Chapter 9

Global Problems

This chapter is on “hard” problems in distributed computing. In sequential computing, there are NP-hard problems which are conjectured to take exponential time. Is there something similar in distributed computing? Using flooding/echo (Algorithms 2.9,2.10) from Chapter 2, everything so far was solvable basically in $O(D)$ time, where $D$ is the diameter of the network.

9.1 Diameter & APSP

But how do we compute the diameter itself? With flooding/echo, of course!

Algorithm 9.1 Naive Diameter Construction
1. all nodes compute their radius by synchronous flooding/echo
2. all nodes flood their radius on the constructed BFS tree
3. the maximum radius a node sees is the diameter

Remarks:
- Since all these phases only take $O(D)$ time, nodes know the diameter in $O(D)$ time, which is asymptotically optimal.
- However, there is a problem! Nodes are now involved in $n$ parallel flooding/echo operations, thus a node may have to handle many and big messages in one single time step. Although this is not strictly illegal in the message passing model, it still feels like cheating! A natural question is whether we can do the same by just sending short messages in each round.
- In Definition 1.8 of Chapter 1 we postulated that nodes should send only messages of “reasonable” size. In this chapter we strengthen the definition a bit, and require that each message should have at most $O(\log n)$ bits. This is generally enough to communicate a constant number of ID’s or values to neighbors, but not enough to communicate everything a node knows!
- A simple way to avoid large messages is to split them into small messages that are sent using several rounds. This can cause that messages are getting delayed in some nodes but not in others. The flooding might not use edges of a BFS tree anymore! These floodings might not compute correct distances anymore! On the other hand we know that the maximal message size in Algorithm 9.1 is $O(n \log n)$. So we could just simulate each of these “big message” rounds by $n$ “small message” rounds using small messages. This yields a runtime of $O(nD)$ which is not desirable. A third possible approach is “starting each flooding/echo one after each other” and results in $O(nD)$ in the worst case as well.
- So let us fix the above algorithm! The key idea is to arrange the flooding-echo processes in a more organized way: Start the flooding processes in a certain order and prove that at any time, each node is only involved in one flooding. This is realized in Algorithm 9.3.

Definition 9.2. (BFS,.) Performing a breadth first search at node $v$ produces spanning tree $BFS_v$ (see Chapter 2). This takes time $O(D)$ using small messages.

Remarks:
- A spanning tree of a graph $G$ can be traversed in time $O(n)$ by sending a pebble over an edge in each time slot.
- This can be done using, e.g., a depth first search (DFS). Start at the root of a tree, recursively visit all nodes in the following way. If the current node still has an unvisited child, then the pebble always visits that child first. Return to the parent only when all children have been visited.
- Algorithm 9.3 works as follows: Given a graph $G$, first a leader $l$ computes its BFS tree $BFS_l$. Then we send a pebble $P$ to traverse tree $BFS_l$. Each time pebble $P$ enters a node $v$ for the first time, $P$ waits one time slot, and then starts a breadth first search (BFS) – using edges in $G$ – from $v$ with the aim of computing the distances from $v$ to all other nodes. Since we start a BFS from every node $v$, each node $v$ learns its distance to all these nodes $v$ during the according execution of BFSs. There is no need for an echo-process at the end of BFSs.

Algorithm 9.3 Computes APSP on $G$
1: Assume we have a leader node $l$ (if not, compute one first)
2: compute $BFS_l$ of leader $l$
3: send a pebble $P$ to traverse tree $BFS_l$. Each time pebble $P$ enters a node $v$ for the first time, $P$ waits one time slot, and then starts a breadth first search (BFS) – using edges in $G$ – from $v$ with the aim of computing the distances from $v$ to all other nodes. Since we start a BFS from every node $v$, each node $v$ learns its distance to all these nodes $v$ during the according execution of BFSs. There is no need for an echo-process at the end of BFSs.
Remarks:
- The depth of node \( u \) in BFS, is \( d(u, v) \)
- Having all distances is nice, but how do we get the diameter? Well, as before, each node could just flood its radius (its maximum distance) into the network. However, messages are small now and we need to modify this slightly. In each round a node only sends the maximal distance that it is aware of to its neighbors. After \( D \) rounds each node will know the maximum distance among all nodes.

**Lemma 9.4.** In Algorithm 9.3, at no time a node \( w \) is simultaneously active for both BFS, and BFSi.

Proof. Assume a BFS is started at time \( t_u \) at node \( u \). Then node \( w \) will be involved in BFS at time \( t_u + d(u, w) \). Now, consider a node \( v \) whose BFS is started at time \( t_v \), then \( u \). According to the algorithm this implies that the pebble visits \( v \) after \( u \) and took some time to travel from \( u \) to \( v \). In particular, the time to get from \( u \) to \( v \) is at least \( d(u, v) \). In addition the pebble waited one time slot at node \( u \) after starting BFS, and we have \( t_v \geq t_u + d(u, v) + 1 \). Using this and the triangle inequality, we get that node \( w \) is involved in BFS strictly after being involved in BFS, since \( t_w + d(v, w) \geq (t_u + d(v, u) + 1) + d(v, w) \geq t_u + d(u, v) + 1 > t_u + d(u, w) \).

**Theorem 9.5.** Algorithm 9.3 computes APS (all pairs shortest path) in time \( \mathcal{O}(n) \).

Proof. Since the previous lemma holds for any pair of vertices, no two BFS “interfere” with each other, i.e., all messages can be sent on time without congestion. Hence, all BFS stop at most \( D \) time slots after they were started. We conclude that the runtime of the algorithm is determined by the time \( \mathcal{O}(D) \) we need to build tree BFS, plus the time \( \mathcal{O}(n) \) that \( P \) needs to traverse BFS, plus the time \( \mathcal{O}(D) \) needed by the last BFS that \( P \) initiated. Since \( D \leq n \), this is all in \( \mathcal{O}(n) \).

Remarks:
- All of a sudden our algorithm needs \( \mathcal{O}(n) \) time, and possibly \( n \gg D \).
- Unfortunately not! One can show that computing the diameter of a network needs \( \Omega(n/\log n) \) time.
- Note that one can check whether a graph has diameter 1 by exchanging some specific information such as degree with the neighbors. However, already checking diameter 2 is difficult.

### 9.2 Lower Bound Graphs

We define a family \( G \) of graphs that we use to prove a lower bound on the rounds needed to compute the diameter. To simplify our analysis, we assume that \( (n - 2) \) can be divided by 8. We start by defining four sets of nodes, each consisting of \( q = q(n) := (n - 2)/4 \) nodes. Throughout this chapter we write \([q]\) as a short version of \([1, \ldots, q]\) and define:

\[
\begin{align*}
L_0 &:= \{l_i \mid i \in [q]\} \quad \text{// upper left in Figure 9.6} \\
L_1 &:= \{l_i' \mid i \in [q]\} \quad \text{// upper right} \\
R_0 &:= \{r_j \mid i \in [q]\} \quad \text{// lower left} \\
R_1 &:= \{r_j' \mid i \in [q]\} \quad \text{// lower right}
\end{align*}
\]

Figure 9.6: The above skeleton \( G' \) contains \( n = 10 \) nodes, such that \( q = 2 \).

We add node \( c_L \) and connect it to all nodes in \( L_0 \) and \( L_1 \). Then we add node \( c_R \), connected to all nodes in \( R_0 \) and \( R_1 \). Furthermore, nodes \( c_L \) and \( c_R \) are connected by an edge. For \( i \in [q] \) we connect \( l_i \) to \( r_j \). Also we add edges such that nodes in \( L_0 \) are a clique, nodes in \( L_1 \) are a clique, nodes in \( R_0 \) are a clique, and nodes in \( R_1 \) are a clique. The resulting graph is called \( G' \).

More formally skeleton \( G' = (V', E') \) is:

\[
V' := L_0 \cup L_1 \cup R_0 \cup R_1 \cup \{c_L, c_R\}
\]

\[
E' := \bigcup_{v \in L_0 \cup L_1} \{v, c_L\} \quad \text{// connections to } c_L
\]

\[
\bigcup_{v \in R_0 \cup R_1} \{v, c_R\} \quad \text{// connections to } c_R
\]

\[
\bigcup_{v \in L_0 \cup L_1} \{l_i, r_j\} \cup \{l_i', r_j'\} \quad \text{// connects left to right}
\]

\[
\bigcup_{S \in \{L_0, L_1, R_0, R_1\}} \{u, v\} \quad \text{// clique edges}
\]

To simplify our arguments, we partition \( G' \) into two parts. \textbf{Part L} is the subgraph induced by nodes \( L_0 \cup L_1 \cup \{c_L\} \). \textbf{Part R} is the subgraph induced by nodes \( R_0 \cup R_1 \cup \{c_R\} \).
Family $G$ contains any graph $G$ that is derived from $G'$ by adding any combination of edges of the form $(l_i, l_j')$ resp. $(r_i, r_j')$ with $l_i \in L_0, l_j' \in L_1, r_i \in R_0$, and $r_j' \in R_1$, and $r_j' \in R_1$.

Since in each round we can transmit $O(\log n)$ bits over each edge (in each direction), the bandwidth between Part L and Part R is $O(n \log n)$.

1. If we transmit the information of the $\Theta(n^2)$ edges in a naive way with a bandwidth of $O(n \log n)$, we need $\Omega(n/\log n)$ time. But maybe we can do better?!? Can an algorithm be smarter and only send the information that is really necessary to tell whether the diameter is 2?

2. It turns out that any algorithm needs $\Omega(n/\log n)$ rounds, since the information that is really necessary to tell that the diameter is larger than 2 contains basically $\Theta(n^2)$ bits.

9.3 Communication Complexity

To prove the last remark formally, we can use arguments from two-party communication complexity. This area essentially deals with a basic version of distributed computation: two parties are given some input each and want to solve a task on this input.

We consider two students (Alice and Bob) at two different universities connected by a communication channel (e.g., via email) and we assume this channel to be reliable. Now Alice and Bob want to check whether they received the same problem set for homework (we assume their professors are lazy and wrote it on the blackboard instead of putting a nicely prepared document online.) Do Alice and Bob really need to type the whole problem set into their emails? In a more formal way. Alice receives an $k$-bit string $x$ and Bob another $k$-bit string $y$, and the goal is for both of them to compute the equality function.

**Definition 9.10. (Equality.)** We define the equality function $EQ$ to be:

$$EQ(x, y) := \begin{cases} 1 & : x = y \\ 0 & : x \neq y \end{cases}$$
Remarks:

- In a more general setting, Alice and Bob are interested in computing a certain function \( f : \{0, 1\}^k \times \{0, 1\}^k \to \{0, 1\} \) with the least amount of communication between them. Of course they can always succeed by having Alice send her whole \( k \)-bit string to Bob, who then computes the function, but the idea here is to find clever ways of calculating \( f \) with less than \( k \) bits of communication. We measure how clever they can be as follows:

Definition 9.11. (Communication complexity \( CC \).) The communication complexity of protocol \( A \) for function \( f \) is \( CC(A, f) := \text{minimum number of bits exchanged between Alice and Bob in the worst case when using } A \). The communication complexity of \( f \) is \( CC(f) := \min CC(A, f) \) for all \( A \) solving \( f \). That is the minimal number of bits that the best protocol needs to send in the worst case.

Definition 9.12. For a given function \( f \), we define a \( 2^k \times 2^k \) matrix \( M_f \) representing \( f \). That is \( M_f := f(x, y) \).

Example 9.13. For EQ, in case \( k = 3 \), matrix \( M^\text{EQ} \) looks like this:

\[
\begin{pmatrix}
000 & 010 & 011 & 100 & 101 & 110 & 111 \\
000 & 1 & 0 & 0 & 0 & 0 & 0 \\
001 & 0 & 1 & 0 & 0 & 0 & 0 \\
011 & 0 & 0 & 1 & 0 & 0 & 0 \\
100 & 0 & 0 & 0 & 1 & 0 & 0 \\
101 & 0 & 0 & 0 & 0 & 1 & 0 \\
110 & 0 & 0 & 0 & 0 & 0 & 1 \\
111 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]

As a next step we define a (combinatorial) monochromatic rectangle. These are “submatrices” of \( M_f \) which contain the same entry.

Definition 9.14. (Monochromatic rectangle.) A set \( R \subseteq \{0, 1\}^k \times \{0, 1\}^k \) is called a monochromatic rectangle, if

- whenever \( (x_1, y_1) \in R \) and \( (x_2, y_2) \in R \) then \( (x_1, y_2) \in R \).
- there is a fixed \( z \) such that \( f(x, y) = z \) for all \( (x, y) \in R \).

Example 9.15. The first three of the following rectangles are monochromatic, the last one is not:

- \( R_1 = \{011\} \times \{011\} \)
- \( R_2 = \{011, 100, 110\} \times \{000, 001\} \)
- \( R_3 = \{000, 001\} \times \{011, 100, 110, 111\} \)
- \( R_4 = \{000, 001\} \times \{000, 001\} \)

Each time Alice and Bob exchange a bit, they can eliminate columns/rows of the matrix \( M_f \) and a combinatorial rectangle is left. They can stop communicating when this remaining rectangle is monochromatic. However, maybe there is a more efficient way to exchange information about a given bit string than just naively transmitting contained bits? In order to cover all possible ways of communication, we need the following definition:

9.3. COMMUNICATION COMPLEXITY

Definition 9.16. (fooling set.) A set \( S \subseteq \{0, 1\}^k \times \{0, 1\}^k \) fools \( f \) if there is a fixed \( z \) such that

- \( f(x, y) = z \) for each \( (x, y) \in S \).
- For any \( (x_1, y_1) \neq (x_2, y_2) \in S \), the rectangle \( \{x_1, x_2\} \times \{y_1, y_2\} \) is not monochromatic. Either \( f(x_1, y_1) \neq z \), \( f(x_2, y_2) \neq z \) or both \( \neq z \).

Example 9.17. Consider \( S = \{(000, 000), (001, 001)\} \). Take a look at the non-monochromatic rectangle \( R_4 \) in Example 9.15. Verify that \( S \) is indeed a fooling set for EQ!

Remarks:

- Can you find a larger fooling set for EQ?
- We assume that Alice and Bob take turns in sending a bit. This results in 2 possible actions (send 0/1) per round and in \( 2^k \) action patterns during a sequence of \( t \) rounds.

Lemma 9.18. If \( S \) is a fooling set for \( f \), then \( CC(f) = \Omega(\log |S|) \).

Proof. We prove the statement via contradiction: fix a protocol \( A \) and assume that it needs \( t < \log(|S|) \) rounds in the worst case. Then there are \( 2^t \) possible action patterns, with \( 2^k < |S| \). Hence for at least two elements of \( S \), let us call them \( (x_1, y_1), (x_2, y_2) \), protocol \( A \) produces the same action pattern \( P \). Naturally, the action pattern on the alternative inputs \( (x_1, y_2), (x_2, y_1) \) will be \( P \) as well, in the first round Alice and Bob have no information on the other party’s string and send the same bit that was sent in \( P \). Based on this, they determine the second bit to be exchanged, which will be the same as the second one in \( P \) since they cannot distinguish the cases. This continues for all \( t \) rounds. We conclude that after \( t \) rounds, Alice does not know whether Bob’s input is \( y_1 \) or \( y_2 \) and Bob does not know whether Alice’s input is \( x_1 \) or \( x_2 \). By the definition of fooling sets, either

- \( f(x_1, y_2) \neq f(x_1, y_1) \) in which case Alice (with input \( x_1 \)) does not know the solution yet,

or

- \( f(x_2, y_1) \neq f(x_1, y_1) \) in which case Bob (with input \( y_1 \)) does not know the solution yet.

This contradicts the assumption that \( A \) leads to a correct decision for all inputs after \( t \) rounds. Therefore at least \( \log(|S|) \) rounds are necessary.

Theorem 9.19. \( CC(\text{EQ}) = \Theta(k) \).

Proof. The set \( S := \{(x, x) \mid x \in \{0, 1\}^k\} \) fools EQ and has size \( 2^k \). Now apply Lemma 9.18.

Definition 9.20. Denote the negation of a string \( z \) by \( \overline{z} \) and by \( x \circ y \) the concatenation of strings \( x \) and \( y \).

Lemma 9.21. Let \( x, y \) be \( k \)-bit strings. Then \( x \neq y \) if and only if there is an index \( i \in \{1, 2, \ldots, k\} \) such that the \( i^{th} \) bit of \( x \circ \overline{y} \) and the \( i^{th} \) bit of \( \overline{x} \circ y \) are both 0.
9.3 COMMUNICATION COMPLEXITY

Part R in this round while executing A on G. Based on this Alice and Bob determine which messages would be sent in round two and so on. For each round simulated by Alice and Bob, they only need to communicate $O(n \log n)$ bits: $O(\log n)$ bits for each of $O(n)$ messages. Since $A$ makes a decision after $o(n/\log n)$ rounds, this yields a total communication of $o(n^2)$ bits. On the other hand, Lemma 9.19 states that to decide whether $x = y$, Alice and Bob need to communicate at least $\Omega\left(\frac{n}{\log n}\right) = \Omega(n^2)$ bits. A contradiction.

Remarks:

• Until now we only considered deterministic algorithms. Can one do better using randomness?

Algorithm 9.25 Randomized evaluation of EQ.

1. Alice and Bob use public randomness. That is they both have access to the same random bit string $z \in \{0, 1\}^k$.
2. Alice sends bit $a := \sum_{i \in [k]} x_i \cdot z_i \mod 2$ to Bob.
3. Bob sends bit $b := \sum_{i \in [k]} y_i \cdot z_i \mod 2$ to Alice.
4. If $a \neq b$ then
5. we know $x \neq y$.
6. end if

Lemma 9.26. If $x \neq y$, Algorithm 9.25 discovers $x \neq y$ with probability at least $1/2$.

Proof. Note that if $x = y$ we have $a = b$ for sure.

If $x \neq y$, Algorithm 9.25 may not reveal inequality. For instance, for $k = 2$, if $x = 01$, $y = 10$ and $z = 11$ we get $a = b = 1$. In general, let $I$ be the set of indices where $x_i \neq y_i$, i.e., $I := \{i \in [k] | x_i \neq y_i\}$. Since $x \neq y$, we know that $|I| > 0$. We have

$$|a - b| = \sum_{i \in I} z_i \mod 2,$$

and since all $z_i$ with $i \in I$ are random, we get that $a \neq b$ with probability at least $1/2$.

Remarks:

• By excluding the vector $z = \theta^k$ we can even get a discovery probability strictly larger than $1/2$.

• Repeating the Algorithm 9.25 with different random strings $z$, the error probability can be reduced arbitrarily.

• Does this imply that there is a fast randomized algorithm to determine the diameter? Unfortunately not!

• Sometimes public randomness is not available, but private randomness is. Here Alice has her own random string and Bob has his own random string. A modified version of Algorithm 9.25 also works with private randomness at the cost of the runtime.
• One can prove an $\Omega(n/\log n)$ lower bound for any randomized distributed algorithm that computes the diameter. To do so one considers the disjointness function $DISJ$ instead of equality. Here, Alice is given a subset $X \subseteq [k]$ and Bob is given a subset $Y \subseteq [k]$ and they need to determine whether $Y \cap X = \emptyset$. $(X$ and $Y$ can be represented by $k$-bit strings $x,y$). The reduction is similar as the one presented above but uses graph $G_{x \cap y}$ instead of $G_{x \cup y}$. However, the lower bound for the randomized communication complexity of $DISJ$ is more involved than the lower bound for $CC(EQ)$.

Since one can compute the diameter given a solution for $APSP$, an $\Omega(n/\log n)$ lower bound for $APSP$ is implied. As such, our simple Algorithm 9.3 is almost optimal!

• Many prominent functions allow for a low communication complexity. For instance, $CC(PARITY) = 2$. What is the Hamming distance (number of different entries) of two strings? It is known that $CC(HAM \geq d) = \Omega(d)$. Also, $CC$(decide whether "$HAM \geq k/2 + \sqrt{k}$" or "$HAM \leq k/2 - \sqrt{k}$") = $\Omega(k)$, even when using randomness. This problem is known as the Gap-Hamming-Distance.

• Lower bounds in communication complexity have many applications. Apart from getting lower bounds in distributed computing, one can also get lower bounds regarding circuit depth or query times for static data structures.

• In the distributed setting with limited bandwidth we showed that computing the diameter has about the same complexity as computing all pairs shortest paths. In contrast, in sequential computing, it is a major open problem whether the diameter can be computed faster than all pairs shortest paths. No nontrivial lower bounds are known, only that $\Omega(n^2)$ steps are needed – partly due to the fact that there can be $n^2$ edges/distances in a graph. On the other hand the currently best algorithm uses fast matrix multiplication and terminates after $O(n^{2.372})$ steps.

9.4 Distributed Complexity Theory

We conclude this chapter with a short overview on the main complexity classes of distributed message passing algorithms. Given a network with $n$ nodes and diameter $D$, we managed to establish a rich selection of upper and lower bounds regarding how much time it takes to solve or approximate a problem. Currently we know five main distributed complexity classes:

• Strictly local problems can be solved in constant $O(1)$ time, e.g., a constant approximation of a dominating set in a planar graph.

• Just a little bit slower are problems that can be solved in log-star $O(\log^* n)$ time, e.g., many combinatorial optimization problems in special graph classes such as growth bounded graphs. 3-coloring a ring takes $O(\log^* n)$.

• A large body of problems is polynomial (or pseudo-local), in the sense that they seem to be strictly local but are not, as they need $O(polylog n)$ time, e.g., the maximal independent set problem.

• There are problems which are global and need $O(D)$ time, e.g., to count the number of nodes in the network.

• Finally there are problems which need polynomial $O(poly n)$ time, even if the diameter $D$ is a constant, e.g., computing the diameter of the network.

Chapter Notes

The linear time algorithm for computing the diameter was discovered independently by [HW12, PRT12]. The presented matching lower bound is by Frischknecht et al. [FHW12], extending techniques by [DBK+11].

Due to its importance in network design, shortest path problems in general and the $APSP$ problem in particular were among the earliest studied problems in distributed computing. Developed algorithms were immediately used, e.g., as early as in 1969 in the ARPANET (see [Lyn96], p.506). Routing messages via shortest paths were extensively discussed to be beneficial in [Taj77, MS79, MRR80, SS80, CM82] and in many other papers. It is not surprising that there is a large body of problems is polynomial time, e.g., many combinatorial optimization problems in special graph classes such as growth bounded graphs. 3-coloring a ring takes $O(\log^* n)$.

Finally there are problems which need polynomial $O(poly n)$ time, even if the diameter $D$ is a constant, e.g., computing the diameter of the network.

Bibliography


