CSCI 5454 Problem Set 2

Robert Werthman

---

1. *Probability boot camp*

   (a) *Prove Markov's inequality, $Pr[X \geq c] \leq E[X]/c$, with $c > 0$*

   The formula for the probability of a continuous random variable X with probability density function $f(x)$ is

$$Pr[x_1 \leq X \leq x_2] = \int_{x_1}^{x_2} f(x)dx$$

   And the formula for the expected value of a continuous random variable X with probability density function $f(x)$ is

$$E[x_1 \leq X \leq x_2] = \int_{x_1}^{x_2} xf(x)$$

   so for $Pr[X \geq c]$ we have

$$Pr[X \geq c] = \int_{c}^{\infty} f(x)dx$$

   and since X is a nonnegative random variable we have

$$E[X] = \int_{0}^{\infty} xf(x)dx$$

   Notice that $0 < c \leq \infty$. This tells us that the bounds of $E[X]$ are greater than $Pr[X \geq c]$. We can break up the integral formed by $E[X]$ to create an inequality that will begin to look similar to the integral of $Pr[X \geq c]$.

$$\begin{aligned} E[X] &= \int_{0}^{\infty} xf(x)dx \\ &= \int_{0}^{c} xf(x)dx + \int_{c}^{\infty} xf(x)dx \\ &\geq \int_{c}^{\infty} xf(x)dx \end{aligned}$$

We can assume that $x \geq c$ because c is one of the bounds of the integral. This means we can substitute c for x.

$$\int_c^{\infty} xf(x)dx \geq \int_c^{\infty} cf(x)dx \geq c\int_c^{\infty} f(x)dx$$

We now have an equation for $E[X]$ that has $Pr[X \geq c]$.

$$E[X] \geq c\int_c^{\infty} f(x)dx = cPr[X \geq c]$$

Dividing both sides by c gives us

$$E[X]/c \geq Pr[X \geq c]$$

which is Markov's inequality. We have just shown that $Pr[X \geq c] \leq E[X]/c$, with $c > 0$ is true based on the probability and expected value of the continuous random variable X.

(b) *Prove Chebyshev's inequality* $Pr[|X-\mu| \geq c \cdot \sigma] \leq 1/c^2$

If we let $k = (c \cdot \sigma)$ we have

$$Pr[|X-\mu| \geq c \cdot \sigma] = Pr[|X-\mu| \geq k]$$

One of the properties of $|a|$ is that it can also be represented as $\sqrt{a^2}$ so we can change

$$Pr[|X-\mu| \geq k] = Pr[\sqrt{(X-\mu)^2} \geq k]$$

If we take the square root of both sides of the inequality we get

$$Pr[(X-\mu)^2 \geq k^2]$$

From here we can use Markov's inequality since we know $(X-\mu)^2$ is nonnegative. If we let $(X-\mu)^2 = X$ and $k^2 = c$ and substitute those into Markov's inequality we have

$$Pr[(X-\mu)^2 \geq k^2] \leq E[(X-\mu)^2]/k$$

The variance σ^2 of a continuous random variable X with mean μ is

$$\sigma^2 = E[(X-\mu)^2]$$

So

$$Pr[(X-\mu)^2 \geq k^2] \leq E[(X-\mu)^2]/k$$

Now becomes

$$Pr[(X-\mu)^2 \geq k^2] \leq \sigma^2/k$$

If we now change k back to $c \cdot \sigma$ we have the equation

$$Pr[(X-\mu)^2 \geq (c \cdot \sigma)^2] \leq \sigma^2/c^2 \cdot \sigma^2$$

Reducing the right side of the equation results in

$$Pr[(X-\mu)^2 \geq (c \cdot \sigma)^2] \leq 1/c^2$$

And taking the square root of both sides of the inequality on the left hand side of the equation results in

$$Pr[\sqrt{(X-\mu)^2} \geq c \cdot \sigma] \leq 1/c^2$$

Which can be further reduced to

$$Pr[|X-\mu| \geq c \cdot \sigma] \leq 1/c^2$$

Which is Chebyshev's inequality. We have just shown that Chebyshev's inequality can be proven using Markov's Inequality.

(c) *Show that for any discrete random variables* $X, X', E[X] = E[E[X|X']]$.

If we let $Y = X'$ we have

$$E[X] = E[E[X|Y]]$$

The expected value of a discrete random variable X is

$$E[X] = \sum_x x \cdot Pr[X = x]$$

The conditional probability for any two discrete random variables X, Y is defined to be

$$Pr[X = x|Y = y] = \frac{Pr[X = x \cap Y = y]}{Pr[Y = y]}$$

The conditional expectation for any two discrete random variable X, Y is defined to be

$$E[X|Y = y] = \sum_x x \cdot Pr[X = x|Y = y]$$

Given the above assertions

$$\begin{aligned}
E[E[X|Y]] &= E[\sum_x x \cdot Pr[X = x|Y]] \\
&= \sum_y [\sum_x x \cdot Pr[X = x|Y = y]] \cdot Pr[Y = y] \\
&= \sum_x x \sum_y Pr[X = x|Y = y] \cdot Pr[Y = y] \\
&= \sum_x x \cdot Pr[X = x] \\
&= E[X]
\end{aligned}$$

Sources I used to complete this problem:

- http://www.maths.qmul.ac.uk/ pettit/MTH5122/notes15.pdf
- https://en.wikipedia.org/wiki/Expected_value
- https://math.stackexchange.com/questions/1353418/expected-value-proof-law-of-total-expectation

(d) *Prove by induction that* $E[X_t] = 0$ *for a martingale.*

We want to show that in a martingale sequence the expected value for a random variable X_{t+1} is the random variable X_t before it.

Base case:

When $t = 0$ with $X_0 = 0$ we have

$$\begin{aligned} E[X_1|X_0] &= \sum_x x \cdot Pr[X_1 = x|X_0 = 0] \\ &= 0 \end{aligned}$$

Inductive step:

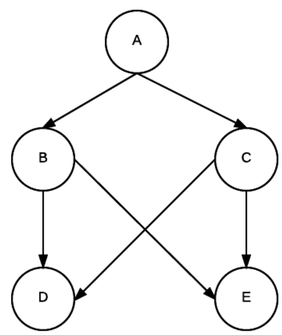
We showed that the base case $E[X_1|X_0] = 0$ is true. Therefore, we can say that $E[X_2|X_1] = E[E[X_1|X0]] = 0$ no matter what X_2 is. This leads to the equation

$$E[X_t|X_0, ..., X_{t-1}] = E[E[...E[X_1|X_0]]] = 0$$

So by induction we have just shown that $E[X_t] = 0$.

2. *Give a graph G with a source vertex s and a set of edges E_π to every vertex in G such that the path from s to v is the shortest path but the set of the edges E_π cannot be produced by running a breadth-first search on G.*

Figure 1: Example of a graph where there exists a set of edges that are the shortest path between vertices but will not be found by a breadth first search.



For this question, the shortest-path from s to v is defined as the minimum number of edges in any path from vertex s to vertex v. If we let the source vertex s be vertex A then the set of edges E_π that cannot be produced by running a breadth first search are

$$\{(A, B), (A, C), (B, D), (C, E)\}$$

This is the case because of how breadth first search discovers vertices. It starts at a source vertex s and then adds every vertex in the adjacency list of s to a queue. In the case above s will be A and it will add B and C to the queue because those are the vertices adjacent to A. An edge from A to B and from A to C will be created. Once it does that it will then pop off the first vertex on the queue which will be either B or C depending on the ordering of the adjacency list of A. It will then add every vertex from the adjacency list of the vertex is pops off to the queue. If either B or C is removed from the queue, D and E are still both added to the queue because there are both in B and C adjacency lists. In either case, edges are created from a single vertex, C or B, to both D and E i.e. $(C, D), (C, E)$. It will never be the case that both C and B will have a only single edge i.e. $(C, D), (B, E)$. It will only be the case that one of them will have two edges so the set of edges E_π above will would never occur with a breadth first search.

3. *Give an efficient algorithm with an input of a graph G and and an edge e and returns a cut where e is a light edge across the cut or NO if no such cut exists*

Psuedocode

```
def LightEdgeCut(G,e):
    minimum spanning tree = Kruskal's algorithm
    if e is in minimum spanning tree:
        create a cut of the graph with e as a light edge
        return cut
    else:
        add e to minimum spanning tree to create a cycle C
        z = largest edge removed from cycle C
        if z is e: // Cycle property of minimum spanning tree
            return NO
        else:
            create a cut of the graph with e as a light edge
            return cut
```

Correctness

The algorithm above depends on two assumptions:

(a) If an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph [1].

(b) For any cycle C in the graph, if the weight of an edge e of C is larger than the individual weights of all other edges of C, then this edge cannot belong to a MST [2].

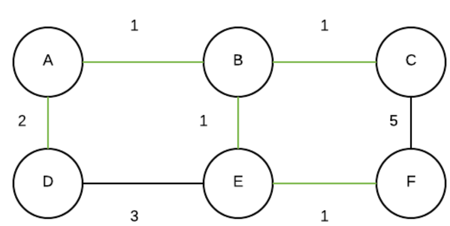
As an example for hypothesis (a) we cut the the graph of Figure 2 between vertices A and D and vertices D and E. After the cut $S = \{D\}$ and $S' = \{A, B, C, E, F\}$. The edge (A, D) is in the minimum spanning tree created by Kruskal's algorithm and it is a light edge of the cut that was made.

As an example for hypothesis (b) if we add the edge (D, E) to the minimum spanning tree it will create a cycle from vertices $A \rightarrow B \rightarrow E \rightarrow D \rightarrow A$. If we remove the largest weighted edge which in this case we say is (A, D) even though it is the same weight as (D, E) we get a new minimum spanning tree that has (A, D) as an edge. If (A, D) had been larger than (D, E) then it would have been removed from the minimum spanning tree and would not be part of any other minimum spanning tree.

We assume Kruskal's algorithm is correct and returns a minimum spanning tree for the graph G. It may be that the edge e LightEdgeCut takes as input is part of the minimum spanning tree returned by Kruskal's algorithm. We then know that the edge e is a light edge crossing some cut of G so we can create that cut with e and return it.

If e is not in the minimum spanning tree returned by Kruskal's algorithm then we have to check to see if e is part of any minimum spanning tree of G. We do this by adding it to the minimum spanning tree returned by Kruskal's algorithm in order to create a cycle. Based on the cycle property of minimum spanning trees the largest edge in a cycle cannot be part of any minimum spanning tree. If e is the largest edge in the cycle we know it is not part of a minimum spanning tree and therefore not a light edge crossing some cut of the graph. If it is not the largest edge then we can conclude it is part of a minimum spanning tree and is a light edge crossing some cut of the graph.

Figure 2: A graph where the edges of a minimum spanning tree created by Kruskal's algorithm are highlighted in green.



Runtime

Kruskal's algorithm runs in $O(V\,log(E))$. The worse case of this algorithm will, after going through Kruskal's algorithm, go to the else statement. Assuming the minimum spanning tree is a disjoint-set data structure adding the edge e to the spanning tree is a set union operation that takes $O(1)$. Finding the edge with the highest weight in the cycle could take $O(V-1)$ if the cycle contains all of the edges in the minimum spanning tree. Checking if the largest edge is e takes $O(1)$. Creating a cut of the graph G with e as a light edge takes O(V) if it has to look at all of there vertices where to make the cut. The worst case runtime would be $O(V\ log(E)+V+(V-1)) = O(V\ log(E)+V)$.

Sources I used to complete this problem.

1 https://en.wikipedia.org/wiki/Minimum_spanning_tree

2 http://test.scripts.psu.edu/users/d/j/djh300/cmpsc465/notes-4985903869437/solutions-to-some-homework-exercises-as-shared-with-students/4-solutions-clrs-23.pdf

4. *Give a $o(|V|^2)$ algorithm to find a universal sink given an adjacency matrix.*

Pseudocode

```
FindUniversalSink(A):
 for k in range(1,|V|):
  vertex k is a universal sink
  for i in range(1,|V|):
   if i != k and A_{ik} == 0: // 0 in the vertex's column that is not a self loop
    vertex k not universal sink
    break
  for j in range(1,|V|):
   if A_{kj} == 1: // 1 in the vertex's row
    vertex k not universal sink
    break
 if vertex k is still a universal sink:
  return vertex k
 elif k == |V|:
  return NO
```

Correctness

In order for this algorithm to be correct I make the following assumptions:

(a) If a graph is represented as an adjacency matrix A then A_{ij} is the edge from vertex i to vertex j.

(b) If a 1 is present in the matrix at row i column j then an edge exists from i to j. If a 0 is present then no edge exists from i to j.

(c) A universal sink will have 0's in its entire row of the adjacency matrix meaning there are no edges from it to any other vertex or to itself in the form of a self-loop.

(d) Every vertex, not including the universal sink, will have a 1 in the column the belongs to the universal sink. This means that they all have edges from themselves to the universal sink.

(e) A vertex that has a 1 in any part of its row in the matrix or a 0 in any part of its column in the matrix other then the cell that is a self-loop then it is not a universal sink.

I claim that FindUniversalSink will always return a universal sink if there is one to be found in the adjacency matrix A otherwise NO is returned. It does this by looking at and assuming every vertex in the adjacency matrix is the universal sink until proven otherwise.

The algorithm begins with a adjacency list. It continues by iterating vertex by vertex through the adjacency list, setting each vertex as the universal sink. Each time a new vertex is set as the universal sink it first looks at the entire column of the vertex and checks to see if there is a 0 in any place other than the self loop cell where row number i equals column number j. This indicates this vertex is not a universal sink because it does not have an edge from every other vertex in the graph going to it. The algorithm then looks at the row of the vertex to see if there are any 1's in any place. This also indicates this vertex is not a universal sink because a universal sink does not have any out-edges. If it is found that this vertex is not a universal sink by either looking at the column or the row of the vertex, the algorithm sets the vertex to not being a universal sink and breaks out of whichever loop it is currently and ends up at a conditional statement.

This conditional statement checks if the vertex is still a universal sink. If it is a universal sink the vertex is returned otherwise two things could happen. We could be at the last vertex in the adjacency list which, if we have reached this point, is not a universal sink so there exists on universal sink the graph. Or we could not be at the last vertex in the adjacency list which means we will go back to the top loop and move to and check the next vertex in the adjacency list.

Runtime
Unfortunately, I was unable to to create an algorithm that had a runtime of $o(|V|^2)$. The first loop of FindUniversalSink runs from $|V|$ times as it goes through all of the

vertices. With in that loop, there are two other loops that also run through all of the vertices in the adjacency graph so they run a total of $2|V|$ times. The rest of the operations like checking the values in the cells of the matrix or seeing if a vertex is a universal sink run in in constant time.

This means this algorithm has a runtime of $O(|V| * 2|V|) = O(|V|^2)$.