Assignment 8

CSCI 270

Paul Kim | pbkim@usc.edu | 1723717002

Question 1

(1) [10 points]
A team of archaeologists is examining fossils at an excavation site. They have discovered remnants of n different species (the set of all species is V), and they believe that these species lived there in two distinct periods. They have been able to use various technologies (such as carbon dating) to pin down the time for *some*, but not all of the species. This partitions the species into three sets: T_1 (who lived in time period 1), T_2 (who lived in time period 2), and U (unknown when they lived). Each species belongs to exactly one of the three sets T_1, T_2, U.

They would now like to label the species in U with which period they likely lived in, although of course, they might not be completely certain. In order to do so, they want to use some uncertain information about which species may have prayed on each other. They inferred this from proximity of bones, some genetic analysis of remnants on teeth, shapes of incisions in bones vs. teeth, etc. The specific techniques are beyond this problem, and of course not relevant. As far as you are concerned, all the relevant information is summarized in some directed edges E on the species: for each edge $e = (u, v)$, they give you a probability $p_e \in [0, 1]$ that species u prayed on v (and hence must have lived at the same time as v).[1] Importantly, they assume that each of their guesses about who prayed on whom is correct *independently*.[2]

They are asking you to use your comprehensive algorithm design skills to build an algorithm that will label each of the unknown species $u \in U$ with whether they lived in period 1 or period 2. Your assignment should maximize the likelihood of the inferred guesses. More specifically, suppose that you partition the species into S_1, S_2, with $T_1 \subseteq S_1$ and $T_2 \subseteq S_2$. For every edge e that goes from S_1 to S_2 or S_2 to S_1, you are basically telling the researchers that they were wrong, which happens with probability $1 - p_e$ (because they believed that their estimate was correct with probability p_e). Therefore, the overall likelihood of your estimate would be $\prod_{e=(u,v):u,v \text{ in different sets}}(1 - p_e)$, so your goal is to partition the species into S_1, S_2 so as to maximize this quantity.

Give (and analyze) a polynomial-time algorithm for solving this problem. If there are multiple equally good assignments[3], you can output any of them.

[Hint: Try to remember how to connect products and sums with each other.]

Answer: *Went to Office Hours religiously.

*Similar to Image Segmentation where we try to partition background and foreground nodes, but instead here, we want to use the idea of a Min-Cut to partition the nodes into two sets.

*The overall likelihood estimate equation is important because it precisely defines the quantity we want maximize.

Problem: Assign all species in U into S1 and S2 (and keep T1 in S1 and T2 in S2) so as to maximize the product given by the formula.

Main Idea: We want to maximize some product given by the formula, by minimizing a sum of some edges. We have a probability on each edge, these edges denoting the probability/likelihood that a species preyed on another species (if it did, then the species were in the same time period). We run Edmond's Karp, and prove that whatever Edmond Karp is maximizing is the thing we are looking for, by taking the Min-Cut.

*Find a way to turn the probability of one species preying on the other, to something that you want to minimize, and this is something that the Min-Cut can help us with. Basically, we have a bunch of edges and we want to use the min cut to minimize the probability.

*Turning Product Into Sums: We can use/apply the Law of Logarithms to convert from sums to products. The Law states that **Log(M*N) = Log(M) + Log(N).**

Goal: We want maximize the quantity given by the formula and the key is to turn the maximization into minimization by using EK and finding the min-cut. The min-cut minimizes the sum of something and that something, will be needed to turn the goal of maximizing the product, into a minimization of sums. We want to partition the graph into two sets S1 and S2 so as to maximize the product. Maximizing the product, is equivalent to minimizing the cut. To convert the product into a sum, we can take the log of it. Since we are trying to maximize the product, we can apply log to it, and it will still conserve the value (still remains maximized). After we take the log of the product, we can convert the edges into a sum of Log(1-P_e) for all edges (and we want to maximize this value). The Min-Cut gets us the minimum value and a partition of two sets. We have a set of edges that cross this partition, and the Min-Cut gives us the sum of those edges such that it is minimal across all edges (Min-Cut is a minimum sum).

*If we convert the product into a sum by taking the log of it, we see that the edges yield a negative value. Multiplying "-1" turns the sum of Log(1-P_e) back into a positive value, and it now turns a maximization problem into a minimization problem.

*We want our edges to have costs (-1*Log(1-P_e)).

*Because the probability of the events are independent, we multiply all of the (1-P_e) values to get the overall probability of the edges involved in the cut. By maximizing this value, we find the most probable solution.

Intuition: We are given 3 sets as current partitions; T1 denotes the set of species from time period 1, T2 denotes the set of species from time period 2, and U is the unknown set of species that we attempt to work with. We want to form bigger sets S1, and S2, which contain T1, T2, as well as the species that were assigned from U to their respective time periods.

Formal Algorithm:

*inf = Infinite.

Build a Flow Network G on V + (S,T)

Add a new source node S, and assign edges of value inf to all the nodes in T1(S1) from S.

Add a new sink node T, and assign edges of value inf to all the nodes in T2(S2) to T.

For the remaining U nodes with unassigned edges, we connect an edge from the Source and Sink to these nodes with values of ”-1*Log(1P$_e$)”. (We don’t want negative numbers so we multiply with -1).

Call Edmond’s-Karp to find the minimum S-T cut (S+(S), S’+(T)) that minimizes the value assigned to these edges.

Return the product of the cut (should be maximized) and it should be optimal partition.

Runtime: Runtime of Edmond’s Karp O(M^2*N) where M is the number of edges, and N is the number of vertices, and trivial pre/post-processing.

Proof of Correctness:

Prove that Edmond’s Karp minimizes indeed what you want to minimize

We can assume that the Law of Logarithms holds true, and it successfully transforms products into sums.

We know that Edmond’s Karp minimizes the total capacity and captures the Min-Cut of the edges that we assigned values too. (We did this in lecture)

Edmond’s Karp is taking the Min-Cut of the edges that we assigned, to minimize the sum as to maximize the product given by the equation. The edges that are cut has the value ”(-1*Log(1P$_e$))”, which is intuitively the probability that two species do not live together and obviously this is what we are trying to minimize. We multiply a negative *1 because if we take the log of a number between 0 to 1, we get a negative value, and so by multiplying a negative number, we can convert the sum into a positive one. Another intuitive aspect about our graph is that we assign edges of inf from S to the nodes in T1 and edges of inf from T2 to T, because we do not want to cut these edges as it was already determined that they are in the set. We have three given sets (T1, T2, and U) and we want to form two big sets S1 (that contains T1) and S2 (that contains T2) from the unknown nodes from U.

So the total capacity/maximized product of the cut is (S+(S), S’+(T)) and this is exactly what we are trying to find.

Because Edmond’s-Karp minimizes the total capacity, it minimizes this summation, and maximizes the product formula.

Therefore, this algorithm finds the optimal partition that separates the set of species into two sets S1 and S2.

Proof Completed.

Question 2

(2) [3+6+6=15 points]
We motivated the Maximum Bipartite Matching problem with a number of applications, including assigning jobs to machines, so as to maximize the total number of jobs that are assigned to a machine. In this problem, we look at a more general version: jobs come with rewards (think amount of money that the owner of the job is willing to pay you), and you want to maximize the total rewards. More specifically, you have a set J of n jobs, and a set M of m machines. Each job j has a reward $r_j \geq 0$; to avoid tie breaking, we assume that all r_j are distinct. In addition, each job comes with a set $S_j \subseteq M$ of machines which would be able to perform job j. Your goal is to find a bipartite matching between jobs and machines (so each machine gets at most one job) that maximizes the total reward $\sum_{j \in T} r_j$ of the set T of jobs that you match.

There is a pretty obvious greedy algorithm to try here. To describe it precisely, we say that a set T of jobs is *matchable* if there is a matching between T and M in which all jobs in T are matched. The

algorithm is then the following: First, sort the jobs by decreasing reward, so $r_1 > r_2 > \cdots > r_n$. Start with the set $T_0 = \emptyset$ of no matched jobs. In each iteration j, try to add job j to the set T_{j-1}; that is, check if $T_{j-1} \cup \{j\}$ is matchable. If so, permanently add j, i.e., set $T_j = T_{j-1} \cup \{j\}$. Otherwise, permanently discard j. In the end, return the final set T.

This algorithm is greedy, so we might be pretty suspicious of it. Amazingly, however, it actually finds the maximum-reward set T. You will prove this. (We won't bother with running time analysis here, as it's pretty straightforward.)

(a) In each iteration j, we need to check if $T_{j-1} \cup \{j\}$ is matchable. Give (and briefly analyze) a polynomial-time algorithm to do this.
[Note: this should be quite straightforward for you at this point.]

(b) Prove the following two key properties:

Lemma 0.1. *(a) Let T, T' be two matchable sets of jobs with $|T| > |T'|$. Then, there exists at least one job $j \in T \setminus T'$ such that $T' \cup \{j\}$ is matchable.*
(b) Let $T \neq T'$ be two different matchable sets of jobs with $|T| = |T'|$. Then, for every job $j' \in T' \setminus T$, there exists a job $j \in T \setminus T'$ such that $T \cup \{j'\} \setminus \{j\}$ is matchable.

[Hint: We highly recommend thinking of what Ford-Fulkerson will do specifically on the types of s-t flow networks that are obtained by the Maximum Bipartite Matching reduction.]

(c) Prove that the Greedy algorithm we gave finds a maximum-reward matching.
[Hint: The lemma you proved will likely be very helpful, and you may want to revisit what you learned earlier about analyzing greedy algorithms. Even if you didn't manage to prove the lemma, you can still use it to solve this part.]

Answer:

*Maximum Bipartite Matching (Similar). Jobs come with rewards and we want to maximize the total reward. Each job comes with a set of machines that would be able to perform job(j).

*Set J of N jobs, and set M of M Machines. Each job(j) has a reward greater than or equal to 0, and we can assume that all values are distinct.

Goal: Find a bipartite matching between jobs and machines (each machine gets at most one job) that maximizes the total reward of the set T of jobs that you can match.

Algorithm Suggested:

Sort the jobs by decreasing reward.

Start with the empty set of no matched jobs.

In each iteration j, attempt to add job(j) to the set T; that is, check if T and J are matchable. If so, permanently add j to the set. Otherwise, permanently discard j.

Return the final set T.

(It actually finds the maximum-reward set T).

1. **Question: How do we check if in each iteration, that the set T is matchable with all jobs.**

   Intuitively, a job and a machine are matchable if a machine is able to perform job j. As a result, to check if $T_(j-1)$ U (j) is matchable, there must be some sort of edge from the machine to the job, and intuitively, we can model this edge as a flow path.

   *We are trying to determine if there can be a match. We can use an algorithm similar to the one used in a maximum bipartite matching problem, which would be Edmond's Karp (Ford Fulkerson works too).

   Matching: A set of edges such that each vertex is incident on at most one edge.

   A (strongly)-polynomial time algorithm that allows us to check if these flow paths exist (connections) could be running Edmond's-Karp that attempts to maximize the flow network from a source to a sink and the nodes in between could be the nodes that represent jobs and the nodes that represent machines, and this could be a potential algorithm that checks if j and the machine are matchable, as the flow from a job to a machine would be modeled as so that such a connection exists. If such a flow path exists, that captures the max flow of the max number of jobs, such that there is a matching between T and M in which all jobs in T are matched, then in each iteration, the set T is matchable.

   Formal Algorithm: Edmond's Karp (Model a General Bipartite Graph Matching Algorithm)

   Construct the original bipartite graph where the left nodes are the J set of Jobs, and the right nodes are the M set of Machines.

   Add a new source S and a new sink T.

   Add a directed edge of capacity 1 from S to each node on the left. (Jobs are unique)

   Add a directed edge of capacity 1 from each node on the right to T. (Machines can only do 1 job)

   Direct all edges of the original bipartite graph from the left to the right in which edges between the two exist only if the machine can complete job J. The problem gives us information about the machines and what jobs they can handle. If the job can run on a machine, then there exist an edge between them. We can give these edges a flow capacity of inf.

   Compute a maximum integer S-T flow F using Edmond's Karp.

   Return Max Flow.

   Runtime: Runtime of Edmond's Karp O(M^2*N) where M is the number of edges, and N is the number of vertices, and trivial pre/post-processing (if there is any). By using Edmond's Karp over Ford Fulkersons, we can achieve a strongly polynomial time.

Using the Proofs we did in class (March 23 Kempe Lecture), we know that this general algorithm returns a matching M and M has a maximum size. We also know that by using Edmond's Karp, we achieve the Max Flow.

Derive to the determination if its matchable or not. And how we can determine something from the result.

Direct Proof:

Matchable means that all the jobs are matched to some machine. By using the algorithm shown we can use the very basic intuition that we are going to given a set of J jobs that have an X number of vertices/nodes of jobs in the left column of our Bipartite Graph. In our algorithm, we directed an edge from the source to each of these nodes on the left (jobs) with a flow capacity of 1. Building our intuition, if we are to claim that every job is matchable, that must mean that all the flow value from S successfully transferred from each of the left nodes, to the sink. Therefore, if all jobs were successfully matched to a machine, it's flow must have passed through the machine to the sink, and we can see that every job is matched if the algorithm returns the Maximum Flow value of all jobs.

To calculate the maximum possible flow value emitted, we can just count the number of jobs that exist in the left column of nodes or set J, and that is how much edges direct out of S (with each flow value of 1), into J jobs. If we are returned the same value of J from our algorithm's output, then we have achieved the maximum flow, and therefore, set T is matchable with all jobs. That all jobs are matched to some machine.

To summarize: We do a bipartite matching. We do a maximum matching and if we find out that if there is a matching max flow value that is the same as the number of jobs, then all jobs were matched.

*In our algorithm, we assign the edge from the nodes in the right column to the sink, with a capacity of one, because it is true that a machine can only do at most 1 job at a time.

2. **Question: Prove the following two key properties. (Gives us two key properties used for Part C of 2)**

   *Office Hours (Yuxing and Adam Aid)

   *Doing Part B first will help us to do Part A.

   Definitions: T and T' are matchable sets of jobs. J is a job.

   Part B:

   Let T and T' be two different matchable sets of jobs with the cardinality being the same. Then for every job in T' not in T, there exists a job in T not in T', such that T is matchable with the job in T' removing its own job.

   Basically the idea here is that since T and T' are two different matchable sets of jobs, then we can see that there are two jobs J and J'. Since the cardinality of both sets are the same, we know that J' will always exist since T is not equal to T'. Suppose that we have some J' in T' that is matchable, that means that J' also has some machine M'. Since we agreed that J' exists and that we know that T' is matchable so that this J' was matched with some machine M', we want to add this job to make it matchable.

Direct Proof:

As the question suggests, we want to add J' to T and keep it matchable:

Two Cases:

1.) The machine M' is not used by any job in T, so this implies that we can just add J' to T to ensure that the job has a machine, and we can therefore, take out the previous job J without any consequences. (Good)

2.) The machine M' is used by a job in T (Tricky Part). Assume that the job in T that uses M' is called J. We know that J is not equal to J', because if they were equal then nothing would change. However, J is in T' and not in T, so they can never be equal. From here, we can just take out the job and give it another job J". (High Level) (Good)

PART A:

Let T, T' be two matchable sets of jobs such that the cardinality of T > T'. Then there exists as least one job J in T not in T' such that T' U (J) is matchable.

To formally prove this statement, we must agree that there is some job J in T and not in T'. This must be true since the set T is greater than T'.

Let us suppose that we have some job J (as we proved above) that is in T and not in T'. Job J was matched to machine M.

Direct Proof:

Let us see what happens if we add job J to T'.

1.) Machine M is initially free, and by simply adding J, we can ensure it is matchable. (Good)

2.) Machine M is initially not free, meaning that M is matched to some job J' in T' and not in T. *From here, we can apply what we derived from part B.

a.) One note is that there is another job J" in T not in T' (Since |T| > |T'|) and J is not J". (This job J" must exist.)

b.) Another note is that there are two jobs J and J" that are in T and not in T'.

c.) Now since we discovered the key facts above, instead of using job J and adding to T', we can use job J" to match into the set T', since ((Since |T| > |T'|)), and we see that this is matchable with some machine M". We can swap that job out and it will still be matchable. (Good)

3. **Question: Prove that the Greedy Algorithm we gave finds a maximum-reward matching. (Exchange Argument)**

*Understand how we can use the lemmas we proved above for this problem. (Lemma A gives intuition, Lemma B helps with optimality (Exchange Argument)).

*Hint: Lemmas are likely helpful, and we want to revisit what we learned earlier about analyzing the greedy algorithm.

Greedy Algorithm Suggested:

Sort the jobs by decreasing reward.

Start with the empty set of no matched jobs.

In each iteration j, attempt to add job(j) to the set T; that is, check if T and J are matchable. If so, permanently add j to the set. Otherwise, permanently discard j.

Return the final set T.

We will assume that the algorithm actually finds the maximum reward set T. (Prompt)

Proof of Correctness of Greedy Algorithm: (Exchange Argument)

Intuition using Lemma A:

Lemma A tells us that if T, T' are two matchable sets of jobs such that the cardinality of T > T'. Then there exists as least one job J in T not in T' such that T' U (J) is matchable. We can use this idea by thinking about constructing larger and larger sets of machines/jobs. If we find that the optimal solution is bigger, then we can use Lemma A to see that there is at least a job in T that is matchable with T'. By performing swaps, once the size are equal, if they are different, then we can perform a series of swaps to maintain match-ability; something similar to an exchange argument. We know that our solution cannot be bigger than the optimal, which gives us a contradiction.

Proof: Exchange Argument (Lemma B helps with)

Intuition using Lemma B:

Lemma B tells us that if T and T' be two different matchable sets of jobs with the cardinality being the same. Then for every job in T' not in T, there exists a job in T not in T', such that T is matchable with the job in T' removing its own job. We used this idea to help enforce Lemma A. This tells us a key property that can be used in an exchange argument, that we can swaps jobs between two sets (OPT and Greedy). We can use this property to let us do swaps and to achieve optimality.

Formal Proof:

Let the set of T (T1,T2,T3...) be the greedy solution that our algorithm returns of the maximum reward set.

Let the set H (H1, H2, H3...) of matched jobs be our OPT solution.

Let Tau be an operation (valid through Lemma B) that allows/makes swaps to turn the OPT solution into our greedy solution.

By applying some operation Tau, we can tell that the solution doesn't get more worse because Tau(H) has the same matching of jobs as set H. Likewise this solution is still valid because we shown/proved in Lemma B and A that there exists at least a job in T that is matchable with a job from another set, and this allows us the possibility to make these swaps. (Allows us to use this exchange argument to prove optimality).

Here we defined a transformation that made OPT closer to the Greedy after applying some Tau operation a bunch of times, and showed that OPT is still optimal, so that our greedy is optimal as well.

Exchange Argument shows that the Greedy is as Good as the OPT. (Greedy Solution we determined is best). Proof Completed.