Conference Session Scheduling Project

Part 1

By Ryan Cole

Summary:

The goal of this work was to provide information about conflicts for a conference that can be used to create a schedule. This conference is very large, with up to 10,000 sessions offered to up to 100,000 attendees.

We estimated that each attendee is planning on taking a fixed number of sessions. These are drawn from one of four distributions: uniform, two-tier, skewed, or the Beta probability distribution with parameters alpha=2 and beta=16.

Two methods are used to eliminate duplicate conflicts created from these samples. An adjacency matrix is used to eliminate conflicts using O(N x N) space, and an adjacency list is used to eliminate conflicts in O(M) space.

After eliminating conflicts, 3 methods are used to color the graphs such that no connected nodes have the same color. This coloring could be used to create a schedule because all sessions with the same color do not conflict with each other and are safe to schedule at the same time. The three methods chosen were the “Smallest-last ordering” from Matula and Beck (JACM, 1983), a “Naïve Largest First” ordering that is like Smallest Last but uses only the initial degrees of vertices, and a random ordering.

Computing Environment:

A laptop with an Intel® i7-7600U @ 2.80GHz, with CPU and 16GB of RAM @ 2.4 mHz, running Windows 10 was used to perform all computations. All code was written in Python and executed either in a Jupyter notebook for plotting or via the command line. Frequently, timing results from the command line were generated and stored on disk to be used in Jupyter.

Definitions:

N = Number of Sessions, max = 10,000

S = Number of Attendees, max = 100,000

K = Number of Sessions Per Attendee, max=10,000 or 1,000 for skewed and Beta distributions

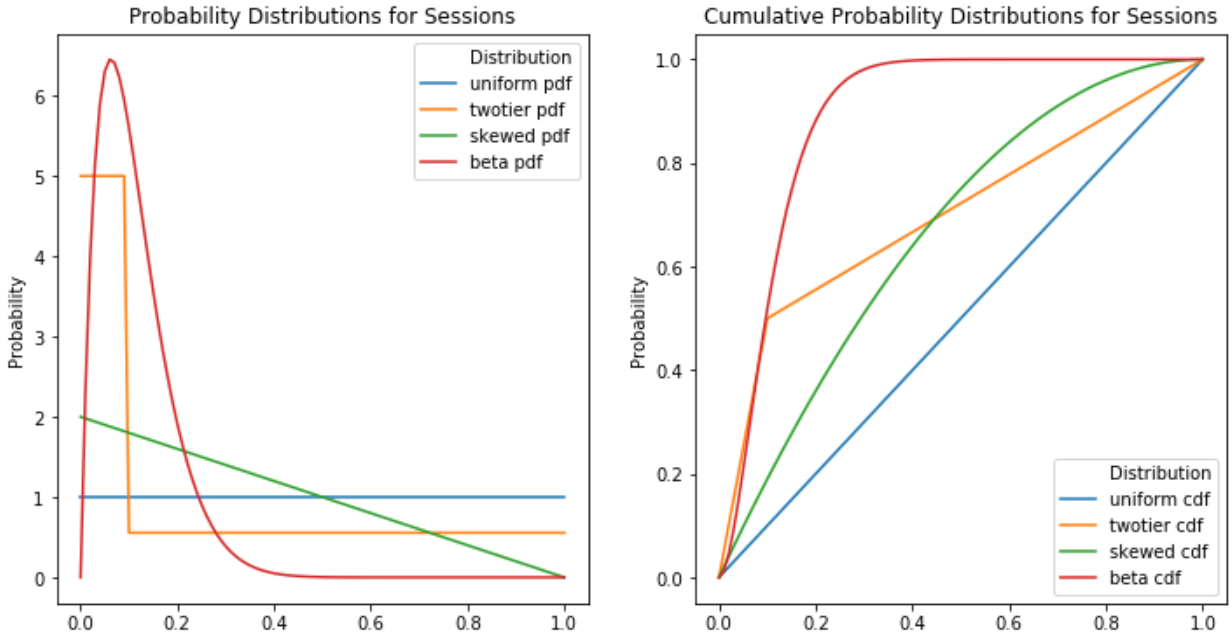
M = Number of Distinct Conflicts

T = Number of Conflicts

Distributions:

Uniform, two-tier, and skewed (linear) distributions were provided to describe how a session number should be sampled from the range 0 to N-1. A uniform distribution assigns an equal probability to all sessions. The two-tier distribution assigns 50% of the probability to the first 10% of sessions, and the remaining 50% probability to the last 90% of sessions. The skewed distribution assigns a probability that linearly decreases to zero for the last session. We choose the Beta distribution as the fourth option. These distributions, and the selected Beta distribution are shown in Figure 1 below. In Figure 1 (left), the height of the Y axis is the relative chance that a particular value on the x-axis will occur.

Figures 1 and 2: Probability Distributions for Sessions

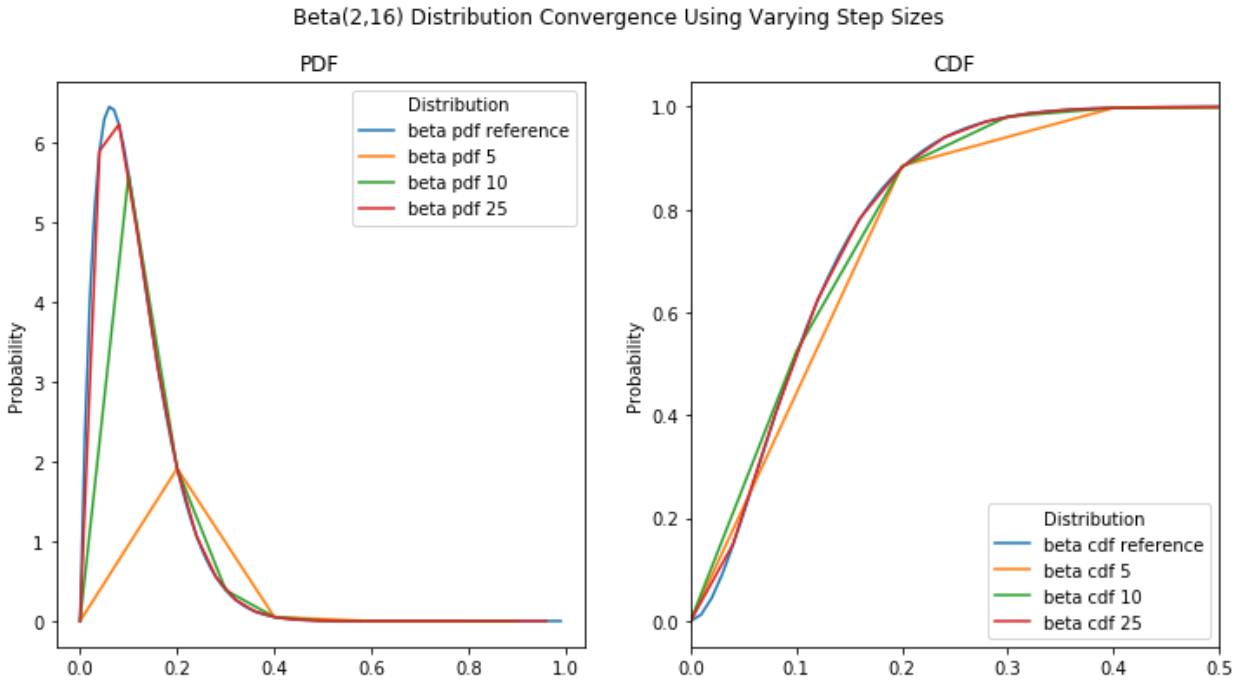


For the uniform and skewed distributions, it is possible to create the probability using a pseudo-random number and simple logic. However, the skewed and Beta distributions require some way to map a random value to an output x. To do so, it is helpful to consider the cumulative probabilities of these distributions, as depicted in Figure 2 (above, right). These cumulative distribution functions (cdf's) are the integral of the probability density functions (pdf's) shown in Figure 1. For the skewed distribution, this integration was performed analytically using the linear function that describes the skewed probability distribution.

The Beta distribution was chosen with parameters that make it analogous to the tiered distribution provided. This distribution trends asymptotically towards zero probability, and may mirror the behavior of a large conference where many conflicts exist for a small subset of sessions, and many sessions will not be in high demand.

The pdf of the Beta function is not integrable analytically, so a simple grid approximation of the curve is used to estimate the cdf. A reference Beta distribution is also obtained from the scientific computation package *scipy* and is compared to the beta distribution created for this project in Figures 3 and 4.

Figures 3 and 4: Beta Distribution Approximation

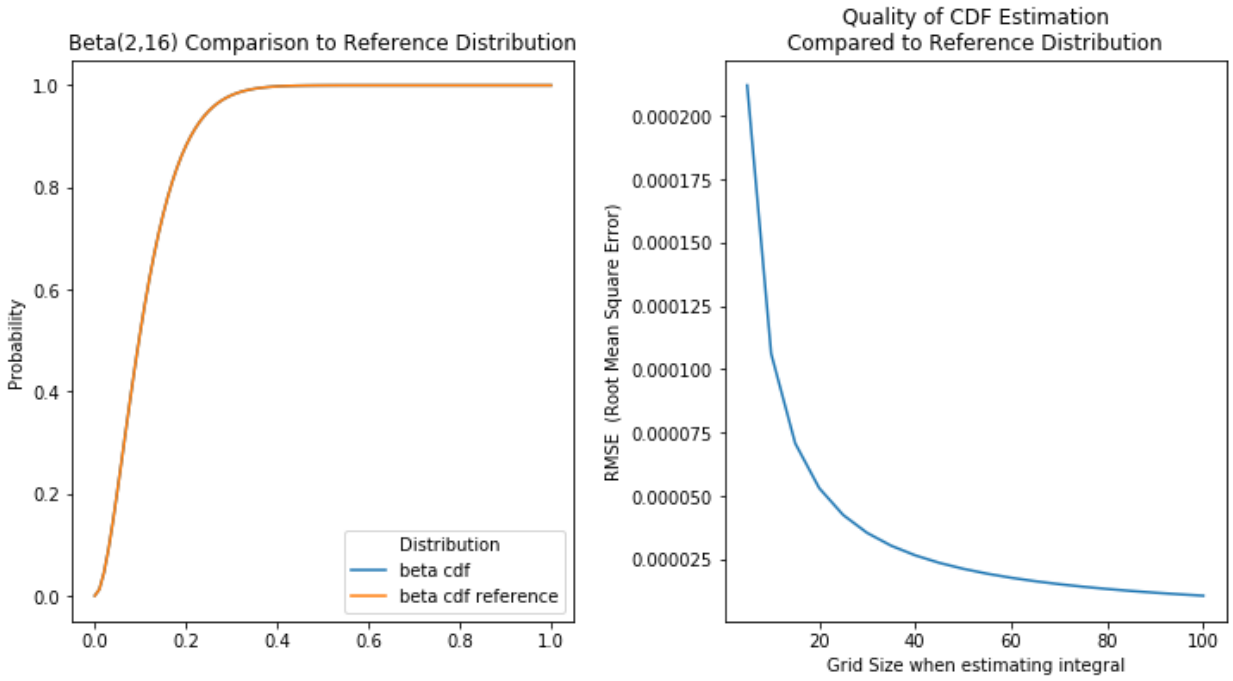


Figures 3 shows the closeness of the fit for the Beta pdf, and Figure 4 (above, right) shows the closeness of the approximation for the Beta cdf, at varying step sizes. The pdf of this function has an analytical form, so it is expected that the points are identical.

The Beta approximation and the reference Beta are shown in Figure 5 (below, left). The reference and approximated cdf curves are too congruent to differentiate in Figure 5, which was created uses a grid of 100 points in between calculated values of the beta pdf to approximate the integral. Figure 6 (below, right) depicts the Root Mean Squared Error, or distance between the reference and approximate

distributions, for a variety of grid sizes. The small RMSE here is evidence that the approximation is sufficient, and we proceed using a grid size of 50.

Figures 5 and 6: Beta Distribution RMSE



It is possible to obtain a sample from the Uniform and Skewed distributions directly, since a sample can be returned directly from one or two pseudo random number generations. To sample the Skewed and Beta distributions, it is necessary to create the intervals of the cdf. These can be created once and then sampled repeatedly. Creating these distributions has the space and time complexities shown in Table 1.

Table 1: Creating Distributions

Distribution	Space Complexity	Time Complexity	Approach
Skewed	O(N)	O(N)	Symbolic integration
Beta	O(N)	O(N x Grid_Size)	Numeric integration

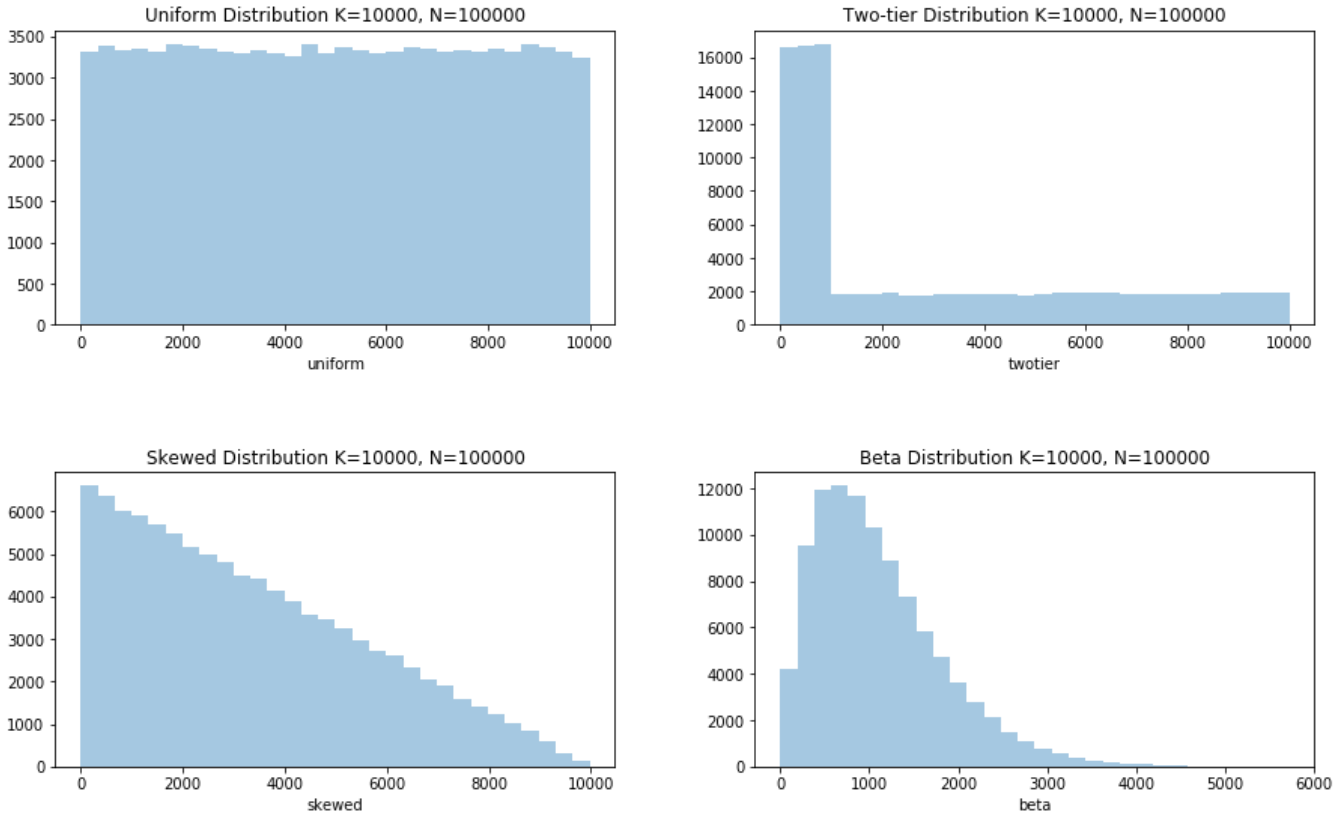
The asymptotic performance of sampling the distributions is included below. Initially a custom binary search was used to map a random number (Y) to the appropriate session (X) on the cdfs for the skewed and Beta functions, but a 2x speedup was observed when using the *bisect* python package. This speedup is likely the result of the non-recursive form of binary search used in the *bisect* package.

Table 2: Sampling Distributions

Distribution	Space Complexity	Time Complexity	Approach
Uniform	O(1)	O(1)	Logic
Two Tier	O(1)	O(1)	Logic
Skewed	O(N)	O(lg N)	Binary search on list
Beta	O(N)	O(lg N)	Binary search on list

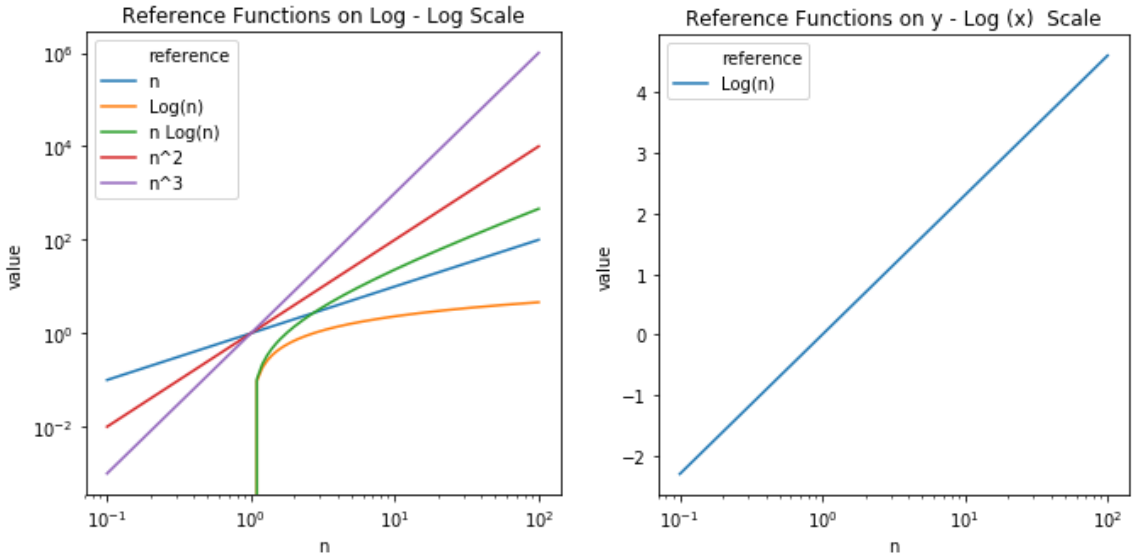
Figure 7 below shows the result of sampling each of the distributions 100,000 times. These figures correspond with the pdfs depicted in Figure 1. Note that the samples of the Beta distribution do not include any number above 6000, due to the extremely low probability of drawing one of these sessions.

Figure 7: Sampling the Distributions



Figures 8 and 9 below show a selection of several functions when plotted on a log-log scale, for reference.

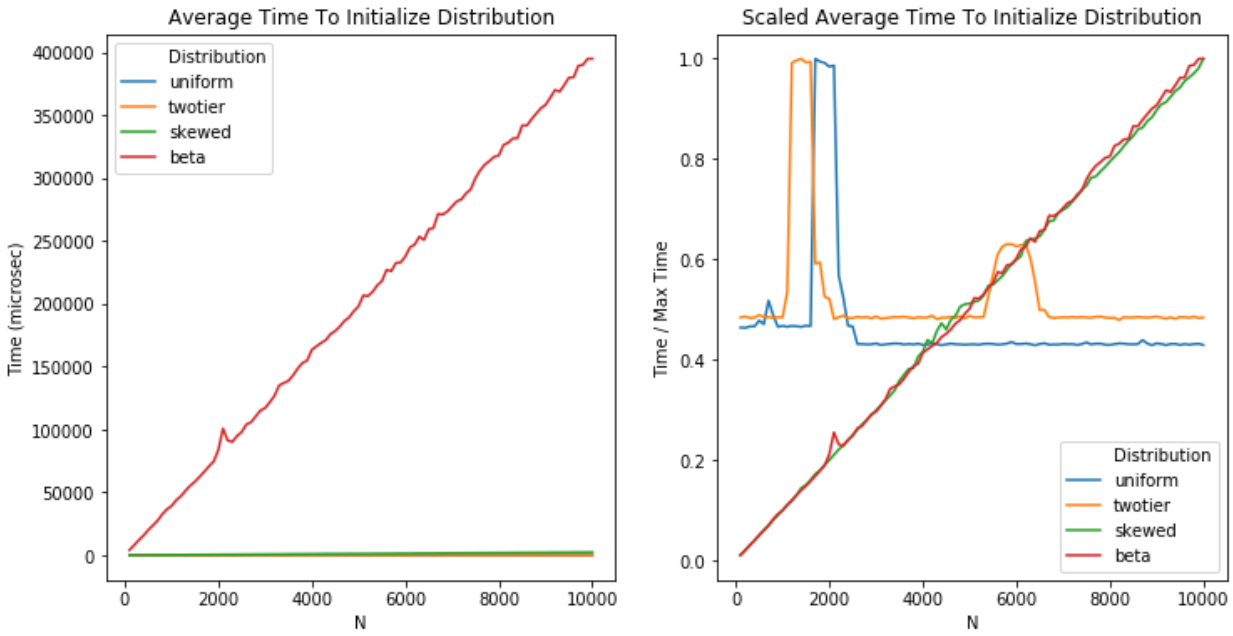
Figures 8 and 9: Log-Log and Log



It is notable that the n^2 and n^3 functions are linear on the log-log scale, and Log(n) is linear on the log-linear scale.

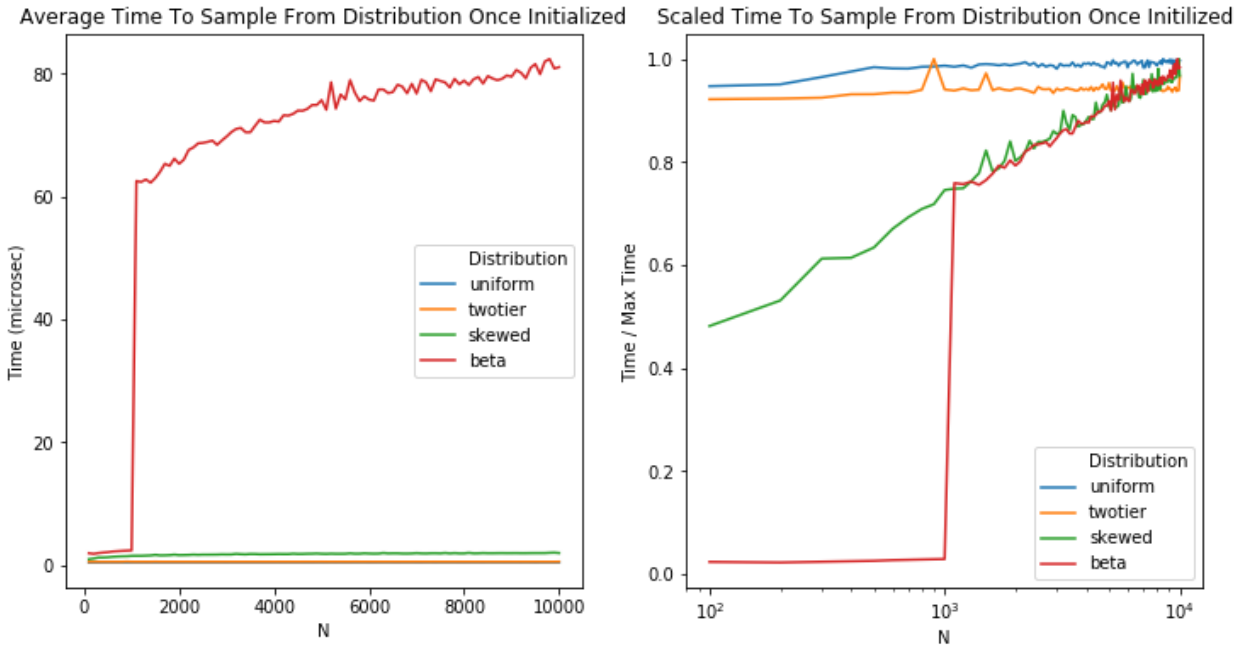
Figure 10 (below, left) depicts the time required to initialize the distributions before they can be sampled. The Uniform and Two-Tier initializations are just placeholders so the functionality of the underlying classes matches for all distributions. Figure 11 (below, right) shows the same data but rescaled using the maximum time for each distribution. The Skewed and Beta initialization is observed to be linear in this rescaled plot, and the Uniform and Two-Tier distributions are constant, excepting some behavior likely due to the real-time behavior of the operation system. It is notable that initializing the beta distribution takes significantly more time than the skewed distribution, due to the complexity of integration. Sampling the Beta distribution is a common statistical task, and it is very likely that the reference *scipy* package will significantly outperform the library written here.

Figures 10 and 11: Initializing Distributions



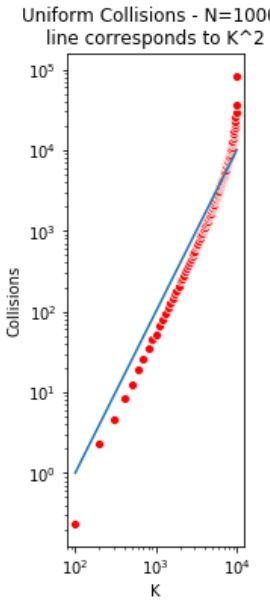
Figures 12 and 13 (below) show the performance of drawing a single sample from these distributions with respect to N after initialization has occured. These timing values do not include the O(N) operation required to build the mapping for the skewed and Beta distributions. Here the rescaled data is plotted in Figure 13 on a log-linear scale. This enables the easy observation of log(n) behavior for the Skewed and Beta distributions.

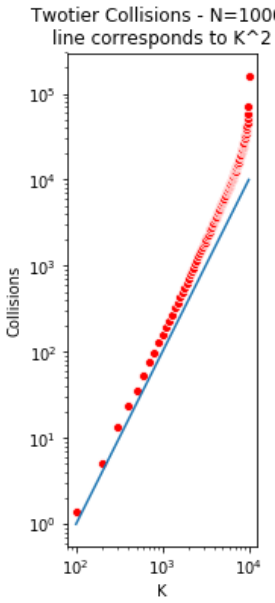
Figures 12 and 13: Sampling One After Initialization

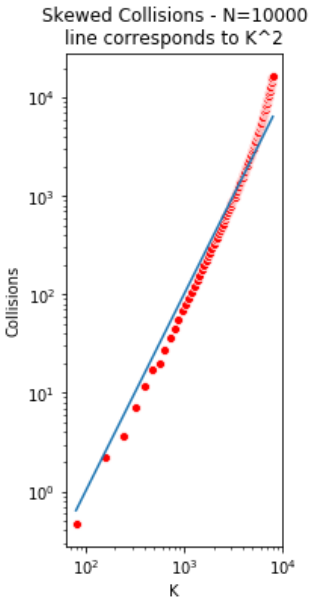


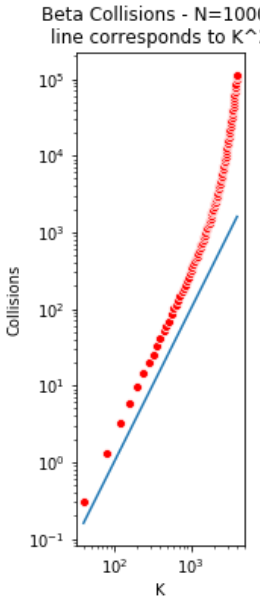
In the plots contained in Figure 14 we consider the number of collisions that occur since each collision results in a resample at a fixed or log(n) cost. This happens because it is required that no session is selected multiple times for a particular attendee. The number of collisions that occur is directly proportional to the time required to generate K sessions, because each collision requires taking another O(1) or O(log(N)) sample. The plots in Figure 11 depict the exponential relationship between collisions and K, in log-log scale with a blue line plotted corresponding to n^2.

Figure 14: Number of Collisions









Note that the Uniform distribution eventually behaves exponentially as the number of samples K approaches the number of sessions N, where the line of red dots skews upwards. The Two-Tier and Skewed distributions begin to behave exponentially at slightly lower values of N. The Beta distribution is particularly affected, because such a small proportion of the probability exists at higher N.

In the case of the Beta and Skewed distributions, this exponential behavior could be alleviated by re-mapping the intervals of the cdf to exclude those already drawn once a certain proportion of the sample locations were chosen, at the same O(N) cost of initialization. This “re-initialization” strategy could also be applied to the Uniform and Two-Tier distributions. This approach is not necessary for this project because although the problem statement allows K values to be up to N or N/10, K values must

be kept small or the scheduling problem becomes trivially hard. This phenomenon is discussed in more detail below.

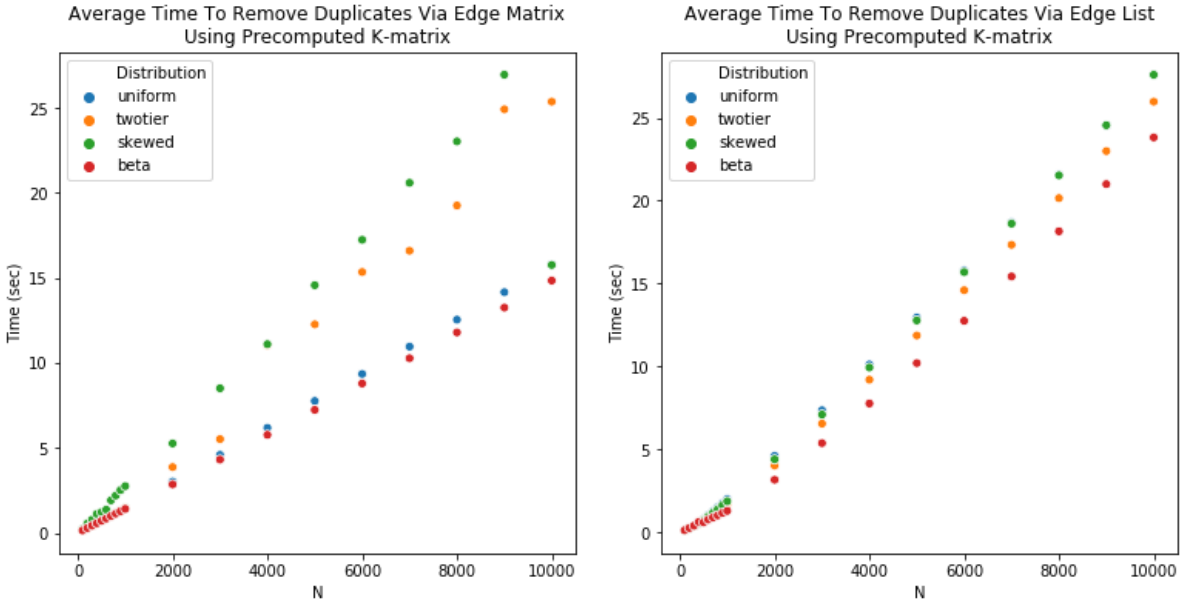
Adjacency Matrix and List:

An NxN adjacency matrix is selected as one of the ways to remove duplicates. In this structure, the element at the i^{th} row and j^{th} column is marked as True if there is an conflict (or edge) between the i^{th} and j^{th} sessions, else it is false. Although this matrix is symmetric, it is implemented as a full matrix to reduce the complexity of indexing into the array and avoid having to compare the sizes of the edges.

An adjacency list is also used to remove duplicates. This structure consists of an array of length N, where each element in the array is a sorted list with O(log(n)) insertion and search time. The *sortedcontainers* package for python was used for this sorted list. The package implements a sorted list using binary trees and was selected to obtain a high performing, robust sorted data structure with O(log(n)) behavior.

To compare these two approaches, a set of test functions were written to use a precomputed set of K-draws. Figures 15 and 16 below depict the time required to process the precomputed S sets of K draws for using the adjacency matrix (left) or adjacency list (right). Here, S = 10 x N, and K = 15 for all distributions.

Figures 15 and 16: Time to Remove Duplicates by N - Using Precomputed K-samples

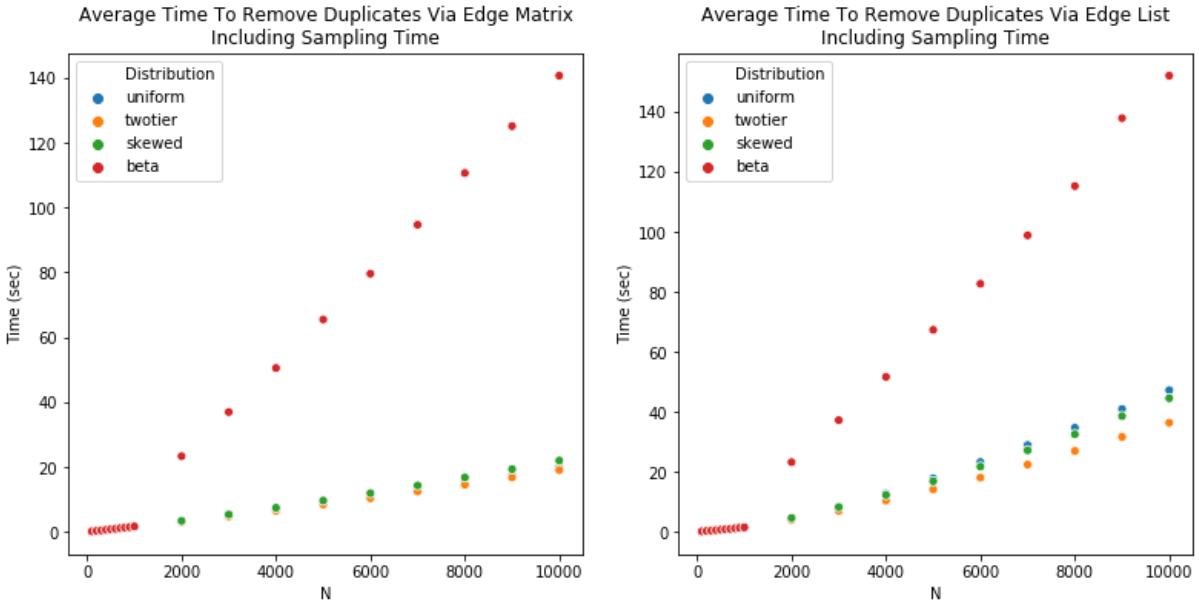


The adjacency matrix performance is shown in Figure 15. The Beta distribution has the best performance here because the algorithm checks if a duplicate conflict exists and performs no further work if it does. Since the draws for the Beta distribution are similar to one another, the program avoids changing as many values in the matrix. This appears to perform slightly faster than when using the Uniform distribution, which should have the least number of redundant conflicts. The behavior to remove duplicates using the adjacency matrix is O(N).

Figure 16 shows the adjacency list performance. Note that the Uniform distribution and Skewed Distributions have very similar performance here. The behavior is O(log(n)), although the skewed distribution appears to have very similar performance using both the matrix and list approaches. Here, the uniform distribution has the worst or second worst behavior, since the lack of redundant conflicts causes more data to be written to the list when using the Uniform distribution compared to the Beta.

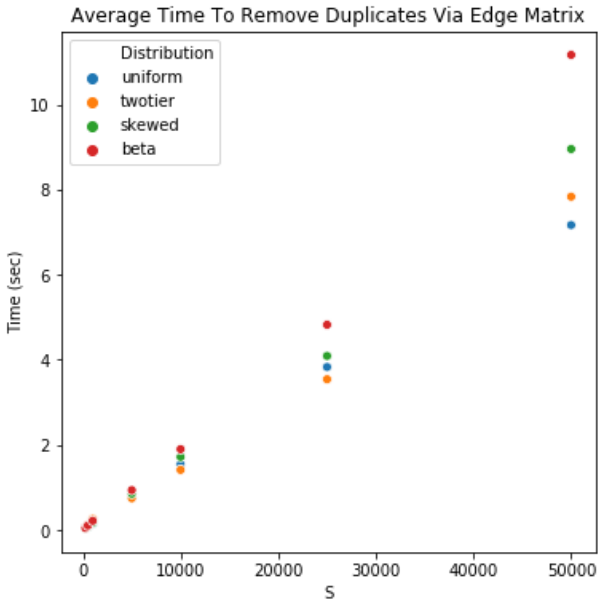
The next figures show the behavior when removing duplicates, including the sampling time. The total time to complete the process is increased substantially, and the cost of sampling from the Beta distribution adds substantially to the time required for completion. The adjacency matrix outperforms the adjacency list when including sampling time, as expected.

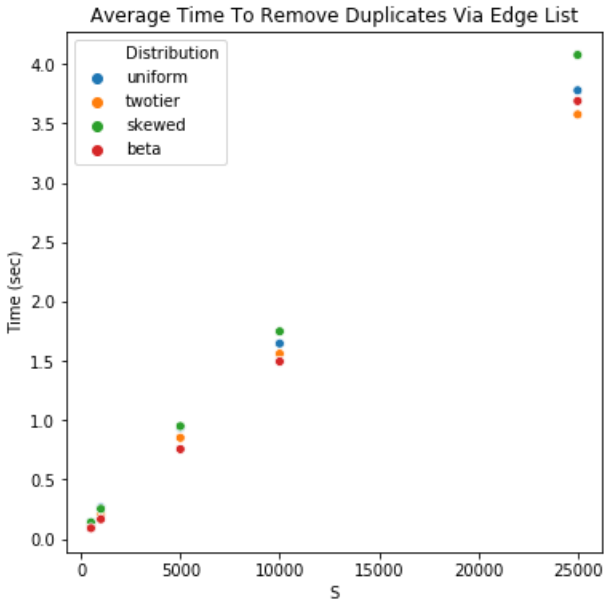
Figure 17 and 18: Time to Remove Duplicates by N - Including Sampling Time



Figures 19 and 20, below, show how the behavior varies with S. Here, we fix N=1,000 and leave K=15. The program appears to perform linearly with S for both adjacency matrix and list.

Figure 19 and 20: Time to Remove Duplicates by S - Including Sampling Time



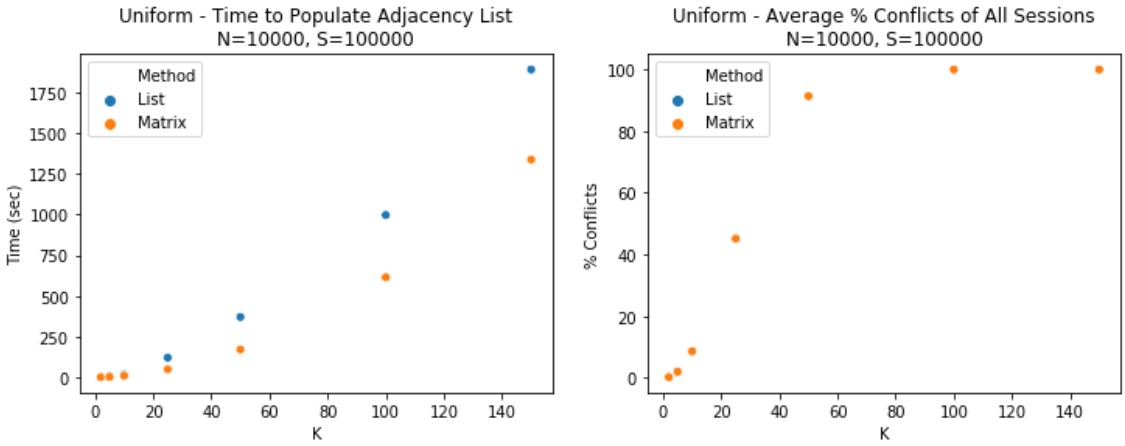


The behavior as K varies is shown in the following series of plots. As discussed previously, the scheduling problem quickly becomes trivially hard as K increases for most distributions. This is because as K increases, the underlying graph quickly becomes fully connected, with every session conflicting with every other session. This results in a trivially hard problem that can be solved only by scheduling every session one after the other, which is obviously not a reasonable solution for a conference with 10,000 sessions to schedule.

Figure 21 shows the behavior for both adjacency matrix and list with N and S equal to the maximum conditions in the original statement using the Uniform distribution. Note that the average percentage conflicts between sessions approaches 100% quickly around K=100. For reference, K was well over 300

before the exponential behavior was observed in Figure 14. We observe the adjacency matrix outperforming the adjacency list, but both methods have identical % Conflict values because they produce identical. In the following plots, K=15.

Figure 21: Time to Remove Duplicates by K and Average % Conflicts – Uniform Distribution



Figures 22-24 below show the same plot for the Two-Tier, Skewed, and Beta Distributions.

Figure 22: Time to Remove Duplicates by K and Average % Conflicts – Two-Tier Distribution

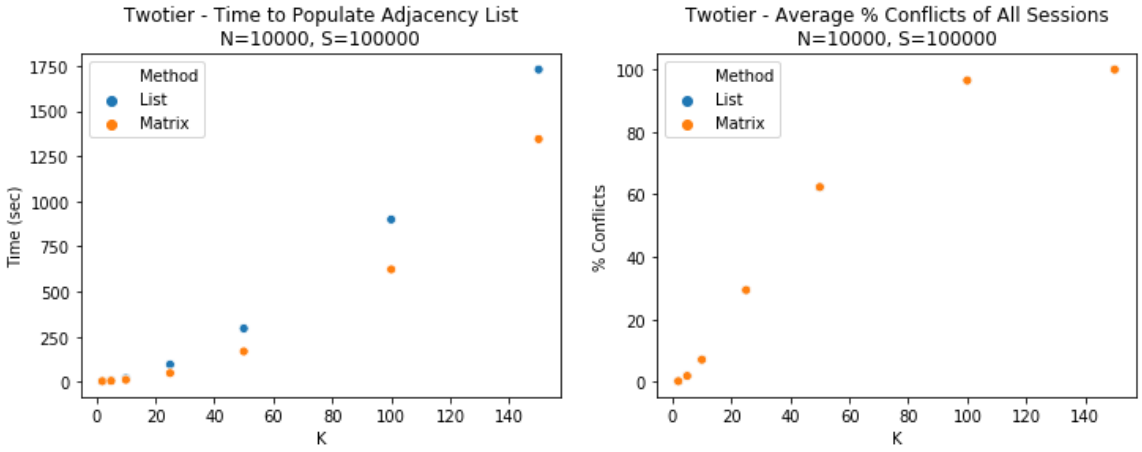


Figure 23: Time to Remove Duplicates by K and Average % Conflicts – Skewed Distribution

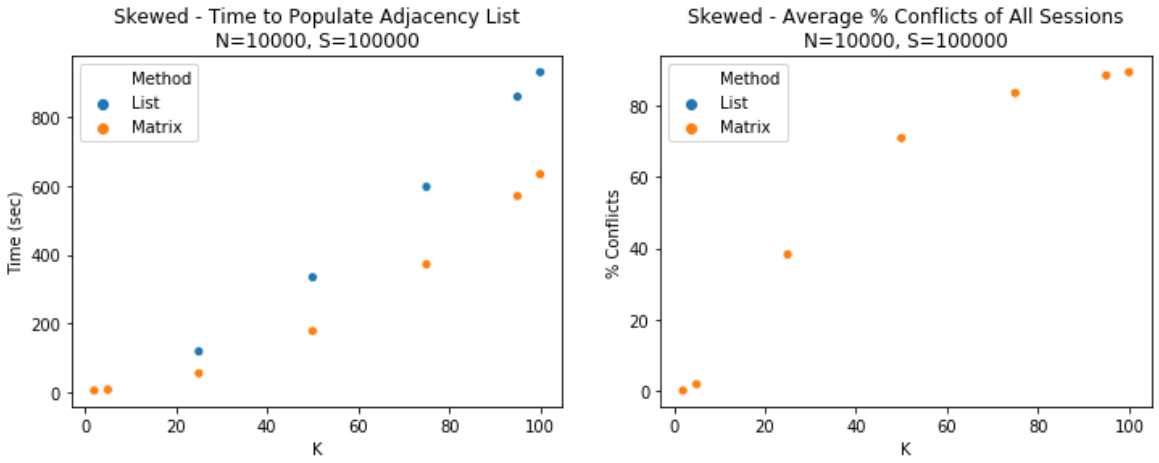
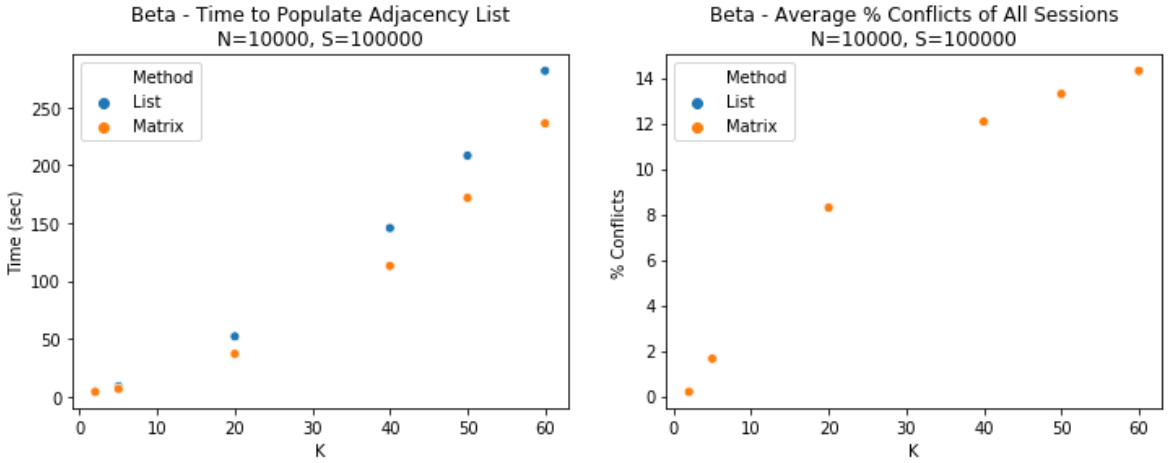


Figure 24: Time to Remove Duplicates by K and Average % Conflicts – Beta Distribution



The Skewed and beta distributions do not approach 100% coverage because the sessions at the tail end of these distributions have a very small probability density. However, these distributions are not robust against this problem but instead have a subset of fully connected sessions within the larger set.

We will proceed with the assumption that the number of sessions chosen by each attendee is significantly smaller than the bounds listed in the original problem (we used K=15 for the plots above). This allows us to avoid spending time analyzing the trivially hard, fully conflicted scheduling problem.

It would be an interesting extension to model the number of sessions chosen K as a variable number using historical data, but that is outside the scope of this work. Even if that were the case, a hard cap on K could be determined as the number of sessions to be scheduled per day times the number of days for the conference.

Coloring The Graph

To this point we have used an adjacency matrix or list to remove all duplicate session conflicts from S attendees who each choose K sessions to attend from a total of N possible sessions. To achieve our goal

of creating a reasonably good schedule for this conference, we will use these structures to color the resulting graphs.

When coloring the graph, we require that each node have a different color than all adjacent nodes. Creating an optimal graph coloring is an NP-hard problem and iterating through all possible subsets to find the optimal is not feasible given the range of N in this problem. Instead we will use several algorithms to select the order in which sessions are colored: Smallest Last ordering, a similar naïve Largest First ordering, and a Random ordering.

The Smallest Last ordering is from Matula and Beck (JACM, 1983), and requires the creation and maintenance of several auxiliary structures to achieve good performance. For this implementation, a list of hashes, one hash for each degree, is used to get O(1) performance for search, insert, and delete of nodes as they are selected and processed. An additional list of the current degree of the vertices was used to allow O(1) identification of which hash should be searched for a particular session. This implementation did not achieve linear performance.

The Largest First ordering is a simplification of the Smallest Last that does not recompute the degree of vertices as adjacent nodes are processed.

The Random ordering simply processes nodes in random order regardless of their degree.

Once the ordering is determined, it is an O(N log(A)) task to assign the colors, where A is the average number of conflicts per session. This process requires that for each session (N) we create an ordered by color list of the A sessions in conflict. We then iterate through the colors to find the first color not used by the adjacent nodes.

Although we choose to use a matrix on the backend for most operations, the following output shows the more concise adjacency list. A verbose output that shows the steps for the Smallest Last ordering is available in the appendix, if desired. Sample orderings for a small problem are included in the appendix.

Figure 25 below shows the time required to generate a Largest First ordering (left) or ordering and coloring (right). Generating the ordering is O(N) in time, and the coloring is presented in log-log scale and is $O(N^2)$. The performance of the Random ordering and coloring is similar.

Figure 25: Time to Generate Largest First Coloring and/or Ordering

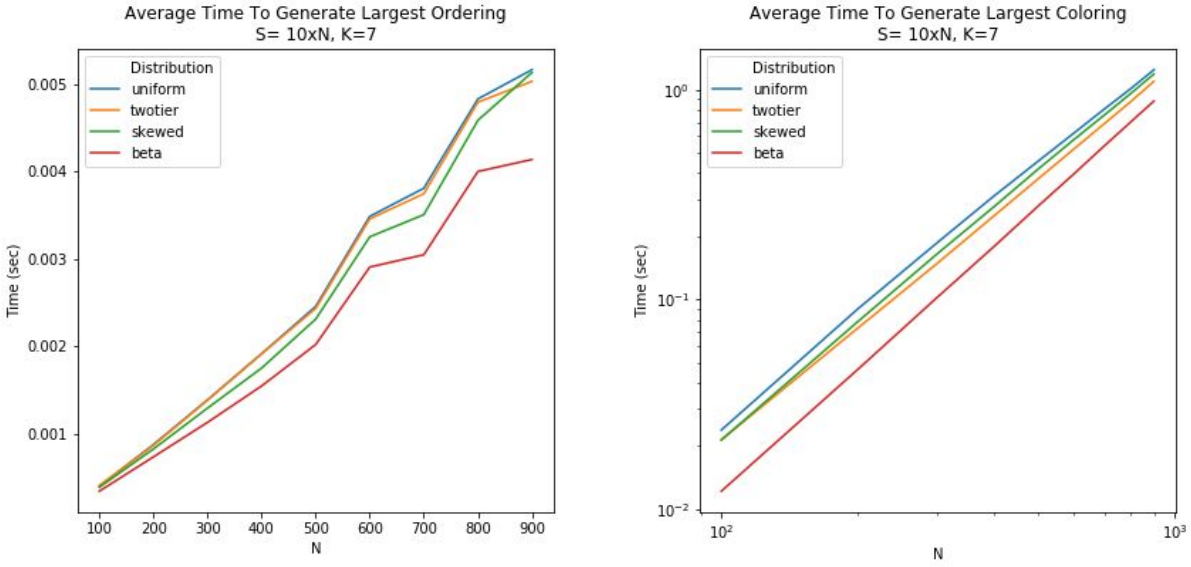
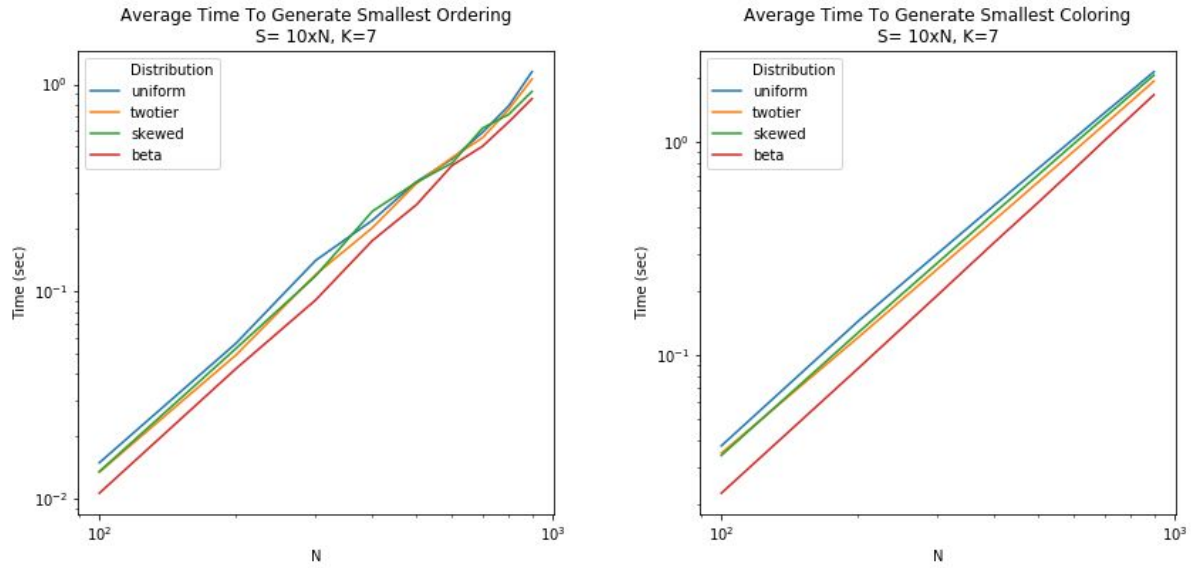


Figure 26 depicts the performance using the Smallest Last algorithm. Note that both plots are presented on the log-log scale, and for this algorithm both appear to be $O(N^2)$. This is likely because the algorithm is O(N x A), where A is the average degree of the sessions and appears to be proportional to N.

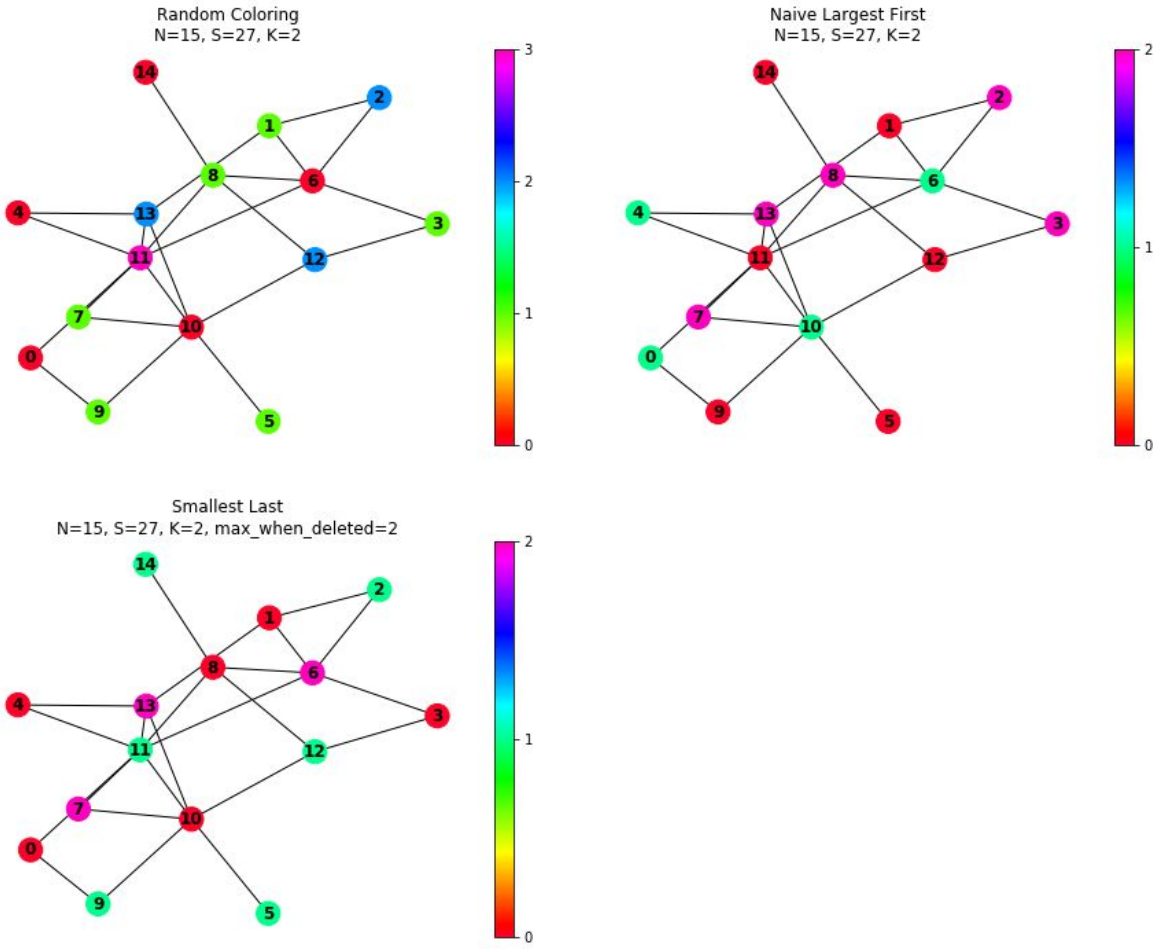
Figure 26: Time to Generate Smallest Last Coloring and/or Ordering



Plots showing the behavior of all three algorithms when S and K vary were preparted but are not included due to the size constraints of this assignment.

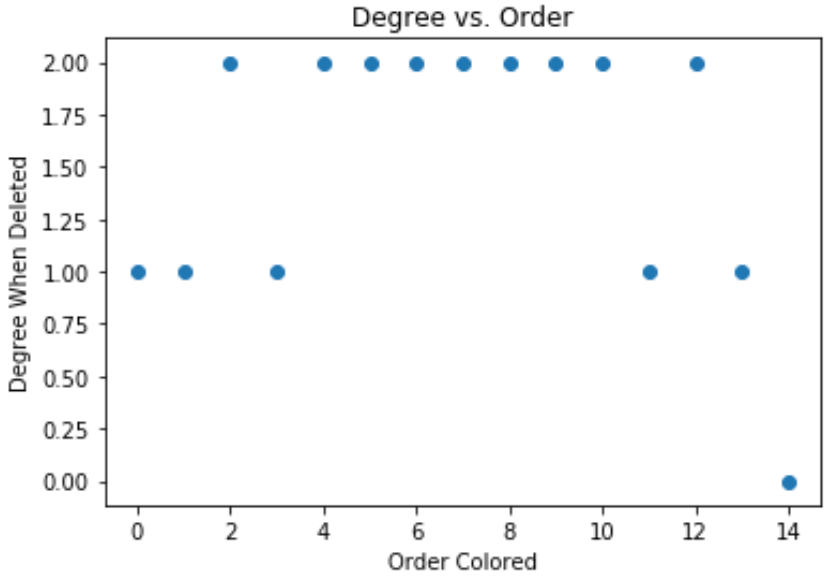
It is easy to represent small graphs visually, as shown in the following pictures. Note that in each case the program derived a different coloring, but selecting the largest or smallest node can be arbitrary for a particular algorithm. In this case, the random algorithm required 4 colors and the Largest First and Smallest Last algorithms required 3.

Figures 27-29: Colored Graphs



The following plot shows the degree of the session vertex when deleted from the graph, as a function of the order colored, for the smallest last ordering. As expected the last node colored has the smallest degree.

Figure 30: Degree When Deleted



CONCLUSION

We have created a program that samples one of four different distributions to determine which K distinct sessions an individual attending a conference will select from a pool of N available sessions. The program repeats this sampling for all S attendees and creates a graph of which sessions have been requested together and should not be scheduled at the same time. Sessions are then 'colored' such that sessions with the same color are safe to schedule concurrently. To accommodate all selections, the number of sessions assigned to the color with the largest number of sessions can be used as a measure of how long the conference will last. In other words, if the color with the largest number of sessions has 24 sessions, then the conference must last long enough to have 24 concurrent sessions.

Although K was allowed to vary substantially in the problem statement, we found that K must be kept rather small to avoid the trivially hard scheduling problem. The trivially hard scheduling problem occurs when all sessions are in conflict with all other sessions, and requires that the conference is scheduled one session at a time.

We did not see the desired linear performance when using the Smallest Last algorithm. This is likely because the average degree of the sessions, A, is equal to a significant fraction of N, and the algorithm is O(N x A). This could be further investigated by plotting the performance using simulated data where N increases but the average number of sessions remains constant.

We also recommend that a more sophisticated sampling be used where the number of sessions selected by each attendee is not constant. Additionally, we recommend determining a maximum K using the longest acceptable conference duration.

APPENDIX

Some of the code used in this project is contained in Jupyter Notebooks that cannot be included here for formatting reasons, as there is no good way to include html in a word document. The github repo for this project is available upon request.

Example Adjacency Matrix and Adjacency List Output

The following output is produced by the program when processing a small test case with N=5, S=7, and K=2.

Output 1: Adjacency Matrix and Adjacency List Sample with Precomputed K

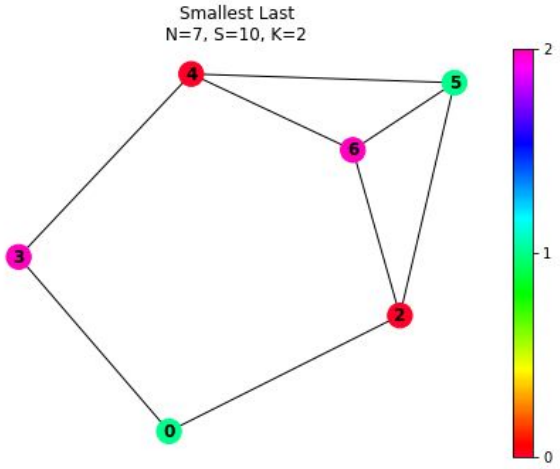
```
--chosen conflicts--
[[4 1]
 [4 2]
 [2 3]
 [0 2]
 [3 1]
 [4 1]
 [0 4]]

--Adjacency List--
Edge collisions: 1
T total conflicts: 7
M distinct conflicts: 6
0 -> 2,4
1 -> 3,4
2 -> 0,3,4
3 -> 1,2
4 -> 0,1,2

E_array: [2, 4, 3, 4, 0, 3, 4, 1, 2, 0, 1, 2]
P_array: [0, 2, 4, 7, 9]


--Adjacency Matrix--
Edge Collisions: 1
T total conflicts: 7
M distinct conflicts: 6
[[False False  True False  True]
 [False False False  True  True]
 [ True False False  True  True]
 [False  True  True False False]
 [ True  True  True False False]]

E_array: [2, 4, 3, 4, 0, 3, 4, 1, 2, 0, 1, 2]
P_array: [0, 2, 4, 7, 9]
```

Example Orderings

Output 1: Orderings

```
E_array: [5, 6, 4, 5, 3, 2, 6, 1, 5, 6, 0, 1, 4, 6, 0, 3, 4, 5]
P_array: [0, 2, 4, 5, 7, 10, 14]
--Converted to Adjacency List--
0 -> 5,6
1 -> 4,5
2 -> 3
3 -> 2,6
4 -> 1,5,6
5 -> 0,1,4,6
6 -> 0,3,4,5
** random: [0, 0, 0, 2, 3, 2, 1] 3
** largest first: [2, 1, 1, 0, 2, 0, 1] 2
** smallest last: [0, 2, 1, 0, 0, 1, 2] 2
```

Example Smallest Last Ordering Graph and Output



The graph above was colored using the Smallest Last ordering. Note that the list immediately following **** is the list of current degrees for each session before removing a node, and the list immediately preceding ^^^^ is the current degree list after removing the node and decrementing the current degree of all adjacent nodes. The list in between these lists is the list of nodes adjacent to the removed list. In this example session 1 did not have any conflicts and is not included in the graph, but is selected first because it has a degree of zero..

Example Verbose Output

```
--Converted to Adjacency List--
0 -> 2,3
1 ->
2 -> 0,5,6
3 -> 0,4
4 -> 3,5,6
```

```
5 -> 2,4,6
6 -> 2,4,5
0: {1: True}
1: {}
2: {0: True, 3: True}
3: {2: True, 4: True, 5: True, 6: True}

****
[2, 0, 3, 2, 3, 3, 3]
[]
[2, -1, 3, 2, 3, 3, 3]
^^^^
0 [(1, 0)]
****
[2, -1, 3, 2, 3, 3, 3]
[0, 4]
[1, -1, 3, -1, 2, 3, 3]
^^^^
1 [(1, 0), (3, 2)]
****
[1, -1, 3, -1, 2, 3, 3]
[2, 3]
[-1, -1, 2, -1, 2, 3, 3]
^^^^
0 [(1, 0), (3, 2), (0, 1)]
****
[-1, -1, 2, -1, 2, 3, 3]
[0, 5, 6]
[-1, -1, -1, -1, 2, 2, 2]
^^^^
1 [(1, 0), (3, 2), (0, 1), (2, 2)]
****
[-1, -1, -1, -1, 2, 2, 2]
[2, 4, 5]
[-1, -1, -1, -1, 1, 1, -1]
^^^^
1 [(1, 0), (3, 2), (0, 1), (2, 2), (6, 2)]
****
[-1, -1, -1, -1, 1, 1, -1]
[2, 4, 6]
[-1, -1, -1, -1, 0, -1, -1]
^^^^
0 [(1, 0), (3, 2), (0, 1), (2, 2), (6, 2), (5, 1)]
****
[-1, -1, -1, -1, 0, -1, -1]
[3, 5, 6]
[-1, -1, -1, -1, -1, -1, -1]
^^^^
0 [(1, 0), (3, 2), (0, 1), (2, 2), (6, 2), (5, 1), (4, 0)]
-1 [(1, 0), (3, 2), (0, 1), (2, 2), (6, 2), (5, 1), (4, 0)]
```

CODE:

init.py

```
print(f'Invoking __init__.py for {__name__}')
import proj1.dist, proj1.adjacency, proj1.coloring
__all__ = [
          'dist',
          'test'
          ]
```

dist.py

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from math import factorial, pi, log
import scipy.stats as stats
import random
import bisect
import time
import os
import timeit
import warnings

from .timer import timer

class distribution_base:
   def __init__(self, max_range=1000):
      self.stored_map = None
      self.max_range = max_range
      self.n_sessions = max_range

   def _upper_binary_bisect(self, x):
      # including code from Python standard lib
      # This bisect algorithm is twice as fast as the recursive code I wrote below even for small n=10, and 5x faster for n=1000000
      # avoids recursion overhead
#################### BEGIN CODE FROM cpython - standard library
   # """Bisection algorithms."""

   # """Return the index where to insert item x in list a, assuming a is sorted.
   # The return value i is such that all e in a[:i] have e <= x, and all e in
   # a[i:] have e > x.  So if x already appears in the list, a.insert(x) will
   # insert just after the rightmost x already there.
   # Optional args lo (default 0) and hi (default len(a)) bound the
   # slice of a to be searched.
   # """
   # def bisect_right(a, x, lo=0, hi=None):
      lo = 0
      hi = self.max_range
      # if lo < 0:
      #     raise ValueError('lo must be non-negative')
      # if hi is None:
      #     hi = self.n_sessions
      while lo < hi:
         mid = (lo+hi)//2
         if x < self.stored_map[mid]: hi = mid
         else: lo = mid+1
      return lo - 1
#################### END CODE FROM cpython

   # prev code is much faster than the recursive algorithm below (>5x as n=10000)
   # def recursive_bisect_right(sorted_list, lower, upper, val):
   #     # recursed to 2 elements,
   #     if upper - lower <= 1:
   #         if sorted_list[lower] >= val:
   #             return lower
   #         else:
   #             return upper
   #     midpoint = lower + int((upper - lower)//2)   # // is flooring division, but faster than math.floor()
   #     if sorted_list[midpoint] < val:
   #         return recursive_bisect_right(sorted_list, midpoint+1, upper, val)
```

```
    #     else:
    #         return recursive_bisect_right(sorted_list, lower, midpoint, val)


    # beta/skewed only
    # given a random number between 0 and 1, find the smallest i such that cdf(i) > rand, using binary search
    def get_samples(self, n=1):
        ret_a = [-1]*n
        for i in range(n):
            ret_a[i] = self._upper_binary_bisect(random.random())
        return ret_a

    # beta/skewed only
    def sample_one(self):
        return self._upper_binary_bisect(random.random())

    # overwritten by all but beta
    def cdf(self, x):
        raise Exception('This function should not be used for this distribution.  Use full_cdf() instead')

    # overwritten by only beta
    def full_cdf(self, grid_size):
        raise Exception('This function should not be used for this distribution.  Use cdf() instead')

    # overwritten by skewed, beta
    def init_sampling(self):
        pass

    # sample distribution K or more times, resampling if receive duplicate item
    # check for uniqueness using hash of size n_sessions.
    # returns K samples, count of collisions
    def get_k_sessions(self, k_sessions_per = 10):
        ret_a = [-1] * k_sessions_per

        # using table instead of hashing since expect collisions
        collision_detect = [False] * self.max_range

        collision_count = 0
        for i in range(k_sessions_per):
            found = False
            # repeatedly sample until get a new value not already seen
            while(not found):
                sample = self.sample_one()
                if collision_detect[sample] == False:
                    collision_detect[sample] = True
                    found = True
                    ret_a[i] = sample
                else:
                    collision_count += 1
        return ret_a, collision_count

class uniform_dist(distribution_base):
    def __init__(self, max_range=1000):
        self.name = 'uniform'
        super().__init__(max_range=max_range)

    def pdf(self, x):
        return 1

    def cdf(self, x):
        return x

    def get_samples(self, n=10):
```

```
        ret_a = [-1]*n
        for i in range(n):
            ret_a[i] = int((self.cdf(random.random())*self.max_range)//1)
        return ret_a

    #significant speedup from moving to single return without list overhead
    def sample_one(self):
        return int((self.cdf(random.random())*self.max_range)//1)


class twotier_dist(distribution_base):
    def __init__(self, max_range=1000, breakpoint=0.1, bottom_prob=0.5):
        super().__init__(max_range=max_range)
        self.breakpoint = breakpoint
        self.bottom_prob = bottom_prob
        self.name = 'twotier'

    def pdf(self, x, breakpoint=0.1, bottom_prob=0.5):
        if x < 0.1:
            return self.bottom_prob / self.breakpoint
        else:
            return (1-self.bottom_prob) / (1-self.breakpoint)

    # calculated symbolic integral
    def cdf(self, x):
        if x < self.breakpoint:
            return self.bottom_prob / self.breakpoint * x
        else:
            return self.bottom_prob + (1-self.bottom_prob) / (1-self.breakpoint) * (x - self.breakpoint)

    # sample distribution to return course number
    def get_samples(self, n=10):
        ret_a = [-1]*n
        for i in range(n):
            rand1 = random.random()
            rand2 = random.random()
            # if in lower tier, resample
            if rand1 <= self.bottom_prob:
                ret_a[i] = int((rand2*self.breakpoint*self.max_range)//1)
            # in upper tier
            else:
                ret_a[i] = int(((self.breakpoint + rand2*(1-self.breakpoint))*self.max_range)//1)

        return ret_a

    #significant speedup from moving to single return without list overhead
    def sample_one(self):
        rand1 = random.random()
        rand2 = random.random()
        # if in lower tier, resample
        if rand1 <= self.bottom_prob:
            return int((rand2*self.breakpoint*self.max_range )//1)
        # in upper tier
        else:
            return int(((self.breakpoint + rand2*(1-self.breakpoint))*self.max_range)//1)


class skewed_dist(distribution_base):
    def __init__(self, max_range=1000):
        super().__init__(max_range=max_range)
        self.name = 'skewed'
    def pdf(self, x):
        return -2. * x + 2
```

```
    def cdf(self, x):
        return -1 * x * x + 2 * x

    # find the cumulative probability from the CDF curve that corresponds to a particular session (i)
    def init_sampling(self):
        self.stored_map = [self.cdf(i/self.max_range) for i in range(self.max_range)]

class beta_dist(distribution_base):
    def __init__(self, max_range=1000, alpha=2, beta=16, use_cache=False):
        self.name = 'beta'
        self.alpha = alpha
        self.beta = beta
        self.use_cache = use_cache
        if use_cache and max_range <= 1000:
            # warnings.warn('Cache disabled for max_range <= 1000')
            self.use_cache = False
        super().__init__(max_range=max_range)

    def _int_gamma(self, n):
        # assert isinstance(n, int)
        return factorial(n-1)

    # alpha, beta as integers >=1
    def pdf(self, x):
                                                                                                                                                    return
(self._int_gamma(self.alpha+self.beta)/self._int_gamma(self.alpha)/self._int_gamma(self.beta))*(x**(self.alpha-1))*((1-x)**(self.beta-1))

    # this can be optimized but is only run once
    # integrate by numeric approximation
    # O(N*grid_size)
    def full_cdf(self, grid_size=100):
        # cache to avoid recreating distribution many times for testing
        if self.use_cache:
            try:
                temp_df = pd.read_parquet('cache/beta_cache_' + str(self.max_range) + '__' + str(grid_size))
                create_cache = False
                cdf_list = temp_df['beta2_16']
            except:
                create_cache = True
        if (not self.use_cache) or create_cache:
            b_coeff = (self._int_gamma(self.alpha+self.beta)/self._int_gamma(self.alpha)/self._int_gamma(self.beta))
            grid_delta = 1/self.max_range/grid_size

            cdf_list = np.full(self.max_range+1, -1.)
            a_1 = self.alpha-1
            b_1 = self.beta-1

            cdf_list[0] = 0
            point_x = 0
            grid_list = np.full(grid_size, -1.)
            for i in range(1,self.max_range+1):
                for j in range(grid_size):
                    point_x = point_x + grid_delta
                    grid_list[j] = b_coeff*(point_x**(a_1))*((1-point_x)**(b_1))
                cdf_list[i] = cdf_list[i-1] + grid_delta * np.sum(grid_list)

        if self.use_cache and create_cache:
            temp_df = pd.DataFrame(cdf_list)
            temp_df.columns = ['beta2_16']
            temp_df.to_parquet('cache/beta_cache_' + str(self.max_range) + '__' + str(grid_size), compression=None)

        # do not include value at 1.0
```

```
        return cdf_list

    def init_sampling(self):
        self.stored_map = self.full_cdf()[:-1]  # don't need interval [1,1]


# distance from reference distribution
def RMSE(estimated, known):
    assert len(estimated) == len(known)
    MSE_acc = 0
    for i in range(len(estimated)):
        MSE_acc += (estimated[i] - known[i])**2
    return (MSE_acc/len(known))**0.5


# initilize and return distribution object that is capable of sampling
def init_dist(chosen_dist, n_sessions=100000, use_cache=False):
    if chosen_dist == 'uniform':
        dist = uniform_dist(max_range=n_sessions)
    elif chosen_dist == 'twotier':
        dist = twotier_dist(max_range=n_sessions)
    elif chosen_dist == 'skewed':
        dist = skewed_dist(max_range=n_sessions)
        dist.init_sampling()
    elif chosen_dist == 'beta':
        dist = beta_dist(max_range=n_sessions, use_cache=use_cache)
        dist.init_sampling()
    else:
        raise ValueError('Invalid distribution input to init_dist()')
    return dist


# show O(1) for uniform, two-tier
def time_to_init(dist_to_init='None', n_sessions=1000, n_per_repeat= 3):
    SETUP_CODE = '''
import proj1
'''

    TEST_CODE = '''
my_dist = proj1.dist.init_dist('{0}', n_sessions={1})
'''.format(dist_to_init, n_sessions)

    times = timeit.repeat(setup=SETUP_CODE,
                  stmt = TEST_CODE,
                  repeat=3,
                  number=n_per_repeat)
    return min(times)/n_per_repeat

def time_to_sample_one(dist_to_init, n_sessions, n_per_repeat=10000):
    if dist_to_init == 'beta':
        SETUP_CODE = '''
import proj1
my_dist = proj1.dist.init_dist('{0}', n_sessions={1}, use_cache=True)
'''.format(dist_to_init, n_sessions)

    else:
        SETUP_CODE = '''
import proj1
my_dist = proj1.dist.init_dist('{0}', n_sessions={1})
'''.format(dist_to_init, n_sessions)

    TEST_CODE = '''
my_dist.sample_one()
```

```
'''.format(n_sessions)

    times = timeit.repeat(setup=SETUP_CODE,
                          stmt = TEST_CODE,
                          repeat=5,
                          number=n_per_repeat)
    return min(times)/n_per_repeat

def time_to_get_k_sessions(dist_to_init, n_sessions, k_sessions_per, n_per_repeat= 1000):
    if dist_to_init == 'beta':
        SETUP_CODE = '''
import proj1
my_dist = proj1.dist.init_dist('{0}', n_sessions={1}, use_cache=True)
'''.format(dist_to_init, n_sessions)

    else:
        SETUP_CODE = '''
import proj1
my_dist = proj1.dist.init_dist('{0}', n_sessions={1})
'''.format(dist_to_init, n_sessions)

    TEST_CODE = '''
my_dist.get_k_sessions(k_sessions_per={0}))
'''.format(k_sessions_per)

    times = timeit.repeat(setup=SETUP_CODE,
                          stmt = TEST_CODE,
                          repeat=3,
                          number=n_per_repeat)
    return min(times)/n_per_repeat


def init_timing(redo_calc=False, make_plots=True):
    cache_file = 'init_testing_N.csv'

    if redo_calc:
        t = timer('sample test')
        my_dists = ['uniform', 'twotier', 'skewed', 'beta']
        n_sess_list = [i*100 for i in range(1, 101)]   #0 to 10,000
        repeat_dict = {'uniform': 1000,
                       'twotier': 1000,
                       'skewed': 100,
                       'beta':5}

        res_df  = pd.DataFrame([n_sess_list[i] for i in range(len(n_sess_list))])
        res_df.columns = ['N']

        time_list = [-1]*len(n_sess_list)
        for this_dist in my_dists:
            print(this_dist)
            for i in range(len(n_sess_list)):
                time_list[i] = time_to_init(this_dist, n_sessions=n_sess_list[i], n_per_repeat=repeat_dict[this_dist])

            res_df[this_dist] = time_list

        # for i in range(len(n_sess_list)):
        #     time_list[i] = i
        res_df['N'] = n_sess_list

        res_df = res_df.melt(id_vars=['N'])
        res_df.columns = ['N', 'Distribution', 'Time (sec)']
        res_df['Time (microsec)'] = res_df['Time (sec)'] / 1e-6
        t.toc(True)
```

```
    res_df.to_csv(cache_file, index=False)

  if make_plots:
    # sampling a distribution once is independent of S, K, M
    res_df2 = pd.read_csv(cache_file)
    max_dict = res_df2[['Distribution','Time (microsec)']].groupby('Distribution').agg('max')['Time (microsec)'].to_dict()
    res_df2['Time / Max Time'] = res_df2.apply(lambda row: row['Time (microsec)'] / max_dict[row['Distribution']], axis=1)

    fig, (ax1, ax2) = plt.subplots(1,2)

    sns.lineplot(data=res_df2[res_df2.Distribution != 'N'], x='N', y='Time (microsec)', hue='Distribution', ax=ax1)
    ax1.set_title('Average Time To Initialize Distribution')

    sns.lineplot(data=res_df2, x='N', y='Time / Max Time', hue='Distribution', ax=ax2)
    ax2.set_title('Scaled Average Time To Initialize Distribution')
    plt.show()


# independet of S, K, M
def sample_timing_after_init(redo_calc=False, make_plots=True):
  cache_file = 'one_sample_testing_N.csv'

  if redo_calc:
    t = timer('init test')
    my_dists = ['uniform', 'twotier', 'skewed', 'beta']
    n_sess_list = [i*100 for i in range(1,101)]   #100 to 10,000
    repeat_dict = {i:100000 for i in my_dists}
    repeat_dict['beta'] = 500
    res_df  = pd.DataFrame([n_sess_list[i] for i in range(len(n_sess_list))])
    res_df.columns = ['N']

    time_list = [-1]*len(n_sess_list)
    for this_dist in my_dists:
      print(this_dist)
      for i in range(len(n_sess_list)):
        time_list[i] = time_to_sample_one(this_dist, n_sessions=n_sess_list[i], n_per_repeat=repeat_dict[this_dist])

      res_df[this_dist] = time_list

    res_df = res_df.melt(id_vars=['N'])
    res_df.columns = ['N', 'Distribution', 'Time (sec)']
    res_df['Time (microsec)'] = res_df['Time (sec)'] / 1e-6
    t.toc(True)
    res_df.to_csv(cache_file, index=False)

  if make_plots:
    fig, (ax1, ax2) = plt.subplots(1,2)
    # sampling a distribution once is independent of S, K, M
    res_df2 = pd.read_csv(cache_file)
    max_dict = res_df2[['Distribution','Time (microsec)']].groupby('Distribution').agg('max')['Time (microsec)'].to_dict()
    res_df2['Time / Max Time'] = res_df2.apply(lambda row: row['Time (microsec)'] / max_dict[row['Distribution']], axis=1)


    sns.lineplot(data=res_df2, x='N', y='Time (microsec)', hue='Distribution', ax=ax1)
    ax1.set_title('Average Time To Sample From Distribution Once Initialized')

    g = sns.lineplot(data=res_df2, x='N', y='Time / Max Time', hue='Distribution', ax=ax2)
    g.axes.semilogx(True)
    ax2.set_title('Scaled Time To Sample From Distribution Once Initilized')
    plt.show()

# create plots for collisions as function of K
def collisions_k(redo_calc=False, make_plots=False, n_trials = 30):
```

```
dist_list = ['uniform', 'twotier', 'skewed', 'beta']
n_sess = 10000

k_vals = {}
k_vals['uniform'] = [i*100 for i in range(1,101)]
k_vals['twotier'] = [i*100 for i in range(1,101)]
k_vals['skewed'] = [i*80 for i in range(1,101)]
k_vals['beta'] = [i*40 for i in range(1,101)]

if redo_calc:
    for dist_name in dist_list:
        my_dist = init_dist(dist_name, n_sessions=n_sess)
        avg = {}
        t = timer(dist_name)

        for k in k_vals[dist_name]:
            tot = 0
            for i in range(n_trials):
                x, cnt = my_dist.get_k_sessions(k_sessions_per=k)
                tot = tot + cnt
            avg[k] = tot/n_trials

        t.toc(True)

        collision_df = pd.DataFrame({'K':list(avg.keys()), 'Collisions':[avg[i] for i in avg.keys()]})
        collision_df.to_csv('collisions_' + dist_name + '.csv', index=False)

if make_plots:
    for dist_name in dist_list:
        collision_df = pd.read_csv('collisions_' + dist_name + '.csv')
        g = sns.scatterplot(data=collision_df, x='K', y='Collisions', color='red')
        collision_df['Collisions'] = (0.01 * collision_df['K'])**2
        g = sns.lineplot(data=collision_df, x='K', y='Collisions')
        g.get_figure().savefig('collisions_' + dist_name)
        g.axes.loglog(True)
        g.axis('scaled')
        g.set_title('{} Collisions - N={}\nline corresponds to K^2'.format(dist_name.capitalize(), n_sess))
        plt.show()
```

adjacency.py

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import timeit
from sortedcontainers import SortedList
from .dist import init_dist
from .timer import timer
import pickle

class adjacency_base():
    def __init__(self, dist=None, s_attendees=10000, k_sessions_per=10):
        self.dist = dist
        self.s_attendees = s_attendees
        self.k_sessions_per = k_sessions_per
        self.edge_collisions = 0
        self.m_conflicts = 0
        self.t_conflicts = 0  # can calculate but counted internally to verify=  sum (i from 1 to k) * n_sessions

    def print_E_P(self):
        x, y = self.E_P()
        print('E_array:', x)
        print('P_array:', y)

    def write_E_P(self, filename='output'):
        with open(filename, 'wb') as fp:
            pickle.dump(self.E_P(), fp)


class adjacency_matrix_base(adjacency_base):
    def __init__(self, dist=None, s_attendees=10000, k_sessions_per=10):
        super().__init__(dist, s_attendees, k_sessions_per)
        self.adj_matrix = np.zeros(self.dist.n_sessions*self.dist.n_sessions, dtype=bool)
        self.adj_matrix = self.adj_matrix.reshape((self.dist.n_sessions, self.dist.n_sessions))

    def print(self):
        print('--Adjacency Matrix--')
        print('Edge Collisions:', self.edge_collisions)
        print('T total conflicts:', self.t_conflicts)
        print('M distinct conflicts:', self.m_conflicts)
        print(self.adj_matrix)

    def E_P(self):
        p_array = [-1]*self.dist.n_sessions
        e_array = []

        p_pntr = 0
        for i in range(self.dist.n_sessions):
            p_array[i] = p_pntr
            for j in range(self.dist.n_sessions):
                if self.adj_matrix[i][j] == True:
                    e_array.append(j)
                    p_pntr += 1
        return e_array, p_array

    def print_adj_list(self):
        adj_list = [[] for i in range(self.dist.n_sessions)]
        for i in range(self.dist.n_sessions):
            for j in range(self.dist.n_sessions):
                if self.adj_matrix[i][j] == True:
                    adj_list[i].append(j)
```

```
        txt = '--Converted to Adjacency List--'
        for i in range(self.dist.n_sessions):
            txt += '\n{} -> {}'.format(i, ','.join([str(x) for x in adj_list[i]]))
        print(txt)

    def to_adj_list(self):
        adj_list = [[] for i in range(self.dist.n_sessions)]
        for i in range(self.dist.n_sessions):
            for j in range(self.dist.n_sessions):
                if self.adj_matrix[i][j] == True:
                    adj_list[i].append(j)
        return adj_list


class adjacency_matrix(adjacency_matrix_base):
    # fill adjacency matrix
    def populate(self):
        # for each attendees
        for i in range(self.s_attendees):
            # get K unique sessions
            # O(N * N - N)
            k_sessions, _ = self.dist.get_k_sessions(k_sessions_per=self.k_sessions_per)
            # add to matrix
            for j in range(self.k_sessions_per-1):
                for k in range(j+1, self.k_sessions_per):
                    self.t_conflicts += 1
                    x = k_sessions[j]
                    y = k_sessions[k]
                    if self.adj_matrix[x][y] == False:
                        self.adj_matrix[x][y] = True
                        self.adj_matrix[y][x] = True
                        self.m_conflicts += 1
                    else:
                        self.edge_collisions += 1 # each x,y pair is x->y and y->x


# class to test 'remove duplicates' functionality
class adjacency_matrix_test(adjacency_matrix_base):
    def get_all_k_sessions(self, use_cache=True):
        cache_file = 'get_all_' + self.dist.name + '_' + str(self.dist.n_sessions)  + '_' +  str(self.s_attendees) + '_' + str(self.k_sessions_per)
        if use_cache:
            try:
                k_matrix = np.load(cache_file)
                create_cache = False
            except:
                create_cache = True
        if create_cache or (not use_cache):
            k_matrix = np.full((self.s_attendees, self.k_sessions_per), -1)
            for i in range(self.s_attendees):
                k_matrix[i,], _ = self.dist.get_k_sessions(k_sessions_per=self.k_sessions_per)
            np.save(cache_file, k_matrix)
        self.k_matrix = k_matrix

    # fill adjacency matrix, but obtain all K-sessions before.  requires K x S additional space
    def populate(self):
        # for each attendees
        for i in range(self.s_attendees):
            # get K unique sessions
            # using prepopulated list
            k_sessions = self.k_matrix[i,]
            # add to matrix
            for j in range(self.k_sessions_per-1):
                for k in range(j+1, self.k_sessions_per):
```

```
                self.t_conflicts += 1
                x = k_sessions[j]
                y = k_sessions[k]
                if self.adj_matrix[x][y] == False:
                    self.adj_matrix[x][y] = True
                    self.adj_matrix[y][x] = True
                    self.m_conflicts += 1
                else:
                    self.edge_collisions += 1 # each x,y pair is x->y and y->x


class adjacency_list_base(adjacency_base):
    def __init__(self, dist=None, s_attendees=10000, k_sessions_per=10):
        super().__init__(dist, s_attendees, k_sessions_per)
        self.adj_list = [SortedList() for i in range(dist.n_sessions)]

    def print(self):
        txt = '--Adjacency List--'
        txt += '\nEdge collisions: {}'.format(self.edge_collisions)
        txt += '\nT total conflicts: {}'.format(self.t_conflicts)
        txt += '\nM distinct conflicts: {}'.format(self.m_conflicts)
        for i in range(self.dist.n_sessions):
            txt += '\n{} -> {}'.format(i, ','.join([str(x) for x in self.adj_list[i]]))
        print(txt)

    def E_P(self):
        p_array = [-1]*self.dist.n_sessions
        e_array = []

        p_pntr = 0
        for i in range(self.dist.n_sessions):
            p_array[i] = p_pntr
            for j in self.adj_list[i]:
                e_array.append(j)
                p_pntr += 1
        return e_array, p_array


class adjacency_list(adjacency_list_base):
    def populate(self):
        # array of sorted lists (tehcnically a list but can index list_array in O(1))

        for i in range(self.s_attendees):
            # get K unique sessions
            # O(N * N - N)
            k_sessions, _ = self.dist.get_k_sessions(k_sessions_per=self.k_sessions_per)
            # add to matrix
            for j in range(len(k_sessions)):
                for k in range(j+1, len(k_sessions)):
                    self.t_conflicts += 1
                    x = k_sessions[j]
                    y = k_sessions[k]

                    if y in self.adj_list[x]:
                        self.edge_collisions += 1  # if find x->y don't check for y->x
                    else:
                        self.adj_list[x].add(y)
                        self.adj_list[y].add(x)
                        self.m_conflicts += 1


# precomputes K-sessions for each S attendee, which takes O(KxS) extra space.  Used for testing.
class adjacency_list_test(adjacency_list_base):
```

```
    def get_all_k_sessions(self, use_cache=True):
        cache_file = 'get_all_' + self.dist.name + '_' + str(self.dist.n_sessions)  + '_' +  str(self.s_attendees) + '_' + str(self.k_sessions_per)
        if use_cache:
            try:
                k_matrix = np.load(cache_file)
                create_cache = False
            except:
                create_cache = True
        if create_cache or (not use_cache):
            k_matrix = np.full((self.s_attendees, self.k_sessions_per), -1)
            for i in range(self.s_attendees):
                k_matrix[i,], _ = self.dist.get_k_sessions(k_sessions_per=self.k_sessions_per)
            np.save(cache_file, k_matrix)
        self.k_matrix = k_matrix

    # fill adjacency matrix, but obtain all K-sessions beforehand.  requires K x S additional space
    def populate(self):
        for i in range(self.s_attendees):
            k_sessions = self.k_matrix[i,]
            for j in range(self.k_sessions_per):
                for k in range(j+1, self.k_sessions_per):
                    self.t_conflicts += 1
                    x = k_sessions[j]
                    y = k_sessions[k]

                    if y in self.adj_list[x]:
                        self.edge_collisions += 1  # if find x->y don't check for y->x
                    else:
                        self.adj_list[x].add(y)
                        self.adj_list[y].add(x)
                        self.m_conflicts += 1

# use pre-computed samples, test only removing duplicates
def time_to_populate_test_matrix(dist_to_init, n_sessions, s_attendees, k_sessions_per, n_per_repeat= 5):
    SETUP_CODE = '''
import proj1
this_dist = proj1.dist.init_dist('{0}', {1}, use_cache=True)
adj_mat_test = proj1.adjacency.adjacency_matrix_test(this_dist, s_attendees={2}, k_sessions_per={3})
adj_mat_test.get_all_k_sessions()
'''.format(dist_to_init, n_sessions, s_attendees, k_sessions_per)

    TEST_CODE = '''
adj_mat_test.populate()
'''

    times = timeit.repeat(setup=SETUP_CODE,
                  stmt = TEST_CODE,
                  repeat=3,
                  number=n_per_repeat)
    return min(times)/n_per_repeat

# use pre-computed k-values, test only removing duplicates
def time_to_populate_test_list(dist_to_init, n_sessions, s_attendees, k_sessions_per, n_per_repeat= 5):
    SETUP_CODE = '''
import proj1
this_dist = proj1.dist.init_dist('{0}', {1}, use_cache=True)
adj_list_test = proj1.adjacency.adjacency_list_test(this_dist, s_attendees={2}, k_sessions_per={3})
adj_list_test.get_all_k_sessions()
'''.format(dist_to_init, n_sessions, s_attendees, k_sessions_per)

    TEST_CODE = '''
adj_list_test.populate()
'''
```

```
    times = timeit.repeat(setup=SETUP_CODE,
                          stmt = TEST_CODE,
                          repeat=3,
                          number=n_per_repeat)
    return min(times)/n_per_repeat

# test time to draw samples, remove dups, but still use cache
def time_to_populate_matrix(dist_to_init, n_sessions, s_attendees, k_sessions_per, n_per_repeat= 5):
    SETUP_CODE = '''
import proj1
this_dist = proj1.dist.init_dist('{0}', {1}, use_cache=True)
adj_mat = proj1.adjacency.adjacency_matrix(this_dist, s_attendees={2}, k_sessions_per={3})
'''.format(dist_to_init, n_sessions, s_attendees, k_sessions_per)

    TEST_CODE = '''
adj_mat.populate()
'''

    times = timeit.repeat(setup=SETUP_CODE,
                          stmt = TEST_CODE,
                          repeat=3,
                          number=n_per_repeat)
    return min(times)/n_per_repeat

# test time to draw samples, remove dups, but still use cache
def time_to_populate_list(dist_to_init, n_sessions, s_attendees, k_sessions_per, n_per_repeat= 5):
    SETUP_CODE = '''
import proj1
this_dist = proj1.dist.init_dist('{0}', {1}, use_cache=True)
adj_list = proj1.adjacency.adjacency_list(this_dist, s_attendees={2}, k_sessions_per={3})
'''.format(dist_to_init, n_sessions, s_attendees, k_sessions_per)

    TEST_CODE = '''
adj_list.populate()
'''

    times = timeit.repeat(setup=SETUP_CODE,
                          stmt = TEST_CODE,
                          repeat=3,
                          number=n_per_repeat)
    return min(times)/n_per_repeat

def N_test_populate_matrix(redo_calc=False, make_plots=True):
    cache_file = 'N_test_populate_matrix.csv'
    if redo_calc:
        my_dists = ['uniform', 'twotier', 'skewed', 'beta']
        repeat_dict = {i:5 for i in my_dists}
        n_sess_list = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]
        s_attend_list = [i*10 for i in n_sess_list]
        k_dict = {i:15 for i in my_dists}

        res_df  = pd.DataFrame(n_sess_list)
        res_df.columns = ['N']

        t = timer('init test')
        time_list = [-1]*len(n_sess_list)
        for this_dist in my_dists:
            print(this_dist)
            for i in range(len(n_sess_list)):
                                time_list[i]  =  time_to_populate_test_matrix(this_dist,  n_sess_list[i],  s_attend_list[i],  k_dict[this_dist],  n_per_repeat=
repeat_dict[this_dist])
            res_df[this_dist] = time_list
```

```
        t.toc(True)


        res_df = res_df.melt(id_vars=['N'])
        res_df.columns = ['N', 'Distribution', 'Time (sec)']
        res_df['Time (microsec)'] = res_df['Time (sec)'] / 1e-6
        res_df.to_csv(cache_file, index=False)

    if make_plots:
        # sampling a distribution once is independent of S, K, M
        res_df2 = pd.read_csv(cache_file)
        max_dict = res_df2[['Distribution','Time (sec)']].groupby('Distribution').agg('max')['Time (sec)'].to_dict()
        res_df2['Time / Max Time'] = res_df2.apply(lambda row: row['Time (sec)'] / max_dict[row['Distribution']], axis=1)

        sns.scatterplot(data=res_df2, x='N', y='Time (sec)', hue='Distribution')
        plt.title('Average Time To Remove Duplicates Via Edge Matrix\nUsing Precomputed K-matrix')
        plt.show()

        sns.scatterplot(data=res_df2, x='N', y='Time / Max Time', hue='Distribution')
        plt.title('Scaled Time To Remove Duplicates Via Edge Matrix\nUsing Precomputed K-matrix')
        plt.show()

def N_test_populate_list(redo_calc=False, make_plots=True):
    cache_file = 'N_test_populate_list.csv'
    if redo_calc:
        my_dists = ['uniform', 'twotier', 'skewed', 'beta']
        repeat_dict = {i:5 for i in my_dists}
        n_sess_list = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]
        s_attend_list = [i*10 for i in n_sess_list]
        k_dict = {i:15 for i in my_dists}

        res_df  = pd.DataFrame(n_sess_list)
        res_df.columns = ['N']

        t = timer('init test')
        time_list = [-1]*len(n_sess_list)
        for this_dist in my_dists:
            print(this_dist)
            for i in range(len(n_sess_list)):
                                time_list[i]  =  time_to_populate_test_list(this_dist,  n_sess_list[i],  s_attend_list[i],  k_dict[this_dist],  n_per_repeat=
repeat_dict[this_dist])
            res_df[this_dist] = time_list
        t.toc(True)

        for i in range(len(n_sess_list)):
            time_list[i] = n_sess_list[i] * np.log(n_sess_list[i])
        res_df['N log(N)'] = time_list

        res_df = res_df.melt(id_vars=['N'])
        res_df.columns = ['N', 'Distribution', 'Time (sec)']
        res_df['Time (microsec)'] = res_df['Time (sec)'] / 1e-6
        res_df.to_csv(cache_file, index=False)

    if make_plots:
        # sampling a distribution once is independent of S, K, M
        res_df2 = pd.read_csv(cache_file)
        max_dict = res_df2[['Distribution','Time (sec)']].groupby('Distribution').agg('max')['Time (sec)'].to_dict()
        res_df2['Time / Max Time'] = res_df2.apply(lambda row: row['Time (sec)'] / max_dict[row['Distribution']], axis=1)

        sns.scatterplot(data=res_df2[res_df2.Distribution != 'N log(N)'], x='N', y='Time (sec)', hue='Distribution')
        plt.title('Average Time To Remove Duplicates Via Edge List\nUsing Precomputed K-matrix')
        plt.show()
```

```
    sns.scatterplot(data=res_df2[res_df2.Distribution != 'N log(N)'], x='N', y='Time / Max Time', hue='Distribution')
    plt.title('Scaled Time To Remove Duplicates Via Edge List\nUsing Precomputed K-matrix')
    plt.show()

def N_populate_matrix(redo_calc=False, make_plots=True):
  cache_file = 'N_populate_matrix.csv'
  if redo_calc:
    my_dists = ['uniform', 'twotier', 'skewed', 'beta']
    repeat_dict = {i:5 for i in my_dists}
    n_sess_list = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]
    s_attend_list = [i*10 for i in n_sess_list]
    k_dict = {i:15 for i in my_dists}

    res_df  = pd.DataFrame(n_sess_list)
    res_df.columns = ['N']

    t = timer('init test')
    time_list = [-1]*len(n_sess_list)
    for this_dist in my_dists:
      print(this_dist)
      for i in range(len(n_sess_list)):
        time_list[i] = time_to_populate_matrix(this_dist, n_sess_list[i], s_attend_list[i], k_dict[this_dist], n_per_repeat= repeat_dict[this_dist])
      res_df[this_dist] = time_list
    t.toc(True)


    res_df = res_df.melt(id_vars=['N'])
    res_df.columns = ['N', 'Distribution', 'Time (sec)']
    res_df['Time (microsec)'] = res_df['Time (sec)'] / 1e-6
    res_df.to_csv(cache_file, index=False)

  if make_plots:
    # sampling a distribution once is independent of S, K, M
    res_df2 = pd.read_csv(cache_file)
    max_dict = res_df2[['Distribution','Time (sec)']].groupby('Distribution').agg('max')['Time (sec)'].to_dict()
    res_df2['Time / Max Time'] = res_df2.apply(lambda row: row['Time (sec)'] / max_dict[row['Distribution']], axis=1)

    sns.scatterplot(data=res_df2, x='N', y='Time (sec)', hue='Distribution')
    plt.title('Average Time To Remove Duplicates Via Edge Matrix\nIncluding Sampling Time')
    plt.show()

    sns.scatterplot(data=res_df2, x='N', y='Time / Max Time', hue='Distribution')
    plt.title('Scaled Time To Remove Duplicates Via Edge Matrix\nIncluding Sampling Time')
    plt.show()

def N_populate_list(redo_calc=False, make_plots=True):
  cache_file = 'N_populate_list.csv'
  if redo_calc:
    my_dists = ['uniform', 'twotier', 'skewed', 'beta']
    repeat_dict = {i:5 for i in my_dists}
    n_sess_list = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]
    s_attend_list = [i*10 for i in n_sess_list]
    k_dict = {i:15 for i in my_dists}

    res_df  = pd.DataFrame(n_sess_list)
    res_df.columns = ['N']

    t = timer('init test')
    time_list = [-1]*len(n_sess_list)
    for this_dist in my_dists:
      print(this_dist)
      for i in range(len(n_sess_list)):
        time_list[i] = time_to_populate_list(this_dist, n_sess_list[i], s_attend_list[i], k_dict[this_dist], n_per_repeat= repeat_dict[this_dist])
```

```
            res_df[this_dist] = time_list
        t.toc(True)

        for i in range(len(n_sess_list)):
            time_list[i] = n_sess_list[i] * np.log(n_sess_list[i])
        res_df['N log(N)'] = time_list

        res_df = res_df.melt(id_vars=['N'])
        res_df.columns = ['N', 'Distribution', 'Time (sec)']
        res_df['Time (microsec)'] = res_df['Time (sec)'] / 1e-6
        res_df.to_csv(cache_file, index=False)

    if make_plots:
        # sampling a distribution once is independent of S, K, M
        res_df2 = pd.read_csv(cache_file)
        max_dict = res_df2[['Distribution','Time (sec)']].groupby('Distribution').agg('max')['Time (sec)'].to_dict()
        res_df2['Time / Max Time'] = res_df2.apply(lambda row: row['Time (sec)'] / max_dict[row['Distribution']], axis=1)

        sns.scatterplot(data=res_df2[res_df2.Distribution != 'N log(N)'], x='N', y='Time (sec)', hue='Distribution')
        plt.title('Average Time To Remove Duplicates Via Edge List\nIncluding Sampling Time')
        plt.show()

        sns.scatterplot(data=res_df2[res_df2.Distribution != 'N log(N)'], x='N', y='Time / Max Time', hue='Distribution')
        plt.title('Scaled Time To Remove Duplicates Via Edge List\nIncluding Sampling Time')
        plt.show()


def S_populate_matrix(redo_calc=False, make_plots=True):
    cache_file = 'S_populate_matrix.csv'
    if redo_calc:
        my_dists = ['uniform', 'twotier', 'skewed', 'beta']
        repeat_dict = {i:30 for i in my_dists}
        n_sess = 1000
        s_attend_list = [250, 500, 1000, 5000, 10000, 25000, 50000]
        k_dict = {i:15 for i in my_dists}

        res_df  = pd.DataFrame(s_attend_list)
        res_df.columns = ['S']

        t = timer('init test')
        time_list = [-1]*len(s_attend_list)
        for this_dist in my_dists:
            print(this_dist)
            for i in range(len(s_attend_list)):
                time_list[i] = time_to_populate_matrix(this_dist, n_sess, s_attend_list[i], k_dict[this_dist], n_per_repeat= repeat_dict[this_dist])
            res_df[this_dist] = time_list
        t.toc(True)


        res_df = res_df.melt(id_vars=['S'])
        res_df.columns = ['S', 'Distribution', 'Time (sec)']
        res_df['Time (microsec)'] = res_df['Time (sec)'] / 1e-6
        res_df.to_csv(cache_file, index=False)

    if make_plots:
        # sampling a distribution once is independent of S, K, M
        res_df2 = pd.read_csv(cache_file)
        max_dict = res_df2[['Distribution','Time (sec)']].groupby('Distribution').agg('max')['Time (sec)'].to_dict()
        res_df2['Time / Max Time'] = res_df2.apply(lambda row: row['Time (sec)'] / max_dict[row['Distribution']], axis=1)

        sns.scatterplot(data=res_df2, x='S', y='Time (sec)', hue='Distribution')
        plt.title('Average Time To Remove Duplicates Via Edge Matrix')
        plt.show()
```

```
    sns.scatterplot(data=res_df2, x='S', y='Time / Max Time', hue='Distribution')
    plt.title('Scaled Time To  Duplicates Via Edge Matrix')
    plt.show()

def S_populate_list(redo_calc=False, make_plots=True):
  cache_file = 'S_populate_list.csv'
  if redo_calc:
    my_dists = ['uniform', 'twotier', 'skewed', 'beta']

    repeat_dict = {i:30 for i in my_dists}
    n_sess = 1000
    s_attend_list = [250, 500, 1000, 5000, 10000, 25000, 50000]
    k_dict = {i:15 for i in my_dists}

    res_df  = pd.DataFrame(s_attend_list)
    res_df.columns = ['S']

    t = timer('init test')
    time_list = [-1]*len(s_attend_list)
    for this_dist in my_dists:
      print(this_dist)
      for i in range(len(s_attend_list)):
        time_list[i] = time_to_populate_list(this_dist, n_sess, s_attend_list[i], k_dict[this_dist], n_per_repeat= repeat_dict[this_dist])
      res_df[this_dist] = time_list
    t.toc(True)

    for i in range(len(s_attend_list)):
      time_list[i] = s_attend_list[i] * np.log(s_attend_list[i])
    res_df['S log(S)'] = time_list

    res_df = res_df.melt(id_vars=['S'])
    res_df.columns = ['S', 'Distribution', 'Time (sec)']
    res_df['Time (microsec)'] = res_df['Time (sec)'] / 1e-6
    res_df.to_csv(cache_file, index=False)

  if make_plots:
    # sampling a distribution once is independent of S, K, M
    res_df2 = pd.read_csv(cache_file)
    max_dict = res_df2[['Distribution','Time (sec)']].groupby('Distribution').agg('max')['Time (sec)'].to_dict()
    res_df2['Time / Max Time'] = res_df2.apply(lambda row: row['Time (sec)'] / max_dict[row['Distribution']], axis=1)

    sns.scatterplot(data=res_df2[res_df2.Distribution != 'S log(S)'], x='S', y='Time (sec)', hue='Distribution')
    plt.title('Average Time To Remove Duplicates Via Edge List')
    plt.show()

    sns.scatterplot(data=res_df2[res_df2.Distribution != 'S log(S)'], x='S', y='Time / Max Time', hue='Distribution')
    plt.title('Scaled Time To  Duplicates Via Edge List')
    plt.show()


def K_populate_matrix(redo_calc=False, make_plots=True):
  cache_file = 'K_populate_matrix.csv'
  my_dists = ['uniform', 'twotier', 'skewed', 'beta']
  k_vals_dict = {'uniform':[2, 5, 10, 25, 50, 100, 150],
           'twotier':[2, 5, 10, 25, 50, 100, 150],
           'skewed':[2, 5, 25, 50, 75, 95, 100],
           'beta':[2, 5, 20, 40, 50, 60]}
  n_sess = 10000
  s_attend = n_sess*10

  for dist_name in my_dists:
    if redo_calc:
```

```
            my_dist = init_dist(dist_name, n_sessions=n_sess)

            pop_time_df  = pd.DataFrame([val for val in k_vals_dict[dist_name]])
            pop_time_df.columns = ['K']

            times = [-1]*len(k_vals_dict[dist_name])
            conflicts = [-1]*len(k_vals_dict[dist_name])
            t = timer(dist_name)
            t.tic()
            for i in range(len(k_vals_dict[dist_name])):
                adj_mat = adjacency_matrix(my_dist, s_attendees=s_attend, k_sessions_per=k_vals_dict[dist_name][i] )
                mytime=timer()
                mytime.tic()
                adj_mat.populate()
                times[i] = mytime.toc()
                conflicts[i] = np.sum(adj_mat.adj_matrix)/(n_sess-1)/n_sess*100
            t.toc(True)

            pop_time_df['Time (sec)'] = times
            pop_time_df['% Conflicts'] = conflicts
            pop_time_df.to_csv(cache_file + dist_name)

        if make_plots:
            pop_time_df = pd.read_csv(cache_file + dist_name)
            f, axes = plt.subplots(1,2)
            sns.scatterplot(data=pop_time_df, x='K', y='Time (sec)', ax=axes[0])
            axes[0].set_title(dist_name.capitalize() + ' - Time to Populate Adjacency Matrix\nN={0}, S={1}'.format(n_sess, s_attend))
            sns.scatterplot(data=pop_time_df, x='K', y='% Conflicts', ax=axes[1])
            axes[1].set_title(dist_name.capitalize() + ' - Average % Conflicts of All Sessions\nN={0}, S={1}'.format(n_sess, s_attend))
            plt.show()


def K_populate_list(redo_calc=False, make_plots=True):
    cache_file = 'K_populate_list.csv'
    my_dists = ['uniform', 'twotier', 'skewed', 'beta']
    k_vals_dict = {'uniform':[2, 5, 10, 25, 50, 100, 150],
                   'twotier':[2, 5, 10, 25, 50, 100, 150],
                   'skewed':[2, 5, 25, 50, 75, 95, 100],
                   'beta':[2, 5, 20, 40, 50, 60]}
    n_sess = 10000
    s_attend = n_sess*10

    for dist_name in my_dists:
        if redo_calc:
            my_dist = init_dist(dist_name, n_sessions=n_sess)

            pop_time_df  = pd.DataFrame([val for val in k_vals_dict[dist_name]])
            pop_time_df.columns = ['K']

            times = [-1]*len(k_vals_dict[dist_name])
            conflicts = [-1]*len(k_vals_dict[dist_name])
            t = timer(dist_name)
            t.tic()
            for i in range(len(k_vals_dict[dist_name])):
                adj_list = adjacency_list(my_dist, s_attendees=s_attend, k_sessions_per=k_vals_dict[dist_name][i] )
                mytime=timer()
                mytime.tic()
                adj_list.populate()
                times[i] = mytime.toc()
                conflicts[i] = sum([len(j) for j in adj_list.adj_list])/(n_sess-1)/(n_sess)*100
            t.toc(True)

            pop_time_df['Time (sec)'] = times
```

```
            pop_time_df['% Conflicts'] = conflicts
            pop_time_df.to_csv(cache_file + dist_name)

        if make_plots:
            pop_time_df = pd.read_csv(cache_file + dist_name)
            f, axes = plt.subplots(1,2)
            sns.scatterplot(data=pop_time_df, x='K', y='Time (sec)', ax=axes[0])
            axes[0].set_title(dist_name.capitalize() + ' - Time to Populate Adjacency List\nN={0}, S={1}'.format(n_sess, s_attend))
            sns.scatterplot(data=pop_time_df, x='K', y='% Conflicts', ax=axes[1])
            axes[1].set_title(dist_name.capitalize() + ' - Average % Conflicts of All Sessions\nN={0}, S={1}'.format(n_sess, s_attend))
            plt.show()


def K_plot_all():
    list_cache_file = 'K_populate_list.csv'
    mat_cache_file = 'K_populate_matrix.csv'
    my_dists = ['uniform', 'twotier', 'skewed', 'beta']
    k_vals_dict = {'uniform':[2, 5, 10, 25, 50, 100, 150],
                   'twotier':[2, 5, 10, 25, 50, 100, 150],
                   'skewed':[2, 5, 25, 50, 75, 95, 100],
                   'beta':[2, 5, 20, 40, 50, 60]}
    # just have to track these, should store in pre-header
    n_sess = 10000
    s_attend = n_sess*10

    for dist_name in my_dists:
        list_df = pd.read_csv(list_cache_file + dist_name)
        list_df['Method'] = 'List'
        mat_df = pd.read_csv(mat_cache_file + dist_name)
        mat_df['Method'] = 'Matrix'
        full_df = list_df.append(mat_df)

        markers = {'List':'X', 'Matrix':'s'}

        f, axes = plt.subplots(1,2)
        sns.scatterplot(data=full_df, x='K', y='Time (sec)', hue='Method', markers=markers, ax=axes[0])
        axes[0].set_title(dist_name.capitalize() + ' - Time to Populate Adjacency List\nN={0}, S={1}'.format(n_sess, s_attend))
        sns.scatterplot(data=full_df, x='K', y='% Conflicts', hue='Method', markers=markers, ax=axes[1])
        axes[1].set_title(dist_name.capitalize() + ' - Average % Conflicts of All Sessions\nN={0}, S={1}'.format(n_sess, s_attend))
        plt.show()
```

coloring.py

```
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import colors
import seaborn as sns
import numpy as np
from math import factorial, pi, log, floor
import scipy.stats as stats
import random
import bisect
import time
import os
from IPython.core.pylabtools import figsize
from sortedcontainers import SortedList
import networkx as nx
import timeit
from .timer import timer

def assign_colors(node_list, adj_mat):
    n_sess = adj_mat.dist.max_range
    color_list = [-1]*len(node_list)

    for i in node_list:
        local_color_list = SortedList()
        for j in range(n_sess):
            if adj_mat.adj_matrix[i][j] == True:
                local_color_list.add(color_list[j])
#                 print('--','i=',i,'color[i]', color_list[j], 'local_color',local_color_list)
        min_color = 0
        for k in range(len(local_color_list)):
            if local_color_list[k] == min_color:
                min_color += 1
            else:
                pass
#         print('-', i, local_color_list, min_color)
        color_list[i] = min_color
    return(color_list)

class dNode():
    def __init__(self, sess, deg=None):
        self.session = sess
        self.degree = deg
    def __gt__(self, x):
        if self.degree > x.degree:
            return True
        else:
            return False
    def __str__(self):
        return '({}:{})'.format(self.session, self.degree)
    def __repr__(self):
        return str((self.session, self.degree))

def create_degree_list(adj_mat):
    n_sess = adj_mat.dist.n_sessions
    deg_list = []
    for i in range(n_sess):
        deg = np.sum(adj_mat.adj_matrix[i,])
        deg_list.append(dNode(i, deg))
    return deg_list

def assign_random(adj_mat, order_only = False):
    n_sess = adj_mat.dist.n_sessions
```

```
    node_list = [i for i in range(n_sess)]
    np.random.shuffle(node_list)    # O(N)
    if order_only:
        return node_list
    else:
        color_list = assign_colors(node_list, adj_mat)
        return color_list, max(color_list)

def assign_largest_first(adj_mat, order_only = False):
    n_sess = adj_mat.dist.n_sessions
    deg_list = create_degree_list(adj_mat)
    # sort by incr degree
    deg_list.sort(reverse=True)
    # flatten to session only
    node_list = [i.session for i in deg_list]
    if order_only:
        return node_list
    else:
        color_list = assign_colors(node_list, adj_mat)
        return color_list, max(color_list)

def assign_smallest_last(adj_mat, order_only = False):
    n_sess = adj_mat.dist.n_sessions
    # create list of hash tables for each possible degree (0 to max_degree)
    smallest_last = SmallestLast(adj_mat)
    # repeatedly get an arbitrary session with the lowest current degree, and update the degrees of all sessions adjacent to selected
    curr = 0
    while curr != -1:
        curr = smallest_last.get_node(curr)
    node_list = [i.session for i in smallest_last.deleted_list]
    node_list.reverse()
    if order_only:
        return node_list
    else:
        color_list = assign_colors(node_list, adj_mat)
        return color_list, max(color_list), smallest_last

# Plot graph
def plot_nodes(G, pos, colors):
    max_color = max(colors)
    len_colors = len(colors)
    # set color mappings
    val_map = {i:colors[i] for i in range(len_colors)}
    values = [val_map.get(node, 0.25) for node in G.nodes()]
    nx.draw(G, pos, with_labels=True, font_weight='bold', cmap=plt.get_cmap('gist_rainbow'), node_color=values)
    sm = plt.cm.ScalarMappable(cmap=plt.get_cmap('gist_rainbow'), norm=plt.Normalize(vmin=0, vmax=max_color))
    sm._A = []
    plt.colorbar(sm, ticks=range(max_color+1))

class SmallestLast():
    def __init__(self, adj_mat):
        self.adj_mat = adj_mat
        self.adj_list = adj_mat.to_adj_list()
        self.n_sess = adj_mat.dist.n_sessions
        self.orig_deg_list = create_degree_list(adj_mat)
        self.curr_deg_list = [i.degree for i in self.orig_deg_list]
        self.sorted_deg_list = sorted(self.orig_deg_list)
        self.max_degree = max(self.sorted_deg_list).degree
        # create list of hash tables for each degree up to max degree
        self.hash_list = [{} for i in range(self.max_degree + 1)]
        self.deleted_list = []

        for i in range(self.n_sess):
```

```
        d_node = self.sorted_deg_list[i]
        self.hash_list[d_node.degree][d_node.session] = True

    def __str__(self):
        s = ''
        for i in range(self.max_degree+1):
            s += '{}: '.format(i) + str(self.hash_list[i]) + '\n'
        return s

    def __repr__(self):
        return 'hash list with max deg ' + str(self.max_degree)

    def max_degree_when_deleted(self):
        return max([i.degree for i in self.deleted_list])

    # keep list of dNodes saving degree when deleted
    def get_node(self, curr_deg_index):
        # try to remove item from dict, starting at curr_index
        while curr_deg_index <= self.max_degree + 1:
            try:
                # get node
                this_session,_ = self.hash_list[curr_deg_index].popitem()
            except:
                this_session = None

            if this_session != None:
#                 print('****')
#                 print(self.curr_deg_list)
                self.deleted_list.append(dNode(this_session, curr_deg_index))
                self.curr_deg_list[this_session] = -1

                # update adjacent.
                adjacent = self.adj_list[this_session]
                for sess in adjacent:
                    # get current degree
                    curr_deg = self.curr_deg_list[sess]
                    # if not deleted
                    if curr_deg != -1:
#                         print(sess, curr_deg)
                        # remove from old hash
                        self.hash_list[curr_deg].pop(sess)
                        # decrement curr degree
                        self.curr_deg_list[sess] -= 1
                        # add to new hash
                        self.hash_list[curr_deg -1][sess] = True
#                 print(adjacent)
#                 print(self.curr_deg_list)
#                 print('^^^^')
                # may now have nodes in previously empty hash
                if curr_deg_index > 0:
                    curr_deg_index -= 1

                break
            else:
                curr_deg_index += 1
        if curr_deg_index > self.max_degree + 1:
            return -1
        else:
            return curr_deg_index

    def get_node_verbose(self, curr_deg_index):
        # try to remove item from dict, starting at curr_index
        while curr_deg_index <= self.max_degree + 1:
```

```
            try:
                # get node
                this_session,_ = self.hash_list[curr_deg_index].popitem()
            except:
                this_session = None

            if this_session != None:
                print('****')
                print(self.curr_deg_list)
                self.deleted_list.append(dNode(this_session, curr_deg_index))
                self.curr_deg_list[this_session] = -1

                # update adjacent.
                adjacent = self.adj_list[this_session]
                for sess in adjacent:
                    # get current degree
                    curr_deg = self.curr_deg_list[sess]
                    # if not deleted
                    if curr_deg != -1:
                        # remove from old hash
                        self.hash_list[curr_deg].pop(sess)
                        # decrement curr degree
                        self.curr_deg_list[sess] -= 1
                        # add to new hash
                        self.hash_list[curr_deg -1][sess] = True
                print(adjacent)
                print(self.curr_deg_list)
                print('^^^^')
                # may now have nodes in previously empty hash
                if curr_deg_index > 0:
                    curr_deg_index -= 1

                break
            else:
                curr_deg_index += 1
        if curr_deg_index > self.max_degree + 1:
            return -1
        else:
            return curr_deg_index

    def plot(self):
        plt.scatter([i for i in range(len(self.deleted_list))], [i.degree for i in self.deleted_list])
        plt.xlabel('Order Colored')
        plt.ylabel('Degree When Deleted')
        plt.title('Degree vs. Order')

def time_to_assign(alg='random', order_only=False, dist_to_init='uniform', n_sessions=100, s_attendees=1000, k_sessions_per = 7,
n_per_repeat= 5):
    if alg == 'random':
        alg_func = 'assign_random'
    elif alg == 'largest':
        alg_func = 'assign_largest_first'
    elif alg == 'smallest':
        alg_func = 'assign_smallest_last'
    else:
        print('invalid input to time_to_assign')
        return -1

    SETUP_CODE = '''
import proj1
this_dist = proj1.dist.init_dist('{0}', {1}, use_cache=True)
am = proj1.adjacency.adjacency_matrix(this_dist, s_attendees={2}, k_sessions_per={3})
am.populate()
```

```
'''.format(dist_to_init, n_sessions, s_attendees, k_sessions_per)

   TEST_CODE = '''
proj1.coloring.{}(am, order_only={})
'''.format(alg_func, order_only)

   times = timeit.repeat(setup=SETUP_CODE,
                         stmt = TEST_CODE,
                         repeat=3,
                         number=n_per_repeat)
   return min(times)/n_per_repeat

def N_coloring_timing(alg='random', order_only=False, redo_calc=False, make_plots=True):
   cache_file = alg + '_assign_N.csv'
   if order_only:
      cache_file += 'order_only_'
   k = 7
   if redo_calc:
      t = timer('random test')
      my_dists = ['uniform', 'twotier', 'skewed', 'beta']
      n_sess_list = [i*100 for i in range(1, 10)]   #100 to 1000
      repeat_dict = {'uniform': 5,
                     'twotier': 5,
                     'skewed': 5,
                     'beta':5}

      res_df  = pd.DataFrame([n_sess_list[i] for i in range(len(n_sess_list))])
      res_df.columns = ['N']

      time_list = [-1]*len(n_sess_list)
      for this_dist in my_dists:
         print(this_dist)
         for i in range(len(n_sess_list)):
                                    time_list[i]  =  time_to_assign(alg=alg,  order_only=order_only,  dist_to_init=this_dist,  n_sessions=n_sess_list[i],
s_attendees=n_sess_list[i]*10, k_sessions_per = k, n_per_repeat=repeat_dict[this_dist])
         res_df[this_dist] = time_list

      # for i in range(len(n_sess_list)):
      #     time_list[i] = i
      res_df['N'] = n_sess_list

      res_df = res_df.melt(id_vars=['N'])
      res_df.columns = ['N', 'Distribution', 'Time (sec)']
      res_df['Time (microsec)'] = res_df['Time (sec)'] / 1e-6
      t.toc(True)
      res_df.to_csv(cache_file, index=False)

   if make_plots:
      if order_only:
         title_word = 'Ordering'
      else:
         title_word = 'Coloring'
      # sampling a distribution once is independent of S, K, M
      res_df2 = pd.read_csv(cache_file)
      max_dict = res_df2[['Distribution','Time (microsec)']].groupby('Distribution').agg('max')['Time (microsec)'].to_dict()
      res_df2['Time / Max Time'] = res_df2.apply(lambda row: row['Time (microsec)'] / max_dict[row['Distribution']], axis=1)

      g = sns.lineplot(data=res_df2[res_df2.Distribution != 'N'], x='N', y='Time (sec)', hue='Distribution')
      if not order_only or alg=='smallest':
         g.axes.loglog(True)
      plt.title('Average Time To Generate {} {}\nS= 10xN, K={}'.format(alg.capitalize(), title_word, k))
      plt.savefig(alg + title_word + '_N.png', pad_inches=0.01)
      plt.show()
```

```
def S_coloring_timing(alg='random', order_only=False, redo_calc=False, make_plots=True):
    cache_file = alg + '_assign_S.csv'
    if order_only:
        cache_file += 'order_only_'
    k = 7
    n_sess = 1000

    if redo_calc:
        t = timer('random test')
        my_dists = ['uniform', 'twotier', 'skewed', 'beta']
        s_list = [500,100,2000,5000]   #100 to 10,000
        repeat_dict = {'uniform': 5,
                       'twotier': 5,
                       'skewed': 5,
                       'beta': 5}

        res_df  = pd.DataFrame([s_list[i] for i in range(len(s_list))])
        res_df.columns = ['S']

        time_list = [-1]*len(s_list)
        for this_dist in my_dists:
            print(this_dist)
            for i in range(len(s_list)):
                        time_list[i]  =  time_to_assign(alg=alg,  order_only=order_only,  dist_to_init=this_dist,  n_sessions=n_sess, s_attendees=s_list[i],
k_sessions_per = k, n_per_repeat=repeat_dict[this_dist])
            res_df[this_dist] = time_list

        # for i in range(len(n_sess_list)):
        #     time_list[i] = i
        res_df['S'] = s_list

        res_df = res_df.melt(id_vars=['S'])
        res_df.columns = ['S', 'Distribution', 'Time (sec)']
        res_df['Time (microsec)'] = res_df['Time (sec)'] / 1e-6
        t.toc(True)
        res_df.to_csv(cache_file, index=False)

    if make_plots:
        if order_only:
            title_word = 'Ordering'
        else:
            title_word = 'Coloring'
        # sampling a distribution once is independent of S, K, M
        res_df2 = pd.read_csv(cache_file)
        max_dict = res_df2[['Distribution','Time (microsec)']].groupby('Distribution').agg('max')['Time (microsec)'].to_dict()
        res_df2['Time / Max Time'] = res_df2.apply(lambda row: row['Time (microsec)'] / max_dict[row['Distribution']], axis=1)

        g = sns.lineplot(data=res_df2[res_df2.Distribution != 'N'], x='S', y='Time (sec)', hue='Distribution')
        # if not order_only:
        #     g.axes.loglog(True)
        plt.title('Average Time To Generate {} {}\nN= {}, K={}'.format(alg.capitalize(), title_word, n_sess, k))
        plt.savefig(alg + title_word + '_S.png', pad_inches=0.01)
        plt.show()


def K_coloring_timing(alg='random', order_only=False, redo_calc=False, make_plots=True):
    cache_file = alg + '_assign_K.csv'
    if order_only:
        cache_file += 'order_only_'

    k_list = [2, 5, 10, 15, 20, 25]
```

```
  # k_vals_dict = {'uniform':[2, 5, 10, 25, 50, 100, 150],
  #               'twotier':[2, 5, 10, 25, 50, 100, 150],
  #               'skewed':[2, 5, 25, 50, 75, 95, 100],
  #               'beta':[2, 5, 20, 40, 50, 60]}
  n_sess = 1000
  s_attend = n_sess*10
  res_df  = pd.DataFrame(k_list)
  res_df.columns = ['K']

  if redo_calc:
    t = timer('K test')
    my_dists = ['uniform', 'twotier', 'skewed', 'beta']
    repeat_dict = {'uniform': 5,
                   'twotier': 5,
                   'skewed': 5,
                   'beta': 5}
    for this_dist in my_dists:
      time_list = [-1]*len(k_list)
      print(this_dist)
      for i in range(len(k_list)):
                    time_list[i] = time_to_assign(alg=alg, order_only=order_only, dist_to_init=this_dist, n_sessions=n_sess, s_attendees=s_attend,
k_sessions_per = k_list[i], n_per_repeat=repeat_dict[this_dist])
      res_df[this_dist] = time_list

    res_df = res_df.melt(id_vars=['K'])
    res_df.columns = ['K', 'Distribution', 'Time (sec)']
    res_df['Time (microsec)'] = res_df['Time (sec)'] / 1e-6
    t.toc(True)
    res_df.to_csv(cache_file, index=False)

  if make_plots:
    if order_only:
      title_word = 'Ordering'
    else:
      title_word = 'Coloring'
    res_df2 = pd.read_csv(cache_file)
    max_dict = res_df2[['Distribution','Time (microsec)']].groupby('Distribution').agg('max')['Time (microsec)'].to_dict()
    res_df2['Time / Max Time'] = res_df2.apply(lambda row: row['Time (microsec)'] / max_dict[row['Distribution']], axis=1)

    g = sns.lineplot(data=res_df2[res_df2.Distribution != 'N'], x='K', y='Time (sec)', hue='Distribution')
    # if not order_only:
    #     g.axes.loglog(True)
    plt.title('Average Time To Generate {} {}\nN={}, S={}'.format(alg.capitalize(), title_word, n_sess, s_attend))
    plt.savefig(alg + title_word + '_K.png', pad_inches=0.01)
    plt.show()
```