Please enter your name and uID below.

Name: Sabath Rodriguez

uID: u1254500

Submission notes

- **(PS 1 specific) This is a warm-up assignment. Full points will be given for answers that have some correct elements and that clearly been given a legitimate effort.**

- *Solutions must be typeset* using one of the template files. For each problem, your answer must fit in the space provided (e.g. not spill onto the next page) *without* space-saving tricks like font/margin/line spacing changes.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the semester, so please retain the original files.
- Please remember that for problem sets, collaboration with other students must be limited to a high-level discussion of solution strategies. If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration. You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.

1. (Analysis of your coding solution)

I tried to approach my algorithm to be as close as possible to the representation in the textbook because of the sheer simplicy and utility of it. I also initially had far more data structures and object than I needed and I think it can sometimes be nice to have the given complexity constrains, that way it forces us to be more creative with our algorithms. For example I ended up using very basic data structures to represent my graph algorithm, I used a stack to hold my "bag", a 2D Boolean array to keep track of the nodes I had visited, a starting location using two ints (an X and a Y), two local variables to keep track of the current best placement for a new monster, as well as a few global variables to keep track of the global best location for a new monster as well as the associated best score(min number of treasure). I used a 2D string array to represent my graph, I took the input and converted into a 2D array, the elements in the outer arrays with representative of the rows, and the elements in the inner array(s) were representative of the columns. My vertices were the individual elements in the inner array of my 2D array, for example: 1.2.3.4.5.6.7.8.9.

```
1.{{"#",".",".","#"},
2.{"#","3",".","#","#"}
...
```

Here my vertices were represented by "#, ".", and digits, there was no need for explicit nodes, it was useful that the input was structured similar to that of a graph, this made traversing the graph relatively straight forward. My edges were essentially the relationships between two vertices, for example, I know that there's an edge on line 1 element 2 to line 1 element 3, essentially all adjacent nodes had edges between each other, that was the only way of traversing the graph, one node and one edge at a time. My WFS algorithm was implemented by first finding the starting position in my graph, afterwards, checking all adjacent nodes if they had been previously visited, and if they haven't then we would add them to a stack, that way we know which vertices to visit next after we finish investigating our current node, we would then repeat this until all nodes that are visitable had been visited. While performing my WFS algorithm, we kept track of how much treasure the player collected by a local variable, and so every time we encountered a digit we would increment our treasure collected, if there was a monster nearby then we would simply collect the variable without adding that vertex to our stack, and therefore, not investigate that node further, we would also mark it as visited. The way I found the best location to place my new monster was my placing a new monster at every possible insertable position and then running WFS and collecting all possible treasure, the way I kept of the best possible solution was through a global variable keeping track of the current min treasure and it's associated location, which translated to the location of a new monster in order to get that amount of treasure, and if we found a new best min, then we would update it with the treasure and the new monster location.

2. (Bipartite Graphs)

3.c) An efficient algorithm to determine if a given graph is bipartisan would be as follows: create some kind of bag or collection, but create 2, one called L and one called R. Afterwards we should do a WFSAll(v) and add vertices to our bag(s) according to a few criteria, we must make sure that each bag does not contain any adjacent nodes to each other, meaning the vertices in each bag, in L and R, must not be adjacent. I will describe WFSAll(v) below. Once we have visited all vertices in our graph, we can then check if each node in each bag connects to the opposing bag, or better said, can $\forall l \in L$ have an adjacent node in R, and vice versa, if this is the case, then we should have a bipartisan graph.

```
WFSAll(v)
    unvisit all V
    create L bag
    create R bag
    add v to L
    for all V
        if v is unvisited:
        perform a WFS(v) and add all nodes to L that are not adjacent to v
        perform another WFS(v) and add all nodes to R that are not already in L
    for all u ∈ L
        if u has no adjacent vertex to any vertex in R
            return false
    for all u ∈ R
        if u has no adjacent vertex to any vertex in L
            return false
    return true
```

3.a) I will attempt to prove that every tree is a bipartite graph. We know that a bipartite graph is partitioned into 2 parts, either a L and R, or red and blue, whichever method we use to label our separate partitions is unimportant, but we must distinguish between the 2 subgraphs. We know that a tree by definition is a graph without any cycles and each node in our tree has a parent, other than itself. If we know that in a bipartite graph each node in our separate partitions has no adjacent nodes to any nodes within that partition, we can use that rule to put each level in our tree into separate parts, our L and R, because know that since each node has a parent, and that node has a parent, meaning: *v->parent(v)->parent(parent(v))->...* that v and parent(parent(v)) are not directly adjacent, we can put them in 2 separate parts without them having a direct adjacent node, however we still know that v and parent(parent(v)) are connected by parent(v), and since both v and parent(parent(v)) are in the same group, and parent(v) is in another group, and we also know that they have an edge from parent(v)->parent(parent(v)) and v->parent(v), and this is true for all nodes in our tree, meaning if we continue adding nodes from each level to separate bags or parts, we will nodes that continue this relationship and we will therefore have a bipartisan graph, and we know that all trees follow the same structure of each node having a parent node other than the root node, and there are no cycles in a tree, then we will have the same result in all possible trees, and that result is they are bipartisan graphs.

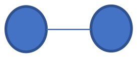
3. (Connected Graphs)

I will prove that if we have an undirected graph with 2k vertices and each vertex must have at least a degree of k must be connected, and I will prove this using Induction. I will start with my base case:

$k = 1$

$|V| = 2k = 2$

$\deg(v) \geq 1$



This will indeed give us a connected undirected graph. Now I will assume that if we have any number *t*, that it will give us a connected graph:

$k = t$

$|V| = 2t$

$\deg(v) \geq t$

We will *assume* this to be true, the next step is going to be to show that this will hold for any number:

$k = t + 1$

$|V| = 2(t + 1) = 2t + 2$

$\deg(v) \geq (t + 1)$

If we have a connected graph, that means that every vertex in our undirected graph is reachable from any other vertex within our graph, or simply, there exists a walk from any node in our graph to any other node in our graph. If we create 2 equally sized clusters, and we know we can create 2 even clusters because 2k will always give us an even number and therefore every even number is divisible by 2, and therefore we can divide our graph into 2 clusters of equal size. If we have 2 clusters of equal size, meaning 2 clusters each of size *t+1*, this coming from our previous equation of *2(t+1)*, giving us the number of vertices in our graph, and we know that we have to have at least *t+1* degrees per vertex, we know that we must have an outgoing edge to the other cluster. This is simply because in any graph, we cannot have any self-edges, and therefore giving us the max number of edges (degrees) a vertex can have is *|V|-1* and if we have t+1 vertices in each cluster, then we know that each node must have a degree of at least the number of vertices in the cluster, which we know is not possible due to our earlier statement of "each node can have a max degree of *|V|-1…"* but we are wanting to know what happens if we have at least k adjacent vertices at each vertex*,* that means that each vertex in our disconnected component must have an outgoing edge to our other disconnected component, and therefore connecting the 2 disconnected components. In other words, if we have 2 subgraphs of size t+1, and each subgraph requires at least t+1 degrees per vertex, then by definition, we must connect our subgraph to another node or subgraph because we cannot have t+1 adjacent nodes in each subgraph, and therefore will give us a connected graph. We have proven that if we have 2k vertices and each vertex must have a degree of at least k, then we must have a connected graph.