorithm $\mathcal{A}$ in parallel. Each node u stores its local variables $L_u^1, \ldots, L_u^k$ as well as all messages received $m_{*,u}^1, \ldots, m_{*,u}^k$ in two tables in RAM. For simplicity, each node u also stores all the sent messages $m_{u,*}^1, \ldots, m_{u,*}^k$ in a third table. If a message or a local variable for a particular phase is unknown, the entry in the table will be marked with a special value $\perp$ ("unknown"). Initially, all entries in the table are $\perp$.

Clearly, in the self-stabilizing model, an adversary can choose to change table values at all times, and even reset these values to $\perp$. Our self-stabilizing algorithm needs to constantly work against this adversary. In particular, each node u runs these two procedures constantly:

- For all neighbors: Send each neighbor v a message containing the complete row of messages of algorithm $\mathcal{A}$, that is, send the vector $(m_{u,v}^1, \ldots, m_{u,v}^k)$ to neighbor v. Similarly, if neighbor u receives such a vector from neighbor v, then neighbor u replaces neighbor v's row in the table of incoming messages by the received vector $(m_{v,u}^1, \ldots, m_{v,u}^k)$.
- Because of the adversary, node u must constantly recompute its local variables (including the initialization) and outgoing message vectors using Functions (13.1) and (13.2) respectively.

The proof is by induction. Let $N^i(u)$ be the i-neighborhood of node u (that is, all nodes within distance i of node u). We assume that the adversary has not corrupted any node in $N^k(u)$ since time t_0. At time t_0 all nodes in $N^k(u)$ will check and correct their initialization. Following Equation (13.2), at time t_0 all nodes in $N^k(u)$ will send the correct message entry for the first round ($m_{*,*}^1$) to all neighbors. Asynchronous messages take at most 1 time unit to be received at a destination. Hence, using the induction with Equations (13.1) and (13.2) it follows that at time $t_0 + i$, all nodes in $N^{k-i}(u)$ have received the correct messages $m^1_{*,*}, \ldots, m^i_{*,*}$. Consequently, at time $t_0 + k$ node u has received all messages of local algorithm $\mathcal{A}$ correctly, and will compute the same result value as in $\mathcal{A}$. □

Remarks:

- Using our transformation (also known as "local checking"), designing self-stabilizing algorithms just turned from art to craft.

- As we have seen, many local algorithms are randomized. This brings two additional problems. Firstly, one may not exactly know how long the algorithm will take. This is not really a problem since we can simply send around all the messages needed, until the algorithm is finished. The transformation of Theorem 13.4 works also if nodes just send all messages that are not $\perp$. Secondly, we must be careful about the adversary. In particular we need to restrict the adversary such that a node can produce a reproducible sufficiently long string of random bits. This can be achieved by storing the sufficiently long string along with the program code in the read only memory (ROM). Alternatively, the algorithm might not store the random bit string in its ROM, but only the seed for a random bit generator. We need this in order to keep the adversary from reshuffling random bits until the bits become "bad", and the expected (or with high probability) efficacy or efficiency guarantees of the original local algorithm $\mathcal{A}$ cannot be guaranteed anymore.

- Since most local algorithms have only a few communication rounds, and only exchange small messages, the memory overhead of the transformation is usually bearable. In addition, information can often be compressed in a suitable way so that for many algorithms message size will remain polylogarithmic. For example, the information of the fast MIS algorithm (Algorithm 37) consists of a series of random values (one for each round), plus two boolean values per round. These boolean values represent whether the node joins the MIS, or whether a neighbor of the node joins the MIS. The order of the values tells in which round a decision is made. Indeed, the series of random bits can even be compressed just into the random seed value, and the neighbors can compute the random values of each round themselves.

- There is hope that our transformation as well gives good algorithms for mobile networks, that is for networks where the topology of the network may change. Indeed, for deterministic local approximation algorithms, this is true: If the adversary does not change the topology of a node's k-neighborhood in time k, the solution will locally be stable again.

- For randomized local approximation algorithms however, this is not that simple. Assume for example, that we have a randomized local algorithm for the dominating set problem. An adversary can constantly switch the topology of the network, until it finds a topology for which the random bits (which are not really random because these random bits are in ROM) give a solution with a bad approximation ratio. By defining a weaker adversarial model, we can fix this problem. Essentially, the adversary needs to be oblivious, in the sense that it cannot see the solution. Then it will not be possible for the adversary to restart the random computation if the solution is "too good".

- Self-stabilization is the original approach, and self-organization may be the general theme, but new buzzwords pop up every now and then, e.g. self-configuration, self-management, self-regulation, self-repairing, self-healing, self-optimization, self-adaptivity, or self-protection. Generally all these are summarized as "self-*". One computing giant coined the term "autonomic computing" to reflect the trend of self-managing distributed systems.

13.2 Advanced Stabilization

We finish the chapter with a non-trivial example beyond self-stabilization, showing the beauty and potential of the area: In a small town, every evening each citizen calls all his (or her) friends, asking them whether they will vote for the Democratic or the Republican party at the next election.[1] In our town citizens listen to their friends, and everybody re-chooses his or her affiliation according to the majority of friends.[2] Is this process going to "stabilize" (in one way or another)?

Remarks:

- Is eventually everybody voting for the same party? No.
- Will each citizen eventually stay with the same party? No.
- Will citizens that stayed with the same party for some time, stay with that party forever? No.
- And if their friends also constantly root for the same party? No.
- Will this beast stabilize at all?!? Yes!

Theorem 13.5 (Dems & Reps)**.** *Eventually every citizen is rooting for the same party every other day.*

Proof: To prove that the opinions eventually become fixed or cycle every other day, think of each friendship between citizens as a pair of (directed) edges, one in each direction. Let us say an edge is currently "bad" if the party of the *advising* friend differs from the next-day's party of the *advised* friend. In other words, the edge is bad if the advisor was in the minority. An edge that is not bad, is "good".

Consider the out-edges of citizen c on day t, during which (say) c roots for the Democrats. Assume that during day t, g out-edges of c are good, and b out-edges are bad. Note that $g + b$ is the degree of c. Since g out-edges were good, g friends of c root for the Democrats on day $t+1$. Likewise, b friends of c root for the Republicans on day $t+1$. In other words, on the evening of day $t+1$

[1]We are in the US, and as we know from The Simpsons, you "throw your vote away" if you vote for somebody else. As a consequence our example has two parties only.

[2]Assume for the sake of simplicity that everybody has an odd number of friends.

citizen c will receive g recommendations for Democrats, and b for Republicans. We distinguish two cases:

- $g > b$: In this case, citizen c will still (or again) root for the Democrats on day $t + 2$. Note that in this case, on day $t + 1$, exactly g in-edges of c are good, and exactly b in-edges are bad. In other words, the number of bad out-edges on day t is exactly the number of bad in-edges on day $t + 1$.

- $g < b$: In this case, citizen c will root for the Republicans on day $t + 2$. Note that in this case, on day $t + 1$, exactly b in-edges of c are good, and exactly g in-edges are bad. In other words, the number of bad out-edges on day t was exactly the number of good in-edges on day $t + 1$ (and vice versa). Since citizen c is rooting for the Republicans, the number of bad out-edges on day t was strictly larger than the number of bad in-edges on day $t + 1$.

We account for every edge as out-edge on day t, and as in-edge on day $t + 1$. Since in both of the above cases the number of bad edges do not increase, the total number of bad edges B cannot increase. In fact, if any node switches its party from day t to $t + 2$, we know that the total number of bad edges strictly decreases. But B cannot decrease forever. Once B hits its minimum, the system stabilizes in the sense that every citizen will either stick with his or her party forever or flip-flop every day – the system "stabilizes". □

Remarks:

- The model can be generalized considerably by, for example, adding weights to vertices (meaning some citizens' opinions are more important than others), allowing loops (citizens who consider their own current opinions as well), allowing tie-breaking mechanisms, and even allowing different thresholds for party changes.

- How long does it take until the system stabilizes?

- Some of you may be reminded of Conway's Game of Life: We are given an infinite two-dimensional grid of cells, each of which is in one of two possible states, *dead* or *alive*. Every cell interacts with its eight neighbors. In each round, the following transitions occur: Any live cell with fewer than two live neighbors dies, as if caused by lonelyness. Any live cell with more than three live neighbors dies, as if by overcrowding. Any live cell with two or three live neighbors lives on to the next generation. Any dead cell with exactly three live neighbors is "born" and becomes a live cell. The initial pattern constitutes the "seed" of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed, births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations. John Conway figured that these rules were enough to generate interesting situations, including "breeders" with create "guns" which in turn create "gliders". As such Life in some sense answers an old question by John von Neumann, whether there can be a simple machine that can build copies of itself. In fact Life is Turing complete, that is, as powerful as any computer.

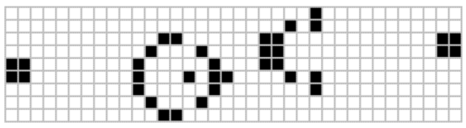


Figure 13.1: A “glider gun”. . .

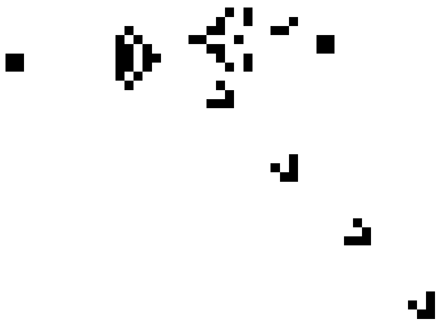


Figure 13.2: . . . in action.