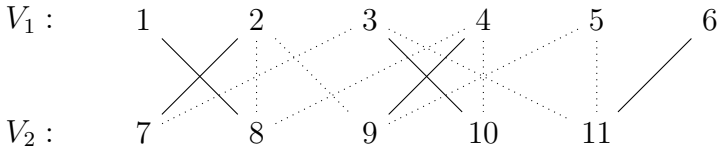
CSCI 3104 Spring 2018 Problem Set 10
Erika BAILON (09/28)

Problem 1

(15 pts total) 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.

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

By the definition we have at the beginning of the problem, each vertex touches at *most* one of the edges in the set. Having this property, we look at the size of each of the partitions of G. We have that $|V_1| = 6$ and $|V_2| = 5$. Therefore, we can guarantee that the maximum number matchings we can find in the graph G, is the value of the minimum size between these partitions of G because every other vertex in the other partition of G will not have a matching since by the definition of the graph, there cannot be more than one edge touching one vertex.

(b) *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 $G = (V_1, V_2; E)$:*

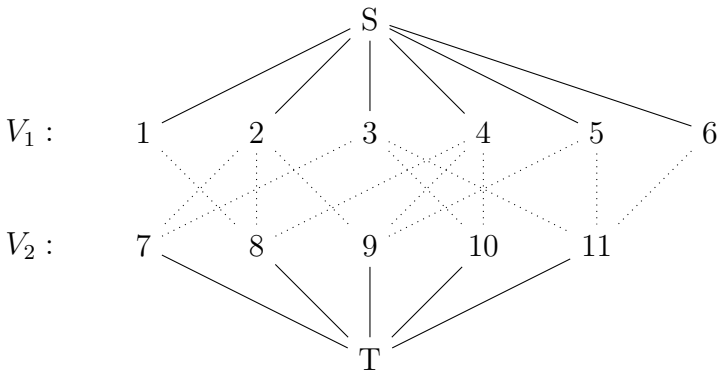
(1) *constructs a directed, weighted graph G' (which need not be bipartite) with weights $w : E(G') \to \mathbb{R}$ as well as two vertices $s; t \in V(G')$,*

(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.*

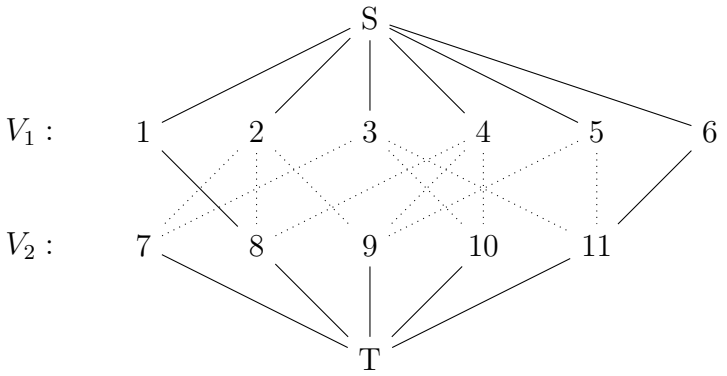
The algorithm would have the following steps:
1^{st} Assign one set to S (start) and the other set to T (end). Find the inflow or the outflow order for the vertices and edges.
2^{nd} Identify the set that has the minimum size, and set it as our start set.
3^{rd} Find the vertices with the lowest inflow on the other set.
4^{th} Connect these vertices to a vertex on the other set that has the same value of flow, otherwise, connect the vertex with a random one from the other set.
5^{th} Update the value of all the vertices and edges and go back to step 2.
6^{th} Stop connecting when you have gone through the entire set that you picked as the starting set (which is the set that had the min value).

(c) *Show the weighted graph constructed by your algorithm on the example bipartite graph above.*
First we assign each set of vertices to S and the other set to T and create a table to keep track of how many edges touch each vertex, their flow.



Vertex:	1	2	3	4	5	6	7	8	9	10	11
Edges (paths):	1	3	3	3	2	1	2	3	3	2	3

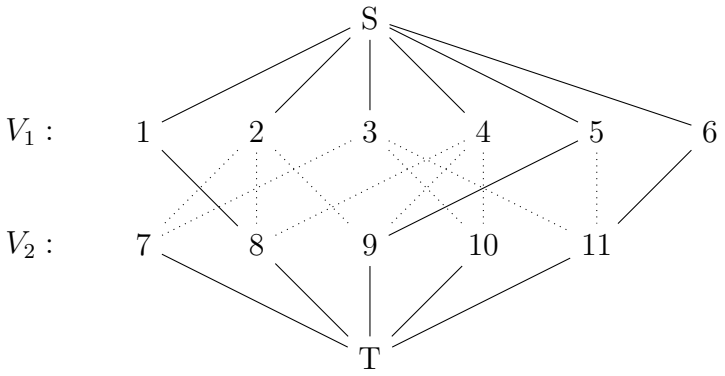
Then we look at the vertices with the smallest path size, in this case is 1 and 6. Then we connect them with the vertex on the other set, obtaining the following:



After we connect these vertices we then update the value of all the vertices and edges by reducing the number of paths each vertex has. Since we have connected 1 to 8, then the number of paths of 8 is going to be reduced to 0, same with 11 because it was connected with 6. In addition we update the number of paths that were connected to these vertexes we have connected. In this case, we update 5 and 3 because they had a path to 11 so we reduce the number of paths of 5 and 3 by one. We do the same with the paths that lead to 8, which were vertices 2 and 4 so we reduce those also by one. Then we go back to step 2 but we have the following updated table of values:

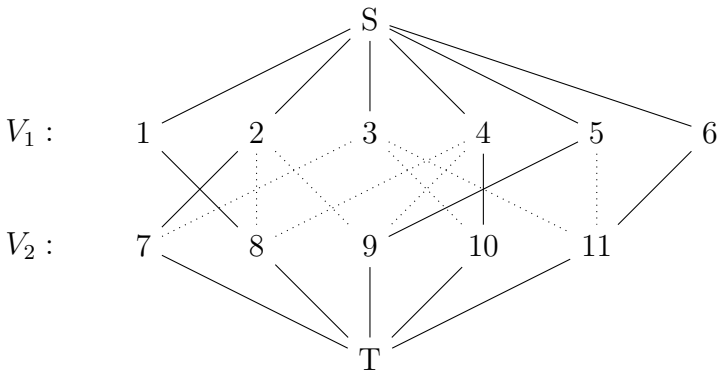
Vertex:	1	2	3	4	5	6	7	8	9	10	11
Edges(paths) :	0	2	2	2	1	0	2	0	3	2	0

As we see, now 5 is the vertex with the smallest number of paths so we connect it with the vertex on the other set and update the table of values again, obtaining the following:



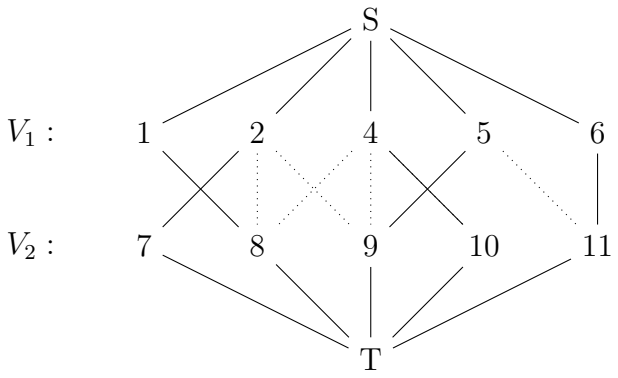
Vertex:	1	2	3	4	5	6	7	8	9	10	11
Edges (paths):	0	1	2	1	0	0	2	0	0	2	0

We go back to step 2 and check the vertices again obtaining the following:



Vertex:	1	2	3	4	5	6	7	8	9	10	11
Edges (paths) :	0	0	0	0	0	0	0	0	0	0	0

Now, when we check the table of values it seems like we finished. However, vertex 3 in the V_1 set, was never connected to anything. This is because as proven in (a), the maximum matching in a bipartite graph, is the size of the minimum size between the sets. In this case is 5. So, the valid graph we will get after running this algorithm is the following:



Problem 2

(20 pts total) 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) *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.*

By definition any function of order 1 has the following property:

$$\mathcal{P} \Rightarrow n^{\mathcal{O}(1)}$$

with a polynomial function the order would be $n^{\mathcal{O}(2)}$ $n^{\mathcal{O}(3)}$ $n^{\mathcal{O}(100)}$
we have a constant so they are the same asymptotically ,$n^{\mathcal{O}(1)}$. Therefore

$$|w| = n^0, n^1, \ldots \Longrightarrow \text{poly}\,(n) \Longrightarrow n^n \therefore \mathcal{P}$$

(b) *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.*

It can be solve by an exponential time algorithm because since the size of the witness is $O(\log n)$ and we know that 2^e is the subset of the edges, then we know that the exponential is in a polynomial form. Also, form part a we know that the solution contains $n^{(1)}$ $n^{(2)}$ ... $n^{(n)}$, then we can use this to know that for exponential we will have k^n, being the bite raised to the power that is the input. therefore it is $\mathcal{NP}$

(c) *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.*

As we have been seeing in class, P is like finding the solution while NP is just finding out if there is a solution. Therefore, NP can be solve by brute force because it will be like reducing the amount of solutions to a valid one.

Problem 3

(30 pts total) The Order of the Phoenix is trying to arrange to watch all the corridors in Hogwarts, to look out for any Death Eaters. Professor McGonagall has developed a new spell, Multi-Directional Sight, which allows a person to get a 360-degree view of where they are currently standing. Thus, if they are able to place a member of the Order at every intersection of hallways, they'll be able to monitor all hallways. In order not to spare any personnel, they want to place as few people as possible at intersections, while still being able to monitor every hallway. (And they really need to monitor every hallway, since Death Eaters could use Apparition to teleport into an arbitrary hallway in the middle of the school.) Call a subset S of intersections "safe," if, by placing a member of the Order at each intersection in S, every hallway is watched.

(a) *Formulate the above as an optimization problem on a graph. Argue that your formulation is an accurate reflection of the problem. In your formulation, show that the following problem is in NP: Given a graph G and an integer k, decide whether there a safe subset of size $\leq k$.*

I don't know

(b) *Consider the following greedy algorithm to find a safe subset:*

```
S = empty
mark all hallways unwatched
while there is an unwatched hallway
    pick any unwatched hallway; let u,v be its endpoints
    add u to S
    for all hallways h with u as one of its endpoints
        mark h watched
    end
end
```

Although this algorithm need not find the minimum number of people needed to cover all hallways, prove that it always outputs a safe set, and prove that it always runs in polynomial time.

I don't know

(c) *Note that, in order to be polynomial-time, an algorithm for this problem cannot simply try all possible subsets of intersections. Prove why not.*

I don't know

(d) *Give an example where the algorithm from (b) outputs a safe set that is strictly larger than the smallest one. In other words, give a graph G, give a list of vertices*

in the order in which they are picked by the algorithm, and a safe set in G which is strictly smaller than the safe set output by the algorithm.

I don't know

(e) *Consider the following algorithm:*

```
S = empty
mark all hallways unwatched
while there is an unwatched hallway
    pick any unwatched hallway; let u,v be its endpoints
    add u,v to S
    for all hallways h with u or v one of their endpoints
        mark h watched
    end
end
```

Although this algorithm need not find the minimum number of people needed to cover all hallways, prove that it always outputs a safe set, and prove that it always runs in polynomial time.

I don't know

(f) *In any safe set of intersections, each hallway is watched by at least one member of the Order. Use this to show that the algorithm from (e) always outputs a safe set whose size is no more than twice the size of the smallest safe set. Note: you don't need to know what the smallest safe set is to prove this! All you need is the fact stated here.*
This is called a "2-approximation algorithm," because it is guaranteed to output a solution that is no worse than a factor of 2 times an optimal solution.

I don't know

(g) *Does the algorithm from (b) always produce a safe set no bigger than that produced by the algorithm in (e)? If so, give a proof; if not, give a counterexample. A counterexample here consists of a graph, and for each algorithm, the list of vertices it chooses in the order it chooses them, such that the safe set output by algorithm (b) is at least as large as the safe set output by algorithm (e). If you are unable to give either a proof or a counterexample, then for partial credit give a plausible intuitive argument for your answer.*

I don't know

(h) *Compare the greedy algorithm from (e) with the greedy algorithm from (b). Show which runs faster asymptotically? Which of these two algorithms would you rather use to solve the Order of the Phoenix's problem and why?*

I don't know

(i) *This problem is, in fact, NP-complete. Why does the 2-approximation polynomial-time algorithm from (e) not show that P=NP?*[1]

I don't know

[1]Interestingly, it is known that if there were a 1.3606...-approximation algorithm for this problem in polynomial time, then it would follow that P=NP, but that is a very nontrivial theorem. Under a standard complexity-theoretic assumption, even if there were a 1.99999-approximation algorithm in polynomial time, it would follow that P=NP, but this assumption remains a conjecture, and opinion in the research community is divided on whether this conjecture is true or false. We will provide references to these results after the problem set has been handed in.

Problem 4

(20 pts extra credit) Every young wizard learns the classic NP-complete problem of determining whether some unweighted, undirected graph $G = (V, E)$ contains a simple path of length at least k (where both G and k are part of the input to the problem), known as the Longest Path Problem. Recall that a simple path is a path $(v_1, v_2, ..., v_\ell)$ where each (v_i, v_{i+1}) in the path is an edge, and all the v_i are distinct; its length is $\ell - 1$ (=the number of edges in the path).

(a) *Ginny Weasley is working on a particularly tricky instance of this problem for her Witchcraft and Algorithms class, and she believes she has written down a "witness" for a particular input (G, k) in the form of a path P on its vertices. Explain how she should verify in polynomial time whether P is or is not simple path of length k. (And hence, demonstrate that the problem of Longest Path is in the complexity class NP.)*

I don't know

(b) *For the final exam in Ginny's class, each student must visit the Oracle's Well in the Forbidden Forest. For every bronze Knut a young wizard tosses into the Well, the Oracle will give a yes or no response as to whether, given an arbitrary graph G and an integer k, G contains a simple path of length k. Ginny is given an arbitrary graph G and must find the longest simple path in G. First, she realizes it would be useful to determine the length of the longest simple path. Describe an algorithm that will allow Ginny to use the Oracle to find the length of the longest simple path in G by asking it a series of questions, each involving a modified version of the original graph G and a number k. Her solution must not cost more Knuts than a number that grows polynomially as a function of the number of vertices in G. (Hence, prove that if we can solve the Longest Path decision problem in polynomial time, we can solve its optimization problem as well.)*

I don't know

(c) *Next, once she knows the length ℓ of the longest simple path in G, Ginny must use the Oracle to actually find a path of length ℓ. Describe an algorithm that will allow Ginny to use the Oracle to find the longest simple path in G by asking it a series of questions, each involving a modified version of the original graph G and a number k of her choosing (for each question she can ask about a different graph G and a different number k). Her solution must not cost more Knuts than a number that grows polynomially as a function of the number of vertices in G. (Hence, prove that if we can solve the Longest Path decision problem in polynomial time, we can solve its search problem as well.)*

I don't know

Problem 5

(20 pts extra credit) Recall that the `MergeSort` *algorithm (Chapter 2.3 of CLRS) 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; see the end of Chapter 5.3.).*

(a) *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.*

I don't know

(b) *For each of* $n = \{2^4, 2^5, ..., 2^{26}, 2^{27}\}$*, run* `MergeSort(randomArray(n),n)` *five 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* $T(n) = An \lg n$*, where you choose the constant* A *so that the function is close to your data.*
Hint: To increase the aesthetics, use a log-log plot.

I don't know

WORKED WITH: Jason Lubrano and Aaron during office hours.