CSCI E-124-Spring, 2004 Homework 2

Russell Lowke, March 2nd 2004

1. A bipartite graph...

To see if a graph is bipartite, we can take advantage of the fact that there can never be an edge between vertices in the same set. Therefore, if we follow an edge from a vertex in V1, we can be assured that the vertex at the other end *must* be in V2 and vice versa. Knowing this, we can do a small modification of the depth first search algorithm that keeps track of the set the current vertex belongs to:

```
Procedure search(v, currentSet)
        vertex v
        int currentSet
        explored(v) := currentSet
        for (v, w) within E
                if explored (w) = 0 then search(w, ~currentSet)
                else if explored (w) = currentSet then This isn't bipartite
        rof
end search

Procedure DFS (G(V, E))
        graph G(v, E)
        for each v within V do
                explored(v) := 0
        rof

        int currentSet := firstSetValue
        for each v within V do
                if explored (v) = 0 then search(v, currentSet)
        rof
end DFS
```

The integer currentSet can take on two values representing the two sets that we are trying to separate the vertices into. They could be 1 and 2, for example, but I'm assuming that the ~ operator will flip between the two possibilities. I'm expanding the role of the explored list to take on 3 values: 0 still means unexplored, and the other two represent which set the node has been put into when it was reached.

The key to the correctness of the algorithm is the fact that, if you grab any given node in a

bipartite graph, let's say from subset V1, all edges coming out of it will be connected to nodes in the *other* subset, which would be V2 in our example case. Therefore, if we *ever* find a node in one subset that's directly connected to a node in **the same** subset, it has violated the definition of a bipartite set. We can notify the user by returning a Boolean value, outputting something to a screen, etc.

The given algorithm is the exact same depth first search algorithm given in class,. The only modification is the passing of an additional parameter (which we assume costs nothing) and the one additional check within the for loop, which will take constant time. So this runs just like the DFS algorithm given in class which has been shown to be O (|V| + |E|).

2. Traverse all of the streets of Sunnydale

Assumptions:

- No one way streets (i.e. graph is undirected)
- No loops on single nodes ((v, v) not an edge)
- If you start anywhere in Sunnydale, you can reach all streets and corners (i.e. graph is connected)

```
Procedure search (v)
        print(v);
        for (v, u) within E
                if traveled(v, u) == 0,
                        traveled(v, u) = 1
                        traveled(u, v) = 1
                        search(u)
                        print(v)
                fi
        rof
end search:
```

The main algorithm would start with a node s, any node s, and call this search function on it. This is similar to a depth first search except that it doesn't mark off vertices as having been visited. Instead, it marks off the exact *edge* you took to reach the vertex. Doing this, it makes sure that you can a) visit a vertex twice, and b) not walk back down the path you took to get there until all others are exhausted.

Why this works: This algorithm is guaranteed to cover every edge in the graph (we assume that all vertices and thus all edges are reachable). The tricky part is crossing every edge twice in opposite directions. I realized that the algorithm can do the forward part easily and the backward part was *implicit* in the returning of the function. However, to make it explicit, I changed the code to print the name of the vertex (the street corner) at the end of every edge (street) so that it will produce a list of vertices that can be used like a set of instructions on how to traverse the map. The first print statement is equivalent to first walking to the corner from any given street. The second print statement is called whenever you run out of new streets to go down, turn around, and go *back the way you came*.. or just when the function returns. Since we assumed that all edges are reachable from any starting point *and* this function doesn't stop until it has crossed all edges, this function is guaranteed to produce (by essentially logging its progress) the instructions we need to do so.

3. Shortest-path algorithm

To solve the single source bottleneck problem, we start with the single source shortest path algorithm offered by Djikstra. However, now instead of storing array dist[], we store array b_neck[] which constantly keeps track of the *largest edge* along the path to vertex v (of course this is also the value pushed on the heap with it). At every node, instead of updating the distance to that point (if that the distance is less than the previous value) we now update the largest bottleneck (if that the bottleneck is now less than the stored value) updating previous just as we did before. The section of the Djikstra algorithm to be changed, the main while loop, is shown below:

```
while h is not empty
    v:= deletemin(h)
    for (v, w) within E
        if b_neck[w] > b_neck [v] and length (v, w)
            b_neck[w] := max {b_neck[v], length(v, w)}, prev[w]:= v, insert(w, b_neck[w], H)
        fi
    rof
end while
```

The statement's changed are the comparison, where you check to see if the current value stored as the largest bottleneck is greater than the largest bottleneck along the path we've currently followed there or the next step it would take to get there (as they are both would be in the final path to w). If so, then we updated the value with the maximum of the two values. We're assuming the existence of some constant time algorithm max.

This keeps the same invariant a the Djikstra algorithm; namely, that the values stored in b_neck[] are *always a conservative overestimate* of the true largest bottleneck to v from s. When a node w is visited, the value of b_neck[w] is immediately checked and updated if it's larger than the true value (which, because of our overestimates, it will always be until it's visited the first time). This runs in the same time as the Djikstra algorithm, which means that it'll be implementation dependent.

4. Risk-free currency exchange

We can model currencies as vertices and their exchange rates as the edges between them. Doing so gives us, for n currencies, a graph with n vertices and n^2 - n edges. A loophole occurs in the currency exchange when, between any two currencies represented by vertices a and b, |(a, b)| * |(b, a)| isn't equal to 1 (where the weight of the edges represents the exchange rate). So, if you want to see if such a loophole exists *anywhere* in the graph, you can simply multiply together all of the edges in the graph and see if they're equal to one. If they're not then we know there's a loophole somewhere. This works because, even though we're multiplying all edges together, we can imagine pair the edges off like (v, w) and (w, v). If each of these (v, w) (w, v) pairs comes out to 1, the entire product will come out to 1 and there will be no errors. If even one of these comes out to something other than 1, the product will be something other than 1 and we'll detect it. This runs in O(|E|) time, but since |E| is n^2 - n, in this case, we can say it runs in O(n^2) time.

5. Consider the shortest paths problem where all edge costs are nonnegative integers. Describe a modification of Dijkstra's algorithm.

In the case where the edge costs are all nonnegative integers, we can use the solution of dividing the edges into smaller edges of length 1 and inserting the necessary dummy nodes in between (L - 1, where L is the length of an edge). What this allows us to do is to run Dijkstra with a DFS algorithm which will be O(|E|+|V|) (but |V|, after our modifications, will now be |V|m where m is the largest edge size. To DFS, we make a key modification: we keep track, every time the function calls itself, of a variable that increases by one. When we reach a node that's not a dummy node, we do our standard Dijkstra check (to see if our value is less than the one that's there). As long as we keep the values initialized to infinity to start this keeps the same Dijkstra invariants and this will work within the asymptotic bounds given.

6. Give an efficient algorithm that takes as input a DAG G and two vertices s and t and outputs the number of paths from s to t in G.

One can perform a depth-first search on G where we input s as the source. When the simple DFS algorithm is run, instead of marking a given node as simply having been visited (explored:= 1), we could keep a counter that starts at 0 and is updated by 1 *every time* we visit a node. So we may go to a node twice, but we'll *never* hit an infinite loop where we constantly cycle through a node because our graph is acyclic. The pseudo code for the search procedure might look like the following:

```
Procedure search(v)
        vertex v
        explored(v)++;
        for (v, w) within E
                search(w)
        rof
end search
```

and at the end you can simply output the value for explored(t), since t is the particular that we're trying to find all paths to. This can be modified slightly to only update the value for the actual vertex t, but since these are constant time modifications both share the same asymptotic bound, namely O(|E|) because, worst case scenario, you go through every edge in the graph.

7. Say that a list of numbers is *k* close to sorted if each number in the list is less than *k* positions from its actual place in the sorted order. (Hence, a list that is 1 close to sorted is actually sorted.) Give an $O(n \log k)$ algorithm for sorting a list of *n* numbers that is *k* close to sorted.

Knowing that the list is k less than sorted, we can do a pass over the list and, for every spot, check k-1 spaces forward, backward, and forward comparing values (swapping value if it's greater or less than the other value, respectively). These operations can be done in constant time. On our next pass over the list, we divide k in half and run the same algorithm (and now we are checking k/2 -1 spaces forward and backward). We stop this process when k = 1 because our list is sorted. The time to iterate through the entire list is only O(n) as the two comparisons are constant time. We pass through the list logk times, so the running time for this algorithm is O (n log k).