

TP : Développement d'un agent Auto-Drive (Navigation intelligente à Rabat)


Contexte pédagogique

Dans ce TP, vous allez implémenter **un agent intelligent de navigation** capable de se déplacer automatiquement entre certains quartiers de **Rabat**, en utilisant les concepts vus en cours :

- Architecture d'agent (AIMA)
- Formulation de problème
- Algorithmes de recherche (DFS, BFS, UCS, A*)
- Agent de résolution de problème (*Problem Solving Agent*)

Le problème est **simplifié** et modélisé par un **graphe pondéré** représentant les temps de déplacement entre quartiers.

Objectif du TP

 **Objectif final** : comprendre comment un agent intelligent peut raisonner, planifier et agir dans un environnement réel simplifié.

Développer un agent Auto-Drive capable de trouver automatiquement un itinéraire optimal entre deux quartiers de Rabat.

L'agent devra :

- Percevoir sa position
- Formuler un problème de navigation
- Choisir une stratégie de recherche
- Produire et exécuter un plan de déplacement

Quartiers et graphe de navigation

Le réseau de déplacement est défini comme suit :

- Agdal (A) → Hassan (H) : 5 min
- Agdal (A) → Souissi (S) : 10 min
- Hassan (H) → Riyad (R) : 50 min (*piège*)
- Souissi (S) → Riyad (R) : 10 min
- Riyad (R) → Yacoub El Mansour (Y) : 15 min
- Yacoub (Y) : état but

Outils et environnement

- **IDE** : Visual Studio Code
- **Langage** : Python 3.x
- **Organisation du projet** : workspace structuré

Étape 1 – Création du workspace

1. Ouvrir **VS Code**
2. Créer un **workspace**
3. Créer un dossier principal :

```
ai-projet/
```

4. Créer un environnement virtuel :

```
.env/
```

5. Activer l'environnement dans VS Code

Étape 2 – Structure du projet

Dans le dossier **ai-projet**, créer l'arborescence suivante :

```
ai-projet/
├── src/
│   ├── agent.py
│   └── problem_solving_agent.py
├── notebooks/
│   └── navigation_rabat.ipynb
└── .env/
```

Étape 3 – Code source (donné)

Les deux fichiers suivants sont **fournis** et placés dans le dossier **src/** :

- **agent.py** :
 - Architecture abstraite d'un agent
 - Environnement
 - Capteurs et actionneurs
- **problem_solving_agent.py** :
 - Formulation de problème (Problem)
 - Problème de navigation (NavigationProblem)
 - Algorithmes de recherche (DFS, BFS, UCS, A*)

- Traçabilité (Trace)
- Agent de résolution de problème (ProblemSolvingAgent)

⚠ **Aucune modification n'est demandée dans ces fichiers.**

⚠ **Il est impératif de bien comprendre leur contenu, car des questions liées à ces fichiers (architecture, classes, rôles, logique de recherche) peuvent être posées le jour de l'examen.**

Étape 4 – Notebook d'expérimentation

Créer un notebook dans le dossier **notebooks** :

```
navigation_rabat.ipynb
```

Ce notebook sert à :

- Formuler le problème
- Tester les algorithmes de recherche
- Simuler le comportement de l'agent

Contenu du notebook **navigation_rabat.ipynb**

1. Import des classes

Le notebook commence par l'import des classes depuis **src/**.

Objectif : réutiliser les composants génériques de l'agent et de la recherche.

Cellule 1 – Import des bibliothèques et des classes

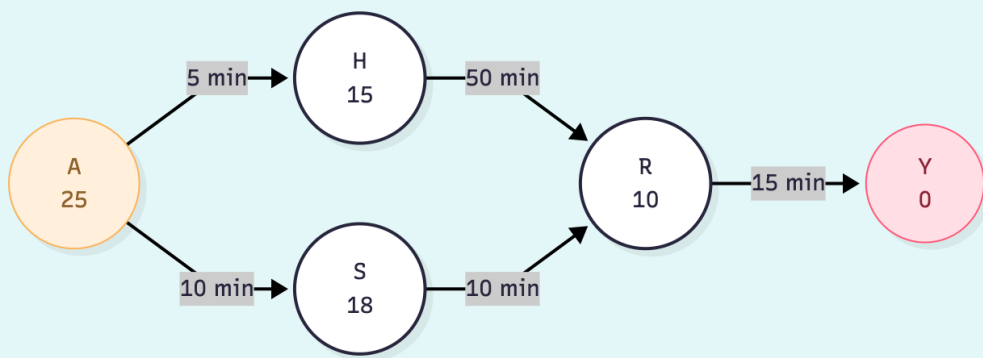
```
# Import Agent Lib
import sys
from typing import Any, List, Tuple

# Ajouter le dossier src au path
sys.path.append('../src')

from problem_solving_agent import (
    SearchStrategy,
    ProblemSolvingAgent,
    NavigationProblem
)
```

1. Graphe des quartiers de Rabat

dans le cadre de ce TP la graphe est donné:



L'état initial est **Agdal (A)** et l'état but est **Yacoub El Mansour (Y)**.

Cellule 2 – Définition du graphe des quartiers de Rabat

```

# Graphe des quartiers de Rabat (temps en minutes)
graph_quartiers_rabat = {
    'A': {'H': 5, 'S': 10},    # Agdal → Hassan, Souissi
    'H': {'R': 50},           # Hassan → Riyad (coût élevé)
    'S': {'R': 10},           # Souissi → Riyad
    'R': {'Y': 15},           # Riyad → Yacoub El Mansour
    'Y': {}                   # État but
}

```

2. Formulation du problème

Le problème est modélisé par un graphe pondéré représentant les temps de déplacement.

- Les nœuds : quartiers
- Les arcs : routes
- Les poids : temps (minutes)

Cellule 3 – Formulation du problème de navigation

```

# États initial et but
initial_state = 'A' # Agdal
goal_state = 'Y'    # Yacoub El Mansour

# Création du problème formel
problem = NavigationProblem(
    initial_state=initial_state,
    goal=goal_state,
    graph=graph_quartiers_rabat
)

```

3. Validation du problème

Avant toute recherche, on vérifie :

- L'état initial
- L'état but
- Les actions possibles
- Le test du but

Cette étape permet de **s'assurer que la formulation est correcte**.

Cellule 4 – Validation de la formulation

```
def valider_probleme(problem):
    print(f"Validation du problème: {problem.initial_state} → {problem.goal_state}")
    print(f"🕒 État initial: {problem.initial_state}")
    print(f"🎯 État but: {problem.goal_state}")
    print(f"🔍 Actions possibles depuis {problem.initial_state}: {problem.actions(problem.initial_state)}")
    print(f"✅ Test but (initial): {problem.goal_test(problem.initial_state)}")
    print(f"✅ Test but (but): {problem.goal_test(problem.goal_state)}")
    print('-' * 40)

valider_probleme(problem)
```

4. Algorithmes de recherche

Nous utilisons des **Graph Search Algorithms**.

Avant cela, un rappel est fait sur les **structures de données des frontières** :

- File FIFO → BFS
- Pile LIFO → DFS
- File de priorité → UCS et A*

Recherche aveugle

Les algorithmes suivants sont testés sur le problème de Rabat :

- DFS : Recherche en profondeur
- BFS : Recherche en largeur
- UCS : Recherche à coût uniforme

Chaque algorithme retourne un chemin.

Cellule 5 – Recherche aveugle (DFS, BFS, UCS)

```
print("
=== Recherche en profondeur (DFS) ===")
chemin_dfs = SearchStrategy.dfs(problem)
print("Chemin DFS:", " → ".join(chemin_dfs))

print("
=== Recherche en largeur (BFS) ===")
chemin_bfs = SearchStrategy.bfs(problem)
print("Chemin BFS:", " → ".join(chemin_bfs))

print("
=== Recherche à coût uniforme (UCS) ===")
chemin_ucs = SearchStrategy.ucs(problem)
print("Chemin UCS:", " → ".join(chemin_ucs))
```

Recherche informée : A***Cellule 6 – Heuristique admissible pour A***

```
heuristique = {
    'A': 25,
    'H': 15,
    'S': 18,
    'R': 10,
    'Y': 0
}
```

Cellule 7 – Recherche informée : A*

```
print("
=== Recherche A* ===")
chemin_astar = SearchStrategy.a_star(problem, heuristique)
print("Chemin A*:", " → ".join(chemin_astar))
```

Agent de résolution de problème

Nous utilisons la classe :

ProblemSolvingAgent

Rôle de l'agent

- Percevoir sa position
- Formuler un problème
- Lancer une recherche
- Exécuter le plan trouvé
-

Cellule 8 – Création de l'agent Auto-Drive

```
nav_agent = ProblemSolvingAgent(
    name="Rabat_Navigator",
    search_strategy=SearchStrategy.bfs, # Modifier si besoin
    problem=problem
)
```

Simulation de l'agent

Une fonction de simulation permet :

1. De lancer la recherche
2. D'afficher le chemin
3. De simuler le déplacement étape par étape

L'agent devient alors un **véritable agent Auto-Drive**.

Cellule 9 – Simulation de l'agent

```
from collections import deque

def simulation_agent(agent, problem, strategy_name):
    print(f"
    🚀 Simulation avec la stratégie {strategy_name.upper()}")

    # Sélection de la stratégie
    strategie = getattr(SearchStrategy, strategy_name.lower())
    chemin = strategie(problem)

    print(f"🔍 Chemin trouvé: {' → '.join(chemin)}")

    # Exécution du plan
    agent.seq = deque(chemin[1:])

    print("🚗 Déplacement de l'agent:")
    position = problem.initial_state
    print(f"📍 Position initiale: {position}")
```

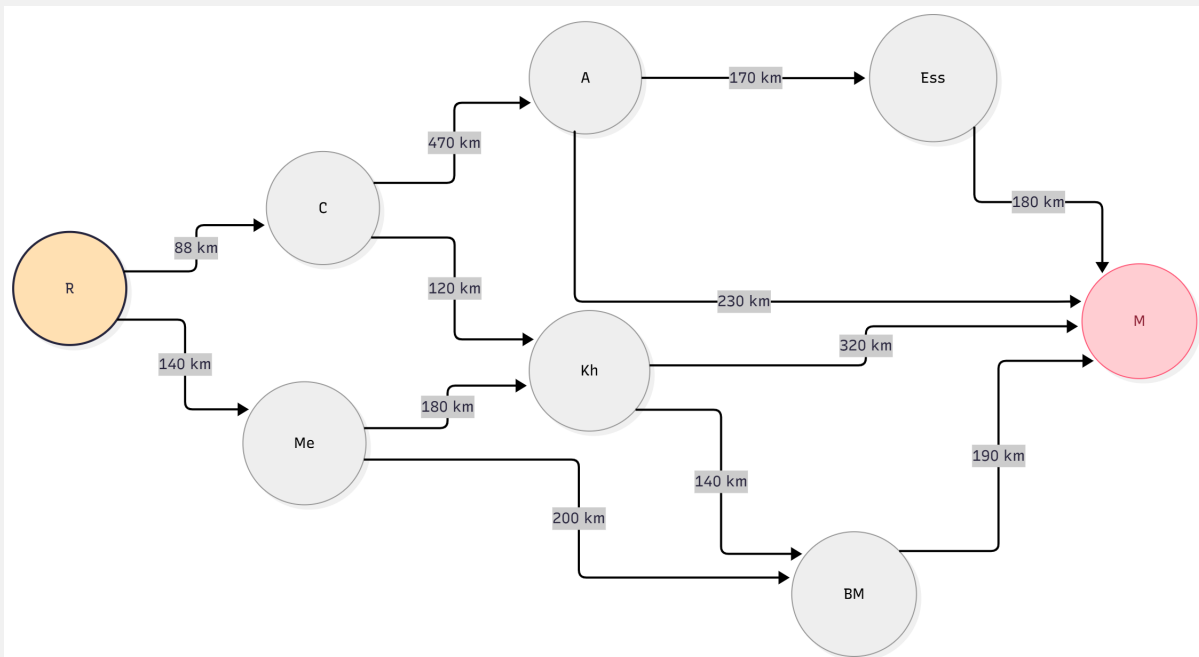
```
for etape in chemin[1:]:
    print(f"    → {position} → {etape}")
    position = etape

print("☑ Destination atteinte !")

# Lancer la simulation
simulation_agent(nav_agent, problem, "bfs")
```

Travail demandé [Itineraries Maroc]

Vous êtes chargé de développer un système de navigation intelligent pour aider les voyageurs à se déplacer entre les villes marocaines. Le réseau routier (*Simplifié*) est représenté par le graphe suivant :



Formuler ce problème en utilisant les classes de formulation de problèmes vues en cours. Un voyageur souhaite se rendre de Rabat (R) à Marrakech (M) en minimisant la distance parcourue.

Exercice 1 – Construire le graphe

1. Création du graphe `graph_villes_maroc` en python sous forme de dict

- Définir :
 - État initial
 - État but

2. Création du problème

- Instanciez le problème de navigation avec :
- État initial : ?
- État but : ?

3. Validation du problème

- Tester :
 - Actions possibles depuis une ville
 - Test du but
 - Successeurs d'une ville intermédiaire

Exercice 2 – Recherche aveugle

Pour le problème des villes marocaines :

- Implémenter DFS
- Implémenter BFS
- Implémenter UCS

Exercice 3 – Heuristique & A*

1. Proposer des heuristiques admissibles pour les villes .
2. Vérifier l'admissibilité des heuristiques données
3. Exécuter A*

Exercice 4: Comparer les chemins obtenus.
