Pset 3

Darwin Do

February 22, 2022

(a) Darwin Do

(b) 919941748

(c) Collaborators: Raffael Davila

(d) I have followed the academic integrity and collaboration policy

(e) Hours: 20

1 Checking Connectivity

(a)
```
function RedFind(G, u, v)
    for all v ∈ V do
        visited[v] ← False
    Q ← MakeQueue()                          ▷ standard FIFO queue
    Q.push(u)
    while Q not empty do
        q ← Q.pop()
        if q == v then return True
        visited[q] ← True
        for all (q, v) ∈ E do
            if color(q, v) == RED and not visited[v] then
                Q.push(v)
     return False
```

Correctness

This algorithm is basically BFS with the added constraint that only nodes connected to the current node via a red edge can be added to the BFS queue. That means this algorithm will discover all nodes in the graph G' where G' is defined as G with only red edges. A path that only contains red edges will exist in G' so this algorithm will discover that path if it exists.

Running Time

$O(|V| + |E|)$

With the following search constraint of only searching red nodes, the worst case scenario is where all edges in the graph are red. If this is the case, this algorithm is exactly BFS and $O(|V| + |E|)$ as discussed in class as each vertex will be added/remove from Q once and the while loop will access $2|E|$ edges as per the handshake lemma.

(b)

```
function DFS(G, u, v)
    for all w ∈ V do
        visited[w] ← False
    visited[u] ← True
    for all (u, w) ∈ E do
        if COLOR(u, w) == Red then
            if EXPLORE(G, w, v, Red) then return True
        else if EXPLORE(G, w, v, Blue) then return True
    return False

function EXPLORE(G, w, v, edgeColor)
    visited[w] ← True
    if w == v then return True
    for all (w, u) ∈ E do
        if edgeColor == Blue then
            if COLOR(w, u) == Blue and not visited[u] then
                if EXPLORE(G, u, v, Blue) then return True
        else                                  ▷ previous edge is still on red path
            if not visited[u] then
                if EXPLORE(G, u, v, COLOR(w, u)) then return True
    return False
```

Correctness

This is algorithm is a modified version of DFS. When the DFS algorithm has crossed a blue edge, it will then only travel along blue edges along that path. Otherwise, it is a normal DFS that will return `True` if the current recursive level or one of its sublevels has detected the target node and `False` otherwise. If there is a path to the target node with the same color edges: the modified-DFS conditions are irrelevant and the algorithm functions as a standard DFS whose correctness was proven in class with the `Explore` lemma that all nodes reachable in the path will be explored. If there is a path that starts with red edges and then later has blue edges, once the algorithm detects the blue edges, it will only recurse on other blue edges. This means that the target node at the end of the blue-edges will be found if it exists.

Running Time

$O(|V| + |E|)$

The same logic from part (a) applies here. The worst case scenario is when all edges in the graph are the same color and no nodes must be skipped in the search. If this is the case, this algorithm is exactly DFS and $O(|V| + |E|)$ as discussed in class as each vertex will be visited once and each edge will be accessed $2|E|$ times per handshake lemma.

2 Road Trip

(a)
```
function IsReachable(G, s, t, ℓ)
    for all v ∈ V do
        visited[v] ← False
    Q ← MakeQueue()                          ▷ standard FIFO queue
    Q.push(s)
    while Q not empty do
        q ← Q.pop()
        if q == t then return True
        visited[q] ← True
        for all (q, v) ∈ E do
            if L ≥ ℓ(v, q) and not visited[v] then
                Q.push(v)
    return False
```

Correctness
This algorithm is very simliar to part (1.a) where it is a modified BFS that only adds nodes to the queue if another condition is met. In this case, the condition is that the car's gas tank is big enough to traverse the edge between two nodes. We can say that this algorithm will perform a full BFS on every node in graph G' where G' is defined as all the nodes with a connecting edge with edge weight $\leq L$. If there is a feasible route from s to t then $t \in G'$ and the algorithm will find it.

Running Time
$O(|V| + |E|)$
The same logic from part (1.a) applies here. The worst case scenario is when all edges are traversable and we must perform BFS on the entire graph leading to $O(|V| + |E|)$ runtime.

(b)
```
function LowestGas(G, ℓ, s, t)
    for all v ∈ V do
        maxL[v] ← ∞
    maxL[s] ← 0
    P ← MakeQueue(V)   ▷ priority queue with maxL values as keys
    while P not empty do
        v ← ExtractMin(P)
        for all (v, w) ∈ E do
            if maxL[w] > maxL[v] and maxL[w] > ℓ(v, w) then
                maxL[w] ←max(maxL[v], ℓ(v, w))
                ChangeKey(P, w)
    return maxL[t]
```

Correctness
This is a modified version of the priority-queue implementation of Dijkstra's algorithm for finding the shortest path in a weighted graph. Suppose

we have a region $R \subseteq V$ where $\forall v \in R$, of all paths to v, maxL[v] contains the value of the heaviest edge in the path to v with the lightest heaviest edge (let's call this the **significant edge**). We grow R greedily by moving to the next node with the lightest significant edge. This is equivlanet to the priority-queue implementation of Dijikstra we talked about in class, but with the heuristic of minimizing the weight of the significant edge instead of the sum of all weights. Note that when updating **maxL** of a neighbor, we set it to either the weight of the significant edge of the current node or the weight of the edge from the current node to the neighbor, whichever is greater. At the end of the algorithm, all nodes will be in R and maxL[t] will return the value of the significant edge of the target node.

Running Time
$O((|V| + |E|)log|V|)$
In class we discussed how the priority queue implementation of Dijikstra to find the shortest path is $O((|V| + |E|)log|V|)$ as the $log|V|$ comes from the priority queue operations. This algorithm uses the same structure as the one in class. We use a priority queue to keep track of nodes not yet in our solved region R and update the priority queue every time a change to the **maxL** array is made. The only difference is the logic in when to update **maxL**. For a worst-case scenario, this does not change and the runtime is the same as mentioned in class.

3 Counting Shortest Paths

We first define a structure that holds both the length of the shortest path to a node and the number of paths with that length. Our algorithm is a modified version of the priority queue implementation of Dijikstra's algorithm. Struct Definition

```
struct {
    int dist, numPaths;
} PathStruct;
```

function NumShortest(G, ℓ, s, t)
  for all $v \in V$ **do**
    paths[v].dist $\leftarrow \infty$
    paths[v].numPaths $\leftarrow 0$
  paths[s].dist $\leftarrow 0$
  paths[s].numPaths $\leftarrow 1$
  $P \leftarrow$ MakeQueue(V) ▷ priority queue with dist values as keys
  while P **not** empty **do**
    $v \leftarrow$ ExtractMin(P)
    for all $(v, w) \in E$ **do**
      if paths[w].dist > paths[v].dist + $\ell(v, w)$ **then**
        paths[w].dist $\leftarrow$ paths[v].dist + $\ell(v, w)$
        paths[w].numPaths $\leftarrow$ paths[v].numPaths
        ChangeKey(w)
      else if paths[w].dist == paths[v].dist + $\ell(v, w)$ **then**
        paths[w].numPaths $\leftarrow$ paths[w].numPaths + paths[v].numPaths
  return paths[t].numPaths

Correctness

This is a modified version of the priority-queue implementation of Dijkstra's algorithm for finding the shortest path in a weighted graph. We do the same steps with keeping track of the shortest path to each node and greedily expanding our solution set R. The additional steps we do are to keep track of the number of paths with the **numPaths** value in our Path struct. If the next smallest-path node not in R, v has a path that is smaller than the currently recorded path for v, the distance of the path will be updated accordingly in **paths**[w]**.dist** and the **numPaths** will be set to the number of paths to the newest member of R. If v has a smallest path equivalent to the value of the smallest path in the newest member of R, v's **numPath** will be set to the sum of the two nodes' **numPaths**. When t is added to R, we will have the number of smallest paths to t.

Running Time

$O((|V| + |E|)log|V|)$

This follows the same logic in part 2B. We use a modified version of Dijikstra that doesn't change the runtime. Here, the number of loop iterations and queue

operations are the same, we just keep track of more data in each iteraetion.

4 Spanning Tree with Leaves

We use a modified version of Kruskal's algorithm.

function LeafSpanningTree(G, U, w)
  for all $v \in V$ **do**
    MakeSet(v)
  $F \leftarrow \{\ \}$
  sort edges E by increasing weight
  $E' \leftarrow \{\{u, v\} \in E | u \notin U \wedge v \notin U\}$
  $E'' \leftarrow \{\{u, v\} \in E | u \in U \oplus v \in U\}$
  for all $\{u, v\} \in E'$ **do**
    if Find(u) $\neq Find(v)$ **then**
      $F \leftarrow F \cup \{\{u, v\}\}$
      Union($\{u, v\}$)
  for all $\{u, v\} \in E''$ **do**
    if Find(u) $\neq Find(v)$ **then**
      $F \leftarrow F \cup \{\{u, v\}\}$
      Union($\{u, v\}$)

Correctness

In class we discussed how Kruskal's algorithm can be used to construct a MST. Let us define G' as the graph of G without any vertices that are in U. If we run Kruskal's on G', we get the lightest spanning tree that can be constructed with $V' = V \setminus U$. Now we must add on the remaining vertices $v \in U$ to form our spanning tree with v as leaves. We remove any edges (v, u) connecting to vertices in U where $v \in U \wedge u \in U$ and run Kruskal's again on this edgeset. The connections from $v \in U$ to G' will be minimal following Kruskal's and v will be guaranteed a leaf as we removed all edges connecting elements in U together.

Running Time

$O((|V| + |E|)log|V|)$

We discussed in class how Kruskal and Prim's algorithm have the same runtime as a P-queue implementation of Dijikstra, $O((|V| + |E|)log|V|)$. Since our algorithm basically performs 2 iterations of the loop in Kruskal's algorithm, we have the same asymptotic runtime.

5 Perfect Matching in a Tree

This algorithm starts from the leaves of the graph and creates pairs of matchings while working its way inwards. If there is a conflict where there is more than 1 way to go "inwards" into the graph, the spot is visited later through the P queue. If a node is found to be unpairable with any of its neighbors, `False` is returned, otherwise, `True` is returned at the end when all nodes are paired up.

```
function CheckPerfectMatching(G)
    Q ← MakeQueue()                                   ▷ standard FIFO queue
    P ← MakeQueue()                                   ▷ standard FIFO queue
    for all v ∈ V do                                  ▷ add all leaves to Q
        isPaired[v] ← False
        if degree of v == 1 then
            Q.push(v)
    while Q not empty and P not empty do
        while Q not empty do
            q ← Q.pop()
            if isPaired[q] then
                continue
            r ← None
            for all {q, v} ∈ E do                     ▷ at most 1 unpaired neighbor exists
                if not isPaired[v] then
                    r ← v
            if r ==  None then return False
            isPaired[q] ← True
            isPaired[r] ← True
            rNumUnpairedN ← 0
            rUnpairedNeighbor ← None
            for all {r, v} ∈ E do                     ▷ check to see if there is a valid parent
                if not isPaired[v] then
                    rNumUnpairedN ← rNumUnpairedN + 1
                    rUnpairedNeighbor ← v
            if rNumUnpairedN == 1 then
                Q.push(rUnpairedNeighbor)
            else
                P.push(r)
        while P not empty do
            p ← P.pop()
            if isPaired[p] then
                continue
            numUnpairedNeighbors ← 0
            for all {p, v} ∈ E do
                if not isPaired[v] then
                    numUnpairedNeighbors ← numUnpairedNeighbors + 1
                if numUnpairedNeighbors > 1 then
                    P.push(p)
                    skip to next p in queue
            if numUnpairedNeighbors == 0 then return False
            else
                Q.push(p)                             ▷ node only has 1 unpaired neighbor
    return True
```

Correctness

This algorithm works by greedily attempting to create pairs of nodes starting from the leaves of the graph. Leaves only have one parent so if a graph is a perfect matching, the leaves will be paired with their direct parent. If the

algorithm encounters a node that has more than one possible matching at that point in time, it will add it to the P queue and "save it" later for processing. After every node in Q is either paired or pushed to P, we can re-examine the nodes in P to see if they are "resolved" (have 1 or 0 unpaired parent).

We will now show that a node in $v \in P$ will always eventually be resolved. By nature of the algorithm, v has more than one neighbor that is unclaimed. That means the degree of v: $d(v) \geq 3$: one neighbor from the claimed path that pushed v to P, and a minimum of 2 unpaired nodes, u and w. Since the graph is acyclic, we know that u and w are not connected and that they branch off into seperate paths that lead to a leaf (u_ℓ and w_ℓ) as going down any acyclic path will eventually lead to a leaf node (there is no cycle to 'loop' back around). We know that u_ℓ and w_ℓ were added to the Q queue as they are leaf nodes. This means that eventually, the algorithm will go down the paths started by these leaves and pair u and w with other partners (or fail before-hand). We know that there cannot be a gridlock of unresolved nodes as the graph is acyclic so all nodes have a path that will claim them (or fail in the case of a non-perfect matching graph).

Elements in P will eventually get resolved and Q will eventually visit all nodes to pair them or fail.

Running Time

$O(|V| + |E|)$

We note that if a node v is popped from Q, we know v will be paired with one if its neighbors and never be visited again. Each iteration in the Q and P while loops look at $O(|E|)$ edges each.