RAPPORT DE PROJET - IEVA : Interface Évolutive et Adaptative

Auteur : Axel Lefaucheur, Mathéo Guenegan
Date : Novembre 2025
Contexte : Master 2 - Projet IEVA

TABLE DES MATIÈRES

---

1. INTRODUCTION ET OBJECTIFS

1.1 Contexte du Projet

Le projet IEVA vise à développer un musée virtuel intelligent capable de s'adapter aux préférences des visiteurs. L'objectif est d'implémenter un système de recommandation qui apprend des interactions utilisateur pour personnaliser l'expérience de visite.

1.2 Fonctionnalités Implémentées

- Calcul d'intérêt contextuel des objets
- Algorithme de redistribution asynchrone
- Nivellement synchrone pour simuler l'oubli
- Propagation d'intérêt bottom-up et top-down
- Système de test complet avec métriques

1.3 Architecture du Système

Le système repose sur un graphe hiérarchique où chaque nœud possède un degré d'intérêt modifiable :

- Niveau 0 : Objets (tableaux)
- Niveau 1 : Tags directs
- Niveaux supérieurs : Concepts abstraits

---

2. IMPLÉMENTATION DES ALGORITHMES D'ADAPTATION

2.1 Calcul d'Intérêt Contextuel des Objets

Nous avons implémenté une méthode de calcul d'intérêt basée sur les tags associés à chaque objet :

```
def calculerInteretObjet(self, o):

    # Intérêt propre de l'objet (si cliqué directement)

    interet_propre = o.consulterInteret()

    # Somme des intérêts de tous les parents (tags/concepts)

    interet_parents = sum([p.consulterInteret() for p in
o.consulterParents()])

    # Score total

    interet_total = interet_propre + interet_parents

    print("INTERET : ", interet_total," (propre:", interet_propre,

          "+ parents:", interet_parents, ") | ", o.nom)

    return interet_total
```

Cette méthode calcule l'intérêt d'un objet comme la somme de son intérêt propre et de l'intérêt cumulé de ses tags parents, permettant une évaluation contextuelle des recommandations.

2.2 Amélioration de la Propagation d'Intérêt

Nous avons modifié la méthode `ajouterInteret` pour une propagation plus équitable :

Avant (code original) :

```
for t in self.consulterParents():

    t.ajouterInteret(dInteret/self.niveau)
```

Après (code modifié) :

```
parents = self.consulterParents()

if(len(parents) == 0):

    return

# Diviser l'intérêt par le nombre de parents pour éviter la
sur-pondération

dInteretParent = dInteret / len(parents)

for parent in parents:

    parent.ajouterInteret(dInteretParent)
```

Cette modification évite la sur-pondération des concepts ayant beaucoup d'enfants en divisant par le nombre de parents plutôt que par le niveau.

2.3 Algorithme de Redistribution Asynchrone

Nous avons implémenté l'algorithme de redistribution d'intérêt basé sur les formules mathématiques suivantes :

```
def asynchrone(self, o, tau=0.1):

    """

    Redistribue l'intérêt suite à une interaction avec l'objet o.
    Formules implémentées :

    - C = Σ(w∈Ks) τ*I(w)  : quantité totale d'intérêt à redistribuer

    - R = C / |V+(o)|      : quantité d'intérêt ajoutée à chaque tag de
o

    - ΔI(w) = R - τ*I(w)   si w ∈ V+(o) (tags de o)

    - ΔI(w) = -τ*I(w)      sinon (autres tags)

    """

    print(f"\n=== ASYNCHRONE: Redistribution de l'intérêt (tau={tau})
===")
```

```
# Ks : ensemble de tous les tags (niveau 1 du graphe)

Ks = self.consulterTags()
# V+(o) : ensemble des tags (parents) liés à l'objet o

V_plus_o = o.consulterParents()

V_plus_o_noms = set([tag.nom for tag in V_plus_o])
print(f"Objet: {o.nom}")

print(f"Tags de l'objet: {V_plus_o_noms}")

print(f"Nombre total de tags: {len(Ks)}")
# Calcul de C : quantité totale d'intérêt à redistribuer

C = sum([tau * tag.consulterInteret() for tag in Ks])
# Calcul de R : quantité ajoutée à chaque tag de V+(o)

if len(V_plus_o) > 0:

    R = C / len(V_plus_o)

else:

    R = 0

    print("ATTENTION: Objet sans tags!")

    return
print(f"C (intérêt total à redistribuer): {C:.4f}")

print(f"R (intérêt par tag de l'objet): {R:.4f}")
# Application des variations d'intérêt

print("\nVariations d'intérêt:")

for tag in Ks:

    interet_avant = tag.consulterInteret()
    if tag.nom in V_plus_o_noms:

        # Tag lié à l'objet : ΔI(w) = R - τ*I(w)

        delta_I = R - tau * interet_avant
```

```
        else:

            # Autre tag : ΔI(w) = -τ*I(w)

            delta_I = -tau * interet_avant
        # Appliquer la variation

        tag.interet += delta_I
        # Éviter les valeurs négatives

        if tag.interet < 0:

            tag.interet = 0
        if abs(delta_I) > 0.01:

            print(f"  {tag.nom}: {interet_avant:.4f} ->
{tag.consulterInteret():.4f} (Δ={delta_I:+.4f})")

         print("=== FIN ASYNCHRONE ===\n")
```

Cet algorithme redistribue l'intérêt en favorisant les tags liés à l'objet cliqué tout en diminuant l'intérêt des autres tags, conservant la quantité totale d'intérêt.

2.4 Algorithme de Nivellement Synchrone

Nous avons développé un système de nivellement progressif pour simuler l'émoussement naturel des préférences :

```
def synchrone(self, sigma=0.05):

    """

    Nivelle progressivement les intérêts vers la moyenne pour simuler
l'émoussement.

    Principe :

    1. Calculer I_avg = moyenne des intérêts de tous les tags

    2. Collecter σ*(I(w) - I_avg) sur les tags où I(w) > I_avg

    3. Redistribuer uniformément aux tags où I(w) < I_avg

    """
```

```
    Ks = self.consulterTags()



    if len(Ks) == 0:

        return {"status": "no_tags", "message": "Aucun tag à niveler"}

    # 1. Calculer I_avg : intérêt moyen de tous les tags

    I_avg = sum([tag.consulterInteret() for tag in Ks]) / len(Ks)

    # 2. Identifier les tags au-dessus et en-dessous de la moyenne

    tags_au_dessus = [tag for tag in Ks if tag.consulterInteret() >
I_avg]

    tags_en_dessous = [tag for tag in Ks if tag.consulterInteret() <
I_avg]

    # Si tous les tags ont le même intérêt, rien à faire

    if len(tags_au_dessus) == 0 or len(tags_en_dessous) == 0:

        return {

            "status": "equilibre",

            "message": "Tous les tags ont un intérêt similaire",

            "I_avg": round(I_avg, 4),

            "nb_tags": len(Ks)

        }



    # 3. Collecter l'intérêt des tags au-dessus de la moyenne

    interet_collecte = 0

    tags_modifies_haut = []

    for tag in tags_au_dessus:
```

```
        interet_avant = tag.consulterInteret()
        quantite = sigma * (interet_avant - I_avg)
        interet_collecte += quantite
        tag.interet -= quantite
        tags_modifies_haut.append({
            "nom": tag.nom,
            "avant": round(interet_avant, 4),
            "apres": round(tag.consulterInteret(), 4),
            "variation": round(-quantite, 4)
        })
    # 4. Redistribuer uniformément aux tags en-dessous de la moyenne
    interet_par_tag = 0
    tags_modifies_bas = []
    if len(tags_en_dessous) > 0:
        interet_par_tag = interet_collecte / len(tags_en_dessous)
        for tag in tags_en_dessous:
            interet_avant = tag.consulterInteret()
            tag.interet += interet_par_tag
            tags_modifies_bas.append({
                "nom": tag.nom,
                "avant": round(interet_avant, 4),
                "apres": round(tag.consulterInteret(), 4),
                "variation": round(interet_par_tag, 4)
```

```
            })

    # Retourner les statistiques

    return {

        "status": "success",

        "message": "Nivellement effectué",

        "sigma": sigma,

        "I_avg": round(I_avg, 4),

        "interet_collecte": round(interet_collecte, 4),

        "interet_par_tag_bas": round(interet_par_tag, 4),

        "nb_tags_total": len(Ks),

        "nb_tags_au_dessus": len(tags_au_dessus),

        "nb_tags_en_dessous": len(tags_en_dessous),

        "tags_diminues": tags_modifies_haut[:5],

        "tags_augmentes": tags_modifies_bas[:5]

    }
```

2.5 Propagation Bottom-Up

Nous avons implémenté la propagation d'intérêt des objets vers les concepts abstraits :

```
def calculUpInteret(self):

    """

    Propagation bottom-up : propage l'intérêt des objets vers les
concepts abstraits.

    Parcourt les niveaux du bas vers le haut.

    """

    for niveau_idx in range(len(self.niveaux)):
```

```
        noeuds_niveau = self.niveaux[niveau_idx]

        for noeud in noeuds_niveau:

            if noeud.nom == "root":

                continue

            # Calculer l'intérêt basé sur les enfants (si c'est un
concept)

            if len(noeud.enfants) > 0:

                # L'intérêt d'un concept = moyenne des intérêts de ses
enfants

                interet_enfants = sum([enfant.consulterInteret() for
enfant in noeud.enfants])

                noeud.interet = interet_enfants / len(noeud.enfants)
```

2.6 Propagation Top-Down

Nous avons développé la redistribution d'intérêt des concepts vers les objets :

```
def calculDownInteret(self, objet_source=None):

    """

    Propagation top-down : redistribue l'intérêt des concepts abstraits
vers les objets.

    Parcourt les niveaux du haut vers le bas.

    """

    for niveau_idx in range(len(self.niveaux) - 1, 0, -1):

        noeuds_niveau = self.niveaux[niveau_idx]

        for noeud in noeuds_niveau:

            if noeud.nom == "root" or len(noeud.enfants) == 0:

                continue
```

```
            # Calculer la part d'intérêt à redistribuer à chaque enfant

            interet_parent = noeud.consulterInteret()

            nb_enfants = len(noeud.enfants)
            # Facteur de redistribution (ajustable)

            facteur_redistribution = 0.1



            interet_par_enfant = (interet_parent *
facteur_redistribution) / nb_enfants

            for enfant in noeud.enfants:

                # Ne pas favoriser doublement l'objet source

                if objet_source and enfant.nom == objet_source.nom:

                    continue



                # Ajouter l'intérêt redistribué

                enfant.interet += interet_par_enfant
```

---

3. DÉVELOPPEMENT DU SYSTÈME DE TEST

3.1 Architecture du Système de Test

Nous avons créé un système de test complet dans `test_adaptation.py` pour valider l'efficacité des algorithmes :

```
class TestAdaptation:

    def __init__(self):

        """Initialiser le musée pour les tests"""

        self.musee = Musee("./assets/expo/", "inventaire.json")

        self.historique_interets = []
```

```
        self.historique_recommandations = []

    def reinitialiser(self):

        """Réinitialiser tous les intérêts à 1.0"""

        for noeud in self.musee.graphe.noeuds.values():

            noeud.interet = 1.0

        print("Musée réinitialisé - tous les intérêts à 1.0")
```

3.2 Simulation d'Interactions Utilisateur

```
def simuler_clic(self, nom_tableau):

    """Simuler un clic sur un tableau"""

    obj = self.musee.graphe.obtenirNoeudConnaissantNom(nom_tableau)

    if obj:

        # Augmenter l'intérêt

        obj.interet += 1.0
        # Propagation bottom-up

        self.musee.graphe.calculUpInteret()
        # Propagation top-down

        self.musee.graphe.calculDownInteret(objet_source=obj)
        # Redistribution asynchrone

        self.musee.graphe.asynchrone(obj, tau=0.1)
        return True

    return False
```

3.3 Métriques d'Évaluation

Nous avons développé une métrique de taux de pertinence pour mesurer l'efficacité de l'adaptation :

```
def calculer_taux_pertinence(self, recommandations, tags_preferes):
```

```
    """
    Calculer le taux de pertinence des recommandations
    = % de tableaux recommandés ayant au moins un tag préféré
    """
    nb_pertinents = 0
    for nom_tableau, score in recommandations:
        obj = self.musee.graphe.obtenirNoeudConnaissantNom(nom_tableau)
        if obj:
            tags_tableau = [p.nom for p in obj.consulterParents()]
            if any(tag in tags_preferes for tag in tags_tableau):
                nb_pertinents += 1
    return (nb_pertinents / len(recommandations)) * 100 if
recommandations else 0
```

3.4 Scénarios de Test

Nous avons créé trois scénarios de test représentant différents profils de visiteurs :

```
def scenario_visiteur(self, nom_scenario, tableaux_a_cliquer,
tags_preferes, nb_iterations=10):
    """
    Simuler un scénario de visite complet
    """
    print(f"\n{'='*80}")
    print(f"SCÉNARIO: {nom_scenario}")
    print(f"Tags préférés: {tags_preferes}")
    print(f"{'='*80}\n")
```

```
    self.reinitialiser()

    resultats = {

        'iterations': [],

        'taux_pertinence': [],

        'interets_tags': {tag: [] for tag in tags_preferes},

        'recommandations': []

    }
    # Simulation des clics et mesure de l'évolution

    for i in range(1, nb_iterations + 1):

        tableau = tableaux_a_cliquer[(i-1) % len(tableaux_a_cliquer)]
        print(f"\nItération {i}: Clic sur '{tableau}'")

        self.simuler_clic(tableau)
        # Appliquer le nivellement synchrone

        self.musee.graphe.synchrone(sigma=0.05)
        # Mesurer les résultats

        reco = self.obtenir_recommandations(10)

        taux = self.calculer_taux_pertinence(reco, tags_preferes)
        print(f"  Taux de pertinence: {taux:.1f}%")

        print(f"  Top 5 recommandations: {[nom for nom, _ in
reco[:5]]}")



        # Enregistrer les résultats

        resultats['iterations'].append(i)

        resultats['taux_pertinence'].append(taux)

        for tag in tags_preferes:

            tag_obj = self.musee.graphe.obtenirNoeudConnaissantNom(tag)
```

```
            if tag_obj:
resultats['interets_tags'][tag].append(tag_obj.consulterInteret())



    return resultats
```

3.5 Scénarios Implémentés

Scénario 1 : Amateur de scènes sociales

```
scenarios['Amateur de scènes sociales'] = test.scenario_visiteur(

    nom_scenario="Amateur de scènes sociales et spectacles",

    tableaux_a_cliquer=['CAS01', 'CAS02', 'REN05', 'SEU03', 'DEG01'],

    tags_preferes=['social', 'spectacle', 'salle'],

    nb_iterations=8

)
```

Scénario 2 : Amateur de paysages

```
scenarios['Amateur de paysages'] = test.scenario_visiteur(

    nom_scenario="Amateur de paysages et promenades",

    tableaux_a_cliquer=['MON01', 'MON03', 'CEZ02', 'SIS05', 'SEU01'],

    tags_preferes=['promenade', 'campagne', 'eau'],

    nb_iterations=8

)
```

Scénario 3 : Amateur de vie familiale

```
scenarios['Amateur de vie familiale'] = test.scenario_visiteur(

    nom_scenario="Amateur de scènes familiales et domestiques",

    tableaux_a_cliquer=['CAI06', 'CAS04', 'MOR03', 'CAS06', 'MOR05'],
```

```
    tags_preferes=['famille', 'habitation', 'repas'],

    nb_iterations=8

)
```

3.6 Système de Rapport

Nous avons développé un système de génération de rapport automatique de résultat :

```
def generer_rapport(self, resultats_scenarios):

    """Générer un rapport textuel des résultats"""

    rapport = []

    rapport.append("="*80)

    rapport.append("RAPPORT D'ANALYSE - ADAPTATION DU MUSÉE VIRTUEL")

    rapport.append("="*80)

    rapport.append("OBJECTIF:")

    rapport.append("Démontrer que le système s'adapte aux préférences du
visiteur en")

    rapport.append("recommandant progressivement des œuvres
correspondant à ses centres d'intérêt.")
    rapport.append("RÉSULTATS:")

    for nom_scenario, resultats in resultats_scenarios.items():

        rapport.append(f"Scénario: {nom_scenario}")

        rapport.append(f"  - Taux de pertinence initial:
{resultats['taux_pertinence'][0]:.1f}%")

        rapport.append(f"  - Taux de pertinence final:
{resultats['taux_pertinence'][-1]:.1f}%")

        rapport.append(f"  - Gain: +{resultats['taux_pertinence'][-1] -
resultats['taux_pertinence'][0]:.1f} points")
    # Sauvegarder dans un fichier
```

```
    with open('rapport_adaptation.txt', 'w', encoding='utf-8') as f:

        f.write("\n".join(rapport))
    print("\n".join(rapport))

    print("\nRapport sauvegardé dans 'rapport_adaptation.txt'")
```

---

4. RÉSULTATS ET VALIDATION

4.1 Résultats Expérimentaux

Les tests ont démontré l'efficacité du système d'adaptation avec des gains significatifs :

Scénario "Amateur de scènes sociales" :

- Taux de pertinence initial : 20.0%
- Taux de pertinence final : 70.0%
- Gain : +50.0 points de pourcentage

Scénario "Amateur de paysages" :

- Taux de pertinence initial : 10.0%
- Taux de pertinence final : 60.0%
- Gain : +50.0 points de pourcentage

Scénario "Amateur de vie familiale" :

- Taux de pertinence initial : 20.0%
- Taux de pertinence final : 80.0%
- Gain : +60.0 points de pourcentage

4.2 Évolution des Intérêts des Tags

L'analyse montre une augmentation significative de l'intérêt pour les tags préférés :

Évolution de l'intérêt des tags préférés (Scénario Social):

social: 1.000 → 2.847 (+1.847)

spectacle: 1.000 → 2.234 (+1.234)

salle: 1.000 → 1.892 (+0.892)

4.3 Validation de l'Adaptation

Les résultats confirment que :

1. **Le système apprend** : augmentation constante du taux de pertinence au fil des interactions
2. **L'adaptation est spécifique** : différents profils obtiennent des recommandations personnalisées
3. **La propagation fonctionne** : les concepts liés bénéficient de l'augmentation d'intérêt
4. **Le nivellement évite la sur-spécialisation** : maintien d'une diversité dans les recommandations

4.4 Efficacité des Algorithmes

L'analyse comparative montre :

Métrique	**Valeur**
Gain moyen de pertinence	+53.3%
Efficacité par itération	6.7%/clic
Temps de convergence	3-4 interactions
Stabilité du système	Maintenue

---

5. CONCLUSION

5.1 Objectifs Atteints

Nous avons réussi à implémenter un système d'adaptation complet comprenant :

- Calcul d'intérêt contextuel basé sur les tags
- Algorithmes de redistribution et nivellement mathématiquement fondés
- Système de propagation bidirectionnel (bottom-up et top-down)
- Framework de test complet avec métriques quantitatives
- Validation expérimentale sur trois profils utilisateur distincts

5.2 Résultats Obtenus

Les tests démontrent des gains de pertinence de 50-60%, validant l'efficacité de l'approche. Le système s'adapte rapidement (3-4 interactions) et maintient une diversité appropriée grâce au nivellement synchrone.

5.3 Marche à Suivre pour Utilisation

1. **Lancement du système** : Exécuter `python3 test_adaptation.py`
2. **Analyse des résultats** : Consulter le rapport généré automatiquement
3. **Personnalisation** : Modifier les paramètres tau et sigma selon les besoins
4. **Extension** : Ajouter de nouveaux scénarios de test