# Projet de programmation : Ariane et le Minotaure

L'objectif de ce projet est d'implémenter un petit jeu de type "puzzle" et un algorithme de recherche automatique de solutions.

## L'histoire du minotaure

Dans la mythologie grecque, le Minotaure est un monstre fabuleux au corps d'homme et à tête de taureau.

### L'histoire officielle

Né des amours de Pasiphaé (épouse du roi Minos) et d'un taureau blanc envoyé par Poséidon, il est enfermé par Minos dans le Labyrinthe. Situé au centre de la Crète, le Labyrinthe est construit spécialement par Dédale afin que le Minotaure ne puisse s'en échapper et que nul ne découvre son existence. Tous les neuf ans, Égée, roi d'Athènes, est contraint de livrer sept garçons et sept filles au Minotaure qui se nourrit de cette chair humaine. Thésée, fils d'Égée, est volontaire pour aller dans le Labyrinthe et tuer le monstre.

La légende raconte que le Minotaure fut tué par Thésée grâce à l'aide d'Ariane, qui rêvait d'épouser Thésée, et lui donna une pelote de fil afin qu'il retrouve son chemin dans le Labyrinthe. On raconte que Thésée parvint à tuer le Minotaure grâce au glaive que Ariane avait volé à son père et à sortir du Labyrinthe. Ensuite il rentra chez son père et oublia complètement Ariane.

### Ce que certaines sources "fiables" nous apprennent

En réalité, il y avait plusieurs minotaures, et Thésée n'est pas du tout parvenu à les tuer et à sortir du Labyrinthe. Mort de peur, il resta coincé sans bouger en attendant les secours, et c'est Ariane qui dut à son tour entrer dans le Labyrinthe pour l'aider à sortir. Ensuite, dégoûtée par tant de lâcheté, elle rentra chez sa mère et oublia complètement Thésée.

## Objectif du projet

L'objectif de ce projet est de réaliser un petit jeu de puzzle tour par tour dans lequel le joueur incarne Ariane. Cette dernière doit parcourir le Labyrinthe pour retrouver Thésée et le guider jusqu'à la sortie tout en échappant aux Minotaures.

Outre l'implémentation du jeu lui-même, le défi principal de ce projet réside dans la programmation d'un *solveur*, ou générateur automatique de solutions.

Quelques exemples de parties et de recherches de solutions sont disponibles sur la chaîne YouTube d'Ariane.

# Règles du jeu

## Principe général

Le Labyrinthe est constitué d'une grille carrée dont les cases peuvent être ou non séparées par des murs. Chaque case du Labyrinthe peut contenir un ou plusieurs des objets ou personnages suivants :

1. Ariane, l'héroïne du jeu, contrôlée par le joueur ;
2. Thésée, le héros qu'Ariane doit sauver du Labyrinthe ;
3. les Minotaures, monstres invincibles mais peu avisés, peuplant le Labyrinthe ;
4. la Porte, la sortie du Labyrinthe.

Chaque tour de jeu se décompose comme suit :

1. Le joueur déplace Ariane ;
2. l'ordinateur déplace Thésée ;
3. si Ariane et Thésée se trouvent ensemble sur la Porte, la partie est gagnée ;
4. l'ordinateur déplace les Minotaures ;
5. si un Minotaure rencontre Thésée ou Ariane, la partie est perdue ;
6. on recommence à l'étape 1 jusqu'à la victoire ou la défaite.

## Déplacement des personnages

Les déplacements des personnages sont régis par les règles suivantes :

**Ariane** se déplace d'une case à chaque tour, vers le haut, le bas, la gauche ou la droite. Elle ne peut pas choisir de rester immobile, et ne peut traverser ni les murs, ni les Minotaures.

**Thésée** peut se déplacer comme Ariane ou bien rester immobile. S'il se trouve à une case de distance d'Ariane (c'est-à-dire, s'il peut la rejoindre en un seul déplacement), il choisit de le faire. Sinon, il reste immobile.

**Les Minotaures** peuvent se déplacer à chaque tour d'autant de cases qu'ils le désirent, dans *une* direction de leur choix (comme une tour au jeu d'échecs). Ils ne peuvent ni traverser les murs, ni occuper une case déjà occupée par un autre Minotaure. Il existe deux types de Minotaures :

- Les Minotaures *verticaux* : à chaque tour, les Minotaures verticaux tentent de se placer sur la même ligne qu'Ariane. Pour ce faire, ils se déplacent verticalement dans la direction (haut ou bas) d'Ariane jusqu'à rencontrer un obstacle (un mur ou un autre Minotaure) ou atteindre la ligne où se trouve Ariane. S'ils se trouvent sur la même ligne qu'Ariane, ils se déplacent horizontalement jusqu'à rencontrer un obstacle ou Ariane.
- Les Minotaures *horizontaux* : ils ont le fonctionnement symétrique des Minotaures verticaux. Ils tentent en priorité de se déplacer horizontalement jusqu'à se retrouver sur la même *colonne* qu'Ariane. S'ils se trouvent sur la même colonne qu'elle, ils se déplacent verticalement jusqu'à l'atteindre ou atteindre un obstacle.

Les Minotaures verticaux jouent *en premier*, suivis par les Minotaures horizontaux. S'il y a plusieurs Minotaures du même type, il est nécessaire de les numéroter pour distinguer quel Minotaure doit jouer en premier.

On remarque que les Minotaures n'essaient pas d'attaquer directement Thésée. Il est cependant possible que de mauvaises décisions de la part d'Ariane conduisent tout de même un Minotaure à rencontrer Thésée, en quel cas le héros est tué et la partie perdue.

**La porte** ne se déplace jamais.

# Réalisation du projet

Le projet se décompose en trois tâches principales (obligatoires).

## Tâche 1 : Représentation et chargement des niveaux

Un niveau du jeu est représenté dans un fichier texte comme suit :

```
10
+-+-+-+-+-+-+-+-+-+-+
|A            |   | |
+ + +-+-+-+-+ +   + +
| |               | |
+ + +-+     +-+   + +
|     |           | |
+     +           + +
|                   |
+         +-+-+ +-+ +
|         |   |     |
+         + + +   + +
|          V|     | |
+-+-+       + +-+ + +
| |  T          |   |
+ + +-+ +   +   +   +
|       |   |       |
+   +-+-+-+-+ + +-+-+
|             |H P  |
+ +-+-+       + +-+ +
|             |   | |
+-+-+-+-+-+-+-+-+-+-+
```

- Le nombre en début de fichier indique le nombre de lignes et de colonnes du Labyrinthe. Le Labyrinthe est toujours carré.
- Les symboles `-`, `|` et `+` représentent les murs. Il n'y a pas de différence de signification entre les trois symboles, mais on choisit pour améliorer la lisibilité du fichier texte d'utiliser `-` pour les murs horizontaux, `|` pour les murs verticaux, et `+` pour les lieux d'intersection possibles entre un mur horizontal et un mur vertical.
- Les symboles `A`, `T`, `V`, `H` et `P` représentent respectivement Ariane, Thésée, les Minotaures Verticaux et Horizontaux et la Porte.

Une représentation graphique du fichier texte donné ci-dessus est proposée sur la Figure 1. Notons que les murs verticaux et horizontaux sont représentés en mode graphique, par de simples lignes. Les lignes et colonnes du fichier texte pouvant contenir des murs ne correspondent pas à des cases du Labyrinthe, et les personnages ne peuvent pas s'y trouver. Le rôle des symboles `+` se limite à rendre le fichier texte plus lisible.

Votre programme doit respecter le format spécifié ci-dessus. Il vous sera ainsi possible de charger les fichiers proposés sur la page du projet, d'échanger des niveaux avec vos camarades, et pourquoi pas de consistuer collectivement une bibliothèque de niveaux intéressants !

**La première tâche du projet** consiste à programmer une fonction permettant de lire un fichier de jeu dans le format indiqué et d'en extraire les informations pertinentes dans une ou plusieurs structures de données bien choisies (on appelle ce genre de fonction un analyseur syntaxique).

La structure de données suggérée pour représenter un Labyrinthe est une liste de listes de dimensions `2*n + 1` par `2*n + 1`, où `n` est le nombre de lignes et de colonnes du Labyrinthe.

Si `laby` est une telle structure de données, la case `j` de la ligne `i` du Labyrinthe sera représentée par l'élément `laby[2*i+1][2*j+1]`. La présence ou l'absence de murs autour de cette case sera représentée par les éléments `laby[2*i][2*j+1]` (mur du haut), `laby[2*i+1][2*j]` (mur de gauche), etc. Les cases voisines seront représentées par les éléments `laby[2*(i-1)+1][2*j+1]` (voisin du haut), `laby[2*i+1][2*(j-1)+1]` (voisin de gauche), etc.

## Tâche 2 : Réalisation du moteur de jeu

**La seconde tâche du projet** consiste à programmer l'interface et les commandes du *jeu*, c'est-à-dire la partie qui propose à l'utilisateur d'essayer de trouver, en déplaçant Ariane, une solution à la grille chargée par l'analyseur syntaxique.

Le programme réalisant cette tâche peut se décomposer en quatre phases principales, que l'on répètera dans cet ordre jusqu'à la fin du jeu :

1. **L'affichage** : on dessine dans une fenêtre bien dimensionnée l'état actuel du jeu, c'est-à-dire la position des murs, de la porte, et des divers personnages.
2. **L'attente d'un coup légal du joueur** : on attend que le joueur appuie sur l'une des flèches du clavier, indiquant le déplacement d'Ariane souhaité. On vérifie que le déplacement est possible, (Ariane ne traverse ni un mur, ni un Minotaure), et on met à jour la position d'Ariane.
3. **Le déplacement de Thésée** : on effectue le déplacement de Thésée (s'il peut bouger), on vérifie la condition de victoire, et on arrête le jeu le cas échéant.
4. **Le déplacement des minotaures** : on effectue le déplacement de chacun des Minotaures verticaux, puis horizontaux, on vérifie la condition de défaite, et on arrête le jeu le cas échéant.

Une interface ergonomique est attendue pour cette étape. Il faudra au minimum permettre au joueur de charger la grille de son choix parmi un ensemble de grilles disponibles, afficher un écran de félicitations en cas de victoire, et afficher un menu proposant au joueur de quitter, de réessayer ou de charger une autre grille en cas de défaite.

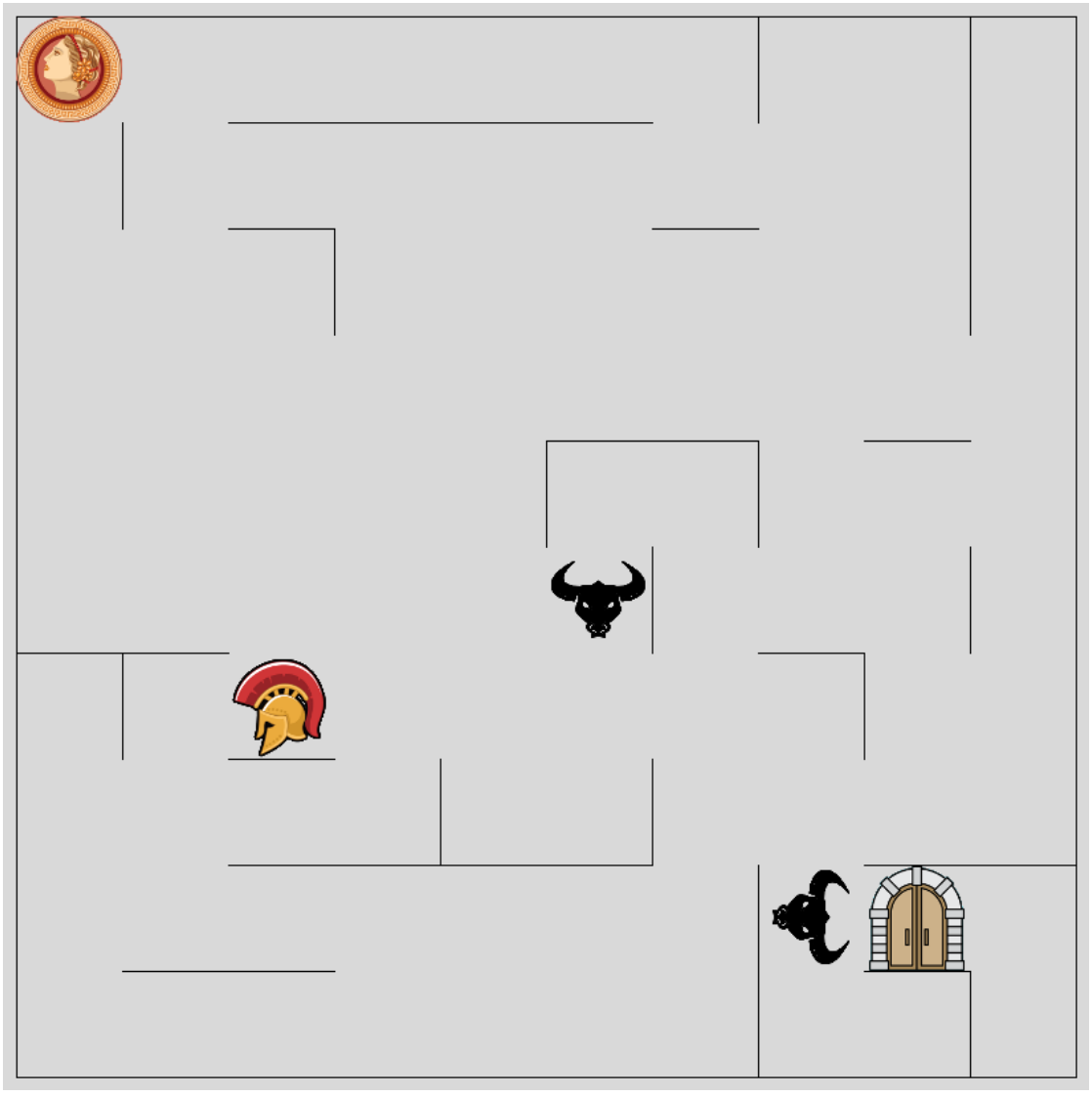

Figure 1: Labyrinthe 1

L'interface devra également proposer une fonction "Annuler", permettant de revenir en arrière d'un coup. Pour mettre cette fonction en place, il sera nécessaire de mémoriser (par exemple dans une liste) l'historique du jeu, c'est-à-dire les positions qu'ont occupées Ariane, Thésée et les Minotaures, à chacun des tours depuis le début de la partie. Ainsi, lorsque le joueur veut annuler son dernier coup, il suffit d'effacer la dernière entrée de l'historique, et de réinitialiser les positions d'Ariane, de Thésée et des Minotaures aux valeurs renseignées dans l'avant-dernière entrée de l'historique.

## Tâche 3 : Recherche de solutions

La troisième tâche du projet consiste à implémenter un algorithme de recherche automatique de solution, ou *solveur*, pour le jeu établi dans la section précédente. On ne s'intéresse pour l'instant qu'à la réalisation d'un solveur simple, permettant de déterminer quelles sont les configurations *gagnantes* du jeu, c'est-à-dire celles à partir desquelles il existe une solution, et inversement, quelles sont les solutions *perdantes*, celles à partir desquelles il n'est pas possible de gagner.

Nous allons commencer par implémenter un algorithme de recherche naïf (du même type que l'algorithme de coloration de zone vu en cours, appelé algorithme de *backtracking*, ou algorithme de recherche en profondeur), qui depuis un état donné du jeu (qu'on appelle une *configuration*) va essayer tous les coups possibles pour Ariane et rechercher récursivement une solution depuis chaque nouvelle configuration obtenue.

Commençons par remarquer que puisque les murs et la Porte ne sont jamais modifiés, une configuration du jeu est entièrement décrite par la donnée des positions d'Ariane, de Thésée et des Minotaures. Notons $C$ une configuration du jeu, et notons $V$ l'ensemble des configurations déjà visitées, (initialement, $V$ est vide). L'ensemble $V$ (qu'on pourra représenter en Python par un objet de type `set`) sert à éviter les appels récursifs infinis.

**Algorithme de recherche (pseudo-code).** L'algorithme exécuté depuis une configuration $C$ procède comme suit :

1. Si dans $C$, Ariane et Thésée se trouvent sur la Porte, répondre `Vrai` : la configuration est évidemment gagnante, puisque le joueur a gagné.

2. Si dans $C$, Ariane ou Thésée ont été tués par un Minotaure, répondre `Faux` : il n'est plus possible de gagner, puisque la partie est perdue.

3. Sinon, on commence par ajouter $C$ dans $V$. Puis pour chacune des directions $d \in \{\text{haut}, \text{bas}, \text{gauche}, \text{droite}\}$ :
   - si $d$ n'est pas un coup valide pour Ariane, on passe à la direction suivante;
   - si $d$ est un coup valide, on calcule la configuration $C'$ obtenue après :
     - le déplacement d'Ariane dans la direction $d$,
     - le déplacement correspondant de Thésée,
     - le déplacement des Minotaures.
   - Si $C'$ est déjà dans $V$, on passe à la direction suivante.
   - Sinon, on relance récursivement la recherche à partir de $C'$.

  - Si la recherche à partir de $C'$ répond `Vrai`, on répond `Vrai` pour $C$ : une solution a bien été trouvée pour $C$. Elle consiste à jouer $d$, puis à suivre une solution possible pour $C'$ (on sait récursivement qu'il en existe au moins une). Il est alors inutile de tester les autres directions.
  - Si la recherche à partir de $C'$ répond `Faux`, on passe à la direction suivante.

- Si aucune direction à partir de $C$ n'aboutit à une solution, on répond `Faux` : il n'existe pas de solution à partir de $C$, puisque tous les coups sont perdants.

Ce solveur simple permet uniquement de savoir si la configuration donnée en argument est gagnante ou non. En s'appuyant sur cet algorithme, on proposera plusieurs niveaux de perfectionnement, par ordre de priorité :

1. Le solveur propose un mode graphique, affichant dans une fenêtre les configurations visitées au fil du calcul. Attention, l'affichage ralentit bien sûr énormément la recherche, c'est pourquoi le mode graphique est seulement une **option** qu'il doit être facile de désactiver au besoin.

2. Lorsque la configuration donnée en argument est gagnante, le solveur renvoie une solution sous la forme d'une liste de coups à jouer au lieu de simplement répondre `Vrai`. Par exemple, une solution possible pour le Labyrinthe 1 est la liste de coups :

```
['Down', 'Down', 'Down', 'Down', 'Right', 'Right', 'Down', 'Up', 'Up', 'Right',
'Up', 'Right', 'Right', 'Right', 'Right', 'Left', 'Down', 'Right', 'Down',
'Down', 'Right', 'Down', 'Down', 'Left', 'Down', 'Right']
```

3. À la fin du calcul, le solveur affiche graphiquement la solution trouvée. Le jeu se lance dans les conditions habituels, mais Ariane est contrôlée par le programme, et joue successivement les coups qui mènent à la victoire.

## ($\star\star$) Optionnel : recherche d'une solution minimale

Il est possible (et même facile : amusez-vous à trouver un contre-exemple !) de prouver que les solutions produites par l'algorithme de recherche en profondeur ne sont pas nécessairement minimales en nombre de déplacements. En d'autres termes, ce n'est pas parce que l'algorithme de recherche en profondeur renvoie une solution en $n$ coups qu'il ne peut pas en exister une autre nécessitant moins de coups.

Pour calculer une solution minimale à une grille donnée, nous allons adopter une nouvelle stratégie d'exploration du Labyrinthe qui considère toutes les séquences de coups possibles par ordre de longueur. Cette méthode est traditionnellement appelée recherche *en largeur* (par opposition à la recherche en profondeur). On peut décrire cette stratégie comme suit (en reprenant les notations du paragraphe précédent) :

**Algorithme de recherche de solution minimale (pseudo-code).**

On déclare une liste `a_traiter` que l'on remplit initialement avec la configuration donnée en argument, et on ajoute cette configuration dans $V$.

Tant que `a_traiter` n'est pas vide :

On retire de la liste `a_traiter` son premier élément : la configuration $C$.

- Si dans $C$ la condition de victoire est remplie, répondre `Vrai`.
- Si dans $C$ la condition de défaite est remplie, passer à la configuration suivante de la liste `a_traiter`.
- Sinon, pour chaque direction $d \in \{\text{haut}, \text{bas}, \text{gauche}, \text{droite}\}$ :
  - On teste si $d$ est un déplacement valide pour Ariane, sinon on passe à la direction suivante.
  - On calcule la configuration $C'$ résultant du déplacement d'Ariane, et des déplacements correspondants de Thésée et des Minotaures.
  - Si $C'$ n'est pas dans $V$, on l'ajoute dans $V$ et en fin de liste `a_traiter`.

Si `a_traiter` est vide, on renvoie `Faux`. (*Note : une liste gérée de cette façon s'appelle une file.*)

Cet algorithme garantit que chaque configuration accessible est visitée dans l'ordre du nombre minimal de coups nécessaires pour l'atteindre, jusqu'à ce qu'une configuration satisfaisant la condition de victoire soit trouvée. On peut bien sûr appliquer à cette méthode les mêmes raffinements qu'à la recherche en profondeur, afin de retourner la solution trouvée, faire l'affichage graphique des configurations visitées et jouer la solution trouvée.

## ($\star$) Optionnel : le Défi d'Ariane

Ariane a pris note pour vous des Labyrinthes les plus difficiles qu'elle a parcourus et vous lance un défi : saurez-vous faire preuve d'autant d'astuce qu'elle pour retrouver Thésée et échapper aux Minotaures ? Sûre d'elle, Ariane a choisi de vous provoquer en révélant la solution à la grille `defi0.txt` sur sa chaîne YouTube. Saurez-vous trouver une solution pour chacune des grilles `defi1.txt`, `defi2.txt` et `defi3.txt` ?

Ces grilles sont d'une difficulté extrême, mais si vous êtes arrivés jusque-là, les résoudre devrait être un jeu d'enfant. Et si vous avez vraiment le goût du défi, essayez donc d'en trouver une solution minimale !

Donnez dans votre rapport la solution produite par votre solveur pour chacune des grilles et indiquez le nombre de coups de votre solution. Indiquez si vous trouvez (à la main par exemple) une meilleure solution et si vous pensez que votre solution est minimale ou non.

## Améliorations possibles

Une fois les trois tâches obligatoires terminées, **et seulement à cette condition**, il est possible de s'attaquer à des améliorations supplémentaires du jeu. Voici quelques suggestions ; attention, certaines sont plutôt faciles tandis que d'autres sont très difficiles.

(⋆) Implémenter une fonction "Indice" qui annonce au joueur si la partie peut encore être gagnée, ou bien annoncer immédiatement la défaite au joueur dès qu'il n'y a plus de solution possible.

(⋆) Ajouter des règles supplémentaires pour enrichir le jeu. Par exemple :

- Objets à usage unique pour tuer les Minotaures. Permet par exemple une nouvelle condition de victoire : tuer tous les Minotaures.
- Nouveaux types de monstres. Par exemple, Fantômes pouvant traverser les murs.
- Clefs et portes fermées à clef.
- Possibilité de déplacer les murs sous certaines conditions (⋆⋆).

(⋆⋆) Implémenter un mode "Survie" : le Labyrinthe n'a pas de sortie. Lorsque le déplacement d'un Minotaure est arrêté par un mur, le mur est détruit. Combien de tours peut-on survivre au maximum ?

(⋆⋆) Permettre au joueur d'enregistrer une partie en cours ou de charger une partie enregistrée. L'enregistrement devra être fait dans un fichier de manière à pouvoir être récupéré lors d'une autre session de jeu.

(⋆⋆) Implémenter un générateur de niveaux aléatoires solubles (c'est-à-dire pour lesquels il existe une solution). **Cette amélioration nécessite un solveur correct.**

(⋆⋆⋆) Implémenter un générateur de niveaux de difficulté choisie. L'utilisateur choisit la longueur de la solution minimale. **Cette amélioration nécessite un solveur en largeur correct.**

(⋆⋆⋆⋆⋆) Thésée ne reste plus immobile. Il joue désormais de façon optimale. Il peut se déplacer comme Ariane, et joue systématiquement de manière à rendre la partie possible (et minimale) en fonction des choix d'Ariane. **Cette amélioration nécessite de repenser entièrement le moteur du jeu et le solveur, et de concevoir des niveaux adaptés.**

(⋆⋆⋆⋆⋆) Jeu en *temps réel* : les Minotaures n'attendent plus leur tour pour jouer, mais se déplacent par exemple d'une case toutes les secondes, tant qu'ils ne sont pas bloqués. Ariane et Thésée sont limités à une certaine vitesse de déplacement. **Cette amélioration nécessite de repenser entièrement le moteur du jeu, la récupération des actions du joueur et de concevoir des niveaux adaptés.**

Si vous choisissez d'implémenter les deux dernières modifications, il vous sera quand même demandé une démonstration du moteur initial. Conservez le !

Toute autre amélioration est envisageable selon vos idées et envies, à condition d'en discuter au préalable avec un de vos enseignants. En effet, il serait dommage que ce que vous considérez comme une amélioration ne présente en réalité que peu d'intérêt de programmation ou ne dénature complètement le reste du travail.