UNIVERSITÉ
CAEN
NORMANDIE

Projet Tutoré

Saute Canton

Thibault Poisson
Éric Hu 21713256

M1 Informatique
Année 2020-2021

Contents

1 Introduction

Dans le cadre de notre première année de Master, nous avons choisi pour projet tutoré le projet du saute canton. Nous avons choisi ce projet car il nous semblait être le plus amusant à réaliser.

De plus ses applications peuvent être très concrètes et les informations que nous pouvons extraire de nos tests, très utiles d'un point de vue générale.

En effet, si un jour nous sommes perdus en voiture sans GPS ou que nous souhaitons visiter le plus possible de grandes villes françaises dans un laps de temps réduit, nous pourrons nous remémorer ce projet. Mais encore, il a la particularité de toucher à de nombreux domaines de l'informatique : le web, la programmation, les bases de données, l'analyse...

Le but de ce projet est en premier lieu de réaliser une réplique du jeu du saute canton. Suite à cela nous devions étudier la forme de la carte du jeu obtenu et réalisée une série de tests statistiques afin de déterminer s'il existe une stratégie optimale pour gagner à tous les coups.

Nous allons donc en premiier lieu présenter globalement notre projet, nous parlerons du jeu original puis des caractéristiques globales de notre implémentation. Ensuite, nous aborderons un à un chaque point technique en passant de la requête SPARQL à nos robots qui joue pour nous à notre implémentation du saute canton selon certaines stratégies. Troisièmement nous discuterons d'une extension que nous avons réalisée afin de "tricher" au jeu original. Enfin, nous finirons par une analyse statistique des parties jouées par nos robots. Nous conclurons par la suite, en indiquant notamment comment avoir plus de chances de gagner. Et nous en profiterons pour énoncer différents axes d'amélioration possibles de notre implémentation et ainsi que d'autres axes d'exploration éventuels de notre implémentation.

2 Présentation globale

2.1 Le jeu du saute canton

Mais qu'est-ce que le saute canton ? Saute canton est un jeu web très simple qui a été développé pendant le confinement de mars 2020. Le but de ce jeu est de voyager vers une ville de plus de 30 000, 40 000 ou 50 000 habitants, avec ou sans indice en partant d'une ville choisie aléatoirement. Moins le joueur parcourt de distance entre la ville d'origine et la "grande" ville, plus il gagne de points à l'arrivée. Pour notre implémentation, nous avons choisi de nous passer des distances en kilomètres et de les substituer par des sauts. Un saut correspond à un aller d'une ville A vers une ville B. Cela a pour avantage de simplifier les calculs de distance et de percevoir la carte du jeu d'une façon différente.

L'une des différences principales de notre jeu par rapport à la version originale est que notre jeu ne possède pas d'interface graphique. Elle nous a paru

contingente sachant que seuls des robots y joueront.

2.2 Langage

Pour notre implémentation, nous avons choisi le langage python pour diverses raisons :

- Nous sommes familiers avec les différents aspects du langage.
- il est très utile pour faire du prototypage rapide et de l'analyse de données.
- Python possède donc un bon nombre bibliothèque, qui nous permette d'accélérer les phases de représentation de données et de calculs comme par exemple Numpy.
- Python est fourni aussi des librairies permettant de gérer, créer des bases de données et de faire des requêtes. Ce qui est un point important et essentiel dans la conception de notre implémentation.

2.3 Architecture globale

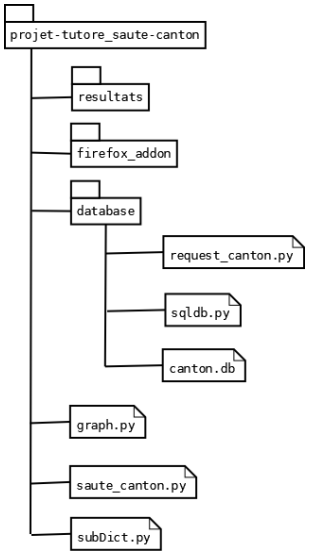

Figure 1: Diagramme des packages

Directement dans notre dossier principal se trouvent les fichiers `graph.py`, `saute_canton.py` et `subDict.py`. Le premier fichier nous permet de construire une représentation de la France, de ses villes et de ses régions. Quant au second, il contient les classes représentant les robots qui vont parcourir la France. Le dernier, `subDict.py` est un utilitaire utilisé pour regrouper les villes et les régions dans un même attribut.

Dans le dossier `resultats` se trouve les résultats, sous format CSV des parcours de la France de nos robots.

Dans le dossier `firefox_addon` se trouve les fichiers nous permettant d'avoir plus d'aide sur le jeu originel.

Enfin, dans `database`, nous avons regroupé les différents fichiers en rapport avec notre base de données `canton.db`. En effet, `request_canton.py` contient les différentes requêtes SPARQL qui nous ont permis de la construire. `sqldb.by` regroupe les différentes requêtes SQL nous permettant de manipuler `canton.db`.

2.4 Diagramme des classes

Nos robots `Voyager` traversant le France à la recherche d'une grande ville implémentent une interface `VoyagerMeta` et utilisent principalement les trois méthodes de cette interface pour pouvoir se mouvoir à travers le graphe représentant la France. Notre classe `FollowADirection` prend un attribut en plus qui est une direction cardinale. Il devra suivre cette direction lors de son parcours du graphe.

Ces `Voyager` prennent en paramètre de `play` un graphe représentant un pays. `Country` représente ce graphe. Il contient deux attributs dont le type est une classe personnalisée de dictionnaire, `subDict`. Ces attributs sont respectivement `cantons` qui est un `Canton` ou une ville et `region` qui est une `Region`.

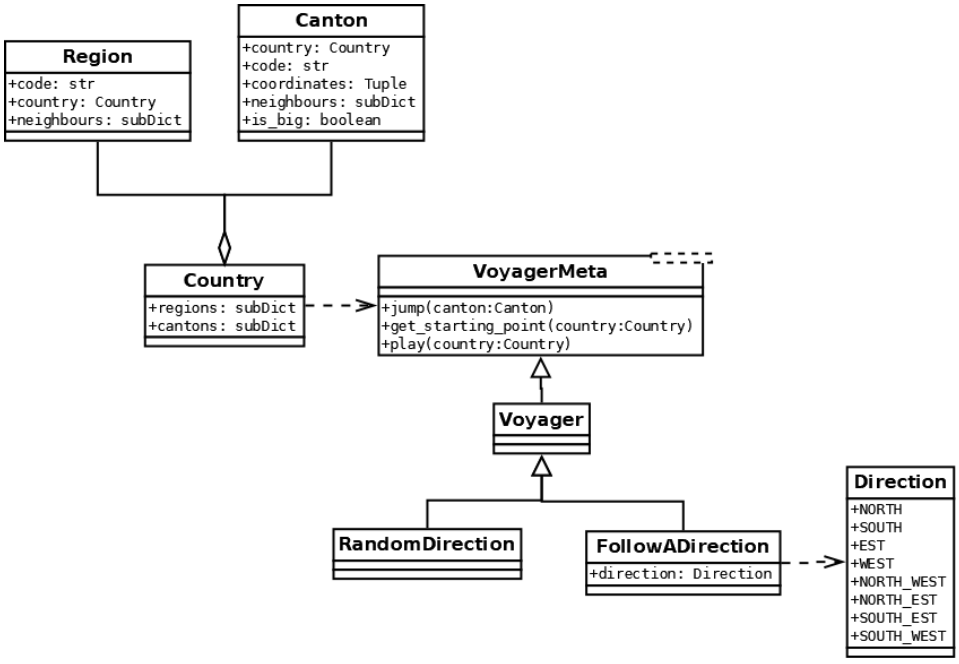

Figure 2: Digramme de classe du saute canton

3 Implémentation

Nous allons maintenant traiter directement des aspects techniques de notre implémentation. Nous commencerons avec SPARQL, un langage de requête qui a été essentiel pour créer la base de données sur laquelle nous avons tout du long travaillé.

3.1 Requêtes SPARQL

SPARQL Protocol and RDF Query Language ou SPARQL, est un langage de requête rendant possible la récupération de données de différents modèles de données à travers le web sémantique. Pour résumer, il permet de récupérer dans des bases de connaissances, des données décrites à travers un certain modèle.

Cela nous a permis d'extraire de wikiData les villes et régions de France ainsi que leurs voisins respectifs. Pour créer notre base de données, nous avons utilisé 3 requêtes : une nous donnant les villes et leurs caractéristiques, une seconde nous donnant les voisins des villes et une dernière pour les régions et leurs voisins.

Cependant, wikiData étant une base de connaissances et non une base de données, certaines données sont incomplètes. Ainsi, certaines villes n'ont pas une liste exhaustive de tous leurs voisins. Il peut ainsi y avoir des différences entre notre version du saute canton et la version officielle.

3.2 Représentation de la carte

Dans notre version du saute canton, nous parcourons un pays qui est représenté par ses régions et ses villes. Comme nous l'avons vu plus haut, ceux-ci composent les attributs d'un pays et ne sont pas liés directement entre eux. Ces deux attributs sont tous les deux des `subDict`. Les clés sont les codes des villes ou des régions et les valeurs sont des `Canton` ou des `Region`.

Comme nous travaillons particulièrement avec les villes, nous détaillerons seulement celles-ci.

3.2.1 Représentation des villes

Les villes sont représentées par la classe `Canton`. Chaque instance de cette classe a plusieurs attributs atomiques essentiels au bon déroulement du voyage des robots de parcours. Ceux-ci sont :

country : Cet attribut fait référence au pays dans lequel est situé la ville.

code : Celui-ci est obtenu à partir de la requête SPARQL. C'est le code avec lequel la ville est identifiée dans wikidata. Nous l'utilisons comme clé pour référencer les différentes villes dans l'attribut `cantons` de nos `Country`.

name : Le nom de la ville tout simplement.

population : Le nombre d'habitants de la ville. Il sert à déterminer si cette ville possède plus de 50 000 habitants.

region : Le nom de la région dans lequel se situe la ville. Comme le nom de la ville, il est seulement d'intérêt informatif.

coordinates : Un tuple représentant la longitude et la latitude de la ville par rapport à son centre.

neighbours : Un `subDict` contentant une référence des voisins de cette ville.

is_big : Ce dernier attribut est un booléen qui nous permet d'identifier directement si la ville a une population supérieure à 50 000 habitants. Ainsi nous pouvons savoir si cette ville est une ville pouvant servir de destination.

Les villes étant reliées à leurs voisines, nous obtenons en quelque sorte un graphe non-orienté avec pour sommet les villes. Une arête entre les sommets est présente si les villes sont voisines.

3.3 Robots de parcours

Nos robots parcourent le pays grâce à trois méthodes :

1. get_starting_point : Cette première méthode permet d'obtenir une ville aléatoire qui servira de point de départ. Cette ville peut être une grande ville comme une petite ville. Pour nos deux `Voyager`, nous choisissons cette ville de départ aléatoirement parmi les villes de l'instance `Country` passée en paramètre.

2. jump : Cette méthode permet d'aller de ville en ville. Elle diffère selon la nature du robot. Pour aller de ville en ville, elle prend en entrée un `Canton` d'origine et nous parcourons la liste de ses voisins pour obtenir la ville idéale selon le type du `Voyager`. Nous détaillerons cette méthode de sélection par la suite.

3. play : C'est la méthode centrale. Elle orchestre les trajets de ville en ville et met fin aussi au voyage. Il y a deux issus possibles pour la fin d'un voyage:

   (a) on arrive dans une grande ville.

   (b) on est coincé, il n'y a plus de voisin disponible dans le pays et on ne peut pas retourner en arrière.

3.3.1 Parcours aléatoire

Pour ce robot, la méthode `jump` renvoie une ville au hasard par rapport aux voisins de la ville courante. Afin que notre robot ne tourne pas en rond indéfiniment, nous avons déterminé qu'il ne peut aller dans une même ville seulement trois fois. Par exemple, si notre robot est déjà allé dans la ville X trois fois, il ne pourra pas y retourner une quatrième fois.

```
while(True):
#on choisit une ville al atoirement
        choosen = random.choice(
            list(canton.neighbours.values()))
        #on regarde si la ville n'a pas ete visite ou a ete visite
    moins de 3 fois
        if choosen.name not in self.visited_cities or    self.
    visited_cities.count(choosen.name) <= 3:
            self.visited_cities.append(choosen.name)
            if choosen.region not in self.visited_regions:
                self.visited_regions.append(choosen.region)
            return choosen
        #si tous les voisins ont ete explores plus de 3 fois, on
    renvoie None
        elif self.tooMuchOccurence(choices_names):
            return None
```

Listing 1: Coeur de la méthode jump pour le robot aléatoire

3.3.2 Parcours dirigé selon les points cardinaux

Ces robots essaient au maximum de suivre un point cardinal : nord, est, ouest, sud. Ainsi que leur combinaison : sud-est, sud-ouest, nord-est et nord-ouest. Le robot choisi donc une ville cible dont les coordonnées GPS correspondent le plus à la direction qu'il doit suivre.

```
choosen = self.compute_closest_city(coord, neighbors)
```

Listing 2: Coeur de la méthode jump pour le robot qui suit une direction

`coord` corresponds aux coordonnées de la ville courante et `neighbors` est un `subDict` contenant les voisins de la ville courante.

Afin de se diriger vers le nord, il suffit de maximiser la latitude. À l'inverse pour aller vers le sud, il faut la minimiser. Il en va de même pour l'ouest et l'est mais avec la longitude.

Pour la direction sud-est on va minimiser somme de la longitude et de la latitude. En effet, dans cette direction, les deux valeurs décroissent. Pour le nord-ouest, on fait de même mais en maximisant cette somme. Sommer nous permet de façon rapide, d'optimiser soit la longitude, soit la latitude soit les deux.

Pour le sud-ouest, on va essayer soit essayer de minimiser la latitude soit de maximiser la longitude. On fait l'inverse pour le nord-est.

Lorsqu'il n'y a aucune ville voisine sur la direction que suit le robot, il choisit une ville au hasard parmi ses voisins.

3.4 Optimisations

À la suite de notre premier prototypage fonctionnel, nous avons observé des problèmes dans certains point clés de notre programme.

3.4.1 Espace

Premièrement du côté la base de données, nous sommes passer par beaucoup itérations pour en arriver à note implémentation actuelle. Effectivement, des erreurs d'inattention et de connaissances ont mené à tout début à une base de données de plus de 1 Go, pour arriver à moins de 8 Mo.

3.4.2 Temps

Après l'espace de la base de données, le second problème a été le temps d'agrégation des données à partir de WikiData, ainsi que la mise à jour des distances après les calculs.

Dans un premier temps nous avons utilisé les mot-clés LIMIT et OFFSET pour récupérer les données partiellement, car une requête unique résulterait en un échec. Malheureusement ce-là n'a pas aidé, plus on avançait dans les villes, plus les requêtes devenaient lentes jusqu'à l'échec.
Notre solution a été d'utiliser le code INSEE des villes comme filtre. Une fois couplés au multi-processsing qu'offre Python, nous sommes passés de 294 secondes(5min) à 91 secondes(1m31), soit une vitesse 3 fois supérieure.
Nous avons donc fais de même pour récupérer les voisins des villes. Mais petite différence, nous récupérons les voisins d'une centaine de villes à la fois grâce à VALUES en concaténant les codes de différentes villes, compressant 100 requêtes en une.

De même pour la mise à jour des distances. Au lieu de faire plus de 37,000 requêtes pour changer les distances une par une, nous groupons les villes par distances pour ne faire qu'une requête par distance(0, 1, 2,...). Soit seulement une vingtaine de requête dans notre cas, nous sommes passés d'une mise à jour interminable, à une mise à jour presque instantanée.

4 L'extension "d'aide" Firefox

Nous avons réalisé une extension pour le navigateur Firefox affichant la distance des grandes villes par rapport à la ville courante.

4.1 Présentation des données

Mais avant de présenter l'extension, analysons quelques donnés:

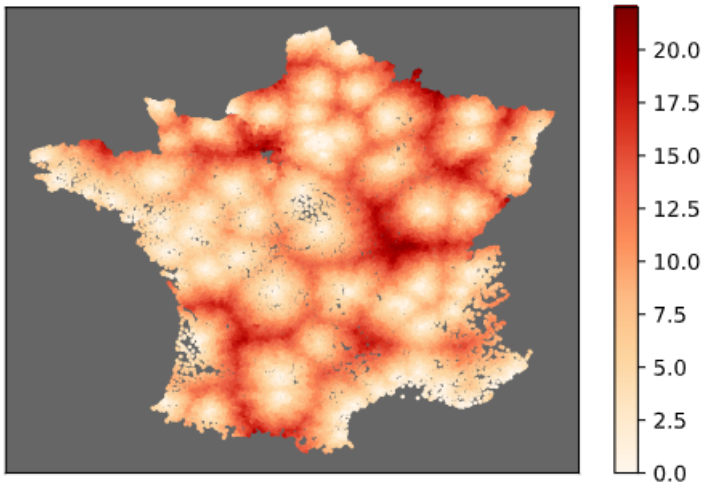

Figure 3: Carte des distances

Voici une carte de la France représentant les distances des villes aux grandes villes. Plus une ville est loin, plus ça représentation est sombre.

Nous pouvons remarquer que la plupart des villes sont aux alentours de 8 à 9 sauts d'une grande ville, mais que certaines zones sont plus peuplés que d'autres, comme par exemple les bords de mer. Au contraire on remarque des creux, par exemple vers Chartres, ce trou pourrait être expliqué par le fait que nous utilisons une base de connaissances et une manière particulière de calculer les distances. Nous partons des grandes villes et sautons vers un voisin incrémentant ainsi la distance. Cependant, si nos données sont incomplètes certaines zones peuvent être donc isolées et non calculées.
Une autre anomalie se présente vers Clermont-Ferrand, étant pourtant une grande ville aux alentours de 140,000 habitants.

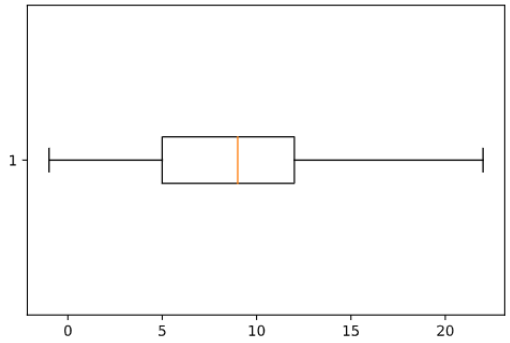

Figure 4: Diagramme en boîte des distances

Nous avons récupéré au total 37,525 villes, dont seulement 119 grandes villes; c'est-à-dire 0.31% , soit un ratio 1:315 grande ville par ville. Avec un nombre de grandes villes si bas on pourrait s'attendre à une moyenne beaucoup plus grande, cela montre donc que dans l'ensemble la France possède une bonne répartition des grandes villes dans son territoire. En moyenne une ville a 5 voisins.
Tout de même, un quart des villes se trouve à plus de 12 sauts d'une grande ville. Dans le pire des cas, la distance la plus grande à une grande ville est de 22 sauts. Cette mesure constitue le diamètre de notre modèle de la France.

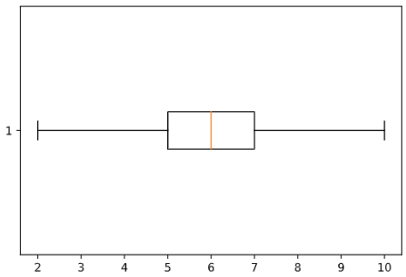

Figure 5: Nombre de voisins

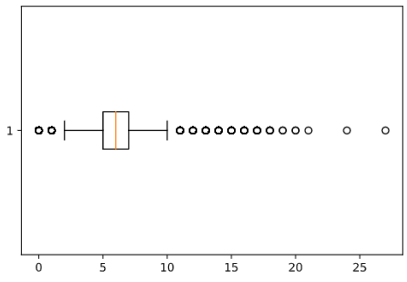

Figure 6: Nombre de voisins(avec extrêmes)

4.2 Implémentation

Pour faire notre extension, nous avons d'abord pensé à faire appel à la base de données directement, mais cela c'est avéré plus compliqué que prévu, utilisation de bibliothèques, projet Node/NPM; Plus compliqué que nécessaire. Nous avons décidé d'exporter les distances dans un fichier JSON pour pouvoir charger les distances immédiatement. Les seuls désavantages que nous voyons et le fait de devoir réexporter les distances à chaque changement de la base de données. Mais ce-là est contrebalancé par une taille bien moins conséquente: 673 Ko vs.

7684 Ko.

Le principe est donc simple, avec un script en Javascript, on récupère les noms des villes à partir de la partie courante de Saute Canton, et les utilisons comme clé dans notre table des distances, puis modifions le texte des villes pour les afficher. Et voilà !

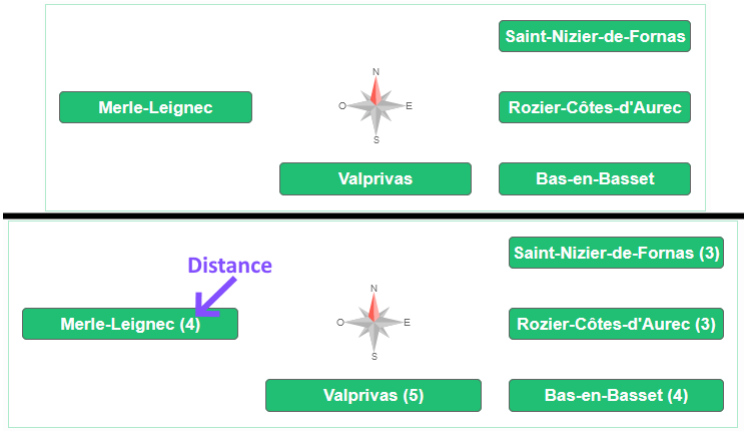

Figure 7: Démonstration de l'ajout des distances

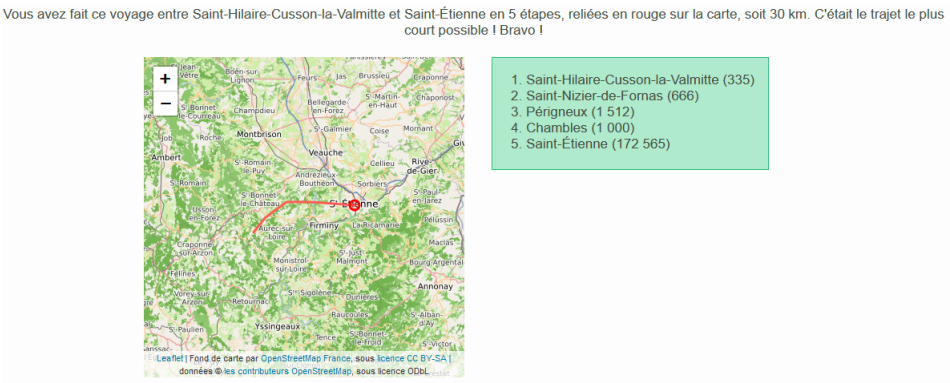

Figure 8: Exemple de partie parfaite

Malheureusement, cette aide n'est pas parfaite. Comme répéter plutôt le fait de ne pas avoir les mêmes données peut entraîner des différences.

Comme on peut le constater sur cet exemple les valeurs sont très variées, ce

qui perturber l'utilisation. Malgré tout, dans ces occasionnels, l'extension permet de savoir la direction générale. Et avec un peu d'expérience, nous concluons qu'en général la meilleure solution est de suivre la valeur la plus.

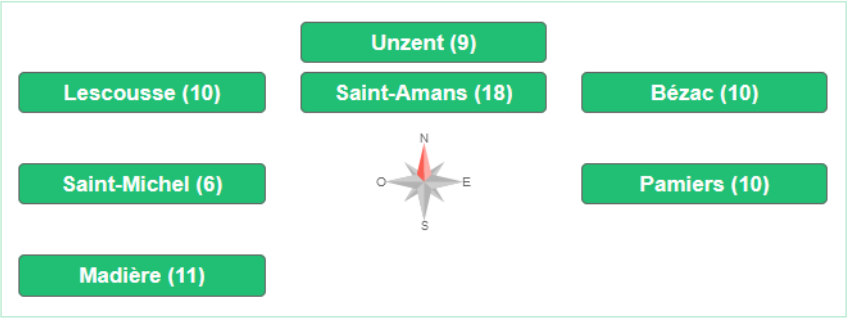

Figure 9: Exemple de partie problématique

Il nous est aussi advenu de rencontrer des problèmes mais cette fois de la part du site, refusant des victoires pourtant légales.

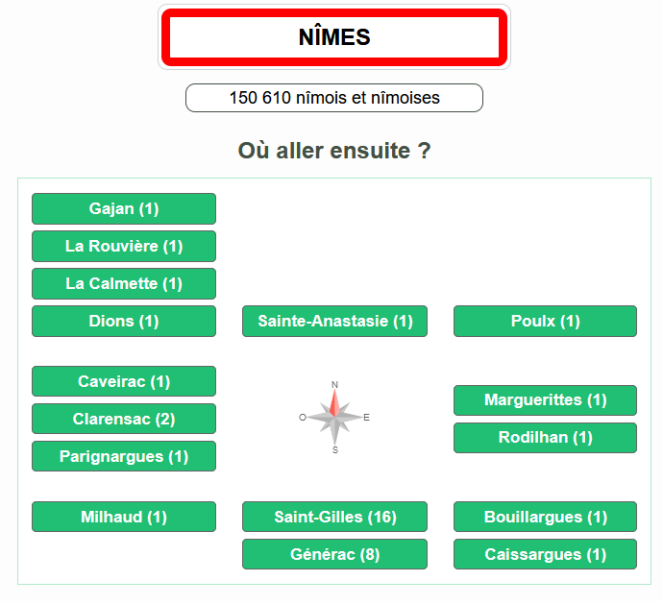

Figure 10: Exemple de problème avec le site

5 Tests Statistiques

Dans un premier temps, nous allons présenter l'ensemble des résultats que nous avons eu. Puis nous nous dirigerons plus en détail vers les différentes stratégies de parcours abordées; c'est-à-dire une stratégie aléatoire et une stratégie se basant sur les points cardinaux.

Nous mesurons le nombre de sauts fait par les robots, le nombre de régions parcourues et indiquons si oui ou non, notre robots arrive dans une grande ville avant de ne plus avoir de voisin déjà parcouru. Pour nos tests, une grande ville est une ville d'au moins 50 000 habitants.

5.1 Résultats globaux

En cumulant nos deux stratégies, nous obtenons un total de 90 000 résultats. Sur ces 90 000 essais seulement un peu plus de 31 000 ont été concluants, donc un peu près 30%. C'est-à-dire, que seuls 31 000 robots sont arrivés dans une grande ville.

La moyenne de sauts des robots étant arrivés à destination est de 51 sauts. Alors que la médiane est de 39. L'écart type entre les valeurs est de 44 sauts. Ces résultats sont très éloignés du diamètre de la France qui est de 22 sauts.

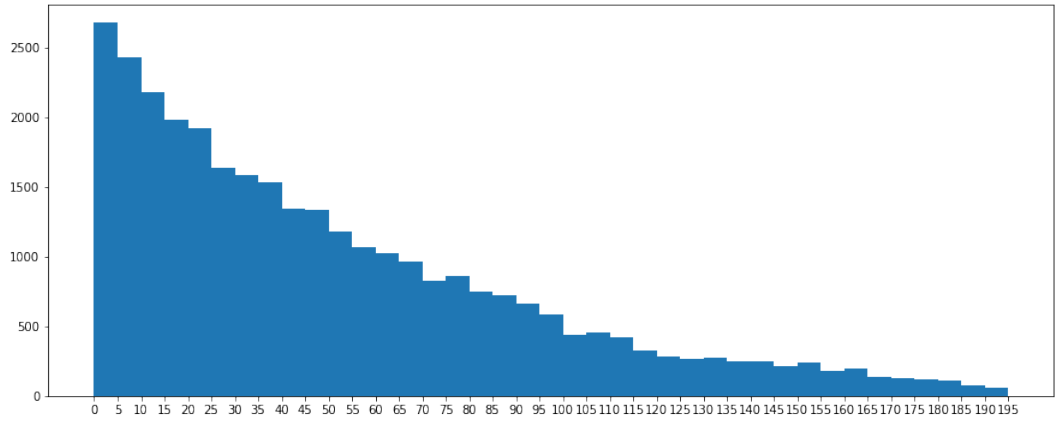

Figure 11: Répartition totale des sauts pour les robots biens arrivés

Comme nous le montre ce graphique 11, la répartition des sauts forme grossièrement une courbe normale centrée autour de 0. On atteint un pic d'un peu plus de 2500 robots qui arrivent à dans une grande ville en 5 sauts maximums. Dans nos tests et comme l'illustre le graphe 11, dans le pire des cas, un robot qui converge prendra entre 190 et 195 sauts pour arriver à destination d'une grande ville.

5.2 Direction aléatoire

En prenant un direction aléatoire à chaque nouvelle ville, une batterie de 10 000 tests, 8% de nos robots arrivent à destination. Malgré ce faible pourcentage, ils prennent en moyenne 28 sauts. Et la médiane des sauts est de 18 et le troisième quartile est 40. Ce qui en fait notre robot le plus "rapide".

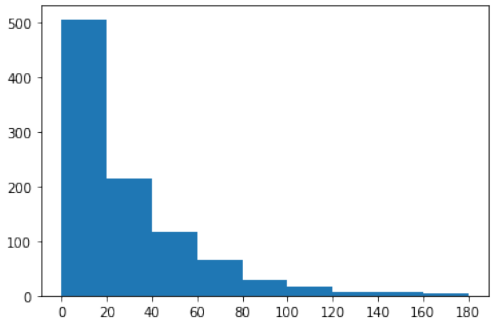

Figure 12: Répartition des sauts pour les robots aléatoires biens arrivés

Comme nous pouvons le voir en nous appuyant sur la répartition des sauts 12, on arrive soit très vite à destination soit très lentement, mais 90% du temps, pas du tout.

5.3 Points cardinaux

Pareillement à la direction aléatoire, chaque direction sera testée 10 000 fois.

Nord : En partant pour le nord, 27% de nos robots sont arrivés à destinations d'une grande ville. Ils ont fait en moyenne 53 sauts et ont traversé en moyenne 4 régions.

Le nord est la direction qui nous donne le moins de chance d'arriver dans une grande ville.

L'histogramme de la répartition du nombre de sauts pour ceux arrivés à destination 13 nous donne une forme similaire aux histogrammes précédents. Les histogrammes suivants nous donnant le même type de courbe et un écart type similaire (autour de 40 sauts), nous prenons le choix de ne pas les afficher afin de rendre le rapport plus compact et lisible.

Nord-Ouest: Partir vers le nord-ouest est un meilleur choix, avec en moyenne 41% des robots qui arrivent dans une grande ville. Le nombre de sauts moyen s'élève à 52,14 et le nombre de régions traversées à 4,45.

Nord-Est: La direction du nord-est ne donne pas de bons résultats. Seulement 37% des robots convergent. Ceux qui convergent font en moyenne 65

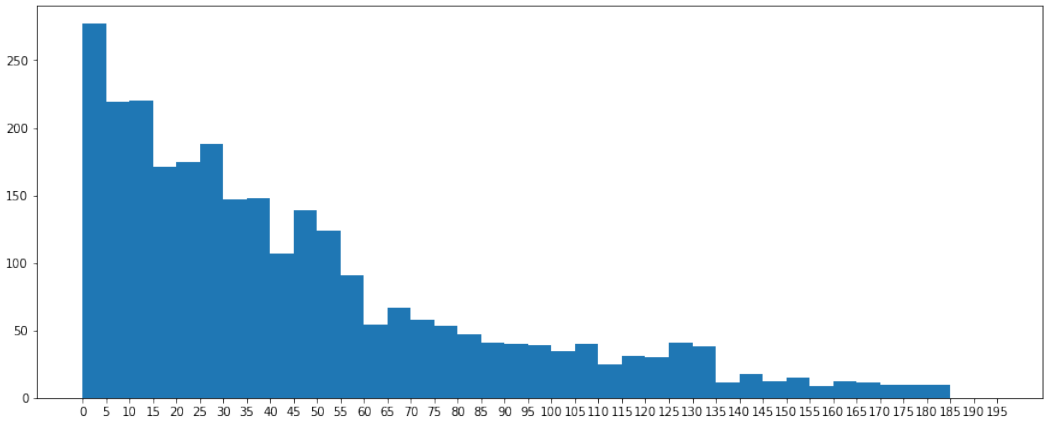

Figure 13: Répartition du nombre de sauts pour les robots bien arrivés se dirigeant vers le nord

sauts, ce qui est le nombre de saut moyen maximum que nous avons obtenu. Ils traversent en moyenne 4,3 régions.

Est : Tout comme le nord-est et le nord,l'est n'est pas une bonne direction à prendre, elle à le second pourcentage de convergence le plus faible : 31%. De même manière que pour les autres directions, prendre la voix de l'est fait traverser un grand nombre de villes : 51,7. Choisir cette direction fait aussi traverser 4,24 régions en moyenne.

Ouest : Étonnamment prendre la direction de l'ouest ne donne pas de bon résultat (29% en moyenne) comparé au nord-ouest(%41) et au sud-ouest (que nous verrons plus tard) . Cependant, lorsqu'ils arrivent à bonne destination, ils font en moyenne un peu moins de sauts, 47,4 que lorsqu'ils suivent les deux directions mentionnés au-dessus.

Sud : En se dirigeant vers le sud, 51% de nos robots atteignent une grande ville. On peut corréler ce pourcentage énorme au fait que de nombreuses grandes villes sont présentes sur les fronts de mer français orienté vers le sud, comme nous l'avons mentionné dans la partie sur l'extension. De plus, ces fronts de mer recouvrent majeure partie des frontières sud de la France. On a ultimement une grande probabilité de tomber à ces endroits (partie bleue du graphique 14).

On notera que la moyenne de sauts est ici de 54,9 et que nos robots traversent en moyenne 5 régions. Ces moyennes font partie des plus hautes que nous avons mesurées.

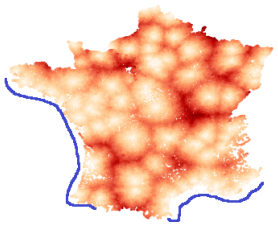

Figure 14: Frontières maritimes face sud

Sud-Ouest : En suivant cette direction, nos robots arrivent dans 46% des cas dans une grande ville. Cela s'explique naturellement lorsque l'on mette ce résultat en corrélation avec la carte 14. La face maritime sud-ouest est parsemée de blancs et donc de grandes villes.

Mais similairement aux robots ayant pris pour direction le sud, on se retrouve avec une grande moyenne de sauts : 63 et 4,8 régions traversées. On peut dire ironiquement que c'est la direction qui fait le plus voyager.

Sud-Est : Finalement, le sud-est est aussi une bonne direction à prendre, avec en moyenne 41% des robots qui réussissent à aller dans des grandes villes. La moyenne des sauts est 51 et la moyenne des régions traversées de 4,7.

5.4 Bilan des tests

Avec une médiane de 9 sauts et un diamètre de 22 sauts, extraits grâce à notre extension, on peut dire que nos robots sont très lents. Globalement, ils font en moyenne 51 sauts, soit plus de 10 fois plus de sauts que la médiane .

Le robot aléatoire a très peu de probabilité d'arriver dans une grande ville. Cependant, c'est celui qui converge le plus rapidement.

Les robots qui suivent une direction précise eux, ont au minimum 27% de chance de converger vers une grande ville, soit 3 fois plus que le robot aléatoire, avec un maximum de chances de 51% pour le sud. Mais, ils convergent plus lentement que le robot aléatoire, avec en moyenne 38 sauts contre 28. Parmi les directions, les plus intéressantes sont celles qui mènent vers le sud.

Lorsque l'on observe les différents histogrammes de répartition des sauts, on comprend surtout que nos robots passent très souvent à côté des grandes villes.

6 Conclusion

Dans ce projet, nous nous sommes fixés pour objectifs de recréer le jeu du saute canton et de trouver un moyen pour gagner le plus rapidement. Pour cela, nous avons dû créer notre propre modèle de la France en extrayant des données de WikiData. Et faire parcourir ce modèle par des robots selon des stratégies que nous aurons développées, afin de trouver la meilleure. Parallèlement à cela, nous avons aussi analyser ces données manuellement pour mesurer les taux de chances selon la direction prises et créer une extension qui nous indique les villes les plus intéressantes vers les quelles se diriger pour gagner.

De manière générale, se diriger aléatoirement est une très mauvaise idée. Aller vers l'une des directions cardinales est l'une des meilleures solutions. Nos tests nous montrent que l'on a une chance sur quatre d'arriver dans une grande ville en suivant la pire des directions, le nord. Afin de maximiser cette chance, il faut de préférence aller dans l'une des trois directions allant vers le sud : le sud, le sud-est et le sud-ouest. Cependant, en ne prenant pour information que la localisation des villes et pour stratégie qu'une direction, on passe souvent à côté des grandes villes.

La pluridisciplinarité de ce projet nous aura permis de renforcer toutes nos compétences en informatique. Et contrairement à ce que nous pensions au départ, apprendre et surtout comprendre SPARQL, construire des requêtes de façon à obtenir les données voulues de WikiData de manière optimale, ce sont avérés être les parties les plus dures du projet. Nous n'avons nous-mêmes pas réussi intégralement à construire cette requête. Nous avons donc dû ruser et la découper en plusieurs sous-requêtes.

Afin d'améliorer nos résultats, une piste d'amélioration et d'exploration de nos stratégies serait de prendre en compte le nombre d'habitants afin de se diriger vers des villes de plus en plus grande. Et par la suite arriver dans une agglomération puis dans la ville centrale de celle-ci.

Pour conclure sur une note plus légère, nous pourrons tirer ou pointer de notre projet que si l'on souhaite visiter beaucoup de grandes villes pendant les vacances le sud est la meilleure destination.