SHEVEK

Zeste de savoir

Introduction à la visualisation de graphes avec VisNetwork

12 mai 2020

Table des matières

1. Introduction 3
- 1.1. Un graphe, kesako ? 3
- 1.2. Utilisation des graphes 4
- 1.3. Faire des graphes en javascript 4
- 1.4. VisNetwork 6

2. Création de notre premier graphe 10
- 2.1. Créer un page vide 10
- 2.2. Créer une instance de VisNetwork 11
- 2.3. Création de noeuds 12
- 2.4. Création de liens 13
- 2.5. Les graphes orientés 14
- 2.6. Exercice 15
- Contenu masqué 16

3. La personnalisation visuelle partie 1 20
- 3.1. Première personnalisation visuelle 20
- 3.2. Premières personnalisations visuel 24
- 3.3. Personnalisation des liens 25
- 3.4. Personnalisation avancée des noeuds 28
- 3.5. Exercice 30
- Contenu masqué 31

4. La personnalisation visuelle partie 2 37
- 4.1. Les groupes 37
- 4.2. Le design des groupes 42
- 4.3. Les images 45
- 4.4. Excercice 48
- 4.5. Défis 54
- 4.6. Récapitulatif des propriétés 54
- Contenu masqué 55

5. Les forces 60
- 5.1. Qu'est qu'une force 60
- 5.2. Les différents algoritmes de force 60
- 5.3. Paramétrer une force 64
- 5.4. Exemple avec barnesHut 65
- 5.5. Gestion de la masse des noeuds 72
- Contenu masqué 74

6. La gestion des évènements 78
6.1. La capture des évènements . . . 78
6.2. Clic et double-clic . . . 78
6.3. Zoom/Dézoom . . . 81
6.4. Exercice . . . 82
Contenu masqué . . . 84

7. Placements hiérarchiques et clusters 88
7.1. Les placements hiérarchiques . . . 88
7.2. Les clusters . . . 91
7.3. Exercice . . . 94
Contenu masqué . . . 100

Dans ce tutoriel, je vais vous présenter la bibliothèque JavaScript VisNetwork. Cette bibliothèque opensource permet de générer des graphes personnalisables et interactifs à partir de code JavaScript.

Prérequis

Un connaissance basique du JavaScript sera nécessaire pour suivre ce tuto, mais il est conçu pour être accessible aux débutants.

Objectifs

Le but est qu'à la fin de la lecture de ce tutoriel, vous soyez en mesure de maîtriser la création de graphe et que vous ayez découvert les principales fonctionnalités de VisNetwork.

Ce tutoriel est très axé sur la pratique, un petit exercice est proposé à la fin de chaque chapitre afin que vous preniez en main la bibliothèque.

1. Introduction

Nous allons commencer par un peu de théorie pour vous présenter tout d'abord ce qu'est un graphe, les différentes bibliothèques JavaScript permettant d'en créer et pourquoi j'ai choisi de vous présenter VisNetwork.

1.1. Un graphe, kesako ?

Les graphes dont les principes viennent de la théorie des graphes sont des formes de représentation graphique permettant de représenter des données liées entre elles, ils sont composés :

— De **noeuds** *(ou sommets)* représentant une donnée.
— De **liens** *(ou arrêtes)* connectant les données entre elles.

Dans VisNetwork les noeuds sont appelés **nodes** et les liens sont appelés **edges**.

Dans les graphes la position des noeuds n'a aucune importance seuls les liens qu'ils ont entre eux compte.

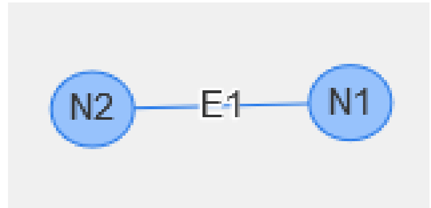


Figure 1.1. – Voici un graphe basique reliant 2 noeuds

Un graphe peut représenter un tas de concepts différents par exemple des relations sociales entre des personnes, un réseaux d'ordinateurs ou encore des liaisons entre des villes, ...

Une notion qui est importante est l'orientation du graphe, c'est à dire que les liens entre nos nœuds peuvent ou pas posséder un sens de parcours.

Dans ce tutoriel je vais parler de graphes pour parler de représentation de données en réseaux en référence à la théorie des graphes, il ne faut pas les confondre avec les représentation graphique de données qu'on appelle communément des graphes et qui couvrent tout un tas de représentations différentes (diagramme, Pareto, ...)

Si vous souhaitez découvrir plus en détail la théorie des graphes, je vous propose ce tutoriel très bien écrit disponible sur Zeste de Savoir .

1.2. Utilisation des graphes

Les graphes peuvent être utilisés pour visualiser toutes sortes de données, mais il sont véritablement utiles lorsque l'on veut mettre en évidence les liens qui relient ces données.

Le logiciel de création de graphe le plus connu et le plus utilisé est gephi . Il est particulièrement utilisé dans les divers domaines de la recherche scientifique.
Son principal inconvénient et d'être assez lourd à utiliser et donc pas adapté pour créer de simples visualisations avec lesquelles des visiteurs pourront interagir.

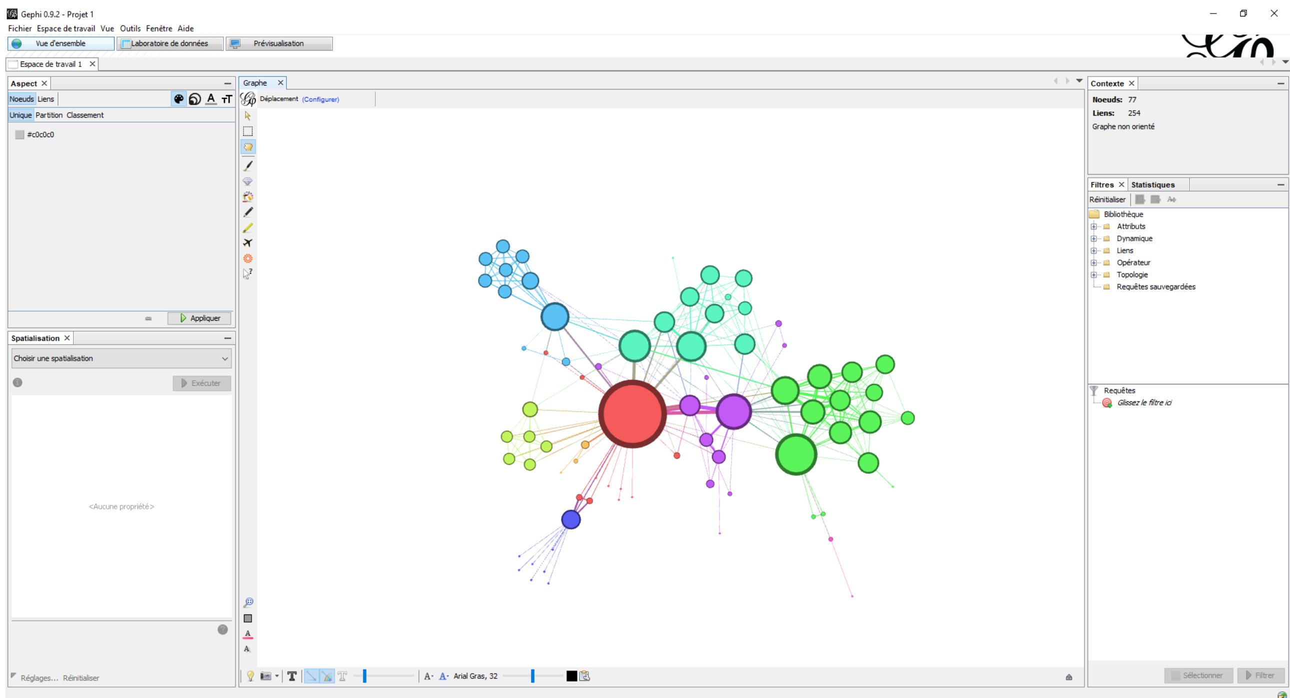


FIGURE 1.2. – Capture d'écran de Gephi

1.3. Faire des graphes en javascript

Il existe plusieurs bibliothèques javascript permettant de créer des graphes, les deux plus connues et utilisées sont D3.js et sigmaJs .

D3.js est sans doute la plus importante bibliothèque spécialisée dans la visualisation graphique de données. Elle permet de créer beaucoup de représentations graphiques différentes, notamment des graphes mais aussi des diagrammes en bâtons, circulaires, courbes et pleins d'autres représentations. Étant plus diversifié ses fonctionnalités liées aux graphes sont plus limitées que celle des bibliothèques spécialisées.

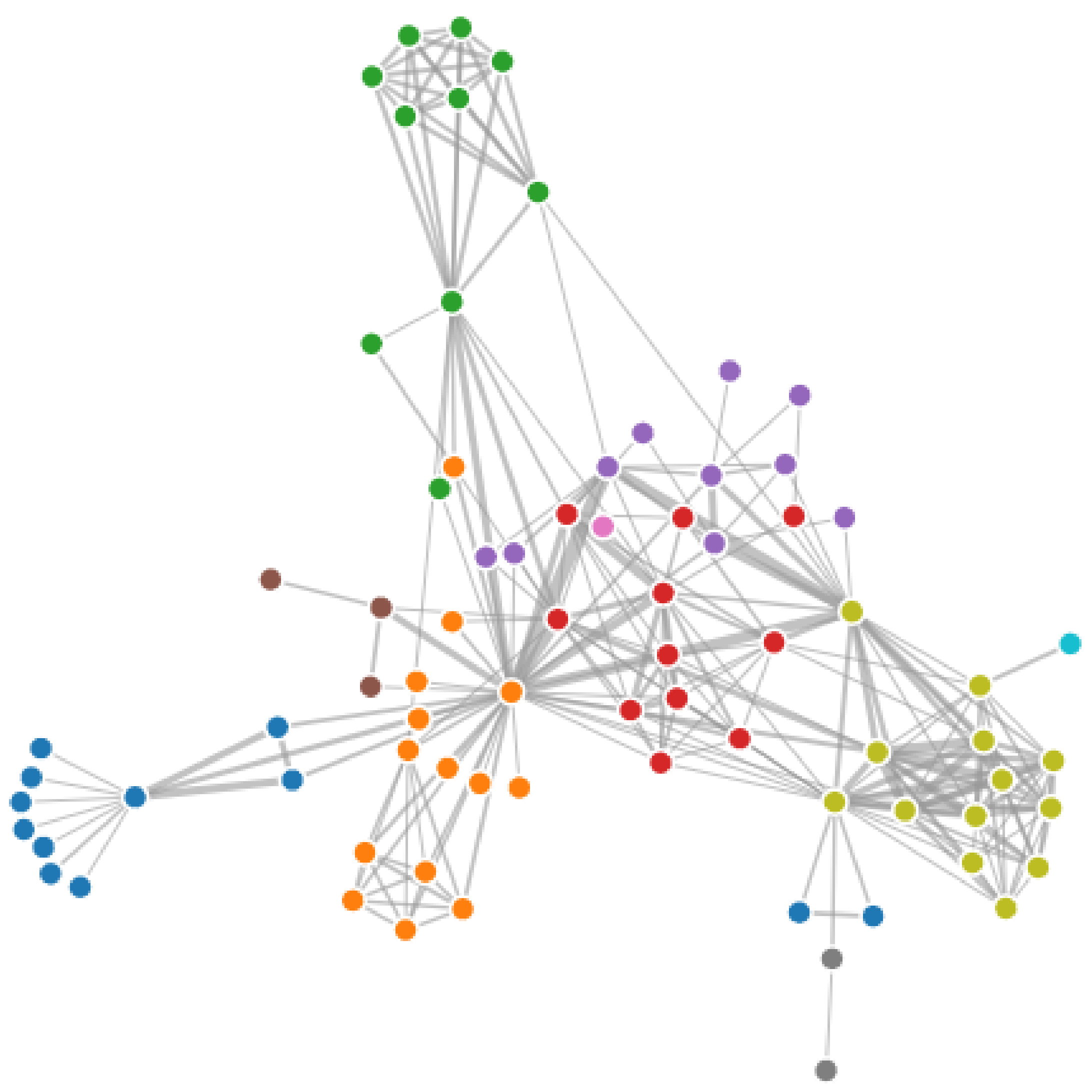


Figure 1.3. – Graphe d'exemple de D3.js

SigmaJs est une bibliothèque spécialisée dans l'affichage des graphes, elle a pour avantage d'être simple d'utilisation et d'avoir beaucoup de modules permettant de personnaliser nos représentations. Son développement est malheureusement en standby.

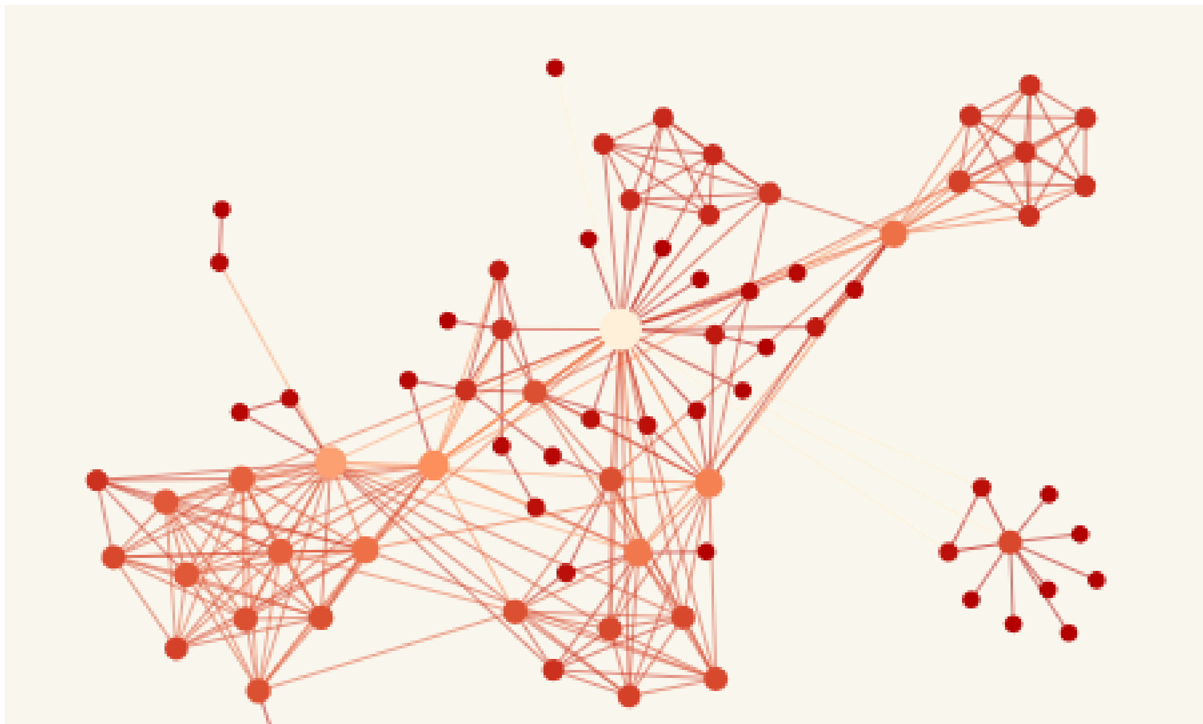


Figure 1.4. – Graphe d'exemple de SigmaJS

Dans ce tutoriel, je vais vous présenter **VisNetwork**, qui fait partie de la bibliothèque Vis.js
Elle a pour avantage d'être relativement simple à utiliser et d'avoir une quantité d'options de personnalisations très impressionnantes, elle est pour moi la meilleure bibliothèque pour créer des graphes, c'est pour cela que j'ai choisi de vous la présenter.

1.4. VisNetwork

La bibliotèque Vis.js est composée de 5 modules :

— `Network` : Permet la représentation de données sous forme de réseaux, c'est ce module que nous allons explorer, ayez à l'esprit que lorsque je vais parler de graphes je ferais référence au module network.
— `DataSet` : Permet l'organisation des données, la module network utilise les dataset pour gérer les tableaux de données de noeuds et des liens.
— `Timeline` : Permet de représenter des frises chronologiques et des évènements dans le temps (frises historiques, calendrier, ...).
— `Graph2d` : Permet la création de représentations graphiques de données en 2 dimensions (diagrammes).
— `Graph3d` : Permet la création de représentations graphiques de données en 3 dimensions (diagrammes).

Dans ce tutoriel nous n'utiliserons pas les trois derniers modules.

Voici quelques liens qui vous serons utiles :

— Le site officiel
— La documentation

— Une grande liste d'exemples d'utilisations
— Le code source sur GitHub

Pour télécharger VisNetwork, allez sur la catégorie téléchargement du site officiel et téléchargez le fichier `vis.min.js`, il suffira alors d'inclure ce fichier dans notre code pour utiliser les fonctionnalités de la bibliothèque.

Vous pouvez bien sûr aussi utiliser nmp si vous le préférez.

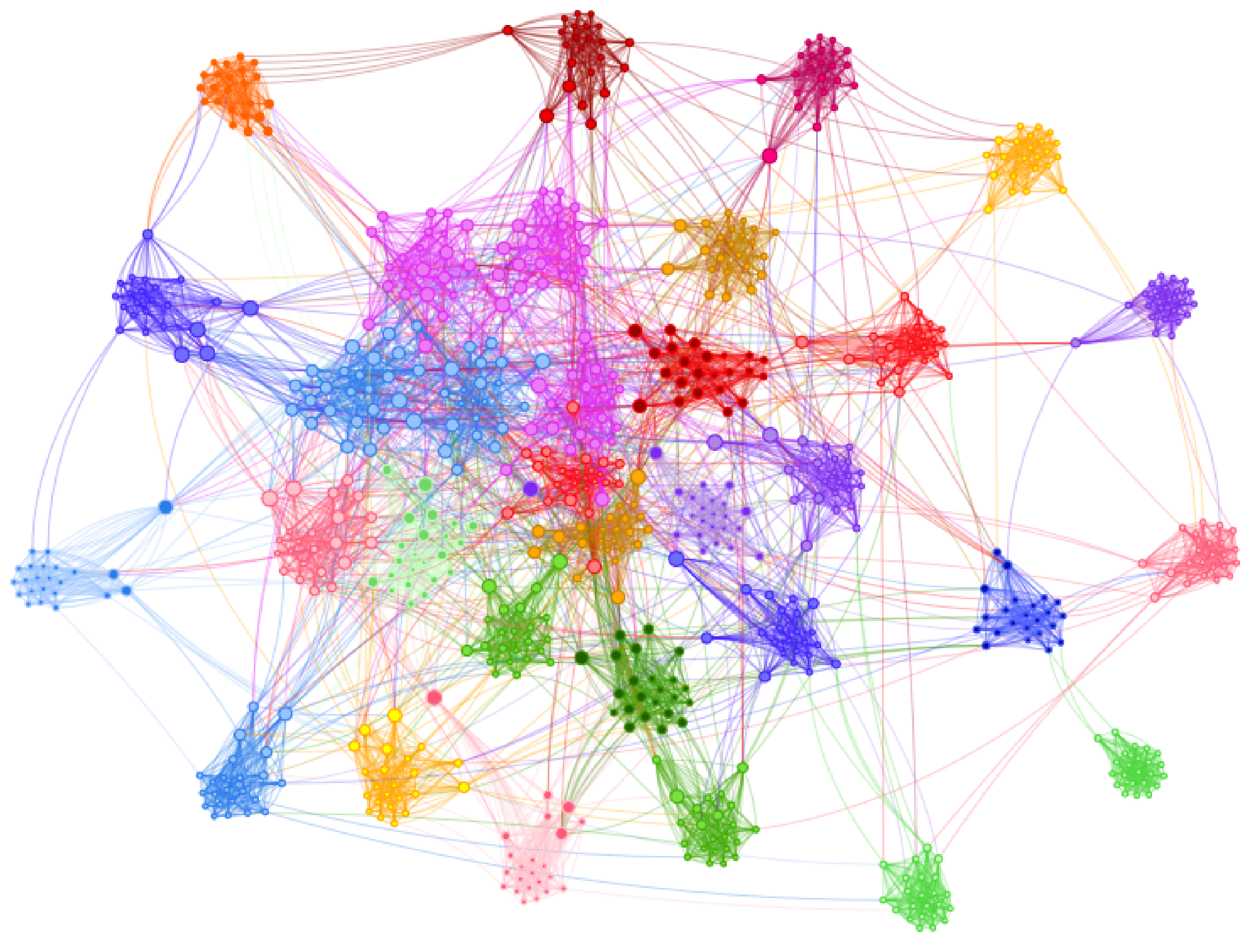


FIGURE 1.5. – Exemple de graphe avec beaucoup de noeuds

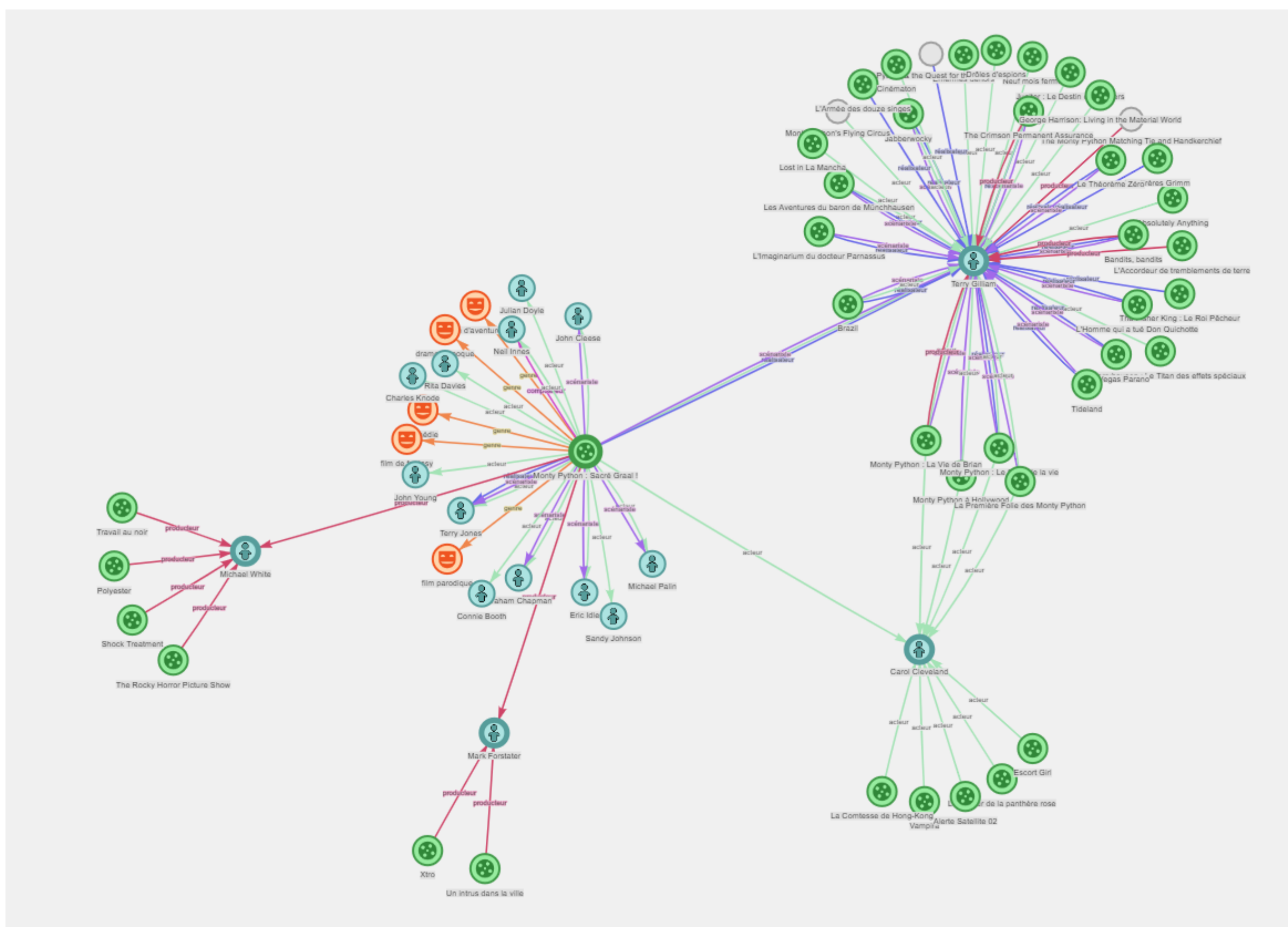


Figure 1.6. – Graphe représentant les liens entre des films et des personnes (données issues de Wikidata)

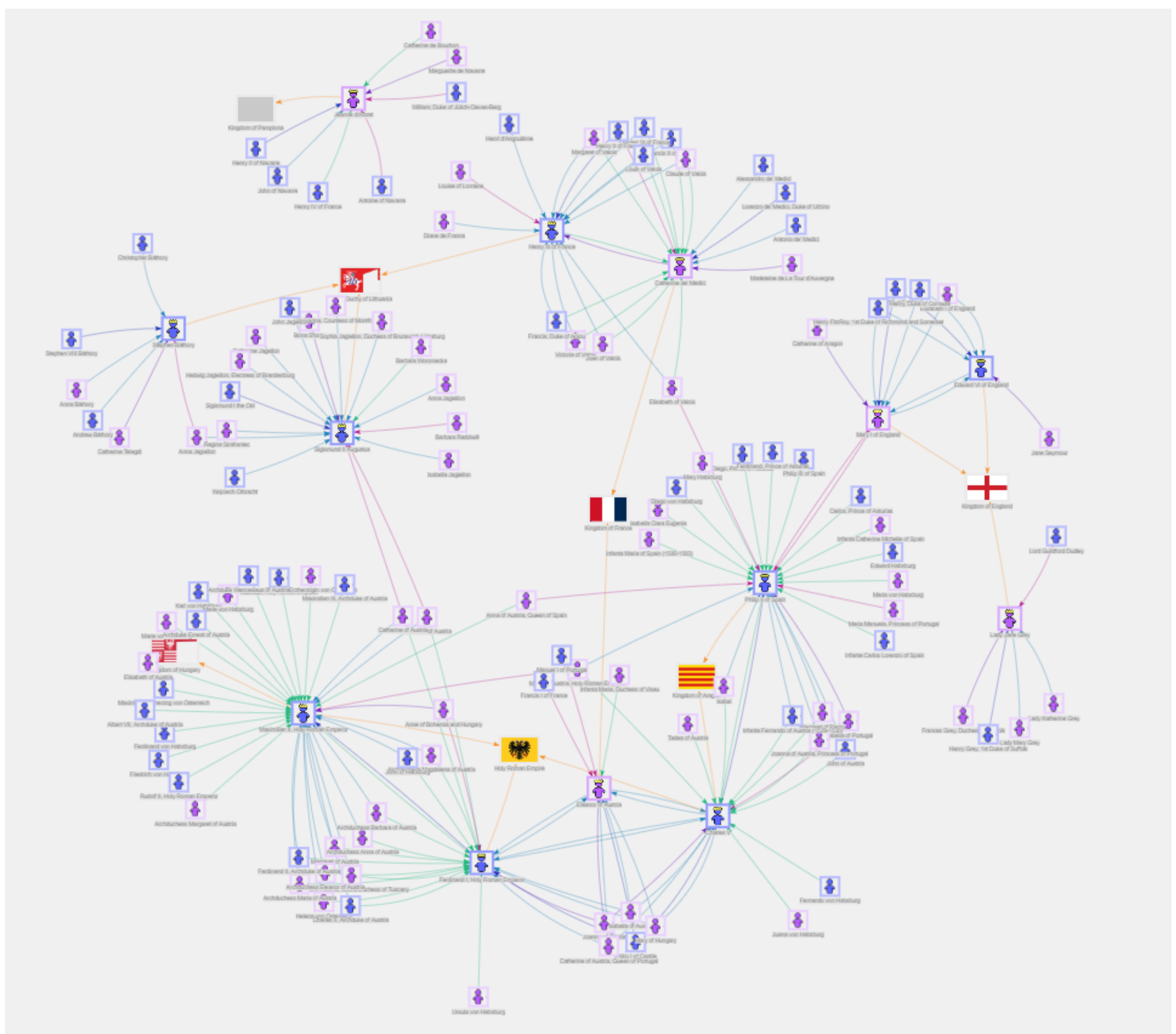


Figure 1.7. – Graphe présentant des liens familiaux entre dynasties royales (données issues de Wikidata)

Ce premier chapitre de théorie est terminé, nous allons pouvoir passer à la partie pratique où je vais vous présenter les différentes fonctionnalités de VisNetwork via des exemples et des exercices.

2. Création de notre premier graphe

Dans ce premier chapitre pratique, je vais vous présenter par étape la création d'un premier graphe.

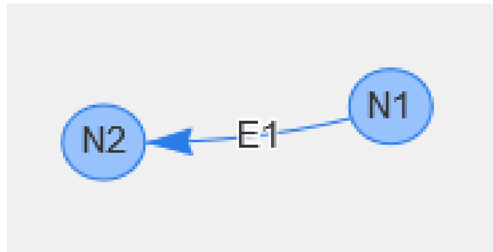


FIGURE 2.1. – Voici le joli petit graphe que nous obtiendrons à la fin de ce chapitre

2.1. Créer un page vide

La première chose à faire est tout d'abord de créer une page vide dans laquelle nous allons importer la bibliothèque VisNetwork. Pour cela créez un nouveau fichier HTML et copier-coller le code ci dessous :

```
<!doctype html>
<html>
  <head>
    <title>Tuto VisNetwork</title>

        <style>
                #graphe
                {
                  display: inline-block;
                  width: 800px;
                  height: 600px;

                  background-color: #f0f0f0;
                }
        </style>

          <script type="text/javascript" src="vis.min.js"></script>
  </head>
  <body>

```

```
21      <div id="graphe"></div>
22
23      <script type="text/javascript">
24
25                  // Notre code sera placé ici
26
27      </script>
28
29    </body>
30  </html>
```

La div html `graphe` permet de contenir notre graphe, sa taille est définie dans le css en haut du fichier.

N'oubliez pas d'inclure le fichier `vis.min.js` et placez le dans le même dossier que votre fichier html.

Si vous ouvrez la page que vous venez de créer avec un navigateur internet, vous devriez apercevoir un rectangle gris qui correspond à la zone dans laquelle nous allons afficher notre graphe.

2.2. Créer une instance de VisNetwork

La seconde chose à faire est de créer une instance de Vis.Network

```
1  let nodes = new vis.DataSet([]);
2  let edges = new vis.DataSet([]);
3
4  let container = document.getElementById('graphe');
5
6  let data = {
7          nodes: nodes,
8          edges: edges
9  };
10 let options = {};
11
12 let network = new vis.Network(container, data, options);
```

Notre instance est composé de :

— `data` : qui contient les données de notre graphe, elle est composé de deux *vis.DataSet* correspondant à nos noeuds et à nos liens que nous allons voir dans la suite de ce chapitre.
— `options` : qui nous permettra de personnaliser notre graphe, nous verrons cela dans le prochain chapitre.

Notre instance `network` va nous permettre d'afficher le graphe et de gérer les actions sur le graphe, nous verrons cela dans le suite du tutoriel.

Dans mes exemples de code j'utilise le mot clé `let` pour déclarer les variables, mais il est possible d'utiliser `var` ou `const` à la place si vous le préférez.

2.3. Création de noeuds

Nous allons maintenant ajouter notre premier noeud, pour cela placer le code ci-dessous après le code précédent.

```
nodes.add(
        [
                {
                        id: 1,
                        label: "N1"
                }
        ]
);
```

Nous ajoutons à notre *Dataset*, via la méthode `add` :

- Un tableau `[]` contenant :
  - Un objet nœud `{}` contenant :
    - `id` : Un identifiant qui doit être unique, ce qui permettra d'identifier le nœud au niveau du code.
    - `label` : Un label qui correspond au texte qui sera affiché sur le nœud, ce qui permet à l'utilisateur d'identifier le nœud.

En rechargeant dans votre page HTML vous devriez obtenir un nœud comme ceci :

FIGURE 2.2. – Notre premier noeud

Vous pouvez déplacer votre nœud en maintenant le clic gauche de la souris enfoncé sur le nœud, si vous faite la même chose à un endroit où il n'y a pas de nœud vous pouvez déplacer la caméra, vous pouvez également effectuer un zoom/dé-zoom avec la molette de la souris.

2.4. Création de liens

Il ne nous reste plus qu'a ajouter un second noeud, puis à créer notre premier lien qui reliera nos deux noeuds et nous obtenons alors notre premier graphe! 😊

```
// Ajout de nos deux noeuds
nodes.add(
        [
                {
                        id: 1,
                        label: "N1"
                },
                {
                        id: 2,
                        label: "N2"
                }
        ]
);

// Ajout du lien entre nos deux noeuds
edges.add(
        [
                {
                        from: 1,
                        to: 2,
                        label: "E1",
                }
        ]
);
```

Les données de notre lien sont composées de:

— `from` : correspond à l'identifiant du premier noeud relié.
— `to` : correspond à l'identifiant du second noeud relié.
— `label` : correspond au texte qui sera affiché sur le lien.

En rechargeant vous page HTML vous devriez obtenir votre premier graphe comme ceci :

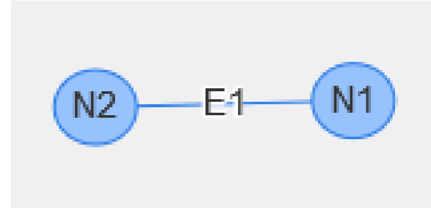


FIGURE 2.3. – Notre premier lien

Si vous déplacez un noeud vous remarquerez que le second est également déplacé, cela est dû à force d'attraction du graphe, nous verrons dans une partie ultérieure comment la personnaliser.

Récapitulatif complet du code :

Contenu masqué n°1

2.5. Les graphes orientés

Un graphe peut être orienté ou non, lorsqu'un graphe est orienté les liens qui le compose sont dirigé c'est-à-dire qu'un vont d'un noeuds vers un autre.

Reprenons l'exemple précédent, en ajoutant une direction à notre relation.

```
// Ajout de nos deux noeuds
nodes.add(
        [
                {
                        id: 1,
                        label: "N1"
                },
                {
                        id: 2,
                        label: "N2"
                }
        ]
);

// Ajout d'un lien entre nos deux noeuds orientés du noeuds N1 vers
    le noeuds N2
edges.add(
        [
                {
                        from: 1, // Source : noeud "N1"
                        to: 2,   // Destination : noeud "N2"
                        label: "E1",
                        arrows: "to"
                }
        ]
);
```

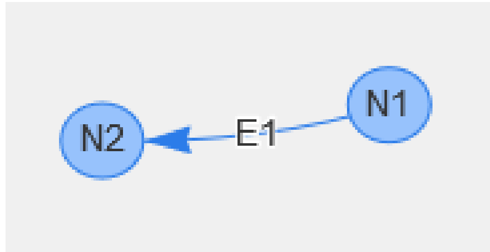


Figure 2.4. – Notre lien est maintenant représenté sous la forme d'une flèche

2.6. Exercice

Pour vous exercer, je vous propose un exercice assez simple, le but est d'obtenir le graphe ci-dessous :

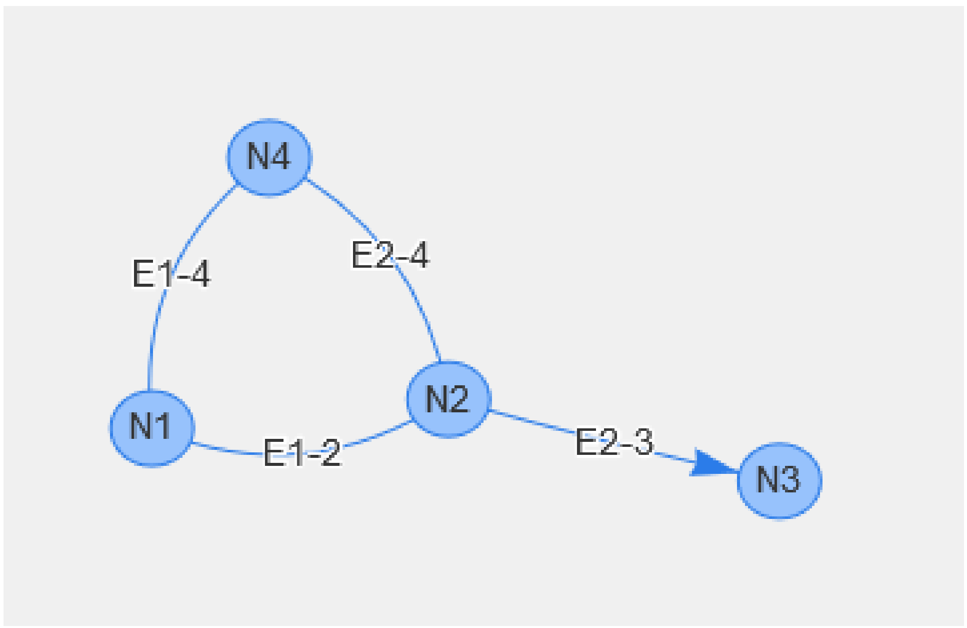


Figure 2.5. – Le résultat à obtenir

Si vous êtes bloqué, vous pouvez retrouver la correction ci-dessous.

Contenu masqué n°2

Avec ce premier chapitre pratique, nous avons vu comment créer notre premier graphe, j'espère qu'il vous a plu, dans le prochain chapitre nous allons découvrir comment personnaliser notre graphe.

Contenu masqué

Contenu masqué n°1

```
<!doctype html>
<html>
  <head>
    <title>Tuto VisNetwork</title>

        <style>
                #graphe
                {
                  display: inline-block;
                  width: 800px;
                  height: 600px;

                  background-color: #f0f0f0;
                }
        </style>

          <script type="text/javascript" src="vis.min.js"></script>
  </head>
  <body>

    <div id="graphe"></div>

    <script type="text/javascript">


                let nodes = new vis.DataSet([]);
                let edges = new vis.DataSet([]);

                let container = document.getElementById('graphe');

                let data = {
                  nodes: nodes,
                  edges: edges
                };
                let options = {};

                // Initialisation de l'instance de VisNetwork
                let network = new vis.Network(container, data,
                    options);

                // Ajout de nos deux noeuds
                nodes.add(
                        [
                                {
```

```
                                        id: 1,
                                        label: "N1"
                                },
                                {
                                        id: 2,
                                        label: "N2"
                                }
                        ]
                );

                // Ajout du lien entre nos deux noeuds
                edges.add(
                        [
                                {
                                        from: 1,
                                        to: 2,
                                        label: "E1",
                                }
                        ]
                );
    </script>

  </body>
</html>
```

Retourner au texte.

Contenu masqué n°2

```
<!doctype html>
<html>
  <head>
    <title>Tuto VisNetwork</title>

        <style>
                #graphe
                {
                  display: inline-block;
                  width: 800px;
                  height: 600px;

                  background-color: #f0f0f0;
                }
        </style>

          <script type="text/javascript" src="vis.min.js"></script>
```

```
</head>
<body>

  <div id="graphe"></div>

  <script type="text/javascript">


              let nodes = new vis.DataSet([]);
              let edges = new vis.DataSet([]);

              let container = document.getElementById('graphe');

              let data = {
                nodes: nodes,
                edges: edges
              };
              let options = {};

              let network = new vis.Network(container, data,
                  options);

              nodes.add(
                      [
                              {
                                      id: 1,
                                      label: "N1"
                              },
                              {
                                      id: 2,
                                      label: "N2"
                              },
                              {
                                      id: 3,
                                      label: "N3"
                              },
                              {
                                      id: 4,
                                      label: "N4"
                              }
                      ]
              );


              edges.add(
                      [
                              {
                                from: 1,
                                to: 2,
                                label: "E1-2"
```

```
                                },
                                {
                                  from: 2,
                                  to: 3,
                                  label: "E2-3",
                                  arrows: "to"
                                },
                                {
                                  from: 1,
                                  to: 4,
                                  label: "E1-4"
                                },
                                {
                                  from: 4,
                                  to: 2,
                                  label: "E2-4"
                                }
                        ]
                );

    </script>

  </body>
</html>
```

Retourner au texte.

3. La personnalisation visuelle partie 1

Dans ce troisième chapitre, je vais vous présenter les options de personnalisation permettant de changer la couleur et la taille des noeuds, l'affichage des liens et plein d'autres paramètres pour créer de jolis graphes.

Dans ce chapitre notre but va être d'obtenir ce diagramme de flux :

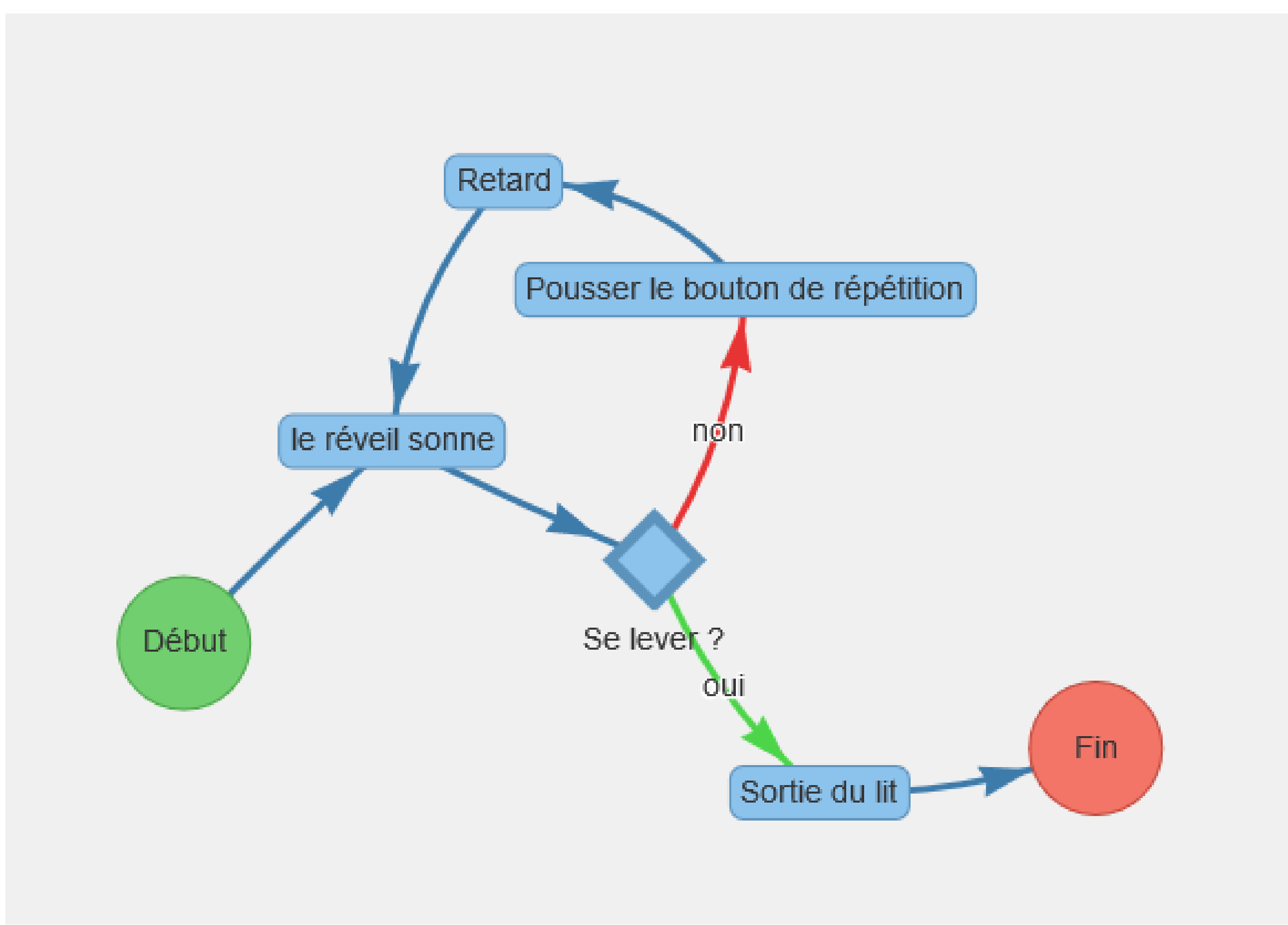


FIGURE 3.1. – Si vous êtes un grand dormeur comme moi, la situation représentée risque de vous être familière

3.1. Première personnalisation visuelle

Avant de commencer nous allons séparer le code JavaScript et HTML en deux fichiers afin de le rendre plus clair, et nous allons ajouter à notre code JavaScript les données de base du graphe présenté au-dessus:

3. La personnalisation visuelle partie 1

Notre code HTML:

```
<!doctype html>
<html>
  <head>
    <title>Tuto VisNetwork</title>

    <style>
        #graphe
        {
          display: inline-block;
          width: 800px;
          height: 600px;

          background-color: #f0f0f0;
        }
    </style>

    <script type="text/javascript" src="vis.min.js"></script>
  </head>
  <body>

    <div id="graphe"></div>

    <script type="text/javascript" src="graph.js"></script>

  </body>
</html>
```

Notre code JavaScript (graph.js) :

```
let nodes = new vis.DataSet([]);
let edges = new vis.DataSet([]);

let container = document.getElementById('graphe');

let data = {
  nodes: nodes,
  edges: edges
};
let options = {
    nodes : {},
    edges : {}
};

// Initialisation de l'instance de VisNetwork
let network = new vis.Network(container, data, options);
```

```

// Ajout des noeuds
nodes.add(
    [
        {
            id: "DEBUT",
            label: "Début"
        },
        {
            id: "REVEIL",
            label: "le réveil sonne"
        },
        {
            id: "LEVER",
            label: "Se lever ?"
        },
        {
            id: "SORTIE_LIT",
            label: "Sortie du lit"
        },
        {
            id: "FIN",
            label: "Fin"
        },
        {
            id: "POUSSER_BOUTON",
            label: "Pousser le bouton de répétition"
        },
        {
            id: "RETARD",
            label: "Retard"
        }
    ]
);

// Ajout des liens
edges.add(
    [
        {
            from: "DEBUT",
            to: "REVEIL",
            label: ""
        },
        {
            from: "REVEIL",
            to: "LEVER",
            label: ""
        },
        {
            from: "LEVER",
```

```
67                 to: "SORTIE_LIT",
68                 label: "oui"
69             },
70             {
71                 from: "SORTIE_LIT",
72                 to: "FIN",
73                 label: ""
74             },
75             {
76                 from: "LEVER",
77                 to: "POUSSER_BOUTON",
78                 label: "non"
79             },
80             {
81                 from: "POUSSER_BOUTON",
82                 to: "RETARD",
83                 label: ""
84             },
85             {
86                 from: "RETARD",
87                 to: "REVEIL",
88                 label: ""
89             }
90         ]
91 );
```

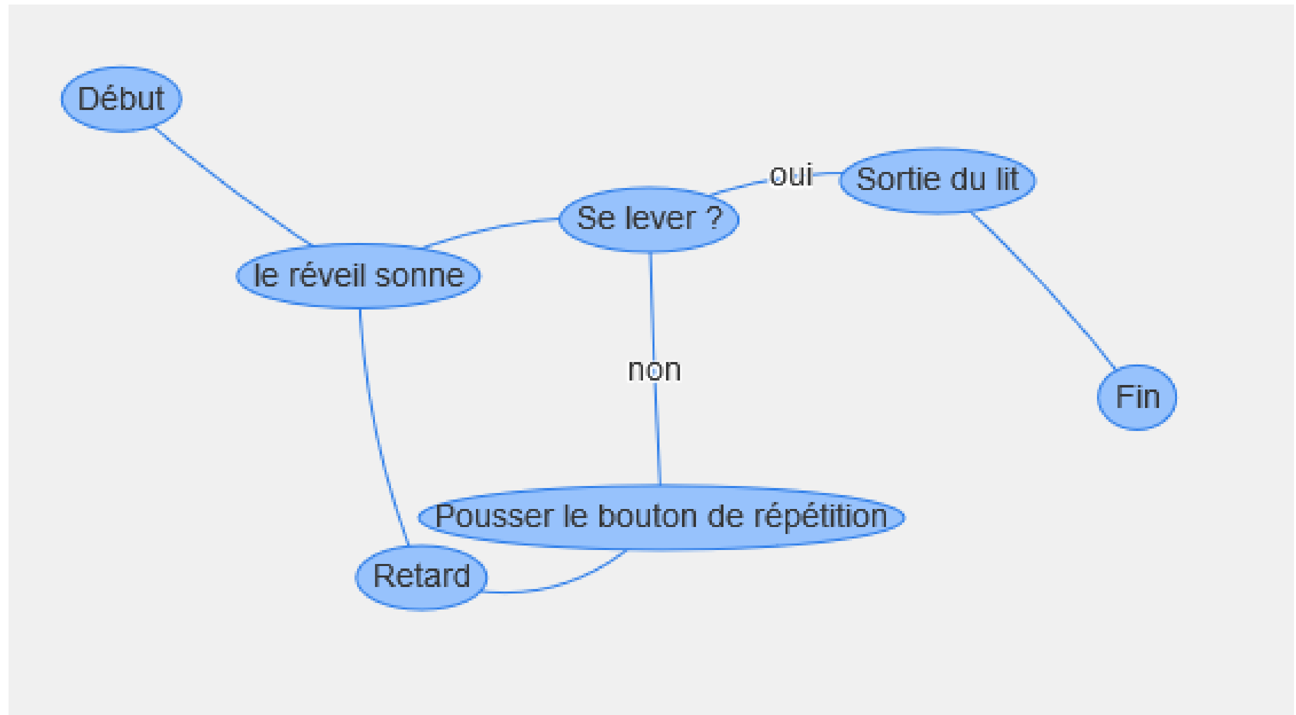


FIGURE 3.2. – Vous devriez obtenir ceci

Vous avez sans doute remarqué que pour les identifiants des noeuds ont été définis sous forme de texte, cela nous permet de créer plus simplement nos liens qu'avec des identifiants numériques.

3.2. Premières personnalisations visuel

Le paramétrage des noeuds et des liens peut se faire de deux façons:
- Soit par les options du graphe qui permettent de définir les styles et paramètres par défaut.
- Soit directement lors de l'ajout d'un noeud et d'un lien, ce qui permet de créer des éléments avec un design unique.

Nous allons commencer par définir les options de notre graphe.

```
let options = {
    nodes : {
        shape: "box",      // Nœud affiché sous forme de boîte
            entourant le label
        color: "#8cc3ed", // Nœud de couleur Bleue
    }
}
```

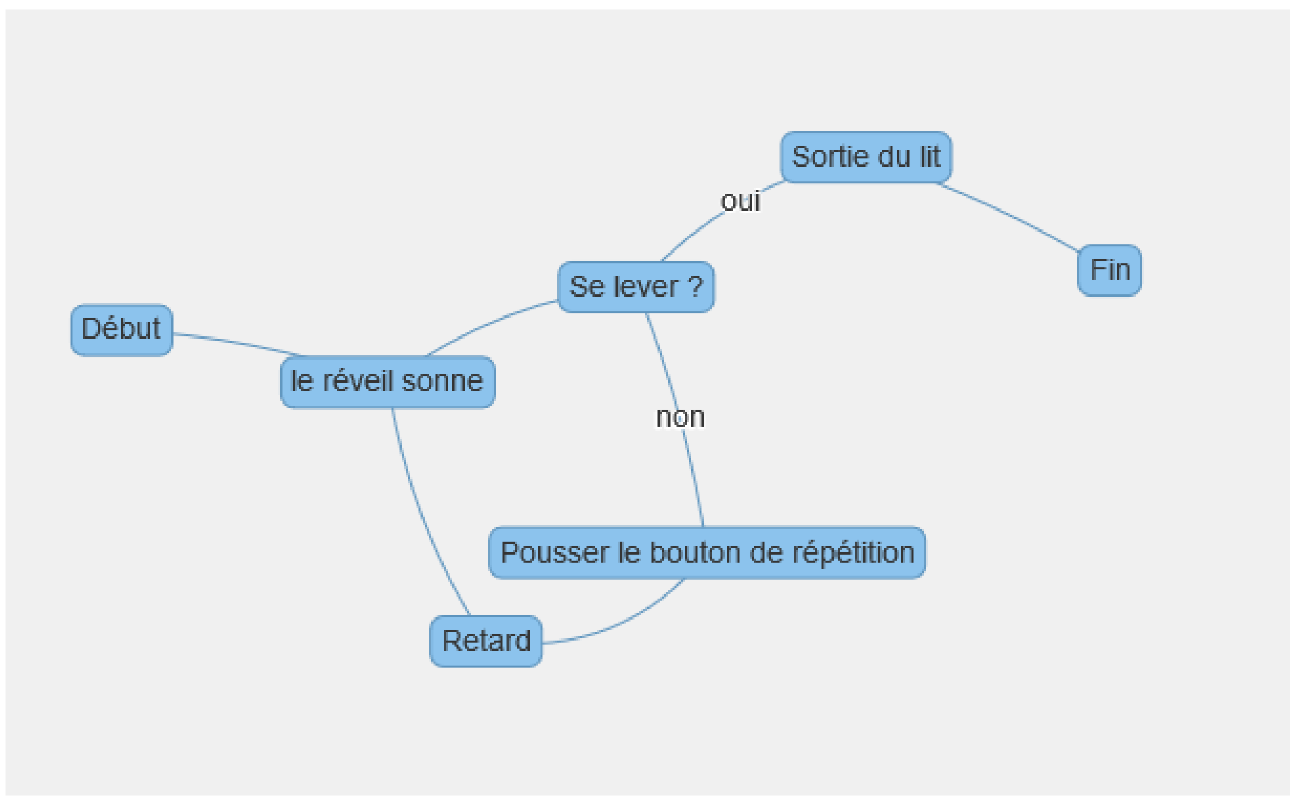


FIGURE 3.3. – Ça ne fait pas un gros changement mais vous voyez que c'est déjà un peu mieux

On a défini dans cet exemple nos deux premières propriétés :

- **`color`** : Choix de la couleur du noeud.
- **`shape`**: Défini la forme d'affichage du noeud.
  - `image`, `circularImage` : Affichage d'images *(nous verrons leur fonctionnement dans le chapitre suivant).*
  - `circle` : Cercle avec le label du nœud au centre.
  - `ellipse` : Ellipse entourant le label du nœud.
  - `box` : Rectangle/boîte entourant le label du nœud.
  - `diamond`, `dot`, `star`, `triangle`, `triangleDown`, `hexagon` et `square` : Symboleavec le label affiché en dessous.

3.3. Personnalisation des liens

Pour créer nos liens on va définir un design général dans les options et un design particulier pour pouvoir définir les couleurs des liens `oui` et `non`.

Pour cela on définit les options de design par défaut des liens :

```
1 let options = {
2   nodes : {
3     shape: "box",           // Nœud affiché sous forme de boite
        entourant le label
```

```
    color: "#8cc3ed",       // Nœud de couleur Bleue
  },
  edges : {
    arrows : "to",          // Flèche dirigée du premier vers le
        second noeud
    color : {
      color: "#3d7bab"      // Lien de couleur Bleue Foncé
    },
    width : 3               // Épaisseur du lien de 3
  }
}
```

Pour définir la couleur de nos liens `oui` et `non` on modifie le code de création des liens :

```
edges.add(
 [
   {
     from: "DEBUT",
     to: "REVEIL",
     label: ""
   },
   {
     from: "REVEIL",
     to: "LEVER",
     label: ""
   },
   {
     from: "LEVER",
     to: "SORTIE_LIT",
     label: "oui",
     color: { color : "#4dd54a" } // Couleur Verte
   },
   {
     from: "SORTIE_LIT",
     to: "FIN",
     label: ""
   },
   {
     from: "LEVER",
     to: "POUSSER_BOUTON",
     label: "non",
     color: { color : "#e73333" } // Couleur Rouge
   },
   {
     from: "POUSSER_BOUTON",
     to: "RETARD",
     label: ""
   },
```

```
35    {
36      from: "RETARD",
37      to: "REVEIL",
38      label: ""
39    }
40  ]
41 );
```

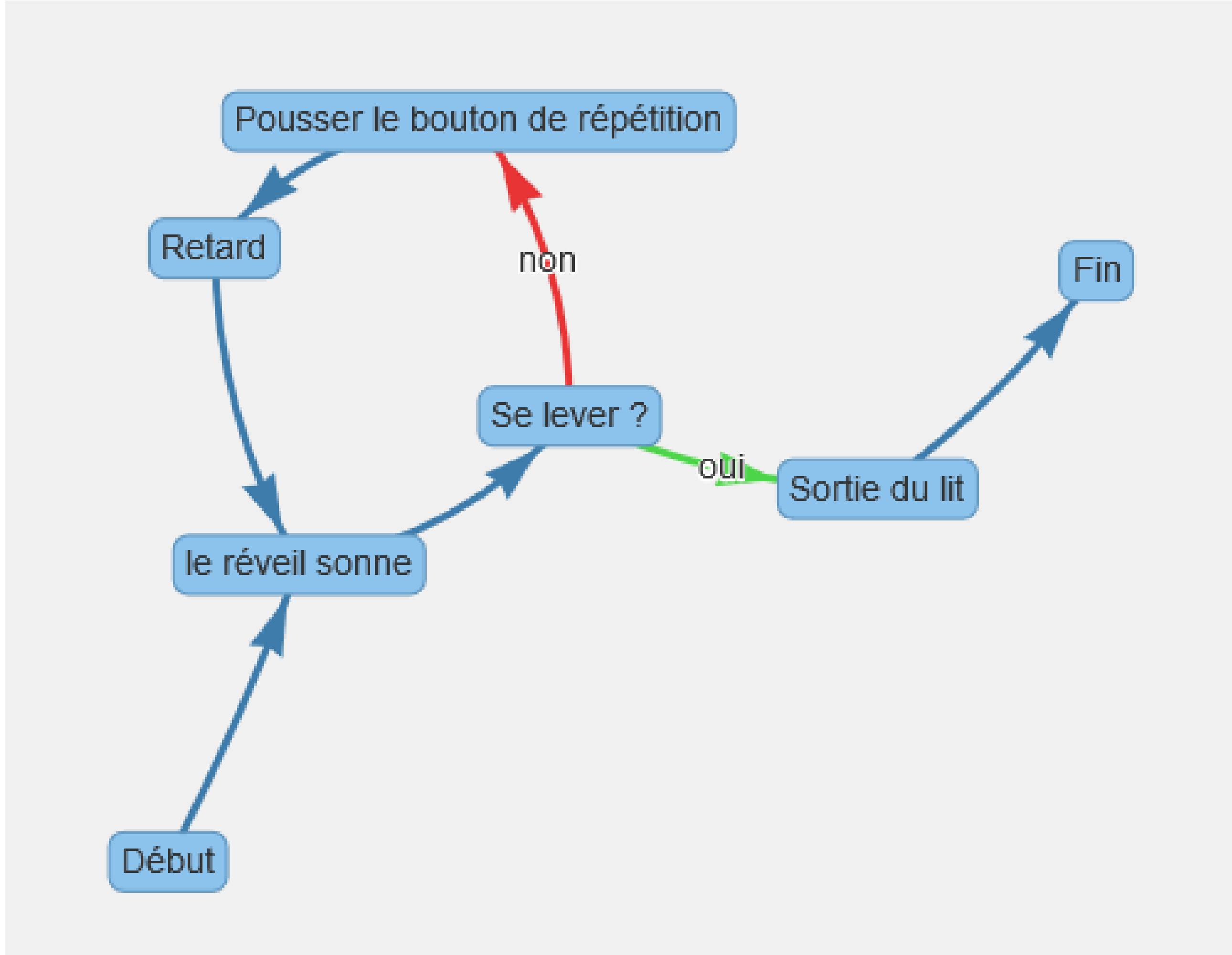


FIGURE 3.4. – Nous obtenons des liens beaucoup plus jolis

Voici une petite description des paramètres utilisés :

- **`arrows`** : Types de représentation du lien, `to` pour avoir une flèche vers le noeud cible ou laisser vide pour ne pas avoir de flèche (lien non dirigé).
- **`color`** : Paramétrages de la couleur du lien.
  - **`color.color`** : Couleur du lien à l'état normal.
  - **`color.highlight`** : Couleur du lien lorsqu'il est sélectionné ou adjacent à un noeud sélectionné.
- **`width`** : Épaisseur du lien *(en pixel)*.

3.4. Personnalisation avancée des noeuds

Pour créer des noeuds avec un design unique, nous allons modifier le code de création des noeuds :

```
nodes.add(
    [
        {
            id: "DEBUT",
            label: "Début",
                        shape: "circle",                    // Affichage sous forme de cercle
                        widthConstraint: { minimum : 50 },  // Taille minimum du noeud
                        color : "#71cf6f" // Couleur Verte
        },
        {
            id: "REVEIL",
            label: "le réveil sonne",
        },
        {
            id: "LEVER",
            label: "Se lever ?",
                        shape: "diamond",                   // Affichage sous forme de losange
                        size: 20, // Taille du losange
                        borderWidth : 5 // Taille de la bordure du losange
        },
        {
            id: "SORTIE_LIT",
            label: "Sortie du lit",
        },
        {
            id: "FIN",
            label: "Fin",
                        shape: "circle",                    // Affichage sous forme de cercle
                        widthConstraint: { minimum : 50 },  // Taille minimum du noeud
                        color : "#f37568"                                // Couleur Rouge
        },
```

```
32          {
33              id: "POUSSER_BOUTON",
34              label: "Pousser le bouton de répétition"
35          },
36          {
37              id: "RETARD",
38              label: "Retard"
39          }
40      ]
41  );
```

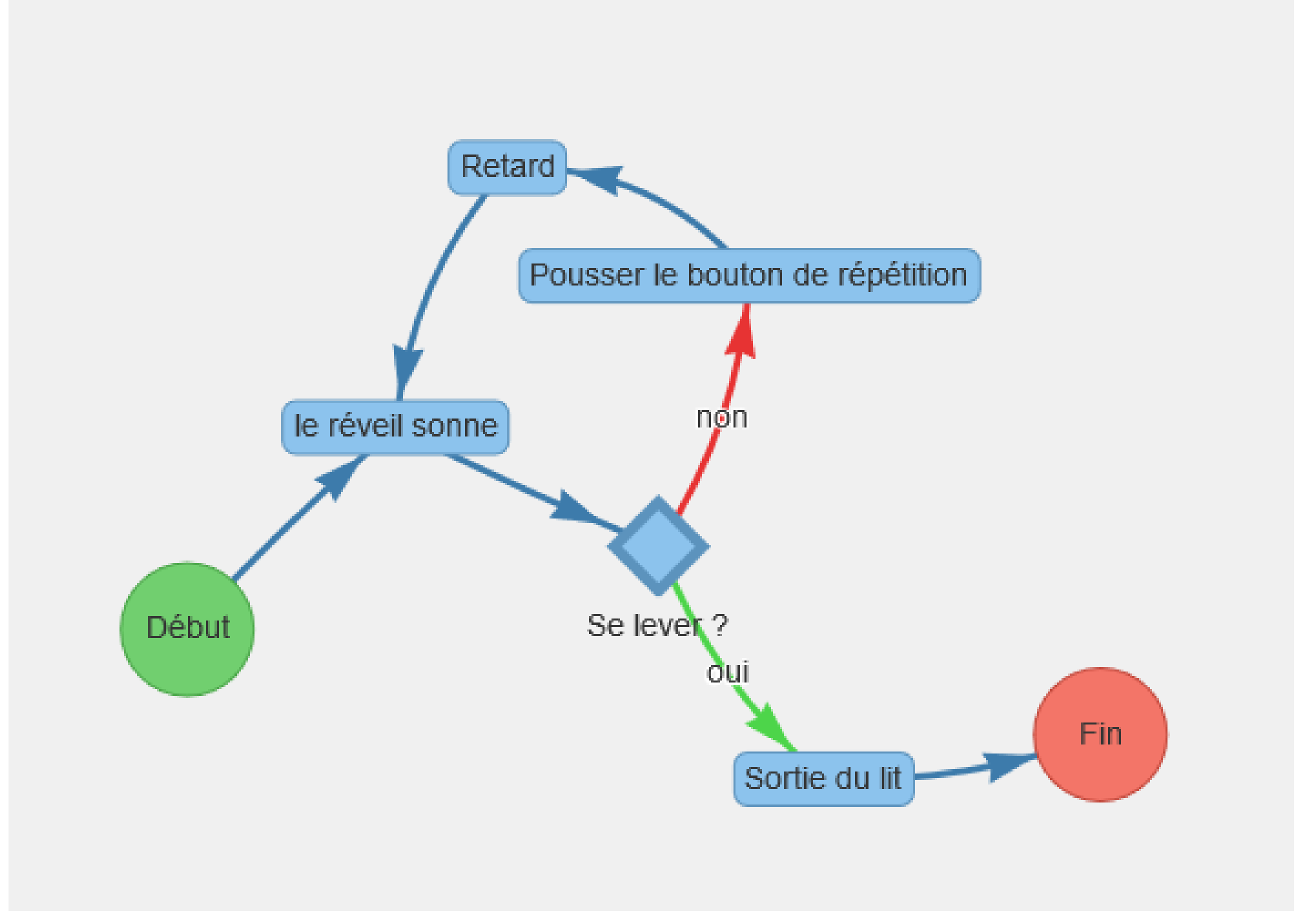


Figure 3.5. – Voila notre résultat finale

Pour cette partie nous avons utilisé trois nouveaux paramètres liés à la taille de noeuds :

- `size` : Taille du nœud, fonctionne uniquement lorsqu'il est affiché sous forme d'un symbole avec du texte au-dessous (`image`, `circularImage`, `diamond`, `dot`, `star`, `triangle`, `triangleDown`, `hexagon`, `square` et `icon`), *exprimé en pixel*.
- `borderWidth`: Taille de la bordure du noeud (1 par défaut), *exprimé en pixel*.
- `widthConstraint` : Objet permettant de définir des contraintes de taille pour le noeud (à utiliser pour lorsque le label est affiché dans le noeuds).
  - `widthConstraint.minimum` : Taille minimum du noeud *(en pixel)*.
  - `widthConstraint.maximum` : Taille maximal du noeud *(en pixel)*.

3.5. Exercice

Récapitulatif du code l'exemple :

Contenu masqué n°3

Dans cet exercice nous allons reprendre le graphe précédent et lui apporter un nouveau design, ce qui vous permettra de vous initier au paramétrage.

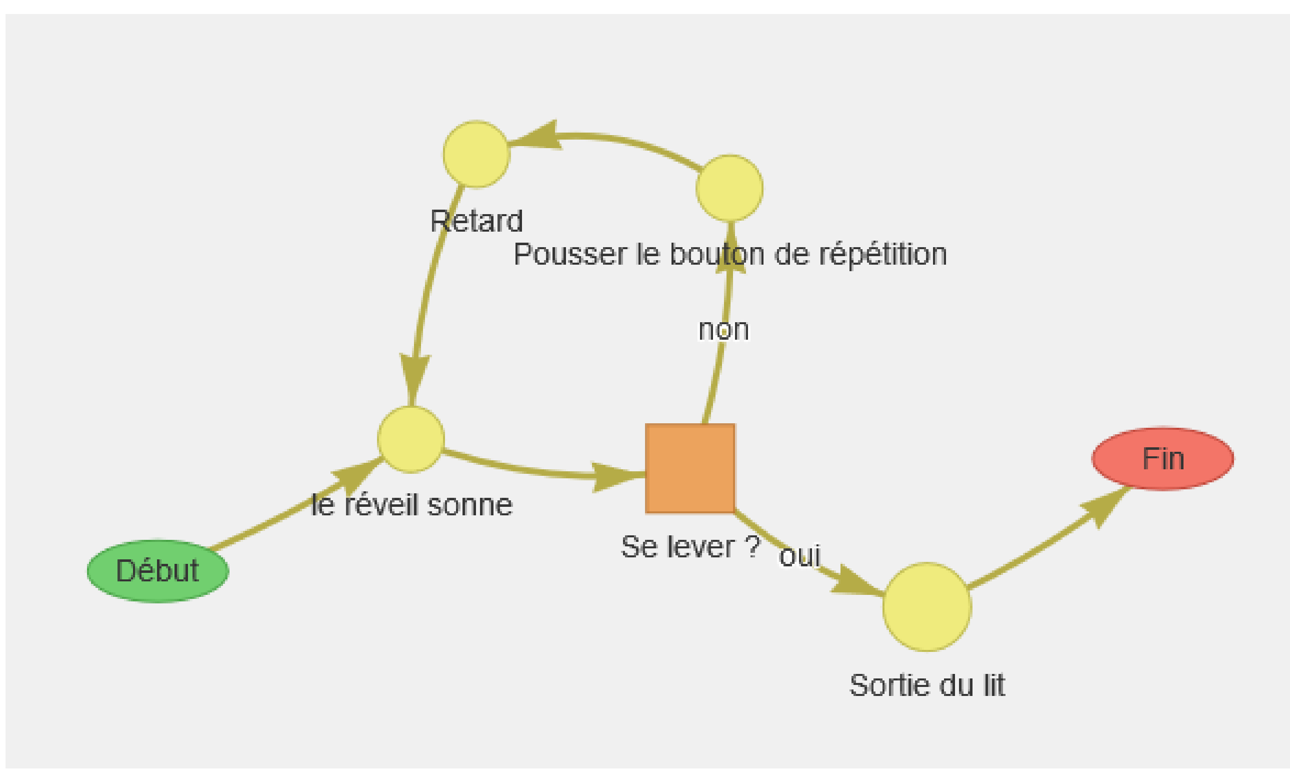


FIGURE 3.6. – L'objectif est d'obtenir ce graphe

Pour vous aider voici une petite description des paramétrages à effectuer :

- Noeuds affichés par défaut sous forme de point, de couleur jaune (#efeb7d) et de taille 15 avec des bordures de taille et de couleur par défaut.
- Noeud `SORTIE_LIT` avec les paramètres par défaut mais de taille 20.
- Noeuds `DEBUT` et `FIN` de type ellipse avec les mêmes couleurs que dans l'exemple précédent.
- Noeud `LEVER` affichés sous forme d'un carré de couleur orange (#eca35d), de taille 20 avec des bordures de taille et de couleur par défaut.
- Liens affichés sous forme de flèches, de couleur jaune foncé (#b5ac47) et de taille 3.

Bonne chance

Vous pouvez retrouver la solution ci dessous :

👁 Contenu masqué n°4

Vous savez maintenant comment paramétrer l'apparence visuelle de votre graphe, dans le prochain chapitre nous allons continuer sur l'apparence visuelle des graphes avec la personnalisation par groupes et les images.

Contenu masqué

Contenu masqué n°3

```
let nodes = new vis.DataSet([]);
let edges = new vis.DataSet([]);

let container = document.getElementById('graphe');

let data = {
 nodes: nodes,
 edges: edges
};
let options = {
 nodes : {
     shape: "box",            // Nœud affiché sous forme de boîte
         entourant le label
     color: "#8cc3ed",        // Nœud de couleur Bleue
 },
 edges : {
   arrows : "to",             // Flèche dirigée du premier vers le
        second noeud
   color : {
       color: "#3d7bab"       // Lien de couleur Bleue Foncé
   },
   width : 3                  // Épaisseur du lien de 3
 }
}

// Initialisation de l'instance de VisNetwork
let network = new vis.Network(container, data, options);


// Ajout des noeuds
nodes.add(
 [
   {
```

```
      id: "DEBUT",
      label: "Début",
      shape: "circle",                          // Affichage sous forme de
          cercle
      widthConstraint: { minimum : 50 },  // Taille minimum du noeud
      color : "#71cf6f"                         // Couleur Verte
    },
    {
      id: "REVEIL",
      label: "le réveil sonne",
    },
    {
      id: "LEVER",
      label: "Se lever ?",
      shape: "diamond",                         // Affichage sous forme de
          losange
      size: 20,                                 // Taille du losange
      borderWidth : 5                           // Taille de la bordure du
          losange
    },
    {
      id: "SORTIE_LIT",
      label: "Sortie du lit",
    },
    {
      id: "FIN",
      label: "Fin",
      shape: "circle",                          // Affichage sous forme de
          cercle
      widthConstraint: { minimum : 50 },  // Taille minimum du noeud
      color : "#f37568"                         // Couleur Rouge
    },
    {
      id: "POUSSER_BOUTON",
      label: "Pousser le bouton de répétition"
    },
    {
      id: "RETARD",
      label: "Retard"
    }
  ]
);

edges.add(
  [
    {
      from: "DEBUT",
      to: "REVEIL",
      label: ""
    },
```

```
      {
        from: "REVEIL",
        to: "LEVER",
        label: ""
      },
      {
        from: "LEVER",
        to: "SORTIE_LIT",
        label: "oui",
        color: { color : "#4dd54a" } // Couleur Verte
      },
      {
        from: "SORTIE_LIT",
        to: "FIN",
        label: ""
      },
      {
        from: "LEVER",
        to: "POUSSER_BOUTON",
        label: "non",
        color: { color : "#e73333" } // Couleur Rouge
      },
      {
        from: "POUSSER_BOUTON",
        to: "RETARD",
        label: ""
      },
      {
        from: "RETARD",
        to: "REVEIL",
        label: ""
      }
    ]
);
```

Retourner au texte.

Contenu masqué n°4

```
let nodes = new vis.DataSet([]);
let edges = new vis.DataSet([]);

let container = document.getElementById('graphe');

let data = {
  nodes: nodes,
```

```
8     edges: edges
9   };
10  let options = {
11    nodes : {
12        shape: "dot",           // Nœud affiché sous forme de
              boîteentourant le label
13        color: "#efeb7d",       // Nœud de couleur Jaune
14        size: 15
15    },
16    edges : {
17      arrows : "to",            // Flèche dirigée du premier vers le
            second noeud
18      color : {
19          color: "#b5ac47"      // Lien de couleur Jaune Foncé
20      },
21      width : 3                 // Épaisseur du lien de 3
22    }
23  }
24
25  // Initialisation de l'instance de VisNetwork
26  let network = new vis.Network(container, data, options);
27
28
29  // Ajout des noeuds
30  nodes.add(
31    [
32      {
33        id: "DEBUT",
34        label: "Début",
35        shape: "ellipse",         // Affichage sous forme d'ellipse
36        color : "#71cf6f"         // Couleur verte
37      },
38      {
39        id: "REVEIL",
40        label: "le réveil sonne",
41      },
42      {
43        id: "LEVER",
44        label: "Se lever ?",
45        shape: "square",          // Affichage sous forme de carré
46        size: 20,                 // Taille du carré
47        color: "#eca35d"          // Couleur du carré en orange
48      },
49      {
50        id: "SORTIE_LIT",
51        label: "Sortie du lit",
52        size: 20,                 // Taille du noeud de 20
53      },
54      {
55        id: "FIN",
```

```
56        label: "Fin",
57        shape: " ellipse",         // Affichage sous forme d'ellipse
58        color : "#f37568"          // Couleur rouge
59      },
60      {
61        id: "POUSSER_BOUTON",
62        label: "Pousser le bouton de répétition"
63      },
64      {
65        id: "RETARD",
66        label: "Retard"
67      }
68    ]
69  );
70
71  edges.add(
72    [
73      {
74        from: "DEBUT",
75        to: "REVEIL",
76        label: ""
77      },
78      {
79        from: "REVEIL",
80        to: "LEVER",
81        label: ""
82      },
83      {
84        from: "LEVER",
85        to: "SORTIE_LIT",
86        label: "oui"
87      },
88      {
89        from: "SORTIE_LIT",
90        to: "FIN",
91        label: ""
92      },
93      {
94        from: "LEVER",
95        to: "POUSSER_BOUTON",
96        label: "non"
97      },
98      {
99        from: "POUSSER_BOUTON",
100       to: "RETARD",
101       label: ""
102     },
103     {
104       from: "RETARD",
105       to: "REVEIL",
```

```
106         label: ""
107       }
108     ]
109 );
```

Retourner au texte.

4. La personnalisation visuelle partie 2

Dans ce second chapitre dédié à la personnalisation visuelle des graphes, je vais vous présenter la gestion des groupes avec de nouvelles options de personnalisations des noeuds et la gestion des images.

Pour les plus courageux d'entre vous, ce chapitre se terminera par un petit défi .

4.1. Les groupes

VisNetwork nous permet de définir des groupes pour permettre la simplification du paramétrage visuel des noeuds, pour cela il suffit d'ajouter un paramètre `group` à nos noeuds et de définir les paramétrages du groupe dans les options.

Dans ce chapitre nous allons travailler à partir d'un autre exemple, représentant les liens entre des tutoriels, des auteurs et des tags issue de Zeste de Savoir :

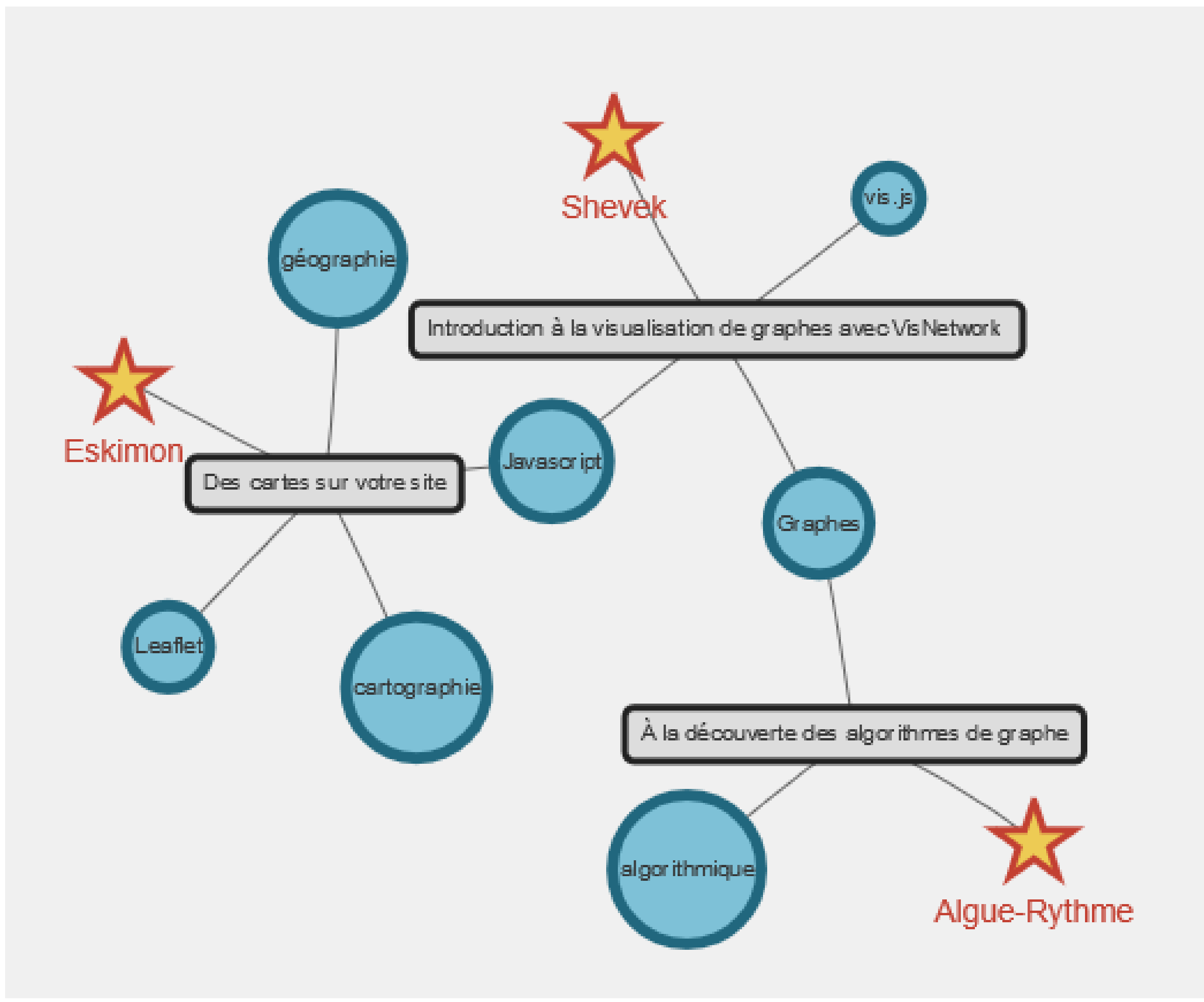


Figure 4.1. – Voici le graphe que nous allons chercher à obtenir

Je vous donne le code de base de ce chapitre avec des noeuds appartenant à des groupes.

```
let nodes = new vis.DataSet([]);
let edges = new vis.DataSet([]);

let container = document.getElementById('graphe');

let data = {
  nodes: nodes,
  edges: edges
};
let options = {
    nodes : {},
    edges : {}
};

// Initialisation de l'instance de VisNetwork
```

```
16 let network = new vis.Network(container, data, options);
17
18 // Ajout des noeuds
19 nodes.add(
20   [
21     {
22       id: "TUTO_VIS",
23       label:
          "Introduction à la visualisation de graphes avec VisNetwork ",
24       group : "tuto"
25     },
26     {
27       id: "USER_SHE",
28       label: "Shevek",
29       group : "user"
30     },
31     {
32       id: "TAG_JS",
33       label: "Javascript",
34       group : "tag"
35     },
36     {
37       id: "TAG_GRA",
38       label: "Graphes",
39       group : "tag"
40     },
41     {
42       id: "TAG_VIS",
43       label: "vis.js",
44       group : "tag"
45     },
46     {
47       id: "TUTO_GRA",
48       label: "À la découverte des algorithmes de graphe",
49       group : "tuto"
50     },
51     {
52       id: "USER_ALG",
53       label: "Algue-Rythme",
54       group : "user"
55     },
56     {
57       id: "TAG_ALG",
58       label: "algorithmique",
59       group : "tag"
60     },
61     {
62       id: "TUTO_CAR",
63       label: "Des cartes sur votre site",
64       group : "tuto"
```

```
    },
    {
      id: "TAG_LEA",
      label: "Leaflet",
      group : "tag"
    },
    {
      id: "TAG_GEO",
      label: "géographie",
      group : "tag"
    },
    {
      id: "TAG_CAR",
      label: "cartographie",
      group : "tag"
    },
    {
      id: "USER_ESK",
      label: "Eskimon",
      group : "user"
    }
  ]
);

// Ajout des liens
edges.add(
  [
    {
      from: "TUTO_VIS",
      to: "USER_SHE",
      label: ""
    },
    {
      from: "TUTO_VIS",
      to: "TAG_JS",
      label: ""
    },
    {
      from: "TUTO_VIS",
      to: "TAG_GRA",
      label: ""
    },
    {
      from: "TUTO_VIS",
      to: "TAG_VIS",
      label: ""
    },
    {
      from: "TUTO_GRA",
      to: "USER_ALG",
```

```
115         label: ""
116       },
117       {
118         from: "TUTO_GRA",
119         to: "TAG_ALG",
120         label: ""
121       },
122       {
123         from: "TUTO_GRA",
124         to: "TAG_GRA",
125         label: ""
126       },
127       {
128         from: "TUTO_CAR",
129         to: "USER_ESK",
130         label: ""
131       },
132       {
133         from: "TUTO_CAR",
134         to: "TAG_LEA",
135         label: ""
136       },
137       {
138         from: "TUTO_CAR",
139         to: "TAG_GEO",
140         label: ""
141       },
142       {
143         from: "TUTO_CAR",
144         to: "TAG_CAR",
145         label: ""
146       },
147       {
148         from: "TUTO_CAR",
149         to: "TAG_JS",
150         label: ""
151       },
152     ]
153 );
```

Nous obtenons ce graphe que nous allons pouvoir personnaliser :

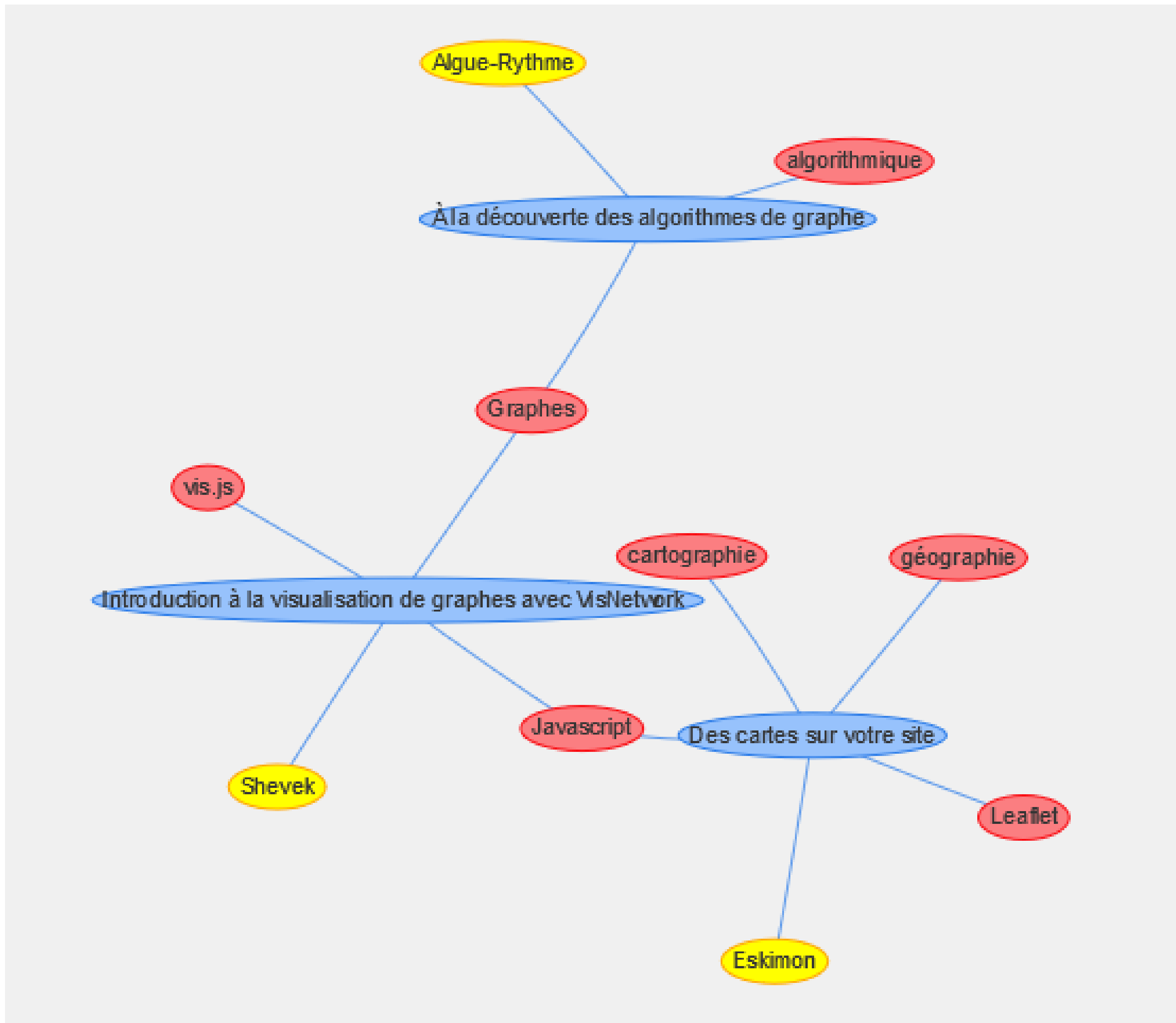


FIGURE 4.2. – VisNetwork attribut automatiquement une couleur à chaque groupe de noeuds

4.2. Le design des groupes

Le design des groupes ce fait directement dans les options, on ajoute un objet `groups` dans lequel on ajoute un objet par groupe (avec pour nom celui du groupe), dans lequel on va définir nos propriétés :

```
let options = {
  nodes : {},
  edges : {},
  groups:
  {
    mongroupe :
    {
      // Paramètres des noeuds du groupe
```

```
        }
    }
```

Dans notre cas on crée 3 groupes de paramétrage : `tuto`, `user` et `tag`.

Voici une petite description des nouvelles propriétés utilisées que nous allons utiliser :

- **`margin`** : Marge entre le label et la bordure du noeud (lorsque le label est contenu dans le noeud), *exprimé en pixel*.
- **`color.highlight`** : Paramètres de couleur du noeud lorsqu'il est sélectionné.
  - **`color.highlight.background`** : Couleur de fond du noeud lorsqu'il est sélectionné.
  - **`color.highlight.border`** : Couleur de bordure du noeud lorsqu'il est sélectionné.
- **`font`** : Paramétrage de la police d'écriture du label, peut-être définie directement *(ex: `14px arial red`)* ou peut être paramétré via ses paramètres enfant :
  - **`font.color`** : Couleur de l'affichage du label.
  - **`font.size`** : Taille de la police de caractère du label.

Voici le code complet :

```
let options = {
  nodes : {},
  edges : {},
  groups:
  {
    tuto:
    {
      shape: "box",                 // Nœud affiché sous forme de boîte entourant le label
      margin : 10,                  // Marge de 10 entre le contour et le label
      color:
      {
        background: '#dddddd',      // Fond de couleur gris clair
        border: '#222222'           // Bordure de couleur noir
      },
      borderWidth:3                 // Bordure de taille 3
    },
    user : {
      shape: "star",                // Nœud affiché sous forme d'étoile
      borderWidth: '3',             // Bordure de taille 3
      color:
      {
        background: '#edcb54',      // Fond de couleur jaune/orange
        border: '#c34032',          // Bordure de couleur rouge
        highlight :
        {
          background: '#c34032',  // Fond de couleur rouge lorsque le noeud est sélectionné
```

```
27            border: '#c34032'        // Bordure de couleur rouge
                  lorsque le noeud est sélectionné
28          }
29        },
30        font: {
31          color: "#c34032",          // Couleur de l'écriture du label
                en rouge
32          size: 20                   // Police d'écriture du label de
                taille 20
33        }
34      },
35      tag: {
36        shape: "circle",             // Nœud affiché sous forme de
              cercle
37        borderWidth: '5',            // Bordure de taille 5
38        color:
39        {
40          background: '#7ec1d7',     // Fond de couleur Bleue
41          border: '#21677e',         // Bordure de couleur Bleue
                foncée
42          highlight :
43          {
44            background: '#7ec1d7',  // Fond de couleur Bleue lorsque
                  le noeud est sélectionné
45            border: '#21677e'        // Bordure de couleur Bleue foncé
                  lorsque le noeud est sélectionné
46          }
47        }
48      }
49    }
50  };
```

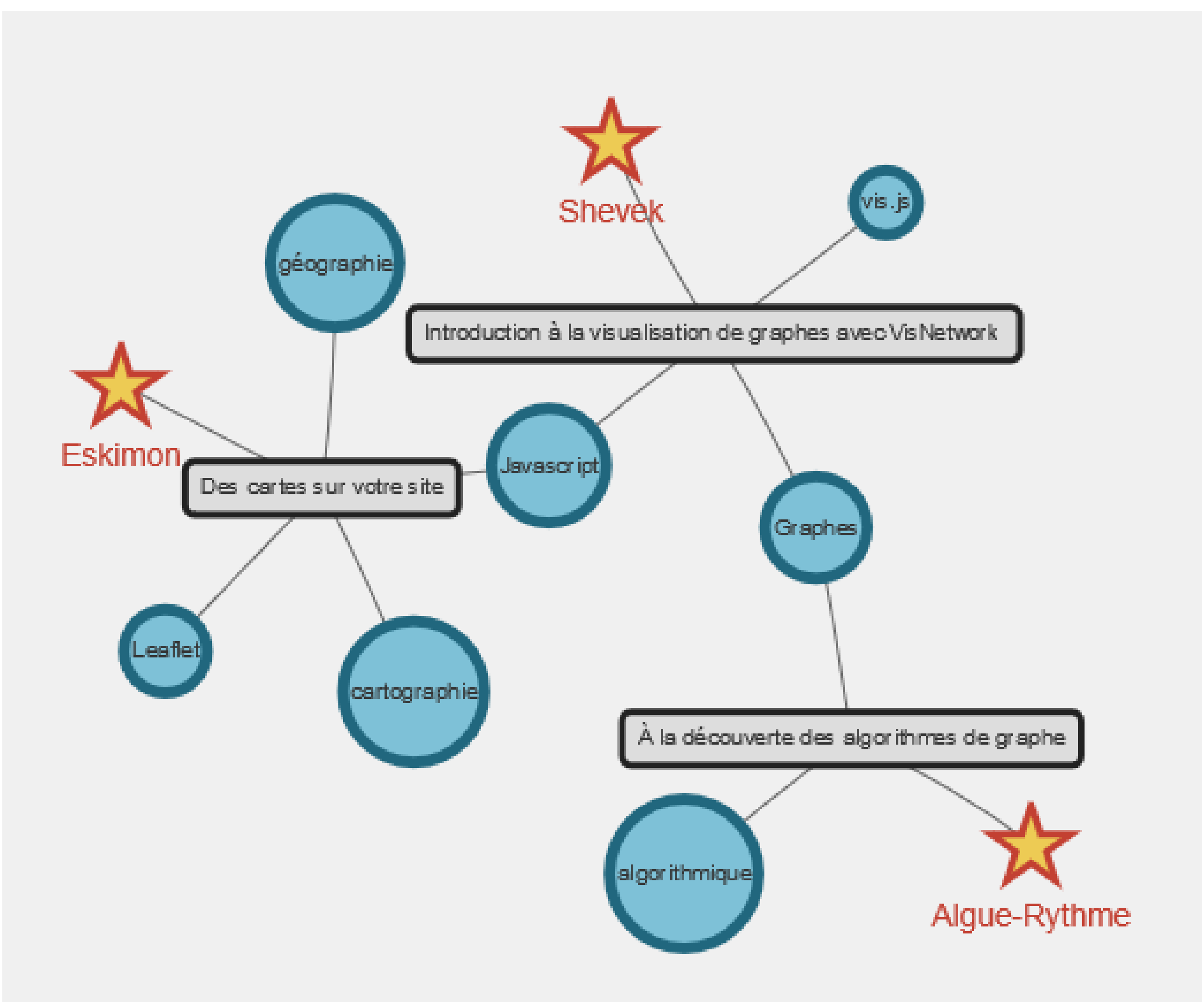


Figure 4.3. – Nous obtenons notre super graphe !

> Je ne vous le présente pas dans ce tuto, mais sachez que vous pouvez également utiliser les groupes pour définir le design des liens.

4.3. Les images

VisNetwork possède deux types d'affichages d'images qui se définissent dans la propriété `shape` du noeud :

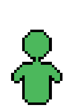
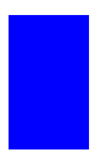
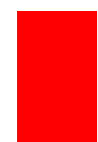
— `image` : Une image est affichée à la place du noeud.
— `circularImage` : Nœud avec une image contenue dans un cercle.

Les images possèdent des options qui leur sont propres :

— `image` : URL de l'image à afficher.

— `brokenImage` : URL de l'image par défaut qui est affichée lorsque le chargement de l'image choisie a échoué.

Pour l'exemple suivant, je vous fournis quelques images qui nous permettront de créer notre graphe :

Voici un exemple de création d'un graphe avec 3 images (deux images circulaires et une image normale) :

```
let nodes = new vis.DataSet([]);
let edges = new vis.DataSet([]);

let container = document.getElementById('graphe');

let data = {
  nodes: nodes,
  edges: edges
};
let options = {
    nodes : {
    },
    edges : { /* Paramètre par defaut des liens */
        arrows: "to",
        color : {
                        color: "#f8944b"
                },
        width : 1,
        selectionWidth: 2,
    }
};

let network = new vis.Network(container, data, options);

nodes.add(
        [
                { /* Noeud représentant une personne */
                        id: 1,
                        label: "Personne",
                        shape: "circularImage",
                        image: "perso.png",
                        borderWidth: 1,
                        size: 15,
                        color: {
                          border: "#42a250",
```

```
36                              background: "#cdf4d2"
37                            },
38                    },
39                    { /* Noeud avec un drapeau représentant la France
                          */
40                            id: 2,
41                            label: "France",
42                            shape: "image",
43                            size: 15,
44                            image: "france.png",
45                    },
46                    { /* Noeud de la ville de Paris */
47                            id: 3,
48                            label: "Paris",
49                            shape: "circularImage",
50                            image: "place.png",
51                            size: 20,
52                            borderWidth: 3,
53                            color: {
54                              border: "#75cbd0",
55                              background: "#caf3f6"
56                            },
57                    }
58            ]
59    );
60
61    edges.add(
62            [
63                    { /* Lien entre la personne et la france */
64                            from: 1,
65                            to: 2,
66                            label: "Nationalitée"
67                    },
68                    { /* Lien entre la personne et Paris */
69                            from: 1,
70                            to: 3,
71                            label: "Née à"
72                    }
73            ]
74    );
```

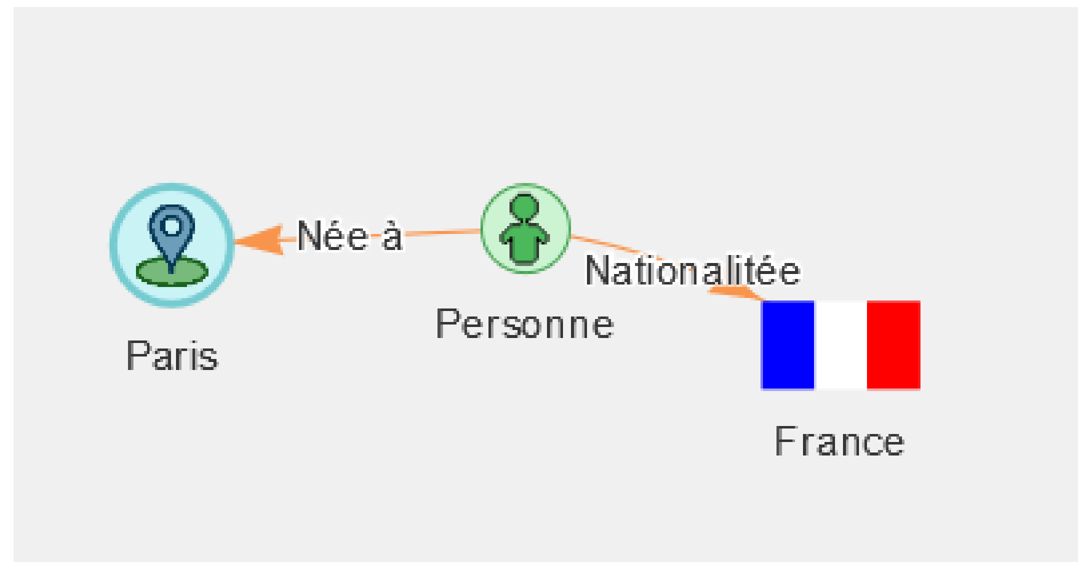


FIGURE 4.4. – Représentation d'une personne née à Paris et de nationalité française

4.4. Excercice

Je vous propose un petit exercice pour mettre en pratique le paramétrage des groupes et des images, le but va être d'obtenir un joli graphe qui représente des espèces d'animaux triés par catégories.

Voici le code de base avec des images issues de Wikipédia :

```
let nodes = new vis.DataSet([]);
let edges = new vis.DataSet([]);

let container = document.getElementById('graphe');

let data = {
  nodes: nodes,
  edges: edges
};
let options = {
    nodes : {},
    edges : {},
    groups : {}
};

// Initialisation de l'instance de VisNetwork
let network = new vis.Network(container, data, options);

// Ajout des noeuds
nodes.add(
  [
    {
```

```
23        id: "Animal",
24        label: "Animal",
25        group : "grande_catégorie"
26      },
27      {
28        id: "Mammifère",
29        label: "Mammifère",
30        group : "moyenne_catégorie"
31      },
32      {
33        id: "Cétacé",
34        label: "Cétacé",
35        group : "petite_catégorie",
36        image:
            "https://upload.wikimedia.org/wikipedia/commons/thumb/2/2b/The_Cetac
37      },
38      {
39        id: "Baleine à bosse",
40        label: "Baleine à bosse",
41        group : "espece",
42        image :
            "https://upload.wikimedia.org/wikipedia/commons/thumb/9/9e/Humpback_
43      },
44      {
45        id: "Dauphin",
46        label: "Dauphin",
47        group : "espece",
48        image :
            "https://upload.wikimedia.org/wikipedia/commons/thumb/1/10/Tursiops_
49      },
50      {
51        id: "Félin",
52        label: "Félin",
53        group : "petite_catégorie",
54        image:
            "https://upload.wikimedia.org/wikipedia/commons/thumb/5/5b/The_Felid
55      },
56      {
57        id: "Tigre",
58        label: "Tigre",
59        group : "espece",
60        image :
            "https://upload.wikimedia.org/wikipedia/commons/thumb/4/41/Siberisch
61      },
62      {
63        id: "Oiseau",
64        label: "Oiseau",
65        group : "moyenne_catégorie"
66      },
67      {
```

```
68          id: "Flamant",
69          label: "Flamant",
70          group : "espece",
71          image :
                "https://upload.wikimedia.org/wikipedia/commons/thumb/4/45/Greater_Fl
72        },
73        {
74          id: "Corbeau",
75          label: "Corbeau",
76          group : "espece",
77          image :
                "https://upload.wikimedia.org/wikipedia/commons/thumb/2/29/Common_rav
78        },
79        {
80          id: "Poisson",
81          label: "Poisson",
82          group : "moyenne_catégorie"
83        },
84        {
85          id: "Requin",
86          label: "Requin",
87          group : "espece",
88          image :
                "https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/White_shar
89        }
90      ]
91    );
92
93    // Ajout des liens
94    edges.add(
95      [
96        {
97          from: "Animal",
98          to: "Mammifère",
99          label: ""
100       },
101       {
102         from: "Mammifère",
103         to: "Cétacé",
104         label: ""
105       },
106       {
107         from: "Mammifère",
108         to: "Félin",
109         label: ""
110       },
111       {
112         from: "Cétacé",
113         to: "Baleine à bosse",
114         label: ""
```

```
115      },
116      {
117        from: "Cétacé",
118        to: "Dauphin",
119        label: ""
120      },
121      {
122        from: "Félin",
123        to: "Tigre",
124        label: ""
125      },
126      {
127        from: "Animal",
128        to: "Oiseau",
129        label: ""
130      },
131      {
132        from: "Oiseau",
133        to: "Flamant",
134        label: ""
135      },
136      {
137        from: "Oiseau",
138        to: "Corbeau",
139        label: ""
140      },
141      {
142        from: "Animal",
143        to: "Poisson",
144        label: ""
145      },
146      {
147        from: "Poisson",
148        to: "Requin",
149        label: ""
150      }
151    ]
152  );
```

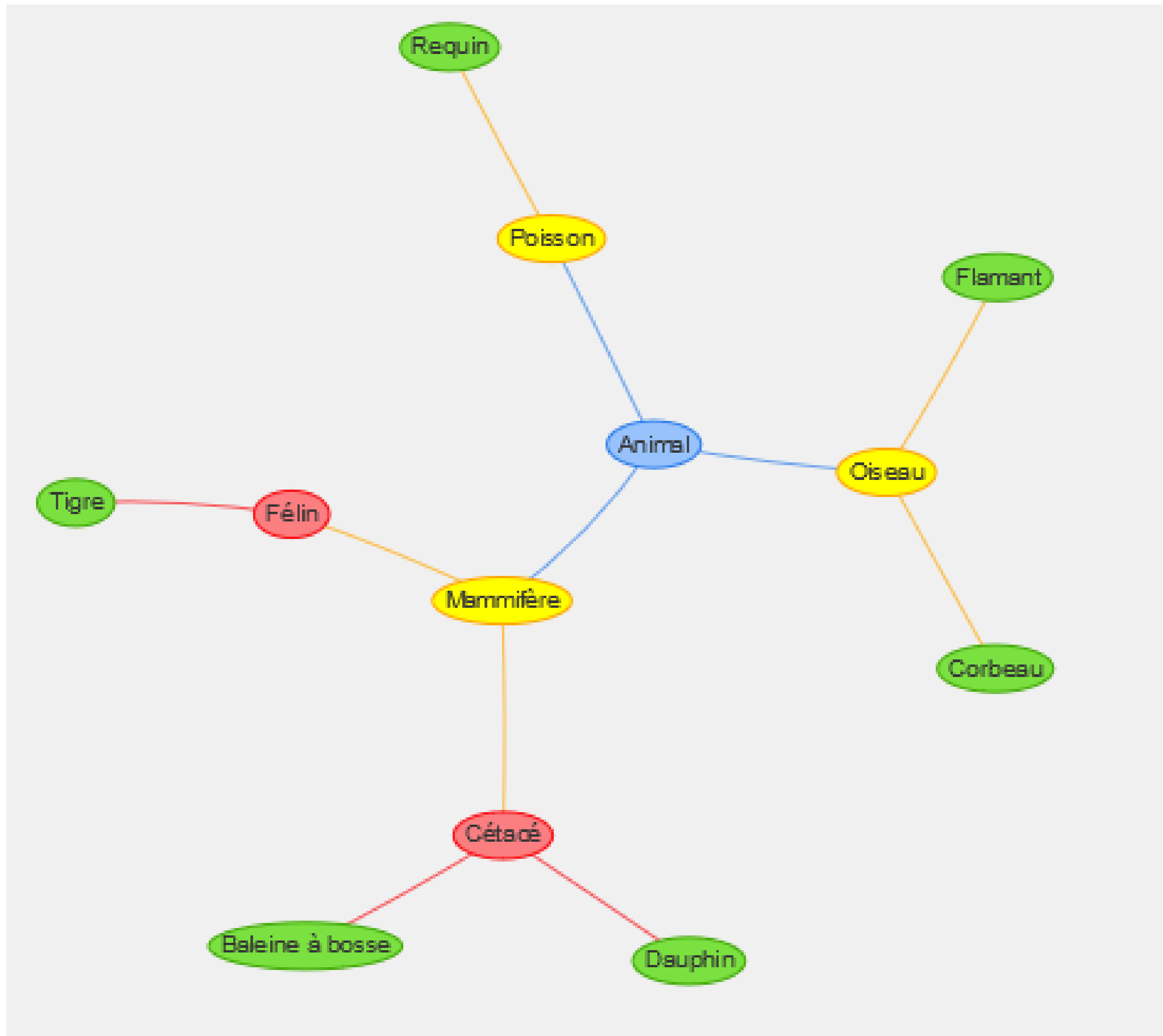


FIGURE 4.5. – Notre graphe de départ

L'objectif va être d'obtenir ce graphe :

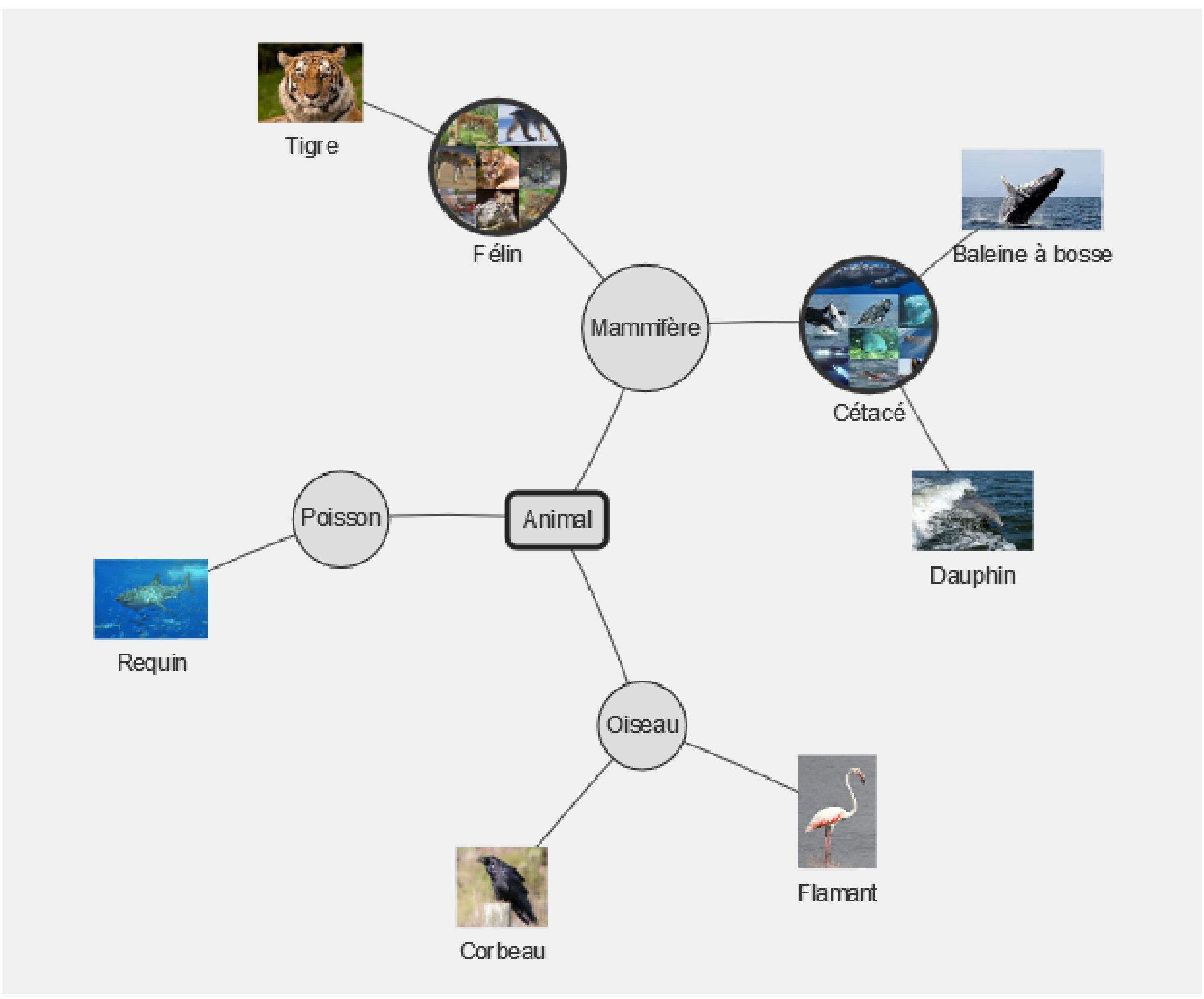


FIGURE 4.6. – Graphe à obtenir

Comme vous pouvez le voir on a 4 groupes :

— `grande_catégorie` : Affiché sous forme de rectangle entourant le label avec une marge de 10. Il a une couleur de fond gris clair (#dddddd) et une bordure gris foncé (#222222) de taille 3.
— `moyenne_catégorie` : Affiché sous forme de cercle à fond gris clair (#dddddd) et à bordure gris foncé (#222222).
— `petite_catégorie` : Affiché sous forme d'image circulaire de taille 40, avec une bordure de taille 6 et de couleur gris foncé (#222222).
— `espece` : Affiché sous forme d'image.

Je vous souhaite bonne chance, si vous avez oublié les propriétés vues lors du chapitre précédent, vous pouvez utiliser le récapitulatif des propriétés présent plus bas.

Vous pouvez retrouver la correction ci-dessous :

Contenu masqué n°5

4.5. Défis

Je propose aux plus motivés d'entre vous un petit défis :

Le but est de reprendre le principe du graphe précédent et d'essayer d'ajouter de nouveaux animaux, de nouvelles catégories et surtout de nouveaux paramétrage de design pour tenter de réaliser le graphe le plus beau possible. 😊

Si vous êtes satisfait du résultat je vous invite à le partager dans les commentaires du tutoriel (En bas de la page d'accueil).

4.6. Récapitulatif des propriétés

Voici un petit récapitulatif des propriétés de paramétrage des noeuds, tous ne sont pas cités. Vous pouvez retrouver les autres dans la documentation de VisNetwork :

- **`shape`** : Défini la forme d'affichage du noeuds :
  - `image` : Affichage d'une image.
  - `circularImage` : Affichage d'une image contenue dans un cercle.
  - `circle` : Cercle avec le label du noeud au centre.
  - `ellipse` : Ellipse entourant le label du nœud.
  - `box` : Rectangle/boîte entourant le label du nœud.
  - `diamond`, `dot`, `star`, `triangle`, `triangleDown`, `hexagon` et `square` : Symboleavec le label affiché en dessous.
- **`color`** : Corresponds à la couleur du noeud, peut être soit une couleur ou un objet contenant des informations plus détaillées :
  - **`color.background`** : Couleur de fond du noeud.
  - **`color.border`** : Couleur de la bordure, s'affichant si le nœud possède une bordure.
  - **`color.highlight`** : Paramètres de couleur du noeud lorsqu'il est sélectionné.
    - **`color.highlight.background`** : Couleur de fond du noeud lorsqu'il est sélectionné.
    - **`color.highlight.border`** : Couleur de bordure du noeud lorsqu'il est sélectionné.
- **`font`** : Paramétrage de la police d'écriture du label, peut-être définie directement *(ex: `14px arial red`)* ou peut être paramétré via ses paramètres enfants :
  - **`font.color`** : Couleur de l'affichage du label.
  - **`font.size`** : Taille de la police de caractère du label *(en pixel)*.
- **`size`** : Taille du noeud *(en pixel)*.
- **`borderWidth`** : Taille de la bordure du noeud *(en pixel)*.
- **`margin`** : Marge entre le label et la bordure du noeud (lorsque le label est contenu dans le noeud), *exprimé en pixel*.

Voici une petite liste des principaux paramétrages des liens, tous ne sont pas cités vous pouvez retrouver les autres dans la documentation de VisNetwork :

- **`arrows`** : Défini le type de représentation du lien :
  - `to` : Flèche sur le noeud choisi dans la propriété `to` du lien.
  - `from` : Flèche sur le noeud choisi dans la propriété `from` du lien.
  - `middle` : Flèche au centre du lien.

- `to;from` : Flèches aux deux extrémités du lien.
- Peut également être paramétré avec d'autres icônes et des tailles personnalisés (voir la doc pour plus d'infos) .
- **`color`** : Paramétrages de la couleur du lien.
  - **`color.color`** : Couleur du lien à l'état normal.
  - **`color.highlight`** : Couleur du lien lorsqu'il est sélectionné ou adjacent à un noeud sélectionné.
- **`font`** : Paramètre de la police d'écriture du label du lien, peut-être défini directement *(ex: `14px arial red`)* ou peut être paramétré via ses paramètres enfants :
  - **`font.color`** : Couleur du label.
  - **`font.size`** : Taille de la police du label *(en pixel)*.
  - **`font.strokeColor`** : Couleur de surbrillance du label.
- **`width`** : Épaisseur du lien *(en pixel)*.
- **`selectionWidth`** : Épaisseur du lien lorsqu'il est sélectionné *(en pixel)*.
- **`smooth`** : Définis la manière dont les liens sont dessinés, je ne vais pas présenter cette option dans ce tutoriel, si vous souhaitez l'utiliser cet exemple qui vous permettra d'expérimenter les différents types de smooth .

Nous en avons fini avec la personnalisation visuelle, si vous souhaitez vous exercer un peu au paramétrage avant de passer au chapitre suivant, vous pouvez retrouver via la doc la liste complète des paramètres sur les noeuds et les liens .

Le prochain chapitre traitera des notions de forces exercés sur les graphes.

Contenu masqué

Contenu masqué n°5

```
let nodes = new vis.DataSet([]);
let edges = new vis.DataSet([]);

let container = document.getElementById('graphe');

let data = {
  nodes: nodes,
  edges: edges
};
let options = {
    nodes : {},
    edges : {},
    groups : {
      grande_catégorie :
      {
```

```
16        shape: "box",                // Nœud affiché sous forme de
              boîte entourant le label
17        margin : 10,                 // Marge de 10 entre le contour
              et le label
18        color:
19        {
20          background: '#dddddd',     // Fond de couleur gris clair
21          border: '#222222'          // Bordure de couleur gris
                foncé
22        },
23        borderWidth:3                // Bordure de taille 3
24      },
25      moyenne_catégorie :
26      {
27        shape: "circle",             // Nœud affiché sous forme de
              cercle entourant le label
28        color:
29        {
30          background: '#dddddd',     // Fond de couleur gris clair
31          border: '#222222'          // Bordure de couleur gris
                foncé
32        }
33      },
34      petite_catégorie :
35      {
36        shape: 'circularImage',     // Nœud affiché sous forme
              d'image contenu dans un cercle
37        borderWidth: 6,             // Bordure de l'image
38        size: 40,                   // Taille du noeud
39        color:
40        {
41          border: '#333333'         // Bordure de couleur gris foncé
42        }
43      },
44      espece :
45      {
46        shape: 'image'             // Nœud affiché sous forme
              d'image
47      }
48    }
49  };
50
51  // Initialisation de l'instance de VisNetwork
52  let network = new vis.Network(container, data, options);
53
54  // Ajout des noeuds
55  nodes.add(
56    [
57      {
58        id: "Animal",
```

```
59        label: "Animal",
60        group : "grande_catégorie"
61      },
62      {
63        id: "Mammifère",
64        label: "Mammifère",
65        group : "moyenne_catégorie"
66      },
67      {
68        id: "Cétacé",
69        label: "Cétacé",
70        group : "petite_catégorie",
71        image:
            "https://upload.wikimedia.org/wikipedia/commons/thumb/2/2b/The_Cetace
72      },
73      {
74        id: "Baleine à bosse",
75        label: "Baleine à bosse",
76        group : "espece",
77        image :
            "https://upload.wikimedia.org/wikipedia/commons/thumb/9/9e/Humpback_s
78      },
79      {
80        id: "Dauphin",
81        label: "Dauphin",
82        group : "espece",
83        image :
            "https://upload.wikimedia.org/wikipedia/commons/thumb/1/10/Tursiops_t
84      },
85      {
86        id: "Félin",
87        label: "Félin",
88        group : "petite_catégorie",
89        image:
            "https://upload.wikimedia.org/wikipedia/commons/thumb/5/5b/The_Felida
90      },
91      {
92        id: "Tigre",
93        label: "Tigre",
94        group : "espece",
95        image :
            "https://upload.wikimedia.org/wikipedia/commons/thumb/4/41/Siberische
96      },
97      {
98        id: "Oiseau",
99        label: "Oiseau",
100       group : "moyenne_catégorie"
101     },
102     {
103       id: "Flamant",
```

```
104         label: "Flamant",
105         group : "espece",
106         image :
                "https://upload.wikimedia.org/wikipedia/commons/thumb/4/45/Greater_F
107       },
108       {
109         id: "Corbeau",
110         label: "Corbeau",
111         group : "espece",
112         image :
                "https://upload.wikimedia.org/wikipedia/commons/thumb/2/29/Common_ra
113       },
114       {
115         id: "Poisson",
116         label: "Poisson",
117         group : "moyenne_catégorie"
118       },
119       {
120         id: "Requin",
121         label: "Requin",
122         group : "espece",
123         image :
                "https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/White_sha
124       },
125 
126       //
127     ]
128 );
129 
130 // Ajout des liens
131 edges.add(
132     [
133       {
134         from: "Animal",
135         to: "Mammifère",
136         label: ""
137       },
138       {
139         from: "Mammifère",
140         to: "Cétacé",
141         label: ""
142       },
143       {
144         from: "Mammifère",
145         to: "Félin",
146         label: ""
147       },
148       {
149         from: "Cétacé",
150         to: "Baleine à bosse",
```

```
151         label: ""
152       },
153       {
154         from: "Cétacé",
155         to: "Dauphin",
156         label: ""
157       },
158       {
159         from: "Félin",
160         to: "Tigre",
161         label: ""
162       },
163       {
164         from: "Animal",
165         to: "Oiseau",
166         label: ""
167       },
168       {
169         from: "Oiseau",
170         to: "Flamant",
171         label: ""
172       },
173       {
174         from: "Oiseau",
175         to: "Corbeau",
176         label: ""
177       },
178       {
179         from: "Animal",
180         to: "Poisson",
181         label: ""
182       },
183       {
184         from: "Poisson",
185         to: "Requin",
186         label: ""
187       },
188 
189       //
190     ]
191 );
```

Retourner au texte.

5. Les forces

Jeune padawan, il est maintenant temps d'apprendre à maitriser la force.

5.1. Qu'est qu'une force

Petite définition :

> Une force modélise, en physique, une action mécanique exercée par un objet sur un autre et capable d'imposer une accélération induisant la modification du vecteur vitesse (une force exercée sur l'objet fait aller celui-ci plus vite, moins vite ou le fait tourner). *source Wikipédia*

Dans le cas des graphes, la force correspond à la physique du graphe, c'est-à-dire aux forces d'attractions qu'exercent les noeuds les un sur les autres modifiant ainsi la position des autres noeuds. Vous avez surement remarqué que lorsque vous déplacez un noeud les autres le suivent, cela est dû tout simplement aux forces.

Ce sont donc ces forces qui vont gérer la manière dont les noeuds du graphe se positionnent.

5.2. Les différents algoritmes de force

VisNetwork nous fourni 4 algorithmes gérant les forces qu'il est possible de paramétrer :

— `barnesHut` : C'est l'algorithme de placement par défaut, il a l'avantage d'être performant et paramétrable et il est recommandé de l'utiliser.
— `forceAtlas2Based` : Est un algorithme de placement développé à la base pour le logiciel Gephi et très utilisé, il est axé sur la notion de centralité des noeuds, plus un noeud a de liens plus il sera central dans le graphe.
— `repulsion` : Algorithme simple dans lequel les noeuds émettent des champs de répulsion sur les autres.
— `hierarchicalRepulsion` : Même principe que la `repulsion` mais à utiliser plutôt pour les placements prédéfinis (layout) que nous verrons dans un autre chapitre.

Je vous met un exemple de chaque type de forces avec ses paramètres par défaut appliqués au même graphe pour que vous voyer les (subtiles) différences :

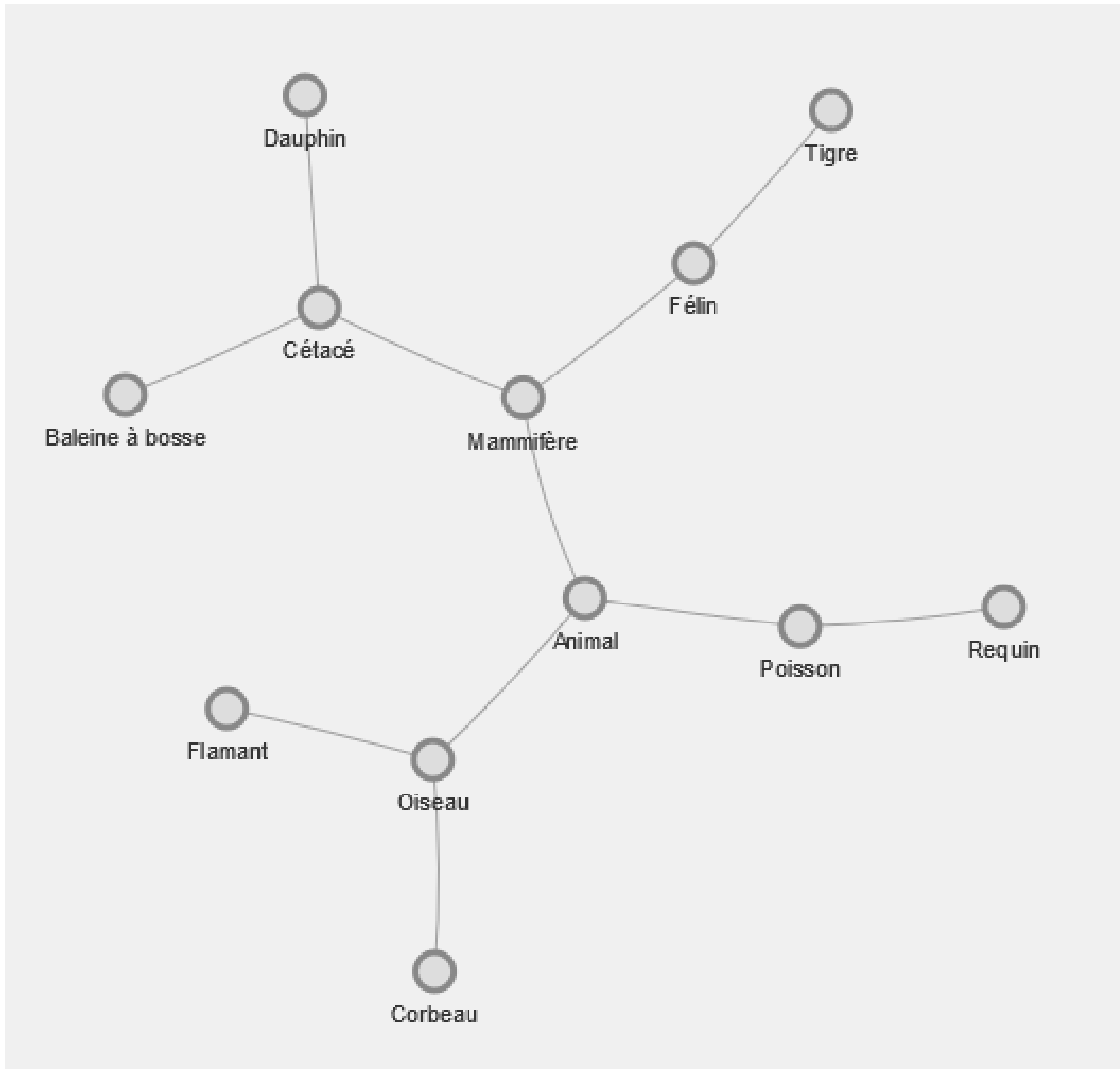


FIGURE 5.1. – barnesHut

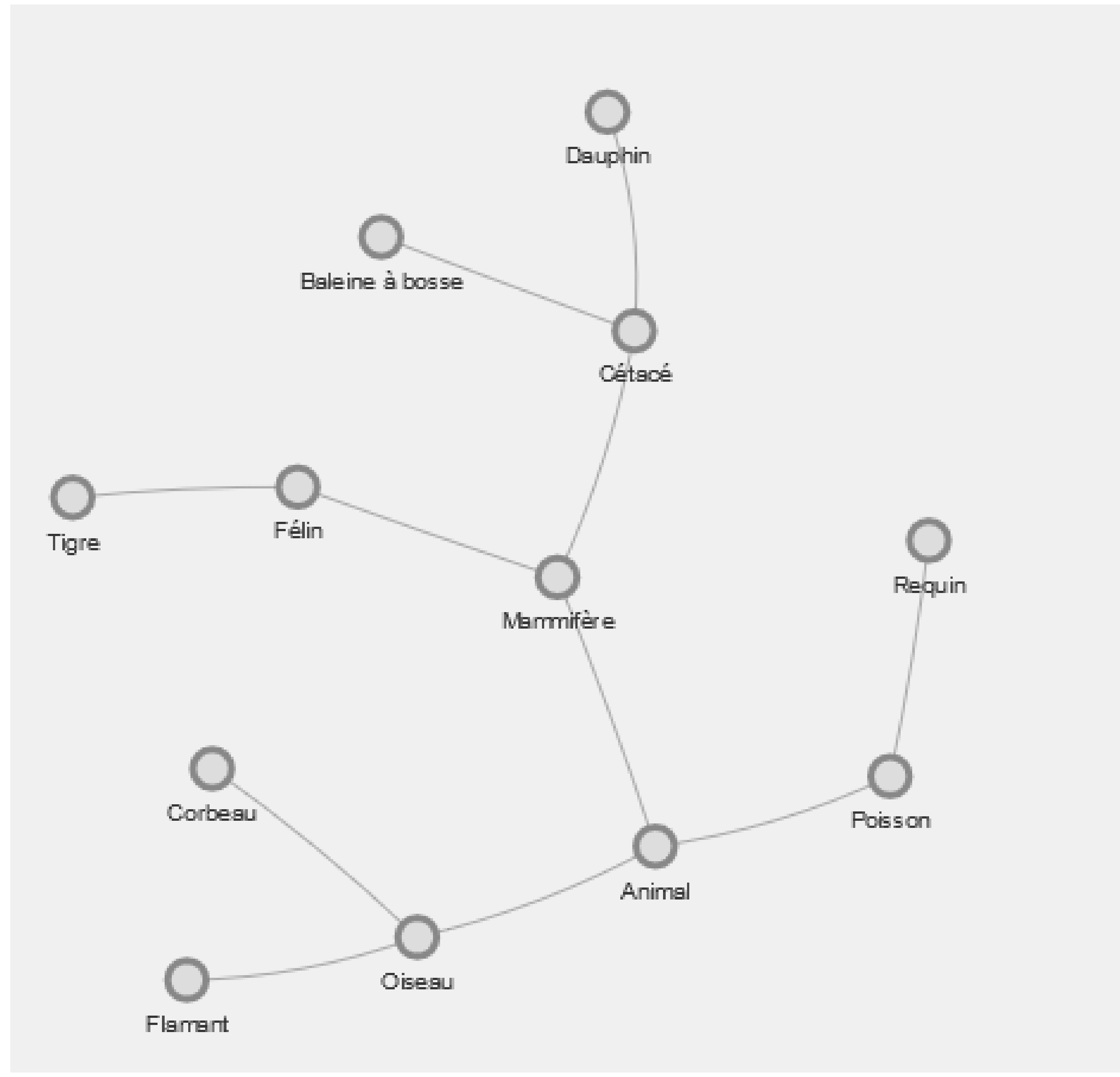


Figure 5.2. – forceAtlas2Based

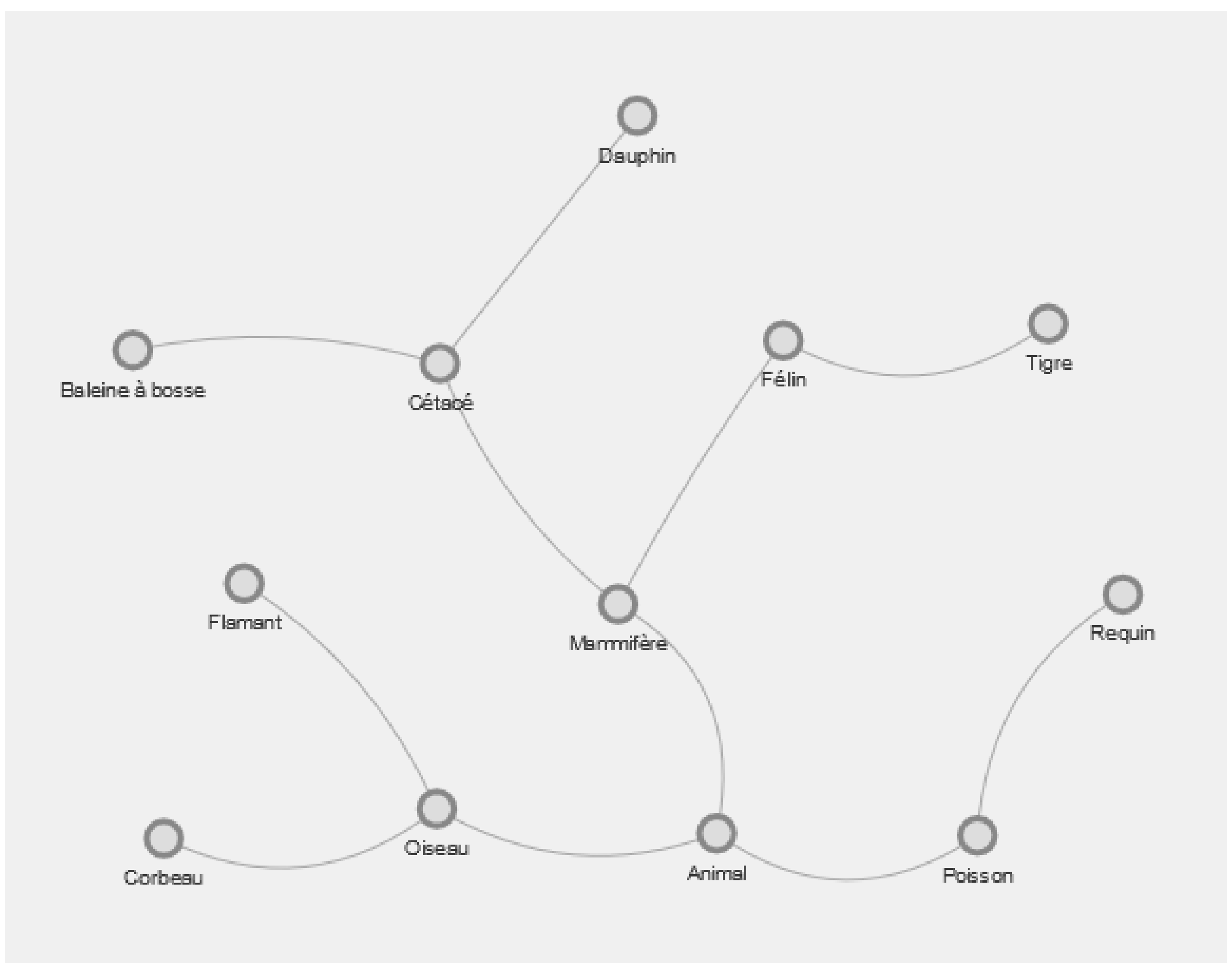


FIGURE 5.3. – repulsion

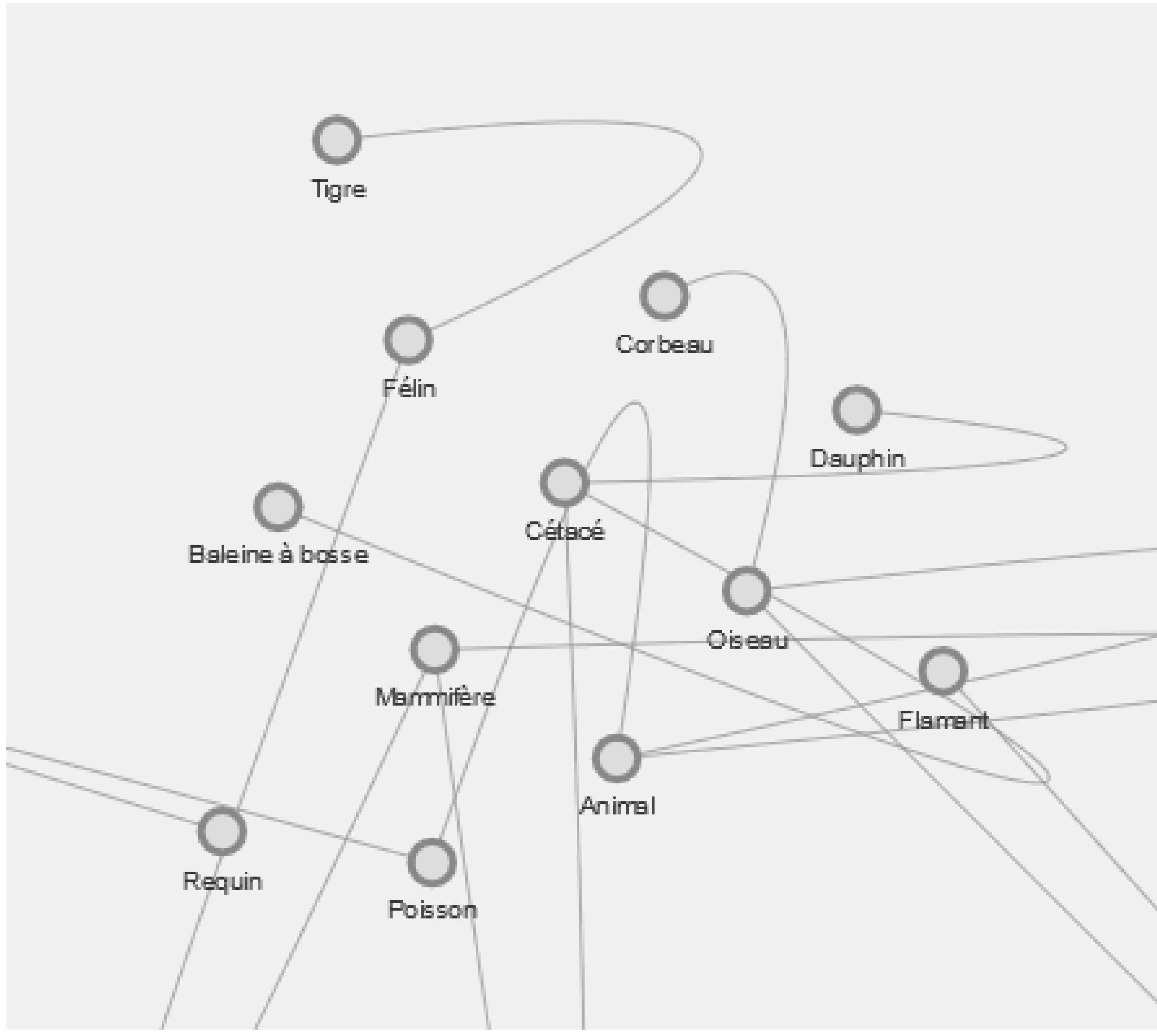


FIGURE 5.4. – hierarchicalRepulsion

Voici le code pour l'algorithme `barnesHut` pour que vous pussiez essayer par vous-même :

Contenu masqué n°6

5.3. Paramétrer une force

Voici les paramètres principaux des forces :

— **`enabled`** : Si à `false` permet de désactiver la physique du graphe, je vous conseille d'essayer pour voir ce que donne un graphe sans force exercée.
— **`solver`** : Définis le type de physique choisie entre les quatre que je vous ai présenté plus haut, par défaut l'algorithme `barnesHut` est utilisé.
— **`barnesHut`** : Choix des paramètres des forces si le solver `barnesHut` a été choisi.
— **`forceAtlas2Based`** : Choix des paramètres des forces si le solver `forceAtlas2Based` a été choisi.

— **`repulsion`** : Choix des paramètres des forces si le solver `repulsion` a été choisi.
— **`hierarchicalRepulsion`** : Choix des paramètres des forces si le solver `hierarchical Repulsion` a été choisi.

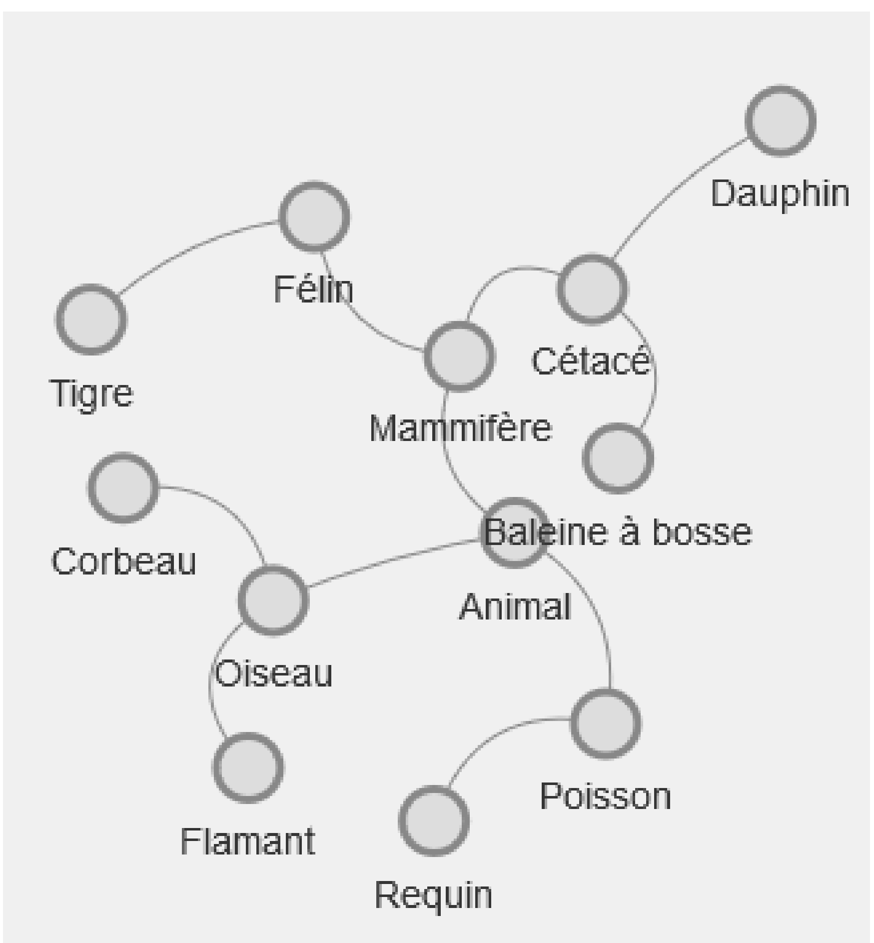
Voici un exemple de paramétrage de la force `barnesHut` avec ses paramètres par défaut :

```
physics:{
  enabled: true,
  solver: 'barnesHut',
  barnesHut : {
    gravitationalConstant : -2000,
    centralGravity : 0.3,
    springLength : 95,
    springConstant : 0.09
  }
}
```

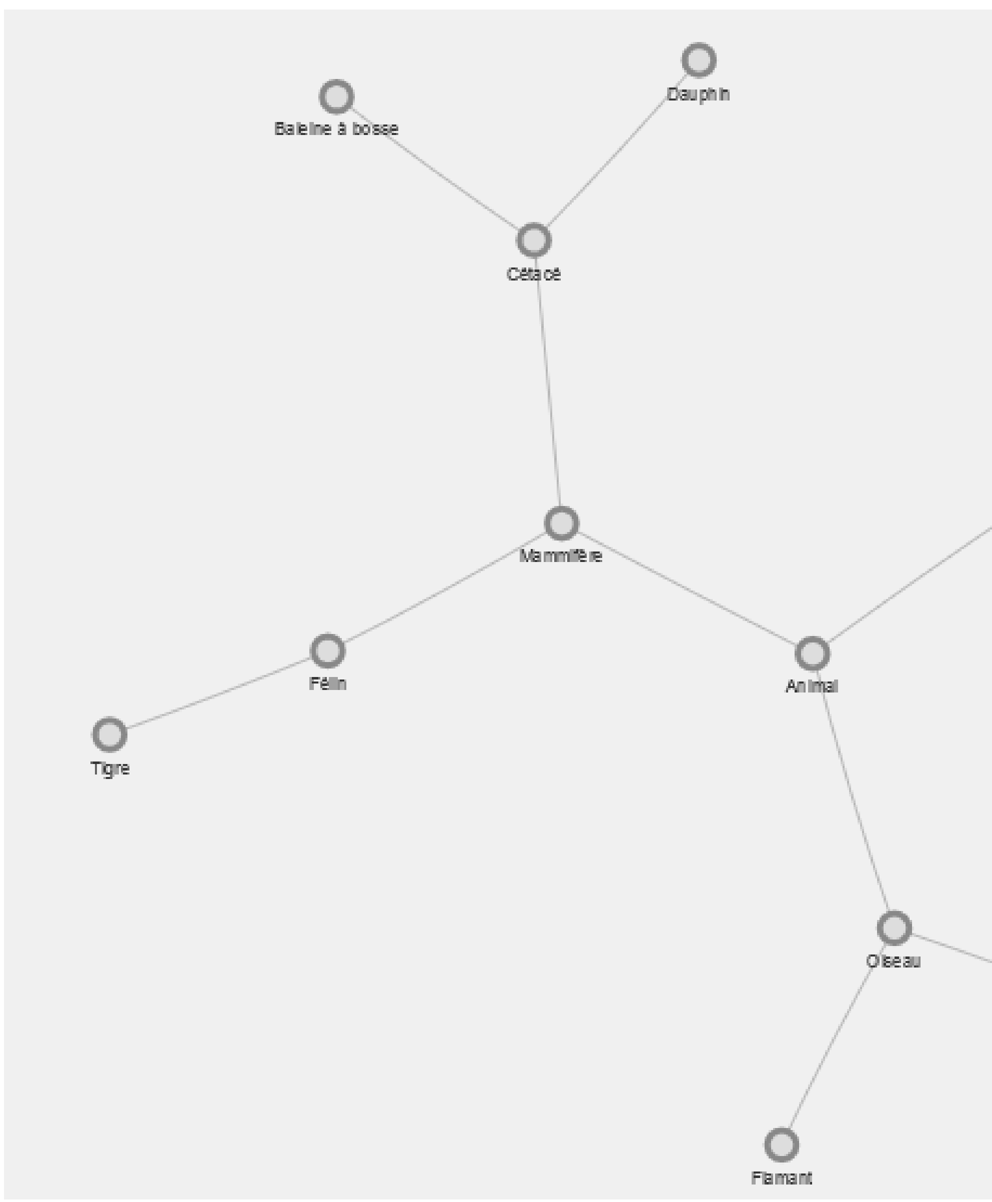
5.4. Exemple avec barnesHut

Je vais vous présenter les principaux paramètres de l'algorithme `barnesHut`, pour que vous voyer concrètement ce qu'apporte le paramétrage des forces.

`barnesHut.gravitationalConstant`: Variable d'attractivité, doit être négatif. Plus la variable est basse plus les noeuds liées sont éloignés les un des autres. Valeur par défaut : `-2000`

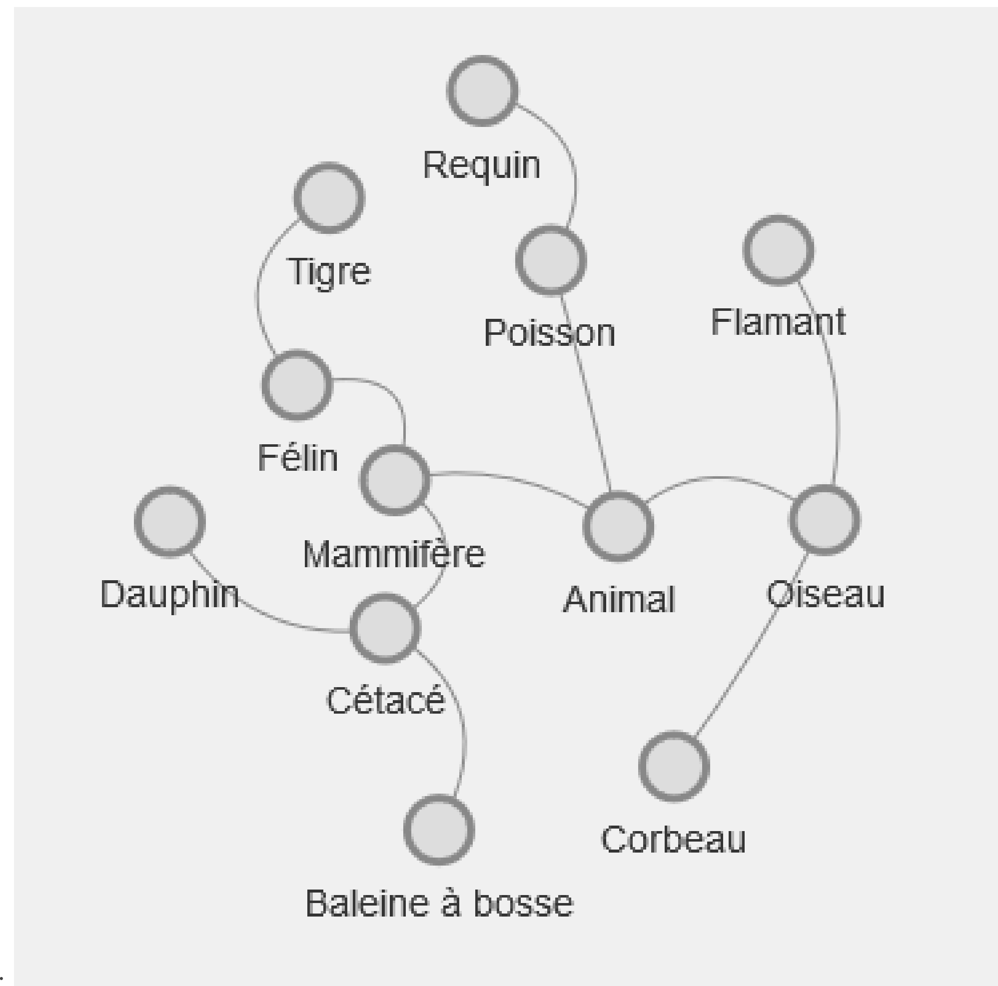
gravitationalConstant à `-300` : 

gravitationalConstant à `-10000` :

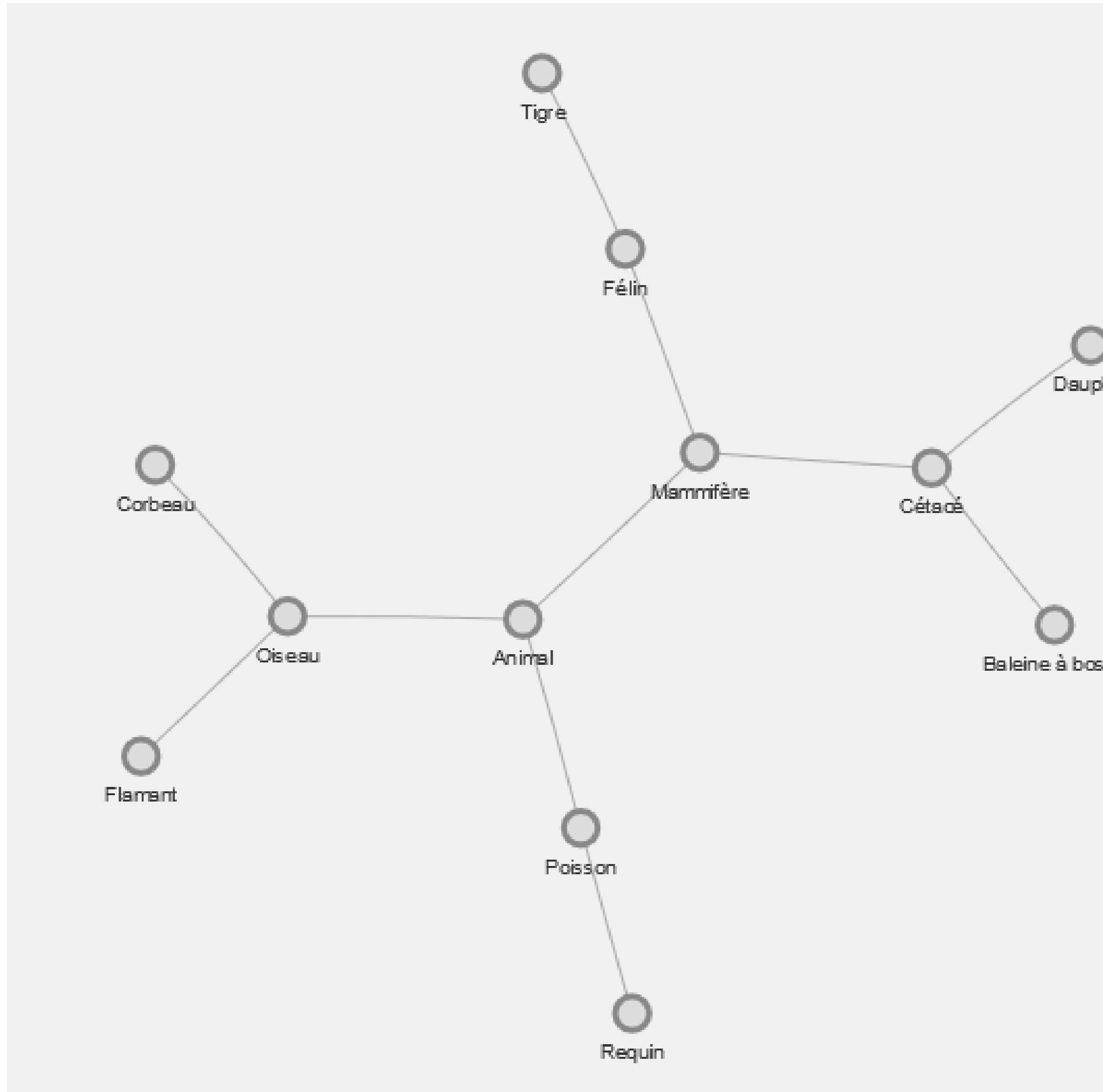


`barnesHut.centralGravity`: Gravité vers le centre, plus la valeur est haute plus les noeuds du graphe sont attirés vers le centre. Valeur par défaut : `0.3`

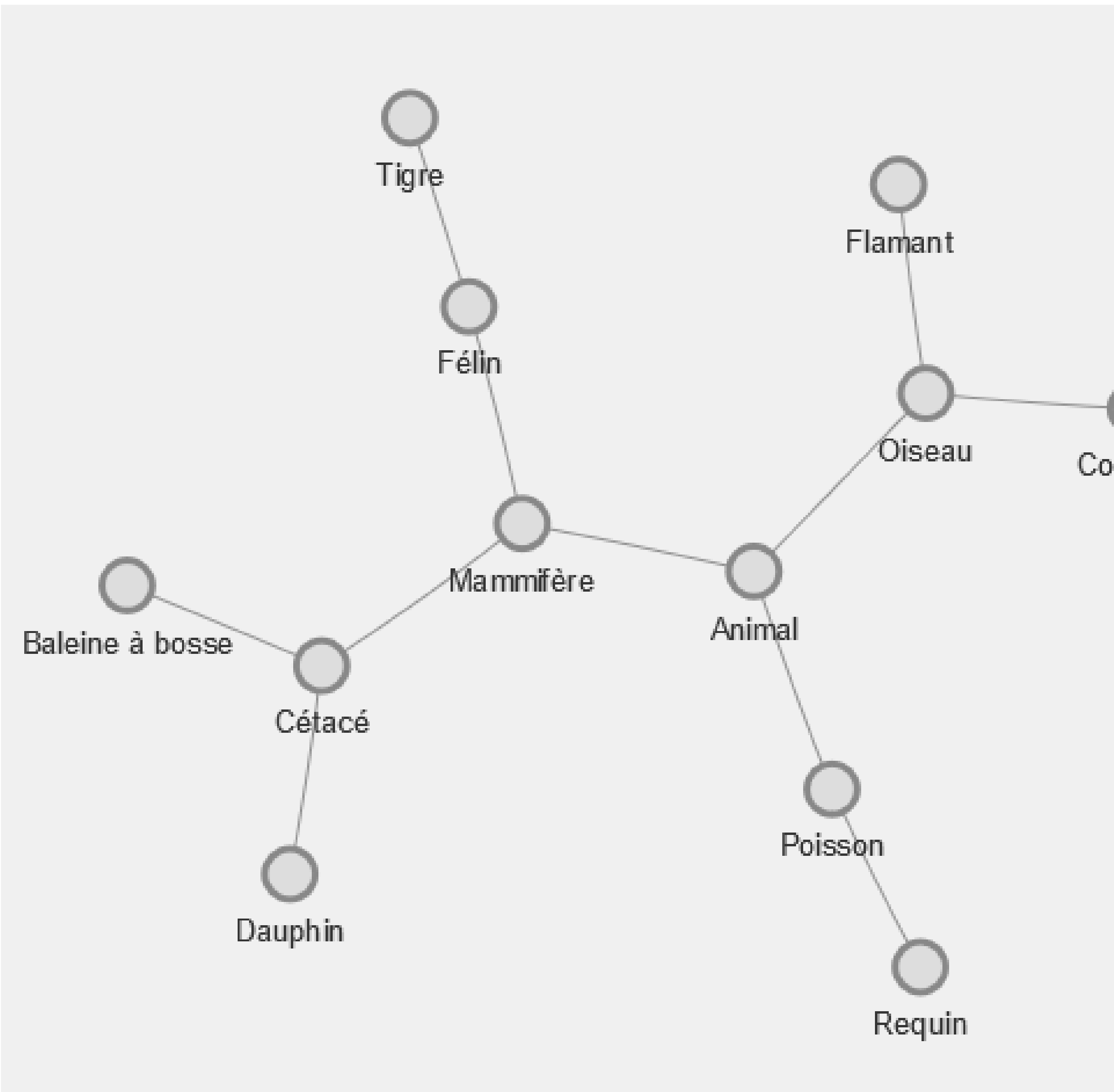
centralGravity à `2` :



centralGravity à `0.05` :



`barnesHut.springLength`: Taille des ressorts des liens, une valeur élevée donnera des liens distendus et une valeur basse des liens très tendus. Valeur par défaut : `95`



springLength à `20` :

springLength à `200` :

`barnesHut.springConstant` : Modifie la robustesse des liens, plus la valeur est élevé, moins les liens sont élastiques. Valeur par défaut : `0.09`.

Lorsque springConstant est à `0.5`, lorsque vous déplacer un noeud vous verrez que tous les autres noeuds se déplacent instantanément.

!(https://jsfiddle.net/shevek/j9ro2eup/4/)

Lorsque springConstant est à `0.005`, les autres noeuds réagissent beaucoup moins à vos déplacements.

!(https://jsfiddle.net/shevek/j9ro2eup/6/)

5.5. Gestion de la masse des noeuds

Un paramètre utile lors de la gestion des forces est le paramètre de **mass** que possède chaque noeuds, par défaut sa valeur est de 1 plus on augmente sa valeur, plus le noeuds va être éloigné dans autre noeuds (il augmente la répulsion appliquée à ses noeuds liés).

Voici un exemple de création de noeuds avec un paramétrage des masses :

```
nodes.add(
  [
    {
      id: "Animal",
      label: "Animal",
    },
    {
      id: "Mammifère",
      label: "Mammifère",
      mass : 10
    },
    {
      id: "Cétacé",
      label: "Cétacé",
      mass : 15
    },
    {
      id: "Baleine à bosse",
      label: "Baleine à bosse"
    },
    {
      id: "Dauphin",
      label: "Dauphin"
    },
    {
      id: "Félin",
      label: "Félin"
    },
    {
      id: "Tigre",
      label: "Tigre"
    },
    {
      id: "Oiseau",
      label: "Oiseau"
    },
    {
      id: "Flamant",
      label: "Flamant"
    },
    {
```

```
42        id: "Corbeau",
43        label: "Corbeau"
44      },
45      {
46        id: "Poisson",
47        label: "Poisson"
48      },
49      {
50        id: "Requin",
51        label: "Requin"
52      }
53    ]
54  );
```

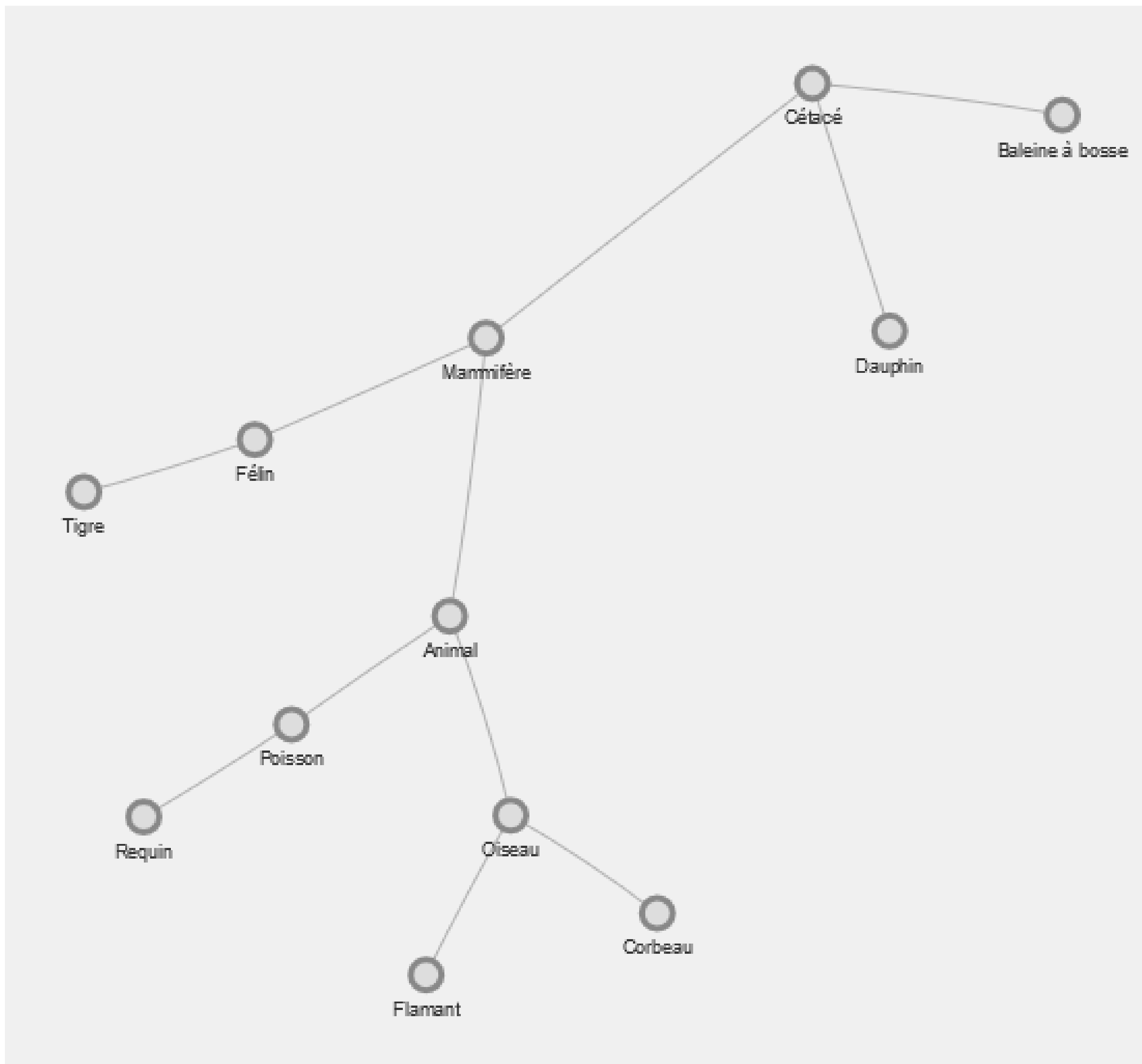


FIGURE 5.5. – Vous voyez que le noeud des Mammifères est éloigné que les autres et celui des Cétacés est encore plus éloigné

Je ne mets exceptionnellement pas d'exercice pour ce chapitre, si vous souhaitez expérimenter les paramétrages des autres types de forces, je vous mets là encore un lien vers la doc qui est très complète et je vous propose cet exemple disponible en ligne qui permet d'expérimenter tous les paramétrages des forces en temps réel .

Dans le prochain nous rendrons notre graphe interactif grâce aux évènements.

Contenu masqué

Contenu masqué n°6

```
let nodes = new vis.DataSet([]);
let edges = new vis.DataSet([]);

let container = document.getElementById('graphe');

let data = {
  nodes: nodes,
  edges: edges
};
let options = {
    nodes : {
      shape: "dot",
      color:
      {
        background: '#dddddd',    // Fond de couleur gris claire
        border: '#888888'         // Bordure de couleur gris foncé
      },
      borderWidth:3,              // Bordure de taille 3
      size : 12                   // Taille de 12
    },
    edges : {},
    physics:{
      enabled: true,              // Rend actif la physique
      solver: 'barnesHut'         // Type de force 'barnesHut'
    },
};

// Initialisation de l'instance de VisNetwork
let network = new vis.Network(container, data, options);

// Ajout des noeuds
```

```
32 nodes.add(
33   [
34     {
35       id: "Animal",
36       label: "Animal"
37     },
38     {
39       id: "Mammifère",
40       label: "Mammifère"
41     },
42     {
43       id: "Cétacé",
44       label: "Cétacé"
45     },
46     {
47       id: "Baleine à bosse",
48       label: "Baleine à bosse"
49     },
50     {
51       id: "Dauphin",
52       label: "Dauphin"
53     },
54     {
55       id: "Félin",
56       label: "Félin"
57     },
58     {
59       id: "Tigre",
60       label: "Tigre"
61     },
62     {
63       id: "Oiseau",
64       label: "Oiseau"
65     },
66     {
67       id: "Flamant",
68       label: "Flamant"
69     },
70     {
71       id: "Corbeau",
72       label: "Corbeau"
73     },
74     {
75       id: "Poisson",
76       label: "Poisson"
77     },
78     {
79       id: "Requin",
80       label: "Requin"
81     }
```

```
  ]
);

// Ajout des liens
edges.add(
  [
    {
      from: "Animal",
      to: "Mammifère"
    },
    {
      from: "Mammifère",
      to: "Cétacé"
    },
    {
      from: "Mammifère",
      to: "Félin"
    },
    {
      from: "Cétacé",
      to: "Baleine à bosse"
    },
    {
      from: "Cétacé",
      to: "Dauphin"
    },
    {
      from: "Félin",
      to: "Tigre"
    },
    {
      from: "Animal",
      to: "Oiseau"
    },
    {
      from: "Oiseau",
      to: "Flamant"
    },
    {
      from: "Oiseau",
      to: "Corbeau"
    },
    {
      from: "Animal",
      to: "Poisson"
    },
    {
      from: "Poisson",
      to: "Requin"
    }
```

```
132     ]
133 );
```

Retourner au texte.

6. La gestion des évènements

Dans cette partie nous allons parler du paramétrage des évènements et des interactions avec le graphe. L'avantage d'avoir un graphe dynamique en javascript est bien de pouvoir interagir avec lui, afin d'afficher des informations supplémentaires, de rediriger l'utilisateur vers une autre page et plein d'autre choses 🍊

6.1. La capture des évènements

Il est très simple d'écouter les évènements de VisNetwork, pour cela il suffit d'ajouter des écouteurs à notre réseau comme ceci :

```
network.on("nomEvenement", mafunction);
```

Il existe un large choix d'évènements pour visnetwork, voici une petite liste de ceux que j'estime intéressants, vous pouvez retrouver les autres dans la doc :

- `click` : Clic gauche de la souris sur la zone d'affichage du graphe
- `doubleClick` : Double clic sur la zone d'affichage du graphe.
- `oncontext` : Clic droit de la souris sur la zone d'affichage du graphe.
- `dragStart` : Évènement déclenché lors du déclenchement du déplacement d'un noeud.
- `dragging` : Évènement déclenché lors du déplacement d'un noeud.
- `dragEnd` : Évènement déclenché lors de l'arrêt du déplacement d'un noeud.
- `zoom` : Déclenché lors de l'action de zoom et de dezoom (molette de la souris).
- `selectNode` : Déclenché lors de la sélection d'un noeud.
- `selectEdge` : Déclenché lors de la sélection d'un lien.
- `deselectNode` : Déclenché lors de la dé-sélection d'un noeud.
- `deselectEdge` : Déclenché lors de la dé-sélection d'un lien.

6.2. Clic et double-clic

Avant de commencer la présentation des exemples, je vous fournis le code JavaScript utilisé afin que vous puissiez reproduire les exemples chez vous.

```
let nodes = new vis.DataSet([]);
let edges = new vis.DataSet([]);
```

```
3
4 let container = document.getElementById('graphe');
5
6 let data = {
7   nodes: nodes,
8   edges: edges
9 };
10 let options = {
11         interaction:{hover:true}
12 };
13
14 // Initialisation de l'instance de VisNetwork
15 let network = new vis.Network(container, data, options);
16
17 // Ajout de nos deux noeuds
18 nodes.add(
19         [
20                 {
21                         id: 1,
22                         label: "N1"
23                 },
24                 {
25                         id: 2,
26                         label: "N2"
27                 }
28         ]
29 );
30
31 // Ajout du lien entre nos deux noeuds
32 edges.add(
33         [
34                 {
35                         from: 1,
36                         to: 2,
37                         label: "E1",
38                 }
39         ]
40 );
```

Nous allons écouter l'action de clic du notre graphe, elle se déclenche lorsque l'utilisateur clique sur la zone d'affichage du graphe.

```
1 network.on("click", function(data)
2 {
3   if(data.nodes.length > 0)
4   {
5         alert(data.nodes[0]);
6   }
```

```
7 });
```

Le paramètre `data` contient un tableau de tout les identifiants des noeuds et un tableau de tout les identifiants des liens dans la zone du clic. Dans notre exemple on vérifie qu'il y a bien un noeud dans la zone du clic et l'on affiche l'identifiant de ce noeud dans un pop-up.

Voici un second exemple pour vous montrer comment récupérer les labels des noeuds et des liens sélectionnés.

```
1  network.on("doubleClick", function(data)
2  {
3    if(data.nodes.length > 0)
4    {
5          var selectedNode = nodes.get(data.nodes[0]);
6
7          var label = selectedNode.label;
8
9          alert(label);
10   }
11   else if(data.edges.length > 0)
12   {
13         var selectedEdges = edges.get(data.edges[0]);
14
15         var label = selectedEdges.label;
16
17         alert(label);
18   }
19 });
```

La méthode `nodes.get(id)` nous permet de récupérer toutes les informations du noeud sur lequel on a cliqué, on récupère ensuite le label du noeud ou du lien, puis on l'affiche.

FIGURE 6.1. – Au clic un pop-up avec le label du noeud ou du lien s'affiche à l'écran.

6.3. Zoom/Dézoom

Dans cet exemple on va afficher le niveau de zoom actuel dans une zone de texte.

> Les paramètres de l'action de zoom ne sont pas les mêmes que les autres actions, ces paramètres sont :
> — `direction` : égale à `+` ou `-`.
> — `scale` : échelle actuelle du zoom (Nombre).
> — `pointer` : {`x` : position de la souris en x, `y` : position de la souris en y}.

Voici un exemple d'affichage du zoom actuel à chaque action de zoom ou de dézoom.

```
// Code Javascript
function zoom(data)
{
        document.getElementById("infos").innerHTML = data.scale;
}

network.on("zoom", zoom);
```

Il faut également ajouter une balise HTML qui contiendra nos informations : `<div id="infos"></div>`

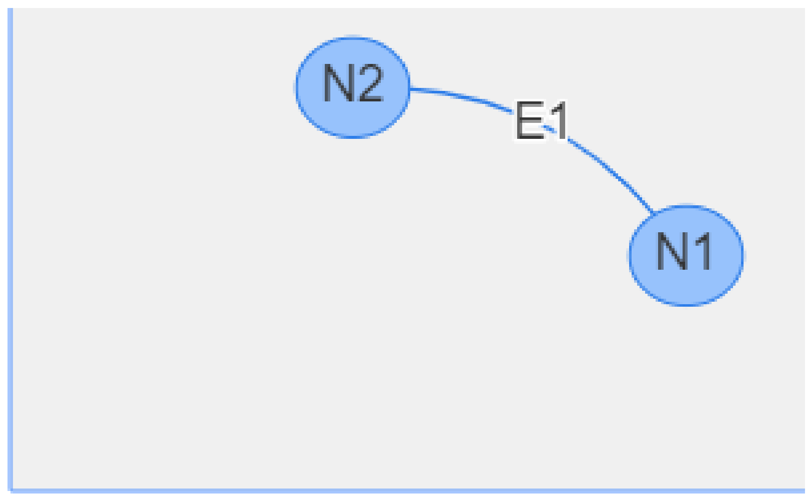


FIGURE 6.2. – Le niveau de zoom est affiché en bas de l'écran

6.4. Exercice

Je vous propose un petit exercice pour mettre en pratique les évènements, à partir du drag and drop (déplacement d'un noeud).

Le but est lors du déplacement d'un noeud de mesurer et d'afficher la valeur de son déplacement, pour cela on va utiliser les évènements `dragStart`, `dragging` et `dragEnd`.

> Petite Astuce : Pour récupérer la position d'un noeud il faut utiliser la méthode `network.getPositions(idNoeud)[idNoeud]`

Procédure à suivre :

- Lors de l'action `dragStart`, stocker la position du noeud.
- Lors de l'action `dragging`, afficher la valeur actuelle du déplacement par rapport à la position initiale du noeud.
- Lors de l'action `dragEnd`, afficher la valeur du déplacement qui vient de se terminer (ex : `Dernier déplacement : X : 215 || Y : 17`).

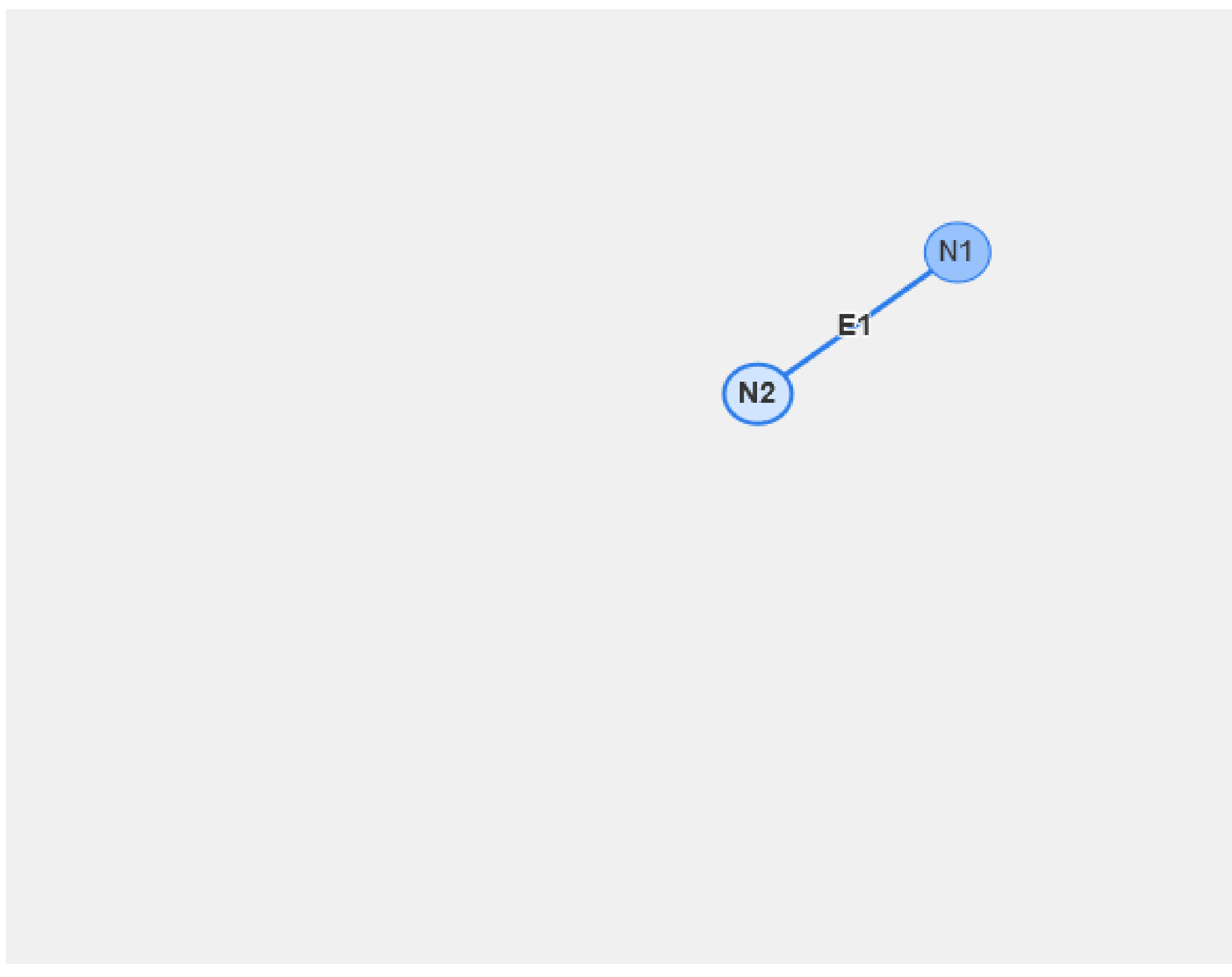


Figure 6.3. – Vous devriez obtenir ce résultat lorsque vous avez terminé un déplacement.

Bon courage 😉

Si vous n'y arrivez pas, vous pouvez retrouver la solution ci dessous :

Contenu masqué n°7

Ce chapitre sur les évènements est maintenant terminé, si vous voulez connaître d'autres évènements rendez vous sur la doc où vous retrouverez une description en anglais de chaque évènements.

Dans la prochaine partie, je vous ferait découvrir les hierarchicals layouts permettant d'ordonnancer les noeuds et le clustering permettant de regrouper les noeuds.

Contenu masqué

Contenu masqué n°7

Code HTML :

```html
<!doctype html>
<html>
 <head>
   <title>Tuto VisNetwork</title>

   <style>
     #graphe
     {
       display: inline-block;
       width: 800px;
       height: 600px;

       background-color: #f0f0f0;
     }
   </style>

   <script type="text/javascript" src="vis.min.js"></script>

 </head>
 <body>

   <div id="graphe"></div>
   <div id="infos"></div>

   <script type="text/javascript" src="graph.js"></script>

 </body>
</html>
```

Code Javascript :

```javascript
let nodes = new vis.DataSet([]);
let edges = new vis.DataSet([]);

let container = document.getElementById('graphe');

let data = {
  nodes: nodes,
  edges: edges
};
```

```
10 let options = {
11         interaction:{hover:true}
12 };
13
14 // Initialisation de l'instance de VisNetwork
15 let network = new vis.Network(container, data, options);
16
17 // Ajout de nos deux noeuds
18 nodes.add(
19         [
20                 {
21                         id: 1,
22                         label: "N1"
23                 },
24                 {
25                         id: 2,
26                         label: "N2"
27                 }
28         ]
29 );
30
31 // Ajout du lien entre nos deux noeuds
32 edges.add(
33         [
34                 {
35                         from: 1,
36                         to: 2,
37                         label: "E1",
38                 }
39         ]
40 );
41
42 // Stockage de la position de départ
43 let startPos = {};
44
45 // Fonction appelé lors du lancement du déplacement
46 function dragStart(data)
47 {
48         if(data.nodes.length > 0)
49         {
50                 // Récupération des informations du noeud en cours
                       de déplacement
51                 let selectedNode = nodes.get(data.nodes[0]);
52
53                 // Mise à jour de la variable stockant la position
                       de départ
54                 startPos =
                       network.getPositions(data.nodes[0])[data.nodes[0]];
55         }
56 }
```

```
57
58 // Fonction s'exécutant en continu au cours du déplacement
59 function dragging(data)
60 {
61         let text = "";
62 
63         if(data.nodes.length > 0)
64         {
65                 // Récupération des informations du noeud en cours
                       de déplacement
66                 let selectedNode = nodes.get(data.nodes[0]);
67 
68                 // Récupération de la position du noeud
69                 let pos =
                       network.getPositions(data.nodes[0])[data.nodes[0]];
70 
71                 // Calcul de la différence de position entre la
                       position actuel du noeud et sa position au
                       debut de déplacement
72                 let posDiffX = pos.x - startPos.x;
73                 let posDiffY = pos.y - startPos.y;
74 
75                 // Affichage du différenciel de position
76                 text += "X:" + posDiffX + " || Y:" + posDiffY;
77         }
78 
79         document.getElementById("infos").innerHTML = text;
80 }
81 
82 // Fonction s’exécutant à la fin de l'action de déplacement (au
       relâchement de la souris)
83 function dragEnd(data)
84 {
85         let text = "";
86 
87         if(data.nodes.length > 0)
88         {
89                 // Récupération des informations du noeud
90                 let selectedNode = nodes.get(data.nodes[0]);
91 
92                 // Récupération de la position du noeud
93                 let endPos =
                       network.getPositions(data.nodes[0])[data.nodes[0]];
94 
95                 // Calcul de la différence de position entre la
                       position actuel du noeud et sa position au
                       début de déplacement
96                 let posDiffX = endPos.x - startPos.x;
97                 let posDiffY = endPos.y - startPos.y;
98 
```

```
99                 // Affichage du différentiel de position
100                text += "Dernier déplacement : X:" + posDiffX +
                       " || Y:" + posDiffY;
101        }
102
103        document.getElementById("infos").innerHTML = text;
104 }
105
106 // Déclenchement des évènements
107 network.on("dragStart", dragStart);
108 network.on("dragging", dragging);
109 network.on("dragEnd", dragEnd);
```

Retourner au texte.

7. Placements hiérarchiques et clusters

Dans ce dernier chapitre je vais vous présenter deux fonctionnalités que je trouve intéressantes

- Les `hierarchicals layouts` qui permettent de créer des graphes de manière ordonnée.
- Les `clusters` qui permettent de créer des groupes de noeuds.

7.1. Les placements hiérarchiques

Les herarchicals layouts qu'on peut traduire par placements hiérarchiques permettent d'ordonner des noeuds de droite à gauche ou de haut en bas.

Cela permet de gérer pas mal de représentations différentes par exemple un arbre généalogique, une hiérarchie sociale ou encore une liste d'instructions.

Pour les activer il faut personnaliser les options `layout` de notre graphe, voici les plus importantes :

- `hierarchical.enabled` : Définit si l'on utilise ou pas les placements hiérarchiques.
- `hierarchical.direction` : Sens dans lequel est ordonnée le graphe.
  - `UD` : De haut en bas (up-down).
  - `DU` : De bas en haut (down-up).
  - `LR` : De gauche à droite (left-right).
  - `RL` : De droite à gauche (right-left).
- `hierarchical.sortMethod` : Méthode de tri du graphe.
  - `hubsize` : Met en premier le noeud avec le plus de lien.
  - `directed` : Ordonné selon l'ordre des noeuds (A->B = A puis B)

Il y a également beaucoup de paramètres qui permettent de gérer l'espacement entre les noeuds à découvrir dans la doc .

> On peut définir la propriété `level` des noeuds par un nombre qui déterminera son niveau de placement dans la hiérarchie. L'ajout du level est optionnel, s'il n'y a pas de level de déterminé, c'est la propriété `directed` qui déterminera la manière dont les noeuds seront placés..

Voici un exemple de création d'un graphe positionné de gauche à droite :

```
let nodes = new vis.DataSet([]);
let edges = new vis.DataSet([]);
```

```

let container = document.getElementById('graphe');

let data = {
  nodes: nodes,
  edges: edges
};
let options = {
        layout: {
                hierarchical : {
                        enabled: true,
                        direction : 'LR',
                        sortMethod : 'hubsize'
                }
        }

};

let network = new vis.Network(container, data, options);

nodes.add(
    [
        {
            id: 1,
            label: "A",
                        level: 1
        },
        {
            id: 2,
            label: "B",
                        level: 2
        },
        {
            id: 3,
            label: "C",
                        level: 2
        },
        {
            id: 4,
            label: "D",
                        level: 3
        },
        {
            id: 5,
            label: "E",
                        level: 3
        }
    ]
);

```

```
53 edges.add(
54     [
55         {
56             from: 1,
57             to: 2
58         },
59         {
60             from: 1,
61             to: 3
62         },
63         {
64             from: 2,
65             to: 4
66         },
67         {
68             from: 2,
69             to: 5
70         },
71         {
72             from: 3,
73             to: 5
74         }
75     ]
76 );
```

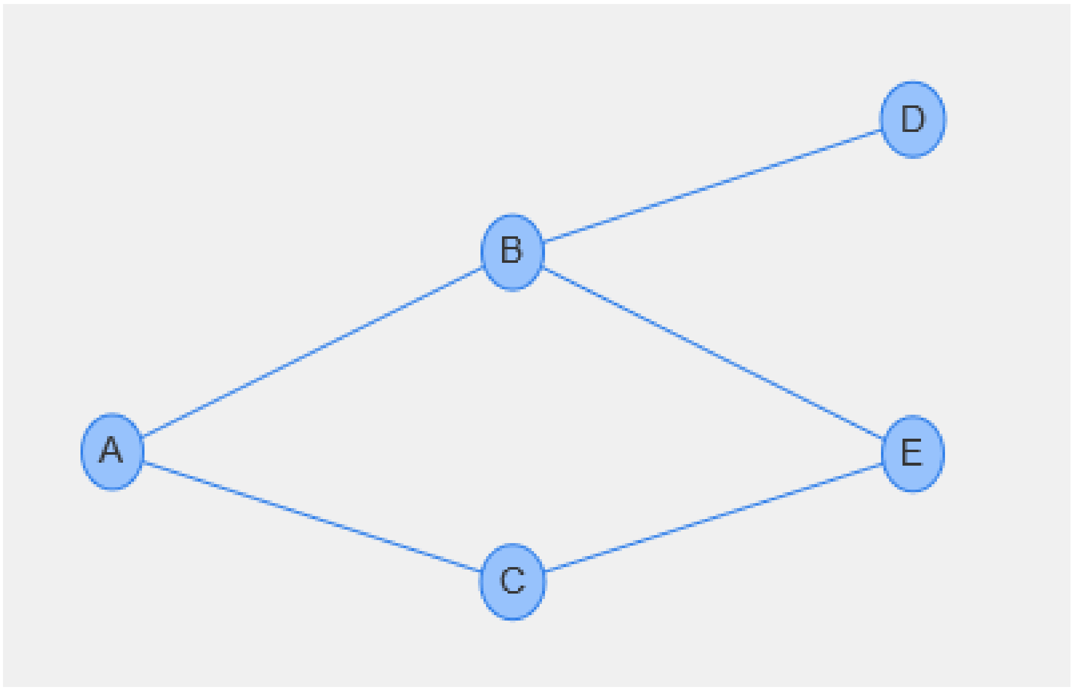


FIGURE 7.1. – Notre premier graphe hiérarchique répartit en 3 niveaux

7.2. Les clusters

Les clusters sont des groupes de noeuds, ils permettent de regrouper plusieurs noeuds en un seul à partir de leurs propriétés, par exemple tous les noeuds de couleur verte réunis en un seul.

Pour faire un cluster on utilise la méthode `cluster()` de notre network, cette méthode prend en paramètre un objet contenant :

- `joinCondition(...)` : Seul paramètre indispensable, cette fonction permet de définir la ou les conditions qui définissent si le noeud sera groupé avec les autres, elle a pour arguments :
  - `nodeOptions` : Propriétés du noeud.
  - *Retour* : Condition de validité du regroupement (voir exemple).
- `processProperties(...)` : Méthode appelé avant l'affichage, elle permet de mettre à jour les propriétés du cluster en fonction des éléments qu'il contient, par exemple la taille du noeud par rapport au nombre de noeuds qu'il contient, cette méthode a pour arguments :
  - `clusterOptions` : Propriétés du cluster.
  - `childNodesOptions` : Propriétés des noeuds contenus dans le cluster.
  - `childEdgesOptions` : Propriétés des liens contenus dans le cluster.
  - *Retour* : Propriétés du cluster mises à jour.
- `clusterNodeProperties` : Objet permettant de personnaliser les options du noeud clusterisé, par exemple pour afficher une couleur ou un label personnalisé.
- `clusterEdgeProperties` : Objet permettant de personnaliser les options des liens du cluster.

Voici un exemple où les noeuds de niveau 2 sont regroupés dans un cluster ayant pour label "Mon cluster".

```
1 let clusterOptionsByData = {
2         joinCondition: function (nodeOptions)
3         {
4                 return nodeOptions.level === 2; // Sélection des
                     noeuds dont le level est 2
5         },
6         clusterNodeProperties: {id: 'cluster', label:'Mon cluster'}
               // Propriété de notre noeud de cluster
7 };
8 network.cluster(clusterOptionsByData);
```

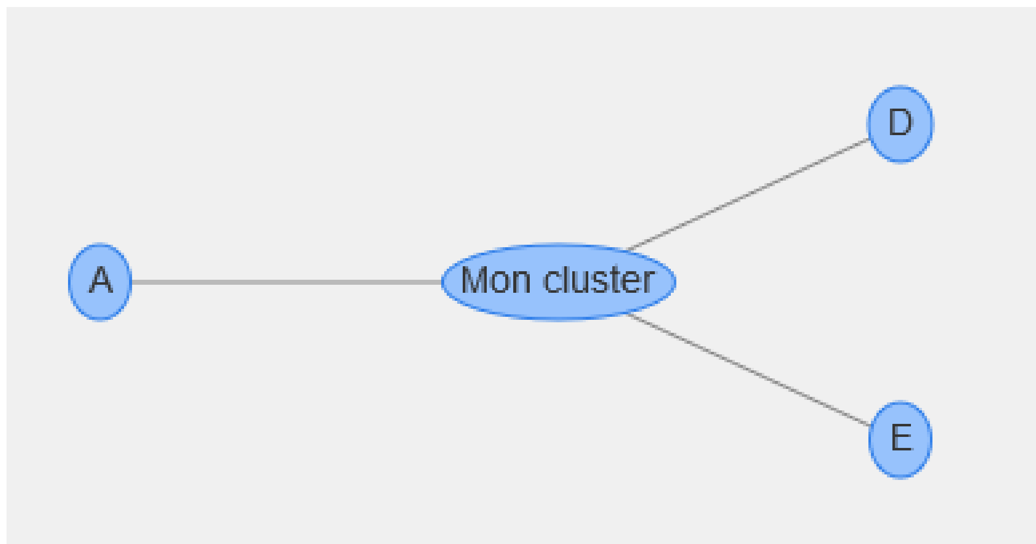


FIGURE 7.2. – Notre premier cluster

Je vous présente un second exemple dans lequel allons reprendre notre exemple précédent mais en y ajoutant deux choses :

— On va gérer la clusterisation et la déclusterisation via des boutons.
— On va afficher le nombre de noeuds contenu dans notre cluster (via la méthode `processProperties`).

FIGURE 7.3. – Nos boutons

Code HTML des boutons :

```
<br/>
<button type="button" onclick="cluster();">Grouper</button>
<button type="button" onclick="uncluster();">Dégrouper</button>
```

Code Javascript :

```
// Action sur le bouton "Grouper", active la clustersation des
    noeuds de niveau 2
function cluster()
{
        let clusterOptionsByData = {
                joinCondition: function (nodeOptions)
                {
```

```
7                       return nodeOptions.level === 2; // Sélection des noeuds dont le level est 2
8               },
9           processProperties: function (clusterOptions, childNodes, childEdges)
10              {
11                      let nbNodes = childNodes.length; // Récupération du nombre de noeuds de notre cluster
12                      clusterOptions.label = 'Mon cluster (' + nbNodes + ')';     // Mise à jour du label de notre cluster
13                      return clusterOptions; // On retourne les options de notre cluster
14              },
15              clusterNodeProperties: {id: 'cluster', label:'Mon cluster'} // Propriétés de notre noeud de cluster
16          };
17          network.cluster(clusterOptionsByData);
18  }
19
20  // Action sur le bouton "Dégrouper", désactive la clustersation, en mettant à jour les données du graphe
21  function uncluster()
22  {
23          network.setData(data);
24  }
```

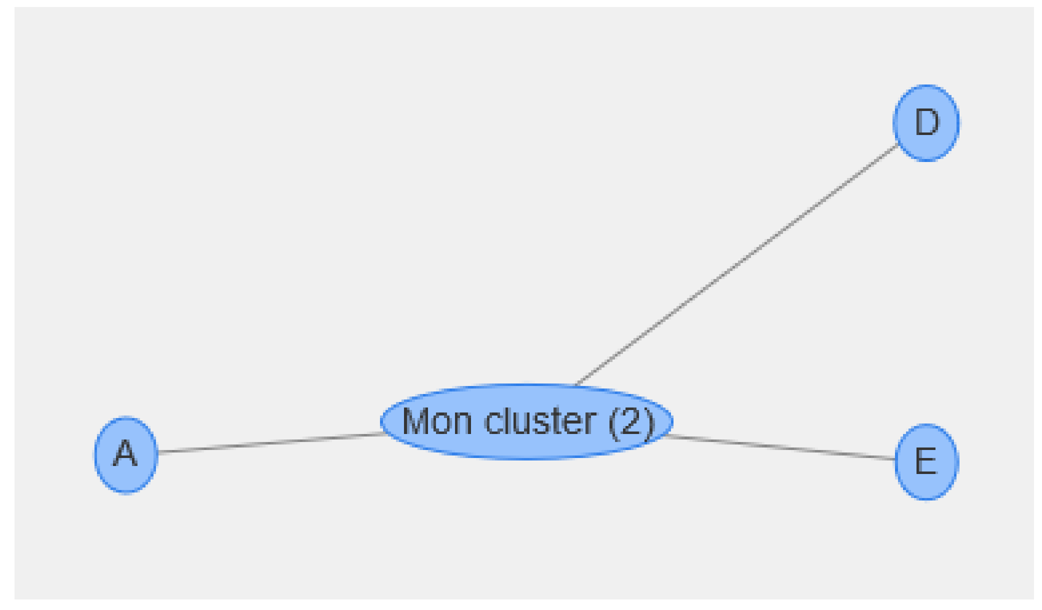


Figure 7.4. – Lors de la clusterisation, on affiche le nombre de noeuds contenu dans notre cluster

Si vous ajoutez d'autres noeuds au niveau 2 de votre graphe (`level:2`) vous verrez que le nombre affiché lors de la clusterisation se mettre à jour selon le nombre de noeuds contenus dans le groupe.

> Il y existe également des types de clusterisation particuliers comme par exemple le `clusterByConnection` qui permet de créer un groupe à partir de tous les noeuds connecté au noeud cible, vous retrouverez comment les utiliser dans la doc .

7.3. Exercice

Nous terminons ce chapitre par le traditionnel exercice qui vous permettra de mettre en pratique les placements hiérarchiques et les clusters.

Dans ce chapitre nous allons créer un graphe généré aléatoirement, le but va être de le trier en le hiérarchisant et en regroupant les noeuds.

Nous allons générer un graphe de 20 noeuds avec 7 couleurs attribuées aléatoirement et 30 liens placés eux aussi aléatoirement entre ces noeuds.

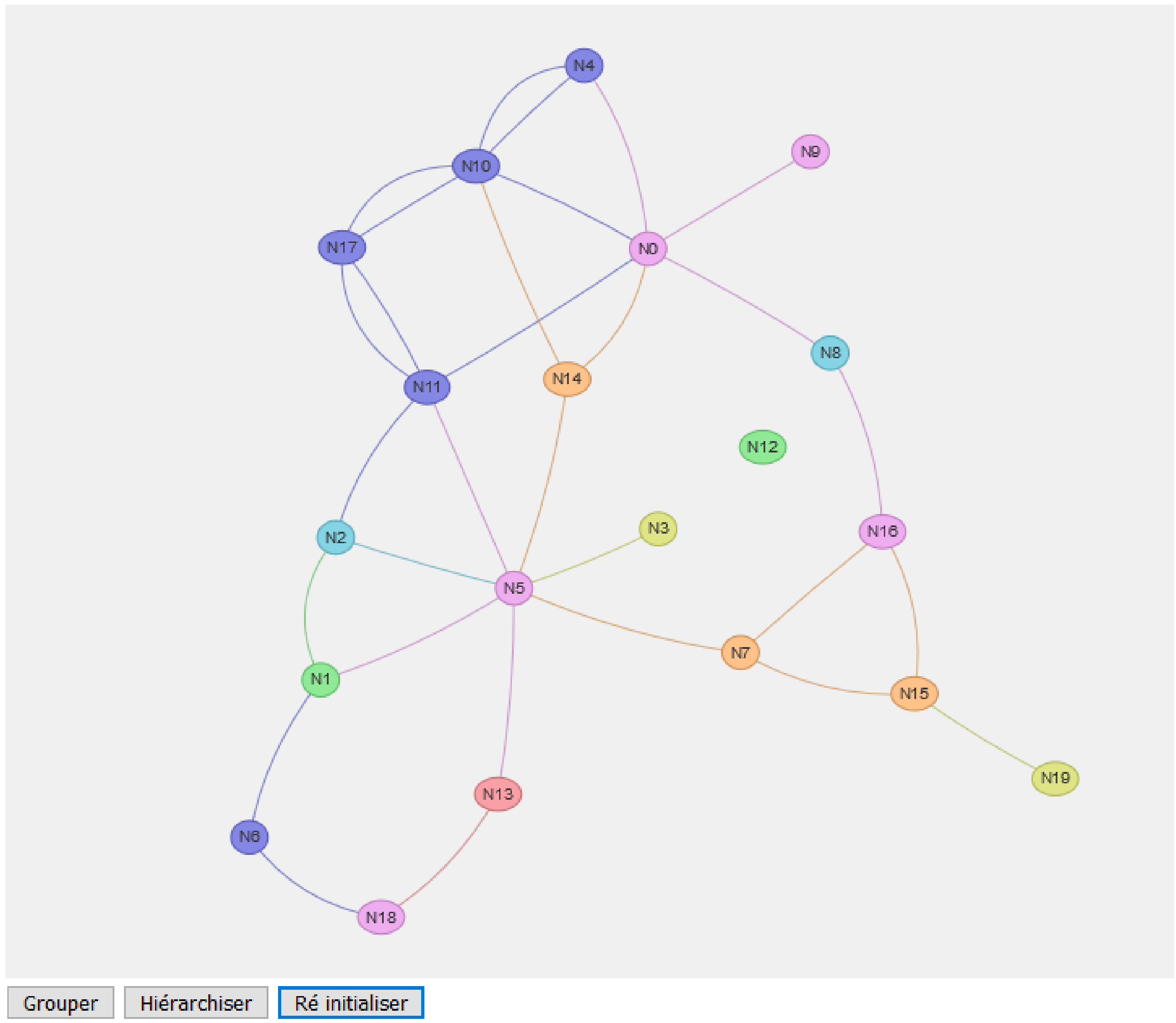


FIGURE 7.5. – Votre graphe devrait ressembler vaguement à celui-ci, si vous rechargez la page il se régenera avec d'autres couleurs et d'autres combinaisons de liens

Objectif

Le but de l'exercice va être de réaliser ces trois boutons :

- **Grouper** : Lance un cluster des noeuds selon leur couleur :
  - Mettre le groupe de noeuds à la couleur cible avec pour label le nom de la couleur.
  - Afficher entre parenthèses le nombre de noeuds du groupe.
  - Modifier la taille des groupes selon leur nombre de noeuds (10 de base avec +3 par noeud regroupé).
- **Hiérarchiser** : Hiérarchise les noeuds selon leur couleur (de haut en bas, avec une ligne par couleur).
- **Ré initialiser** : Ré-initialise le graph à son état initiale.

Je vous donne le code HTML de nos boutons :

```
<button type="button" onclick="grouper();">Grouper</button>
<button type="button"
    onclick="hierarchiser();">Hiérarchiser</button>
<button type="button" onclick="reinit();">Ré initialiser</button>
```

Je vous donne également le code javascript de génération du graph :

```
let nodes = new vis.DataSet([]);
let edges = new vis.DataSet([]);

let container = document.getElementById('graphe');

// Initialisation du graphe
let data = {
  nodes: nodes,
  edges: edges
};
let options = {};
let network = new vis.Network(container, data, options);

// Définition du nombre de noeuds et de liens à générer
let nbNodes = 20;
let nbEdges = 30;

// Définition du tableau des couleurs et des noms de celle-ci (pour
    les labels des groupes)
let colors = [
"#f99fa6",  // Rouge
"#ffc389",  // Orange
"#8fea96",  // Vert
"#8587e5",  // Bleu
"#eeadef",  // Violet
"#dfe585",  // Jaune
"#85d4e5"   // Cyan
];
let colorsName = ["Rouge", "Orange", "Vert", "Bleu", "Violet",
    "Jaune", "Cyan"];

// Génération de nos noeuds
for(let i = 0; i < nbNodes; i++)
{
        // Génération aléatoire de la couleur
        let numColor = Math.floor(Math.random() * (colors.length -
            0) + 0);
        let color = colors[numColor];

        // Création du noeud
```

```
38          nodes.add(
39      [
40          {
41              id: i,
42              label: "N"+i,
43                          color: color
44          }
45          ]);
46  }
47
48  // Génération de nos liens
49  for(let i = 0; i < nbEdges; i++)
50  {
51          let n1 = Math.floor(Math.random() * (nbNodes - 0) + 0); //
                Génération aléatoire de l'id du premier noeud
52          let n2 = Math.floor(Math.random() * (nbNodes - 0) + 0); //
                Génération aléatoire de l'id du second noeud
53
54          if(n1 != n2) // Ne pas prendre en compte les noeuds liés à
                eux-mêmes
55          {
56                  // Ajout du lien entre les deux noeuds
57                  edges.add(
58                  [
59                          {
60                                  from: n1,
61                                  to: n2
62                          }
63                  ]);
64          }
65          else
66          {
67                  i--;
68          }
69  }
```

Lors de la clusterisation, votre graphe doit resembler à ceci :

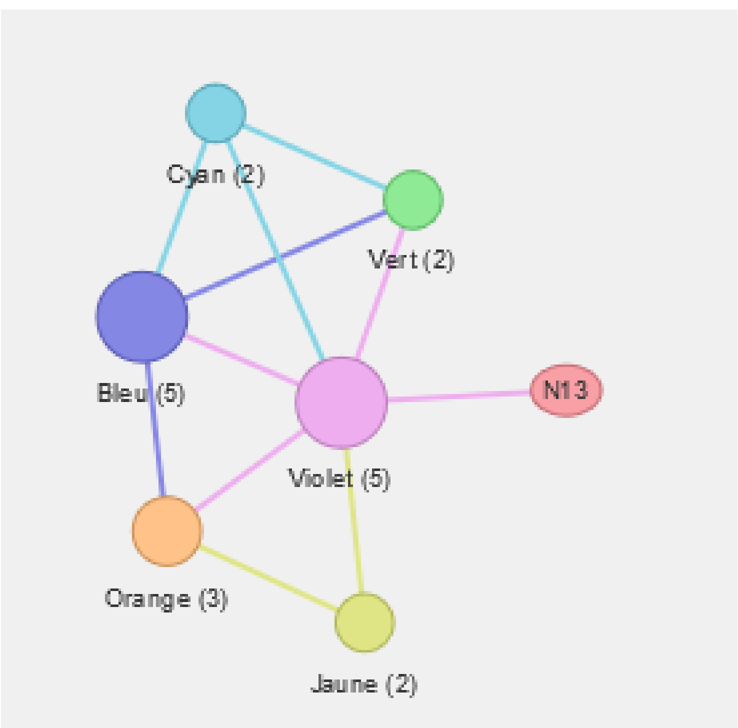


Figure 7.6. – Nos noeuds sont clusterisés par couleur, et sont plus ou moins grands selon leur nombre de noeuds regroupés

Lors de la hiérarchisation,, vous devriez obtenir un résultat qui ressembler à celui-ci :

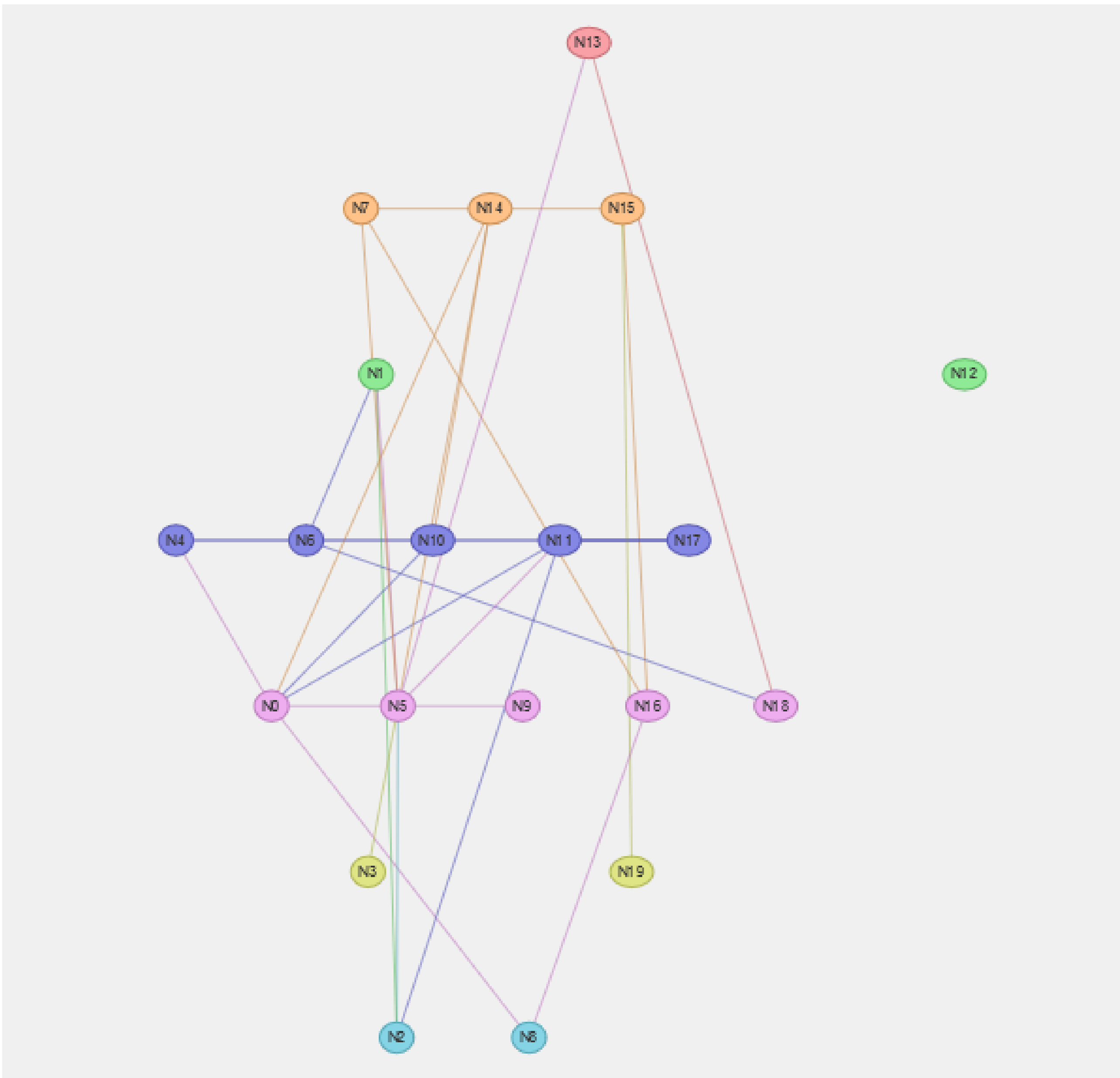


Figure 7.7. – Nos noeuds hiérarchisés par couleurs

Si vous êtes bloqué, voici la solution :

Contenu masqué n°8

Cette dernière partie est terminée, de nouveaux chapitres viendront peut-être rejoindre le tutoriel plus tard.

Je félicite les plus courageux qui ont été jusqu'au bout de ce petit tutoriel, en espérant qu'il vous a plu

---

Si vous souhaitez poursuivre votre apprentissage, rendez vous sur la documentation et sur la liste des exemples de VisNetwork.

Contenu masqué

Contenu masqué n°8

Code HTML

```
<!doctype html>
<html>
        <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8" />
 <head>
   <title>Tuto VisNetwork</title>

   <style>
       #graphe
       {
         display: inline-block;
         width: 800px;
         height: 600px;

         background-color: #f0f0f0;
       }
   </style>

   <script type="text/javascript" src="vis.min.js"></script>
 </head>
 <body>

   <div id="graphe"></div>
        </br>
        <button type="button" onclick="grouper();">Grouper</button>
        <button type="button"
            onclick="hierarchiser();">Hiérarchiser</button>
        <button type="button" onclick="reinit();">Ré
            initialiser</button>

   <script type="text/javascript" src="graph.js"></script>

 </body>
</html>
```

Code Javascript :

```
let nodes = new vis.DataSet([]);
let edges = new vis.DataSet([]);

let container = document.getElementById('graphe');

// Initialisation du graphe
let data = {
 nodes: nodes,
 edges: edges
};
let options = {};
let network = new vis.Network(container, data, options);

// Définition du nombre de noeuds et de liens à générer
let nbNodes = 20;
let nbEdges = 30;

// Définition du tableau des couleurs et des noms de celle-ci (pour
    les labels des groupes)
let colors = [
"#f99fa6",  // Rouge
"#ffc389",  // Orange
"#8fea96",  // Vert
"#8587e5",  // Bleu
"#eeadef",  // Violet
"#dfe585",  // Jaune
"#85d4e5"   // Cyan
];
let colorsName = ["Rouge", "Orange", "Vert", "Bleu", "Violet",
    "Jaune", "Cyan"];

// Génération de nos noeuds
for(let i = 0; i < nbNodes; i++)
{
        // Génération aléatoire de la couleur
        let numColor = Math.floor(Math.random() * (colors.length -
            0) + 0);
        let color = colors[numColor];

        // Création du noeud
        nodes.add(
   [
       {
           id: i,
           label: "N"+i,
                        color: color,
                        level: numColor
       }
        ]);
```

```
47 }
48 
49 // Génération de nos liens
50 for(let i = 0; i < nbEdges; i++)
51 {
52         let n1 = Math.floor(Math.random() * (nbNodes - 0) + 0); //
               Génération aléatoire de l'id du premier noeud
53         let n2 = Math.floor(Math.random() * (nbNodes - 0) + 0); //
               Génération aléatoire de l'id du second noeud
54 
55         if(n1 != n2) // Ne pas prendre en compte les noeuds liés à
               eux-mêmes
56         {
57                 // Ajout du lien entre les deux noeuds
58                 edges.add(
59                 [
60                         {
61                                 from: n1,
62                                 to: n2
63                         }
64                 ]);
65         }
66         else
67         {
68                 i--;
69         }
70 }
71 
72 // Action de "Clustering" des noeuds
73 function grouper()
74 {
75         // Parcours nos couleurs afin de créer un cluster par
               couleur
76         for (let i = 0; i < colors.length; i++)
77         {
78                 let clusterOptionsByData = {
79                         joinCondition: function (nodeOptions)
80                         {
81                                 return nodeOptions.color.background
                                       === colors[i]; // Sélection des
                                       noeuds de la couleur cible
82                         },
83                         processProperties: function
                               (clusterOptions, childNodes,
                               childEdges)
84                         {
85                                 // Mise à jour de la taille et du
                                       label du noeud selon le nombre
                                       d'éléments groupés
86                                 let size = 10;
```

```
 87                                 for (let j = 0; j <
                                        childNodes.length; j++)
 88                                 {
 89                                         size += 3;
 90                                 }
 91                                 clusterOptions.size = size;
 92                                 clusterOptions.label =
                                        colorsName[i] + " (" +
                                        childNodes.length + ")";
 93                                 return clusterOptions;
 94                         },
 95                         clusterNodeProperties:
 96                         {
 97                                 // Définition des propriétés de
                                        notre noeud de cluster
 98                                 id: 'cluster_'+colors[i],
 99                                 shape:'dot',
100                                 color:colors[i],
101                                 level:i
102                         },
103                         clusterEdgeProperties:
104                         {
105                                 // Définition des propriétés des
                                        liens de nos clusters
106                                 color:{
107                                         color : colors[i]
108                                 },
109                                 width : 3
110                         }
111                 };
112                 network.cluster(clusterOptionsByData);
113         }
114 }
115 
116 // Hiérarchisation du graph
117 function hierarchiser()
118 {
119         // Mise à jour des options (activation de la hiérarchie)
120         options.layout = {
121        hierarchical : {
122            enabled: true,
123            direction : 'UD'
124        }
125    }
126 
127         // Ré-initialisation du réseau avec les nouvelles options
128         network = new vis.Network(container, data, options);
129 }
130 
131 // Réinitialisation de graphe
```

```
132 function reinit()
133 {
134         // Mise à jour des options (retrait de la hiérarchie)
135         options.layout = {
136        hierarchical : {
137            enabled: false
138                 }
139         }
140
141         // Ré-initialisation du réseau avec les nouvelles options
142         network = new vis.Network(container, data, options);
143 }
```

Retourner au texte.