# A* appliqué à Frogger

Roc de Larouzière et Charles Cazenave

22 août 2025

## 1 Présentation du jeu

*Frogger* est un jeu vidéo classique, apparu en 1981, où le joueur contrôle une grenouille devant traverser une route et une rivière pour atteindre une zone sûre. Le jeu combine réflexion et rapidité, nécessitant d'éviter des voitures sur la route et de sauter sur des troncs flottants sur la rivière. La version que nous utilisons est basée sur un projet Python utilisant `Pygame` pour modéliser le jeu : https://github.com/pecurka/Frogger-AI-bot.

À l'origine, ce projet utilisait une IA génétique pour déplacer la grenouille automatiquement. Nous avons modifié le code afin d'implémenter l'algorithme A*, en remplaçant la logique génétique par une recherche de chemin informée. Cette adaptation nécessite de représenter la grille, de prévoir le mouvement des obstacles et de sélectionner des mouvements sûrs.

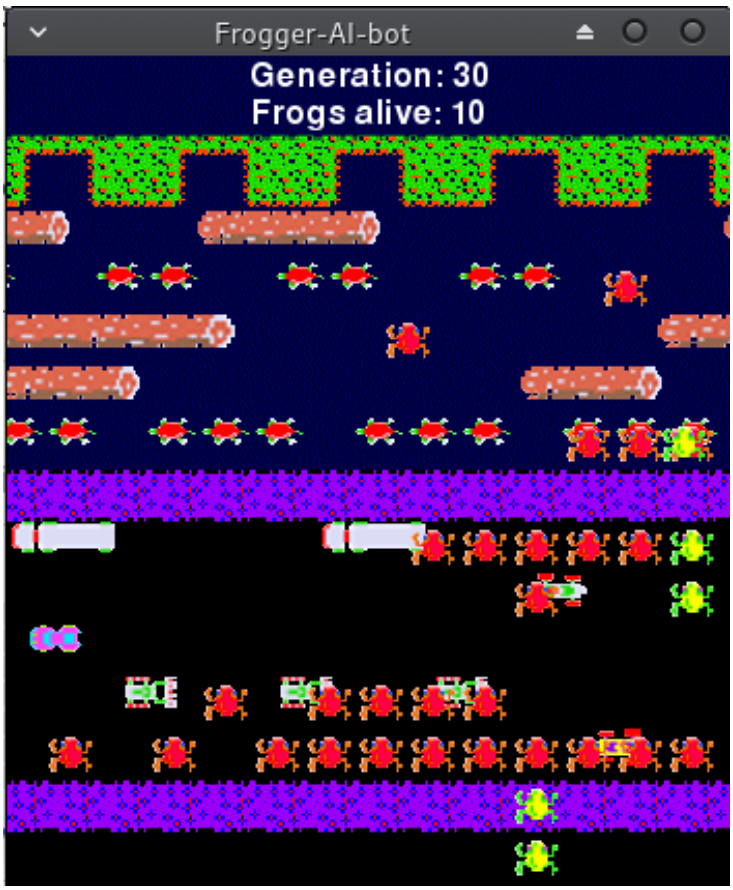


FIGURE 1 – Interface du jeu Frogger sous PyGame avec implémentation de l'algorithme génétique original

## 2 Algorithme utilisé : A*

A* est un algorithme de recherche informée qui combine le coût réel depuis l'état initial $g(s)$ et une estimation heuristique du coût restant $h(s)$ pour atteindre l'objectif. Le score total est donné par :

$$f(s) = g(s) + h(s)$$

L'algorithme explore en priorité les états avec le plus petit $f(s)$, garantissant un chemin optimal si l'heuristique est admissible.

### 2.1 Implémentation spécifique pour Frogger

Adapter l'algorithme A* à un environnement dynamique comme celui de *Frogger* exige une modélisation du monde qui tienne compte du mouvement constant des obstacles. Une représentation purement statique de la grille est insuffisante, car une case sûre à un instant t peut devenir mortelle à l'instant t+1. Notre approche a donc évolué à travers plusieurs itérations pour aboutir à une solution robuste. Notre développement s'est articulé autour de trois fonctions clés pour la génération de la grille de navigation :

- **create_grid** (l'approche statique) : Notre première implémentation se basait sur cette fonction, qui génère une grille représentant l'état du jeu à l'instant précis où elle est appelée. Elle marque correctement les voitures, les bûches et l'eau comme obstacles ou supports. Sa limite fondamentale est son incapacité à anticiper. La grenouille pouvait ainsi décider de sauter vers une case libre, sans savoir qu'un véhicule à grande vitesse allait l'occuper à la frame suivante, rendant de nombreux chemins "optimaux" en réalité mortels.
- **create_predictive_grid** (l'approche prédictive) : Pour pallier ce défaut, nous avons développé une fonction prédictive qui calcule la position des obstacles à la prochaine frame (t+1). En se basant sur la vitesse de chaque objet, elle offre une vision future du terrain. Cependant, cette méthode seule n'était pas parfaite non plus. En ne considérant que le futur, l'algorithme pouvait ignorer un danger immédiat. Cette faille s'explique par l'ordre d'exécution dans la boucle de jeu : le mouvement de la grenouille est un téléport instantané qui s'effectue avant la mise à jour des positions des autres obstacles pour la même frame. Par exemple, une case occupée par un camion à l'instant t mais qui sera libre à t+1 pouvait être considérée à tort comme une destination viable, alors que la grenouille y serait morte avant même d'atteindre l'état futur.
- **conservative_grid** (la synthèse robuste) : La solution la plus efficace a été de combiner les deux approches. Cette fonction fusionne les grilles de l'état actuel et de l'état futur. Une case n'est considérée comme sûre que si elle est praticable à la fois à l'instant t ET à l'instant t+1. Cette approche "conservatrice" élimine les mouvements risqués en ne retenant que les options garanties comme étant sûres sur les deux horizons temporels. C'est cette fonction qui est utilisée par notre agent final pour prendre ses décisions.

L'implémentation de A* dans ce contexte suit les étapes suivantes :

1. Initialisation de la liste ouverte avec l'état initial de la grenouille.
2. Sélection de l'état avec le plus petit $f(s)$ et génération de ses successeurs selon les déplacements possibles (haut, bas, gauche, droite, rester).
3. Vérification de la sécurité de chaque successeur grâce à `conservative_grid` et estimation du coût via l'heuristique (distance de Manhattan vers la zone sûre).
4. Mise à jour des coûts et ajout des successeurs sûrs à la liste ouverte.

5. Répétition du processus jusqu'à atteindre un état sûr.

Cette approche permet de gérer efficacement les obstacles dynamiques sans modéliser explicitement le temps, et offre une solution optimale dans la grille discrète du jeu.

### 2.2 Version $(x, y, t)$

Une version plus ambitieuse, intégrant le temps dans l'état $(x, y, t)$ pour gérer précisément la dynamique des obstacles, a été testée. Malheureusement, cette implémentation n'a pas produit de résultats corrects, probablement en raison de problèmes d'implémentation ou de prédiction des mouvements. Une future amélioration consisterait à mieux synchroniser la grille prédictive avec le temps et à gérer les collisions de manière plus robuste.

## 3 Résultats

Les performances ont été évaluées sur plusieurs niveaux avec des vitesses et densités d'obstacles variables.

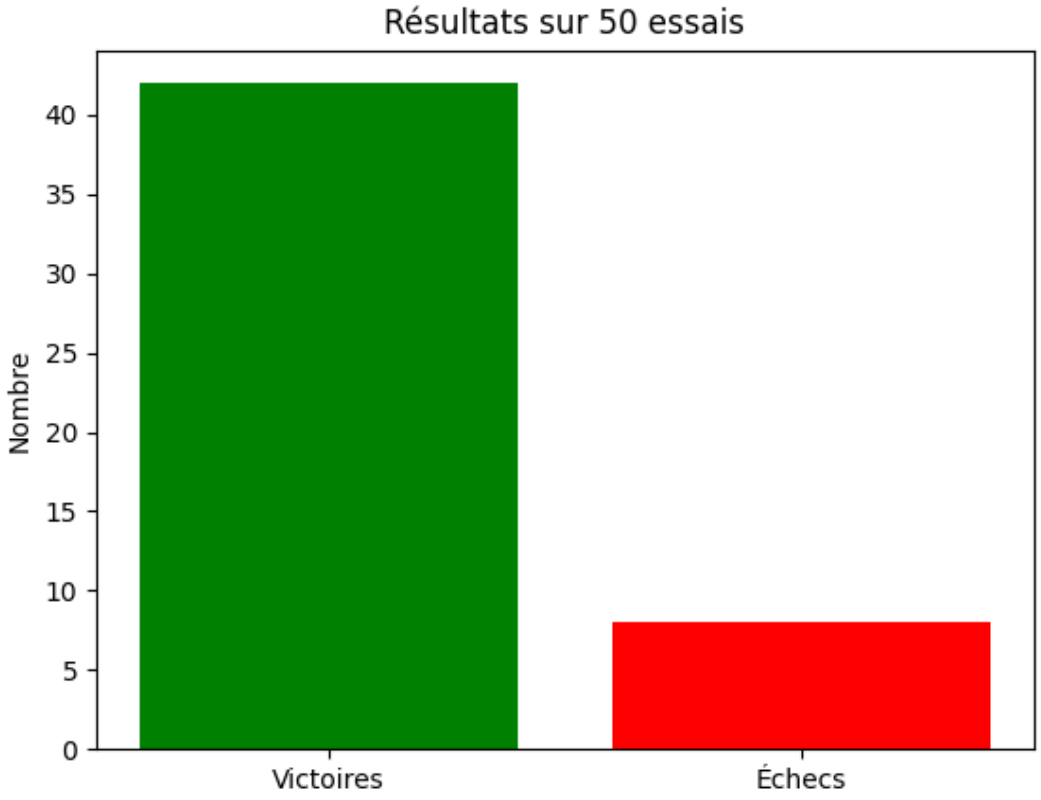


FIGURE 2 – Résultats A* sur 50 essais, plus de 80% de victoires grâce à *conservative_grid*

Notre implémentation de la version spatio-temporelle n'est pas tout à fait au point, les résultats sont censés être bien meilleurs, mais voici ce qu'on obtient :

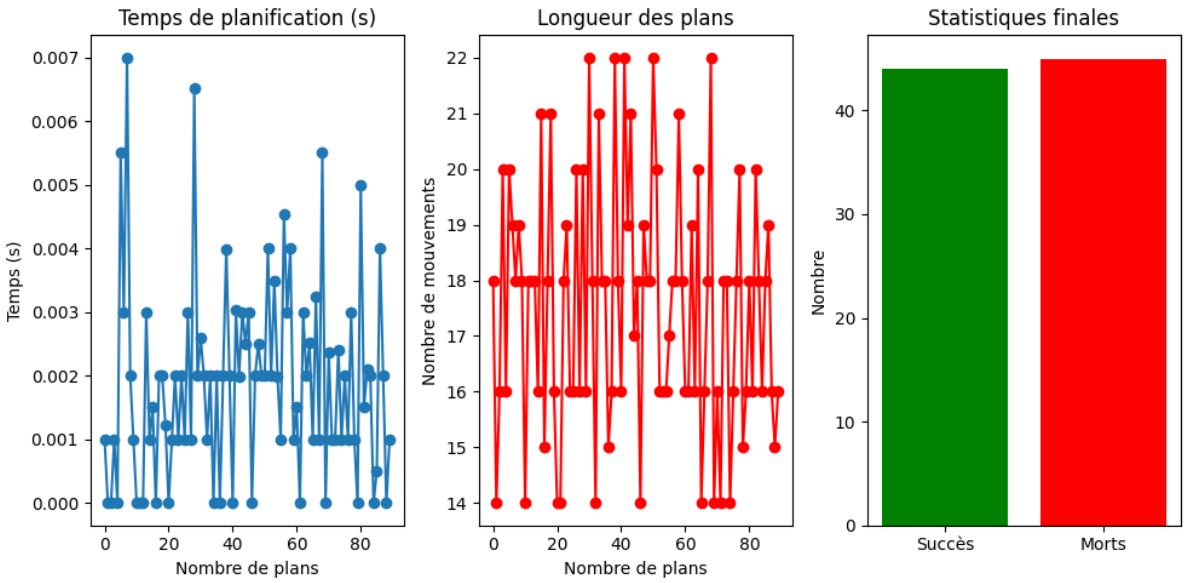


FIGURE 3 – Résultats A* Spatio-temporel

# 4 Conclusion

A* permet à la grenouille d'atteindre la zone sûre en minimisant les risques et en trouvant un chemin optimal. L'implémentation actuelle sur $(x, y)$ fonctionne bien, tandis que la version $(x, y, t)$ nécessite encore des améliorations pour gérer correctement la dynamique des obstacles. Nous avons également essayé d'atteindre les 100% de victoires avec l'implémentation de la grille en (x,y) en ayant remarqué certains patterns de morts qui se répétaient. Par exemple, la grenouille arrive dans un coin de la carte sur une bûche ou une tortue, qui se situe juste avant la ligne d'arrivée, et peut mal calculer le mouvement d'après (sûrement en raison d'un problème de coordonnées de la grille et de l'espace du jeu). Elle choisit alors de rester même si la bûche ou la tortue bouge et disparait, entrainant sa mort. Les pistes d'amélioration pour éviter ce "bug" pourront être étudiées ultérieurement.