1 Shuffle de árboles

En esta seción describiremos el producto tensorial $\Omega[S] \otimes \Omega[T]$ para todo par de árboles S y T de Ω. Para ello deberemos introducir la noción del conjunto de shuffles de S y T, que será una gran ayuda para encontrar el producto tensorial. Así podremos entender que es el producto tensorial de conjuntos dendroidales.

1.1 Producto tensorial de árboles lineales

Antes de ver el producto tensorial para árboles en general, estudiaremos el caso de árboles lineales ya que resulta una tarea más sencilla y la podemos relacionar con el producto cartesiano de conjuntos simpliciales.

Sean $S = L_n$ y $T = L_m$ dos árboles lineales, entonces por la proposición 3.30(i),

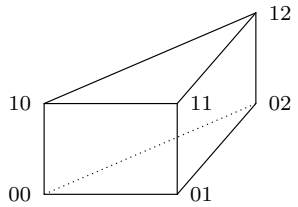
$$\Omega[L_n] \otimes \Omega[L_m] = i_!(\Delta[n]) \otimes i_!(\Delta[m]) \cong i_!(\Delta[n] \times \Delta[m])$$

Los simplices no degenerados del producto de dos representables en conjuntos simpliciales son computados mediante un *shuffle*. Un *(n,m)-shuffle* es un camino de longitud máxima en el conjunto de orden parcial $[n] \times [m]$. Los $(n + m)$-simplices no degenerados de $\Delta[n] \times \Delta[m]$ corresponde a los $(n,\ m)$-shuffles. De hecho,

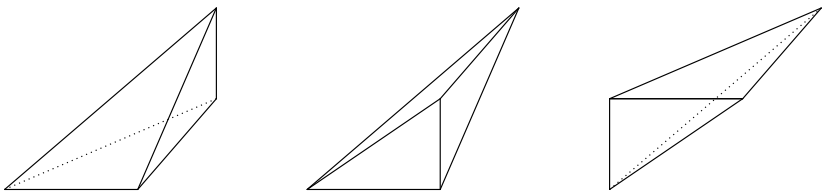
$$\Delta[n] \times \Delta[m] = \bigcup_{(n,m)} \Delta[n + m]$$

Donde la unión recorre todos los posibles (n, m)-shuffles.

Ejemplo 1.1. Sean $n = 2$ y $m = 1$. Existen tres (2, 1)-shuffles en $[2] \times [1]$, (00,01,02,12), (00,01,11,12) y (00,10,11,12). Tenemos la siguiente figura que representa $\Delta[2] \times \Delta[1]$



Podemos ver que cada (2, 1)-shuffle corresponde a un tetraedro, y hacen una descomposición de $\Delta[2] \times \Delta[1]$ como la unión de tres copias de $\Delta[3]$:



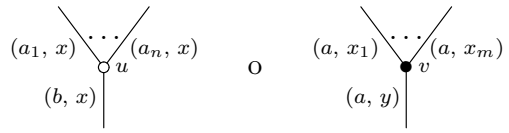
(00, 01, 02, 12) (00, 01, 11, 12) (00, 10, 11, 12)

1.2 Producto tensorial de árboles

1.2.1 Shuffles y conjuntos de shuffles

En este apartado vamos a introducir la noción de un shuffle entre dos árboles y la coleción de todos ellos. También daremos ejemplos extensos de como calcular dichos shuffles.

Definición 1.2. Sea S y T dos objetos de Ω. Un *shuffle* de S y T es un árbol R cuyo conjunto de aristas es un subconjunto de $E(S) \times E(T)$. La raíz de R es $(a,\ x)$, donde a es la raíz de S y x es la raíz de T, y sus hojas son todos los pares $(l_S,\ l_T)$, donde l_S es una hoja de S y l_T es una hoja de T. Los vértices son de la forma



Donde u es un vértice de S con entradas $a_1, \ldots, a_n$ y salida b, y v es un vértice de T con entradas $x_1, \ldots, x_m$ y salida y. Nos referiremos a los dos tipos de vértices como *vértices blancos* y *vértices negres*, respectivamente. Para diferenciarlos visualmente los pintaremos con $\circ$ y $\bullet$, respectivamente.

Observamos que existe una biyección entre los shuffles de dos árboles lineales L_n y L_m con los $(n,\ m)$-shuffles de $[n] \times [m]$.

Definición 1.3. Sean S y T dos árboles. El *conjunto de shuffles de S y T* es la colección de todos los shuffles posibles entre S y T. La cardinalidad de este conjunto la denotaremos por $sh(S,\ T)$.

Proposición 1.4. *El número de shuffles $sh(S,\ T)$ de dos árboles S y T satisface tres propiedades:*

(i) $sh(S,\ T) = sh(T,\ S)$

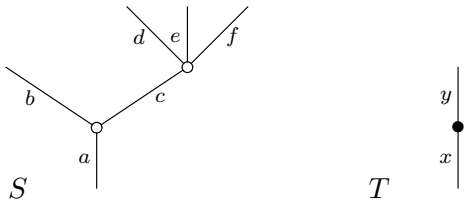
(ii) *Si T es un árbol unitario η, entonces* $sh(S,\ \eta) = 1$

(iii) *Si* $S = C_n[S_1, \ldots, S_n]$ *y* $T = C_m[T_1, \ldots, T_m]$, *entonces*

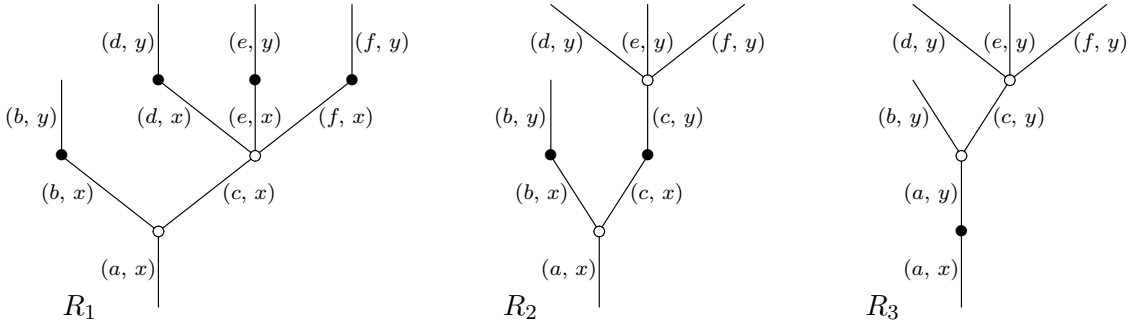
$$sh(S,\ T) = \prod_{i=1}^{n} sh(S_i,\ T) + \prod_{j=1}^{m} sh(S,\ T_j)$$

Donde C_n y C_m son n y m-corolas, respectivamente; y $C_n[S_1, \ldots, S_n]$ es una n-corola que cada hoja i-esima la conectamos con la raíz del árbol S_i.

Ejemplo 1.5. Sean S y T los árboles

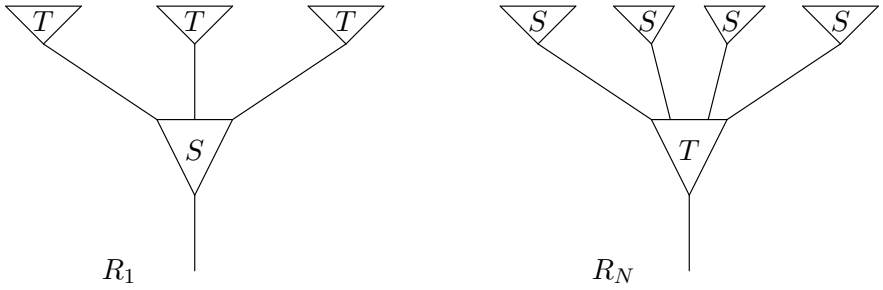


El conjunto de shuffles de S y T consiste de los siguientes tres árboles:

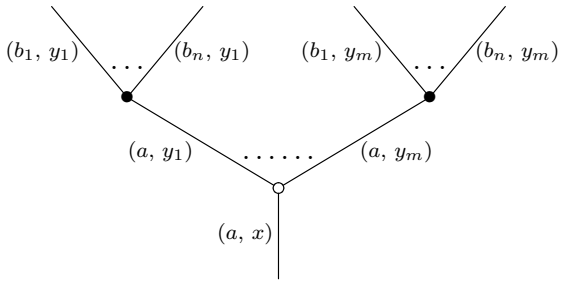


(1.1)

El conjunto de shuffles de S y T es *ordenado parcialmente*. El árbol minimal R_1 en el conjunto ordenado parcialmente se obtiene mediante la inserción de una copia del árbol negro T en cada entrada del árbol blanco S. Es decir, primero hacemos una copia del árbol S de la forma $S \otimes r_T$, donde todas sus aristas han sido renombradas como $(\_,\ r_T)$, siendo r_T la raíz del árbol T. Luego hacemos una copia del árbol T de la forma $l \otimes T$, para toda hoja l de S; donde todas sus aristas han sido renombradas como $(l,\ \_)$. Finalmente, obtenemos el árbol R_1 encajando las últimas copias encima de las hojas de la forma $(l,\ r_T)$ de la primera copia. El árbol maximal R_N en el conjunto ordenado parcialmente se obtiene mediante la inserción de una copia del árbol blanco S en cada entrada del árbol negro T. Los árboles R_1 y R_n deberían lucir de la siguiente manera

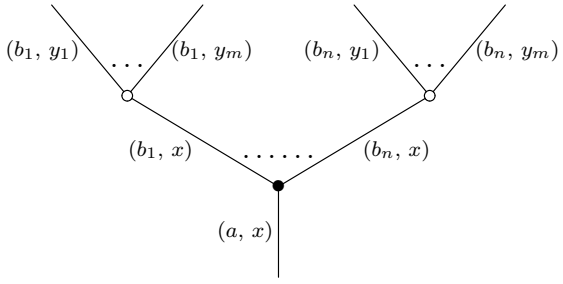


Existen los *shuffles intermediarios* R_k $(1 < k < N)$ entre R_1 y R_N obtenidos filtrando los vértices negros en R_1 hacia la raíz del árbol mediante intercambios con los vértices blancos. Todo R_k se obtiene desde un R_l anterior. Es decir, cada intercambio se basa en transformar una configuración de R_l



(1.2)

A una configuración de R_k

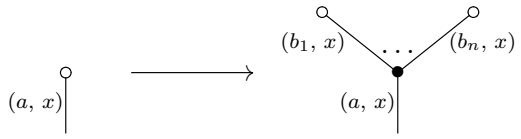


(1.3)

Si un shuffle R_k se obtiene the otro shuffle R_l mediante la norma de arriba, entonces decimos que R_k se obtiene mediante *un solo intercambio* y lo denotaremos por $R_l \leq R_k$. Así, obtenemos un orden parcial en el conjunto de todos los shuffles.

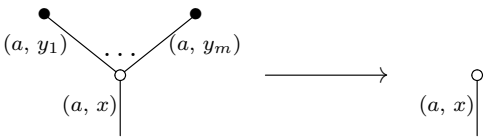
Tenemos que espicificar el caso de un intercambio con un árboles sin entradas, es decir,

$n = 0$ o $m = 0$. Si $m = 0$ y $n \neq 0$, entonces tenemos el intercambio



(1.4)

Si $n = 0$ y $m \neq 0$, entonces tenemos el intercambio



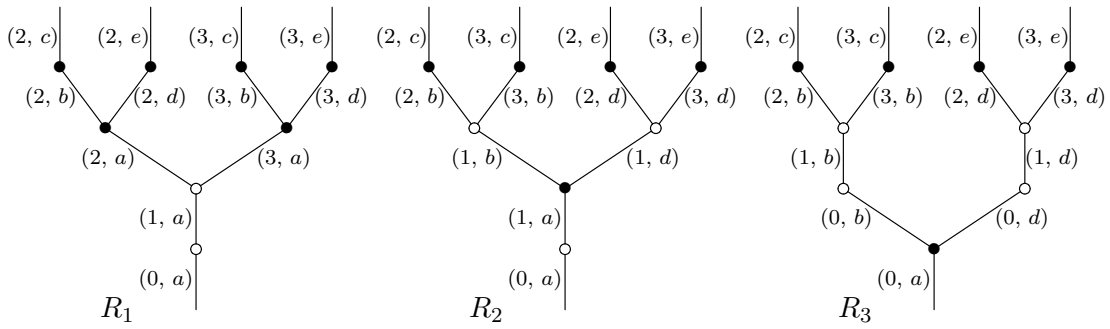
(1.5)

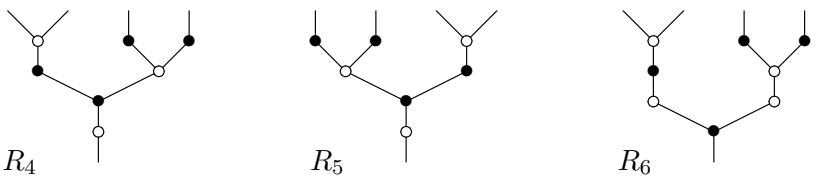
Finalmente, si $n = m = 0$, entonces tenemos el intercambio

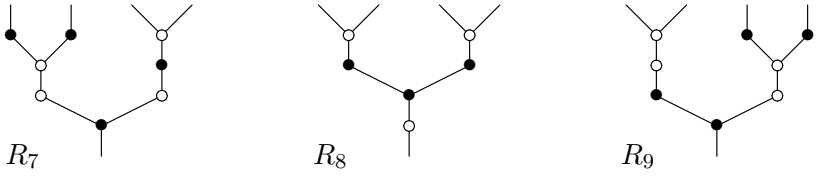
(1.6)

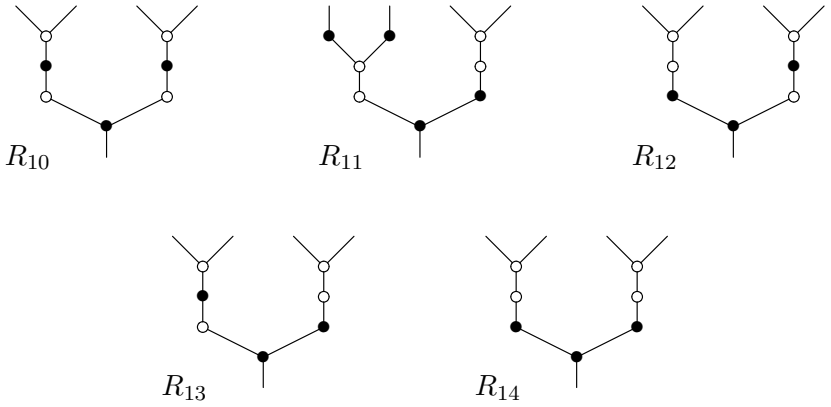
Ejemplo 1.6. Sean S y T los árboles

Existen catorce shuffles $R_1, \ldots, R_{14}$ de S y T. Mostramos una lista completa de ellos. Marcaremos los nombres de las aristas en los tres primeros shuffles.

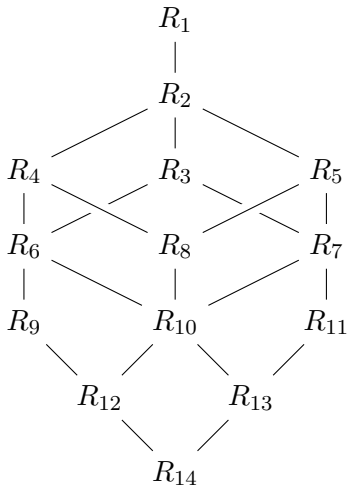








Tenemos la siguiente estructura dentro del conjunto ordenado parcialmente.



Producto tensorial de árboles

Ahora podemos dar una descripción completa del producto tensorial entre dos objetos representables en Ω mediante el cómputo de su conjunto de shuffles.

Lema 1.7. *Para todo shuffle R_i de S y T tenemos un monomorfismo*

$$m\colon \Omega[R_i] \rightarrowtail \Omega[S] \otimes \Omega[T]$$

El subconjunto dendroidal, que viene dado por la imágen de este monomorfismo, lo denotaremos $m(R_i)$.

Proof. Los vérties del conjunto dendroidal $\Omega[R_i]$ son las aristas del árbol R_i. La función m envía aristas nombradas como $(a,\ x)$ en R_i a la arista con el mismo nombre en $\Omega[S]\otimes\Omega[T]$. Vemos que es un monomorfismo. No entiendo. □

Corolario 1.8. *Para todo objeto T y S en Ω, tenemos que*

$$\Omega[S] \otimes \Omega[T] = \bigcup_{i=1}^{N} m(R_i)$$

donde la unión recorre todos los posibles shuffles de S y T.

1.3 Shuffle de árboles en Python

No es complicado ver que tanto encontrar el producto tensorial de conjuntos dendroidales o, equivalentemente, calcular el conjunto de shuffles para dos árboles cualesquiera, resulta una tarea tediosa si los árboles son grandes. Para tal problema, el uso de un programa informático, capaz de almanezar grandes cantidades de información al momento de ejecución, nos resulta cómodo, fácil y rápido.

En este apartado describiré de manera breve el código que he escrito para poder tratar con opéradas, árboles, shuffles y finalmente con el conjunto de shuffles. También, el código incluye una función para formar figuras con el paquete *xy* de *Latex* de un árbol mediante una descripción básica.

Finalmente, dicho código lo podréis encontrar tanto en el Anexo 1 como en el repositorio público de código de Github: Trees Shuffling. Hace falta comentar que habrán diferencias entre el código completo que podréis encontrar en el anexo y los pseudocódigos usados a continuación, ya que nos quedamos con la estructura fundamental del algoritmo para una facilitar la lectura.

Clases del paquete

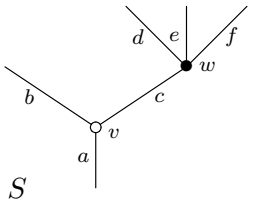
Definición 1.9. Una *clase* es una abastración de propiedades y funciones de un objeto en concreto. Siguiendo tal definición, tenemos las siguientes clases en nuestro paquete:

- *Operad*: Espacio que guarda los colores y las operaciones que forman un árbol.
- *Tree*: Clase abstracta que define un árbol con propiedades tipo: raíz, hojas...
- *TreeMerger*: Clase para juntar dos árboles S y T, uno encima del otro.
- *TreeManipulator*: Clase para buscar y hacer intercambios.
- *ShuffleLattice*: Clase para generar todos los shuffles entre dos árboles S y T.

Utilidades

Antes de describir las clases que puedes encontrar en el paquete, debemos comentar que existe un fichero llamado *utils.py* donde hay una colección de funciones y algoritmos *utiles* que usaremos a lo largo del código.

Vale la pena mencionar la función que servirá como la puerta de entrarda a los árboles, nombrada: *string_to_tree_space*. Tiene como entrada una descripción completa de un árbol y devuelve una instáncia de la clase *Tree*. Por ejemplo, la cadena de cáracteres "$vW(b,c;a)|wB(d,e,f;c)$" describe el árbol S



Donde el vértice v de color blanco (W) tiene como entradas b y c, y salida a; y el vértice w de color negro (B) tiene como entradas d, e y f, y salida c. Durante la explicación vamos llamar *operaciones* a "$vW(b, c; a)$" o "$wB(d, e, f; c)$".

En esta coleccíon se encuentra el algoritmo *sorted* que nos va a ordenar las operaciones que generan un árbol. El orden que vamos a describir nos asegura que todo color de salida en cada operación, aparezca como entrada de otra operación en la izquierda, salvo el color correspondiente a la raíz ya que esa operación en concreto será la primera. Hemos realizado un pseudocódigo para que se entienda mejor el algoritmo *sorted*.

Algorithm 1 Pseudocódigo del algoritmo para ordenar las operaciones de un árbol T. Podéis encontrar el código completo en el anexo.

Input: Requires an array of operations of a tree

```
1:  function SORTED(operations)
2:      n ← Length of operations
3:      i ← 0                                                    ▷ Starting index
4:      while i < n do
5:          for j in [i + 1, ..., n) do
6:              if operations[i].trunk in operations[j].branches then
7:                  operation ← Pop operation on index i from operations
8:                  opertations.insert(operation, j)        ▷ Insert operation on new index
9:                  break
10:         else        ▷ Note that this else only occurs when the for hasn't been broken
11:             i ← i + 1
12:     return operations
```

Output: Returns the list of operations sorted

Por ejemplo, todas las siguientes descripciones hablan de un mismo árbol S:

1. "$2W(3;2)|1W(2,4,6;1)|3W(5;4)|4W(7;6)|0W(1;0)$"
2. "$1W(2,4,6;1)|2W(3;2)|3W(5;4)|0W(1;0)|4W(7;6)$"
3. "$3W(5;4)|2W(3;2)|4W(7;6)|0W(1;0)|1W(2,4,6;1)$"

Si aplicamos el algoritmo *sorted* a estas entradas obtendremos la siguiente cadena de carácteres "$0W(1;0)|1W(2,4,6;1)|4W(7;6)|3W(5;4)|2W(3;2)$" que describe el árbol S de manera estándar.

Clase *Tree*

La clase *Tree* tiene las propiedades que define un árbol como una opérada coloreada.

- *trunk*: Color de salida.
- *branches*: Lista de colores de entrada.
- *node*: Nombre de la operación.

Sea S un árbol con más de una operación, cada operación será una instancia de *Tree* y estarán relacionadas entre si mediante las ramas (*branches*) y los troncos (*trunk*). Es

decir, sea $S =$ "$vW(b, c; a)|wB(d, e, f; c)$", la rama c de la operación "$vW(b, c; a)$" esta relacionada con el tronco c de la operación "$wB(d, e, f; c)$".

De esta manera tenemos una estructura de árbol en forma de grafo, que podremos explotar con los algoritmos que vienen a continuación. Y además, tendremos todo el árbol a mano mediante la operación que contienen la raíz. Falta mencionar que usaremos la clase *uTree* que describe un árbol sin *node* ni *branches*.

Clase *TreeMerger*

Esta clase tiene unos algoritmos, que no describiremos, para poder unir dos árboles de la misma manera descrita en el apartado 4.2.1. Es decir, la clase genera el shuffle R_1 si usamos los dos árboles S y T de entrada; y a la inversa, la clase genera el shuffle R_N si usamos los árboles T y S de entrada.

Clase *TreeManipulator*

Esta clase es importante para la clase que explicaremos a contiunación *ShuffleLattice*, ya que será su alimento. Es decir, esta clase nos busca en un shuffle R los vértices disponibles para realizar un intercambio, explicado en el apartado 4.2.1; y también la función necesaria para realizar tal intercambio.

El algoritmo de búsqueda se llama *find_percolations*. Es el encargado de buscar los vértices disponibles para realizar un intercambio en un shuffle R cualquiera. Es un algoritmo recursivo donde va acumulando las operaciones donde se encuentran los intercambios posibles. Seguidamente podemos ver el pseudocódigo.

Algorithm 2 Pseudocódigo del algoritmo para encontrar los vértices para hacer un intercambio de un shuffle R. Podéis encontrar el código completo en el anexo.

Input: Requires a *Tree* R and an optional array $found$

```
function FIND_PERCOLATIONS(R, found)        ▷ Note that R is the rooted operation
    if found is None then
        found ← Initialize empty array
    if R.node is from S.operations then
        for branch in R.branches do                    ▷ Note that branch is a Tree
            if branch.node not in T.operations then
                break
        else        ▷ Note that this else only occurs when the for hasn't been broken
            found ← Append R
    for R' in R.branches do
        find_percolations(R', found)                   ▷ Note that R' is a Tree
    return found
```

Output: Returns the list $found$ of locations

La función de acción se llama *make_percolation*. Es la encargada de realizar el intercambio, préviamiente encontrado por el algoritmo *find_percolations*. Es decir, realiza un cambio de operaciones y un cambio de los nombres de las aristas afectadas siguiendo la norma de intercambios.

Clase *ShuffleLattice*

Finalmente, la clase más importante del paquete que usa diractamente o indirecatamente todas las clases que hemos comentado anteriormente. Esta clase va a generar todos los shuffles R_i entre dos árboles cualesquiera S y T.

La clase tiene una propiedad llamada *dictionary* donde vamos acumulando todos los shuffles encontrados. También tiene una propiedad llamada *skeleton* donde guardamos las relaciones entre shuffles $R_l < R_k$ y así poder tener la estructura del conjunto de shuffles ordenado parcialmente.

La clase se basa en la ejecución del algoritmo *generate_shuffle*, que básicamente es el algoritmo clásico de búsqueda en anchura, más conocido como *BFS* en inglés. Es decir, es un algoritmo de búsqueda no informada que se alimenta con nuestro algoritmo *find_percolations* y actúa con la función *make_percolation*. Seguidamente podemos ver el pseudocódigo.

Algorithm 3 Pseudocódigo del algoritmo para generar todos los shuffles entre S y T. Podéis encontrar el código completo en el anexo.

Input: Requires the class instance of *ShuffleLattice*

```
1:  function GENERATE_SHUFFLES(self)
2:      queue ← Append R_1
3:      self.dictionary ← Save initial tree
4:      while queue not empty do
5:          tree ← Pop the first tree in queue
6:          for location in tree.find_percolations() do
7:              new_tree ← Apply make_percolation on location
8:              if new_tree not in self.dictionary then
9:                  queue ← new_tree Append new tree
10:                 self.dictionary ← Save new tree
```

Output: The algorithm does not return anything because the shuffles have been stored on the *dictionary*

Finalmente, comentaremos el algoritmo *sh* que usa la proposición 4.4. Este algoritmo devuelve el número de shuffles entre dos árboles S y T. Seguidamente podemos ver el pseudocódigo.

Algorithm 4 Pseudocódigo del algoritmo para computar el número de shuffles entre S y T. Podéis encontrar el código completo en el anexo.

Input: Requires two trees S and T

```
1: function SH(S, T)
2:     if S is a unitary Tree then
3:         return 1
4:     if T is a unitary Tree then
5:         return 1
6:     prod_S ← 1
7:     for branch in S.get_branches() do
8:         prod_S ← prod_S * sh(branch, T)        ▷ Note that every branch is a Tree
9:     prod_T ← 1
10:    for branch in T.get_branches() do
11:        prod_T ← prod_T * sh(S, branch)        ▷ Note that every branch is a Tree
12:    return prod_S + prod_T
```

Output: Returns the sum of the products