Applied Algorithms Assignment 3

Fei Fan (Peter) Chen

January 24, 2021

Exercise 1 (10 pts)

Plugging the numbers into the approximation equation

$$n(p; d) \approx \sqrt{2d \cdot ln\frac{1}{1-p}}$$

$$n(0.5; 2^{128}) \approx \sqrt{2^{128} \cdot 2ln2}$$

The number of operations per day is 10^{12}, therefore the number of days that you would generate the same number with probability half is approximately 60,000 years.

Exercise 2 (10 pts)

a) m and w have to be matched as they are first on each other's preference list. If this was not the case, then there would be an instability and this would not be a stable matching, which is an contradiction.

b) In the case where m have the same preference list $[w_1, w_2, w_3, ...]$, w_1 get its first choice, w_2 gets the first choice of {all m \ m chosen by w_1}, and so on. Otherwise, an instability is produced as there is one m_i who could have matched with a more preferred w. This solution is unique since each w can choose only one m without creating an instability.

Exercise 3 (10 pts)

Imagine m, m', m" and w, w', w".

w: m" > m' > m

w': m' > m" > m

w": m' > m" > m

m: w > w' > w"

m': w > w' > w"

m": w' > w > w"

If w does not lie, the matchings are (m’,w), (m”, w’), (m, w”). However if w switches her preference to m” > m > m’ then the bond (m”, w’) is broken as m’ is now free to propose to w’. This enables w to get her higher preference m”.

Exercise 4 (10 pts)

Input (Proposer):

[[2, 1, 3, 0], [0, 1, 3, 2], [0, 1, 2, 3], [0, 1, 2, 3]]

Input (Accepter):

[[0, 2, 1, 3], [2, 0, 3, 1], [3, 2, 1, 0], [2, 3, 1, 0]]

Trace

0 proposed to 2 [2, -1] Accepted

1 proposed to 0 [0, -1] Accepted

2 proposed to 0 [0, 1] Accepted

1 proposed to 1 [1, -1] Accepted

3 proposed to 0 [0, 2] Rejected

3 proposed to 1 [1, 1] Accepted

1 proposed to 3 [3, -1] Accepted

Output (key w, values m):

2: 0, 0: 2, 1: 3, 3: 1

Exercise 5 (10 pts)

See Figure 1 and 2. The average goodness for the proposer (men) is $Log(N)$ while goodness for accepter (women) is $N/Log(N)$. The algorithm runtime, excluding creation of the preference matrices ($O(N^2)$, is O($NLogN$). The kink at N=5000 is when swap kicked in.

The results is similar to the coupon collector problem where it is also O($NLogN$). In a way, each round of the algorithm where the man proposes, the chance that he proposes to a new woman who’ve not been proposed to before (thereby advancing the algorithm forward towards completion) is k/N where k is number of woman still left unmatched. This is logically similar to the coupon collector problem.

Finally, we tried up to N=15000, but for N > 7000, the preference matrix generation would take too long.

See code at the end.

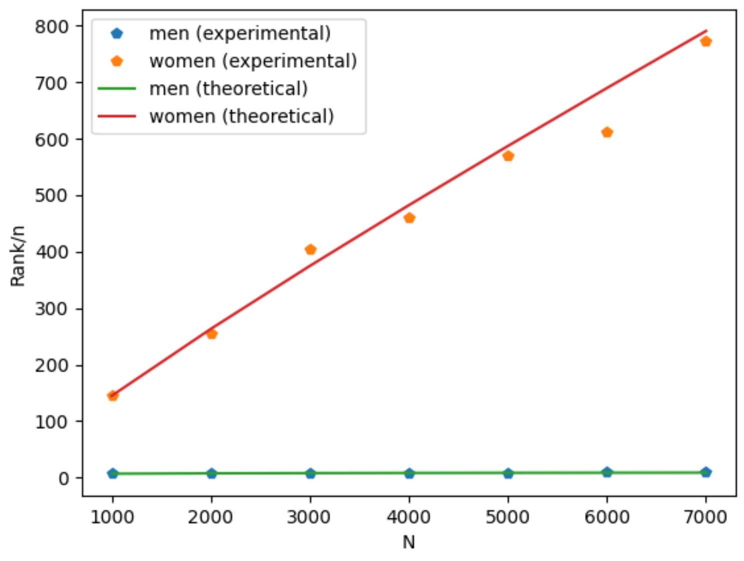

Figure 1: Goodness of matches for men and women.

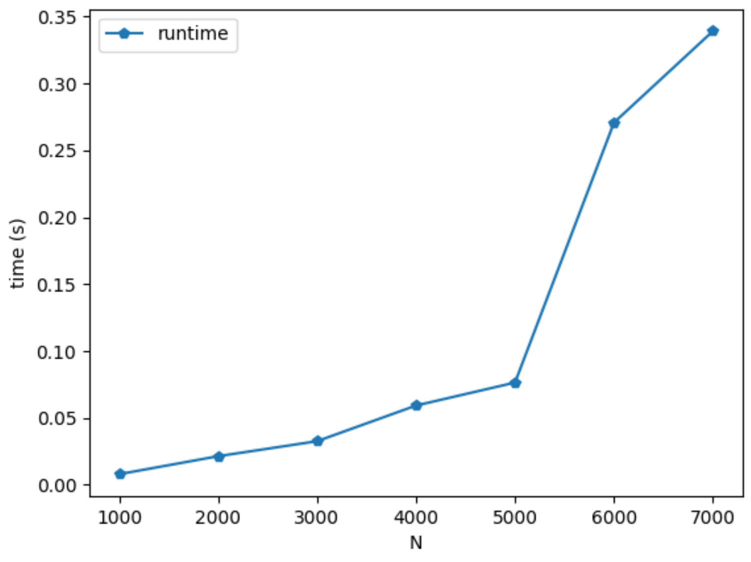

Figure 2: Runtime O(NlogN) until around N=5000 when swap kicks in.

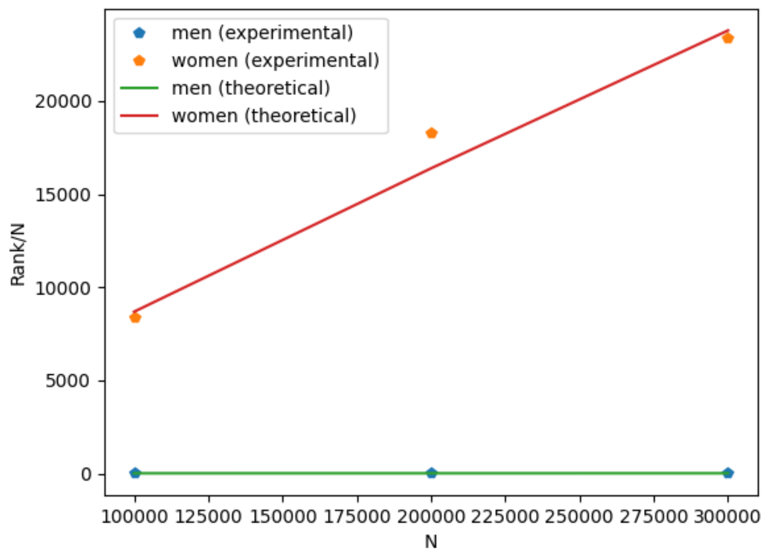

Figure 3: Part I: Goodness of matches for men and women for large N.

Exercise 6 (10 pts)

The results are shown in Figure 3, 4, 5 and 6. The tests were run in chunks because the time it took and python garbage collection once you hit the swap isn't great and impacted performance. Finally we tried N equal to 3 million that completed in 1623.41 seconds. The runtime seem to increase at rate of NlogN for sub 2 million, whereas at 3 million, we hit the swap. It is unclear why we hit the swap since the total memory used should be 2N*log(N)+3N which for even 4 million sets works out to 1.4 Gb but in practice exceeds 16 Gb. At 3 million, this was around 12 Gb even though it should be 1 Gb given integers have 8 bytes.

The goodness findings were the same as question 5. The insight to scaling up to larger N is that for the men, only the women he has proposed to up to the current round in the algorithm need to be generated. Similarly women need only keep track of their preference of men who've proposed. Both these numbers are log(N) which significantly decreases memory needed from N^2 to $Nlog(N)$. To maintain a reasonable runtime, we also have to be clever on how we draw uniformly from N given a list of numbers we've already drawn. To keep the runtime reasonable, this cannot be O(N). Therefore we use another randomized algorithm that repeatedly draw numbers until one that is not seen has been drawn. Since no more than Log(N) numbers are expected to be drawn per man or woman, we have a high probability that this is an O(1) operation. Similarly to keep track of the free men list, we use a linked list to maintain O(1) operation.

One interesting observation is that the algorithm is extremely fast until the number of free men become less than a few thousand at which point each man must try almost every woman before they find one that will match and repeat this over many rounds until an unmatched woman was found that can advance the progress of the algorithm. Most of the attempts seem to be in this fat tail within this group.

See code at the end.

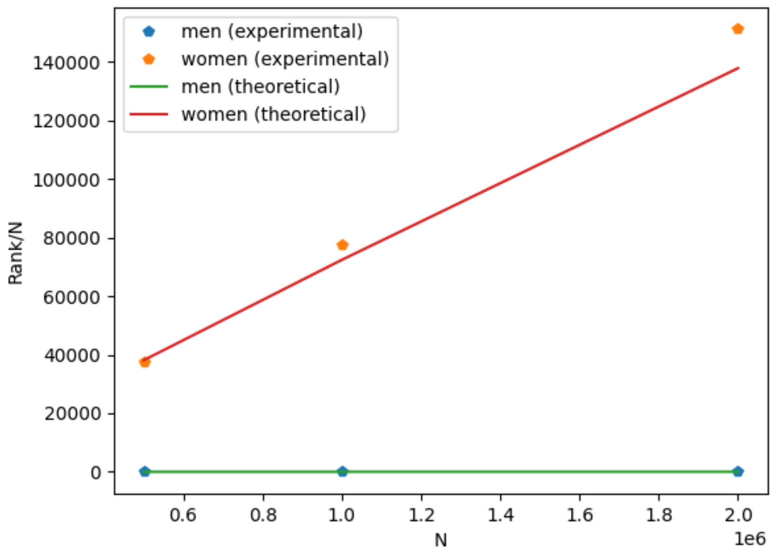

Figure 4: Part II: Goodness of matches for men and women for larget N.

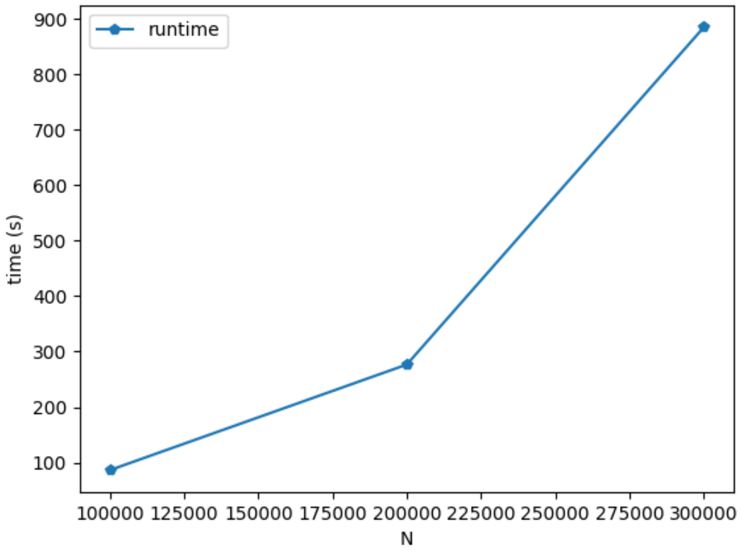

Figure 5: Part I: Runtime for large N.

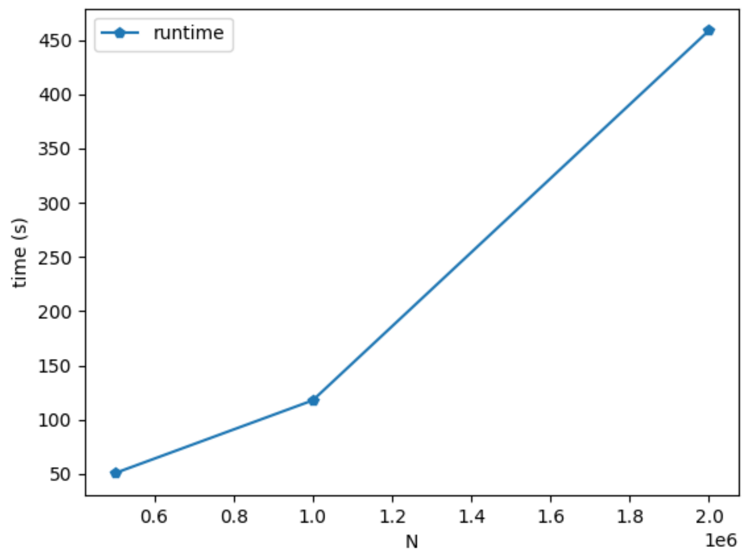

Figure 6: Part I(: Runtime for large N.

Code for Question 5

```
def generate(N: int):
  arr = [i for i in range(N)]

  for i in range(N):
    j = int(random.uniform(0, i+1))
    tmp = arr[i]
    arr[i] = arr[j]
    arr[j] = tmp

  return arr

class StableMatcher():
  """
  p_matrix: proposer matrix
  a_matrix: accepter matrix
  """
  def __init__(self, p_matrix, a_matrix):
    self._p = p_matrix
    self._a = a_matrix
    self.N = len(p_matrix)

  def analyze(self, results: dict,
    p_rank: list, a_matrix_invert=False):
    MRank = 0
    WRank = 0

    for i in p_rank:
      MRank += i

    for w, m in results.items():
      if a_matrix_invert:
        WRank += self._a[w].index(m)
      else:
        WRank += self._a[w][m]

    return MRank/self.N, WRank/self.N

  def match(self, a_matrix_invert=False, log=False):
    if log:
      print("========================================")
      print("Input (Proposer):")
      print(self._p)
      print("Input (Accepter):")
      print(self._a)
      print("Trace")

    # every proposer start free with their top preference
    p_free = deque( [ i for i in range(self.N) ] )
     matched = {}

    p = 0
    st = time.time()
    while len(p_free) > 0:
      # the order of proposers do not matter,
      # we pick first from a random permutation
      p = p_free.popleft()
      w = self._p[p][p_next[p]]

      if w not in matched:
        matched[w] = p
        if log:
          print(f"{p} proposed to {w} [{w}, -1] Accepted")
      else:
        p_prime = matched[w]
        if a_matrix_invert:
          p_index = self._a[w].index(p)
          p_prime_index = self._a[w].index(p_prime)
        else:
          p_index = self._a[w][p]
          p_prime_index = self._a[w][p_prime]
        if p_index < p_prime_index:
          matched[w] = p
          p_free.append(p_prime)
          if log:
            print(f"{p} proposed to {w} [{w}, {p_prime}] Accepted")
```

```
                else:
                    p_free.append(p)
                    if log:
                        print(f"{p} proposed to {w} [{w}, {p_prime}] Rejected")
            p_next[p] += 1

        if log:
            print("Output:")
            print(matched)

        return matched, p_next, time.time() - st
```

Code for Question 6

```
class StableMatcher():
    def __init__(self, N):
        self._p = { i: set() for i in range(N) }
        self._p_free = {}
        self._p_size = 0
        self._p_index = 0
        # tracks the preference ranking already assigned for a
        self._a = { i: set() for i in range(N) }
        self._a_free = {}
        self._a_size = 0
        self._a_free = {}
        self._a_index = 0
        # tracks the preference ranking of p for a
        self._a_p = { i: {} for i in range(N) }
        self._prefset = { i for i in range(N) }
        self.threshold = 0.999
        self.N = N
        self.N2 = self.N*self.N
        self.matched = {}

    def generateNextA(self, p):
        p_pref = self._p[p]

        if len(p_pref) > self.threshold*self.N:
            if p not in self._p_free:
                self._p_free[p] = list(self._prefset - p_pref)
                for i in range(len(self._p_free[p])):
                    j = int(random.uniform(0, i+1))
                    tmp = self._p_free[p][i]
                    self._p_free[p][i] = self._p_free[p][j]
                    self._p_free[p][j] = tmp
            a = self._p_free[p][self._p_index]
```

```
      self._p_index += 1
    else:
      a = None
      while a is None or a in p_pref:
        a = int(random.uniform(0, self.N))

    p_pref.add(a)
    self._p_size += 1
    return a

  def generateRankP(self, a, p):
    a_pref = self._a[a]
    a_p_pref = self._a_p[a]
    if p in a_p_pref:
      return a_p_pref[p]
    if len(a_pref) > self.threshold*self.N:
      if a not in self._a_free:
        self._a_free[a] = list(self._prefset - a_pref)
        for i in range(len(self._a_free[a])):
          j = int(random.uniform(0, i+1))
          tmp = self._a_free[a][i]
          self._a_free[a][i] = self._a_free[a][j]
          self._a_free[a][j] = tmp
      a = self._a_free[a][self._a_index]
      self._a_index += 1
    else:
      r = None
      while r is None or r in a_pref:
        r = int(random.uniform(0, self.N))

    a_pref.add(r)
    a_p_pref.update({p:r})
    self._a_size += 1
    return r

  def analyze(self):
    MRank = 0
    WRank = 0
    for p in self._p:
      MRank += len(self._p[p])
    for a, p in self.matched.items():
      WRank += self._a_p[a][p]
    return MRank/self.N, WRank/self.N

  def match(self, log=False):
    """
    Args:
      log - True to print trace of matchings
```

```
"""
# every proposer start free with their top preference
p_free = deque( [i for i in range(self.N)] )

while len(p_free) > 0:
  p = p_free.popleft()
  w = self.generateNextA(p)
  if w not in self.matched:
    self.generateRankP(w, p)
    self.matched[w] = p
  else:
    p_prime = self.matched[w]
    p_index = self.generateRankP(w, p)
    p_prime_index = self.generateRankP(w, p_prime)
    if p_index < p_prime_index:
      self.matched[w] = p
      p_free.append(p_prime)
    else:
      p_free.append(p)
```