CSEP 521, Winter 2021: Homework 3

Krishan Subudhi (ksubudhi@uw.edu) - 2040900

January 28, 2021

Problem 1

Unique keys possible with 128 bits = d = 2^{128}

$$p = 1\% = 0.01$$

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

$$= \sqrt{2 * 2^{128} \ln \frac{1}{1-0.01}}$$

Here n is the number of keys that need to be generated until there is a 1% probability that two keys are same.

n is the number of transactions. 1 billion users are generating 1 thousand transactions per day. Hence the user can record data until $n/1E9/1000/365$ years.

```
n/1E9/1000/365 = math.sqrt(math.pow (2,129) * math.log(1/0.99))/1E12/365
```

= 7165.263 years

Hence Mark can record data until 7165 years until there is 1% chance of having two transactions with the same key.

Problem 2

a) m is ranked first on w's preference list.
w is ranked first on m's preference list.

We have to prove that for every stable matching for $I(M, W)$, (m, w) must be the matching.

Let's prove this through contradiction. Let's assume that in one of the stable matching instances S, m is matched with w' and w is matched with m'.

However, since m prefers w over w' and w prefers m over m', (m, w') , (m', w) is an instability.

This contradicts our claim that S is stable and hence proves that every stable matching instance must match m with w.

b) In the given instance of stable matching problem $I = (M, W)$,

$$\text{Preference list of all } m \in M \text{ is } [w_1, w_2, ..., w_n] \tag{2.1}$$

For a particular I, preference list is fixed for for every $m \in M$ and every woman $w \in W$

We have to prove that *There is a unique solution to this instance*

Let's say in the stable matching problem I for any random $0 < i, j < n$,

$$w_1 \text{ prefers } m_i \text{ over } m_j \tag{2.2}$$

Let assume that there are two stable matching possible for instance I - S and S'. In S, w_1 is matched with m_i and in S' w_1 is matched with m_j.

For $w', w'' \in [w_2, w_3, ..., w_n]$

$$S : (m_i, w_1), (m_j, w')..$$
$$S' : (m_j, w_1), (m_i, w'')..$$

Since w_1 prefers m_i over over m_j (as per 2.2), and m_i prefers w_1 over w' and w'' (as per 2.1), S' will not be stable match - contradicting our initial assumption. Hence the only match possible for w_1 is (w_1, m_i).

Now since (w_1, m_i) is the only stable match possible for w_1 and m_i, we can remove w_1 and m_i from the preference lists and match others. After removing w_1, the preference list for men becomes $[w_2, ..., w_n]$. Applying the same logic as above, we can prove that there is an unique stable match for w_2 too. hence it can be proven by induction that each w_i will have an a unique stable match. Since each w_i has a unique stable match, there is only one unique stable solution possible for I.

Problem 3

Q: Show that a participant can improve its outcome by lying about its preferences.

In Gale-Shapley algorithm, participants can imrpove their chances of achieving a better match if they switch the order of their preferences in some cases. Let's say the preference for women is in the following order:

$$\begin{aligned} w &: [m'', m, m'] \\ w' &: [m, m'', m'] \\ w'' &: [m, m'', m'] \end{aligned}$$

Let's say preference list of men is

$$\begin{aligned} m &: [w, w', w''] \\ m' &: [w, w', w''] \\ m'' &: [w', w, w''] \end{aligned}$$

If we run the Gale-Shapley on the preference lists above, we obtain the final match to be $m - w, m'' - w', m' - w''$

In this case m is higher in preference list of women other than w. But since w is first preference of m, and m proposes first in Gayle Shapely algorithm, w will accept m's proposal and w' will accept m'''s proposal. The algorithm still produces stable match even though w does not get her first preference.

Now let's say w lies about her preference.

$$\begin{aligned} w &: [m'', m', m] \\ w' &: [m, m'', m'] \\ w'' &: [m, m'', m'] \end{aligned}$$

If we run the Gale-Shapley on the updated preference list of women, we obtain the match to be $m - w', m'' - w, m' - w''$, noticing that w1 ends up with her first preference.

Hence w was matched m'' which she prefers higher than both m and m'. This proves that lying about preference can increase the chances to get a better match.

Problem 4

Q) Implement the stable matching algorithm.

Here I have implemented the Gayle Shapely algorithm. Men propose to women and women accept the proposal if they are either unmatched or the proposal is preferred compared to the current match. The specific implementation uses a set of unmatched men.

```
men
[[2 1 3 0]
 [0 1 3 2]
 [0 1 2 3]
 [0 1 2 3]]

women
[[0 2 1 3]
 [2 0 3 1]
 [3 2 1 0]
 [2 3 1 0]]
0 proposes to 2 [2,-1] Accepted
1 proposes to 0 [0,-1] Accepted
2 proposes to 0 [0,1] Accepted
3 proposes to 0 [0,2] Rejected
3 proposes to 1 [1,-1] Accepted
1 proposes to 1 [1,3] Rejected
1 proposes to 3 [3,-1] Accepted

matches
 [2, 3, 0, 1]
```

```python
# question4.py

import numpy as np
import dataclasses


@dataclasses.dataclass
class Person:
    index: int
    pref_list: list
    current_match: int = -1  # accepted

class Man(Person):
    total_proposed: int = 0  # may or may not be accepted
    def get_next_woman(self):
        return self.pref_list[self.total_proposed]
    @property
    def mrank(self):
        return self.total_proposed
```

```
class Woman(Person):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.pref_dict = {m: i for i, m in enumerate(self.pref_list)}
        del self.pref_list
    def __str__(self):
        return super().__str__() + str(self.pref_dict)
    def __repr__(self):
        return super().__repr__() + str(self.pref_dict)
    @property
    def wrank(self):
        return self.pref_dict[self.current_match]


class GayleShapely:
    def __init__(self, m: list, w: list, trace=True):

        m = np.array(m)
        w = np.array(w)
        self.men = [Man(i, pref) for i, pref in enumerate(m)]
        self.women = [Woman(i, pref) for i, pref in enumerate(w)]
        self.trace = trace

        self.free_men = set([m.index for m in self.men])
        if trace:
            print('men')
            print(m)
            print('women')
            print(w)

    def _get_next_free_m(self):
        return self.men[self.free_men.pop()] if len(self.free_men) >0 else None

    def _matched(self, man: Man, woman: Woman):
        man.current_match = woman.index
        if woman.current_match > -1:
            self.men[woman.current_match].current_match = -1
            self.free_men.add(woman.current_match)
        woman.current_match = man.index




    def _propose(self, man, woman):
        man.total_proposed += 1
        result = "Rejected"
        woman_prev_match = woman.current_match
        if (
            woman.current_match == -1
            or woman.pref_dict[man.index] < woman.pref_dict[woman.current_match] #this
                                                    is rank so lower is better
        ):
            self._matched(man, woman)
            result = "Accepted"

        if self.trace:
            print(
                f"{man.index} proposes to {woman.index} [{woman.index},{woman_prev_match
```

```
                                                                }] {result}"
            )
        return result == "Accepted"

    def match(self):
        """
        m and w are objects
        """
        m = self._get_next_free_m()

        while m is not None:
            accepted = False
            while not accepted:
                # select next woman
                w = self.women[ m.get_next_woman()]
                # _propose
                accepted = self._propose(m, w)
            m = self._get_next_free_m()
        if self.trace:
            print('matches\n',self.get_matches())

    def get_matches(self):
        return [m.current_match for m in self.men]

    def get_mranks(self):
        return [m.mrank for m in self.men]


    def get_wranks(self):
        return [w.wrank for w in self.women]

    @property
    def MGoodness(self):
        return sum(self.get_mranks())/len(self.men)

    @property
    def WGoodness(self):
        return sum(self.get_wranks())/len(self.women)

def main():
    m = [
        [2,1,3,0],
        [0,1,3,2],
        [0,1,2,3],
        [0,1,2,3]
    ]

    w = [
        [0,2,1,3],
        [2,0,3,1],
        [3,2,1,0],
        [2,3,1,0]
    ]
    algo = GayleShapely(m,w)
    algo.match()

if __name__ == "__main__":
    main()
```

Problem 5

Q. Write an input generator which creates completely random preference lists, so that each M has a random permutation of the W's for preference, and vice-versa

```
from question4 import Man, Woman, GayleShapely
import random
import numpy as np
import time
import pandas as pd
import matplotlib.pyplot as plt
import math
plt.style.use('ggplot')

def generate_random_permutations(n):
    return [np.random.permutation(range(n)).tolist() for i in range(n)]

def main():
    results = {}
    for n in [8000]:
        random.seed(46)

        m = generate_random_permutations(n)
        w = generate_random_permutations(n)

        algo = GayleShapely(m, w, trace = False)

        start = time.time_ns()/1000
        algo.match()
        end = time.time_ns()/1000

        total_time = end-start

        results[n]= pd.Series({
                'time_micros':total_time,
                'MGoodness':algo.MGoodness,
                'WGoodness':algo.WGoodness,
                'time_ms/CouponCollector':total_time/theoritical_cn(n)
            })
    df =  pd.DataFrame(results).T
    return df

def theoritical_cn(n):
    return n * math.log(n) + 0.57 * n

if __name__=='__main__':
    result = main()
    print(result)

    fig, axes = plt.subplots(2,2, sharex = True)
    axes = axes.reshape(-1)
    # print(axes)
    for i, col in enumerate(result.columns):
        axes[i].plot(result.index, result[col])
        axes[i].set_ylabel(col)
        axes[i].set_xlabel('n')
    plt.show()
```

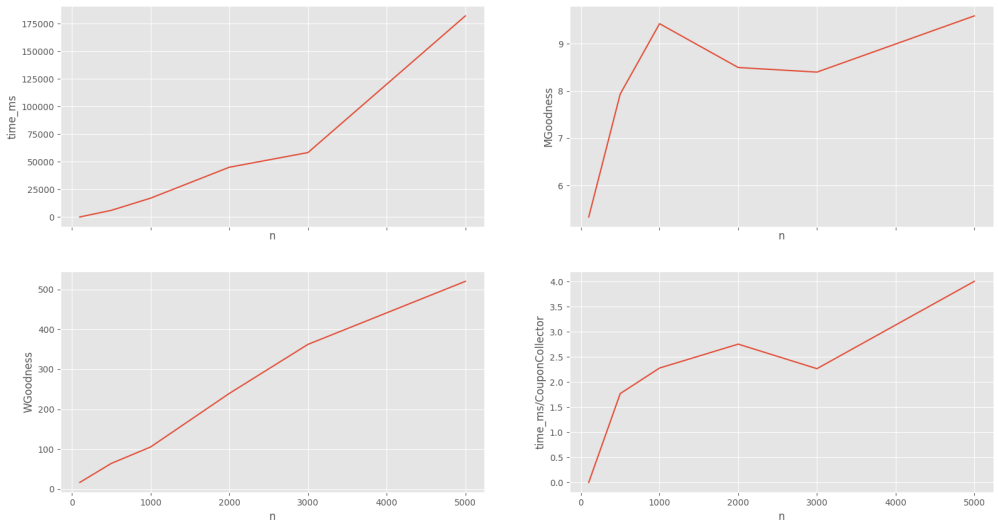

Figure 1: Growth rate of Goodness and run time

Table 1: Question 5 results

n	time(μs)	MGoodness	WGoodness	time/CouponColl	MGoodness/logn	Wgoodness/n	time/nlogn
500	6000.25	7.936000	64.3340	1.768783	1.276991	0.128668	1.931015
1000	17031.25	9.421000	105.5240	2.277589	1.363829	0.105524	2.465526
2000	44999.00	8.495500	239.4845	2.753613	1.117696	0.119742	2.960109
3000	58276.50	8.398667	362.2550	2.265003	1.048998	0.120752	2.426256
5000	181921.75	9.585400	519.9838	4.003915	1.125418	0.103997	4.271871
8000	270089.50	7.641625	1026.2355	3.532541	0.850279	0.128279	3.756587

1. Q. how does the goodness change for M and W?

   MGoodness grows at the rate of $O(logn)$ while WGoodness grows at a rate around $O(n)$. The growth rates might not be exact but the constant ratio gives an indication of the upper bound. Clearly the algorithm is not favourable for women.

2. Q. How does the run time of the code vary as a function of n?

   Run time grows approximately at a rate of $O(nlogn)$

3. Q. How do your results relate to result from the coupon collector problem?

   The ratio of runtime and Coupon Collector theoritical value is almost a constant at every n. Approximately the growth rates seem to match.

   Note: the constant values for growth rates wrt time are slightly higher for higher n. But there can be other constraints like ineffecient memory management, processor throttling which might increase runtime at higher n. But the overall constant ratio makes this algorithm runtime comparable with coupon collector.

Problem 6

Q. Rewrite your stable matching generator, so that you compute stable matchings for very large, random instances.

```
import numpy as np
import dataclasses
import random
import sys

'''
Rewrite your stable matching generator, so that you compute stable matchings for very
                                        large, random instances. The key idea is to
                                        generate the random permutations
                                        incrementally, so that you only construct
                                        the random preferences when they are needed
'''
@dataclasses.dataclass
class Man():
    index: int
    n:int
    proposed_set: set = dataclasses.field(default_factory=set)
    current_match: int = -1  # accepted

    def get_random_proposee(self):
        random_woman = random.randint(0,self.n-1)
        if random_woman in self.proposed_set:
            return self.get_random_proposee()
        self.proposed_set.add(random_woman)
        return random_woman

    def accepted(self,w):
        self.current_match = w

    def dumped(self):
        self.current_match = -1

    @property
    def mrank(self):
        assert self.current_match > -1
        return len(self.proposed_set)

    def __repr__(self):
        return str(self.index)

@dataclasses.dataclass
class Woman():
    index: int
    n:int
    proposed_set: set = dataclasses.field(default_factory=set)
    current_match: int = -1  # accepted
    current_match_rank: int = -1

    def accept_proposal(self, m):
        preference = self.get_random_rank()
        if preference > self.current_match_rank:
            self.current_match = m
            self.current_match_rank = preference
```

```
            return True
        return False

    def get_random_rank(self):
        random_rank = random.randint(0,self.n-1)
        if random_rank in self.proposed_set:
            return self.get_random_rank()
        self.proposed_set.add(random_rank)
        return random_rank

    @property
    def wrank(self):
        return self.current_match_rank


    def __repr__(self):
        return str(self.index)


class GayleShapelyRandom:
    def __init__(self, n, trace=True):

        self.men = [Man(i, n) for i in range(n)]
        self.women = [Woman(i, n) for i in range(n)]
        self.trace = trace

        self.free_men = set([m.index for m in self.men])
        if trace:
            print('men')
            print(self.men)
            print('women')
            print(self.women)

    def _get_next_free_m(self):
        return self.men[self.free_men.pop()] if len(self.free_men) >0 else None

    def _matched(self, man: Man, woman: Woman, woman_prev_match:int):
        man.accepted(woman.index)
        if woman_prev_match > -1:
            self.men[woman_prev_match].dumped()
            self.free_men.add(woman_prev_match)


    def _propose(self, man, woman):

        woman_prev_match = woman.current_match
        accepted = woman.accept_proposal(man.index)

        if accepted:
            self._matched(man, woman, woman_prev_match)

        if self.trace:
            print(
                f"{man.index} proposes to {woman.index} [{woman.index},{woman_prev_match
                                                                    }] {accepted}"
            )
        return accepted
```

```
    def match(self):
        """
        m and w are objects
        """
        m = self._get_next_free_m()

        while m is not None:
            accepted = False
            while not accepted:
                # select next woman
                w = m.get_random_proposee()
                w = self.women[w]
                # _propose
                accepted = self._propose(m, w)
            m = self._get_next_free_m()
        if self.trace:
            print('matches\n',self.get_matches())

    def get_matches(self):
        return [m.current_match for m in self.men]

    def get_mranks(self):
        return [m.mrank for m in self.men]


    def get_wranks(self):
        return [w.wrank for w in self.women]

    @property
    def MGoodness(self):
        return sum(self.get_mranks())/len(self.men)

    @property
    def WGoodness(self):
        return sum(self.get_wranks())/len(self.women)

import time
import pandas as pd
def main2():
    results = {}
    random.seed(47)
    for n in [1_000_000]:#[1_000, 10_000, 100_000]:
        print(f'n =  {n}')
        total_time = 0
        MGoodness = 0
        WGoodness = 0
        iterations = 2
        for i in range(iterations):
            print(f'iteration {i+1}')
            algo = GayleShapelyRandom(n, trace = False)
            start = time.time()
            algo.match()
            end = time.time()

            total_time += (end-start)
            MGoodness += algo.MGoodness
            WGoodness += algo.WGoodness
```

```
        results[n]= pd.Series({
                'time_s':total_time/iterations,
                'MGoodness':MGoodness/iterations,
                'WGoodness':WGoodness/iterations
            })
    df =  pd.DataFrame(results).T
    print(df)
    return df

if __name__ == "__main__":
    main2()
```

Q. Write a summary of your results for large n, including both the statistics for rank/goodness, as well as recorded run-time of the program.

Table 2: Question 6 results

n	time_s	MGoodness	WGoodness	mGoodness/logn	Wgoodness/n	time/nlogn
1000	0.043	7.026	861.776	1.017	0.861	6E-6
10000	0.602	11.491	9087.755	1.247	0.908	7E-6
100000	14.337	11.838	91496.482	1.028	0.914	1.2E-5
1000000	267.126	14.845	932650.559	1.074	0.932	1.9E-5

Q. What does the estimated growth rate in runtime

All results are averaged over 5 iterations. Growth rate of MGoodness is $O(logn)$ and growth of Wgoodness is $O(n)$. Runtime is higher than $O(nlogn)$ but lower than $O(n^2)$. Run time is calculated as system time and can involve CPU throttling and memory inefficiency at higher n too. Random preference creation is also part of runtime which might be contributing to higher than $nlogn$ run time.

Q. In your write up describe your process for incrementally generating random permutations.

Instead of creating all the random permutations beforehand which required $O(n^2)$ memory, the permutations are generated incrementally.

When a particular man m had to propose a new woman, the woman w was selected randomly from the set of all women to which m had never proposed previously. Each men only stored information of women they had previously proposed. Since each m proposes $logn$ women on average, the space requirement reduced to $O(nlogn)$.

Similarly, for a woman w, only ranks of those men were stored which had already proposed her. When a new man proposed, his rank was selected randomly from a set of all possible ranks minus the ranks of men who had previously proposed w. Since each m proposes $logn$ women on average, each woman was proposed $logn$ times on average too. The space requirement for women also reduced to $O(nlogn)$.