Joakim Rosenqvist
Assignment 3
1DV516 -HT19

Exercise 1

I will split the analysis for directed and undirected graphs, since my implementation varies. However addVertex() and addEdge() are the same on both.

Undirected

addVertex: Vertices are stored in an ArrayList, as objects of type Vertex. The constructor of Vertex assigns an id of generic type and creates an ArrayList that stores adjacent vertices. These are all constant time operations, giving the addVertex() method a time complexity of **O(1).**

addEdge: Since I've opted to do the generic implementation, I have to find the corresponding vertices in the ArrayList, meaning I perform a linear search operation on it. When that is done, an object of type Edge is created and added to the adjacent lists of both vertices. An Edge object holds references to two vertices. Construction of Edge object and addition of it into the vertices' lists are constant time operations. Searching for the two vertices is linear time. Overall complexity of the operation is **O(N).**

isConnected: This method starts at the first vertex in the list of vertices, checks for adjacent vertices, and recursively visits them if they are not previously visited. Visited vertices are added to a HashSet. When every reachable vertex has been visited, the size of the HashSet is compared to number of vertices in the graph.

The HashSet is initialized with a size of 2N to ensure no rehashing needs to be done. I have verified that every vertex has a unique hashcode, which according to Java SE 8 API documentation ensures O(1) time-complexity for add() and contains(). For every visited node the list of edges needs to be iterated through, but since the number of edges in an undirected graph has to be smaller than the number of vertices, we can disregard this and focus on how many times the method recursively is called, which is a worst-case of **O(N).**

isAcyclic: This method iterates through every unvisited vertex, and calls on a recursive boolean method that will call on itself for every unvisited adjacent vertex, with the current vertex as a parent. If somewhere in the recursive tree, a node finds a visited adjacent that is not its parent, it will return true, and this means a cycle has been found.

As with the previous method, a HashSet is used to keep track of visited vertices, and it will at maximum have a workload of 0.5, meaning it will not have to rehash. The operations used on the set are O(1). The recursive method will be called on a maximum of N times, giving a time complexity of O(N). I will also disregard the iteration over list of edges since it will always be smaller than N. Overall the time complexity of this operation is **O(N).**

connectedComponents: A HashSet is used to keep track of visited vertices in each connected component. As before the operations on this set will be O(1) time. The algorithm will start by iterating over the list of vertices. When an unvisited vertex is found, it will check if there is a previous component that needs to be added to the return list. If so, it will iterate over the set of connections and add their id's to a list of type T, and then add that list to the return list, and clear the

set holding the component. A search for new connected vertices starting from the unvisited vertex will then begin, using the same recursive BFS method as isConnected. After every vertex has been examined, the last component will be added to the return list.

The recursive BFS method will visit every vertex once. Every vertex will then have to be iterated over one more time when the component lists are created. This gives us 2N + C, or **O(N).**

Directed

isAcyclic: This method has two HashSets, one to keep track of visited vertices, and one to keep track of recursion history. It will iterate through every vertex and call on a recursive method that adds the vertex to visited and its recursion set. For every incoming edge it will recursively call on itself with the source of the incoming edge and ultimately return true when an edge is reached that visits the vertex without having been visited at the time of first level recursion, but also is part of the recursion set. HashSet operations are O(1) for this application. Every recursive call checks first if the vertex has been visited before, and only further evaluates unvisited vertices. This means every vertex should only be visited once, for an overall time-complexity of O**(N).**

connectedComponents: This method makes use of a HashSet and a Stack. It iterates over every vertex and sends unvisited vertices to a recursive helper method that examines its outgoing edges and calls on itself recursively for every unvisited vertex. Every vertex is added to the set of visited in this method, and after the recursive call is finished, it is pushed to the stack. This ensures that the stack contains vertices with components grouped together.

A recursive assignment method is then used to create the component lists. An iterator pops the stack and for every unvisited vertex the assign method is called and adds every vertex that is reachable through incoming edges to a component set. After every reachable vertex has been found the set is added to a list, which in turn is added to a list of lists.

The first phase, when vertices are added to the stack, contains several constant time operations and the iterator and recursive method only visits previously unvisited vertices, making it O(N). The second phase again only visits unvisited vertices, but again iterates over them when added do their component lists. Overall we have O(3N), which is simplified to **O(N).**

isConnected: Since a strongly connected graph requires every vertex to be part of a single component, this method simply returns if connectedComponent returns a list smaller than 2. Since this is completely reliant on the time-complexity of connectedComponents, this method is also **O(N).**

Exercise 2

I am under the assumption that a graph needs at least two vertices of non-zero degree to contain a valid Euler path, so my solution will reflect that.

hasEulerPath: This method will iterate over the list of vertices and count degrees. To return true, I require two vertices of non-zero degree, and either only vertices of even degree or exactly two of odd degree. This operation iterates through the list of vertices once for a time-complexity of **O(N).**

Joakim Rosenqvist
Assignment 3
1DV516 -HT19

eulerPath: The algorithm starts with choosing a good start point, which in case of odd vertices will be one of those, or in other case it will choose a vertex of non-zero degree. The start vertex is pushed to a stack. An iterator will iterate while stack has elements, and take the top element and iterate over its edges. If an edge has not been traversed before, it will be added to the list of used edges, its first vertex pushed to the stack and its second vertex becomes the new "current". After every connecting edge of a current vertex has been fully explored, it will be added to a list that makes up the path. Finally the path list is iterated over and the vertex id:s are added to a new list which is then returned. In total, this method will iterate over the full list of vertices one time, and then two times for what is a maximum of number of edges. Since edges in an unconnected graph are always a maximum of N-1, we can simplify them as O(N) as well. The operations on set, stack and list are constant time. Total time is 3N + C, or **O(N).**

Exercise 3

numberOfPeopleAtFriendshipDistance: This method iterates through the list of vertices to find the requested vertex, since the implementation is generic. It then calls on a method that returns a set of people at requested distance. That method in turn calls on another method which takes a vertex and a distance, and generates sets of people for every distance up to the requested distance.

The way that method works is by keeping track of visited nodes through a HashSet, taking the start vertex and add all its adjacent vertices to a set representing distance 1. Then it visits every vertex in that set and adds every unvisited adjacent vertex into a set representing distance 2, and so on until it has created sets up to the requested distance. All the operations on set and list are constant time, or amortized constant time. Since the method at worst will only have to iterate over every vertex and every edge (which is N – 1), time-complexity of the operation is **O(N).**

furthestDistanceInFriendshipRelationships: Since there is no way to know beforehand when to stop, I make use of the same helper methods as in previous method, and check for when the size of a distance is 0. A worst-case scenario here is a straight graph, meaning for N inputs, it will require N calls of a method that will go from 1 to N-1. Overall this means a time-complexity of **$O(N^2)$.**

possibleFriends: This method is always a distance of 2, so it will use the above mentioned helper method to obtain a list of sets for distance 1 and distance 2. It will then iterate over the set of distance 2, check its adjacent vertices and compare to the set of distance 1 vertices. When a distance 2 vertex has an adjacent in the distance 1 set, we know they are a common friend and this is counted. When every adjacent of a vertex has been checked for common, they will be added to a list of possible friends if they have 3 or more commons. The list of possible friends is then returned after every vertex in distance 2 has been visited. Generating the sets is as mentioned before O(N), and so is iterating over the vertices and adjacents of that set. Operations on the sets are constant time. Overall this operation has a time-complexity of **O(N).**