Advanced Algorithms Problems and Solutions

Mattia Setzu Giorgio Vinciguerra

October 2016

Contents

1 Hogwarts

The Hogwarts School[1] is modeled as a graph $G = (V, E)$ where V is the set of castle's rooms and $E \subseteq V \times V$ is the set of the stairs. Each stair is labelled with the time of appearance and disappearance, and can be walked in both directions, therefore the graph is undirected. The goal is to find, if possible, the minimum amount of time required to go from the first to the last room.

1.1 Solution 1: Preprocessing-then-Dijkstra

Dijkstra is able to find the shortest path in a graph with non-negative weights on its edges. Our main idea is to create a Dijkstra compatible graph through a NORMALIZE function, then apply Dijkstra to it in order to find the shortest path. The core of the preprocessing is the NORMALIZE function which computes traversal times between nodes at a given time *time*:

```
**function** NORMALIZE($from$, $to$, $time$):
    $t \leftarrow \infty$
    **if** $start[v'] \leq t < end[v']$ **then**                ▷ No waiting time
        $t \leftarrow t + 1$
    **else if** $t < start[v']$ **then**                         ▷ Waiting time
        $t \leftarrow start[v'] + 1$
    **else**
        $t \leftarrow \infty$                          ▷ Available time already expired
    **return** $t$
```

The normalize function is then applied to a node traversal:

1.1.1 Pseudo-code

```
create vertex set $Q$ of unvisited nodes
create vertexes set $E'$ of edges weight
$time \leftarrow 0$                                        ▷ Initial time for traversal
$edges \leftarrow$ STAIRS_OF(0)    ▷ Get incoming/outgoing edges of the source node
**function** PROCESS($node$, $time$)
    **if** $edge \in visited\_edges$ **then**
        **return**
    $traversal\_time \leftarrow \infty$
    **for all** $neighbor \in neighbors\_of\_node$ **do**
        $traversal\_time \leftarrow$ TRAVERSAL_TIME($node$, $neighbor$, $time$)
        $E'[0][node] \leftarrow traversal\_time$     ▷ $E'[i][j]$ holds the weight/traversal
                                              ▷ time for the stair between $i$ and $j$
        **for all** $new\_neighbor \in neighbors\_of\_neighbor$ **do**
            NORMALIZE($neighbor$, $new\_neighbor$, $traversal\_time$)
    **if** DIJKSTRA$(V, E') = \infty$ **then**
```

[1] http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistraleinformatica/alg2/algo2_16/hogwarts.pdf

```
        return -1
    else
        return $t$
```

Computational cost: $\Theta(n^2)$ if the vertex set in DIJKSTRA is implemented as an array. $O(|E| + |V| \log |V|)$ with Fibonacci heap.

1.2 Solution 2: HogwartsDijkstra

```
function HOGWARTSDIJKSTRA($G$):
    create vertex set $Q$ of unvisited nodes
    for all vertex $v \in V$ do                          ▷ initialization
        $time[v] \leftarrow \infty$                      ▷ unknown time from source to v
        add $v$ to $Q$                                   ▷ all nodes initially in Q
    $time[0] \leftarrow 0$                               ▷ time from source to source
    while $Q \neq \emptyset$ do
        $u \leftarrow x \in Q$ with $\min\{time[x]\}$
        remove $u$ from $Q$
        for all neighbor $v$ of $u$ do:
            if $time[u] \leq appear[v]$ then
                $alt \leftarrow appear[v] + 1$           ▷ wait the appearance of the stair
            else if $time[u] < disappear[v]$ then
                $alt \leftarrow time[u] + 1$             ▷ use the stair
            else
                $alt \leftarrow \infty$                  ▷ the stair has already disappeared
            if $alt < time[v]$ then
                $time[v] \leftarrow alt$                 ▷ a quicker path to $v$ has been found
    return $time[|N| - 1]$
```

Computational cost. See the previous section.

1.3 Solution 3: BFS-like traversal

```
function REACH($N$, $M$, $A[]$, $B[]$, $appear[]$, $disappear[]$)
    for $i = 0$ to $M - 1$ do
        $edges\_[A[i]].push\_back(make\_pair(i, B[i]))$
        $edges\_[B[i]].push\_back(make\_pair(i, A[i]))$
    for $i = 0$ to $N - 1$ do
        $done\_[i] \leftarrow false$
        $distance\_[i] \leftarrow \infty$
    $reached\_[0].push\_back(0)$
    $distance\_[0] \leftarrow 0$
    for $t = 0$ to $MAX\_TIME$ do
        for all $v \in reached\_[t]$ do
            if not $done\_[v]$ then
                for all $edge \in edges\_[v]$ do
                    $staircase \leftarrow edge.first$
                    $neighbor \leftarrow edge.second$
                    $time \leftarrow \max(distance_{[}v], appear[staircase]) + 1$
                    if not $done\_[neighbor]$
                                          and $distance\_[v] < disappear[staircase]$
                                            and $time < distance\_[neighbor]$ then
                        $distance\_[neighbor] \leftarrow time$
                        $reached\_[time].push\_back(neighbor)$
                $done\_[v] \leftarrow true$
    return $(distance\_[N-1] = \infty)? - 1 : distance\_[N-1]$
```

Computational cost: $O(m + MAX\_TIME)$.

2 Paletta

Paletta ordering[2] is a peculiar ordering technique: given a 3-tuple of elements, paletta takes the central element as pivot and swaps the two elements right before and next to it. To make an example:

$$(3, 2, 1) \xrightarrow{paletta} (1, 2, 3)$$

We now want to develop an algorithm to order any array through paletta ordering with the minimal number of swaps. You should see as not every array can be ordered (e.g. [1, 3, 2]).

2.1 Solution 1: Split and count-inversions

We should note that the following properties hold:

1. Every element can be a pivot, but the first and the last one, as they have respectively no elements before and after them.

2. Every element can be swapped as many times as necessary, but only with elements of the same 2-remainder (numbers in even positions can only be swapped with numbers in even positions, the same holds for odd indexes). More formally, if n is the size of the array A we want to sort, $i, j \in [1, n-2]$, $A[i]$ can be swapped with $A[j]$ if and only if $i \equiv j \pmod 2$.

[2] `http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistraleinformatica/alg2/algo2_16/paletta.pdf`

3. The least number of swaps does not backtrack any element. Formally, let k be the minimal number of swaps applied to an array, backtracks included. By hypothesis, k is minimal, but at least m, $m > 0$ backtrack swaps have been operated, therefore we found a $k' = k - m : k' < k$, a new minimal number of swaps: contradiction.

Given item 2, we can split our array in two, even and odd numbers, and order them counting the swaps. In our example we'll use *mergesort*, as it runs in $O(n \log n)$, does backtrack elements, and is very well-known. Clearly, given an array, a swap happens when an element is pushed back, pulling the one between its new position and the old one ahead: we can map this behaviour in the merge routine of mergesort: the array merged is able to push back elements from its right pointer to the new array, moving them back of $(m-i)+(j-m)$ positions, where m is the dimension of the current two sub-arrays to merge. Provided that our edited version of mergesort ran successfully on both the even-index and odd-index, we now need to verify if by merging them we obtain an ordered array. Intuitively, the merged array will start with the first element of the even-index arrays, followed by the first of the odd-index array, followed by the second of the even-index array, and so on. To check for these elements is pretty trivial and can be done in linear time. Follows the pseudo-code for the edited version and SNAKE_CHECK function:

```
function MERGE_WITH_PALETTA($left$, $right$, $k$):
    ...                                                  ▷ merge instructions
    if $right > left$ then
        $paletta\_count \leftarrow paletta\_count + 1$
        ...
```

```
function SNAKE_CHECK
    $even, odd \leftarrow 0$
    for ;$even, odd < N; even = even + 1, odd = odd + 1$ do
        if $a[even] > a[odd]$ then
            return $-1$
    return $paletta\_count$
```

Computational cost: $\Theta(n \log n)$.

3 Range updates

Consider an array C of n integers, initially all equal to zero. We want to support the following operations:

- **update(i, j, c):** where $0 \leq i \leq j \leq n-1$ and c is an integer: it changes C such that $C[k] = C[k] + c$ for every $i \leq k \leq j$.
- **query(i)** where $0 \leq i \leq n-1$: it returns the value of $C[i]$.
- **sum(i,j)** where $0 \leq i \leq j \leq n-1$: it returns $\Sigma_{k=1}^{j}(C[k])$.

Design a data structure that uses $O(n)$ space and implements each operation above in $O(\log(n))$ time. Note that $query(i) = sum(i,i)$ but it helps to reason. [Hint to further save space: use an implicit tree such as the Fenwick tree (see wikipedia).]

3.1 Solution 1: Fenwick lazy a-b sums

Let T be a segmented binary tree over a continuous interval $I : [0, N-1]$ s.t. its leafs are the points in I, and the parent of two nodes comprises of their interval:

$$n' \cup n'' = n, n' \cap n'' = \emptyset \text{ s.t. } n \text{ is the parent of } n', n''$$

T will keep track of the prefix sums for every interval. We define a function

$$s' : [0, n-1] \to \mathbb{N} \tag{1}$$

that given a node in T returns the value associated with I, namely the cumulative sum of that interval.

In order to reduce the computational cost, we introduce a lazy algorithm that doesn't propagate sums over T as they are streamed in the input, which means $s'(i)$ might not be accurate at a given time t for any of the requested operation.

We'll instead either compute over T or update T as necessary. Let us define a function to do so:

$$l : \mathbb{N} \to (\mathbb{N} \cup \{\epsilon\}, \mathbb{N}) \tag{2}$$

to keep track of our lazy sums:

$$s(n) = \begin{cases} \epsilon, \_ & \text{if no lazy prefix sum is in that interval} \\ k, m & \text{if a lazy sum of k is to be propagated to m} \end{cases}$$

The QUERY function is then trivial:

```
function QUERY($I$, $i$, $sum$):
    if $I.size = 1$ then                                  ▷ Return found value
        return $I.sum$
    if lazy(I), $i \in I.left, i \notin I.right$ then     ▷ Lazy on left child
        $lazy(I) \leftarrow False$
        QUERY($I.left$, $i$, $sum + I.sum$)
    if lazy(I), $i \in I.right, i \notin I.left$ then        ▷ Lazy on right child
        $lazy(I) \leftarrow False$
        QUERY($I.right$, $i$, $sum + I.sum$)
    if lazy(I), $i \in I.right, i \in I.left$ then        ▷ Lazy on both
        $lazy(I) \leftarrow False$
        QUERY($I.right$, $i$, $j$, $sum + I.sum$) + QUERY($I.left$, $i$, $j$, $sum + I.sum$)
    if !lazy(I), $i \in I.left$ then        ▷ Not lazy on left child
        QUERY($I.left$, $i$, $sum$)
    if !lazy(I), $i \in I.right$ then        ▷ Not lazy on right child
        QUERY($I.right$, $i$, $sum$)
    if !lazy(I), $i \in I.right, i \in I.left$ then        ▷ Not lazy on both
        SUM($I.right$, $i$, $sum$)
```

```
function SUM($I$, $i$, $j$, $sum$):
    if $I.size = 1$ then        ▷ Return found value
        return $I.sum + sum$
    if lazy(I), $i \in I.left, i \notin I.right$ then        ▷ Lazy on left child
        $lazy(I) \leftarrow False$
        SUM($I.left$, $i$, $j$, $sum + I.sum$)
    if lazy(I), $i \in I.right, i \notin I.left$ then        ▷ Lazy on right child
        $lazy(I) \leftarrow False$
        SUM($I.right$, $i$, $j$, $sum + I.sum$)
    if lazy(I), $i \in I.right, i \in I.left$ then        ▷ Lazy on both
        $lazy(I) \leftarrow False$
        SUM($I.right$, $i$, $j$, $sum + I.sum$) + SUM($I.left$, $i$, $j$, $sum + I.sum$)
    if !lazy(I), $i \in I.left$ then        ▷ Not lazy on left child
        SUM($I.left$, $i$, $sum$)
    if !lazy(I), $i \in I.right$ then        ▷ Not lazy on right child
        SUM($I.right$, $i$, $sum$)
    if !lazy(I), $i \in I.right, i \in I.left$ then        ▷ Not lazy on both
        SUM($I.right$, $i$, $sum$)
```

```
function UPDATE($I$, $i$, $j$, $k$):
    if $I.size = 1$ then        ▷ Return found value
        return $I.val \leftarrow I.val + update$
    if lazy(I), $i \in I.left, i \notin I.right$ then        ▷ Lazy on left child
        $lazy(I.left) \leftarrow True$
        $I.left.val \leftarrow k$
    if lazy(I), $i \in I.right, i \notin I.left$ then        ▷ Lazy on right child
        $lazy(I.right) \leftarrow True$
        $I.right.val \leftarrow k$
    if lazy(I), $i \in I.right, i \in I.left$ then        ▷ Lazy on both
```

```
        $lazy(I) \leftarrow True$
        $I.val \leftarrow k$
    if !lazy(I), $i \in I.left$ then                          ▷ Not lazy on left child
        UPDATE($I.left$, $i$, $update$)
    if !lazy(I), $i \in I.right$ then                         ▷ Not lazy on right child
        UPDATE($I.right$, $i$, $update$)

    if !lazy(I), $i \in I.right, i \in I.left$ then                  ▷ Not lazy on both
        UPDATE($I.right$, $i$, $update$)
```

4 Karp-Rabin

Given a string $S : |S| = n$, and two positions $0 \leq i < j \leq (n - 1)$, the longest common extension $lceS(i, j)$ is the length of the maximal run of matching characters from those positions, namely: if $S[i]6 = S[j]$ then $lceS(i, j) = 0$; otherwise, $lceS(i, j) = \max l \geq 1 : S[i...i + l - 1] = S[j...j + i - 1]$. For example, if S = abracadabra, then $lceS(1, 2) = 0$, $lceS(0, 3) = 1$, and $lceS(0, 7) = 4$. Given S in advance for preprocessing, build a data structure for S based on the Karp-Rabin fingerprinting, in $O(n \ln(n))$ time, so that it supports subsequent online queries of the following two types:

- $lceS(i, j)$: it computes the longest common extension at positions i and j in O(log n) time.
- $equals(i, j, c)$: it checks if $S[i...i + ‘ - 1] = S[j...j + ‘ - 1]$ in constant time.

Analyze the cost and the error probability. The space occupied by the data structure can be $O(n \log(n))$ but it is possible to use $O(n)$ space. [Note: in this exercise, a onetime preprocessing is performed, and then many online queries are to be answered on the fly.]

4.1 Solution 1: Cumulative shift

4.1.1 Construction

In order to save computational cycles on checks over ranges we use a similar structure to the one in the range updates: we compute the hashing on the first character in $O(1)$ time, then roll the hash through the $n - 1$ remaining characters through $nO(1)$ operations. We call H this array; we also denote h_k as the function ca^i computating the Rabin-Karph hash of a string s. The reader shall now see that $\exists h^{-1}(s)$: that is, h is invertible in $O(1)$. The entries $h[i] = \sum_{i \in [0, n-1]}(h(i))$ have cumulative hash and the following properties hold:

- $h[s[i]] = (h[i] - h[i - 1])/a^{-1}, a^{-1} = a^1$
- $h[i..j] - h[k..l] = (h[l] - h[k - 1])/a^{-1} - (h[j] - h[i - 1])/a^{-1}, a^{-1} =$ modular inverse

4.1.2 equals(i, j, l)

EQUALS works on cumulative hashes, subtracting them and scaling them accordingly, as our *rabin* function multiplies by an a^i costant.

```
function EQUALS($i$, $j$, $length$):
    $h_i = h[i + length] - h[i - 1]$
    $h_j = h[j + length] - h[j - 1]$
    $h^i = h_i / inv(a, i, l)$
    $h^j = h_j / inv(a, j, l)$
     return $h^i - h^j == 0$
function INV($h$, $k$, $l$): return $h^{k-l}$
```

4.1.3 lce(i, j)

LCE works on cumulative hashes, checks the equality on the middle element of the strings and runs recursively on the half with different hashing. We define LCE as an auxiliary function

```
function LCE($i$, $j$, $l$):
    eq = EQUALS($i$, $j$)
    if eq then return $l$
    else if ¬ EQUALS($(j-i)/2$, $(n-j)/2$, $l$) then return EQUALS($(j-i)/2$,
$(n-j)/2$)
    else$(j-i)/2+$ return EQUALS($(j-i)/2$, $(n-j)/2$)
    $h_i = h[i + length] - h[i-1]$
    $h_j = h[j + length] - h[j-1]$
    $h^i = h_i/inv(a, i, l)$
    $h^j = h_j/inv(a, j, l)$
      return $h^i - h^j == 0$
function INV($h$, $k$, $l$): return $h^{k-l}$
```

5 Depth of a node in a random search tree

A random search tree for a set S can be defined as follows: if S is empty, then the null tree is a random search tree; otherwise, choose uniformly at random a key $k \in S$: the random search tree is obtained by picking k as root, and the random search trees on $L = x \in S : x < kandR = x \in S : x > k$ become, respectively, the left and right subtree of the root k. Consider the randomized QuickSort discussed in class and analyzed with indicator variables CLRS 7.3, and observe that the random selection of the pivots follows the above process, thus producing a random search tree of n nodes. Using a variation of the analysis with indicator variables, prove that the expected depth of a node (i.e. the random variable representing the distance of the node from the root) is nearly $2\log_2(n)$. Prove that the probability that the expected depth of a node exceeds $2\log_2(n)$ is small for any given constant $c > 1$. [Note: the latter point can be solved after we see Chernoff's bounds.]

5.1 Recursive balanced proof

Let n be the number of nodes in the input list l, $h = \log_2(n)$ the height of a balanced tree over l, $T(p)$ the tree built over the permutation p of pivots, $d(m)$ be the positional distance of a value m of a partition from the median value of the said partition. Then the following holds:

- $height(T) = h \iff |T.left| = |T.right| \pm 1$ Trivially, let r be the root of a 3-nodes partition: then, if the partition is unbalanced, the lesser one will comprise of 0 nodes, while the greater one of 2, which implies that $height(T.right) == 2$.
- P = pivot, $d(m) = \pm k \implies height(T.left) = height(T.right) \pm k$. Recursively from the previous statement, a partition unbalanced of one element generates subtrees whose levels differ on a factor of 1. By iterating recursively, their subtrees, if unbalanced by 1, will yield one more level difference. Over k unbalanced pivots on a single subtree, at most k levels will be added to h.
- By the previous statement, it follows that $\nexists T, T' : height(T) >= height(T')$, T balanced, T' unbalanced. As stated, let T', T be the unbalanced/balanced tree respectively; let us cheat with T and switch the root pivot with the first element in its subtree. Now, let us prove by contradiction that T can't stay balanced and that its height will increase. By shifting the tree to the left we have deprived $T.right$ of either 0 levels (in case $T.right$ is able to switch every pivot in its tree with its right subtree root, ending with the rightmost leaf in its subtree) or 1, in case no rightmost leaf is present. Therefore $height(T) <= height(T')$.
- The completely unbalanced tree is the tree with the most levels. By taking partitions of size 0 we costantly force, at each level, one subtree to

disappear. Therefore, its level(s) has to be necessarily transferred to its brother. We then have exactly one node per level, therefore n levels.

Behaviour on random permutations Now let us analyse how the tree depth varies according to random pivot selection. We start by applying the 5.1k-distance to a tree T with $n = 3$ nodes. Trivially, $height(T)$ with balanced tree is equal to two. Now, let us pick either the lowest or the greatest pivot possible: the tree is unbalanced towards either the left or the right, but $height(T) = 2$ in both cases. As the reader can see from 5.1, the distance works in absolute value; it is then clear how, at every permutation for a pivot p, out of the n, there are 2 that generate a tree of the same height: $p = d(P) + k, p = d(P) - k$. Given that at every iteration a node x in a completely unbalanced tree T’ has a probability of $\frac{1}{n-i}$, we can define the probability of x being a pivot at level l as:

$$P(x_k) = \frac{1}{n - l} \tag{3}$$

Now, in order for x not to be chosen as pivot in the previous $l - 1$ levels we have:

$$P(x_k) = \Sigma_{k=1}^{l-1}(\frac{1}{n - l + 1}) \tag{4}$$

Given the height of T, the (harmonic) partial series converges to $\ln(n) + 1$. Let us now add a root r s.t. $T'.right = T, T'.left = T$. We now have to consider the mirror case $\ln(n') + \ln(n')$, given by the previous $n' = n/2$ in the logarithm, since the number of nodes doubled, the $+1$ removed for both, since now neither of $T'.left, T.right$ is the root, and a $+1$ added since a new level has been added.

5.2 Upper bound

By hypothesis,

$$E[d(x) > 2c\ln(n)] <<< 1 \tag{5}$$

By definition the ancestor of a node i are indipendent random variables, and we can apply the Chernoff bounds over the set $x : d(x) >= 2\ln(n)$ of random variables determining the expected distance of nodes.

$$\mathsf{P}[X \geq c\mathsf{E}[X]] < e^{-c\ln(\frac{c}{e})\mathsf{E}[X]}$$

Let us consider $X = 1 \forall i == \ln(n)$, the expected depth of $\ln(n)$, then

$$\mathsf{P}[X \geq c\ln(n)] < e^{-c\ln(\frac{c}{e})\ln(n)}$$

6 Hashing sets

Your company has a database $S \subseteq U$ of keys. For this database, it uses a randomly chosen hash function h from a universal family H (as seen in class); it also keeps a bit vector B_S of m entries, initialized to zeroes, which are then set $B_S[h(k)] = 1 \forall k \in S$ (note that collisions may happen). Unfortunately, the database has been lost, thus only BS and h are known, and the rest is no more accessible. Now, given $k \in U$, how can you establish if k was in S or not? What is the probability of error? (Optional: can you estimate the size $|S|$ of S looking at h and B_S and what is the probability of error?) Later, another database R has been found to be lost: it was using the same hash function h, and the bit vector BR defined analogously as above. Using h, B_S, B_R, how can you establish if k was in $S \cap R$ (union), $S \cup R$ (intersection), or S
R (difference)? What is the probability of error?

6.1 $k \in S$

Let $k \in [0, m-1] : B_s[k] = 1 \implies h(k) \in S$, but not conclusive, as $\exists h \in H_{a,b}, l \in U : h(l) = k$. Out of the $|H_{a,b}| = (p)(p-1)$ functions $\in H_{a,b}$ the probability

$$\mathsf{P}(h_{a_i,b_i}(l) = h_{a_0,b_0}(k)) \tag{6}$$

6.2 $|S|$

Since h is bijective, and we know h, we can identify a lower bound, but are not able to estimate correctly the exact number. Trivially, we count the insertions in B_S with a $\phi(B_S)$, and we'll have $|S| \geq \phi(B_S)$. Since we have no access to the collision list, we dont know how many collisions there are on every B_S entry to 1.

6.3 Set operations

- **Union** Trivially proven by 6.1 applied to R, S. If any of them $\in [m]$, then $k \in S \cup R$.
- **Intersection** Trivially proven by 6.1 applied to R, S. If both $\in [m]$, then $k \in S \cap R$.
- **Difference** Trivially proven by complement on union.

7 Family of uniform hash functions

The notion of pairwise independence says that, for any $x_1 \neq x_2, c_1, c_2 \in \mathbb{Z}_p$, we have that

$$\mathsf{P}(h(x_1 = c_1), h(x_2 = c_2)) = \mathsf{P}(h(x_1 = c_1)) * \mathsf{P}(h(x_2 = c_2)) \tag{7}$$

In other words, the joint probability is the product of the two individual probabilities. Show that the family of hash functions $H = h_{a,b}(x) = ((ax + b) \mod p) \mod m : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p^*$ is *pairwise dependent* where p is a sufficently large prime number $(m + 1 \leq p \leq 2m)$.

7.1 Equal probability

By linear algebra, $ak \mod p = c \forall k \in \mathbb{N}$. If we were to cap $ak \mod p = c \forall k \in K, K_N = k_i : k_i < N$

$$\mathsf{P}(h(x_i = c_i)) = \frac{1}{m^2} = \mathsf{P}(h(x_1 = c_1), h(x_2 = c_2)) \tag{8}$$

Given x_i we define as $m_i = (ax_i + b) \mod p$; since $(ax_i + b)$ is a *linear transformation* m_i is unique. It follows trivially that $\mathsf{P}((ax_i + b) = d) = \frac{1}{p}$. Now, the same goes for x_1, x_2:

$$(ax_1 + b) \equiv d \tag{9}$$

$$(ax_2 + b) \equiv e \tag{10}$$

By the Chinese reminder theorem the above system has only one solution, therefore $\mathsf{P}((ax_1 + b) = d) = \frac{1}{p}$ and e become independent of d. Having

$$\mathsf{P}(h(x_1 = c_1), h(x_2 = c_2)) = \mathsf{P}(h(x_1 = c_1)) * \mathsf{P}(h(x_2 = c_2)) = \frac{1}{p} * \frac{1}{p} = \frac{1}{p^2} \tag{11}$$

Which, since $m + 1 \leq p \leq 2m$, proves the assumption.