Problem Set 6

All parts are due May 5th, 2016 at 11:59PM.

Name: Arturo Chavez-Gehrig

Collaborators: Sabrina Ibarra, Julian Alverio

Part A

Problem 6-1.

(a) Since we know the 'U' shape of a parabola, if we select η correctly, we will bounce continuously between symmetrical points of the parabola.

$f(x) = 4x^2 - 12x + 0$

$f'(x) = 8x - 12$

$x_0 = 0$

$x_1 = 0 - \eta f'(x_0)$

$3 = 0 + \eta 12$

$\eta = \frac{1}{4}$

Sequence

$x_0 = 0, x_1 = 3, x_2 = 0, x_3 = 3, x_4 = 0, ...$

(b) $f'(x) = 8x - 12$

$-\eta f'(x) = -\frac{1}{2}(x - \frac{3}{2})$

$-\eta(8x - 12) = -\frac{1}{2}(x - \frac{3}{2})$

$-\eta 8 = -\frac{1}{2}$

$\eta = \frac{1}{16}$

In this case the error is divided in two at each step leading to log number of steps.

$e_0 =$ initial error

$e_0 = |(f(x_0) - f(x^*)|$

$e_i = \frac{e_0}{2^i} \leq \epsilon$

$\frac{e_0}{\epsilon} \leq 2^i$

$log(\frac{e_0}{\epsilon}) \leq i$

So for all j greater than or equal to i, it'll be $\epsilon-$good. The smallest index i that is $\epsilon-$good is bounded by $Olog(\frac{1}{\epsilon})$

(c) $x^* = \frac{-b}{2a}$

$f'(x) = 2ax + b = 2a(x + \frac{b}{2a})$

$x_{i+1} = x_i - \eta f'(x_i)$

$-\eta f'(x_i) = \frac{1}{2}(x_i + \frac{b}{2a})$

$\eta 2a(x + \frac{b}{2a}) = \frac{1}{2}(x_i + \frac{b}{2a})$

$\eta = \frac{1}{4a}$

$e_0 = |(f(x_0) - f(x^*)| = a\,|x_0 - x^*|$

$e_i = \frac{e_0}{2^i} \leq \epsilon$

$\frac{e_0}{\epsilon} \leq 2^i$

$log(\frac{e_0}{\epsilon}) \leq i$

So for all j greater than or equal to i, it'll be $\epsilon-$good. The smallest index i that is $\epsilon-$good is bounded by $Olog(\frac{a|x_0-x^*|}{\epsilon})$

pseudocode

def GoodStep($function, x_0, a, b, c, \epsilon$)

$x = x_0$

while $\left|f(x) - f(\frac{-b}{2a})\right| \leq \epsilon$

$x_{old} = x$

$x = x_{old} - \frac{1}{4a}(2ax_{old} + b)$

return x

(d) Given x_0 has $f'(x_0) < 0$ and x_1 has $f'(x_1) > 0$, we know that the quadratic function's minimum value falls between the two x values. We can apply binary search to reduce the set of possible solutions in half at each step. We will take the middle value between x_0 and x_1 and update until we satisfy the stopping condition $|f(x) - f(x^*)| \leq \epsilon$. The

run time of this algorithm is bounded by the log of the difference between the starting values of x, because each step reduces the difference between the two x values by one half so effectively log base two. The run time of the algorithm is also bounded by the degree of accuracy, so the total run time of this algorithm is $O(log(\frac{a-x_0-x_1}{\epsilon})$

Def Binary-search($function, x_0, x_1, a, b, c$)

newx = $(x_0 + x_1)\frac{1}{2}$

slope = $2a(newx) + b$

if $\left|f(newx) - f(\frac{-b}{2a})\right| \leq \epsilon$

return newx

elif slope > 0:

return Binary-search($x_0, newx, a, b, c$)

else slope < 0:

return Binary-search($newx, x_1, a, b, c$)

(e) Taylor's approximation

$f(x + \epsilon) = f(x) + \epsilon f'(x) + \frac{\epsilon^2}{2} f''(x)$

$\frac{d}{d\epsilon} = f'(x) + \epsilon f''(x) = 0$

$\epsilon = -\frac{f'(x)}{f''(x)} = -\eta f'(x)$

$\eta = \frac{1}{f''(x)} = \frac{1}{2a} = \frac{1}{8}$

$x_1 = 0 - \frac{1}{8}(-12) = \frac{12}{8} = \frac{6}{4} = \frac{3}{2}$

(f) To efficiently approximate the optimal step size, we start with the minimum η. With the minimum η, we calculate the x value. Then we double the current η and then calculate the mean squared error. If the doubled η's mean squared error is larger than the preceding ηs, the previous η (with the smaller mean squared error) becomes the approximate η^*. We average η with its doubled value calculate its mean squared error. We compared this result our previous approximate for η^*. We can use mean squared error to perform binary search effectively reducing the possible η^* solution set in half. We continue this binary search process until the difference between the mean squared error and the η is smaller than a defined error constant.

(g) We know gradient descent updates based on traversing the opposite direction of the gradient vector towards the minimum. In x^3, there is no convergence on a global min. The only place that will allow gradient descent to converge is at $x = 0$. For this to happen, we must start wiht an x greater than or equal to 0, but smaller than a value of x^{max} where it will overshoot 0 and never converge. Since $f'(x) = 3x^2$, we know the y values scale faster than the x values so as x increases it will only overshoot further. To find the upper bound on the values for x that will converge:

$0 \leq x - \eta 3x^2$

η

$3x^2 - x \geq 0$

$x(3x - 1)$

solve we get that $x = 0, \frac{1}{3}$

$f'(x^3) - 3x^2$

Problem 6-2.

(a) We will exhaustively search the set of possible arrangements by representing the problem as a collection of subproblems. This sequence of problems can be organized into a binary tree. We use recursion to traverse each level. We stop when one list is empty and the other list has only one element.

Time analysis:

Dependency graph has $O(M + N)$ length branches and number of branches is $\binom{M+N}{N}$ giving run time of $O((M + N) * \binom{M+N}{N})$

def start(A,B,MA,MB):

A-start= A

B-start= B

merged-total= $BestOrdering(0, 0)$

def bestOrdering(A,B):

if A=length(A-start) and B = length(B-start)-1

return 0

if A=length(A-start)-1 and B = length(B-start)

return 0

if A and B are not the ends of the list, do the following transitions

return max((MA[A][B]+bestOrdering(A+1, B)), (MB[B][A]+bestOrdering(A, B+1)))

(b) We can be more efficient by not repeating previous computations. To do this, we will track the index of the remaining restaurants in each list. We will create a dictionary that maps the indices to their optimal donation values. This algorithm has $O(M * N)$ subproblems with each taking $O(1)$ time. Total run time is $O(M * N)$.

def start(A,B,MA,MB):

A-start= A

B-start= B

sub-problems = dict()

sub-problems[length(A),length(B)-1]= 0

sub-problems[length(A)-1,length(B)]= 0

merged-total= $BestOrdering(0, 0)$

def bestOrdering(A,B):

if A=length(A-start) and B = length(B-start)-1

return 0

if A=length(A-start)-1 and B = length(B-start)

return 0

if $(A + 1, B)$ and $(A, B + 1)$ are keys in sub-problems:

return max((MA[A][B]+bestOrdering(A+1, B)), (MB[B][A]+bestOrdering(A, B+1)))

else:

if A and B are not the ends of the list, do the following transitions

sub-problem-solution=max((MA[A][B]+bestOrdering(A+1, B)), (MB[B][A]+bestOrdering(A, B-1)))

sub-problem[A,B]=sub-problem-solution

return sub-problem-solution

(c) To employ the bottom-up approach, we will replace the dictionary that we used for memoization with a table. The table will have M rows and N columns. The bottom left of the table will the optimal path when we have included 0 restaurants from both lists and the top right entry of the table will hold the optimal donation total when all $M+N$ restaurants are included in our ranking. The algorithm will fill the table starting in the bottom left and moving up and to the right until reaching (M,N). This structure takes advantage of the optimal substructure property of this dynamic programming problem. This is evident since solutions from below or to the left are the inputs to compute the maximum donation in the table at position (i,j). Each time step takes $O(1)$ time. There are $O(M * N)$ subproblems. This gives the total run time of $O(M * N)$.

```
def bottom-updonations(A,B,MA,MB):
    merged=dict()
    for i in range(length(A)):
        for j in range(length(B)):
            if i or j = 0:
                merged[(i,j)]
            else:
                cellbelow=MA[i][j]+merged[(i − 1,j)]
                cellleft=MB[j][i]+merged[(i,j − 1)]
                merged(i,j)=max(cellbelow,cellleft)
    return merged(length(A)−1,length(B)−1)
```

(d) The run time of this algorithm is still $O(M * N)$. Adding parent pointers $O(1)$ and iterating through the list of orderings to recreate the path is $O(M + N)$.

```
def bestMergeLists(A,B,MA,MB):
    m=length(A)
    n=length(B)
    ranking=[]
    merged=dict()
    parent=dict()
parent[(0, 0)]=none
    for i in range(length(A))
```

```
    merged[(i,0)]= 0
    parent[(i,0)]=((i − 1, 0), i − 1))
for j in range(length(B))
    merged[(0,j)]= 0
    parent[(0,j)]=((0, j − 1), j − 1))
for i in range(length(A))
    for j in range(length(B))
        transup=MA[i][j]+merged[(i − 1, j)]
        transright=MB[j][i]+merged[(i, j − 1)]
        best=max(transup,transright)
        if best=transup:
            parent[(i,j)]=((i − 1, j),i − 1)
            restaurant=A[m − 1]
        else:
            parent[(i,j)]=((i, j − 1),j − 1)
            restaurant=B[n − 1]
        merged[(i,j)]=best
p=(length(A)−1,length(B)−1)
ranking.append(restaurant)
while parent[p] not = None:
    rank.append(parent[p][1])
    p=parent[p][0]
return ranking
```

Problem 6-3.

(a)

```
def findpathcost(V,E,start,target,k,w,weights)
    weight= ∞
    if k= 1:
```

if (start,target) exists in E:

return(weights[start,target]−w[k])^2

else:

return ∞

for destination in E[start]:

next=findpathcost(V,E,destination,target,$k-1$,weight,w)

if weight>(weights[start,destination]−w[k])^2+next:

weight=(weights[start,destination]−w[k])^2+next

return weight

The run time for this algorithm is bounded by the path length k and the number of vertices in the path. The upper bound for this is $O(K * V^k)$.

(b) memo=dict()

def findpathcost(V,E,start,target,k,w,weights)

weight= ∞

if k= 1:

if (start,target) exists in E:

return(weights[start,target]−w[k])^2

else:

return ∞

for destination in E[start]:

if destination in memo:

next=memo[destination]

else:

next=findpathcost(V,E,destination,target,$k-1$,weight,w)

if weight>(weights[start,destination]−w[k])^2+next:

weight=(weights[start,destination]−w[k])^2+next

memo[destination]=weight

return weight

This algorithm is more efficient than in part a because we do not repeat computations. We keep track of solutions we've already computed in the memo dictionary, then before computing the answer to a subproblem, we check if it already exists in memo. This way, we only calculate it once. For this reason, the run time of this algorithm is now $O(K * V^2)$.

(c) In the bottom-up approach, we will be computing shortest paths from the start to all reachable vertices. We will be doing this exhaustively, and unlike in part b, this will be on a reversed graph where edge directions are reversed and start and target are flipped. Otherwise, the algorithm is very much the same, including its run time $O(K * V^2)$.

(d) The memoized algorithm is much more efficient than the bottom-up algorithm in the case that the number of edges and vertices near the target is very dense relative to the rest of the graph. If we chose to use bottom-up, we would be exhaustively searching the nodes near to the target before expanding outwards to find the start.

(e)

```
memo=dict()
global variable Z
def findpathcost(V,E,start,target,k,w,weights)
    weight= ∞
    path=[]
    if k= 1:
        if (start,target) exists in E:
            memo[start]=((weights(start,target)−w[Z − k])^2,[target,start])
            return((weights[start,target]−w[Z − k])^2,[target,start])
        else:
            return ∞
    for destination in E[start]:
        if destination in memo:
            next=memo[destination]
            nextpath=memo[destination]
        else:
            next=findpathcost(V,E,destination,target,k − 1,weight,w)
```

nextpath=findpathcost(V,E,destination,target,$k-1$,weight,w)

if weight>(weights[start,destination]−w[$Z-k$])^2+next:

weight=(weights[start,destination]−w[$Z-k$])^2+next

path=nextpath

memo[destination]=(weight,path.append(start))

return (weight,path.append(start))

The true path is the reverse of the path that was returned. The run time of this algorithm is $O(K * V^2)$. Taking the reverse of the path takes $O(V)$ so it is not one of the high ordered terms that determines the asymptotic running time.

Part B

Problem 6-4.

(a) $J(x) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta_i x)^2$

$\nabla J(x) = \frac{-1}{n}\sum_{i=1}^{n} 2(y_i - \theta_i x)\theta_i = \frac{-2}{n}\sum_{i=1}^{n} 2(y_i - \theta_i x)\theta_i$

(b)

(c)

(d)

(e) $\frac{1}{n}(y-\theta x)^T(y-\theta x)$

$\frac{1}{n}(y^T - \theta^T x^T)(y-\theta x)$

$\frac{1}{n}\frac{\partial}{\partial x}(y^Ty - \theta^T x^T y - y^T\theta x + \theta^T x^T \theta x)$

$\frac{1}{n}(y^Ty - \theta^T(x^Ty - x^T\theta x) - y^T\theta x)$

$\frac{1}{n}(0 - (\frac{\partial(x^Ty)}{\partial x} - \frac{\partial(x^T\theta x)}{\partial x})\theta^T - y^T\theta)$

$\frac{1}{n}(-(y - (\theta + \theta^T)x)\theta^T - y^T\theta)$

$\frac{1}{n}(\theta x\theta^T + \theta^T x\theta^T - y^T\theta - y\theta^T)$

Alternatively:

$\nabla J(X) = 0 = \frac{-2}{n}\theta^T(y-\theta x)$

$0 = \theta^T y - \theta^T\theta x$

$\theta^T y = \theta^T\theta x$

$x = (\theta^T\theta)^{-1}\theta^T y$