COMP251 Assignment 4

Adam Hooper
260055737

April 5, 2005

1. Problem 11-4: Universal hash and authentication

a. 2-universal → universal

If the family $\mathcal{H}$ of hash functions is 2-universal, that means each member $h \in \mathcal{H}$ maps the universe U of keys to $\{0, 1, \ldots, m-1\}$. For every pair $k, l \in U$ and for $h \in \mathcal{H}$ chosen at random, the sequence $\langle h(k), h(l) \rangle$ is equally likely to be any of the m^2 sequences of length 2 with elements drawn from $\{0, 1, \ldots, m-1\}$).

We want to prove that $\mathcal{H}$ is universal: that is, that the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}|/m$.

We know $|\mathcal{H}| = m^2$. We are given two distinct keys $k, l \in U$. Drawing a hash function $h \in \mathcal{H}$ at random, we will see that $h(k) = x$ for some x. Since $\langle h(k), h(l) \rangle$ is equally likely to be any of the m^2 sequences of length 2, but we know that $h(k) = x$, that means $\langle x, h(l) \rangle$ is equally likely to be any of the m sequences of length 2 starting with x. Thus the probability that $h(l) = x$ is exactly $1/m$. This means our 2-universal family of hash functions $\mathcal{H}$ is universal.

b. A 2-universal function

We are given a family of hash functions and asked to prove it is 2-universal. So we should write out the 2-key version of the family:

Let U be the set of pairs of values drawn from $\mathbb{Z}_p$, and let $B = \mathbb{Z}_p$, where p is prime. For any pair $a = \langle a_0, a_1 \rangle$ of values from $\mathbb{Z}_p$ and for any $b \in \mathbb{Z}_p$, define the hash function $h_{a,b} : U \to B$ on an input pair $x = \langle x_0, x_1 \rangle$ from U as $h_{a,b}(x) = (a_0 x_0 + a_1 x_1 + b) \bmod p$. $\mathcal{H} = \{h_{a,b}\}$.

There are a total of p^3 combinations of a_0, a_1 and b. We want to prove that if we choose at random from these p^3 combinations our $h_{a,b}$, the probability that $h_{a,b}(x_0) = h_{a,b}(x_1)$ is $1/p$.

Choose a fixed x_0 at random, and vary x_1. This will give us $h_{a,b}(x) = (a_0 x_0 + a_1 x_1 + b) \bmod p$, where $a_0 x_0$ is a constant. Let $c = a_0 x_0 + b \bmod p$. Since b is chosen at random, the value of c is equally likely to be any of the values in $\mathbb{Z}_p$. And so $h_{a,b}(x) = (a_1 x_1 + c) \bmod p$.

Now, we investigate what happens when we vary x_1, leaving c fixed. Since p is prime, a_1 does not divide into p if $a_1 \neq 0$. The value of $a_1 x_1$ will cycle over all possible values mod p in $\mathbb{Z}_p$ as x_1 takes on different values from $\mathbb{Z}_p$. So the probability of $(a_1 x_1 + c) \bmod p$ of being a particular value within $\mathbb{Z}_p$ is exactly $1/p$.

This all depends upon a_1 not being equal to 0. But if a_0 is fixed and $a_1 = 0$, there are still p possible hashing functions; each one will map x to a different value, and so the probability that $h_{a,b}(x_0) = h_{a,b}(x_1)$ is still exactly $1/p$.

c. Hashing a message

We are given a universal family $\mathcal{H}$ of hash functions. Each $h \in \mathcal{H}$ maps the universe of keys U to $\mathbb{Z}_p$. Given a message $m \in U$ and an authentication tag $t = h(m)$ for some $h \in \mathcal{H}$, we must argue that the probability of generating a different pair of m' and a corresponding t' without knowing h is at most $1/p$.

We can use the property that $\mathcal{H}$ is 2-universal to prove this problem. There are two hash functions in question: $h \in \mathcal{H}$, the hash function chosen by Alice and Bob, and $h' \in \mathcal{H}$, the hash function chosen by the attacker. There are two keys: m, the original key, and m', the attacker's key. The attacker only knows $h(m)$, and so wants to choose an h' which behaves similarly to h. But $\mathcal{H}$ is 2-universal: given a sequence of keys $\langle m, m' \rangle$, the attacker's chosen h' which maps m to $h(m)$ is equally likely to map m' to any value in $\mathbb{Z}_p$. In other words, there is a $1/p$ chance that the attacker chose the correct h' such that $h'(m') = h(m')$.

When Bob receives the value of m' and computes the value of $h(m')$, it thus has a $1/p$ chance of being the same as the value of $h'(m')$.

2. Exercise 16.3-2

We are given a set of frequencies. They can easily be put into a Huffman tree:

Letter	Frequency	Code
a	1	1111111
b	1	1111110
c	2	111110
d	3	11110
e	5	1110
f	8	110
g	13	10
h	21	0

The pattern is obvious; it arises from the following formula, where f_n is the n^{th} Fibonacci number:

$$\sum_{i=0}^{n} f_i = f_{n+2} - 1$$

The inductive proof is not too difficult. When $n = 0$, $0 = 1 - 1$. When $n = 1$, $1 = 2 - 1$. When $n > 1$, we have:

$$\begin{aligned}\sum_{i=0}^{n} f_i &= \sum_{i=0}^{n-1} f_i + f_n \\ &= f_{n+1} - 1 + f_n \\ &= f_{n+2} - 1\end{aligned}$$

From this identity, we can see that HUFFMAN will produce the pattern of 1's illustrated in the above table. After a and b are placed in the tree, subsequent calls to EXTRACT-MIN(Q) will always extract the next letter (corresponding to the next number in the Fibonacci sequence) and the working sub-tree (corresponding to $f_{n+2} - 1$, which is 1 less than the *next* number in the sequence).

The Huffman code can be defined backwards. The last letter in the sequence gets a code of 0. Then, each previous letter in the sequence (until the first) gets the code of its successor with a prefix of 1. The first letter in the sequence gets the code of its successor with 0 replaced by 1.

Problem 23-4: Alternative MST algorithms

a. MAYBE-MST-A

MAYBE-MST-A relies on a loop invariant analogous to the one in section 23.1: Prior to each iteration, T is a superset of some minimum spanning tree.

This is trivial to prove. At the beginning of the algorithm, T contains all edges and is thus a superset of any minimum spanning tree. And each time an edge e is removed from T, the remaining T is a superset of a minimum spanning tree, because the largest possible weight edge (e) was removed from it. In other words, if we were building a MST and were considering e for inclusion, we would have an alternative edge with a smaller or equal weight, and so e would not be needed in the MST.

So when the loop has finished iterating, T is a superset of a minimum spanning tree. But no edge can be removed from T without decreasing the number of nodes it spans (since we tried removing every edge); therefore, T *is* a minimum spanning tree. Therefore, MAYBE-MST-A does compute a minimum spanning tree.

The algorithm should be implemented as KRUSKAL is. Unfortunately, it is difficult to detect when removing an edge breaks the tree; the entire tree would need to be traversed each time to see if it missed any vertices. This would lead to a very slow algorithm: for each edge $e \in E$, it would need to check the remaining edges (an $O(|E|)$ operation) to see if they make a spanning tree. Thus, the most efficient algorithm would run in $O(|E|^2)$ time.

b. MAYBE-MST-B

This algorithm does not compute a minimum spanning tree. We can easily construct a graph which highlights its fault. Create a graph with vertices a, b and c, and weights $w(a, b) = 0$, $w(b, c) = 0$, $w(a, c) = 1$. The proper minimum spanning tree would be a tree which connects a to b and b to c (total weight 0); MAYBE-MST-B could connect a to c and a to b, which has weight 1.

We can use sets to implement this algorithm, as done in KRUSKAL. Here is the pseudo-code:

```
MAYBE-MST-B(G, w)
1  T ← ∅
2  for each vertex v ∈ V[G]
3        do MAKE-SET(v)
4  for each edge (u, v) ∈ E, taken in arbitrary order
5        do if FIND-SET(u) ≠ FIND-SET(v)
6              then T ← T ∪ {(u, v)}
7                   UNION(u, v)
8  return A
```

As with KRUSKAL, the running time is $O(|E| \lg |V|)$.

Maybe-MST-C

This algorithm does compute a minimum spanning tree. This can be proven by contradiction. Suppose the algorithm does not produce a minimum spanning tree. In that case, there is a path p from vertex u to vertex v which contains an edge (x, y) which is not part of a minimum spanning tree. But that is contradiction: if that were the case, MAYBE-MST-C would have either removed the edge or never included it in the first place. Therefore all edges in the tree are safe, and it is a minimum spanning tree.

As above, we can use sets to implement the algorithm:

```
MAYBE-MST-B(G, w)
 1  T ← ∅
 2  for each vertex v ∈ V[G]
 3        do MAKE-SET(v)
 4  for each edge (u, v) ∈ E, taken in arbitrary order
 5        do if FIND-SET(u) = FIND-SET(v)
 6              then c ← 1
 7              else  c ← 0
 8           T ← T ∪ {(u, v)}
 9           UNION(u, v)
10           if c = 1
11              then T ← T − {(u, v)}
12  return A
```

As above, the running time is $O(|E| \lg |V|)$; we don't perform any more operations here slower than $O(1)$.

4. Problem 21-2: Depth determination

a. Worst-case running time

The running time of MAKE-TREE is $\Theta(1)$: it just sets $p[v] = v$.

The running time of FIND-DEPTH on a tree of size m is in the worst case $\Theta(m)$: just as in a worst-case binary tree search, the worst case consists of a linked list of vertices of length m; FIND-DEPTH must iterate over them all.

The running time of GRAFT is $\Theta(1)$: it just sets $p[r] = v$.

The total worst-case running time consists of am calls to MAKE-TREE, bm calls to FIND-DEPTH, and cm calls to GRAFT, where $a + b + c = 1$. FIND-DEPTH will run in $\Theta(cm)$ time, and it will be called bm times. The total worst-case running time is therefore $\Theta(bcm^2) = \Theta(m^2)$ since b and c are constants.

b. MAKE-TREE

MAKE-TREE is exactly like MAKE-SET, except it adds a $d[v]$ to track the element's depth. Also, we need not concern ourselves with $rank[v]$.

```
MAKE-TREE(v)
1  p[v] ← v
2  d[v] ← 0
```

c. Find-Depth

```
FIND-DEPTH(v)
1  if x ≠ p[x]
2      then d[x] ← d[x] + FIND-DEPTH(p[x])
3           p[x] ← p[p[x]]
4  return d[x]
```

d. Graft

Graft looks like Union, but uses Find-Depth to flatten nodes instead of Find-Set:

```
GRAFT(r, x)
1  FIND-DEPTH(r)
2  FIND-DEPTH(x)
3  LINK(r, x)
```

Link looks a bit different from before. It need only increment the pseudo-depth of $p[r]$, which will have the effect of incrementing the depth of all children when Find-Depth is called on them (that is, the depth of all children of $p[r]$ will be incremented by $1 + d[x]$).

Also, Graft takes ordered parameters, and so Link uses the given order instead of using ranks. It does not assume that the given node r is the parent of its set, but it assumes that Find-Depth has been called on r (and thus $p[r]$ *is* the parent of its set).

```
LINK(r, x)
1  d[p[r]] ← d[p[r]] + d[x] + 1
2  p[p[r]] ← p[x]
```

e. Running time

We make n calls to Make-Tree and $m - n$ calls to both Graft and Find-Depth.

Make-Tree has running time $\Theta(1)$.

Graft has the same running time as a two calls to Find-Depth, since Link runs in $O(1)$ time.

Find-Depth is thus called as many as $2(m - n)$ times. Each time, the worst-case running time is, surprisingly, $\Theta(1)$. This is because it is impossible to construct a set with a height greater than 3. Any call to Graft will automatically call Find-Depth on both arguments; Find-Depth will ensure that its arguments (the sets) have depth of at most 2. Since no nodes may be added to any existing sets without calling Find-Depth on their parents (guaranteeing a set depth of at most 2), and since all new sets have height 1, it is impossible to construct a set with a height greater than 3. Therefore, Find-Depth's worst-case running time is a constant.

We have a total of n calls to MAKE-TREE ($\Theta(1)$), at most $m-n$ calls to GRAFT, and (through GRAFT) at most $2(m-n)$ calls to FIND-DEPTH. All these procedures run in $\Theta(1)$ time; the total worst-case running time is thus $\Theta(n+m-n+2(m-n)) = \Theta(3m-2n) = \Theta(m)$.

5. Problem 24-5: Karp's minimum mean-weight cycle algorithm

a. $\mu^* = 0$

If $\mu^* = 0$, then $\min_c \mu(c) = 0$. This means there are no negative-weight cycles; a simple proof by contradiction will suffice. Suppose there is a negative-weight cycle in G. Then there exists a cycle $c = \langle e_1, e_2, \ldots, e_k \rangle$ of edges in E for which $\sum_{i=1}^{k} w(e_i) < 0$, and so $\mu(c) = \frac{1}{k}\sum_{i=1}^{k} w(e_i) < 0$. Since we assumed $\min_c \mu(c) = 0$, and we have here a cycle c with $\mu(c) < 0$, we have a contradiction. Therefore, if $\mu^* = 0$, the graph G contains no negative-weight cycles.

We are also asked to prove that $\delta(s,v) = min_{0 \le k \le n-1}\, \delta_k(s,v)$ for all vertices $v \in V$. This is much more obvious when translated to English: it says that the shortest path from s to v is the shortest of all shortest paths from s to v of different lengths. The only possible hitch would be that a path $\delta_j(s,v)$ existed, with $j \ge n$, such that $\delta_j(s,v) < \delta(s,v)$. But that is impossible, since G contains no negative-weight cycles.

b.

We are asked to prove that:

$$\max_{0 \le k \le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} \quad \ge \quad 0$$

From the above proofs, we can easily see that $\delta_n(s,v) > \delta_k(s,v)$ for at least one value of k: at some value of k, $\delta_k(s,v) = \delta(s,v) \le \delta_n(s,v)$. And since $n-k$ is positive, the resulting fraction is also positive for some value of k. Therefore the maximum over $0 \le k \le n-1$ is greater than or equal to 0.

c. 0-weight cycle

We are presented with a 0-weight cycle c which contains vertices u and v. We are told that $\mu^* = 0$; that means that there are no negative-weight cycles. We are given that the weight along the cycle from u to v is x. We must prove that $\delta(s,v) = \delta(s,u) + x$. In other words, a shortest path from s to v consists of the shortest path from s to u followed by the part of the cycle c travelling from u to v.

We can prove this by proving an upper bound and a lower bound. $\delta(s,v) \le \delta(s,u) + x$, obviously, since there exists a path from s to u and there exists a path from u to v, so there

definitely exists a path from s to v through u with a certain weight. $\delta(s, v)$ is less than or equal to that weight.

Slightly less trivially, we can show that $\delta(s, v) \geq \delta(s, u) + x$. This is because $\delta(s, u) \leq \delta(s, v) - x$ (i.e., take a path from s to v and then from v to u along c; the path from v to u along c has weight $-x$ since the entire cycle c has weight 0 from v to u and back). We can simply rearrange terms:

$$\begin{aligned} \delta(s, u) &\leq \delta(s, v) - x \\ \delta(s, u) + x &\leq \delta(s, v) \\ \delta(s, v) &\geq \delta(s, u) + x \end{aligned}$$

And so, since $\delta(s, v) \geq \delta(s, u) + x$ and $\delta(s, v) \leq \delta(s, u) + x$, $\delta(s, v) = \delta(s, u) + x$.

d. Shortest path on a minimum mean-weight cycle

In this problem, we are told that $\mu^* = 0$; thus, each minimum mean-weight cycle is a 0-weight cycle.

We are asked to prove, given v on a minimum mean-weight cycle:

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0$$

Using the previous proof, we know that $\delta(s, v) = \delta(s, u) + x$, where u is another node along the cycle and x is the weight along the cycle from u to v. We also know from part b that $\delta(s, v) = \delta_k(s, v)$ for some k, $0 \leq k \leq n - 1$. So if we define u such that the distance from u to v along the 0-weight cycle is $n - k$, we have a path from s to v through u and along the cycle; the path's length is n. (If $n - k$ is greater than the length of the cycle, we can simply choose u such that the distance from u to v is $n - k \bmod l$, where l is the total length of the cycle.) So for any n, we can construct a shortest path: a path with weight $\delta(s, v)$.

Since $\delta_n(s, v) = \delta(s, v)$ (i.e., any path from s to v of length n has the same weight as a minimum path), and since $\delta_k(s, v) \geq \delta(s, v)$, the given fraction will be negative or zero, depending on the value of k. Since for some k, $0 \leq k \leq n - 1$, $\delta_k(s, v) = \delta(s, v)$, the fraction will be 0 for some value of k. Therefore, the maximum value of the fraction will be 0.

e.

We are again asked to prove the fraction above, but over all vertices.

$$\min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0$$

We know already that at least two v's exist on a minimum mean-weight cycle (since $\mu^* = 0$), and so there exists some v for which the maximum fraction equals 0. We must now prove that it is a lower bound: that is, that there exists no v for which the maximum is less than 0.

We know that for some k, $0 \le k \le n-1$, $\delta_k(s,v) = \delta(s,v)$. And we know that, since $\mu^* = 0$, there are no negative-weight cycles on G. That means that $\delta_n(s,v) \ge \delta(s,v)$, by the definition of $\delta(s,v)$. Since $\delta_n(s,v)$ is constant, the greatest term in the maximum is $\frac{\delta_n(s,v)-\delta(s,v)}{n-k}$ (for some k, $0 \le k \le n-1$), which must be positive or zero.

It should be obvious at this point that the statement is true.

f. Add t to the weight of each edge

Start with $\mu^* = 0$, and add t to the weight of each edge on G. The original sum,

$$\mu(c) = \frac{1}{k} \sum_{i=1}^{k} w(e_i)$$

becomes:

$$\begin{aligned} \mu(c) &= \frac{1}{k} \sum_{i=1}^{k} (w_0(e_i) + t) \\ &= \frac{1}{k} \left(kt + \sum_{i=1}^{k} w_0(e_i) \right) \\ &= t + \frac{1}{k} \sum_{i=1}^{k} w_0(e_i) \end{aligned}$$

where $w(e_i) = w_0(e_i) + t$, and w_0 is a weighting function such that $\mu^* = 0$.

We know that $\mu^* = \min_c(\mu(c))$, where c ranges over all cycles in G. If, with weighting function w_0, $\mu^* = 0$, then $\min_c(\mu(c)) = 0$ for some cycle c in G and $\mu(c) \ge 0$ for all others. With weighting function w, such that $w(e) = w_0(e) + t$, $\mu(c)$ increases by t, for all cycles. So $\mu^* = \min_c(\mu(c)) = t$.

Now, for any vertex v on a minimum mean-weight cycle, we can show that

$$\max_{0 \le k \le n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = t$$

This follows from the proof in part d. $\delta_n(s, v)$ will consist of $\delta_r(s, u)$ to reach the minimum mean-weight cycle (for some integer r), and then s edges around the cycle (plus, potentially, more iterations *around* the cycle). When transitioning from the $\mu^* = 0$ condition, we must increase $\delta_r(s, u)$ by the number rt, since each edge has increased in weight by t and r edges have been traversed. Also, the path of s edges around the minimum mean-weight cycle will increase in weight by st. Finally, any traversals around the minimum mean-weight cycle will give an increase in weight of lt, l being the number of edges in the cycle. So in total, $\delta_n(s, v) = \delta_{n_0}(s, v) + t(r + s + xl)$, x being the number of loops around the minimum mean-weight cycle. However, by the same logic, $\delta_k(s, v) = \delta_{k_0}(s, v) + kt$. So we can set up and prove a simple equality which will make our desired maximum fraction equal to t:

$$r + s + xl = (n - k) + k$$

And since we defined r, s, xl in such a way that their sum is n, this is obvious.

g. An algorithm to compute μ^*

This algorithm will construct a vector of δ_k's to each vertex, and then use the equation given above to find μ^*.

FIND-MIN-MEAN-WEIGHT(G, w)

```
1  n ← |V[G]|
2  s ← any vertex from V[G]
3  for k ← 0 to n
4        do for v in V[G]
5              do compute δ_{v,k}, relying on previous values of δ
6  for v in V[G]
7        do m_v ← max((δ_{v,n} − δ_{v,k}) / (n − k)) for all k
8  return min(m_v)
```

This algorithm is more to give an idea than to implement in a real programming language. However, running-time analysis may still be performed. The first **for** loop iterates $|V|$ times; in each iteration, as many as $|E|$ edges will be traversed to calculate the new k iteration of $\delta_{v,k}$. The running time of the δ computation is therefore $O(|V||E|)$. The next loop, to calculate the maximum, runs an $O(|V|)$ operation $|V|$ times, and therefore has running time $O(|V|^2)$. The final part, the **return**, simply iterates once per vertex, and thus has a running time of $O(|V|)$. Putting it all together, we get a running time of $O(|V||E|)$, since $|V| \leq |E|$.