CSCE 629-601 Analysis of Algorithms

Fall 2022

Instructor: Dr. Jianer Chen
Office: PETR 428
Phone: (979) 845-4259
Email: chen@cse.tamu.edu
Office Hours: MWF 3:50 pm - 5:00 pm

Teaching Assistant: Vaibhav Bajaj
Office: EABC 107B
Phone: (979) 739-2707
Email: vaibhavbajaj@tamu.edu
Office Hours: T,TR 2:00 pm - 3:00 pm

Solutions to Assignment #2

1. Let U be a (large) set with $|U| \geq nm$. Suppose that we are storing a set of n elements of U using a hash function from U to $[0..m-1]$. Show that no matter what hash function h we use, there is a subset S_h of n elements of U that are all hashed by h to the same slot. Note that this shows that in the worst case, searching in a set of n elements by hashing can be very bad and take time $\Theta(n)$.

Solution. This problem is trying to indicate that you should not expect a "universally good" hash function. In particular, every hash function is bad for *some* sets. In particular, when you are building a hash structure for your set, you should be careful in choosing a hash function. By the way, the study of universal family of hash functions was motivated by this concern and considered how we could avoid this bad situation with a large probability.

Thus, what we want to show is that for *any* hash function h, there is a bad set S_h, i.e., a set S_h such that the hash function h hashes all elements of S_h into the same slot in the hash table. More precisely, suppose we use the function h to store a set of n elements into a hash structure $H[0..m-1]$, where the set is a subset of the set U of at least nm elements and the hash function h is a mapping from U to $[0..m-1]$. Let U_k be the subset of elements in U that are mapped by h to k, for $0 \leq k \leq m-1$. Since U has at least nm elements, while there are only m subsets U_k, $0 \leq k \leq m-1$, at least one subset U_{k_0} contains at least n elements. Let S_h be the set of any n elements in this subset U_{k_0}, then h hashes all n elements in S_h into the same slot $H[k_0]$.

Therefore, if you use a bad hash function h for your set S of n elements (or, alternatively, if you pick a hash function h such that your set S happens to be a bad set for the hash function h), then searching an element in the set S using the hash structure may need to go through the entire linked list of length n in the hash structure, i.e., searching in the worst case will take time $\Theta(n)$.

2. Let S_n be a set of n elements in $[0..N-1]$. Suppose that a prime number $p \geq N$, and an array $H[0..n^2-1]$ are also given, where all elements in $H[0..n^2-1]$ have an initialized value -1. Develop a randomized algorithm of time $O(n)$ that, with a probability at least 99%, constructs a hash function h from $[0..N-1]$ to $[0..n^2-1]$ such that h is perfect from S_n. (**Hint** Your randomized algorithm is allowed to randomly pick any number, with equal probability, from a domain $[N_1..N_2]$.)

Solution. As studied in class, the following collection of hash functions is a universal family of hash functions from $[0..N-1]$ to $[0..n^2-1]$ (Theorem 11.5 in the textbook):

$$\mathcal{H}_{univ} = \{h_{a,b,n^2} \mid 1 \leq a \leq p-1, 0 \leq b \leq p-1\}, \tag{1}$$

where h_{a,b,n^2} is defined as $h_{a,b,n^2}(x) = ((ax+b) \bmod p) \bmod n^2$. Moreover, as we proved in class (Theorem 11.9 in the textbook), if we randomly pick a hash function from $\mathcal{H}_{univ}$, then the probability that the picked hash function is perfect from S_n is larger than $1/2$, or, in other words, the probability that the picked hash function is not perfect from S_n is less than $1/2$.

Therefore, if we randomly pick 7 hash functions from $\mathcal{H}_{univ}$, then the probability that all of them are not perfect from S_n is less than $(1/2)^7 = 1/128 < 1\%$. That is, the probability that at least one of these 7 hash functions is perfect from S_n is larger than 99%.

Also, as we explained in class, randomly picking a hash function $h_{a,b,n^2}(x)$ from $\mathcal{H}_{univ}$ can be implemented by randomly picking two integers a and b, where $1 \leq a \leq p-1$, and $0 \leq b \leq p-1$.

What remains is how we can identify a hash function h from these 7 randomly picked hash functions such that h is perfect from S_n. This is actually simple: we can use the hash function h to hash each of the elements in S_n, and mark values $h(x)$ for all x in S_n (the values $h(x)$ are in the range $[0..n^2-1]$ so we can use the array $H[0..n^2-1]$ to record these values). It is easy to see that the hash function h is perfect from S_n if and only If no value in $[0..n^2-1]$ is marked more than once.

We summarize the above discussion in the following algorithm.

```
PerfectHash(S_n, p, H[0..n^2 - 1])
\\ S_n is a subset of n elements in [0..N - 1], p ≥ N is a prime, H[i] = −1 for all i.
1.  for (i = 1; i ≤ 7; i++)
1.1    randomly pick a and b, where 1 ≤ a ≤ p − 1, and 0 ≤ b ≤ p − 1;
1.2    let h_i be the hash function h_{a,b,n^2};  collide = false;
1.3    for (each element x in S_n)
           k = h_i(x);
           if (H[k] == i) then collide = true; \\ h_i is not perfect from S_n
           else H[k] = i;
1.4    if (not collide) return (h_i); \\ h_i is perfect from S_n
2.  return (''no perfect hash function from S_n is found'').
```

It is easy to see that the algorithm `PerfectHash` runs in time $O(n)$: Step 1.1 takes time $O(1)$ because the algorithm is randomized so it can pick a random number in constant time. For step 1.3, note that the value $h_i(x) = h_{a,b,n^2}(x)$ can be computed in constant time using the formula $h_{a,b,n^2}(x) = ((ax+b) \bmod p) \bmod n^2$. Thus, step 1.3 runs in time $O(n)$. Since steps 1.1-1.4 are executed 7 times, the total time by step 1, thus by the algorithm `PerfectHash` is $O(n)$.

3. Based on Breadth-First-Search, write algorithms that solve the following problems, respectively:

(1) Given an undirected graph G, decide if G is connected.
(2) Given an undirected graph G, decide if G is a tree.
(3) Given an undirected graph G, decide if G is bipartite.
(4) Given an unweighted and undirected graph G and two vertices v and w in G, either construct a shortest path from v to w in G, or report that there is no path from v to w in G.

Solution. All problems can be solved by modifying the BFS algorithm.

(1) Test if a graph G is connected.

Suppose that we call BFS on any vertex v. If G is connected, then after BFS on v, all vertices will become black. Otherwise, the vertices not reachable from v would remain white. Thus, we just have to do a BFS plus a checking on the vertex colors. The algorithm is given as follows, where steps 1-3 give the standard BFS (with an arbitrary vertex s picked), and steps 4-5 check if there are still white vertices.

```
CONN(G) \\ Q is a queue
1.  for (each vertex v) color[v] = white;
2.  pick any vertex s; color[s] = gray; EnQueue(Q, s);
3.  while (Q is not empty)
      w = DeQueue(Q);
      for (each edge [w, v])
        if (color[v] == white)
          color[v] = gray; EnQueue(Q, v);
      color[w] = black;
4.  for (each vertex v)
      if (color[v] == white) return("not connected");
5.  return("connected").
```

Steps 1-3 give the standard BFS, so take time $O(m+n)$. Steps 4-5 obviously take time $O(n)$. Therefore, the algorithm CONN(G) runs in time $O(m+n)$ and tests the connectivity of the graph G.

(2) Test if a graph G is a tree.

If the graph G is a tree, then G is connected. We can use the idea for (1) to test the connectivity of G. Under the condition that G is connected, if G is a tree, then the BFS-tree, starting from any vertex s, is that tree. Therefore, if we find any edge that does not belong to the BFS-tree, then the graph G is not a tree. The algorithm is given below. We use the array dad[*] to record the parent of a vertex. Step 3.1 creates a new edge in the BFS-tree, while in step 3.2, when the vertex v is not white, then only if v is the parent of w then the edge $[w, v]$ is an edge in the BFS-tree. Again steps 4-5 check the connectivity of the graph G.

```
TREE(G) \\ Q is a queue
1.  for (each vertex v) color[v] = white; dad[v] = NIL;
2.  pick any vertex s; color[s] = gray; EnQueue(Q, s);
3.  while (Q is not empty)
      w = DeQueue(Q);
      for (each edge [w, v])
3.1     if (color[v] == white)
          color[v] = gray; EnQueue(Q, v); dad[v] = w;
```

```
3.2     else if (v ≠ dad[w])
          return("not a tree");
      color[w] = black;
4.  for (each vertex v)
      if (color[v] == white) return("not a tree");
5.  return("tree").
```

Again steps 1-3 are small modifications of BFS that do not change the asymptotic complexity. So they take time $O(m+n)$. Steps 4-5 take time $O(n)$. Therefore, the algorithm TREE(G) runs in time $O(m+n)$ and tests if the graph G is a tree.

(3) Test if a graph G is bipartite.

We partition the vertices of graph G into two sets V_0 and V_1 by assigning them a number 0 or 1, and check if all vertices can be consistently assigned. As we explained in class, when we start BFS with a vertex s, we can simply assign 0 to s becuase there is no enforced condition, yet. When we apply BFS and look at an edge $[w, v]$ where w already got an assigned number, then the number assigned to v is uniquely determined: it must be opposite to that of w. This gives the following algorithm testing bipartiteness of a graph, where the array RB is used to record the number assigned to each vertex. The algorithm consists of a function BFS and a main algorithm (note that a bipartite graph may not be connected).

In step 2 of the main algorithm, if we encounter a new white vertex v, then we assign v with 0 and start a new BFS from v. Note that once a vertex becomes non-white, it gets a number in RB[ ]. In particular, in step 2.1 of the function BFS when we look at an edge $[w, v]$ where the vertex v has not assigned a number yet (vertex w is gray so it already got a number), we assign the number opposite to that of w to the vertex v. On the other hand, if v already got a number, then step 2.2 checks if that number is consistent, i.e., if that number is opposite to that of w. If not, then the graph is not bipartite (because all assigned numbers are enforced). Finally, if all edges can pass the consistency test in the calls to BFS, then the graph is bipartite, which is reported in step 3 of the main algorithm.

```
BFS(s) \\ Q is a queue
1.  Q = ∅; color[s] = gray; EnQueue(Q, s);
2.  while (Q is not empty)
      w = DeQueue(Q);
      for (each edge [w, v])
2.1     if (color[v] == white)
          color[v] = gray; EnQueue(Q, v); RB[v] = 1 − RB[w];
2.2     else if (RB[v] ≠ 1 − RB[w]) return("not bipartite");
      color[w] = black;

main BIPARTITE( )
1.  for (each vertex v) color[v] = white; RB[v] = -1;
2.  for (each vertex v)
    if (color[v] == white) RB[v] = 0; BFS(v);
3.  return("bipartite").
```

Again the algorithm BIPARTITE is a simple modification of BFS that does not change the complexity. Thus, the algorithm BIPARTITE takes time $O(m+n)$.

(4) Shortest path from v to w.

As we explained in class, if we start BFS from the vertex v, then the BFS-tree gives the shortest path from v to every vertex in the tree. Thus, we only need to record the parent of

each vertex in the BFS, as we did in (2). If the vertex w is included in the BFS-tree, then by following the parent pointers, we will find the shortest path from v to w (in the reversed order). If w is not included in that BFS-tree, then there is no path from v to w in the graph G.

```
SHORTEST(v, w) \\ Q is a queue
1.  for (each vertex x) color[x] = white; dad[x] = NIL;
2.  Q = ∅; color[v] = gray; EnQueue(Q, v);
3.  while (Q is not empty)
      x = DeQueue(Q);
      for (each edge [x, y])
          if (color[y] == white)
          color[y] = gray; EnQueue(Q, y); dad[y] = x;
4.  if (color[w] == white)
      return("no path from v to w");
5.  t = w; print(t);
6.  while (dad[t] ≠ NIL) t = dad[t]; print(t).
    \\ the shortest path from v to w is printed backwards.
```

The algorithm is a simple modification of BFS without changing the complexity. Thus, the algorithm SHORTEST takes time $O(m + n)$.

4. Based on Depth-First-Search, write algorithms that solve the following problems, respectively:

(1) Given an undirected graph G, decide if G is connected.
(2) Given an undirected graph G, decide if G is a tree.
(3) Given an undirected graph G, decide if G is bipartite.
(4) Given an undirected graph G, either construct a cycle in G or report that G contains no cycle.

Solution. Again, all problems can be solved by modifying the DFS algorithm.

(1) Test if a graph G is connected.

Similar to BFS, if we call DFS on a vertex s and the graph is connected, then after DFS(s), all vertices should become black. This is tested by the following algorithm.

```
DFS(v)
1.  color[v] = gray;
2.  for (each edge [v, w])
      if (color[w] == white) DFS(w);
3.  color[v] = black;

main CONN(G)
1.  for (each vertex v) color[v] = white;
2.  pick any vertex s; DFS(s);
3.  for (each vertex v)
      if (color[v] == white) return("not connected");
4.  return("connected").
```

The algorithm is a simple modification of DFS. Thus, the algorithm CONN runs in time $O(n+m)$.

(2) Test if a graph G is a tree.

Again, if G is a tree, then G is the DFS-tree, starting from any vertex s. As we did in BFS, we use array dad[ ] to record the parent of each vertex in the DFS-tree. Whenever we find an edge not in the DFS-tree, we stop and report that G is not a tree. The main algorithm also checks the connectivity after the call to DFS(s).

```
DFS(v)
1.  color[v] = gray;
2.  for (each edge [v, w])
      if (color[w] == white) dad(w) = v; DFS(w);
      else if (w ≠ dad[v]) return("not a tree");
3.  color[v] = black;

main TREE(G)
1.  for (each vertex v) color[v] = white; dad[v] = NIL;
2.  pick any vertex s; DFS(s);
3.  for (each vertex v)
      if (color[v] == white) return("not a tree");
4.  return("tree").
```

The algorithm is a simple modification of DFS. Thus, the algorithm TREE runs in time $O(n+m)$.

(3) Test if a graph G is bipartite.

We apply DFS. When we start DFS on a vertex v, we assign v by 0. During DFS, we assign 0 and 1 to each vertex and check the consistency on each edge.

```
DFS(v)
1.  color[v] = gray;
2.  for (each edge [v, w])
      if (color[w] == white) RB[w] = 1 − RB[v]; DFS(w);
      else if (RB[w] ≠ 1 − RB[v]) return("not bipartite");
3.  color[v] = black;

main BIPARTITE(G)
1.  for (each vertex v) color[v] = white; RB[v] = -1;
2.  for (each vertex v)
      if (color[v] == white) RB[v] = 0; DFS(v);
3.  return("bipartite").
```

The algorithm is a simple modification of DFS without changing complexity. Thus, the algorithm BIPARTITE runs in time $O(n + m)$.

(4) Find a cycle in a graph G.

When we call DFS, starting from any vertex v, if we find an edge $[d, a]$ not in the DFS-tree rooted at v, then we find a cycle. As we discussed in class, this kind of edges are called *back edges* and must connect a descendant d to an ancestor a in the DFS-tree. Thus, the tree path from a to d plus the edge $[d, a]$ forms a cycle in the graph. Note that the graph can be not connected but still contain cycles.

```
DFS(v)
1.  color[v] = gray;
2.  for (each edge [v, w])
      if (color[w] == white) dad[w] = v; DFS(w);
      else if (w ≠ dad[v]) \\ find a cycle
           t = v; print(v);
           while (t ≠ dad[w]) t = dad[t]; print(t);
           return;
3.  color[v] = black;

main CYCLE(G)
1.  for (each vertex v) color[v] = white; dad[v] = NIL;
2.  for (each vertex v)
      if (color[v] == white) DFS(v);
3.  return("no cycle").
```

The algorithm is a simple modification of DFS without changing complexity. Thus, the algorithm CYCLE runs in time $O(n + m)$.