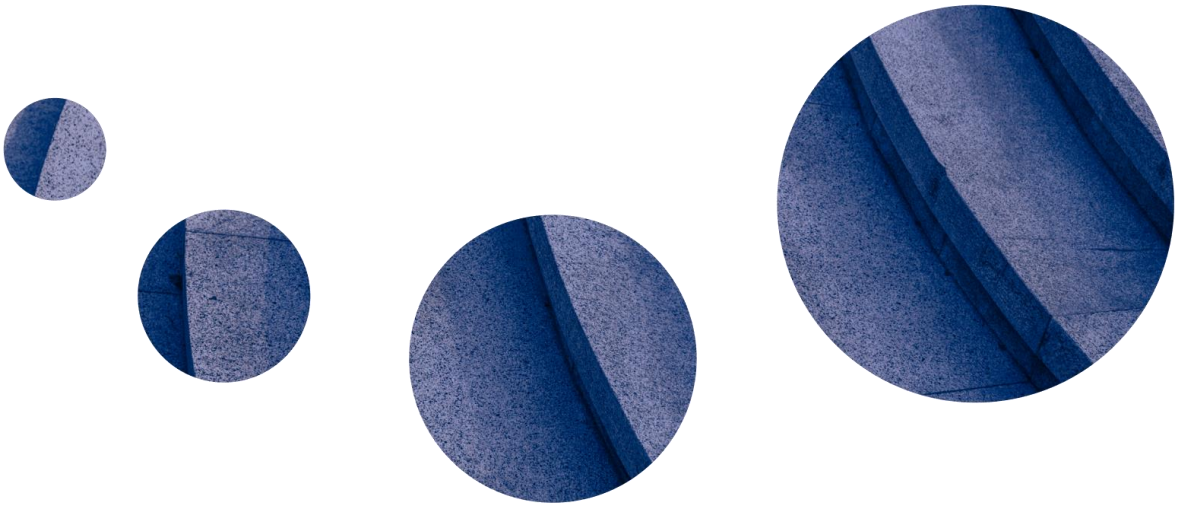


Compte Rendu SAE 2.02 : Exploitation algorithmique d'un problème

HODIN Dorian, ASTOLFI Vincent, JACQUELIN Bastien

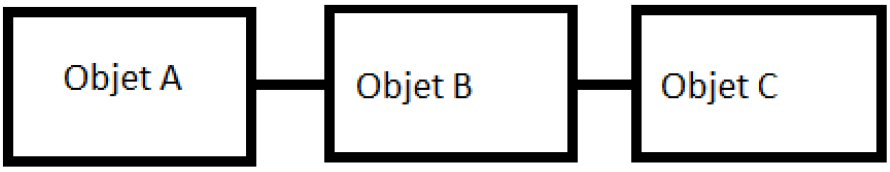
Partie 1 : Description des implémentations

Nous devions modéliser un réseau de transformations d'objets dans un jeu vidéo de différentes façons. Dans un premier temps nous ajoutons tous nos objets dans notre réseau à l'aide d'un vecteur d'objet. Nous avons choisi d'utiliser un vecteur car nous partons du principe qu'il ne sera pas possible de modifier les objets de notre réseau après que nous l'avoir créé. Ainsi, au début du programme il faut définir le nombre d'objet que l'on souhaite avoir dans le réseau puis on les définies en donnant leurs noms. Les transformations sont, quant à elles, définie dans l'un des deux réseaux soit en liste soit sous la forme d'une matrice. Les transformations possibles sont définies une fois le la modélisation souhaitée soit choisie, comme expliqué ultérieurement. Nous avons choisi de faire une application où l'utilisateur est libre. Il choisit donc les objets ainsi que leurs transformations puis test si un objet est transformable en un autre ou non.

L'application propose aussi un mode automatique où tout le réseau d'objet est créé automatiquement ainsi que les transformations possibles de chaque objet puis test toutes les transformations possibles pour ce réseau d'objet. Le réseau et les tests en question sont expliqué plus bas.

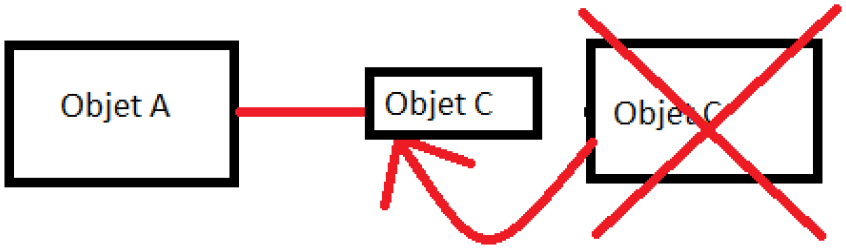
Pour cette SAE, nous avons décidé d'implémenter une liste d'objets où chaque transformation directe est représentée par un pointeur vers l'objet d'arrivée. Pour réaliser cela, nous avons créé une classe qui se nomme liste qui est constitué d'une liste de pointeurs vers l'objet maillon. L'objet maillon est un objet qui est constitué de son nom et d'une liste de pointeur vers les transformations directes. Nous avons donc la classe liste qui

représente tous les objets de notre univers et des objets Maillon qui stocke pour chaque objet toute les transformations directes possibles. Nous avons choisi d'utiliser des listes de pointeur sur Maillon pour la liste de transformations d'un objet car il est possible, dans notre algorithme, de supprimer et d'ajouter à sa guise des objets à cette liste et donc de modifier dès qu'on le souhaite les transformations possibles d'un objet. Ainsi, l'utilisation d'une liste plutôt qu'un vecteur va permettre de réduire la complexité de l'algorithme car, dans une liste, il est possible de supprimer directement un élément et de simplement refaire le lien entre les maillons suivant et précédant de celui que l'on vient de supprimer en redirigeant les objets pointés par la liste. Là où, avec un vecteur, il faudrait replacer tous les éléments suivants celui que l'on vient de retirer ce qui augmenterai fortement la complexité.



Dans l'exemple suivant, si l'on veut supprimer l'objet B avec une matrice on fait :

Cela signifie donc simplement qu'on enlève l'élément de la liste puis qu'on redéfinie le pointeur qui pointait de A vers B en un pointeur qui pointe de A vers C. Or dans un tableau il faudrait :



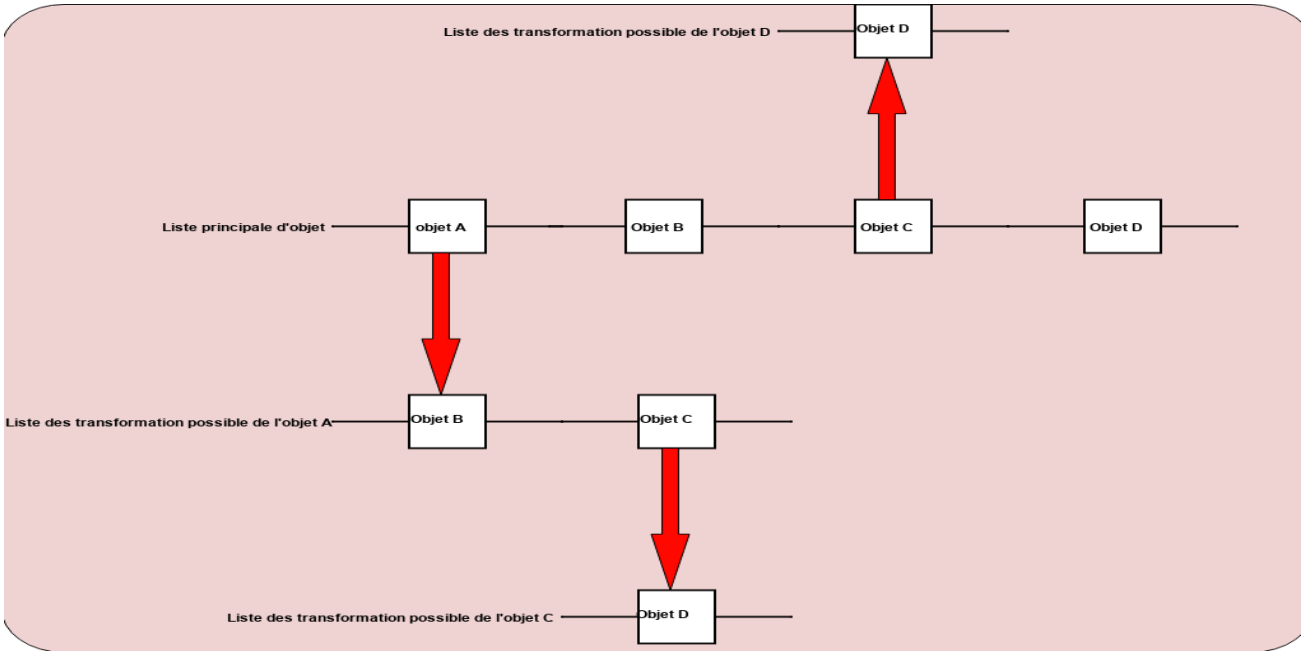
Il faudrait donc supprimer l'élément B puis replacer l'élément C. On remarque donc que juste pour un tableau de 4 éléments il faudra faire 3 opérations, et ce nombre ne fera qu'augmenter avec la taille du tableau là où il sera toujours à 2 opérations pour des listes.

En deuxième solution, nous avons choisis d'utiliser des matrices pour stocker les transformations directes. Nous avons donc créé une classe matrice qui contient un vecteur à 2 dimensions. Chaque élément du vecteur est composé de deux valeurs qui donnent l'emplacement de la case à regarder. Pour une case (i,j), nous avons le résultat de la comptabilité de l'objet n°i avec l'objet n°j. Si l'objet n°j est une transformation directe de

l'objet n°i, on écrit 1 dans la case. Si l'objet n°j n'est pas une transformation directe de l'objet n°i, on écrit 0 dans la case en enfin si l'objet n°i est égal à l'objet n°j, donc que j=i, on écrit -1 dans la case de coordonnées (i,j).

Partie 2 : Explication des algorithmes

Liste : Le principe du réseau par liste est que nous connaissons chaque nom de nos objets via un tableau les contenants. Ces objets sont particuliers. En effet, ils comportent chacun deux informations. Dans un premier temps leurs nom, qui nous permets de les reconnaitre et enfin d'un autre tableau comprenant toutes les transformations possibles de cet objet. Par exemple, des planches peuvent servir à fabriquer des tables ou des épées en bois. Ainsi, dans le tableau que nous pouvons retrouver lié à l'objet nommé planche se trouve les deux objets suivants : table et épée. Chacun de ses deux objets comprenant lui-même un tableau avec toute ses transformations possibles. Ainsi, on peut parcourir les transformations de chacun de nos objets pour transformer un objet en un autre. Par exemple du bois peut servir à faire des planches qui peuvent-elles même servir à faire des épées. Donc on peut transformer du bois en table en transformant le bois en planche puis en épée.

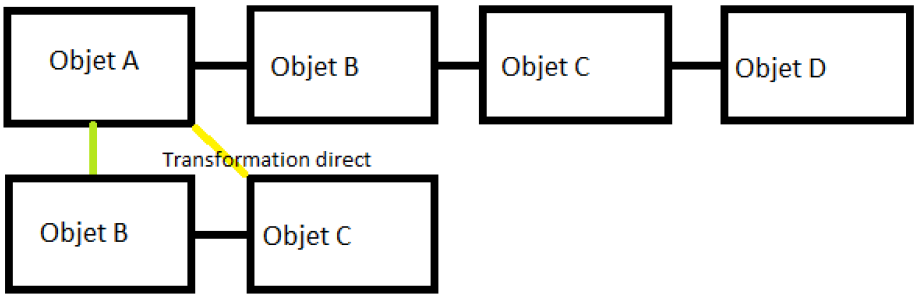


Dans cet exemple schématisé on possède 4 objets nommée respectivement objet A, B, C, D. L'objet A possède deux transformations, il peut se transformer en l'objet B et en l'objet C. L'objet C, quant à lui peu se transformer en l'objet D. Ainsi, avec ce schéma on peut constater qu'on peut transformer un objet A en un objet D par l'intermédiaire d'un objet C. Ce principe fonctionne quel que soit le nombre d'objet de base.

La fonction qui permet de vérifier qu'une transformation possible fonctionne réellement de la fonction suivante. Lorsqu'on l'appelle on lui donne le nom de l'objet que l'on cherche ainsi qu'un tableau vide qui nous servira à l'affichage plus tard.

Puis, dans un premier temps, l'algorithme vérifie que l'objet que l'on souhaite transformer n'est pas un objet qui est transformable en aucun autre. Auquel cas l'algorithme nous indiques une erreur et nous invite à sélectionner un autre objet.

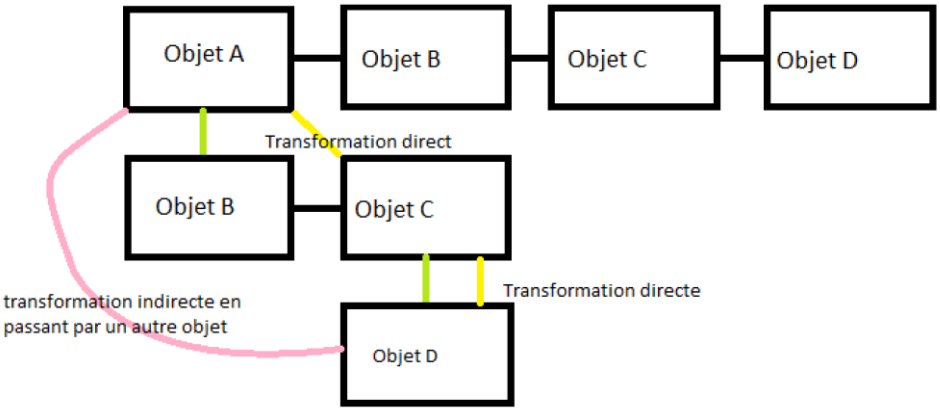
Après cela, l'algorithme parcours les transformations directes possible de notre objet de base et, s'il trouve directement l'objet que nous cherchions, alors il écrit qu'il a bien trouver l'objet en question et donc que la transformation et possible. Pour reprendre l'exemple ci-dessus, l'objet A est transformable directement en l'objet B et en l'objet C. Ainsi si l'on cherche l'un de ces deux objets on le trouvera en un seul tour de boucle.



Dans l'exemple ci-dessus on voit qu'il y a une transformation directe entre l'objet A et l'objet C.

S'il ne trouve pas directement dans le premier tableau de transformations alors il va chercher dans ceux de ses propres transformations. Afin d'être plus clair reprenant l'exemple du réseau simple au-dessus.

Admettons que l'on souhaite transformer l'objet en A en objet D. L'algorithme va commencer par parcourir les transformations possibles de l'objet A, donc l'objet B et l'objet C. Ainsi, il ne trouve pas directement l'objet D. Il parcourt donc les transformations possibles de l'objet B mais ce rend compte qu'il n'en possède pas. L'algorithme va alors essayer les transformations possibles de l'objet C et va donc trouver l'objet D en question. Ainsi, l'objet A est transformable en l'objet C, et l'objet C est transformable en l'objet D, donc A est transformable en D.



On voit donc qu'il y a une transformation directe entre l'objet A et l'objet B ainsi qu'une transformation directe entre l'objet C et l'objet D. Il y a donc une transformation indirecte mais possible entre A et D.

Si une transformation est impossible, comme par exemple transformer C en A, l'algorithme va essayer toutes les possibilités puis sortir du programme car la transformation est introuvable.

L'affichage de cette recherche se fait donc à l'aide d'un tableau nommée "tmp". Il est vidé à chaque nouvel appel de la fonction et va nous servir à retenir le nom des objets par lesquels nous sommes passé pour arriver à finaliser notre transformation. Par exemple dans le cas où l'on souhaite transformer A en D notre tableau contiendra les informations suivantes.

Nom de l'objet	A	C	D

Ainsi, lors de l'affichage, on affichera la chose suivante :

"A est transformable en C
C est transformable en D,
donc A est transformable en D"

On remarquera que si la transformation est directe on n'affiche pas la ligne "Donc A est transformable en D"

Matrice :

Une fois tous les items rentrés, ils se trouvent dans un tableau nommé Items.

Place	0	1	2	3	4
Nom	Obj1	Obj2	Obj3	Obj4	Obj5

Lorsque la demande d'une recherche est effectuée, la première chose à faire est de vérifier si les items renseignés sont bien présents dans le tableau. En plus de vérifier, on renvoie à quelle place l'item de départ et l'item recherché sont situés.

Ici, on recherche un objet ObjY comme objet de départ et un objet ObjX comme item recherché.

Leur place dans le tableau est respectivement 1 et 3 et leur valeur : Obj2 et Obj4.

Une fois toutes les transformations, on se retrouve avec une matrice comme celle-ci :

Transformation de l'Objet de départ / Transformation vers l'Objet d'arrivée

	Obj 1	Obj 2	Obj 3	Obj 4	Obj 5

Obj1	-1	0	0	0	1
Obj2	1	-1	0	0	0
Obj3	0	0	-1	0	0
Obj4	1	0	0	-1	0
Obj5	0	1	1	1	-1

On lit horizontalement les transformations d'un objet, (flèche bleue)

Un 1 représente une transformation, tandis qu'un 0 signifie que l'objet en question ne peut se transformer en l'autre objet. Le -1 est présent lorsque l'on demande à un objet de se transformer en lui-même.

On va donc commencer par parcourir les transformations directes de Obj2, Il faut savoir que le parcours se fait de la gauche vers la droite

On voit que Obj2 ne peut se transformer directement en Obj4 : 0

On cherche en quoi Obj2 pourrait se transformer :

1 - 1 0 0 0

Obj2 se transforme directement en obj1, faisons de même pour voir si ce dernier propose des transformations :

-1 0 0 0 1

Il admet une seule transformation en Obj5, on réitère le même processus :

0 1 1 1 -1

La première transformation est en Obj2. Or comme nous venons de le voir, nous avons déjà testé Obj2. Une boucle se créerait donc. Pour parer cela, nous avons mis en place un nouveau tableau « Tested » qui stock chaque Obj déjà testé. Avant de passer à l'objet suivant, on ajoute l'objet dans le tableau tested. Une fois que l'on trouve une nouvelle relation, on vérifie si l'item est présent dans ce tableau, s'il l'est, on parcourt les transformations suivantes de l'objet précédent :

0 ~~1~~ 1 1 -1

Il se trouve que Obj5 se transforme également Obj3 :

0 0 -1 0 0

Or cet item ne possède aucune transformation. On rembobine à Obj5 et on parcourt de nouveau les transformations suivantes :

0 ~~1~~ ~~1~~ 1 -1

Il se trouve que Obj5 possède une autre transformation, cette fois si en Obj4

Une fois l'objet d'arrivée trouvé, on ajoute dans un nouveau tableau tous les items par lesquels on est passé, en commençant par l'objet le plus proche de l'objet recherché ici Obj4. On remonte et on ajoute Obj1, et pour finir Obj2.

Récap des transformations et du remplissage des tableaux :

Transformation de Obj2 en Obj1 : Tested : ø ;Required : ø

Transformation de Obj1 en Obj5 : Tested : Obj2 ; Required : ø

~~Transformation de Obj5 en Obj2 :~~ -Tested : Obj2 Obj1 ; Required : ø

~~Renvoie faux : pas de transformation~~

~~Transformation de Obj5 en Obj3 :~~ Tested : Obj2 Obj1 Obj3 ~~;~~Required : ø

~~Renvoie faux : pas de transformation~~

Transformation de Obj5 en Obj4 ; Tested : Obj2 Obj1 Obj3 ~~;~~Required : ø

Renvoie Vrai, Obj5 se transforme bien en Obj4 ; Tested : Obj2 Obj1 Obj3- Required : ø

Renvoie Vrai, se transforme bien en Obj4 ; Tested : Obj2 Obj1 Obj3–Required : Obj5

Renvoie Vrai, se transforme bien en Obj4 ; Tested : Obj2 Obj1 Obj3–Required : Obj5, Obj1

Renvoie Vrai, se transforme bien en Obj4 ; Tested : Obj2 Obj1 Obj3-Required : Obj5, Obj1, Obj2

Les Objets à parcourir sont dans l'ordre inverse de celui du tableau, c'est-à-dire qu'on commence avec Obj2 qui se transforme en Obj1, lui-même se transforme en Obj5, qui finit par se transformer en Obj4, l'objet recherché.

Partie 3 : Comparaison de la complexité des 2 méthodes (au pire des cas) laquelle est la plus pertinente ?

Liste : Pour le constructeur de l'objet Liste, la complexité est constante, donc de O(1). Pour la fonction getNbMaillon, la complexité est aussi constante (O(1)), on return juste une valeur. Cependant pour la fonction creerList, la complexité est de O(n) car on a une boucle

for qui parcourt toute la liste d'objet. C'est aussi le cas de la fonction affichListe, sa complexité est linéaire. Le fonction creerDependance, malgré sa longueur, n'utilise qu'une seule boucle for. Mais dans cette dernière on appelle une autre fonction qui a une complexité linéaire. Donc cela revient à une boucle for dans un for. Cependant, on appelle aussi une dernière fonction transformer qui est déjà composé d'une boucle for dans une autre boucle for. Donc au final nous avons une boucle for qui lance une boucle for dans une autre boucle fort, la complexité de cette fonction est donc cubique, d'ordre O(n**3).

Maillon : Pour le constructeur de l'objet Maillon, la complexité est constante comme pour la fonction getDesignation. La fonction creerListeTransfo est composé d'une boucle for dans une boucle while, la complexité est donc quadratique, d'ordre O(n**2). La fonction affichTransfoListe n'as qu'une seule boucle for, la complexité est donc de O(n). La fonction transformer est composé d'une boucle for dans une autre, la complexité est par conséquent quadratique, O(n**2). Les surcharges des opérateurs !=, == et << sont toutes de complexité d'ordre O(1), donc de complexité constante. La fonction viderLesListes qui sert à liberer les listes de transformation des objets est de complexité O(1), car il y aucune boucle for. Et pour finir la fonction supprimerListeTransfo est de complexité linéaire, une seule utilisation d'une boucle for.

Matrice : Les constructeurs de la classe Matrice est comme tous les autres constructeurs de complexité constante, donc O(1). La fonction remplirMatriceAuto est aussi de complexité O(1). Ensuite, nous avons la fonction remplirMatrice qui est composé d'une boucle while dans une boucle for dans une boucle for, ce qui augmente grandement la complexité de cette fonction. On en déduit donc, vu que l'on imbrique 3 boucles entre elle, que la complexité est cubique, donc d'ordre O(n**3). La fonction afficherMatrice a une complexité de O(n**2) car il y a une boucle dans une boucle (deux boucles for). Nous avons également les fonctions AfficherTab, research et present ont des complexités linéaires, donc d'ordre O(n). La fonction researchTransfo est une fonction récursive, donc la complexité est forcément de O(n**2). Par conséquent la fonction transfoDirect qui appelle researchTransfo en dehors d'une boucle for, a aussi une complexité quadratique. Et finalement nous avons la fonction menu, qui appelle dans une boucle while les fonctions research, transfoDirect, AfficherTab et afficherMatrice. Cependant aucune des fonctions appelées est imbriquée dans une autre, donc la plus grosse est celle de la fonction afficherMatrice dans la boucle while, donc qui a une complexité de O(n**), complexité cubique.

Conclusion : Pour conclure, nous voyons qu'utiliser le réseau en matrice appelle beaucoup plus de fonction à la complexité assez élevé comparé au réseau en liste. Par conséquent la solution avec le réseau en liste sera la plus pertinente et la plus rapide pour répondre à ce problème.

Partie 4 : Nos programme peuvent résoudre d'autres problèmes similaires ?

Nos programmes peuvent servir à résoudre tout type de problème qui relève de la même base. C'est à dire tout problème qui demande de partir d'un point A pour arriver à un point

B avec des éléments qui s'appelle les uns les autres. Cela peut correspondre à un voyage avec des correspondance par exemple. Si un bus partant de Clermont-Ferrand veut aller à Paris mais doit pour ça passer par une correspondance par Vichy alors notre algorithme pourra nous dire qu'il est possible d'aller de Clermont-Ferrand à Paris en passant par Vichy. Le nombre d'exemple peut être très élevés mais chacun d'eux ne nécessitera presque aucune modification de l'algorithme de base si ce n'est les éléments d'affichage. Les fonctions de recherches de chemin elle ne nécessiteront aucun changement pour pouvoir fonctionner correctement quel que soit le type de réseau que l'on décide d'utiliser.

Par exemple, le réseau aérien pourrait fonctionner avec notre application pour expliquer quel chemin il faudrait utiliser pour aller d'un point A à un point B. Par exemple, on peut aller de l'aéroport de Paris-Orly à celui de Bastia en faisant une correspondance à celui de Montpellier.

Explication du fonctionnement de la partie automatique.

Comme expliqué dans la partie 1, notre programme propose un réseau déjà initialisé qui fonctionne automatiquement afin de tester les différentes fonctionnalités sans avoir à créer tous les objets d'un réseau et sans avoir à rentrer leurs transformations possibles. La liste suivante donne toutes les valeurs que nous avons créées pour cette fonction de test automatique ainsi que leurs transformations possibles.

Nom des objets --> noms de ses transformations possibles

Arbre --> Planche

Planche --> Bâton, Table, Bouclier

Bâton --> Echelle, Canne

Fer --> Pioche, Epée, Boussole, Armure

Pioche --> Aucune transformation

Epée --> Aucune transformation

Table --> Bouclier, Planche

Bouclier --> Planche, Table

Echelle --> Bâton

Or --> Lingots d'or, Couronne

Lingots d'or -- > Or, Couronne

Boussole --> Fer

Couronne --> Or, Lingots d'or

Armure --> Fer

Canne à pêche --> Bâton

Puis on affiche toute la liste de transformation de chacun de ses objets pour voir si les transformations se sont faites pour les deux types d'algorithmes (en liste ou en matrice).

Suite à cela, la fonction automatique va nous proposer si l'on souhaite tester des transformations ou non. Si l'on répond oui alors nous allons être redirigé vers un menu annexe où on pourra tester nos transformations sans pouvoir les modifier pour autant car elles ont déjà été implémentées automatiquement.