6.854 **Dan Ports**

Advanced Algorithms drkp@mit.edu

2004/09/24 Collab: {sarahl,bhebert,pramook}

Problem Set 3

P1: 7.5 P2: 10 P4: 10 P3: 10 P5: 10 47.5

Problem 1:

a) To identify the number of occurrences of a m-length pattern in a text T, we will first preprocess the text T to generate all possible substrings of length m, compute their fingerprints, and store the number of substrings per hash value in the table. We begin by creating a hash table of size p, where p is a prime whose size will be discussed below, and initialize every element of the array to zero. We use Rabin-Karp fingerprinting: we assume without loss of generality that the string is composed of the alphabet $\{0, 1\}$, and define the fingerprint of a substring as the number represented in binary by the substring, modulo a prime p. To compute the fingerprints of every substring in $O(|T|)$ time, we use the Rabin-Karp technique of using a sliding window, at each stage subtracting the first digit of the previous substring then multiplying by two and adding the new last digit of the new substring. The correctness of this procedure and its asymptotic runtime are well-known. As each fingerprint is computed, we increment the integer in the array indexed by the fingerprint.

When the pattern is given, we compute its fingerprint (mod p) in the standard way; this requires time proportional to the length of the pattern. We then use this fingerprint to index into the hash table, and return the count that is stored in the appropriate field.

To analyze the probability of correctness of this algorithm, we must consider the possibility of a fingerprint collision. This occurs if two different substrings have the same fingerprint value, and in this case the algorithm may return the incorrect value. This occurs if the difference in fingerprint values $n = f(x_1) - f(x_2)$ between two different strings is a multiple of p. This occurs only if p is a prime factor of n. n has at most $\log n$ factors. If we choose p randomly from $[2, P]$, there are approximately $\frac{P}{\log P}$ primes, so using the union bound we find that the probability of a fingerprint collision is less than $\frac{|T| \log n \log P}{P}$. Since n is the difference between two binary strings of length m, $n \leq 2^m$ and so $\log n = O(m)$. So the probability of a collision is $O(\frac{|T| m \log P}{P})$. If we choose P sufficiently large, we can make the probability of a collision arbitrarily small, and since finding the prime requires $\log P$ time (we assume that our machine word is sufficiently long to perform all operations in constant time), if we choose $P = O(2^{|T|})$, we do not affect the $O(|T|)$ runtime of the preprocessing.

How do you initialize the array? not linear -2

b) We now show that given a text T, we can preprocess it in time polynomial in $|T|$ in order to output all locations where a pattern occurs in worst case time linear in the length of the pattern size and the number of occurrences.

Our preprocessing is the following: for every possible pattern size $m < |T|$, we compute all m-length substrings of T and their locations, and sort them to identify all identical substrings and group each set of identical substrings into a linked list. We then build a perfect hash table mapping the substring to its linked list of positions. This can be done using 2-level hashing in $O(|T|)$ space and expected $O(|T|)$ time. This procedure is performed $|T|$ times in generating a total of $|T|$ perfect hash tables (one for each substring length), so the runtime of the preprocessing is expected polynomial in $|T|$.

To perform a lookup, we determine the length of the pattern X, and access the hash table containing substrings of length $|X|$. We compute the hash value of X in $O(|X|)$ time, and use this to find the linked list containing the occurrences (if any) of X as a substring in the perfect hash table. Since we are using perfect hashing, there are no collisions; all bounds are worst-case bounds.

We have to check with notation...

We then output the positions of the occurrences from the linked list in time proportional to the number of occurrences.

Problem 2:

a) Using a random function to map each item to buckets, we show that bashing produces an expected constant number of collisions. Suppose that we are inserting items $x_1, x_2, \ldots, x_n$ in that order.

Define A_k to be a Bernoulli random variable for the event that x_k collides with some other element in $x_1, \ldots x_{k-1}$. Note that this can only happen if x_k maps to buckets in each hash table that are *both* occupied. Thus, $A_k \leq B_{k,1}B_{k,2}$, where $B_{k,2}$ represents the event that one of the first $k-1$ elements maps to the same bucket in the first hash table as x_k, and similarly for $C_{k,2}$ in the second hash table. We consider B_k. Since the hash value of x_k can be considered fixed, the probability that an element i ($i < k$) has the same hash value is $\Pr[C_{ik}] = \frac{1}{n^{3/2}}$. Using linearity of expectation,

$$\mathrm{E}[B_{k,1}] = \mathrm{E}\left[\sum_{i<k} C_{ik}\right] = \sum i < k \mathrm{E}[C_{ik}] = \sum i < k \Pr[C_{ik}] \leq \frac{k}{n^{3/2}}$$

By the same argument, $\mathrm{E}[B_{k,2}] = \mathrm{E}[B_{k,1}] = \frac{k}{n^{3/2}}$. So

$$A_k \leq B_{k,1}B_{k,2} = (B_{k,1})^2 = \left(\frac{k}{n^{3/2}}\right)^2 = \frac{k^2}{n^3}$$

And since the number of collisions X is the sum of the indicator random variables A_k

$$\mathrm{E}[X] = \sum_{k=0}^{n} \mathrm{E}[A_k] \leq \sum_{k=0}^{n} \frac{k^2}{n^3}$$

good 4/4

We apply the identity $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1) = \frac{2n^3+3n^2+n}{6}$. So

$$\mathrm{E}[X] \leq \frac{2n^3+3n^2+1}{6n^3} = \frac{1}{3} + \frac{1}{2n} + \frac{1}{n^2}$$

Thus for sufficiently large n, we can make the upper bound (for example) $\mathrm{E}[X] \leq \frac{1}{2}$ (or, indeed, any constant greater than 1/3).

b) Note that the only assumption we make about the independence of the hash values is pairwise independence for establishing $\Pr[C_{ik}] = \frac{1}{n^{3/2}}$. Thus, it suffices to use any hash function that provides pairwise independence. We can therefore choose two randomly selected hash functions from the 2-universal hash family to use as the hash functions for the two component hash tables. This allows for an efficient implementation with the same results.

2/2

c) We can produce a perfect bash function for n items by attempting to use the procedure above, and repeating until we find a bash function that provides no collisions. We show that this can be done in expected $\Theta(n)$ time. Note from above that the expected number of collisions was less than 1/2. So by the Markov inequality, the probability of one collision must be less than 1/2, and thus the probability of zero collisions must be at least 1/2. So the expected number of bash functions that must be generated and tested is less than 2. Note that generating a bash function requires constant time, since it merely requires generating two 2-universal hash functions, and testing it requires linear time, since each of the n elements in the set must be hashed twice and inserted into the table (or at least until a collision is found). So the expected runtime to find a perfect bash function is $\Theta(n)$.

4/4

d) Consider the general problem of m-bucket hashing ($m$ashing, I suppose). We determine the required size p of the hash table. The critical difference in the analysis is the following change:

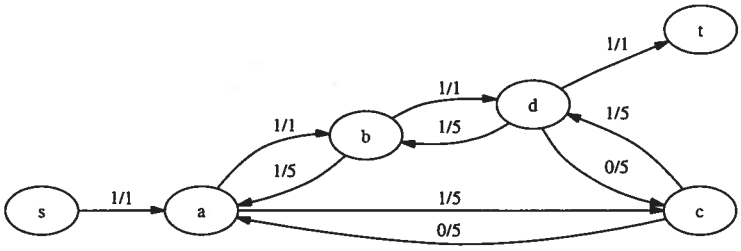
$$A_k \leq B_{k,1}B_{k,2}\cdots B_{k,m} = (B_{k,1})^m = \left(\frac{k}{n^p}\right)^m = \frac{k^m}{n^{pm}}$$

We must ensure that if $k = n$, $\frac{k^m}{n^{pm}} = \frac{1}{n}$ so that the remainder of our analysis will hold. For this to be true, p must equal $\frac{m+1}{m}$.

The only other necessary change in the analysis is that we can no longer use the identity $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)$. However, we need only observe that $\sum_{i=1}^{n} i^m$ is a constant factor smaller than n^m. This being an optional problem, we leave the proof as an exercise for the grader.

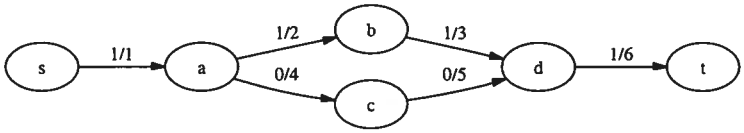
Problem 3:

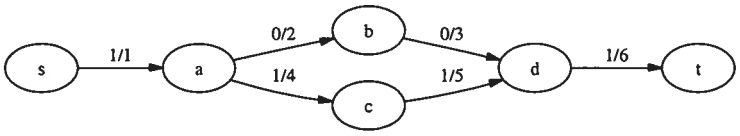
a) False. The maximum flow of the following network is clearly 1 because of the bottleneck edges. Note that it does not satisfy the condition that for all v and w, either the gross flow from v to w or the gross flow from w to v is zero.



b) True. Any maximum flow can be converted to a flow that satisfies the condition that for all v and w, either the gross flow from v to w or the gross flow from w to v is zero. To do so, note that any maximum flow given in the gross flow representation can be converted into the net flow representation (call the new flow function f_N). The net flow representation satisfies skew-symmetry, i.e. $f_N(v,w) = -f_N(w,v)$. We can convert this back into a gross flow representation by defining $f_G(v,w) = \max\{f_N(v,w), 0\}$. This clearly satisfies the capacity axiom, since the net flow representation did so. It also satisfies the conservation algorithm since every negative-flow edge of the net flow is removed, but corresponds to a positive-flow edge in the opposite direction. Thus the net flow requirement that the sum of the flow from each vertex is zero is mapped to the gross flow requirement that the sum of the outgoing flow is equal to the sum of the incoming flow. So f_G is a valid gross flow representation of a maximum flow. By our definition of f_G, it satisfies the condition that for all v and w, either the gross flow from v to w or the gross flow from w to v is zero.

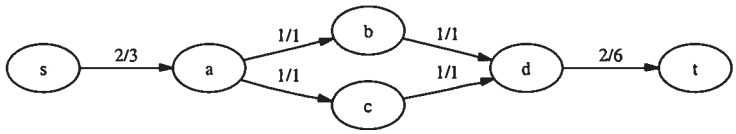
c) False. The following two flows are both maximum flows over the same network that has distinct capacities for all edges:





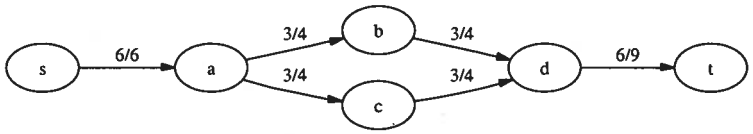
d) False. Consider the network that consists only of a source s, sink t, and a directed edge of any (non-zero) capacity connecting t to s. Then the max flow is zero. But replacing each directed edge with two directed edges in opposite directions connects s to t, giving a non-zero max flow.

e) False. Consider the following network, with max flow as shown below.



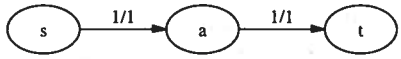
The maximum flow is 2, and a minimum capacity cut can be drawn between $\{s, a\}$ and $\{b, c, d, t\}$.

However, if we add $\lambda = 3$ to the capacities of all edges, the resulting network has the maximum flow shown below. The minimum cut is between $\{s\}$ and $\{a, b, c, d, t\}$, with capacity 6.



Problem 4:

a) Not every network contains an upward critical edge. The following network, for example, contains two edges, but raising the capacity of either of them will not increase the maximum flow of the network.

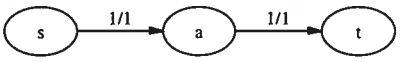


Since increasing the capacity of an upward critical edge must increase the flow, in the residual network of the maximum flow any upward critical edge will connect a node reachable by some path from the source to a node reachable from some path from the sink. We can therefore identify the upward critical edges by the following procedure:

1. Compute a maximum flow of the network
2. Generate the residual network under maximum flow
3. Identify the partitions of nodes reachable from the source and from the sink by performing breadth-first searches from the source and sink.
4. For each node reachable from the source, check whether any edges exist connecting it to a node reachable from the sink. If so, it is an upward critical edge.

Each step in this algorithm except the first requires $O(m)$ time, where m is the number of edges in the network. Thus, the dominating factor in the runtime is computing the maximum flow. This can be done using your favorite max-flow algorithm.

b) The set of upward and downward critical edges are different. Consider the same graph as above:



Here both edges are downward critical, but neither is upward critical.

To identify downward critical edges, we make two observations. First, a downward-critical edge must not appear in the residual network. If it did, then the edge would not be saturated in the maximum flow, and its capacity could be reduced without any effect. Moreover, there must not be *any* path between the two endpoints of the edge in the residual network. If there were, then if the edge capacity is decreased, the flow through the edge could be rerouted through the alternate path without affecting the total flow of the network. This condition is both necessary and sufficient.

Thus, to identify downward critical edges, we must compute the maximum flow and the residual network, then for each edge in the network determine whether there is a path from one endpoint to the other in the residual network. This can be done by computing the all-pairs shortest paths over the residual network, using the Floyd-Warshall algorithm.

The asymptotic runtime required is $O(n^3)$ to perform Floyd-Warshall, plus whatever the runtime of your favorite max-flow algorithm is. 8/5

Problem 5:

a) Suppose all people are currently in a room s and need to reach a single exit t in a system described by a graph $G = (V, E)$. Define G_k to be an auxiliary graph with k vertices for each vertex in the original graph. Each vertex represents one room at one time step. If there exists a corridor connecting u and v in G, then we create an edge in G_k between u_i and v_{i+1} for all $i < k - 1$. We make G_k into a flow network by defining the capacity of every edge to be c. Thus, a flow from s to t corresponds to moving people from s to t over k time steps, with never more than c persons in a corridor at any time. Thus, all people can be moved from s to t in time k if and only if the maximum flow from s to t is at least the number of people.

We can apply this mapping to create an algorithm for finding the fastest way to get everyone out of the building. We repeatedly generate the graphs G_k for increasing k starting with G_1 and compute the max flows over each, until we find one whose maximum flow is greater than the number of people. Then we convert the flow over the graph to a sequence of movements of people using the bijection described above.

b) With multiple starting locations and multiple exits, we use the standard technique of creating a supersource s that connects to each of the starting locations $s_1, s_2, \ldots, s_n$ by an infinite-capacity edge. Similarly, we create a supersink t that connects via infinite-capacity edges to each of the exits $t_1, t_2, \ldots, t_n$. We then apply the algorithm as described above.

c) We now generalize this approach to the case in which the corridors have different integral transit times, and their capacity represents the number of people who can enter the corridor per time step. As before, we again define G_k to be the auxiliary graph with k vertices for each vertex in the original graph where each vertex represents one room at one time step. However, for each edge $e = (v, w)$ in G representing a corridor with transit time $t(e)$, we connect v_i to $v_{i+t(e)}$ for all i such that $i + t(e) \leq k$. Thus each edge represents traveling from v to w during the time interval from i to $i + t(e)$.

As before, we can incrementally generate the graphs G_k for increasing k and compute their max flows until we find one whose max flow is greater than the total number of people that need to be moved.