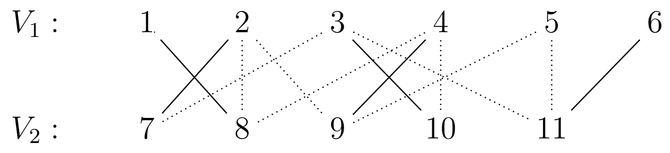
Name: $\boxed{\text{Jon Abrahamson}}$
ID: $\boxed{107084898}$

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 10 **Spring 2019, CU-Boulder**

Quick links 1a 1b 1c 2a 2b 2c 3a 3b

1. A *matching* in a graph G is a subset $E_M \subseteq E(G)$ of edges such that each vertex touches at most one of the edges in E_M. Recall that a bipartite graph is a graph G on two sets of vertices, V_1 and V_2, such that every edge has one endpoint in V_1 and one endpoint in V_2. We sometimes write $G = (V_1, V_2; E)$ for this situation. For example:



The edges in the above example consist of all the lines, whether solid or dotted; the solid lines form a matching.

The *bipartite maximum matching* problem is to find a matching in a given bipartite graph G , which has the maximum number of edges among all matchings in G.

Name: Jon Abrahamson
ID: 107084898

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 10 **Spring 2019, CU-Boulder**

(a) (6 pts total) Prove that a maximum matching in a bipartite graph $G = (V_1, V_2; E)$ has size at most $\min\{|V_1|, |V_2|\}$.

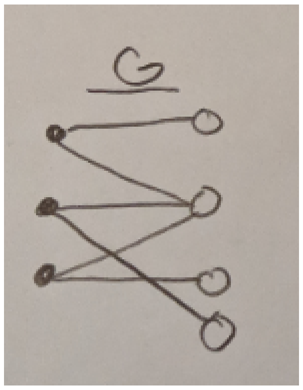
The maximum matching problem allocates each vertex into a pair of vertices $[V_a, V_b]$ connected by an edge from E_M Once a vertex has been paired, it cannot be connected to any other vertex by an edge from E_M For two sets of vertices $[V_1, V_2]$ there can't be any other edges added to E_M once the smaller set of vertices have been completely matched because a vertex from the smaller set would then have more than one edge. Therefore, there can only be at most $\min\{|V_1|, |V_2|\}$ edges in E_M.

Additionally each vertex can only have a maximum of one matching edge. Therefore, the size of maximum matching can't exceed the vertices in either V_1 or V_2. So $min|V_1|, |V_2|$ is valid. Even if we assume $min|V_1|, |V_2| = |V_1|$ as a matching has more than $|V_1|$ edges, the matching must have different end points in V_1, which is greater than V_1. Therefore, there must be at least two vertices with common end points making the matching not valid. So the assumption is false and the matching can have at most $|V_1|$ edges.

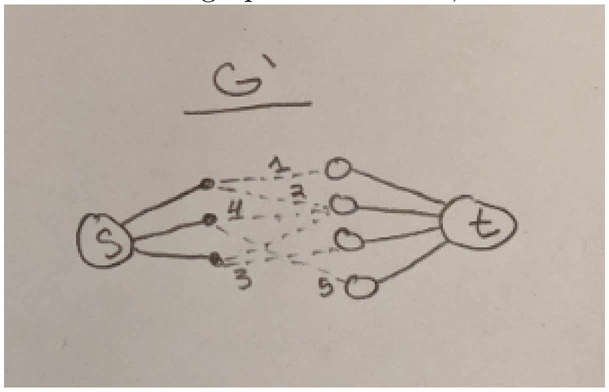
Name: Jon Abrahamson
ID: 107084898

(b) (8 pts total) Show how you can use an algorithm for max-flow to solve bipartite maximum matching on undirected simple bipartite graphs. That is, give an algorithm which, given an undirected simple bipartite graph $G = (V_1, V_2; E)$, (1) constructs a directed, weighted graph G' (which need not be bipartite) with weights $w : E(G') \to \mathbb{R}$ as well as two vertices $s, t \in V(G')$, (2) solves max-flow for $(G', w), s, t$, and (3) uses the solution for max-flow to find the maximum matching in G. Your algorithm may use any `max-flow` algorithm as a subroutine.

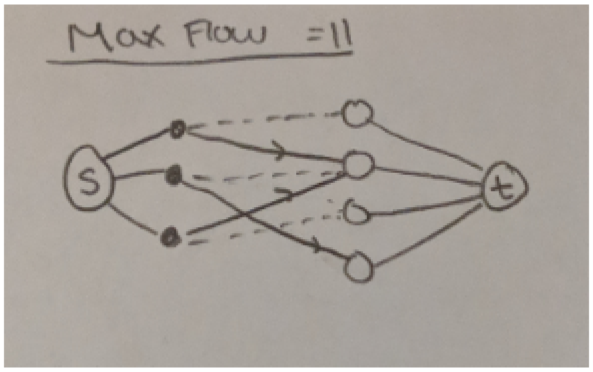
Given Graph G :



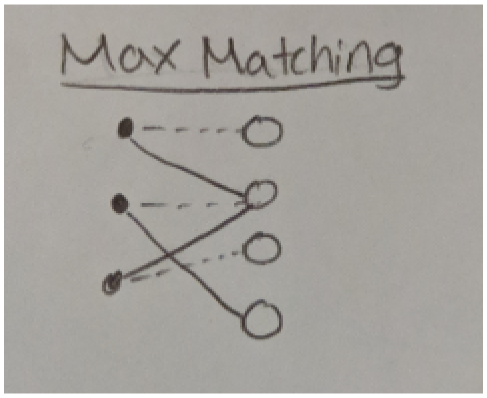
1) We first create vertices s, t where s has edges that connect to V_1 and t has edges to V_2. Next we assign weights to each edge. Here the weights will be assigned randomly from 1 and 10 (the weights could represent flow of water in pounds). The algorithm would also assign directions to the graph. In this case, the direction of the edges will all be towards t.



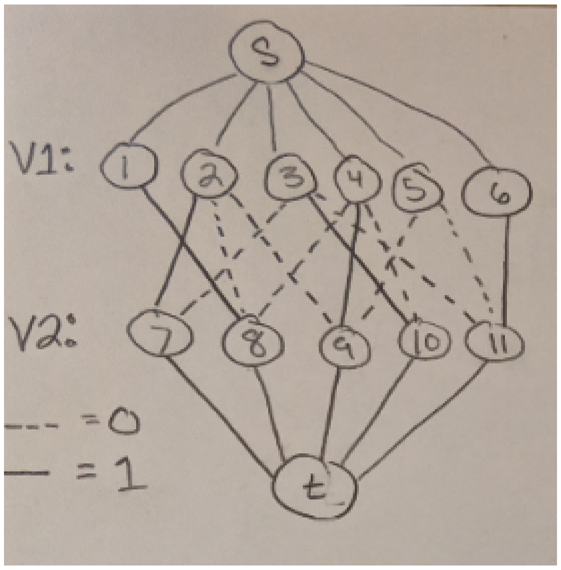
2) We then solve for max-flow using ford-fulkerson. The greedy algorithm would choose a s-t path that has the greatest weight. It would then check both that each vertex in $(|V_1|, |V_2|)$ have at max one edge connecting them and that the number of paths between $(|V_1|, |V_2|)$ does not exceed $\min\{|V_1|, |V_2|\}$.



3) Using the max-flow, we add the edges from the solution paths to E_M which represents the edges that solve bipartite max matching in G.



(c) (7 pts total) Show the weighted graph constructed by your algorithm on the example bipartite graph above.



By giving weight 1 to the dark lines and weight 0 to the dotted lines, my algorithm identifies the max-flow through the weighted lines and stops assigning lines to E_M once their is $\min\{|V_1|, |V_2|\}$ edges.

Name: $\boxed{\text{Jon Abrahamson}}$
ID: $\boxed{107084898}$

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 10 **Spring 2019, CU-Boulder**

2. In the review session for his Deep Wizarding class, Dumbledore reminds everyone that the logical definition of NP requires that the number of *bits* in the witness w is polynomial in the number of bits of the input n. That is, $|w| = poly(n)$. With a smile, he says that in beginner wizarding, witnesses are usually only logarithmic in size, i.e., $|w| = O(\log n)$.

(a) (7 pts total) Because you are a model student, Dumbledore asks you to prove, in front of the whole class, that any such property is in the complexity class P.

In general, $P \subseteq NP$ which means that any w that can be solved in NP can be solved in P. For any w with input n in the logical definition of NP, $|w| = poly(n)$. A basic example that shows $|w| = O(log(n))$s solving for a numbers greatest common divisor; where $n = 2$ for the two integer inputs and w is their gcd. Solving for the gcd would be $O(log(n))$ based on the number of inputs which is in Polynomial time. Any such property that is a subset of NP is in the complexity class P.

(b) (6 pts total) Well done, Dumbledore says. Now, explain why the logical definition of NP implies that any problem in NP can be solved by an exponential-time algorithm.

Exponential time means that every possible type of witness is used to solve the problem. In other words, any problem can be solved in exponential time. Because $P \subseteq NP \subseteq Exponential$,any problems that can be solved in NP can be solved in exponential time.

(c) (6 pts total) Dumbledore then asks the class: “So, is NP a good formalization of the notion of problems that can be solved by brute force? Discuss.” Give arguments for both possible answers.

No - A better formalization of this notion is that Exponential Time encompasses problems that can be solved by pure brute force.

Yes - The range of questions that can be solved by NP can also be solved by EXP (by the same logic in part b), therfore they can be nearly solved in the same brute force method.

Name: Jon Abrahamson
ID: 107084898

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 10 **Spring 2019, CU-Boulder**

3. (20 pts) Recall that the *MergeSort* algorithm is a sorting algorithm that takes $\Theta(n \log n)$ time and $\Theta(n)$ space. In this problem, you will implement and instrument MergeSort, then perform a numerical experiment that verifies this asymptotic analysis. There are two functions and one experiment to do this.

   (i) `MergeSort(A,n)` takes as input an unordered array A, of length n, and returns both an in-place sorted version of A and a count t of the number of atomic operations performed by `MergeSort`.

   (ii) `randomArray(n)` takes as input an integer n and returns an array A such that for each $0 \leq i < n$, $A[i]$ is a uniformly random integer between 1 and n. (It is okay if A is a random permutation of the first n positive integers.)

Name: Jon Abrahamson
ID: 107084898

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 10 **Spring 2019, CU-Boulder**

(a) (10 pts total) From scratch, implement the functions `MergeSort` and `randomArray`. You may not use any library functions that make their implementation trivial. You may use a library function that implements a pseudorandom number generator in order to implement `randomArray`.

Submit a paragraph that explains how you instrumented `MergeSort`, i.e., explain which operations you counted and why these are the correct ones to count.

```
def randomArray(n):
    A = np.random.uniform(1, n, size=n)
    return A

def merge(A, left, mid, right, count):
    p = []
    L = MergeSort(A[:mid])
    R = MergeSort(A[mid:])
    lengthL = len(L)
    lengthR = len(R)
    count += 4

    while lengthL != 0 and lengthR != 0:
        if L[0] < R[0]:
            p.append(L[0])
            L.remove(L[0])
            count += 3
        else:
            p.append(R[0])
            R.remove(R[0])
            count += 2

    if lengthL == 0:
        p += R
        count += 3
    else:
        p += L
        count += 1
    return p, count
```

```
def MergeSort(A, n, count):
    mid = math.floor(n/2)
    left = len(A[:mid])
    right = len(A[mid:])
    count += 1
    if (left < right):
        MergeSort(A, left, count)
        MergeSort(A, right, count)
        A, count = merge(A, left, mid, right, count)
    return (A, count)
```

```
I implemented merge sort by constructing an array out of the left and
    right parts of the original array. This algorithm operates
    recursively. I counted the atomic operations by adding up the
    number of append, remove, and equals operations in the merge
    function.
```

(b) (10 pts total) For each of $n = \{2^4, 2^5, \ldots, 2^{26}, 2^{27}\}$, run `MergeSort(randomArray(n),n)` fives times and record the tuple $(n, \langle t \rangle)$, where $\langle t \rangle$ is the average number of operations your function counted over the five repetitions. Use whatever software you like to make a line plot of these 24 data points; overlay on your data a function of the form $T(n) = A\, n \log n$, where you choose the constant A so that the function is close to your data.

Hint 1: To increase the aesthetics, use a log-log plot.
Hint 2: Make sure that your *MergeSort* implementation uses only two arrays of length n to do its work. (For instance, don't do recursion with pass-by-value.)

```
def sim(n):
    count = 0
    averages = []
    trials = [2**i for i in range(4, n)]
    model = [14 * (j * np.log2(j)) for j in trials]
    for i in trials: #2^(4-26)
        sum = 0
        for j in range(0, 5):
            print("Trial: ", i, "-", j)
            sum += MergeSort(randomArray(i), i, count)[1]
        averages.append(sum/5)
    print(averages)
    plt.loglog(trials, averages)
    plt.show()

sim(26)
```

```
Trial:  16 - 0
Trial:  16 - 1
Trial:  16 - 2
Trial:  16 - 3
Trial:  16 - 4
Trial:  32 - 0
Trial:  32 - 1
Trial:  32 - 2
Trial:  32 - 3
Trial:  32 - 4
Trial:  64 - 0
Trial:  64 - 1
```

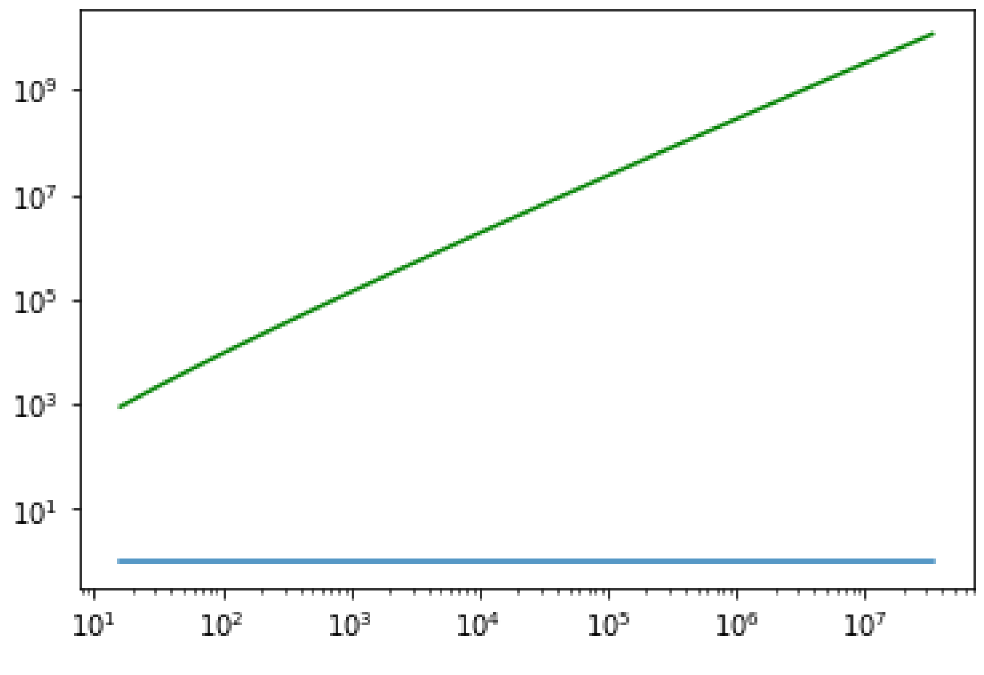
```
Trial: 64 - 2
Trial: 64 - 3
Trial: 64 - 4
Trial: 128 - 0
Trial: 128 - 1
Trial: 128 - 2
Trial: 128 - 3
Trial: 128 - 4
Trial: 256 - 0
Trial: 256 - 1
Trial: 256 - 2
Trial: 256 - 3
Trial: 256 - 4
Trial: 512 - 0
Trial: 512 - 1
Trial: 512 - 2
Trial: 512 - 3
Trial: 512 - 4
Trial: 1024 - 0
Trial: 1024 - 1
Trial: 1024 - 2
Trial: 1024 - 3
Trial: 1024 - 4
Trial: 2048 - 0
Trial: 2048 - 1
Trial: 2048 - 2
Trial: 2048 - 3
Trial: 2048 - 4
Trial: 4096 - 0
Trial: 4096 - 1
Trial: 4096 - 2
Trial: 4096 - 3
Trial: 4096 - 4
Trial: 8192 - 0
Trial: 8192 - 1
Trial: 8192 - 2
Trial: 8192 - 3
Trial: 8192 - 4
Trial: 16384 - 0
Trial: 16384 - 1
```

```
Trial: 16384 - 2
Trial: 16384 - 3
Trial: 16384 - 4
Trial: 32768 - 0
Trial: 32768 - 1
Trial: 32768 - 2
Trial: 32768 - 3
Trial: 32768 - 4
Trial: 65536 - 0
Trial: 65536 - 1
Trial: 65536 - 2
Trial: 65536 - 3
Trial: 65536 - 4
Trial: 131072 - 0
Trial: 131072 - 1
Trial: 131072 - 2
Trial: 131072 - 3
Trial: 131072 - 4
Trial: 262144 - 0
Trial: 262144 - 1
Trial: 262144 - 2
Trial: 262144 - 3
Trial: 262144 - 4
Trial: 524288 - 0
Trial: 524288 - 1
Trial: 524288 - 2
Trial: 524288 - 3
Trial: 524288 - 4
Trial: 1048576 - 0
Trial: 1048576 - 1
Trial: 1048576 - 2
Trial: 1048576 - 3
Trial: 1048576 - 4
Trial: 2097152 - 0
Trial: 2097152 - 1
Trial: 2097152 - 2
Trial: 2097152 - 3
Trial: 2097152 - 4
Trial: 4194304 - 0
Trial: 4194304 - 1
```

```
Trial: 4194304 - 2
Trial: 4194304 - 3
Trial: 4194304 - 4
Trial: 8388608 - 0
Trial: 8388608 - 1
Trial: 8388608 - 2
Trial: 8388608 - 3
Trial: 8388608 - 4
Trial: 16777216 - 0
Trial: 16777216 - 1
Trial: 16777216 - 2
Trial: 16777216 - 3
Trial: 16777216 - 4
Trial: 33554432 - 0
Trial: 33554432 - 1
Trial: 33554432 - 2
Trial: 33554432 - 3
Trial: 33554432 - 4
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

Name: Jon Abrahamson
ID: 107084898

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 10 **Spring 2019, CU-Boulder**



10^9
10^7
10^5
10^3
10^1
10^1
10^2
10^3
10^4
10^5
10^6
10^7