Práctica Final: Letras

Gustavo Rivas Gervilla

UNIVERSIDAD
DE GRANADA

DECSAI

Contenido

La práctica consiste en implementar un juego similar al Scrabble o al juego Cifras y Letras. Con esta práctica:

- practicaremos la abstracción.
- podremos hacer uso de la `STL`.
- seguiremos documentando el código que desarrollemos con la herramienta `Doxygen`.
- pondremos en práctica una de las habilidades más importantes para un ingeniero: el diseño de la solución más adecuada a un problema.
  - Esto pasa por el diseño y uso de una **estructura de datos adecuada**.

E E T S M R E D T E I C O N A S I O L O O D

E S T E R N O C L E I D O M A S T O I D E O

Esta práctica la podríamos dividir en 3 partes distintas (**que no módulos**, habrá que desarrollar tantos módulos como se consideren necesarios):

1. Almacenamiento de la información.
2. Generador de letras aleatorias.
3. Inteligencia Artificial.

Contenido

Se ha de desarrollar un `T.D.A` Diccionario y se tendrá que implementar un test que pruebe el correcto funcinamiento del `T.D.A`.

Aquí sería donde se almacenan las palabras válidas para el juego.

Tanto para este módulo como para el resto de módulos de esta práctica tenéis total libertad para su implementación.

Tenéis algunos diccionarios de ejemplo en PRADO, pero vosotros podéis usar los vuestros:

- Nombres de Pokémon.
- Nombres de artistas (sin espacios).
- ...

Contenido

Para jugar necesitamos un conjunto de letras aleatorias, para ello en el guion se propone el diseño de tres módulos, aunque tenéis total libertad:

- Módulo Letra.
- Módulo Conjunto de Letras: un conjunto de `Letra`.
- Módulo Bolsa de Letras: de donde se obtiene el conjunto de letras aleatorias.

En un fichero de letras tenemos la siguiente información para cada letra:

- La letra.
- La frecuencia (fabs.) de la letra.
- La puntuación de la letra.

Cada letra aparecerá en la `Bolsa de Letras` tantas veces como indique su fabs.. El conjunto aleatorio de letras lo muestrearemos **con reemplazamiento** de la `Bolsa de Letras`[1].

[1] Si empleáis la fabs. de la letra con otro criterio o generáis el conjunto aleatorio de otro modo, explicadlo (pero debería poder haber letras repetidas en el conjunto).

Contenido

Este es el módulo más importante pues es donde un ingeniero ha de brillar. Este módulo es el que se encarga de

$$\{\text{letras}\}_{\text{random}} \longrightarrow \{\text{mejores palabras}\} \subseteq \texttt{Diccionario}$$

Es decir, encontrar

- Las mejores palabras del diccionario. Para ello hay dos criterios:
  - La longitud de la palabra.
  - La puntuación de la palabra $= \sum_{\text{letra} \in \text{palabra}} \text{puntuación(letra)}$.
- Que se pueden formar con las letras de las que se dispone (**no es obligatorio usar todas las letras del conjunto**).

El algoritmo para encontrar esas palabras se puede implementar de diversas formas, y empleando distintas estructuras de datos (que como sabemos influye en la eficiencia del mismo).

Evidentemente tenemos dos enfoques triviales por fuerza bruta:

- 1. Recorrer el diccionario al completo.
  2. Escoger las mejores palabras del mismo que puedan formarse con mis letras.
- 1. Para cada posible combinación de mis letras.
  2. Ver si es una palabra del diccionario y quedarme con las mejores posibles.

Evidentemente ambos enfoques son $\mathcal{O}(n)$ donde:

- n = número de palabras en el diccionario.
- n = número de posibles combinaciones de letras que puedo obtener con mis letras.

¿Cuál de las dos soluciones es mejor? Pues dependerá del número de palabras en el diccionario y del número de combinaciones posibles que haya en nuestro diccionario.

Veamos para 8 letras no repetidas cuántas "palabras" podemos formar:[2]

$$\sum_{t=1}^{8} \frac{8!}{(8-t)!} = 109600$$

[2] Calculado con WolframAlpha.

Podríamos obviar el orden y ordenar las letras de cada palabra del diccionario de modo que:

$$\{\text{ave, vea, eva}\} \longrightarrow \text{aev}$$

- Ahora el número de posibles combinaciones (donde ya no se tiene en cuenta el orden) disminuye notablemente:[3]
  $\sum_{t=1}^{8} \binom{8}{t} = 255.$
- Pero esto supone trabajar con una **estructura de datos adicional**, de modo que dada una combinación devolvamos finalmente todas las palabras del diccionario que se formen con dicha combinación. No vamos a devolver "aev", sino "ave", "vea" y "eva".
  - Podría ser una tabla hash que esté indexada por la combinación de letras ordenada ("aev"), y en la que cada celda apunte a un conjunto que contenga las palabras del diccionario que se forman con esa combinación ({ave, vea, Eva}).

[3] Calculado con WolframAlpha.

Contenido

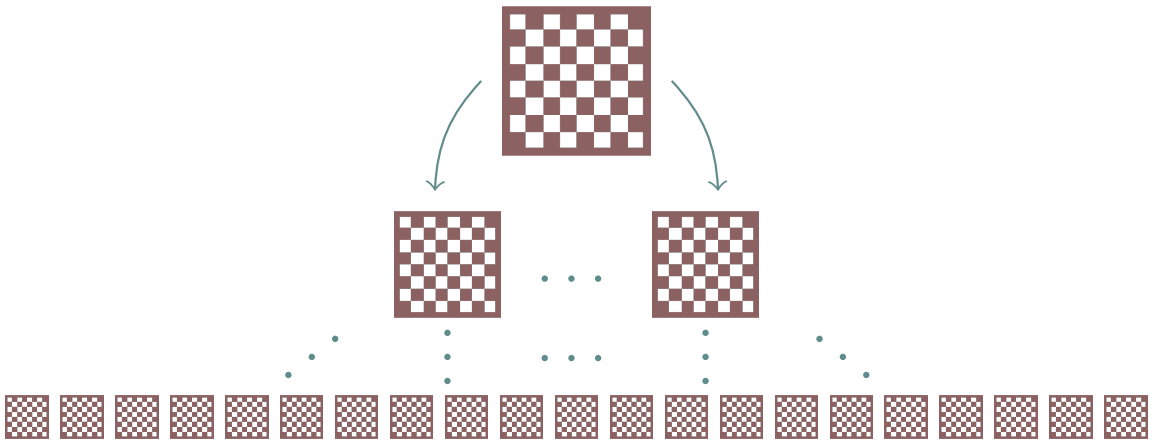
Vamos a ver una solución más elaborada y que funciona mejor que una solución trivial, podéis desarrollar vuestra propia solución y encontrar las deficiencias de esta.

En este caso vamos a tener en cuenta el orden en las combinaciones de letras, aunque podríamos incorporar a esta solución la idea comentada anteriormente para no tener en cuenta el orden.

La solución está inspirada en el uso de árboles para representar el espacio de estados de un juego[4].

Y así poder desarrollar una inteligencia artificial capaz de jugar a dicho juego, explorando su espacio de estados y eligiendo el mejor movimiento posible a partir del estado actual. Podemos pensar por ejemplo en el ajedrez.

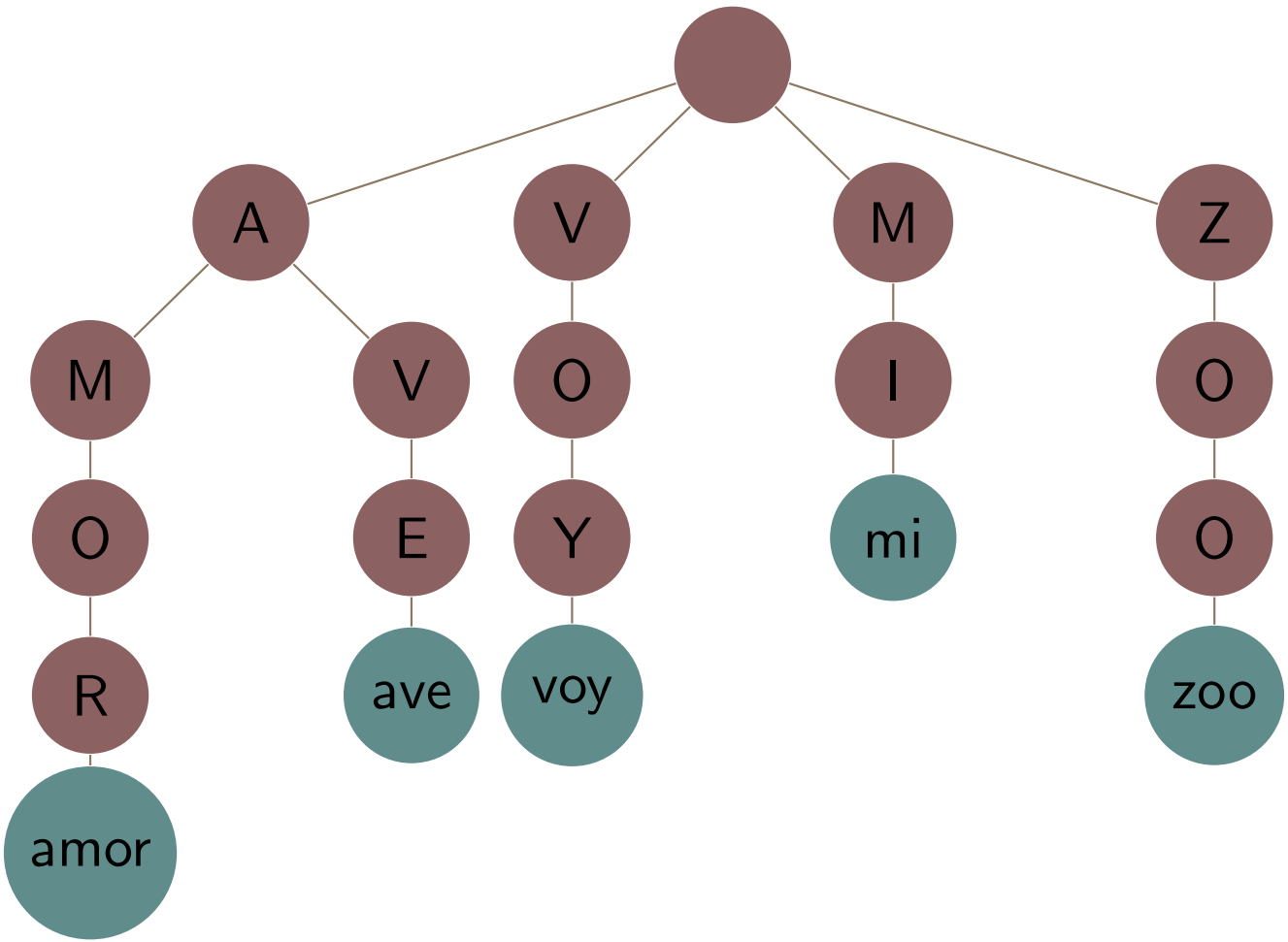
[4] Podemos ver este vídeo para entender mejor de qué estamos hablando.



- A cada nodo[5] se le puede asociar una puntuación basada en la posición y figuras actuales, tanto propias como del adversario.
- O bien basada en la puntuación de los estados futuros a partir de ese nodo.
- Un jugador será mejor cuanta más profundidad del árbol sea capaz de analizar. $\Longrightarrow$ Limitando la profundidad de análisis tenemos una I.A. que juega mejor o peor.

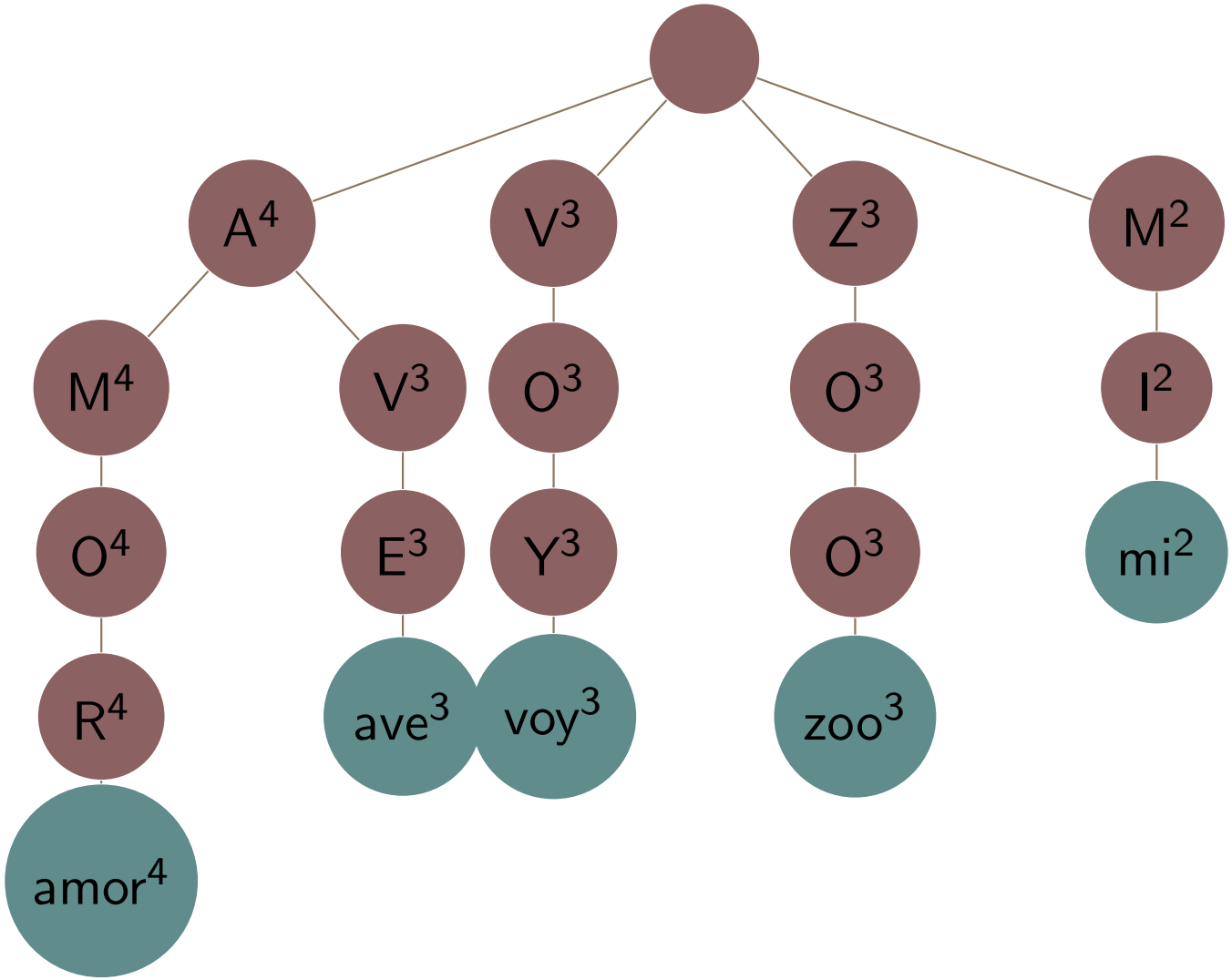
[5] El icono del tablero usado en los nodos ha sido creado por freepik de www.flaticon.com.

Pues bien, dado, por ejemplo, el diccionario {amor, ave, mi, voy, zoo}. La propuesta es crear un árbol como el siguiente:



- En cada hoja del árbol tenemos una palabra del diccionario.
- Y los nodos en el camino desde la raíz a cada hoja contienen las letras que forman la palabra.

- Ahora se puntúa cada una de las hojas según el criterio elegido (en este ejemplo vamos a usar la longitud).
- Promocionamos esa puntuación de los hijos a sus padres, de modo que el padre hereda la puntuación del mejor de sus hijos.
- Ordenaremos los hijos de cada nodo de izquierda a derecha, de mayor a menor puntuación. Favoreciendo las estrategias de poda que veremos a continuación.


A4
V3
Z3
M2
M4
V3
O3
O3
I2
O4
E3
Y3
O3
mi2
R4
ave3
voy3
zoo3
amor4

La ventaja que tiene esta estrategia o esta estructura de datos es que nos permite encontrar las mejores soluciones posibles para nuestro conjunto de letras aleatorias sin necesidad de explorar el diccionario al completo.

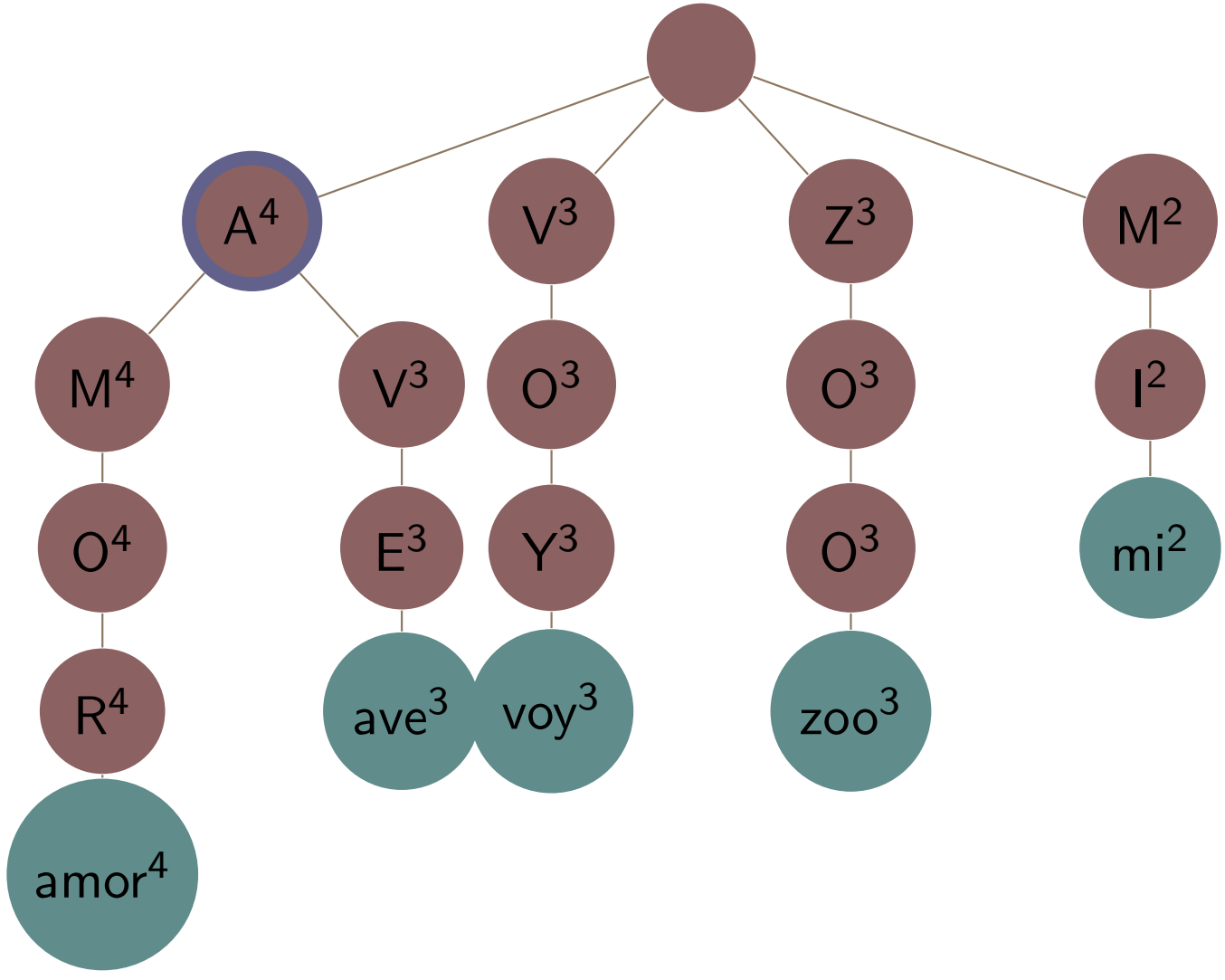
Para ello establecemos dos estrategias de poda:

- Si no dispongo de la letra del siguiente nodo a explorar $\Longrightarrow$ no sigo explorando ese camino y vuelvo hacia atrás.
- Si la puntuación de ese nodo (la mejor solución a la que puedo llegar a partir de él) es peor que las encontradas hasta el momento $\Longrightarrow$ tampoco exploro ese camino.

Vamos a suponer que disponemos del siguiente conjunto de letras:

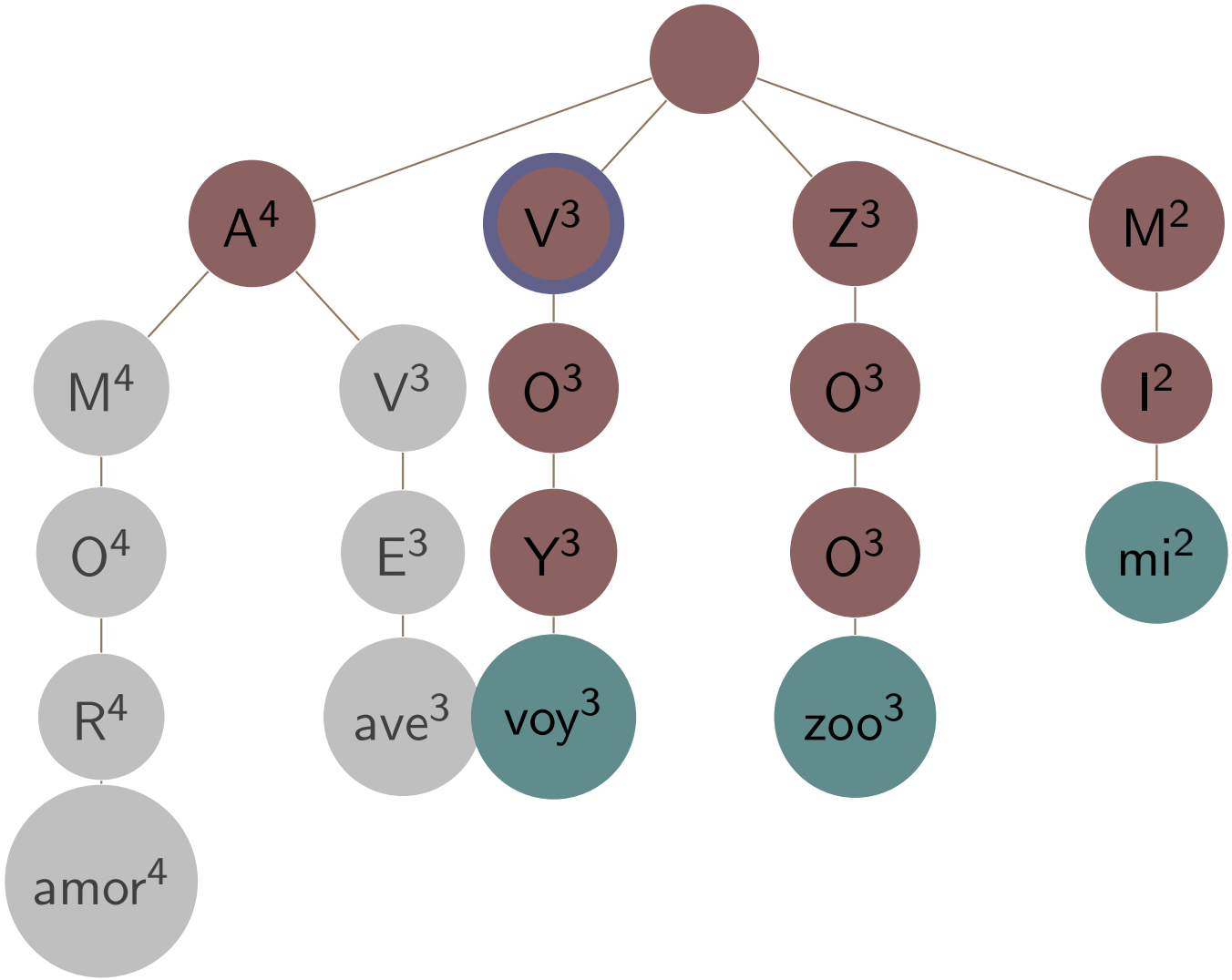
O V Z C Y M

Comenzaremos a explorar el árbol desde la raíz, y de izquierda a derecha.



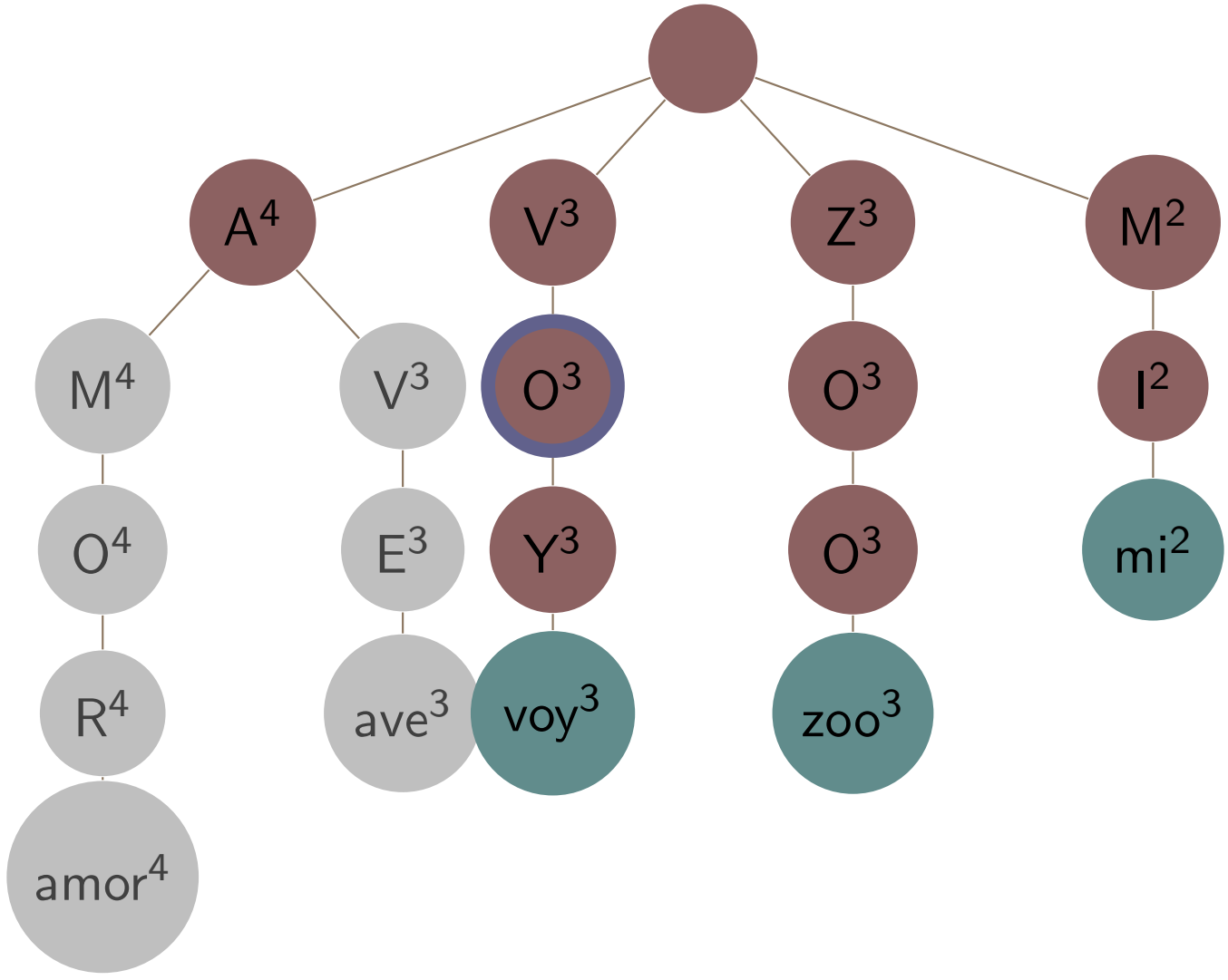
No disponemos de la A con lo que este camino no lo exploramos.

$\Longrightarrow$ Descartamos dos palabras del diccionario en un solo paso.

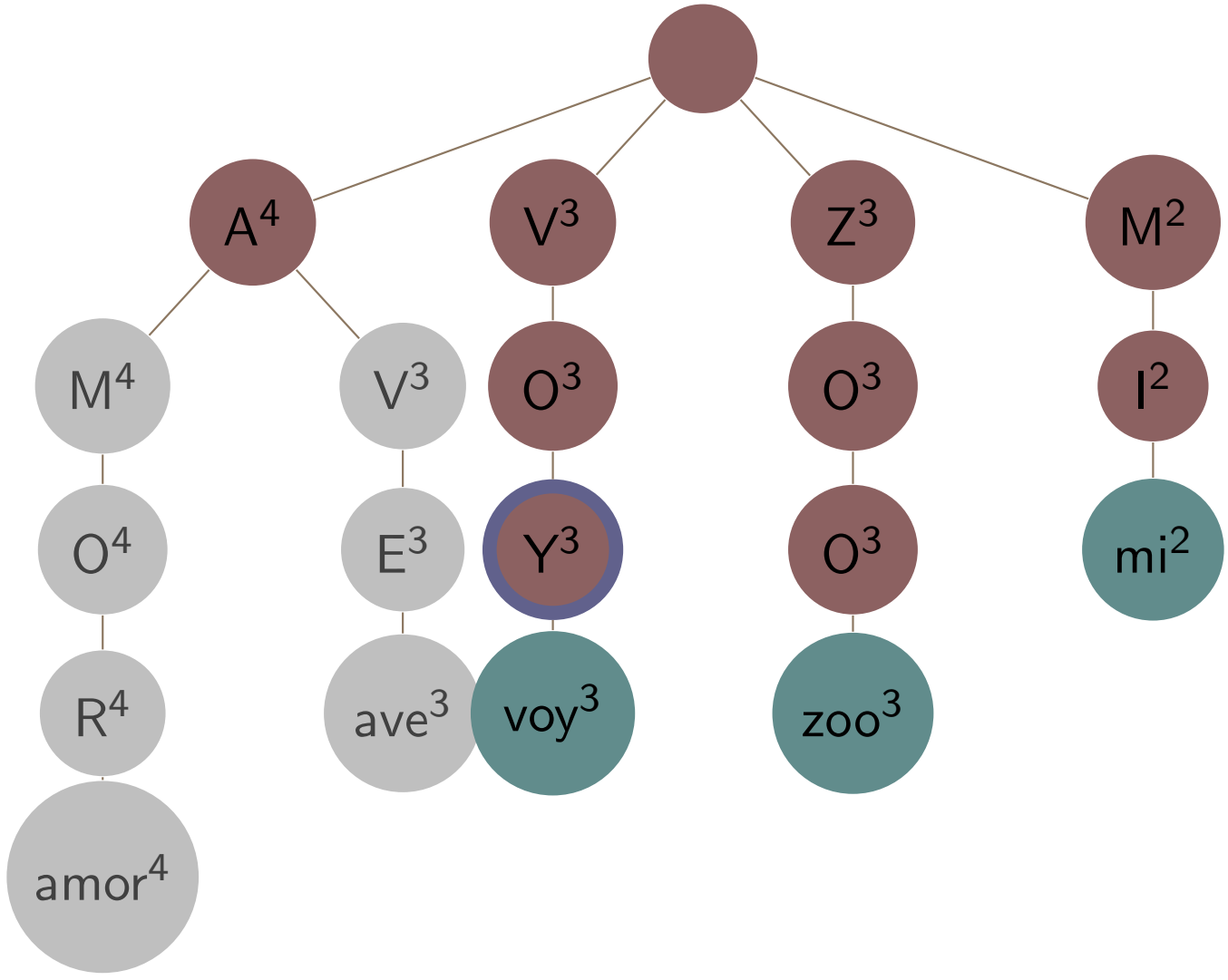


Tengo la V $\Longrightarrow$ Paso a explorar este nodo. $\Longrightarrow$ Saco la letra de mi conjunto (no puedo usar la misma letra varias veces).

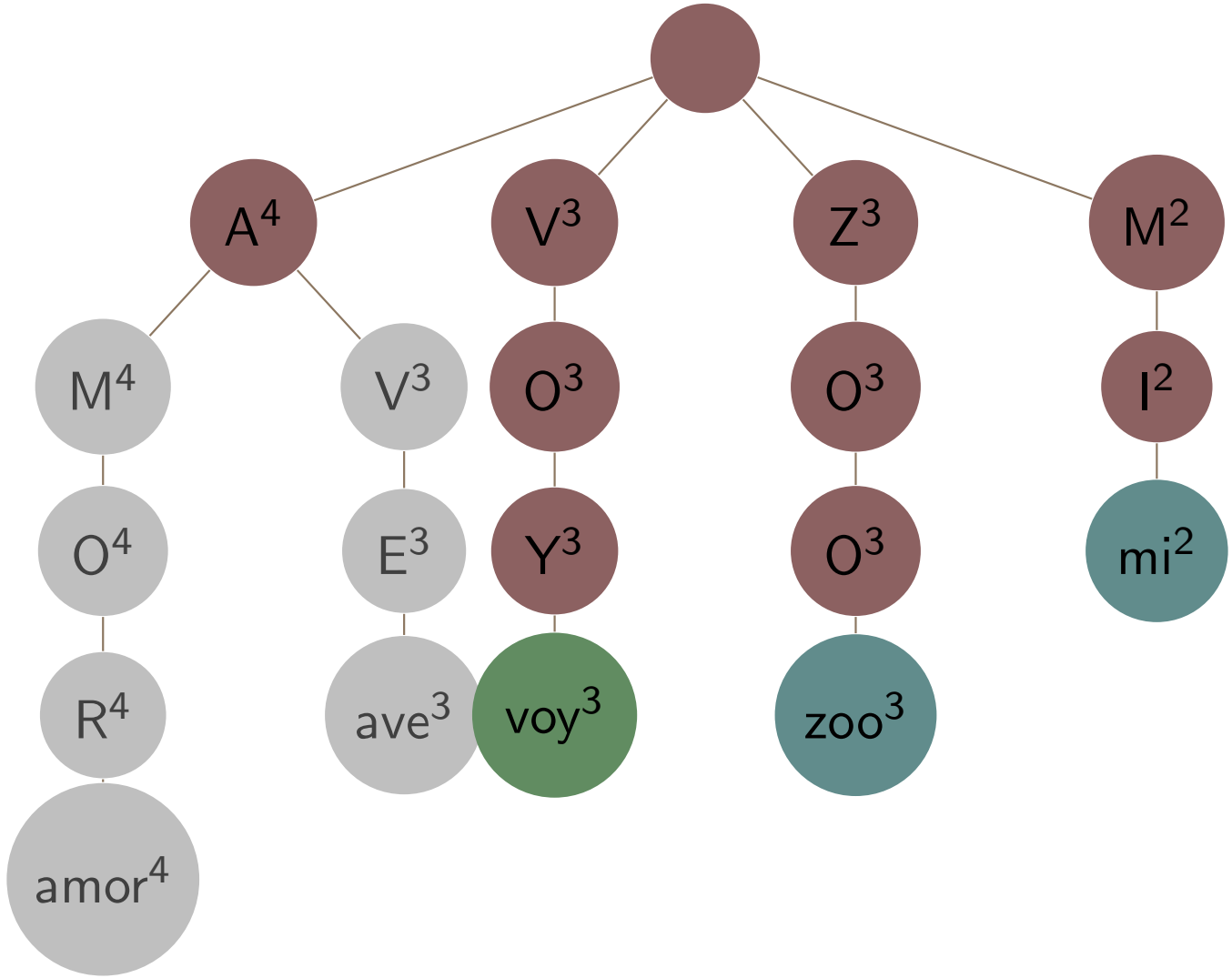
O V Z C Y M



O V Z C Y M

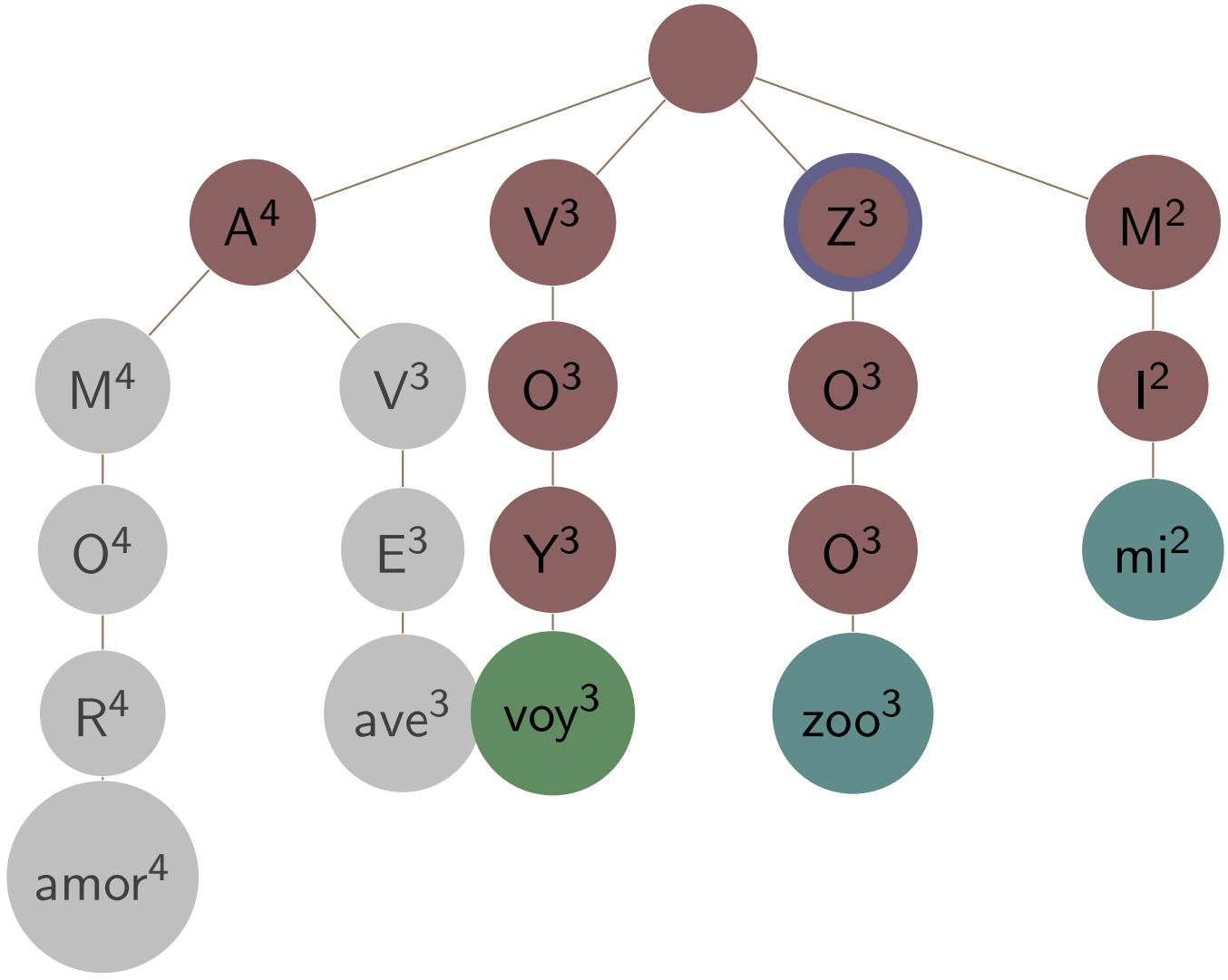


O V Z C Y M

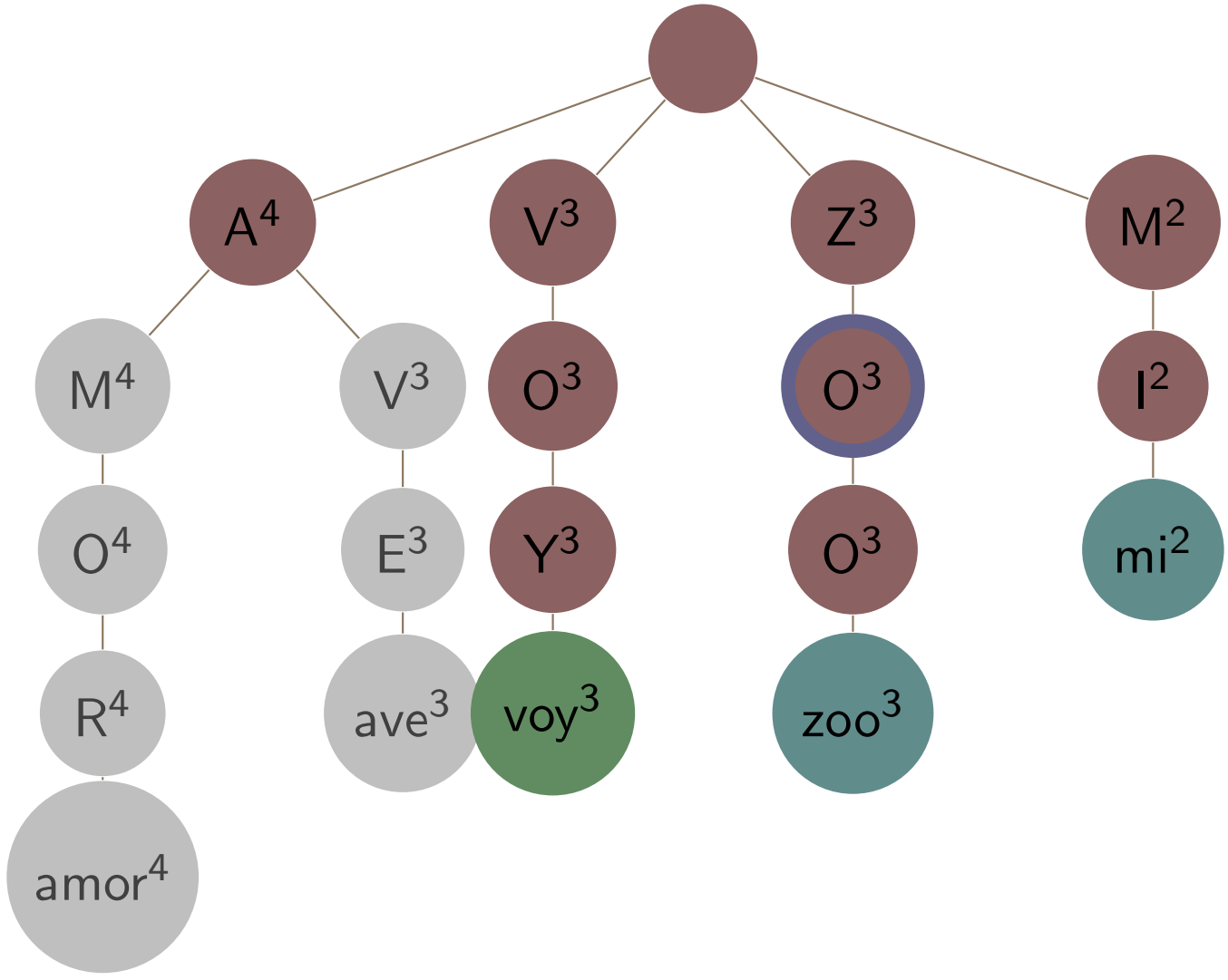


He llegado a un nodo hoja con lo que añado esta palabra a mi conjunto de soluciones encontradas.

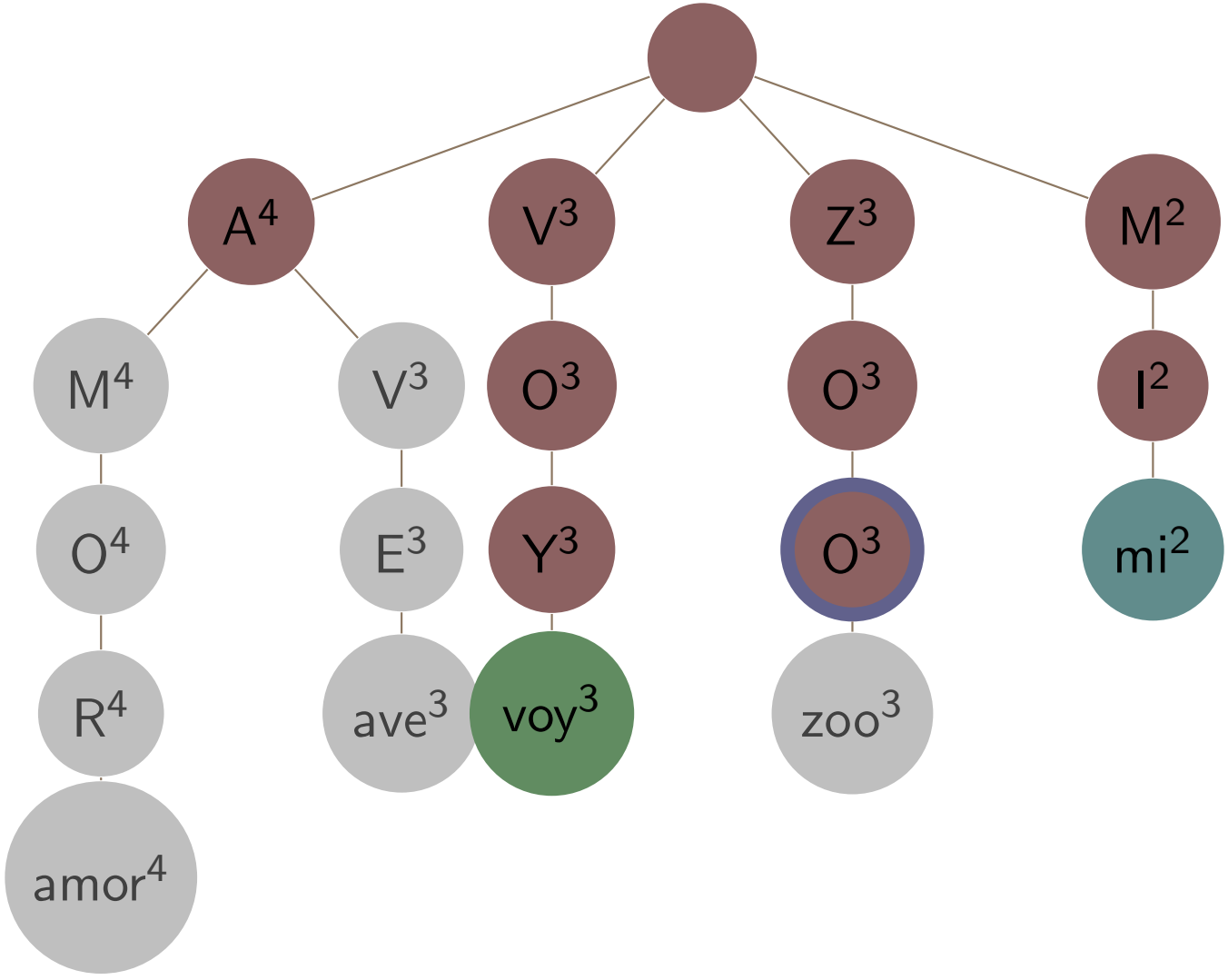
Vuelvo hacia atrás recuperando las letras que he ido utilizando.



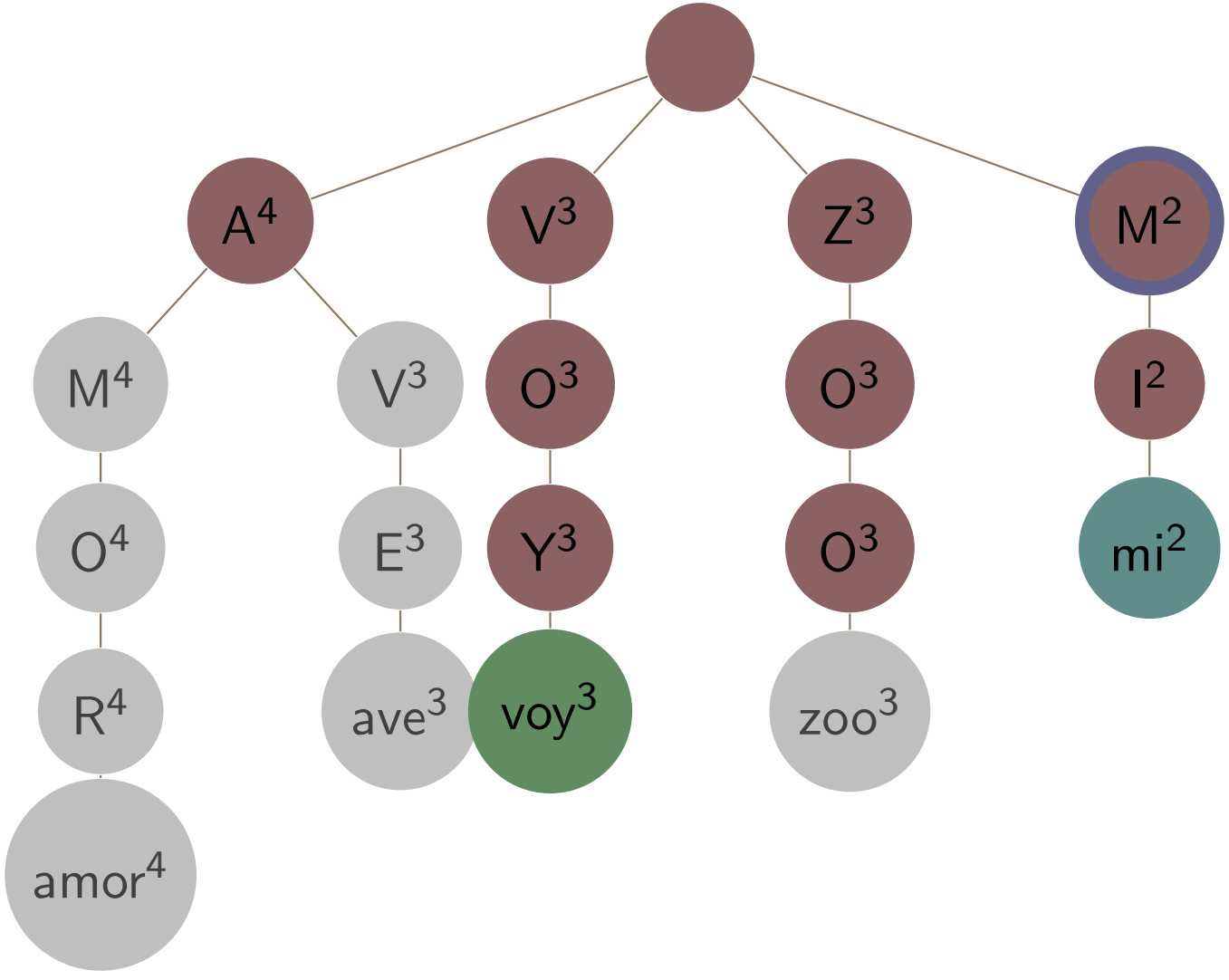
O V Z C Y M

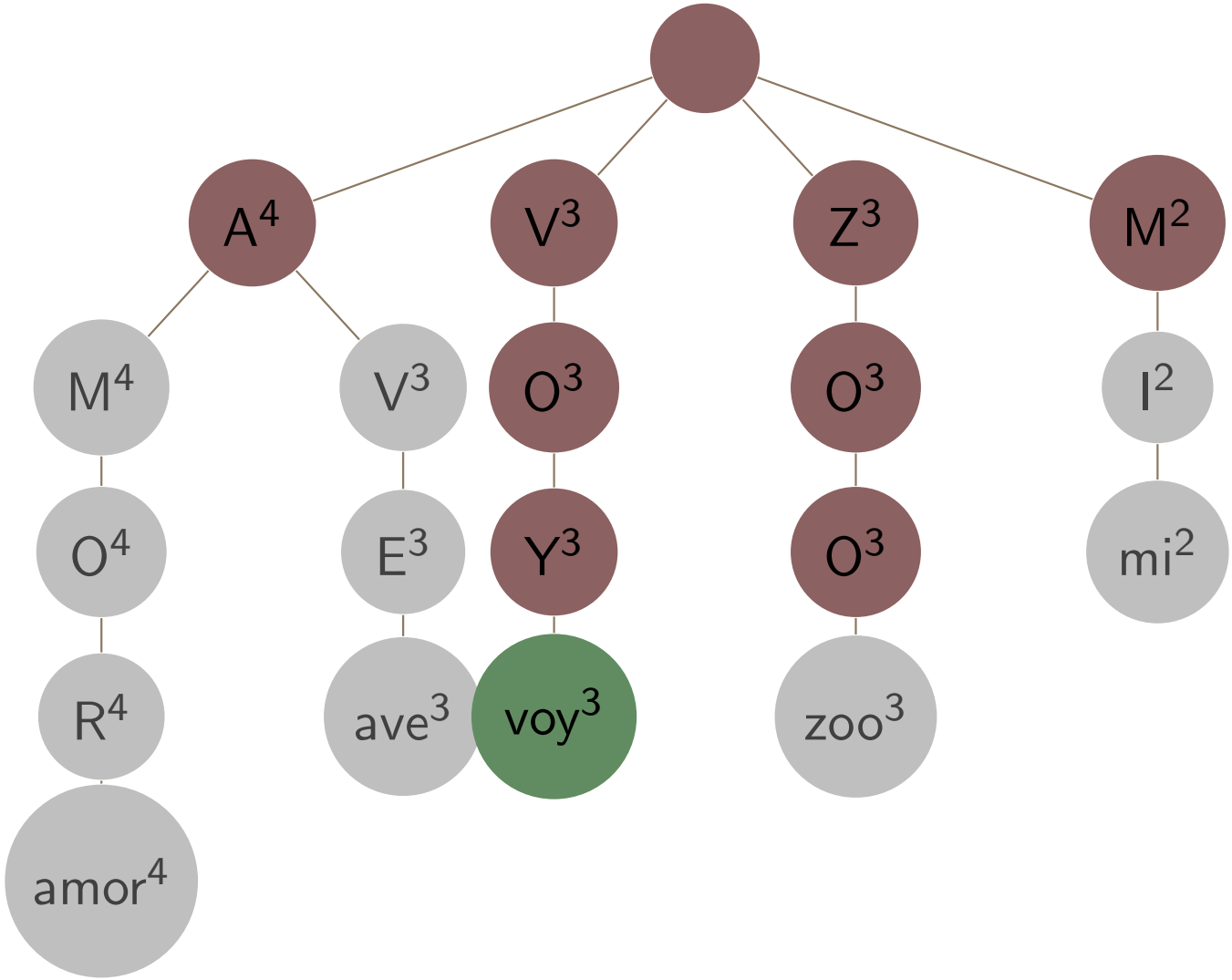


O V Z C Y M



No tengo otra O con lo que no puedo seguir explorando este camino.


A4
V3
Z3
M2
M4
V3
O3
O3
I2
O4
E3
Y3
O3
mi2
R4
ave3
voy3
zoo3
amor4



Aunque dispongo de una M, desde este nodo no puedo llegar a ninguna solución que sea, al menos, igual de buena que las que ya he encontrado, con lo que no exploro este camino.

- Podemos observar las ventajas de usar esta estrategia para no explorar todo el diccionario.
- Evidentemente la construcción del árbol consumirá algún tiempo, pero en la mayoría de los casos merecerá la pena. Sobre todo si jugamos varias rondas con el mismo diccionario.

- Un problema que tiene esta estructura de datos es lo costoso que resultan las modificaciones de un nodo concreto.
- Los nodos están ordenados por puntuación y no por orden alfabético. $\Longrightarrow$ La búsqueda de un nodo por su etiqueta es $\mathcal{O}(n)$.
- Además, para mantener un `set` de la STL ordenado tras una modificación, tendremos que sacar el elemento y volver a meterlo (no podemos aplicar el método `algorithm::sort` sobre un `set`).
- Una solución para tener accesos $\mathcal{O}(1)$ a los hijos de un nodo por su etiqueta es usar un `unordered_set`. En mi caso sería un `unordered_set<Node**>`, generando una especie de tabla hash indexada con la etiqueta de los nodos y que apunta a los hijos de un `Node` que son `Node**`.

En cualquier caso esto es sólo una propuesta, que puede ser mejorada, criticada y cambiada por una diferente. El trabajo de un ingeniero es buscar la mejor solución al problema que se le plantea.

- Esta práctica son 1.5 puntos de la calificación final.
- La fecha de entrega es el 7/1/19.
- Se puede hacer por parejas.