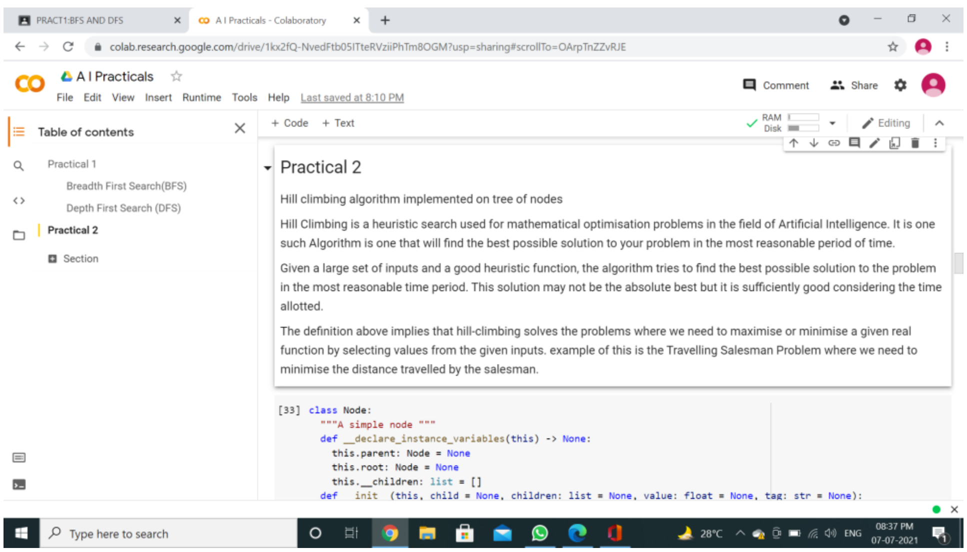
Name: Zeenat Class: TYIT Roll no:578

Practical 2

Aim: Hill climbing algorithm implemented on tree of nodes

Google colab link: https://colab.research.google.com/drive/1kx2fQ-NvedFtb05ITteRVziiPhTm8OGM#scrollTo=OArpTnZZvRJE




```
class Node:
  """A simple node """
  def __declare_instance_variables(this) -> None:
    this.parent: Node = None
    this.root: Node = None
    this.__children: list = []
  def __init__(this, child = None, children: list = None, value: float = None, tag: str = None):
    """child: Node, children: List[Node]"""
    this.__declare_instance_variables()
    this.tag = tag
    this.value = value
    if (child != None):
      this.add(child)
    if (children != None):
      this.add_children(children)

  def get_neighbors(this) -> list:
    """Returns the neighbor nodes"""
    if this.parent == None:
      return [this]
    children = this.parent.get_children()
    if children == None:
      return []
    return children
```

[33]

```
def get_first(this):
  """Returns the first children of this node"""
  if (this.is_empty): return None
  return this.__children[0]

def is_root(this) -> bool:
  return this.parent == None

def is_leaf(this) -> bool:
  if (this.__children == None): return True
  return this.is_empty()

def is_inner(this) -> bool:
  return not (this.is_leaf() or this.is_root())

def get_children(this) -> list:
  return this.__children

def get_root(this):
  """Returns -> Node"""
  if (this.is_root()):
    return this
  else:
    return this.parent.root
```

[33]

```
def get_height(this) -> int:
  if (this.is_empty()):
    return 0
  maxHeight: int = 0
  children: list = this.get_children()
  for element in children:
    height: int = element.get_height()
    if (height > maxHeight):
      maxHeight = height
  return maxHeight + 1

def get_depth(this) -> int:
  if (this.is_root()):
    return 0
  return this.parent.get_depth() + 1

def is_empty(this) -> bool:
  return len(this.__children) == 0

def is_not_empty(this) -> bool:
  return not this.is_empty()

def add(this, child) -> None:
  """child: Node"""
  assert child != None
```

```
[33]     if (this.__children == None):
           this.__children = []
         child.parent = this
         child.root = this.get_root()
         this.__children.append(child)

       def add_children(this, children: list) -> None:
         assert children != None
         if (len(children) == 0):
           return
         if (this.__children == None):
           this.__children = []
         for element in children:
           element.parent = this
           element.root = this.get_root()
           this.__children.append(element)

       def __len__(this) -> int:
         if (len(this.__children) != 0 and this.__children != None):
           maxLength: int = 1
           for child in this.__children:
               maxLength += len(child)
           return maxLength
         else:
           return 1
```

```
[33]   def __str__(this) -> str:
         return f"Node({this.value})"
```

```
[32] def node_to_string(node: Node, islast=False):
       pretab = '' if node.get_depth() == 0 else ' ' * (node.get_depth())
       prefix = f'{pretab}:{node.get_depth()} ──'
       value = node.value
       depthTab: str = ' ' * (node.get_depth() + 1)
       children_str = ''
       for child in node.get_children():
         ischildlast = node.get_children()[-1] == child
         children_str += f'{depthTab}{node_to_string(child, ischildlast)}'
       return (
           f'{prefix} {node.tag} = {value}\n'
           f'{children_str}'
       )
```

```
[31] def evaluate(node: Node):
       """returns the value of the node"""
       if(isinstance(node.value, float) or isinstance(node.value, int)):
         assert node.value != None, "Node must have a value"
         return node.value
       elif(isinstance(node.value, str)):
         raise NotImplementedError
```

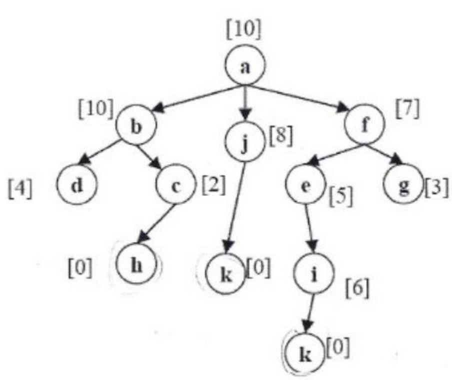
Simple hill climbing algorithm

```
[30] from math import inf
     def hill_climbing(start_node) -> Node:
       """
       Pseudo-code for the algorithm

       ```
       algorithm hill Climbing is
           currentNode := startNode
           loop do
               L := NEIGHBORS(currentNode)
               nextEval := -INF
               nextNode := NULL
               for all x in L do
                   if EVAL(x) > nextEval then
                       nextNode := x
                       nextEval := EVAL(x)
               if nextEval ≤ EVAL(currentNode) then
                   // Return current node since no better neighbors exist
                   return currentNode
               currentNode := nextNode
       ```
       """
```

```
[30]   current_node = start_node
       best_value = -inf
       best_node = None
       while True:
         current_value = evaluate(current_node)
         if current_value > best_value:
           best_node = current_node
           best_value = current_value
         else:
           # this node has a value smaller than the best node.
           # stopping search with local maxima
           return best_node
         childrens = current_node.get_children()
         for child in childrens:
           child_value = evaluate(child)
           if child_value > best_value:
             best_value = child_value
             best_node = child
           else:
             return best_node
           # every neighbour of this child is traversed.
           # Setting current_node as the last child traversed
           current_node = child
```

Graph



```
[29] # implemented the above tree of nodes
     tree1: Node = Node(
         value=10, tag='a',
         children=[
             Node(
                 value=10, tag='b',
                 children=[
                     Node(value=4, tag='d'),
                     Node(value=2, tag='c',
                          child=Node(value=0, tag='h')
                     ),
                 ]
             ),
             Node(value=8, tag='j',
                 child=Node(value=0, tag='k')
             ),
             Node(value=7, tag='f',
                 children=[
                     Node(value=5, tag='e',
                         child=Node(
                             value=6, tag='i',
                             child=Node(value=0, tag='k')
                         )
                     ),
                     Node(value=3, tag='g')
```

```
[29]             ]
             )
         ]
     )
```

```
[11] # implemented an another tree of nodes
     tree2: Node = Node(
         value=2, tag='a',
         children=[
             Node(
                 value=4, tag='b',
                 children=[
                     Node(value=5, tag='d'),
                     Node(value=6, tag='c',
                          child=Node(value=8, tag='h')
                     ),
                 ]
             ),
             Node(value=9, tag='j',
                 child=Node(value=0, tag='k')
             ),
```



```
[11]         Node(value=7, tag='f',
                 children=[
                     Node(value=12, tag='e',
                         child=Node(
                             value=6, tag='i',
                             child=Node(value=0, tag='k')
                         )
                     ),
                     Node(value=3, tag='g')
                 ]
             )
         ]
     )
```

```
[37] print('Tree - 1: representation')
     print('pattern -> :<depth> ── <value>', end='\n\n')
     print(node_to_string(tree1))
```

```
Tree - 1: representation
pattern -> :<depth> ── <value>

:0 ── a = 10
  :1 ── b = 10
    :2 ── d = 4
```

+ Code + Text RAM Disk Editing

```
[37]    :1 ── b = 10
          :2 ── d = 4
          :2 ── c = 2
            :3 ── h = 0
        :1 ── j = 8
          :2 ── k = 0
        :1 ── f = 7
          :2 ── e = 5
            :3 ── i = 6
              :4 ── k = 0
          :2 ── g = 3
```

```
[36] print('Tree - 2: representation')
     print('pattern -> :<depth> ── <value>', end='\n\n')
     print(node_to_string(tree2))
```

```
Tree - 2: representation
pattern -> :<depth> ── <value>

:0 ── a = 2
  :1 ── b = 4
    :2 ── d = 5
    :2 ── c = 6
      :3 ── h = 8
  :1 ── j = 9
    :2 ── k = 0
```

+ Code + Text RAM Disk Editing

```
[36]    :2 ── k = 0
      :1 ── f = 7
        :2 ── e = 12
          :3 ── i = 6
            :4 ── k = 0
        :2 ── g = 3
```

```
[35] print('For Tree - 1')
     best_solution = hill_climbing(tree1)
     print(f"Best solution is {best_solution.value} with tag {best_solution.tag}")
```

```
For Tree - 1
Best solution is 10 with tag a
```

```
[34] print('For Tree - 2')
     best_solution = hill_climbing(tree2)
     print(f"Best solution is {best_solution.value} with tag {best_solution.tag}")
```

```
For Tree - 2
Best solution is 9 with tag j
```