Opiii

February 7, 2025

Contents

1 Dynamic Networks

Dynamic graph networks are graph networks that change over time. Communication is in synchronous, asynchronous or semi-synchronous rounds. Additionally shared memory is possible. Network elements may be failure-free or failure-prone. A classical example are mobile ad-hoc networks. Those are temporary interconnection networks of mobile wireless nodes without a fixed infrastructure. Communication happens whenever mobile nodes come within the wireless range of each other.

Example 1.1. *In mobile ad hoc networks, one may want to colour the graph or maintain a routing mechanism for communication to any particular destination in the network.*

1.1 Almost constant message-passing vertex colouring in a tree

Let T be a tree network with n labelled vertices in $[n]$. Colouring the graph can be done in almost constant, i.e. in $\log^*$ time.

Definition 1.2. $\log^*(x)$ is defined as the number of log functions that need to be applied to x such that the result is at most 1. E.g. $\log^*(16) = 3$ and $\log^* 2^{65536} = 5$.

1. begin by rooting the tree at vertex 0. This defines an order on the tree
2. each parent sends its number to all of its children
3. each child computes the smallest index i where its number differs from the parent's number. It is important to note that this can be done in constant time with suitable hardware
4. It computes a new ID for itself consisting of a trailing bit corresponding to the bit where IDs disagreed. The new ID begins with the binary representation of the digit where the Ids differed.
5. the new ID is now only $\log \log n$ bits long. This is repeated until there are only six distinct numbers left. This takes $\log^*$ rounds each taking only constant time.
6. each parent sends its number to its children which relabel themselves accordingly
7. This is repeated another time and the IDs are taken $\mod 3$ resulting in a three colourin

Definition 1.3. The collection of the initial states of all nodes in the r-neighbourhood of a node v is the r-hop view of v.

Definition 1.4. Let $\mathcal{G}$ be a family of network graphs. The r-neighbourhood graph $N_r(\mathcal{G})$ is defined as follows:
The node set is the set of all possible labelled r-neighbourhoods (i.e. all possible r-hop views). There is an edge between tow labelled r-neighbourhoods V_r and V_r' if V_r and V_r' can be the r-hop views of adjacent nodes.

Lemma 1.5. *For a given family of network graphs $\mathcal{G}$ there is an r-round algorithm that colours graphs of $\mathcal{G}$ with c colours of the chromatic number of the neighbourhood graph is $\chi(N_r(\mathcal{G})) \leq c$.*

Definition 1.6. We define a directed graph B_k which is closely related to the neighbourhood graph. The vertex set is made up of all k-tuples consisting increasing node labels. For two nodes $\alpha = (\alpha_1, ..., \alpha_k)$ and $\beta = (\beta_1, ..., \beta_k)$ there is an edge from α to β if $\forall i$ it holds that $\beta_i = \alpha_{i+1}$.

Lemma 1.7. *Viewed as an undirected graph, B_{2r+1} is a subgraph of the r-neighbourhood graph of directed rings with n nodes.*

Lemma 1.8. *If $n > k$ the graph B_{k+1} can be defined as the line graph $\mathcal{L}(B_k)$ of B_k.*

Lemma 1.9. *It holds that*

$$\chi(\mathcal{L}(G)) \geq \log_2(\chi(G))$$

Lemma 1.10. *For all $n \geq 1$ it holds that $\chi(B_1) = n$. Further for $n \geq k \geq 2$ it holds that $\chi(B_k) \geq \log^{(k-1)} n$.*

Theorem 1.11. *Every deterministic distributed algorithm to colour a directed ring with at most 3 colours needs at least $\log^*(\frac{n}{2}) - 1$ rounds.*

Corollary 1.12. *Every deterministic distributed algorithm to compute a maximal independent set on a directed ring needs at least $\log^*(\frac{n}{2}) - \mathcal{O}(1)$ rounds.*

1.2 MIS

The following randomized algorithm gives a good solution to the maximum independent set.

1. the algorithm operates in synchronous rounds grouped into phases
2. each node marks itself with probability $\frac{1}{2d(v)}$
3. if no higher degree neighbour of v is marked, node v unmarks itself again

4. delete all nodes that joined the MIS and their neighbours as the cannot join the MIS any more

Lemma 1.13. *A node v joins the MIS in step 3 with probability $p \geq \frac{1}{4d(v)}$*

Lemma 1.14. *A node is called good if*

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

A good node will be removed in Step 4 with probability $p \geq \frac{1}{36}$.

Lemma 1.15. *An edge is called bad if both its endvertices are bad. Otherwise it's called good. At any time at least half of the edges are good.*

Lemma 1.16. *A bad node has out-degree at least twice its in-degree.*

Lemma 1.17. *The algorithm terminates in expectation in $\mathcal{O}(\log n)$ rounds.*

2 Consensus

In a distributed system with each node starting with input x_i, we speak of consensus if an algorithm can achieve the following properties

1. Agreement: all alive nodes decide on a single value x
2. Validity: the decided value x is one of the initial inputs
3. Termination: each vertex terminates at some point (either voting for one value or crashing)

The following randomized consensus algorithm works in an asynchronous setting with less than half the nodes crashing

1. input bit $v_i \in \{0, 1\}$, $round = 1$, decided = false
2. broadcast $(v_i, round)$
3. while true
4. wait until majority of messages of current round arrived
5. if all messages contain the same value v:
6. propose $(v, round)$, decided = true

```
7. else:
8. propose $(\perp, round)$ //$\perp$ is a signal of disagreement
9. end if
10. wait until a majority of proposals of current round arrived
11. if all messages propose the same value $v$:
12. $v_i = v$, decide = true
13. else if there is at least one proposal for $v$:
14. $v_i = v$
15. else:
16. choose $v_i$ uniformly at random
17. end if
18. $round = round + 1$
19. broadcast $(v_i, round)$
20. end while
```

Theorem 2.1. *The above algorithm satisfies validity, termination and comes to an agreement. In expectation it takes exponential time.*

2.1 shared coin

The following algorithm allows a dynamic network to use the same coin for all vertices at the same time. Here f is the number of nodes that can turn byzantine. It should hold that $f \leq \frac{n}{3}$.

```
1. choose local coin $c_u = 0$ with probability $\frac{1}{n}$
2. broadcast $c_u$
3. wait for $n - f$ coins and store them in the local coin set $C_u$
4. broadcast $C_u$
5. wait for $n - f$ coin sets
```

```
6. if at least one coin is 0 among all coins in $C_u$:
7. return 0
8. return 1
9. end if
```

2.2 byzantine consensus

Definition 2.2. A node which can have arbitrary or malicious behaviour is called byzantine. This includes not sending messages, sending wrong messages, sending different messages to different neighbours and many more. A node that is not byzantine is called correct or truthful.

The following probabilistic algorithm achieves consensus in an asynchronous setting with $< \frac{n}{9}$ byzantine nodes.

```
1. $x_i \in \{0,1\}$, $r = 1$, decided = false
2. propose($x_i, r$)
3. while not decided
4. wait until $n - f$ proposals of current round $r$ arrived
5. if at least $n - 2f$ proposals contain the same value $x$: $x_i = x$ decided = true
6. elseif at least $n - 4f$ proposals contain the same value $x$: $x_i = x$
7. else: choose $x_i$ randomly with $\mathbb{P}[x_i = 0] = \mathbb{P}[x_i = 1] = \frac{1}{2}$
8. endif
9. $r = r + 1$, propose($x_i, r$)
10. endwhile
11. decision = $x_i$
```

Lemma 2.3. *Let $f < \frac{n}{9}$. If a correct node chooses value x in line 6, then no other correct node chooses value $y \neq x$ in line 6.*

Theorem 2.4. *The algorithm solves binary agreement for up to $f < \frac{n}{9}$ byzantine nodes.*

Definition 2.5. $N[u] = N(u) \cup \{u\}$

3 Dominating Set

The following algorithm gives an approximation to a minimal dominating set. To this end, we colour vertices white in the beginning, black if they are added to the dominating set S and grey if they are covered by a neighbour in S. For a vertex u we define $W(u) = \{v \in N[u] \mid v \text{ is white}\}$.

```
1. while $v$ has white neighbours
2. compute $|W(v)|$ and send it to all neighbours at distance at most 2
3. if $|W(v)|$ is largest among neighbours of distance 2
4. join $S$
5. endif
6. endwhile
```

Theorem 3.1. *Let S^* be the optimal dominating set and S the one returned by the algorithm. Then $\frac{|S|}{|S^*|} \leq \ln \Delta + 2$. The algorithm takes $\Theta(n)$ rounds.*

In the following we try to push this runtime to sublinear.

3.1 Fast Dominating Set Algorithm

```
1. $W(v) = N[v]$, $w(v) = |W(v)|$
2. while $W(v) \neq \varnothing$
3. $w'(v) = w(v)$ rounded down to the nearest power of 2
4. if $w(v) = \max_{u \in N_2(v)} w'(u)$ then $v.active = true$
5. else $v.active = false$
6. endif
7. compute active neighbours $a(v) = \{u \in N(v) \mid u.active\}$
8. $v.candidate = false$
9. if $v.active = true$ then
10. $v.candidate = true$ with probability $\frac{1}{\max_{u \in W(v)} a(u)}$
```

```
11. endif
12. compute $c(v) = |\{u \in W(v) \mid u.candidate\}|$
13. if $v.candidate$ and $\sum_{u \in W(v)} c(u) \leq 3w(v)$ then
14. node $v$ joins dominating set
15. endif
16. update $W$, $w$
17. endwhile
```

Theorem 3.2. *The algorithm computes a dominating set of size at most* $(6 \ln \Delta + 12)\,|S^*|$.

Lemma 3.3. *Consider an iteration of the while loop. Suppose that a node u is white and that $2a(u) \geq \max_{v \in C(u)} \max_{y \in W(y)} a(y)$ where*

$$C(u) = \{v \in N(u) \mid v.candidate\}$$

Then the probability that u becomes dominated in this iteration is larger than $\frac{1}{9}$.

4 Maximal matching

This section introduces a new technique called rounding. The idea is to solve a given integral problem as a continuous problem and then rounding the results to the nearest integer. This often allows for polylogarithmic time complexity.

Definition 4.1. A maximal matching is a subset S of edges s.t. no vertex has two incident edges in S. Furthermore, there are no edges $e \in E \setminus S$ that can be added to S without breaking the first condition.

This problem is a typical integer linear problem, i.e. one where variables x_e for $e \in E$ are in $\{0, 1\}$. The idea is now to allow continuous variables and fix the result by rounding. A non-integral result is called a fractional matching.

Definition 4.2. In a fractional matching we call a vertex v loose if $c_v = \sum_{e \in E(v)} x_e \leq \frac{1}{2}$. An edge is called loose if both its vertices are loose. We call a fractional matching f-fraction if $c_v \geq f \ \forall v \in V$.

The following algorithm runs in $\mathcal{O}(\log n)$ time and yields a 4-approximation to for a fractional matching.

```
1. $x_e = 2^{-(\lceil \log \Delta \rceil)}$
2. while both endpoints of $e$ are loose:
3. $x_e = 2x_e$
4. endwhile
```

Theorem 4.3. *The algorithm computes a 4-approximation $\frac{1}{\Delta}$-fractional matching in $\mathcal{O}(\log \Delta)$ time.*

The idea is now to get rid of the fractional edges. This works by starting to either multiply all edges of value $\frac{1}{\Delta}$ by a factor of 2 or by rounding them down to 0. In the next step this is done for edges of value $\frac{2}{\Delta}$ and so on.

Definition 4.4. We define the subgraph G_f as the graph induced by all edges of value f in G.

Definition 4.5. Rounding the graph G_f means to identify a subset of edges of value f in E_f which will be doubled. All other edges in (which are also of value f!) will be assigned value 0. The resulting graph should still be a valid $2f$-fractional matching.

Definition 4.6. A perfect rounding of a graph G is a rounding of the graph such that for all nodes v half of its edges are assigned twice its value and the other half 0. Notice that c_v and the size of the matching remain unchanged.

We start by introducing the idea for bipartite graphs. For this, we need to construct a 2-decomposition of the graph G_f.

Definition 4.7. For a graph G we define a decomposition G' of G by copying each vertex v $\frac{d(v)}{2}$ times. The edges incident to v are distributed among all these copies such that each copy has degree 2 (in the case that $d(v)$ is odd, one copy may have degree 1).

Note that in G' each vertex v has $d(v) \in \{1, 2\}$. I.e. G' is a disjoint union of cycles and paths.

Definition 4.8. A cycle or path is called short if its length is at most $l = 24 \log \Delta$ and long otherwise.

Since we assumed that the starting graph is bipartite, all cycles are even. Therefore we would want the edge values to be raised and dropped alternately along the cycle. This is a perfect rounding. If the cycle is short, the following algorithm achieves this in $\mathcal{O}(\log \Delta)$.

1. Orient the cycle in one direction
2. if e goes from a node with colour 1 to colour 2:
3. $x_e := 2x_e$
4. else: $x_e := 0$

In a long cycle the first line is not easy to solve. Therefore, we contend ourselves with a common direction only on subpaths of length at least l on long cycles.

Definition 4.9. Consider a cycle with an orientation of each edge. A maximal directed path is a directed path that can not be extended since both neighbouring edges have inconsistent orientation.

Starting from a random orientation we can achieve long directed paths by determining the length of a subpath and compute the length of the subpath it points towards. By flipping the edges of the shorter path, we create a longer directed path.

1. orient e arbitrarily
2. for $i = 1, ..., \log(l)$:
3. compute the length of the path pointing in the opposite direction and flip the edges of the shorter path
4. endfor

Since we can now compute long subpaths of long cycles, we discuss rounding of long cycles in the next step.

1. compute long paths
2. if e is a boundary edge or goes from a node of colour 2 to colour 1: $x_e := 0$
3. else: $x_e := 2x_e$

Lemma 4.10. *Rounding long cycles leads to a loss of* $\leq \frac{3}{l} \sum_{e \in E} x_e$.

Obviously the same approach works for long paths.
The algorithm for short paths is a bit more complicated.

1. orient the graph with start node s and end note t
2. if e is first edge:

```
3. if $s$ is tight (*not loose*): $x_e := 0$
4. else: $x_e := 2x_e$
5. else if $e$ is the last edge:
6. if $t$ is tight: $x_e := 0$
7. else: $x_e := 2x_e$
8. else if $e$ is an even edge: $x_e := 0$
9. else: $x_e := 2x_e$
```

Lemma 4.11. *Rounding short paths results in a loss* $\leq 4f \sum_{e \in E} x_e$

Lemma 4.12. *In the rounding step going from an f-fractional matching to a $2f$-fractional matching the matching decreases by a factor of at most* $(1 - \frac{3}{l} - 4f)$ *and the rounding step takes* $\mathcal{O}(\Delta)$ *time.*

Lemma 4.13. *This results in a $\frac{1}{16}$-fractional matching which is a constant factor smaller than the initial $\frac{1}{\Delta}$ matching.*

Lemma 4.14. *A constant factor approximation 16-fractional matching can be computed in $\mathcal{O}(\log^2 \Delta)$ in a 2-coloured bipartite graph.*

Lemma 4.15. *A constant factor approximation matching can be computed in $\mathcal{O}(\log^2 \Delta)$ time in a 2-coloured bipartite graph with maximum degree Δ.*

What can we say about general graphs?
The idea is to decompose the graph in a similar way to how it was done in the exercises. Assuming that the vertices are labelled, we can direct the edges from smaller to larger ID. Then each vertex is copied once. The first copy only keeps in-going edges, while the other copy only keeps out-going edges. It is clear that this construction generates a bipartite graph. This motivates the following definition.

Definition 4.16. The previous construction is called a bipartite double cover. The bipartition classes are obviously the set of vertices with in-edges or out-edges respectively. Since a vertex has only in-neighbours or only out-neighbours, it is easy to assign a 2-colouring.

Since we can compute a c-approximation matching for bipartite graphs, the following lemma follows quickly.

Lemma 4.17. *A 3c-approximation matching can be computed in* $\mathcal{O}(\log^* n + \log^2 \Delta)$ *time in general graphs.*

By repeatedly applying this lemma, we can even prove the following:

Theorem 4.18. *A maximal matching can be computed in* $\mathcal{O}((\log^* n + \log^2 \Delta) \log n)$ *time.*

5 Wireless networks

In a wireless network nodes can send messages only to nodes within a given range. Even if we assume that the transmission graph is complete, multiple nodes sending messages at the same time can lead to interference. Therefore we want that in each round all but one vertex only receive messages while the remaining one is free to communicate. How can we guarantee message transmissions? Communication is possible if a vertex of ID j transmits a message in round j but this needs a linear number of rounds. Trough randomization this can be improved substantially.

5.1 Wireless leader election

1. In the first phase, the nodes transmit with probability $\frac{1}{2^{2^0}}, \frac{1}{2^{2^1}}, \frac{1}{2^{2^2}}, \ldots$ until no node transmits which yields a first approximation of the number of nodes
2. in the second phase, a binary search is performed to determine an even better approximation of n. The first phase returned the search window $[l, u]$ in which the number of vertices must lie.
3. in the third phase we use a biased random walk to give a final approximation of n. The second phase returned an approximation d of the number of nodes. With probability $\frac{1}{2^d}$ each node now communicates. If nothing was transmitted, decrease d and if more than one node communicated, increase d until exactly one message was sent.

Let X denote the random variable representing the number of nodes transmitting in the same time slot.

Lemma 5.1. *During phase 2, if the current approximation j of n is larger than* $\log n + \log\log n$ *or if during the first phase the current approximation i is larger than* $2 \log n$ *it holds that* $\mathbb{P}[X > 1] \leq \frac{1}{\log n}$.

Lemma 5.2. *If* $j < \log n - \log\log n$ *or* $i < \frac{1}{2}\log n$ *then* $\mathbb{P}[X = 0] \geq \frac{1}{n}$

Lemma 5.3. *Let v be such that $2v - 1 \leq n \leq 2v$. If during the third phase the current approximation d satisfies $d > v + 2$ then $\mathbb{P}[X > 1] \leq \frac{1}{4}$.*

Lemma 5.4. *If $d < v - 2$ then $\mathbb{P}[X = 0] \leq \frac{1}{4}$*

Lemma 5.5. *If $v - 2 \leq d \leq v + 2$ then $\mathbb{P}[X = 1]$ is constant.*

Lemma 5.6. *With probability $1 - \frac{1}{\log n}$ we find a leader in phase 3 in $\mathcal{O}(\log \log n)$ time.*

Theorem 5.7. *The algorithm elects a leader with probability of at least $1 - \frac{\log \log n}{\log n}$ in $\mathcal{O}(\log \log n)$ time.*

5.2 Advice Algorithms

In an asynchronous deterministic setting with at least one crash it is not possible to solve consensus. In the following we will see that in a deterministic setting that allows for advice we can achieve different results.
There are two possibilities for consensus. In the first case we have advice from the beginning on. That means, we always have some kind of trustworthy information. In case two, we deal with eventual advice. That means there will be a point in time at which there is secure information on which we can fall back on.
The following chapter deals with an asynchronous message-passing system. At most f processes may fail by halting. Each process has access to a failure detector. This is an unreliable oracle that gives the vertex continuous updates about the status of other processes. For each node the oracle can tell its process if it suspects the node to have crashed. If not, that node is considered trusted. A correct process never fails and a non-faulty one may fail but has not as of this point in time. A variant of this are limited-scope failure detectors. These can only give an answer to process that are "reachable" what ever that may mean.

Definition 5.8 (configuration)**.** We say that a system is fully defined at any point in time by its configuration C. This includes the state of every node and all messages that are in transit.

Definition 5.9 (univalent)**.** We call a configuration C univalent if the decision value (think consensus) is determined independently of what happens later on.

Remark 5.10.

- a configuration C that is univalent with value v is called v-valent
- C can be univalent even though no node knows about it

Definition 5.11. A configuration C that is not univalent is called bivalent (assuming that the decision space is $\{0, 1\}$).

Lemma 5.12. *There is at least one selection of input vales such that the according initial configuration C_0 is bivalent if the number of crashes is $f \geq 1$.*

Definition 5.13 (transition)**.** A transition from a configuration C to a following configuration C_τ is characterized by an event $\tau = (u, m)$ meaning that node u receives message m.

Remark 5.14.

- A transition $\tau = (u, m)$ is only applicable to configuration C if m was still in transit in C.
- The difference between C and C_τ is that in C_τ u might have a different state, m is no longer in transit and there are potentially new messages in transit sent from u.

Remark 5.15.

1. the set of configurations can be viewed as vertices of a graph and the transitions can be viewed as edges. The resulting graph is the so-called transition tree
2. leaves are configurations where the execution terminates

Lemma 5.16. *Assume two transitions τ_1, τ_2 for $u_1 \neq u_2$ are both applicable to to C. Let $C_{\tau_1\tau_2}$ be the configuration that follows C by first applying τ_1 and then τ_2 and define $C_{\tau_2\tau_1}$ analogously. Then $C_{\tau_1\tau_2} = C_{\tau_2\tau_1}$.*

Definition 5.17 (critical configuration)**.** We say that a configuration C is critical, if C is bivalent but all configurations that are direct children of C in the configuration tree are univalent.

Lemma 5.18. *In a system with bivalent configurations, it has to reach a critical point in finite time otherwise it does not reach consensus.*

Lemma 5.19. *If a configuration tree contains a critical configuration, crashing a single node can create a bivalent leaf, i.e. a crash prevents the algorithm from reaching an agreement.*

Theorem 5.20. *There is no deterministic algorithm which always achieves consensus in the asynchronous model with $f > 0$.*

Here, we are considering a system that clusters the vertices. Each cluster includes one correct process that is never suspected to have failed by any other process. However, that may only be the case after a time period t. A model with q clusters can be understood as a network composed of w disjoint LANs.

Definition 5.21 (strong completeness)**.** At some point every crashed participant will be suspected to be permanently dead.

Definition 5.22 (k-set agreement)**.** In a network where every vertex starts with an initial value that is not necessarily binary, the vertices should agree on at most k different values. For $k = 1$ this is simply consensus.

The solvability of k-set agreement now depends on q, k and $x = \left|\bigcup_{j=1}^{Q} q_j\right|$ the total size of all clusters. This system satisfies the above strong completeness and one of the following two weak accuracies.

Definition 5.23 (perpetual weak (x, q)-accuracy)**.** Some correct process in each cluster is never suspected by any process in that cluster.

Definition 5.24 (eventual weak (x, y)-accuracy)**.** There is a time after which some correct process in each cluster is never suspected by any process in that cluster.

We focus on two different classes of failure detection:

- $S_{x,q}$: strong completeness and perpetual weak (x, q)-accuracy
- $\diamond S_{x,q}$: strong completeness ad eventual weak (x, q)-accuracy

Under the first system, it is possible to solve k-set agreement with up to $f < k - 1 + x - q$ failures if $q < k$ and $f < x$ otherwise. Under the second system however, $f < \min\{\lceil \frac{n+1}{2} \rceil, k - 1 + x - q\}$ failures can be tolerated.

5.3 <u>k-set-agreement with advice</u>

Definition 5.25. Suppose each vertex i starts with a value x_i. A solution to the k-set agreement problem satisfies

- <u>agreement</u>: all vertices agree on at most k many distinct values
- <u>all-same-validity</u>: the k values must be a subset of the initial values
- <u>termination</u>: every correct node eventually decides

The following algorithm works for th $S_{x,q}$ failure detector.

```
1. $r = 1; s = 0; e = x_i; ifq \leq k, b = \min\{n, k - 1 + x - q\} elseb = \min\{n, x - 1\}$
2. for each set $S$ of size $b$ do
3. if $i \in S$
4. for each $X \subset S$ of size $\min\{k, q\}$ do
5. if $i \in X$ then $\forall p \in S$ broadcast $(e, r, s)$
```

```
6. else $e = e'$ from $(r, s)$-broadcast where $e'$ is from $p \in X$ not suspected
7. $s = s + 1$
8. endfor
9. broadcast $(e, r)$
10. else $e = e'$ from some other broadcast
11. $r = r + 1$
12. endfor
```

Theorem 5.26. *This algorithm solves k-set agreement in an asynchronous setting with a failure tolerance of up to $f < \min\{n, k - 1 + x - q\}$ if $q \leq k$ and $f < \min\{n, x - 1\}$ otherwise.*

The following algorithm can solve k-set-agreement with $\diamond S_{x,q}$ fault detection.

```
1. $r = 1$, $e = x_i$
2. while TRUE do
3. $h = \varnothing$
4. $e = S_{x,q}(id, e)$
5. for all permutations $t$ of $[n]$ do
6. broadcast $(r, t, i, e, h)$, $r = r + 1$
7. define $M, S, H$ as the sets of messages, participants and histories of the first $\lceil \frac{n+1}{2} \rceil$
   messages received
8. $e =$ message received by the participant in the first permutation $t$ of $[n]$
9. $E =$ estimates of $S$ from last round and last permutation in $H$
10. if $0 < |E| \leq k$ broadcast $e$, return $e$
11. if $e'$ received, then $e = e'$ and broadcast $e$, return $e$
12. endfor
13. endwhile
```

Theorem 5.27. *This algorithm solves k-set-agreement asynchronously while tolerating up to $fy \min\{\frac{n}{2}, k - 1 + x - q\}$ if $q \leq k$ and $f < \min\{\frac{n}{2}, x - 1\}$ otherwise.*

Definition 5.28. An n-simplex S is a space spanned by a collection of $n+1$ affine independent points $\{v_1, ..., v_{n+1}\}$. That is

$$S = \left\{ \sum_{i=1}^{n+1} c_i v_i \mid \sum_{i=1}^{n+1} c_i = 1,\ c_i \geq 0\ \forall i \right\}$$

A simplicial subdivision of S is a partition of S into smaller sub-simplices such that two of them are either disjoint or share a full face. Finally, a Sperner-colouring of a simplicial subdivision of S is one that assigns different colours to all of the initial $n+1$ vertices of S and assigns each subdivision vertex of a face F of S one of the colours of the boundary vertices of F. There are no restrictions on the colours of internal vertices.

Lemma 5.29. *A simplicial complex of an n-simplex S as in the previous definition together with a Sperner-colouring always admits a simplicial cell with $n+1$ differently coloured vertices.*

6 Sorting and Counting Networks

Definition 6.1. A comparator is a device with two inputs, x and y and two outputs x', y' such that $x' = \min\{x, y\}$ and $y' = \max\{x, y\}$. We construct so-called comparison networks that consist of wires that connect comparators.

Definition 6.2. The depth of an input wire is 0. Teh depth of a comparator is the maximum depth of its input wires plus one. The depth of an output wire is the depth of the previous comparator. The depth of a comparison network is the maximum depth of the output wires.

Definition 6.3. A bitonic sequence s a sequence of numbers that first monotonically increases and then monotonically decreases or vice versa.

Definition 6.4 (half-cleaner)**.** A half-cleaner is a comparison network of depth 1 that compares wire i with wire $i + \frac{n}{2}$.

In the following, we will consider binary bitonic sequences, that is of the form $0^j 1^k 0^l$ or $1^j 0^k 1^l$.

Lemma 6.5. *Feeding a bitonic sequence into a half-cleaner will clean either the upper or the lower half of the n-wires. That is, it makes half the sequence all 0 (or all 1 respectively). The other half is bitonic.*

Definition 6.6 (Bitonic sequence sorter)**.** A bitonic sequence sorter of width $n(= 2^k)$ consists of a half cleaner of width n and then two bitonic sequence sorters of width $\frac{n}{2}$ each.

Lemma 6.7. *A bitonic sequence sorter of width n has depth $\log n$.*

In order to sort arbitrary sequences, we now need to introduce another concept, called merging networks.

Definition 6.8 (merger)**.** A merger is a network of depth 1 that compares wire i with wire $n - i + 1$.

Definition 6.9. A merging network is a merger of depth 1 followed by two bitonic sequence sorters of depth $\frac{n}{2}$ each. That is, it is a bitonic sequence sorter where we replace the first half-cleaner by a merger.

Lemma 6.10. *A merging network of depth n merges two sorted input sequences of length $\frac{n}{2}$ each into one sorted sequence of length n.*

It should now be obvious that applying the previous lemma recursively allows us to fully sort any sequence.

Definition 6.11 (Batcher's network)**.** A Batcher's sorting network of width n consists of two Batcher's sorting networks of width $\frac{n}{2}$ followed by a merger. A network of width 1 is empty.

Lemma 6.12. *In order to sort a sequence of length n we need a sorting network of depth $\mathcal{O}(\log^2 n)$.*

7 Minimum Spanning Tree

The idea is to use an approach similar to Kruskal. We start with n components. In each round we can merge components by picking the smallest edge leaving a component. This however, may create very large components along the way. Therefore, it would be good if we were able to restrict the size of components to $\sqrt{n}$. This ensures that communication is sublinear in each round. In a general setting there can be at most $\sqrt{n}$ components of size more than $\sqrt{n}$ so this is quite the challenge. As usual, the key is randomization.

7.1 Aggregation Problem

This is a different kind of distributed problem that allows for a reduction from MST. Thus, by solving this problem in a distributed setting, we can also compute an MST.

Definition 7.1. In the problem formulation, there are N so-called aggregation messages and M machines that can store at most S such messages. That means in each round, a machine M_i can not send or receive more than S messages. Further, $M \in \mathcal{O}(\frac{N}{S} \log_S N)$. An aggregation message m is also given a value v_m, a target machine t_m and a group g_m. Messages of one group all have the same target machine and a machine M_i is target of at most one group. A solution to the problem is reached if every target machine has received a message of minimum value from its group.

How does this translate to MST?
The idea is still to form clusters and merge them in the best way (i.e. using minimum weighted edges). We therefore put each machine in charge of handling one of the already computed minimum spanning tree clusters. We need to achieve the following: Each edge is stored by some machine that needs to figure out, which two (not necessarily different) clusters the endpoints of this edge lie in and each machine responsible for a cluster needs to find the edge of minimal weight leaving the cluster.
For each cluster c an aggregation group is created whose target machine is the one responsible for c. Each machine storing an edge with exactly one endpoint in x sends a message to the target machine containing some identifier of the edge and its weight (the aggregation value). The target machine know the minimal edge leaving x at the end of this phase and can merge with the respective cluster x'. It then broadcasts the creation of the new cluster $x \cup x'$ and the next phase works similarly.
Given that $N = \mathcal{O}(n^2)$ and that $S = \mathcal{O}(n^c)$ for some constant c we get that $\mathcal{O}(\log_S N) = \mathcal{O}(\frac{1}{c}) = \mathcal{O}(1)$ if the runtime to solve the aggregation problem and to do the broadcast through the tree is $\mathcal{O}(\log_S N)$. This results in a runtime of $\mathcal{O}(\log n)$ for MST.
It remains to show the runtime of the aggregation problem. We aim to find an algorithm that works in $\mathcal{O}(\log_S N)$ time and that in expectation does not require any machine to send or receive more than $\frac{S}{2}$ messages per round. We use advice to solve this. Thus, assume that there is a sufficiently long string of random bits that is accessible to all machines.
We partition the machines into different levels each of size $2\lceil\frac{N}{S}\rfloor+2$ machines which implies that there are $\mathcal{O}(\log_S N)$ levels in total. It is further assumed that the machines initially storing the aggregation messages are in the bottom-most level while the target machines are placed at the top.
Every machine in the lowest level L_1 belonging to an aggregation group g now picks a

random subset $L_{2,g}$ of machines from the next level L_2. This subset should be of size $N\left(\frac{S}{2}\right)^2$ and should be the same for all machines in g. This is possible via utilization of the advice string. Note that a machine can be part of multiple groups depending on the aggregation messages it holds. From $L_{2,g}$ each machine from group g on level 1 then picks a parent machine (which is no longer the same for all machines in g). At this point, each machine now has a parent machine from the next level for each group it participates in. A machine on level i then belongs to g if it an element of $L_{i,g}$.
We inductively repeat this construction. Machines in the set $L_{i,g}$ also pick a random subset of size $N\left(\frac{S}{2}\right)^{i+1}$ from level $i+1$ and then proceed to choose a parent independently and uniformly at random from there. If $i+1=l$ the last level, it is only necessary to connect machines of group g to their respective target machine. In each round, a machine of level i sends the minimum valued aggregation message m it knows of to its parent on level $i+1$.
This implies the correct running time and since at each level the global minimum message is relayed, it will surely reach its target proving correctness. Finally since the probability that a machine on level i belongs to $L_{i,g}$ for a given g is $\frac{2^{i-1}}{S^{i-1}} \geq \frac{|L_{i,g}|}{|L_i|}$ while the probability that a machine from L_i is chosen as the parent of a fixed machine is $\frac{1}{L_{i,g}}$. Thus the probability that a machine is chosen as parent by a fixed machine from L_{i-1} is at most $\frac{S}{2N}$. Therefore, a machine will be the parent of at most $\frac{S}{2}$ machines thus respecting the restriction on the storage space of the machines.

8 Dynamic networks

In the following we are working with a dynamic network graph $G=(V,E)$ where V is a static set of vertices as usual and where $E:\mathbb{N}\to\mathcal{P}\left(\binom{V}{2}\right)$ is a function mapping the round number to a set of edges.

Definition 8.1. A dynamic graph G is called T-interval connected for $T\in\mathbb{N}$ if for all $r\in\mathbb{N}$ the static graph $G_{r,T}=\left(V,\bigcap_{i=r}^{r+T-1}E(i)\right)$ is connected. If G is 1-interval connected we say that G is always connected.

Theorem 8.2. *Suppose one vertex of a graph wants to broadcast a message. In a 1-interval connected graph G, it is obvious that after r failure-free communication rounds, at least $r+1$ vertices are aware of the message. Thus, after at most $n-1$ rounds, the broadcast is complete.*

This lays the foundation for a counting algorithm. The goal is that one vertex initiates a count of the number of vertices. For vertex v this algorithm achieves that

```
1. $r = 0, A = \{v\}$
2. repeat
3. $r = r + 1$, reliably broadcast $A$
4. receive id sets $A_1, ..., A_j$ from neighbours
5. $A = A \cup \bigcup_{k=1}^{j} A_k$
6. if $|A| \leq r$ output $|A|$
7. until $|A| \leq r$
```

Theorem 8.3. *In a 1-interval connected failure-free dynamic graph, this algorithm terminates and correctly outputs n in round $r = n$.*

Theorem 8.4. *Asynchronous counting is impossible in 1-interval connected graphs.*

It is rather obvious that 1-interval connectivity is the hard case. If the dynamic graph is T-interval connected for a $T > 1$, there is more information about successive rounds and so the algorithm can be sped up.
The following deals with the search for a black hole in a network. For this we interpret the dynamic graph as a street network that can be traversed by so-called agents. An agent located in a vertex v in round r has the decision to move to a neighbouring node u in round $r + 1$ or to stay at v. Since we are dealing with dynamic graphs however, an edge $e = (u, v)$ may exist in round r but is removed for round $r + 1$. If the agent wanted to move through e but the adversary removed it for the following round, the agent has to stay in vertex v. A byzantine vertex b which agents can enter but never leave is called a black hole. We are now interested in finding a black hole in a dynamic graph. Specifically, we want to design algorithms that can locate a black hole in the shortest possible time while needing as few agents as possible.

8.1 black hole detection

Definition 8.5. If at the start of the black hole search all agents are located in a common vertex v, we say they are co-located.

Lemma 8.6. *Two co-located agents can not find the black hole in a 1-interval connected cycle.*

Lemma 8.7. *Three co-located agents can find the black hole in a 1-interval connected cycle in $\mathcal{O}(n^2)$ time.*

Definition 8.8. A torus-graph is a $n \times n$ grid graph where the sides of the grid are identified with one another in the same way as in the topological construction of a torus. Let $G_{n\times n}$ be a grid graph on n^2 vertices and let T_n be the corresponding torus graph. T_n can be constructed from $G_{n\times n}$ by adding edges $(v_{1,j}, v_{n,j})$ and $(v_{i,1}, v_{i,n})$ between vertices in the top and bottom row (reps. left-most and right-most column). T_n consists of n row and n column cycles.

Lemma 8.9. *A number of $n+2$ agents that are initially co-located ca find a single black hole in a 1-interval connected torus graph.*

Switching back to static graphs, we can achieve better results.

Theorem 8.10. *Two co-located agents can find the black hole asynchronously in a 2-connected static graph of known topology in $\mathcal{O}(n \log n)$.*

8.2 Gatherings in 1-interval connected graphs

Definition 8.11. In the gathering problem a set of k agents is initially placed arbitrarily on different vertices of a graph and they have to meet at one node within finite time.

Definition 8.12. In the weak gathering problem we also allow agents to meet at different ends of one edge and not necessarily in the same node.

Lemma 8.13. *Let G be a 1-interval connected graph containing exactly one cycle of size $|C| > 3$. Then it is impossible to solve the gathering problem.*

Lemma 8.14. *Let G be a 1-interval connected graph containing at least two cycles. Then the weak gathering problem is impossible.*

By inspection it should be quite obvious that the weak gathering problem is solvable on unicyclic graphs. Note that we work under the assumptions that all agents know the number of nodes n and the number of agents k. The first assumption is not actually necessary but the second one is:

Theorem 8.15. *If k is unknown, the weak gathering problem is unsolvable.*

Lemma 8.16. *There is a $\mathcal{O}(n^3 \log n)$ two-phase algorithm solving the weak gathering problem in unicyclic graphs if n and k are known. This algorithm also utilizes the concept of pebbles.*

Definition 8.17. A pebble is an abstract marker that agents can carry around with them and leave at specific nodes to mark them. Pebbles are not distinct, i.e. if an agent encounters a pebble, they can not know whose it is, not even if it is a pebble they left there themself.

9 Distributed minimum cut

Lemma 9.1. *Let $G = (V, E)$ be a graph with edge-connectivity λ. Sample edges from G with probability $p \geq \frac{20 \log n}{\lambda}$ and call the set of sampled edges S. Then the graph $G' = (V, S)$ is connected with probability at least $1 - \frac{1}{n}$.*

In each sampling step, we also assign each edge e to a layer $l \in \{0, 1, ..., L-1\}$ as well, where $L = 20 \log n$. We then define S_i as the set of sampled edges in S that have been assigned layer i and $S_{\leq i} = \bigcup_{j \leq i} S_j$. Finally, for each component C of each graph $G_i = (V, S_i)$ corresponding to a layer i, we collect the cut $(C, V \setminus C)$ in a set F. Note that this amounts to only about $n \log n$ cuts.

Theorem 9.2. *Let $\varepsilon \in (0, 1)$. This sampling process with $p = \varepsilon \frac{\log n}{2\lambda}$ computes a set of cuts F that satisfies*

$$\mathbb{P}[F \textit{ contains an } \mathcal{O}(\varepsilon^{-1}) \textit{ approximation to a minimum cut}] \geq n^{-\frac{\varepsilon}{2}}$$

10 Lazy Random Walks in Algorithms

This is only a very brief overview of the chapter. It is based on lecture 18 which is not relevant for the exam. We therefore omit the more difficult results and content ourselves with an introduction to random walks as they can be a handy tool for many applications.

Definition 10.1 (Lazy Random Walk)**.** Starting from a vertex $v \in V$ we perform a sequence of random experiments. Each round consists of at most two random choices.

1. With probability $\frac{1}{2}$ stay in the current node and skip step 2
2. otherwise uniformly choose a neighbour u of v and go there

Definition 10.2. For a probability distribution P_v^t of a random walk on G starting in v after t rounds, we define the mixing time τ_{mix} as follows the minimal t such that

$$\left| P_v^t(w) - \frac{1}{2md(u)} \right| \leq \frac{1}{2nmd(u)} \; \forall u, w \in V$$

Definition 10.3 (2Δ-regular Random Walk)**.** Assuming a graph on n nodes with maximum degree Δ we add $\Delta - d(v)$ loops to each node v. This multigraph is denoted by G'. Now, a 2Δ-regular random walk on G is simply a lazy random walk on G'. We define $\tau_{mix}(G) = \tau_{mix}(G')$.

Lemma 10.4. *The mixing time $\overline{\tau_{mix}}$ of a 2Δ-regular random walk on G is*

$$\overline{\tau_{mix}} \leq 8\frac{\Delta^2}{h^2(G)} \ln n$$

where $h(G)$ is the so-called edge expansion of G.

Ultimately we have access to parallel processing, so using only one random walk seems like a waste of computing power. To this end, we allow for parallel random walks which is not easily possible in a centralized setting.

Lemma 10.5. *Suppose we run at most $n^{\mathcal{O}(1)}$ synchronous steps of a collection of independent parallel random walks on a graph G. If each node v is the starting point of at most $kd(v)$ random walks, then w.h.p. after each parallel step there are at most $\mathcal{O}(kd(v)+\log n)$ random walks at each node v.*

Lemma 10.6. *When performing $T = n^{\mathcal{O}(1)}$ steps of a collection of independent random walks in parallel where each node v is the starting node of at most $kd(v)$ random walks then w.h.p. the T steps of all random walks can be performed in $\mathcal{O}((k + \log n)T)$ rounds in a distributed system.*

11 MST in complete graphs

Naturally, in a complete graph it is possible to improve on our results about computing MST. The simple reason is that any node can send its message to any other node in just one round. In our previous discussion, we partitioned the graph into multiple fragments and elect a leader in each fragment. This leader is then responsible for all communication with other fragments. This resulted in a $\mathcal{O}(\log n)$ algorithm. Now, let F be a fragment and L_F its leader (think the one with highest ID). The idea is that the leader connects F to more than one fragment in each round. To that intent, we construct an algorithm that allows each node to learn about the $|F|$ lightest outgoing edges of F in $\mathcal{O}(1)$ time. The following algorithm is computed in parallel by all nodes v in fragment F.

1. repeat
2. $\forall F^* \neq F$ compute lightest edge e_{F^*} connecting v with F^*.
3. $\forall F^* \neq F$ send e_{F^*} to L_{F^*}
4. if $v = L_F$
5. $\forall F^* \neq F$ compute lightest edge e_{F,F^*} between F and F^*

```
6. let $E(F)$ be the set of the $|F|$ lightest edges including $e_{F,F^*}$ $\forall F \neq F^*$
7. send each edge from $E(F)$ to a distinct member of $F$
8. endif
9. send the edge received from $L_F$ to all other nodes
10. AddEdges($E^*$)
11. until all nodes are in the same fragment
```

In this algorithm we use a subroutine AddEdges (explained below) that allows each node to decide for all edges in E^* whether it is save to add the edge to the MST. Such edges are called safe. A fragment F that has safe outgoing edges is also called safe. F is same precisely if the original fragments composing F are incident to to at least one safe edge in E^* that has not yet been considered. To keep track of edges that have been considered, we implement a counter $c(F')$ of edges incident to fragment F'. Note that if edge e is incident to F' and F'' and e is added to the MST in any given round, we need to decrement both $c(F')$ and $c(F'')$.
The subroutine AddEdges looks as follows.

```
1. Let $f_1 = F_1, ..., f_s = F_r$ be the original $s$ many fragments of the current round $r$.
2. $\forall i \in [s]$ let $c(f_i)$ be the number of edges in $E^*$ that are incident to $F_i$ and set
   $safe(F_i) = true$
3. while $E^* \neq \varnothing$
4. let $e$ be the lightest edge in $E^*$ between the original fragments $F_i$ and $F_j$
5. $E^* = E^* \setminus \{e\}$
6. $c(F_i) = c(F_i) - 1, c(F_j) = c(F_j)$-1
7. if $e$ connects fragments $f' \neq f''$ and either of them are safe
8. add $e$ to the MST
9. merge $f'$ and $f''$ into larger fragment $f_{new}$
10. if $f', f''$ are both safe and $c(F_i) > 0, c(F_j) > 0$ then set $safe(f_{new}) = true$
11. else $safe(f_{new}) = false$
12. endif
```

```
13. else if $f' = f''$ and $c(F_i) = 0$ or $c(F_j) = 0$ then $safe(f') = false$
14. endif
15. endwhile
```

An edge e that connects two distinct fragments is added if and only if at least one of the fragments is safe. By adding this edge, we merge the two fragments. If both of them are safe and e is not the last safe edge from E^* that connects the two original fragments, the newly created is safe as well.

Theorem 11.1. *The algorithm has a running time of* $\mathcal{O}(\log \log n)$.

12 Robot mutual visibility

12.1 Introduction

Definition 12.1. The gist of this problem is that we want to place robots on pairwise different nodes of a graph G. The shortest path between any two robots should be unobstructed. That is, there can be no other robots placed on any shortest path. The idea is that this allows for secure communication.

Definition 12.2. Given a graph G and $X \subseteq V$ a set of robots placed on G is called X-visible if no shortest path between robots intersects X. We call X a geodesic mutual visibility set (GMV). The maximal cardinality of such sets X is denoted by $\mu(G)$.

12.2 GMV on unit interval graphs

Definition 12.3. Let M be a set of unit intervals on the real line. We define the unit interval graph $G(M)$ as the graph whose vertices are the intervals and where vertices share an edge if and only if the corresponding intervals intersect.

Now, since the vertices of the graph are intervals, we have a natural ordering on the nodes (e.g. the left interval boundary). We can thus construct an ordered set $C = \{C_1, C_2, ..., C_k\}$ of maximal cliques of G. Furthermore, we define the set $R = \{R_1, R_2, ..., R_k\}$ as a set of subsets of V where in each R_i lie the vertices of the intersection of neighbouring cliques. We denóte by $r(G) = \min_{R_i \in R} |R_i|$.

Lemma 12.4. *In a unit interval graph it holds that* $\mu(G) = n - r(G)$.

12.3 GMV on chordal graphs

Definition 12.5. A chordal graph is a graph G without induced cycles of length ≥ 4. In other words: every large cycle is subdivided by chords.

We can characterize chordal graphs by a set of overlapping maximal cliques $C_1, C_2, ..., C_\alpha$. We define sets $R(G) \subseteq V(G)$ as sets of vertices such that

$$R(G) \cap (C_i \cap C_j) \neq \varnothing \; \forall i \neq j$$

and we define $\mathcal{R}(G)$ as the set of such sets $R(G)$. Finally, we denote $\tilde{R}(G)$ as the minimal set from $\mathcal{R}(G)$.

Definition 12.6. Let C_i be a clique such that $\forall j \neq k$ it holds that

$$C_i \cap C_j \neq \varnothing \wedge C_i \cap C_k \neq \varnothing \Rightarrow C_j \cap C_k \neq \varnothing$$

then we call C_i a boundary clique. Intuitively, all cliques intersecting C_i also intersect each other.
Let C_i be a clique such that $\forall j \neq k$ it holds that

$$C_i \cap C_j \neq \varnothing \wedge C_i \cap C_k \neq \varnothing \Rightarrow C_j \cap C_k = \varnothing$$

then we call C_i an intermediate clique.

Lemma 12.7. *Let C_1, C_2, C_3, C_4 be cliques such that $C_1 \cap C_i \neq \varnothing$ and $C_2 \cap C_i \neq \varnothing$ for $i \in \{3, 4\}$, then it holds that $C_1 \cap C_3 = C_1 \cap C_4$ or $C_2 \cap C_3 = C_2 \cap C_4$.*

Definition 12.8. We call a clique C free if $\exists v \in C$ such that $v \notin C'$ for any other clique C'. We call v a free vertex of C.

Lemma 12.9. *Let S be the set of free vertices from all free cliques. Then S is a maximum visibility set and $V(G) \setminus S = \overline{S} = \tilde{R}(G)$ for some minimal $\tilde{R}(G) \in \mathcal{R}(G)$.*

Theorem 12.10. *Finding an optimal GMV in chordal graphs is solvable in polynomial time.*

12.4 GMV on Hypercubes

Definition 12.11. A hypercube graph Q_d is a graph with $n = 2^d$ vertices that are labelled with d digit bitstrings. There is an edge between u and v if the Hamming distance $d_H(u, v)$ is 1.

Lemma 12.12. *$\mu(Q_d) \leq 2\mu(Q_{d-1})$ and by considering the case $d = 5$ one can determine $\mu(Q_d) \leq 2^{d-1}$ $\forall d \geq 5$.*

Theorem 12.13. *$\mu(Q_d) \geq \binom{d}{\lfloor \frac{d}{2} \rfloor} + \binom{d}{\lfloor \frac{d}{2}+3 \rfloor}$.*

Corollary 12.14. *There exists an algorithm for the GMV problem in hypercubes with an approximation factor of $\frac{2^d}{\sqrt{\frac{\pi}{2}d}}$. That is, there is a $\mathcal{O}(\sqrt{d})$-approximation algorithm.*

12.5 GMV on Cube-Connected Cycles

Definition 12.15. A Cube-Connected-Cycle graph of order d can be visualized as hypercube of dimension d where each vertex is replaced by a d-cycle.
A vertex of a CCC graph is a tuple (i, v) where v is a d digit bitstring (analogous to hypercubes) and i is an integer in $[d]$. Intuitively, v indicates which cycle the node is on and i gives the index of the node on that cycle.
There is an edge between (i, v) and (j, u) if

1. $i = j$ and $d_H(u, v) = 1$ or
2. $u = v$ and $|i - j| = 1 \mod d$.

Lemma 12.16. *$\mu(CCC_d) \leq 3 \cdot 2^{d-2}$ $\forall d \geq 3$.*

Theorem 12.17. *$\mu(CCC_d) \geq 2^{\lceil \frac{d}{2} \rceil}$, $\forall d \geq 3$.*

Corollary 12.18. *There exists a $3 \cdot 2^{\lfloor \frac{d}{2} \rfloor - 2}$-approximation algorithm for the GMV problem.*

12.6 GMV on Butterfly graphs

Definition 12.19. A butterfly graph BF_d of order d is a graph with vertices (l, c) where $l \in \{0, 1, ..., d\}$ is called the level and $c \in \{0, 1\}^d$ is called the column. There are edges between (l, c) and (l', c') if

1. $l = l + 1$ and $c = c'$ or
2. $l = l + 1$ and c and c' differ in the l-th bit.

Definition 12.20. We define $A_i = \{(l, i) \mid l \in [d + 1]\}$ as the set of vertices in column i and $L_j = \{(j, c) \mid c \in \{0, 1\}^d\}$ as the vertices of level j.

Lemma 12.21. *Let X be a GMV set. Then*

$$|X \cap A_i| \leq 2 \ \forall i \in [d + 1]$$

Lemma 12.22. *$X = (L0 \cup L_d) \setminus \{(0, 1^d), (d, 1^d)\}$ is a GMV set.*

Lemma 12.23. *$\mu(BF_d) = 2^{d+1} - 2$.*