Final Solutions

- Do not open this exam booklet until you are directed to do so. Read all the instructions first.
- The exam contains 10 problems, with multiple parts. You have 180 minutes to earn 180 points.
- This exam booklet contains 20 pages, including this one.
- This exam is closed book. You may use three double-sided letter ($8\frac{1}{2}'' \times 11''$) or A4 crib sheets. No calculators or programmable devices are permitted. Cell phones must be put away.
- Do not waste time deriving facts that we have studied. Just cite results from class.
- When we ask you to "give an algorithm" in this exam, describe your algorithm in English or pseudocode, and provide a short argument for correctness and running time. You do not need to provide a diagram or example unless it helps make your explanation clearer.
- Do not spend too much time on any one problem. Generally, a problem's point value is an indication of how many minutes to spend on it.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Please be neat.
- Good luck!

Q	Title	Points	Parts	Grade	Q	Title	Points	Parts	Grade
1	True or False	56	14		6	Be the Computer	14	3	
2	Überstructure	10	1		7	Startups are Hard	20	3	
3	Meancorp	15	2		8	Load Balancing	15	2	
4	Forgetful Forrest	15	3		9	Distributed Coloring	20	3	
5	Piano Recital	15	3		Total		180		

Name: _____

Problem 1. True or False. [56 points] (14 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false and briefly explain why.

(a) T F [4 points] Suppose algorithm $\mathcal{A}$ has two steps, and $\mathcal{A}$ succeeds if both the steps succeed. If the two steps succeed with probability p_1 and p_2 respectively, then $\mathcal{A}$ succeeds with probability $p_1 p_2$.

Solution: False. Unless the two steps are independent.

(b) T F [4 points] If the divide-and-conquer convex hull algorithm (from Lecture 2) used a $\Theta(n^2)$ strategy to discover the maximum and minimum tangents, the overall algorithm would run in $\Theta(n^2 \log n)$ time.

Solution: False. The recurrence would be $T(n) = 2T(\frac{n}{2}) + \Theta(n^2)$ whose solution is $T(n) = \Theta(n^2)$.

(c) T F [4 points] In order to get an expected $\Theta(n \log n)$ runtime for "paranoid" quicksort (from Lecture 3), we require the recursive divide step to split the array into two subarrays each of at least $\frac{1}{4}$ the size of the original array.

Solution: False. As long as it is a constant fraction of the original array, we can get the bound.

(d) T F [4 points] A binary min-heap with n elements supports INSERT in $O(\log n)$ amortized time and DELETE-MIN in 0 amortized time.

Solution: True. Same amortization as in class for insert/delete in 2-3 trees.

(e) T F [4 points] The hash family $H = \{h_1, h_2\}$ is universal, where $h_1, h_2 : \{1, 2, 3\} \to \{0, 1\}$ are defined by the following table:

	1	2	3
h_1	0	1	0
h_2	1	0	1

(For example, $h_1(3) = 0$.)

Solution: False. Consider elements 1 and 3: h_1 and h_2 both cause a collision between them, so in particular a uniformly random hash function chosen from H causes a collision between 1 and 3 with probability 1, greater than the $1/2$ allowed for universal hashing (since there are 2 hash buckets).

(f) T F [4 points] Recall the $O(n^3 \lg n)$ matrix-multiplication algorithm to compute shortest paths, where we replaced the matrix-multiplication operator pair $(*, +)$ with $(+, \min)$. If we instead replace the operator pair with $(+, *)$, then we compute the product of the weights of all paths between each pair of vertices.

Solution: False. If the graph has a cycle, there are infinitely many paths between some pairs of vertices, so the product ought to be $\pm\infty$, yet the matrix-multiplication algorithm will compute finite values if the original matrix has all finite values (e.g., a clique).

(g) T F [4 points] Negating all the edge weights in a weighted undirected graph G and then finding the minimum spanning tree gives us the *maximum*-weight spanning tree of the original graph G.

Solution: True.

(h) T F [4 points] In a graph with unique edge weights, the spanning tree of second-lowest weight is unique.

Solution: False, can construct counter-example.

(i) T F [4 points] In the recursion of the Floyd–Warshall algorithm:

$$d_{uv}^{(k)} = \min\{d_{uv}^{(k-1)}, d_{uk}^{(k-1)} + d_{kv}^{(k-1)}\},$$

$d_{uv}^{(k)}$ represents the length of the shortest path from vertex u to vertex v that contains at most k edges.

Solution: False. $d_{uv}^{(k)}$ is the length of the shortest path from vertex u to vertex v that only uses vertex $\{1, 2, \cdots k\}$ as intermediate nodes.

(j) T F [4 points] Consider a network of processes based on an arbitrary undirected graph $G = (V, E)$ with a distinguished vertex $v_0 \in V$. The process at each vertex $v \in V$ starts with a positive integer x_v. The goal is for the process at v_0 to compute the maximum $\max_{v \in V} x_v$. There is an asynchronous distributed algorithm that solves this problem using $O(diam^2 d)$ time and $O(E + diam \cdot n)$ messages.

Solution: True.

Using the algorithm from Problem 10-2, we can construct a BFS tree rooted at v_0 within the given time and message bounds. The root process can broadcast a signal telling all the processes that the tree is completed. Then the processes can use the tree for convergecasting their values, computing the max as the messages move up the tree. The broadcast and convergecast phases do not exceed the bounds for the BFS construction.

(k) T F [4 points] Suppose a file server stores a hash of every file in addition to the file contents. When you download a file from the server, you also download the hash and confirm that it matches the file. This system securely verifies that the downloaded file has not been modified by an adversary, provided the hash function has collision resistance.

Solution: False. This scheme is not secure because the adversary can simply replace the file with any file and the hash of that file, and you cannot tell the difference.

(l) T F [4 points] Suppose Alice, Bob, and Charlie secretly generate a, b and c, respectively, and publish $g^a \bmod p$, $g^b \bmod p$, and $g^c \bmod p$, where p is a prime. Then, Alice, Bob, and Charles can each compute $g^{abc} \bmod p$ as a shared secret known only to the three of them.

Solution: False. For example, Alice only knows a, g^b and g^c, so she can compute g^{ab} and g^{ac} but not g^{abc}.

(m) T F [4 points] The number of memory transfers used by the best cache-oblivious algorithm is always at least the number of memory transfers used by the best external-memory algorithm for the same problem.

Solution: True. Make implicit memory transfers explicit, using LRU.

(n) T F [4 points] If there is a time-optimal divide-and-conquer algorithm for a problem, then that algorithm is also optimal with respect to memory transfers in the cache-oblivious model.

Solution: False. Example: binary search.

Problem 2. Überstructure [10 points] (1 part)

Design a data structure that maintains a dynamic set S of n elements subject to the following operations and time bounds:

Operation	Effect	Time Bound	
1. INSERT(x, S)	Insert x into S.	$O(\log n)$	expected amortized
2. DELETE(x, S)	Delete x from S.	$O(\log n)$	expected amortized
3. SUCCESSOR(x, S)	Find the smallest element in S larger than x.	$O(\log n)$	worst-case
4. FIND-MIN(S)	Return the smallest element in S.	$O(1)$	worst-case
5. SEARCH(x, S)	Return TRUE if element x is in S.	$O(1)$	expected

Describe how the operations are implemented on your data structure and justify their runtime.

Solution: Use a balanced binary search tree and a hash table. Augment the root of the balanced binary search tree with the value of the minimum element.

INSERT: Insert the element in both the balanced binary search tree and hash table. If the element is smaller than the current min, update the root's stored min value. Insertion into the tree requires $O(\log n)$ worst-case time, and insertion into the hash table requires $O(1)$ expected amortization time, for a total of $O(\log n)$ expected amortized. (In fact, with high probability, insertion into the hash table requires at most $O(\log n)$.)

DELETE: Find the item in the tree and the hash table, and delete it from both. Rebalance the tree as necessary and update the root's min value if the minimum element has been deleted. Deletion, including rebalancing, costs $O(\log n)$, and it takes $O(\log n)$ to find the minimum element using the binary search tree.

FIND-MIN: Return the min value stored at the root node. This takes $O(1)$ worst-case time.

SEARCH: Check whether the element exists in the hash table. This takes $O(1)$ expected time.

Problem 3. Meancorp [15 points] (2 parts)

You are in charge of the salary database for Meancorp, which stores all employee salaries in a 2-3 tree ordered by salary. Meancorp compiles regular reports to the Department of Fairness about the salary for low-income employees in the firm. You are asked to implement a new database operation AVERAGE(x) which returns the average salary of all employees whose salary is at most x.

(a) [10 points] What extra information needs to be stored at each node? Describe how to answer an AVERAGE(x) query in $O(\lg n)$ time using this extra information.

Solution: Each node x should store $x.size$ — the size of the subtree rooted at x — and $x.sum$ — the sum of all the key values in the subtree rooted at x. For a value $x > 0$, let S_x be the set of all keys less than or equal to x. Let A_x and B_x be the sum and the size of S_x.

We can compute A_x as follows. Let u be the leaf with smallest key larger than x. Finding u from the root only takes $O(\lg n)$ time by using SEARCH in a 2-3 tree. Now consider the path from the root of the tree to u. Clearly, A_x is the sum of all leaves that are on the left of this path. Therefore, A_x can be computed by summing up all $y.sum$'s for every node y that is a left sibling of a node in the path. Since there are only $\lg n$ such nodes y's, computing A_x only takes $O(\lg n)$ time.

Computing B_x is similar: instead of summing up $y.sum$, we sum up $y.size$. Therefore, it also takes $O(\lg n)$ time to compute B_x.

Therefore, AVERAGE(x) which is $\frac{A_x}{B_x}$ can be answered in $O(\lg n))$ time.

(b) [5 points] Describe how to modify INSERT to maintain this information. Briefly justify that the worst-case running time for INSERT remains $O(\lg n)$.

Solution: Maintaining $x.size$ is similar to what was covered in recitation and homework. Maintaining $x.sum$ is exactly the same: when a node x gets inserted, we simply increase $y.sum$ for every ancestor y of x by the amount $x.key$. When a node splits, we recompute the $x.sum$ attribute for the split nodes and its parent. Hence, INSERT still runs in worst-case time $O(\lg n)$.

Problem 4. Forgetful Forrest [15 points] (3 parts)

Prof. Forrest Gump is very forgetful, so he uses automatic calendar reminders for his appointments. For each reminder he receives for an event, he has a 50% chance of actually remembering the event (decided by an independent coin flip).

(a) [5 points] Suppose we send Forrest k reminders for each of n events. What is the expected number of appointments Forrest will remember? Give your answer in terms of k and n.

Solution: These are all independent events. So linearity of expectation applies. Each given event has been remembered with probability $1 - 2^{-k}$. So in expectation $n(1 - 2^{-k})$ appointments are remembered.

(b) [5 points] Suppose we send Forrest k reminders for a *single* event. How should we set k with respect to n so that Forrest will remember the event with high probability, i.e., $1 - 1/n^{\alpha}$?

Solution: This problem is equivalent to how many times we must flip a coin to get a head with high probability. The probability of k tails in a row is $1/2^k$. Thus exactly $\alpha \lg n$ coin flips suffice.

(c) [5 points] Suppose we send Forrest k reminders for each of n events. How should we set k with respect to n so that Forrest will remember *all* the events with high probability, i.e., $1 - 1/n^{\alpha}$?

Solution: We must send at least $k = \Omega(\lg n)$ reminders, because we needed this many reminders to remember one event with high probability.

If we send $k = (\alpha + 1) \lg n$ reminders, then each event is remembered with probability $1 - 1/n^{\alpha+1}$. By a union bound, we know that all events are remembered with probability $1 - 1/n^{\alpha}$. So, the number of reminders needed is $k = O(\lg n)$.

Problem 5. Piano Recital [15 points] (3 parts)

Prof. Chopin has a piano recital coming up, and in preparation, he wants to learn as many pieces as possible. There are m possible pieces he could learn. Each piece i takes p_i hours to learn.

Prof. Chopin has a total of T hours that he can study by himself (before getting bored). In addition, he has n piano teachers. Each teacher j will spend up to t_j hours teaching. The teachers are very strict, so they will teach Prof. Chopin only a single piece, and only if no other teacher is teaching him that piece.

Thus, to learn piece i, Prof. Chopin can either (1) learn it by himself by spending p_i of his T self-learning budget; or (2) he can choose a unique teacher j (not chosen for any other piece), learn together for $\min\{p_i, t_j\}$ hours, and if any hours remain ($p_i > t_j$), learn the rest using $p_i - t_j$ hours of his T self-learning budget. (Learning part of a piece is useless.)

(a) [6 points] Assume that Prof. Chopin decides to learn exactly k pieces. Prove that he needs to consider only the k lowest p_is and the k highest t_js.

Solution: Assume there exists a selection of teachers and pieces for learning k pieces. Let the set of lowest k pieces be P_k. If there is a piece in our selection that is $\notin P_k$, then we must have a piece in P_k not in the final selection. If we swap the one with the higher cost ($\notin P_k$) with the one with lower cost ($\in P_k$), the new selection thus made will still be valid, because if the higher time cost was fulfilled in the previous selection, the lower time cost in the new selection will still be fulfilled. In this way, we can swap pieces until all of them are $\in P_k$.

Similarly, we can swap the teachers for those of higher value until they are the ones with the k highest times.

(b) [5 points] Assuming part (a), give an efficient greedy algorithm to determine whether Prof. Chopin can learn exactly k pieces. Argue its correctness.

Solution: Let us sort all the teachers and pieces in increasing order beforehand. Call the sorted lists P and T. We see that if a solution exists, there is also one in which P_1 is paired with T_{n-k+1}, P_2 is paired with T_{n-k+2} and so on.
So for each $1 \leq i \leq k$, the greedy algorithm checks if $P_i \leq T_{n-k+i}$. If it is, then we don't need to use the shared time for this piece. If it is not, we need to use $T_{n-k+i} - P_i$ of the shared time. We can add up these values. In the end, if the total shared time we need is $> T$, we return false. Otherwise, we return true.

This takes $O(k)$ time, apart from the initial sorting.

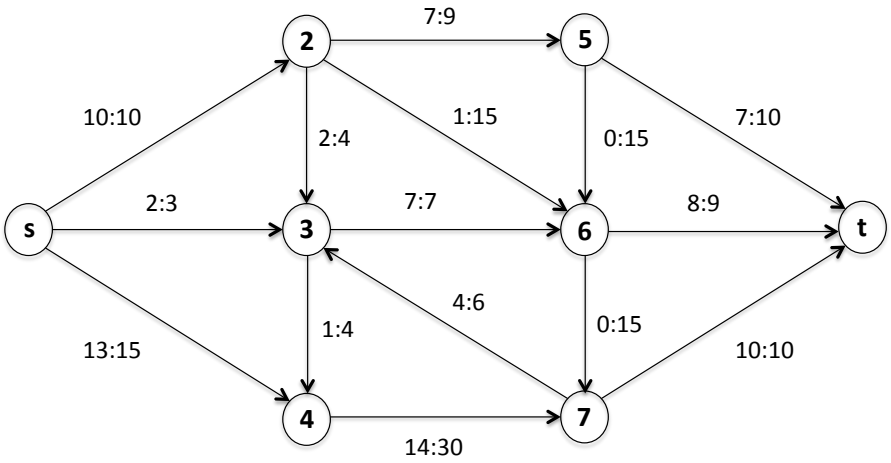
(c) [4 points] Using part (b) as a black box, give an efficient algorithm that finds the maximum number of pieces Prof. Chopin can learn. Analyze its running time.

Solution: Notice that if k_{max} is the maximum value of pieces we can learn, we can also learn k pieces for any $k \leq k_{max}$. This suggests that we binary search over the value of k.

We try $O(\log n)$ values during the binary search, and checking each value takes $O(n)$ time. This takes $O(n \log n)$ time. The sorting also took $O(n \log n)$ time, so the algorithm takes $O(n \log n)$ time overall.

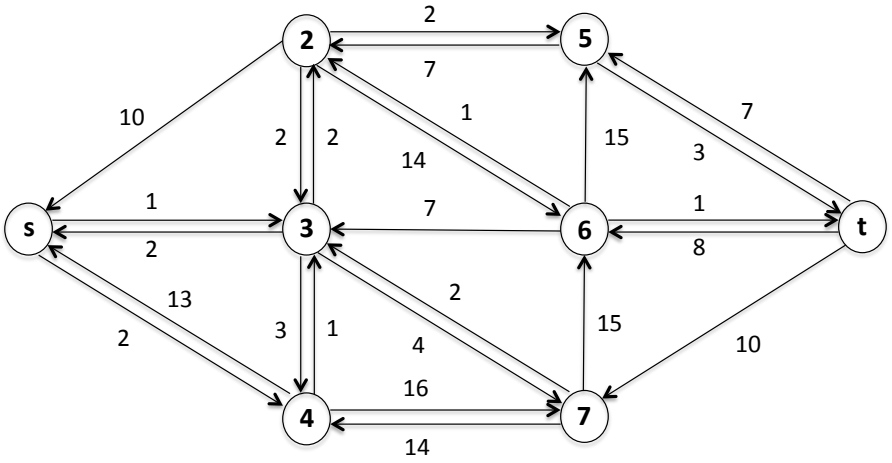
Problem 6. Be the Computer [14 points] (3 parts)

Consider the following flow network and initial flow f. We will perform one iteration of the Edmonds–Karp algorithm.



(a) [5 points] Draw the residual graph G_f of G with respect to f.

Solution:



(b) [4 points] List the vertices in the shortest augmenting path, that is, the augmenting path with the fewest possible edges.

Solution:

$$s \to 3 \to 2 \to 5 \to t$$

or

$$s \to 3 \to 2 \to 6 \to t$$

(c) [5 points] Perform the augmentation. What is the value of the resulting flow?

Solution: 26. The augmenting flow has value 1.

Problem 7. Startups are Hard [20 points] (3 parts)

For your new startup company, *Uber for Algorithms*, you are trying to assign projects to employees. You have a set P of n projects and a set E of m employees. Each employee e can only work on one project, and each project $p \in P$ has a subset $E_p \subseteq E$ of employees that must be assigned to p to complete p. The decision problem we want to solve is whether we can assign the employees to projects such that we can complete (at least) k projects.

(a) [5 points] Give a straightforward algorithm that checks whether any subset of k projects can be completed to solve the decisional problem. Analyze its time complexity in terms of m, n, and k.

Solution: For each $\binom{n}{k}$ subsets of k projects, check whether any employee is required by more than one project. This can be done simply by going each of the k projects p, marking the employees in E_p as needed, and if any employee is marked twice, then this subset fails. Output "yes" if any subset of k project can be completed, and "no" otherwise.

The time complexity is $\binom{n}{k} \cdot m$ because there are $\binom{n}{k}$ subsets of size k and we pay $O(m)$ time per subset (because all but one employee will be marked only once). Asymptotically, this is $(n/k)^k m$.

(b) [5 points] Is your algorithm in part (a) fixed-parameter tractable? Briefly explain.

Solution: No. An FPT algorithms requires a time complexity of $n^{O(1)} f(k)$. By contrast, in our running time, the exponent on n increases with k.

(c) [10 points] Show that the problem is NP-hard via a reduction from 3D matching.

Recall the 3D matching problem: You are given three sets X, Y, Z, each of size m; a set $T \subseteq X \times Y \times Z$ of triples; and an integer k. The goal is to determine whether there is a subset $S \subseteq T$ of (at least) k disjoint triples.

Solution: Each $(x, y, z) \in T$ becomes a project that requires employees $E_{(x,y,z)} = \{e_x, e_y, e_z\}$. Thus $n = |T|$, $E = X \cup Y \cup Z$, and $m = |X| + |Y| + |Z|$. We set k to be the same in both problems. The size of the matching is equal to the number of projects that can be completed because both problems model disjointness: if k projects can be completed, a subset S of size k can be found, and vice versa. The reduction takes polynomial time.

Problem 8. Load Balancing [15 points] (2 parts)

Suppose you need to complete n jobs, and the time it takes to complete job i is t_i. You are given m identical machines $M_1, M_2, \ldots, M_m$ to run the jobs on. Each machine can run only one job at a time, and each job must be completely run on a single machine. If you assign a set $J_j \subseteq \{1, 2, \ldots, n\}$ of jobs to machine M_j, then it will need $T_j = \sum_{i \in J_j} t_i$ time. Your goal is to partition the n jobs among the m machines to minimize $\max_i T_i$.

(a) [5 points] Describe a greedy approximation algorithm for this problem.

Solution: Let J_j to be the set of jobs that M_j will run, and T_j to be the total time it machine M_j is busy (i.e., $T_j = \sum_{i \in J_j} t_i$). Initially, $J_j = \emptyset$, and $T_j = 0$ for all j.

For $i = 1, \ldots, n$, assign job i to machine M_j such that $T_j = \min_{1 \le k \le m}(T_k)$. That is, $J_j = J_j \cup i$ and $T_j = T_j + t_i$. Output J_j's.

This runs in $O(n \lg m)$ time by keeping a min-heap of the machines based on the current total runtime of each machine.

Solution: Alternate solution: Sort jobs in non-increasing order. Without loss of generality, let the jobs in order be $t_1, \ldots, t_n$. Let $J_j = \{t_{k:k \equiv j \mod m}\}$. Variations of this algorithm also works, with different sorting orders and assignments. This takes $O(n \lg n)$ time to sort the jobs.

(b) [10 points] Show that your algorithm from part (a) is a 2-approximation algorithm.
Hint: Determine an ideal bound on the optimal solution OPT. Then consider the machine M_ℓ with the longest T_ℓ, and the last job i^* that was added to it.

Solution: A lower bound to the optimal is $L = \max(\frac{1}{m}\sum_{1\le i\le n} t_i, \max_i(t_i))$ since the best you can do is to evenly divide the fractional jobs, and it has to run for at least as long as the longest job.
Now let M_ℓ be the machine that runs for the longest, and let i^* be the last job that was assigned to M_ℓ using the greedy algorithm. Let T_j^* be the total run time of all jobs of M_j immediately before assigning i^*; $T_\ell^* = \min_j T_j^*$. Then we have

$$m \cdot T_\ell^* \le \sum_{1\le j\le m} T_j^* = \sum_{1\le i\le i^*} t_i \le \sum_{1\le i\le n} t_i \le m \cdot L,$$

which implies that $T_\ell^* \le L$. Putting it together, we have $T_\ell = T_\ell^* + t_{i^*} \le L + t_{i^*} \le 2L \le 2OPT$. Therefore, this is a 2-approximation algorithm.

Solution: Proof for alternate solution: Let L be the lower defined above. Consider the longest job t_n. Let $k \equiv n \mod m$, and let $S_k = T_k - t_n$. It must be that $S_k \le T_j$ for all j: for $j > k$, we only added elements at least as large as every element of S_k. For $j < k$, there are $a = \lceil \frac{m}{n} \rceil$ jobs, and the last $a-1$ jobs in J_j are greater the first $a-1$ jobs of J_k due to the jobs being sorted, which shows that $S_k \le T_j$. Then

$$\sum_{i=1}^{n} t_i = t_n + S_k + \sum_{j\ne k} T_j \ge t_n + mS_k.$$

Therefore, $mS_k \le \sum_{i=1}^n t_i$, which implies $S_k \le L$. Since $t_n \le L$ also, we get that $t_n + S_k \le 2L \le 2OPT$.

Problem 9. Distributed Coloring [20 points] (3 parts)

Consider an undirected graph $G = (V, E)$ in which every vertex has degree at most Δ. Define a new graph $G' = (V', E')$, the ***Cartesian product*** of G with a clique of size $\Delta + 1$. Specifically, V' is the set of pairs (v, i) for all vertices $v \in V$ and integers i with $0 \leq i \leq \Delta$, and E' consists of two types of edges:

1. For each edge $\{u, v\} \in E$, there is an edge between (u, i) and (v, i) in E', for all $0 \leq i \leq \Delta$. (Thus, each index i forms a copy of G.)
2. For each vertex $v \in V$, there is an edge between (v, i) and (v, j) in E', for all $i \neq j$ with $0 \leq i, j \leq \Delta$. (Thus each v forms a $(\Delta + 1)$-clique.)

Here is an example of this transformation with $\Delta = 3$:

Figure 1: Graph G.

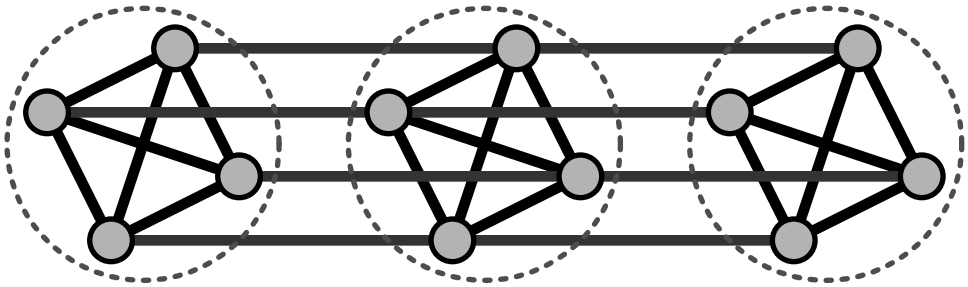


Figure 2: The Cartesian product G' of G and a clique of size 4.

(a) [8 points] Let S be any *maximal* independent set of G' (i.e., adding any other vertex to S would violate independence). Prove that, for each vertex $v \in V$, S contains exactly one of the $\Delta + 1$ vertices in V' of the form (v, i). *Hint:* Use the Pigeonhole Principle.

Solution: It cannot contain more than one, since all of these are connected in G' and that would violate independence.

Now suppose for contradiction that, for some particular u, S contains no vertices of the form (u, i). Then by maximality, every vertex of the form (u, i) must have some G'-neighbor in S. Since that neighbor is not of the form $(u, *)$, it must be of the form (v, i), for some v with $(u, v) \in E$.

Thus, each of the $\Delta+1$ vertices of the form (u, i) has some neighbor of the form (v, i) in S, where $(u, v) \in E$. Since u has at most Δ neighbors in G, by the Pigeonhole Principle, there must be two different values of i, say i_1 and i_2, for which there is a single v such that (u, i_i) is a G'-neighbor of (v, i_1), (u, i_2) is a G'-neighbor of (v, i_2), and both (v, i_1) and (v, i_2) are in S. That is a contradiction because S can contain at most one vertex of the form $(v, *)$.

(b) [8 points] Now consider a synchronous network of processes based on the graph G, where every vertex knows an upper bound Δ on the degree. Give a distributed algorithm to find a vertex $(\Delta + 1)$-coloring of G, i.e., a mapping from vertices in V to colors in $\{0, 1, \ldots, \Delta\}$ such that adjacent vertices have distinct colors. The process associated with each vertex should output its color. Argue correctness.

Hint: Combine part (a) with Luby's algorithm.

Solution: The "colors" will be chosen from $\{0, 1, \ldots, \Delta\}$.

The nodes of G simulate an MIS algorithm for G'. Specifically, the node associated with vertex u of G simulates the $\Delta + 1$ nodes associated with vertices of the form (u, i) of G'. The algorithm produces an MIS S for G', where each node of G learns which of its simulated nodes correspond to vertices in S. By Part (a), for each vertex u of G, there is a unique color i such that $(u, i) \in S$; the node associated with u chooses this color i.

Obviously, this strategy uses at most $\Delta + 1$ colors. To see that no two neighbors in G are colored with the same color, suppose for contradiction that neighbors u and v are colored with the same color, say i. That means that both (u, i) and (v, i) are in S. But (u, i) and (v, i) are neighbors in G', contradicting the independence property for S.

An alternative solution that many students wrote involved executing $\Delta + 1$ instances of Luby's MIS directly on G, in succession. In each instance i, the winners are colored with color i. Then we remove just the winners before executing the next instance. This works, but leaves out some details w.r.t. synchronizing the starts of the successive instances. Also, its performance is quite a bit worse than the recommended solution above.

(c) [4 points] Analyze the expected time and communication costs for solving the coloring problem in this way, including the cost of Luby's algorithm.

Solution: The costs are just those of solving MIS on G'; the final decisions are local and don't require any extra rounds.

Time (number of rounds): The expected time to solve MIS on G' is $O(\lg(n \cdot \Delta))$, because the number of nodes in G' is $n \cdot (\Delta + 1)$. The $O(\lg(n \cdot \Delta))$ bound can be simplified to $O(\lg n)$.

Communication (number of messages): The expected number of messages is $O(E \lg n)$, corresponding to $O(\lg n)$ rounds and messages on all edges (in both directions) at each round.

SCRATCH PAPER

SCRATCH PAPER

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015