# Question 1

To prove that is not **NP**-hard under , a problem will be appropriately chosen, that is in **NP**, and it will be shown that that problem cannot be reduced to . In this case, the chosen problem will be CIRCUIT-SAT, which is already proven to be **NP**-complete. So, it will be shown that CIRCUIT-SAT , which proves that is not **NP**-hard.

CIRCUIT-SAT is the decision problem where given a Boolean circuit, does there exist an assignment of its inputs that makes the output true. In other words, does there exist inputs of and to the circuit that evaluates to . To solve this problem, an algorithm has to try every possible combination of inputs and evaluate the circuit each time to see if it outputs . So, given an input of assignments, although the computation of the circuit to determine the output can be done in constant time, the determination of the inputs is done in exponential time, as the algorithm would have to try every possible combination in the worst-case.

With that being said, now attempt to reduce CIRCUIT-SAT to via . It is stated that the function has the following property: for every , where is the set of all bit-strings. So, in this case, since , then this means that the input to is always a valid bit-string, thus will always evaluate to . This reveals the discrepancy between these two problems: CIRCUIT-SAT will output either or , whereas always outputs . In addition, CIRCUIT-SAT must compute its output based on its input, meaning it must process and determine whether the input is valid, which leads to its exponential time as mentioned earlier, whereas will always evaluate to , leading to a constant time of , putting it in the complexity class **P**.

Since runs in constant time, whereas CIRCUIT-SAT runs in exponential time, it is not possible to say that CIRCUIT-SAT can reduce to . Another perspective: take the bit-string input of , break them up into individual bits, and assign them to some circuit as the inputs. In this case, will evaluate to no matter what the setup of the circuit is, whereas CIRCUIT-SAT may evaluate to or depending on the circuit. Thus, CIRCUIT-SAT cannot reduce to because not only is it more computationally complex than , but it also could result in different outputs for the same input. Therefore, CIRCUIT-SAT , hence is not **NP**-hard.

# Question 2

To prove that the connected components decision problem is **P**-complete under , two properties will be shown:

1. This problem is itself **P**.
2. Every problem **P** reduces to this problem.

To show that this problem is indeed in **P**, a deterministic polynomial-time algorithm will be proposed that solves the problem. To show that it is **P**-hard, a reduction will be carried out from some other problem in **P** to this problem. Proving these two together is enough evidence to claim that this problem is indeed **P**-complete.

To prove (i), the following is a deterministic polynomial-time algorithm (written in C++) that solves the connected components problem:

void dfs(int v, vector<bool> visited, vector<vector<int>> adj) {

visited[v] = true;

for (auto it = adj[v].begin(); it != adj[v].end(); ++it) {

if (!visited[\*it]) {

dfs(\*it, visited, adj);

}

}

}

int countConnectedComponents(vector<bool> visited, vector<vector<int>> adj) {

const int n = visited.size();

int connectedComponents = 0;

for (int i = 0; i < n; i++) {

visited[i] = false;

}

for (int i = 0; i < n; i++) {

if (!visited[i]) {

dfs(i, visited, adj);

connectedComponents++;

}

}

return connectedComponents;

}

bool hasKConnectedComponents(const int connectedComponents, const int k) {

if (connectedComponents == k) {

return true;

}

return false;

}

As mentioned, this algorithm was written in C++ rather than pseudocode to be able to test with various inputs of non-empty undirected graphs and positive integers . The algorithm consists of three functions: , , and . The first function is given a starting vertex, visits it, and then visits every neighbouring vertex that has not yet been visited through a recursive strategy, which results in a depth-first search of the graph. The function initially sets all visited nodes to false, as well as a connected components counter to , and then for each non-visited node, calls the method and increments the counter. This works because the function naturally visits all the vertices that are connected to the starting vertex, so once that function is finished, it is guaranteed that that is one connected component. Finally, compares the number of connected components computed in to the positive integer , and if they are equal then return true, else return false.

Analyzing the time efficiency of this algorithm, it performs a depth-first search through the entire graph , visiting all the nodes once, leading to a time efficiency of , where are the vertices and are the edges of the graph . Thus, this algorithm is linear in time, and in turn, safely upper bounded polynomial in time. Therefore, the connected components problem is proven to be itself **P**, and (i) is satisfied.

To prove (ii), let be any problem in **P**. Now, need to show that reduces to the connected components problem with Definition 1 from Lecture 9(c). Once this is achieved, then this proves that the problem is **P**-hard, and together with (i) showing that it is in **P**, then a conclusion can be reached about the problem, namely that it is **P**-complete under .

Since is assumed to be any problem in **P**, then there exists a polynomial-time algorithm, call it , that given any instance of , correctly outputs true or false. Now, one can leverage to reduce to the connected components problem. Let represent the connected components problem, with being the polynomial-time algorithm discussed above. Furthermore, let be the set of all instances of and be the set of all instances of . Now, suppose has the following decision problem: given the same positive integer from , does there exist a function such that it only outputs true on a certain value equivalent to , and false otherwise. Let this function be .

Now, given the same inputs as , the problem will return true if and only if returns true, and false otherwise. Applying this algorithm to the problem , this will also result in the same output, namely that it will return true if and only if there are indeed connected components in the given graph for problem . This extends for any instance of , where the function will return true for any positive integer that is satisfied by its desired equivalence. Thus, the following property holds: is true if and only if is true. In other words, any instance of evaluates to true if and only if the proposed algorithm evaluates to true when applied to problem , which it indeed does.

Next, it can be proven that the algorithm is indeed polynomial-time computable, which will complete the reduction. The goal of is, given some input, determine whether or not it can achieve some value equivalent to the positive integer common to both problems. Hence, the algorithm must iterate through the entirety of its given input to determine such a value. However, it only has to do this once, i.e., visit each element once, thus also running at linear time in the size of its input. In other words, it does not have to revisit nodes it has already processed. Therefore, it can be stated that is indeed polynomial-time computable.

At this point, the following claims have been proven: (1) is true if and only if is true, and (2) is polynomial-time computable. Thus, it can be said that reduces to , or , where is the connected components problem. Since is represented as any problem in **P**, then this holds in the general case, namely that every problem **P** reduces to this problem, therefore the problem is **P**-hard, and (ii) is satisfied.

Since properties (i) and (ii) are both satisfied, then the connected components decision problem is indeed **P**-complete under .

# Question 3

# Question 4

Example of a valid automorphic mapping:

A picture containing text, clock

Description automatically generated

* So long as at least one vertex is mapped to a different vertex than itself, it is a valid automorphism, because the mapping is no longer the identity.
* is an edge in the one graph if and only if is an edge in the other.

Need to properly modify and , for a polynomial number of times and invoke for each pair. Each modification/transformation should also take polynomial-time.

* Suppose you have and a copy of it . Now, pick a pair of distinct vertices . And now suppose you want to force a mapping of to , and for the other vertices, you don’t care how they are mapped.
* That’s the approach. For every pair of distinct vertices , try and force a mapping of in to in . You know that a valid mapping exists if and only if is automorphic.
* To force a mapping, “hang” something off of each of and that forces any isomorphic mapping to map to only. By “hang something off” we mean construct some kind of graph that forces such a mapping of to only.

# Question 5