CSE 101 Winter 2014 Implementation Project Final Report

I. Team Member Names

- **Member 1:** Jaiswal, Atyansh (A98055205)
- **Member 2:** Nicholson, Connor (A10913032)

II. Project Design

- **Implementation Platform-**
  - **Programming Languages-** Java for everything.
  - **IDESs Used-** IntelliJ for Deliverables #1A, #1B, #1C. Standard Vim for #2A, #3A. Using an IDE was just personal preference b/w coding partners nothing was used outside of the Java Standard Library.
  - **Machine performance specs-**
    - We used two different machines.
    - Deliverables #1A, #2A, #3A were calculated using the following specs-
      - 1.7GHz dual-core Intel Core i7
      - 8GB 1600MHz DDR3
    - Deliverables #1B and #1C were calculated using the following specs-
      - 3.5GHz quad-core Intel Core i7-3770K
      - 24GB 1600MHZ DDR3

- **Deliverable #1A**
  - **Algorithmic Approach-** We created our own Graph class, created a factory method to generate Graphs of size n with random edges. We then ran DFS on the graph and used that to count the number of Connected Components.
  - **Challenges Encountered-** No challenges encountered. Everything was really simple.
  - **Observations and Inferences-** The value of the mean number of CCs drops very quickly to 1. This is because the possible number of edges is much higher than the number of (n choose 2 number of edges). For example, in case of n = 1000, there are C(1000, 2) number of possible edges, which is 499500 amount of possible edges. Even with p = 0.02, there are 9900 expected edges, which will probably create a connected graph for 1000 edges. The standard deviation is high when the mean number of connected components is somewhere between 0 and n, otherwise when p = 0 or when p is sufficiently large that number of CCs is close to 1, then standard deviation in minimized because most trials have the similar values of number of CCs.
  - **Runtimes –** Short for small values of n. Almost 15 minutes for n = 1000

- **Deliverable #1B**
  - **Algorithmic Approach-** After having found the connected components through the method in 1A, we divided the edges into separate lists based on the connected compoonent. Once we had separate lists of edges, we sorted them based on their weight, and then used Kruskal's Algorithm to find the MST, and then subsequently added the weights of all the edges. Challenges 1B: After making the optimization of ending Kruskal's after it found |E|-1 edges, no challenges were found.
  - **Challenges Encountered-** No challenges encountered. Everything was really simple.
  - **Observations and Inferences-** Average MST weight obviously starts at 0 when there are no edges. It should reach its maximum quickly, as there is a tradeoff between dividing the weight of connected components which is small initially, but it is outweighed by the number of connected components (Which decreases rapidly as probability rises and are used to divide into the weight of the connected components). As the number of nodes increases, the spike and drop becomes more and more dramatic due to connecting all the components more quickly.
  - **Runtimes-** Short for small values of n. Approximately 1 hour for a 1000 nodes.

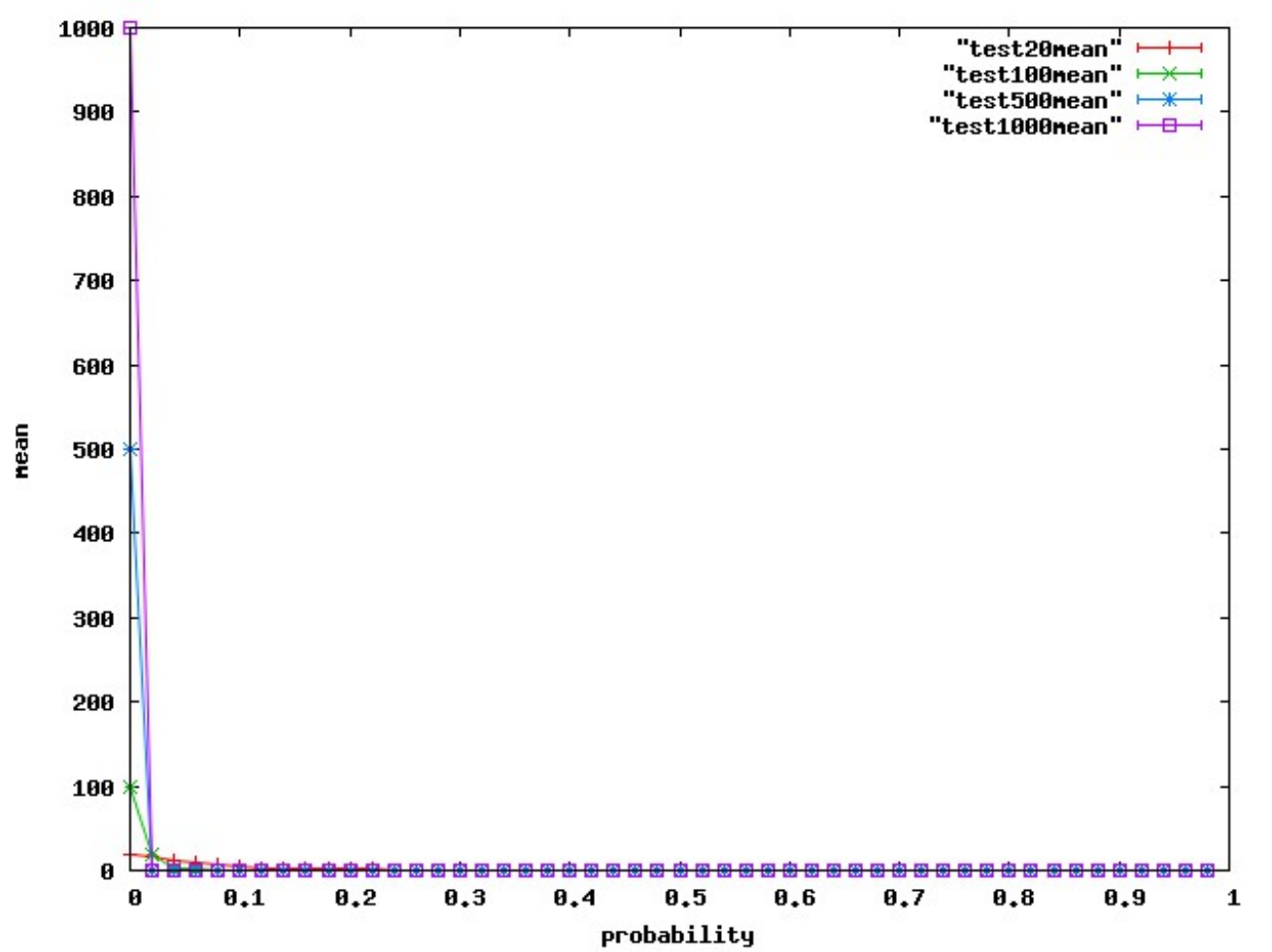
- **Deliverable #1C**
  - **Algorithmic Approach-** Since all edges for this part can be counted as having weight 1, we initialized the distances between any nodes with an edge connecting them to '1', then we iterated over all distances found so far. For each distance found, we iterated over all edges connected to the end vertex of the last-updated distance. (Effectively, a modified BFS whereby all edges are the starting points)
  - **Challenges Encountered-** Due to the high runtime of this algorithm, we divided all probabilities to be run into separate threads to take advantage of all four cores in computations.
  - **Observations and Inferences-** In the beginning, the MST diameter starts low due to the fact that for each connected component, the MST is rather small and thus has a small diameter. As probability increases, there are fewer connected components, ending at zero, thus it stabilizes due to there eventually being only one connected component. As N increases, the amount of probability needed for having only one connected component decreases, thus making the time to stabilization smaller.
  - **Runtimes-** Short for small values of n. Approximately 5 hours for n = 1000.

- **Deliverable #2A**
  - **Algorithmic Approach-** We created our own Point and PointSet classes to simulate a PointSet and added a Factory method to create a random Pointset. We created our own Convex Hull algorithm that implements Graham Scan to create a Convex Hull of the pointset. We then removed the convex hull points and iterated again until all points were deleted. We made helper methods like turn to calculate "left" or "right" turn using simple vector mathematics, we also implemented our own Comparator to sort based on angle.
  - **Challenges Encountered-** No challenges encountered. Everything was extremely simple.
  - **Observations and Inferences-** The graph rises slowly. This is because as n rises, probability of number of points being on the convex hull also increases to some amount so a higher percentage of points are on the hull, leading a slower increase in iteration.
  - **Runtimes-** Ran extremely quickly. Took approximately second.

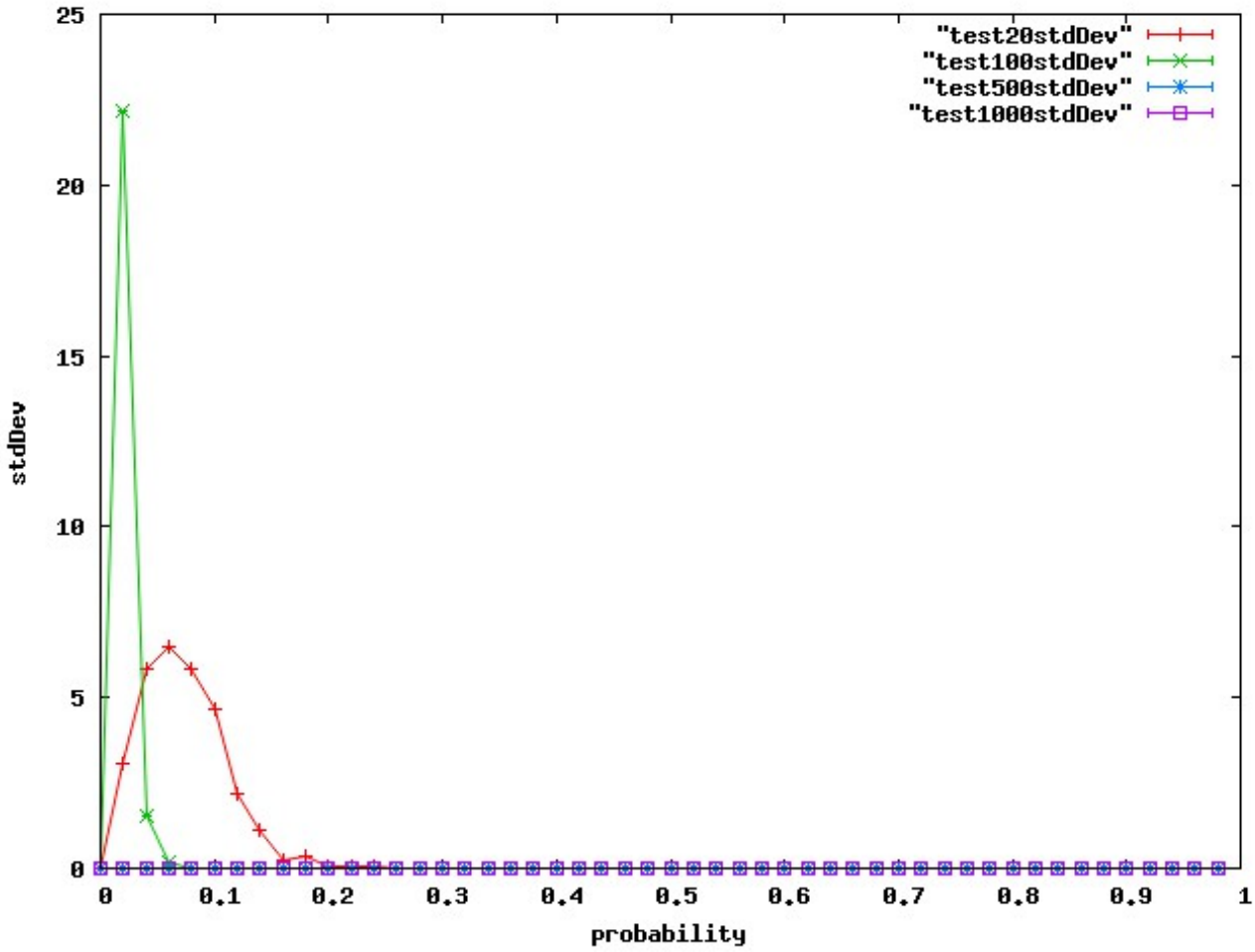
- **Deliverable #3A**
  - **Algorithmic Approach-** We used the same PointSet and Point Classes from #2A and made a small change to it. The factory method was edited to make sure the points were present in the annulus. After that, it was a simple deal to run the trials and calculating the mean and standard deviation of the number of iterations.
  - **Challenges Encountered-** No challenges encountered, everything was extremely simple.
  - **Observations and Inferences-** The number of iterations decreases when value of r increases. This is because as r increases, annulus gets tighter, therefore less space points to be generated, leading to more points in the convex hull.
  - **Runtimes-** Ran moderately. Takes a few seconds for each value of r.

III. Results

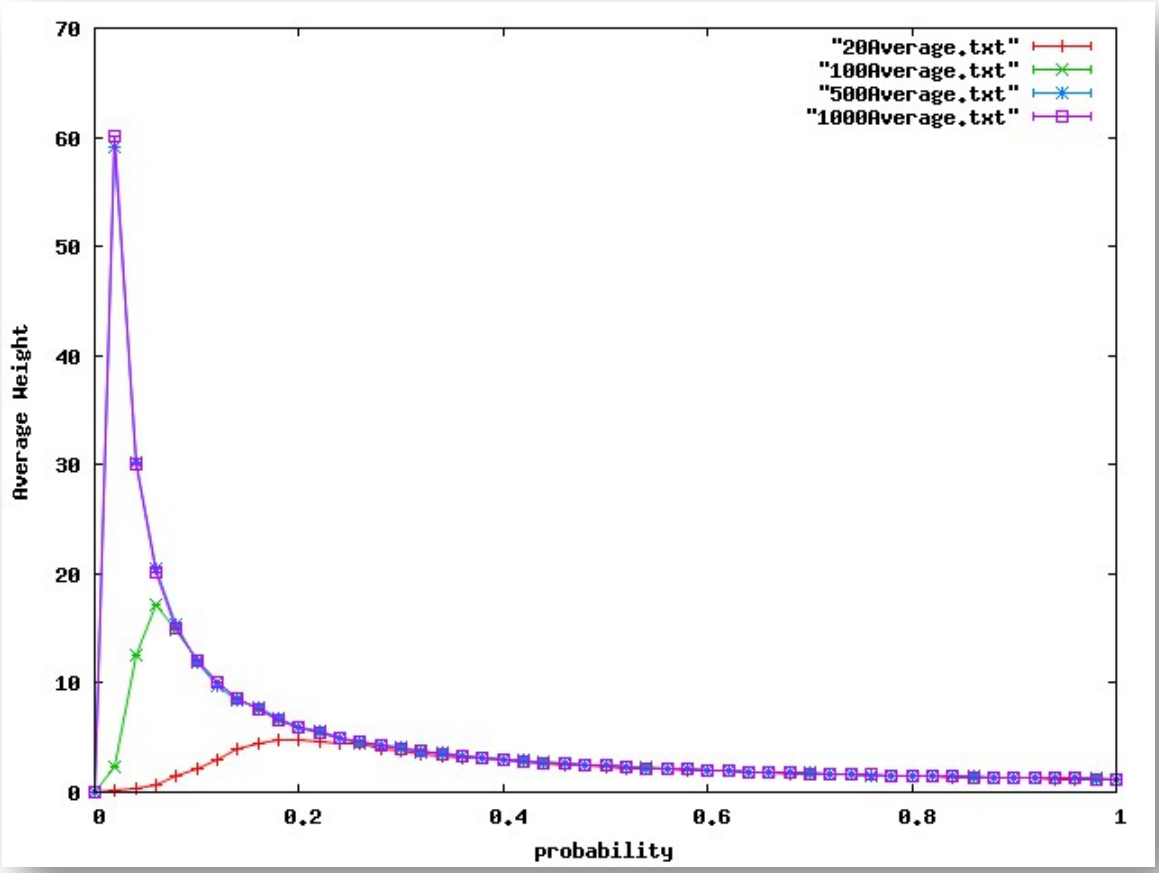
- **Deliverable #1A**
  - **Plot 1- Mean vs. Probability**



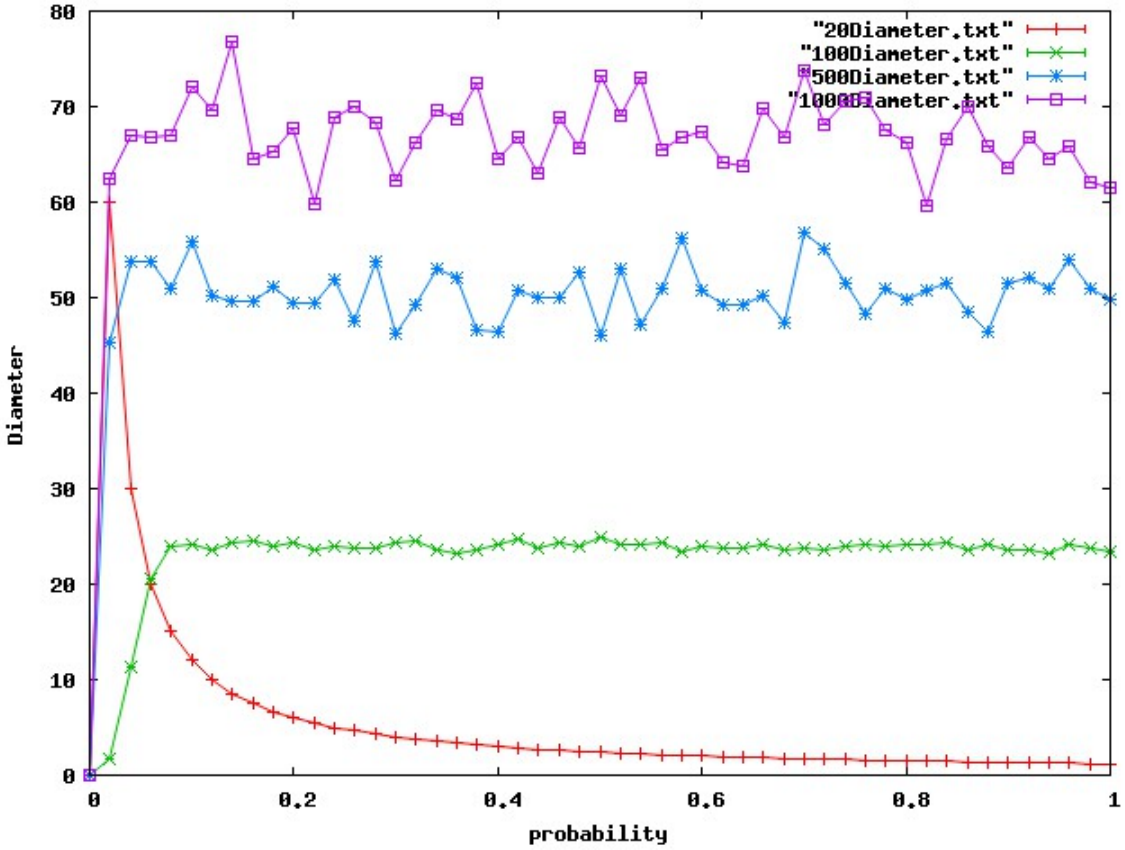
- **Plot 2- StdDev vs. Probability**



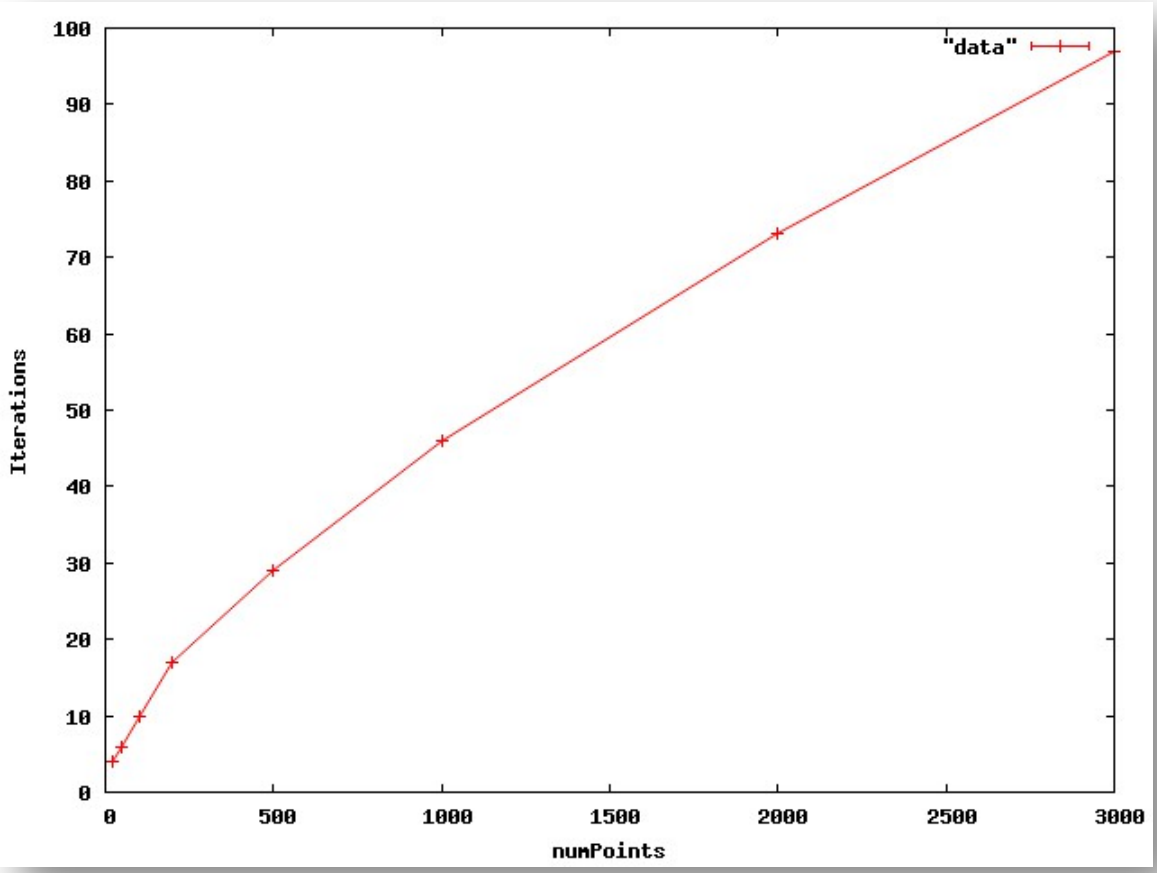
- **Deliverable #1B**
  - **Plot (Average Weight vs. probability)**



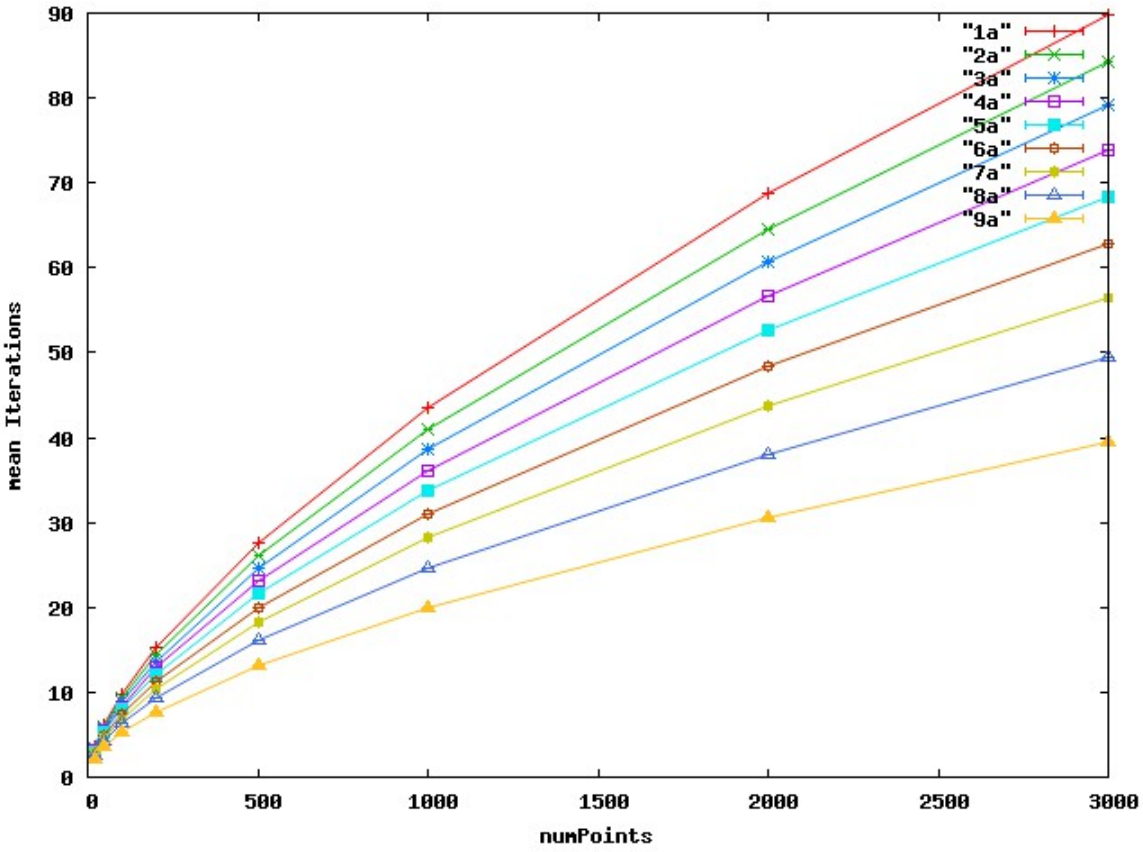
- **Deliverable #1B**
  - **Plot (Diameter vs. probability)**



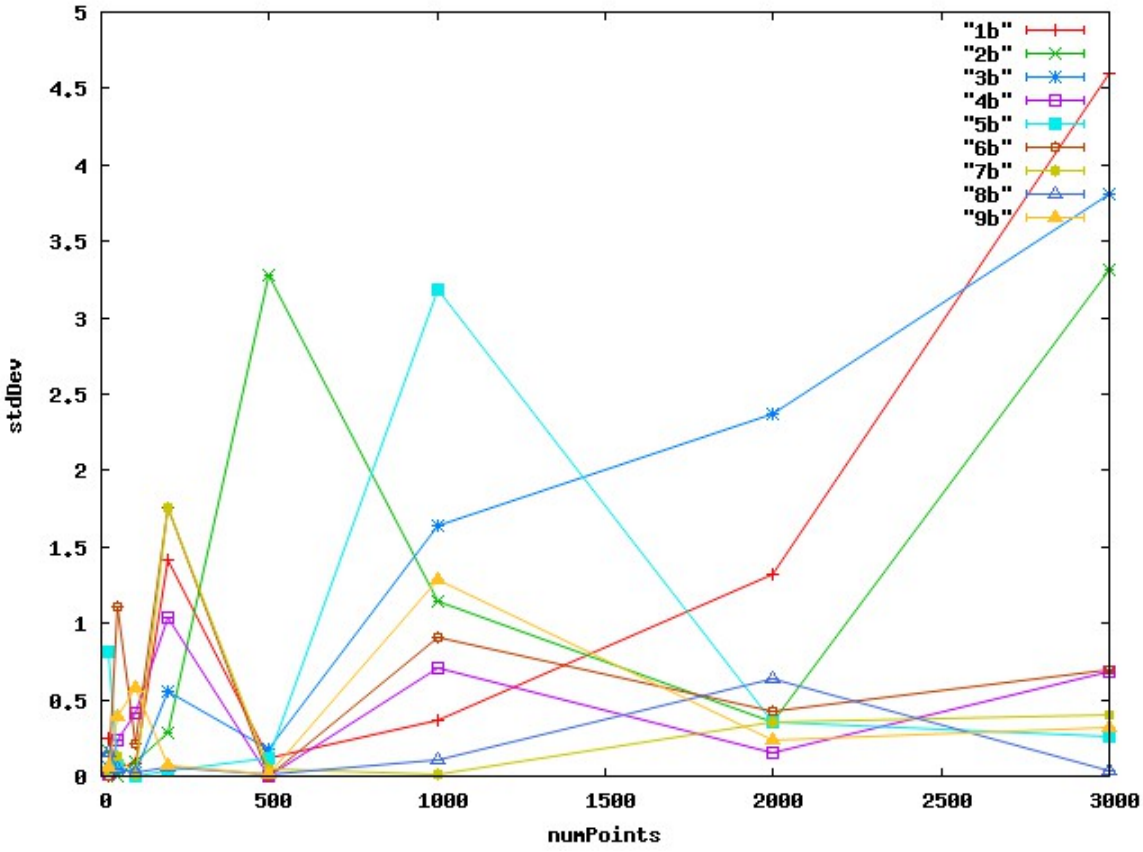
- **Deliverable #2A**
  - **Plot (Iterations vs. number of points)**



- **Deliverable #3A**
  - **Plot 1 (mean iterations vs. n) for different r values**



- **Plot 2 (stdDev iterations vs. n) for different r values**



IV. Acknowledgments

a. Everything was done using only Java and the Java standard Library.

b. We used no external libraries, no external research papers. Even algorithms like Kruskal's and Graham Scan were implemented from knowledge learnt in 101 and 21 lectures.