[Projet] – LLMs-Garous

206.3 Formal and Natural Languages (FNL)

Objectifs

- **But :** Développer en groupe de 3 étudiant.e.s un chatbot qui joue au jeu "Les Loups-Garous de Thiercelieux"
- **Durée totale :** 12 periodes (4 leçons: 21/05, 28/05, 04/06, 11/06)
- **Rendu :** Une implementation de la classe `WerewolfPlayer` qui simule un joueur du jeu "Les Loups-Garous de Thiercelieux".
- **Evaluation:**
  - 25% qualité du code, en particulier qualité des prompts
  - 25% pertinence des réponses des joueurs pendant les parties
  - 50% nombre de points gagnés par les joueurs

Introduction

- Nous utiliserons ces règles du jeu légèrement modifiées pour accomoder le setup de notre jeu.
- Il y aura initialement 7 joueurs avec ces rôles: 2 loups-garous, 1 voyante, et 4 villageois. Le but est de monter à 14 joueurs (2 par équipe).
- Les joueurs peuvent gagner un point par partie. Nous allons faire plusieurs parties par tour.

Architecture générale

Le fichier central à modifier est `werewolf.py`. Ce fichier est utilisé par `werewolf_server.py` pour exposer une API REST pour chaque joueur. Le meneur de jeu (`game_leader.py`) appelle l'API de chaque joueur pour les informer des événements de la partie et recevoir les actions des joueurs.

Vous pouvez ainsi lancer les joueurs avec `python werewolf_server.py` qui va lancer 7 joueurs (services REST) sur les port 5021 à 5027.

Vous lancerez ensuite le meneur de jeu avec `python game_leader.py` qui va gérer les événements de la partie et appeler les API des joueurs.

Le seul endroit à modifier pour vous est dans `werewolf.py`, où vous implémenterez la classe `WerewolfPlayer`.

Communication

Le **meneur** gère les événements suivants:

- la distribution des rôles (aléatoire) au début du jeu.

- le flux de la conversation entre les joueurs (prise de parole, demande de parole, demande d'interruption)
- la gestion des phases de jeu (jour, nuit)
- la gestion des votes
- la propagation de rumeurs (pour pimenter la partie)
- modération: les joueurs qui insultent ou font preuve d'agressivité sont éliminés.

Les **joueurs** utilisent les fonctions suivantes (dans `WerewolfPlayer`):

```python
    def __init__(self, name: str, role: str, players_names: List[str], werewolves_count: int, werewolves: List[str]) -> None:

    def speak(self) -> str:

    def notify(self, message: str) -> Intent:
```

- `__init__()` : le meneur appelle cet endpoint pour créer une nouvelle partie
- `speak()` : le meneur appelle cet endpoint quand il donne la parole au joueur
- `notify()` : le meneur appelle cet endpoint pour
    1. donner des informations au joueur *sous forme de texte* (rôle, qui a parlé et qu'est-ce qu'il a dit, qui a été tué, etc. Voir les messages du meneur ci-dessous)
    2. recevoir les actions du joueur *sous forme d'un Intent* (prendre la parole, interrompre, voter, etc.)

Regardez maintenant très attentivement le code dans `werewolf.py` pour la description de la classe `WerewolfPlayer`. J'ai pris soin de bien la documenter.

Messages du meneur

Le meneur envoie des messages aux joueurs, qui les reçoivent avec la fonction `notify()`. Voici la liste **exhaustive** et **exacte**[1] des messages que le meneur peut envoyer avec `return Intent(want_to_speak=..., want_to_interrupt=..., vote_for=...)`:

Messages autour de la voyante:

- "La Voyante se réveille, et désigne un joueur dont elle veut sonder la véritable personnalité !" -> envoyé à tout le monde; si vous êtes la voyante, vous devez vote_for pour demander le rôle d'un joueur à sonder; les autres joueurs ne doivent pas répondre (return Intent())
- "Le rôle de {player_to_check} est {player_to_check_role}" -> rien besoin de répondre (return Intent()). Ce message est envoyé *uniquement à la voyante*.

Messages autour des loup-garous:

- "Les Loups-Garous se réveillent, se reconnaissent et désignent une nouvelle victime !!!" -> envoyé à tout le monde; rien besoin de répondre (return Intent())
- "Les Loups-Garous votent pour une nouvelle victime !!! {last_vote}" -> envoyé seulement aux loup-garous; vote_for pour désigner une nouvelle victime. {last_vote} est le résultat du dernier vote, par exemple: "Dernier vote: Aline a voté pour Benjamin, Benjamin a voté pour Frédéric"

Messages à tout le monde:

- "C'est la nuit, tout le village s'endort, les joueurs ferment les yeux." -> rien besoin de répondre (return Intent())

[1] Il n'y a pas de variation possible dans ces messages, sauf les *rumeurs* qui peuvent comporter n'importe quelle information.

- “C’est le matin, le village se réveille, tout le monde se réveille et ouvre les yeux… Cette nuit, {victim.name} a été mangé.e par les loups-garous. Son rôle était {victim.role}. {rumors}” -> want_to_speak pour parler et/ou want_to_interrupt pour interrompre
- “C’est le matin, le village se réveille, tout le monde se réveille et ouvre les yeux… Cette nuit, personne n’a été mangé.e par les loups-garous. {rumors}” -> want_to_speak pour parler et/ou want_to_interrupt pour interrompre
- “Le vote va bientôt commencer. Chaque joueur peut encore prendre la parole s’il le souhaite.” -> want_to_speak pour parler et/ou want_to_interrupt pour interrompre
- “Il est temps de voter. Donnez maintenant votre intention de vote.” -> vote_for pour voter
- “{msg_voted_for}. Ainsi, {victim.name} est mort(e) et son rôle était {victim.role}.” -> rien besoin de répondre (return Intent()); {msg_voted_for} contient le dernier vote, par exemple: “Dernier vote: Aline a voté pour Benjamin, Benjamin a voté pour Frédéric”
- “{msg_voted_for}. Il n’y a pas de victime.”-> rien besoin de répondre (return Intent())
- “{speaker.name} a dit: {speech}” -> want_to_speak pour parler et/ou want_to_interrupt pour interrompre
- “{speaker.name} avec le rôle {speaker.role} n’a pas répondu à temps. Il/elle a été éliminé de la partie.” -> want_to_speak pour parler et/ou want_to_interrupt pour interrompre

Les valeurs possibles pour les rôles sont `villageois`, `voyante`, `loup-garou`.

Flux d’un tour

De manière simplifiée, le flux d’un tour de parole est le suivant:

1. Le meneur **sélectionne** le prochain joueur à parler.
2. Le meneur **donne la parole** (`speak()`) au joueur sélectionné.
3. Le joueur sélectionné **répond** au meneur avec son `speech`.
4. Le meneur appelle tous les *autres* joueurs (`notify()`) avec le message du joueur sélectionné.
5. Chaque joueur répond avec des flags (`want_to_speak`, `want_to_interrupt`, `vote_for`).
6. Retour à l’étape 1 jusqu’au vote, qui se déclenche également via un `notify`.

Viennent ensuite les particularités suivantes:

Les phases de nuit et de jour:
- le meneur décide quand changer de phase et l’annonce aux joueurs via (`notify()`), par exemple: ” C’est la nuit, tout le monde s’endort. Les loups-garous se réveillent.”
- pendant la nuit, seuls les loups-garous seront appelés à parler (`notify()`).

Les votes:
- le meneur décide quand lancer le vote (plus aucune demande de parole ou nombre de messages dans la phase dépasse un max: MAX_ROUNDS) - il annonce aux joueurs via (`notify()`), par exemple: `"Le vote va bientôt commencer"`…
- il laisse la possibilité de faire une dernier prise de parole à chaque joueur si ceux-ci le souhaitent. - les joueurs renvoient leurs votes via `notify()` avec le flag `vote_for`
- le meneur collecte les votes valides et calcule le résultat du vote
- c’est le meneur qui annonce le résultat du vote et le rôle du joueur éliminé

La gestion des rumeurs: Le meneur peut au matin injecter des rumeurs (inventées ou semi-vraies) dans le jeu via (`notify()`), par exemple: “On a entendu un bruit du côté de Chloé cette nuit.”

Gestion de la parole

Principes directeurs:

- **Interruption** : demandée avec le flag `want_to_interrupt`. Autorisée très rarement (2×/partie/joueur). Le meneur a le droit de refuser une interruption de parole. Le meneur refusera la requête si le quota est épuisé.
- **Main levée** uniquement : seuls les joueurs ayant demandé la parole (avec le flag `want_to_speak`) sont priorisés.
- **Interaction directe** : priorité aux joueurs mentionnés/accusés récemment pour leur permettre de répondre. TODO pas encore implémenté
- **Silence prolongé** : joueurs n'ayant pas parlé depuis plusieurs tours gagnent progressivement en probabilité de parole.
- **Équité** : le meneur évite absolument deux prises de parole consécutives d'un même joueur.

Gestion du temps et pénalisation

Les joueurs qui répondent trop lentement seront pénalisés. TODO pour l'instant max 4s hardcoded.

Cas de tricherie:

- Si un joueur fait preuve d'agressivité ou d'insultes, il est automatiquement éliminé de la partie. TODO: implémenter ou faire manuellement
- Si un joueur tente d'extraire des informations du meneur de jeu, il est automatiquement éliminé de la partie.
- Si une équipe communique avec son `WerewolfPlayer`, par exemple en lui révélant des informations sur ses rôles, elle est automatiquement éliminée de toutes les parties. TODO: implémenter ou faire manuellement
- Si une équipe utilise l'API d'OpenAI dans un but autre que de jouer au jeu, ou de manière abusive, elle est automatiquement éliminée de toutes les parties. TODO: check audit logs after games

Groupes

Les groupes sont formés de 3 étudiant.e.s et sont assignés par l'enseignant selon un critère d'hétérogénéité basé sur les notes des étudiant.e.s. au CC.

Chaque groupe crée un chatbot. Les noms des joueurs sont:

- **A**line
- **B**enjamin
- **C**hloe
- **D**avid
- **E**lise
- **F**rédéric
- **G**abrielle
- **H**ugo
- **I**nès
- **J**ulien
- **K**arine
- **L**éo
- **M**anon
- **N**oé

Déroulement du projet

21/05: Présentation du projet et prise en main.

28/05, 14h: Premier tour de jeu avec les joueurs qui sont prêts. Critère pour pouvoir participer: avoir une démo tournant avec 7 versions de votre chatbot qui converge, validé par l'enseignant.

04/06, 14h: Deuxième tour de jeu. Tous les joueurs doivent participer.

11/06, 14h: Dernier tour de jeu avec tous les joueurs. Si la classe a bien avancé à ce moment-là, possibilité de faire une démo le vendredi sur la pause de midi ouverte au autres étudiant.e.s et enseignant.e.s (similaire à la démo de votre algo de jeu d'échec).

TODOs

- tuner l'algorithme de sélection du joueur à parler
- ajouter des rumeurs
- tester tunnelling avec ngrok pour pouvoir faire des démos en ligne
- webapp pour visualiser la partie