ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

FS 2019

Prof. Dr. Mohsen Ghaffari,
Prof. Dr. Roger Wattenhofer

Exam
Principles of Distributed Computing

Monday, August 12, 2019
09:00 – 12:00

Do not open or turn until told to by the supervisor!

The exam lasts 180 minutes, and there is a total of 180 points. The maximal number of points for each question is indicated in parentheses. Your answers must be in English. Be sure to always justify (prove) your answers. Algorithms can be specified in high-level pseudocode or as a verbal description. You do not need to give every last detail, but the main aspects need to be there. Big-O notation is acceptable when giving algorithmic complexities. Please write legibly. If we cannot read your answers, we cannot grade them.

Please write down your name and Legi number (your student ID) in the following fields.

Name	Legi-Nr.

Exercise	**Achieved Points**	**Maximal Points**
1 - Multiple Choice		22
2 - Leader Election		12
3 - Sorting Networks		16
4 - No 5-Clique		30
5 - Asynchronous Broadcast		28
6 - Scheduling		20
7 - All-to-All Communication		30
8 - 1-Bit Adjacency Labels		22
Total		**180**

1 Multiple Choice (22 points)

Evaluate each of the following statements in terms of correctness. Indicate whether a statement is true or false by ticking the corresponding box. Each correct answer gives one point. Each wrong answer and each unanswered question gives 0 points.

A) [3] Starting from the same initial rooted tree, we execute Arrow and Ivy for exactly 1 request, from the same requesting node.

	true	**false**
The distance traveled by the find request is the same for Arrow and Ivy.	☐	☐
After the request is served, Arrow and Ivy always end up with the same rooted tree.	☐	☐
After the request is served, Arrow and Ivy always end up with different rooted trees.	☐	☐

B) [3] Consider the parent pointers of nodes in the Arrow algorithm during concurrent find requests.

	true	**false**
The parent pointers form a rooted tree at any moment, even during a find operation.	☐	☐
During a request, the parent pointer changes for two nodes only: the requesting node, and the node currently holding the token.	☐	☐
It is possible that at a given point in time, there are multiple nodes which have a pointer to themselves.	☐	☐

C) [3] Consider the setting of distributed computing, with unique $O(\log n)$-bit identifiers, where per round each node can send one unbounded-size message to each of its neighbors.

	true	**false**
We can compute a maximal independent set of any graph with maximum degree $\Delta = O(1)$, in constant rounds.	☐	☐
We can compute an $O(\log n)$-coloring of any graph with maximum degree $\Delta = O(1)$, in constant rounds.	☐	☐
We can compute an $O(\log \log n)$-coloring of any graph with maximum degree $\Delta = O(1)$, in constant rounds.	☐	☐

D) [3] Consider the setting of distributed computing, with unique $O(\log n)$-bit identifiers, where per round each node can send one unbounded-size message to each of its neighbors.

	true	**false**
For any monotonically non-decreasing function $T(n) \in \mathbb{N}^+$, if there is a $T(n)$-round algorithm for solving maximal independent set in n-node graphs, then there is a $T(n^2)$-round algorithm for solving maximal matching in n-node graphs.	□	□
For any monotonically non-decreasing function $T(n) \in \mathbb{N}^+$, if there is a $T(n)$-round algorithm for solving $(\Delta+1)$ coloring in n-node graphs with maximum degree at most Δ, then there is a $T(n^2)$-round algorithm for solving maximal independent set in n-node graphs.	□	□
For any monotonically non-decreasing function $T(n) \in \mathbb{N}^+$, if there is a $T(n)$-round algorithm for solving maximal independent set in n-node graphs, then there is a $T(n^2)$-round algorithm for computing a $(\Delta+1)$ coloring in n-node graphs with maximum degree at most Δ.	□	□

E) [3] Consider the setting of distributed computing, with unique $O(\log n)$-bit identifiers, where per round each node can send one $O(\log n)$-bit message to each of its neighbors.

	true	**false**
In any graph with constant diameter, we can compute a maximal forest in constant rounds.	□	□
In any graph with constant maximum degree Δ that is not a complete graph or an odd cycle, we can compute a Δ-coloring in $O(\log n)$ rounds.	□	□
In any graph with constant maximum degree, we can determine whether the graph is planar or not in $O(\sqrt{n})$ rounds.	□	□
In any graph with constant diameter, we can identify all of the cut-edges in constant rounds.	□	□

F) [3] Let P be a problem defined on synchronous networks. Let A be a deterministic algorithm solving P, assuming all nodes have nonidentical identifiers and A's time complexity is $C(n)$ where n is the number of nodes in the network.

	true	**false**
Algorithm A still works if there are only two nodes having the same identifier and their distance is larger than $3C(n)$.	□	□
There always exists a randomized algorithm which can solve P in less than $C(n)$ rounds in expectation.	□	□
Algorithm A still works if we add more edges into this network.	□	□

G) [3] Assume two functions $f, g : \{0,1\}^k \times \{0,1\}^k \mapsto \{0,1\}$, and Alice has a k-bit string x and Bob another k-bit string y. The communication complexities of computing $f(x,y)$ and $g(x,y)$ are c_f and c_g respectively.

	true	**false**
If the goal of Alice and Bob is to compute $h = g \oplus f$, then the communication complexity $c_h \leq c_f + c_g$.	☐	☐
If the goal of Alice and Bob is to compute $h = g \oplus f$, then the communication complexity $c_h \geq \min\{c_f, c_g\}$.	☐	☐
Suppose, apart from x, Alice also knows $g(x,y)$, then the communication complexity to compute $f(x,y)$ must be smaller than c_f.	☐	☐

Solutions

A) [3] Starting from the same initial rooted tree, we execute Arrow and Ivy for exactly 1 request, from the same requesting node.

	true	false
The distance traveled by the find request is the same for Arrow and Ivy. *Reason: Both algorithms operate on the same rooted tree, so the find request goes through the same edges in the two cases.*	✓	
After the request is served, Arrow and Ivy always end up with the same rooted tree. *Reason: In general, the two algorithms behave very differently.*		✓
After the request is served, Arrow and Ivy always end up with different rooted trees. *Reason: If the request was only 1 hop away, then the two algorithms produce the same rooted tree.*		✓

B) [3] Consider the parent pointers of nodes in the Arrow algorithm during concurrent find requests.

	true	false
The parent pointers form a rooted tree at any moment, even during a find operation. *Reason: When a find operation travels on an edge, the corresponding nodes are not connected.*		✓
During a request, the parent pointer changes for two nodes only: the requesting node, and the node currently holding the token. *Reason: The parent pointer of every node on the path between these two nodes changes.*		✓
It is possible that at a given point in time, there are multiple nodes which have a pointer to themselves. *Reason: There can indeed be multiple such nodes: the node currently holding the token, and any number of nodes that have initiated a (yet unfinished) find request.*	✓	

C) [3] Consider the setting of distributed computing, with unique $O(\log n)$-bit identifiers, where per round each node can send one unbounded-size message to each of its neighbors.

	true	false
We can compute a maximal independent set of any graph with maximum degree $\Delta = O(1)$, in constant rounds. *Reason:*		✓
We can compute an $O(\log n)$-coloring of any graph with maximum degree $\Delta = O(1)$, in constant rounds. *Reason:*	✓	
We can compute an $O(\log \log n)$-coloring of any graph with maximum degree $\Delta = O(1)$, in constant rounds. *Reason:*	✓	

D) [3] Consider the setting of distributed computing, with unique $O(\log n)$-bit identifiers, where per round each node can send one unbounded-size message to each of its neighbors.

	true	**false**
For any monotonically non-decreasing function $T(n) \in \mathbb{N}^+$, if there is a $T(n)$-round algorithm for solving maximal independent set in n-node graphs, then there is a $T(n^2)$-round algorithm for solving maximal matching in n-node graphs. *Reason:*	✓	
For any monotonically non-decreasing function $T(n) \in \mathbb{N}^+$, if there is a $T(n)$-round algorithm for solving $(\Delta+1)$ coloring in n-node graphs with maximum degree at most Δ, then there is a $T(n^2)$-round algorithm for solving maximal independent set in n-node graphs. *Reason:*		✓
For any monotonically non-decreasing function $T(n) \in \mathbb{N}^+$, if there is a $T(n)$-round algorithm for solving maximal independent set in n-node graphs, then there is a $T(n^2)$-round algorithm for computing a $(\Delta+1)$ coloring in n-node graphs with maximum degree at most Δ. *Reason:*	✓	

E) [3] Consider the setting of distributed computing, with unique $O(\log n)$-bit identifiers, where per round each node can send one $O(\log n)$-bit message to each of its neighbors.

	true	**false**
In any graph with constant diameter, we can compute a maximal forest in constant rounds. *Reason:*	✓	
In any graph with constant maximum degree Δ that is not a complete graph or an odd cycle, we can compute a Δ-coloring in $O(\log n)$ rounds. *Reason:*		✓
In any graph with constant maximum degree, we can determine whether the graph is planar or not in $O(\sqrt{n})$ rounds. *Reason:*		✓
In any graph with constant diameter, we can identify all of the cut-edges in constant rounds. *Reason:*	✓	

F) [3] Let P be a problem defined on synchronous networks. Let A be a deterministic algorithm solving P, assuming all nodes have nonidentical identifiers and A's time complexity is $C(n)$ where n is the number of nodes in the network.

	true	**false**
Algorithm A still works if there are only two nodes having the same identifier and their distance is larger than $3C(n)$. *Reason: Consider P = output different numbers*		✓
There always exists a randomized algorithm which can solve P in less than $C(n)$ rounds in expectation. *Reason: Not every problem has a more efficient randomized algorithm*		✓
Algorithm A still works if we add more edges into this network. *Reason: Some algorithm works only for trees, but not general graphs. We can make a tree to become a general graph by adding edges.*		✓

G) [3] Assume two functions $f, g : \{0,1\}^k \times \{0,1\}^k \mapsto \{0,1\}$, and Alice has a k-bit string x and Bob another k-bit string y. The communication complexities of computing $f(x,y)$ and $g(x,y)$ are c_f and c_g respectively.

	true	**false**
If the goal of Alice and Bob is to compute $h = g \oplus f$, then the communication complexity $c_h \leq c_f + c_g$. *Reason: We can first compute g and f.*	✓	
If the goal of Alice and Bob is to compute $h = g \oplus f$, then the communication complexity $c_h \geq \min\{c_f, c_g\}$. *Reason: Consider the case $f = g$.*		✓
Suppose, apart from x, Alice also knows $g(x,y)$, then the communication complexity to compute $f(x,y)$ must be smaller than c_f. *Reason: If $g = g(x)$ and $f = f(y)$*		✓

2 Leader Election (12 points)

We are given an undirected tree, where each node has a unique ID, in the asynchronous model. There is no distinguished root node. We want to elect a leader among the nodes, such that all nodes learn the ID of this leader. Consider the following algorithm.

1: A node v with $d(v) = 1$ sends a message to its neighbor, where $d(v)$ is the degree of node v.
2: A node v with $d(v) > 1$ **waits** until it receives $d(v) - 1$ messages from its neighbors:
3: Then node v sends a message to the remaining neighbor that has not yet sent a message
4: If a node v receives $d(v)$ messages, then v broadcasts its own ID and becomes the leader.

A) [6] When and why does this algorithm fail?

B) [6] Fix the algorithm. You do not need to prove its correctness.

Solutions

A) In a simple tree with 2 nodes, both would declare themselves as leaders, if both send their message before receiving the message from the other. This also happens when two adjacent nodes send their messages of line 3 around the same time. Then they both satisfy the condition of line 4 and become the leader.

B) Use the same algorithm as above with the following changes. First, nodes send messages containing the largest ID they have seen so far. Note that this message is sent once. When a node receives messages from all of its neighbors, it recognizes that all nodes have participated and therefore whatever ID is the largest for this node so far is the largest ID in the network. So it propagates the maximum ID, which is the leader's ID.

3 Sorting Networks (16 points)

A) [8] We can represent an n-input sorting network with c comparators as a list of c pairs of integers, each integer in $\{1, \ldots, n\}$. If two pairs contain the same integer, the order of the corresponding comparators in the network is determined by the order of the pairs in the list. Given this representation, describe a simple algorithm for determining the depth of the sorting network.

B) [8] Suppose that we have $2n$ elements $a_1, a_2, \ldots, a_{2n}$ and wish to partition them into the n smallest and the n largest. Prove that we can do this in constant additional depth after separately having sorted $a_1, a_2, \ldots, a_n$ respectively $a_{n+1}, a_{n+2}, \ldots, a_{2n}$.

Solutions

A) Create an array with n element, $A[n]$ and initialize it with 0s. Traverse the list of pairs once and fill the array as follows: let (i, j) be the pair, then $A[i] = A[j] = max\{A[i], A[j]\} + 1$. Once you have traversed the list of pairs, return the largest value of the array which is the depth of the comparison network. The time complexity of traversing the list is $O(c)$ because c is the number of pairs and the time complexity of searching the array for the largest value is $O(n)$. Thus, in total the running time of the algorithms is $O(n + c)$.

B) To partition the n smallest from the n largest elements, given the two sorted sequences, it is enough to compare each element just once. Specifically, we compare the pairs (a_1, a_{2n}), $(a_2, a_{2n-1}), \ldots, (a_n, a_{n+1})$. The reason is the following: Since the sequences are sorted, there is an element $a_i, \quad i \leq n$, such that every element after that in the first sequence belongs to the n largest elements. Thus, the $n - i$ elements of the first sequence belong to the n largest elements. But then, the first $n - i$ elements of the second sequence also belong to the n smallest elements. And since a_i is the smallest element of the first list and is still larger that a_{n+i} which is the larger element of the second list, the comparison network described above will swap all the elements below a_i and above a_{n+i}, and hence will return a correct division of the n largest and n smallest elements.

4 No 5-Clique (30 points)

Consider the setting of distributed computing where n processors, with unique $O(\log n)$-bit identifiers, are connected as a network $G = (V, E)$ and can exchange messages in synchronous rounds. Per round, each node can send an unbounded size message to each of its neighbors in G. Initially, each node knows only its neighbors, the network size n, and its maximum degree Δ.

The objective is to compute a maximal set of edges $E' \subseteq E$ that does not include a 5-node complete graph K_5. That is, for any set of 5 nodes, the edge-set E' should have at most 9 connections among these 5 nodes. Moreover, we should not be able to add any edge to E' without violating this property. At the end, each node should know which of its edges are in E'.

A) [15] Suppose that G has maximum degree $\Delta = O(1)$. Devise a deterministic distributed algorithm that, in $O(\log^* n)$ rounds, computes a maximal set of edges $E' \subseteq E$ that does not include a 5-node complete graph K_5.

B) [15] Suppose that the maximum degree Δ can be arbitrarily large (up to $n - 1$). Devise a deterministic distributed algorithm that, in as few rounds as possible as a function of n, computes a maximal set of edges $E' \subseteq E$ that does not include a 5-node complete graph K_5. Any algorithm with a round complexity exceeding $O(n^{0.1})$ receives zero points.

Solutions

A) We start by finding an edge coloring where any two edges of the same color have distance at least 3, as follows. Let $L(G)$ denote the line graph of the input graph G, i.e., the vertex set of $L(G)$ consists of the edges of G and there is an edge between two vertices of $L(G)$ if the corresponding edges in G share an endpoint. Consider $L(G)^2$, the graph obtained from $L(G)$ by taking the vertices of $L(G)$ and connecting any two vertices that have distance at most 2 in $L(G)$ by an edge. Now by applying Linial's algorithm in $L(G)^2$, we obtain a proper coloring of the vertices of $L(G)^2$ with $O(\Delta'^2)$ colors, where Δ' denotes the maximum degree of $L(G)^2$. Such a coloring corresponds to a coloring of the vertices in $L(G)$ where nodes of the same color have distance at least 3, and hence to an edge coloring of G where edges of the same color have distance at least 3. In G, we can simulate the execution of Linial's algorithm in $L(G)^2$ by making each node v in G responsible for all edges in G (i.e., all vertices in $L(G)^2$) of which v is the endpoint with the larger identifier. Since distances in $L(G)^2$ differ only by a constant factor from the corresponding distances in G, the simulation of Linial's algorithm in G incurs only a constant-factor overhead in the runtime compared to the execution in $L(G)^2$. We will ignore this constant factor in the following. Note also that the number of nodes of $L(G)^2$ is at most n^2 (where n denotes the number of nodes of G), and that since $\Delta = O(1)$, we have $\Delta' = O(1)$, by the construction of $L(G)^2$. Hence the runtime incurred so far is $O(\log^* n)$ (for Linial's algorithm), and the number of colors in our edge coloring is constant.

Now, we iterate through the color classes, one by one, and for each color class, we process all edges of that color in parallel. Set $E' = \emptyset$. When we process an edge e, we include it into the set E' if it does not complete a K^5 (i.e., if there is no set of 5 vertices including both endpoints of e such that each of the 10 possible edges between these vertices except e is already present in E'), and we do not include it otherwise. This concludes the description of the algorithm.

In the following, we argue that the algorithm is correct. Since, in a K^5, any two edges have distance at most 2, and no two edges of distance at most 2 are processed in parallel, we have the following: The set E' does not contain a K^5 since if E' contained a K^5, then one of the 10 edges of that K^5 must have been processed strictly after the other 9 edges and, by the construction of our algorithm, would not have been added to E'. Furthermore, E' is maximal as the construction of our algorithm ensures that the only case in which an edge e is not added to E' is when E' already contains 9 edges which, together with e, would form a K^5.

What is left is to determine the runtime of our algorithm. Finding the edge coloring can be done in $O(\log^* n)$ rounds, as argued above, and iterating through the color classes can be done in $O(1)$ rounds as we have a constant number of colors and processing an edge e only requires to check whether adding e would complete a K^5 which only depends on edges in distance at most 2 from e and hence can be performed in constant time. Thus, the total runtime is $O(\log^* n)$ rounds.

B) We will use the terminology and notation of part A). We would also like to use the same process as in A) for deciding whether to add an edge to our set E', but unfortunately finding an edge coloring as above would require too many colors to yield a good runtime. Hence, we instead start by finding a $(2^{O(\sqrt{\log n \log\log n})}, 2^{O(\sqrt{\log n \log\log n})})$ strong-diameter network decomposition of $L(G)^2$. Using Theorem 1.36 from the lecture notes and applying our knowledge about simulating algorithms in G and the number of nodes of $L(G)^2$ from part A), this can be done in $2^{O(\sqrt{\log n \log\log n})}$ rounds.

Now, as in part A), we iterate through the color classes, processing clusters of the same color in parallel. When we process a cluster, a selected node v (e.g., the node with maximum identifier; in $L(G)^2$ one can assume that a node has a unique identifier by concatenating the IDs of the endpoints of the edge in G it corresponds to) in that cluster aggregates all information about the cluster and the cluster's 2-hop neighborhood. Then, v internally (i.e., in 0 rounds) iterates through the edges one by one and, as in part A), includes an edge into the set E' if and only if it does not complete a K^5. Finally v informs everyone in its cluster

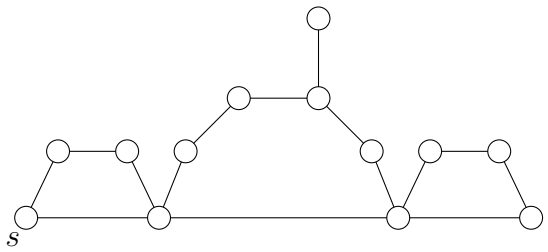
about the decision which edges are added to E'. Now the same arguments as in part A) show that the algorithm is correct.

For the runtime, observe that v can aggregate the cluster information (and also disseminate its decisions) in $2^{O(\sqrt{\log n \log\log n})}$ rounds as this is the diameter of the cluster. As we iterate through $2^{O(\sqrt{\log n \log\log n})}$ color classes, we obtain a runtime of $2^{O(\sqrt{\log n \log\log n})} \cdot 2^{O(\sqrt{\log n \log\log n})} = 2^{O(\sqrt{\log n \log\log n})}$ for the part of the algorithm after finding the network decomposition. Together with the time for finding the network decomposition, we obtain a total runtime of $2^{O(\sqrt{\log n \log\log n})}$ rounds.

It is also possible to construct an algorithm based on a network composition of G^2 instead of $L(G)^2$ with the same runtime.

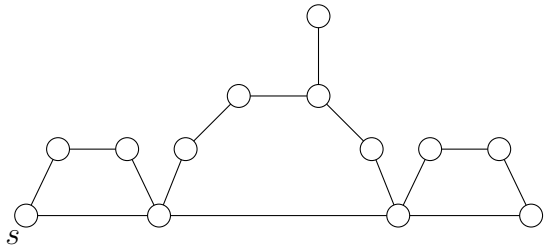
5 Asynchronous Broadcast (28 points)

In this exercise, we study the Broadcast/Echo algorithm in the asynchronous model. Consider the algorithm on the following graph, with node s as the source node:

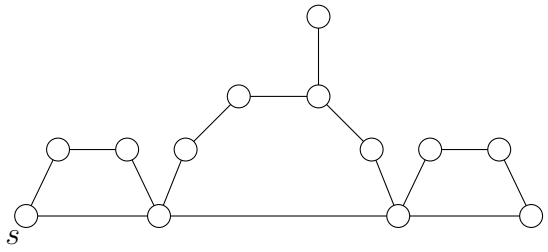


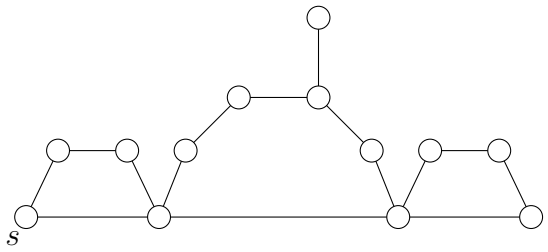
For convenience, we provide copies of this graph at many of the subquestions below; you can use these to sketch your answers.

A) [3] What is the maximal running time of the Broadcast algorithm in this graph?

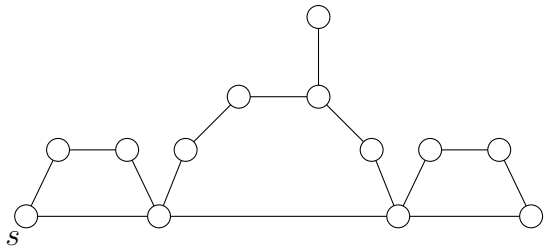


B) [5] Consider the tree obtained from a Broadcast algorithm. What is the minimum number of leaf nodes in such a tree? What is the maximum number of leaf nodes?

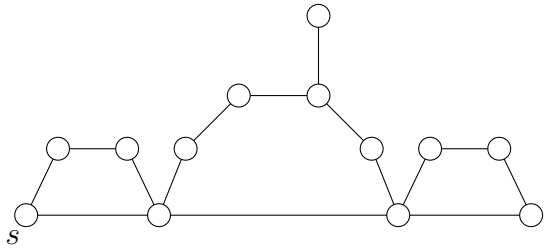




C) [4] Let us call a node v of the graph a *closing node* if there exists an execution of the asynchronous Broadcast algorithm (from s) where v is the last node to receive a broadcast message. Which of the nodes in the graph are closing nodes?

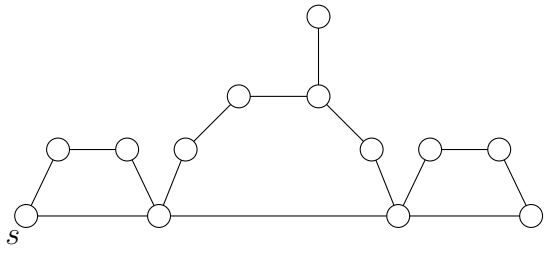


D) [6] Assume that before starting the broadcast, each node already knows who its neighbors are. We now decide to also run an Echo algorithm after the Broadcast: whenever a node has realized that it is a leaf node, it starts the Echo phase immediately. What is the maximal running time of this Broadcast/Echo algorithm?



For questions **E)** and **F)**, assume that we know that the running time of a Broadcast algorithm was at least 4.3 time units.

E) [4] What is the maximal possible running time of an Echo algorithm following the Broadcast in this case?

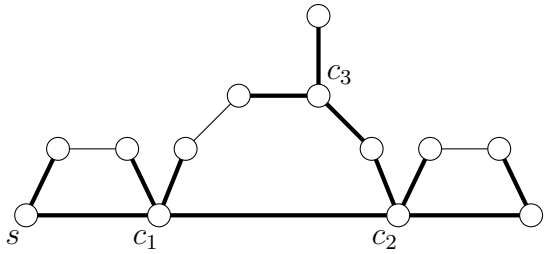


F) [6] What is the number of different spanning trees that we can possibly obtain from such a Broadcast?

Solutions

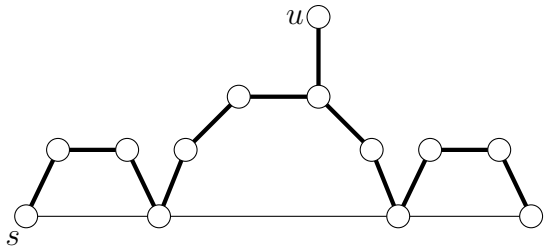
A) The maximal running time of Broadcast is the radius of the tree from node s, which is 5.

B) The following broadcast tree is one of the options to get the maximal number of 7 leaves:



If in the Broadcast algorithm, every message on the edges of this spanning tree is delivered in essentially 0 time, and all other messages travel for 1 time unit, then indeed becomes the broadcast tree. Nodes c_1, c_2 and c_3 can never be leaves, since they are the only connection from s to specific nodes of the graph. Furthermore, at most one of the two neighbors of c_3 can be a leaf node. Finally, in both the left-side and right-side cycle, there can be at most 2 leaves in any spanning tree. Thus, the maximal number of leaves is indeed 7 altogether.

The number of leaves is minimized if the broadcast tree is (almost) a very long path, as shown below:

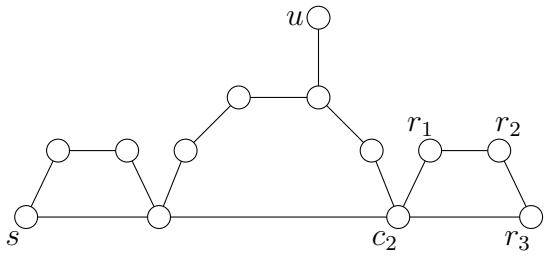


Again, this broadcast tree is achievable if messages on these edges are very fast, and very slow on all other edges. The number of leaves in this tree is 2 (not counting s). Any other broadcast tree must also have at least 2 leaves, because both the uppermost node u and at least one of the nodes in the right-side cycle is always a leaf.

Note: if we apply the graph-theoretic definition of leaf (node of degree 1), the root node s is also a leaf, and thus the correct answer is 3.

C) As discussed before, nodes c_1, c_2 and c_3 can never be leaves of the broadcast tree, so they cannot be closing nodes either. All other nodes can indeed be the last ones to receive a broadcast message. For a specific node v, this is easiest to show if we select a broadcast tree where v is a leaf, deliver all messages (except for the one to v) in essentially 0 time, and deliver the final broadcast messages to v very slowly. Note that if we also consider the definition for s, then s also qualifies as a closing node, since e.g. in the leaf-minimal tree of exercise **B)**, it receives a broadcast message from c_1, which might arrive very late. Therefore, all nodes except for c_1, c_2 and c_3 are closing nodes.

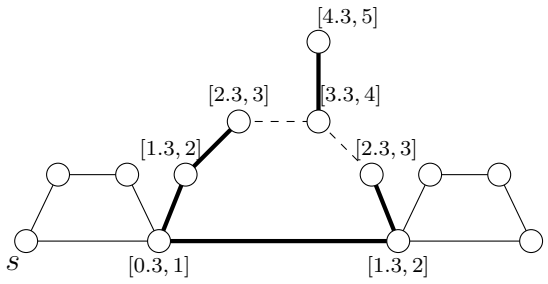
D) In our Broadcast algorithm, a node realizes that it is a leaf when it has received a broadcast message from all of its neighbors. In case of maximal running time, the Echo phase happens as slowly as possible in the broadcast tree developed, taking 1 time unit to travel on each edge. The total running time of Broadcast/Echo is determined by a leaf node v of the broadcast tree which only begins the Echo process after t_1 time units, and is at a distance of t_2 from s in the broadcast tree; such a node v then ensures that the total running time can indeed be at least $t_1 + t_2$. We consider different cases based on this leaf node v whose Echo message is (one of) the last ones to arrive to s.



For node u, Broadcast takes at most 5 time units, and the longest path to s is 7 edges, so u could cause a total running time of 12 at most. For all other nodes in the left-side cycle or the middle part of the graph, this sum is even smaller than 12. This only leaves the right-hand cycle to discuss. As for node r_1 (or r_3 symmetrically), there are two cases. If r_1 is a leaf connected to c_2 in the broadcast tree, then it receives its last broadcast message after at most 5 time units, and has a path of length 9 to s. If r_1 is a leaf connected to r_2 in the broadcast tree, then it receives its last broadcast message after at most 3 time units, and has a path of length 11 to s. This adds up to 14 in both cases. Finally, node r_2 is reached by all broadcast messages after 4 time units, and its maximal distance to s in the broadcast tree is 10, which also adds up to 14.

Therefore, the maximal running time of Broadcast/Echo is 14. It is achieved e.g. if the leaf-minimal tree of exercise **B)** is developed in 3 time units, and node r_3 sends its Echo message to s in 11 time units.

E) As node u is the only node that has a distance larger than 4 from s, this means that u is the node which was only reached after (at least) 4.3 time units. This also implies that any node at distance d from u is reached only after at least $4.3-d$ time units. On the other hand, the distance of each node from s is an upper bound on the time when they receive the first message. Thus, for the nodes in the middle part of the graph, we have the following bounds on the time when they receive the first broadcast message:



Note that if a node w only has two neighbors w_1 and w_2, and the upper bound on the reaching time of w_1 is smaller by 1 than the lower bound on the reaching time of w_2, then w will certainly obtain its first broadcast message from w_1. Hence, the thick edges in the figure are certainly contained in our broadcast tree, and furthermore, the tree contains exactly one of the dashed edges. This shows that in the broadcast tree, the maximal distance of u from s is at most 7. Also, due to the thick edge (c_1, c_2), any node in the right-side cycle has a distance of at most 7 from s. Therefore, the maximal running time of Echo is 7 time units.

F) As discussed before, there are two possible spanning tree variants in the middle part of the graph (corresponding to the two dashed edges). At both sides of the graph, there are 4 possible spanning trees, corresponding to the 4 edges we can omit from the tree; note that all 4 of these trees are indeed obtainable in a broadcast, even with our restriction that the right-hand neighbor of s can only be reached after 0.3 time units at earliest. Altogether, any combination of these forms a valid broadcast tree, so the number of possible broadcast trees is $2 \cdot 4 \cdot 4 = 32$.

6 Scheduling (20 points)

Consider the setting of distributed computing where n processors, with unique $O(\log n)$-bit identifiers, are connected as a network $G = (V, E)$ and can exchange messages in synchronous rounds. Per round, each node can send an unbounded size message to each of its neighbors in G. Initially, each node knows only its neighbors, the network size n, and its maximum degree Δ.

We define a K-day schedule to be an assignment of multiple days from $\{1, 2, \ldots, K\}$ to each of the nodes. A *good* K-day schedule is one in which each node has at least $\log n$ days that are assigned to it but not to any of its neighbors.

Devise and analyze a randomized algorithm that computes a good K-day schedule for $K = 100\Delta \log^2 n$, with probability at least $1 - 1/n$. Each node should know its own assigned dates. An algorithm that needs more than $O(\log \log n)$ rounds receives zero points.

Solutions

There is actually a very fast algorithm for this problem: each node simply chooses each day with probability $1/(2\Delta)$. With the runtime of this algorithm obviously being 0 rounds, in the following we argue about its correctness.

Consider an arbitrary node v. Let us call a day d *excellent* if v selects d, but none of v's neighbors selects d. For each day d, the probability that d is excellent is at least

$$1/(2\Delta) \cdot (1 - 1/(2\Delta))^{\Delta} \geq 1/(2\Delta) \cdot 1/2 = 1/(4\Delta) \ .$$

Hence, for each collection of $100\Delta \log n$ days, the probability that there is no excellent day in this collection is at most

$$(1 - 1/(4\Delta))^{100\Delta \log n} \leq (e^{-1/(4\Delta)})^{100\Delta \log n} \leq 1/(n^{25}) \ .$$

By bucketizing our $100\Delta \log^2 n$ days into $\log n$ buckets of $100\Delta \log n$ days each and applying a union bound, we obtain that the probability that there is a bucket that does not contain an excellent day is at most

$$(\log n)/(n^{25}) \leq 1/(n^{24}) \ .$$

Thus, also the probability that there are fewer than $\log n$ excellent days is upper bounded by $1/(n^{24})$. By union bounding over all nodes, we obtain that the probability that there is a node that has fewer than $\log n$ days that are assigned to it but not to any of its neighbors is at most

$$n/(n^{24}) = 1/(n^{23}) \ .$$

Hence, the computed K-day schedule is good with probability at least

$$1 - 1/(n^{23}) \geq 1 - 1/n \ ,$$

which concludes the analysis.

Note that by applying a Chernoff Bound one can get rid of some of the above calculations.

7 All-to-All Communication (30 points)

Consider the setting of distributed computing with all-to-all communication, where n nodes, which have identifiers in $\{1, 2, \ldots, n\}$, can communicate in synchronous rounds. Per round each node can send $O(\log n)$ bits to each other node. Suppose that these nodes are given a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, in a distributed fashion where each node $v \in V$ knows its neighbors in G.

A) [20] Devise a deterministic distributed algorithm that, in $O(\log n)$ rounds of all-to-all communication, identifies the connected components of G. Each connected component should have one leader (a node in that component) and each node should know the identifier of its component leader.

B) [10] Suppose that G is connected. Devise a deterministic distributed algorithm that, in $O(1)$ rounds of all-to-all communication, identifies all *cut-edges* of G, that is, any edge $e \in E$ such that $G \setminus \{e\}$ is disconnected. You can assume that you are already given an arbitrary spanning tree $T \subseteq G$, in a distributed fashion, where each node v knows its neighbors in T.

Solutions

A) First, we find a coordinator c, e.g. the node with smallest ID. In the beginning, each node forms its own component. Assume that in every step, each node knows the ID of its components leader, as well as the IDs of the leaders for all its neighbors. One step proceeds as follows: Every node v selects one of its neighbors u, that belongs to a different component, and sends the information about this edge (u, v) to the coordinator c. With all the information from these messages, c now merges all components that are connected by an edge. It then informs the nodes about the ID of their new leader. Lastly, every node informs their neighbors about the ID of its new leader. In every step, except for the components that are already maximal, every new component contains at least two components from the previous step. Thus, this process takes $O(\log n)$ rounds.

B) First, we observe the following: A spanning tree has exactly $n-1$ edges, and every cut edge of G must be contained in T. Our goal will be that all nodes learn T. Assume that we have a leader node l, e.g. the node with the smallest ID, and note that we can find such a node l in 1 round. Further assume that we have one node "responsible" for each tree edge, e.g. the endpoint with smaller ID. Let r_v be the number of such edges that a node v is responsible for and note that $\sum_v r_v = n-1$. Each node now sends r_v to the leader l. Using these values, l creates a routing scheme, by assigning r_v distinct nodes to each v. So lets say v receives the nodes $\{u_1, \ldots, u_{r_v}\}$. Now, v sends the first edge e_1 it is responsible for to u_1, the second e_2 to u_2 and so on. From now on, lets say that u_i is the *representative* of e_i. This will result in a state where every node holds (at most) one edge of T, and in one more round, every node can learn all edges of T.

Now, for every tree edge e we can determine if e is a cut-edge as follows: If a node is incident to some edge e' whose addition to T creates a cycle containing e, we know that e is not a cut edge. If no such edge e' exists, e must be a cut edge. In a distributed fashion, we can perform the above, by having each node v that notices that some edge e is not a cut edge, inform the representative u of v. If a representative u receives no such information, then its edge e must be a cut edge.

8 1-Bit Adjacency Labels (22 points)

We want to label nodes for determining adjacency (whether two nodes are neighbors) based only on their node labels. Each label is exactly 1 bit, i.e. every node is labeled with either 0 or 1. Clearly, many nodes will have the same label. Assume there are no queries asking whether a node is adjacent to itself. If we can obtain one graph from another by shuffling node ID's, the graphs are considered identical.

A) [10] Determine the largest k, such that there is a 1-bit adjacency labeling scheme for *all* graphs with k nodes.

B) [10] Consider graphs with exactly 10 nodes. Give a 1-bit adjacency labeling scheme such that you can label as many graphs as possible. How many different graphs with 10 nodes can you label?

C) [2] How many graphs with exactly 20 nodes can you label (using the same labeling scheme as in B)?

Solutions

For a given labeling scheme, by $a(x, y) = 1$ (respectively $a(x, y) = 0$) we denote that nodes with labels x and y are connected (respectively not connected).

A) With a labeling scheme such that $a(1, 1) = 1$ and $a(0, 0) = 0$ we can label all 2 graphs with $k = 2$ nodes.

Suppose for contradiction that all graphs with $k = 3$ nodes can be labeled with some scheme. In any graph with 3 nodes, at least 2 have to have the same label. Without loss of generality, assume two nodes in a fully connected graph have the label 1. Then, $a(1, 1) = 1$. Hence, in a graph with no edges, at least two nodes have the label 0. Hence, $a(0, 0) = 0$.

Suppose $a(1, 0) = 1$. Then, any node being labeled with a 1 would mean at least two edges are present, making it impossible to label the graph with exactly one edge. Suppose $a(1, 0) = 0$. Then, any node being labeled with a 0 would mean at least two edges are absent, making it impossible to label the graph with exactly two edges.

Hence, $k = 2$ is the biggest number, such that there is a labeling scheme for all graphs of size k.

B) Note that given a labeling scheme and a labeling of nodes, the associated graph can be determined. Suppose for contradiction there are two different (label-able) graphs with the same number of nodes labeled by a 1 (and the same number labeled by a 0). Shuffle the nodes so that the graphs are labeled identically. Since the graphs are different, there is a pair of nodes with labels x, y such that $a(x, y) = 1$ in one graph, and $a(x, y) = 0$ in the other, a contradiction. Hence, for any labeling scheme, each number $m = 0, \ldots, 10$ of nodes labeled with a 1 can correspond to at most one graph that can be labeled.

Suppose for contradiction that each $m = 0, \ldots, 10$ corresponds to a different graph. Then, $m = 0$ corresponds to a different graph than $m = 1$, so $a(0, 0) \neq a(1, 0)$. Analogously for $m = 9, 10$ we conclude $a(1, 1) \neq a(1, 0)$. Hence, $a(0, 0) = a(1, 1)$, and graphs corresponding to $m = 0, 10$ must be the same, a contradiction.

Hence, at most 10 different graphs can be labeled.

Let $a(1, 1) = 1$, $a(0, 0) = 0$, $a(1, 0) = 1$. Increasing m from 0 up to 9 increases the number of edges, producing different graphs. ($m = 9, 10$ correspond to the same graph since $a(1, 1) = a(1, 0)$). Hence, 10 graphs can be labeled.

C) Substituting $m = 0, \ldots, 20$ in the point B), we obtain 20 graphs that can be labeled.