Apprentissage des Protocoles de Communication pour les Algorithmes Distribués

Table des matières

1 Introduction

Les algorithmes distribués jouent un rôle fondamental dans la gestion et l'analyse des réseaux à grande échelle. Dans le modèle LOCAL, chaque nœud d'un graphe ne communique qu'avec ses voisins immédiats pour résoudre un problème global tout en minimisant le nombre de tours de communication. Toutefois, la communication représente souvent un goulot d'étranglement dans ces algorithmes, et la défaillance d'un ou plusieurs nœuds peut perturber considérablement le processus.

L'objectif de ce travail est d'explorer comment les techniques d'apprentissage automatique peuvent être utilisées pour prédire les bits transmis par les nœuds voisins dans un contexte distribué. Cela permettrait de remplacer les informations manquantes et d'assurer une plus grande robustesse des algorithmes contre les pannes.

2 État de l'Art

Les algorithmes distribués classiques tels que le MIS (Maximal Independent Set) et le Graph Coloring fonctionnent en minimisant les communications tout en assurant l'optimisation des résultats globaux. Cependant, ces algorithmes sont vulnérables aux pannes des nœuds et nécessitent des stratégies avancées pour la gestion des erreurs.

Les approches existantes utilisent principalement des techniques de redondance ou de recalcul partiel, mais elles restent coûteuses en termes de ressources. L'intégration de l'apprentissage automatique pourrait apporter une alternative plus efficace en prédisant les communications perdues.

3 Modélisation du Problème

Nous avons défini notre problématique principale autour des questions suivantes :

— Comment prédire les bits de communication échangés entre nœuds ?
— Quels modèles d'apprentissage sont les plus efficaces pour cette tâche ?
— Comment intégrer ces modèles dans un environnement distribué avec des contraintes de temps réel ?

4 Algorithme Alice–Bob

Dans le modèle classique d'un réseau pair-à-pair (P2P), deux nœuds – nommés Alice et Bob – échangent des bits (0 ou 1). Cependant, l'un des deux (par exemple, Bob) peut se déconnecter à tout moment et cesser d'envoyer de nouveaux bits. Dans ce cas, l'autre nœud (Alice) doit deviner les bits que Bob aurait pu envoyer. Étant donné que ces bits sont générés de façon aléatoire (avec une probabilité `p_one` d'être égaux à 1), il est impossible de les prédire parfaitement.

4.1 Problématique principale

L'objectif est de minimiser l'erreur de transmission malgré cette déconnexion. Chaque nœud maintient un historique des bits récents reçus sous forme d'un tampon (buffer). Quand un nœud se déconnecte, l'autre tente de prédire les bits manquants grâce aux statistiques de son buffer. L'aspect aléatoire de la génération des bits entraîne inévitablement une incertitude dans la prédiction, mais cette méthode permet de poursuivre la simulation ou la communication sans interrompre complètement le processus.

4.2 Paramètres et implémentation du code

Chaque nœud (Alice ou Bob) est caractérisé par :

- `p_one` : la probabilité d'envoyer un bit valant 1.
- `buffer_size` : la taille du buffer pour stocker les bits récemment reçus.
- `is_disconnected` : un indicateur booléen précisant si le nœud est déconnecté.
- `count_ones` et `count_total` : le décompte de bits 1 et le nombre total de bits reçus, utiles pour calculer une probabilité de prédiction.

Lorsqu'un nœud est déconnecté, il ne peut plus envoyer de bits, et l'autre nœud doit appeler une fonction de « devinette » (guess). Ci-dessous, nous présentons uniquement les deux méthodes principales pour envoyer et deviner un bit :

Extrait de code pertinent :

```
# --- Création du message ---
def send_message(self):
    if self.is_disconnected:
        return None
    bit = 1 if random.random() < self.p_one else 0
```

```
    return bit

# --- Devinette du message ---
def guess_message(self):
    if self.count_total == 0:
        return 0
    if self.count_total >= self.buffer_size:
        prob_one = self.count_ones / self.buffer_size
    else:
        prob_one = self.count_ones / self.count_total
    return 1 if random.random() < prob_one else 0
```

4.3 Conclusion et brève allusion aux travaux futurs

Cet algorithme classique montre comment gérer la déconnexion éventuelle d'un nœud dans un réseau P2P en conservant un estimé des bits manquants. Certes, la précision n'est pas garantie en raison de l'aléatoire inhérente à la génération des bits, mais cette approche maintient la cohérence des échanges et évite l'arrêt brutal de la communication. Enfin, on peut noter qu'une étape ultérieure consistera à calculer et analyser la quantité de données perdues (*miss*) et à mettre en place des méthode plus avancées pour la reconstruction des données manquantes, si nécessaire.

5 Algorithme Alice–Bob

Dans un réseau pair-à-pair (P2P) où chaque nœud génère aléatoirement des bits (0 ou 1) avec une certaine probabilité p_{one}, l'algorithme Alice–Bob sert à gérer les déconnexions soudaines des nœuds. Lorsque Bob se déconnecte, Alice essaie de deviner les bits que Bob aurait envoyés. Dans la version *classique*, cette prédiction se base simplement sur la proportion de 1 reçus dans le buffer (méthode `guess_message`). Cependant, nous proposons ici une version améliorée où Alice utilise un modèle de **régression logistique** pour prédire les bits manquants.

5.1 Pourquoi la Régression Logistique ?

— **Principe de base :** La régression logistique permet de modéliser la probabilité qu'une variable binaire (ici, le bit vaut 1) soit égale à 1 en fonction de certaines variables explicatives (par exemple, l'identifiant

de Bob, le nombre de bits déjà reçus, etc.). Elle calcule donc :

$$\mathbb{P}(Y = 1 \mid X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n)}}$$

où Y est la variable binaire (le bit), et $X_1, \ldots, X_n$ les variables explicatives.

— **Pourquoi utiliser un modèle d'apprentissage ?**
  — *Approche naïve* : On estime la probabilité d'obtenir 1 par $\frac{\text{count\_ones}}{\text{count\_total}}$.
  — *Régression logistique* : On cherche à faire mieux en exploitant des *features* plus complexes (par exemple, plusieurs Bob, l'historique par Bob, etc.).

  Même si, dans ce projet, les bits sont *effectivement* générés de façon aléatoire selon une probabilité fixe p_{one}, la régression logistique peut s'avérer utile lorsqu'on souhaite tenir compte de plus de paramètres (nombre de déconnexions, contextes, interactions multiples, etc.).
— **Amélioration incertaine :** Dans notre cas précis, puisque la distribution des bits reste purement aléatoire (avec une probabilité fixe), on ne s'attend pas forcément à des gains considérables par rapport à la méthode *compter le nombre de 1 dans le buffer*. Néanmoins, cette approche sert de tremplin pour des scénarios plus complexes ou réalistes (où la probabilité génératrice peut varier avec le temps ou l'environnement).

5.2 Architecture du Code

5.2.1 Héritage : Classe `Alice` dérivée de `Node`

Dans notre implémentation, chaque nœud du réseau (Alice ou Bob) est représenté par la classe `Node`, qui propose les fonctionnalités de base :

— **Attributs principaux :**
  — `p_one` : probabilité d'envoyer 1.
  — `buffer_size` : taille du buffer pour stocker les bits reçus.
  — `is_disconnected` : booléen indiquant si le nœud est déconnecté.
  — `received_bits`, `count_ones`, `count_total` : pour suivre l'historique de bits reçus.
— **Méthodes :**
  — `send_message()` : envoie un bit en se fondant sur `p_one`.
  — `receive_message(bit)` : met à jour le buffer et les compteurs de bits reçus.
  — `guess_message()` : devine un bit lorsqu'un nœud est déconnecté (version de base : probabilité = $\frac{\text{count\_ones}}{\text{count\_total}}$).

La classe `Alice` **hérite** de `Node` pour ajouter les capacités suivantes :

- **Attributs spécifiques :**
  - `bob_history` : dictionnaire pour stocker l'historique par Bob.
  - `model` : instance d'un modèle de régression logistique (issu de `scikit-learn`).
  - `training_data`, `training_labels` : pour enregistrer des exemples et entraîner le modèle.
- **Méthodes supplémentaires :**
  - `receive_message_from_bob(bob_id, bit)` : enregistre les bits reçus de chaque Bob, alimente la base d'apprentissage.
  - `train_model()` : entraîne la régression logistique sur les données recueillies.
  - `predict_message(bob_id)` : utilise la régression logistique pour deviner le bit manquant d'un Bob déconnecté.

5.2.2 Nouvelle méthode de prédiction (`guess_message`) dans la classe `Alice`

```
def predict_message(self, bob_id):
    """
    Utilise le modèle entraîné pour prédire le bit d'un Bob déconnecté.
    Si le modèle n'est pas encore entraîné, on retombe sur la méthode traditionn
    """
    if not self.is_model_trained:
        return self.guess_message()

    bob_msg_count = len(self.bob_history.get(bob_id, []))
    features = np.array([[bob_id, bob_msg_count]])

    try:
        proba = self.model.predict_proba(features)[0][1]
        bit_prediction = 1 if random.random() < proba else 0
        return bit_prediction
    except Exception as e:
        print(f"Error making prediction: {e}")
        return self.guess_message()
```

Remarque : On conserve la possibilité de repasser par la fonction `guess_message` de la classe parente (`Node`) si le modèle n'est pas (ou mal) entraîné.

5.3 Performances et Discussion

- **Pas de gain significatif sur des bits strictement aléatoires :** Les bits envoyés par Bob sont générés selon une loi de Bernoulli de paramètre p_{bob}. Par conséquent, la régression logistique ne trouve pas de corrélations notables, aboutissant à des performances proches de la méthode naïve.
- **Maintien des nœuds déconnectés :** Les nœuds "fantômes" restent pris en compte dans les prédictions. Cette politique permet de réintégrer un nœud si celui-ci revient en ligne, en conservant son historique.
- **Prochaines étapes :**
  - Reconstruction de bits avec un dataset complexe (MNIST).
  - Calcul du taux d'erreur ("miss") et colorisation d'un graphe pour analyser la propagation des erreurs.

5.4 Conclusion

Dans cette version du rapport, nous avons introduit une extension de l'algorithme Alice–Bob classique, où la prédiction des bits d'un nœud déconnecté s'appuie sur un modèle de régression logistique.

Avantages : Cette approche prouve la faisabilité d'un apprentissage supervisé dans un contexte P2P, avec un code modulaire et extensible.

Limites : Quand les bits sont générés indépendamment et aléatoirement, le modèle ne dispose pas de réelles corrélations à exploiter.

Ce travail prépare néanmoins l'introduction d'approches plus avancées (autoencodeurs, reconstruction d'images MNIST, calcul du taux d'erreurs "miss", etc.), cruciales lorsque les données possèdent des structures spatiales ou temporelles plus riches.

6 Algorithme Alice–MNIST

Dans cette section, notre objectif principal est de concevoir et de mettre en œuvre une approche d'apprentissage automatique pour qu'un réseau pair-à-pair (Peer-to-Peer) puisse transmettre efficacement des données d'images et, en cas de perte de certains bits ou pixels, soit capable de reconstruire l'image initiale autant que possible. L'idée directrice est que si, dans un réseau P2P, un pourcentage élévé de nœuds (pixels) se déconnectent temporairement, un modèle d'apprentissage puisse estimer les données manquantes et reconstituer l'image originale.

6.1 Problématique

Question principale :
"Comment exploiter des modèles d'apprentissage automatique pour améliorer la transmission et la reconnaissance de données dans un réseau distribué (Peer-to-Peer) et permettre la récupération de parties manquantes ?"

Ici, nous considérons la situation où un grand nombre de nœuds (c'est-à-dire de pixels) sont déconnectés, et le modèle doit estimer au mieux ces éléments manquants en s'appuyant sur les informations restantes.

6.2 Description

Dans cet algorithme, nous utilisons le **jeu de données MNIST**, qui contient des images manuscrites des chiffres de 0 à 9. Ce jeu de données comporte 60 000 images pour l'entraînement et 10 000 pour le test. Chaque image est en niveaux de gris (une seule composante) au format 28×28 pixels. Le choix de MNIST est motivé par sa simplicité et sa popularité pour valider des méthodes de reconnaissance d'images.

6.2.1 Technologies utilisées

1. **Bibliothèque PyTorch :**
   Pour la construction et l'entraînement des modèles d'apprentissage profond (réseaux de neurones convolutifs, etc.).
2. **Régression logistique (Logistic Regression) :**
   Utilisée dans certaines parties pour estimer la probabilité qu'un bit (ou un pixel) soit 1, au cas où nous n'utiliserions pas l'architecture CNN.
3. **Réseau de neurones convolutif (CNN) de type Autoencoder :**
   Un CNN est un réseau neuronal conçu pour extraire automatiquement des caractéristiques spatiales d'une image à travers des filtres convolutifs. Un Autoencoder, quant à lui, est un modèle composé d'un encodeur et d'un décodeur, qui apprend à compresser puis reconstruire les données. Dans notre travail, nous utilisons un autoencoder convolutif pour reconstruire les zones manquantes dans les images, en s'appuyant sur les informations partielles disponibles. C'est la partie centrale du modèle pour prédire les pixels déconnectés.

6.2.2 Pourquoi avoir choisi un CNN et un Autoencoder ?

— **Représentation compacte (Encodage) et reconstruction (Décodage) :**
Dans le problème de récupération de pixels perdus, un autoencodeur peut prendre en entrée une image partielle (Partial) et produire une reconstruction (Reconstruction) proche de la version initiale.

— **Robustesse face au bruit et aux données manquantes :**
La structure d'un CNN et ses filtres de convolution sont très efficaces pour apprendre les caractéristiques spatiales d'une image, et peuvent assez bien réparer ou compléter les régions détruites ou manquantes.

— **Rapidité et simplicité :**
Les Autoencoders convolutifs sont relativement simples à implémenter pour des données d'image et offrent un bon compromis entre la précision de reconstruction et la rapidité d'entraînement.

6.2.3 Aperçu du code (extraits principaux)

Voici un exemple de la classe `CNNAutoencoder`, sans inclure tous les détails afin de ne pas allonger excessivement le rapport :

```
class CNNAutoencoder(nn.Module):
    """
    Un autoencodeur convolutionnel simple:
         - Encoder: 2 blocs Conv -> MaxPool
         - Decoder: 2 blocs ConvTranspose
    """
    def __init__(self):
        super(CNNAutoencoder, self).__init__()

        # --- Encoder ---
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )

        # --- Decoder ---
        self.decoder = nn.Sequential(
```

```
            nn.ConvTranspose2d(64, 32, kernel_size=2, stride=2),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 1, kernel_size=2, stride=2)
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

Dans la partie **Encoder**, on trouve deux couches de convolution suivies de **MaxPool**, permettant d'extraire et de compresser les caractéristiques visuelles de l'image.
Dans la partie **Decoder**, grâce aux couches `ConvTranspose2d`, on restaure les dimensions originales (28×28).

Les classes comme `Node` et `Alice` simulent la logique d'un nœud P2P : gestion de la déconnexion (inconnue de l'état d'un pixel) et recours à un modèle entraîné (CNN ou logistic regression) pour deviner les bits manquants. L'essentiel du code se trouve dans le fichier `p2pmnist.py`.

La fonction `train_cnn_model_with_dynamic_disconnections(...)` illustre la méthode d'entraînement : à chaque *batch*, un pourcentage de pixels est aléatoirement déconnecté, et le modèle apprend à reconstruire l'image malgré cette perte partielle.

6.3 Résultats

Après avoir entraîné le modèle sur MNIST en simulant 60 % de déconnexions, l'Autoencoder a atteint une précision moyenne d'environ **94.7 %** dans la prédiction des pixels perdus (calculée comme le pourcentage de bits correctement prédits parmi les pixels déconnectés). De plus, jusqu'à **60 % de données manquantes**, le modèle parvient à reconstruire assez fidèlement la forme générale du chiffre.
Figure (6-1) : Voici les **précisions moyennes** du modèle en fonction de chaque chiffre (de 0 à 9). On observe une précision allant d'environ 92 % à 98 % selon le chiffre.

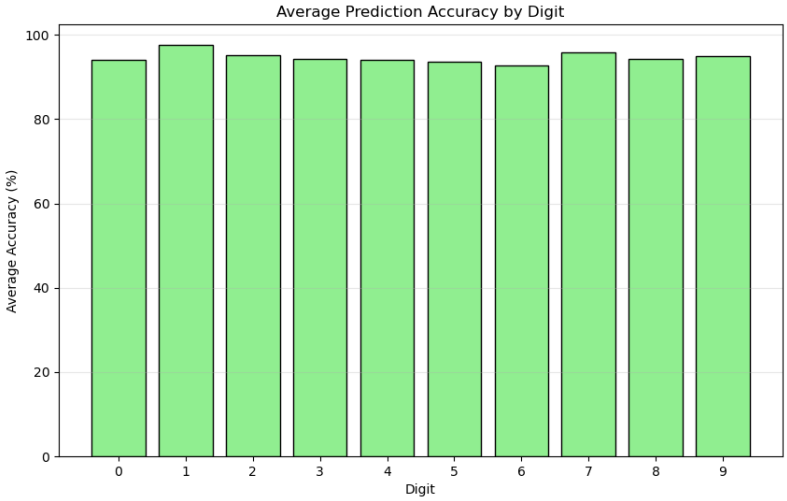

Figure (6-1) : Précision moyenne de la reconstruction des pixels manquants pour les chiffres 0 à 9 (en pourcentage).

Figure (6–2) : Elle illustre la distribution des précisions (*Accuracy Distribution*) sur différents échantillons de test. La plupart des échantillons se situent entre 90 % et 98 % de précision, et la moyenne globale est d'environ 94,79 %.

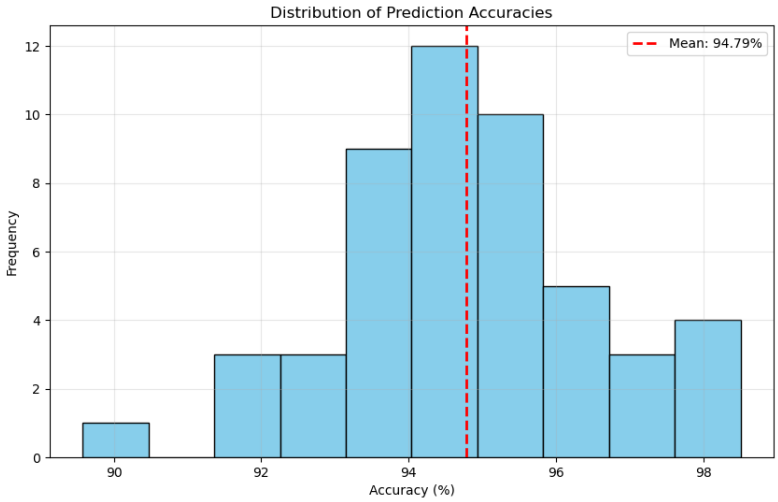

Figure (6-2) : Distribution de la fréquence de précision de prédiction sur les échantillons de test.

Figure (6-3) : Un exemple concret pour le chiffre « 5 » présenté sous trois formes :

1. L'image originale (à gauche)
2. L'image partielle (au milieu) avec 60 % de pixels déconnectés
3. L'image reconstruite (à droite) par le modèle, montrant sa capacité à reconstituer l'essentiel du chiffre.

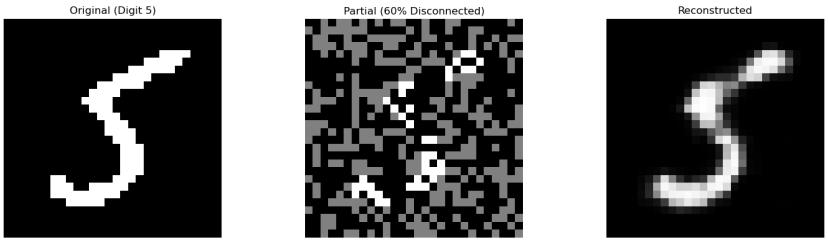

Figure (6-3) : Exemple d'image originale, image avec 60 % de pixels déconnectés et image reconstruite par l'Autoencoder.

6.4 Limites (Limitations)

1. **Forte dépendance au type de jeu de données :**
   Bien que le modèle fournisse d'excellents résultats sur MNIST (images binaires noir et blanc), dans des cas plus réalistes avec un spectre plus large (images RGB, etc.), il faudra un réseau plus profond et des ensembles de données beaucoup plus volumineux pour reproduire ce type de performance.
2. **Besoin de ressources de calcul plus puissantes :**
   MNIST est un petit jeu de données relativement simple. Pour obtenir un modèle « complet » couvrant divers types d'images, il est nécessaire de disposer d'une infrastructure de calcul importante (GPU avancé ou clusters) ainsi que d'ensembles de données massifs.
3. **Baisse de performances au-delà de 60 % de déconnexion :**
   Bien que la reconstruction reste satisfaisante jusqu'à environ 70 % de déconnexion pour la majorité des chiffres, on commence à observer une diminution progressive des performances. À partir de 90 % de perte de données, le modèle n'arrive plus qu'à reconstruire uniquement le chiffre 1.

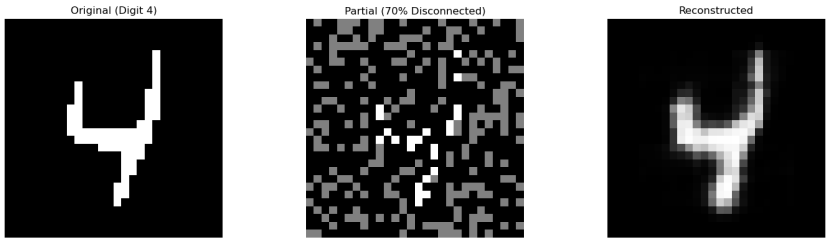

Figure (6-4) : Exemple d'image originale, image avec 70 % de pixels déconnectés et image reconstruite par l'Autoencoder.

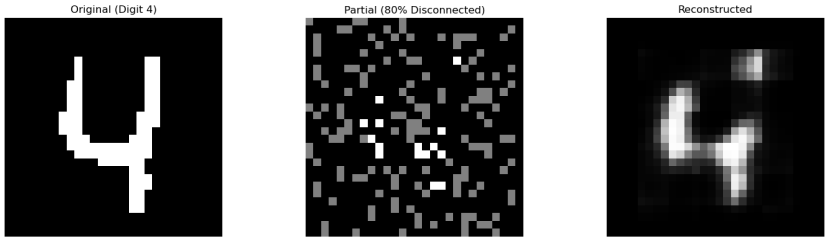

Figure (6-5) : Exemple d'image originale, image avec 80 % de pixels déconnectés et image reconstruite par l'Autoencoder.

Conclusion

Dans cette partie du projet (section 6), nous avons appliqué un réseau de neurones convolutif de type Autoencoder sur le jeu de données MNIST afin de reconstituer les pixels manquants (jusqu'à 60 % de perte). Le modèle atteint en moyenne 94.7 % de précision pour la prédiction et la reconstruction. Toutefois, pour aborder des cas réels plus complexes, il faudra envisager des jeux de données bien plus étendus et des architectures plus sophistiquées.

7 Algorithme MIS

7.1 Problématique

Comment construire un ensemble indépendant maximal (MIS) de manière efficace et résiliente dans un environnement distribué ?

7.2 Description

L'algorithme MIS est essentiel pour de nombreuses tâches en réseaux distribués, notamment pour la gestion des ressources et la planification des transmissions.

7.3 Implémentation et Résultats

8 Conception et Développement de l'Application

8.1 Introduction

Dans le cadre de notre projet de fin d'études, nous avons développé une application web permettant de visualiser et de manipuler des graphes distribués. Cette application a été conçue en utilisant Django comme framework backend, HTML et JavaScript pour le frontend, et la bibliothèque GoJS pour la représentation graphique des graphes. Cette section décrit en détail la conception et le développement de l'application.

8.2 Problématique

Comment concevoir une interface interactive et intuitive pour la visualisation et la gestion des algorithmes distribués ?

8.3 Architecture de l'Application

L'application suit une architecture MVC (Modèle-Vue-Contrôleur) classique, où Django gère le modèle et le contrôleur, tandis que le frontend (HTML, JavaScript) gère la vue. GoJS est utilisé pour la partie graphique, permettant de créer, modifier et visualiser les graphes de manière interactive.

- **Backend (Django)** : Gère la logique métier, la gestion des données, et les API nécessaires pour interagir avec le frontend.
- **Frontend (HTML, JavaScript)** : Gère l'interface utilisateur et les interactions avec l'utilisateur.
- **GoJS** : Bibliothèque JavaScript pour la création de diagrammes interactifs, utilisée pour représenter les graphes.

8.4 Fonctionnalités de l'Application

L'application offre plusieurs fonctionnalités clés pour la manipulation et la visualisation des graphes distribués :

- **Création de Graphes** : Les utilisateurs peuvent créer des graphes en ajoutant des nœuds et des arêtes de manière interactive.
- **Visualisation des Graphes** : Les graphes sont affichés de manière claire et interactive, avec des options de zoom et de déplacement.

— **Manipulation des Graphes** : Les utilisateurs peuvent modifier les graphes en déplaçant des nœuds, en ajoutant ou en supprimant des arêtes, et en modifiant les propriétés des nœuds et des arêtes.
— **Sauvegarde et Chargement** : Les graphes peuvent être sauvegardés dans la base de données et rechargés ultérieurement.
— **Algorithmes sur les Graphes** : L'application permet d'exécuter des algorithmes de graphes (comme le parcours en largeur, en profondeur, ou la détection de cycles) et d'afficher les résultats.

8.5 Développement du Backend avec Django

Le backend de l'application a été développé en utilisant Django, un framework web Python. Django gère la logique métier, la gestion des données, et les API nécessaires pour interagir avec le frontend.

— **Modèles** : Les modèles Django représentent les graphes, les nœuds, et les arêtes. Chaque graphe est composé de plusieurs nœuds et arêtes, stockés dans la base de données.
— **Vues** : Les vues Django gèrent les requêtes HTTP et renvoient les réponses appropriées. Elles sont utilisées pour créer, lire, mettre à jour, et supprimer des graphes.
— **API** : Des API REST ont été développées pour permettre au frontend d'interagir avec le backend. Ces API permettent de récupérer, créer, modifier, et supprimer des graphes.

8.6 Développement du Frontend avec HTML, JavaScript et GoJS

Le frontend de l'application a été développé en utilisant HTML pour la structure, JavaScript pour la logique interactive, et GoJS pour la représentation graphique des graphes.

— **HTML** : La structure de base de l'interface utilisateur est définie en HTML. Cela inclut les formulaires pour créer et modifier des graphes, ainsi que la zone d'affichage du graphe.
— **JavaScript** : JavaScript est utilisé pour gérer les interactions utilisateur, comme le clic sur un nœud pour le déplacer, ou le clic sur un bouton pour exécuter un algorithme. Il est également utilisé pour interagir avec les API Django.
— **GoJS** : GoJS est une bibliothèque JavaScript puissante pour la création de diagrammes interactifs. Elle a été utilisée pour représenter les graphes de manière visuelle, avec des options de zoom, de déplacement, et de modification interactive.

8.7 Intégration de GoJS

GoJS a été intégré dans l'application pour fournir une interface graphique interactive pour les graphes. Voici comment GoJS a été utilisé :

- **Initialisation du Diagramme** : Un diagramme GoJS est initialisé dans une zone HTML spécifique. Ce diagramme est configuré pour permettre l'ajout, la suppression, et la modification des nœuds et des arêtes.
- **Gestion des Événements** : Des gestionnaires d'événements sont définis pour gérer les interactions utilisateur, comme le clic sur un nœud pour le déplacer, ou le double-clic pour modifier ses propriétés.
- **Synchronisation avec le Backend** : Les modifications apportées au graphe dans GoJS sont synchronisées avec le backend Django via des appels API. Cela permet de sauvegarder les modifications dans la base de données.

8.8 Exemple de Code

Voici un exemple de code pour initialiser un diagramme GoJS et gérer les interactions utilisateur :

```
// Initialisation du diagramme GoJS
var $ = go.GraphObject.make;
var diagram = $(go.Diagram, "myDiagramDiv", {
    "undoManager.isEnabled": true,
    "click": function(e) {
        var node = e.diagram.selection.first();
        if (node instanceof go.Node) {
            console.log("Node clicked:", node.data.
               key);
        }
    }
});

// D finition du mod le de donn es
diagram.model = new go.GraphLinksModel(
    [
        { key: "Node1" },
        { key: "Node2" }
    ],
    [
        { from: "Node1", to: "Node2" }
```

```
    ]
);
```

8.9 Conclusion

L'application développée dans le cadre de ce projet permet de visualiser et de manipuler des graphes distribués de manière interactive. En combinant Django pour le backend, HTML et JavaScript pour le frontend, et GoJS pour la représentation graphique, nous avons créé une application robuste et facile à utiliser. Cette application peut être étendue avec des fonctionnalités supplémentaires, comme l'intégration d'algorithmes de graphes plus complexes, ou l'ajout de fonctionnalités de collaboration en temps réel.

9 Expérimentations et Résultats

10 Applications et Perspectives

Les applications possibles incluent :

— Amélioration des protocoles P2P pour assurer une meilleure qualité de service (QoS).
— Extension aux systèmes massivement distribués.
— Développement de stratégies hybrides combinant redondance et prédiction.

11 Conclusion

Ce travail a permis de démontrer l'intérêt de l'apprentissage automatique pour améliorer la robustesse des algorithmes distribués face aux pannes de communication. Les résultats obtenus sont prometteurs et ouvrent la voie à de nouvelles approches pour optimiser les échanges d'informations dans des systèmes distribués.

12 Annexes

— Code source et configurations.
— Compléments théoriques sur les algorithmes et les modèles utilisés.