M2 MPRI - Rapport projet langages de programmation probabilistes

Josué Moreau, Léo Kulinski

18 Février 2022

1 Structure du projet

Le projet est un repo git qui se trouve à cette adresse. Il est constitué des fichiers suivants:

- `distribution.py` contient une petite bibliothèque de distributions.
- `inference.py` contient toutes les méthodes d'inférences implémentées.
- `utils.py` contient des fonctions utiles dans la définition des distributions et des méthodes d'inférence.
- `test_inference.py` contient certaines fonctions utilisées lors des tests.
- `funny_bernoulli.py`, `coin.py`, `laplace.py`, `hmm.py`, `branching.py`, `gaussian.py`, `rel_lin.py` et `bouncing_ball.py` sont des fichiers de tests. (Chacun de ces exemples peut être directement exécuté pour afficher la distribution de probabilité renvoyée par différentes méthodes d'inférence. Pour afficher la suivante, il suffit de fermer l'image générée.)

Ce projet a été réalisé en Python 3.10. Nous avons utilisé de la programmation orientée objet afin de représenter les distributions ainsi que les méthodes d'inférence. Ce style de programmation a été essentiel dans notre cas car nous avons utilisé le typage fourni par la bibliothèque `typing` de Python, ainsi que l'outil `mypy` pour vérifier statiquement le typage des fichiers. Utiliser des objets pour représenter les méthodes d'inférences et les distribution a donc ensuite permis, avec des classes abstraites, de les manipuler dans le typage et d'écrire le code le plus générique possible lors des tests.

2 Méthodes d'inférence

2.1 Méthode d'inférence abstraite

Concrètement, une méthode d'inférence est une classe implémentant la classe abstraite `InferenceMethod`, définie de la manière suivante :

```
class InferenceMethod(ABC, Generic[A, B]):

    @abstractmethod
    def __init__(self, model: Callable[[Prob, A], B], data: A):
        pass

    @abstractmethod
    def infer(self, n: int = 1000) -> Distrib[B]:
        pass

    @staticmethod
    @abstractmethod
    def name() -> str:
        pass
```

Un modèle manipulé par une fonction d'inférence est une fonction qui prend en argument une instance de la classe `Prob`, ainsi qu'une donnée, et renvoie une valeur. `Prob` est en réalité également une classe abstraite mettant à disposition de la fonction du modèle les fonctions `sample`, `factor`, `observe` et `assume`. Elle est définie de la manière suivante :

```
class Prob(ABC):

    @abstractmethod
    def assume(self, p: bool) -> None:
        pass

    @abstractmethod
    def observe(self, d: Distrib[A], x: A) -> None:
        pass

    @abstractmethod
    def sample(self, d: Distrib[A]) -> A:
        pass
```

Cette classe classe `Prob` est implémentée, pour chaque méthode d'inférence, par une classe définie dans la classe de la méthode d'inférence. Procéder ainsi a l'avantage de permettre, lors de l'écriture d'un modèle, d'avoir un typage ne dépendant pas de la méthode d'inférence et de cacher les méthodes auxiliaires utilisées dans l'implémentation de la classe `Prob` liée au modèle.

2.2 Échantillonnage par énumération

```
class EnumerationSampling(InferenceMethod[A, B]):

    class EnumSampProb(Prob):
    _score: float
    _path: List[_SampleState]

        def init_next_path(self) -> None: ...

        def sample(self, d: Distrib[A]) -> A: ...
```

L'échantillonnage par énumération fonctionne ainsi.

Les fonctions assume, et observe sont similaires à celles de l'inférence par rejet.

On commence par exécuter le modèle une première fois, et à chaque nouvelle instruction sample trouvée (y compris celles cachées dans la méthode observe), on examine le support de la distribution en question. On sauvegarde les valeurs qu'on peut y prendre, ainsi que les choix qu'on a déjà effectués. Le score attribué à une exécution est le produit des probabilités de chaque choix effectué, ou de manière équivalente la somme des logits, calculé au fur et à mesure par `_score`.

Puis, lorsqu'une exécution se termine, on vérifie si on a déjà effectué toutes les exécutions possibles. C'est le rôle de `init_next_path`. Si ce n'est pas le cas, il faut planifier notre prochaine exécution : quelle succession de tirage faire pour aboutir à un scénario qui ne s'est encore jamais produit ? Cette information est enregistrée dans `_path`, que l'on peut voir comme une pile. Pour être sûr d'être exhaustif, on cherche à faire prendre au dernier sample rencontré toutes les valeurs possibles. S'il les a toutes prises, on revient au sample précédent, et ainsi de suite jusqu'à épuisement des possibilités.

Bien entendu, si après avoir suivi toutes les consignes de `_path`, on rencontre un sample qu'il n'avait pas prédit, on doit également à présent en examiner toutes les possibilités, donc on l'empile sur `_path`.

Ainsi, on garantit d'explorer toutes les exécutions possibles du programme.

2.3 Algorithme de Metropolis-Hastings

La classe implémentant `Prob` dans le cas de l'algorithme de Métropolis-Hastings est de la forme suivante :

```
class MHProb(Prob):
    _scores: List[float]
    _weights: List[float]
    _sampleResults: List[Any]
    ...
```

```
    def sample(self, d: Distrib[A]) -> A: ...

    def go_back_to_step(self, i: int, new_id: int) -> None: ...

    def pick_random_step(self) -> int: ...

    def computeScore(self) -> float: ...
```

Une adaptation est tout d'abord nécessaire dans la fonction `sample`, par rapport à Importance Sampling. En effet, Metropolis-Hastings permet de recommencer l'exécution du modèle à partir d'un certain point. Pour ce faire, nous utilisons une liste `_sampleResults` qui stocke les résultats des appels à `sample`, afin de pouvoir les réutiliser lorsque l'on demande à ré-exécuter le modèle à partir d'un certain point. Il est à noter que, dans le cas d'une implémentation en OCaml, il n'aurait pas été nécessaire d'utiliser une telle liste, car on pourrait utiliser le passage par continuation avec les clôtures pour reprendre l'exécution du modèle à partir d'un certain point. Ici, lorsqu'on parle de ré-exécuter le modèle à partir d'un certain point, il s'agit en fait d'exécuter le modèle à nouveau. Lorsque des appels à `sample` sont faits, si l'exécution courante du modèle n'est pas encore au point à partir duquel on va échantillonner à nouveau les variables, il suffit alors de restaurer la valeur qui avait été choisie aléatoirement et stockée auparavant dans `_sampleResults`.

Outre les classiques fonctions demandées par la classe `Prob`, diverses fonctions, qui ne pourront être appelées qu'à partir de la méthode `infer`, sont nécessaires. La fonction `go_back_to_step` permet de réinitialiser l'instance de la classe `MHProb` pour qu'elle soit réutilisée dans l'itération suivante, et qu'elle conserve les `i` premières variables. Cette méthode change également l'identifiant afin de ne pas modifier le score de l'itération précédente de l'algorithme. La fonction `computeScore` calcule le score tel qu'utilisé pour décider si l'on conserve ou non la valeur obtenue dans l'itération courante de Métropolis-Hastings. Le calcul de celui-ci fait appel au même score, contenu dans le tableau `_scores`, que celui calculé par Importance Sampling, ainsi qu'au `logpdf` de chaque variable obtenue par `sample`, qui se trouvent dans le tableau `_weights`.

2.4 Hamiltonian Monte Carlo

Après avoir réalisé l'essentiel des tâches demandées dans le projet, nous avons essayé d'implémenter l'algorithme Hamiltonian Monte Carlo. Après quelques recherches, nous avons appris que cette méthode nécessitait un nombre fixe d'appels à `sample`, ce qui n'est pas toujours le cas dans un programme probabiliste. Nous avons donc essayé d'implémenter la variante décrite dans cet article [1], qui corrige ce problème.

Cependant, nous n'avons pas réussi à obtenir des résultats avec cette implémentation. D'après ce que nous avons compris, l'algorithme HMC fait varier les résultats des appels à `sample` pour générer de nouvelles valeurs. Le soucis que nous avons est que ces variables que l'algorithme fait varier sont censées suivre des distributions qu'elles ne semblent plus suivre si on fait varier leurs valeurs. Nous supposons donc qu'il s'agit de la raison pour laquelle notre implémentation ne fonctionne pas, mais nous n'avons pas réussi à trouver comment permettre de faire varier ces valeurs tout en faisant en sorte qu'elles continuent de suivre une distribution donnée. Nous laissons les fichiers dans le dossier mais aucun des tests définis dans `test_hmc.py` n'est concluant.

3 Exemples et résultats

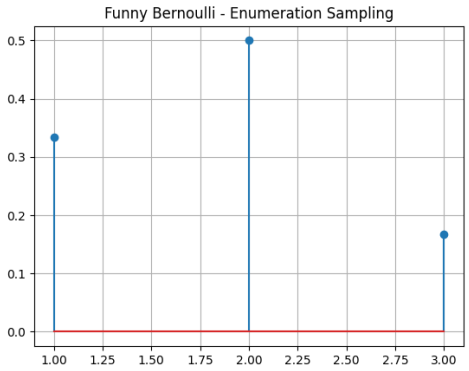
3.1 Exemples classiques du cours

Dans l'exemple `coin.py` se trouve le modèle `coin` du cours, mais aussi un modèle `discrete_coin`. C'est le même que le précédent, à la différence que le biais de la pièce qu'on cherche à trouver est tiré selon une loi discrète qui s'approche de la loi uniforme. Ainsi, il est possible d'utiliser les méthodes d'inférence discrètes comme Rejection et Enumeration Sampling dessus.

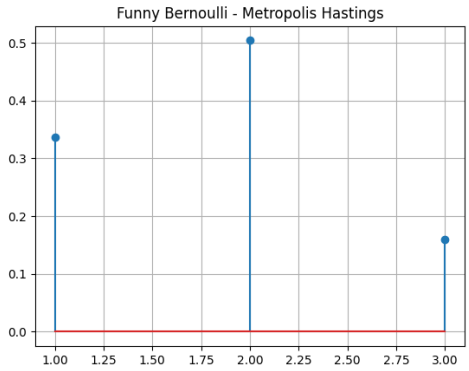
Enumeration Sampling et Métropolis-Hastings se comportent très bien sur les exemples Funny Bernoulli et Coin. Dans le cas de l'exemple Laplace, la majeure partie du temps, Métropolis-Hastings trouve un résultat très similaire à Importance Sampling ; cependant, en testant plusieurs fois de suite l'exemple, la distribution obtenue est parfois un peu moins précise. Pour ce qui est de

[1] Nonparametric Hamiltonian Monte Carlo, *Carol Mak, Fabian Zaiser, Luke Ong, Proceedings of the 38th International Conference on Machine Learning*, PMLR 139:7336-7347, 2021.

l'exemple hidden markov model, en exécutant successivement plusieurs fois le modèle, il semblerait qu'en moyenne Métropolis-Hastings et Importance Sampling soient de la même efficacité.

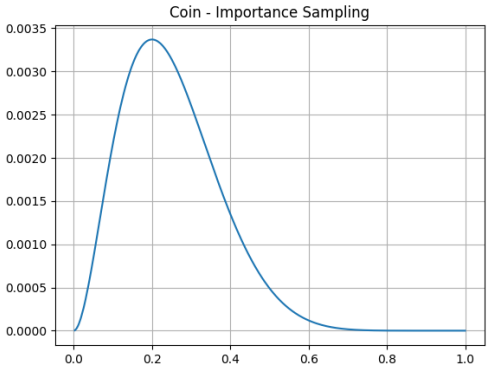


(a) Enumeration Sampling

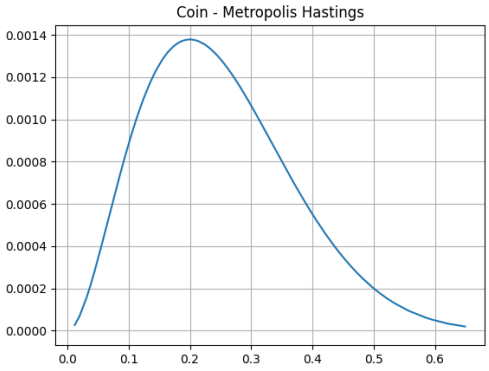


(b) Métropolis-Hastings

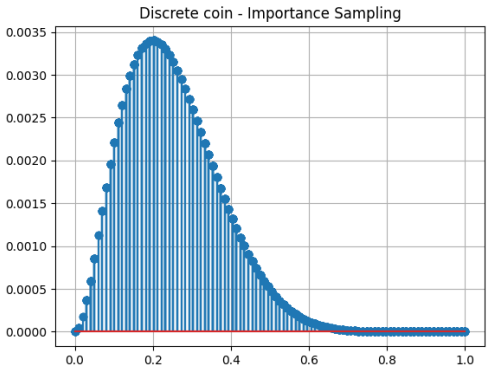
Figure 1: Funny Bernoulli



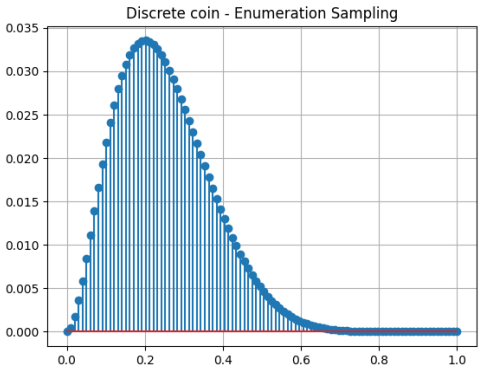
(a) Coin - Importance Sampling



(b) Coin - Métropolis-Hastings

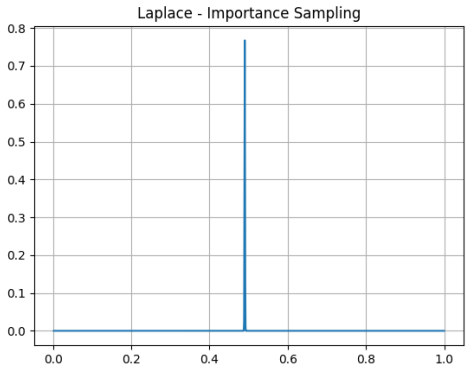


(c) Discrete Coin - Importance Sampling

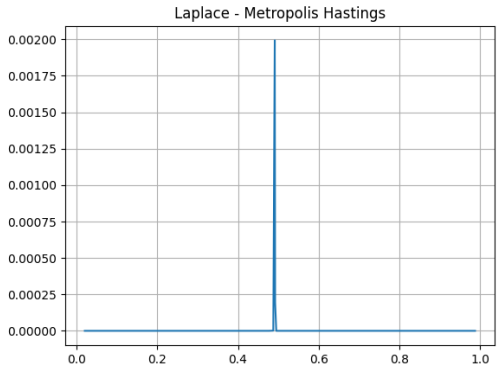


(d) Discrete Coin - Enumeration Sampling

Figure 2: Coin



(a) Importance Sampling



(b) Métropolis-Hastings

Figure 3: Laplace

```
 0.00000 >> 0.00000
 1.05263 >> 0.96565
 2.10526 >> 2.27618
 3.15789 >> 2.94145
 4.21053 >> 3.07147
 5.26316 >> 2.39646
 6.31579 >> 2.86714
 7.36842 >> 2.79719
 8.42105 >> 3.58989
 9.47368 >> 5.32409
10.52632 >> 9.23163
11.57895 >> 8.47337
12.63158 >> 9.04129
13.68421 >> 10.27966
14.73684 >> 11.67216
15.78947 >> 13.72597
16.84211 >> 15.11842
17.89474 >> 15.26178
18.94737 >> 14.99519
20.00000 >> 15.15904
```

(a) Importance Sampling

```
 0.00000 >> 0.00000
 1.05263 >> 1.46368
 2.10526 >> 4.20068
 3.15789 >> 5.15743
 4.21053 >> 5.65205
 5.26316 >> 6.07756
 6.31579 >> 7.17788
 7.36842 >> 7.32570
 8.42105 >> 8.17849
 9.47368 >> 9.04020
10.52632 >> 8.79149
11.57895 >> 9.72571
12.63158 >> 9.67666
13.68421 >> 10.25567
14.73684 >> 10.60707
15.78947 >> 11.47216
16.84211 >> 11.76923
17.89474 >> 13.92129
18.94737 >> 12.17048
20.00000 >> 12.12221
```

(b) Métropolis-Hastings

Figure 4: Hidden Markov Model

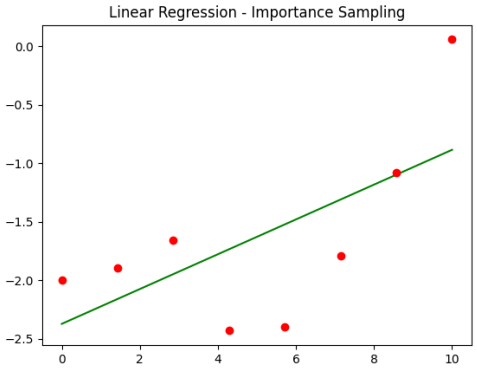
3.2 Embranchements

La raison d'être du modèle `branching` est de permettre de vérifier que l'inférence par énumération fonctionne correctement : si une boucle if ou for dépend d'un tirage aléatoire précédent, et qu'elle change le reste de l'exécution du programme (en augmentant le nombre de samples réalisés par exemple), alors on examine bien toutes les exécutions possibles.

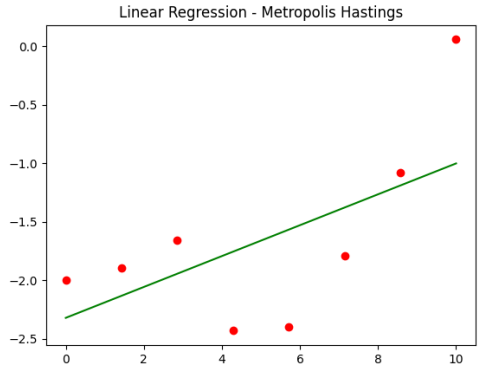
3.3 Régression linéaire

Nous avons implémenté un exemple de régression linéaire semblable à celle présentée lors du TP avec Stan. Il choisit les paramètres représentant la droite et l'erreur dans une distribution uniforme, puis applique un observe pour chaque point donné en entrée et enfin retourne les paramètres choisis en valeurs. Ces paramètres sont ensuite récupérés et triés par leur score associé, qui représente l'adéquation de la droite représentée par ces paramètres avec les points donnés en entrée. On peut ensuite choisir de ne garder que les valeurs associées aux meilleurs scores. Enfin on effectue une moyenne pondérée par les scores des paramètres représentant la droite. Les graphiques en figure 5 montrent les résultats obtenus par Importance Sampling et Métropolis-Hastings sur cet exemple.

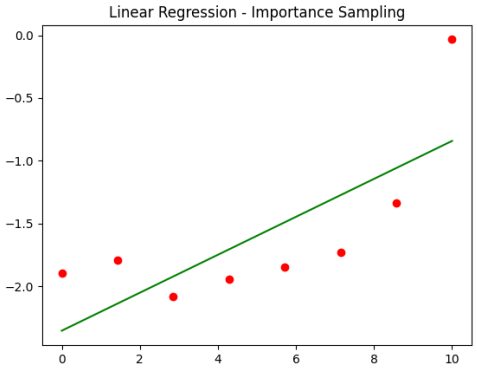
Uniquement les 100 meilleurs scores ont été gardés. Dans la majorité des cas les deux fonctions ont des résultats très proches. Il arrive cependant quelques cas où les résultats sont davantage différents, comme on peut le voir dans les figures 5c et 5d.



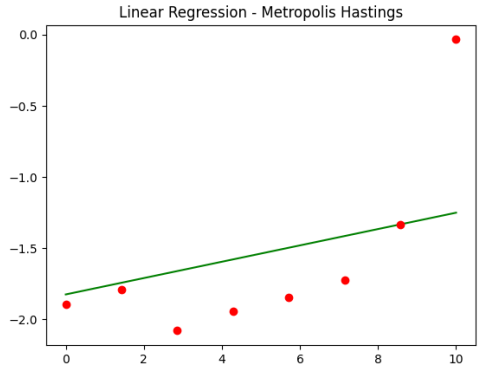
(a) Importance Sampling



(b) Métropolis-Hastings



(c) Importance Sampling



(d) Métropolis-Hastings

Figure 5: Régression linéaire

3.4 Balles rebondissantes

Il s'agit là du problème original suggéré dans le sujet.

La fonction `deterministic_bouncing_ball(ball_pos, dt, ground, bucket, obstacles)` permet de simuler physiquement une balle lachée depuis la position initiale `ball_pos` sans vitesse initiale . Cette balle a un rayon nulle, et elle est soumise à la gravité ainsi qu'à des frottements quadratiques avec l'air. A chaque pas de temps `dt`, la nouvelle position et vitesse de la balle sont calculés par la méthode d'Euler explicite. `ground`, `bucket`, et les éléments d'`obstacles` sont du même type : ce sont des obstacles contre lesquelles la balle peut rebondir.

La simulation se termine dès que la balle touche le sol, et nous n'expliqueront pas plus en détail son implémentation. Notons simplement la présence d'une fonction `plot_traj` qui affiche le résultat de la simulation : le sol en marron, le seau en noir, les plateformes en vert, et en rouge la trajectoire de la balle (fig 6).

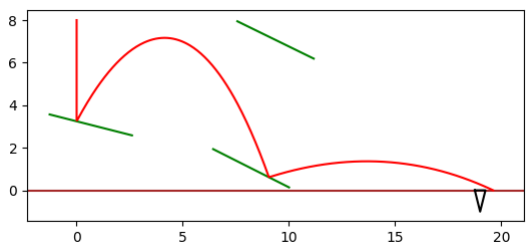
`bouncing_ball` est la modèle probabiliste utilisant la simulation. Son but est de se comporter comme un algorithme d'approximation, et d'optimiser le placement et l'angle des plateformes pour que la balle tombe au plus près du seau. Pour cela, on calcule la distance du seau à l'endroit où la balle est tombée (cette distance étant $+\infty$ si la simulation n'a pas pu terminer), et on utilise la méthode factor pour l'utiliser comme score que l'on attribue à une exécution.

De multiples données sont passées au modèle. D'une part celles utiles à la simulation comme la position initiale de la balle, le pas de temps, la position du sol et du seau. D'autre part, celles utiles à la création de plateformes, comme leur nombre, la zone dans laquelle elles peuvent se trouver, et leur largeur.

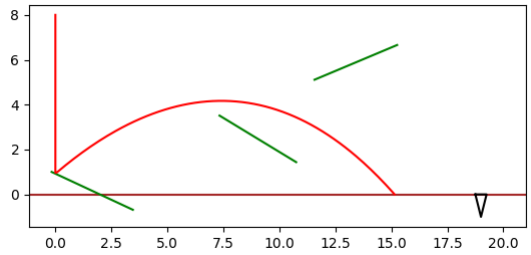
Le modèle retourne un vecteur de taille 3 fois le nombre de plateformes, indiquant pour chacune d'entre elles son angle de rotation ainsi que les deux coordonnées de sa position.

Pour augmenter les chances d'avoir des configurations intéressantes de plateforme, on met le seau suffisamment loin pour encourager les méthodes d'inférence à utiliser plusieurs plateformes, et on en rajoute une de plus aux 2 initialement voulues pour que la balle ait plus de chance de rencontrer plus de plateformes.

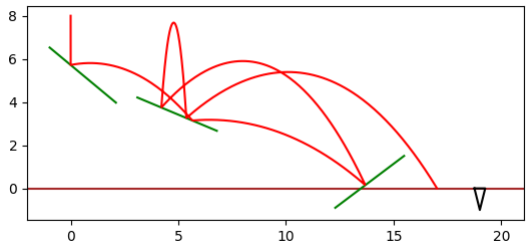
On obtient alors des distributions pour lesquelles on peut afficher la meilleure trajectoire obtenue (fig 6).



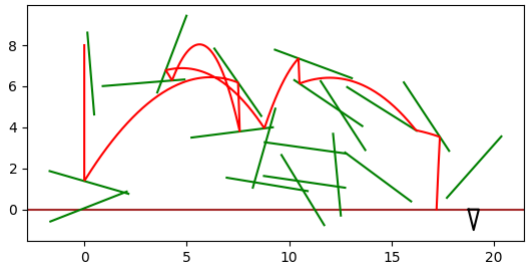
(a) Inference Sampling



(b) Metropolis-Hasting



(c) Rare cas où les 3 plateformes sont utilisées



(d) Le chaos

Figure 6: Balle rebondissante

On constate que sauf exception (fig 6c), on n'utilise que très rarement les trois plateformes.

Inference Sampling parvient généralement à trouver une solution utilisant deux plateformes parmi les trois, alors que Metropolis Hastings a plus de difficultés et n'en utilise souvent qu'une. C'est sûrement dû à son fonctionnement, qui l'encourage à ne retirer qu'un sample à chaque itération, limitant son exploration de toutes les valeurs. De plus, la plupart du temps, la nouvelle configuration d'une plateforme n'aura pas d'incidence sur la trajectoire si elle n'était pas déjà sur la trajectoire de la balle.