Question 1

(a) i) We can build a graph that is unweighted and undirected where the vertices are pictures of dogs. Each vertex will store a field that represents what breed the dog is, if that information is available. If it is unknown, the field is left blank.

ii) In this graph, the unweighted, undirected edges will represent pictures of dogs that are identified as the same breed by a person. This means that in this graph, there will be an edge connecting pictures that were identified as pairs. For example, if A and B were identified by the person as the same dog, then there would be an edge connecting them.

iii) To generate this graph, we can assume that we are given a list of pictures with the correct breed if available for each one, and all the person's pairings. To begin, we can iterate through each pairing, and for a pairing (A, B), if either picture does not already exist in the graph, then we can create them as vertices in constant time. If the vertex is a known breed, we can label the vertex with the known breed in constant time. We can then generate an edge between the two vertices in constant time. Iterating through every pairing will take O(k) time, so this entire process therefore takes O(k) time as well. We will not have to iterate through the pictures as iterating through every pairing will give us enough information to construct the graph. This correctly generates the graph because it ensures that we account for the actual breed and the persons' identification for every picture it exists for- if there are any pictures that the person did not create a pairing for, they cannot be part of an inconsistency.

(b) In our graph, we can identify if we are ever given inconsistent responses by checking if any connected component contains more than one breed. To check for this, we can first create a set of all the vertices in the graph that we know the breeds of, or all labelled pictures and a set of visited vertices that is originally empty (in k time and constant time respectively as iterating through all paired, labelled vertices takes k time). We randomly select an element in the set of labelled pictures, remove it, and add it to the visited vertices set. We then conduct BFS on this vertex, and for every unvisited vertex that we can reach, check if the breed is different than that of the source vertex. If so, then we have found an inconsistency. If not, then we add the new vertex we just checked to the visited set and continue until we cannot reach any more nodes that are not part of the visited set. We then remove another vertex from the set of all vertices repeatedly until we have a vertex that is not part of the visited set. If we find one, we then repeat this operation on it and continue until every labelled vertex has been added to the visited set. We know that there are no inconsistencies as we can ensure that all vertices in a connected component are of the same breed. If there is ever a connection between two differently labelled dogs (through any number of pairings), then they are part of a connected component. We then find that this component contains multiple breeds, regardless of which labelled vertex we start at, leading us to return that there is an inconsistency. This algorithm runs in $O(k+l+p)$ time as in the worst case scenario, we will have to traverse along every edge and vertex, or every pairing and picture.

Question 2

We can build a graph that is unweighted and undirected where the vertices are pictures of llamas or alpacas. The vertices will contain species as labels. In this graph, the edges will represent pictures that are compared by a person. This means that in this graph, there will be an edge connecting pictures that were compared. Each edge will contain boolean data representing if they were paired to be the same or different. For example, if A and B were paired, then there would be an edge connecting them labelled as same/different accordingly.

To generate this graph, we can assume that we are given all the person's pairings. To begin, we can iterate through each pairing, and for a pairing (A, B), if either picture does not already exist in the graph, then we can create them as vertices in constant time. We can then generate an edge between the two vertices and label it accordingly in constant time. All vertices are naturally unlabelled. Iterating through every pairing will take O(k) time. This correctly generates the graph because it ensures that we account for all possible pairings and use our algorithm to check for inconsistencies or under-specifications.

In our graph, we can identify if we are ever given inconsistent responses by checking to make sure that this graph is bipartite. However, there is a modification to the standard 2-color algorithm where we do not simply check that every other level is opposite color, but rather that vertices separated by a 'different' edge are opposite colors and those connected by a 'same' edge as the same color. If we do find that the graph does not fit our modified definition of bipartite, then there is an inconsistency.

To check for this, we can first create a set of all the vertices that have been part of a pairing, a set of visited vertices that is originally empty, and two empty sets representing species A and B. Initializing these sets takes $O(s+d)$ time as we have to iterate through every same/different pairing. We randomly select an element in the set of paired vertices pictures, label it as species A, remove it, add it to the visited vertices set, and it to the correct species set. We then conduct BFS on this vertex, and for every edge coming out of it, check if the label is 'same' or 'different'. If it is 'same', then check that the vertex it connects to is either the same species (A in this case) or unlabelled, and if it is 'different' then check that it is the other species (B in this case) or unlabelled. If the vertex is unlabelled, then label it with the corresponding species and add it to the corresponding set. If the vertex is already labelled and is different from the species we want to label it as, then we have found an inconsistency and can return so. If it is the same as what we want to label it as, then we add the new vertex we just checked to the visited set and continue until we cannot reach any more nodes that are not part of the visited set. We then remove another vertex from the set of paired vertices repeatedly until we have a vertex that is not part of the visited set. If we find one, we then repeat this operation on it and continue until every vertex has been added to the visited set. This will correctly identify inconsistencies because if we ever encounter two paired species that are incorrectly labelled because of another pairing, then this algorithm will identify it. This part of the algorithm will run in $O(s+p+d)$ time because you will have to visit every edge and vertex, or every same pairing, different pairing, and picture.

If we are sure the graph is consistent, to check if there is underspecification, we can check to make sure that the entire graph is one connected component. If this is false, then we can return so. To check for this, we can pick an arbitrary start vertex and run the regular BFS algorithm on it. If at

the end of the runthrough of BFS, the size of the visited vertices set is less than the total number of pictures, p, then the entire graph is not one connected component. In other words, there is underspecification, and we can return so. This algorithm works because if the graph were completely connected, then the size of the visited vertices set after the BFS run through from any vertex should include every vertex, or have size p. If it doesn't, then we know there is underspecification. This part of the algorithm runs in $O(s+p+d)$ time as well because you will have to visit every edge and vertex, or every same pairing, different pairing, and picture.

If the graph is both bipartite and has no connected components that only contain 'same' edges, then there is an exact answer. We can return the two sets that are produced by the 2-color traversal in $O(s+p+d)$ time. The worst case run through of our algorithm is therefore $O(s+p+d)$ because identifying inconsistencies, underspecification, and the exact answer all take this long.

Question 3

(a) Since the graph G is strongly connected, we can reach any vertex regardless of which vertex we start at. Therefore, the path with the maximum possible points is the one where we visit every vertex. One way to do this is to conduct BFS on our start vertex u and when we run BFS, since G is strongly connected, we know we will reach every vertex in the graph, and therefore maximize our point total. This algorithm has a worst-case runtime of $O(V + E)$ because we will have to visit every edge and every vertex.

(b) To begin, we need to find the optimal start point for the path. The best starting point will be from the vertex that allows us to reach all other vertices in the graph. To do this, we run a topological sort on the graph in $O(E + V)$ time and then select the vertex that is first in the topological sort and use it as our source vertex on it. We will need to keep track of all the possible paths that we can take and then select the path that results in the most total points. To do this, we keep track of all the vertices on the current path and the total points so far for every edge added in the DFS. When we backtrack and remove an edge, we remove however many points are lost by removing the destination vertex from the current total. We compare every possible combination of points and keep track of the largest one. After going through all possible paths, we can return the one that produced the highest point total. This can be done $O(E + V)$, the standard for DFS, so our total runtime is $O(E + V)$.

(c) For any directed graph that is acyclic or cyclic, we are given the start vertex u. This means that from the previous part b), we do not need to find the optimal start point for the path as it is given to us. From here, we can effectively repeat the algorithm but from the fixed start vertex. We will need to keep track of all the possible paths that we can take and then select the path that results in the most total points. To do this, we keep track of all the vertices on the current path and the total points so far for every edge added in the DFS. When we backtrack and remove an edge, we remove however many points are lost by removing the destination vertex from the current total. After going through all possible paths, we can return the one that produced the highest point total. This can be done $O(E + V)$, the standard for DFS, so our total runtime is $O(E + V)$.

Question 4

(a) There can't be a forward edge because in an undirected graph, forward and back edges are treated the same way and both will be part of the DFS tree or will be treated as backward edges. There can't be cross edges because if node A is already explored, if A and B are connected, then node B would have been explored when continuing DFS from A.

(b) We will prove this by contradiction by assuming that there is a cut edge e in an undirected graph G that is not a tree edge. By the definition of a cut edge, this means that removing e will result in there being more connected components. In other words, if e connects two nodes u and v, then now u and v are no longer part of the same connected component and have no way of reaching each other now. Since u and v were in two separate components once e was removed, this means that the only way to get from one component to other was across e. Since e is also not a tree edge, then e should not appear as one of the edges traversed in the DFS of G. However, if the only way to get from any one component to the other is across e, then to explore the entirety of the graph with DFS, e must appear in the DFS tree. Therefore, e is a tree edge, so any cut edge must also be a tree edge.

(c) start begins at 1 u.start –¿ the level of u v.start –¿ the level of v where (u,v) is a back edge–¿ the minimum of all back edges d.low –¿ low of some descendant of u that is calculated later –¿ the minimum of all descendants Basis of recursion based DFS to solve this because need to use the fact that u know the low of all descendants of u to get low of u

THE DFS code can be modified by using the recursion-based DFS rather than the stack-based DFS. When starting, we can measure the start field of our first vertex as 1 and retain that as the value will not change. We can then set the low field equal to the minimum of start, the minimum start value among all neighboring nodes that are connected by back edge (there are none for the first vertex), and the minimum result of the DFS calls on each of its direct neighbors (this is equal to the minimum low value of all its descendants). We then return the low field so that after every DFS call, we are returning the value that can be compared to the node's start and the minimum back edge's vertex start and then be set to the node's low. If we repeat this until we visit every node in the graph, we know the source vertex's low value.

Vertex	Start	End	Low	Back-Edges	Descendants
a	1	18	1	None	b,c,d,e,f,g,h,i
b	2	17	2	None	c,d,e,f,g,h,i
c	3	16	2	None	d,e,f,g,h,i
d	4	11	4	None	e,f,i
e	5	10	4	None	f,i
f	6	9	4	(f-d)	i
g	12	15	2	None	h
h	13	14	2	(h-b)	None
i	7	8	7	None	None