Grafos

Búsqueda en anchura

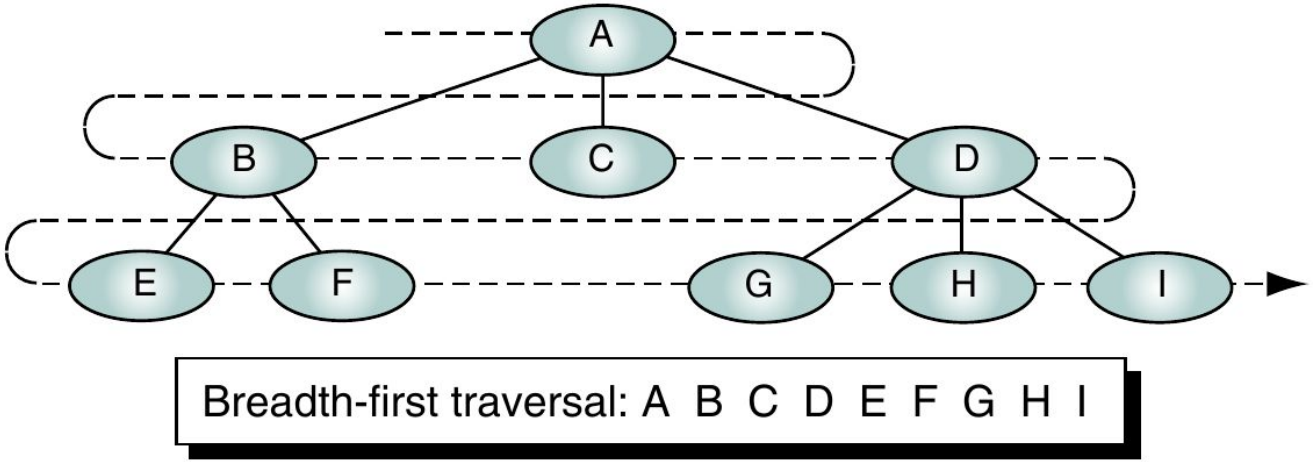
Profesor: Néstor Suat-Rojas. Ing., M.Sc.

Recorridos

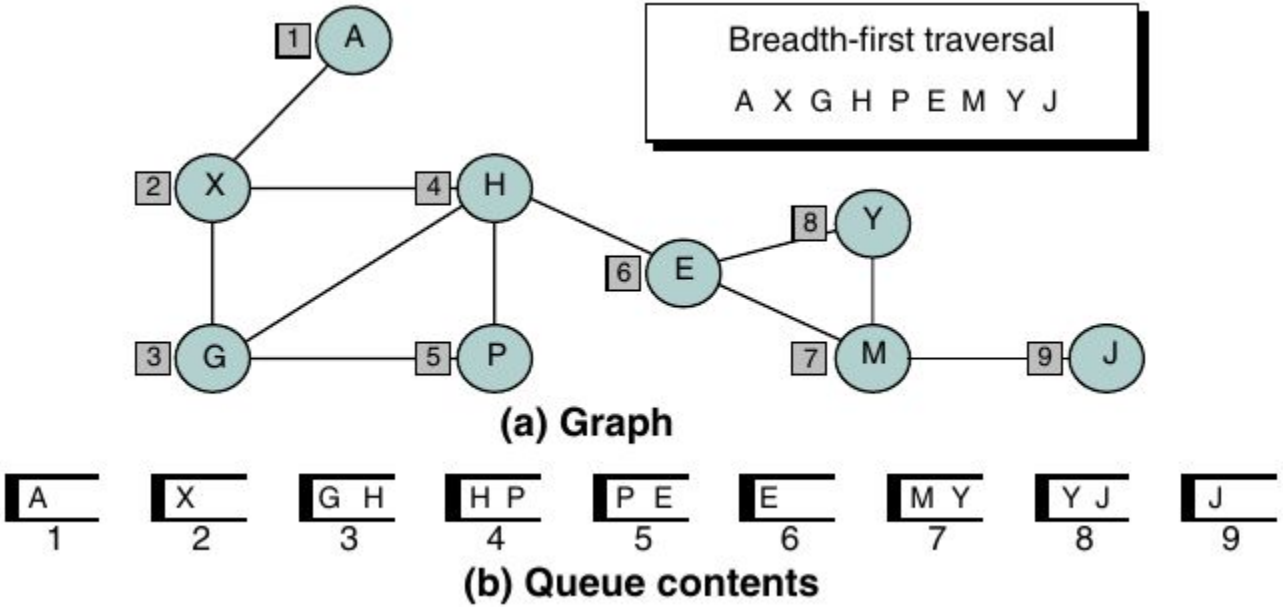
- Más de un algoritmo requiere recorrer todos los vértices en el grafo, procesarlos una sola vez.
- Sin embargo, dada la definición de un grafo, tenemos más de una ruta para llegar a un vértice.
- Soluciones tradicionales incluyen una bandera en cada vértice para determinar si el vértice ya fue visitado o no.

Recorrido en anchura

Procesa todos los vértices adyacentes de un vértice antes de continuar con el siguiente nivel.



Recorrido en anchura

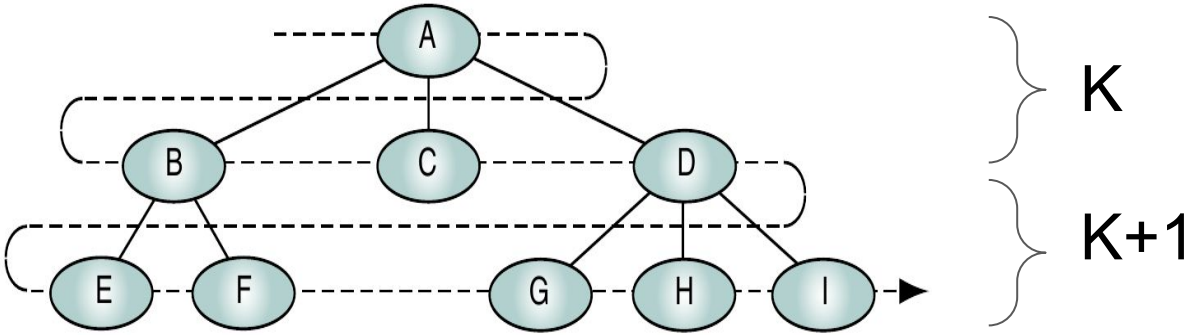


Búsqueda en anchura

Búsqueda en anchura (BEA)

Encontrar todos los vértices para los cuáles hay una ruta a partir de un vértice inicial.

1. Encuentra primero todos los vértices a K distancia antes de los K+1 distancia.

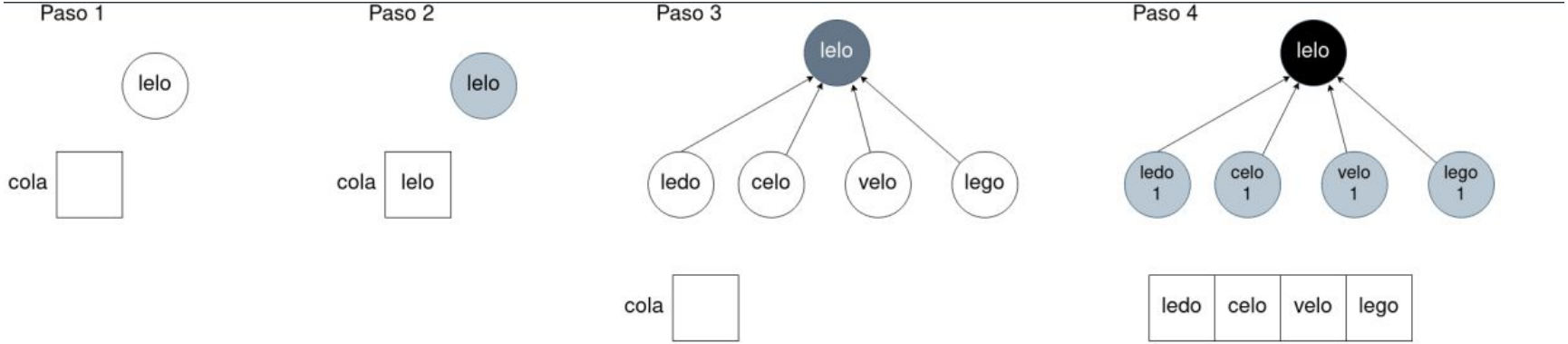


Búsqueda en anchura (BEA)

2. BEA pinta los vértices de blanco, gris o negro.

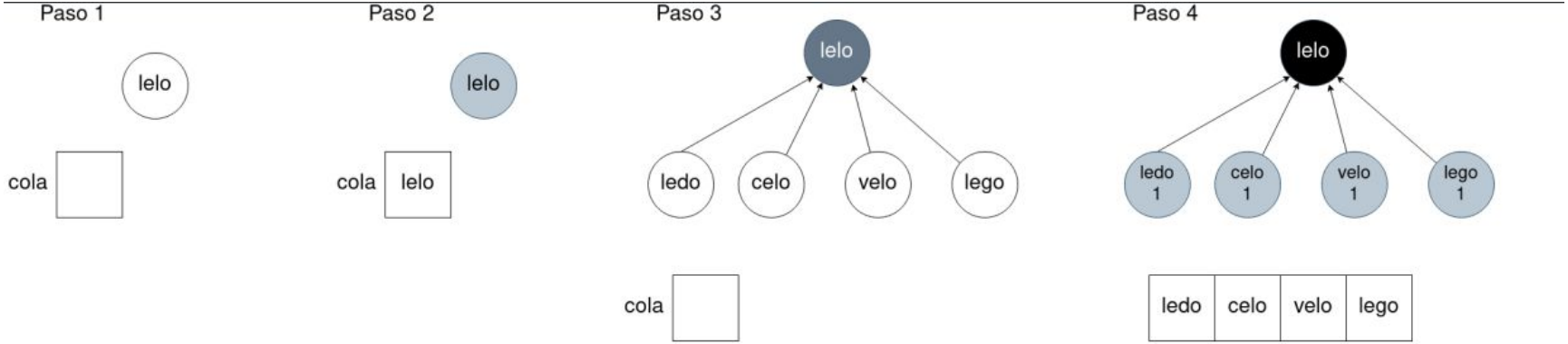
- **Blanco:** no explorado.
- **Gris:** En exploración.
- **Negro:** Explorado por completo.

Cuando un vértice es negro no tiene vértices blancos adyacentes a él.



Búsqueda en anchura (BEA)

3. Utiliza una cola para realizar seguimiento del siguiente vértice por explorar.



Búsqueda en anchura (BEA)

4. Agrega atributos a la clase Vértice.

```
template<class T>
class Vertex{
public:
    T data;
    int inDegree;
    int outDegree;
    vector<Edge<T>*> connectedTo;

    Vertex<T>* predecessor;
    int distance;
    char color;
};
```

El problema de la escalera de palabras

Transformar la palabra “LELO” en la palabra “AGIL”.

1. El cambio se produce una letra a la vez.
2. La nueva palabra en cada cambio debe existir en el diccionario.

El problema de la escalera de palabras

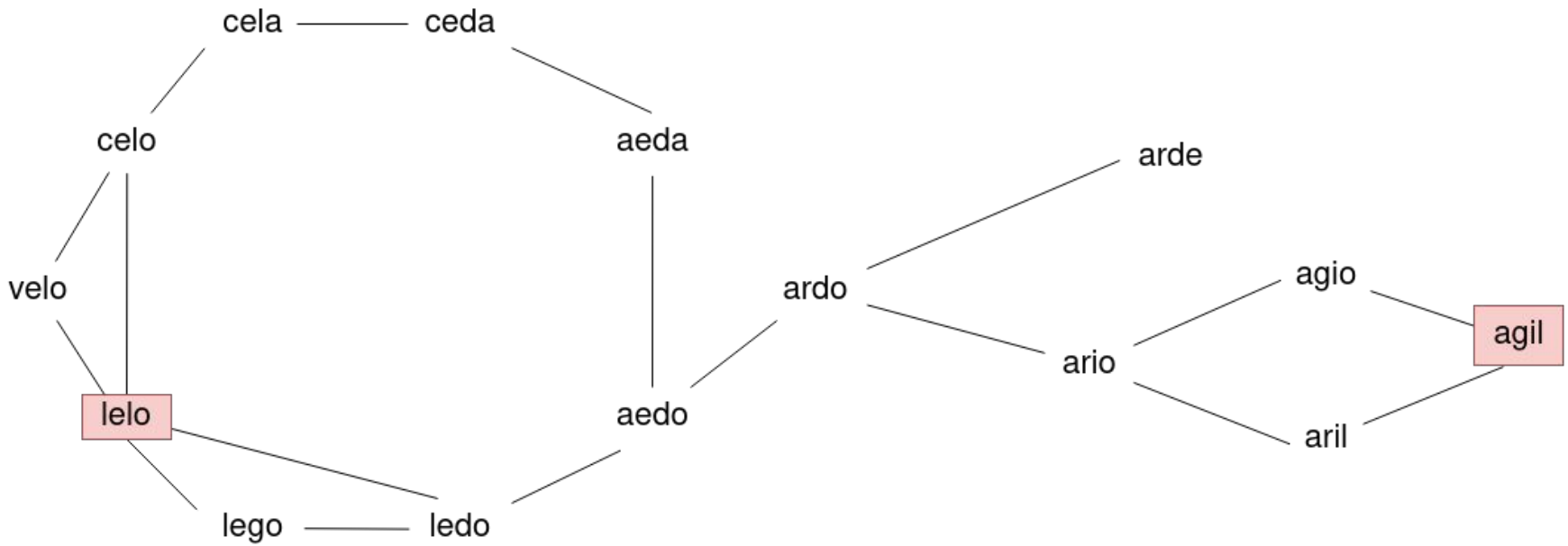
Transformar la palabra “LELO” en la palabra “AGIL”.

1. El cambio se produce una letra a la vez.
2. La nueva palabra en cada cambio debe existir en el diccionario.

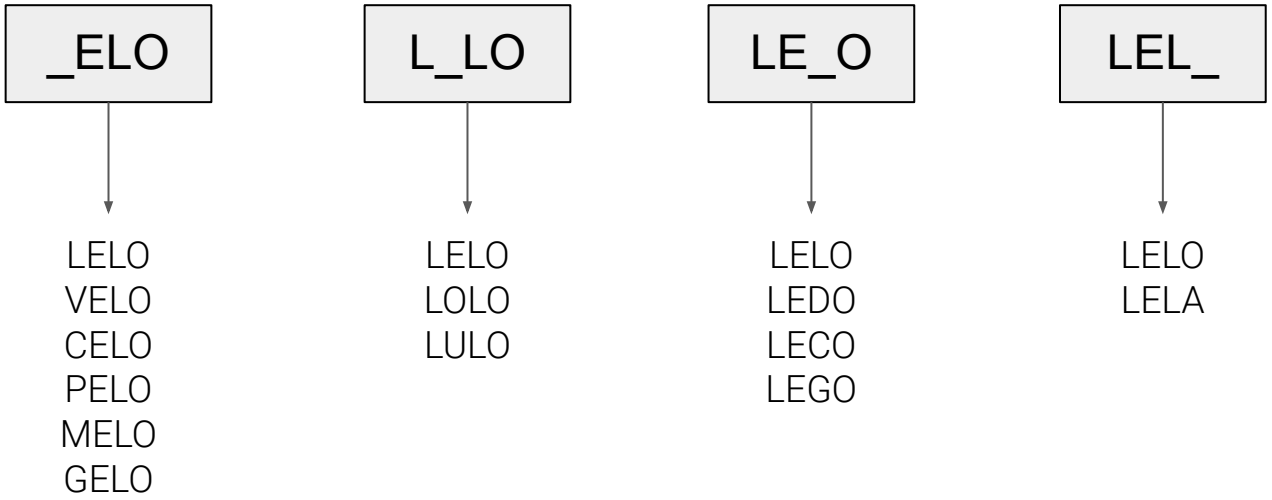
```
LELO
LEDO
AEDO
ARDO
ARIO
AGIO
AGIL
```

- Representar las relaciones entre las palabras en un grafo.
- Usar BEA para encontrar la ruta eficiente.

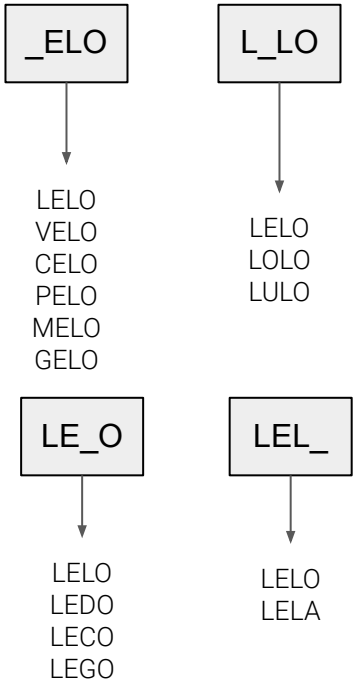
Construcción del grado de la escalera de palabras



Construcción del grado de la escalera de palabras



Construcción del grafo de la escalera de palabras




```
Graph<string> buildGraph (){
    map<string, vector<string> > dict = {};
    Graph<string> g;
    string word = "";
    ifstream file("/listapalabras.txt" , ifstream::in);
    if(!file){
        cout << "Error reading file"  << endl;
    }
    if(file.is_open()){
        // crear baldes de palabras que se diferencian por una letra
        while(getline(file, word)){
            for(int i=0; i<word.size(); i++){
                string bucket = word.substr(0,i)
                              + "_"
                              + word.substr(i+1, word.size());
                dict[bucket].push_back(word);
            }
        }
        file.close();
    }
    //# agregar vértices y aristas para palabras en el mismo balde
    for(auto[k,v]: dict){
        for(string word1: dict[k]){
            for(string word2: dict[k]){
                if(word1 != word2){
                    g.addEdge(word1, word2);
                }
            }
        }
    }
    return g;
}
```

Implementación de la búsqueda en anchura

Algoritmo:

1. Comienza con el vértice inicial (`begin`) y lo pinta de gris para mostrar que está siendo explorado.
2. A `begin` los valores de `distancia` y `predecesor` se inicializan:
   a. `distancia` = 0
   b. `predecesor` = **None**
3. `begin` se coloca en una cola (`queue`).

Implementación de la búsqueda en anchura

Algoritmo:

1. Comienza con el vértice inicial (`begin`) y lo pinta de gris para mostrar que está siendo explorado.
2. A `begin` los valores de `distancia` y `predecesor` se inicializan:
   a. `distancia` = 0
   b. `predecesor` = **None**
3. `begin` se coloca en una cola (`queue`).

```
begin->distance = 0;
begin->predecessor = 0;
queue<Vertex<string>*> queueVertex;
queueVertex.push(begin);
```

Implementación de la búsqueda en anchura

Algoritmo:

A medida que se examina cada nodo en la lista de adyacencia, se comprueba su color. Si es `blanco`, el vértice no ha sido explorado, y suceden cuatro cosas:

4. El nuevo vértice inexplorado vecino (`neighbor`), es coloreado de gris.
5. Al `predecesor` de `neighbor` se le asigna el nodo actual `curVertex`.
6. A la distancia de `neighbor` se le asigna la distancia de `curVertex + 1`.
7. `neighbor` se agrega al final de una cola.

Implementación de la búsqueda en anchura

Algoritmo:

4. El nuevo vértice inexplorado vecino (`neighbor`), es coloreado de gris.
5. Al `predecesor` de `neighbor` se le asigna el nodo actual `curVertex`.
6. A la distancia de `neighbor` se le asigna la distancia de `curVertex + 1`.
7. `neighbor` se agrega al final de una cola.

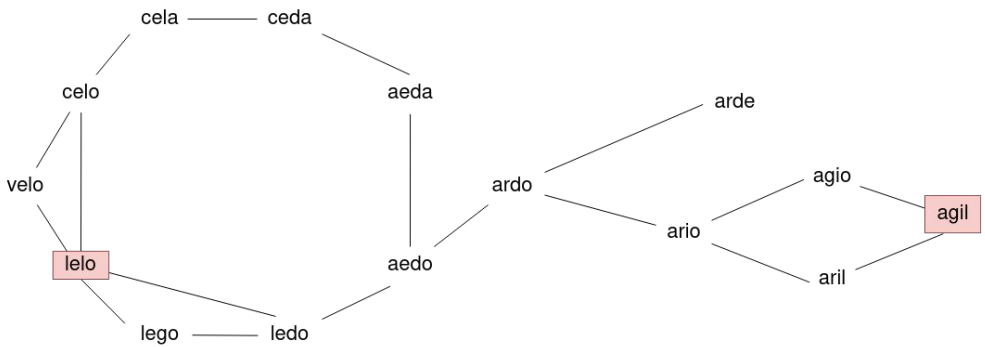
```
while(queueVertex.size() > 0){
    Vertex<string>* curVertex = queueVertex.front();
    queueVertex.pop();
    for(Edge<string>* neighbor: curVertex->connectedTo){
        if(neighbor->to->color == 'w') { // white
            neighbor->to->color = 'g'; // grey
            neighbor->to->distance = curVertex->distance +
1;
            neighbor->to->predecessor = curVertex;
            queueVertex.push(neighbor->to);
        }
    }
    curVertex->color = 'b'; //black
}
```

BEA

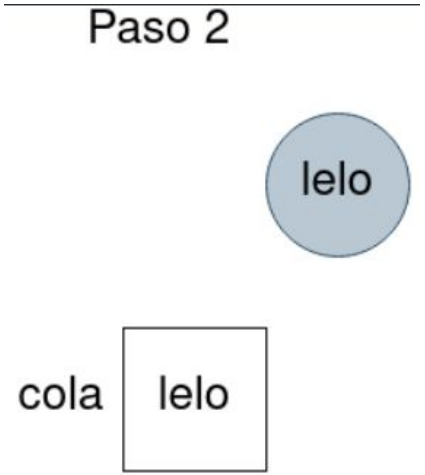
Paso 1

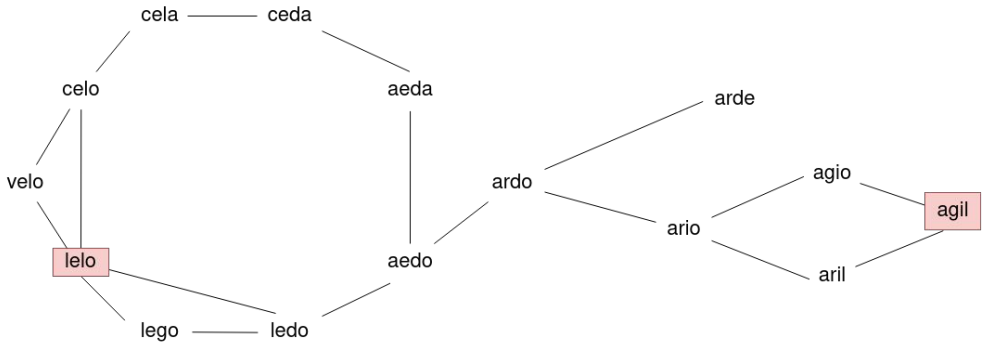
lelo

cola

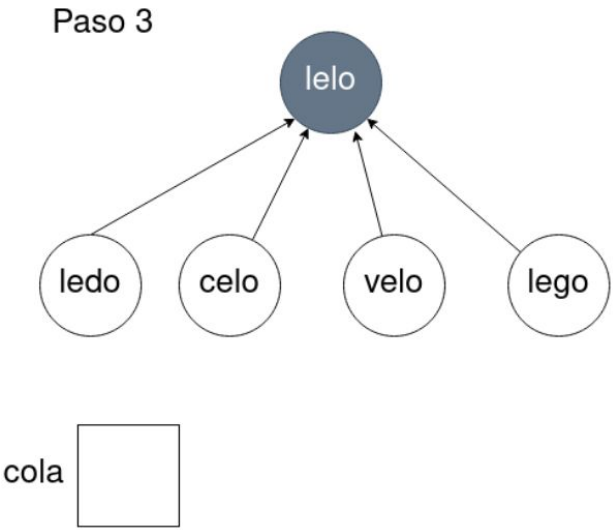


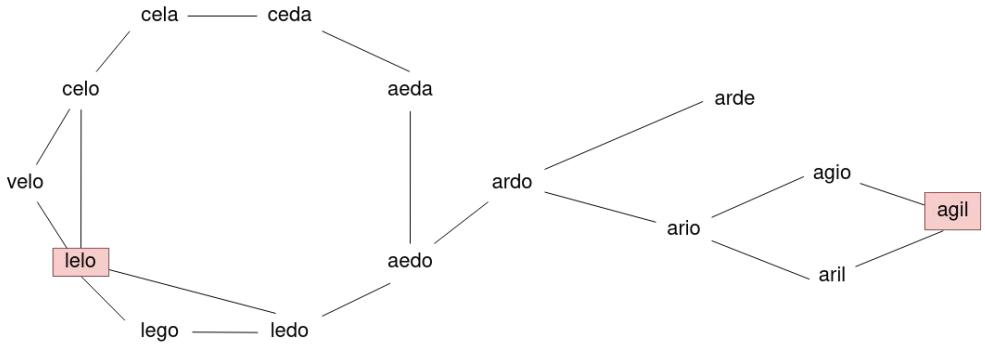
BEA


Paso 2
lelo
cola
lelo


cela
ceda
celo
aeda
velo
ardo
arde
agio
agil
ario
lelo
aedo
aril
lego
ledo

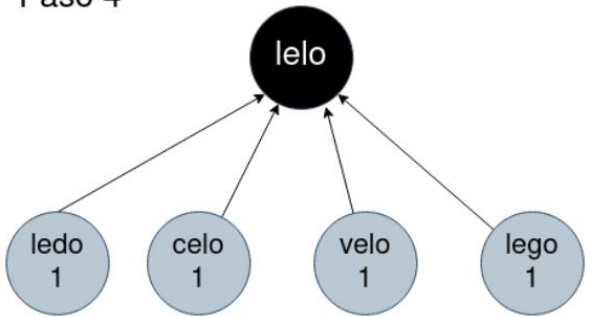
BEA


Paso 3
lelo
ledo
celo
velo
lego
cola

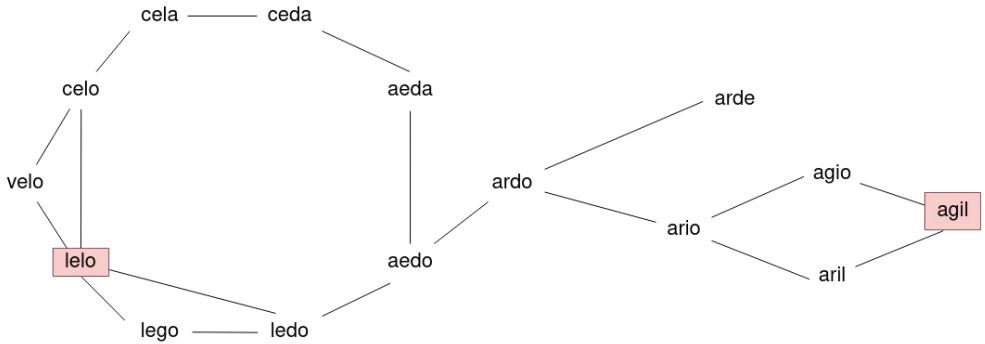

cela
ceda
celo
aeda
velo
ardo
arde
agio
agil
ario
lelo
aedo
aril
lego
ledo

BEA

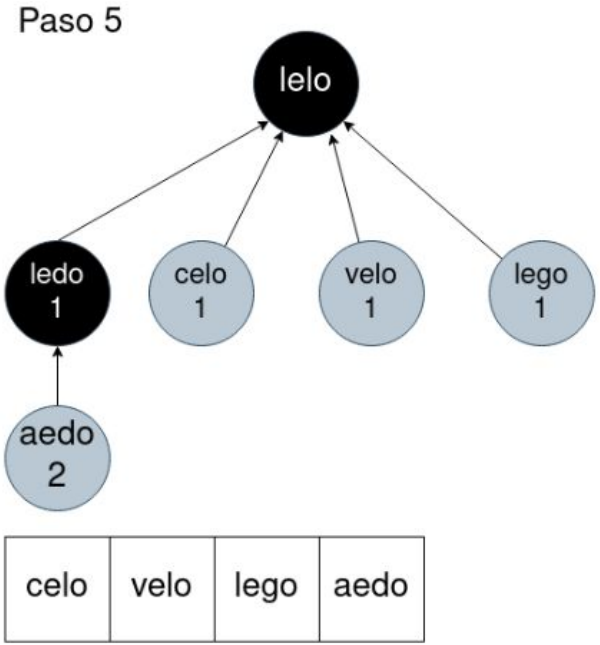
Paso 4

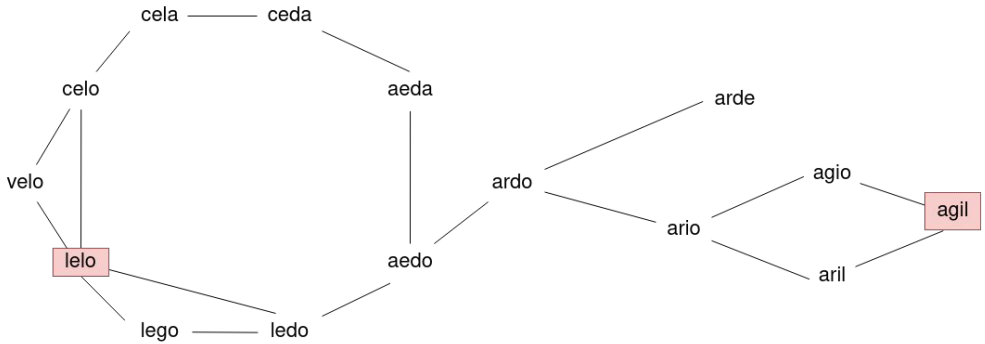


ledo	celo	velo	lego

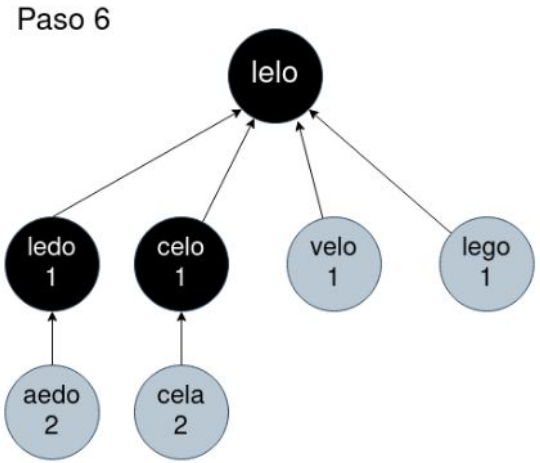


BEA

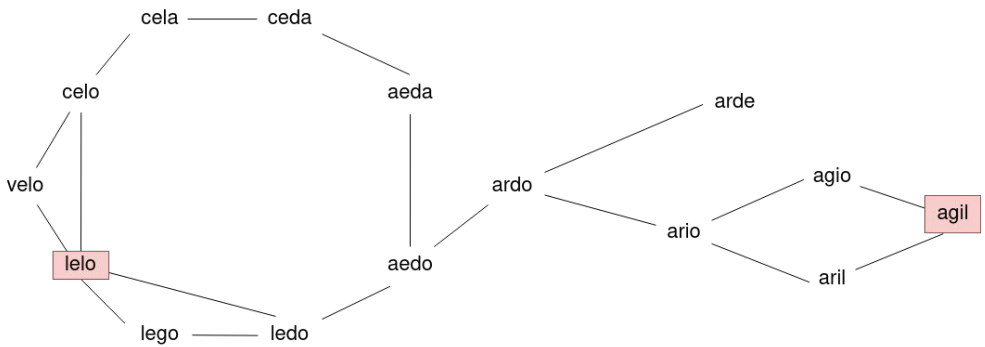

Paso 5
lelo
ledo 1
celo 1
velo 1
lego 1
aedo 2
celo
velo
lego
aedo


cela
ceda
celo
aeda
velo
ardo
arde
agio
agil
ario
lelo
aedo
aril
lego
ledo

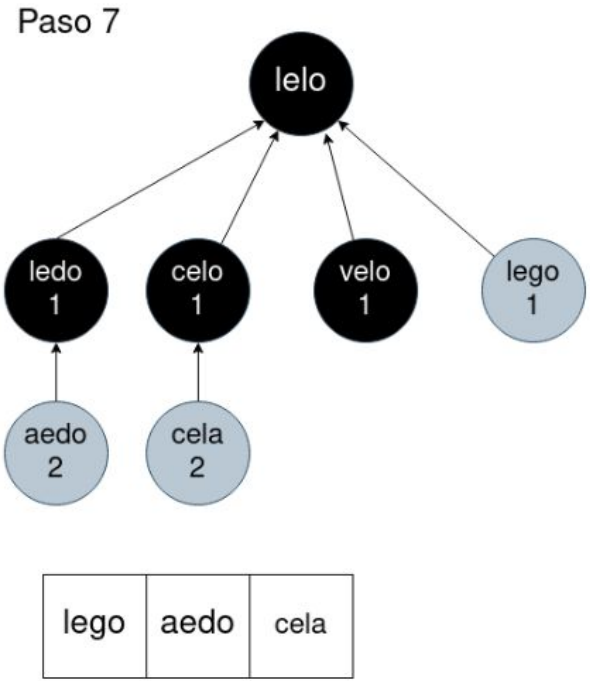
BEA

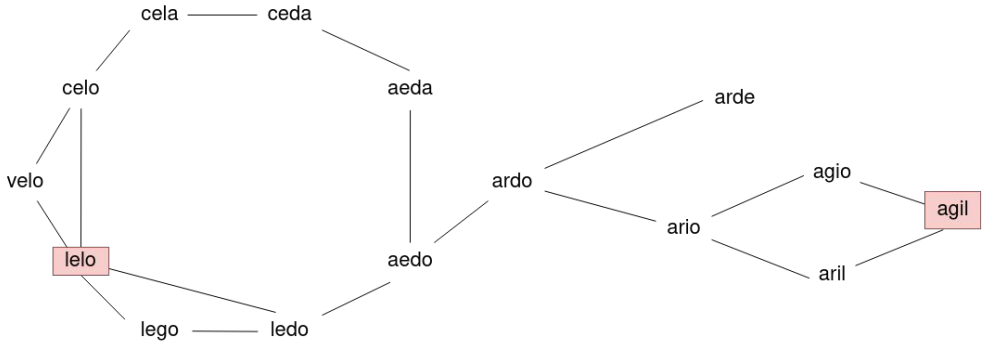


velo	lego	aedo	cela

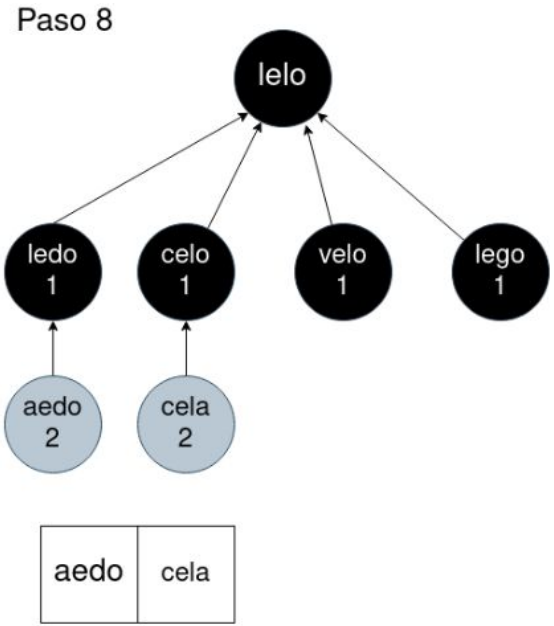


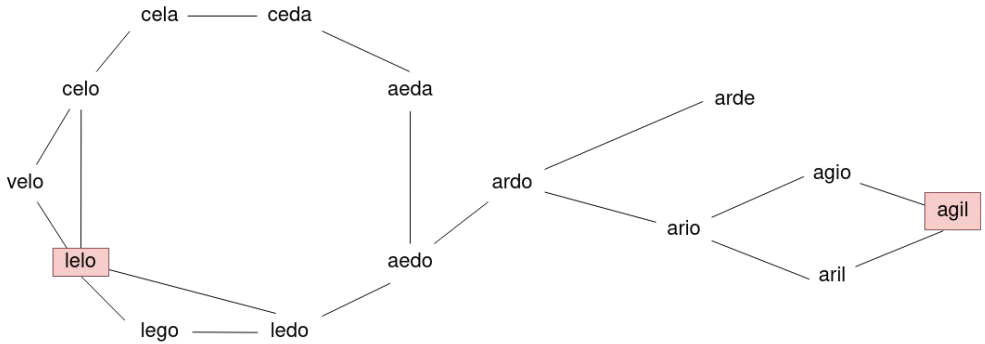
BEA


Paso 7
lelo
ledo 1
celo 1
velo 1
lego 1
aedo 2
cela 2
lego
aedo
cela

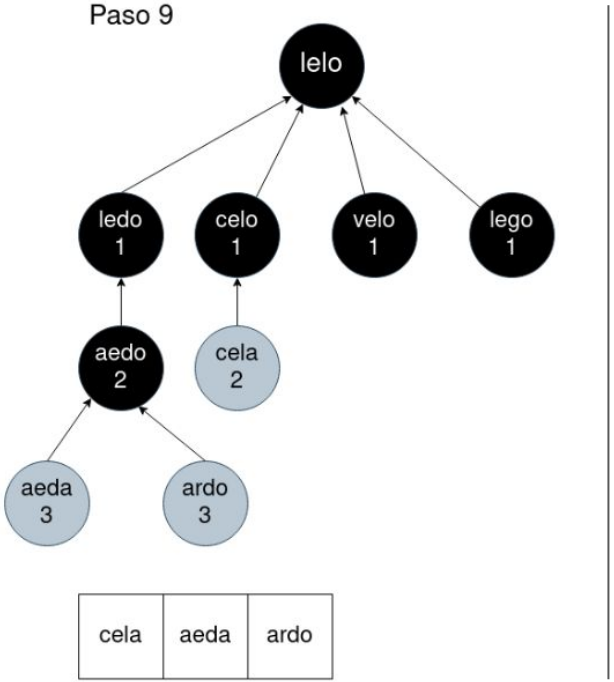

cela
ceda
celo
aeda
velo
lelo
aedo
lego
ledo
ardo
arde
ario
agio
agil
aril

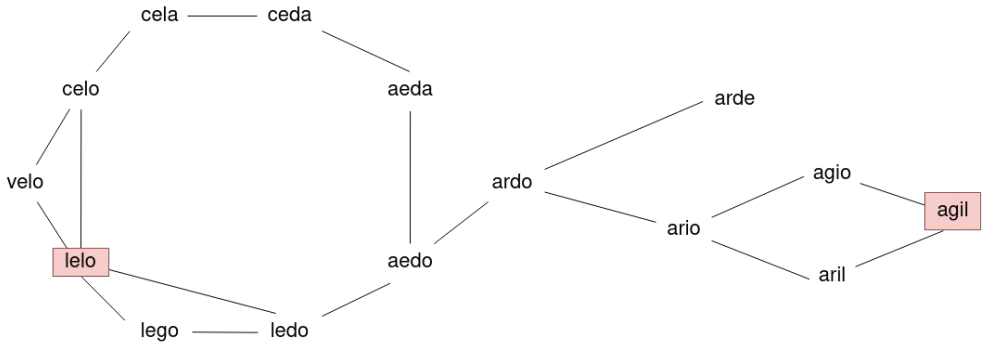
BEA


Paso 8
lelo
ledo 1
celo 1
velo 1
lego 1
aedo 2
cela 2
aedo
cela

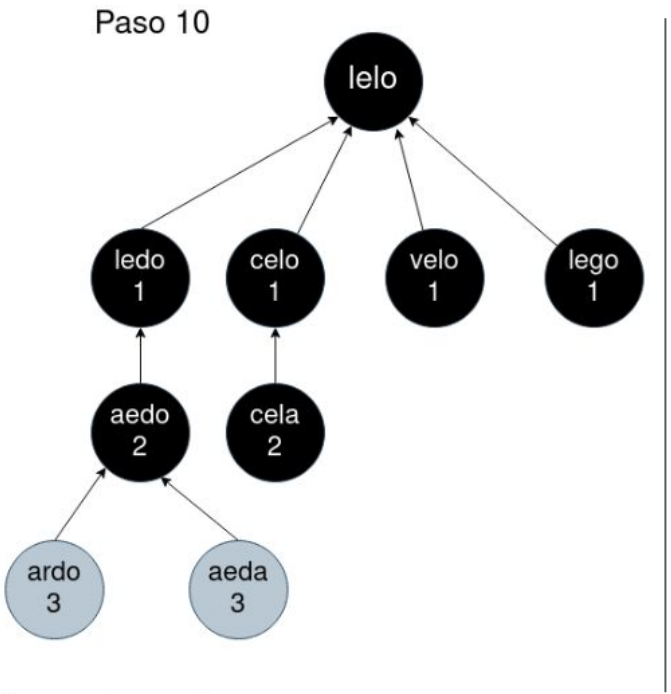

cela
ceda
celo
aeda
velo
ardo
arde
agio
agil
ario
lelo
aedo
aril
lego
ledo

BEA

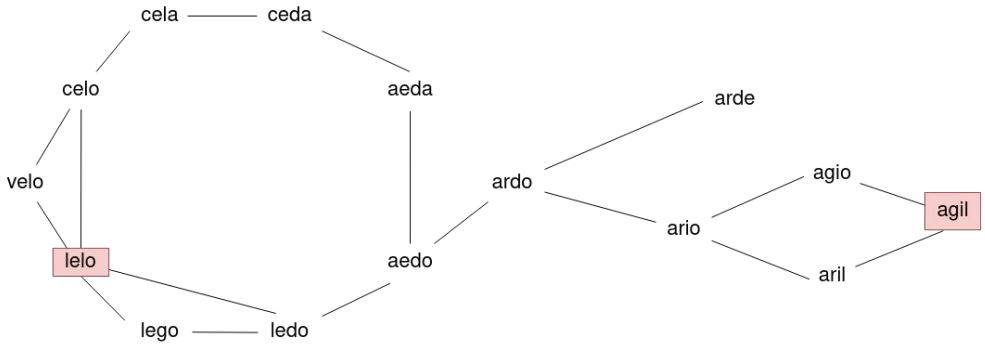

Paso 9
lelo
ledo 1
celo 1
velo 1
lego 1
aedo 2
cela 2
aeda 3
ardo 3
cela
aeda
ardo


cela
ceda
celo
aeda
velo
ardo
arde
agio
agil
ario
lelo
aedo
aril
lego
ledo

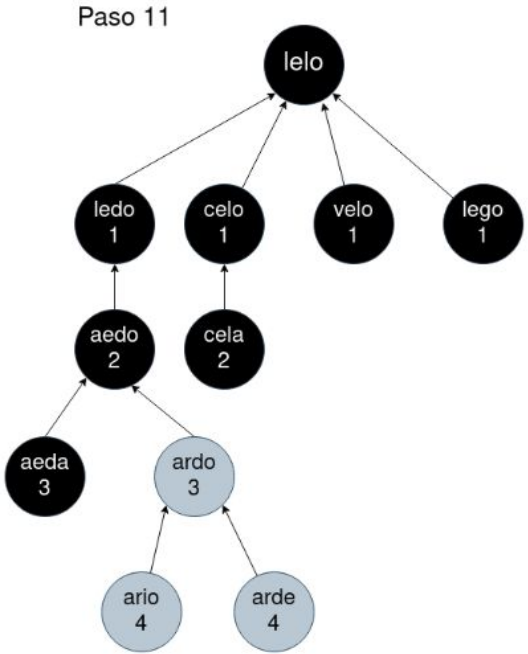
BEA



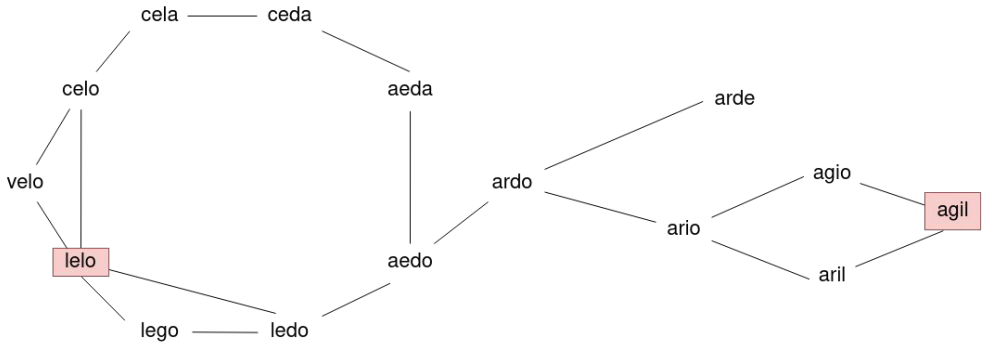
aeda	ardo



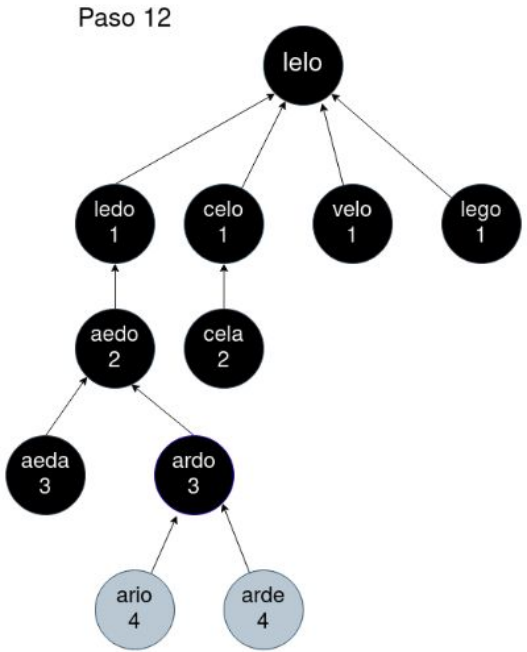
BEA



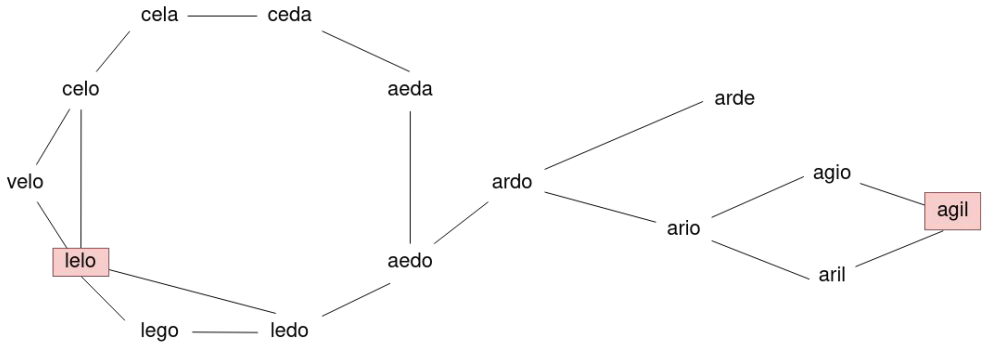
ardo	ario	arde



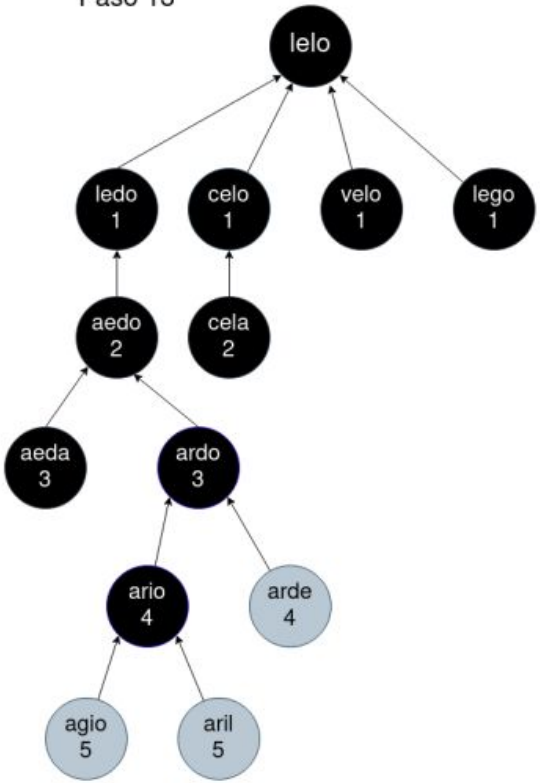
BEA

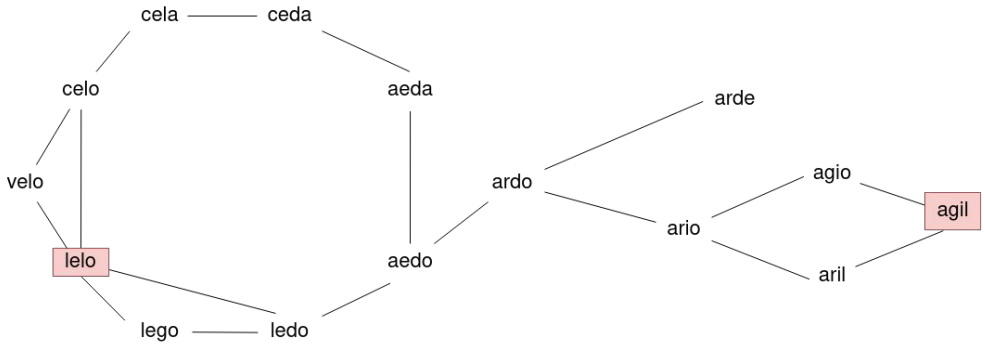


ario	arde



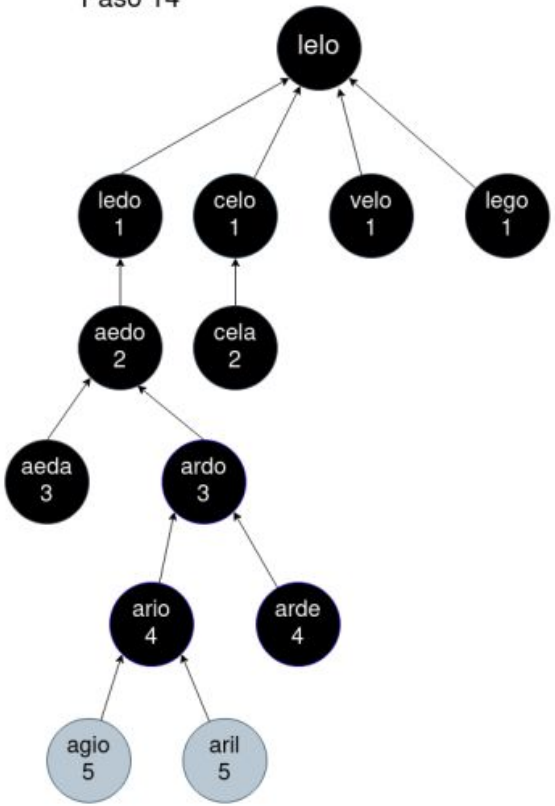
Paso 13

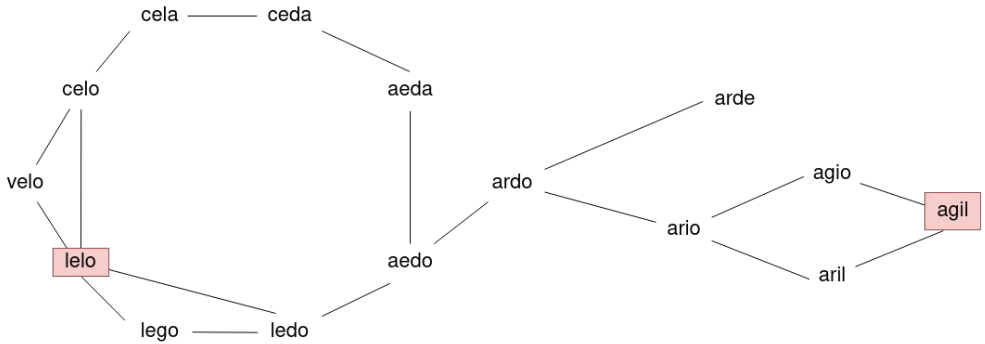




arde	agio	aril

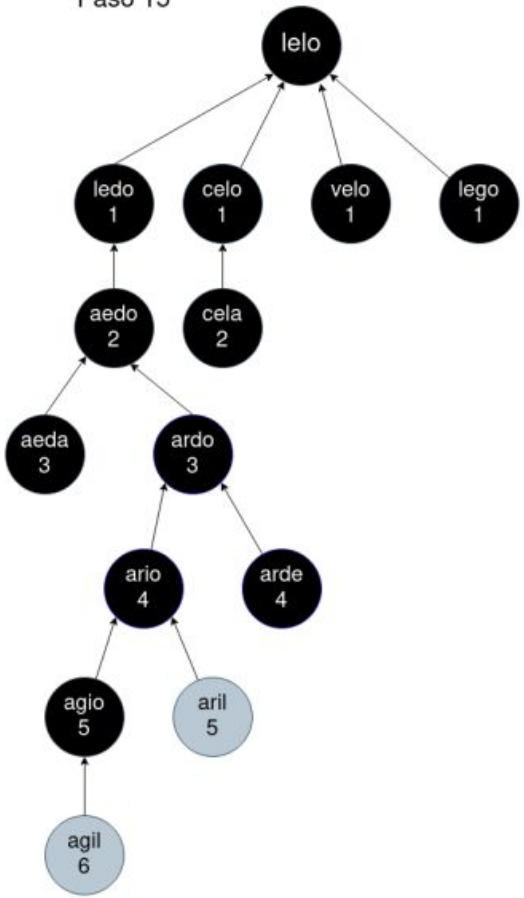
Paso 14



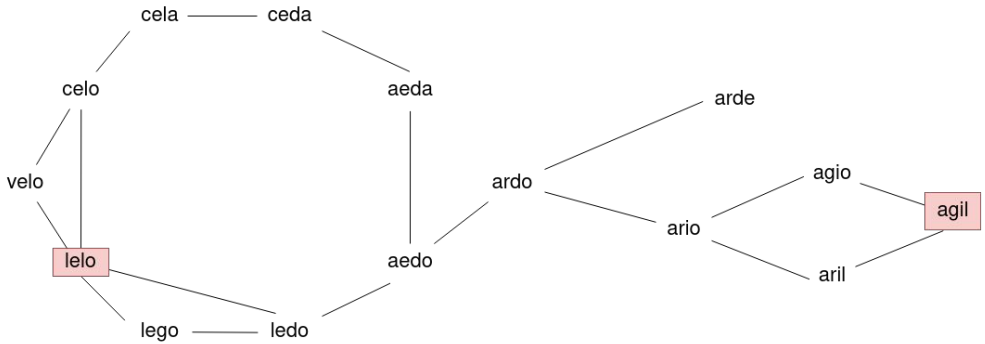


agio	aril

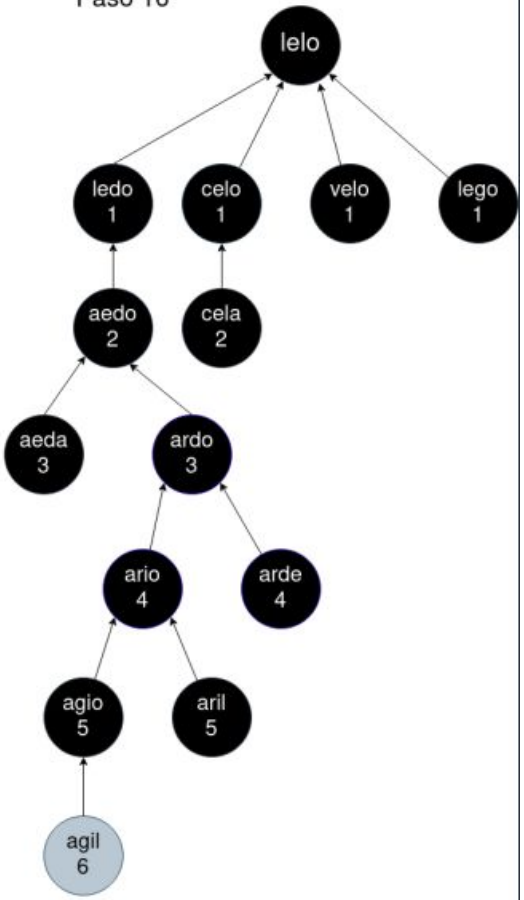
Paso 15



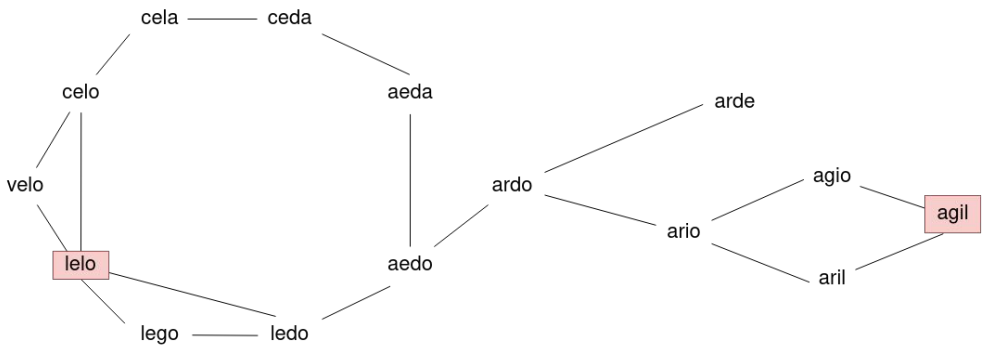
aril	agil



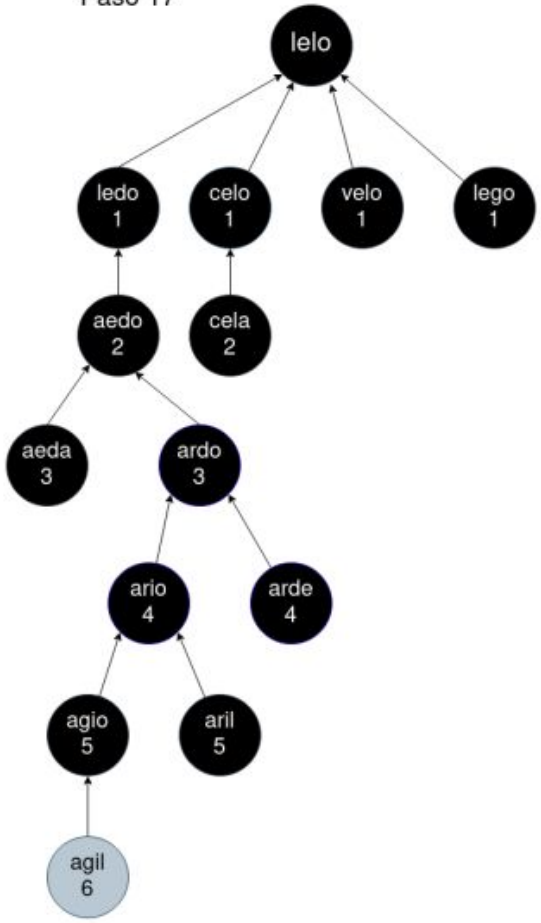
Paso 16


lelo
ledo 1
celo 1
velo 1
lego 1
aedo 2
cela 2
aeda 3
ardo 3
ario 4
arde 4
agio 5
aril 5
agil 6

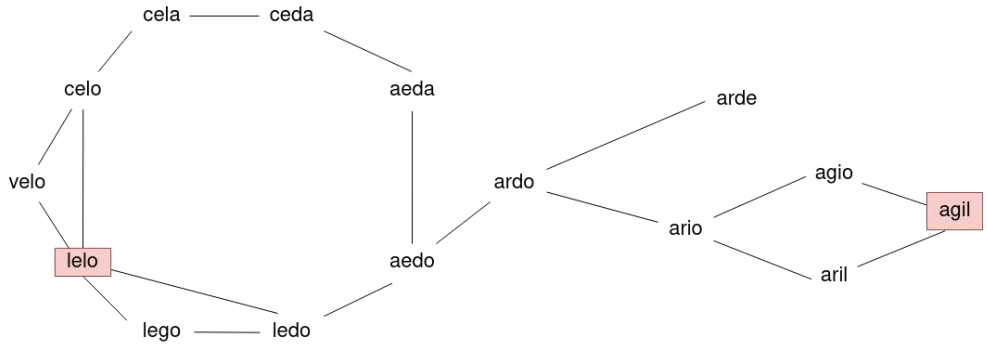
agil


cela
ceda
celo
aeda
velo
ardo
arde
agio
agil
ario
lelo
aedo
aril
lego
ledo

Paso 17


lelo
ledo 1
celo 1
velo 1
lego 1
aedo 2
cela 2
aeda 3
ardo 3
ario 4
arde 4
agio 5
aril 5
agil 6

cola


cela
ceda
celo
aeda
velo
ardo
arde
agio
agil
ario
lelo
aedo
aril
lego
ledo

Recorrer

La siguiente función muestra cómo seguir los enlaces del predecesor para imprimir la escalera de palabras.

```
void traversal(Vertex<string>* vertex){
   while(vertex->predecessor){
       cout << vertex->data << endl;
       vertex = vertex->predecessor;
   }
   cout << vertex->data << endl;
}
```

Implementación final

```
int main() {
    cout << "Comienza construcción del grafo" << endl;
    Graph g = buildGraph();
    cout << "Finaliza construcción del grafo" << endl;
    cout << "Realizando BEA desde vertice" << endl;
    bea(g.getVertex("lelo")); // palabra inicial
    cout << "Imprimiendo recorrido" << endl;
    traversal(g.getVertex("agil")); // palabra final
    return 0;
}
```

output:

```
Comienza construcción del grafo
Finaliza construcción del grafo

Realizando BEA desde vertice

Imprimiendo recorrido
agil
agio
ario
ardo
aedo
ledo
lelo
```

Gracias