INF-2 CWR2

s2265910 JIAJUN LI

March 2023

1 Part A

1.1 Ai

The subplot(from Figure 1 to Figure 4) of the execution of GreedyIS on the first graph.

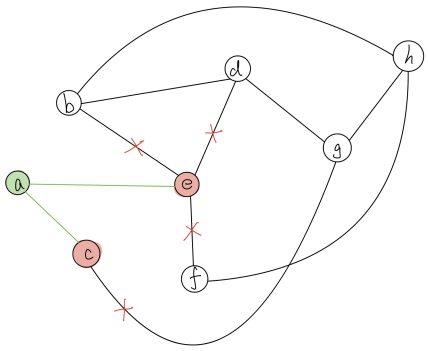

Figure 1: The original plot.

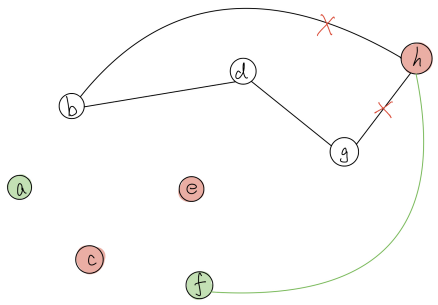

Figure 2: The picked node and its neighbours in second iteration.

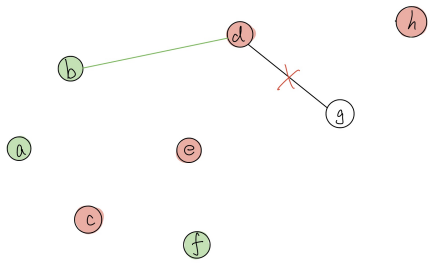

Figure 3: The picked node and its neighbours in third iteration.

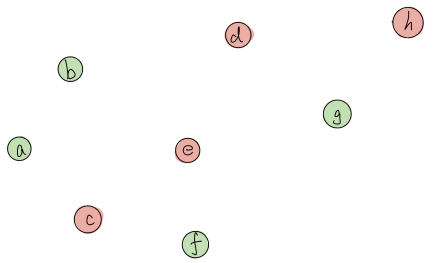

Figure 4: The picked node and its neighbours in fourth iteration.

1.2 Aii

The leftover subgraph(from Figure 6 to Figure 9) of interest after each iteration of the while loop of Figure 5 using the weighted GreedyIS (in other words, using criterion (a)). Figure 5 is the original graph with each node's weight.

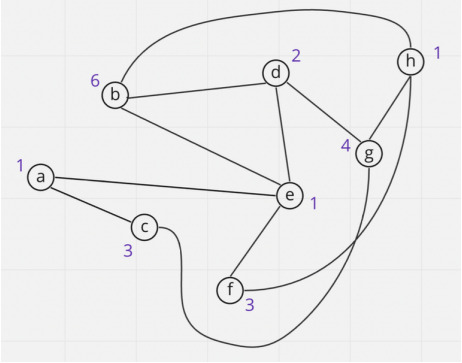

Figure 5: The graph with each node's weight.

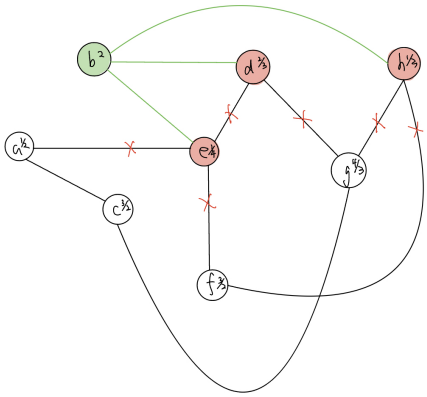

Figure 6: The picked node and its neighbours in first iteration using criterion (a).

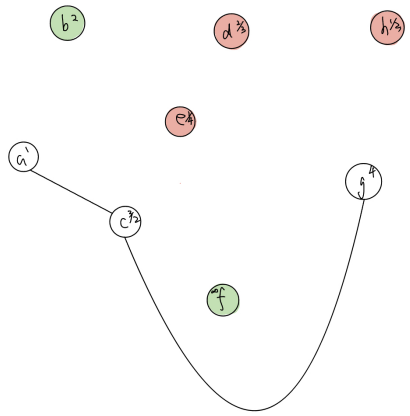

Figure 7: The picked node and its neighbours in second iteration using criterion (a).

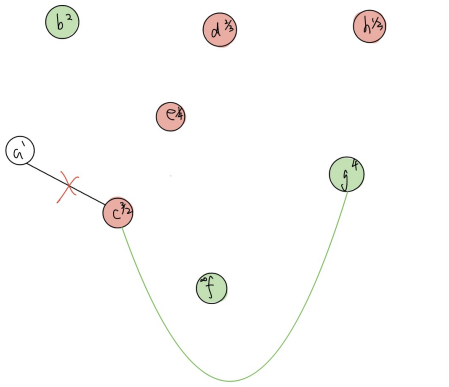

Figure 8: The picked node and its neighbours in third iteration using criterion (a).

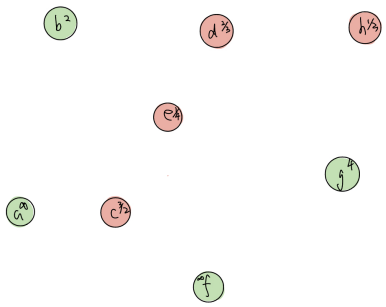

Figure 9: The picked node and its neighbours in fourth iteration using criterion (a).

1.3 Aiii

The leftover subgraph(from Figure 11 to Figure 14) of interest after each iteration of the while loop of Figure 5 using the weighted GreedyIS (in other words, using criterion (b)). Figure 5 is the original graph with each node's weight.

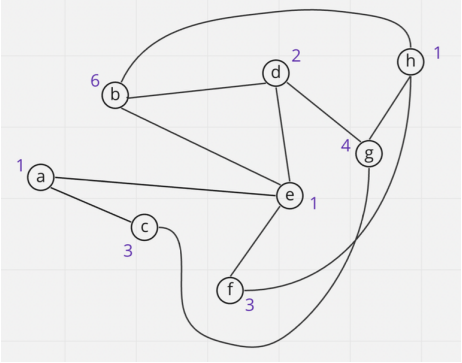

Figure 10: The original graph with each node's weight.

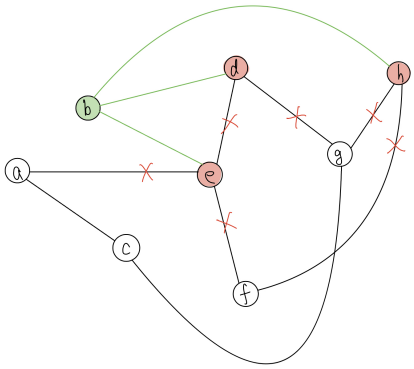

Figure 11: The picked node and its neighbours in first iteration using criterion (b).

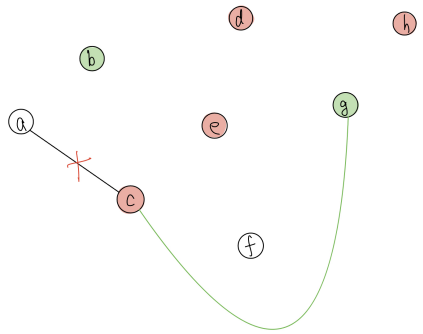

Figure 12: The picked node and its neighbours in second iteration using criterion (b).

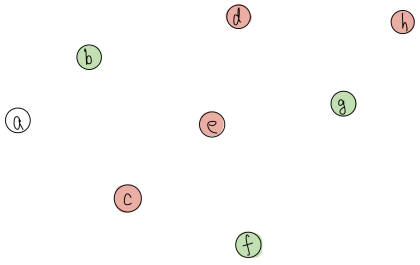

Figure 13: The picked node and its neighbours in third iteration using criterion (b).

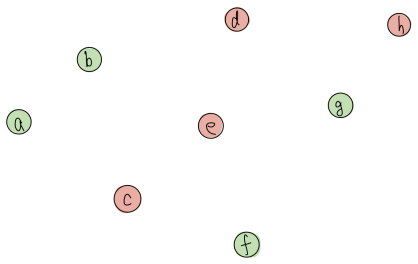

Figure 14: The picked node and its neighbours in fourth iteration using criterion (b).

1.4 Aiv

Here, in line 1, we get the number of vertex, which take $O(n)$ times. In Line 2, we initialize IS and deg list, which also take $O(n)$ times. In Line 3 and 4, we set the initial deg values for all nodes, which takes $O(n)$ times. In Line 5, the worst case that we might got is no connection between any two nodes, so it takes $O(n)$ times. In Line 6, the worst case is that we need to check all nodes, so combined with outer iteration, it takes $O(n^2)$ times. In Line 7, it takes $O(n)$ times combined with outer iteration. From Line 8 to Line 10, we set the picked the IS of node's neighbour to be -1 and delete the edges of deleted nodes, so the worst case is that all the nodes' edge need to be deleted, which takes $O(n+m)$ times. From Line 11 to Line 12, we correct the output set IS, which takes $O(n)$ times.

$$O(n) + O(n) + O(n) + O(n^2) + O(n) + O(n+m) + O(n) = O(n^2 + m).$$

Algorithm GreedyIS(G=(V, E))

1. $n \leftarrow |V|$
2. Adj is an adjacency list structure for E; IS and deg are initialised to all-0s
3. **for** $u \leftarrow 0$ **to** $n-1$ **do**
4. $\quad deg[u] = len(Adj[u])$ // (set the initial deg values for all nodes)
5. **while** (some vertices remain in the graph) **do**
6. $\quad u \leftarrow \arg\min\{deg[v]; v \in V, IS[v] == 0\}$ // (node with min *residual* degree
7. $\quad IS[u] \leftarrow 1$ // gets added to the IS)
8. $\quad$ **for** $(w \in Nbd(u))$ **do**
9. $\quad\quad IS[w] \leftarrow -1$ // (mark u's neighbours as disallowed)
10. $\quad\quad$ Update Adj, deg to reflect deletion of $\{u\} \cup Nbd(u)$
11. **for** $u \leftarrow 0$ **to** $n-1$
12. $\quad IS[u] \leftarrow \max\{0, IS[u]\}$ // (tidy: delete marks of disallowed vertices)
13. **return** IS

Algorithm

Figure 15: The Greedy Independent Search Algorithm.

1.5 Av

Here, in Figure 17, it is a simple example of an unweighted *bipartite* graph $G = (L \cup R, E)$, with $L = \{u_1, \ldots, u_{\lceil n/3 \rceil}\} \cup \{w_1, w_2, w_3, w_4\}$, and $R = \{v_1, \ldots, v_n\}$.

The edge set $E = E_1 \cup E_2$ consists of the following edges:

$$\begin{aligned} E_1 &= \{(u_i, v_i) : i = 1, \ldots, \lceil n/3 \rceil\} \cup \\ &\quad \{(u_i, v_{i+\lceil n/3 \rceil}) : i = 1, \ldots, \lceil n/3 \rceil\} \cup \\ &\quad \{(u_i, v_{i+2\lceil n/3 \rceil}) : i = 1, \ldots, \lceil n/3 \rceil\} \\ E_2 &= \{(w_i, v_j) : i = 1, \ldots, 4, j = 1, \ldots, n\} \end{aligned}$$

Here, it is clear that the upper nodes all have 3 edges, the middle nodes have 5 edges and the bottom nodes have 9 nodes. We can normalize this graph with $n = 9$. Then, the upper nodes all have 3 edges, the middle nodes have 5 edges and the bottom nodes have n nodes. From the GreedyIS search, it is clear that we will pick all upper nodes first, since they have less edges. Picking the upper nodes will make no effect to remaining nodes in middle, but the bottom nodes' edges will minus three. If we have picked all upper nodes, then there will be no middle nodes, since they are neighbours of the upper nodes. Then, we will pick all bottom nodes. Finally, we get $\lceil n/3 \rceil + 4$ nodes in IS. However, the true maximum IS is picking all the middle nodes with n nodes. Thus,

$$\frac{IS}{TrueMaximumIS} = \frac{\lceil n/3 \rceil + 4}{n}.$$

If we take n to infinity, we will get

$$\lim_{x \to \infty} \frac{\lceil n/3 \rceil + 4}{n} = \frac{1}{3}.$$

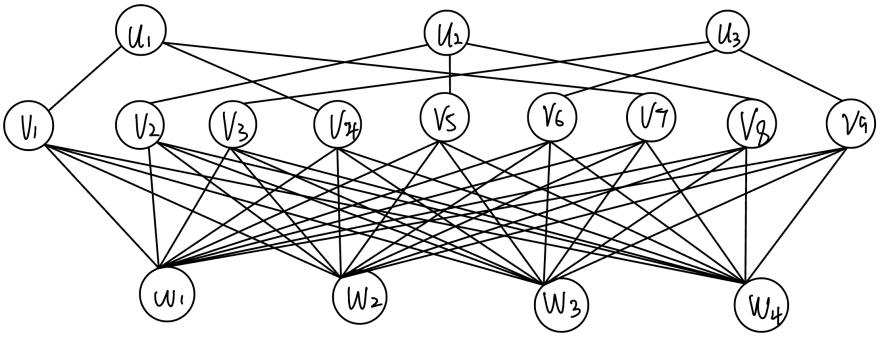

Figure 16: The Greedy Independent Search Algorithm.

1.6 Avi

Here, we can get progressively worse "approximation factors" for the result returned by GreedyIS from the last question(Av). From the formula $\frac{IS}{TrueMaximumIS} =$

$\frac{\lceil n/3 \rceil + 4}{n}$, in order to make the "approximation factors" worse, we can enlarge the number of adjacent edges of each of the vertices $\{u_1, \ldots, u_{\lceil n/a \rceil}\}$ to another number such as 4, 5 or even larger. Thus, we will get approximate $\frac{1}{a}$, where a is the number of adjacent edges of each of the vertices $\{u_1, \ldots, u_{\lceil n/a \rceil}\}$. we can also make bottom nodes(w_i) be less to make the "approximation factors" a little bit worse.

2 Part D

2.1 Di

Here, we can consider $\kappa_{v,0}$ first. $\kappa_{v,0}$ means the maximum-weight Independent set of subtree Tv which does not include v itself. Thus, we only need to make its children nodes to have a maximum-weight Independent set whether or not its children' maximum-weight Independent set include its children themselves. Thus, we can get $\kappa_{v,0}$ by $\kappa_{v,0} = \sum maxmium(\kappa_{child,0}, \kappa_{child,1})$. It is similar for $\kappa_{v,1}$, but this will be more simple. Since we must include the top node itself, its children should all not include themselves by the rule of Independent Set, which means all its children are $\kappa_{child,0}$. Thus, we can get $\kappa_{v,0} = W(v) + \sum \kappa_{child,0}$.

Here, it is easy to see that our pseudocode go from the bottom of the Tree to the root of the Tree. If the vertex is leaf, then we let $\kappa_{v,0}$ to be 0 and $\kappa_{v,1}$ to be W(v), since they do not have child. For the middle of nodes of the Tree, we use dynamic programming to computing the two recurrences from bottom to root. Finally, if we reach the root, we get the maximum of maximum-weight Independent set with root included and not included.

Algorithm 1 WIS

```
1:  function WIS(T)
2:      if v is leaf then
3:          κ_{v,0} ← 0
4:          κ_{v,1} ← W(v)                          ▷ W(v) is the weight of v
5:      else
6:          for each node v in T in postorder traversal except leaves v do
7:              for each child of v do
8:                  κ_{v,0} ← Σ maximum(κ_{child,0}, κ_{child,1})
9:                  κ_{v,1} ← W(v) + Σ κ_{child,0}
10:                 if v is the root of Tree then return max κ_{v,0}, κ_{v,1}
```

The running-time for this algorithm is simple. From the pseudocode, it is clear that WIS go through every nodes in the tree from bottom to top and do sum in each node, so the running-time will be $O(|V|)$

2.2 Dii

We first need to know the structure of the tree. Each child will have one edge with its parent and the root will not have this edge, which shows that there are n-1 edges in the whole tree. There should not be any cycle in the tree graph.

Algorithm 2 Is Tree

```
1: function IS_TREE(V, E)
2:     if |E| is not equal to |V| - 1  then return False
3:     else
4:         visited ← initial with an empty list        ▷ a list to record all visited nodes
5:         v ← a random node in tree
6:         queue ← [v]
7:         while queue is not empty do
8:             current_node ← the first element of queue
9:             remove the current_node in the queue
10:            add current_node to visited
11:            for each neighbour of current_node do
12:                if neighbour is not in visited then
13:                    add neighbour to visited
14:                    add neighbour to the back of queue
15:                else
16:                    return False
17:                end if
18:            end for
19:        end while
20:        return True
```

Here, we use BFS from an arbitrary node in the tree. We then expand this node to its neighbours and add all its neighbour to the back of queue. Then, we will go through all the nodes. Here, the visited list will help us to check whether there is a cycle in the tree. In other words, if there is a cycle in the tree, then the BFS will find a node visited twice.
The running-time of this code is also clear. We will go through each vertex to check each edge, so the running-time is $O(|V| + |E|)$.
If we want to convert such graph into a tree-like data structure. We need to find its root and build the whole tree by constructing all the children recursively till the leaves. We know that the node's neighbour are either parent or children. If we find from root, we can mark all parents and mark the result nodes as children. Here, the running-time is also $O(|V| + |E|)$, since we find all nodes and construct them with their nodes (equal to the number of edges).

2.3 Diii

By the criterion a, with a unweighted tree, the greedyIS will pick all leaves first. This means that we will pick the leaves and delete all their parents. Then, pick the new leaves in the remaining graph. It is same picking the grandparent of leaves with picking leaves, since if we pick leaves and delete all their neighbours, then the new leaves are the grandparents. Thus, we only need to care about the parents of the leaves.
Here, we can prove this by strong induction.

Proof. Base case: the tree only has one vertex.
In this case, the GreedyIS will only pick one node, which is also the only node in the graph, so it is true that GreedyIs is true maximum Independent Set in base case.

Normarl Case:
By strong induction, we assume that GreedyIs is successful for all $depth <= k$ and we need to prove it is also true for $depth = k + 1$.

We can get all sub-tree with $depth = 2$ from all leaves in the tree with $depth = k + 1$. Then, the remaining tree after deleting all sub-tree that we got before, is a tree with maximum $depth = k - 1$. The Independent Set of this remaining tree got from GreedyIs should be true maximum Independent Set, since we assumed before. All the sub-trees' Independent Set got from GreedyIS are also true maximum Independent Set, since the *depth* of them is 2 as the assumption before. All the sub-trees' root will not be chosen, so deleting all the sub-tree will not affect the remaining tree if we connect them on. Based on our discussion before, we know that all the trees that we split out are all true maximum Independent Set got from GreedyIS. If we connect them on, there will be one more restriction between each two of them. Thus, the Independent Set of the whole graph should be no greater than the sum of all split sub-trees. However, we discussed before that all the sub-tree will not affect the remaining leaves when they connect together under picking all the sun-trees with $depth = 2$ from leaves. Therefore, add all sub-trees together is the true maximum Independent Set of the graph with $depth = k + 1$.

Here, we also need to prove that the sum of all Independent Set added up together is the same as the Independent Set that we get from GreedyIS. The GreedyIS will pick all leaves of a graph and delete all parents of leaves, where the graph deleting all leaves and their parents is same as the graph that deleting all sub-trees with $depth = 2$ from all leaves in the tree. Thus, the independent set above is the same as the independent set got from GreedyIS.

Finally, since base case and normal case are all successful, we come out that the GreedyIS got the true maximum independent set from an unweighted tree graph. □