a5 CSC263

Tony and Andrew

March 2024

1

Given a list V and list E that contain the nodes and edges, the algorithm will make a disjoint set to solve the problem. Before the algorithm is made, we make an augmentation on each DisjointSet. Now, the head of this disjoint set will contain the size of the set, in other words, how many nodes are in the set. This will require modifications to the Union and MakeSet operations.

The algorithm will find n given that the vertices are labeled 1 through n through the edges. Then, it will instantiate each node in V as their own disjoint set using MakeSet. Then, in the order $e_m, ..., e_2, e_1$, in other words in backwards order, for every edge it will use the disjoint set(DS) operations FindSet(DS, v) and Union(DS, x, y) to merge the sets that contain the nodes in the given edge. For example, if e_m = (1,2) the algorithm will call Union(DS, 1, 2) on the element nodes of 1 and 2. Then, the algorithm will check to see if any of the disjoint sets have a size greater than floor(n/4) where n is the number of nodes. If that is the case, the algorithm terminates and returns the edge that was last inserted (e_i). This edge is e_i because adding it breaks the condition of all the sets have less than floor(n/4) nodes.

Below is the pseudo code implementation of the algorithm and any changes to the disjoint set operations.

```
def MakeSet(x):
    // IMPORTANT: Each index for the arrays earlier represent a node in the graph.
    x.p = x
    x.rank = 0
    x.size = 1

def Find(x)
    \\ Nothing changes for Find(x), assume it is the implementation from class

def Union(x, y):
    rootX = Find(x)
    rootY = Find(y)
    if rootX != rootY:
        // Union by Rank
        if rootX.rank < rootY.rank:
            ...
            rootY.size = rootX.size + rootY.size
        else:
            ...
            rootX.size = rootX.size + rootY.size
        return rootX.size

def findCriticalEdge(edges):
    n = 0
    for edge in edges:
        if max(edge[0], edge[1]) > n:
            n = max(edge[0], edge[1])

    // Make each node its own set
    for i from 1 to n:
        MakeSet(i)

    // Process edges in reverse order
    rev = reverse(edges) \\ The edges in reverse order
    for each edge (u, v) in rev:
        if Find(u) != Find(v) Then
            if Union(u, v) > floor(n/4):
                return (u, v)
```

```
return NULL  // No critical edge found but this should not happen if input is correct
```

In terms of the size augmentation, all of the changes to MakeSet and Union with reference to size are all constant time. The implementation of MakeSet, Find, and Union derive from their in class examples and are translated into pseudo-code.

In terms of overall correctness, the worst case running time of the algorithm is $O(m \times \alpha(n))$. We know from class that for a sequence of m disjoint set operations after n MakeSet operations, the worst case runtime of the sequence is $O(m \times \alpha(n))$. To find the number of nodes n takes m time. Initializing all of the elements takes n time since it is a for loop over n elements that take constant time. Note that because the graph is connected, the number of edges m will be at least $n - 1$. Since $n - 1 \leq m$ and $n \in O(n - 1)$, $n \in O(m)$.

Processing the edges in reverse order for m edges results in $2m$ Find(x) operations and m Union operations at most (though because of the condition of the inner if statement there will never be this many operations). We are told in class a sequence of m find and union operations takes $O(m \times \alpha(n))$ time. As such, a sequence of $3m$ operations after scaling by a constant also takes $O(m \times \alpha(n))$ time. In total, the worst case runtime is $(m + n + 3m)$. Since $n \in O(m)$ in this case, $(m + n + 3m) \in O(m \times \alpha(n))$. So the worst case running time is $O(m \times \alpha(n))$.

2

a) To determine whether a graph is connected, we can simply keep track of the number of unique nodes visited during BFS and return TRUE when that number is equivalent to the total vertices in the graph. To reduce the runtime of BFS for a complete graph, we can utilize the fact that every vertex is connected to every other vertex. This means that running BFS on the graph after the first node will result in visiting/enqueuing all of the vertices. For the sake of finding connectivity, and reducing the runtime for connected graphs, we can end the algorithm early if the number of white nodes found is equal to the number of vertices. For the code below, assume that ... represents code from the implementation from class.

```
function ModifiedBFS(G, s):
   n = 0
   Initialize vertices as shown in class
        Increment n such that it represents the number of vertices
    ...
    num_whites_found = 1 # we start at 1 for the root being discovered
    while Q not empty:
        u=Dequeue(Q)
        ...
        for each v in G.adj(u):
            if v.colour == White:
                ...
                num_whites_found += 1
            if num_whites_found == n:
                Return True
    Return False
```

Over the running of the entire program, the for loop iterates at most $2m$ times, taking constant time each iteration (same as the in-class breadth search). The initialization of the vertices takes $O(n)$ time, as is the case for determining n. A maximum of n items can be enqueued. Considering all of the this, the runtime for an arbitrary graph is $O(n + m)$. However, for a complete graph, since the source node is connected to every other node in the graph, the function terminates immediately after n-1 iterations of the inner for loop (where only the source node has been dequeued), taking $\Theta(n)$ time.

b) One disconnected graph on n vertices where the running time is $\Theta(n^2)$ is one where $n - 1$ elements form a complete graph and there is one element that is not connected. Suppose we start on one of the $n - 1$ connected elements for our ModifiedBFS. This would mean that the nth element would never be enqueued and is also not connected to the graph. The algorithm would never terminate early since the number of whites will never be equal to n and will reach at most $n - 1$. Since the $n - 1$ nodes are connected, for each of the $n - 1$ enqueued nodes there will be $n - 2$ iterations of the inner for loop. Since $(n - 1)(n - 2) \in \Theta(n^2)$, this qualifies as an example of a graph which answers the question.

3

a)

```
Bipartite(G):
    # ... the beginning is verbatim from the BFS in class, except we pick the root node
    # arbitrarily rather than specifying it as an argument ...
    odd_cycle_not_found = TRUE
    while Q is not empty:
        u = Dequeue(Q)
        ...
        for vertex v in G.adj[u]:
            if v.colour == white:
                ...
            else if v.d == u.d:
                # as stated in class, v.d and u.d are the minimum distances from v and u
                # to the source node, respectively
                odd_cycle_not_found = FALSE
                return odd_cycle_not_found
        ...
    return odd_cycle_not_found
```

We want to prove that after the algorithm finishes, TRUE is returned iff G is bipartite. Since G is bipartite iff there exists no odd cycle in G, it is sufficient to prove that after the algorithm finishes, FALSE is returned iff G contains an odd cycle. The key insight driving the algorithm is that for breadth-first search, an odd cycle is found if and only if a node visits a neighbor with the same minimum distance from the source node ("level") as it. Additionally, we note that before nodes at a particular level are dequeued, the nodes in the level below are always dequeued first. This allows us to define the following "level invariant", an invariant that is true just after all the nodes in a particular level have just been dequeued:

- After all the nodes in the n^{th} level have been dequeued, we construct a graph G' from the vertices of G and edges containing at least one vertex from level n or below. G' does not have an odd cycle iff odd_cycle_not_found is TRUE.
- Also, no edges in G' connect vertices in the same level iff odd_cycle_not_found is TRUE.

We prove this level invariant by induction. Note that level refers to the "current" depth of the BFS tree.

Base case: Let $n = 0$. After the source node (the only node in the 0th level) has been dequeued, the edges of G' comprise only the ones with the source node as an endpoint. Therefore, there is no odd cycle in G'. Also, since the neighbors visited from the source are all initially white, odd_cycle_not_found is TRUE. The second part of the level invariant also holds, since all the edges in G' are formed by connecting the source node with nodes in level 1.

Inductive step: Let $n \in N^{\geq 1}$. Assume the level invariant holds after the $n - 1^{th}$ level has been completely dequeued. Also, assume that at least one node in the n^{th} level will be dequeued. Then before any nodes in the nth level are dequeued, odd_cycle_not_found must be TRUE (otherwise the program would terminate and FALSE returned). We construct G' from the vertices of G and edges containing at least one vertex from level $n-1$ or below. By the LI, G' contains no odd cycles and does not have edges with vertices in the same level. We now consider the sequential dequeuing of the nodes in the nth level and represent visiting each of their adjacent nodes as "adding" edges to G'. Consider the dequeuing process of a node v where it visits one of one of its adjacent nodes u. We divide this into cases:

- Case 1: u is white. Then this is the first time u is visited. No cycles are newly formed from adding (v, u) to G'. Also, odd_cycle_not_found remains TRUE.
- Case 2: u is black. We divide this into subcases based on u's level:
  - Subcase 1: $u.d < v.d$. Then G' already contains (v, u), and no new odd cycles are formed. Also, odd_cycle_not_found remains TRUE.

- Subcase 2: $u.d = v.d$. Then odd_cycle_not_found becomes FALSE and the program terminates. After adding (v, u) to G', we prove that an odd cycle is present. Consider the minimum distance paths $P_1 = su_1u_2...u_{n-1}u$ and $P_2 = sv_1v_2...v_{n-1}v$ from the source node s to u and v, respectively. P_1 and P_2 share at least one vertex, the source node. Let S be the set of all shared vertices between P_1 and P_2. We note that the path position of each $s \in S$ is the same, ie. if s is the x^{th} vertex passed through in P_1, it must also be the x^{th} vertex passed through in P_2 (if not, then either P_1 or P_2 is not a minimum distance path). Let $c \in S$ be the shared vertex closest to the end of P_1 and P_2 and let y be its minimum distance from u and v. From this, we can now construct the path $P_3 = uu_{n-1}...c...v_{n-1}vu$, which is an odd cycle of length $2y + 1$.
- Subcase 3: $u.d > v.d$. Then there exists some (v', u) in G' where v' is distinct from v but has the same level. After adding (v, u) to G', we note that any new cycle formed in G' must follow the pattern $v'uv...v'$, and now show that the length of such a path must be even. Consider an arbitrary path P connecting v and v' using only edges in G'. Each vertex in the path must differ in level from the previous node by exactly one (this can be shown through a proof by contradiction which incorporates the level invariant). Since v and v' are at the same level, any change in level must be "counteracted" at some point by an equal change in the other direction, resulting in the length of P to be even. Therefore, any new cycle created must be even. Also, odd_cycle_not_found remains TRUE.

From all of this, it is clear that odd_cycle_not_found is FALSE if and only if an odd cycle-creating edge is added to G'. Additionally, only when odd_cycle_not_found is FALSE will there be edges connecting nodes in the same level. Therefore, the level invariant holds for the n^{th} level.

Partial Correctness: Once the last level has been completely dequeued, because of the level invariant, we know that odd_cycle_not_found is TRUE (and therefore the program returns TRUE) iff G does not contain an odd cycle. And so the program is correct.

Runtime analysis: From class, the runtime of breadth first search was shown to be $O(n + m)$, where $n = |V|$ and $m = |E|$. At most, our implementation of Bipartite(G) only adds a constant number of steps for each for loop iteration (when the neighbors of a node are checked), so the runtime is still $O(n+m)$. Since G is assumed to be connected, $m \geq n - 1$. Therefore, the algorithm's runtime is $O(m)$.

b)

```
Better-Bipartite(G): # main program
    # instantiate the vertices as stated in class
    ...
    S1 = {}
    S2 = {} # S1 and S2 are empty sets to store the bi-partitions
    for vertex v in G.V:
        if v.colour == white:
            BFS_result = Bipartite-Helper(G, v, S1, S2):
            if BFS_result[0] == TRUE:
                S1, S2 = BFS_result[1], BFS_result[2]
            else:
                return FALSE, BFS_result[1]
    return TRUE, S1, S2

Bipartite-Helper(G, root, S1, S2):
    Q = Queue() # instantiate an empty queue
    root.d = 0
    Add(S1, root)
    Enqueue(Q, root)
    odd_cycle_not_found = TRUE
    while Q is not empty:
        u = Dequeue(Q)
        ... same as in class ...
```

```
        for vertex v in G.adj[u]:
            if v.colour == white:
                ... same as in class ...
                if v.d is even:
                    Add(S1, root)
                else if v.d is odd:
                    Add(S2, root)
            else if v.d == u.d:
                odd_cycle_not_found = FALSE
                return odd_cycle_not_found, Make-Linked-List(u,v)
        ... same as in class ...
    return odd_cycle_not_found, S1, S2

Make-Linked-List(u,v):
    curr1, curr2 = LL_Node(u), LL_Node(v) # instantiates linked list nodes that have
    # empty "next"and "previous" attributes
    head = curr1
        while u.p != v.p: # iterates until the first shared vertice is found
            curr1.next, curr2.next = LL_Node(u.p), LL_Node(v.p)
            curr2.next.prev = curr2 # assumes each node can track previous node in list
            curr1, curr2 = curr1.next, curr2.next
            u, v = u.p, v.p
        curr1.next = LL_Node(u.p)
        curr1 = curr1.next
        while curr2 is not NIL:
            curr1.next = curr2
            curr1, curr2 = curr1.next, curr2.prev
        curr1.next = head
    return head
```

Description and Correctness: The backbone of the algorithm is the same as Bipartite(G); it checks if an odd cycle is present in G. The bulk of the algorithm from the previous part is carried over into Bipartite-Helper. Better-Bipartite is the main program that calls Bipartite-Helper to begin breadth-first search calls that traverse a single connected component each. To prevent a breadth-first search from being called on a connected component multiple times, Bipartite-Helper is only called on white vertices, which have not been reached by previous BFS calls. S1 and S2 store a bi-partition of the nodes in G as they are traversed. Even level nodes are added to S1, while odd level nodes are added to S2 (so S1 and S2 are disjoint).

If a Bipartite-Helper call does not encounter an odd cycle in its connected component, it returns TRUE as well as the updated S1 and S2 sets containing all nodes traversed so far. If no Bipartite-Helper calls encounter odd cycles (ie. if there exists no odd cycles in G), all nodes in G are traversed and (TRUE, S1, S2) is returned. As explained previously in Q3 part a, for a graph containing no odd cycles, each node's neighbors differ in level from the node by exactly 1. Thus, each edge contains exactly one even-level node and odd-level node, and therefore has an endpoint in both S1 and S2. Then S1 and S2 are valid bi-partitions.

In contrast, if there exists an odd cycle in G, when it is detected, Bipartite-Helper calls on Make-Linked-List to construct a linked list out of the exact path described in the Q3a) inductive step, case 2, subcase 2. Since a node can only backtrack by visiting its parent, we create two linked lists, reverse one and append it to the other to get the cycle. Bipartite-Helper then returns (FALSE, 'head of the odd cycle'), which is also returned by Better-Bipartite, and the algorithm terminates.

Explanation for runtime: Instantiating each vertex at the beginning of Better-Bipartite and later checking to see if each node has already been visited takes $O(|V|)$ time. Each node is enqueued at most once in all Bipartite-Helper calls. Therefore, over the time Better-Bipartite runs, the number of for loop iterations within all Bipartite-Helper calls is at most $2|E|$. Since the minimum distance of a node from its BFS root is bounded by the number of nodes in the graph, traversing 2 minimum distance paths and constructing a linked-list from them in Make-Linked-List takes $O(|V|)$ time. Therefore, the runtime of the algorithm is $O(|V| + |E|)$.