- This exam has 5 questions for a total of 135 marks, of which you can score at most 100 marks.
- The deadline for emailing your answers is 14:15 hours on November 21, 2020. Email your answers to: algo2020cmi@gmail.com. **Do not** use this email address for any other communication.
- You may answer any subset of questions or parts of questions. All answers will be evaluated.
- You are free to refer to the lectures/your notes/any other material that you wish, but you are **not** permitted to confer with one another in any manner. Note that "referring" to material does *not* cover **copying** material from elsewhere, except for statements of theorems, lemmas and such. In particular, do not copy arguments wholesale from external sources.
- Warning: CMI's academic policy regarding cheating applies to this exam.

Unstated assumptions and lack of clarity in solutions can and will be used against you during evaluation. You may freely refer to statements from the lectures in your arguments. You don't need to reprove these unless the question explicitly asks you to, but you must be precise. That is, "As we saw in class ... " will not get you any credit, but "As we saw in Exercise x of Lecture y" will. You may assume that comparing any two integers takes constant time. Please feel free to ask us via the designated channels if you have questions about the questions.

Whenever you are asked to design an algorithm, you must make it as efficient as possible. The credit will depend on the running time of your algorithm as well, apart from its correctness.

1. Let G be a connected and undirected graph on n vertices and m edges (that is: $|V(G)| = n, |E(G)| = m$). The *distance* $d(u, v)$ between two vertices u, v in G is the *number of edges* in a shortest path in G between the two vertices. Vertices u, v in G are said to

   1. form a *remote pair* if $d(u, v) > \frac{n}{2}$, and
   2. form a *tenuous pair* if there exists a vertex $x \notin \{u, v\}$ in G such that the graph $G - x$ obtained by *deleting* x from G contains no path between u and v.

   This question has two parts. Part (a) is the "full" question. Part (b) is a lighter variant with fewer marks to score. If you find part (a) too difficult, then you can try the easier part (b). We will evaluate all answers, and marks from (only) the part with the higher score will be added to the total.

(a) This is the "full" version. [25]

i. Describe a modification to the BFS algorithm that takes (i) the adjacency list of graph G and (ii) a vertex u of G as input, runs in $\mathcal{O}(m+n)$ time, and decides whether there is a vertex v in G such that u, v form a remote tenuous pair. Further, *if* there is such a vertex v then the algorithm must output (i) one such vertex v **and** (ii) a vertex $x \notin \{u, v\}$ such that $G - x$ contains no path between u and v.

ii. Prove that your algorithm is correct, and that it runs in $\mathcal{O}(m+n)$ time.

(b) This is the "light" version.

i. Describe a modification to the BFS algorithm that takes (i) the adjacency list of graph G and (ii) a vertex u of G as input, runs in $\mathcal{O}(m+n)$ time, and decides whether there is a vertex v in G such that u, v form a remote—not necessarily tenuous—pair. Further, *if* there is such a vertex v then the algorithm must output one such vertex v. Prove that your algorithm is correct, and that it runs in $\mathcal{O}(m+n)$ time. [5]

ii. Describe an algorithm that takes (i) the adjacency list of graph G and (ii) a vertex u of G as input, runs in time polynomial in n, and decides whether there is a vertex v in G such that u, v form a remote tenuous pair. Further, *if* there is such a vertex v then the algorithm must output (i) one such vertex v **and** (ii) a vertex $x \notin \{u, v\}$ such that $G - x$ contains no path between u and v. Prove that your algorithm is correct, and that it runs in time polynomial in n. [5]

Warning: Read the instructions below before *you attempt a solution!*

You will get the credit for the above problem only if your algorithm and its analysis are correct, *and* it runs within the required bound. You will get the credit for a modification of the BFS algorithm if you either (i) write correct pseudocode for the entire algorithm, *OR* (ii) write pseudocode for BFS and describe *how* to make the required changes to your BFS code (and not merely *what* these changes are) to get an algorithm that solves the problem.

2. A digraph is *connected* if its underlying undirected graph is connected. All digraphs in this problem are connected, and have no parallel arcs or self-loops. We say that a vertex v in a digraph D is a *prince* if there are directed paths in D from v to every other vertex in D. Note that if D is *strongly connected* then every vertex in D is a prince. In this problem we see how to quickly identify princes in digraphs which may not be strongly connected.

(a) Suppose D is digraph that does have at least one prince. Suppose we do a DFS on D starting from an arbitrary vertex; recall that for digraphs we repeat—without [15]

resetting the clock—the DFS from unvisited vertices till every vertex is visited. Let v be a vertex that is assigned the largest *finishing time* by this run of DFS. Prove that v must then be a prince.

(b) Write the pseudocode for an algorithm that takes the adjacency list of a digraph D with n vertices and m arcs as input, runs in $\mathcal{O}(m+n)$ time, and decides whether D has at least one prince. If D does have a prince then the algorithm must return (any) one prince. You *do not* have to prove the correctness or running time bound of this algorithm, but you will get the credit for this part *only* if your algorithm is correct and runs within the required bound. [10]

3. A *cycle-hitting set* of an undirected graph G is any subset $S \subseteq E(G)$ of the edge set of G such that the graph $G - S$ obtained by deleting all edges of S from G is a *forest*. The *weight* $w(S)$ of a cycle-hitting set S is the sum of the weights of all edges in S. [25]

   Write the pseudocode for an algorithm which, given a connected undirected graph G with edge weights $w : E(G) \to \mathbb{N}$ as input, runs in $\mathcal{O}(|E(G)| \log |V(G)|)$ time, and finds and returns a cycle-hitting set S of G *of the smallest weight*. That is, S must be a cycle-hitting set of G, and there must not exist a cycle-hitting set S' of G such that $w(S') < w(S)$.

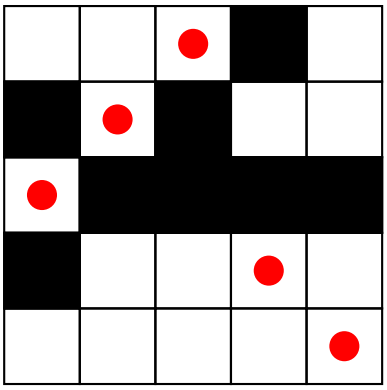
   Prove that your algorithm is correct, and that it runs within the stated bound.

   Note: You will get credit for this question only if your argument for the correctness of the algorithm is correct; you won't get any credit for just producing correct pseudocode out of thin air.

4. You are given an $n \times n$ square grid with some squares black and the remaining ones white. Describe a polynomial-time algorithm *using network flows* to place tokens on the squares of this grid, satisfying the following conditions: [25]

   1. every token is on a white square
   2. every row and every column of the grid contains exactly one token

   Your algorithm should detect if no such placement is possible. The figure on the next page shows an example.



5. You are given n courses. There is a directed edge from course p to course q if p is a prerequisite of q, which means q cannot be started unless p is completed. Each course also has a duration (number of weeks required for completion of the course) associated with it, and different courses may have different durations. Describe a linear-time algorithm to determine the minimum number of weeks required to complete all the courses, if any number of courses can be taken simultaneously as long as all their prerequisites have been met. [25]