

Intelligence Artificielle (IA2) Rapport

Noah STAPLE
Anaïs ESPICIER

Encadrant: **Karim TAMINE**



Université
de Limoges

Master Cryptis
Université de Limoges

11 *Mai* 2025

1 Explication des algorithmes

1.1 Algorithme A* et Greedy-BFS (Best First Search pour N-Puzzle)

1.1.1 Rappel BFS

L'algorithme **BFS** ou Best First Search permet de résoudre des problèmes d'optimisation à l'aide d'une heuristique, ce qui permet d'approcher une solution optimale à des problèmes qui sont NP ou NP-complets.

1.1.2 Application au N-Puzzle

On utilise l'heuristique $h(s)$ qui compte le nombre de tuiles mal placées dans l'état s .

Fonctionnement de l'algo

1. Initialiser l'ensemble **OPEN** avec l'état initial, priorisé par $h(s)$, et **CLOSED** à vide.
2. Tant que **OPEN** n'est pas vide :
 - (a) Extraire de **OPEN** l'état e de priorité minimale ($h(e)$).
 - (b) Si $h(e) = 0$, la solution est trouvée ; on s'arrête.
 - (c) Ajouter e à **CLOSED**.
 - (d) Générer tous les successeurs e' de e (Tous les déplacements possibles pour notre état actuel).
 - (e) Pour chaque e' , si e' n'est pas déjà dans **CLOSED**, l'insérer dans **OPEN** en calculant sa valeur $h(e')$.

L'algorithme s'arrête dès qu'un état atteint $h = 0$.

Pour préciser, notre heuristique et le fait que l'on ait une liste **OPEN** et **CLOSED** permettent de ne pas rester bloqué car si on explore des successeurs qui mènent à une solution non optimale, on pourra le voir si on garde la liste des anciennes branches qui mènent à d'autres solutions.

1.1.3 Algorithme A*

L'algorithme **A*** combine le coût déjà parcouru et une heuristique pour garantir la solution optimale (longueur minimale) tant que l'heuristique est admissible.

L'évaluation de chaque état e est

$$f(e) = g(e) + h(e),$$

où

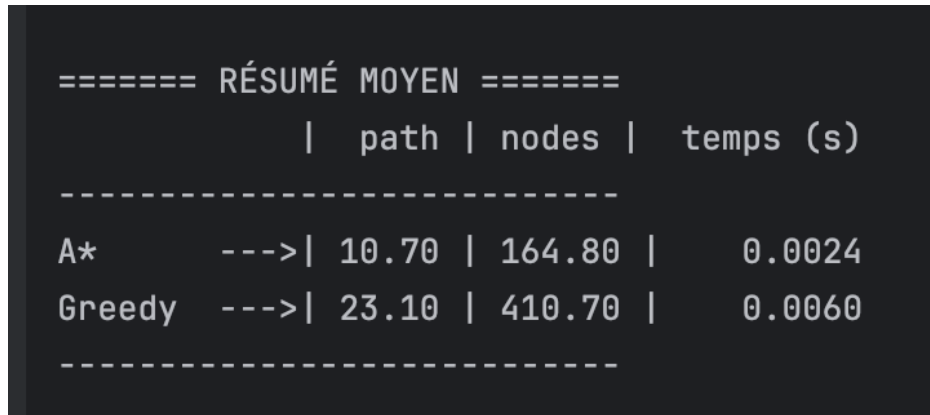
- $g(e)$ est le coût exact depuis l'état initial jusqu'à e (ici : nombre de déplacements effectués);
- $h(e)$ est une estimation du coût restant. Deux heuristiques classiques pour le N-Puzzle sont :
 1. *Tuiles mal placées*: nombre de tuiles dont la valeur diffère de la case but;

2. *Distance de Manhattan*: somme des distances de Manhattan de chaque tuile à sa position finale.

Dnas notre cas du N -Puzzle on a utilisé les tuiles mal placées.

L'algorithme reprend la structure OPEN / CLOSED du Greedy Best-First Search, mais il ordonne **OPEN** avec la clé $f(e)$ plutôt que $h(e)$ seul, ce qui assure l'optimalité de la solution.

1.1.4 Comparaison des deux algorithmes



	path	nodes	temps (s)
A*	---> 10.70	164.80	0.0024
Greedy	---> 23.10	410.70	0.0060

Figure 1: Comparaison entre A* et Greedy-BFS

Le benchmark a été effectué sur 20 puzzles générés aléatoirement.

- **Path**: Compte le nombre moyen d'étapes parcouru avant de trouver la solution.
- **Nodes**: Le nombre de noeuds explorés par chaque algorithme.
- **Temps**: Le temps d'exécution moyen de chaque algorithme.

On voit que A* bat Greedy-BFS dans chaque métrique et surtout dans le nombre de nœuds explorés.

1.2 Algorithme génétique pour le problème du voyageur de commerce

Important: Cet algorithme a été initialement réalisé par Anaïs ESPICIER en collaboration avec le groupe de Tommy Debord et Matthéo Jaouen avant qu'il ne soit décidé qu'elle rejoindrait Noah Staple qui était seul pour réaliser le projet.

1.2.1 Rappels

Le problème du voyageur de commerce (TSP) se formule ainsi :

- **Données** : un ensemble de N villes et, pour chaque paire de villes, une distance (ou un coût) de déplacement.

- **Objectif** : trouver un circuit qui passe une et une seule fois par chacune des N villes et revient au point de départ = **circuit hamiltonien**, tout en minimisant la distance totale parcourue.
- **Complexité** : c'est un problème NP-difficile ; le nombre de circuits possibles croît en $O(N!)$, si bien qu'une recherche exhaustive devient vite inabordable dès que N dépasse 10–12.

Nous allons utiliser dans cette section une méthode de résolution par méta-heuristique qui explore efficacement l'espace des solutions : un algorithme génétique.

1.2.2 Structure générale

Le programme se compose de cinq fichiers principaux :

- `graphe.py` : définition d'un graphe (villes et distances) et méthodes de parcours exhaustif (DFS) pour trouver tous les chemins hamiltoniens.
- `arete.py` : heuristique gloutonne (plus proche voisin) et version récursive brute-force pour comparaison.
- `algo_genetique.py` : implémentation de l'algorithme génétique pour le TSP :
- `main.py` et `main2.py` : programmes d'exemple, l'un pour un algorithme génétique très simple sur un entier cible, l'autre pour résoudre le problème du voyageur de commerce sur un graphe.

1.2.3 Fonctionnement du programme

- **Encodage** (représentation des individus) : chaque individu de la population représente une solution candidate, i.e un circuit hamiltonien sous forme d'une permutation = **liste** de N villes

- **Initialisation** :

```

1 def generate_population(self):
2     pop = []
3     for _ in range(self.taille_population):
4         individu = list(range(self.graphe.nombre_ville))
5         random.shuffle(individu)           # permutation alatoire
6         pop.append(individu)
7         self.distances[tuple(individu)] = self.distance_individu(individu)
8     return pop

```

- **Évaluation** (fitness) = distance du chemin. La fitness est inversement liée à la distance totale : plus la distance est faible, meilleur est l'individu.

```

1 def distance_individu(self, individu):
2     if tuple(individu) in self.distances:
3         return self.distances[tuple(individu)]
4     d = 0
5     for i in range(len(individu)-1):
6         d += self.graphe.villes[individu[i]][individu[i+1]]
7     self.distances[tuple(individu)] = d
8     return d

```

- **Sélection** d'un individu avec une roulette russe en fonction de son évaluation (fitness)

```

1 def selection(self, population, inversed=False):
2     total = sum(self.distance_individu(ind) for ind in population)
3     # Si inversed=True, on transforme en "attractivité" : total distance
4     # Puis on procède à une sélection par roulette russe.
5     # Retourne l'indice de l'individu sélectionné.

```

- **Croisement** : on génère un nombre fixé (hyperparamètre du modèle) d'enfants à chaque itération en croisant deux individus parents dont la probabilité d'être sélectionnés augmente avec la fitness des individus

```

1 def croisement(self, p1, p2):
2     stack = []
3     child = []
4     for i in range(len(p1)):
5         if (random.random() < 0.5 or p2[i] in child) and p1[i] not in child:
6             child.append(p1[i])
7             if p2[i] not in child and p2[i] not in stack:
8                 stack.append(p2[i])
9             elif p2[i] not in child:
10                child.append(p2[i])
11                if p1[i] not in child and p1[i] not in stack:
12                    stack.append(p1[i])
13            else:
14                # Remplissage des trous avec le contenu de stack en LIFO
15                while len(child) != i+1:
16                    child.append(stack.pop())
17    return child + stack[::-1]

```

Le croisement en lui-même s'effectue ainsi : on choisit aléatoirement une ville du parent 1 ou du parent 2 en fonction de la valeur d'un `random.random()` dans `[0,1]` qu'on affecte à l'enfant, et le gène non sélectionné de l'autre parent est stocké dans une pile (et éventuellement dépilé et ajouté à l'enfant par la suite) afin que l'on obtienne bien à la fin un chemin hamiltonien comme enfant.

- **Mutation**: avec une très petite probabilité α qui est un hyperparamètre du modèle, chaque individu mute en ayant deux de ses villes permutées dans le parcours du chemin hamiltonien.

```

1 def mutation(self, individu):
2     i, j = random.sample(range(len(individu)), 2)
3     individu[i], individu[j] = individu[j], individu[i]

```

- **Elimination**: A chaque itération on élimine les individus les plus mauvais pour garder une population de taille fixe.
- **Resultat** : On parcourt la population finale pour afficher l'individu le mieux évalué, i.e qui possède le chemin hamiltonien de distance minimale : il s'agit de notre solution au problème du voyageur de commerce.

```

1 def shortest_path(self):
2     best = min(self.population, key=self.distance_individu)
3     d = self.distance_individu(best)
4     print(f"Meilleure distance = {d} km, chemin = {best}")

```

- **Benchmark** : Ci-dessous figure un résumé des performances de l'algorithme génétique sur un graphe de 5 puis de 20 villes comparé à l'algorithme Best First Search dans un graphe.

```

=== Résumé Moyen ===
      | path | nodes | temps (s)
plus court circuit hamiltonien : coût = 394 et chemin : [1, 2, 4, 0, 3]
Algorithme génétique -> 394 | 5 | 0.045966499999999991 sec
plus court circuit hamiltonien : coût = 4022 et chemin : [9, 4, 11, 16, 13, 1, 14, 12, 6, 18, 8, 7, 5, 15, 2, 17, 0, 3, 10]
Algorithme génétique -> 4022 | 20 | 0.063826582999999999 sec
Best First Search -> 762 | 5 | 3.917000000019932e-06
Best First Search -> 2539 | 20 | 3.133399999999398e-05

```

Figure 2: Test unitaire de notre algorithme génétique comparé au BFS.

1.3 Algorithme des colonies de fourmis

1.3.1 Principe:

L'algorithme fonctionne en simulant le comportement réel qu'ont les fourmis, avec pour but de se rapprocher d'une solution optimale, dans notre cas, le problème du voyageur de commerce.

- Fourmis: Agents qui construisent des solutions (des chemins) en se déplaçant d'un sommet à un autre.
- Phéromone $T_{i,j}$: La quantité déposée sur l'arête (x_i, x_j) , ca nous dis à quel point un chemin est attractif ou pas.
- Heuristique $n_{i,j}$: Information statique, ici $\frac{1}{a_{i,j}}$, l'inverse du coût entre deux sommets.

```

1 eta = (1.0 / matrice_distance[ville_actuelle][ville]) ** beta
2

```

- Évaporation: à chaque itération on "oublie" une partie de la phéromone déposée.

```

1 self.matrice_pheromone *= (1 - self.rho)
2

```

Notre algorithme s'arrête lorsque nous avons un certain nombre d'itérations (100 dans notre cas).

Notre programme commence par générer un problème aléatoire

```

1
2 def creer_probleme_aleatoire(nombre_villes: int) -> Tuple[np.ndarray, List[Tuple[float,
3     float]]]:
4     """
5     Cr e un probl me du voyageur de commerce al atoire
6
7     Args:
8         nombre_villes: Nombre de villes dans le probl me
9
10    Returns:

```

```

10     Un tuple contenant la matrice de distance et les coordonnées des villes
11     """
12     coordonnees = [(random.uniform(0, 100), random.uniform(0, 100)) for _ in range(
nombre_villes)]
13
14     # Calculer la matrice de distance
15     matrice_distance = np.zeros((nombre_villes, nombre_villes))
16     for i in range(nombre_villes):
17         for j in range(nombre_villes):
18             if i != j:
19                 dx = coordonnees[i][0] - coordonnees[j][0]
20                 dy = coordonnees[i][1] - coordonnees[j][1]
21                 matrice_distance[i][j] = np.sqrt(dx * dx + dy * dy)
22
23     return matrice_distance, coordonnees

```

Une fois notre problème créé, on va instancier une colonie de fourmis avec les paramètres suivants.

```

1 # param tres
2 nombre_fourmis = 20
3 alpha = 1.0 # importance des ph romones
4 beta = 2.0 # importance de l'heuristique (distance)
5 rho = 0.5 # taux d' evaporation des ph romones
6 q = 100 # constante pour le d p t de ph romones
7 iterations = 100
8
9 colonie = ColonieDefourmis(matrice_distance, nombre_fourmis, alpha, beta, rho, q,
iterations)

```

On exécute ensuite notre colonie de fourmis, la boucle principale est la suivante

```

1 for iteration in range(self.iterations):
2     # r initialiser les fourmis
3     for fourmi in self.fourmis:
4         fourmi.reset()
5
6     # construire les solutions
7     for fourmi in self.fourmis:
8         fourmi.construire_solution(self.matrice_pheromone, self.matrice_distance,
self.alpha, self.beta)
9
10
11     # mettre jour la meilleure solution si n cessaire
12     if fourmi.distance_totale < self.meilleure_distance:
13         self.meilleure_distance = fourmi.distance_totale
14         self.meilleur_chemin = fourmi.chemin.copy()
15
16     self.historique_distances.append(self.meilleure_distance)
17     self.mettre_a_jour_pheromones()
18
19 return self.meilleur_chemin, self.meilleure_distance

```

À chaque itération, on évalue à nouveau notre solution en faisant choisir la prochaine ville à nos fourmis, en calculant notre heuristique et en appliquant l'évaporation des phéromones.

1.3.2 Affichage graphique des résultats

Nous avons aussi implémenté un graphe pour visualiser le résultat trouvé par notre algorithme en utilisant seaborn.

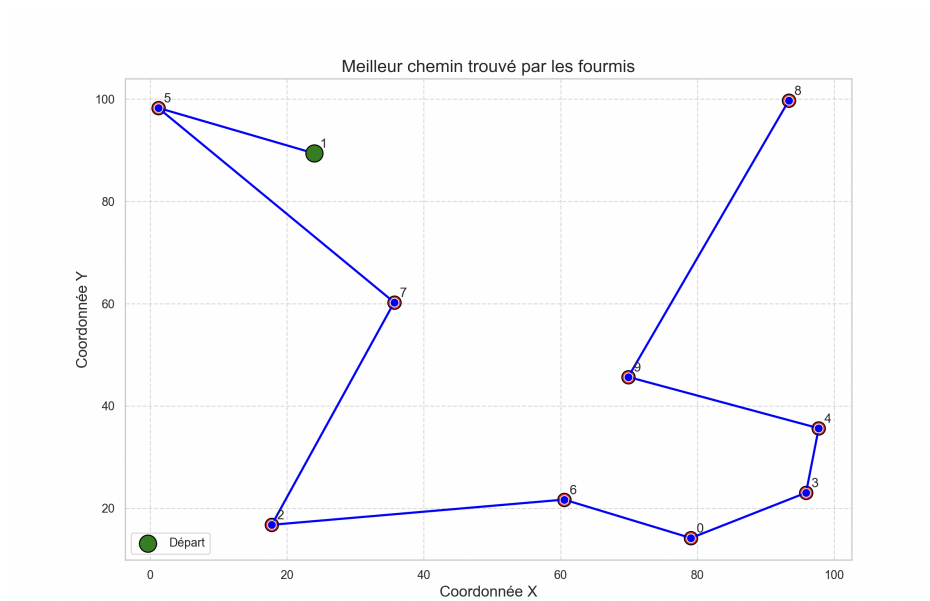


Figure 3: Test unitaire de notre algorithme des fourmis.

1.3.3 Benchmark

Nous avons mis en place deux choses pour tester notre solution.

- Un test unitaire pour voir si notre algorithme est capable de trouver une solution optimale pour une problème simple.

```
=== Test sur un problème carré (solution optimale connue) ===  
Meilleur chemin trouvé: [0, 1, 2, 3]  
Distance trouvée: 40.00  
Distance optimale: 40.00  
Erreur relative: 0.00%  
Test réussi! L'erreur (0.00%) est dans la limite acceptable (5%).
```

Figure 4: Test unitaire de notre algorithme des fourmis.

- Un simple test permettant de calculer le temps d'exécution de notre algorithme.

References