1) Describe tradeoffs among possible implementations of the Graph data structure, including the implementation that you provided and at least one other possible implementation. Your comments should refer to the Big-O complexity of Graph operations, and to the possible implications that this may have for running your program on very large graphs.

> There are two main implications of Graph: matrix (two dimensional array) implementation and list implementation. Matrix implementation can execute addEdge, removeEdge, and hasEdge in constant time, and inEdges and outEdges take O(n) time. However, the space complexity is not efficient, especially when there are not many edges; it takes n^2 of space no matter what. On the other hand, the list implementation can be more efficient with the space complexity and can be implemented for specific cases by using different data structures for the lists, including stack, DLLis, and array. At the same time, inEdges takes O(n+m) time while other methods are executed in constant time or O(deg(i)). Matrix implementation also has an application by using mathematical properties of matrices, such as calculating shortest paths between two nodes by matrix multiplication.
> In this program, the methods most frequently used are outEdges(i) in reachable(i,j), addEdge, and hasEdge. Especially, outEdges(i) is called in a while loop for BFS every time reachable (i,j) is called, increasing the proportion of the total time complexity. Therefore, the list implementation that allows outEdges(i) to be executed in the constant time and addEdge(i) seems preferred. However, if we expect the number of the edges to be increasing at a higher rate as we increase the size, matrix implementation may function well by having less extra space in n^2 space and allowing addEdge(i) to be more efficient since I am using DLList as a list implementation.

2) Although you were instructed not to change the code in Main, randomlyAddUntilReachable() could be implemented in different ways. Can you see how the task that it is performing could be accomplished more efficiently? Discuss.

> One issue is that the more edges we add, the more likely that the edge newly produced already exists. Therefore, although the result of the simulation does not change by explicitly avoiding the duplicates, the efficiency would increase especially if we use a bigger number of vertices. Additionally, the randomly created edge may have the same starting and ending node, making the program less efficient. This can also be explicitly prevented (although it would have less effect than the first point due to the smaller probability).

3) What might simulations of the sort that your program is performing tell us about some real-world process? Give at least one example of a real-world process for which this might be an appropriate model. What do the results of the simulation suggest with respect to that process? If modifications to the simulation would produce a more accurate model of the process you have chosen, then describe them.

> I imagine that this simulates any random walks quite well. One specific example I think of is a particle scattering within a system, for example electrons or photons inside a star. In a star, energy is produced at the center of the star due to nuclear fusion, emitting in the form of photons. These photons will eventually escape from the star as radiation, yet they scatter quite randomly by hitting other atoms and particles (precisely speaking, energy gets absorbed and re-emitted since

photons are concurrently waves), taking much more time than what speed of light would predict. This simulation gives how many scattering steps it takes to move from center (source) to the outside (destination) in the most randomized situation. The time it takes can also be calculated if the number density of atoms and/or mean free path are known.

To improve this simulation and approximate even closer to the real-life events, we need to implement the forces that photons experience within a star. There is gravity pulling them inside and stronger radiation pressure pushing them outward, resulting in faster escape than a total random walk would predict. The number of vertices can be controlled depending on the number density of atoms within a star, making the simulation even more accurate.