

Implémentation du papier de recherche : Dynamic Address Allocation Algorithm for Mobile Ad hoc Networks

Travail réalisé par : Ilyes Rezgui
Groupe: M1 recherche

Vous trouverez le code ici:

<https://colab.research.google.com/drive/19aRWrvVg17K7x9KuJwgY53jFe-DQIPk?usp=sharing>

1) Création de la structure de données pour stocker les nœuds.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

def create_tree():
    # Create the head node
    head = TreeNode("192.168.0.0")

    # Add children nodes to the head node
    for i in range(1, 256):
        child_value = f"192.168.{i}.0"
        child_node = TreeNode(child_value)

        # Add grandchildren nodes to each child node
        for j in range(1, 256):
            grandchild_value = f"192.168.{i}.{j}"
            grandchild_node = TreeNode(grandchild_value)
            child_node.children.append(grandchild_node)

        head.children.append(child_node)

    return head

def print_tree(node, indent=0):
    print(" " * indent + node.value)
    for child in node.children:
        print_tree(child, indent + 1)

# Create the tree
tree_head = create_tree()
# Print the tree
print_tree(tree_head)
```


Ce code Python crée une structure d'arbre représentant les adresses IP dans le contexte des adresses IP de classe C du réseau privé 192.168.0.0. L'arbre est créé avec des instances de la classe `TreeNode`, où chaque nœud représente une partie d'une adresse IP. Voici une explication du code :

La classe `TreeNode` : La classe `TreeNode` représente un nœud de l'arbre. Chaque nœud a une valeur (`value`), qui est une partie de l'adresse IP, et une liste d'enfants (`children`), qui sont d'autres nœuds.

La fonction `create_tree()` : Cette fonction crée un arbre avec pour racine le nœud représentant l'adresse IP "192.168.0.0". Ensuite, elle ajoute des nœuds enfants à la racine, représentant les adresses IP "192.168.1.0" à "192.168.255.0". Pour chaque nœud enfant, elle ajoute des nœuds petits-enfants, représentant les adresses IP "192.168.i.1" à "192.168.i.255".

La fonction `print_tree(node, indent=0)` : Cette fonction imprime l'arbre de manière récursive en utilisant un espacement proportionnel à la profondeur du nœud dans l'arbre. Elle commence par imprimer la valeur du nœud actuel avec un certain niveau d'indentation. Ensuite, elle appelle récursivement la fonction pour imprimer les enfants du nœud actuel.


Création et impression de l'arbre : L'arbre est créé en appelant la fonction `create_tree()`, et la racine de l'arbre est stockée dans la variable `tree_head`. Enfin, l'arbre est imprimé en appelant la fonction `print_tree(tree_head)`. En résumé, ce code génère et affiche une structure d'arbre représentant les adresses IP dans un réseau privé de classe C. Chaque niveau de l'arbre représente une partie de l'adresse IP, de la partie réseau à la partie hôte.



```
192.168.0.0
  192.168.1.0
    192.168.1.1
    192.168.1.2
    192.168.1.3
    192.168.1.4
  192.168.2.0
    192.168.2.1
    192.168.2.2
    192.168.2.3
    192.168.2.4
  192.168.3.0
    192.168.3.1
    192.168.3.2
    192.168.3.3
    192.168.3.4
  192.168.4.0
    192.168.4.1
    192.168.4.2
    192.168.4.3
    192.168.4.4
```

Exemple avec seulement 5 nœuds, dont chacun a 5 nœuds fils .

2) Recherche d'un noeud dans un cluster:



```
def search_node(root, target_value):
    if root is None:
        return False

    if root.value == target_value:
        return True

    for child in root.children:
        if search_node(child, target_value):
            return True

    return False

node_to_search = "192.168.255.254"
node_exists = search_node(tree_head, node_to_search)
print(node_exists)
```

True

1s



```
def search_node(root, target_value):  
    if root is None:  
        return False  
  
    if root.value == target_value:  
        return True  
  
    for child in root.children:  
        if search_node(child, target_value):  
            return True  
  
    return False  
  
node_to_search = "192.168.255.256"  
node_exists = search_node(tree_head, node_to_search)  
print(node_exists)
```

False

- Pour noeud 192.168.255.254 : existe dans le cluster donc la fonction retourne True
- Pour noeud 192.168.255.256 : n'existe dans le cluster donc la fonction retourne False

3) Supprimer un noeud dans un cluster:

Ce code Python définit une fonction `delete_node` pour supprimer un nœud spécifique d'un arbre représentant des adresses IP.

L'arbre est supposé avoir été créé par une fonction `create_tree()` qui n'est pas fournie dans ce code, mais dont l'utilisation est illustrée.

Voici une explication détaillée du code :

`delete_node` fonction: Cette fonction prend en entrée la racine de l'arbre (`root`) et une valeur cible (`target_value`) correspondant à la valeur d'un nœud que l'on souhaite supprimer. Si la racine est `None`,

cela signifie que l'arbre est vide, et un message est affiché indiquant que le nœud cible n'existe pas. Si la valeur de la racine est égale à la valeur cible, cela signifie que la racine doit être supprimée, et la fonction retourne None. Sinon, la fonction est appliquée récursivement à tous les enfants de la racine, en supprimant le nœud cible de chaque sous-arbre. Ensuite, elle filtre les enfants pour supprimer les nœuds qui ont été marqués comme None (ceux qui ont été supprimés) et retourne la racine mise à jour.

`print_tree` function: Cette fonction imprime l'arbre de manière récursive avec une indentation pour représenter la structure arborescente.

```
# Assume the create_tree() function is defined elsewhere
def delete_node(root, target_value):
    if root is None:
        print(f"Node {target_value} doesn't exist. Cannot delete.")
        return None
    if root.value == target_value:
        return None
    root.children = [delete_node(child, target_value) for child in root.children]
    root.children = [child for child in root.children if child is not None]
    return root

def print_tree(node, indent=0):
    print(" " * indent + str(node.value))
    for child in node.children:
        print_tree(child, indent + 1)

# Example usage:
tree_head = create_tree() # You need to implement the create_tree function

# Delete a specific node, for example, "192.168.255.256"
node_to_delete = "192.168.255.255"
if search_node(tree_head, node_to_delete) == True:
    deleted_tree_head1 = delete_node(tree_head, node_to_delete)
    print("\nTree after deletion of", node_to_delete + ":")
    if deleted_tree_head1 is not None:
        print_tree(deleted_tree_head1)
    print("node deleted successfully")
else:
    print("node doesn't exist")
```

Exemple de suppression ou le noeud n'existe pas :

```
# Delete a specific node, for example "192.168.255.256"
node_to_delete = "192.168.255.256"
if search_node(tree_head, node_to_delete) == True:
    deleted_tree_head1 = delete_node(tree_head, node_to_delete)
    print("\nTree after deletion of", node_to_delete + ":")
    if deleted_tree_head1 is not None:
        print_tree(deleted_tree_head1)
    print("node deleted successfully")
else:
    print("node Doesn't exist")
```

node Doesn't exist

Exemple de suppression ou le noeud existe:

- Suppression du noeud 192.168.255.255

```
192.168.255.250
192.168.255.251
192.168.255.252
192.168.255.253
192.168.255.254
```

node deleted successfully

4) Ajouter un noeud a un cluster:

node_exists function: Cette fonction vérifie si un nœud avec une valeur spécifique (target_value) existe dans l'arbre à partir d'un nœud donné (root). Si le nœud actuel (root) est None, cela signifie que l'arbre est vide, et la fonction retourne False. Si la valeur du nœud actuel est égale à la valeur recherchée (target_value), la fonction retourne True. Sinon, la fonction est appliquée récursivement à tous

les enfants du nœud actuel, et elle retourne True si le nœud recherché est trouvé dans l'un des enfants, sinon elle retourne False.

add_node function: Cette fonction ajoute un nouveau nœud à l'arbre à partir d'un nœud parent spécifié (parent_value). Elle commence par vérifier si le nouveau nœud existe déjà dans l'arbre en utilisant la fonction node_exists. Si le nœud existe, un message est affiché, et la fonction retourne la racine inchangée. Si la racine est None, cela signifie que l'arbre est vide, et la fonction retourne None. Si la valeur de la racine est égale à la valeur du nœud parent, un nouveau nœud est créé avec la valeur spécifiée (new_value) et ajouté à la liste des enfants du nœud parent. Sinon, la fonction est appliquée récursivement à tous les enfants du nœud actuel.

- L'arbre est créé à l'aide de la fonction create_tree. Le nœud avec la valeur "192.168.255.3" est supprimé de l'arbre à l'aide d'une fonction hypothétique delete_node. Ensuite, un nouveau nœud ("192.168.255.3") est ajouté sous le nœud parent spécifié ("192.168.2.0") en utilisant la fonction add_node. Un message est affiché indiquant si le nœud a été ajouté avec succès.

Exemple d'ajout d'un noeud qui existe déjà :

```
print("\nTree after adding", new_node_value, "under", parent_node + ":")  
# Note: The tree itself is not reprinted here, only the message is printed.
```

Original Tree:

192.168.0.0

Node 192.168.255.3 already exists. Not adding.

Exemple d'ajout d'un noeud qui n'existe pas :

J'ai supprimé le nœud, puis je l'ai ajouté dans cet exemple

```
# Add a new node only if it doesn't already exist, for example, add "192.168.2.0" under "192.1  
delete_node(tree_head, "192.168.255.3")  
parent_node = "192.168.2.0"  
new_node_value = "192.168.255.3"  
tree_head = add_node(tree_head, parent_node, new_node_value)  
  
# Print a message if the node exists, without re-showing the tree  
# (the tree itself is not reprinted in this case)  
print("\nTree after adding", new_node_value, "under", parent_node + ":")  
# Note: The tree itself is not reprinted here, only the message is printed.
```

⇒ Original Tree:
192.168.0.0

Tree after adding 192.168.255.3 under 192.168.2.0:

Merci pour votre attention,
Ilyes REZGUI