Contents

Lecture 1

Vertex Coloring

1.1 The Problem

Nowadays multi-core computers get more and more processors, and the question is how to handle all this parallelism well. So, here's a basic problem: Consider a doubly linked list that is shared by many processors. It supports insertions and deletions, and there are simple operations like summing up the size of the entries that should be done very fast. We decide to organize the data structure as an array of dynamic length, where each array index may or may not hold an entry. Each entry consists of the array indices of the next entry and the previous entry in the list, some basic information about the entry (e.g. its size), and a pointer to the lion's share of the data, which can be anywhere in the memory.

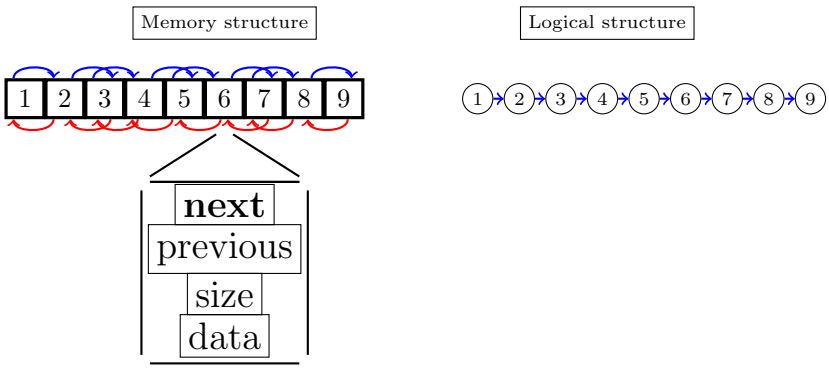

Figure 1.1: Linked listed after initialization. Blue links are forward pointers, red links backward pointers (these are omitted from now on).

We now can quickly determine the total size by reading the array in one go from memory, which is quite fast. However, how can we do insertions and deletions fast? These are *local* operations affecting only one list entry and the pointers of its "neighbors," i.e., the previous and next list element. We want to be able to do many such operations concurrently, by different processors, while maintaining the link structure! Being careless and letting each processor act independently invites desaster, see Figure 1.3.

Figure 1.2: State after many insertion and deletion operations. There may also be "dead" cells which are currently not part of the list. These are not shown; we assume that these are taken care of every now and then to avoid wasting too much memory.

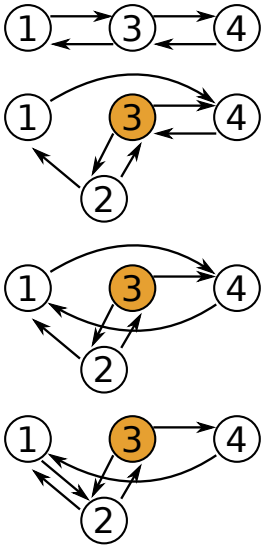

Figure 1.3: Concurrent insertion of a new entry 2 between 1 and 3, and (logical) deletion of entry 3. The deletion of 3 requires to change the successor and predecessor of 1 and 4, but the insertion of 2 changes the successor of 1 as well. Not doing this in a consistent order messes up the list. Note that entry 3 might get physically deleted as well, rendering many of our pointers invalid.

On the other hand, *any* set of concurrent modifications that does *not* involve neighbors is fine: The result is a neat doubly linked list. Clearly, we want to be able to manipulate arbitrary list entries. This can be rephrased as an (in)famous graph problem.

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: $e = \{v, w\} \in E \Rightarrow c_v \neq c_w$.*

We then can "cycle" through the colors and perform concurrent operations on all "nodes" (a.k.a. list entries) of the same color without worrying. Once we're done with all colors, we color the new list, and so on. We now have a challenging task:

- We want to use very few colors, so cycling through them is completed quickly. Coloring with a minimal number of colors is in general very hard,

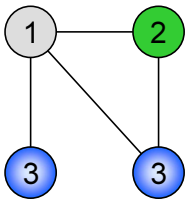

Figure 1.4: 3-colorable graph with a valid coloring.

but fortunately we're dealing with a very simple graph.

- The coloring itself needs to be done fast, too. Otherwise we'll be waiting for the new coloring to be ready all the time.

- That means we want to use all our processors. It's easy to split up responsibility for the list entries by splitting up the array. The downside of this is that the processors receive only fragmented parts of the list (see Figure 1.5).

- Trying to get consecutive pieces under the control of a single processor requires to *break symmetry:* List fragments get longer only if more nodes are added than removed. If the list is fragmented into single nodes, this roughly means that we want to find a *maximal independent set*, i.e., a set containing no neighbors to which we cannot add a node without destroying this property. This turns out to be essentially the same problem as coloring, as we will see in the exercises.

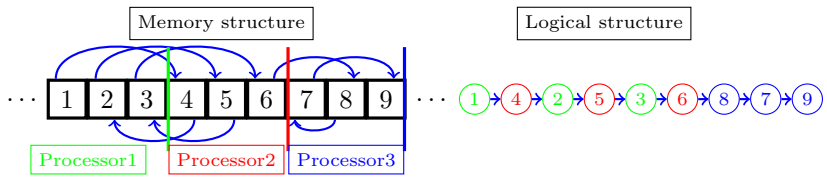

Figure 1.5: List split. We may get lucky in some places of the list (as for the blue processor), but in wide parts the list will be fragmented between processes.

As the list is fragmented among the processors anyway, it's useful to pretend that we have as many processors as we want. That means each of the nodes can have its "own" processor! If we can deal with this case efficiently, it will certainly work out with fewer processors! Oh, and one more thing: We have some additional information we can glean from the setup. Each node has a *unique identifier* associated with it, namely its array index. Note that this means nodes already "look different" initially, which is crucial for coloring deterministically without starting from the endpoints only.

Remarks:

- In distributed computing, we often take the point of view that the system is a graph whose nodes are processors and whose edges are both representing relations with respect to the problem at hand *and* communication links. This will come in handy here as an abstraction, but in many systems it is literally true.

- The assumption of unique identifiers is standard, the reason being that deterministic distributed algorithms can't even do basic things without them (for instance coloring a list quickly). On the other hand, using randomization it's trivial to generate unique identifiers with overwhelming probability. Nonetheless, it is also studied how important such identifiers actually are; more about that in another lecture!

- The linked list here is a toy example, but an entire branch of distributed computing is occupied with finding efficient data structures for *shared memory* systems like the one informally described above. We'll have another look at such systems further into the course!

1.2 2-Coloring the List

Clearly, we can color the list with two colors, simply by passing through the list and alternating. Since this is sequential, i.e., only one process is actually working, it takes $\Theta(n)$ steps in a list of n nodes. We can parallelize this strategy, however. For $i \in [n] := \{0, \dots, n-1\}$, denote by v_i the array index of the i^{th} node in the list. As always in this course, log denotes the base-2 logarithm. First, we add "shortcuts" to our linked list.

Algorithm 1 Parallel pointer jumping

```
1: for j = 0, ..., ⌈log n⌉ − 1 do
2:   for each node v_i in parallel do
3:     if i − 2^j ≥ 0 and i + 2^j < n then
4:       {have node v_i create shortcuts between v_{i−2^j} and v_{i+2^j}}
5:       store a pointer to v_{i−2^j} at element v_{i+2^j}
6:       store a pointer to v_{i+2^j} at element v_{i−2^j}
7:     end if
8:   end for
9: end for
```

Here we assume that processes share a common clock to coordinate the execution of the outer loop, or somehow simulate this behavior. We'll examine this issue more closely in the next lecture; let's assume for now that we can handle this and call each iteration of the outer loop a *round*.

Let's have a closer look at what this algorithm does.

Lemma 1.2 (Shortcuts from pointer jumping)**.** *After r rounds of Algorithm 1, at each array index v_i with $i \geq 2^r$ the index v_{i-2^r} is stored. Likewise, at each index v_i with $i < n - 2^r$, v_{i+2^r} is stored.*

Proof. We show the claim by induction. The base case is $r = 0$, i.e., the initial state. As $2^0 = 1$, the statement is just another way of saying that we have a doubly linked list, so we're in the clear. Now assume that the claim is true for some $0 \leq r < \lceil \log n \rceil$. In the r^{th} round, we have $j = r - 1$. For any $i \geq 2^r$, (the process responsible for) node $v_{i-2^{r-1}}$ will add v_{i-2^r} to the entry at array index v_i. It can do this, because by induction hypothesis v_{i-2^r} and v_i can be looked up at the array index $v_{i-2^{r-1}}$. Note that processes dealing with a v_i with $i < 2^{r-1}$ will not get confused: they will know that $i < 2^{r-1}$ because $v_{i-2^{r-1}}$ was not added to the entry corresponding to v_i. Similarly, for each $i < n - 2^r$, $v_{i+2^{r-1}}$ will add v_{i+2^r} to the entry at index v_i. □

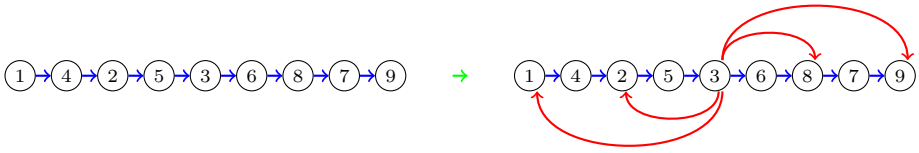

Figure 1.6: Parallel pointer jumping. Depicted are the additional pointers/links of node 3 only.

With these shortcuts, we can color the list quickly.

Algorithm 2 2-coloring the list

```
1: execute Algorithm 1
2: color the list head by 0 (i.e., index v_0)
3: for j = ⌈log n⌉ − 1, ..., 1 do
4:    for each node v_i colored 0 in parallel do
5:       if i + 2^j < n then
6:          color index v_{i+2^j} by 0
7:       end if
8:    end for
9: end for
10: color all remaining nodes by 1
```

Theorem 1.3 (Correctness of Algorithm 2)**.** *Algorithm 2 colors the doubly linked list with* 2 *colors.*

Proof. From Lemma 1.2, we know that array elements will store the necessary information to execute the for-loops. By induction, we see that after r rounds of the loop, all nodes v_i with $i \mod 2^{\lceil \log n \rceil - r} = 0$ are colored 0. The loop runs for $\lceil \log n \rceil - 1$ rounds, i.e., until $j = 1$. Thus, all nodes in even distance from the list head are colored 0, while the remaining nodes get colored 1. □

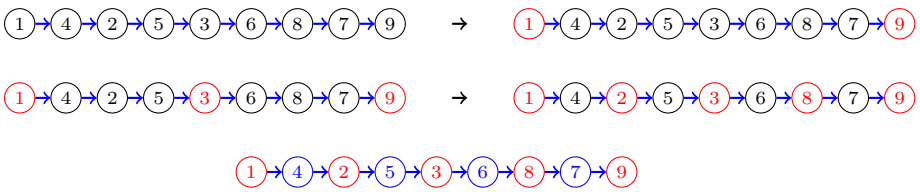

Figure 1.7: Execution of the 2-coloring algorithm. Each step uses a different "level" of pointers constructed with the pointer jumping algorithm; the final steps just uses the neighbor pointers.

Remarks:

- In the above algorithms, we referred to n. However, n is unknown due to parallel insertions and deletions (maintaining a shared counter is another fundamental problem!). This can be resolved by letting v_i *terminate* when it knows that its work is done, which is the case when not both v_{i-2^j} and v_{i+2^j} are written to v_i in round j. The processors then just need to notify each other once all their associated nodes are terminated.

- For 2-coloring, the $\mathcal{O}(\log n)$ rounds of this algorithm are the best we can get: another straightforward induction shows that following pointers, it takes $\lceil \log h \rceil$ rounds to "see" something that is h "hops" in the list away, and unless individual processors read large chunks of memory, this has to be done.

- The issue is that 2-coloring is too rigid. Once we color a single node, all other nodes' colors are determined. The problem is not *local*.

- This is also bad for another reason: if we have only small changes in the list, we would like to avoid having to recolor it from scratch. It would be nice to have an algorithm where the output depends only on a small number of hops around each node. This would most likely also yield a fast and efficient algorithm!

- We can use the pointer jumping technique to speed up algorithms that are more local in this sense: if in round r nodes write everything they know to the array entries of nodes in distance 2^{r-1}, it takes only $\lceil \log h \rceil$ steps until the output of an algorithm depending on nodes in distance at most h can be determined. However, this is only practical if h is small, as otherwise a lot of work is done!

1.3 Using 3 Colors

What good does it do to get down to two colors, but at a large overhead? None, as we have to do it again after each change of the list. Let's be a bit more relaxed and permit $c > 2$ colors. This means that, no matter what the neighbors' colors are, there's always a free one to pick! Given that we start with a valid coloring – the array indices – we can use this to reduce the number of colors to 3. Let's assume in the following that $v_{-1} = v_{n-1}$ and $v_n = v_0$ (i.e.,

head and tail of the list also have pointers to each other), since this will simplify describing algorithms.

Algorithm 3 color reduction

```
1: for each node v_i in parallel do
2:    c_{v_i} := v_i
3: end for
4: while ∃v_i : c_{v_i} > 2 do
5:    for each node v_i with c_{v_i} > max{c_{v_{i-1}}, c_{v_{i+1}}, 2} in parallel do
6:       c_{v_i} := min([3] \ {c_{v_{i-1}}, c_{v_{i+1}}})
7:    end for
8: end while
```

Lemma 1.4. *Algorithm 3 computes a* 3*-coloring. It terminates in* c *rounds, where* c *is the number of different colors in the initial coloring (here* n*, because the array indices are unique).*

Proof. No two neighbors can change their color in the same round, as this would require that each of their colors is larger than the other. Thus, the coloring is valid after each round (given that it was valid initially). A node with the current maximum color will change its color (because no neighbor can have this color, too). Note also that no colors other than 0, 1, or 2 are ever picked by a node. It follows that the algorithm completes after at most c rounds and the result is a valid 3-coloring. □

Remarks:

- A time complexity of (almost) n rounds is attained if we still have a nice, well-ordered list, i.e., $v_i = i$ for all i. In other words, if we're unlucky, we color the list sequentially.
- The algorithm is, however, good to reduce the number of colors to 3 if we only have a few colors to begin with.
- If we have an arbitrary graph of *maximum degree* Δ (i.e., no node has more than Δ neighbors), the same approach can be used to find a $(\Delta+1)$-coloring (see Figure 1.8).
- It's not hard to show that if the initial coloring is random, the algorithm will finish in $\Theta(\log n / \log\log n)$ rounds with a very large probability. Can you prove it?
- One can construct such an initial coloring by picking colors randomly at each node from a sufficiently large range.
- Combining with pointer jumping, we get a running time of $\Theta(\log\log n)$ for 3-coloring, exponentially faster than for 2-coloring!

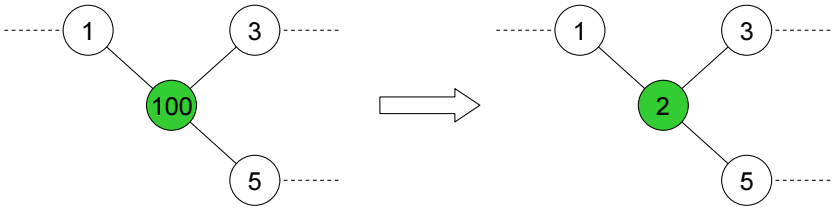

Figure 1.8: Vertex 100 receives the lowest possible color.

1.4 Cole-Vishkin

The previous algorithm reduced the number of colors in each step, starting from a valid coloring. We can now ask: Can this be done more quickly/efficiently? The answer turns out to be yes, as shown by the following algorithm, which is based on a simple, but ingenious idea.

Algorithm 4 Cole-Vishkin color reduction

1: **for** each node v_i in parallel **do**
2: $c_{v_i} := v_i$
3: **end for**
4: **while** $\exists v_i : c_{v_i} > 5$ for all nodes in parallel **do**
5: interpret c_{v_i} and $c_{v_{i-1}}$ as (infinite) little-endian bit-strings, i.e., starting with the least significant bit
6: let j be the smallest index where they differ
7: concatenate j (as bitstring) and the differing bit itself, yielding color c
8: $c_{v_i} := c$
9: **end while**

Example:

Part of an execution of Algorithm 4:

v_{i-2}	0010110000	$\rightarrow$	...	$\rightarrow$	...
v_{i-1}	1010010000	$\rightarrow$	01010	$\rightarrow$	...
v_i	0110010000	$\rightarrow$	10001	$\rightarrow$	0001

The trick is that either the first or the second part of (the bit string of) the new color saves the day.

Lemma 1.5 (Correctness of Cole-Vishkin)**.** *Algorithm 4 computes a valid coloring.*

Proof. Since the initial coloring is valid, we need to show that a valid coloring enables to compute the new colors and the new coloring is valid. The first part readily follows from the fact that two different colors must have differing bit strings, so the index j can be computed. Now consider two neighbors v_i and v_{i-1}. If they determine different indices j for which the current colors differ from v_{i-1} and v_{i-2} respectively, the front part of the new colors is different. Otherwise, the second part of their new colors consists precisely of the least significant *differing* bit! □

This algorithm terminates in (almost) $\log^* n$ time. Log-Star is the *number* of times one needs to take the logarithm (to the base 2) to get to at most 1, starting with n:

Definition 1.6 (Log-Star)**.**
$\forall x \leq 1 : \ \log^* x := 0 \quad \forall x > 1 : \ \log^* x := 1 + \log^*(\log x)$

Theorem 1.7. *Algorithm 4 computes a valid* 6*-coloring in* $\log^* n + \mathcal{O}(1)$ *rounds.*

Proof. Correctness is shown in Lemma 1.5. The time complexity follows from the fact that if the original color had b bits, the new color has at most $\lceil \log b \rceil + 1$ bits: the number of bits to encode an index in a b-bit string plus the appended bit. The $\mathcal{O}(1)$ term addresses the fact that we don't actually apply the base-2 logarithm in each step. (The non-exciting computations showing that this makes only a minor difference are omitted.) The reason why we end up with 6 colors is simple: encoding an index of a 3-bit value yields 00, 01, or 10 as leading parts; appending a bit yields 6 possibilities. It's simple to check that for any larger number of initial colors, fewer possibilities will remain. □

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. Hence, for all practical purposes, it's constant.
- One can use Algorithm 3 to reduce the number of colors to 3 in 3 rounds.
- As stated, the algorithm has a termination condition that cannot be *checked* efficiently based on local information. Fortunately, we can just get rid of this condition and run the algorithm for the right number of rounds given by Theorem 1.7.
- This does not work if n is unknown. This issue has two different solutions: a practical one and a theoretical one. Can you figure out both?
- For a change, the $\mathcal{O}(1)$ term is actually hiding only a *small* constant. The time complexity of the problem has been nailed down to be precisely $1/2 \cdot \log^* n$ for infinitely many values of n [RS15].
- Another detail here is that instead of n, the argument of the $\log^*$ is, in fact, the initial *range* of colors. In our case, this is the current size of the array, which may be larger than n, typically by some constant factor. However, even if it would be exponentially larger, this would mean we need to do just one or two more rounds of the algorithm to handle this.
- A simple modification results in running time $1/2 \cdot \log^* n + \mathcal{O}(1)$ (see exercises).
- Using pointer jumping, the running time can be reduced to $\log(\log^* n) + \mathcal{O}(1)$. Shockingly, this is *not* the most ridiculously slow-growing function I've encountered in a statement that is not deliberately about slow-growing functions.
- The technique is not limited to lists. It can be used to color oriented trees and constant-degree graphs in $\mathcal{O}(\log^* n)$ rounds, too (see exercises).

1.5 Linial's Lower Bound

If we can color the list *that* fast, can't we find an algorithm that does it in truly constant time? The answer is no, and we're going to see now why. We'll focus on the case where interactions are solely with neighbors, in which one requires $\Omega(\log^* n)$ rounds. Such algorithms are called *message-passing algorithms*, for reasons that will be discussed in the next lecture. With shared memory, the variant of Cole-Vishkin with pointer jumping is asymptotically optimal [FR90]. We also restrict to deterministic algorithms.

Before we do the proof, let's simplify the situation a bit. First, observe that all information the output of v_i can be influenced by in a T-round message passing algorithm is the information that's initially available at nodes $v_{i-T}, v_{i-T+1}, \ldots, v_{i+T}$. In the worst case, every content stored is identical,[1] so the only real difference are the actual array indices (and memory addresses). Note also that the order is relevant: We have forward and backward pointers, i.e., we can distinguish directions, and obviously it's possible to count the number of "hops" traversed. Consequently, even if we don't know anything about a coloring algorithm except that it is a deterministic T-round algorithm $\mathcal{A}$ (with neighbor-neighbor interactions only), we can conclude that there is a function

$$f\colon (x_0, \ldots, x_{2T}) \to [c]$$

so that $c_{v_i} = f(v_{i-T}, v_{i-T+1}, \ldots, v_{i+T})$ when executing $\mathcal{A}$. Here, c is the number of colors used by $\mathcal{A}$ and we assume *without loss of generality* (w.l.o.g.) that $[c]$ is the set of colors produced by the algorithm.[2]

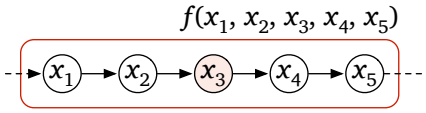

Figure 1.9: Interpreting a 2-round coloring algorithm as a coloring function f mapping 5-tuples to colors.

If $\mathcal{A}$ produces a valid coloring, we also know that

$$f(x_0, \ldots, x_{2T}) \neq f(x_1, \ldots, x_{2T+1})$$

provided that $x_i \neq x_j$ for $i \neq j$, $i, j \in [2T+2]$: The two arguments could be the views of adjacent nodes in the list, and they must not compute the same color.

Now comes the clever bit making our lives much easier: We restrict the problem without actually taking away what makes it hard. This will simplify our key argument, as it has an algorithmic component – and it would be more challenging to come up with an algorithm for the more general setting. For $c, k \in \mathbb{N}$, we say that g is a *k-ary c-coloring function* if

$$\forall 0 \leq x_1 < x_2 < \ldots < x_k < n\colon\ g(x_1, x_2, \ldots, x_k) \in [c]$$

[1] Even if that wasn't true, the same argument applies taking this content into account.
[2] As opposed to, e.g., {pink, elephant, turtle}.

and

$$\forall 0 \leq x_1 < x_2 < \ldots < x_{k+1} < n : \; g(x_1, x_2, \ldots, x_k) \neq g(x_2, x_3, \ldots, x_{k+1}).$$

For $k = 2T+1$, these are the exact same requirements as to f, however, only for ascending addresses $x_1 < \ldots < x_{k+1} < n$. Note that by restricting the domain of f to such inputs, we see that the existence of a T-round algorithm $\mathcal{A}$ using c colors implies the existence of a $(2T+1)$-ary c-coloring function f.

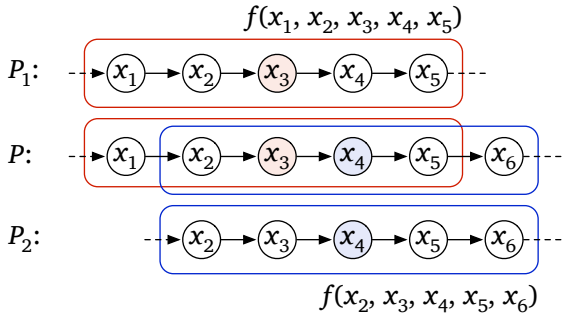

Figure 1.10: 5-tuples that correspond to possible views of adjacent nodes must result in different colors.

Using this connection, we can now move on to the proof of the lower bound, which consists of showing that if c is small, then T cannot be arbitrarily small, too.

Lemma 1.8 (1-ary functions require many colors)**.** *If f is a 1-ary c-coloring function, then $c \geq n$.*

Proof. By definition, $f(x_1) \neq f(x_2)$ for all $0 \leq x_1 < x_2 < n$, i.e.,

$$\forall x_1 \neq x_2 \in [n] : \; x_1 \neq x_2 \Leftrightarrow f(x_1) \neq f(x_2).$$

In other words, f is an injection, which is only possible if $c \geq n$. □

The main step of the proof is to show that we can construct $(k-1)$-ary 2^c-coloring functions out of k-ary c-coloring functions. That is, we can "pay" for saving time by using more colors.

Lemma 1.9 (k-ary c-coloring enables $(k-1)$-ary 2^c-coloring)**.** *If f is a k-ary c-coloring function for some $k > 0$, then a $(k-1)$-ary 2^c-coloring function g exists.*

Proof. First, let h be a bijection from the subsets of $[c]$ to $[2^c]$. Concretely, we may choose for $S \subseteq [c]$ as $h(S)$ the string of c bits in which the i^{th} bit is 1 if and only if $i - 1 \in S$ (but any other bijection would do, too).

Next, define

$$g'(x_1, \ldots, x_{k-1}) := \{f(x_1, \ldots, x_k) \,|\, x_{k-1} < x_k < n\},$$

i.e., g' is the *set* of all colors that can possibly be assigned by f when all but the last argument of f are specified. These are the colors that might cause trouble

when g assigns a color to $x_1, \ldots, x_{k-1}$ without considering x_k. Using h, we can interpret this set as a new color:[3]

$$g(x_1, \ldots, x_{k-1}) := h \circ g'(x_1, \ldots, x_{k-1}) = h(g'(x_1, \ldots, x_{k-1})).$$

It's straightforward to check that this is a well-defined function with range $[2^c]$: $g'(x_1, \ldots, x_{k-1}) \subseteq [c]$ and h maps such sets to a color from $[2^c]$. In order to verify that g is indeed a $(k-1)$-ary 2^c-coloring function, we thus must show that

$$\forall 0 \leq x_1 < x_2 < \ldots, x_k < n : \; g(x_1, \ldots, x_{k-1}) \neq g(x_2, \ldots, x_k).$$

Let $0 \leq x_1 < x_2 < \ldots < x_k < n$. Clearly, $f(x_1, \ldots, x_k) \in g'(x_1, \ldots, x_{k-1})$. On the other hand, we have that $f(x_1, \ldots, x_k) \neq f(x_2, \ldots, x_{k+1})$ for any $x_k < x_{k+1} < n$, because f is a coloring function. This is equivalent to saying that $f(x_1, \ldots, x_k) \notin g'(x_2, \ldots, x_k)$. We conclude that $g'(x_1, \ldots, x_{k-1}) \neq g'(x_2, \ldots, x_k)$. Since h is a bijection, this is equivalent to

$$g(x_1, \ldots, x_{k-1}) = h(g'(x_1, \ldots, x_{k-1})) \neq h(g'(x_2, \ldots, x_k)) = g(x_2, \ldots, x_k). \quad \square$$

With these lemmas, it's a piece of cake to obtain the lower bound.

Theorem 1.10 (Linial's lower bound)**.** *Coloring a list with a message passing algorithm that uses (at most) 4 colors requires at least* $1/2 \cdot \log^* n - 1$ *rounds.*

Proof. Assume that $\mathcal{A}$ is a T-round coloring algorithm using 4 colors. Thus, a $(2T+1)$-ary 4-coloring function exists. We apply Lemma 1.9 for $2T$ times, to see that then a 1-ary $\left({}^{2T}2\right)^4$-coloring function exists. Here, ${}^a2$ denotes the tetration or "power tower," the a-fold iterated exponentiation by 2. From Lemma 1.8, we know that

$${}^{2T+2}2 = \left({}^{2T}2\right)^4 \geq n,$$

yielding

$$2T + 2 \geq \log^* n$$

and finally

$$T \geq \frac{\log^* n}{2} - 1. \qquad \square$$

Remarks:

- More colors don't help a lot. If we consider c colors in the above proof, we get that it requires at least $1/2 \cdot (\log^* n - \log^* c)$ rounds to color with c colors.
- Randomization doesn't help either. Naor extended the lower bound to randomized algorithms [Nao91].
- I've been a bit sloppy, as I haven't defined the model precisely. This can easily lead to mistakes, so I will make amends in the next lecture. The given proof works in the so-called *message passing* model, which we get to know in more detail in the next lecture.
- If one permits non-neighbor interactions, the lower bound weakens to $\lceil \log(1/2 \cdot (\log^* n - \log^* c)) \rceil$ [FR90], just like we could speed up the Cole-Vishkin algorithm using pointer jumping.

[3] Note that h doesn't really do anything but "rename" the sets such that they are easy to count. That's why, rather than turtles or sets, we like our colors to be numbers!

What to take Home

- Exploiting parallelism, distributed algorithms can be extremely fast.
- *Symmetry breaking* is a fundamental challenge in distributed computing, and a coloring is a basic structure that breaks symmetry between neighbors.
- The key to understanding parallelism is to understand what is possible based on limited (in particular local) information.
- What can and can't be done is quite sensitive to the model. When considering running time bounds, impossibility results, etc. it is thus important to keep in mind that changing an aspect of the model may have a dramatic impact. Try always to understand what aspects of a model cause a certain result, and wonder whether changing them would change the game!
- On the other hand, we can frequently prove *unconditional* lower bounds in distributed computing, such as Theorem 1.7. If we *do* figure out what the suitable model of computation is for a given system, we may be able to understand precisely how fast things can be done. Contrast this with lower bounds on sorting (which restrict the feasible operations) or impossibilities in the sequential world that rest on conjectures like P $\neq$ NP or the unique games conjecture!
- Math is going to be our friend in this lecture. If your reflex is to disagree, try to imagine figuring out how fast the list can be colored by concurrent processes without the tools we used. Moreover, coming up with a proof requires us to reflect on our assummptions and crystallize ideas; that's difficult, but *very* useful when dealing with more complex problems later on!

Bibliographic Notes

The basic technique of the log-star algorithm is by Cole and Vishkin [CV86]. The technique can be generalized and extended, e.g., to a ring topology or to graphs with constant degree [GP87, GPS88, KMW05]. Using it as a subroutine, one can solve many problems in log-star time.

The lower bound of Theorem 1.7 is due to Linial [Lin92]. Linial's paper also contains a number of other results on coloring, e.g., that any message passing algorithm for coloring d-regular trees of radius r that runs in time at most $2r/3$ requires at least $\Omega(\sqrt{d})$ colors. The presentation here is based on a more streamlined version by Laurinharju and Suomela [LS14].

Figures 1.9 and 1.10 are courtesy of Jukka Suomela and under a creative commons license.[4] Figures 1.4 and 1.8 are courtesy of Roger Wattenhofer; substantial parts of today's lecture are based on material from his course at ETH Zurich. Wide parts of today's lecture are covered by books [CLR90, Pel00].

[4]CC BY-SA 3.0, see HTTPS://CREATIVECOMMONS.ORG/LICENSES/BY-SA/3.0/.

Bibliography

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *18th annual ACM Symposium on Theory of Computing (STOC)*, 1986.

[FR90] Faith E. Fich and Vijaya Ramachandran. Lower bounds for parallel computation on linked structures. In *Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 1990)*, pages 109–116, 1990.

[GP87] Andrew V. Goldberg and Serge A. Plotkin. Parallel (Δ+1)-coloring of constant-degree graphs. *Inf. Process. Lett.*, 25(4):241–245, June 1987.

[GPS88] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel Symmetry-Breaking in Sparse Graphs. *SIAM J. Discrete Math.*, 1(4):434–446, 1988.

[KMW05] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. On the Locality of Bounded Growth. In *24th ACM Symposium on the Principles of Distributed Computing (PODC), Las Vegas, Nevada, USA*, July 2005.

[Lin92] N. Linial. Locality in Distributed Graph Algorithms. *SIAM Journal on Computing*, 21(1)(1):193–201, February 1992.

[LS14] Juhana Laurinharju and Jukka Suomela. Brief Announcement: Linial's Lower Bound Made Easy. In *Symposium on Principles of Distributed Computing (PODC)*, pages 377–378, 2014.

[Nao91] Moni Naor. A Lower Bound on Probabilistic Algorithms for Distributive Ring Coloring. *SIAM J. Discrete Math.*, 4(3):409–412, 1991.

[Pel00] David Peleg. *Distributed computing: a locality-sensitive approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[RS15] Joel Rybicki and Jukka Suomela. Exact bounds for distributed graph colouring. *CoRR*, abs/1502.04963, 2015.

Lecture 2

Synchronizers

In the previous lecture, we have seen that results may vary quite a bit depending on the model. Today, we study one very important distributed model in more detail. We will see that it can be a good approach to first figure out how to *simulate* a more powerful model before attempting to solve a difficult problem.

2.1 Synchronous Message Passing

As promised in the previous lecture, we'll be more precise about the considered model(s) today. If you expect that this means tedious definitions, that's not actually true. Many useful models are useful *because* they are simple, implying that understanding them means to learn something about a wide variety of systems. This is also the case for the following well-studied model of distributed computing.

Definition 2.1 (The LOCAL model)**.** *The network is modeled as a simple*[1] *graph $G = (V, E)$ of n nodes. Each node has a unique identifier of $\mathcal{O}(\log n)$ bits. An algorithm is executed in* synchronous rounds*, where in each round, each node (a.k.a. processor) takes the following steps:*

1. *Do some local computations.*
2. *Send messages to its neighbors in the graph G.*
3. *Receive messages (that were sent by neighbors in step 2 of the same round).*

In addition, nodes may determine a (local) output and terminate at the end of a round. The time complexity *of a synchronous message passing algorithm is the time until all nodes have terminated. Finally, some problems will also have some additional input information, e.g., edge weights. Nodes will then initially know the local part of the input, e.g., the weight of incident edges.*

Clearly, this model is questionable for many reasons:

- We do not put restrictions on local computations or memory, which means that nodes could solve NP-hard problems locally! Some authors consider this cheating and require computations polynomial in n.

[1] Meaning: undirected, unweighted, loop-free, and without parallel edges.

- Nodes can send messages of arbitrary size. In particular, they simply can send everything they know to all neighbors in each round. This can easily result in unrealistically large messages of size $\Omega(n^2)$, e.g., in a complete graph.

- We assume perfectly synchronous execution, but many practical systems simply cannot operate this way or, if forced to do so, would have extremely long rounds (because one needs to wait for the slowest computations to finish and all messages to arrive before moving on to the next round).

So, why is the LOCAL model useful at all? First of all, it was originally designed for proving *lower bounds*, just like the one given in Theorem 1.7. This makes the statement only more powerful: Even if all the above crazy things were possible, it would *still* take at least $1/2 \cdot \log^* n - \mathcal{O}(1)$ rounds to color the list with few colors. Such results are surprisingly useful, simply because they tell us what we *shouldn't* spend our time on trying.

Second, algorithms designed in the LOCAL model frequently turn out to use very little computations and memory, as well as small messages. The Cole-Vishkin coloring algorithm is a prime example for this. After all, only so much can be done with little information! Hence, it's not a bad approach to design a (fast) algorithm in this model and worry about things like message size later.

Finally, even if the designed algorithms are fast (in terms of the number of rounds), but use large messages, we can either try to get rid of the "cumbersome" aspects of the algorithm or know that an algorithm being slow must be caused by a limitation in bandwith or computation.

2.2 Asynchronous Message Passing

Still, all that doesn't address the issue of synchrony, which, to put it mildly, turns out to be more of a problem. Consequently, there is a "sister" to the LOCAL model, the asynchronous message passing model.[2]

Definition 2.2 (The asynchronous message passing model)**.** *The network is modeled as a simple graph $G = (V, E)$ of n nodes. Each node has a unique identifier of $\mathcal{O}(\log n)$ bits. An algorithm is executed based on* events*. An* event at node $v \in V$ *is either the node starting to execute the algorithm or receiving a message from a neighbor (nodes start the algorithm at the latest on reception of the first message). Upon an event at node v, v does the following:*

1. *It does some local computations.*

2. *It may send messages to its neighbors in the graph G.*

Each sent message will eventually be received, but it is completely arbitrary which of the messages in transit will arive next. Just like before, a node may also determine a (local) output and terminate upon an event.

Observe that an asynchronous algorithm can also operate in the synchronous model: If all nodes start the execution at time 0 and each message is under way

[2] The LOCAL model is also called *sychronous message passing* model, but that's a mouthful and I'm lazy.

for exactly 1 time unit, this is exactly the same as executing the algorithm in a synchronous system.

It is a crucial aspect of the asynchronous model that the time for a message to reach its destination is unbounded. However, we still would like to figure out what algorithms are "fast" in this model. We do this by giving the algorithm more slack.

Definition 2.3 (Asynchronous time complexity)**.** *An algorithm in the asynchronous message passing model has time complexity T, if in all executions in which all nodes start the algorithm at time 0 and each message is received at most one time unit after it was sent, all nodes terminate by time T.*

Remarks:

- An asynchronous algorithm of time complexity T has synchronous time complexity at most T.
- One can extend this definition to allow for not all nodes "waking up" at time 0.
- Assuming asynchrony is typically unrealistic, too. However, this time we're overly pessimistic, which means that algorithms can deal with "more synchronous" models, while lower bounds cannot.
- A lower bound or impossibility result based on asynchrony thus means that one can/should look for adding constraints that make the system "more synchronous," enable better algorithms, and yet are still pessimistic enough to be realistic.
- The synchronous and asynchronous models are two extremes, so understanding them also helps understanding what's in between.

2.3 Simulating Synchrony

Our goal is to be able to "pretend" that the system is synchronous when coming up with algorithms. In other words, we would like to figure out a (generic) way of transforming a synchronous algorithm into an asynchronous one. The new algorithm should behave just like the old, which is captured by the following definition.

Definition 2.4 (Simulation)**.** *Algorithm $\mathcal{A}$* simulates *algorithm $\mathcal{B}$, if, given the same inputs, both algorithms compute the same outputs.*

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

Definition 2.5 (valid clock pulse)**.** *Assume that (upon an event), node v can "trigger clock pulse i," for $i \in \mathbb{N}$ and nodes simulate a synchronous algorithm $\mathcal{A}$. When generating clock pulse i, node v will send all messages it would send in round i according to $\mathcal{A}$; to each of these messages it will attach the round number i. Clock pulse i (at node v) is* valid *if it is generated after v generated all pulses $j < i$, it has not been generated before, and it is generated after v received all the messages of the synchronous algorithm sent to v by its neighbors in rounds $j < i$.*

Given a mechanism that generates valid clock pulses $1, \dots, T+1$ at all nodes, a T-round synchronous algorithm can be simulated: Each node can compute its output based on its local history of messages and clock pulses.

Lemma 2.6. *If all generated clock pulses are valid according to Definition 2.5 and all non-terminated nodes keep generating pulses, we can simulate the respective synchronous algorithm.*

Proof. Suppose synchronous Algorithm $\mathcal{A}$ terminates in T rounds at node v. When v generates pulse $T+1$, it has received all messages sent by its neighbors during the execution of $\mathcal{A}$, neatly labeled by their round numbers. It can thus locally perform exactly the computations that $\mathcal{A}$ would, output the result, and terminate. □

Note that the messages a synchronous algorithm sends in round i are simply the "output" generated by executing the algorithm partially up to round $i-1$; if $i-1$ is 0, this just means to compute the first set of messages based on the input. In particular, it's safe to generate pulse 1 right away at each node. In other words, all we need to do is to provide a method that generates clock pulse $i+1$ at each node provided that clock pulse $i \in \mathbb{N}$ has been generated at each node and the respective messages of $\mathcal{A}$ have been sent.

The main issue with generating clock pulse $i+1$ at a node v is that (in general) v cannot know which of its neighbors send a message in round i of $\mathcal{A}$. As messages may be in transit arbitrarily long, this means that if it produces a clock pulse without hearing from some neighbor, it risks generating an invalid pulse. On the other hand, if there is no such message, but it plays things safe, it may wait indefinitely and we have a *deadlock*.

The most simple solution to this dilemma is to make sure that there is *always* a message from *each* neighbor in *each* round. Denote by $m_{\mathcal{A}}(v, w, i)$ the message v sends to w in round i when executing $\mathcal{A}$; if $\mathcal{A}$ sends no such message or has already terminated at v, we write $m_{\mathcal{A}}(v, w, i) = \perp$. With this notation, this strategy is cast into Algorithm 5.

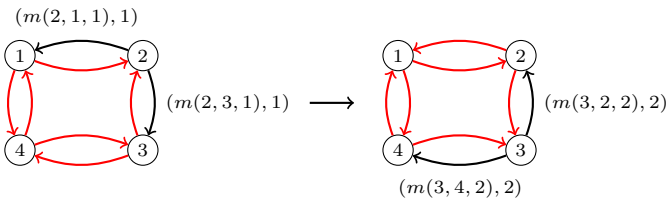

Figure 2.1: First two "rounds" of an execution of the α-synchronizer. Black arrows correspond to messages sent by the original algorithm, while red edges indicate $(\perp, i)$ messages. Actually, one can just send $\perp$ messages, where the receivers count the number of $\perp$ messages received from each neighbor; for this reason, we refrained from depicting the message contents for red arrows.

Algorithm 5 α-synchronizer simulating $\mathcal{A}$ (code for node $v \in V$)

```
1: if v just woke up then
2:    pulse_v := 1
3:    for {v, w} ∈ E do
4:       term_v(w) := ∞
5:       compute m_A(v, w, 1)
6:       send (m_A(v, w, 1), 1) to w
7:    end for
8: end if
9: if v received (m_A(w, v, i), i) from w then
10:   store (m_A(w, v, i), i)
11: end if
12: if v received (term, i) from w then
13:   term_v(w) := i
14: end if
15: if v stores (m_A(w, v, i), i) for each {v, w} ∈ E with term_v(w) > i and
    pulse_v = i then
16:   pulse_v := i + 1
17:   check if A terminates at v at the end of round i (from stored messages,
      ignoring ⊥ and term)
18:   if A terminates at v at the end of round i then
19:      for {v, w} ∈ E with term_v(w) > i + 1 do
20:         send (term, i + 1) to w
21:      end for
22:      compute output of A (from stored messages) and terminate
23:   else
24:      for {v, w} ∈ E with term_v(w) > i + 1 do
25:         compute m_A(v, w, i + 1) (from stored messages)
26:         send (m_A(v, w, i + 1), i + 1) to w
27:      end for
28:   end if
29: end if
```

Theorem 2.7 (Synchronizer α)**.** *Given a synchronous Algorithm $\mathcal{A}$ of running time T, synchronizer α simulates it in an asynchronous system with a running time of T. The number of additional messages send compared to an execution of $\mathcal{A}$ is at most $2(T+1)|E|$.*

Proof. To prove simulation, we will show that

1. All generated pulses are valid.

2. If node v terminates at the end of round T_v in $\mathcal{A}$, it generates pulses $1, \ldots, T_v + 1$, terminates when generating pulse $T_v + 1$, and outputs the correct result.

3. For each pulse $i \in \{1, \ldots, T_v\}$, v sends $(m_{\mathcal{A}}(v, w, i), i)$ to all neighbors upon generating pulse i.

4. At pulse $T_v + 1$, v sends $(\text{term}, T_v + 1)$ to all non-terminated neighbors.

5. If, for $\{v, w\} \in E$, w terminates with pulse T_w+1, v will wait for a message from w in rounds $1, \dots, T_w + 1$.

We prove this by induction. The base case is $i = 1$. Clearly, all nodes generate valid pulse 1 and compute and transmit the messages for round 1 upon wake-up; they wait for messages from all neighbors w, since initially $\text{term}_v(w) = \infty$. Also, note that in the event that a node would terminate immediately according to $\mathcal{A}$ (i.e., "after 0 rounds"), it also does so here, computes the same output, and sends a $(\text{term}, 1)$ message to each neighbor.

Now suppose all statements are true for pulses $1, \dots, i$ of each node and consider pulse $i + 1$. As all messages for round i were computed and sent, eventually they arrive. Neighbors not sending a message have, by the induction hypothesis, terminated in an earlier round, so v sent $\text{term}_v(w) = j$ for some $j \leq i$ and will not wait for such a message. Hence, eventually each v will generate pulse $i+1$. As the content of all messages for pulses $1, \dots, i$ was correct, nodes correctly store them and compute and send the messages $(m_{\mathcal{A}}(v, w, i+1), i+1)$. Likewise, termination and corresponding outputs are determined correctly. This completes the induction step and thus the proof that $\mathcal{A}$ is simulated.

Concerning the time complexity, observe that all messages for pulse/round 1 are sent at time 0 and received by time 1 (assuming delays of at most 1). Hence all (non-terminated) nodes generate pulse 2 by time 1 and the corresponding messages are received by time 2, and so on. By time T, all nodes terminated or generated pulse $T+1$, the latter also implying that they terminated because they completed T rounds of $\mathcal{A}$.

Regarding the number of sent messages, note that node v sends in pulses $i \in \{1, \dots, T_v + 1\}$ at most one message over each incident edge. Denoting by $\delta_v := |\{w \in V \mid \{v, w\} \in E\}|$ the *degree of* v, the total number of messages is at most

$$\sum_{v \in V} \sum_{i=1}^{T_v+1} \delta_v = \sum_{v \in V} (T_v + 1)\delta_v \leq (T + 1) \sum_{v \in V} \delta_v = 2(T + 1)|E|,$$

where the last equality uses that each edge has exactly two endpoints. □

Remarks:

- Despite its length, this proof is quite simple. The most difficult part is to figure out all the conditions that must be satisfied to perform the induction step and make them part of the induction hypothesis (which then makes things tedious).

- The same problem transpired when I wrote the pseudo-code for Algorithm 5. It's idea is straightfoward, but I made several mistakes. Allowing for all the cases and treating them properly can be tiresome, and makes it challenging to show correctness of more involved asynchronous algorithms.

- Fortunately, Theorem 2.7 shows that we have to do it only once! We now can devise synchronous algorithms and make them into asynchronous algorithms using the synchronizer.

- The argument extends to randomized algorithms in the following way. Interpret a randomized algorithm as a deterministic algorithm in which each

node has an additional input: a (sufficiently long) string of independent, unbiased random bits. Now synchronizer α simulates a randomized synchronous algorithm. The algorithm has to fulfill that if a node is given the same randomness (i.e., the strings are the same) throughout the simulated "asynchronous rounds," it must show the same behavior.

- Note that this can be subtle: If in a program one calls a standard function producing a random value, it may compute the "random" value by taking into account the system clock's time!
- Of course, that's not the end of the story. This equivalence between asynchronous and synchronous systems breaks down if we take into account other factors, such as the number of messages sent by an algorithm (we'll look into this now) or the possibility of failures (we'll look into this next lecture).

2.4 Synchronizing Globally

Synchronizer α is "expensive" in terms of messages. This may be fine if the simulated algorithm also sends a lot of messages, implying that there's not much of a difference. Similarly, if $\mathcal{A}$ communicates only over a subset of the edges that are known in advance (e.g. a spanning tree), we can also run the synchronizer on the induced subgraph only. However, if neither is the case and we care about the number of messages, we might want to look for something else.

An obvious way of reducing the number of messages per round related to the synchronizer is to communicate control messages over fewer edges. Since we need to make sure that all (potential) neighbors are synchronized, the respective edge set must connect all nodes. A connected graph with the fewest number of edges is a tree.

Definition 2.8 (Distributed representation of a rooted tree)**.** *A distributed* rooted tree *is defined as follows. There is a distinguished* root *node $v_0 \in V$. Each $v \in V \setminus \{v_0\}$ has a neighbor p_v as* parent*; p_v is known to v and vice versa.*

Note that it's easy to root an unrooted tree (i.e., one where nodes don't know p_v) at a node v_0 in time equal to it's depth with r as root, by sending messages "down" the tree. If we also need to figure out which node becomes root, we can use, e.g., the node with largest identifier. We then start the rooting procedure at each node, including the corresponding identifier into each message, and always let the currently largest known identifier "win."

How do we use a given rooted tree for synchronization? We let the root orchestrate the execution of the algorithm. Nodes will send their messages for a given round, wait for acknowledgements from the recipients, and then consider themselves "safe" for the current round.

Definition 2.9 (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.*

If all nodes are safe, we can move on to the next round. So we let the root know when all nodes are safe and then, in turn, distribute the information that

this is the case to all nodes – both via the tree. The details of synchronizer β are given in Algorithm 6.

Theorem 2.10 (β-synchronizer)**.** *Denote by d the depth of the tree employed by Algorithm 6. The algorithm simulates the synchronous T-round Algorithm $\mathcal{A}$ in $\mathcal{O}(d(T+1))$ asynchronous rounds. In total $\mathcal{O}(M+(T+1)n)$ messages are sent, where M is the number of messages sent by $\mathcal{A}$.*

Proof sketch. We give the main idea of the proof here; working out the details is similar to the proof of Theorem 2.7.

When the root issues a new pulse by sending a pulse message, it is forwarded to each node in the tree, as each node sends it to its children upon reception. Hence, if the root issues a pulse, eventually all nodes issue the pulse. Upon a pulse, nodes send the their messages for the respective round of $\mathcal{A}$, wait for the acknowledgements, and then set their safe-variable for the pulse to **true**. Note that this will eventually happen (no message is lost, acknowledgements are always sent), and it implies that the respective node is indeed safe. A safe node will send a safe message to its parent as soon as it received safe messages from all of its children. Thus, by induction on decreasing distance from the root (i.e., starting at leaves), all nodes will send safe messages to their parents; the induction also shows that this entails that the subtree rooted at the respective node consists of safe nodes only. Consequently, eventually the root will generate the next pulse, and at this point all nodes are safe.

This way the algorithm will proceed until all nodes have determined that $\mathcal{A}$ has locally terminated. This information is forwarded to the root in a similar fashion to the information that nodes are safe (using the done messages), the only difference being that not necessarily all nodes terminate in the same simulated round of $\mathcal{A}$. However, all nodes participate in the synchronizer until the root initiates the distribution of term messages, which happens when it learns that all nodes' simulation of $\mathcal{A}$ has locally terminated. We conclude that Algorithm 6 simulates $\mathcal{A}$.

Now consider the time complexity. Suppose pulse i starts at time t. By time $t+d$, all nodes have received their (pulse, i) message. Thus, by time $t+d+2$, the messages of $\mathcal{A}$ for round i have been received, acknowledged, and also these acknowledgements have arrived. Now, starting from the leaves, we see that by time $t+2d+2$, the root will have received safe messages from all its children and issue pulse $i+1$. Since $\mathcal{A}$ terminates in T rounds, all nodes will notice this when generating pulse $T+1$. Then it takes at most d time until the root learns that all nodes have completed their part of the execution of $\mathcal{A}$ and another d time to terminate, for a total of $\mathcal{O}(dT+d)=\mathcal{O}(d(T+1))$ asynchronous rounds.

Finally, let us check the message complexity. In each pulse, over each tree edge a pulse and a safe message is sent. We also have one done and one term message per tree edge. All other messages are either messages of $\mathcal{A}$ or acknowledgements for such messages. Hence, the total number of messages is

$$\begin{aligned} & \sum_{\text{tree edges}} 2 + \sum_{i=1}^{T+1} \sum_{\text{tree edges}} 2 + \sum_{\text{messages of } \mathcal{A}} 2 \\ = \;& 2(n-1) + 2(T+1)(n-1) + 2M \\ \in \;& \mathcal{O}(M+(T+1)n), \end{aligned}$$

Algorithm 6 β-synchronizer simulating $\mathcal{A}$ (code for node $v \in V$). For simplicity, we adopt the convention that v stores all received information for later reference and performs necessary computations.

if v just woke up **then**
  $\text{done}_v := \textbf{false}$
  if v is root **then**
    send $(\text{pulse}, 1)$ to self (i.e., execute code for "received $(\text{pulse}, 1)$")
  end if
end if
if received (pulse, i) **then**
  $\text{pulse}_v := i$
  $\text{safe}_v(i) := \textbf{false}$
  send (pulse, i) to children
  for $\{v, w\} \in E$ with $m_{\mathcal{A}}(v, w, i) \neq \perp$ **do**
    send $(m_{\mathcal{A}}(v, w, i), i)$ to w
  end for
end if
if v received $(m_{\mathcal{A}}(w, v, i), i)$ from w **then**
  send (ACK, i) to w
end if
if v received (ACK, i) from all w with $m_{\mathcal{A}}(v, w, i) \neq \perp$ **then**
  $\text{safe}_v(i) := \textbf{true}$
end if
// do the following only once for each pulse i
if $\text{safe}_v(i) = \textbf{true}$ and v received (safe, i) messages from all children **then**
  if v is the root **then**
    send $(\text{pulse}, i+1)$ to self (i.e., execute code for "received $(\text{pulse}, i+1)$")
  else
    send (safe, i) to parent
  end if
  if $\mathcal{A}$ terminates at v at the end of round i **then**
    $\text{done}_v := \textbf{true}$
  end if
end if
if $\text{done}_v = \textbf{true}$ and received done from all children **then**
  if v is root **then**
    send term to all children
    compute output and terminate
  else
    send done to parent
  end if
end if
if v received term **then**
  send term to children
  compute output and terminate
end if

provided that no node ever sends a message related to pulses larger than $T+1$.

To achieve this, we bundle up the synchronizer-related messages a node sends to its parent (i.e., safe and done) in each pulse. This means that the root cannot learn that pulse $T+1$ is complete without also noticing that $\mathcal{A}$ has terminated. The root then will just send the term message and terminate without issuing another pulse. □

Remarks:

- The last paragraph of this proof highlights again how careful one needs to be with asynchrony. If safe and done messages are handled independently, it could happen that a done message is incredibly slow and we execute a huge number of pointless pulses!
- Strictly speaking, the theorem is therefore about a slightly different version of Algorithm 6. Since this means the theorem is technically wrong, I hope that this way of presenting it is at least very didactic.
- Note that nodes cannot terminate just because they're done with simulating $\mathcal{A}$. They have to relay information to and from the root.
- The β-synchronizer is an example of repeated use of the *flooding/echo* routine. The root "floods" the information to do something through the tree, and the result is then collected by the "echo" emanating from the leaves. Here, the job is to make sure that all messages of $\mathcal{A}$ have arrived, so some additional waiting can be involved.
- Flooding/echo is extremely useful when sufficient time is available (i.e., we don't mind waiting for d time). One can use the principle for detecting termination of algorithms or determining sums (including the number of nodes), averages, maxima, etc., and then making the result known to everyone.
- One can do the flooding also without having a tree at hand, instead constructing it "on the fly." In this version, the first received message determines the parent, and one speculatively sends a message to all neighbors at this point, as they might become children. This costs $\Theta(|E|)$ messages. If we start from a "clean slate" (no knowledge of the network topology), this number of messages is necessary to make sure that no node is missed: an unexplored edge may lead to a node with no other connection to the network.
- Surely, synchronizer β is message-optimal (up to constants)? Nope! We *could* just collect and make known the entire graph (including identifiers and inputs) to each node by collecting and distributing it using echo/flooding on the tree, using $\mathcal{O}(n)$ messages. Afterwards, each node can simulate the complete algorithm locally!
- "That's cheating!!" you might exclaim. Rightfully so, because if we could do things centrally, then we wouldn't need to think about a distributed algorithm in the first place. Still, in practice this is something to always check before rushing to the fancy solutions from this course!

2.5 BFS Tree Construction

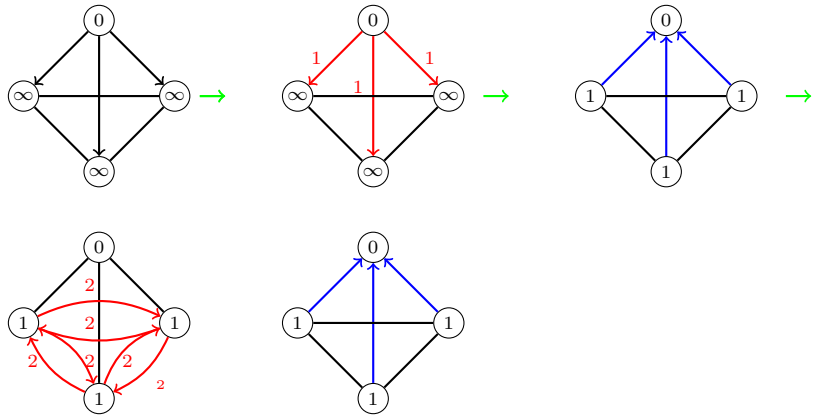

Figure 2.2: Trivial synchronous BFS tree construction. The root sends "1" to all neighbors. Nodes receiving a message "d" know they are in distance d from the root, send $d+1$ to all neighbors, and terminate. Here we see the algorithm in the complete graph; in general, it requires D synchronous rounds, where D is the diameter of the network.

Synchronizer β requires a tree to operate, preferably one with small depth d. In synchronous systems, flooding is a simple yet efficient method to construct a breadth-first search (BFS) spanning tree using at most $2|E|$ messages and ensuring that $d \leq D$, where

$$D := \max_{v,w \in V} \{\text{dist}(v,w)\} \tag{2.1}$$

is the network *diameter* and $\text{dist}(v,w)$ is the length of a shortest path from v

Algorithm 7 Dijkstra BFS

1: The algorithm proceeds in phases. In phase i the nodes with distance i to the root are detected. Let T_i be the tree in phase i. We start with T_0 which is just the root.
2: **repeat**
3: The root starts phase i by broadcasting "start i" within T_i.
4: When receiving "start i" a leaf node v of T_i (that is, a node that was newly discovered in the last phase) sends a "join $i+1$" message to all quiet neighbors. (A neighbor w is quiet if v has not yet "talked" to w.)
5: A node v receiving the first "join $i+1$" message replies with "ACK" and becomes a leaf of the tree T_{i+1}.
6: A node v receiving any further "join" message replies with "NACK."
7: The leaves of T_i 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

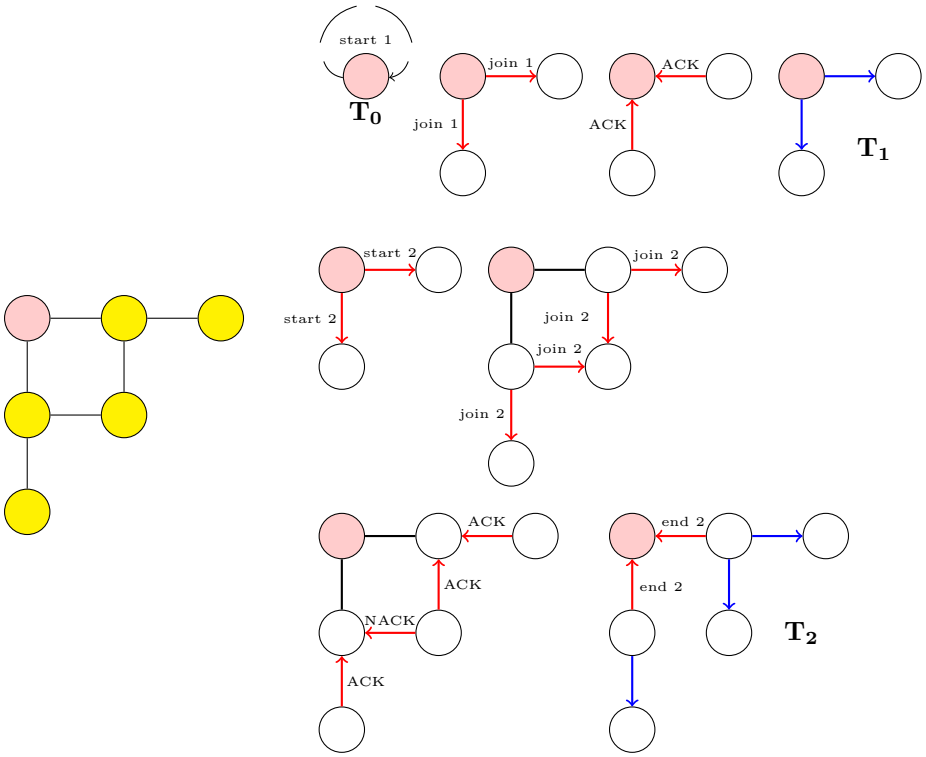

Figure 2.3: A run of Dijkstra's algorithm on the graph depicted on the left. In the first phase, the neighbors of the root join the tree, resulting in T_1. The second phase uses the existing tree edges to communicate its start and finish messages, very similar to the β-synchronizer. If there were more distant nodes w.r.t. the root, there would be more phases following the same pattern.

to w. However, in asynchronous systems the spanning tree constructed by the flooding algorithm may be far from BFS; in the worst case, we construct a line in a complete graph (see Figure 2.4)!

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 parallelize this idea by developing the BFS tree layer by layer. In Algorithm 7, by "broadcast" we denote the operation that the root uses flooding to distribute some information throughout the (current) tree. "Echo" means the process of all leaves sending some information to the root, which interior nodes aggregate from all children before forwarding it.

Theorem 2.11 (Distributed Dijkstra)**.** *The time complexity of Algorithm 7 is* $\mathcal{O}(D^2)$*. Its message complexity is* $\mathcal{O}(|E| + nD)$*.*

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 depth of the BFS tree is bounded by the diameter, we have at most D phases, giving a total time

complexity of $\mathcal{O}(D^2)$.

The broadcast/echo routine uses each edge of a tree twice, i.e., at most $2(n-1)$ such messages are sent in each phase. Since there are D phases, this amounts to $\mathcal{O}(nD)$ messages. 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 $\mathcal{O}(|E| + nD)$. □

Remarks:

- The description of the algorithm is less formal than before, but highlights the structure of the algorithm better. This helps with explaining the idea, but don't be fooled: This style can easily trick the reader (or writer!) into believing some things will happen in a certain order, while in fact asynchrony could cause something entirely different to happen!

- We haven't specified how the root is selected. Either one has to specify this in advance, or the problem of *leader election* has to be solved. This is another fundamental problem in distributed computing.

The basic idea of the Moore-Bellman-Ford algorithm is even simpler. We keep book on the distance to the root. If a node has found a better route to the root, its neighbors update their distance accordingly.

Algorithm 8 Bellman-Ford BFS

1: Each node v stores an integer d_v which corresponds to the distance $\text{dist}(v, v_0)$ to the root v_0. Initially $d_{v_0} = 0$, and $d_v = \infty$ for $v \in V \setminus \{v_0\}$.
2: The root starts the algorithm by sending "1" to all neighbors.
3: **if** v receives "d" with $d < d_v$ from w **then**
4: $\quad d_v := d$
5: $\quad p_v := w$
6: $\quad v$ sends "$d + 1$" to all neighbors
7: **end if**

Theorem 2.12 (Bellman-Ford BFS)**.** *The time complexity of Algorithm 8 is $\mathcal{O}(D)$, the message complexity is $\mathcal{O}(n|E|)$.*

Proof. We 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 w at distance $d-1$. Node w, by the induction hypothesis, sends a message "d" to v by time $d-1$, which is then received by v by time d.

Regarding message complexity, a node can reduce its distance at most $n-1$ times; each of these times it sends a message to all of its neighbors. If all nodes do this we have $\mathcal{O}(n|E|)$ messages. □

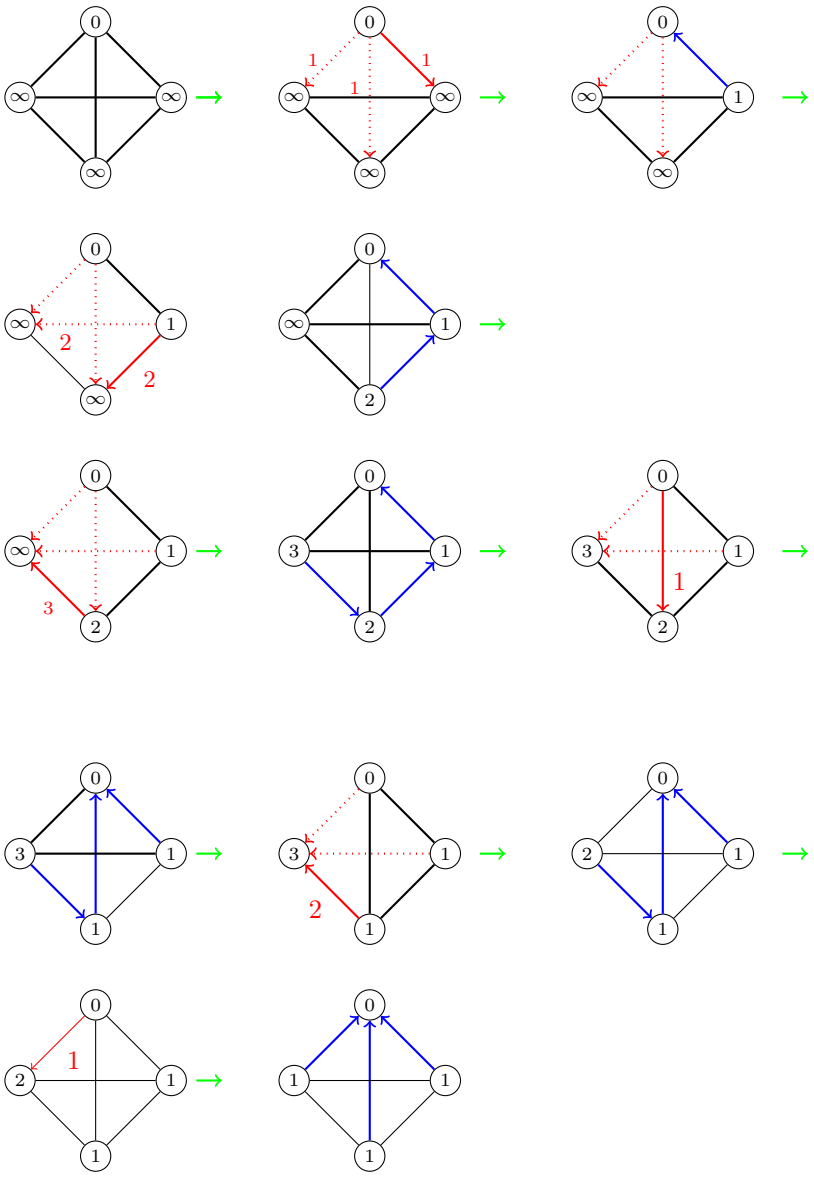

Figure 2.4: "Bad" execution of the Bellman-Ford algorithm on the complete graph of 4 nodes. Red arrows indicate messages that cause nodes' estimated distances to change upon reception. Dotted arrows indicate that such messages are delayed. Note that terminating when the first distance estimate is obtained (i.e., running a naive synchronous BFS algorithm) would yield erroneous results. Moreover, on average each node changes its label $n/2$ times (here $n = 4$), implying a message complexity of $n \cdot n \cdot n/2 \in \Theta(n^3)$.

Remarks:

- Here's another "wrong" algorithm/proof: I didn't say in the algorithm or

the theorem how termination is detected. Can you see how to fix this? (If not, don't worry.)

- Algorithm 7 has the better message complexity and Algorithm 8 has the better time complexity. The currently best algorithm (optimizing both) needs $\mathcal{O}(|E| + n \log^3 n)$ messages and $\mathcal{O}(D \log^3 n)$ time. This "trade-off" algorithm is beyond the scope of this lecture.

- As such an advanced algorithm is quite efficient and one usually can construct the tree in advance, in many cases it is not a big deal. Still, one needs to keep in mind that this initial overhead exists and might not be worth it.

2.6 Hybrid Synchronizers

Synchronizer α is fast, but costs a lot of messages. Synchronizer β is slow, but efficient in terms of messages. Can we compromise? The answer is yes and called (surprise!) synchronizer γ.

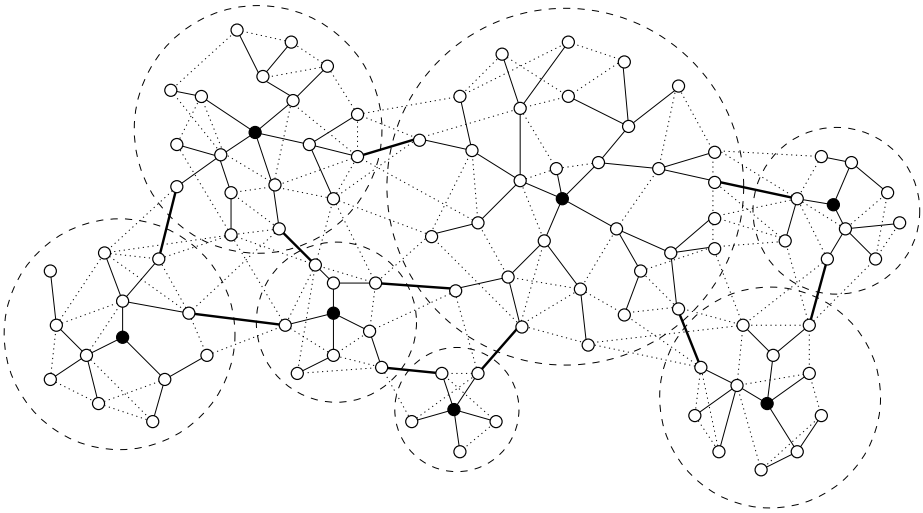

Figure 2.5: A cluster partition of a network: The dashed circles specify the clusters, cluster leaders are black, the solid edges are the edges of the intracluster trees, and the bold solid edges are the intercluster edges.

We will now briefly discuss the key ideas (there has been enough detail for one lecture already!). 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 2.5 illustrates this partitioning into clusters.

We say that a cluster is safe if all its nodes are safe. Let's start the description from all nodes generating a pulse (which, of course, may happen at

very different times). In at most 2 time units, all nodes will be safe. As in synchronizer β, we now let the leader of each cluster learn that the cluster is safe (we use acknowledgements again). Then, the leader will let the leaders of adjacent clusters know that its cluster is safe. This is done by flooding this information through the own cluster using the tree, then communicating it over the intercluster edges, and using the same approach as in the β-synchronizer to collect the information that all adjacent clusters are safe within the cluster (i.e., a "safe" message is sent to the parent once "safe" via all incident intercluster edges and from all children has been received). Once a cluster leader knows that all adjacent clusters and its own are safe, it tells all nodes in its cluster to generate the next pulse. Hence, we essentially apply synchronizer α between clusters.

Theorem 2.13 (Synchronizer γ)**.** *Let E_C be the set of intercluster edges and let k be the maximum cluster radius (i.e., the maximum distance of a leaf to its cluster leader). Simulating a synchronous T-round algorithm sending M messages using synchronizer γ then takes $\mathcal{O}((T+1)k)$ time and requires $\mathcal{O}(M + (T+1)(|E_C| + n))$ messages.*

Proof sketch. From the above description, we see that each pulse requires a constant number of flooding and echo operations on the trees (which have depth at most k and in total at most $n-1$ edges), 2 messages over each intercluster edge, and the messages of the simulated algorithm plus acknowledgements. Hence, each pulse takes $\mathcal{O}(k)$ time, $\mathcal{O}(|E_C| + n)$ synchronizer messages, and two times the number of messages sent by the algorithm in the simulated round. Summing up, over T rounds (and taking into account handling of termination), we get the stated bounds. □

In the exercises, we will see that one can have $|E_C| \in \mathcal{O}(n^{1+1/k})$, which is pretty impressive.

Corollary 2.14 (Synchronizer γ)**.** *For $k \in \{1, \ldots, \lceil \log n \rceil\}$, there is a network partion so that synchronizter γ simulates a synchronous T-round algorithm sending M messages in $\mathcal{O}((T+1)k)$ time and requires $\mathcal{O}(M + (T+1)n^{1+1/k})$ messages.*

Remarks:

- For $k = 1$, this is just like synchronizer α.
- For $k = \lceil \log n \rceil$, this uses as few messages as synchronizer β, but pays only a factor of $\mathcal{O}(\log n)$ in time, regardless of D!

Remarks:

- It can be shown that the trade-off between cluster radius and number of intercluster edges of Corollary 2.14 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 synchronizers 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. In such scenarios, it is possible to achieve multiplicative time and message complexity overheads of $\mathcal{O}(\log^3 n)$ (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 maximal independent sets can be based on some kind of network partition or cover.

What to take Home

- Asynchrony does not affect solvability of problems – if there are no faults.
- It comes at a cost in time and/or message complexity, though.
- Simulation is a powerful tool for designing algorithms. Designing and analyzing advanced asynchronous algorithms can be very challenging. If it's ok to run a synchronizer (which simulates synchrony), things can become astronomically simpler.
- Not all problems have a single best solution. Frequently, there is a trade-off between quality measures that cannot be compared in a straightforward way, e.g. time vs. messages.
- Then again, there might be a "sweet spot," where the cost in either measure is very small. We had this with time vs. number of colors for list coloring, and with the γ-synchronizer using a good network partition.
- On the way to a good solution, it may turn out that one needs to solve another problem that did not appear to be of relevance initially.
- Totally unrelated: network partitions rock!

Bibliographic Notes

The idea behind synchronizers is quite intuitive and as such, synchronizers α and β were implicitly used in various asynchronous algorithms [Gal76, Cha79, CL85] before being proposed as separate entities. The general idea of applying synchronizers to run synchronous algorithms in asynchronous networks was first introduced by Awerbuch [Awe85]. His work also formally introduced the synchronizers α, β, and γ. Improved synchronizers that exploit inactive nodes or hypercube networks were presented in [AP90, PU87].

Trees are one of the oldest graph structures, already appearing in the first book about graph theory [Koe36]. *Broadcasting* (i.e., flooding some information

through the network) in distributed computing is younger, but not that much [DM78]. Overviews about broadcasting can be found for example in Chapter 3 of [Pel00] and Chapter 7 of [HKP+05]. Overviews of distributed tree construction can be found in Chapter 5 of [Pel00] or Chapter 4 of [Lyn96]. The classic papers on routing are [For56, Bel58, Dij59].

Figure 2.5 is courtesy of Roger Wattenhofer. Wide parts of this lecture are based on the corresponding lecture of his course "Principles of Distributed Computing."

Bibliography

[AP90] Baruch Awerbuch and David Peleg. Network Synchronization with Polylogarithmic Overhead. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science (FOCS)*, 1990.

[Awe85] Baruch Awerbuch. Complexity of Network Synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, October 1985.

[Bel58] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[Cha79] E.J.H. Chang. *Decentralized Algorithms in Distributed Systems.* PhD thesis, University of Toronto, 1979.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 1:63–75, 1985.

[Dij59] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[DM78] Y.K. Dalal and R.M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 12:1040–148, 1978.

[For56] Lester R. Ford. Network Flow Theory. *The RAND Corporation Paper P-923*, 1956.

[Gal76] Robert Gallager. Distributed Minimum Hop Algorithms. Technical report, Lab. for Information and Decision Systems, 1976.

[HKP+05] Juraj Hromkovic, Ralf Klasing, Andrzej Pelc, Peter Ruzicka, and Walter Unger. *Dissemination of Information in Communication Networks - Broadcasting, Gossiping, Leader Election, and Fault-Tolerance.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2005.

[Koe36] Denes Koenig. *Theorie der endlichen und unendlichen Graphen.* Teubner, Leipzig, 1936.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Pel00] David Peleg. *Distributed computing: a locality-sensitive approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[PU87] David Peleg and Jeffrey D. Ullman. An Optimal Synchronizer for the Hypercube. In *Proceedings of the sixth annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 77–85, 1987.

Lecture 3

Impossibility of Consensus

In the previous lecture, we saw that it is possible to simulate synchronous algorithms in asynchronous systems. Today, we will see that a basic fault-tolerance task, *consensus*, is unsolvable in asynchronous systems. In the exercises, we will see that consensus is straightforward in synchronous systems, separating synchronous and asynchronous systems beyond differences in efficiency.

3.1 The Problem

A standard formulation of the (binary) consensus problem is given as follows. Each of n nodes is given a binary input b_i, $i \in \{1, \ldots, n\}$. Nodes may *crash* during the execution. A node that crashes is *faulty*, while nodes that do not crash are *correct*. Correct nodes $i \in [n]$ are to compute an output o_i such that the following properties hold.

Agreement Correct nodes i output the same value $o = o_i$.

Validity If all nodes have the same input b, then $o = b$.

Termination All correct nodes decide on an output and terminate.

Being able to solve this problem can, e.g., be useful for control of a plane. For safety reasons, there are several computers in case some of them fail. Suppose they need to decide between two possible courses for the plane, at least one of which is safe. If the computers each compute an opinion b_i based on the data they have, you surely want the decision to satisfy all three properties:

Validity If the data clearly prefers one route over the other, this decision should be taken! Otherwise: plane crash.

Agreement *Some* decision must be taken even if the data is inconclusive (the computers compute different values b_i). The plane must take one of the two routes! Otherwise: plane crash.

Termination This decision must be taken at some point. In fact, probably soon, which is why the time complexity of consensus algorithms is important, too! Otherwise: plane crash.

Figure 3.1: This is not supposed to happen!

Note that this problem is a no-brainer in absence of faults. Just pick a leader (e.g., the node with smallest identifier) and decide on its input! But what if this node crashes? In a synchronous system someone will notice, but in an asynchronous system there is no way to be sure that it's not just a bad case of excruciatingly slow message delivery...

We need to specify the model in which we want to consider the problem. We will use a model that is stronger than the message passing model (we will see later why), the asynchronous *shared memory* model. Here, there is some common memory accessible by all n nodes that is used to communicate. Nodes read and write *registers* of this memory *atomically*. This means that nodes read the entire register in one go or (over)write the content of a register without anyone else interfering. For convenience, we assume that all registers are initialized with a special symbol $\perp$. The catch now is that a scheduler decides who's next – and since we're talking asynchrony here, it is under no obligation regarding which other nodes it schedules (and how often) before it picks a specific node that wants to read or write. However, it is required to schedule non-crashed nodes *eventually*. Any node that intends to read or write is scheduled (or crashes) after finitely many steps. This property is called *fairness*. The scheduler may also decide to *crash* a node, simply meaning that it will not be scheduled again.

As usual, nodes have unique identifiers, initially know their input value only, and local compuations are "free." It's convenient to assume that a node performs all its initial local computations and those after a read/write instantaneously. Thus, a node is always either waiting to perform a read or write operation, is crashed, or is terminated; local termination occurs when a node decides at the end of a step that it's done and outputs a value.

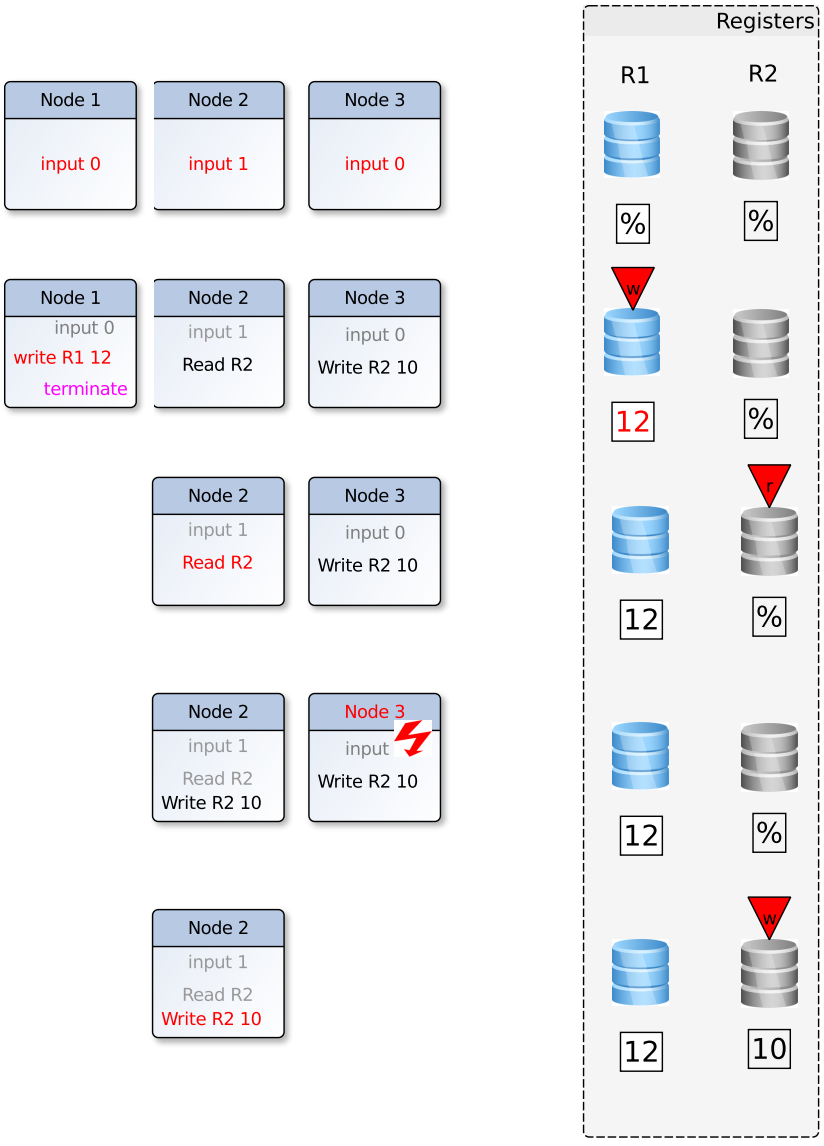

Figure 3.2: Sample execution of shared memory system with 3 nodes and 2 shared registers. The depicted execution is $(b_1 = 0, b_2 = 1, b_3 = 0, \text{write}_1(R1, 12), \text{term}_1, \text{read}_2(R2), \text{crash}_3)$. The currently executed operation is marked red, gray operations are already executed and black operations are currently outstanding.

Remarks:

- Dropping any of the requirements of agreement, validity, or termination renders the problem trivial. Think of the respective "solutions!"
- Observe that fairness basically means that it's not ok to crash a node without saying so. This is relevant because a crashed node does not have to decide or, if it already decided, have the same output as others.
- With fairness, one can define asynchronous rounds like for message passing: within one time unit, each node is guaranteed to be scheduled at least once.
- We assume the powerful shared memory communication and benign faults (there's much worse than clean crashes out there, but that's a tale for another day!). This makes the impossibility we will show a strong result.
- On the other hand, we consider asynchronous communication and deterministic algorithms, so do not despair!

3.2 Getting Started

Today's main result was surprising and a big deal when it was shown first. It was surprising both because it's not easy to show and because quite a few people believed that asynchronous consensus *is* possible. It will be much easier for us, and that's because the right definitions will point us in the right direction.[1]

Definition 3.1 (Executions)**.** *An execution of an algorithm is given by a sequence of read and write operations, crashes, and terminations, alongside the initial inputs given to the nodes; naturally, the decision whether a node terminates, reads, or writes (and if so what) in its next step is made by the algorithm.*

Note that, since we require that all nodes that do not crash must terminate, all executions that are relevant to us are of finite length. Also, as stated earlier, we will consider fair executions only.

We now can state the main result.

Theorem 3.2 (FLP (Fischer, Lynch & Patterson))**.** *There is no algorithm that solves the consensus problem in all fair executions with at most one fault.*

As mentioned, good definitions are pivotal. We will need two key concepts. The first is called *indistinguishability*. Note that while Definition 3.1 is about the entire network, the following definition is about how an execution looks like at a specific node:

Definition 3.3 (Indistinguishable Executions)**.** *Two executions are* indistinguishable *at node i, iff in both executions i has the same input, performs the same sequence of read and write operations, and all the read operations return the same values in both executions.*

[1] The professor of one of my math courses once said that *definitions* are even more important than theorems, because the right definition tells us how to look at things and paves the way for the big results.

If two executions are indistinguishable at node i, it must behave the same way in both executions.

Lemma 3.4. *If two executions are indistinguishable at node i, the write operations of i in both executions are identical. If it terminated, the output values are identical. If it hasn't terminated yet, its next action is the same in both executions.*

Proof. By induction, the memory state of and values written by i are the same in the respective steps of each execution. Hence i's output value or next step, respectively, is also the same. □

The second definition looks even simpler.

Definition 3.5 (Bivalency and Univalency)**.** *For $b \in \{0, 1\}$, an execution of a consensus algorithm is* b-valent*, if any possible continuation of the execution results in output b. It is* univalent*, if it is b-valent for some b. Otherwise, it is* bivalent*.*

Combining these two notions, we obtain a crucial observation that will be at the heart of our reasoning.

Corollary 3.6. *If two executions are indistinguishable at all non-crashed nodes and each shared register contains the same value at their end, they have the same valency (i.e., both are 0-, both are 1-, or both are bivalent).*

Proof. By (inductive use of) Lemma 3.4, any extension of one execution is also a valid extension of the other, and the result will be two indistinguishable executions: every read operation will return the same value in both executions. Thus, outputs in such a pair of executions must be identical. Now the claim readily follows from the definition of bi- and univalency. □

Here's the plan:

1. Show that there are bivalent executions or validity is violated.
   (a) If validity holds, use it to show that there are 0- and 1-valent executions.
   (b) Infer that there must be a configuration for which one node's input makes the difference.
   (c) Conclude that crashing/not crashing the node must result in different outputs in some execution.
2. For any node i, show that we can extend any bivalent execution to another bivalent execution such that i takes another step; alternatively, there is an execution violating agreement.
   (a) For a bivalent execution that has no bivalent extension with another step of i, there are 0- and 1-valent extensions involving another step of i.
   (b) Infer that there must be a configuration for which swapping the steps of nodes i and some $j \neq i$ makes the difference between 0- and 1-valency.

   (c) Perform a case analysis proving that agreement is violated in some execution (using Lemma 3.4, Corollary 3.6, and the 0-/1-valency of the extensions).

3. Conclude that if agreement and validity hold, an infinite fair execution exists (i.e., termination does not hold).

As you can see, many of the above statements require that some of the properties of a consensus algorithm hold. For simplicity, we will assume that we have an algorithm solving consensus and ultimately derive a contradiction. Apart from this, the structure of the proof remains exactly as outlined above.

3.3 Step 1: Bivalent Executions Exist

In the following, let $\mathcal{A}$ be a consensus algorithm, i.e., one that satisfies agreement, validity, and termination. First, we use validity to show that there must be at least one bivalent execution.

Lemma 3.7. *$\mathcal{A}$ has a bivalent execution without crashes.*

Proof. For $j \in [n+1]$, consider the execution $\mathcal{E}_j$ that's simply given by the inputs $b_i = 0$ for all $i > j$ and $b_i = 1$ for $i \leq j$ (i.e., nothing has happened yet except for the inputs being specified). If any of these executions is bivalent, we're done, so let's suppose for contradiction that they are all univalent.

If $j = 0$, $b_i = 0$ for all $i \in \{1, \ldots, n\}$, and validity implies that the output is 0. Likewise, the execution with $j = n$ is 1-valent. Hence, there must be some $j \in [n]$ such that $\mathcal{E}_j$ is 0-valent and $\mathcal{E}_{j+1}$ is 1-valent. Since nothing has happened yet, both executions are indistinguishable to all nodes but j, which has different input in both executions. Thus, crashing j yields two executions of different valency that are indistinguishable at all non-crashed nodes, where the shared registers haven't been touched yet. This contradicts Corollary 3.6! □

Remarks:

- We used the *possibility* of a fault to show that there is a bivalent execution. However, we didn't "use up" the fault, we have a fault-free bivalent execution!

3.4 Step 2: Extending Bivalent Executions

Next, we show that, given a bivalent execution and a node i, a "follow-up" execution exists that is also bivalent and in which i performs a step. This last bit is crucial, because it ensures that a bivalent execution can be "kept bivalent" *even in a fair schedule.*

We start with a helper lemma ensuring that we can extend a bivalent execution to force either decision without crashing a node.

Lemma 3.8. *Given a bivalent execution $\mathcal{E}$ of $\mathcal{A}$ and $b \in \{0,1\}$, we can extend $\mathcal{E}$ to a b-valent execution $\mathcal{E}'$ without any further crashes.*

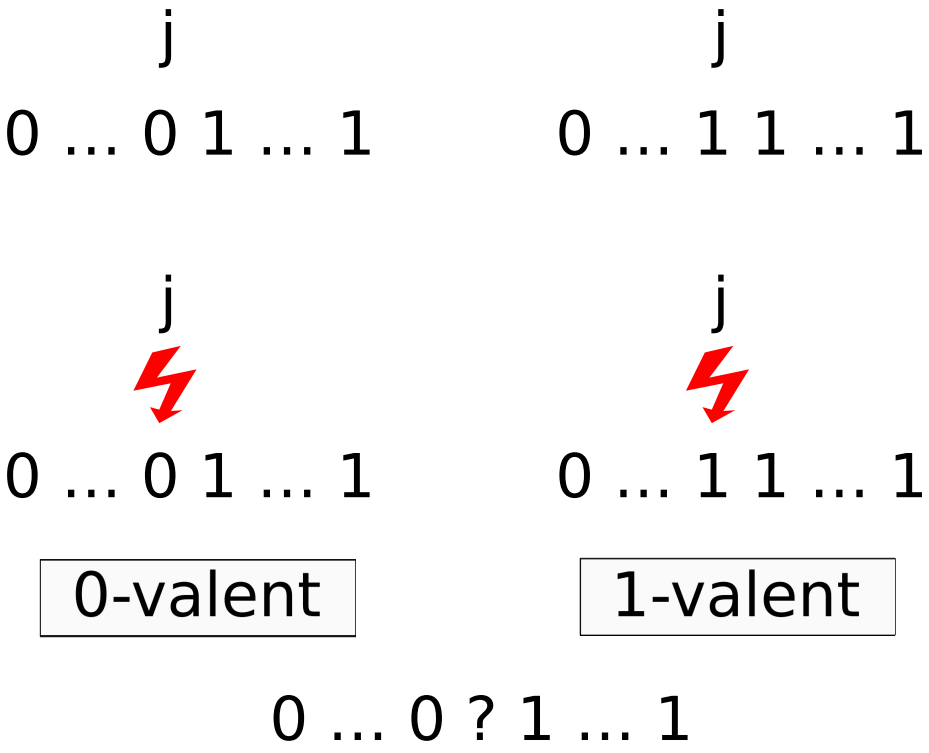

Figure 3.3: Key argument of Lemma 3.7. Assuming that no bivalent execution exists, validity implies that we can find a pair of "executions" (i.e., inputs) for which only the input of a single node differs, but one execution is 1- and the other 0-valent. Crashing this node, which is the only one knowing about the difference, right away, yields a contradiction.

Proof. By definition of bivalency, there is *some* execution $\mathcal{E}_b$ extending $\mathcal{E}$ that is b-valent. However, it might contain crashes. We extend $\mathcal{E}_b$ further to $\mathcal{E}'_b$, in which some node decides on b and terminates; this is feasible by termination of $\mathcal{A}$. Now we remove all crashes from $\mathcal{E}'_b$, resulting in execution $\mathcal{E}'$. By Lemma 3.4 and the fact that the crashed nodes do not change the contents of registers in either execution, the node still decides on b and terminates. Thus, $\mathcal{E}'$ must be b-valent by agreement, and by construction it contains no further crashes. □

Now we can proceed to extending (fault-free) bivalent executions in a way keeping them bivalent (and fault-free).

Lemma 3.9. *Given a bivalent execution $\mathcal{E}$ of $\mathcal{A}$ and a non-crashed node $i \in [n]$, we can construct a bivalent execution with an additional step of i. If $\mathcal{E}$ is fault-free, so is the new execution.*

Proof. Refer to Figure 3.4. Clearly, i cannot be terminated in $\mathcal{E}$, as otherwise the execution must be univalent by agreement. Let i take an additional step. If the extended execution $\mathcal{E}_0$ is still bivalent, we're done. Otherwise, assume w.l.o.g. that $\mathcal{E}_0$ is 0-valent. Because $\mathcal{E}$ is bivalent, by Lemma 3.8 there is also

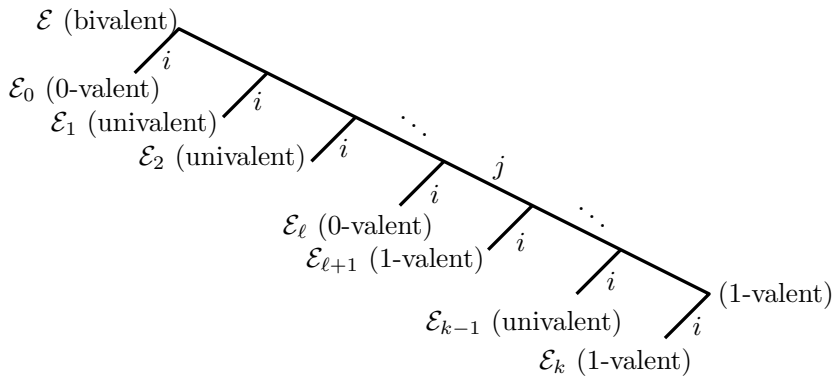

Figure 3.4: By assumption there is no bivalent extension of $\mathcal{E}$ that contains an additional step of node i. The switch from 0- to 1-valency happens between $\mathcal{E}_\ell$ and $\mathcal{E}_{\ell+1}$.

an extension of $\mathcal{E}$ that is 1-valent and contains no further crashes. Take such an extension, denote by k the number of additional steps, and let $\mathcal{E}_k$ be this extension plus an additional step of i.[2] Note that $\mathcal{E}_k$ is 1-valent, as any extension of a 1-valent execution is 1-valent. In summary, we have two extensions of $\mathcal{E}$, both with a step of i at the end, and either 0 ($\mathcal{E}_0$) or $k \neq 0$ ($\mathcal{E}_k$) intermediate steps. $\mathcal{E}_0$ is 0-valent and $\mathcal{E}_k$ is 1-valent.

Next consider the executions $\mathcal{E}_\ell$, $\ell \in [k+1]$, which are $\mathcal{E}$ followed by the next ℓ steps that happen in $\mathcal{E}_k$, and each of them with a final step of i (Figure 3.4). If any of these are bivalent, we're done. Otherwise, as we know that $\mathcal{E}_0$ is 0-valent and $\mathcal{E}_k$ is 1-valent, there must be some $\ell \in [k]$ so that $\mathcal{E}_\ell$ is 0-valent and $\mathcal{E}_{\ell+1}$ is 1-valent.

Suppose that j is the node that takes the final step before node i in execution $\mathcal{E}_{\ell+1}$. Note that both executions are indistinguishable at all nodes but i and j, and any difference must come from the final one or two steps. To complete the proof, we go through all possible cases and lead each of them to a contradiction with Corollary 3.6.

$i = j$: In this case, letting i take another step in $\mathcal{E}_\ell$ results in $\mathcal{E}_{\ell+1}$. But one is 0- and the other is 1-valent. Contradiction!

j does not write: We crash j at the end of both $\mathcal{E}_\ell$ and $\mathcal{E}_{\ell+1}$. The executions are indistinguishable at all nodes but the crashed j, and i did the same write (or read) in both executions, resulting in identical content of the shared registers. However, one execution is 0- and the other 1-valent. Contradiction!

i does not write: We let j take another step in $\mathcal{E}_\ell$, which is the same as the one in $\mathcal{E}_{\ell+1}$; since i didn't write, it is the only node that can distinguish the two executions, and again the shared registers have identical contents in both executions. Crashing i thus yields a contradiction.

[2] Again, as soon as i terminates, the execution must become univalent, so either i can still take a step or i just terminated in the last step; in the latter case, we just use the execution directly without adding a step of i.

i and j write to different registers: We let j take its step in $\mathcal{E}_\ell$ and obtain an execution that is indistinguishable at all nodes. Since the two writes do not interfere, the shared registers' content is identical, too. One execution is 0-, the other 1-valent. Contradiction!

i and j write to the same register: As i overwrites j's write in $\mathcal{E}_{\ell+1}$, j is the only node that can distinguish $\mathcal{E}_\ell$ and $\mathcal{E}_{\ell+1}$, and the register contains the value written by i at the end of both executions. Crashing j yields a contradiction!

Since all possibilities lead to a contradiction, we must have had the situation that we encountered a bivalent execution earlier on. Also, all the executions $\mathcal{E}_\ell$, $\ell \in [k+1]$, contained at least one more step of i than $\mathcal{E}$. □

Remarks:

- This proof critically relies on the assumption that only a *single* register can be written atomically. What happens if it's possible to write concurrently to several?

3.5 Step 3: Reaching Contradiction

All that remains is to wrap things up.

Proof of Theorem 3.2. Assume for contradiction that a consensus algorithm $\mathcal{A}$ exists that tolerates a single fault, i.e., in all fair executions with at most one crash agreement, validity, and termination hold. By Lemma 3.7, there is a bivalent execution of $\mathcal{A}$ without crashes. By Lemma 3.9, we can extend any such execution to a bivalent execution without crashes that includes an additional step of an arbitrary node $i \in [n]$. We apply the lemma inductively in a round-robin fashion; in the k^{th} step of the induction, we add a step of node $k \mod n$. The result is an infinite, fair, bivalent execution without crashes. This contradicts the condition that the algorithm must terminate in *all* fair executions with at most one crash! □

3.6 How about Message Passing?

Fine, we can't do it in this shared memory setting. But is consensus possible in the asynchronous message passing model? At least for *some* graphs? To answer this question, we need to specify what it means that a node crashes in the message passing model.

Definition 3.10 (Crash Faults in the Message Passing Model)**.** *A node may crash at any point in the execution, after which it does not respond to any further events. It may also crash when responding to an event. In this case, it sends an arbitrary subset of the messages it would send if it did not crash.*

This definition takes into account that it's virtually impossible to make sure that a crashing node sends either everything or nothing – that would be very similar to writing *multiple* registers atomically! Each individual message *is*

sent and received "atomically," which is justified since any message that is not transmitted and received completely can simply be dropped.

It may not seem like it, but basically we already have the answer. We use a simulation argument!

Lemma 3.11 (Simulation of Message Passing)**.** *If, for any simple graph $G = (V, E)$, an asynchronous message passing algorithm solving consensus with at most one crash fault on G exists, then there is an asynchronous shared memory algorithm on $|V|$ nodes that solves consensus in all fair executions with at most one fault.*

Proof. We "translate" the message passing system to a shared memory system. We use the same set of nodes. For each edge $e = \{v, w\}$, we add registers $R_{v,w,i}$ and $R_{w,v,i}$, $i \in \mathbb{N}$, initialized to $\perp$ (meaning not used). For each neighbor w, v maintains two local counters $s_{v,w}$ and $r_{v,w}$, the number of sent and received messages for this node, respectively (initially 0). We simulate the message passing algorithm as follows. Initially, each node v performs its local computations and decides on the messages to send. Then, for each neighbor w to which it sends a message, it increases $s_{v,w}$ and writes the content of the message to $R_{v,w,s_{v,w}}$. Once this is complete, it executes a *busy-wait*. Cycling through its neighbors, w, it keeps reading $R_{v,w,r_{v,w}+1}$ until $R_{v,w,r_{v,w}+1} \neq \perp$ for some w. When this happens, it executes the code of the asynchronous algorithm for reception of a message with the content equal to that of $R_{v,w,r_{v,w}+1}$ from w and increases $r_{v,w}$. Resulting messages are resolved as above and the busy-wait recommences (unless the node terminates, of course).

It's straightforward to see that each "sent message" is eventually "received" (unless the receiving node terminates or crashes before this happens, which is ok), and since the shared memory algorithm does the same computations and "sends" the same messages, it will produce the same outputs as some corresponding execution of the message passing algorithm. Thus, agreement, validity, and termination of the shared memory version are inherited from the original algorithm. □

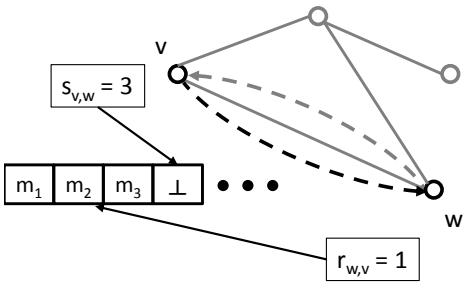

Figure 3.5: Construction for simulation of a message passing algorithm in shared memory. Depicted are only the registers for the edge $\{v, w\}$ for the direction from v to w. Node v will write each message to a new register using its local counter $s_{v,w}$. Node w will increase its counter $r_{v,w}$ whenever it reads a value that is not $\perp$, meaning it "received" the next message from v.

This lemma extends the previous impossibility to the asynchronous message passing model.

Corollary 3.12. *There is no algorithm that solves the consensus problem in the asynchronous message passing model with at most one crash fault.*

Proof. If such an algorithm existed, by Lemma 3.11 there would also be an algorithm solving consensus in all fair executions of the asynchronous shared memory model with at most one crash fault. By Theorem 3.2, such an algorithm does not exist. □

Remarks:

- We're doing something that might seem weird here. In the simulation, we use infinitely many shared registers (as there can be an unbounded number of messages under way in the message passing system), and these registers have infinite size (as messages may be arbitrarily large). However, we're talking about an impossibility result here: *Even* with such an impossible-to-build system, we *still* couldn't solve the problem!
- Note also that the simulation will actually ensure FIFO (first-in-first-out) order of message reception. Again, this makes the impossibility result only stronger. Also if message delivery is guaranteed to happen in FIFO order, the problem cannot be solved!
- Originally, the FLP theorem was shown for the message passing model. Showing it for shared memory and then using a simulation argument as done here is much simpler, yet we get the result for the more powerful shared memory model along the way!

What to take Home

- Knowing that certain things *cannot* be done is really important, as it keeps us from trying to do these things.
- Actually, it will not really keep us from trying, as it's important to solve these problems. However, such results show where one can change the model (i.e., add some helpful, hopefully realizable assumptions), so that they become solvable.
- Finding the right definitions can be the most important part of the job.
- Simulation arguments are also very powerful tools for lower bounds. FLP is a great example for this, as it's much easier to prove the result for shared memory and transfer it to message passing than taking the message passing model head on!

Bibliographic notes

Fischer, Lynch, and Patterson showed the original theorem about message passing systems, in a model slightly, but insubstantially different from the asynchronous message passing model given in Lecture 2 [FLP85]. Loui and Abu-Amara [LAA87] extended the result to the shared memory setting; strictly

speaking, Theorem 3.2 is to be attributed to them, but Fischer, Lynch, and Patterson developed the underlying technique. Later it was discovered that the impossibility of consensus and generalizations can be shown using topological tools [HS99].

Bibliography

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

[HS99] Maurice Herlihy and Nir Shavit. The Topological Structure of Asynchronous Computability. *J. ACM*, 46(6):858–923, 1999.

[LAA87] Michael C Loui and Hosame H Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4(163–183), 1987.

Lecture 4

Reaching Consensus

4.1 The Problem

In the previous lecture, we've seen that consensus can't be solved in asynchronous systems. That means asking "how can we solve consensus?" is pointless. But it doesn't mean we should give up! Impossibility results and lower bounds tell us what questions we should not ask, but at the same time, we learn a lot about the questions we *should* ask!

For starters, you saw in the exercises that the question "Can we solve consensus *in synchronous systems*?" makes a lot of sense. And having a positive answer to this already implies something very important: In the presence of faults, synchronous and asynchronous systems are fundamentally different. Not just in terms of message or time complexity, but in terms of what type of problems can be solved! In particular, we cannot hope for a convenient tool like synchronizers.

This immediately raises more important questions. Given that synchrony is a strong (read: often unrealistic) assumption, where exactly does the boundary between consensus being solvable and unsolvable lie? This has been studied quite extensively, but today we're going to focus on something else. We're asking the question

> "Can consensus be solved *almost* certainly despite asynchrony?"

Put differently, instead of making stronger assumptions on the system, we make our problem easier. We permit randomization[1] and relax the requirements of the binary consensus problem:

Agreement Correct nodes i output the same value $o = o_i$.

Validity If all corrrect nodes have the same input b, then $o = b$.

Termination *With probability 1,* all correct nodes decide on an output and terminate.

Wait a second – doesn't termination with probability 1 mean that it is *certain* that the algorithm will eventually terminate? This is not true, as seen by the

[1] Ok, ok. Depending on your point of view, we *do* make the system more powerful.

following simple example. We take an *unbiased* coin and flip it. If the result is "heads," we stop. Otherwise, we repeat this. In principle, it is possible that this goes on forever. However, the probability for this to happen is, for each $r \in \mathbb{N}$, bounded by the probability of getting "heads" r times in a row. This probability is 2^{-r}, which goes to 0 for $r \to \infty$. Hence, with probability 0, we never stop, but it is not impossible that we never stop!

Ok, so the above definition may make some sense. However, it *does* open a new can of worms. What do we even *mean* by probability here? An algorithm is not just a sequence of coin flips! Changing the system model such that the next node that makes a step or that crashes is random might be too optimistic (there could be systematic faults), and it's not clear what the respective probability distributions should look like.

4.2 The Model

The simplest way to resolve the above conundrum is to interpret randomized algorithms as deterministic ones with additional random input. Each node gets a separate input, which is an infinite string of independent, unbiased random bits (the node will generate the finite prefix of bits it uses during the course of the algorithm). The nodes now execute a "deterministic" algorithm that may, at any time, use some of these random bits to decide on its next step. Our probability space now consists of all the possible combinations of such strings with the probability measure induced by each individual bit being 0 with independent probability 1/2.

Formally, we can now say what "termination with probability 1" means as follows. We fix our randomized algorithm (i.e., our "deterministic" algorithm that is given the random input strings), draw the random strings from the distribution, and then check whether there is *any* execution for which termination is violated. The latter must happen with probability 0 in terms of the distribution of random input strings. Of course, we will also have to show that agreement and validity are *always* satisfied.

Moreover, we get rid of another, often unrealistic, assumption, namely that failing nodes "crash" nicely. In fact, we're going all the way, to so-called *Byzantine* failures. This means that failing nodes can behave in *any* way: They can crash, they can send erroneous and conflicting messages to different nodes, and they can violate the protocol in an arbitrary way. They can even stick to the protocol (for a while), making it hard or impossible to figure out that they are faulty. If you think this sounds crazy, you are not *completely* wrong. However, if we can handle this scenario, we don't have to worry about the type of faults that can actually happen!

Let's say we have f faulty nodes. Faulty nodes can pretend to be correct, but with different input. By the validity condition, this implies that in a fault-free execution, we must decide on 0 (or 1), if $n - f$ nodes have input 0 (or 1), respectively. This immediately implies that consensus cannot be solved if $f \geq n/2$! In fact, more careful reasoning shows that $f \geq n/3$ also breaks our necks: If we have two sets of f correct nodes with inputs 0 and 1, respectively, the faulty nodes can play to each set the role of correct nodes with their input; the correct nodes in each set then cannot figure out whether the respective other set of nodes is faulty or the real baddies, and thus cannot distinguish from $n-f$

correct nodes having input 0 respectively 1. The surprise is that, indeed, it is possible to handle any number f of faults strictly smaller than $n/3$! Throughout this lecture, we will hence assume that $f < n/3$.

Finally, we will not design our algorithms in the shared memory model from the previous lecture, but in the asynchronous message passing model. However, we assume that "everyone can talk to everyone," i.e., the communication graph is complete.

Remarks:

- One way to visualize the probability space we're using is to think of the input string of each node as encoding a uniformly random real number from $[0, 1]$. The first bit says whether it's from $[0, 0.5]$ or $[0.5, 1]$, etc.[2] The complete space is then the product of all the individual ones (for correct nodes), i.e., a hypercube of dimension $n - f$; the bit strings determine a uniformly random point in this hypercube.
- A subset of the hypercube with measure 0 is e.g. given by a side of the cube, which is equivalent to fixing a single node's string to be all 0s or all 1s. As we already saw, this happens with probability 0!
- Since this hypercube has exactly volume 1, the probability to draw bit strings corresponding to a given subset of the hypercube is exactly the volume of the subset. For instance, requiring that the first bit of some fixed node is 0 defines a "half-cube" and happens with probability 1/2.

4.3 First Thoughts and a Key Ingredient

Let's start with some basic observations:

- No matter how the algorithm is going to look like, we cannot trust that any information that comes from fewer than $f + 1$ nodes is definitely correct (unless there's a way to be sure that not all faulty nodes are among them).
- Since the system is asynchronous, waiting until this many messages arrived is the only way of doing this.
- However, one must not wait to hear from everyone! Byzantine nodes may decide to not send anything, so a node cannot be sure that more than $n - f$ messages arrive (if every node is supposed to send one)!
- We are going to think about our algorithms in terms of rounds again, where in each round, each node is supposed to send one and only one message to each other node. But as described above, each node can only expect to receive $n - f$ of these messages. You can view this as a "deteriorated" version of the α-synchronizer.
- It's important to understand that if the faulty nodes *do* send messages, it may be the good guys' messages that get discarded while waiting for the end of the round. Only $n - 2f$ of the messages a node receives for a round are certainly legit!

[2] This way we have several encodings for the same number, as, e.g., $0.1\overline{0} = 0.0\overline{1}$. Throwing some measure theory at this shows that this is ok, though.

- That means if actually all correct nodes agree on something (e.g., they have the same input), a node will "observe" at least $n - 2f$ being in agreement.
- Since the algorithm does not know f, it will have to use an upper bound on f whenever counting messages as above. We'll slightly abuse notation and still use the variable f in algorithms, but remember that in fact our algorithms should use a parameter t (instead of f) and the corresponding assumption in the analysis is that $f \leq t < n/3$.

Ok, so let's get started. Think about validity first. We need to make sure that if everyone has input 0, this is also the output – deterministically. So, all nodes announce their inputs; for simplicity, we assume that nodes also send messages "to themselves." If out of the $n - f$ messages received by a node, $n - 2f$ say "my input is 0," it's possible that all nodes had input 0. Say a node in this situation decides that it will output 0, tells the others, and terminates. This means that if *not* all inputs were 0, we need to make sure that agreement holds, i.e., everyone else decides on 0, too!

Given this insight, consider now the case that some correct node following the above rule decides on 0 and terminates. It must have received at least $n - 3f > 0$ messages saying "my input is 0" from correct nodes. That means that at least one node indeed had input 0, so this is alright. But a single node having input 0 is not enough to convince anyone else, as other nodes have no way of being certain that this node is not faulty and in fact all nodes had input 1!

To resolve this, let us be a bit more lenient and make the stronger assumption that $f < n/5$. Now, a correct node seeing $n - 2f$ times "0" means that $n - 3f$ correct nodes indeed have input 0. And since each node waits for all but f messages, this means each correct node receives $n - 4f \geq f + 1$ times "0." Now we let nodes that see $f + 1$ times "0" change their "opinion" to 0, even if their input was 1.

Algorithm 9 Subroutine for trying to decide on 0, code at node $i \in [n]$. Each node i is given an input "opinion" $op_i \in \{0, 1\}$, which initially will be the input of i.

```
1: send "op_i" to all nodes // also to yourself
2: wait until received "op_j" messages from n − f nodes // drop all but one
   message per node
3: if received ≥ n − 2f "0" messages then
4:     decide(0)
5: else if received ≥ n − 4f "0" messages then
6:     op_i := 0
7: end if
```

Let's summarize the properties of this basic procedure, under the assumption that $f < n/5$.

Lemma 4.1. *If a correct node decides on 0 in Algorithm 9, then all correct nodes have decided 0 or have opinion 0 at the end of the algorithm.*

Proof. The node received $n - 2f$ times "0," $n - 3f$ of which are from correct nodes. Since each node drops up to f messages, all nodes receive at least $n - 4f$ times "0" and hence execute either the IF or the ELSEIF statement. □

Lemma 4.2. *If all correct nodes have opinion 0 at the beginning of Algorithm 9, then all correct nodes decide 0.*

Proof. If all (at least $n - f$) correct nodes have opinion 0, each node receives at least $n - 2f$ times "0" and executes the IF statement. □

Lemma 4.3. *If all correct nodes have opinion 1 at the beginning of Algorithm 9 and $f < n/5$, then they all keep this opinion and none of them decide.*

Proof. If all correct nodes have opinion 1, no node will receive more than f times "0." As $n > 5f$, we have $n - 4f > f$, implying that no correct node executes the IF statement or the ELSEIF statement. □

Remarks:

- This is nice: we can achieve termination if all nodes agree on 0 without destroying an existing agreement on 1! And if some node decides, we can be sure that all nodes agree afterwards and will decide on 0 when running the subroutine again.
- Alternating with the "twin" algorithm which tries to agree on 1 makes sure that validity is satisfied. Since neither of the twins destroys an existing agreement, we have ensured agreement and validity.
- Unfortunately, it is possible that the algorithm doesn't terminate. We know from the last lecture that this cannot be avoided, since so far we haven't used randomness!

4.4 Shared Coins

We now have a way of deciding in a safe way. The issue is that we can guarantee that nodes decide only if all of them already have the same opinion. Hence, we need a mechanism to establish this common opinion if it's not initially present.

Definition 4.4 (Shared Coin)**.** *A* shared coin with defiance $\delta > 0$ *is a subroutine that generates at each node a bit such that the probability that the bit is 0 at all correct nodes is at least δ and the probability that it is 1 at all correct is at least δ. The coin is* strong*, if it is guaranteed that all nodes output the same bit. Otherwise it is* weak*.*

Remarks:

- A strong shared coin essentially generates a common "random" bit which may be influenced by the faulty nodes, but not too badly. One can interpret it as a random experiment in which with independent proability $1-2\delta$ an "adversary" determines the outcome, but with probability 2δ an unbiased coin is flipped to determine the result.
- If the coin is weak, the adversary can even make it so that if the first case applies, different correct nodes observe different outcomes. Of course that's less useful, but easier to achieve.

- Surprisingly, strong shared coins with constant defiance exist. Unfortunately, they all need some additional assumptions: states of and communication between correct nodes is secret (i.e., the faulty nodes may act based on the inputs and communication they have seen only), cryptography can be used to prevent messages to be understood until the sender reveals the respective key, etc.

- If one makes such additional assumptions, one must be more careful in defining the probability space over which one argues. We dodge this bullet today, by assuming that the adversary is *oblivious* to the random choices of the algorithm.

- If the adversary knows the random decisions of the correct nodes in advance, can you see how to *deterministically* prevent termination?

- You will show how to obtain a better shared coin in the exercises.

Lemma 4.5. *There is a weak shared coin with defiance 2^{-n} that requires no communication.*

Proof. All nodes flip an unbiased coin independently, i.e., output a fresh bit from their random input strings. The probability that all these bits are 0 (respectively 1) is 2^{-n}. □

Given this coin, we can solve the problem, can't we? Let's look at Algorithm 10. Once we used Algorithm 9 and its counterpart to make sure that we terminate with the right output if all inputs agreed, we simply flip the weak shared coin to obtain new opinions and repeat. Eventually, all nodes have the same opinion and will decide on the same output. Almost, except that we can get into trouble if some node already decided and then we mess up the opinions using the coin flips. We make sure that this doesn't happen by checking this with another voting step.

Lemma 4.6. *The last IF statement in the WHILE loop of Algorithm 10 does not change any opinions if all correct nodes had the same opinion. If the coin has defiance δ and $n > 5f$, all correct nodes have the same opinion after execution of the WHILE loop with probability at least δ.*

Proof. If all correct nodes agree, each of them receives at least $n - 2f$ messages with the same value; hence previous agreement is not destroyed.

For the second statement of the lemma, assume for contradiction that two correct nodes with different opinions ignore the shared coin.[3] This entails that each of them received $n - 3f$ messages with 0 respectively 1 from correct nodes. However,

$$2(n - 3f) = n - f + (n - 5f) > n - f,$$

i.e., there are not sufficiently many correct nodes to make this possible. We conclude that for at most one opinion value correct nodes may ignore the shared coin. With probability at least δ, the value of the shared coin at all correct nodes is this opinion value (if no opinion is fixed, having the same value at all nodes is good enough), yielding the second statement of the lemma. □

[3]Here we exploit that the adversary is oblivious of the nodes' randomness, and hence the outcome of the coin matches any predetermined opinion with probability δ!

Algorithm 10 Consensus algorithm using weak shared coin. Note that messages must be numbered, so receivers can figure out to which "round" a message belongs; for readability, this is omitted from the code.

```
1: op_i := b_i // set opinion to input
2: while not decided do
3:    send "op_i" to all nodes
4:    wait until received "op_j" messages from n − f nodes
5:    if received ≥ n − 2f "0" messages then
6:       decide(0)
7:    else if received ≥ n − 4f "0" messages then
8:       op_i := 0
9:    end if
10:   send "op_i" to all nodes
11:   wait until received "op_j" messages from n − f nodes
12:   if received ≥ n − 2f "1" messages then
13:      decide(1)
14:   else if received ≥ n − 4f "1" messages then
15:      op_i := 1
16:   end if
17:   compute a (weak) shared coin b_i
18:   send "op_i" to all nodes
19:   wait until received "op_j" messages from n − f nodes
20:   if received < n − 2f "op_i" messages then
21:      op_i := c_i // c_i ∈ {0, 1} uniformly random
22:   end if
23: end while
24: send the messages for the next iteration of the loop, where the opinion
    remains fixed to the decided value (i.e., do not wait)
25: terminate and output decided value
```

Theorem 4.7. *Given a weak shared coin and that $f < n/5$, Algorithm 10 solves consensus. All nodes terminate in expected time $\mathcal{O}(1/\delta)$, where δ is the defiance of the shared coin.*

Proof. Let us verify validity, agreement, and termination one by one.

Validity If all correct nodes have input 0, by Lemma 4.2, in the first "round" of the first loop iteration all nodes decide on 0. If all correct nodes have input 1, by Lemma 4.3, no opinions change and no node decides in the first round. Applying Lemma 4.2, all correct nodes then decide on 1 in the second round.

Agreement By Lemma 4.1, once a correct node decides on value b, all correct nodes adopt opinion b. By Lemmas 4.3 and 4.6, no correct node will change its opinion any more after this happens. Hence, no correct node can observe more than $f < n - 2f$ messages $1 - b$ in any future "round." Thus, no correct node will decide $1 - b$.

Termination First, note that no correct node gets "stuck" waiting for $n - f$ messages of a "round" provided that all correct nodes send a message for

that round. By the previous observations, if some correct node decides, all nodes will adopt that opinion and in the next iteration of the loop they will decide on the respective value by Lemma 4.2. As deciding nodes will complete the current iteration of the loop and then send the (known) messages for the next iteration, the algorithm will thus terminate provided that some node decides. By Lemma 4.6, with probability δ all nodes have the same opinion at the end of the loop, irrespectively of previous iterations. Once this happens, all nodes decide in the next loop iteration by Lemmas 4.2 and 4.3. Thus, as the probability that an infinite number of shared coins "fail" $\lim_{k\to\infty}(1-\delta)^k = 0$, the algorithm terminates with probability 1.

Finally, we bound the expected time complexity. In each iteration, every correct node constantly often sends a message to every node and waits for $n - f$ responses. Since messages are sent in parallel, it takes $\mathcal{O}(1)$ time per round and thus $\mathcal{O}(k)$ time for k rounds. The probability that "the algorithm decides" in round k (i.e., all nodes obtain the same opinion) is $\delta(1-\delta)^{k-1}$; once this happens, all nodes terminate within $\mathcal{O}(1)$ additional rounds. All of this means that the expected time until termination is bounded by

$$\sum_{k=1}^{\infty} \mathcal{O}(k) \cdot \delta(1-\delta)^{k-1} = \mathcal{O}\left(\frac{\delta}{\delta^2}\right) = \mathcal{O}\left(\frac{1}{\delta}\right). \qquad \square$$

Corollary 4.8. *Using randomness and given that $f < n/5$, consensus can be solved in asynchronous systems with Byzantine faults with probability 1.*

Proof. Plug the weak shared coin from Lemma 4.5 into Theorem 4.7. $\square$

Remarks:

- Two things are not satisfying here. One is that using the simplistic shared coin from Lemma 4.5, the expected running time is astronomically large: for many inputs, it's $\Theta(2^n)$. The other is that we can tolerate only $f < n/5$ faults, but it might be possible to handle $f < n/3$.
- We will now see how to get down to $f < n/4$ using a new subroutine, safe broadcast. Combining the various voting ideas and a strong shared coin in a more clever way, one can indeed get down to $f < n/3$.
- There was some mild cheating: I silently omitted discussing what happens if the weak shared coin communicates and expects correct nodes to send messages. This is easy to resolve by making sure that the shared coin can somehow terminate using default messages from correct nodes (without maintaining any guarantees on the output), as it becomes irrelevant as soon as some node terminated.

4.5 Safe Broadcast

One of the issues we had in Algorithm 10 was that faulty nodes could announce different values to different nodes. Let's take this power away! This is essentially the following task, named *safe broadcast*:

- The source node s is globally known.
- s is given a message M.
- Each correct node i outputs at most one message M.
- If s is correct, all correct nodes eventually output M.
- If any correct node outputs a message M', all correct nodes eventually output M'.

Using this subroutine as a wrapper for each message sent, we can, in effect, force each faulty node to either send the same message to all nodes in a given "round," or just stay silent. Let's see how we can use this to improve Algorithm 10.

Algorithm 11 Consensus algorithm using weak shared coin and safe broadcasts.

1: $op_i := b_i$ // set opinion to input
2: **while** not decided **do**
3: safely broadcast "op_i"
4: wait until received "op_j" messages from $n - f$ nodes
5: **if** received $\geq n - 2f$ "0" messages **then**
6: decide(0)
7: **else if** received $\geq n - 3f$ "0" messages **then**
8: $op_i := 0$
9: **end if**
10: safely broadcast "op_i"
11: wait until received "op_j" messages from $n - f$ nodes
12: **if** received $\geq n - 2f$ "1" messages **then**
13: decide(1)
14: **else if** received $\geq n - 3f$ "1" messages **then**
15: $op_i := 1$
16: **end if**
17: compute a (weak) shared coin b_i
18: safely broadcast "op_i"
19: wait until received "op_j" messages from $n - f$ nodes
20: **if** received $< n - 2f$ "op_i" messages **then**
21: $op_i := b_i$
22: **end if**
23: **end while**
24: safely broadcast the messages for the next iteration of the loop, where the opinion remains fixed to the decided value (i.e., do not wait)
25: terminate and output decided value

Corollary 4.9. *Given a weak shared coin, safe broadcast, and that $f < n/4$, Algorithm 11 solves consensus. All nodes terminate in expected time $\mathcal{O}(1/\delta)$, where δ is the defiance of the shared coin.*

Proof. The reasoning is the same as before, so we need to revisit only the steps that required that $f < n/5$ in the previous proofs, as well as check the effect of

the $n-3f$ thresholds that replaced $n-4f$ thresholds. This concerns Lemmas 4.1, 4.3, and Lemma 4.6.

Regarding Lemma 4.1, observe that if a node receives $n-2f$ times "0," then at most $2f$ nodes – including faulty ones! – broadcast "1." Hence, any $n-f$ received values contain $n-3f$ times "0," i.e., all correct nodes adopt opinion 0. For Lemma 4.3, we just need to note that $n-3f > f$ holds, so the faulty nodes will not be able to change anyone's opinion if all correct nodes have opinion 1. Finally, Lemma 4.6 still holds because two nodes with different opinions ignoring the shared coin would mean that there have been at least $2(n-f) = n+(n-4f) > n$ many distinct broadcast messages in the respective round. □

Remarks:

- Altough the broadcast problem is very similar to consensus (the input is send to nodes by the source), there is a crucial difference: it is ok that no correct node "terminates," i.e., no correct node outputs a message. That's all it takes to make it much simpler.
- In the exercises, you will solve the safe broadcast problem.
- By adding the round number and source's identifier to the messages sent by the safe broadcast algorithm, we can tell the instances apart. Any "excess" messages for a given round number/source ID combination are simply discarded. The same holds for communication that is obviously not conform with the algorithm.
- By introducing the safe broadcast statements into the code, I also introduced a bug: the algorithm may not terminate anymore! Can you see why?
- Can you come up with a fix? This is not too difficult, but again underlines how careful one needs to when reasoning about asynchronous algorithms.

What to take Home

- Randomization can work miracles in distributed systems. Maybe even more so than in case of "classic" centralized algorithms!
- Impossibility results and lower bounds usually can and will be circumvented by changing some details – often those that we do not mind in practice or that are the smallest burdens. Only sometimes one hits hard walls, as with the $\log^*$ lower bound for coloring (but this one we don't mind in practice either).
- The running times we get here are terrible. However, strong shared coins with constant defiance and large resilience exist! This yields algorithms that can solve consensus in constant expected time, and with overwhelming probability in time $\mathcal{O}(\log n)$!
- Dealing with Byzantine faults always involves voting using thresholds requiring, e.g., $n-f$ or $n-2f$ nodes claiming a certain opinion.

- What you've seen here are some core ideas for handling Byzantine faults. There are myriads of variations and generalizations of the consensus problem or other fault-tolerance primitives. The basic ideas and the approach to reasoning about Byzantine faults you've seen today typically resurface in one way or another when considering these.
- One needs to be extremely careful in terms of how randomness and the adversary may interact. For instance, even if the adversary cannot magically predict future coin flips, node may reveal the outcomes of these coin flips prematurely due to asynchrony. This is especially important when designing systems that are supposed to be resilient against deliberate attacks.

Bibliographic Notes

The consensus algorithm presented in this lecture is a variant of Bracha's algorithm [Bra87]. However, Bracha mistakenly claimed that the algorithm tolerates $f < n/3$ faults. The issue is that in Bracha's algorithm nodes terminate only when receiving at least $2f+1$ messages supporting the decided value in a certain step, but for $n = 3f + 1$ this is exactly $n - f$; given that Byzantine nodes can always claim a different value and only $n - f$ messages are evaluated (otherwise the algorithm might deadlock), termination can never be guaranteed.

For a solution that tolerates $f < n/3$ faults using a strong shared coin, see Mostéfaoui et al. [MMR14]. A cryptographically safe, optimally resilient, constant expected time strong shared coin is given by Cachin et al. [CKS05]. Together this yields a constant expected time optimally resilient solution to consensus. Both algorithms are also very efficient in terms of messages. The caveat is that a trusted dealer is needed in a setup phase for the shared coin.

Bibliography

[Bra87] Gabriel Bracha. Asynchronous Byzantine Agreement Protocols. *Information and Computation*, 75(2):130–143, November 1987.

[CKS05] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[MMR14] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free Asynchronous Byzantine Consensus with $t < n/3$ and $\mathcal{O}(n^2)$ Messages. In *Proc. 34th Symposium on Principles of Distributed Computing (PODC)*, pages 2–9, 2014.

Lecture 5

Maximal Independent Set

5.1 The Problem

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

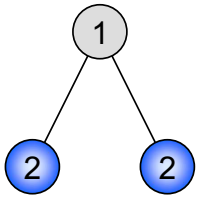

Figure 5.1: Example graph with 1) a maximal independent set and 2) a maximum independent set.

From a centralized perspective, an MIS is an utterly boring structure: Just pass over all nodes and greedily select them, i.e., pick every node without a previously selected neighbor. Voilà, you've got an MIS! However, in the distributed setting, things get more interesting. As you saw in the first exercise, finding an MIS on a list is essentially the same as finding a 3-coloring, and if we don't care about message size, being able to find an MIS on general graphs in T rounds can be used to determine a $(\Delta + 1)$-coloring in T rounds. This means that despite it being easy to *verify* that a set is an MIS locally (i.e., some node will notice if there's a problem just by looking at its immediate neighborhood), it takes $\Omega(\log^* n)$ rounds to compute an MIS!

Today, we're going to study this problem in the synchronous message passing model (without faults), for an arbitrary simple graph $G = (V, E)$. Recall that

we can use synchronizers to transform any algorithm in this model into an asynchronous algorithm, so the result will make a very versatile addition to our toolbox! Like last week, we're going to allow for randomness. This can be done in the same way, by assuming that each node has an infinite (read: sufficiently large) supply of unbiased random bits.

Remarks:

- Note that an MIS can be very small compared to a maximum independent set. The most extreme discrepance occurs in a star graph, where the two possible MIS are of size 1 and $n-1$ (cf. Figure 5.1)!
- One can apply a coloring algorithm first and then, iterating over colors, concurrently add all uncovered nodes of the current color to the independent set. Once this is complete, all nodes are covered, i.e., we have an MIS. However, this can be quite slow, as the number of colors can be large.

5.2 Fast MIS Construction

Algorithm 12 MIS construction algorithm. Of course we cannot implement an algorithm that operates with real values. We'll fix this later.

```
// each iteration of the while-loop is called a phase
  while true do
    choose a uniformly random value r(v) ∈ [0, 1] and send it to all neighbors
    if r(v) < r(w) for each r(w) received from some neighbor w then
      notify neighbors that v joins the independent set
      return(1) and terminate
    end if
    if a neighbor joined the independent set then
      return(0) and terminate
    end if
  end while
```

Lemma 5.2. *Algorithm 12 computes an MIS. It terminates with probability 1.*

Proof. We claim that at the beginning of each phase, the set of nodes that joined the set and terminated is an independent set, and the set of all nodes that terminated is the union of their (inclusive) neighborhoods; we show this by induction. Trivially, this holds initially. In each phase, it cannot happen that two adjacent non-terminated nodes enter the set. By the induction hypothesis, no active (i.e., not terminated) node has a neighbor in the independent set. Together this implies that at the end of the phase, the set of nodes that output 1 is still an independent set. As the active neighbors of joining nodes output 0 and terminate, the induction step succeeds and the claim holds true. We conclude that the algorithm computes an independent set.

To see that the independent set is maximal, observe that a node can only terminate if it enters the set or has a neighbor in the set. Thus, once all nodes

have terminated, no further nodes could be added to the set without violating independence.

Finally, note that the probability that two random real numbers from $[0, 1]$ are identical is 0. By the union bound, this yields that with probability 1, in a given phase all random numbers are different. This entails that the non-terminated node with the smallest number joins the set. This implies that within finitely many phases in which the numbers differ, all nodes terminate. Using the union bound once more, it follows that the algorithm terminates with probability 1. □

Remarks:

- Simple stuff, but demonstrating how to reason about such algorithms.
- The *union bound* states that the probability of one (or more) of several (more precisely: countably many) events happening is at most the sum of their individual probabilities. It is tight if the events are disjoint.
- The union bound can't even be called a theorem. It's obvious for discrete random variables, and for continuous random variables it's just paraphrasing the fact that the total volume of the union of countably many sets is bounded by the sum of their individual volumes, a property that any measure (in particular a probability measure) must satisfy.
- We'll frequently use the union bound implicitly in the future.
- That's enough with the continuous stuff for today, now we're returning to "proper" probabilities.
- Note that the algorithm can be viewed as selecting in each phase an independent set in the subgraph induced by the still active nodes. This means that all we need to understand is a single phase of the algorithm on an arbitrary graph!

5.3 Bounding the Running Time of the Algorithm

Before we can do this, we need a (very basic) probabilitistic tool: linearity of expectation.

Theorem 5.3 (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. We'll do the proof for a discrete random variable; for a continuous one, simply replace the sums

by integrals. We compute

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

where in the second last step we used that

$$\sum_{X=x} P[X=x \mid Y=y] = \sum_{Y=y} P[Y=y \mid X=x] = 1,$$

as X and Y are random variables (i.e., *something* happens with probability 1, even when conditioning on another variable's outcome). □

Denote by $N_v := \{w \in V \mid \{v,w\}\} \in E$ the neighborhood of $v \in V$ and by $\delta_v := |N_v|$ its *degree*. Now it's alluring to reason as follows. Since $\delta_v + 1$ nodes are removed "because of v" if v joins the set, by linearity of expectation we have that

$$\sum_{v \in V} P[v \text{ joins the set}] \cdot (\delta_v + 1) = \sum_{v \in V} \frac{\delta_v + 1}{\delta_v + 1} = |V| \qquad \text{wrong!!}$$

nodes are removed in expectation. This is utter nonsense, as it would automatically mean that all nodes are eliminated in a single step (otherwise the expectation must be smaller than $|V|$), which clearly is false!

The mistake here is that we counted eliminated nodes multiple times: it's possible that *several* neighbors of a node join the set. In fact, there are graphs in which only a small fraction of the nodes gets eliminated in a phase, see Figure 5.2.

In summary, we cannot hope for many nodes being eliminated in each phase. We might be able to reason more carefully, over multiple phases, but how? It turns out that the easiest route actually goes through figuring out the expected number of *edges* that are deleted (i.e., one of their endpoints is deleted) from the graph in a given phase. Still, we need to be careful about not counting deleted edges repeatedly!

Lemma 5.4. *In a single phase, we remove at least half of the edges in expectation.*

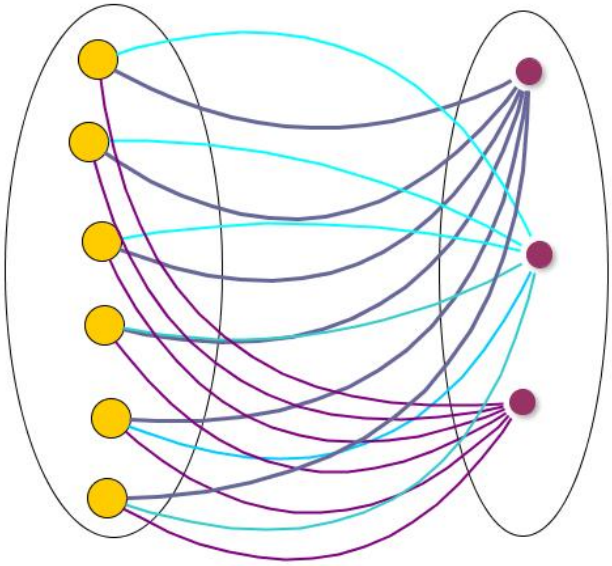

Figure 5.2: A graph where it's highly unlikely that more than a small fraction of the nodes gets deleted in the first phase. The graph is fully bipartite, but with few nodes on the right side. If $R \ll n$ nodes are on the right, the expected number of such nodes that gets selected is $R/(n-R+1) \ll 1$. This means the probability that at least one node on the right is selected is at most $R/(n-R+1) \ll 1$, too. From the left side, in expectation $(n-R)/R$ nodes are selected. If $1 \ll R \ll n$, this means that it's very likely that only nodes on the left side are selected and the selected nodes are much fewer than $n-R$; as $R \ll n$, the fact that all nodes on the right side are deleted does not affect the tally substantially.

Proof. To simplify the notation, at the start of our phase, denote the subgraph induced by the non-terminated nodes by $G = (V, E)$, i.e., ignore all terminated nodes and their incident edges. In addition, think of each undirected edge $\{v, w\}$ as being replaced by the two directed edges (v, w) and (w, v); we make sure that we count each such directed edge at most once when bounding the expected number of removed edges.

Suppose that a node v joins the MIS in this phase, i.e., $r(v) < r(w)$ for all neighbors $w \in N_v$. Consider a fixed $w \in N_v$. If we also have $r(v) < r(x)$ for all $x \in N_w \setminus \{v\}$, we call this event $(v \to w)$. The probability of event $(v \to w)$ is at least $1/(\delta_v + \delta_w)$, since (i) $\delta_v + \delta_w$ is the maximum number of nodes adjacent to v or w (or both) and (ii) ordering by $r(\cdot)$ induces a uniformly random permutation on V. As v joins the MIS, all (directed) edges (w, x) with $x \in N_w$ will be removed; there are δ_w such edges.

We now count the removed (directed) edges. 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 δ_w, if not it has the value 0. For each undirected edge $\{v, w\}$, we have two such variables, $X_{(v \to w)}$ and $X_{(w \to v)}$. Due to Theorem 5.3, the expected value of the sum X of all these random variables

is at least

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

In other words, in expectation $|E|$ directed edges are removed in a single phase! Note that we did not count any edge removals twice: we attribute the directed edge (v,w) being removed to an event $(u\to v)$, which inhibits a concurrent event $(u'\to v)$, because then $r(u) < r(u')$ for all $u'\in N_v$. We may have counted an undirected edge at most twice (once in each direction). So, in expectation at least half of the undirected edges are removed. □

This tells us that in expectation we make good progress. But how do we derive some explicit bound on the time until *all* edges are eliminated (and thus all nodes are, too) from this? We need another very basic tool, Markov's inequality.

Theorem 5.5 (Markov's Inequality)**.** *Let X be a non-negative random variable (in fact, $P[X\geq 0]=1$ suffices). Then, for any $K>1$,*

$$P[X\geq K\mathbb{E}[X]]\leq\frac{1}{K}.$$

Proof. We'll prove the statement for a discrete random variable with values from $\mathbb{N}_0$; it's straightforward to generalize to arbitrary discrete variables and continuous variables. If $P[X=0]=1$, the statement is trivial; hence, assume w.l.o.g. that $P[X=0]<1$, implying that $\mathbb{E}[X]>0$. We bound

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{i=0}^{\infty} P[X=i]\cdot i \\
&\geq \sum_{i=\lceil K\mathbb{E}[X]\rceil}^{\infty} P[X=i]\cdot i \\
&\geq K\mathbb{E}[X]\sum_{i=\lceil K\mathbb{E}[X]\rceil}^{\infty} P[X=i] \\
&= K\mathbb{E}[X]P[X\geq K\mathbb{E}[X]].
\end{aligned}$$

Dividing by $K\mathbb{E}[X]>0$ yields the statement of the theorem. □

Using Markov's bound, we can now infer that the bound on the expected number of removed edges in a given phase also implies that a constant fraction of edges must be removed with constant probability.

Corollary 5.6. *In a single phase, we remove at least a third of the edges with probability at least* $1/4$.

Proof. Fix a phase. For simplicity, denote the remaining graph by $G = (V, E)$. Denote by X the random variable counting the number of removed edges and by $\bar{X}$ the random variable counting the number of surviving edges, i.e., $|E| = X + \bar{X}$. By Lemma 5.4,

$$\mathbb{E}[\bar{X}] = |E| - \mathbb{E}[X] \leq \frac{|E|}{2}.$$

By Markov's inequality (Theorem 5.5, for $K = 4/3$),

$$P\left[X \leq \frac{|E|}{3}\right] = P\left[\bar{X} \geq \frac{2|E|}{3}\right] \leq P\left[\bar{X} \geq \frac{4\mathbb{E}[\bar{X}]}{3}\right] \leq \frac{3}{4}.$$

Hence,

$$P\left[X > \frac{|E|}{3}\right] = 1 - P\left[X \leq \frac{|E|}{3}\right] \geq \frac{1}{4}. \qquad \square$$

In other words, we translated the bound on the expected number of eliminated edges into a bound on the probability that a constant fraction of the remaining edges gets eliminated. This now can happen only $\mathcal{O}(\log n)$ times before we run out of edges!

Theorem 5.7. *Algorithm 12 computes an MIS in* $\mathcal{O}(\log n)$ *rounds in expectation.*

Proof. Irrespectively of how the remaining graph looks like, Corollary 5.6 tells us that with probability at least $1/4$, at least a fraction of $1/3$ of the remaining edges is removed in each phase. Let's call a phase in which this happens *good*. After

$$\log_{3/2} |E| < \frac{\log n^2}{\log(3/2)} < 4 \log n$$

good phases, we can be sure that there is at most a single remaining edge; in the following phase the algorithm will terminate. As each phase requires 2 rounds, it thus suffices to show that the expected number of phases until $4 \log n$ good phases occurred is in $\mathcal{O}(\log n)$.

Computing this expectation precisely is tedious. However, consider, say, $40 \log n$ phases. The expected number of good phases in $40 \log n$ phases is, by Corollary 5.6, at least $10 \log n$. Hence, the expected number of phases that are not good is at most $30 \log n$, and by Markov's inequality the probability that more than $36 \log n$ phases are bad is at most $30/36 = 5/6$. In fact, this holds for *any* set of $40 \log n$ phases. Consequently, the expected number of phases until at least $4 \log n$ good phases occurred is bounded by

$$\begin{aligned} 40 \log n \sum_{i=0}^{\infty} \left(\frac{5}{6}\right)^i \cdot \frac{1}{6} \cdot (i+1) &= \frac{20 \log n}{3} \left(\sum_{i=0}^{\infty} i \left(\frac{5}{6}\right)^i + \sum_{i=0}^{\infty} \left(\frac{5}{6}\right)^i \right) \\ &= \frac{20 \log n}{3} \left(\frac{5}{6} \cdot 6^2 + 6 \right) \\ &= 240 \log n. \end{aligned}$$

We conclude that the expected running time of the algorithm is bounded by $240 \log n \in \mathcal{O}(\log n)$ rounds. $\square$

Remarks:

- This analysis is somewhat heavy-handed with respect to constants. I'd guess that the probability to eliminate at least half of the edges is at least $1/2$, and that the algorithm terminates in expected time smaller than $4 \log n$. Proving this, however, might be quite difficult!

- The analysis is also tight up to constants. If one samples uniformly at random from the family of graphs of uniform degree Δ for, say, $\Delta = n^{1/100}$, there will be extremely few short cycles. In such a graph, it's easy to show that the degree of surviving nodes falls almost exactly by a constant factor in each phase (until degrees become fairly small).

- No algorithm that has expected running time $o(\log n)$ on all graphs is known to date. Plenty of research has been done and better bounds for many restricted graphs classes (like, e.g., trees) are known, but nothing that *always* works.

- The strongest known lower bound on the number of rounds to compute an MIS is $\Omega(\sqrt{\log n / \log \log n})$ or $\Omega(\log \Delta / \log \log \Delta)$ (depending on whether one parametrizes by the number of nodes or the maximum degree). It holds also for randomized algorithms; closing this gap is one of the major open problems in distributed computing.

- Embarrassingly, the best known upper bound on the time to compute an MIS deterministically is $2^{\mathcal{O}(\sqrt{\log n})}$, i.e., there's an exponential gap to the lower bound! Even worse, the respective algorithm may end up collecting the topology of the entire graph locally, i.e., using huge messages!

5.4 Exploiting Concentration

We have bounded the expected running time of the algorithm, but often that is not too useful. If we need to know when we can start the next task and this is not controlled by termination of the algorithm (i.e., we need to be ready for something), we need to know that all nodes are essentially *certain* to be finished! Also, there are quite a few situations in which we don't have such a trivial-to-check local termination condition. Yet, we would like to have an easy way of deciding when to stop!

Definition 5.8 (With high probability (w.h.p.))**.** *We say that an event occurs* with high probability (w.h.p.)*, if it does so with probability at least $1 - 1/n^c$ for any (fixed) choice of $c \geq 1$. Here c may affect the constants in the $\mathcal{O}$-notation because it is considered a "tunable constant" and usually kept small.*

This weird definition asks for some explanation. The reason why the probability bound depends on n is that it makes applying the union bound *extremely* convenient. Think for instance that you showed that each node terminates w.h.p. within $\mathcal{O}(\log n)$ rounds (the constant c being absorbed by the $\mathcal{O}$-notation). Then you pick $c' := c + 1$, apply the union bound over all nodes and conclude that *everyone* terminates with probability at least $1 - 1/n^{c'} \cdot n = 1 - 1/n^c$, i.e., w.h.p.!

The cool thing here is that this works for any polynomial number of events, as c is "tunable." For instance, if something holds for each edge w.h.p., it holds for all edges w.h.p. Even better, we can condition on events that happen w.h.p. and basically pretend that they occur deterministically. The probability that they do not happen is so small that any dependencies that might exist have negligible effects!

After this sales pitch, the obvious question is where we can get such strong probability bounds from. Chernoff's bound comes to the rescue! It holds for sums of *independent* random variables.

Definition 5.9 (Independence of random variables)**.** *A set of random variables $X_1, \dots, X_k$ is* independent*, if for all i and $(x_1, \dots, x_k)$ it holds that*

$$\begin{aligned} & P[X_i = x_i] \\ = \; & P[X_i = x_i \,|\, (X_1, \dots, X_{i-1}, X_{i+1}, \dots, X_k) = (x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_k)]. \end{aligned}$$

In words, the probability that $X_i = x_i$ is independent of what happens to the other variables.

Theorem 5.10 (Chernoff's Bound)**.** *Let $X = \sum_{i=1}^{k} X_i$ be the sum of k independent Bernoulli (i.e., 0-1) variables. Then, for $0 < \delta \leq 1$,*

$$\begin{aligned} P\big[X \geq (1+\delta)\mathbb{E}[X]\big] &\leq e^{-\delta^2 \mathbb{E}[X]/3} \\ P\big[X \leq (1-\delta)\mathbb{E}[X]\big] &\leq e^{-\delta^2 \mathbb{E}[X]/2}. \end{aligned}$$

Let's see Chernoff's bound in action.

Corollary 5.11. *Algorithm 12 terminates w.h.p. in $\mathcal{O}(\log n)$ rounds.*

Proof. By Lemma 5.4, with probability at least $1/4$, a third of the remaining edges is removed from the graph in a given phase. This bound holds *independently* of anything that happened before! We reason as in the proof of Theorem 5.7, but bound the number of phases until $4 \log n$ phases are good using Chernoff's bound.

For $c \geq 1$, the probability that we need more than $k := 32\lceil c \log n \rceil$ phases for $4 \log n$ of them to be good is bounded by the probability that the sum $X := \sum_{i=1}^{k} X_i$ of independent Bernoulli variables X_i with $P[X_i = 1] = 1/4$ is smaller than $4 \log n$. We have that $\mathbb{E}[X] = 8\lceil c \log n \rceil$. Hence, by Chernoff's bound for $\delta = 1/2$,

$$P[X < 4 \log n] \leq P\left[X < \frac{\mathbb{E}[X]}{2}\right] \leq e^{-\mathbb{E}[X]/8} \leq e^{-c \log n} < n^{-c}.$$

Thus, the probability that the algorithm does *not* terminate within $2(k+1) \in \mathcal{O}(\log n)$ rounds is at least $1 - 1/n^c$. Since $c \geq 1$ was arbitrary, this proves the claim. □

Remarks:

- Chernoff's bound is *exponentially* stronger than Markov's bound. However, it requires independence!
- Surprisingly, in fact Chernoff's bound is just a very clever application of Markov's bound (see exercises)!
- **Careful:** Pairwise independence of random variables $X_1, \dots, X_k$ does *not* imply that they are independent! A counterexample are two independent Bernoulli variables X_1 and X_2 with $P[X_1 = 1] = P[X_2 = 1] = 1/2$ (i.e., unbiased coin flips), and $X_3 := X_1$ XOR X_2. If one fixes *either* X_1 or X_2, X_3 is determined by the respective other (independent) coin flip, and hence remains independent. If one fixes both X_1 and X_2, X_3 is already determined!

5.5 Bit Complexity of the Algorithm

We still need to fix the issue of the algorithm using random real numbers. We don't want to communicate an infinite number of bits! To resolve this, recall the relation between uniformly random real numbers from $[0, 1]$ and infinite strings of unbiased random bits: The latter is a binary encoding of the former. Next, note that in order to decide whether $r_v > r_w$ or $r_v < r_w$ for neighbors $v, w \in V$ ($r_v = r_w$ has probability 0 and thus does not concern us here), it is sufficient to communicate only the leading bits of both strings, until the first differing bit is found! What is the expected number of sent bits? Easy:

$$\sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^i \cdot i = 2.$$

Inconveniently, the number of bits that need to be exchanged between each pair of nodes among three nodes that form a triangle are not independent. This is a slightly more elaborate example showing that pairwise independence does not imply "collective" independence.

Our way out is using that the probability to exchange many bits is so small that the dependence does not matter, as it also must become very small. The probability that a pair of nodes needs to exchange more than $1 + (c + 2) \log n$ bits in a given phase is

$$2^{-(c+2)\log n} = n^{-(c+2)}.$$

By the union bound, the probability that *any* pair of nodes needs to exchange more than this many bits is thus no larger than n^{-c} (there are fewer than n^2 edges). Applying the union bound over all rounds, we can conclude it suffices for each node to broadcast $\mathcal{O}(\log n)$ bits per round. But we can do better!

Corollary 5.12. *If n is known, Algorithm 12 can be modified so that it sends 1-bit messages and terminates within $\mathcal{O}(\log n)$ rounds w.h.p.*

Proof. Before running Algorithm 12, each pair of neighbors v, w determines for each of the $\mathcal{O}(\log n)$ phases the algorithm will require whether $r_v < r_w$ or vice versa. This is done by v sending the leading bits of r_v (and receiving the ones from w) until the first difference is noted. Then the nodes move on to

exchanging the leading bits for the next phase, and so on. The total number of bits that needs to be exchanged is in expectation $\mathcal{O}(\log n)$ (2 times the number of phases). By Chernoff's bound, for any fixed $K \geq 2$ the probability that the sum X of $K \log n$ independent unbiased coin flips is smaller than $K \log n/2$ is in $n^{-\Omega(K)}$. Note that the probability that the exchange for a given phase ends is indeed $1/2$ and independent of earlier such events. Choosing K suitably, we conclude that w.h.p. v and w exchange at most $\mathcal{O}(\log n)$ bits in total.

Applying the union bound, we conclude that w.h.p. *no* pair of nodes exchanges more than $\mathcal{O}(\log n)$ bits. Knowing n, we can determine the respective number, run the exchange algorithm for the respective number of rounds, and then run Algorithm 12 without having to communicate the values r_v, $v \in V$, at all. □

Remarks:

- Using this trick means that nodes need to communicate *different* bits to different neighbors.
- It's possible to get rid of needing to know n.[1]
- It's not hard to see that at least some nodes will have to send $\Omega(\log n)$ bits across an edge: in a graph consisting of $n/2$ pairs of nodes connected by an edge, the expected number of edges for which $\log(n/2)$ random bits are required to break symmetry is 1.

5.6 Applications

We know from the first exercise that we can use an MIS algorithm to compute a $(\Delta + 1)$-coloring.

Corollary 5.13. *On any graph G, a $(\Delta + 1)$-coloring can be computed in $\mathcal{O}(\log n)$ rounds w.h.p.*

MIS are not only interesting for graph coloring. As we will see in the exercises, they can also be very helpful in finding small *dominating sets*[2] in some graph families. Moreover, an MIS algorithm can be used to compute a *maximal matching*.

Definition 5.14 (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 incident to 2 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.*

[1] This is done by alternating between bit exchange rounds and rounds of Algorithm 12. However, then some nodes may not yet have succeeded in comparing the random values for the "current" phase of Algorithm 12. This can be understood as the exchange of the random numbers happening asynchronously (nodes do not know when they will be ready to compare their value to all neighbors), and thus can be resolved by running the α-synchronizer version of Algorithm 12. Now one needs to avoid that the communication of the synchronizer dominates the complexity. Solution: Exploiting synchrony, we can count "synchronizer rounds" by locally counting how many rounds neighbors completed and just communicating "I advance to the next round" (plus the respective message content) and otherwise remaining silent.

[2] A dominating set $D \subseteq V$ satisfies that each $v \in V \setminus D$ has a neighbor in D.

Corollary 5.15. *On any graph G, a maximal matching can be computed in $\mathcal{O}(\log n)$ rounds w.h.p.*

Proof. For a simple graph $G = (V, E)$, the line graph is (E, L), where $\{e, e'\} \in L$ if and only if $e \cap e' \neq \emptyset$. In words, edges become nodes, and a pair of "new" nodes is connected by an edge exactly if the corresponding edges in the original graph share an endpoint. We simulate an MIS algorithm on the line graph. It's straightforward to show that this can be done at an overhead of factor 2 in time complexity. □

Another use of MIS is in constructing small *vertex covers*.

Definition 5.16 (Vertex cover)**.** *A vertex cover $C \subseteq V$ is a set of nodes covering all edges, i.e., for all $e \in E$, $e \cap C \neq \emptyset$. It is minimal, if no node can be removed without violating coverage. It is minimum, if it is of minimum size.*

Corollary 5.17. *On any graph G, a vertex cover that is at most a factor 2 larger than a minimum vertex cover can be computed in $\mathcal{O}(\log n)$ rounds w.h.p.*

Proof. Compute a maximal matching using Corollary 5.15 and output the endpoints of all its edges. Clearly, this is a vertex cover, as an uncovered edge could be added to the matching. Moreover, any vertex cover must contain for each matching edge at least one of its endpoints. Hence the cover is at most a factor of 2 larger than the optimum. □

Finally, we can also use a maximal matching to approximate a maximum matching.

Corollary 5.18. *On any graph G, a matching that is at most a factor 2 smaller than a maximum matching can be computed in $\mathcal{O}(\log n)$ rounds w.h.p.*

Proof. Compute and output a maximal matching using Corollary 5.15. Now consider a maximum matching. A minimum vertex cover must be as least as large as this matching (as no node can cover two matching edges). As the endpoints of the edges of a maximal matching form a vertex cover, they must be at least as many as those of a minimum vertex cover and thus the size of a maximum matching. The number of edges in the maximal matching is half this number. □

Remarks:

- Given the important role of all these graph structures in "traditional" algorithms, it comes hardly as a surprise that being able to construct an MIS fast in distributed systems comes in very handy on many occasions!

What to take Home

- Finding an MIS is a trivial task for centralized algorithms. It's a pure *symmetry breaking* problem.
- An MIS is typically not useful by itself, but MIS algorithms are very helpful in the construction of many other basic graph structures.

- Your basic toolbox for analyzing randomized algorithms:
  - linearity of expectation
  - Markov's bound
  - probabilistic independence (and its convenient properties)
  - probabilistic domination (see below)
  - Chernoff's bound
  - union bound
- Probabilistic domination simply means that you replace a random variable by some other variable "dominating" it. We did this with the probability that a constant fraction of the edges is removed: whether this happens might depend on the graph, so we could not apply Chernoff to collections of such variables. However, we knew that *no matter what*, the respective probability is constantly bounded *independently* of the graph, so we used independent Bernoulli variables to indicate whether a phase was good or not, possibly treating some good phases as "bad," to be on the safe side.
- One particularly useful toolchain is probabilistic domination $\rightarrow$ Chernoff $\rightarrow$ union bound. It's not uncommon to use the other tools just to feed this chain, like we did today, too!
- That's almost everything you need to analyze the majority of randomized distributed algorithms out there. The difficulty typically lies in how to apply these tools properly, not in finding new ones![3]

Bibliographic Notes

In the 80s, several groups of researchers came up with more or less the same ideas: Luby [Lub86], Alon, Babai, and Itai [ABI86], and Israeli and Itai [II86]. Essentially, all of these works imply fast randomized distributed algorithms for computing an MIS. The new MIS variant (with a simpler analysis) presented here is by Métivier, Robson, Saheb-Djahromi, and Zemmari [MRSDZ11]. With some adaptations, also the algorithms [Lub86, MRSDZ11] only need to transmit a total of $\mathcal{O}(\log n)$ bits per node, which is asymptotically optimal, even on unoriented trees [KSOS06].

The state-of-the-art running time on general graphs, due to Ghaffari, is $\mathcal{O}(\log \Delta + 2^{\mathcal{O}(\sqrt{\log \log n})})$. However, when the maximum degree $\Delta \in n^{\Omega(1)}$, this is not asymptotically faster than the presented algorithm. Also, while this nearly matches the lower bound of $\Omega(\min\{\log \Delta / \log \log \Delta, \sqrt{\log n / \log \log n}\})$ [KMW10] with respect to Δ, it does not yield further insight for the range $\Delta \gg 2^{\sqrt{\log n}}$. Deterministic MIS algorithms are emberassingly far from the lower bounds: the best known upper bound is $2^{\mathcal{O}(\sqrt{\log n})}$ [PS96].

[3] Admittedly, this might be a bad case of "if all you have is a hammer, everything looks like a nail." In a course on probabilistic algorithms I took I drove the TA mad by solving most of the exercises utilizing Chernoff's bound (or trying to), even though the questions were handcrafted to strongly suggest otherwise.

In growth-bounded graphs,[4] an MIS is a constant-factor approximation to a minimum dominating set. In this graph family, an MIS (and thus small dominating set) can be computed in $\mathcal{O}(\log^* n)$ rounds [SW08]; the respective algorithm leverages the Cole-Vishkin technique. On graphs that can be decomposed into a constant number of forests, a constant-factor approximation to a minimum dominating set can be computed with the help of an MIS algorithm, too [LPW13] (see exercises).

Wide parts of the script for this lecture are based on material kindly provided by Roger Wattenhofer. Thanks!

Bibliography

[ABI86] Noga Alon, László Babai, and Alon Itai. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *J. Algorithms*, 7(4):567–583, 1986.

[II86] Amos Israeli and Alon Itai. A Fast and Simple Randomized Parallel Algorithm for Maximal Matching. *Inf. Process. Lett.*, 22(2):77–80, 1986.

[KMW10] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local Computation: Lower and Upper Bounds. *CoRR*, abs/1011.5470, 2010.

[KSOS06] Kishore Kothapalli, Christian Scheideler, Melih Onus, and Christian Schindelhauer. Distributed coloring in $O(\sqrt{\log n})$ Bit Rounds. In *20th international conference on Parallel and Distributed Processing (IPDPS)*, 2006.

[LPW13] Christoph Lenzen, Yvonne-Anne Pignolet, and Roger Wattenhofer. Distributed Minimum Dominating Set Approximations in Restricted Families of Graphs. *Distributed Computing*, 26(2):119–137, 2013.

[Lub86] Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986.

[MRSDZ11] Yves Métivier, John Michael Robson, Nasser Saheb-Djahromi, and Akka Zemmari. An optimal bit complexity randomized distributed MIS algorithm. *Distributed Computing*, 23(5-6):331–340, 2011.

[PS96] Alessandro Panconesi and Aravind Srinivasan. On the Complexity of Distributed Network Decomposition. *J. Algorithms*, 20(2):356–374, 1996.

[SW08] Johannes Schneider and Roger Wattenhofer. A Log-Star Distributed Maximal Independent Set Algorithm for Growth-Bounded Graphs. In *27th ACM Symposium on Principles of Distributed Computing (PODC), Toronto, Canada*, August 2008.

[4] A graph is growth-bounded, if the maximal number of independent nodes in r-hop neighborhoods is bounded as a function of r, independently of n or Δ.

Lecture 6

Minimum Spanning Trees

In this lecture, we study another classic graph problem from the distributed point of view: *minimum spanning tree* construction.

Definition 6.1 (Minimum Spanning Tree (MST))**.** *Given a simple weighted connected graph $G = (V, E, W)$, $W\colon E \to \{1, \dots, n^{\mathcal{O}(1)}\}$, a* minimum spanning tree *$T \subseteq E$ minimizes $\sum_{e \in t} W(e)$ among all spanning trees of G. Here* spanning *means that all nodes are connected by the respective edge set.*

Our goal today is to develop efficient MST algorithms in the Congest model, on an arbitrary simple weighted connected graph G. The Congest model is identical to the synchronous message passing model, but with the additional restriction that $\mathcal{O}(\log n)$ bits are sent in each round along each edge.

Remarks:

- In the Congest model (for a connected graph), essentially every problem can be solved in $\mathcal{O}(|E|)$ rounds by collecting and distributing to all nodes the entire topology, and then having each node solve the problem at hand locally.
- Initially nodes know the weights of their incident edges. In the end, nodes need to know which of their incident edges are in the MST.
- If you're wondering about the weird constraint on the range of the weight function: This is so we can encode an edge weight in a message. Fractional edge weights are fine; we simply multiply all weights by the smallest number that is an integer multiple of the denominators of all fractions (where w.l.o.g. we assume that all fractions are reduced).
- One can handle weights with range $1, \dots, 2^{n^{\mathcal{O}(1)}}$ by rounding up to the next integer power of $1 + \varepsilon$ for $\varepsilon > n^{-\mathcal{O}(1)}$. Then an edge weight can be encoded with

  $$\left\lceil \log \log_{1+\varepsilon} 2^{n^{\mathcal{O}(1)}} \right\rceil = \log \left(\frac{n^{\mathcal{O}(1)}}{\log(1+\varepsilon)} \right) \subseteq \mathcal{O}\left(\log n + \log \frac{1}{\varepsilon} \right) = \mathcal{O}(\log n)$$

  bits as well. However, then we will get only $(1+\varepsilon)$-approximate solutions.

6.1 MST is a Global Problem

Recall that in the message passing model without restrictions on message size, an r-round algorithm is some mapping from r-neighborhoods[1] labeled by inputs, identifiers, and, for randomized algorithms, the random strings to outputs. Thus, r puts a bound on the *locality* of a problem: If topology or inputs are locally changed, the outputs of the algorithm change only up to distance r (both in the new and old graph). In contrast, if no r-round algorithm exists for $r \in o(D)$, a problem is *global*.

Theorem 6.2. *MST is a global problem, i.e., any distributed algorithm computing an MST must run for $\Omega(D)$ rounds. This holds even when permitting outputs that are merely spanning subgraphs (i.e., not necessarily trees) and, for any $1 \leq \alpha \in n^{\mathcal{O}(1)}$, α-approximate.*

Proof. Consider the cycle consisting of n nodes, and denote by e_1 and e_2 two edges on opposite sides of the cycle (i.e., in maximal distance from each other). For $1 \leq \alpha \in n^{\mathcal{O}(1)}$, define

$$W_1(e) := \begin{cases} 2\alpha^2 n & \text{if } e = e_1 \\ \alpha n & \text{if } e = e_2 \\ 1 & else \end{cases} \qquad W_2(e) := \begin{cases} \alpha n & \text{if } e = e_2 \\ 1 & else. \end{cases}$$

On the cycle with weights W_1, the MST consists of all edges but e_1. In fact, *any* solution containing e_1 has approximation ratio at least

$$\frac{W_1(e_1)}{W_2(e_2) + n - 2} > \frac{2\alpha^2 n}{(\alpha+1)n} \geq \alpha.$$

Thus, any α-approximate solution must output all edges but e_1. Likewise, on the cycle with weights W_2, the MST consists of all edges but e_2, and any solution containing e_1 has approximation ratio at least

$$\frac{\alpha n}{n-1} > \alpha.$$

Thus, any α-approximate solution must output all edges but e_2. As by construction the nodes of e_1 and those of e_2 are in distance $\Omega(D) = \Omega(n)$, finding any spanning subgraph that is by at most factor α heavier than an MST is a global problem. □

Remarks:

- The restriction that $\alpha \in n^{\mathcal{O}(1)}$ is only a formality in this theorem. We decided that only polynomial edge weights are permitted in the problem description, so we will be able to *talk* about the weights. But if they are so large that we can't talk about them, this doesn't make our job easier!
- W.l.o.g., we assume in the following that all edge weights are distinct, i.e., $W(e) \neq W(e')$ for $e \neq e'$. This is achieved by attaching the identifiers of the endpoints of e to its weight and use them to break symmetry in case of identical weights. This also means that we can talk of *the* MST of G from now on, as it must be unique.

[1] Here, edges connecting two nodes that are both exactly in distance r are not included.

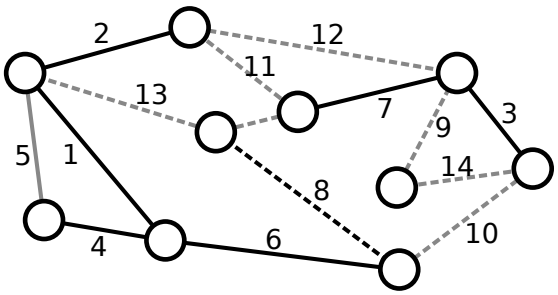

Figure 6.1: Snapshot of Kruskal's algorithm. Solid black edges have been added to T, the solid gray edges have been discarded. The dashed black edge is being processed and will be added to T since it does not close a cycle; the dashed gray edges will be processed in later iterations.

6.2 Being Greedy: Kruskal's Algorithm

Considering greedy strategies is always a good starting point. In the case of an MST, this means to always add the cheapest useful edge first! As closing cycles is pointless, this yields Kruskal's algorithm, compare Figure 6.1:

Algorithm 13 Kruskal's algorithm (centralized version)

```
1: sort E in ascending order of weights; denote the result by (e_1, ..., e_|E|)
2: T := ∅
3: for i = 1, ..., |E| do
4:    if T ∪ {e_i} is a forest then
5:       T := T ∪ {e_i}
6:    end if
7: end for
8: return T
```

Lemma 6.3. *If an edge is heaviest in a cycle, it is not in the MST. If all such edges are deleted, the MST remains. In particular, Kruskal's algorithm computes the MST.*

Proof. Denote the MST by M and suppose for contradiction that $e \in M$ is heaviest in a cycle C. As $C \setminus \{e\}$ connects the endpoints of e, there must be an edge $e' \in C$ so that $(M \setminus \{e\}) \cup \{e'\}$ is a spanning tree (i.e., e' connects the two components of $M \setminus \{e\}$). However, as $W(e') < W(e)$, $(M \setminus \{e\}) \cup \{e'\}$ is lighter than M, contradicting that M is the MST.

Thus, when deleting all edges that are heaviest in some cycle, no MST edge is deleted. As this makes sure that there is no cycle contained in the obtained edge set, the result is a forest. As this forest contains the MST, but a tree is a maximal forest, the forest must in fact be the MST. □

Let's make this into a distributed algorithm! Of course we don't simply collect all edges and execute the algorithm locally. Still, we need to somehow collect the information to compare. We do this, but drop all edges which are certainly "unnecessary" on the fly.

Algorithm 14 Kruskal's algorithm (distributed version)

compute an (unweighted) BFS, its depth d, and n; denote the root by R
for each $v \in V$ in parallel **do**
  $E_v := \{\{v, w\} \in E\}$
  $S_v := \emptyset$ // sent edges
end for
for $i = 1, \ldots, n + d - 2$ **do**
  for each $v \in V \setminus \{R\}$ in parallel **do**
    $e := \text{argmin}_{e' \in E_v \setminus S_v}\{W(e')\}$ // lightest unsent edge
    send $(e, W(e))$ to v's parent in the BFS tree
    $S_v := S_v \cup \{e\}$
  end for
  for each $v \in V$ in parallel **do**
    for each received $(e, W(e))$ **do**
      $E_v := E_v \cup \{e\}$ // also remember $W(e)$ for later use
    end for
    for each cycle C in E_v **do**
      $e := \text{argmax}_{e' \in C}\{W(e')\}$ // heaviest edge in cycle
      $E_v := E_v \setminus \{e\}$
    end for
  end for
end for
R broadcasts E_R over the BFS tree
return E_R

Intuitively, MST edges will never be deleted, and MST edges can only be "delayed," i.e., stopped from moving towards the root, by other MST edges. However, this is not precisely true: There may be other lighter edges that go first, as the respective senders do not know that they are not MST edges. The correct statement is that for the k^{th}-lightest MST edge, at most $k - 1$ lighter edges can keep it from propagating towards the root. Formalizing this intuition is a bit tricky.

Definition 6.4. *Denote by B_v the subtree of the BFS tree rooted at node $v \in V$, by d_v its depth, by $E_{B_v} := \bigcup_{w \in B_v}\{\{w, u\} \in E_w\}$ the set of edges known to some node in B_v, and by $F_{B_v} \subseteq E_{B_v}$ the lightest maximal forest that is a subset of E_{B_v}.*

Lemma 6.5. *For any node v and any $E' \subseteq E_{B_v}$, the result of Algorithm 13 applied to E' contains $F_{B_v} \cap E'$.*

Proof. By the same argument as for Lemma 6.3, Kruskal's algorithm deletes exactly all edges from a given edge set that are heaviest in some cycle. F_{B_v} is the set of edges that survive this procedure when it is applied to E_{B_v}, so $F_{B_v} \cap E'$ survives for any $E' \subseteq E_{B_v}$. □

Lemma 6.6. *For any $k \in \{0, 1, \ldots, |F_{B_v}|\}$, after $d_v + k - 1$ rounds of the second FOR loop of the algorithm, the lightest k edges of F_{B_v} are in E_{B_v}, and have been sent to the parent by the end of round $d_v + k$.*

Proof. We prove the claim by double induction over the depth d_v of the subtrees and k. The claim holds trivially true for $d_v = 0$ and all k, as for leaves v we have $E_{B_v} = F_{B_v}$. Now consider $v \in V$ and assume that the claim holds for all $w \in V$ with $d_w < d_v$ and all k. It is trivial for d_v and $k = 0$, so assume it also holds for d_v and some $k \in \{0, \dots, |F_{B_v}| - 1\}$ and consider index $k + 1$.

Because

$$E_{B_v} = \{\{v, w\} \in E_v\} \cup \bigcup_{w \text{ child of } v} E_{B_w},$$

we have that the $(k+1)^{th}$ lightest edge in F_{B_v} is already known to v or it is in E_{B_w} for some child w of v. By the induction hypothesis for index $k + 1$ and $d_w < d_v$, each child w of v has sent the $k + 1$ lightest edges in F_{B_w} to v by the end of round $d_w + k + 1 \leq d_v + k$. The $(k + 1)^{th}$ lightest edge of F_{B_v} must be contained in these sent edges: Otherwise, we can take the sent edges (which are a forest) and edges $k + 1, \dots, |F_{B_v}|$ out of F_{B_v} to either obtain a cycle in which an edge from F_{B_v} is heaviest or a forest with more edges than F_{B_v}, in both cases contained in E_{B_v}. In the former case, this contradicts Lemma 6.5, as then an edge from F_{B_v} would be deleted from $E' \subseteq E_{B_v}$. In the latter case, it contradicts the maximality of F_{B_v}, as all maximal forests in E_{B_v} have the same number of edges (the number of nodes minus the number of components of (G, E_{B_v})). Either way, the assumption that the edge was not sent must be wrong, implying that v learns of it at the latest in round $d_v + k$ and adds it to E_v. By Lemma 6.5, it never deletes the edge from E_v. This shows the first part of the claim.

To show that by the end of the round $d_v + k + 1$, v sends the edge to its parent, we apply the induction hypothesis for d_v and k. It shows that v already sent the k lightest edges from F_{B_v} before round $d_v + k + 1$, and therefore will send the next in round $d_v + k + 1$ (or has already done so), unless there is a lighter unsent edge $e \in E_v$. As F_{B_v} is a maximal forest, $F_{B_v} \cup \{e\}$ contains exactly one cycle. However, as e does not close a cycle with the edges sent earlier, it does not close a cycle with all the edges in F_{B_v} that are lighter than e. Thus, the heaviest edge in the cyccle is heavier than e, and deleting it results in a lighter maximal forest than F_{B_v}, contradicting the definition of F_{B_v}. This concludes the induction step and therefore the proof. □

Theorem 6.7. *For a suitable implementation, Algorithm 14 computes the MST M in $\mathcal{O}(n)$ rounds.*

Proof. The algorithm is correct, if at the end of the second FOR loop, it holds that $E_R = M$. To see this, observe that $E_{B_R} = \bigcup_{v \in V} \{\{v, w\} \in E\} = E$ and hence $F_{B_R} = M$. We apply Lemma 6.6 to R and round $n + d - 2 = |M| + d_R - 1$. The lemma then says that $M \subseteq E_R$. As in each iteration of the FOR loop, all cycles are eliminated from E_R, E_R is a forest. A forest has at most $|M| = n - 1$ edges, so indeed $E_R = M$.

Now let us check the time complexity of the algorithm. From Lecture 2, we know that a BFS tree can be computed in $\mathcal{O}(D)$ rounds using messages of size $\mathcal{O}(\log n)$.[2] Computing the depth of the constructed tree and its number of nodes n is trivial using messages of size $\mathcal{O}(\log n)$ and $\mathcal{O}(D)$ rounds. The second

[2] If there is no special node R, we may just pick the one with smallest identifier, start BFS constructions at all nodes concurrently, and let constructions for smaller identifiers "overwrite" and stop those for larger identifiers.

FOR loop runs for $n+d-1 \in n+\mathcal{O}(D)$ rounds. Finally, broadcasting the $n-1$ edges of M over the BFS tree takes $n+d-1 \in n+\mathcal{O}(D)$ rounds as well, as in each round, the root can broadcast another edge without causing congestion; the last edge will have arrived at all leaves in round $n-1+d$, as it is sent by R in round $n-1$. As $D \leq n-1$, all this takes $\mathcal{O}(n+D) = \mathcal{O}(n)$ rounds. □

Remarks:

- A clean and simple algorithm, and running time $\mathcal{O}(n)$ beats the trivial $\mathcal{O}(|E|)$ on most graphs.
- This running time is optimal up to constants on cycles. But (non-trivial) graphs with diameter $\Omega(n)$ are rather rare in practice. We shouldn't stop here!
- Also nice: everyone learns about the entire MST.
- Of course, there's no indication that we can't be faster in case $D \ll n$. We're not asking for everyone to learn the entire MST (which trivially would imply running time $\Omega(n)$ in the worst case), but only for nodes learning about their incident MST edges!
- To hope for better results, we need to make sure that we work concurrently in many places!

6.3 Greedy Mk II: Gallager-Humblet-Spira (GHS)

In Kruskal's algorithm, we dropped all edges that are heaviest in some cycle, because they cannot be in the MST. The remaining edges form the MST. In other words, picking an edge that cannot be heaviest in a cycle is always a correct choice.

Lemma 6.8. *For any $\emptyset \neq U \subset V$,*

$$e_U := \operatorname*{argmin}_{e \in (U \times V \setminus U) \cap E} \{W(e)\}$$

is in the MST.

Proof. As G is connected, $(U \times V \setminus U) \cap E \neq \emptyset$. Consider any cycle $C \ni e_U$. Denoting $e_U = \{v, w\}$, $C \setminus \{e_U\}$ connects v and w. As $e_U \in U \times V \setminus U$, it follows that $(C \setminus \{e_U\}) \cap (U \times V \setminus U) \neq \emptyset$, i.e., there is another edge $e \in C$ between a node in U and a node in $V \setminus U$ besides e_U. By definition of e_U, $W(e_U) < W(e)$. As $C \ni e_U$ was arbitrary, we conclude that there is no cycle C in which e_U is the heaviest edge. Therefore, e_U is in the MST by Lemma 6.3. □

This observation leads to *another* canonical greedy algorithm for finding an MST. You may know the centralized version, Prim's algorithm. Let's state the distributed version right away.

Admittedly, this is a very high-level description, but it is much easier to understand the idea of the algorithm this way. It also makes proving correctness

Algorithm 15 GHS (Gallager–Humblet–Spira)

1: $T := \emptyset$ // T will always be a forest
2: **repeat**
3: $\mathcal{F}$:= set of connectivity components of T (i.e., maximal trees)
4: Each $F \in \mathcal{F}$ determines the lightest edge leaving F and adds it to T
5: **until** T is a spanning subgraph (i.e., there is no outgoing edge)
6: **return** T

very simple, as we don't have to show that the implementations of the individual steps work correctly (yet). We call an iteration of the REPEAT statement a *phase*.

Corollary 6.9. *In each phase of the GHS algorithm, T is a forest consisting of MST edges. In particular, the algorithm returns the MST of G.*

Proof. It suffices to show that T contains only MST edges. Consider any connectivity component $F \in \mathcal{F}$. As the algorithm has not terminated yet, F cannot contain all nodes. Thus, Lemma 6.8 shows that the lightest outgoing edge of F exists and is in the MST. □

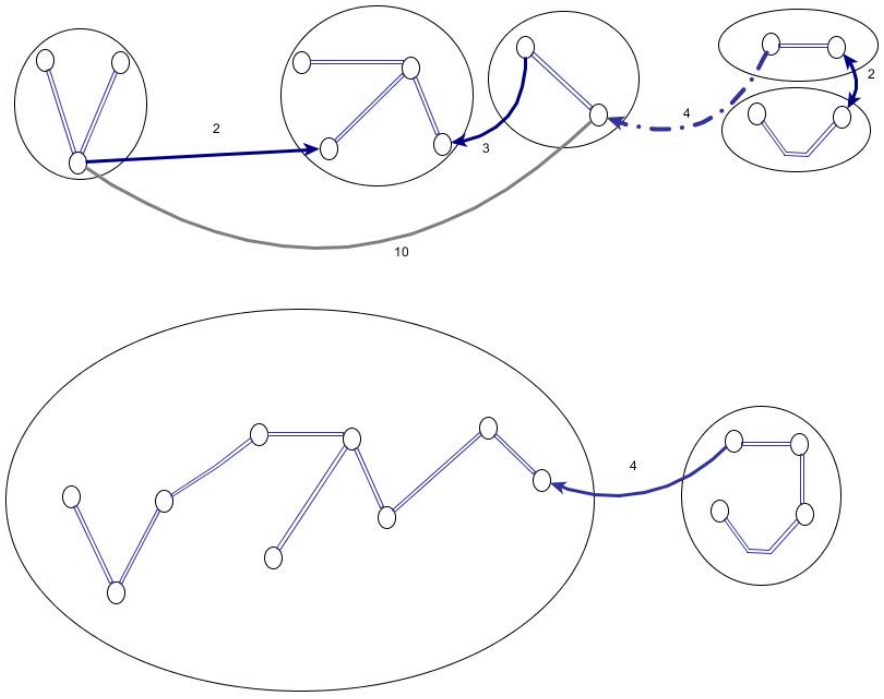

Figure 6.2: Two iterations of the GHS algorithm. The circled areas contain the components at the beginning of the iteration, connected by the already selected MST edges. Each component selects the lightest outgoing edge into the MST (blue solid arrows). Other edges within components, like the grey one after the first iteration, are discarded, as they are heaviest in some cycle.

Great! But now we have to figure out how to implement this idea efficiently. It's straightforward to see that we won't get into big trouble due to having too many phases.

Lemma 6.10. *Algorithm 15 terminates within* $\lceil \log n \rceil$ *phases.*

Proof. Denote by n_i the number of nodes in the smallest connectivity component (maximal tree) in $\mathcal{F}$ at the end of phase i. We claim that $n_i \geq \min\{2^i, n\}$. Trivially, this number is $n_0 := 1$ "at the end of phase 0," i.e., before phase 1. Hence, it suffices to show that $n_i \geq \min\{2n_{i-1}, n\}$ for each phase i. To this end, consider any $F \in \mathcal{F}$ at the beginning of phase i. Unless F already contains all nodes, it adds its lightest outgoing edge to T, connecting it to at least one other component $F' \in \mathcal{F}$. As $|F| \geq n_{i-1}$, $|F'| \geq n_{i-1}$, and connectivity components are disjoint, $|F \cup F'| \geq 2n_{i-1}$. The claim follows. □

Ok, so what about the phases? We need to do everything concurrently, so we cannot just route all communication over a single BFS tree without potentially causing a lot of "congestion." Let's use the already selected edges instead!

Lemma 6.11. *For a given phase* i*, denote by* D_i *the maximum diameter of any* $F \in \mathcal{F}$ *at the beginning of the phase, i.e., after the new edges have been added to* T*. Then phase* i *can be implemented in* $\mathcal{O}(D_i)$ *rounds.*

Proof. All communication happens on edges of T that are selected in or before phase i. Consider a connectivity component of T at the beginning of phase i. By Corollary 6.9, it is a tree. We root each such tree at the node with smallest identifier and let each node of the tree learn this identifier, which takes $\mathcal{O}(d)$ time for a tree of depth d. Clearly, $d \leq D_i$. Then each node learns the identifiers of the roots of all its neighbors' trees (one round). On each tree, now the lightest outgoing edge can be determined by every node sending their lightest edge leaving its tree (alongside its weight) to the root; each node only forwards the lightest edge it knows about to the root. Completing this process and announcing the selected edges to their endpoints takes $\mathcal{O}(D_i)$ rounds. As the communication was handled on each tree separately without using external edges (except for exchanging the root identifiers with neighbors and "marking" newly selected edges), all this requires messages of size $\mathcal{O}(\log n)$ only. □

We've got all the pieces to complete the analysis of the GHS algorithm.

Theorem 6.12. *Algorithm 15 computes the MST. It can be implemented in* $\mathcal{O}(n \log n)$ *rounds.*

Proof. Correctness was shown in Corollary 6.9. As trivially $D_i \leq n - 1$ for all phases i (a connectivity component cannot have larger diameter than the number of its nodes), by Lemma 6.3 each phase can be completed in $\mathcal{O}(n)$ rounds. This can be detected and made known to all nodes within $\mathcal{O}(D) \subseteq \mathcal{O}(n)$ rounds using a BFS tree. By Lemma 6.10, there are $\mathcal{O}(\log n)$ phases. □

Remarks:

- The $\log n$ factor can be shaved off.
- The original GHS algorithm is asynchronous and has a message complexity of $\mathcal{O}(|E| \log n)$, which can be improved to $\mathcal{O}(|E| + n \log n)$. It was celebrated for that, as this is way better than what comes from the use of an α-synchronizer. Basically, the constructed tree is used like a β-synchronizer to coordinate actions within connectivity components, and only the exchange of component identifiers is costly.
- The GHS algorithm can be applied in different ways. GHS for instance solves leader election in general graphs: once the tree is constructed, finding the minimum identifier using few messages is a piece of cake!

6.4 Greedy Mk III: Garay-Kutten-Peleg (GKP)

We now have two different greedy algorithms that run in $\mathcal{O}(n)$ rounds. However, they do so for quite different reasons: The distributed version of Kruskal's algorithm basically reduces the number of components by 1 per round (up to an additive D), whereas the GHS algorithm cuts the number of remaining components down by a factor of 2 in each phase. The problem with the former is that initially there are n components, the problem with the latter is that components of large diameter take a long time to handle.

If we could use the GHS algorithm to reduce the number of components to something small, say $\sqrt{n}$, quickly without letting them get too big, maybe we can then finish the MST computation by Kruskal's approach? Sounds good, except that it may happen that, in a single iteration of GHS, a huge component appears. We then wouldn't know that it is so large without constructing a tree on it or collecting the new edges somewhere, but both could take too long! The solution is to be more conservative with merges and apply a symmetry breaking mechanism: GKP grows components GHS-like until they reach a size of $\sqrt{n}$. Every node learns its component identifier (i.e., the smallest ID in the component), and GKP then joins them using the pipelined MST construction (where nodes communicate edges between connected components instead of all incident edges).

Let's start with correctness.

Lemma 6.13. *Algorithm 16 adds only MST edges to T. It outputs the MST.*

Proof. By Lemma 6.8, only MST edges are added to C in any iteration of the FOR loop, so at the end of the looop T is a subforest of the MST. The remaining MST edges thus must be between components, so contracting components and deleting loops does not delete any MST edges. The remaining MST edges are now just the MST of the constructed multigraph, and Kruskal's algorithm works fine on (loop-free) multigraphs, too. □

Also, we know that the second part will be fast if few components remain.

Corollary 6.14. *Suppose after the FOR loop of Algorithm 16 k components of maximum diameter $D_{\max}$ remain, then the algorithm terminates within $\mathcal{O}(D + D_{\max} + k)$ additional rounds.*

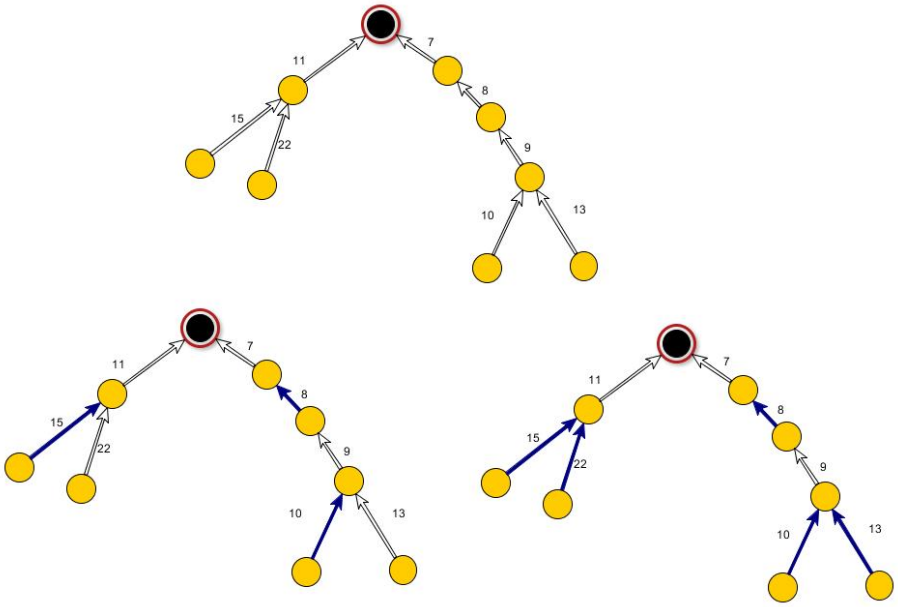

Figure 6.3: Top: A component of $(\mathcal{F}, C)$ in a phase of the first part of the GKP algorithm. "Large" components that marked no MST edge for possible inclusion can only be roots; we have such a root here. Bottom left: The blue edges show a matching constructed using the Cole-Vishkin algorithm. Bottom right: The blue edges are the final set of selected edges in this phase.

Algorithm 16 GKP (Garay–Kutten–Peleg). We slightly abuse notation by interpreting edges of C also as edges between components F. Contracting an edge means to identify its endpoints, where the new node "inherits" the edges from the original nodes.

1: $T := \emptyset$ // T will always be a forest
2: **for** $i = 0, \dots, \lceil \log \sqrt{n} \rceil$ **do**
3: $\mathcal{F} :=$ set of connectivity components of T (i.e., maximal trees)
4: Each $F \in \mathcal{F}$ of diameter at most 2^i determines the lightest edge leaving F and adds it to a candidate set C
5: Add a maximal matching $C_M \subseteq C$ in the graph $(\mathcal{F}, C)$ to T
6: If $F \in \mathcal{F}$ of diameter at most 2^i has no incident edge in C_M, it adds the edge it selected into C to T
7: **end for**
8: denote by $G' = (V, E', W')$ the multigraph obtained from contracting all edges of T (deleting loops, keeping multiple edges)
9: run Algorithm 14 on G' and add the respective edges to T
10: **return** T

Proof. The contracted graph has exactly k nodes and thus $k-1$ MST edges are left. The analysis of Algorithm 14 also applies to multigraphs, so (a suitable implementation) runs for $\mathcal{O}(D+k)$ additional rounds; all that is required is that

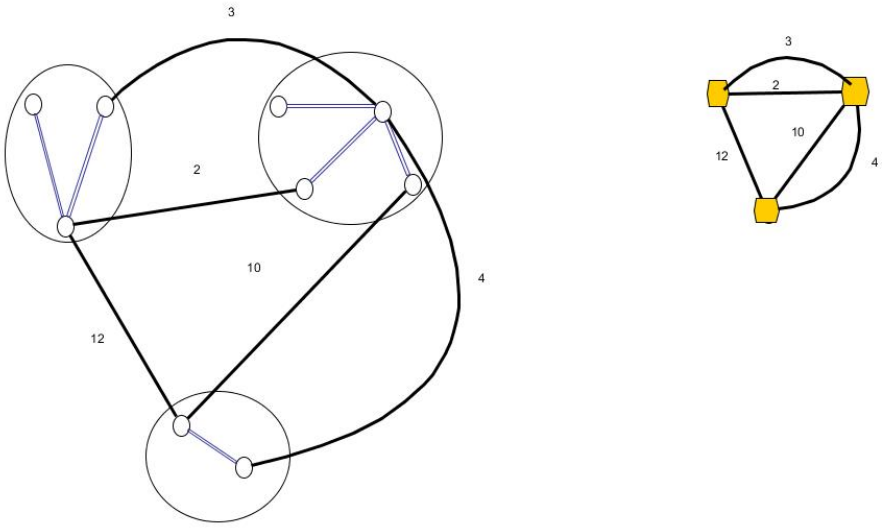

Figure 6.4: Left: Components and their interconnecting edges after the first stage of the GKP algorithm. Right: The multigraph resulting from contraction of the components.

the nodes of each component agree on an identifier for their component. This can be done by figuring out the smallest identifier in each component, which takes $\mathcal{O}(D_{\max})$ rounds. □

It remains to prove three things: (i) components don't become too large during the FOR loop, (ii) we can efficiently implement the iterations in the FOR loop, and (iii) few components remain after the FOR loop. Let's start with (i). We again call one iteration of the FOR loop a phase.

Lemma 6.15. *At the end of phase i, components have diameter $\mathcal{O}(2^i)$.*

Proof. We claim that at the end of phase i, no component has diameter larger than $6 \cdot 2^i \in \mathcal{O}(2^i)$, which we show by induction. Trivially, this holds for $i = 0$ (i.e., at the start of the algorithm). Hence, suppose it holds for all phases $j \leq i \in \mathbb{N}_0$ for some i and consider phase $i + 1$.

Consider the graph $(\mathcal{F}, C)$ from this phase. We claim that each component of $(\mathcal{F}, C)$ has diameter at most 3. To see this, observe that if an unmatched $F \in \mathcal{F}$ adds an edge $\{F, F'\}$, F' must be matched: otherwise, $\{F, F'\}$ could be added to the matching, contradicting its maximality. Thus, all non-matching edges added to T "attach" some $F \in \mathcal{F}$ that was isolated in the graph $(\mathcal{F}, C_M)$ to some F' that is not picking a new edge in this step. This increases the diameter of components from at most 1 (for a matching) to at most 3.

Next, we claim that no component contains more than one F of diameter larger than 2^i (with respect to G). This can be seen by directing all selected edges away from the $F \in \mathcal{F}$ that selected it (breaking ties arbitrarily). We obtain a directed graph of outdegree at most 1, in which any F of diameter

larger than 2^i has outdegree 0 and is thus the root of a component that is an oriented tree.

Now consider a new component at the end of the phase. It is composed of at most one previous component of – by the induction hypothesis – size at most $6 \cdot 2^i$, while all other previous components have size at most 2^i. The longest possible path between any two nodes in the new component thus crosses the large previous component, up to 3 small previous components, and up to 3 edges between previous components, for a total of $6 \cdot 2^i + 3 \cdot 2^i + 3 \leq 6 \cdot 2^{i+1}$ hops. □

Together with an old friend, we can exploit this to show (ii).

Corollary 6.16. *Each iteration of the FOR loop can be implemented with running time $\mathcal{O}(2^i \log^* n)$.*

Proof. By Lemma 6.15, in phase i components are of size $\mathcal{O}(2^i)$. We can thus root them and determine the edges in C in $\mathcal{O}(2^i)$ rounds. By orienting each edge in C away from the component $F \in \mathcal{F}$ that selected it (breaking ties by identifiers), $(\mathcal{F}, C)$ becomes a directed graph with outdegree 1. We simulate the Cole-Vishkin algorithm on this graph to compute a 3-coloring in $\mathcal{O}(2^i \log^* n)$ rounds. To this end, component F is represented by the root of its spanning tree and we exploit that it suffices to communicate only "in one direction," i.e., it suffices to determine the current color of the "parent." Thus, for each component, only one color each needs to be sent and received, respectively, which can be done with message size $\mathcal{O}(\log n)$ over the edges of the component. The time for one iteration then is $\mathcal{O}(2^i)$. By Theorem 1.7, we need $\mathcal{O}(\log^* n)$ iterations; afterwards, we can select a matching in $\mathcal{O}(2^i)$ time by going over the color classes sequentially (cf. Exercise 1) and complete the phase in additional $\mathcal{O}(2^i)$ rounds, for a total of $\mathcal{O}(2^i \log^* n)$ rounds. □

It remains to show that all this business really yields sufficiently few components.

Lemma 6.17. *After the last phase, at most $\sqrt{n}$ components remain.*

Proof. Observe that in each phase i, each component of diameter smaller than 2^i is connected to at least one other component. We claim that this implies that after phase i, each component contains at least 2^i nodes. This trivially holds for $i = 0$. Now suppose the claim holds for phase $i \in \{0, \dots, \lceil \sqrt{n} \rceil - 1\}$. Consider a component of fewer than 2^{i+1} nodes at the beginning of phase $i+1$. It will hence add an edge to C and be matched or add this edge to T. Either way, it gets connected to at least one other component. By the hypothesis, both components have at least 2^i nodes, so the resulting component has at least 2^{i+1}.

As there are $\lceil \log \sqrt{n} \rceil$ phases, in the end each component contains at least $2^{\log \sqrt{n}} = \sqrt{n}$ nodes. As components are disjoint, there can be at most $n/\sqrt{n} = \sqrt{n}$ components left. □

Theorem 6.18. *Algorithm 16 computes the MST and can be implemented such that it runs in $\mathcal{O}(\sqrt{n} \log^* n + \mathcal{O}(D))$ rounds.*

Proof. Correctness is shown in Lemma 6.13. Within $\mathcal{O}(D)$ rounds, a BFS can be constructed and n be determined and made known to all nodes. By Corollary 6.16, phase i can be implemented in $\mathcal{O}(2^i)$ rounds, so in total

$$\sum_{i=0}^{\lceil \log \sqrt{n} \rceil} \mathcal{O}(2^i \log^* n) = \mathcal{O}(2^{\log \sqrt{n}} \log^* n) = \mathcal{O}(\sqrt{n} \log^* n)$$

rounds are required. Note that since the time bound for each phase is known to all nodes, there is no need to coordinate when a phase starts; this can be computed from the depth of the BFS tree, n, and the round in which the root of the BFS tree initiates the main part of the computation. By Lemmas 6.15 and 6.17, only $\sqrt{n}$ components of diameter $\mathcal{O}(\sqrt{n})$ remain. Hence, by Corollary 6.14, the algorithm terminates within additional $\mathcal{O}(\sqrt{n} + D)$ rounds. □

Remarks:

- The use of symmetry breaking might come as a big surprise in this algorithm. And it's the only thing keeping it from being greedy all the way!
- Plenty of algorithms in the CONGEST model follow similar ideas. This is no accident: The techniques are fairly generic, and we will see next time that there is an inherent barrier around $\sqrt{n}$, even if D is small!
- The time complexity can be reduced to $\mathcal{O}(\sqrt{n \log^* n} + D)$ by using only $\left\lceil \log(\sqrt{n / \log^* n}) \right\rceil$ phases, i.e., growing components to size $\Theta(\sqrt{n / \log^* n})$.
- Be on your edge when seeing $\mathcal{O}$-notation with multiple parameters. We typically *want* it to mean that no matter how the parameter combination is, the expression is an asymptotic bound where the constants in the $\mathcal{O}$-notation are *independent* of the parameter choice. However, in particular with lower bounds, this can become difficult, as there may be dependencies between parameters, or the constructions may apply only to certain parameter ranges.

What to take Home

- Studying sufficiently generic and general problems like MIS or MST makes sense even without an immediate application in sight. When I first encountered the MST problem, I didn't see the Cole-Vishkin symmetry breaking technique coming!
- If you're looking for an algorithm and don't know where to start, check greedy approaches first. Either you already end up with something nontrivial, or you see where it goes wrong and might be able to fix it!
- For global problems, it's very typical to use global coordination via a BFS tree, and also "pipelining," the technique of collecting and distributing k pieces of information in $\mathcal{O}(D + k)$ rounds using the tree.

Bibliographic Notes

Tarjan [Tar83] coined the terms *red* and *blue edges* for heaviest cycle-closing edges (which are not in the MST) and lightest edges in an edge cut[3] (which are always in the MST), respectively. Kruskal's algorithm [Kru56] and Prim's algorithm [Pri57] are classics, which are based on eliminating red edges and selecting blue edges, respectively. The distributed variant of Kruskal's algorithm shown today was introduced by Garay, Kutten, and Peleg [GKP98]; it was used in the first MST algorithm of running time $\mathcal{O}(o(n) + D)$. The algorithm then was improved to running time $\mathcal{O}(\sqrt{n}\log^* n + D)$ by introducing symmetry breaking to better control the growth of the MST components in the first phase [KP00] by Kutten and Peleg. The variant presented here uses a slightly simpler symmetry breaking mechanism. Algorithm 15 is called "GHS" after Gallager, Humblet, and Spira [GHS83]. The variant presented here is much simpler than the original, mainly because we assumed a synchronous system and did not care about the number of messages (only their size). As a historical note, the same principle was discovered much earlier by Otakar Boruvka and published in 1926 – in Czech (see [NMN01] for an English translation).

Awerbuch improved the GHS algorithm to achieve (asynchronous) time complexity $\mathcal{O}(n)$ at message complexity $\mathcal{O}(|E| + n\log n)$, which is both asymptotically optimal in the worst case [Awe87]. Yet, the time complexity is improved by the GKP algorithm! We know that this is not an issue of asynchrony vs. synchrony, since we can make an asynchronous algorithm synchronous without losing time, using the α-synchronizer. This is not a contradiction, since the "bad" examples have large diameter; the respective lower bound is *existential*. It says that for any algorithm, there *exists* a graph with n nodes for which it must take $\Omega(n)$ time to complete. These graphs all have diameter $\Theta(n)$! A lower bound has only the final word if it, e.g., says that *for all* graphs of diameter D, any algorithm must take $\Omega(D)$ time.[4] Up to details, this can be shown by a slightly more careful reasoning than for Theorem 6.2. We'll take a closer look at the $\sqrt{n\log^* n}$ part of the time bound next week!

Bibliography

[Awe87] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 230–240, New York, NY, USA, 1987. ACM.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[GKP98] Juan A Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal on Computing*, 27(1):302–316, 1998.

[3] An edge cut is the set of edges $(U \times V \setminus U) \cap E$ for some $\emptyset \neq U \subset V$.

[4] Until we start playing with the model, that is.

[KP00] Shay Kutten and David Peleg. Fast Distributed Construction of Small k-Dominating Sets and Applications, 2000.

[Kru56] Jr. Kruskal, Joseph B. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[NMN01] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar Boruvka on Minimum Spanning Tree Problem Translation of Both the 1926 Papers, Comments, History. *Discrete Mathemetics*, 233(1–3):3–36, 2001.

[Pri57] R. C. Prim. Shortest Connection Networks and some Generalizations. *The Bell Systems Technical Journal*, 36(6):1389–1401, 1957.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms*, chapter 6. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

Lecture 7

Hardness of MST Construction

In the previous lecture, we saw that an MST can be computed in $\mathcal{O}(\sqrt{n \log^* n} + D)$ rounds using messages of size $\mathcal{O}(\log n)$. Trivially, $\Omega(D)$ rounds are required, but what about this $\mathcal{O}(\sqrt{n \log^* n})$ part? The $\Omega(D)$ comes from a *locality*-based argument, just like, e.g., the $\Omega(\log^* n)$ lower bound on list coloring we've seen. But this type of reasoning is not going to work here: *All* problems can be solved in $\mathcal{O}(D)$ rounds by learning the entire topology and inputs!

Hence, if we want to show any such lower bound, we need to reason about the *amount* of information that can be exchanged in a given amount of time. So we need a problem that is "large" in terms of *communication complexity*, then somehow make it hard to talk about it efficiently, and still ensure a small graph diameter (since we want a bound that is not based on having a large diameter).

7.1 Reducing 2-Player Equality to MST Construction

These are quite a few constraints, but actually not too hard to come by. Take a look at the graph in Figure 7.1. There are two nodes A and B, connected by $2k \in \Theta(\sqrt{n})$ disjoint paths p_i, $i \in \{1, \dots, 2k\}$ of length k, and a balanced binary tree with $k+1$ leaves, where the i^{th} leaf is connected to the i^{th} node of each path. Finally, there's an edge from A to the leftmost leaf of the tree and from B to the rightmost leaf of the tree. This graph has a diameter of $\mathcal{O}(\log n)$, as this is the depth of the binary tree. Also, it's clear that all communication between A and B that does not use tree edges will take k rounds, and using the tree edges as "shortcuts" will not yield a very large bandwidth due to the $\mathcal{O}(\log n)$ message size.

So far, we haven't picked any edge weights. We'll use these to encode a difficult problem – in terms of 2-player communication complexity – in a way that keeps the information on the inputs well-separated. We're going to use the *equality problem*.

Definition 7.1 (Deterministic 2-Player Equality)**.** *The* deterministic 2-player equality *problem is defined as follows. Alice and Bob are each given N-bit*

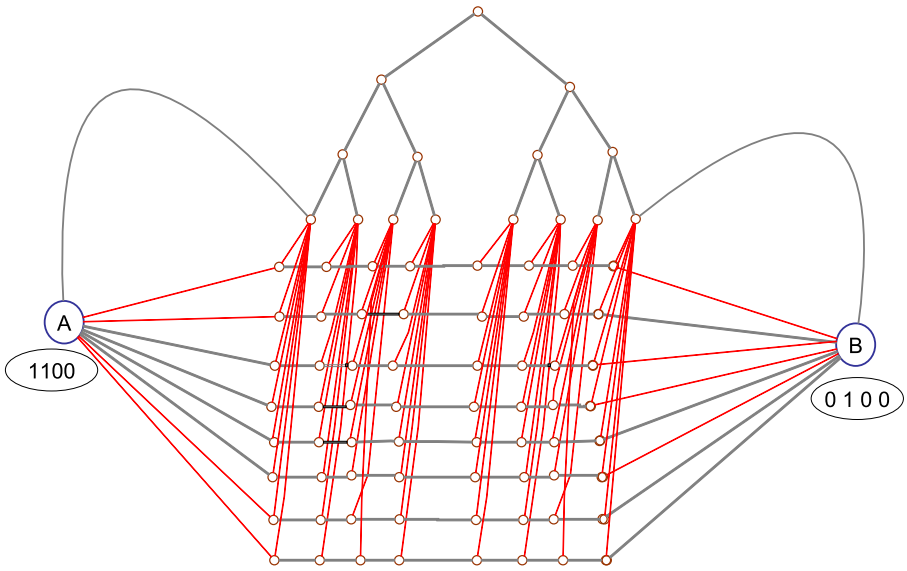

Figure 7.1: Graph used in the lower bound construction. Grey edges have weight 0, red edges weight 1. Only the weight of edges incident to nodes A and B depends on the input strings x and y of Alice and Bob, respectively. The binary tree ensures a diameter of $\mathcal{O}(\log n)$.

strings x and y, respectively. They exchange bits in order to determine whether $x = y$ or not. In the end, they need to output 1 if and only if $x = y$. The communication complexity *of the protocol is the worst-case number of bits that are exchanged (as function of N).*

Let's fix the weights. We'll only need two different values, 0 and 1. Given $x, y \in \{0,1\}^k$, we use the following edge weights:

- All edges between path nodes and the binary tree have weight 1.
- For $i \in \{1, \dots, k\}$, the edge from A to the i^{th} path p_i has weight x_i and the edge from A to p_{k+i} has weight $\overline{x_i}$, where $\overline{x_i} := 1 - x_i$.
- For $i \in \{1, \dots, k\}$, the edge from B to p_i has weight y_i and the edge from B to p_{k+i} has weight $\overline{y_i}$.
- All other edges have weight 0.

This encodes the question whether $x = y$ in terms of the weight of an MST.

Lemma 7.2. *The weight of an MST of the graph given in Figure 7.1 is k if and only if $x = y$.*

Proof. By construction, the binary tree, A, and B are always connected by edges of weight 0. Likewise, for each $i \in \{1, \dots, 2k\}$, the nodes of path p_i are connected by edges of weight 0. Hence, we need to determine the minimal weight of $2k$ edges that interconnect the paths p_i and the component containing the binary tree, A, and B.

Suppose first that $x = y$. Then, for each bit $x_i = y_i$, either $x_i = y_i = 1$ or $\overline{x_i} = \overline{y_i} = 1$. Thus, either the cost of all edges from p_i to the remaining graph is 1, while for p_{i+k} there are 0-weight edges leaving it, or vice versa. Thus, the cost of a minimum spanning tree is exactly k.

On the other hand, if $x \neq y$, there is at least one index i for which $x_i \neq y_i$. Thus, both p_i and p_{i+k} have an outgoing edge of weight 0, and the weight of an MST is at most $k - 1$. □

This looks promising in the sense that we have translated the equality problem to something an MST algorithm has to be able to solve. However, we cannot simply argue that if we let A and B play the roles of Alice and Bob in a network, the (to-be-shown) hardness of equality implies that the problem is difficult in the network, as the nodes of the network might do all kinds of fancy things. Similar to when we established that consensus is hard in message passing systems from the same result for shared memory, we need a clean simulation argument.

Theorem 7.3. *Suppose there is a deterministic distributed algorithm that solves the MST problem on the graph in Figure 7.1 for arbitrary x and y in $T \in o(\sqrt{n}/\log^2 n)$ rounds using messages of size $\mathcal{O}(\log n)$. Then there is a solution to deterministic 2-player equality of communication complexity $o(N)$.*

Proof. We only consider the case where the algorithm requires at most $k/2$ rounds, otherwise it would finish in $\Omega(\sqrt{n})$ time. Alice and Bob simulate the MST algorithm on the graph in Figure 7.1 for $N = k$. Alice knows the entire graph but the weights of the edges incident to B and Bob knows everything but the weights of the edges incident to A. In round $r \in \{1, \ldots, T\}$, Alice simulates the algorithm at the nodes A, the $k+1-r$ nodes on each path closest to it, and a subset of the nodes of the binary tree; the same holds for Bob with respect to B.

Because Alice and Bob know only what's going on for a subset of the nodes, they need to talk about what happens at the boundary of the region under their control. However, since in each round the subpaths they simulate become shorter, the path edges are already accounted for: For the new boundary edges that are in paths, their communication can be computed locally, as the messages that have been sent over them in previous rounds are known – they were in the simulated region!

Hence, we only have to deal with edges incident to nodes of the binary tree. Consider Alice; Bob behaves symmetrically. In round r, Alice simulates the smallest subtree containing all leaves that connect to path nodes she simulates as well. Observe that this rids Alice and Bob of talking about edges between the tree and the rest of the graph as well. The only issue is now that the "simulation front" does not move as fast within the tree as it does in the remaining graph. This implies that Alice needs some information only Bob knows: the messages sent to tree nodes by neighbors she did not simulate in the previous round. Since the tree is binary, it has depth $\lceil \log(k+1) \rceil \in \mathcal{O}(\log n)$ and for each node on the "simulation front," there is at most one message that needs to be communicated. Therefore, Alice and Bob can simulate the MST algorithm communicating only $\mathcal{O}(\log^2 n)$ bits per round,[1] provided that $T \leq k/2$.

[1] In Figure 7.2 this only happens for one tree edge per round, because the tree has depth 3; asymptotically, however, $\Theta(\log n)$ messages are sent on average.

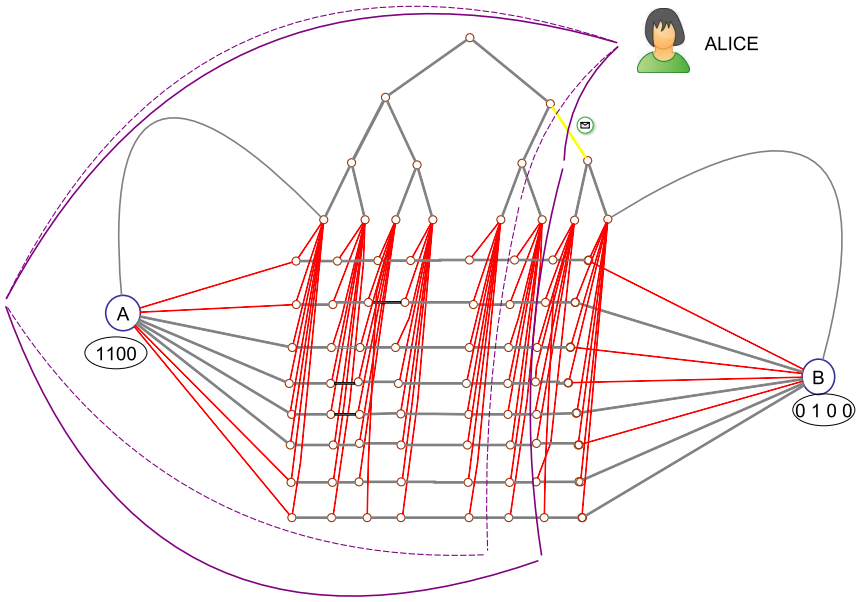

Figure 7.2: The regions of the graph Alice simulates in rounds 2 (solid purple) and 3 (dotted purple), respectively. Bob needs to tell Alice what message is sent over the yellow edge by the node outside the region of round 2.

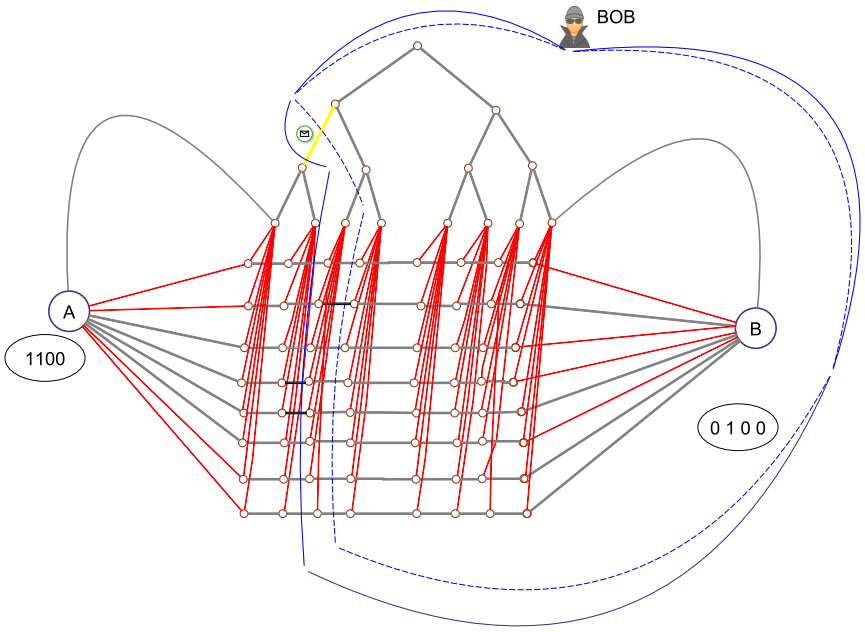

Figure 7.3: The regions of the graph Bob simulates in rounds 2 (solid blue) and 3 (dotted blue), respectively. Alice needs to send Bob up to $\mathcal{O}(\log n)$ bits the algorithm sends over the yellow edge.

For $T < k/2$, the subgraphs Alice and Bob have knowledge of cover the graph. Therefore, for a suitable partition of the edge set $E = E_A \dot{\cup} E_B$, Alice can count and communicate the weight of all MST edges in E_A, and Bob can do so for E_B. This requires at most $2 \log k \in \mathcal{O}(\log n)$ communicated bits. By Lemma 7.2, Alice and Bob now can output whether $x = y$ by outputting whether the MST has weight k or not. The total communication cost of the protocol is $\mathcal{O}(T \log^2 n + \log^2 k) \subseteq o(N)$. □

Remarks:

- In the previous lecture, we required non-zero edge weights. This doesn't change anything, as picking, e.g., 1 and 2 will change the weight of an MST by exactly $n - 1$.
- The simulation approach used for Theorem 7.3 is very flexible. Not only can it be used for different weights of the edges incident to A and B, but also for different topologies of a similar flavor. In fact, it is the underlying technique for almost all the non-locality-based lower bounds we know in the distributed setting without faults!

7.2 Deterministic Equality is Hard

We know now that any fast deterministic MST algorithm using small messages implies a protocol solving deterministic equality at small communication cost. Hence, showing that the communication complexity of this problem is large will yield that MST cannot be solved quickly in all graphs of n nodes, even if D is small.

Theorem 7.4. *The communication complexity of deterministic equality is $N+1$ (respectively N, if we are satisfied with one player learning the result).*

Proof. Clearly, N $(N + 1)$ bits suffice: Just let Alice send x to Bob and decide (and tell the result to Alice). Now assume for contradiction that there is a protocol communicating $N - 1$ bits in which one player decides correctly. As there are $2^N > 2^{N-1}$ possible values of x, there must be two inputs (x, x) and (x', x') (i.e., both with $x = y$) with $x' \neq x$ so that the sequence of $N-1$ exchanged bits (including who sent them)[2] must be identical. By definition, in both cases the output is 1. Now consider the input (x, x'). By induction on the communicated bits and using indistinguishability, we see that Alice cannot distinguish the execution from the one for inputs (x, x), while Bob cannot distinguish it from the one for inputs (x', x'). This is a contradiction, as then one of them decides on output 1, but $x \neq x'$ implies that the output should be 0. To see that one more bit needs to be communicated if both players need to know the output, observe that for an N-bit protocol, one player would have to decide knowing only $N - 1$ bits, yielding the same contradiction. □

Corollary 7.5. *There is no deterministic distributed MST algorithm that uses messages of size $\mathcal{O}(\log n)$ and terminates in $o(\sqrt{n}/\log^2 n + D)$ rounds on all graphs of n nodes and diameter D (unless D is smaller than in the graph from Figure 7.1).*

[2] This follows by induction: Both Alice and Bob must know who sends next, so this must be a function of the transmitted bits.

Proof. Theorem 6.2 shows that running time $o(D)$ is impossible, which shows the claim if $D \geq \sqrt{n}/\log^2 n$. Theorem 7.3 shows that an $o(\sqrt{n}/\log^2 n)$-round algorithm implied a solution to deterministic equality using $o(N)$ bits. By Theorem 7.4, this is not possible, covering the case that $D \leq \sqrt{n}/\log^2 n$. □

7.3 Randomized Equality is Easy

One might expect that the same approach extends to randomized MST algorithms. Unfortunately, the equality problem defies this intuition: It can be solved extremely efficiently using randomization.

Definition 7.6 (Randomized 2-Player Equality)**.** *In the* randomized 2-player equality *problem, Alice and Bob are each given N-bit strings x and y, respectively. Moreover, each of them has access to a (sufficiently long) string of unbiased random bits. They exchange bits in order to determine whether $x = y$ or not. In the end, they need to determine whether $x = y$ correctly with error probability at most ε (for any x and y!).*

The communication complexity *of the protocol is the worst-case number of bits that are exchanged (as function of N). We talk of* public randomness *if Alice and Bob receive the same random bit string, otherwise the protocol uses* private randomness *(and the strings are independent).*

Public randomness is a strong assumption which makes designing an algorithm very simple.

Lemma 7.7. *For any $k \in \mathbb{N}$, randomized equality can be solved with $\varepsilon = 2^{-k}$ using $k + 1$ bits of communication assuming public randomness.*

Proof. Let $a \cdot b := \sum_{i=1}^{N} a_i b_i$ denote the scalar product over $\{0,1\}^N$. Consider the probability that for a uniformly random vector v of N bits, it holds that $v \cdot x = v \cdot y \bmod 2$. If $x = y$, this is always true: $P[v \cdot x = v \cdot y \bmod 2 \mid x = y] = 1$. Otherwise, we have

$$v \cdot x - v \cdot y \bmod 2 = v \cdot (x - y) \bmod 2 = \sum_{\substack{i \in \{1,\dots,N\} \\ x_i \neq y_i}} v_i \bmod 2,$$

and as v is a string of independent random bits, $P[v \cdot x = v \cdot y \bmod 2 \mid x \neq y]$ is the probability that the number of heads for $|\{i \in \{1,\dots,N\} \mid x_i \neq y_i\}| > 0$ unbiased coin flips is even. This is exactly $1/2$ for a single coin flip and $P[v \cdot x = v \cdot y \bmod 2 \mid x \neq y] = 1/2$ can be shown by induction.

In summary, testing whether $v \cdot x = v \cdot y \bmod 2$ reveals with probability $1/2$ that $x \neq y$ and will never yield a false negative if $x = y$. With kN public random bits, Alice and Bob have k independent random vectors. The probability that the test fails k times is 2^{-k}. It remains to show that only $k + 1$ bits need to be exchanged. To this end, Alice sends, for each of the k random vectors v, $v \cdot x \bmod 2$ (i.e., 1 bit) to Bob. Bob then compares to $y \cdot v$ for each v and sends the result to Alice (1 bit). □

This is great, but what's up with this excessive use of public random bits? Of course, we can generate public random bits by communicating private random bits, but then the communication complexity of the protocol would become

worse than the trivial solution! It turns out that there's a much more clever way of doing this.

Theorem 7.8. *Given a protocol for equality that uses public randomness and has error probability ε, we can construct a protocol for randomized equality with error probability 2ε that uses $\mathcal{O}(\log N + \log 1/\varepsilon)$ public random bits.*

Proof. For simplicity, assume that $6N/\varepsilon$ is integer (otherwise round up). Select $6N/\varepsilon$ random strings uniformly and independently at random and fix this choice.

Now consider an input (x, y) to the equality problem. For most of the fixed random strings, the original protocol will succeed, for some it may fail. Let us check the probability that it fails for more than a 2ε-fraction of these strings. The number of such "bad" strings is bounded from above by the sum X of $6N/\varepsilon$ independent Bernoulli variables that are 1 with probability ε. Thus, $E[X] = 6N$. By Chernoff's bound,

$$P[X \geq 12N] = P[X \geq 2E[X]] \leq e^{-E[X]/3} = e^{-2N} < 2^{-2N}.$$

By the union bound, the probability that there is *any* pair (x, y) for which there are more than $12N$ "bad" strings among the $6N/\varepsilon$ selected ones is at most

$$\sum_{(x,y)} P[X \geq 12N] < \sum_x \sum_y 2^{-2N} = 1.$$

Thus, with non-zero probability, our choice of $6N/\varepsilon$ random strings is "good" for all inputs (x, y). In particular, there *exists* a choice for which this holds! Fix such a choice of $6N/\varepsilon$ (now non-random) strings, i.e., for no (x, y) there are more than $12N$ strings for which the original algorithm with these strings as "public random bits" outputs the wrong result. Picking one such string uniformly at random and executing the protocol will thus fail with probability at most

$$\frac{12N}{6N/\varepsilon} = 2\varepsilon.$$

We make the list of these strings part of the new algorithm's code. Alice and Bob now simply pick one entry from the list uniformly at random (using public randomness) and execute the original algorithm with this string as random input. This errs with probability at most 2ε and requires

$$\left\lceil \log\left(\frac{6N}{\varepsilon}\right) \right\rceil \in \mathcal{O}(\log N + \log 1/\varepsilon)$$

public random bits. □

Corollary 7.9. *Randomized equality can be solved with error probability $N^{-\Theta(1)}$ with private randomness and $\mathcal{O}(\log N)$ bits of communication.*

Proof. We apply Theorem 7.8 to the algorithm obtained from Lemma 7.7 for a choice $k \in \Theta(\log N)$. We obtain an algorithm using $\mathcal{O}(\log N)$ bits of public randomness and achieving error probability $N^{-\Theta(1)}$. To make the randomness private, we let Alice choose the $\Theta(\log N)$ random bits and communicate them; this does not affect the asymptotic bit complexity. □

Remarks:

- Apart from showing off with Chernoff's bound, we got to see the *probabilistic method* in action here. We used a probabilistic argument to show that something happens with non-zero probability. Regardless of how small the probability is, it means that there *exists* some deterministic choice achieving the property that held with non-zero probability.
- Communication complexity people suffer from the same illness as distributed computing folks: They don't care about local computations.
- Here, this is quite bad. When *constructing* the algorithm, we cannot be *sure* that it actually has the desired guarantee on the error probability without explicitly checking for all inputs, which requires exponential computations.
- Even if we do this in advance, this causes the additional trouble that we need to assume a bound on N. If Alice and Bob get larger inputs, they are screwed!
- On top of this, Alice and Bob require polynomial memory. The simple algorithm using shared randomness can handle everything using only $\mathcal{O}(\log N)$ bits besides the one for the inputs and random bit string.
- In the exercises, you will see a deterministic polynomial time construction.

7.4 Handling Randomization and Approximation

So, equality is not good enough to handle randomization. It also does not cope very well with approximation algorithms, at least not in the construction we've seen. We need a communication complexity problem that is hard even for randomized algorithms – and ideally, it should yield an all-or-nothing construction for which the MST has non-zero weight only if the answer to the communication complexity problem is "yes."

Definition 7.10 (2-Player Set Disjointness)**.** *The deterministic and randomized versions of the* set disjointness *problem are defined as for equality, with the difference that the goal is to decide whether x and y encode disjoint sets, i.e., whether $\exists i \in \{1, \dots, N\}$ so that $x_i = y_i = 1$.*

This problem is hard also for randomized algorithms.

Theorem 7.11 ([KS92, Raz92])**.** *The communication complexity of set disjointness is $\Omega(n)$, even for randomized algorithms with error probability $1/3$.*

How can we encode this in our graph? It's even easier than before:

- Use the same topology, but with only k paths.
- Pick all edge weights as before, except for the edges from Alice and Bob to the endpoints of paths.
- For $i \in \{1, \dots, k\}$, the edge from Alice to path p_i has weight x_i.

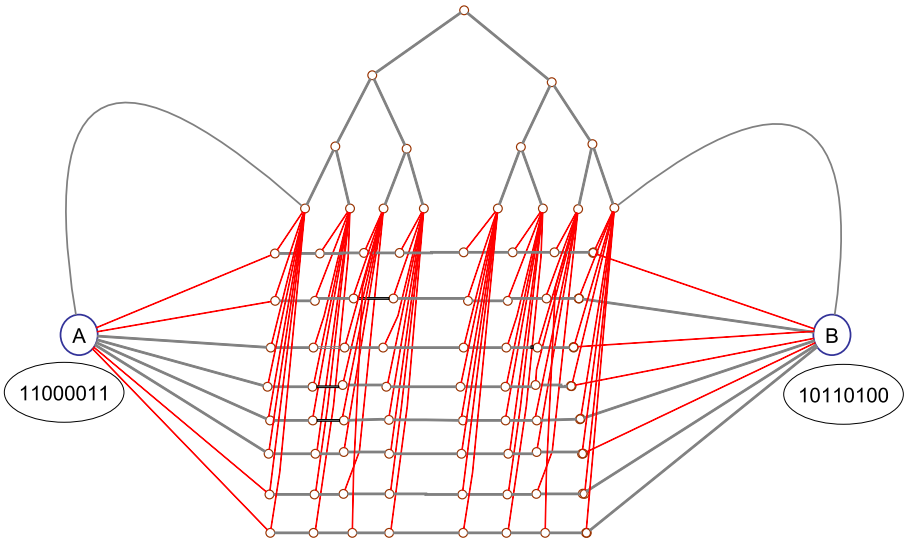

Figure 7.4: How to use the same topology as in Figure 7.1 to encode a set disjointness instance. Now there is a one-to-one correspondence between input bits and paths p_i.

- For $i \in \{1, \ldots, k\}$, the edge from Bob to path p_i has weight y_i.

Lemma 7.12. *The weight of an MST of the modified graph is 0 if and only if x and y encode disjoint sets.*

Proof. As before, the question is how expensive it is to connect the paths to the rest of the graph. If x and y encode disjoint sets, then for all $i \in \{1, \ldots, k\}$ we have that $x_i = 0$ or $y_i = 0$, implying that there is an edge leaving the path of weight 0. If the sets are not disjoint, there is a path p_i with $x_i = y_i = 1$, which therefore cannot be connected to the remaining graph by a 0-weight edge. □

Corollary 7.13. *There is no distributed MST approximation algorithm that uses messages of size $\mathcal{O}(\log n)$ and terminates in $o(\sqrt{n}/\log^2 n + D)$ rounds on all graphs of n nodes and diameter D (unless D is smaller than in the graph from Figure 7.1).*

Proof. Theorem 6.2 shows that running time $o(D)$ is impossible, which shows the claim if $D \geq \sqrt{n}/\log^2 n$. Analogously to Theorem 7.3, based on Lemma 7.12 we can show that an $o(\sqrt{n}/\log^2 n)$-round algorithm implied a solution to set disjointness using $o(N)$ bits; this also holds for approximation algorithms, as the weight of the MST is 0 if x and y represent disjoint sets. By Theorem 7.11, this is not possible. This covers the case that $D \leq \sqrt{n}/\log^2 n$. □

- Unfortunately, showing that set disjointness is hard is much more involved than the straightforward argument for deterministic equality.

- In some sense, this bound means that we hit the wall. The hardness comes from set disjointness, not any fancy aspect of the model.

- On the other hand, one can refine the granularity further by taking into account other parameters. We will see an example for this in a future lecture.

What to take Home

- The machinery demonstrated today produces essentially the same lower bound for plenty of other important graph problems. There will be some examples in the exercises. In some cases, the techniques shows even bounds that are (almost) $\Omega(n)$!
- As a result, communication complexity lower bounds are *the* tool for showing distributed lower bounds arising from congestion. This is very natural, as distributed graph problems are essentially peculiar n-player communication complexity problems, with the addition of a notion of time!
- In many cases, deriving lower bounds of this type is quite easy once one is familiar with the technique. Typically, set disjointness is the source of hardness, Theorem 7.3 works for any graph where the bandwidth available for algorithms between the "input-encoding" parts is small if the running time is small, and all one needs to do is find a suitable graph and encode the instance. Even better: the graph shown works for lots of problems as off-the-shelf topology, only the weights need to be adjusted!
- The probabilistic method, which was only supporting actor today, is also very useful. There are more "constructive" variants, like the celebrated (constructive versions of the) Lovász Local Lemma.
- Another bunch of examples for the utility of simulation results. Both derivation of lower bounds and algorithms become easier this way, as obstacles are separated and handled in individual steps.

Bibliographic Notes

The first lower bound on MST construction, by Peleg and Rubinovich [PR00], applied only to deterministic exact algorithms. This boils down to the fact that they exploited the hardness of equality, not set disjointness. Elkin [Elk06] extended the result to randomized approximation algorithms. However, in his construction the lower bound deteriorated depending on the approximation ratio of the algorithm; this was resolved by Das Sarma et al. [SHK$^+$12], who list a large number of related problems for which the technique also yields "the" lower bound of roughly $\Omega(\sqrt{n})$.

For the basics of communication complexity, see, e.g., [KN97]. The first strong lower bound on the randomized communication complexity of set disjointness is due to Babai, Frankl, and Simon [BFS86], showing that $\Omega(\sqrt{N})$ bits are required. They sampled x and y independently from the N-bit strings with roughly $\sqrt{N}$ non-zeros. They show that one *has* to look for more complex distributions; essentially, the birthday paradoxon is the monkey wrench in the works. The $\Omega(N)$ hardness was shown by Kalyanasundaram and Schintger [KS92]; a simplified proof was given by Razborov [Raz92].

One can dial it up to eleven and show quantum distributed computing complexity lower bounds [EKNP14] or derive bounds on multi-party set disjointness in the message passing model [BEO+13], which in turn permits to show hardness by reduction from such problems.

Bibliography

[BEO+13] Mark Braverman, Faith Ellen, Rotem Oshman, Toniann Pitassi, and Vinod Vaikuntanathan. A Tight Bound for Set Disjointness in the Message-Passing Model. In *54th Symposium on Foundations of Computer Science (FOCS)*, pages 668–677, 2013.

[BFS86] Laszlo Babai, Peter Frankl, and Janos Simon. Complexity Classes in Communication Complexity Theory. In *27th Symposium on Foundations of Computer Science (FOCS)*, pages 337–347, 1986.

[EKNP14] Michael Elkin, Hartmut Klauck, Danupon Nanongkai, and Gopal Pandurangan. Can Quantum Communication Speed Up Distributed Computation? In *Proc. 2014 Symposium on Principles of Distributed Computing (PODC)*, pages 166–175, 2014.

[Elk06] Michael Elkin. An Unconditional Lower Bound on the Time-Approximation Trade-off for the Distributed Minimum Spanning Tree Problem. *SIAM Journal on Computing*, 36(2):433–456, 2006.

[KN97] E. Kushilevitz and N. Nisan. *Communication complexity*. Cambridge University Press, 1997.

[KS92] B. Kalyanasundaram and G. Schintger. The Probabilistic Communication Complexity of Set Intersection. *SIAM Journal on Discrete Mathematics*, 5(4):545–557, 1992.

[PR00] David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM Journal on Computing*, 30(5):1427–1442, 2000.

[Raz92] A. A. Razborov. On the Distributional Complexity of Disjointness. *Theoretical Computer Science*, 106(2):385–390, 1992.

[SHK+12] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM Journal on Computing*, 41(5):1235–1265, 2012.

Lecture 8

Distance Approximation and Routing

Knowing how to construct a minimum spanning tree is very useful to many problems, but it is not always enough. Cheaply connecting all nodes is one thing, but what about finding a short path between an arbitrary pair of nodes? Clearly, an MST is not up to the task, as even for a single edge, the route in the MST might be factor $n-1$ more costly: Just think of a cycle!

Trivially, finding the distance between two nodes is a global problem, just like MST. However, the connection runs deeper. As we saw in the exercises for the previous lecture, even just *approximating the weight* of a shortest s-t path requires $\Omega(\sqrt{n}/\log^2 n + D)$ rounds in the worst case (with messages of size $\mathcal{O}(\log n)$).

Doing this every time a routing request is made would take too long and cause a lot of work. We're too lazy for that! So instead, we preprocess the graph and construct a distributed data structure that helps us serving such requests.

Definition 8.1 (All-Pairs-Shortest-Paths (APSP))**.** *In the distributed* all-pairs-shortest-paths (APSP) problem, *we are given a weighted, simple, connected graph $G = (V, E, W)$. The task is for each node $v \in V$ to compute a* routing table*, so that given an identifier $w \in V$, v can determine* $\operatorname{dist}(v, w)$*, the (weighted) distance from v to w, i.e., the minimum weight of a path from v to w, and the next node on a shortest path from v to w. For $\alpha > 1$, an α-approximation merely guarantees the that the stated distance d satisfies* $\operatorname{dist}(v, w) \leq d \leq \alpha \operatorname{dist}(v, w)$ *and that the routing path has weight at most* $\alpha \operatorname{dist}(v, w)$*.*

We will solve this problem in the synchronous message passing model without faults. In other words, we accept some preprocessing out of necessity, but refuse to fall back to a centralized algorithm!

8.1 APSP is Hard

As mentioned above, we know that we should not even try to find an algorithm faster than (roughly) $\Omega(\sqrt{n})$.

Corollary 8.2. *An α-approximate solution to APSP with $\mathcal{O}(\log n)$-bit messages requires $\Omega(\sqrt{n}/\log^2 n + D)$ rounds, regardless of α.*

However, things are much worse. Even in an unweighted (i.e., $W(e) = 1$ for all $e \in E$) *tree of depth 2*, solving APSP requires $\Omega(n/\log n)$ rounds!

Theorem 8.3. *Any deterministic α-approximation to APSP with $\mathcal{O}(\log n)$-bit messages requires $\Omega(n/\log n)$ rounds, even in trees of depth 2.*

Proof. Consider a tree whose root has two children, which together in total have k children with identifiers $1, \dots, k$. We consider all such graphs. Note that the root can only tell them apart by the bits that its two children send, which are $\mathcal{O}(R \log n)$ for an R-round algorithm. The number of different routing tables the root may produce is thus $2^{\mathcal{O}(R \log n)}$. Note also that for each of the considered graphs, we need a different routing table: Any two partitions of k nodes must differ in at least one node, for which the routing table then must output a different routing decision. How many such partitions are there? Well, 2^k – just decide for each node $1, \dots, k$ to which child of the root it's attached. Hence,

$$k \in \mathcal{O}(R \log n),$$

or $R \in \Omega(k/\log n) = \Omega(n/\log n)$, as the considered graph family has $n = k + 3$ nodes. □

Will randomization save us? Not today. If the number of bits received by the root is too small, it will in fact err with a large probability.

Corollary 8.4. *Any randomized α-approximation to APSP with $\mathcal{O}(\log n)$-bit messages requires $\Omega(n/\log n)$ rounds, even in trees of depth 2.*

Proof. Suppose there's a randomized algorithm that terminates in $o(n/\log n)$ rounds. Fix the random bit strings of all nodes, and execute the resulting *deterministic* algorithm, on a *uniformly random* topology as in the proof of Theorem 8.3. Now, as there are $2^{o(n)}$ different bit strings the root can possibly receive in $R \in o(n/\log n)$ rounds, the probability that the algorithm computed a correct table is at most $2^{o(n)}/2^n = 2^{(1-o(1))n}$, i.e., astronomically small. As we used the same random distribution of topologies *irrespectively of the assigned random bits*, we can now argue that choosing the random bit strings of the nodes uniformly and independently at random *after* we picked the topology yields the same result. □

Remarks:

- The above corollary is an application of *Yao's principle*. If one provides a distribution of inputs, no randomized algorithms can perform better than the best deterministic algorithm for this distribution.
- For exact algorithms with weights, the bound becomes $\Omega(n)$: Just add a weight from $1, \dots, n$ to each edge to a leaf, resulting in n^k distinct combinations, even with a single child!
- This also shows that if we only care about distances (and not how to route), we're still skrewed. Even for approximate distances we can make sure that there are at least two different "classes" of distances for each node that need to be distinguished.

- Essentially, the bound still holds even if we permit *dynamic* routing (without knowing distances), where nodes on the routing path may attach some routing information to the message. This way, one can "check" whether the destination is attached to a child and return to the root if the decision was wrong. One then uses $\Theta(\rho)$ children of the root to show that a ρ-approximation (even on average) is not possible in $o(n/(\rho^2 \log n))$ rounds.
- In n rounds, everyone can learn the entire tree, so at least for this family of graphs the bound is tight. Let's see what we can do for arbitrary graphs!

8.2 Exact APSP in Unweighted Graphs

If a problem appears to be difficult, one shouldn't always try to take on the most general form first. We start by considering unweighted graphs.[1] In a nutshell, solving APSP here is equivalent to constructing for each node a BFS tree rooted at it.

The setting is synchronous, so we know how to do this for a single node in $\mathcal{O}(D)$ rounds. The challenge is that the different constructions might interfere. We have seen that we cannot avoid this completely, as it will take $\Omega(n)$ rounds even if $D \in \mathcal{O}(1)$, but we can still hope for a running time that is much faster than the trivial solution of running n instances of the Bellman-Ford algorithm sequentially, i.e., $\Theta(Dn)$ rounds.

It turns out that there is a straightforward solution to this problem.[2] We employ Bellman-Ford for all sources concurrently, where always the seemingly most useful piece of information is communicated. "Seemingly most useful" here means to always announce the closest node that hasn't been announced before, breaking ties by identifiers. "Source" refers to a node $s \in S \subseteq V$; as we will see, the algorithm works very well for the more general setting where only distances to a subset $S \subseteq V$ of nodes are to be determined.

Definition 8.5 (Total order of distance/node pairs)**.** *Let $(d_v, v), (d_w, w) \in \mathbb{N}_0 \times V$ be two distance/node pairs. Then*

$$(d_v, v) < (d_w, w) \quad \Leftrightarrow \quad (d_v < d_w) \vee (d_v = d_w \wedge v < w).$$

Here the comparison "$v < w$" means to numerically compare the identifiers of v and w.

In the following, we consider all sets of distance/node pairs to be ordered ascendingly according to the above definition (and consequently refer to them as lists).

Let's fix some helpful notation.

Definition 8.6. *For each node $v \in V$ and each round $r \in \mathbb{N}$, denote by L_v^r the content of v's L_v variable at the end of round r; by L_v^0 we denote the value at initialization. Furthermore, define $L_v := \{(\mathrm{dist}(v, s), s) \mid s \in S\}$.[3] For $h \in \mathbb{N}_0$, denote by $L_v(h)$ the sublist of L_v containing only elements $(\mathrm{dist}(v, s), s)$ with*

[1] That's not how we did it, but there's no reason you shouldn't learn from our mistakes!

[2] Actually several, but we're going for the one that will be most useful later on.

[3] This is slight abuse of notation; we will show that the algorithm returns exactly this L_v, though.

Algorithm 17 Pipelined Bellman-Ford, code at node v. Initially, v knows whether it is in S, as well as parameters $H, K \in \mathbb{N}$. Remembering the sender for each entry in L_v reveals the next routing hop on a shortest path to the respective source w.

```
1: if v ∈ S then
2:    L_v := {(0, v)}
3: else
4:    L_v := {}
5: end if
6: for i = 1, ..., H + K − 1 do
7:    (d_s, s) := smallest element of L_v not sent before (⊥ if there is none)
8:    if (d_s, s) ≠ ⊥ then
9:       send (d_s + 1, s) to all neighbors
10:   end if
11:   for each (d_s, s) received from a neighbor do
12:      if ∄(d'_s, s) ∈ L_v : d'_s ≤ d_s then
13:         L_v := L_v ∪ {(d_s, s)}
14:      end if
15:      if ∃(d'_s, s) ∈ L_v : d'_s > d_s then
16:         L_v := L_v \ {(d'_s, s)}
17:      end if
18:   end for
19: end for
20: return L_v
```

$\text{dist}(v,s) \leq h$. *For $k \in \mathbb{N}$ denote by $L_v(h,k)$ the sublist of the (up to) k first elements of $L_v(h)$.*

We will show that Algorithm 17 guarantees that after r rounds, for $h + k \leq r + 1$, the first $|L_v(h,k)|$ entries of L_v^r are already correct. Inserting $h = D$ and $k = n$, we will then see that the algorithm indeed returns the lists L_v.

With the right induction hypothesis, the proof is actually going to be quite simple. Let's assemble the pieces first.

Lemma 8.7. *If $(d_w, w) \in L_v^r$ for any $r \in \mathbb{N}_0$, then $w \in S$ and $d_w \geq \text{dist}(v,w)$.*

Proof. We never add entries for nodes that are not in S. Moreover, initially for each $s \in S$ only s has an entry $(0,s) \in L_s^0$. As we increase the d-values by one for each hop, it follows that $d_s \geq \text{dist}(v,s)$ for any entry $(d_s, s) \in L_v^r$. □

Corollary 8.8. *If for any $s \in S$ and $v \in V$, it holds that v receives $(\text{dist}(v,s), s)$ from a neighbor in round $r \in \mathbb{N}$ (or already stores it on initialization), then $(\text{dist}(v,s), s) \in L_v^{r'}$ for all $r' \geq r$. Moreover, if $L_v(h,k) \subseteq L_v^r$ for any $r \in \mathbb{N}_0$, it is in fact the head of the list L_v^r.*

Lemma 8.9. *For all $h, k \in \mathbb{N}$ and all $v \in V$,*

$$L_v(h,k) \subseteq \{(\text{dist}(w,s)+1, s) \mid (\text{dist}(w,s), s) \in L_w(h-1,k) \wedge \{v,w\} \in E\} \cup \{(0,v)\}.$$

Proof. Since $(\text{dist}(v,v),v) = (0,v)$, the case of $v \in S$ is covered. Hence, suppose $(\text{dist}(v,s),s) \in L_v(h,k)$ for some $s \neq v$. Consider a neighbor w of v on a shortest path from v to s. We have that $\text{dist}(w,s) = \text{dist}(v,s) - 1 \leq h-1$. Hence, it suffices to show that $(\text{dist}(w,s),s) \in L_w(h-1,k)$. Assuming otherwise, there are k elements $(\text{dist}(w,s'),s') \in L_w(h-1,k)$ satisfying that $(\text{dist}(w,s'),s') \leq (\text{dist}(w,s),s)$. Hence, $(\text{dist}(v,s'),s') \leq (\text{dist}(w,s')+1,s') \leq (h,s')$, and if $\text{dist}(v,s') = \text{dist}(v,s)$, then also $\text{dist}(w,s') = \text{dist}(w,s)$ and thus $s' < s$. It follows that $(\text{dist}(v,s'),s') < (\text{dist}(v,s),s)$. But this means there are at least k elements in $L_v(h,k)$ that are smaller than $(\text{dist}(v,s),s)$, contradicting the definition of $L_v(h,k)$! □

Now we can prove the statement sketched above.

Lemma 8.10. *For every node $v \in V$, $r \in \{0,\dots,H+K-1\}$, and $h+k \leq r+1$,*

(i) $L_v(h,k) \subseteq L_v^r$, and

(ii) v has sent $L_v(h,k)$ by the end of round $r+1$.

Proof. We show the statement by induction on r. It trivially holds for $k=0$, as well as for $h=0$ and all k, as $L_v(0,k) = \{(0,v)\}$ if $v \in S$ and $L_v(0,k) = \emptyset$ otherwise, and clearly this will be sent by the end of round 1. In particular, the claim holds for $r=0$.

Now suppose both statements hold for $r \in \mathbb{N}_0$ and consider $r+1$. As the case $h=0$ is already covered, we may assume that $h > 0$. By the induction hypothesis (Statement (ii) for r), for $h+k \leq r+1$, node v has already received the lists $L_w(h-1,k+1)$ and $L_w(h,k)$ from all neighbors w. By Lemma 8.9, v thus has received all elements of $L_v(h,k+1)$ and $L_v(h+1,k)$. By Corollary 8.8, this implies Statement (i) for $h+k \leq r+2 = (r+1)+1$.

It remains to show Statement (ii) for $h+k = r+2$. Since we just have shown (i) for $h+k = r+2$, we know that $L_v(h,k) \subseteq L_v^{r+1}$ for all $h+k = r+2$. By Corollary 8.8, these are actually the first elements of L_v^{r+1}, so v will sent the next unsent entry in round $r+2$ (if there is one). By the induction hypothesis, v sent $L_v(h,k-1)$ during the first $r+1$ rounds (where $L_v(h,0) := \emptyset$), hence only $L_v(h,k) \setminus L_v(h,k-1)$ may still be missing. As $|L_v(h,k) \setminus L_v(h,k-1)| \leq 1$ by definition, this proves (ii) for $h+k = r+2$. This completes the induction step and thus the proof. □

Corollary 8.11. *APSP on unweighted graphs can be solved with message size $\mathcal{O}(\log n)$ in $n + \mathcal{O}(D)$ rounds.*

Proof. We construct a BFS tree, count the number of nodes and determine the depth d of the BFS tree; this takes $\mathcal{O}(D)$ rounds, and we have that $d \leq D \leq 2d$. The root then initiates Algorithm 17 with $S = V$, $H = 2d$, and $K = n$, so that all nodes jointly start executing it in some round $R_0 \in \mathcal{O}(D)$. As for $S = V$, $L_v = L_v(D,n) = L_v(2d,n)$ (and remembering senders yields routing information), Lemma 8.10 shows that this solves APSP. □

Remarks:

- Somewhat depressing, but we have seen that this is essentially optimal.
- We've actually shown something stronger. For *any* $S \subseteq V$ and *any* $h, k \in \mathbb{N}$, we can determine $L_v(h,k)$ at all nodes $v \in V$ in $h + k - 1$ rounds.
- There's a straightforward example showing that this is the best that's possible for *any* h and k. Even more depressing!
- What do we do when we're getting depressed due to lower bounds? We change the rules of the game!

8.3 Relabeling

Basically, the lower bound might mean that we haven't asked the right question. The problem is that we insisted on using the original identifiers. If there are bottleneck edges – like in the above construction the edges between the root and its children – this dooms us (modulo nitpicking over details) to transmit them all over these edges. The problem is easily resolved if we permit *relabeling*.

Definition 8.12 (APSP with Relabeling)**.** *The APSP problem with* relabeling *is identical to the APSP problem from Definition 8.1, except that each node now also outputs a* label*. The task is now to construct a routing table and a label $\lambda(v)$ at each node v so that, given $\lambda(w)$ of some node w, v can determine the distance and next routing hop to w. Approximate solutions are defined as before.*

How does this help us? Let's consider a peculiar special case first: in a tree, we want to be able to route from the root to each node.

Lemma 8.13. *Suppose we are given a tree (V,E) of depth d. Using messages of size $\mathcal{O}(\log n)$, in $\mathcal{O}(d)$ rounds we can determine routing tables and assign labels $1,\dots,|V|$ such that given the label $\lambda(v)$ of node $v \in V$, we can route from the root to v.*

Proof. We enumerate the tree nodes in a pre-order depth-first-search manner.[4] In a distributed fashion, this is done as follows.

1. Determine for each $v \in V$ the number of nodes in its subtree. This is done in a bottom-up fashion in $\mathcal{O}(d)$ rounds: each node announces the number of nodes in its subtree to its parent, starting from the leaves.
2. The root labels itself 1 and assigns to each child a range of labels matching the size of its subtree. Each child then takes the first label from its assigned range and splits the remaining labels between its children in the same way. This top-down procedure takes $\mathcal{O}(d)$ rounds, too. Note that since the assigned ranges are consecutive, they can be communicated using $\mathcal{O}(\log n)$ bits by announcing the smallest and largest element of the respective interval.

The tables at each node store the label ranges assigned to the children. Hence, given a label $\lambda(w)$ of a node w, each node on the unique path from the root to w can determine the next routing hop. □

[4] First list the root, then recursively list the subtrees rooted at its children, one by one.

Remarks:

- This construction is inefficient in terms of memory, i.e., the size of tables. In a tree, one can be much more efficient and have tables of size $\log^{\mathcal{O}(1)} n$, without increasing label size significantly.
- We can also make distances available. Each node learns its distance to the root ($\mathcal{O}(d)$ rounds; simply do a "flooding" that sums up the weights of traversed edges) and adds it to the label. The resulting labels have size $\mathcal{O}(\log n)$.
- While handling trees does not seem very impressive, the labels help circumvent the bottleneck problem we might experience with the original identifiers. Let's now handle general (unweighted) graphs!

8.4 Fast APSP with Relabeling: The Unweighted Case

From a previous exercise, we know that approximating the diameter of an unweighted graph better than factor 3/2 takes $\Omega(n/\log n)$ rounds. Hence, we'll have to live with getting only an approximation if we want to obtain a faster algorithm. The key idea in Algorithm 18 is to use a small set of "landmarks" to navigate to distant nodes, while handling close-by nodes directly:

The landmarks $S \subseteq V$ are sampled. Each node $v \in V$ is assigned to the landmark $s_v \in S$ that is closest to v. Now each node learns the following things: (i) its landmark s_v, (ii) the next hop on a shortest path to all nodes closer than some threshold, and (iii) the next hop on a shortest path to any landmark $s \in S$, no matter how far away it is. Landmarks use the routing trick from Section 8.3 to reach all nodes assigned to them. The label of v consists of three parts: (i) the ID of s_v, (ii) a bit indicating whether $v \in S$, and (iii) the label for the routing tree of s_v as in Section 8.3. Also see Figure 8.1. The key idea is that v either knows a shortest path to w because $\text{dist}(v, w)$ is small enough, or that w is so far away that it becomes acceptable to take the detour via s_w (extracted from $\lambda(w)$) and then to w (s_w knows how to get there).

Algorithm 18 5-approximate APSP with relabeling in unweighted graphs. By c we denote a sufficiently large constant.

1: determine n and $\tilde{D} \in [D, 2D]$ and make both known to all nodes
2: sample each node into $S \subseteq V$ with independent probability $c\sqrt{\log n/n}$
3: determine $|S|$ and make it known to all nodes
4: add to each node's identifier a bit indicating whether it is in S
5: for source set S, compute $L_v(\tilde{D}, |S|)$ for all $v \in V$
6: for each $v \in V$, $s_v := \text{argmin}_{s \in S}\{\text{dist}(v, s)\}$
7: for each $s \in S$, compute labels $\lambda_s(v)$ for routing from/distances to the root s of the (partial) BFS tree with nodes $\{v \in V \mid s_v = s\}$ rooted at s
8: relabel each $v \in V$ by $\lambda(v) := (s_v, \lambda_{s_v}(v))$
9: for source set V, compute $L_v(\sqrt{n \log n}, \sqrt{n \log n})$
10: **return** labels $\lambda(v)$ and all computed tables

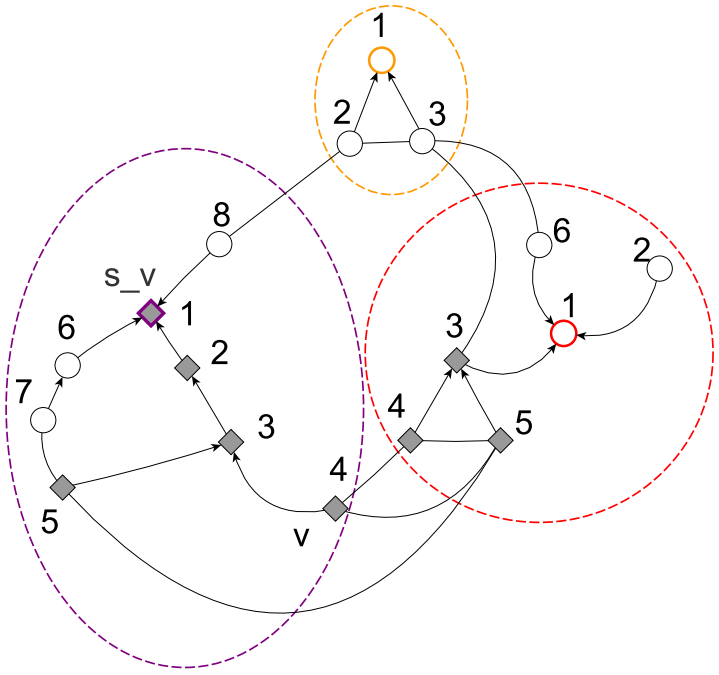

Figure 8.1: An example of the "clustering" constructed for the hierarchical routing scheme. The dotted ovals indicate the regions belonging to the sampled node framed in the same color. The oriented edges are part of the shortest-path tree rooted at that node. Each node is labeled by the identifier of its root and the number assigned to it according to the DFS enumeration of the trees (only these are written next to the nodes). The grey nodes are those for which node v (labeled $(s_v, 4)$) knows how to route directly to.

The algorithm is for label and table construction. Before we discuss that it can be implemented quickly, let's first explain how we can route and estimate distances with approximation factor at most 5. For routing and distance estimation, given a label $\lambda(w)$ at a node v, v does the following:

- If $\exists(\text{dist}(v,w),w) \in L_v(\sqrt{n\log n}, \sqrt{n\log n})$ for sources V, then v knows $\text{dist}(v,w)$ and knows the next hop on a shortest path to w.
- Otherwise, we first route from v to s_w (s_w is part of $\lambda(w)$) using the tables for $L_v(\tilde{D}, |S|) = L_v(D, |S|)$ and then from s_w to w using the tree label $\lambda_{s_w}(w)$. The distance is estimated as $\text{dist}(v, s_w) + \text{dist}(s_w, w)$, where $\text{dist}(s_w, w)$ is available from $\lambda_{s_w}(w)$.

Let's first show that this is indeed a factor-5 approximation if for each v, s_v is close enough.

Lemma 8.14. *Suppose* $(\text{dist}(v, s_v), s_v) \in L_v(\sqrt{n\log n}, \sqrt{n\log n})$ *with source set* V *for all* $v \in V$*, then the above routing and distance approximation scheme has approximation factor at most 5.*

Proof. If $(\text{dist}(v,w),w) \in L_v(\sqrt{n\log n},\sqrt{n\log n})$, then the solution is optimal. If not, the assumption that $(\text{dist}(v,s_v),s_v) \in L_v(\sqrt{n\log n},\sqrt{n\log n})$ implies that $(\text{dist}(v,s_v),s_v) \leq (\text{dist}(v,w),w)$. In particular, $\text{dist}(v,s_v) \leq \text{dist}(v,w)$. As by definition $\text{dist}(w,s_w) \leq \text{dist}(w,s_v)$, the triangle equality yields that

$$\begin{aligned}\text{dist}(v,s_w)+\text{dist}(s_w,w) &\leq \text{dist}(v,w)+\text{dist}(w,s_w)+\text{dist}(s_w,w)\\ &= \text{dist}(v,w)+2\,\text{dist}(w,s_v)\\ &\leq \text{dist}(v,w)+2(\text{dist}(w,v)+\text{dist}(v,s_v))\\ &\leq 5\,\text{dist}(v,w).\end{aligned}$$

□

Using Chernoff's bound, it's straightfoward to see that the prerequisite of this lemma is satisfied w.h.p.

Lemma 8.15. *W.h.p.,* $(\text{dist}(v,s_v),s_v) \in L_v(\sqrt{n\log n},\sqrt{n\log n})$ *for all* $v \in V$.

Proof. We sampled nodes into S with independent probability $c\sqrt{\log n/n}$. The expected number of nodes from S among a set of at least $\sqrt{n\log n}$ nodes – in particular the nodes indicated by $L_v(\sqrt{n\log n},\sqrt{n\log n})$ for a given $v \in V$ – is thus at least $c\log n$. By Chernoff's bound, the probability that the number of such nodes is fewer than $c\log n/2$ is $2^{-\Omega(c\log n)} = n^{-\Omega(c)}$. As the constant c is assumed to be sufficiently large, we conclude that for each $v \in V$, it holds that $(\text{dist}(v,s_v),s_v) \in L_v(\sqrt{n\log n},\sqrt{n\log n})$ w.h.p. By the union bound, the joint event that this holds for all $v \in V$ occurs w.h.p., too. □

It remains to understand the time complexity of the construction. An immediate consequence of the above lemma is that the partial BFS trees rooted at the nodes in S are not too deep.

Corollary 8.16. *W.h.p., the partial BFS trees rooted at the nodes* $s \in S$ *containing the nodes* $\{v \in V \mid s_v = s\}$ *all have depth* $\mathcal{O}(\sqrt{n\log n})$.

Now we just need to check the complexities of the individual steps.

Corollary 8.17. *Algorithm 18 can be implemented such that it terminates in* $\mathcal{O}(\sqrt{n\log n}+D)$ *rounds w.h.p.*

Proof. Lines 2, 4, 6, 8, and 10 are local computations only. Lines 1 and 3 can be done in $\mathcal{O}(D)$ rounds by constructing and using a BFS tree. By Lemma 8.10, calling Algorithm 17 with source set S, $H = \tilde{D} \in \Theta(D)$, and $K = |S|$ will handle Line 5. By Chernoff's bound, $|S| \in \Theta(\sqrt{n\log n})$ w.h.p., i.e., this takes $\mathcal{O}(\sqrt{n\log n}+D)$ rounds w.h.p. By Lemma 8.13 and Corollary 8.16, Line 7 can be completed in $\mathcal{O}(\sqrt{n\log n}+D)$ rounds. By Lemma 8.10, calling Algorithm 17 with source set V and $K = H = \sqrt{n\log n}$ will yield lists containing $L_v(\sqrt{n\log n},\sqrt{n\log n})$; by Lemma 8.7, we can obtain the lists $L_v(\sqrt{n\log n},\sqrt{n\log n})$ by discarding all entries (d_w,w) with $d_w > \sqrt{n\log n}$ and truncating the list to (at most) $\sqrt{n\log n}$ elements. Summing this all up, we get the claimed running time bound. □

Theorem 8.18. *In unweighted graphs, we can find a 5-approximate solution to APSP using messages of size* $\mathcal{O}(\log n)$ *in* $\mathcal{O}(\sqrt{n\log n}+D)$ *rounds w.h.p.*

Proof. By Lemmas 8.14 and 8.15, the approximation guarantee holds w.h.p. By Corollary 8.17, the time bound is satisfied w.h.p. □

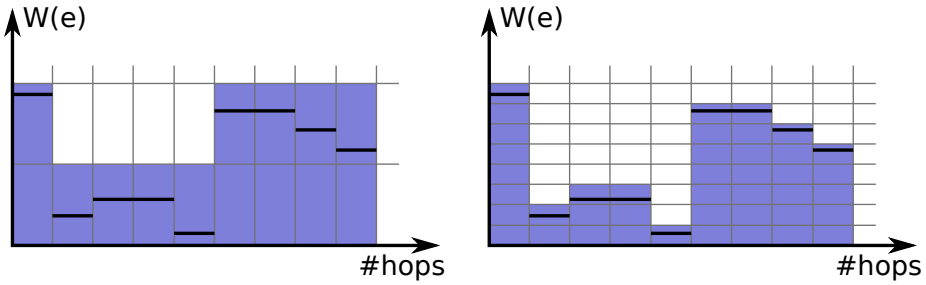

Figure 8.2: Approximating paths using coarse (left) and fine (right) weight classes. Coarse weight classes induce a larger error and require fewer hops, fine weight classes yield a better approximation at the expense of using more hops.

Remarks:

- One can reduce the approximation factor to 3 if one permits access to routing tables of *both* source *and* destination when determining where to route/approximating the distance, as then one can route via s_v or s_w, whatever is shorter.

- This is typically done in centralized constructions, where the main point is to make the tables small. In this context it makes sense to be able to access both tables, but in the distributed setting this would defeat the purpose.

- The argument in Lemma 8.14 can be used repeatedly for a sampling hierarchy of k levels (i.e., each node makes it to the next level with probability roughly $n^{-1/k}$), resulting in an $\mathcal{O}(k)$-approximation. This yields a running time of $\mathcal{O}(k(n^{1/k}\sqrt{\log n} + D))$. And one can make the tables to have size about $n^{1/k}$, too!

8.5 Weighted APSP*

In order to handle the weighted case, we reduce it to a small number of unweighted instances. Denote by $W_{\max} := \max_{e \in E}\{W(e)\}$ the maximum edge weight. Fix any constant $0 < \varepsilon \leq 1$. Set $i_{\max} := \lceil \log_{1+\varepsilon} W_{\max} \rceil$ and define for for $x \in \mathbb{R}$ and $i \in \{0, \ldots, i_{\max}\}$ that $\lceil\!\lceil x \rceil\!\rceil_i := (1+\varepsilon)^i \lceil W(e)/(1+\varepsilon)^i \rceil$, i.e., $\lceil\!\lceil \cdot \rceil\!\rceil_i$ rounds up to multiples of $b_i := (1+\varepsilon)^i$.

Now, given $G = (V, E, W)$, we define $G_i := (V, E, W_i)$ by $W_i(e) := \lceil\!\lceil W(e) \rceil\!\rceil_i$. Denoting by $\mathrm{dist}_i(v, w)$ the distance of v and w in G_i, obviously we have that $\mathrm{dist}_i(v, w) \geq \mathrm{dist}(v, w)$. The interesting bit is that there's a "sweet spot" for which $\mathrm{dist}_i(v, w) \leq (1+\varepsilon)\,\mathrm{dist}(v, w)$, yet $\mathrm{dist}_i(v, w)$ is roughly $\mathrm{hop}(v, w) b_i$, where $\mathrm{hop}(v, w)$ denotes the hop count of a shortest path from v to w in G, compare Figure 8.2.

Lemma 8.19. *For* $i(v, w) := \max\{0, \lfloor \log_{1+\varepsilon}(\varepsilon\,\mathrm{dist}(v, w)/\,\mathrm{hop}(v, w)) \rfloor\}$*, it holds that* $\mathrm{dist}_{i(v,w)}(v, w) \leq (1+\varepsilon)\,\mathrm{dist}(v, w) \in \mathcal{O}(b_{i(v,w)}\,\mathrm{hop}(v, w))$.

Proof. If $i(v,w) = 0$, we have that $\text{dist}_{i(v,w)} = \text{dist}(v,w)$. Otherwise,

$$\begin{aligned}\text{dist}_{i(v,w)}(v,w) &\leq \text{dist}(v,w) + b_{i(v,w)} \,\text{hop}(v,w)\\ &= \text{dist}(v,w) + (1+\varepsilon)^{i(v,w)} \,\text{hop}(v,w)\\ &\leq (1+\varepsilon)\,\text{dist}(v,w)\\ &\in \mathcal{O}(\text{dist}(v,w)).\end{aligned}$$

Regarding the second inequality, observe that

$$\begin{aligned}\text{dist}(v,w) &= \frac{\text{hop}(v,w)}{\varepsilon} \cdot \frac{\varepsilon\,\text{dist}(v,w)}{\text{hop}(v,w)}\\ &\leq \frac{\text{hop}(v,w)}{\varepsilon} \cdot (1+\varepsilon) b_{i(v,w)}\\ &\in \mathcal{O}(b_{i(v,w)} \,\text{hop}(v,w)).\end{aligned}$$

□

Theorem 8.20. *For any constant $\varepsilon > 0$, we can $(1+\varepsilon)$-approximate APSP in $\mathcal{O}(n \log n)$ rounds with messages of size $\mathcal{O}(\log n)$.*

Proof. By Lemma 8.19, for all $v, w \in V$ we have that

$$\text{dist}_{i(v,w)}(v,w) \leq (1+\varepsilon)\,\text{dist}(v,w) \in \mathcal{O}(b_{i(v,w)} \,\text{hop}(v,w)).$$

Replace for each G_i each edge of weight kb_i by a virtual path of k edges of weight 1. The result is an unweighted graph $\tilde{G}_i$. Denote by $L_{i,v}(h,k)$ the list for $\tilde{G}_i$; the lemma states that if we determine $L_{i(v,w),v}(\mathcal{O}(\text{hop}(v,w)),n) = L_{i(v,w),v}(\mathcal{O}(n),n)$, then there is an entry $(d,w) \in L_{i(v,w),v}(\mathcal{O}(n),n)$ such that $b_{i(v,w)}d \leq (1+\varepsilon)\,\text{dist}(v,w)$. Note also that we have $\text{dist}(v,w) \leq b_i d$ for any i and $(d,w) \in L_{i,v}(\mathcal{O}(n),n)$ (as we rounded weights up), as well as $i(v,w) \leq i_{\max}$, because $\varepsilon\,\text{dist}(v,w)/\,\text{hop}(v,w) \leq W_{\max}$. Consequently, for all $v, w \in V$ it holds that

$$\text{dist}(v,w) \leq \min_{i \in \{1,\ldots,i_{\max}\}} \{b_i d \,|\, (d,w) \in L_{i,v}(\mathcal{O}(n),n)\} \leq (1+\varepsilon)\,\text{dist}(v,w).$$

As the G_i are unweighted graphs and rounding edge weights can be done locally, we can compute for each i the lists $L_{i,v}(\mathcal{O}(n),n)$ concurrently in $\mathcal{O}(n)$ rounds by Corollary 8.11; the virtual nodes "on" edges are simply simulated by one of the nodes incident to the corresponding edge in G. As ε is a constant,

$$i_{\max} = \lceil \log_{1+\varepsilon} W \rceil \in \mathcal{O}(\log W) \subseteq \mathcal{O}(\log n).$$

□

Remarks:

- One can use this rounding approach also to construct faster approximate solutions.

What to take Home

- Sometimes a simplified version of the problem is worth studying, as the ideas turn out to be useful for more general cases, too.

- On the other hand, special cases may admit better solutions, and sometimes this is all we care about. For instance, in unweighted graphs one can solve APSP with small messages up to factor $\mathcal{O}(\log n)$ in $D \log^{\mathcal{O}(1)} n$ rounds. On weighted graphs, *any* algorithm that fast may hit the fan!
- If you want small messages: pipelining, pipelining, pipelining! Throw in some pipelining for good measure.

Bibliographic Notes

The almost linear lower bound for the APSP problem (without renaming) was shown independently and concurrently in two papers [Nan14, PSL13]. The exact unweighted APSP algorithm given here is from [LP13]. An elegant previous solution solving APSP in the same time was given concurrently and independently in two papers [HW12, PRT12]. However, this algorithm requires $\Omega(n)$ time for computing the lists $L_v(h,k)$ for *any* $h > 0$, $k > 1$, and $|S|$. The paper by Holzer and Wattenhofer [HW12] contains a second algorithm that achieves running time $\mathcal{O}(h + |S|)$ for that task. This is ok for a large variety of applications, but if we have $S = V$, the algorithm is slow. For the fast APSP approximation with relabeling, we need the algorithm presented here.

The tree relabeling scheme in this lecture is nothing more than a composition of folklore results. In contrast, the *compact* (i.e., little-memory and small-labels) tree labeling scheme by Thorup and Zwick [TZ01] is more clever and at the heart of many compact routing schemes!

The rounding technique for transforming the ($(1+\varepsilon)$-approximate) weighted problem into a collection of unweighted problems was used by Nanongkai [Nan14] in the distributed context. However, it found earlier application for the centralized APSP problem [Zwi02], for which the fastest known algorithms are based on fast matrix multiplication.

Bibliography

[HW12] Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *Proc. 31st ACM Symp. on Principles of Distributed Computing*, 2012.

[LP13] Christoph Lenzen and David Peleg. Efficient distributed source detection with limited bandwidth. In *Proc. 32nd ACM Symp. on Principles of Distributed Computing*, 2013.

[Nan14] Danupon Nanongkai. Distributed Approximation Algorithms for Weighted Shortest Paths. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC '14, pages 565–573, 2014.

[PRT12] David Peleg, Liam Roditty, and Elad Tal. Distributed algorithms for network diameter and girth. In *Proc. 39th Int. Colloq. on Automata, Languages, and Programming*, 2012.

[PSL13] B. Patt-Shamir and C. Lenzen. Fast Routing Table Construction Using Small Messages [Extended Abstract]. In *Proc. 45th Symposium on the Theory of Computing (STOC)*, 2013.

[TZ01] Mikkel Thorup and Uri Zwick. Compact Routing Schemes. In *Proc. 13th ACM Symp. on Parallel Algorithms and Architectures*, 2001.

[Zwi02] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, 2002.

Lecture 9

Self-Stabilization and Recovery

Previously, we have seen ways of handling permanent faults: nodes crashed and never booted up again, or they turned Byzantine, which was an untreatable condition. However, not every job needs the same tool. For instance, Byzantine tolerance needs that $f < n/3$ nodes are faulty, and that might not be true over extended periods of time. Instead, nodes or communication links (i.e., edges) may undergo *transient* failures. They recover eventually, but that doesn't mean the state of their volatile memory hasn't been corrupted!

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?*

Definition 9.1 (Self-Stabilization)**.** *Denote by $\mathcal{S}$ the* state space *of a distributed system, i.e., the product space over all individual nodes' state spaces. An* execution *of an algorithm is a (bounded or unbounded) sequence of state transitions that is valid in that it follows the rules given by the algorithm and the execution model. Define a set $\mathcal{C}$ of* correct executions*, i.e., executions that satisfy some desirable properties. The system/algorithm is called* self-stabilizing *(w.r.t. $\mathcal{C}$) iff every possible execution, no matter what the initial state, has a correct suffix.*

Usually, the information that the algorithm stabilizes is of limited use, as the time for it to do so might be unbounded. This means that typically we are interested in algorithms that have a small *stabilization time*. The definition of self-stabilization given above is very general, so there wasn't any notion of time. It needs to be derived from the execution model, which we didn't specify! So let's define what we mean for synchronous and asynchronous message passing systems.

Definition 9.2 (Stabilization time)**.** *Fix a set of parameters, e.g., the number of nodes n and the diameter of the network D. The* stabilization time *of a synchronous self-stabilizing system is the maximum number of rounds R so that removing the first R rounds from the execution results in a correct execution, taken over all executions on n-node graphs of diameter D. If this maximum doesn't exist, the algorithm has* unbounded *stabilization time. For an asynchronous system, we do the same with respect to asynchronous rounds, i.e., after*

normalizing the maximum message delay to 1, we check the maximum number of time units we need to cut off of the front of an execution to make the suffix correct.

Remarks:

- $\mathcal{S}$ needs not be complicated. For instance, for the MIS algorithm from the lecture, it was merely a few bits: Is the node in the MIS? Is it terminated? Are we in a round for trying to join the MIS or just propagating the information which nodes did? Etc. Self-stabilization then means that even if who's in the MIS and who's not is completely messed up, the algorithm will re-construct an MIS.
- This definition is quite abstract, but can be applied to essentially any state-based description of a system. Of course, $\mathcal{C}$ should be a set of executions in which the system is considered to behave "correctly."
- The requirement here is very strong. Ending up in $\mathcal{C}$ from *any* state is equivalent to saying that *anything* can happen to the volatile memory (a.k.a. states) during a period of transient failures.
- The program code of the nodes and their hardware capabilities must not be changed. Also, the communication infrastructure must operate correctly after transient faults ceased. Otherwise there's no way of guaranteeing that the nodes behave as intended by the algorithm, and we can't hope to guarantee recovery!
- This seems obvious, but it means that one has to be careful. What if a simple bit flip tells the node to stop running the algorithm?
- Some things can't be self-stabilizing. For instance, the (single-shot) consensus algorithms we discussed generates some output based on the inputs. There is no way that we can ensure that everything works out if we make everyone believe the inputs were actually different! Let's try to capture this issue better.

9.1 Self-stabilizing Algorithms can't Terminate

Lemma 9.3. *A self-stabilizing algorithm can never terminate, unless for each node there is a single output that is always correct.*

Proof. Suppose there are two possible conflicting outputs for a node v. More precisely, there are system/input combinations for which the algorithm must output different values. Any terminating algorithm we can let run until it terminates. Then we switch the system/input combination, but set the state of some node w to the state of v from above (a transient fault may cause this). We have found an execution in which w doesn't change its state any more (as it has terminated), but has incorrect output. Hence, the execution has no correct suffix, i.e., the algorithm can't be self-stabilizing! □

Remarks:

- For instance, the only self-stabilizing coloring algorithm that terminates assigns a unique color to each node identifier – e.g., the identifier itself. Boring!
- That's also why consensus doesn't make a lot of sense here, at least not in the usual way the problem is posed.
- However, having inputs *does* make sense. One can give the inputs in registers that cannot be touched and require the algorithm to stabilize to correct outputs for these inputs. If the input registers are affected by a fault, it's also okay that the outputs change accordingly.

9.2 Dijkstra's Token Ring

One of the first self-stabilizing algorithms was Dijkstra's token ring network. A token ring is an early form of a local area network where nodes are arranged in a directed ring, communicating by a token. The system is correct if there is exactly one token in the ring, which keeps being passed around. Let's have a look at a simple solution. Given an oriented ring, we simply call the clockwise neighbor *child c,* and the counterclockwise neighbor *parent p.* Also, there is a leader node v_0. Every node v is in a state $S(v) \in \{0, 1, \ldots, n-1\}$, 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 19 Self-stabilizing token ring. A node with $S(v) \neq S(p)$ holds a token – except for v_0, which has the token if $S(v) = S(p)$. In each round, each node announces its state to its child, which then updates its own state as below.

```
1: if v = v_0 then
2:    if S(v) = S(p) then
3:       S(v) := S(v) + 1 (mod n)
4:    end if
5: else
6:    S(v) := S(p)
7: end if
```

Theorem 9.4. *In a synchronous system, Algorithm 19 is self-stabilizing with stabilization time at most* $3n$.

Proof. Suppose the leader is in a state s that nobody else has in some round. By the rules of the algorithm, the state propagates around the ring, and exactly n rounds later the leader switches to state $(s+1) \bmod n$; at this time, all other nodes are in state s. From that point on, the algorithm circulates a single token around the ring.

Now assume that for n rounds, the leader does *not* attain a state that nobody else had initially. Within n rounds, the original state of the leader propagates around the ring, reaching the leader again. Any state that is not eliminated during that time must be attained by the leader, which increases its state by

1 mod n each time it switches its state. Note that the leader cannot perform this operation n times without reaching a state that no node in the system initially had. Therefore, after n rounds the leader is in a state s such that state $s+1 \bmod n$ is not present in the system. At most n rounds later, the leader will increase its state again, implying stabilization within a total of $3n$ rounds. □

Remarks:

- This is asymptotically optimal, as two tokens could be on opposite sides of the ring. It takes at least $n/2$ rounds to distinguish this from a 1-token system.
- The algorithm also works in asynchronous systems. Can you see how to generalize the above proof?
- We defined the stabilization time as a specific value. Usually determining it precisely is very hard (or just tedious), so we tend to give upper bounds on the stabilization time. Language then deteriorates a bit and we tend to talk of an algorithm "having stabilization time S" when actually meaning that it has stabilization time at most S.
- It can be a lot of fun designing self-stabilizing algorithms, and it's not always as difficult as one might expect. However, it's essential to do baby steps. Initially, one shows seemingly very weak statements, but the resulting increased degree of organization in the system makes it much easier to follow up with stronger properties. The above proof exemplifies this; reversing the order helped, as it made clear why we wanted to show the intermediate claim that eventually the leader attains a state that wasn't present before.

9.3 Synchronous = Self-stabilizing Asynchronous!

Finding and proving correct self-stabilizing algorithms can be difficult. Let's automate the process!

We want to transform an arbitrary synchronous R-round message-passing algorithm $\mathcal{A}$ into a self-stabilizing asynchronous one. This way we can reason about things in a simplified setting (synchronous message-passing), immediately obtain a result in the asynchronous model, and self-stabilization comes for free. The hard part is to make sure that all nodes think they are in the same round of $\mathcal{A}$; instead of solving this problem, though, each node v simulates R copies $v_1, \ldots, v_R$ of itself, where v_i is in round i. Each round, v simply communicates all messages all v_i send as a bundle, and uses the received message bundles in the next round.

Theorem 9.5. *Given a deterministic synchronous message-passing algorithm $\mathcal{A}$ that runs for R rounds, we can construct an asynchronous self-stabilizing algorithm $T(\mathcal{A})$ that stabilizes in R time. If the message size of $\mathcal{A}$ is at most M_i in round i, the message size of $T(\mathcal{A})$ is $\sum_{i=1}^{R} M_i$.*

Proof. For each round i of $\mathcal{A}$, each node $v \in V$ simulates a copy v_i of itself in round i. Each copy sends messages to some neighbors w_i, $\{v, w\} \in V$, and receives the messages of them. Then it computes its state for round $i+1$, i.e., the state of v_{i+1}. The state of v_1 is determined solely by the input of v, and from the state of v_R the output of v can be determined. For this simulation, v needs to keep communicating the messages of all copies $v_1, \ldots, v_R$, yielding message size $\sum_{i=1}^{R} M_i$.

We know from Lemma 9.3 that a self-stabilizing algorithm cannot terminate. Hence, v will just keep sending the message vectors to all neighbors and updating the state of each of its copies v_i whenever receiving a message vector from a neighbor. Now why is the algorithm self-stabilizing? Well, we know that once transient faults cease, each node will correctly compute the messages of the first round and send them in the message vectors. Once a node received such vectors from all neighbors, it locally simulates round 2 correctly and sends message vectors with correct messages for rounds 1 and 2. This takes at most one time unit in terms of asynchronous time complexity. By induction, after R time units the correct results on termination are determined by all nodes. □

Remarks:

- Using this transformation, also known as *local checking*, designing self-stabilizing algorithms just turned from art to craft.
- Note that if R is a function of, e.g., n, (an upper bound on) n must be known and hardwired into the algorithm.
- The asynchronous model needs to be slightly augmented here to make sense. Transmitting "continuously" in the message passing model would mean to send an infinite number of messages. Instead, one assumes that nodes can perform a "busy-wait," for which they keep executing a loop. The next loop execution is one of the possible actions the scheduler can trigger (apart from receiving a message).
- Note that a node may send a huge number of messages during the stabilization phase. However, in principle it's ok to mitigate this problem by nodes waiting and collecting messages for some time before processing them.
- In the asynchronous shared memory model this problem doesn't exist, simply because the busy-wait is already a necessary part of the model.
- This transformation does not work for randomized algorithms. Execution and output would change all the time.
- In practice, one simply uses "pseudo-randomness." We fix sufficiently long random bit strings in advance, giving them to the nodes as part of the program memory.
- Of course the result is – technically – a deterministic algorithm, and all respective restrictions apply. One can always find a bad input/schedule combination, unless guarantees are, in fact, deterministic. However, as soon as the adversary is "oblivious," i.e., needs to make its decisions independently of the pseudo-random strings, we're good again. This usually is fine, unless there is an *actual* adversary who exploits this weakness!

- The impact on the message size is small for very fast algorithms.
- If there is a local fault or the topology or inputs change locally, this will only cause corrections in an R-hop neighborhood of the faults for an R-round algorithm.
- Ergo, this transformation rocks when applied to very fast algorithms!

9.4 Non-local Recovery

Recall that there are global problems, like MST computation. Using the above transformation would yield prohibitively large messages, even when starting from an algorithm using small messages! One way to handle this would be to keep executing a respective algorithm repeatedly, controlling it via a BFS tree. However, this would imply large stabilization times.

Can we have something simpler? If everything can be messed up, the answer is no. The lower bounds tell us that we can't be faster in the worst case. But if there are only few changes, we can exploit the convenient structure of the MST problem again.

Lemma 9.6. *If at most k edges in a (connected) weighted graph change their weight, appear, or are deleted (such that the graph is still connected), the new MST differs in at most k edges from the previous one.*

Proof. Consider a single edge e and suppose it increases its weight or gets deleted. If it's not in the MST M, nothing changes. Otherwise, it might not be in the MST any more. Denote by e' the lightest edge that does not close a cycle with $M \setminus \{e\}$. Then $(M \setminus \{e\}) \cup \{e'\}$ is the new MST: Running Kruskal on the new graph will select all edges from $E \cap M$ that are lighter than e', then e', and then (as the connectivity components at this point are identical to the run on the original graph) the remaining edges of $M \setminus \{e\}$.

Now suppose an edge e appears or decreases its weight. If it already was in the MST M or is not in the new MST, nothing changes. Otherwise, there is a unique edge $e' \in M$ heaviest in a cyle in $M \cup \{e\}$. The same reasoning as above shows that the new MST is $(M \setminus \{e'\}) \cup \{e\}$.

This shows the claim for $k = 1$. By induction on k, it follows for any k. □

This yields a very easy way of fixing a "broken" MST depending on the number of changes.

Theorem 9.7. *Assume that we have a fixed BFS tree that does not change. There is an MST algorithm that can recover from k changes in edge weights, deletions, or insertions in $\mathcal{O}(k + D)$ rounds, assuming that it had terminated before.*

Proof sketch. Run the distributed version of Kruskal on the BFS tree until termination. Now suppose k changes are made. The nodes noticing these will start sending corrections to their parents concerning their local forests (i.e., report insertions, deletions, and weight changes). As Lemma 9.6 generalizes to the min-weight maximal forests maintainted locally by the BFS tree nodes in the distributed version of Kruskal's algorithm, none of these forests change by more than k edge "swaps." Hence, each node needs to send at most k messages

to report updates to its parent. They may send some intermediate incorrect values, but the pipelining argument generalizes to this setting, too. Hence the root will learn about the new MST within $\mathcal{O}(k + D)$ rounds. □

Remarks:

- This demonstrates how important insights into the structure of problems are. This idea is very straightforward once the distributed version of Kruskal's algorithm is known, which relies on the matroid structure of the problem. This is another way of exploiting this structure!
- As pointed out, the lower bounds imply that a self-stabilizing algorithm cannot be that fast. The issue is that the entire computation needs to be repeated, as any pre-computed information cannot be trusted!
- Handling changes in the BFS tree is more involved. While adding or deleting a single edge can also only change a single BFS tree edge (just apply Lemma 9.6 with all edges having weight 1), adapting the distributed data structure given by the states of the nodes becomes a nuisance.

9.5 2-Party Systems Stabilize

We finish the chapter with another 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)?

- 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 9.8 (Dems & Reps)**.** *Eventually every citizen is rooting for the same party every other day.*[3]

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,

[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.

[3] Hence the term "swing voter."

the edge is bad if the advised friend did not follow the advisor's opinion (which means that 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 $t+1$ 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 does 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), adding weights to edges (meaning the influence between some citizens is stronger than between others), allowing loops (citizens who consider their own current opinions as well), allowing tie-breaking mechanisms, and even allowing different thresholds for party changes.

- 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" which 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.

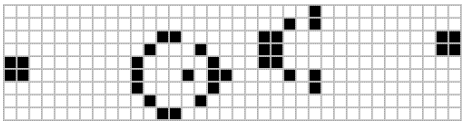

Figure 9.1: A "glider gun"...

Figure 9.2: ... in action.

Bibliographic Notes

Self-stabilization was first introduced in a paper by Edsger W. Dijkstra in 1974 [Dij74], in the context of a token ring network. It was shown that the ring stabilizes in time $\Theta(n)$. For his work Dijkstra received the 2002 ACM PODC Influential Paper Award. Shortly after receiving the award he passed away. 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. In 1991 Awerbuch et al. showed that any algorithm can be modified into a self-stabilizing algorithm that stabilizes in the same time that is needed to compute the solution from scratch [APSV91]; this construction is explained in Theorem 9.5.

For fast "distributed correction" of MSTs see the work by Peleg [Pel98]. This paper in fact deals with the more general case of matroids, and hence spotlights how the matroid structure is exploited by the distributed version of Kruskal's algorithm as well as the adaption to changing edge sets and weights.

The Republicans vs. Democrats problem was popularized by Peter Winkler, in his column "Puzzled" [Win08]. Goles et al. already proved in [GO80] that any configuration of any such system with symmetric edge weights will end up in a situation where each citizen votes for the same party every second day. The understanding of this problem has been significantly extended recently [FKW13, KPW14]. Closely related to this puzzle is the well known Game of Life which was described by the mathematician John Conway and made popular by Martin Gardner [Gar70].

This lecture is in wide parts based on material by Roger Wattenhofer. Thanks!

Bibliography

[APSV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-Stabilization By Local Checking and Correction. In *In Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):943–644, November 1974.

[FKW13] Silvio Frischknecht, Barbara Keller, and Roger Wattenhofer. Convergence in (Social) Influence Networks. In *27th International Symposium on Distributed Computing (DISC), Jerusalem, Israel*, October 2013.

[Gar70] M. Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game Life. *Scientific American*, 223:120–123, October 1970.

[GO80] E. Goles and J. Olivos. Periodic behavior of generalized threshold functions. *Discrete Mathematics*, 30:187–189, 1980.

[KPW14] Barbara Keller, David Peleg, and Roger Wattenhofer. How even Tiny Influence can have a Big Impact! In *7th International Conference on Fun with Algorithms (FUN), Lipari Island, Italy*, July 2014.

[Pel98] David Peleg. Distributed Matroid Basis Completion via Elimination Upcast and Distributed Correction of Minimum-Weight Spanning Trees. In *Proc. 25th Colloquium on Automata, Languages and Programming (ICALP)*, pages 164–175, 1998.

[Win08] P. Winkler. Puzzled. *Communications of the ACM*, 51(9):103–103, August 2008.

Lecture 10

Mutual Exclusion and Store & Collect

In the previous lectures, we've learned a lot about message passing systems. We've also seen that neither in shared memory nor message passing systems consensus can be solved deterministically. But what makes them *different?* Obviously, the key difference to message passing is the shared memory: Different processors can access the same register to store some crucial information, and anyone interested just needs to access this register. In particular, we don't suffer from locality issues, as nodes are just one shared register away. Think for instance about pointer jumping, which is not possible in a message passing system, or about MST construction, where the diameter of components matters.

Alas, great power comes with its own problems. One of them is to avoid that newly posted information is overwritten by other nodes before it's noticed.

Definition 10.1 (Mutual Exclusion)**.** *We are given a number of nodes, each executing the following code sections:*
<Entry> → <Critical Section> → <Exit> → <Remaining Code>,
where <Remaining Code> means that the node can access the critical section multiple times. A mutual exclusion *algorithm consists of code for entry and exit sections, such that the following holds*[1]

Mutual Exclusion *At all times* at most one *node is in the critical section.*

No Deadlock *If some node manages to get to the entry section, later* some *(possibly different) node will get to the critical section (in a fair execution).*

Sometimes we in addition ask for

No Lockout *If some node manages to get to the entry section, later* the same *node will get to the critical section.*

Unobstructed Exit *No node can get stuck in the exit section.*

[1] Assuming that nodes finish the <Critical Section> in finite time.

Remarks:

- We're operating in the asynchronous model today, as is standard for shared memory. The reason is that the assumption of strong memory primitives and organization of modern computing systems (multiple threads, interrupts, accesses to the hard drive, etc.) tend to result in unpredictable response times that can vary dramatically.

10.1 Strong RMW Primitives

Various shared memory systems exist. A main difference is how they allow nodes 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(R)/store-conditional(R, x): Load-link returns the current value of the specified register R. A subsequent store-conditional to the same register will store a new value x (and return **true**) only if the register's content hasn't been modified in the meantime. Otherwise, the store-conditional is guaranteed to fail (and return **false**), even if the value read by the load-link has since been restored.

An operation being atomic means that it is only a single step in the execution. For instance, no other node gets to execute the "fetch" part of the fetch-and-add primitive while another already completed it, but hasn't executed the addition yet.

Using RMW primitives one can build mutual exclusion algorithms quite easily. Algorithm 20 shows an example with the test-and-set primitive.

Algorithm 20 Mutual exclusion using test-and-set, code at node v.

Given: some shared register R, initialized to 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 10.2. *Algorithm 20 solves mutual exclusion and guarantees unobstructed exit.*

Proof. Mutual exclusion follows directly from the test-and-set definition: Initially R is 0. Let p_i be the i^{th} node to execute the test-and-set "successfully," i.e., such that the result is 0. Denote by t_i the time when this happens and by t'_i the time when p_i resets the shared register R to 0. Between t_i and t'_i no other node can successfully test-and-set, hence no other node can enter the critical section during $[t_i, t'_i]$.

Proving no deadlock works similar: One of the nodes loitering in the entry section will successfully test-and-set as soon as the node 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 ensured by this algorithm. Even with only two nodes there are asynchronous executions in which always the same node wins the test-and-set.
- Algorithm 20 can be adapted to guarantee this, essentially by ordering the nodes in the entry section in a queue.
- The power of RMW operations can be measured with the *consensus number*. The consensus number k of an RMW operation is defined as the number of nodes for which one can solve consensus with k (crashing) nodes using basic read and write registers alongside the respective RMW operations. For example, test-and-set has consensus number 2, whereas the consensus number of compare-and-swap is infinite.
- It can be shown that the power of a shared memory system is determined by the consensus number ("universality of consensus"). This insight has a remarkable theoretical and practical impact. In practice, for instance, after this was known, hardware designers stopped developing shared memory systems that support only weak RMW operations.

10.2 Mutual Exclusion using only RW Registers

Do we actually need advanced registers to solve mutual exclusion? Or to solve it efficiently? It's not as simple as before,[2] but can still be done in a fairly straightforward way.

We'll look at mutual exclusion exclusion for two nodes p_0 and p_1 only. We discuss how it can be extended to more nodes in the remarks. The general idea is that node 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 node is not interested ($W_{1-i} = 0$) access is granted. To avoid deadlocks, we add a priority variable Π enabling one node to enter the critical section even when the "want" registers are saying that none shall pass.

Theorem 10.3. *Algorithm 21 solves mutual exclusion and guarantees both no lockout and unobstructed exit.*

[2] Who would have guessed, we're talking about a non-trivial problem here.

Algorithm 21 Mutual exclusion: Peterson's algorithm.

Given: shared registers W_0, W_1, Π, all initialized to 0.
Code for node p_i, $i \in \{0, 1\}$:
<**Entry**>
1: $W_i := 1$
2: $\Pi := 1 - i$
3: **repeat** *nothing* **until** $\Pi = i$ or $W_{1-i} = 0$ // "busy-wait"
<**Critical Section**>
4: ...
<**Exit**>
5: $W_i := 0$
<**Remainder Code**>
6: ...

Proof. The shared variable Π makes sure that one of the nodes can enter the critical section. Suppose p_0 enters the critical section first. If at this point it holds that $W_1 = 0$, p_1 has not yet executed Line 1 and therefore will execute Line 2 before trying to enter the critical section, which means that Π will be 0 and p_1 has to wait until p_0 leaves the critical section and resets $W_0 := 0$. On the other hand, if $W_1 = 1$ when p_0 enters the critical section, we already must have that $\Pi = 0$ at this time, i.e., the same reasoning applies. Arguing analogously for p_1 entering the critical section first, we see that mutual exclusion is solved.

To see that there are no lockouts, observe that once, e.g., p_0 is executing the spin-lock (i.e., is "stuck" in Line 3), the priority variable is not going to be set to 1 again until it succeeds in entering and passing the critical section. If p_1 is also interested in entering and "wins" (we already know that one of them will), afterwards it either will stop trying to enter or again set Π to 0. In any event, p_0 enters the section next.

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

Remarks:

- Line 3 in Algorithm 21 is a *spinlock* or *busy-wait,* like Lines 1-3 in Algorithm 20. Here we have the extreme case that the node doesn't even try to do anything, it simply needs to wait for someone else to finish the job.

- Extending Peterson's Algorithm to more than 2 nodes can be done by a tournament tree, like in tennis. With n nodes every node needs to win $\lceil \log n \rceil$ matches before it can enter the critical section. More precisely, each node 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.

- This solution inherits the additional nice properties: no lockouts, unobstructed exit.

- On the downside, more work is done than with the test-and-set operation, as the binary tree has depth $\lceil \log n \rceil$. One captures this by counting

asynchronous rounds or the number of actual changes of variables,[3] as only signal *transitions* are "expensive" (i.e., costly in terms of energy) in circuits.

10.3 Store & Collect

Informally, the STORE & COLLECT problem can be stated as follows. There are n nodes $p_1, \ldots, p_n$. Every node 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 nodes. Further, there is an operation by which a node can *collect* (i.e., read) the values of all the nodes 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 10.4 (Store and Collect)**.** *There are two operations: A* STORE(val) *by node p_i sets val to be the latest value of its register R_i. A* COLLECT *operation returns a* view*, i.e., a function $f\colon V \to VAL \cup \{\perp\}$ from the set of nodes V to a set of values VAL or the symbol $\perp$, which means "nothing written yet." Here, $f(p_i)$ is intended to be the latest value stored by p_i, for each node p_i. For a* COLLECT *operation* cop*, the following validity properties must hold for every node p_i:*

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

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

Attention: A COLLECT operation is not atomic, i.e., consists of multiple (atomic) operations! This means that there can be reads that neither precede nor follow a COLLECT. Such overlapping operations are considered *concurrent*. In general, also a write operation can be more involved, to simplify reads or achieve other properties, so the same may apply to them.

We assume that the read/write register R_i of every node 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 shown in Algorithm 22.

[3] There may be an unbounded number of read operations due to the busy-wait, and it is trivial to see that this cannot be avoided in a (completely) asynchronous system.

Algorithm 22 Trivial COLLECT.

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

Remarks:

- Obviously,[4] Algorithm 22 works. The step complexity of every STORE operation is 1, the step complexity of a COLLECT operation is n.
- The step complexities of Algorithm 22 is optimal: There 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 is unnecessarily large. Assume that there are only two nodes p_i and p_j that have stored a value in their registers R_i and R_j. Then, in principle, COLLECT needs to read the registers R_i and R_j only.

10.3.1 Splitters

Assume that up to a certain time t, $k \leq n$ nodes have started at least one operation. We call an operation completing at time t *adaptive* to contention, if its step complexity depends on k only.

To obtain adaptive COLLECT algorithms, we will use a symmetry breaking primitive called a *splitter*.

Definition 10.5 (Splitter)**.** *A splitter is a synchronization primitive with the following characteristics. A node entering a splitter exits with either **stop**, **left**, or **right**. If k nodes enter a splitter, at most one node exits with **stop** and at most $k-1$ nodes exit with **left** and **right**, respectively.*

This definition guarantees that if a single node enters the splitter, then it obtains **stop**, and if two or more nodes enter the splitter, then there is at most one node obtaining **stop** and there are two nodes that obtain different values

[4] Be extra careful whenever such a word pops up. If it's not indeed immediately obvious, it may translate to "I believe it works, but didn't have the patience to check the details," which is an excellent source of (occasionally serious) blunders. One of my lecturers once said: "If it's trivial, then why don't we write it down? It should not take more than a line. If it doesn't, then it's not trivial!"

Algorithm 23 Splitter Code

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

Splitter access by node p_i:

```
1:  X := i;
2:  if Y then
3:    return right
4:  else
5:    Y := true
6:    if X = i then
7:      return stop
8:    else
9:      return left
10:   end if
11: end if
```

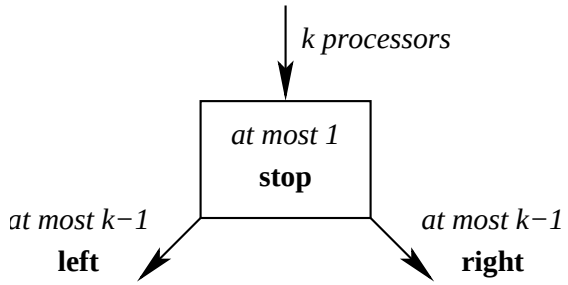

Figure 10.1: A Splitter

(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 10.1. Algorithm 23 implements a splitter.

Lemma 10.6. *Algorithm 23 implements a splitter.*

Proof. Assume that k nodes enter the splitter. Because the first node that checks whether $Y =$ **true** in line 2 will find that $Y =$ **false**, not all nodes return **right**. Next, assume that i is the last node 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 node that does not return **left**.

It remains to show that at most 1 node returns **stop**. Suppose p_i decides to do this at time t, i.e., p_i reads that $X = i$ in Line 6 at time t. Then any p_j that sets $X := j$ after time t will (re)turn **right**, as already $Y =$ **true**. As any other node p_j will not read $X = j$ after time t (there is no other way to change X to j), this shows that at most one node will return **stop**. Finally, observe that if $k = 1$, then the result for the single entering node will be **stop**. □

10.3.2 Binary Splitter Tree

Assume that we are given $2^n - 1$ splitters and that for every splitter S, there is an additional shared variable $Z_S \colon \{\perp\} \cup \{1, \ldots, n\}$ that is initialized to $\perp$ and an additional shared variable $M_S \colon$ **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 24.

Theorem 10.7. *Algorithm 24 implements* STORE *and* COLLECT*. Let k be the number of participating nodes. The step complexity of the first* STORE *of a node p_i is $\mathcal{O}(k)$, the step complexity of every additional* STORE *of p_i is $\mathcal{O}(1)$, and the step complexity of* COLLECT *is $\mathcal{O}(k)$.*

Proof. Because at most one node can stop at a splitter, it is sufficient to show that every node 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 $k - i$ nodes enter a subtree at depth i (i.e., a subtree where the root has distance i to the root of the whole tree). This follows by induction from the definition of splitters, as not all nodes entering a splitter can proceed to the same subtree

Algorithm 24 Adaptive collect: binary tree algorithm

Operation STORE(val) (by node p_i) **:**

```
1:  R_i := val
2:  if first STORE operation by p_i then
3:    v := root node of binary tree
4:    α := result of entering splitter S(v);
5:    M_S(v) := true
6:    while α ≠ stop do
7:      if α = left then
8:        v := left child of v
9:      else
10:       v := right child of v
11:     end if
12:     α := 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 ≠ ⊥ then
19:     i := Z_S; f(p_i) := R_i                    // read value of node p_i
20:   end if
21: end for                                        // f(p_i) = ⊥ for all other nodes
```

rooted at a child of the splitter. Hence, at the latest when reaching depth $k-1$, a node is the only node entering a splitter and thus obtains **stop**.

Note that the step complexity of executing a splitter is $\mathcal{O}(1)$. The bound of $k-1$ on the depth of the accessed subtree of the binary splitter tree thus shows that the step complexity of the initial STORE is $\mathcal{O}(k)$ for each node, and each subsequent STORE requires only $\mathcal{O}(1)$ steps.

To show that the step complexity of COLLECT is $\mathcal{O}(k)$, 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 when $k \in \mathbb{N}_0$ nodes access the root. We claim that $x_k \leq \max\{2k-1, 0\}$, which is trivial for $k = 0$. Now assume the inequality holds for $0, \ldots, k-1$. Splitters guarantee that neither all nodes turn **left** nor all nodes turn **right**, i.e., $k_l \leq k-1$ nodes 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

$$x_k \leq x_{k_l} + x_{k_r} + 1 \leq \max\{2k_l - 1, 0\} + \max\{2k_r - 1, 0\} + 1 \leq 2k - 1,$$

concluding induction and proof. □

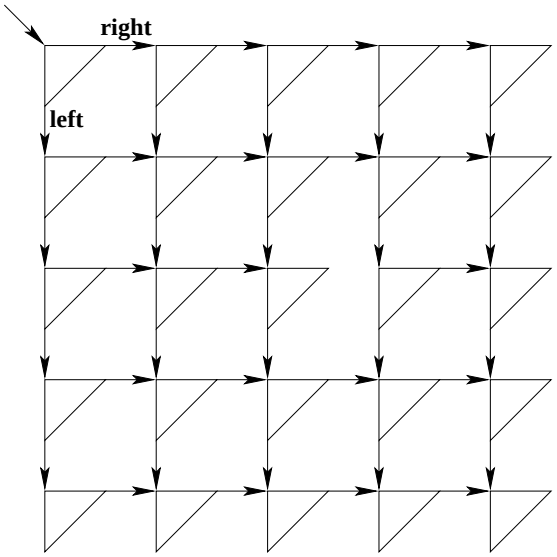

Figure 10.2: 5×5 Splitter Matrix

Remarks:

- The step complexities of Algorithm 24 are very good. Clearly, the step complexity of the COLLECT operation is asymptotically optimal.[5] 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 24 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.

10.3.3 Splitter Matrix

In order to obtain quadratic memory consumption (instead of the exponential memory consumption of the splitter tree), we arrange n^2 splitters in an $n \times n$ matrix as shown in Figure 10.2. The algorithm is analogous to Algorithm 24. The matrix is entered at the top left. If a node receives **left**, it next visits the splitter in the next row of the same column. If a node receives **right**, it next visits the splitter in the next column of the same row. Clearly, the space complexity of this algorithm is $\mathcal{O}(n^2)$. The following theorem gives bounds on the step complexities of STORE and COLLECT.

Theorem 10.8. *Let k be the number of participating nodes. The step complexity of the first* STORE *of a node p_i is $\mathcal{O}(k)$, the step complexity of every additional* STORE *of p_i is $\mathcal{O}(1)$, and the step complexity of* COLLECT *is $\mathcal{O}(k^2)$.*

Proof. Let the top row be row 0 and the left-most column be column 0. Let x_i be the number of nodes entering a splitter in row i. By induction on i, we show

[5] Here's another clearly to watch carefully. While the statement is correct, it's not obvious that we chose the performance measure wisely. We could refine our notion again and ask for the step complexity in terms of the number of writes that did not precede the most recent COLLECT operation of the collecting process. But let's not go there today.

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 node visits the splitter in row $i-1$ and column j. By the properties of splitters, not all nodes entering the splitter in row $i-1$ and column j obtain **left**. Therefore, not all nodes entering a splitter in row $i-1$ move on to row i. Because at least one node stays in every row, we get that $x_i \leq k - i$. Similarly, the number of nodes entering column j is at most $k-j$. Hence, every node 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 $\mathcal{O}(k^2)$. Because the longest path in the splitter matrix is $2k$, the step complexity of STORE is $\mathcal{O}(k)$. □

Remarks:

- With a slightly more complicated argument, it is possible to show that the number of nodes 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.

- Recently, it has been shown that with a considerably more complicated deterministic algorithm, it is possible to achieve $\mathcal{O}(k)$ step complexity and $\mathcal{O}(n^2)$ space complexity.

What to take Home

- Obviously, more powerful RMW primitives are extremely useful. However, their implementation might be more costly than an implementation using read/write registers only. At the end of the day, RMW primitives solve mutual exclusion at some level of the system hierarchy.

- Naturally, atomic read/write registers do not fall out of the sky either. They are implemented from non-atomic registers using similar techniques.

Bibliographic Notes

Already in 1965 Edsger Dijkstra gave a deadlock-free solution for mutual exclusion [Dij65]. Later, Maurice Herlihy suggested consensus numbers [Her91], where he proved the "universality of consensus," i.e., the power of a shared memory system is determined by the consensus number. Peterson's Algorithm is due to [PF77, Pet81], and adaptive COLLECT was studied in the sequence of papers [MA95, AFG02, AL05, AKP+06].

Again, a big thanks goes to Roger Wattenhofer, whose lecture material today's topic is based on!

Bibliography

[AFG02] Hagit Attiya, Arie Fouren, and Eli Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

[AKP+06] Hagit Attiya, Fabian Kuhn, C. Greg Plaxton, Mirjam Wattenhofer, and Roger Wattenhofer. Efficient adaptive collect using randomization. *Distributed Computing*, 18(3):179–188, 2006.

[AL05] Yehuda Afek and Yaron De Levie. Space and Step Complexity Efficient Adaptive Collect. In *DISC*, pages 384–398, 2005.

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[Her91] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[MA95] Mark Moir and James H. Anderson. Wait-Free Algorithms for Fast, Long-Lived Renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995.

[Pet81] J.L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.

[PF77] G.L. Peterson and M.J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 91–97. ACM, 1977.

Lecture 11

Shared Counters

Maybe the most basic operation a computer performs is adding one, i.e., to count. In distributed systems, this can become a non-trivial task. If the events to be counted occur, e.g., at different processors in a multi-core system, determining the total count by querying each processor for its local count is costly. Hence, in shared memory systems, one may want to maintain a *shared counter* that permits to determine the count using a single or a few read operations.

11.1 A Simple Shared Counter

If we seek to implement such an object, we need to avoid that increments are "overwritten," i.e., two nodes increment the counter, but only one increment is registered. So, the simple approach of using one register and having a node incrementing the counter read the register and write the result plus one to the register is not good enough with atomic read/write registers only. With more powerful registers, things look differently.

Algorithm 25 Shared counter using compare-and-swap, code at node v.

Given: some shared register R, initialized to 0.
Increment:
1: **repeat**
2: $r := R$
3: success := compare-and-swap$(R, r, r + 1)$
4: **until** success = **true**
Read:
5: **return** R

11.1.1 Progress Conditions

Basically, this approach ensures that the read-write sequence for incrementing the counter behaves as if we applied mutual exclusion. However, there is a crucial difference. Unlike in mutual exclusion, no node obtains a "lock" and needs to release it before other nodes can modify the counter again. The algorithm is *lock-free*, meaning that it makes progress regardless of the schedule.

Definition 11.1 (Lock-Freedom)**.** *An operation is* lock-free*, if whenever any node is executing an operation, some node executing the same operation is guaranteed to complete it (in a bounded number of steps of that node). In asynchronous systems, this must hold even in (infinite) schedules that are not fair, i.e., if some of the nodes executing the operation may be stalled indefinitely.*

Lemma 11.2. *The increment operation of Algorithm 25 is lock-free.*

Proof. Suppose some node executes the increment code. It obtains some value r from reading the register R. When executing the compare-and-swap, it either increments the counter successfully or the register already contains a different value. In the latter case, some other node must have incremented the counter successfully. □

This condition is strong in the sense that the counter will not cease to operate because some nodes crash or are stalled for a long time. Yet, it is pretty weak with respect to read operations: It would admit that a node that just wants to read never completes this operation. However, as the read operations of this algorithm are trivial, they satisfy the strongest possible progress condition.

Definition 11.3 (Wait-Freedom)**.** *An operation is* wait-free *if whenever a node executes an operation, it completes if it is granted a bounded number of steps by the execution. In asynchronous systems, this must hold even in (infinite) schedules that are not fair, i.e., if nodes may be suspended indefinitely.*

Remarks:

- Wait-freedom is extremely useful in systems where one cannot guarantee reasonably small response times of other nodes. This is important in multi-core systems, in particular if the system needs to respond to external events with small delay.
- Consequently, wait-freedom is the gold standard in terms of progress. Of course, one cannot always afford gold.
- From the FLP theorem, we know that wait-free consensus is not possible without advanced RMW primitives.

11.1.2 Consistency Conditions

Progress is only a good thing if it goes in the right direction, so we need to figure out the direction we deem right. Even for such a simple thing as a counter, this is not as trivial as it might appear at first glance. If we require that the counter always returns the "true" value when read, i.e., the sum of the local event counts of all nodes, we cannot hope to implement this distributedly in any meaningful fashion: whatever is read at a single location may already be outdated, so we cannot satisfy the "traditional" sequential specification of a counter. Before we proceed to relaxing it, let us first formalize it.

Definition 11.4 (Sequential Object)**.** *A* sequential object *is given by a tuple* (S, s_0, R, O, t)*, where*

- S *is the set of states the object can attain,*

- s_0 *is its initial state,*
- R *is the set of values that can be read from the object,*
- O *is the set of operations that can be performed on the object, and*
- $t\colon O \times S \to S \times R$ *is the transition function of the object.*

A sequential execution *of the object is a sequence of operations* $o_i \in O$ *and states* $s_i \in S$*, where* $i \in \mathbb{N}$ *and* $(s_i, r_i) = t(o_i, s_{i-1})$*; operation* o_i returns *value* $r_i \in R$*.*

Definition 11.5 (Sequential Counter)**.** *A* counter *is the object given by* $S = \mathbb{N}_0$*,* $s_0 = 0$*,* $R = \mathbb{N}_0 \cup \{\perp\}$*,* $O = \{\mathit{read}, \mathit{increment}\}$*, and, for all* $i \in \mathbb{N}_0$*,* $t(\mathit{read}, i) = (i, i)$ *and* $t(\mathit{increment}, i) = (i + 1, \perp)$*.*

We could now "manually" define a distributed variant of a counter that we can implement. Typically, it is better to apply a generic *consistency condition*. In order to do this, we first need "something distributed" we can relate the sequential object to.

Definition 11.6 (Implementation)**.** *A (distributed)* implementation *of a sequential object is an algorithm*[1] *that enables each node to access the object using the operations from* O*. A node completing an operation obtains a return value from the set of possible return values for that operation.*

So far, this does not say anything about whether the returned values make any sense in terms of the behavior of the sequential object; this is addressed by the following definitions.

Definition 11.7 (Precedence)**.** *Operation* o precedes *operation* o' *if* o *completes before* o' *begins.*

Definition 11.8 (Linearizability)**.** *An execution of an implementation of an object is* linearizable*, if there is a sequential execution of the object such that*

- *there is a one-to-one correspondence between the performed operations,*
- *if* o *precedes* o' *in execution of the implementation, the same is true for their counterparts in the sequential execution, and*
- *the return values of corresponding operations are identical.*

An implementation of an object is linearizable if all its executions are linearizable.

Theorem 11.9. *Algorithm 25 is a linearizable counter implementation. Its read operations are wait-free and its increment operations are lock-free.*

Proof. All claims but linearizability are easily verified from the definitions and the algorithm. For linearizability, note that read operations are atomic, so we only need to worry about when we let a write operation take place in the linearization. This is easy, too: we choose the point in time when the successful compare-and-swap actually incrementing the value stored by R occurs. □

[1] Or rather a suite of subroutines that can be called, one for each possible operation.

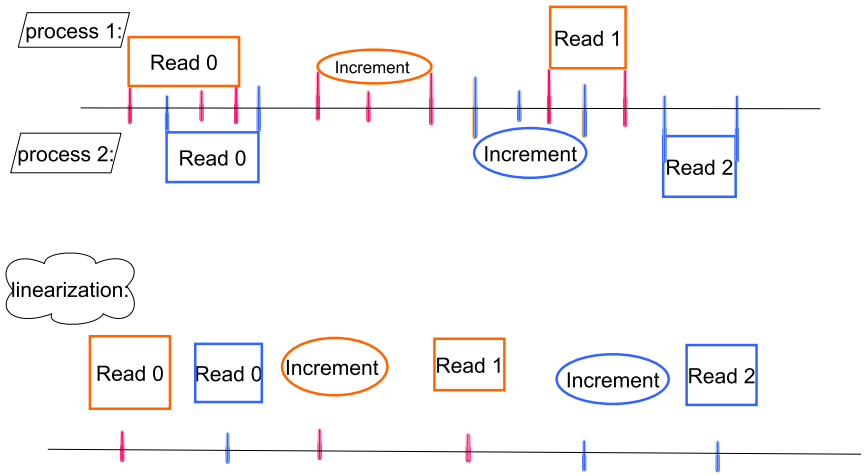

Figure 11.1: Top: An execution of a distributed counter implementation. Each mark is one atomic step of the respective node. Bottom: A valid linearization of the execution. Note that if the second read of node 1 would have returned 2, it would be ordered behind the increment by node 2. If it had returned 0, the execution would not be linearizable.

Remarks:

- Linearizability is extremely useful. It means that we can treat a (possibly horribly complicated) distributed implementation of an object as if it was accessed atomically.
- This makes linearizability the gold standard in consistency conditions. Unfortunately, also this gold has its price.
- Put simply, linearizability means "simulating sequential behavior," but not just any behavior – if some operation completed in the past, it should not have any late side effects.
- There are many equivalent ways of defining linearizability:
  - Extend the partial "precedes" order to a total order such that the resulting list of operation/return value pairs is a (correct) sequential execution of the object.
  - Assign strictly increasing times to the (atomic) steps of the execution of the implementation. Now each operation is associated with a time interval spanned by its first and last step. Assign to each operation a *linearization point* from its interval (such that no two linearization points are identical). This induces a total order on the operations. If this can be done in a way consistent with the specification of the object, the execution is linearizable.
- One can enforce linearizability using mutual exclusion.

- In the store & collect problem, we required that the "precedes" relation is respected. However, our algorithms/implementations were *not* linearizable. Can you see why?
- Coming up with a linearizable, wait-free, and efficient implementation of an object can be seen as creating a more powerful shared register out of existing ones.
- Shared registers are linearizable implementations of conventional registers.
- There are many weaker consistency conditions. For example one may just ask that the implementation behaves like its sequential counterpart only during times when a single node is accessing it.

11.2 No Cheap Wait-Free Linearizable Counters

There's a straightforward wait-free, linearizable shared counter using atomic read/write registers only: for each node, there's a shared register to which it applies increments locally; a read operation consists of reading all n registers and summing up the result.

This clearly is wait-free. To see that it is linearizable, observe that local increments require only a single write operation (as the node knows its local count), making the choice of the linearization point of the operation obvious. For each read, there must be a point in time between when it started and when it completes at which the sum of all registers equals the result of the read; this is a valid linearization point for the read operation.

Here's the problem: this seems very inefficient. It requires $n-1$ accesses to shared registers just to *read* the counter, and it also requires n registers. We start with the bad news. Even with the following substantially weaker progress condition, this is optimal.

Definition 11.10 (Solo-Termination)**.** *An operation is* solo-terminating *if it completes in finitely many steps provided that only the calling node takes steps (regardless of what happened before).*

Note that wait-freedom implies lock-freedom and that lock-freedom implies solo-termination.

Theorem 11.11. *Any linearizable deterministic implementation of a counter that guarantees solo-termination of all operations and uses only atomic read/write shared registers requires at least $n-1$ registers and has step complexity at least $n-1$ for read operations.*

Proof. We construct a sequence of executions $\mathcal{E}_i = \mathcal{I}_i\mathcal{W}_i\mathcal{R}_i$, $i \in \{0, \ldots, n-1\}$, where $\mathcal{E}_i$ is the concatenation of $\mathcal{I}_i$, $\mathcal{W}_i$, and $\mathcal{R}_i$. In each execution, the nodes are $\{1, \ldots, n\}$, and execution $\mathcal{E}_i$ is going to require i distinct registers; node n is the one reading the counter.

1. In $\mathcal{I}_i$, nodes $j \in \{1, \ldots, i\}$ increment the counter (some of these increments may be incomplete).

2. In $\mathcal{W}_i$, nodes $j \in \{1, \ldots, i\}$ each write to a different register R_j once and no other steps are taken.

3. In $\mathcal{R}_i$, node n reads the registers $R_1, \ldots, R_i$ as part of a (single) read operation on the counter.

As in $\mathcal{E}_{n-1}$ node n accesses $n-1$ different registers, this shows the claim.

The general idea is to "freeze" nodes $j \in \{1, \ldots, i\}$ just before they write to their registers R_j. This forces node $i+1$ to write to another register if it wants to complete many increments (which wait-freedom enforces) in a way that is not

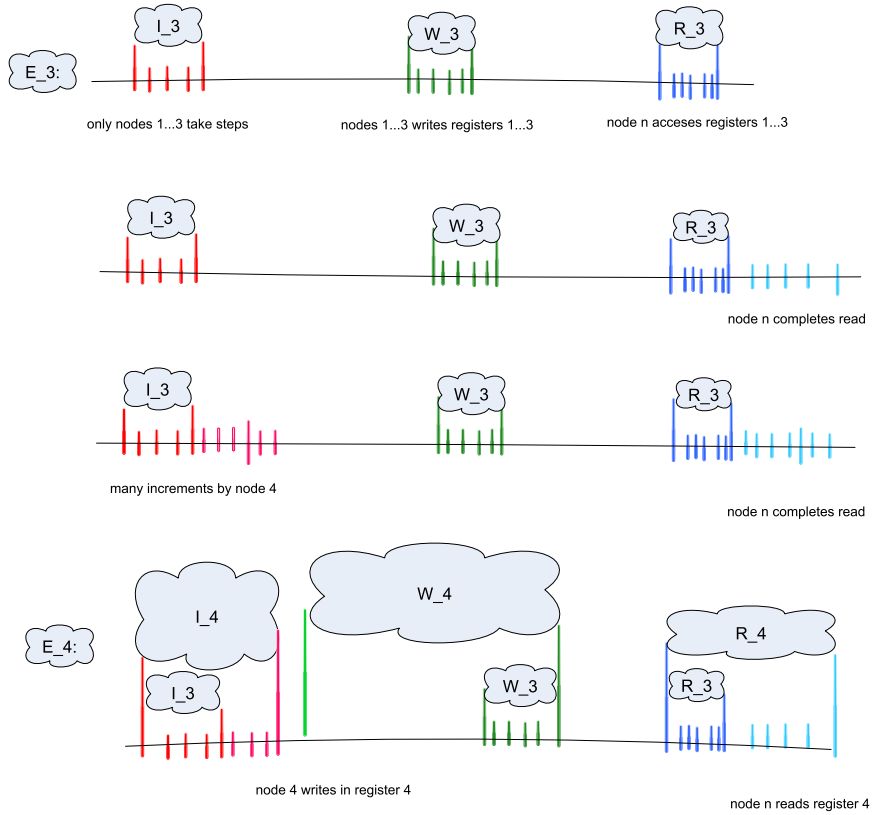

Figure 11.2: Example for the induction step from 3 to 4. Top: Execution $\mathcal{E}_3$. Second: We extend $\mathcal{E}_3$ by letting node n complete its read operation. Third: We consider the execution where we insert many increment operations by some unused node between $\mathcal{I}_3$ and $\mathcal{W}_3$. This might change how node n completes its read operation. However, if node n would not read a register not overwritten by $\mathcal{W}_3$ to which the new node writes, the two new executions would be indistinguishable to node n and its read would return a wrong (i.e., not linearizable) value in at least one of the them. Bottom: We let the new node execute until it writes the new register first and node n perform its read until it accesses the register first, yielding $\mathcal{E}_4$.

overwritten when we let nodes $1, \ldots, i$ perform their stalled write steps. This is necessary for node n to be able to complete a read operation without waiting for nodes $1, \ldots, i$; otherwise n wouldn't be able to distinguish between $\mathcal{E}_i$ and $\mathcal{E}_{i+1}$, which require different outputs if $i+1$ completed more increments than have been started in $\mathcal{E}_i$.

The induction is trivially anchored at $i = 0$ by defining $\mathcal{E}_0$ as the empty execution. Now suppose we are given $\mathcal{E}_i$ for $i < n-1$. We claim that node n must access some new register R_{i+1} before completing a read operation (its single task in all executions we construct). Assuming otherwise, consider the following execution. We execute $\mathcal{E}_i$ and then let node n complete its read operation. As the implementation is solo-terminating, this must happen in finitely many steps of n, and, by linearizability, the read operation must return at most the number k of increments that have been started in $\mathcal{E}_i$; otherwise, we reach a contradiction by letting these operations complete (one by one, using solo-termination) and observing that there is no valid linearization.

On the other hand, consider the execution in which we run $\mathcal{I}_i$, then let some node $j \in \{i+1, \ldots, n-1\}$ complete $k+1$ increments running alone (again possible by solo-termination), append $\mathcal{W}_i$, and let node n complete its read operation. Observe that the state of all registers $R_1, \ldots, R_i$ before node n takes any steps is the same as after $\mathcal{I}_i\mathcal{W}_i$, as any possible changes by node j were overwritten. Consequently, as n does not access any other registers, it cannot distinguish this execution from the previous run and thus must return a value of at most k. However, this contradicts linearizability of the new execution, in which already $k+1$ increments are complete. We conclude that when extending $\mathcal{E}_i$ by letting node n run alone, n will eventually access some new register R_{i+1}.

Define $\mathcal{R}_{i+1}$ as the sequence of steps n takes in this setting up to and including the first access to register R_{i+1}. W.l.o.g., assume that there exists an extension of $\mathcal{I}_i$ in which only nodes $i+1, \ldots, n-1$ take steps and eventually some $j \in \{i+1, \ldots, n-1\}$ writes to R_{i+1}. Otherwise, R_{i+1} is never going to be written (by a node different from n) in any of the executions we construct, i.e., node n cannot distinguish any of the executions we construct by reading R_{i+1}; hence it must read another register by repetition of the above argument. Eventually, there must be a register it reads that is written by some node $i+1 \leq j \leq n-1$ (if we extend $\mathcal{I}_i$ such that only node j takes steps), and we can apply the reasoning that follows.

W.l.o.g., assume that $j = i+1$ (otherwise we just switch the indices of nodes j and $i+1$ for the purpose of this proof) and denote by $\mathcal{I}_{i+1}w_{i+1}$ such an extension of $\mathcal{I}_i$, where w_{i+1} is the write of $j = i+1$ to R_{i+1}. Setting $W_{i+1} := w_{i+1}W_i$ and $\mathcal{E}_{i+1} := \mathcal{I}_{i+1}\mathcal{W}_{i+1}\mathcal{R}_{i+1}$ completes the induction and therefore the proof. □

Remarks:

- There was some slight cheating, as the above reasoning applies only to unbounded counters, which we can't have in practice anyway. Arguing more carefully, one can bound the number of increment operations required in the construction by $2^{\mathcal{O}(n)}$.
- The technique is far more general:
  - It works for many other problems, such as modulo counters, fetch-and-add, or compare-and-swap. In other words, using powerful RMW

registers just shifts the problem.

  - This can also seen by using reductions. Algorithm 25 shows that compare-and-swap cannot be easy to implement, and load-link/store-conditional can be used in the very same way. A fetch-and-add register is even better: it trivially implements a wait-free linearizable counter.
  - The technique works if one uses *historyless* objects in the implementation, not just RW registers. An object is historyless, if the resulting state of any operation that is not just a read (i.e., does never affect the state) does not depend on the current state of the object.
  - For instance, test-and-set registers are historyless, or even registers that can hold arbitrary values and return their previous state upon being written.
  - It also works for *resettable consensus* objects. These support the operations propose(i), $i \in \mathbb{N}$, reset, and read, and are initiated in state $\perp$. A propose(i) operation will result in state i if the object is in state $\perp$ and otherwise not affect the state. The reset operation brings the state back to $\perp$. This means that the hardness of the problem is not originating in an inability to solve consensus!
  - The space bound also applies to randomized implementations. Basically, the same construction shows that there is a positive probability that node n accesses $n-1$ registers, so these registers must exist. However, one can hope to achieve a small step complexity (in expectation or w.h.p.), as the probability that such an execution occurs may be very small.

- By now you might already expect that we're going to "beat" the lower bound. However, we're not going to use randomization, but rather exploit another loophole: the lower bound crucically relies on the fact that the counter values can become very large.

11.3 Efficient Linearizable Counter from RW Registers

Before we can construct a linearizable counter, we first need to better understand linearizability.

11.3.1 Linearizability "=" Atomicity

As mentioned earlier, a key feature of linearizability is that we can pretend that linearizable objects are atomic. In fact, this is the reason why it is standard procedure to assume that atomic shared registers are available: one simply uses a linearizable implementation from simpler registers. Let's make this more clear.

Definition 11.12 (Base objects)**.** *The* base objects *of an implementation of an object* **O** *are all the registers and (implementations of) objects that nodes may access when executing any operations of* **O**.

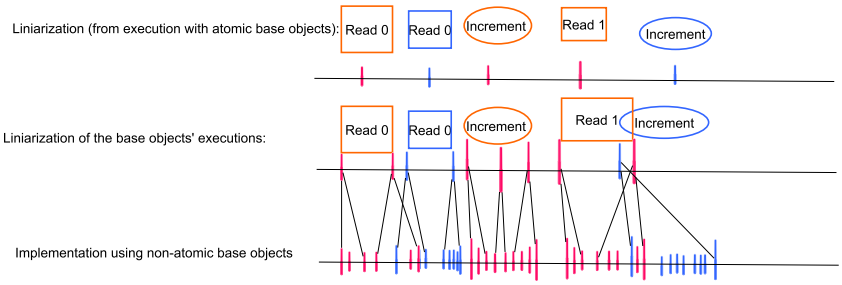

Figure 11.3: Bottom: An execution of an implementation using linearizable base objects. Center: Exploiting linearizability of each base object, we obtain an execution of a corresponding implementation from atomic base objects. Top: By linearizability of the assumed implementation from atomic base objects, this execution can be linearized, yielding a linearization of the original execution at the bottom.

Lemma 11.13. *Suppose some object* $\mathbf{O}$ *has a linearizable implementation using atomic base objects. Then replacing any atomic base object by a linearizable implementation (where each atomic access is replaced by calling the respective operation and waiting for it to complete) results in another linearizable implementation of* $\mathbf{O}$*.*

Proof. Consider an execution $\mathcal{E}$ of the constructed implementation of $\mathbf{O}$ from linearizable implementations of its base objects. By definition of linearizability, we can map the (sub)executions comprised of the accesses to (base objects of) the implementations of base objects to sequential executions of the base objects that preserve the partial order given by the "precedes" relation.

We claim that doing this for all of the implementations of base objects of $\mathbf{O}$ yields a valid execution $\mathcal{E}'$ of the given implementation of $\mathbf{O}$ from atomic base objects. To see this, observe that the view of a node in (a prefix of) $\mathcal{E}$ is given by its initial state and the sequence of return values from its previous calls to the atomic base objects. In $\mathcal{E}$, the node calls an operation once all its preceding calls to operations are complete. As, by definition of linearizability, the respective return values are identical, the claim holds true.

The rest is simple. We apply linearizability to $\mathcal{E}'$, yielding a sequential execution $\mathcal{E}''$ of $\mathbf{O}$ that preserves the "precedes" relation on $\mathcal{E}'$. Now, if operation o precedes o' in $\mathcal{E}$, the same holds for their counterparts in $\mathcal{E}'$, and consequently for *their* counterparts in $\mathcal{E}''$; likewise, the return values of corresponding operations match. Hence $\mathcal{E}''$ is a valid linearization of $\mathcal{E}$. □

Remarks:

- Beware side effects, as they break this reasoning! If a call to an operation affects the state of the node (or anything else) beyond the return value, this can mess things up.
- For instance, one can easily extend this reasoning to randomized implementations. However, in practical systems, randomness is usually not

"true" randomness, and the resulting dependencies can be... interesting.

- Lemma 11.13 permits to abstract away the implementation details of more involved objects, so we can reason hierarchically. This will make our live much, *much* easier!

- This result is the reason why it is common lingo to use the terms "atomic" and "linearizable" interchangably.

- We're going to exploit this to the extreme now. Recursion time!

11.3.2 Counters from Max Registers

We will construct our shared counter using another, simpler object.

Definition 11.14 (Max Register)**.** *A* max register *is the object given by* $S = \mathbb{N}_0$, $s_0 = 0$, $R = \mathbb{N}_0 \cup \{\bot\}$, $O = \{read, write(i) \mid i \in \mathbb{N}_0\}$, *and, for all* $i, j \in \mathbb{N}_0$, $t(read, i) = (i, i)$ *and* $t(write(i), j) = (\max\{i, j\}, \bot)$. *In words, the register always returns the maximum previously written value on a read.*

Max registers are not going to help us, as the lower bound applies when constructing them. We need a twist, and that's requiring a bound on the maximum value the counter – and thus the max registers – can attain.

Definition 11.15 (Bounded Max Register)**.** *A* max register with maximum value $M \in \mathbb{N}$ *is the object given by* $S = \{0, \ldots, M\}$, $s_0 = 0$, $R = S \cup \{\bot\}$, $O = \{read, write(i) \mid i \in S\}$, *and, for all* $i, j \in S$, $t(read, i) = (i, i)$ *and* $t(write(i), j) = (\max\{i, j\}, \bot)$.

Definition 11.16 (Bounded Counter)**.** *A* counter with maximum value $M \in \mathbb{N}$ *is the object given by* $S = \{0, \ldots, M\}$, $s_0 = 0$, $R = S \cup \{\bot\}$, $O = \{read, increment\}$, *and, for all* $i \in S$, $t(read, i) = (i, i)$ *and* $t(increment, i) = (\min\{i + 1, M\}, \bot)$.

Before discussing how to implement bounded max registers, let's see how we obtain an efficient wait-free linearizable bounded counter from them.

Lemma 11.17. *Suppose we are given two atomic counters of maximum value M that support k incrementing nodes (i.e., no more than k different nodes have the ability to use the increment operation) and an atomic max register of maximum value M. Then we can implement a counter with maximum value M and the following properties.*

- *It supports $2k$ incrementing nodes.*

- *It is linearizable.*

- *All operations are wait-free.*

- *The step complexity of reads is 1, a read of a max register.*

- *The step complexity of increments is 4, where only one of the steps is a counter increment.*

Proof. Denote the counters by C_1 and C_2 and assign k nodes to each of them. Denote by R the max register. To read the new counter C, one simply reads R. To increment C, a node increments its assigned counter, reads both counters, and writes the sum to R. Obviously, we now support $2k$ incrementing nodes, all operations are wait-free, and their step complexity is as claimed. Hence, it remains to show that the counter is linearizable.

Fix an execution of this implementation of C. We need to construct a corresponding execution of a counter of maximum value M. At each point in time, we rule that the state of C is the state of R.[2] Thus, we can map the sequence of read operations to the same sequence of read operations; all that remains is to handle increments consistently. Suppose a node applies an increment. Denote by σ the sum of the two counter values right after it incremented its assigned counter. At this point, $r < \sigma$, where r denotes the value stored in R, as no node ever writes a value larger than the sum it read from the two counters to R. As the node reads C_1 and C_2 after incrementing its assigned counter, it will read a sum of at least σ and subsequently write it to R. We conclude that at some point during the increment operation the node performs on C, R will attain a value of at least σ, while before it was smaller than σ. We map the increment of the node to this step.

To complete the proof, we need to check that the result is a valid linearization. For each operation o, we have chosen a linearization point $l(o)$ during the part of the execution in which the operation is performed. Thus, if o precedes o', we trivially have that $l(o) < l(o')$. As only reads have return values different from $\perp$ and clearly their return values match the ones they should have for a max register whose state is given by R, we have indeed constructed a valid linearization. □

Corollary 11.18. *We can implement a counter with maximum value M and the following properties.*

- *It is linearizable.*
- *All operations are wait-free.*
- *The step complexity of reads is* 1*.*
- *The step complexity of each increment operation is* $3\lceil \log n \rceil + 1$*.*
- *Its base objects are $\mathcal{O}(n)$ atomic read/write registers and max registers of maximum value M.*

Proof. W.l.o.g., suppose $n = 2^i$ for some $i \in \mathbb{N}_0$. We show the claim by induction on i, where the bound on the step complexity of increments is $3i + 1$. For the base case, observe that a linearizable wait-free counter with a single node that may increment it is given by a read/write register that is written by that node only, and it has step complexity 1 for all operations.

Now assume that the claim holds for some $i \in \mathbb{N}_0$. By the induction hypothesis, we have linearizable wait-free counters supporting 2^i incrementing nodes

[2]This is a slight abuse of notation, as it means that multiple increments may take effect at the same instant of time. Formally, this can be handled by splitting them up into individual increments that happen right after each other.

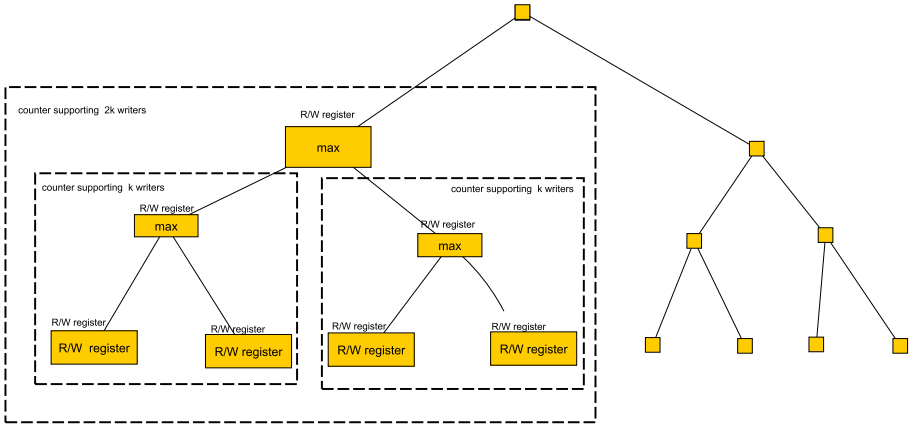

Figure 11.4: The recursive construction from Corollary 11.18, resulting in a tree. The leaves are simple read/write registers, which can be used as atomic counters with a single writer. A subtree of depth d implements a linearizable counter supporting 2^d writers, and by Lemma 11.13 it can be treated as atomic. Using a single additional max register, Lemma 11.19 shows how construct a counter supporting 2^{d+1} writers using 2 counters supporting 2^d writers.

(with the "right" step complexities and numbers of registers). If these were atomic, Lemma 11.17 would immediately complete the induction step. Applying Lemma 11.13, it suffices that they are linearizable implementations, i.e., the induction step succeeds. □

Remarks:

- This is an application of a reliable recipe: Construct something linearizable out of atomic base objects, "forget" that it's an implementation, pretend its atomic, rinse and repeat.
- Doing it without Lemma 11.13 would have meant to unroll the argument for the entire tree construction of Corollary 11.18, which would have been cumbersome and error-prone at best.

11.3.3 Max Registers from RW Registers

The construction of max registers with maximum value M from basic RW registers is structurally similar.

Lemma 11.19. *Suppose we are given two atomic max registers of maximum value M and an atomic read/write register. Then we can implement a max register with maximum value $2M$ and the following properties from these.*

- *It is linearizable.*
- *All operations are wait-free.*
- *Each read operation consists of one read of the RW register and reading one of the max registers.*

- *Each write operation consists of at most one read of the RW register and writing to one of the max registers.*

The construction is given in Algorithm 26. The proof of linearizability is left for the exercises.

Algorithm 26 Recursive construction of a max register of maximum value $2M$ from two max registers of maximum value M and a read/write register.

Given: max registers $R_<$ and $R_\geq$ of maximum value M, and RW register switch, all initialized to 0.

read

```
 1: if switch = 0 then
 2:    return R_<.read
 3: else
 4:    return M + R_≥.read
 5: end if
```

write(i)

```
 6: if i < M then
 7:    if switch = 0 then
 8:       R_<.write(i)
 9:    end if
10: else
11:    R_≥.write(i − M)
12:    switch := 1
13: end if
14: return ⊥
```

Corollary 11.20. *We can implement a max register with maximum value M and the following properties.*

- *It is linearizable.*
- *All operations are wait-free.*
- *The step complexity of all operations is $\mathcal{O}(\log M)$.*
- *Its base objects are $\mathcal{O}(M)$ atomic read/write registers.*

Proof sketch. Like in Corollary 11.18, we use Lemmas 11.13 and 11.19 inductively, where in each step of the induction the maximum value of the register is doubled. The base case of $M = 1$ is given by a read/write register initialized to 0: writing 0 requires no action, and writing 1 can be safely done, since no other value is ever (explicitly) written to the register; since both reads and writes require at most one step, the implementation is trivially linearizable. □

Theorem 11.21. *We can implement a counter with maximum value M and the following properties.*

- *It is linearizable.*
- *All operations are wait-free.*
- *The step complexity of reads is $\mathcal{O}(\log M)$.*

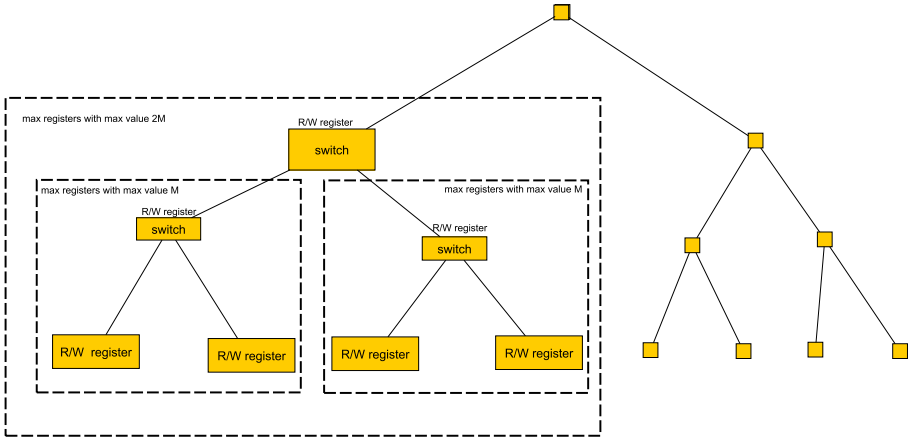

Figure 11.5: The recursive construction from Corollary 11.20, resulting in a tree. The leaves are simple read/write registers, which can be used as atomic max registers with maximum value 1. A subtree of depth d implements a linearizable max register of maximum value 2^d, and by Lemma 11.13 it can be treated as atomic. Using a single read/write register "switch," Lemma 11.19 shows how to control access to two max registers with maximum value 2^d to construct one with maximum value 2^{d+1}.

- *The step complexity of each increment operation is* $\mathcal{O}(\log M \log n)$.
- *Its base objects are* $\mathcal{O}(nM)$ *atomic read/write registers.*

Proof. We apply Lemmas 11.13 and 11.18 to the implementations of max registers of maximum value M given by Corollary 11.20. The step complexities follow, as we need to replace each access to a max register by the step complexity of the implementation. Similarly, the total number of registers is the number of read/write registers per max register times the number of used max registers (plus an additive $\mathcal{O}(n)$ read/write registers for the counter implementation that gets absorbed in the constants of the $\mathcal{O}$-notation). □

Remarks:

- As you will show in the exercises, writing to $R_<$ only if switch reads 0 is crucial for linearizability.
- If M is $n^{\mathcal{O}(1)}$, reads and writes have step complexities of $\mathcal{O}(\log n)$ and $\mathcal{O}(\log^2 n)$, respectively, and the total number of registers is $n^{\mathcal{O}(1)}$. As for many algorithms and data structures only polynomially many increments happen, this is a huge improvement compared to the linear step complexity the lower bound seems to imply!
- If one has individual caps c_i on the number of increments a node may perform, one can use respectively smaller registers. This improves the space complexity to $\mathcal{O}(\log n \sum_{i=1}^{n} c_i)$, as on each of the $\lceil \log n \rceil$ hierarchy levels (read: levels of the tree) of the counter construction, the max registers must be able to hold $\sum_{i=1}^{n} c_i$ *in total*, but not individually.

- For instance, if one wants to know the number of nodes participating in some algorithm or subroutine, this becomes $\mathcal{O}(n \log n)$.

- One can generalize the construction to cap the step complexity at n. However, at this point the space complexity is already exponential.

What to take Home

- This is another example demonstrating how lower bounds do more than just giving us a good feeling about what we've done. The lower bound was an essential guideline for the max register and counter constructions, as it told us that the bottleneck was the possibility of a very large number of increments!

- Advanced RMW registers are very powerful. At the same time, this means they are very expensive.

- Understanding and proving consistency for objects that should behave like they are accessed sequentially is challenging. One may speculate that we're not seeing the additional computational power of multi-processor systems with many cores effectively used in practice, due to the difficulty of developing scalable parallel software.

Bibliographic Notes

This lecture is based largely on two papers: Jayanti et al. [JTT00] proved the lower bounds on the space and step complexity of counters and, as discussed in the remarks, many other shared data structures. The presented counter implementation was developed by Aspnes et al. [AACH12]. In this paper, it is also shown how to use the technique to implement general *monotone* circuits (i.e., things only "increase," like for a counter), though the result is not linearizable, but satisfies a weaker consistency condition. Moreover, the authors show that randomized implementations of max registers with maximum value n must have step complexity $\Omega(\log n / \log \log n)$ for read operations, assuming that write operations take $\log^{\mathcal{O}(1)} n$ steps. In this sense their deterministic implementation is almost optimal!

Randomization [AC13] or using (deterministic) linearizable snapshots that support only polynomially many write operations of the underlying data structure [AACHE12] are other ways to circumvent the linear step complexity lower bound. Finally, any implementation of a max register of maximum value M from historyless objects requires $\Omega(\min\{M, n\})$ space [AACHH12].

Bibliography

[AACH12] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *Journal of the ACM*, 59(1):2:1–2:24, 2012.

[AACHE12] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Faster Than Optimal Snapshots (for a While): Preliminary Version. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, pages 375–384, 2012. Journal preprint at http://cs-www.cs.yale.edu/homes/aspnes/papers/limited-use-snapshots-abstract.html.

[AACHH12] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Danny Hendler. Lower bounds for restricted-use objects. In *Twenty-Fourth ACM Symposium on Parallel Algorithms and Architectures*, pages 172–181, 2012.

[AC13] James Aspnes and Keren Censor-Hillel. Atomic Snapshots in $O(\log^3 n)$ Steps Using Randomized Helping. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, pages 254–268, 2013.

[JTT00] P. Jayanti, K. Tan, and S. Toueg. Time and Space Lower Bounds for Nonblocking Implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.

Lecture 12

The Port Numbering Model

Today we're looking at a particularly weak model of computation: deterministic algorithms in the message passing model *without node identifiers*. This means that we have to specify whether (and how) nodes can tell each other apart: while nodes are anonymous, there is the question whether they can recognize if two messages originate from the same neighbor or not. We assume that they can, and model this by *port numbers*. A node of degree δ_v has a bijection p from $1, \ldots, \delta_v$ to its edges. Whenever it receives a message, it "sees" the port on which it arrives, and thus knows it was sent by the node incident to the respective edge. Likewise, whenever it sends a message, it specifies the port on which it sends the message, and the other endpoint of the respective edge is the receiver of the message.

We care neither about message size nor the number of messages sent. Hence we can run an α-synchronizer, which in turn means that it's fine to assume that the system is synchronous to begin with. Altogether, this is called the *port numbering model*. Let's wrap up how it works:

- The network is described by a simple connected graph $G = (V, E)$.
- For each node $v \in V$, there is a bijection $p_v \colon \{w \in V \mid \{v, w\} \in E\} \to 1, \ldots, \delta_v$.
- The system operates in synchronous rounds. In each round, each node v

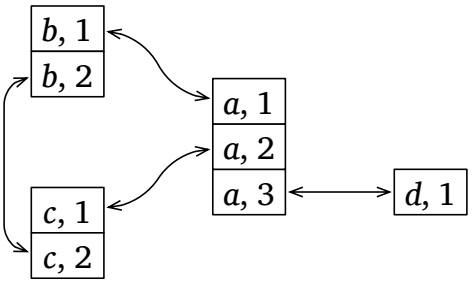

Figure 12.1: A port numbering network consisting of nodes a, b, c, d.

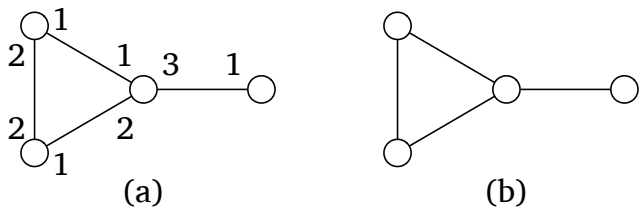

Figure 12.2: The same network represented as a labeled graph (a) and without indicating port numbers (b).

1. performs arbitrary (finite) local computations,
2. sends a message to each port $1, \ldots, \delta_v$ (sending none is ok, too), and
3. receives, for each neighbor w, on port $p_v(w)$ the message w sent to its own port $p_w(v)$.

- As usual, nodes may be given additional inputs, and should eventually terminate and return a value so that all outputs together describe a solution of the problem at hand.

Remarks:

- Randomization is out of the question this time, simply because it permits to generate unique identifiers with high probability.
- We study this model primarily for understanding the relative power of the models.
- Lower bounds in the port numbering model can also be a good starting point for ones in stronger models. They are usually easier to show, and in some cases it's possible to "lift" the result to a more powerful one by using simulation (i.e., showing that at least for the considered problem the "stronger" model is not actually stronger).

12.1 What we can't do

Having no identifiers is quite the bummer. We cannot break symmetry, so we can basically do nothing at all.[1]

Theorem 12.1. *In general, it is impossible to break symmetry in the port numbering model. In particular, one cannot always*

- *solve leader election,*
- *find proper vertex or edge colorings,*
- *determine a non-empty independent set,*
- *determine a dominating set that does not contain all nodes,*

[1] Or *do* we?

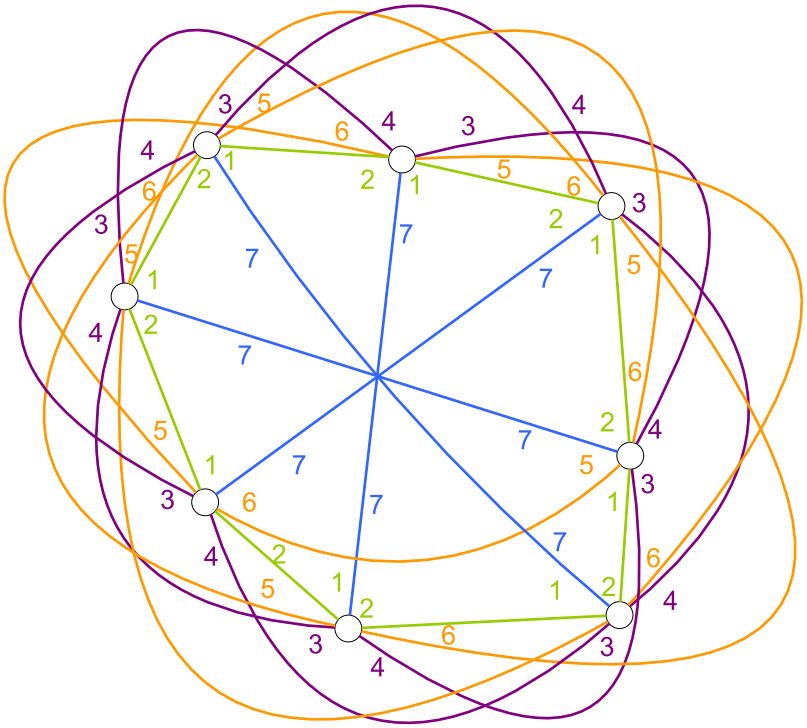

Figure 12.3: A symmetric port numbering network with 8 nodes of degree 7.

- *find a non-empty matching, or*
- *compute a minimum vertex cover.*

This holds also when restricting to graphs of uniform degree Δ*, for any* $1 < \Delta \in \mathbb{N}$ *and* $n > \Delta$*.*

Proof. Assume that initially, all nodes are in the same state and all nodes have the same degree Δ. Then each node will send the same message to a given port $i \in \{1, \ldots, \Delta\}$. Thus, if each port i connects to the other endpoint of the corresponding edge with the same port number $j(i) \in \{1, \ldots, \Delta\}$, each node receives the same message on port $j(i)$, implying that all nodes are in the same state at the end of the first round. By induction, this shows that symmetry cannot be broken and all the above statements readily follow.

Hence, all we need to do is to show that, for any fixed $\Delta > 1$ and any $n > \Delta$, connected, simple port-numbered graphs with the following property exist: there is a bijection $b\colon \{1, \ldots, \Delta\} \to \{1, \ldots, \Delta\}$ so that for each edge $\{v, w\} \in E$, it holds that $p_v(w) = b(p_w(v))$.

Consider the following graphs and port numberings:

- For even Δ, connect for each $h \in \{1, \ldots, \Delta/2\}$ node $i \in \{1, \ldots, n\}$ to node $(i + h) \bmod n$ with port number $2h - 1$ at node i and port number $2h$ at node $(i + h) \bmod n$.

- For odd Δ, observe that n must be even (as the sum of degrees must be even). Do the same as for even Δ for $h \in \{1, \ldots, \lfloor \Delta/2 \rfloor\}$. Then add the perfect matching $\{1, \lceil n/2 \rceil\}, \{2, \lceil n/2 \rceil + 1\}, \ldots, \{\lfloor n/2 \rfloor, n\}$ with port number Δ for both endpoints of each edge.

The bijection b is then given as follows

- If Δ is even:
$$b(i) = \begin{cases} i+1 & \text{if } i \text{ is odd, and} \\ i-1 & \text{if } i \text{ is even.} \end{cases}$$
- And for odd Δ:
$$b(i) = \begin{cases} i+1 & \text{if } i < \Delta \text{ and } i \text{ is odd,} \\ i-1 & \text{if } i \text{ is even, and} \\ i & \text{if } i = \Delta. \end{cases}$$
□

Remarks:

- Of course, the list of things one cannot do in this model given in the theorem could go on forever.
- For $\Delta \leq n/2$, one can also construct bipartite graphs with $b(i) = i$ in a similar fashion. So nothing solvable in this case either? Can we do anything at all?!?

12.2 Bipartite Matching

Let's make our life a little bit easier. We consider 2-colored graphs now, where each node has its color as input. This is still fairly natural: Think of relations such as client/server, VIP/fan, or hypergraphs,[2] where we represent each hyperedge by a node (on one side of the bipartite graph) connected to its constituent nodes (on the other side).

Now finding a maximal matching is straightforward.

Theorem 12.2. *On 2-colored graphs of maximum degree Δ, Algorithm 27 computes a maximal matching in 2Δ rounds.*

Proof. Each white node is incident to at most one matching edge, because in each iteration it proposes only a single edge and terminates if it is selected. Each black node is incident to at most one matching edge, because it accepts only a single proposal. Any edge will be proposed, unless its white endpoint is matched before it gets to proposing it. Any proposed edge will be accepted, unless its black endpoint is already matched. Hence, any edge that is not in the matching is adjacent to a matching edge, implying that the matching is maximal.

The time complexity is 2Δ, as there are Δ iterations, each consisting of one round for proposals and one for accepts. □

[2] A hypergraphs is a structure $H = (V, E)$ in which each *hyperedge* $e \in E$ is an arbitrary subset of the nodes.

Algorithm 27 Matching in 2-colored graphs of maximum degree Δ using port numberings, code at node v. Nodes return their matched port or $\perp$ if none of their incident edges is in the matching.

```
1:  for i = 1, ..., Δ do
2:    if v is white then
3:      send propose to port number i
4:    else if v receives propose then
5:      send accept to minimal port j at which propose was received
6:      return j
7:    end if
8:    if v receives accept on port i then
9:      return i
10:   end if
11: end for
12: return ⊥
```

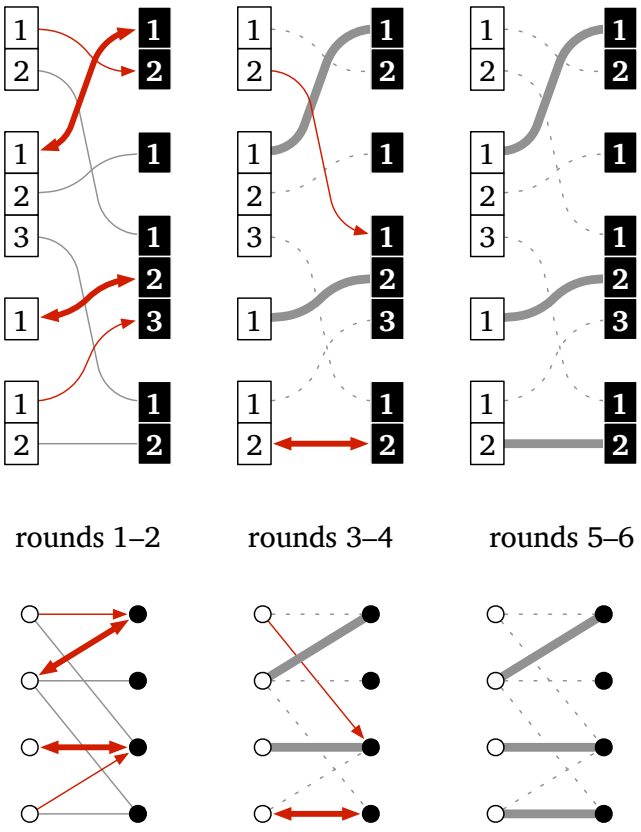

Figure 12.4: An execution of the matching algorithm. Red edges are proposed in this round, grey edges have not yet been proposed, and grey dotted lines did not make it; thick edges are matched.

- Not very fast if Δ is large. However, if Δ is a constant, so is the running time of the algorithm.
- The assumption that we have a 2-coloring does some intial symmetry breaking for us. For instance, we already have the best coloring one could get, and each color is a maximal independent set (as the graph is connected).
- However, we still cannot solve leader election, regardless of how much time we spend. Based on identifiers, this is possible, so the model is still weaker than the standard message passing model!
- This feels like cheating. Let's do something without requiring a 2-coloring!

12.3 3-Approximating Minimum Vertex Cover

We know that we cannot solve minimum vertex cover precisely, but that's ok – it's an NP-complete problem anyway. It's even NP-hard to approximate within a constant and, assuming the unique games conjecture, NP-hard to approximate better than factor $2 - o(1)$. On the other hand, obtaining a 2-approximation is easy: just output a maximal matching!

...except that we can't do that. We cannot compute *any* non-trivial matching. We first need to transform the graph into something we can handle: a 2-colored graph. Once this goal is set, it is actually not too hard to achieve.

- Replace each node by 2 copies, a white copy and a black copy.
- For an edge $\{v, w\}$, connect the white copy of v to the black copy of w and the black copy of v to the white copy of w.
- The new edges inherit their port numbers from the originals.

Lemma 12.3. *For a port-numbered graph $G = (V, E, \{p_v\}_{v \in V})$, denote the (port-numbered) graph constructed above by G'. Then the constructed port numbering on G' is feasible. Moreover, G' is 2-colored (by nodes being white and black), has the same maximum degree as G, and the port-numbering model on G' can be simulated on G without overhead in round complexity.*

Proof. All properties are straightforward. Neighbors in G' have different colors by construction, the new nodes have the same degree as the originals, inheriting port numbers results in port numbers $1, \ldots, \delta_v$ for a node of degree δ_v, and each node can simulate both of its copies, where communication on new edges is performed via the original edges. □

Theorem 12.4. *A 4-approximation to vertex cover can be computed in $\mathcal{O}(\Delta)$ rounds of the port numbering model.*

Proof. We construct G' and simulate Algorithm 27. The algorithm can be made to terminate without knowledge of Δ by non-matched white nodes terminating once they proposed on all their ports, letting their neighbors know, and non-matched black nodes terminating once all their neighbors terminated. By

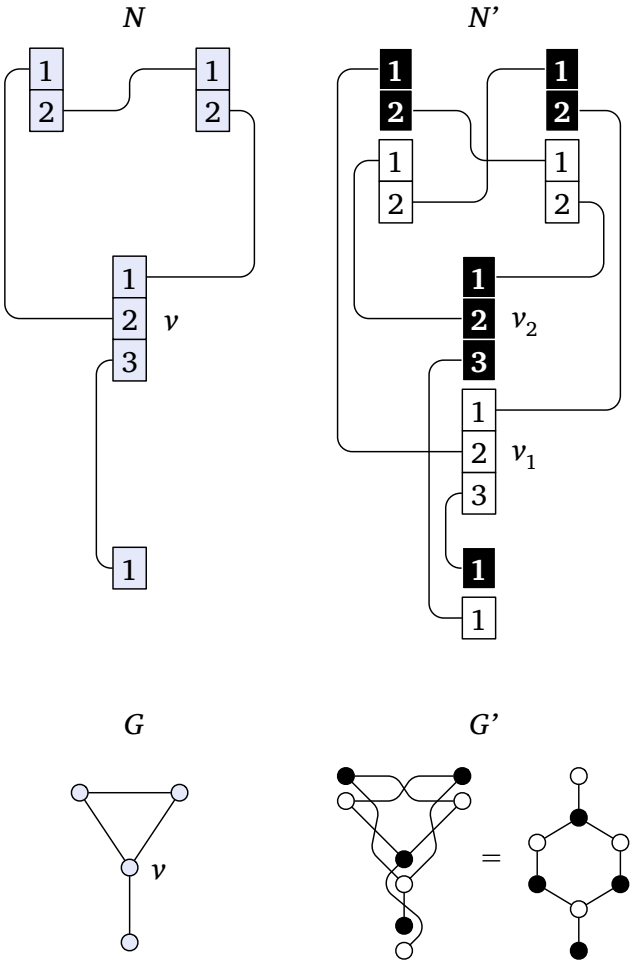

Figure 12.5: The construction of G' from G.

Theorem 12.2, Lemma 12.3, and this modification, the algorithm terminates in $\mathcal{O}(\Delta)$ rounds.

If a black or white copy of a node is incident to a matching edge in G', the node is in the vertex cover, otherwise it is not. Recall that the endpoints of a maximal matching form a 2-approximate vertex cover (as shown in Corollary 5.17). Because any vertex cover of G induces a vertex cover of G' of at most twice the size, the returned set has at most 4 times the size of a minimum vertex cover. As all edges in G' have at least one incident node in the vertex cover of G', the same is true in the computed node set of G, i.e., we did indeed find a vertex cover of G. □

If we look a bit closer, there's another surprise. The result is, in fact, a 3-approximation!

Corollary 12.5. *The algorithm from Theorem 12.4 returns a vertex cover that is at most factor* 3 *larger than the optimum.*

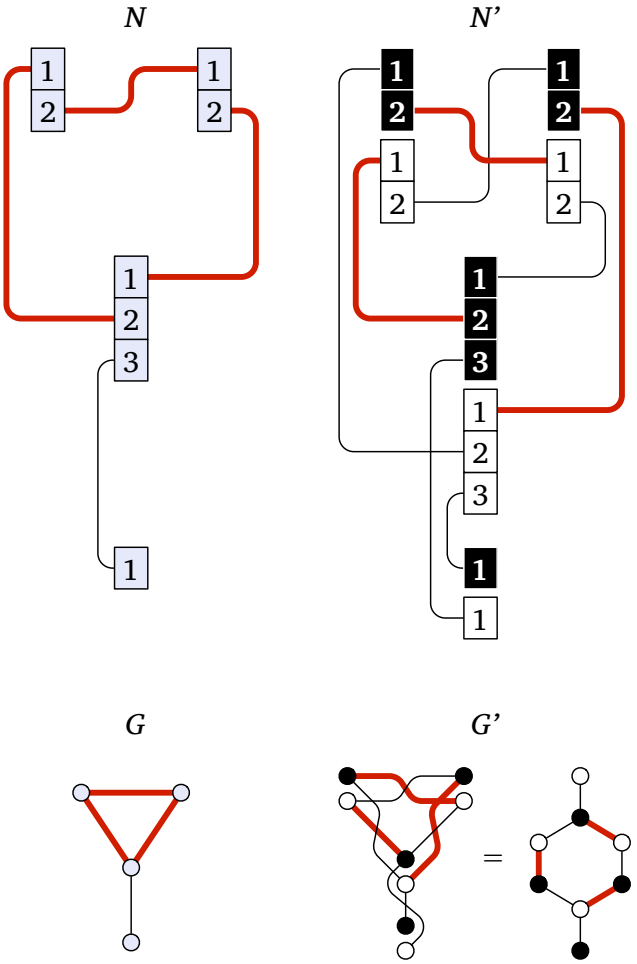

Figure 12.6: The computed matching in G' and the corresponding edges in G.

Proof. Consider the originals of the matching edges in the maximal matching of G'. These induce a subgraph of G of maximum degree 2, which is a disjoint union of paths and cycles of $k \geq 2$ nodes. To cover all edges of G, one needs to cover in particular these edges, and they can only be covered by the nodes on the respective paths and cycles. To cover a cycle of k nodes, at least $k/2$ of its nodes must be selected. To cover a path of k nodes, at least $\lfloor k/2 \rfloor$ of its nodes must be selected. As we choose all these nodes (and no others), the size of the computed vertex cover is at most factor $3 = \max_{2 \leq k \in \mathbb{N}} \{k/\lfloor k/2 \rfloor\}$ larger than the optimum. □

What to take Home

- Even in very restricted models, some things can be done efficiently.

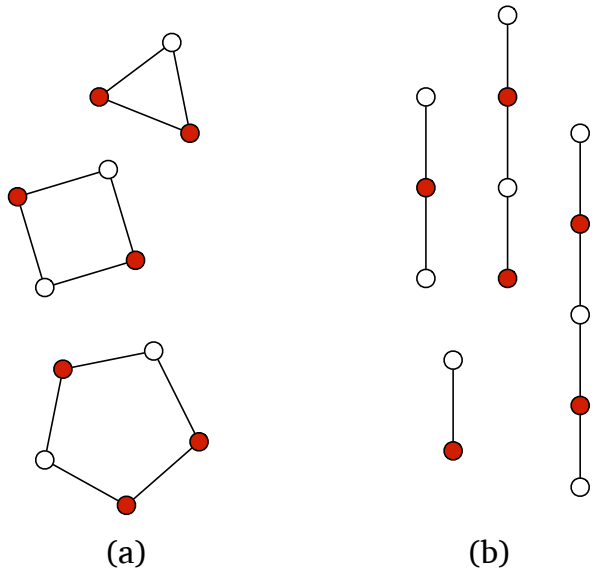

Figure 12.7: Examples for minimum vertex covers of cycles (a) and paths (b).

- Frequently, the resulting algorithms are very clean and simple, making them easy to implement.
- Such models tend to highlight what makes stronger models stronger. This can be useful in finding lower bounds or better algorithms, or just in pointing out that, e.g., one very important use of having node identifiers is the ability to execute Cole-Vishkin.
- Basic concepts such as simulation and indistinguishability are the name of the game also here.

Bibliographic Notes

The presented vertex cover algorithm is due to Polishchuk and Suomela [PS09]. Using Cole-Vishkin *on edge weights*, it is possible to obtain a 2-approximation of the weighted version of the problem [ÅS10]. On the negative side, one cannot hope for anything better than a 2-approximation in the port-numbering model: in an even cycle an optimum solution chooses half of the nodes, but a symmetric port numbering causes all nodes to be selected. Even if identifiers and randomization are available, a classic construction shows that any distributed algorithm finding a reasonable approximation requires $\Omega(\sqrt{\log n})$ and $\Omega(\log \Delta)$ rounds.[3] Recently, it has been shown that there is some constant $\delta > 0$ so that finding a $(1+\delta)$-approximation cannot be done in $o(\log n)$ rounds [GS12], even if the graph is 2-colored and has degree 3! Note that this an unconditional lower

[3]This is the same construction. Choosing the maximum feasible value of Δ for a given value of n yields the $\Omega(\sqrt{\log n})$ bound.

bound. It does not depend on P≠NP or similar assumptions, but arises from locality issues.

All figures but Figure 12.1 are courtesy of Jukka Suomela and under a creative commons license.[4] Large parts of today's lecture are my own narrative of a part of Jukka's course on deterministic distributed algorithms. A reference to his survey of local algorithms [Suo13] is also in order; a local algorithm is a distributed algorithm whose running time is bounded by a constant.

Bibliography

[ÅS10] Matti Åstrand and Jukka Suomela. Fast Distributed Approximation Algorithms for Vertex Cover and Set Cover in Anonymous Networks. In *Proc. 22nd Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 294–302, 2010.

[GS12] Mika Göös and Jukka Suomela. No Sublogarithmic-time Approximation Scheme for Bipartite Vertex Cover. In *Proc. 26th Conference on Distributed Computing (DISC)*, pages 181–194, 2012.

[PS09] Valentin Polishchuk and Jukka Suomela. A Simple Local 3-approximation Algorithm for Vertex Cover. *Information Processing Letters*, 109(12):642–645, 2009.

[Suo13] Jukka Suomela. Survey of Local Algorithms. *ACM Computing Surveys*, 45(2):24:1–24:40, March 2013.

Appendix A

Notation and Preliminaries

This appendix sums up important notation, definitions, and key lemmas that are not the main focus of the lecture.

A.1 Numbers and Sets

In this lecture, zero is not a natural number: $0 \notin \mathbb{N}$; we just write $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$ whenever we need it. $\mathbb{Z}$ denotes the integers, $\mathbb{Q}$ the rational numbers, and $\mathbb{R}$ the real numbers. We use $\mathbb{R}^+ = \{x \in \mathbb{R} \mid x > 0\}$ and $\mathbb{R}_0^+ = \{x \in \mathbb{R} \mid x \geq 0\}$, with similar notation for $\mathbb{Z}$ and $\mathbb{Q}$.

Rounding down $x \in \mathbb{R}$ is denoted by $\lfloor x \rfloor := \max\{z \in \mathbb{Z} \mid z \leq x\}$ and rounding up by $\lceil x \rceil := \min\{z \in \mathbb{Z} \mid z \geq x\}$.

For $n \in \mathbb{N}_0$, we define $[n] := \{0, \ldots, n-1\}$, and for a set M and $k \in \mathbb{N}_0$, $\binom{M}{k} := \{N \subseteq M \mid |N| = k\}$ is the set of all subsets of M that contain exactly k elements.

A.2 Graphs

A finite set of *vertices*, also referred to as *nodes* V together with *edges* $E \subseteq \binom{V}{2}$ defines a *graph* $G = (V, E)$. Unless specified otherwise, G has $n = |V|$ vertices and $m = |E|$ edges. Since edges are sets of exactly two vertices $e = \{v, w\} \subseteq V$,[1] our graphs have no *loops*, are *undirected* and have no *parallel edges*. This definition does not include *edge weights*, either. All of this together is equivalent of saying that we deal with *simple* graphs.

If $e = \{v, w\} \in E$, the vertices v and w are *adjacent*, and e is *incident* to v and w, furthermore, $e' \in E$ is *adjacent* to e if $e \cap e' \neq \emptyset$. The *neighboorhood* of v is

$$N_v := \{w \in V \mid \{v, w\} \in E\}, \tag{A.1}$$

i.e., the set of vertices adjacent to v. The *degree* of v is

$$\delta_v := |N_v|, \tag{A.2}$$

[1] Still, we occasionally write edges as tuples: $e = (v, w)$.

the size of v's neighborhood. We denote by

$$\Delta := \max_{v \in V} \delta_v \tag{A.3}$$

the maximum degree in G.

A v_1-v_d-*path* p is a set of edges $p = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{d-1}, v_d\}\}$ such that $|\{e \in p \mid v \in e\}| \leq 2$ for all $v \in V$. p has $|p|$ *hops*, and we call p a *cycle* if it visits all of its nodes exactly twice. The *distance* between $v, w \in V$ is

$$\text{dist}(v, w) := \min\{|p| \mid p \text{ is a } v\text{-}w\text{-path}\}, \tag{A.4}$$

which gives rise to the *diameter* D of G,

$$D := \max_{v,w \in V} \text{dist}(v, w), \tag{A.5}$$

the maximum pairwise distance between nodes.

A.2.1 Weighted Graphs

A *weighted graph* is a graph (V, E) together with weighting function $W \colon E \to \mathbb{R}$; we write $G = (V, E, W)$. An edge $e \in E$ has *weight* $W(e)$, and an edge set $E' \subseteq E$ has weight $W(E') := \sum_{e \in E'} W(e)$. Observe that since paths are sets of edges, this definition captures the weight of a path p: $W(p) = \sum_{e \in p} W(e)$.

In weighted graphs, distances are more complex than in simple graphs, because there are several measures: the smallest weight of a path, and the number of hops. The *distance* between $v, w \in V$ is the weight of a shortest v-w-path

$$\text{dist}(v, w) := \min\{W(p) \mid p \text{ is a } v\text{-}w\text{-path}\}, \tag{A.6}$$

and the *hop distance* is the smallest number of hops required to attain a shortest v-w-path is

$$\text{hop}(v, w) := \min\{|p| \mid p \text{ is } v\text{-}w\text{-path} \wedge W(p) = \text{dist}(v, w)\}. \tag{A.7}$$

Note that shortest paths can be very long in terms of hops, even if there is some (non-shortest) path with few hops: Even if $\{v, w\} \in E$, $\text{hop}(v, w) = n - 1$ is still possible (think about a circle with one heavy and $n - 1$ light edges).

A.2.2 Trees and Forests

A *forest* is a cycle-free graph, and a *tree* is a connected forest. Trees have $n - 1$ edges and a unique path between any pair of vertices. The tree $T = (V, E)$ is *rooted* if it has a designated root node $r \in V$; in which case it has a *depth* d

$$d := \max_{v \in V} \text{dist}(r, v), \tag{A.8}$$

which is the maximum distance from any node to the root node.

A.2.3 Cuts

Given a graph $G = (V, E)$ and distinct vertices $s, t \in V$, an s-t *cut* is a non-trivial partition of the vertices $V_s \dot\cup V_t = V$, such that $s \in V_s$ and $t \in V_t$. The *weight* of the cut is $|E \cap (V_s \times V_t)|$, i.e., the number of the edges connecting a vertex in V_s with to a vertex in V_t (in a weighted graph, the weight of the cut is the sum of those edges' weights). Alternatively, cuts can be represented as only the set V_s ($V_t = V \setminus V_s$), or as the edge set $E \cap (V_s \times V_t)$.

A.3 Logarithms and Exponentiation

Logarithms are base 2 logarithms, unless specified otherwise. Iterated exponentiation is denoted by ${}^a b$, which is the a-fold ($a \in \mathbb{N}_0$) exponentiation of b:

$${}^a b := \begin{cases} 1 & \text{if } a = 0 \\ b^{({}^{a-1}b)} & \text{otherwise.} \end{cases} \tag{A.9}$$

This is commonly referred to as *power tower* ${}^a 2 = 2^{2^{\cdot^{\cdot^{\cdot^2}}}}$. $\log_b^* x$ answers the inverse question of how often the logarithm has to be iteratively applied to end up with a result of at most 1:

$$\log_b^* x := \begin{cases} 0 & \text{if } x \leq 1 \\ 1 + \log_b^*(\log_b x) & \text{otherwise.} \end{cases} \tag{A.10}$$

A simple inductive check confirms $\log_b^* {}^a b = a$.

A.4 Probability Theory

We use some basic tools from probability theory in order to analyze randomized algorithms. The first of these tools states that the probability that at least one of k events occur is bounded by the sum of the individual events' probabilities.

Theorem A.1 (Union Bound)**.** *Let $\mathcal{E}_i$, $i \in [k]$ be events. Then*

$$P\left[\bigcup_{i \in [k]} \mathcal{E}_i\right] \leq \sum_{i \in [k]} p_i, \tag{A.11}$$

which is tight if $\mathcal{E}_{[k]}$ are disjoint.

Another key property is that expactation is compatible with summation:

Theorem A.2 (Linearity of Expectation)**.** *Let X_i for $i \in [k]$ denote random variables. Then*

$$\mathbb{E}\left[\sum_{i \in [k]} X_i\right] = \sum_{i \in [k]} \mathbb{E}[X_i]. \tag{A.12}$$

Markov's inequality and Chernoff's bound both bound the probability that a random variable attains a very different value from its expected value. The preconditions for Markov's inequality are much weaker than those for Chernoff's bound, but the latter is stronger than the former.

Theorem A.3 (Markov's Inequality)**.** *Let X be a positive random variable (in fact, $P[X \geq 0] = 1$ and $P[X = 0] < 1$ suffice). Then for any $K > 1$,*

$$P[X \geq K\mathbb{E}[X]] \leq \frac{1}{K}. \tag{A.13}$$

Theorem A.4 (Chernoff's Bound)**.** *Let $X = \sum_{i\in[k]} X_i$ be the sum of k independent Bernoulli variables (i.e., 0-1-variables). Then we have, for any $0 < \delta \leq 1$,*

$$P[X \geq (1+\delta)\mathbb{E}[X]] \leq e^{-\delta^2\mathbb{E}[X]/3}, \text{ and} \tag{A.14}$$

$$P[X \leq (1-\delta)\mathbb{E}[X]] \leq e^{-\delta^2\mathbb{E}[X]/2}. \tag{A.15}$$

The concept of something happending *with high probability (w.h.p.),* i.e., with proability at least $1 - n^{-c}$, is the following. First, the larger your input n, the larger the probability that the event occurs. Second, c can be picked at will, meaning that the probabilities can be picked as close to 1 as desired. This is useful for randomized algorithms. Suppose you have a randomized algorithm $\mathcal{A}$ that succeeds with probability p. If you want to use $\mathcal{A}$ several times (e.g. to construct a new algorithm) the probability that all of these calls succeed decreases. But if *each* call of $\mathcal{A}$ succeeds w.h.p. and there only are polynomially many of them, you can use the union bound and pick a large enough c to show that *all* calls of $\mathcal{A}$ succeed w.h.p. as well.

Definition A.5 (With high Probability)**.** *The event $\mathcal{E}$ occurs* with high probability (w.h.p.) *if $P[\mathcal{E}] \geq 1 - 1/n^c$ for any fixed choice of $1 \leq c \in \mathbb{R}$. Note that c typically is considered a constant in terms of the $\mathcal{O}$-notation.*

A.5 Asymptotic Notation

We require asymptotic notation to reason about the complexity of algorithms. This section is adapted from Chapter 3 of Cormen et al. [CLR90]. Let $f, g\colon \mathbb{N}_0 \to \mathbb{R}$ be functions.

A.5.1 Definitions

$\mathcal{O}(g(n))$ is the set containing all functions f that are bounded from above by $cg(n)$ for some constant $c > 0$ and for all sufficiently large n, i.e. $f(n)$ is *asymptotically bounded from above* by $g(n)$.

$$\mathcal{O}(g(n)) := \{f(n) \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}_0 \colon \quad \forall n \geq n_0 \colon \quad 0 \leq f(n) \leq cg(n)\} \tag{A.16}$$

The counterpart of $\mathcal{O}(g(n))$ is $\Omega(g(n))$, the set of functions *asymptotically bounded from below* by $g(n)$, again up to a postive scalar and for sufficiently large n:

$$\Omega(g(n)) := \{f(n) \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}_0 \colon \quad \forall n \geq n_0 \colon \quad 0 \leq cg(n) \leq f(n)\} \tag{A.17}$$

If $f(n)$ is bounded from below by $c_1 g(n)$ and from above by $c_2 g(n)$ for positive scalars c_1 and c_2 and sufficiently large n, it belongs to the set $\Theta(g(n))$; in this case $g(n)$ is an *asymptotically tight* bound for $f(n)$. It is easy to check that $\Theta(g(n))$ is the intersection of $\mathcal{O}(g(n))$ and $\Omega(g(n))$.

$$\begin{aligned}\Theta(g(n)) := \{f(n) \mid \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}_0 \colon \quad \forall n \geq n_0 \colon \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}\end{aligned} \tag{A.18}$$

$$f(n) \in \Theta(g(n)) \quad \Leftrightarrow \quad f \in (\mathcal{O}(g(n)) \cap \Omega(g(n))) \tag{A.19}$$

For example, $n \in \mathcal{O}(n^2)$ but $n \notin \Omega(n^2)$ and thus $n \notin \Theta(n^2)$.[2] But $3n^2 - n + 5 \in \mathcal{O}(n^2)$, $3n^2 - n + 5 \in \Omega(n^2)$, and thus $3n^2 - n + 5 \in \Theta(n^2)$ for $c_1 = 1$, $c_2 = 3$, and $n_0 = 4$.

In order to express that an asymptotic bound is not tight, we require $o(g(n))$ and $\omega(g(n))$. $f(n) \in o(g(n))$ means that for any postitive constant c, $f(n)$ is strictly smaller than $cg(n)$ for sufficiently large n.

$$o(g(n)) := \{f(n) \mid \forall c \in \mathbb{R}^+ \colon \exists n_0 \in \mathbb{N}_0 \colon \forall n \geq n_0 \colon \quad 0 \leq f(n) < cg(n)\} \tag{A.20}$$

As an example, consider $\frac{1}{n}$. For arbitrary $c \in \mathbb{R}^+$, $\frac{1}{n} < c$ we have that for all $n \geq \frac{1}{c} + 1$, so $\frac{1}{n} \in o(1)$. A similar concept exists for lower bounds that are not asymptotically tight; $f(n) \in \omega(g(n))$ if for any positive scalar c, $cg(n) < f(n)$ as soon as n is large enough.

$$\omega(g(n)) := \{f(n) \mid \forall c \in \mathbb{R}^+ \colon \exists n_0 \in \mathbb{N}_0 \colon \forall n \geq n_0 \colon \quad 0 \leq cg(n) < f(n)\} \tag{A.21}$$

$$f(n) \in \omega(g(n)) \quad \Leftrightarrow \quad g(n) \in o(f(n)) \tag{A.22}$$

A.5.2 Properties

We list some useful properties of asymptotic notation, all taken from Chapter 3 of Cormen et al. [CLR90]. The statements in this subsection hold for all $f, g, h \colon \mathbb{N}_0 \to \mathbb{R}$.

Transitivity

$$f(n) \in \mathcal{O}(g(n)) \wedge g(n) \in \mathcal{O}(h(n)) \quad \Rightarrow \quad f(n) \in \mathcal{O}(h(n)), \tag{A.23}$$

$$f(n) \in \Omega(g(n)) \wedge g(n) \in \Omega(h(n)) \quad \Rightarrow \quad f(n) \in \Omega(h(n)), \tag{A.24}$$

$$f(n) \in \Theta(g(n)) \wedge g(n) \in \Theta(h(n)) \quad \Rightarrow \quad f(n) \in \Theta(h(n)), \tag{A.25}$$

$$f(n) \in o(g(n)) \wedge g(n) \in o(h(n)) \quad \Rightarrow \quad f(n) \in o(h(n)), \text{ and} \tag{A.26}$$

$$f(n) \in \omega(g(n)) \wedge g(n) \in \omega(h(n)) \quad \Rightarrow \quad f(n) \in \omega(h(n)). \tag{A.27}$$

Reflexivity

$$f(n) \in \mathcal{O}(f(n)), \tag{A.28}$$

$$f(n) \in \Omega(f(n)), \text{ and} \tag{A.29}$$

$$f(n) \in \Theta(f(n)). \tag{A.30}$$

Symmetry

$$f(n) \in \Theta(g(n)) \quad \Leftrightarrow \quad g(n) \in \Theta(f(n)). \tag{A.31}$$

Transpose Symmetry

$$f(n) \in \mathcal{O}(g(n)) \quad \Leftrightarrow \quad g(n) \in \Omega(f(n)), \text{ and} \tag{A.32}$$

$$f(n) \in o(g(n)) \quad \Leftrightarrow \quad g(n) \in \omega(f(n)). \tag{A.33}$$

[2] We write $f(n) \in \mathcal{O}(g(n))$ unlike some authors who, by abuse of notation, write $f(n) = \mathcal{O}(g(n))$. $f(n) \in \mathcal{O}(g(n))$ emphasizes that we are dealing with *sets* of functions.

Bibliography

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.