Dossier développement efficace

Jeu de stratégie en console

Luc FILIAO
Nathan LIEGEON
G1S3

Objet du projet

Ce projet vise à implémenter les structures de données suivantes en Python :

- Arbre
- Pile
- File
- Liste chaînée

Pour ce faire, nous avons choisi de réaliser un jeu de stratégie en console.

Explication du jeu

Le jeu se joue en console, sur un seul ordinateur. Il se joue au tour par tour, et les joueurs se passent l'ordinateur en fin de tour.

Comment gagner ?

Chaque joueur est un seigneur régnant sur une cité. Pour gagner, il faut avoir le score le plus élevé dans l'une de ces catégories dans le nombre de tours impartis :

- Domination : Les points de domination représentent les prouesses guerrières, et sont gagnés en attaquant les autres joueurs
- Richesse : Les points de richesse représentent le développement économique de sa cité (à travers la construction de marchés, de ports, etc…)
- Savoir : Les points de savoir représentent les technologies débloquées.

Seul le score le plus élevé de chaque joueur est comptabilisé pour déterminer la victoire, il existe donc plusieurs façons de gagner.

Composition d'une cité

Les joueurs commencent tous avec les mêmes ressources. Une cité est composée de bâtiments, qui peuvent être construits avec des ressources. Ces bâtiments peuvent eux-même permettre de rapporter des ressources (tous les *x* tours), ou donner accès à de nouvelles fonctionnalités.
Les cités ont aussi une limite de population, qui peut être étendue en construisant des habitations.

Bâtiments spéciaux

Habitations : Augmente la capacité maximale de la cité.
Caserne : Permet de former des soldats.
Temple : Permet de prêter allégeance à une divinité, et de débloquer certains bonus (une seule divinité autorisée par cité).
Marché : Permet d'échanger avec des marchands itinérants, ou d'autres joueurs. (Non implémenté)

Utilisation des structures de données

Les structures implémentées sont les suivantes :

Arbre

L'arbre permet de stocker toutes les technologies existantes, et d'ordonner l'obtention de celles-ci. Débloquer une Technologie permettra de consulter et de débloquer les technologies qui en découlent, permettant d'organiser la progression en proposant des Technologies de plus en plus puissantes, tout en laissant le choix à l'utilisateur de prioriser la technologie qu'il juge la plus importante.
De plus, la navigation de l'utilisateur dans les différents menus est également un arbre, où chaque nœud correspond à la visualisation d'un prompt / d'une information.

Liste chaînée

Les listes chaînées sont utilisées à de multiples reprises, par exemple :

- Stocker les bâtiments d'une ville
- Stocker les troupes d'un joueur (soldats)
- Stocker les troupes sélectionnées par le joueur lors des batailles
- etc…

File

Utilisées pour les tours des joueurs (le joueur actuel est récupéré, et renvoyé à l'arrière, afin d'effectuer la rotation). Tous les **"% nbJoueurs == 0"**, le compteur de tour est incrémenté.

Pile

La pile est utilisée pour le parcours de l'arbre afin de garder en mémoire tous les nœuds parcourus lors des multiples changements de vue. Chaque vue propose de naviguer vers ses enfants, et une option "**Retour**" qui permet de retourner à son parent. Plutôt que de devoir rechercher le parent depuis la racine à chaque retour en arrière, un noeud est placé dans une pile à chaque fois que l'on se dirige vers l'un de ses enfants, ainsi, pour revenir au noeud précédent, il suffit de lire le haut de la pile (peut s'apparenter à un historique).

Classes variées

De nombreuses classes ont été implémentées afin de supporter les différentes fonctionnalités du jeu, et ne seront pas traitées ici. On peut notamment citer :

- **Player** : Informations des joueurs
- **City** : La cité du joueur
- **Resources** : Représentant des ressources, un apport de ressources ou un coût
- **Facility** : Un bâtiment de la cité
- **Troup** : Un soldat

Fonctionnement de l'arbre d'actions

Cet arbre représente donc les actions que l'utilisateur peut effectuer. Chaque nœud comporte différents attributs d'affichage (**name**, **title**, **desc**) qui serviront à l'affichage console, ainsi qu'un attribut **required**. Cet attribut peut être null, ou un tableau de taille deux, comprenant :

- En premier : le nom de la technologie nécessaire pour accéder à cette action.
- En deuxième : le nom d'un type de bâtiment, à avoir absolument construit pour accéder à cette action.

L'un et l'autre pouvant éventuellement être nuls.

Ainsi, avant chaque affichage d'un nœud, on vérifie lesquels sont actuellement accessibles par l'utilisateur, puis on les affiche. Si il débloque la technologie adéquate, alors les enfants seront affichés.

```
{
    "name" : "Renforcement",
    "desc" : "Formez des soldats afin de protéger votre cité... ou attaquer vos adversaires.",
    "type" : "TechnologyNode",
    "cost" : 1,
    "children" : [
        {
            "name" : "Minage",
            "desc" : "Apprenez à creuser et récoltez les ressources souterraines.",
            "type" : "TechnologyNode",
            "cost" : 2,
            "children" : []
        },
        {
            "name" : "Forge",
            "desc" : "Apprenez à frapper le métal, et améliorez vos unités.",
            "type" : "TechnologyNode",
            "cost" : 3,
            "children" : [

            ]
        }
    ]
}
```

L'attribut **callback** correspond à une fonction à appeler à cette étape de la navigation (à travers un switch/case ma foi pas très élégant).

Complexités

Dans cette partie seront démontrées les complexités des fonctions estimées les plus intéressantes, ou jouant un rôle crucial dans le projet.

Il est important de noter que les diverses fonctions **length**, **size**, etc… Permettant de connaître la taille des structures que nous avons codé ne représentent pas un parcours total de la structure. En effet, nous avons jugé plus sage de créer un attribut **len : int**, servant de

compteur. À chaque insertion ou suppression réussie, ce compteur est modifié, permettant un accès direct à la taille de n'importe quelle structure.

Arbres

isUnlocked

Cette fonction prend une chaîne de caractères en argument et retourne **True** si la technologie a été trouvée et est débloquée. Cette fonction est utilisée à de nombreuses reprises afin de vérifier si l'utilisateur a le droit de faire telle ou telle action.

```
def isUnlocked(self, technologyName : str|None) -> bool|None :
    """
    Returns True if the technology is unlocked, else False.
    Must enter the technology name.
    """
    node = self.getNodeByName(technologyName)
    if node is not None :
        return node.unlocked
    return True
```

```
def getNodeByName(self, name : str) -> _TreeNode|None :
    """
    Finds a node in the Tree
    returns the Node if it exists, None if it doesn't
    """
    if self.root is not None :
        return self.root.searchChild(name)
    return None
```

```
def searchChild(self, name : str) -> Self:
    stack = Stack()
    stack.push(self)
    while not stack.isEmpty() :
        node = stack.pop()
        if node.name == name :
            return node
        nodeList = node.children.getValuesInRange()
        for i in range(nodeList.size()):
            stack.push(nodeList.pop())
```

Complexité :

La fonction **isUnlocked** fait dans un premier temps appel à la fonction **getNodeByName**, faisant elle-même appel à la fonction **searchChild**. La complexité dépend donc de cette dernière.

La fonction **searchChild** est composée de trois boucles consécutives, toutes trois imbriquées dans une quatrième boucle.

En se plaçant dans le pire des cas, s'exécute i fois, avec n le nombre d'enfants du nœud. La deuxième boucle s'exécutera également i fois, étant donné qu'elle parcourt la File retournée afin d'en extraire chaque valeur. La troisième boucle s'exécute également i fois, puisqu'elle parcourt la File afin d'insérer chaque valeur dans une Pile.

Pour finir, la quatrième boucle englobante s'exécute tant que la valeur cherchée n'est pas trouvée. Dans le pire des cas, la valeur recherchée se trouve à la toute fin de l'arbre. Ce qui signifie que la boucle s'exécutera autant de fois que la taille de l'arbre, que nous appellerons k.
Pour conclure, la recherche d'une valeur dans l'arbre donne une complexité de (3i)*k, soit une complexité en O(n).

```
def getValuesInRange(self, indexStart : int = 0, indexEnd : int | None = None) -> Queue:
    queue = self.getNodesInRange(indexStart, indexEnd)
    for i in range(queue.size()):
        queue.push(queue.pop().value)
    return queue


def getNodesInRange(self, indexStart : int = 0, indexEnd : int | None = None) -> Queue:
    indexEnd = self.len-1 if indexEnd is None else indexEnd

    ret = Queue()
    if indexStart < 0 or indexStart >= self.len or indexEnd < 0 or indexEnd > self.len or indexEnd < indexStart:
        return ret

    if (indexStart <= self.len/2):
        i = 0
        current = self.head
        while (i <= indexEnd):
            if i >= indexStart:
                ret.push(current)
            current = current.next
            i += 1
    else:
        i = self.len-1
        current = self.tail
        while (i >= indexStart):
            if i <= indexEnd:
                ret.push(current)
            current = current.prev
            i -= 1

    return ret
```

Navigate

Cette fonction est utilisée pour faire naviguer l'utilisateur dans un arbre, que ce soit l'arbre d'action ou l'arbre de technologies.

```
def navigate(self) -> None :
    """
    Displays the tree in a little menu that lets the player simply navigate through it,
    read information about Technologies he has access to, or unlock new ones.
    author : Nathan
    """
    if self.root is None :
        print("L'arbre est vide...")
    else :
        self._processNavigation()
```

Complexité :

La fonction **navigate** fait appel à la fonction **_processNavigation** (qui peut être redéfinie dans les enfants de **_Tree**, afin d'avoir une navigation personnalisée), la complexité dépend donc de cette dernière. Ci-dessous la fonction redéfinie dans la class **ActionTree** :

```gdscript
def _processNavigation(self) -> None :

    node = self.root
    nodeStack = Stack()
    done = False

    nodeStack.push(node)

    while not nodeStack.isEmpty() :

        promptStatus(node.title)
        print(node.desc + "\n")

        self._execCallback(node, nodeStack)

        unlockedNodes = self._getUnlockedChildren(node)

        select = userInputInt(unlockedNodes.printChildren(nodeStack.size() != 1), 0, unlockedNodes.children.len)

        # Get back
        if select == 0 :
            node = nodeStack.pop()
            done = True if nodeStack.isEmpty() else False
        else :
            nodeStack.push(node)
            node = unlockedNodes.children.get(select - 1)

    Campaign.getInstance().nextPlayer()
```

```gdscript
def _getUnlockedChildren(self, node : ActionNode) -> ActionNode :

    unlockedChildren = ActionNode(node.name, node.title, node.desc, node.callback, node.required)

    for i in range(node.nbChildren()) :
        n = node.children.get(i)
        if (n.required is None or (n.required[0] is None or self.player.technoTree.isUnlocked(n.required[0]))
            and (n.required[1] is None or self.player.city.contains(n.required[1]))) :
            unlockedChildren.children.add(n)

    return unlockedChildren
```

Cette fonction est composée de deux boucles imbriquées, dont sa boucle boucle externe qui est une boucle tournant tant que l'utilisateur n'a pas décidé de finir son tour. L'étude de la complexité de cette boucle sera donc ignorée. Les trois opérations principales se répétant durant la navigation sont l'exécution du **callback**, la recherche des enfants débloqués (**unlockNodes**) et les affiche, et ensuite l'insertion du noeud courant sur la pile dans le cas ou on visite un de ses enfants, sinon le dernier noeud devient la dépile de la pile.
La complexité de la fonction **callback** sera ignorée, puisque les fonctions appelées sont trop diverses et variées, nous allons partir du principe qu'il n'y a pas d'appel à un **callback** sur le nœud courant.

Dans le pire des cas, la fonction est alors composée de deux boucles consécutives, parcourant l'entièreté des enfants à deux reprises dans le pire des cas (une fois pour sélectionner les enfants débloqués, et l'autre fois pour rechercher l'enfant sélectionné), soit i fois.

Au final, chaque opération de l'utilisateur lancera dans le pire des cas 2*i opérations, soit une complexité en O(n).

_build

Cette fonction créer un arbre à partir d'un fichier JSON.

```python
def _build(self, jsonPath : str|None) -> bool :
    """
    Builds the tree from a json file, it will first create the root, then recursively
    let each node create its children
    author : Nathan
    """
    if jsonPath is not None :
        with open(getcwd() + jsonPath) as jsonFile :
            data = load(jsonFile)
            root = data["root"]
            if root is None :
                return False

            self.root = TreeNodeFactory.create(root)
            if self.root is None :
                print("self.root est nul.")
            else :
                self.root.addChildrenRec(root)
```

```python
def addChildrenRec(self, data : dict) -> None :
    """
    Generates a node from the data contained in a dictionnary
    (extracted from a json).
    The node will then create its children, until the tree is entirely filled.
    author : Nathan
    """
    childrenList = data["children"]
    for nodeData in childrenList:
        node = TreeNodeFactory.create(nodeData)
        self.addChild(node)
        node.addChildrenRec(nodeData)
```

La fonction **_build** fait appel à la fonction **addChildrenRec**, qui ajoute tous les enfants du nœud courant dans l'arbre. La fonction s'arrête lorsqu'un nœud n'a pas d'enfant (feuille), sinon elle ajoute les enfants à leur parent avant d'importer les petits-enfants. Cette fonction est donc constituée d'une boucle, qui s'exécutera autant de fois que d'enfants dans l'arbre, soit n - 1 fois, avec n la taille de l'arbre.

La complexité est donc en O(n).