Assignment 3

Danilo Vladicic - dvladici@uwo.ca - 250861339

April 1, 2020

Question 1: Next[] Table

The pattern string is $P = babbabbabbababbabb$

```
i       = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
P[i]    = b a b b a b b a b b  a  b  a  b  b  a  b  b
next[i] = 0 0 1 1 2 3 4 5 6 7  8  9  2  3  4  5  6  7
```

Question 2: String Matching

Modify KMP to find the largest prefix of P that matches a substring of T.
Let $P = p_1p_2...p_m$
Let $T = t_1t_2...t_n$

Modified KMP

```
-Initialize variable longest_match to 0
-Initialize variable current_match to 0
-Start comparing characes in P and T:
    -For each match:
        - update current_match = current_match + 1
    -For mismatch at P[q + 1]:
        -update longest_match = max{longest_match, current_match}
        -update current_match = next[q]
        -q = next(q)
        -Same character in T is compared to P[next[q] + 1]
        -Only increase T when mismatch occurs on P[1]
-Once all of T has been checked update
 longest_match = max{longest_match, current_match}
 once more for the case of the longest prefix of P in T
 is at the end of T
```

Correctness

Suppose there is a mismatch at $P[q+1]$ and $T[i]$. For each match *current_match* will be incremented by 1. Before the mismatch $current\_match = q$ and suppose *current_match* is the longest match. The mismatch occurs at $q + 1$, I update both q and *current_match* to $next[q]$, which is always strictly $< q+1$. If $P[next[q] + 1]$ matches T[i] then the $current\_match = next[q] + 1$, since before rechecking T[i] *current_match* was set to $next[q]$ after the new check it will update to $current\_match + 1 = next[q] + 1$. If the mismatch occurs at $P[1]$ then $next[q] = 0$ and $current\_match = 0$. Early termination could

be added for when *longest_match* is the length of P, since there will be no longer prefix of P that is longer than P itself. If there is never a match then *current_match* is never updated and always stays 0. Consider

```
index =   1 2 3 4 5 6 7 8 9 10
P =       a b b a b b b
next[q] = 0 0 0 1 2 3 0
T =       a b a b b a b b a b
```

longest_match and *current_match* = 0. The first mismatch to occur is $i = 3$, $T[3] = a$ & $P[3] = b$. At this point *current_match* = 2, and *longest_match* = 2. $q = next(1) = 0$ and *current_match* = 0. Now $T[2]$ is compared to $P[1]$. The first mismatch after this point is at *index* = 9 and $q + 1 = 7$. *current_match* = 6 since $P[1]$ to $P[6]$ match. l*ongest_match* = 6 since it's > 2. *current_match* = $q = next(6) = 3$. So $T[8]$ is to be compared with $P[4]$ (q+1). Mismatch on $T[8]$.

```
i = 1 2 3 4 5 6 7 8 9 10
T = a b a b b a b b a b
P =     a b b a b b b
```

After updating q = 4

```
i = 1 2 3 4 5 6 7 8 9 10
T = a b a b b a b b a b
P =           a b b a b b b
q =           1 2 3 4 5 6 7
```

Since $P[4]$ matches $T[8]$ *current_match* is updated to be 4. The last match is $P[5]$ and $T[9]$ and so *current_match* = 5. Since T has no more characters to check, the final part of the algorithm update *longest_match* to the final match and so the returned value is 5 characters.

Complexity

The complexity of calculating the next table is O(m) and the complexity of the search is O(n). For the search suppose you have no matches. Every $T[i]$ will be compared to $P[1]$ and each comparison results in i being incremented. Thus only, n elements are compared. When there is a match, both i and q are incremented. When the mismatch happens i, is left unchanged and q is set to $next[q-1]+1$. For P's longest prefix to be a substring of T then the largest possible prefix is n (When $P = T$). If a mismatch occurs at $T[1]$ then the longest prefix is $m-1$. Since $next[q-1]+1 < q$, the maximum number of rollbacks for P is $m-1$ (case of $T = bbba$, $P = bbbb$). Then when

the mismatch occurs at $T[4]$, first $P[3]$ is compared then $P[2]$, then $P[1]$, which also mismatches, but increments i, and the algorithm finishes. It can be seen then that there can be no more than $2 * n$ comparisons and so it always runs O(n).

Question 3: Longest Common Subsequence

Give pseudocode to reconstruct an LCS from the completed c table, and the original sequences $X =< x_1x_2...x_m >$ and $Y =< y_1y_2...y_n >$ in O$(m + n)$ without using the b table

Reconstruct LCS

```
INPUT: X and Y are sequences
       i is the length of X
       j is the length of Y
       c is the c table of X and Y
Output: A valid LCS of X and Y
RECONSTRUCT-LCS(X, Y, i, j, c):
    if i = 0 and j = 0 then
        return
    else if X[i] = Y[j] then
        RECONSTRUCT-LCS(X, Y, i - 1, j - 1, c)
        print X[i]
    else if c[i-1, j] >= c[i, j-1]
        RECONSTRUCT-LCS(X, Y, i - 1, j, c)
    else
        RECONSTRUCT-LCS(X, Y, i, j - 1, c)
```

The first call would be $RECONSTRUCT - LCS(X, Y, m, n, c)$.

Correctness

Given X and Y of sizes m and n respectively, then the longest LCS has at most length $k = min\{m, n\}$ which implies that either all of X is in Y or all of Y is in X. Let Z_k be this LCS. Suppose that $x_m = y_n$. This element will always be the last element of all the possible LCS, and the remaining Z_{k-1} are strictly a subproblem of X_{m-1} and Y_{n-1}. If $x_m \neq y_n$ then the LCS is either a subproblem of X_m and Y_{n-1} or X_{m-1} and Y_n. The cost table, $c[i, j]$ is defined to be an LCS of the sequences X_i and Y_j. Thus, if $c[m - 1, n] > c[m, n - 1]$ then Z_k is a problem of X_{m-1} and Y_n. Otherwise Z_k is a subproblem of X_m and Y_{n-1}. If $c[m - 1, n] = c[m, n - 1]$ then both subproblems are valid solutions to Z_k so the algorithm will still be correct. The recursion is expanded until both $i = 0$ and $j = 0$ where it

starts collapsing, in which case only elements that are both in X and Y and part of Z_k are output.

Complexity

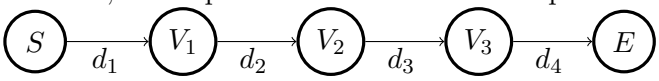
The worst case for the algorithm I described above is whenever no elements of X exist in Y. This is because whenever there is a shared element, both sequences can remove that character from the subproblem. In the worst case either only i is reduced or only j is reduced. Since $i = m$ and $j = n$ initially and the recursive calls end when $i = 0$ and $j = 0$ then it can be seen the total complexity is $\mathrm{O}(n + m)$.

Question 4: Greedy Algorithm

The Problem

Prof. Gekko is planning to **inline** skate across a highway. He can carry 2 liters of water and can skate m miles before running out of water. He starts with a full 2L. He has a *map* of all the places he can refill his bottle, and the distances between these locations. The profs **goal** is to minimize the number of water stops along his route.

To start, it's important to see how this map looks.



So we can see there are a couple of variables we'll need to consider.

$$capacity = 2L$$

$$distance = m\,miles$$

$$G = (V, E)$$

$$V = \{v_1, ...v_n\}$$

$$E = \{d_1, ..., d_{n-1}\}$$

$$dist(v_i, v_j) = \Sigma_{x=i}^{j} d_x < m$$

Where $v_1 = s$, $v_n = e$, $d_1 = dist(\{s, v_2\})$, and $d_{n-1} = dist(\{v_{n-1}, v_n\})$. A couple things to extract from this, is if $d_i > distance$ then the prof will necessarily need to refill at v_{i+1} for any d_i. However if $d_i < distance$ then the does not necessarily need to refill his water.

The above section describes what is given to us. The required output is which stops the prof. needs to make. Thus, the output of my algorithm will be a set $S = \{s_1, s_2, ..., s_n\}$, containing some subset of V.

The Algorithm

Considering everything above, a simple algorithm for computing the stops is as follows.

```
-Let S be an empty set
-Let the profs current position be x = 1
-while x < n:
    -Find the furthest stop i, reachable from x, that is before or at n
    -Add stop i to S
    -Set x to i
-return S
```

The farthest stop reachable from x is the stop t whose $dist(v_x, v_t) \geq m$ and $dist(v_x, v_{t-1}) < m$ and $dist(v_{t-1}, v_t) < m$. This means upon reaching stop v_{t-1} the prof still has some water left over, and he proceeds to v_t. However, between his journey from v_{t-1} and v_t he consumes all of his remaining water.

Correctness

The correctness of the algorithm depends on the metrics described above, especially in regards to how to handle points where, the prof doesn't have enough water to cover an entire distance. To handle those points my algorithm only refills water at the points where the prof has no water. It could also be adjusted to only refill at points where the prof doesn't have enough water to proceed to the next point. In the latter case, any distance ¿ m, with a full capacity, would cause the prof. to stall, since he can't find a path between those points. This is not what the prof. wants, hence the former metrics gurantee a path. Another assumption could also be made that all the stops are within m miles. In any case to prove correctness, we must prove that the algorithm always finds a path.

Proof: The algorithm always finds a path.

Suppose it did not find a path.

Let the position of the stops $S = \{s_1, s_2, ..., s_m\}$ be the stops taken by the prof.

Since the stops are inline then $s_1 < s_2 < ... < s_m$, and we don't have a path, the algorithm stopped at some v_i and couldn't proceed to v_{i+1}. This means that $d_i + m < d_{i+1}$

Since there is a path from the start, v_1, to the end, v_n, there must be a stop that happens before v_{i+1} and ends at or after v_{i+1}. This stop can't be

v_i, so it must be at some v_s where $s < i$. Since all our vertices are **inline** then, $d_s + m < d_i + m < d_{i+1}$. Therefore, this stop can't exist, so the assumption is wrong and the algorithm always finds a path.

Optimalness

The algorithm yields an optimal solution.
Let $S = s_1, ..., s_n$ be the path returned by my algorithm.
Let $S^* = s_1^*, ..., s_m^*$ be an optimal path.
Let $|S|$ and $|S^*|$ denote the number of stops in S and S^* respectively.
Let $p(i, S)$ denote the profs position after the first i stops.

We assume that S is not optimal. This implies that

$$|S| \geq |S^*|$$

since if $|S| < |S^*|$ then S^* is not optimal.

Lemma 1: For all i: $0 \leq i \leq |S^*|$ we have $p(i, S) \geq p(i, S^*)$.

Proof by Induction:
Base: $i = 0$ then

$p(0, S) = 0 \geq 0 = p(0, S^*)$.
since the prof has not moved.

Induction: We assume this holds for $i = k$

Case 1: $p(i, S) \geq p(i + 1, S^*)$

Since each stop is farther than the next, then $p(i+1, S) \geq p(i, S)$. Then, $p(i + 1, S) \geq p(i + 1, S^*)$.

Case 2: $p(i, S) < p(i + 1, S^*)$

Each stop S and S^* is at most, m miles apart. Consider $p(1, S*) = m$ and $p(1, S) = 2m$. This cannot occur, since my algorithm refills every time the prof has no water, which occured at stop 1 in $|S^*|$, which means he would not have continued to stop 1 in S.
With that in mind, it can be seen $p(i+1, S^*) \leq p(i, S)+m$. Thus, the greedy algorithm can at least get to $p(i + 1, S^*)$ and by the induction hypothesis $p(i + 1, S) \geq p(i + 1, S^*)$.

Now we can continue to the more important proof.

Theorem: My solution S is optimal. So $|S| = |S*|$.

Proof: Since S^* is optimal, we know $|S^*| \leq |S|$ (*Optimalness*).
Suppose $|S*| < |S|$
Let $k = |S*|$
By *Lemma 1*, we know

$$p(k, S^*) \leq p(k, S)$$

Since the prof reaches position n (the end), after k stops along S^*, we know $n = p(k, S^*)$. From *Lemma 1*, we know $p(k, S^*) \leq p(k, S)$, and since $|S^*| < |S|$ then the last stop in S is beyond the point n. This means that my algorithm must have made an extra stop after the $k - th$, which is a contradiction to the algorithm's correctness which gurantees to stop at n. Therefore this is a contradiction to the claim $|S^*| < |S|$ and so $|S| = |S^*|$ and the solution provided by my algorithm is optimal.

Complexity

Suppose there are n vertices in the graph. The prof. starts at vertex 1. The while loop executes exactly n times when all the pairs of vertices are $dist(v_i, v_{i+1}) \geq m$. In this case, the algorithm runs in O(n). When vertices are $dist(v_i, v_{i+1}) < m$, the body of the while loop, would run until it found a vertex whose total distance is or $m \geq total\_dist < 2m$, but the profs current position is updated to that vertex's index, and so in any case, the total execution of the loop will be O($|V|$) = O(n).

Question 5: Word Wrapping Problem

The Problem

Each line has a width of M chars. You are given a list of n words $W = \{w_1, ..., w_n\}$. Each words w_i has length L_i chars.

The problem is to break W into lines of length at most M, in a way that minimizes the total cost.
The cost of using a single line to hold words $w_i, w_{i+1}, ..., w_j$ is given by the function

$$c(i, j) = (M - (j - i) - \Sigma_{k=i}^{j}(L_k))^3$$

The parts of the equation:
$(j - i)$: The additional amount of spaces needed for the words.
$\Sigma_{k=i}^{j}(L_k)$: The total amount of characters each word uses.

If $c(i, j) < 0$ then the words cannot fit on the line and we set $c(i, j) = \infty$ If $c(i, j) > 0$ and $j = n$ then $c(i, j) = 0$ which indicates there is no penalization

for wasted space in the last paragraph.

Now consider the case when the words occupy just two lines at some breakpoint i. The $total\_cost = c(1,i) + c(i+1,n)$. Since the last line is always 0, this is equal to $total\_cost = c(1,i)$. This is important because we observer to get the **minimum** total cost, we want the last line to have its **maximum** possible cost. This really means we want $c(1,i) > 0$ and $c(1,i+1) = \infty$.
Next we consider the case of 3 lines with n elements at breakpoints i, j. Then the $total\_cost = c(1,i) + c(i+1,j) + c(j+1,n)$. Again $c(j+1,n) = 0$. If we follow the greedy algorithm described above, then we want $c(1,i+1) = \infty$ and $c(i+1,j+1) = \infty$. However, it is possible that $c(1,i-1) > c(1,i+1)$ and $c(i,j) < c(i+1,j)$ since the word w_i can fit in the second line without moving w_j to the last line.
To see this consider this example:
$W = \{aa, bb, c, dddd\}$ and $M = 5$.
The greedy algorithm would suggest the following orientation (cost per line on the right):

$$\begin{bmatrix} a & a & \_ & b & b & (0)^3 \\ c & \_ & \_ & \_ & \_ & (4)^3 \\ d & d & d & d & \_ & 0 \end{bmatrix} \; total\_cost = (0)^3 + (4)^3 = 84.$$

However the least total cost would come from

$$\begin{bmatrix} a & a & \_ & \_ & \_ & (3)^3 \\ b & b & \_ & c & \_ & (1)^3 \\ d & d & d & d & \_ & 0 \end{bmatrix} \; total\_cost = (3)^3 + (1)^3 = 28.$$

From this we extract a very important piece of information. That being that the minimum total cost depends more-so on the balance of where the line starts and ends rather than just how much the cost is of an individual line. That being rather than considering minimizing the cost of each line, we want to minimize the total cost with respect to each lines starting point. Thus, my algorithm will involve two parts, first constructing a cost table for all the possible lines, and second for finding the endpoints for the least *total_cost* for all the lines using this precomputed table.

The Algorithm

Part 1: Preprocessing

```
Input: The list of words W, and the cost function c(i, j).
Output: Array cost[1..n, 1..n]
COST-TABLE(W, c):
    cost[1...n, 1..n] = [INF, ..., INF] #set all entries to INF
    for i = 1 to n:
        for j = i to n:
```

```
        cost[i, j] = c(i, j)
    return cost
```

An important thing to note is that the $cost[i, j] = \infty$ when $i > j$. Another thing to note is that $cost[i, i]$ is the cost of storing just word w_i on the line. Now to construct the starting points of these lines. We know the maximum number of lines is n, (one line per word), and the minimum is 1 line (all words fit on a single line). Suppose my output is T. Since we want to return the break points where the least cost for that line starts, then when its one word/line, then $T[i] = i$. When this isn't the case, $T[n] = i$ means the last line is given by $w_i w_{i+1} ... w_n$. Similarly if $T[n-1] = t$ means the second last line is given by $w_t, ..., w_{T[n]-1}$

```
Input: c the cost table
Output: T[1, ..., n] the points where the line starts
COMPUTE-LC(c):
    total_cost[1..n] = [INF, ..., INF]
    T[1, ..., n] = [0, ..., 0]
    total_cost[1] = 0
    T[1] = 1
    for i = 2 to n:
     for j = 1 to i:
         if total_cost[j] != INF
            and c[j, i] != INF
            and total_cost[j] + c[j, i] < total_cost[i] then
               total_cost[i] = total_cost[j] + c[j, i]
               T[i] = j
    return T[i]
```

To output the lines all that is required is

```
Input: T the breakpoints, n is the number of total words
PRINT-LINES(T, n):
   if T[n] == 1:
      line = 1
   else
      line = 1 + PRINT-LINES(T, T[n]-1)
   print "Line " + line + " Starts at " T[n] " and Ends at " n
   return line
```

Correctness

Suppose all the words require their own line. The cost table $c[i, j] = \infty$ for all values when $j \neq i$. Thus, the internal loops if condition can only be met when $i = j$. Since $total\_cost[i]$ won't be updated till this condition is

met, then the only update that occurs to $T[i]$ is when $i = j$ and so the break points returned will be $1, 2, ..., n$. Which when calling $PRINTLINES(T, n)$ the output will be
"Line 1 Starts at 1 and Ends at 1"
...
"Line n Starts at n and Ends at n"

Suppose all the words fit on a single line. First we notice that $c[i, j] \neq \infty$ for all $j \leq i$. Consider the first iteration, $i = 2$, $j = 1$, $total\_cost[i] = \infty$. Then $total\_cost[2] = c(1, 2)$ and $T[2] = 1$. Next $j = 2$. then then condition $total\_cost[j] + c[j, i] < total_cost[i]$ will never execute since $c(1, 2) + c(2, 2) < c(1, 2)$. This is repeated for all values of i and so $T[i] = 1$ for all i. When calling $PRINT - LINES(T, n)$ then $T[n] = 1$ and the output is
"Line 1 Starts at 1 and Ends at n"

Finally consider the case $W = ((aa), (bb), (c), (dddd))$ with $M = 5$. The algorithm starts with $i = 2$ and $j = 1$. After the first iteration $total\_cost[2] = c(1, 2)$ and $T[2] = 1$. Next the $total\_cost[2] + c(2, 2) < total\_cost[2]$ will not be met. The next iteration $i = 3$, and $j = 1$. The $c(1, 3) = \infty$ so the if condition is not met and now $j = 2$. At this point $total\_cost[j] = c(1, 2)$ and $total\_cost[3] = \infty$ since it has not yet been updated. Since all the conditions are met $total\_cost[3] = c(1, 2) + c(2, 3)$ and $T[3] = 2$. Next $j = 3$. At this point $total\_cost[i] = total\_cost[j]$ and the 3rd condition of the if statement will not be met. For the last word, $i = 4$, for all $j < i$ $c[j, i] = \infty$. Thus $T[4]$ will only get updated when $j = i$. Calling the $PRINT - LINES(T, n)$ will result in
"Line 1 Starts at 1 and Ends at 1"
"Line 2 Starts at 2 and Ends at 3"
"Line 3 Starts at 4 and Ends at 4"

For any case the algorithm will output correctly since for each iteration of i, it finds the least cost starting point of the i-th line storing some amount of words j where $j \leq i$. Since the goal is to minimize the $total\_cost$ across all the lines, the first j that fits all the words up to i is stored in T. All starting points x after j, $j < x \leq i$ will have a $c(x, i) > c(j, i)$ the minimum $total\_cost$ for that line will always be $c(j, i)$. However, $c(j, i+1)$ may or may not fit in the line as well. This gets checked each time i is incremented. To reconstruct the lines, we know each $T[i]$ stores where that line starts. We know the last line always ends at w_n and its starting point will be $T[n]$. Then the second last line will start at $T[n-1]$ and end at $T[n]-1$. Suppose that $T[n] = 3$. This indicates when $i = n$ the if conditions were first met when $j = 3$. Since this is true when $i = n$ then it also would have been true when $i = 3$ and $j = 3$, as the $c(i, j)$ only gets smaller as j increases. Once we know $T[n] = 3$ the only time $T[i]! = 3$ is when $i < 3$, and the first occurence of

that is $T[n]-1$. So the next recursive call for $PRINT-LINES(T, T[n]-1)$ is correct as the next line's starting point will first occur then.

Complexity

In any case, the outer loop will run n-1 iterations. The inner loop runs i times for each iteration of the outer loop. In total the inner loop executes $1 + 2 + ... + n$ iterations once the outer loop has finished. Then it's easy to see that the algorithm runs in $O(n^2)$.

Question 6: Maximum Spanning Tree

We want the maximum spanning tree, which means we want the set of edges that connects all the vertices together, with the maximum total edge weight, and no cycles.

Modified MST

```
-Let G = (V, E)
-Let T be the set of vertices who are in the Maximum Spanning Tree
-Let parent[1,..., n] store the node used to reach the node at any index
-Arbitrarily select a starting vertex r
-Initialize the parent[1,...,n] to NULL
-Initialize the key for r to be 0
-Initialize the remaining keys to -INF
-Let Q be a max heap storing the keys of vertices
-While Q is not empty:
   -Let v be the vertex with the maximum key from Q
   -Add v to T
   -For each (v, u) in E and w not in T:
    -if w(v, u) > Q[u]:
    -update key[u] = w(v, u) (in Q)
    -update parent[u] = v
```

Correctness

Initially all the keys for the vertices are $-\infty$ except for the starting vertex r. Q stores all these keys in a max heap. The first iteration removes r from the heap and updates all the nodes connecting to r with the weights of those edges. The next node removed has the highest edge weight from r. If this new node u connects to any nodes that r connected to, with higher weights, then the keys for those nodes will be update to $w(u, v)$ and their parents will be updated as well. Since each node is only updated if they are not a part of T, and in each iteration the node with the maximum key is removed,

then we gurantee that each node will only appear in T once, with the node that has highest edge weight connecting it stored as the parent of that node.

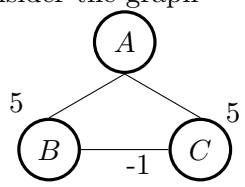
Complexity

Let $|E| = number\ of\ edges$ and $|V| = number\ of\ vertices$. We observe that the total number of times the while loop runs is $|V|$ since each time 1 node is removed. If every vertex v has the maximum number of edges then $|E|_v = |V| - 1$. Then for the first node the inner for loop executes $|V| - 1$ times. The next iteration of the while loop the inner for loop will run a maximum of $|V| - 2$ times, since every vertice lost the edge with r. For the last vertex, the inner for loop will run 0 times, as there a no nodes that are not in T except for itself. Similarly the second last iteration will only have 1 edge to evaluate. So in total the inner loop runs $|V|-1+|V|-2+...+1+2$ which is the same as the total number of edges $|E| = (|V|)(|V-1|)/2)$. The top part of the while loop runs in O(1) so the majority of the complexity comes from updating the key in the heap, which runs in $O(\log |V|)$. So in total the complexity would be $O(|E| \log |V|)$.

Question 7: Dijkstra's

Counter-Example

Consider the graph



Discussion

In terms of Dijkstra's it's relitevely unimportant which edge has the negative edge since it'll return an incorrect solution one way or another. We suppose that the algorithm starts at A. The edges for B and C are both updated to 5. The next iteration removes suppose B is selected by some tie-breaking scheme (it won't matter later). This results in C key (or cost) being updated to 4 since $5 < 5 + (-1)$ and C's parent is updated to B. The algorithm terminates with the shortest path to B being $p_1 =< A, B >$ and the shortest path to C being $p_2 =< A, B, C >$. If we *accept* that negative weights are well-defined, then p_1 is incorrect as the actual shortest path to B would then be $p_3 =< A, C, B >$. This is a contradiction to Dijkstra's algorithm and so the algorithm can not work with negative weights.

Question 8: All-Pairs-Shortest-Path with Negative Weights

The Problem

Given a weighted directed graph $G = (V, E)$, suppose that there are negative weights in G, but there is no negative cycle in G. Is all-pairs-shortest-path algorithm still correct? Try to prove your answer.

The Algorithm

A simple algorithm uses induction to design a direct solution for APSP.

–We know the shortest path between a set of k vertices (V_k)
–Want to add new vertex u

–Find shortest path from u to all vertices $\in V_k$:
shortest–path(u, w) = $min_{v \in V_k, (u,v) \in E}$(w(u, v) + shortest–path(v, w))

–Find shortest path to u from all the vertices $\in V_k$
shortest–path(w, u) = $min_{v \in V_k, (w,v) \in E}$(w(w, v) + shortest–path(v, u))

–Update all shortest–path(w_1, w_2) for w_1,w_2 $\in V_k$

shortest–path(w_1, w_2) = min(
shortest–path(w_1, u) + shortest–path(u, w_2),
shortest–path(w_1, w_2))

Proof

APSP algorithm is still correct when there are negative weights. To prove this we will need to consider 3 cases. The first case is when the new vertex u is reachable by some negative weighted edge $w(v_i, u) < 0$. The second case is when the new vertex u reaches some v_i along a negative weighted edge $w(u, v_i) < 0$. And the last case would be all edges in and out of u are negative $w(u, v_i) < 0$ *and* $w(v_i, u) < 0$ $\forall v_i \in V_k$.

Let S be the set storing the shortest-paths for all the vertices $v \in V_k$.
Let a path be denoted by $p(i, k) = (v_i, ..., v_k)$
Let $S(x, y) = p(x, y)$ deonte the shortest path between two vertices x, y $\in$ V_k.
Let $w(p)$ deonte the sum of the cost of edges along a path

Lemma 1: No positive weighted cycles are in S

A positive weighted cycle is the path $c =< p_i, p_{i+1}, ..., p_i >$, where $w(c) > 0$. Suppose there is a cycle in a path from x to y in S.
Then $S(x, y) = p(x, y) = (v_x, ..., v_{i-1}, v_i, ..., v_i, v_{i+1}, ..., v_y)$
This is a contradiction to the APSP since the cycle can be removed and the new path would be $p'(x, y) = (v_x, ..., v_{i-1}, v_i, v_{i-1}, ..., v_y)$ where $w(p') < w(p)$ and so there can be no positive weighted cycles in S.
Lemma 2: All negative edges must be unidirectional

Suppose $w(u, v) = w(v, u) < 0$ then this is a negative cycle since any path through v to any node reachable by v can find paths $p' = (w, v, u, v, x)$, $p'' = (w, v, u, v, u, v, u, v, x)$ where $w(p') > w(p'')$.

Case 1: New vertex u is only reachable by some negative edge from $v \in V_k$ and u can only reach other nodes along a positive edge.

The only paths effected by the negative edge, must pass through v. So any node w that can reach v will have $S(w, u) = (w, ..., v, u)$. This path is well defined since there is no negative cycle, and $S(w, v)$ is the shortest-path from w to v and $S(v, u)$ is the shortest-path from v to u. Any node w that has v along its path may be effected by this negative edge. Suppose $S(w_1, w_2) = (w_1, ..., v, ..., w_2)$. The new shortest-path between these two nodes may be replaced by $S(w_1, w_2) = S(w_1, v) + S(v, u) + S(u, w_2))$. We know from lemma 1 that this path will contain no positive cycles in it. We also know the graph will contain no negative cycles. Suppose the path $S(w_2, w_1) = (w_2, ..., w_1)$ also existed. If this path also went through v it will not be change since the negative edge only reaches u. However the cycle $p = (S(w_1, v), S(v, u), S(u, w_2), S(w_2, w_1))$ must have $w(p) > 0$ otherwise there is a negative cycle. Since there is no negative cycles, then the new path through u for $S(w_1, w_2)$ is correct and well defined.

Case 2: New vertex u reaches some $v \in V_k$ along a negative edge but is only reachable by positive edge.

From this, any path to u, from any node w will have $w(p(w, u)) > 0$ which are all well defined. The only paths effected by the new negative edge will contain the path $S(w_i, v) = (p(w_i, u), p(u, v))$. Suppose $S(w_i, v)$ was previously not reachable, but the path $S(v, w_i)$ existed. Then $w(p(w_i, v)) + w(p(u, v)) + w(p(v, w_i)) > 0$ or there is a negative cycle. Since there are no negative cycles then even in the smallest case of 3 vertices $S(w, v) = (p(w, u), p(u, v))$ is well defined since any previously existing paths between w and v must at minimum create a 0-cycle in the new shortest path including u. Thus any path that includes the path through u along a negative edge will be well-defined.

Case 3: All of the edges on u are negative

From case 1 and 2 we know that any path $S(w_1, u)$ and $S(u, w_2)$ are well defined for negative edges $\forall\ w_1, w_2 \in V_k$. Suppose the $S(w_1, w_2)$ is the path $p = (w_1, w_2)$. Since all of the edges on u are negative, the new $S(w_1, w_2) = (S(w_1, u), S(u, w_2))$ which would be the path $p' = (w_1, u, w_2)$. Since both $S(w_1, u)$ and $S(u, w_2)$ are well defined then this new path is also well-defined. Suppose that this new path is incorrect. Then $w(p) < w(p')$. Since both edges $w(w_1, u) < 0$ and $w(u, w_2) < 0$ then the edge $w(w_1, w_2) < w(w_1, u) + w(u, w_2) < 0$ is a contradiction to the graph not having negative cycles. Since the graph doesn't have negative cycles, then the shortest path $S(w_1, w_2) = p'$ and APSP algorithm worked correctly.

Therefore since all the paths that start or ending at u with a negative edge, and the paths containing u and only negative edges are well defined, then the APSP algorithm works and is correct for weighted directed graphs that contain negative edges but no negative cycles.

Question 9

Time Complexity

The main while loop runs one time for every vertex. The *delete_min*() operation runs in $O(\log |V|)$ since the heap property of the heap storing the vertices has changed. Similar to the analysis given above for the modified MST, the body of the for loop will in total be evaluated $|E|$ times once the entire outer loop has completed. The *decrease_key*() operation runs in $O(\log |V|)$. The remaining operations are all either $O(1)$ (*get_edges*() and *min_id*()). So the total complexity of the algorithm (not incl heap construction) is $O(|E| \log |V| + |V| \log |V|) = O((|E| + |V|) \log |V|)$