Sorbonne Université
Master ANDROIDE

Parallélisation de l'apprentissage par renforcement basé sur des populations

UE de projet M1

Charles LIN – Zhe WANG

2023

Table des matières

Chapitre 1

Introduction

Le projet s'inscrit dans le domaine de l'apprentissage par renforcement, une technique d'apprentissage automatique où un agent interagit avec son environnement pour apprendre à prendre des décisions. Les algorithmes de recherche de nouveauté (NS) et de qualité-diversité (QD) sont des approches alternatives à l'optimisation basée sur la performance seule. Ce sont des algorithmes évolutionnaires qui demandent une importante capacité computationnelle, car leur leur apprentissage ne s'appuit que sur l'expérience. Il est donc important d'utiliser des méthodes de parallélisation pour accélérer le temps d'exécution des expériences, dans notre projet on s'intéressera à l'optimisation de la phase d'évaluation qui sont les plus coûteuses pour nos deux algorithmes.

La librairie diversity-algorithms [1] porpose une implémentation des méthodes NS et QD parrallélisé sous CPU.

BRAX [2] est une librairie développé en JAX [3] par Google, pour simuler des environnement physique en apprentissage par renforcement. Les mechanisme de parralélisation introduite par la librairie permettent de réduire considérablement le temps d'exécussion des algorithmes d'apprentissage.

Sous l'encadrement de Stéphane Doncieux et de Johann Huber, notre projet aura donc pour objectif d'intégrer des environnement brax à la librairie diversity-algoritghms.

Lien github : https://github.com/lincharles123/PROJET_ANDROIDE

Chapitre 2

Etat de l'art

2.1 Novelty Search (NS)

Novelty Search (NS) est l'un des principaux algorithmes de recherche divergente, ou l'objectif est laissé de côtés pour développer des solutions nouvelles.

En effet, NS calcule la nouveauté comme la distance moyenne d'un individu à ses voisins les plus proches dans un espace de comportement. Par conséquent, NS pousse les individus à se déplacer constamment dans l'espace de comportement, explorant de nouvelles zones inexplorées.

Algorithm 1 Novelty Search [4]

INPUTS : population size μ, number of offspring λ, environment *env*, number of neighbors for novelty calculation k, number of generations G
RESULT : Generated policies.
POP $\leftarrow$ $RandomPopulation(\mu)$
ARCH $\leftarrow \emptyset$
GEN $= 0$
while GEN $< G$ **do**
  OFF $\leftarrow$ generateOffspring(POP, λ)
  for AGENT in (POP $\cup$ OFF) **do**
    (agent.fit, agent.bd) = evaluate(agent, env)
  end for
  novRefSet $\leftarrow$ POP $\cup$ OFF $\cup$ ARCH
  for agent in (POP $\cup$ OFF) **do**
    agent.novelty $\leftarrow$ getNovelty(agent.bd, novRefSet, k)
  end for
  / Update archive, either with random samples or with the most novel ones : */*
  ARCH $\leftarrow$ ARCH $\cup$ sample(off)
  POP $\leftarrow$ selectMostNovel(POP $\cup$ OFF, μ)
  GEN $=$ GEN $+ 1$
end while

2.2 Quality Diversity (QD) - MAP-Elites

L'optimisation de la Qualité-Diversité (QD) propose une approche novatrice à la génération d'un ensemble de solutions diversifiées et de haute qualité. Elle se distingue par sa capacité à offrir une multitude de solutions performantes, au lieu de se concentrer sur une seule.

A chaque génération, la population évalué sera ajouté à une archive, les agents qui aurons une meilleur performance que les agents précédents de l'archive vont les remplacés.

Algorithm 2 MAP-Elites ($N_{\mathcal{B}}$: Batch size) [5]

for iteration $\in [1, I]$ **do**
  if first iteration **then**
    $\mathcal{B} \leftarrow$ random solutions
  else
    $\mathcal{B} \leftarrow$ select solutions from archive $\mathcal{A}$
  end if
  $\widetilde{\mathcal{B}} = \left(\widetilde{\boldsymbol{\theta}}_j\right)_{j \in [1, N_{\mathcal{B}}]} \leftarrow \text{VARIATION}(\mathcal{B})$
  $\forall j \in [1, N_{\mathcal{B}}]$ run episode of $\pi_{\widetilde{\boldsymbol{\theta}}_j}$, compute return $R^{(\boldsymbol{\theta}_j)}$ and trajectory $\boldsymbol{\tau}^{(\boldsymbol{\theta}_j)}$
  for $j \in 1, N_{\mathcal{B}}$ **do**
    cell $\leftarrow$ get grid cell of descriptor $\tilde{d}\left(\boldsymbol{\tau}^{(\boldsymbol{\theta}_j)}\right)$
    $\boldsymbol{\theta}_{\text{cell}} \leftarrow$ get content of cell
    if $\boldsymbol{\theta}_{\text{cell}}$ is None **then**
      Add $\boldsymbol{\theta}_j$ to cell
    else if $R^{(\boldsymbol{\theta}_j)} > R^{(\boldsymbol{\theta}_{\text{cell}})}$ **then**
      Replace $\boldsymbol{\theta}_{\text{cell}}$ with $\boldsymbol{\theta}_j$ in cell
    else
      Discard $\boldsymbol{\theta}_j$
    end if
  end for
end for
return archive $\mathcal{A}$

Chapitre 3

Contribution

3.1 Méchanisme important

Nous allons ici présenté les méchanisme importante et les contraintes a respéctées pour profiter au maximum de l'optimisation proposé par JAX[3].

3.1.1 Just In Time Compilation (JIT)

La compilation à la volée (JIT) est un mode de compilation où le code source est traduit en code machine pendant l'exécution du programme, plutôt que d'être préalablement compilé avant l'exécution. L'objectif principal de la compilation à la volée est d'améliorer la performance en compilant le code à exécuter au moment le plus approprié.

Le méchanisme de JIT est un point clés de JAX, dont nous aurons besoin tout au cours de l'implémentation de notre projet.

Pour toutes fonctions à entrées de structure statique, la compilation ne sera efféctué qu'une unique fois au première appel de la fonction. Ainsi, si une fonction et appellé de manière récurentes dans notre projet, on aura intêret à la coder avec JIT.

Inconvénient :

1. Temps de compilation : Pour les fonctions longues ou complexes, le processus de compilation peut être plus longue que l'exécution réel de la fonction.
2. Consommation de mémoire : La compilation à la volée peut augmenter la consommation de mémoire car elle doit stocker à la fois le code source et le code machine.
3. Entrées statiques : Comme mentionnez pour tirer profit de la JIT il faut que les entrés soit de structure statique, sinon la fonction devra être recompilé à chaque changement de structure.

3.1.2 Parallélisation

La parallélisation des fonctions en JAX[3] est réalisé à l'aide de la fonction jax.vmap. C'est un outil qui permet d'appliquer une fonction à un ensemble de vecteurs ou de tenseurs de manière parallèle et efficace.

La fonction vmap prend en entrée une fonction f et un ensemble de vecteurs ou de tenseurs. Elle applique ensuite cette fonction à chaque élément de l'ensemble, en retournant un nouveau vecteur ou tenseur contenant les résultats correspondants.

La parallélisation est réalisée grâce à une transformation automatique des opérations effectuées par la fonction f. JAX décompose le calcul en plusieurs sous-calculs indépendants, les exécute en parallèle sur les cœurs du processeur, puis réassemble les résultats pour former le nouveau vecteur ou tenseur.

Cela permet d'accélérer considérablement les calculs lorsque la fonction f est appliquée à un grand nombre d'éléments, en exploitant les capacités de parallélisme offertes par les processeurs modernes.

Il faut cependendant que les entrées et sorties de la fonction soit des types JAX valide (Tableaux ndarray JAX, Scalaires JAX, ...), si ce n'est pas le cas ces valeurs devrons être converties au préalable. La fonction en entrée devra également être pure, c'est à dire produire pour une entrée donné un même résultat à chaque appel.

3.2 Implémentation

3.2.1 Changement apporté à diversity_algorithms

Notre code dérive de la librairie diversity_algorithms, l'objectif principale du projet consistait en la parralélisation de l'étape d'évaluation des agents qui est la plus couteuse. Ainsi nous avons d'abord implémenter un controlleur en flax [6] librairie écrite en jax pour la création de réseau de neuronnes, elle servira à inférer une action pour un agent et un état donné de l'environnement.

L'environnement sera simulé à l'aide d'environnement BRAX[2] codé en JAX qui afin d'effectuer les evaluations en parallèle.

Après optimisation de l'évaluation nous avons observés qu'en augmentant le nombre d'agents pour profiter un maximum de la parallélisation, les autres étapes dans l'implémentation originelle deviennent trop longue. Voir même beaucoup plus longues que l'exécution d'évaluation qui était à l'origines l'étape la plus couteuse de l'algorithmes.

Nous avons du donc recoder d'autres étapes de l'algorithmes avec jax[3].

Evaluation

Elle est composé d'un controlleur perceptron multicouche codé en flax[6] et d'un environnement BRAX[2]. L'évaluateur est initialisé au lancement du programme est sera appelé à chaque étape d'évaluation. Il utilise les mécanisme de jit et vmap pour optimiser le calcul, ainsi les paramètres devront être de type JAX valide et la fonction d'évaluation pure.

Variation

Les fonction VarOr et mutate de DEAP[7] sont séquentielles donc très lente et pas adapté à des populations de grande taille, nous avons donc réimplémenter leurs équivalents en JAX. De plus, nous avons remarquer que cloner un agent et beaucoup plus lent que

de créer un nouvel agents, et un comportement bizarre de DEAP est que l'initialisation d'un agent de type de base ndarray et plus rapide si on le met dans une liste puis de la ressortir pour bien avoir le ndarray. Sur une population de taille supérieur à 1000 le temps de variation initiale est de plus de 10 secondes ce qui n'est pas acceptable, après réimplémentation même pour une population de 8000 agents sa durée reste à 1-2 secondes (le gros du temps est consacrés au clonage de l'agent)

Mise à jour de l'archive

Pour novelty search, la mise à jour de l'archive nécessite de calculé l'indice de nouveauté de chaque agents et l'opération était réalisé itérativement, nous avons donc recodé les fonction pour que le processus soit parallélisé.

Du côté de qd l'achive structuré ne nécessite pas d'optimisation et l'archive nonstructuré n'a pas été étudié.

3.2.2 Visualisation

Nous avons implémenter deux visualisation pour la carte de descripteur comportementaux, une visualisation pour ant-omni qui a un descripteur 2D et un pour ant-uni qui a un descripteur 4d. Nous avons utilisons heatmap pour cela, la visualisation du descripteur 4 dimensions est réalisé en utilisant plusieurs représentation 2d dans les cellules d'une autre représentation 2d.

3.3 Expérience et résultat

Nous avons essayé de reproduire au mieux les metrics de QDAX pour s'y comparer.

3.3.1 Mise en place des expériences

Environnement

Les environnement simulé sont ant-uni et ant-omni telle que décrit dans QDAX[5], la librairie est open-source et libre de modification, nous avons donc repris le wrapper FeetContactWrapper qui va permettre d'extraire le contact de chaque patte du robot avec le sol à chaque step 0, 1.

ant-uni :

- But : trouver des comportement ou le robot avance le plus rapidement possible.
- Descripteur comportemental : porportion de temps en contact avec le sol de chaque patte [0,1].
- Fonction objectif : la somme d'une récompense de survie(reward_survive), récompense de vitesse de déplacement vers l'avant(reward_froward) et du coût des force appliqués à chaque step de l'épisode(reward_torque).

ant-omni :

- But : trouver des robot qui peuvent se déplacer de manière efficace dans toutes les directions.
- Descripteur comportemental : position final du robot (x, y)

— Fonction objectif : la somme d'une récompense de survie(reward_survive) et du coût des forces appliqués à chaque step de l'épisode(reward_torque).

Nous n'avons pas trouvé comment le calcul du coût des forces est calculé, nous avons donc pris comme cout la somme de la moyenne de valeur absolu du couple appliqué sur chaque articulation à chaque step. Les autres metriques sont données dans l'implémentation de BRAX[2].

	ant-uni	ant-omni
longeur d'épisode	300	100
Min-BD	[0,0,0,0]	[-15,-15]
Max-BD	[1,1,1,1]	[15,15]
Taille de coverage map	10^4	100^2

Paramètrage

Les paramètres ont été choisi de sorte à collés avec ceux de QDAX[5] et par rapport aus paramètres originel de diversity_algorithms[1].

Réseau de neuronnes : Perceptron multicouches 2 couche cachés de 64 neuronnes, fonction d'activation relu et fonction de sortie tanh.

Pour novelty search, nous prenons lambda_nov à 6% de la taille de la population (nombre d'agents ajouté à l'archive par itération) et lambda de 1 (multiplicateur du nombre de descendant créer à chaque itération).

Pour quality diversity, nous prennons une archive de type structuré, nous mettons à jour l'archive avec tous les agents. Lors de la sélection d'agents nous tirons autant d'agents aléatoirement que demandés contrairement à l'implémentation initial qui rend tous les agents présent dans l'archive s'il n'y en a pas assez.

Nous fixons un budget computationnel de 100 000 évaluations.

Equipement

Les expériences sont réalisé avec une carte graphique NVIDIA GeForce RTX 3060

3.4 Résultat

3.4.1 ant-uni

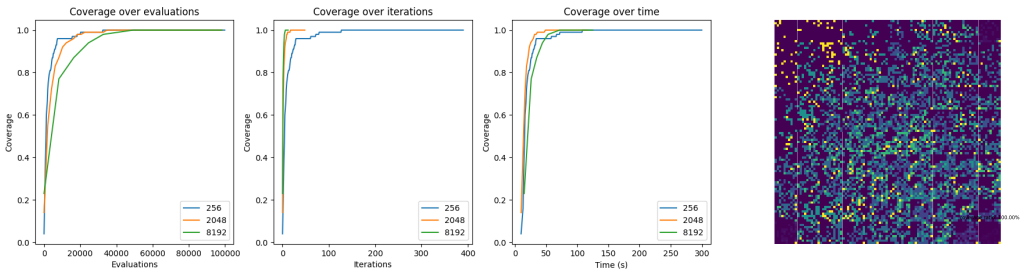

FIGURE 3.1 – ant-uni NS

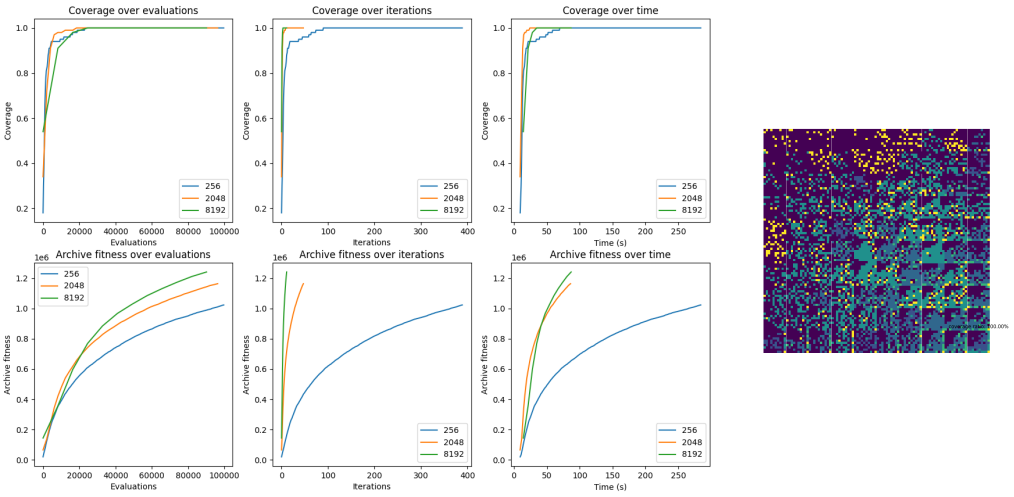

FIGURE 3.2 – ant-uni QD

3.4.2 ant-omni

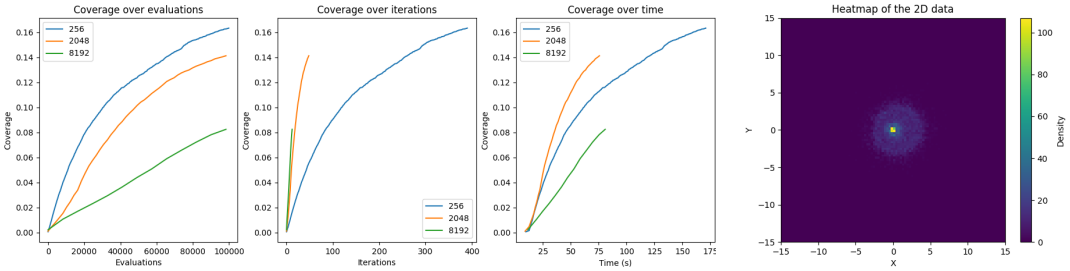

FIGURE 3.3 – ant-omni NS

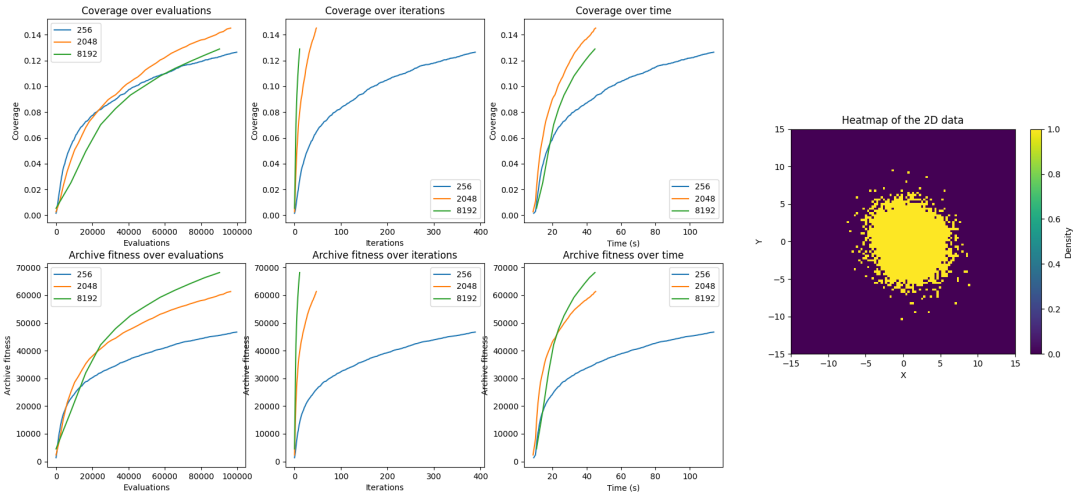

FIGURE 3.4 – ant-omni QD

3.5 Discussion

Nous observons que pour un même nombre d'évaluation augmenter la taille de la population le temps de calcul est réduit significativement. Cependant pour taille de population trop grande le vitesse d'excution devient n'est plus optimé, ce phénomène vient du faite que les autres étapes de l'algorithme deviennent trop lente malgrés le faite que le calcul de l'evaluation soit optimisé.

Nous observons également qu'en prenant une grande population les agents appris ont globalement une meilleur performance 3.2 3.4, ceci est du fait qu'avec une plus grande population on remplace plus rapidement les agents de mauvaise perfomance.

La couverture des descripteurs comportemantaux est également meilleur pour un algorithme qd où l'on tire aléatoirement des agens dans l'archive, généré beaucoup d'agents permet d'augmenter nos chances d'obtenir des agents nouveaux et donc de remplir l'archive plus rapidement, on peut ensuite tiré des agents dans une plus diverse.

Par contre pour un algorithme de type NS, le nombre d'itération réalisé sera plus important 3.3, généré beaucoup plus d'agent à partir des agents les plus nouveaux a un intérêt moidre dans la mesure ou les agents généré seront proches de ceux-ci, il conviendrait donc d'en générer un nombre correct mais pas trop non plus.

Néanmoins si on ne se fixait pas un budget computationnel mais un temps d'exuction, il serait intéressant de générer plus d'agents et réalisé deux fois plus d'évaluation pour un même temps d'exécution et ainsi obtenir plus d'agents diverses.

Une population de 2024 agents et 8192 agents est observé au même temps d'exécution, ce phénomène est causé par la lenteur des étape de variations et de mise à jour de l'archive qui ne sont pas optimiser, le temps gagner par la parralélisation de l'évolution a été annulé par les autres composantes.

Chapitre 4

Conclusion

Nous n'avons pas pu comparé nos résultat à des environnement de diversity_algorithms initialement définit afin d'étudier les gain en temps d'exécution par rapport à une parallélisation sur CPU. Cependant on peut dire que la parralélisation de l'évaluation est réussit en observant les temps d'execution qui est de l'ordre de quelques secondes même pour des milliers d'agents.

L'implémentation d'environnement BRAX[2] reste réussi même si les autres étapes des algorithmes ralentissent l'apprentissage.

4.1 Amélioration envisagé

La structure de stockage d'information de diversity-algorithms [1] n'est pas adapté à brax, c'est à dire stockées toute les information dans une entités, genotype, descripteur comportementale, fitness, etc. Il faudrait penser à séparer les données en plusieurs tableau multidimensionnel jax ndarray, cela facilitera la parallélisation des opérations en nous évitant d'effectuer systématiquement des transformations des données avant de les passer au fonction vmap.

Le fait de stocker les informations dans des ndarray va également faciliter le clonage et la création d'agents qui n'est pas commode avec deap (notamment lors de la varition et de la création d'agents).

Implémenter les environnement en brax.v2 qui est parru au cours du projet et que nous avons préférer abandonner en raison de lenteur à l'exécution que nous n'avons pas su identifié.

Bibliographie

[1] Stéphane Doncieux, Giuseppe Paolo Alban Laflaquière et Alexandre Coninx. *diversity$_a$lgorithms*. 2020. URL : https://github.com/robotsthatdream/diversity_algorithms (pages 1, 7, 10).

[2] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch et Olivier Bachem. *Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation*. Version 0.9.0. 2021. URL : http://github.com/google/brax (pages 1, 5, 7, 10).

[3] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne et Qiao Zhang. *JAX : composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL : http://github.com/google/jax (pages 1, 4, 5).

[4] Stephane Doncieux, Giuseppe Paolo, Alban Laflaquière et Alexandre Coninx. « Novelty Search Makes Evolvability Inevitable ». In : (2020). DOI : 10.1145/3377930.3389840. URL : https://doi.org/10.1145/3377930.3389840 (page 2).

[5] Bryan Lim, Maxime Allard, Luca Grillotti et Antoine Cully. « Accelerated Quality-Diversity for Robotics through Massive Parallelism ». In : *arXiv preprint arXiv :2202.01258* (2022) (pages 3, 6, 7).

[6] Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner et Marc van Zee. *Flax : A neural network library and ecosystem for JAX*. Version 0.6.9. 2023. URL : http://github.com/google/flax (page 5).

[7] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau et Christian Gagné. « DEAP : Evolutionary Algorithms Made Easy ». In : *Journal of Machine Learning Research* 13 (juill. 2012), p. 2171-2175 (page 5).

[8] Eisuke Kita. *Evolutionary Algorithms*. Rijeka : IntechOpen, avr. 2011. DOI : 10.5772/627. URL : https://doi.org/10.5772/627 (page 12).

Annexe A

Cahier des charges

A.1 Contexte et définition du projet

Le projet s'inscrit dans le domaine de l'apprentissage par renforcement, une technique d'apprentissage automatique où un agent interagit avec son environnement pour apprendre à prendre des décisions. Les algorithmes de recherche de nouveauté (NS) et de qualité-diversité (QD) sont des approches alternatives à l'optimisation basée sur la performance seule. Ce sont des algorithmes évolutionnaires qui demandent une importante capacité computationnelle, car leur leur apprentissage ne s'appui que sur l'expérience. Il est donc important d'utiliser des méthodes de parallélisation pour accélérer le temps d'exécution des expériences, dans notre projet on s'intéressera à l'optimisation de la phase d'évaluation qui sont les plus coûteuses pour nos deux algorithmes. Actuellement la li-

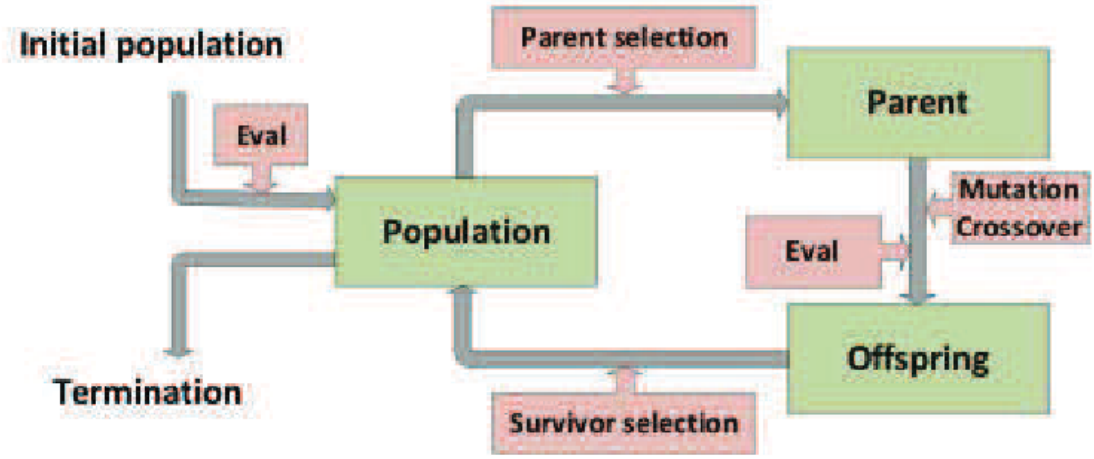

FIGURE A.1 – Scheme of Evolutionary Algorithms [8]

brairie est parallélisé sur CPU, et notre but serait de le réaliser sur GPU, qui ont été spécialement conçus pour le traitement parallèle massif de données. JAX est une librairie d'apprentissage automatique développée par Google qui permet de paralléliser massivement des calculs sur GPU. BRAX est une librairie développée en JAX pour simuler des environnements standards en apprentissage par renforcement, notamment pour la robo-

tique. La parallélisation mise en œuvre dans les environnements BRAX permet de réduire significativement les temps d'exécution des algorithmes d'apprentissage.

A.2 Objectif du projet

Le but principal de ce projet est d'intégrer les environnements de simulation BRAX à la librairie diversity_algorithms pour accélérer les temps de calcul et donc d'améliorer l'efficacité des algorithmes d'apprentissage par renforcement basés sur la recherche de nouveauté et de qualité-diversité. Les objectifs spécifiques incluent :

1. Modifier la librairie diversity_algorithm pour permettre l'utilisation d'environnements BRAX.
2. Tester les environnements BRAX, comparer les performances de diversity_algorithm avec BRAX et leurs équivalents dans les simulateurs précédemment utilisés, comme MUJOCO.
3. Si possible, implémenter des environnements jouets supplémentaires en utilisant JAX.

Le résultat attendu du projet est une version de la librairie diversity_algorithms qui intègre des environnements BRAX. L'analyse des choix de conception, les visualisations interprétables des résultats et les études comparatives en termes de temps et de coût computationnel, de complexité en usage mémoire et en calculs devront être présentées pour justifier les choix effectués pendant le projet.

A.3 Périmètre du projet

Dans le cadre de ce projet, le périmètre comprend les tâches suivantes :

— Intégrer les environnements de simulation BRAX à la librairie diversity_algorithm pour améliorer les performances des algorithmes d'apprentissage par renforcement basés sur la recherche de nouveauté et de qualité-diversité.

— Effectuer des tests comparatifs des performances des environnements BRAX avec celles des simulateurs précédemment utilisés, tels que MUJOCO.

— Si possible, implémenter de nouveaux environnements jouets en utilisant JAX.

Les tâches qui ne sont pas incluses dans le périmètre du projet sont :

— Développer de nouveaux algorithmes d'apprentissage par renforcement basés sur la recherche de nouveauté et de qualité-diversité.

— Modifier BRAX ou la librairie diversity_algorithm de manière à changer leur fonctionnement fondamental.

— Effectuer des tests de performance sur des ordinateurs autres que ceux qui sont mis à disposition dans le cadre du projet.

A.4 Besoins et contraintes

Besoins :

— Connaissance de base des algorithmes NS et QD.

— Connaissance en programmation en Python.
— Connaissance de la bibliothèque BRAX et de la parallélisation GPU.
— Connaissance de la librairie diversity_algorithms

Contraintes :

— La structure de la librairie diversity_algorithms doit être préservée autant que possible pour garantir son intégrité.
— L'intégration de BRAX doit être réalisée de manière cohérente
— Les résultats doivent être facilement interprétables à travers des visualisations et une analyse des choix de conception

Annexe B

Manuel utilisateur

B.1 Installation

— Cloner le projet

— Créer un nouvel environnement et activer environnement :

```
conda create --name your\_env python
conda activate your\_env
```

— Clonner le repo google/brax en local puis lancer la commande :

```
cd brax
pip install --upgrade pip
pip install -e .
```

— Installation de jax pour gpu :

```
pip install --upgrade "jax[cuda11_pip]" -f
    https://storage.googleapis.com/jax-releases/jax_cuda_releases.html
```

— Installation de diversity-algorithms :

```
cd diversity_algorithms_dev-master
pip install -e .
```

B.2 Utilisation

Pour lancer le code il faut se mettre dans le dossier diversity_algorithms_dev-master/diversity_algorithm
puis lancer la commande :

```
python3 brax_novelty.py -e env
# ou
python3 brax_qd.py -e env
```

Les environnement disponible acutellement sont :

— ant-uni et ant-omni

Les paramètre réglabes sont :

— -p : la taille de la population

— -L : le nombre de step par evaluation

— -g : le nombre de génération d'évaluation

— -e : le nom de l'environnement

— l : coefficient de descendant à générer par rapport à la population (NS)

Les notebook nov_test et qd_test sont également disponible pour manipuler les algorithmes.