|  |  |
| --- | --- |
| SMA  Résolution du Taquin | Résumé  Dans ce rapport est exposé une méthode de résolution du Taquin à l’aide d’un système multi agents.  Jérémy Dollé, Etienne Debard  5A Informatique |

Table des matières

[Introduction 2](#_Toc501648440)

[Librairie MadKit 2](#_Toc501648441)

[Approche naïve 2](#_Toc501648442)

[Principe général 2](#_Toc501648443)

[Exemple 3](#_Toc501648444)

[Etape 1 : Déplacement simple 3](#_Toc501648445)

[Etape 2 : Déviation 3](#_Toc501648446)

[Etape 3 : Amélioration du déplacement 4](#_Toc501648447)

[Limite 4](#_Toc501648448)

[Approche multi-agent avec dialogue 4](#_Toc501648449)

[Principe général 4](#_Toc501648450)

[Système d’agression 4](#_Toc501648451)

[Heuristique de choix d’une contrainte 5](#_Toc501648452)

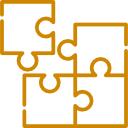
[Heuristique de choix d’une place 5](#_Toc501648453)

[Exemple 6](#_Toc501648454)

[Résultats 7](#_Toc501648455)

[Conclusion 8](#_Toc501648456)

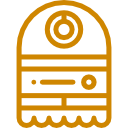
# Introduction

Dans ce rapport nous allons exposer les choix que nous avons pris pour la résolution du système du Takin. Dans un premier temps nous allons formuler une approche naïve en exposant le principe général ainsi que le résultat obtenu. Ensuite, nous allons expliquer une approche multi-agent en commençant par le principe général ainsi que le résultat obtenu.

# Librairie MadKit

Le but de ce TP étant l’implémentation d’un algorithme d’apprentissage, nous avons décidé de choisir une librairie pour tout ce qui est question d’affichage pour éviter de perdre du temps à concevoir une interface et aussi pour vérifier rapidement nos résultats. La librairie MadKit est très simple d’utilisation est colle parfaitement à nos besoins, c’est pour cette raison que nous avons opté pour cette dernière.

# Approche naïve

Pour commencer, nous avons implémenté un algorithme naïf que nous avons petit à petit amélioré. Dans la suite, nous avons commencé par expliquer le principe général avec les améliorations apporté petit à petit à notre solution. Ensuite des exemples afin d’illustrer notre propos et pour finir les limites à cette solution.

## Principe général

Premièrement, notre agent agissait de la manière suivante : l’agent se déplace sur la bonne colonne, s’il est sur la bonne colonne alors il se déplace sur la bonne ligne. De cette façon, si le nombre d’agents est élevé alors nos agents se bloquaient les uns les autres assez rapidement. Nous avons donc décidé d’implémenter de façon naïve une déviation. Pour ce faire, nous avons fait en sorte que lorsqu’un agent veut se déplacer sur une case déjà occupée alors il choisit une autre case paris celle disponibles. Afin d’éviter que nos agents fassent les mêmes déplacements les cases indisponibles sont celle qui sont occupées + la case précédemment occupée par l’agent. Afin d’améliorer l’efficacité de notre algorithme, nous avons changé la stratégie de déplacement des agents (bonne colonne puis bonne ligne). Nous avons dit à notre agent de se déplacer sur la bonne ligne ou la bonne colonne en fonction de ce qui est le plus proche : si la colonne est plus proche que la ligne alors l’agent va d’abord se placer sur la colonne et inversement.

## Exemple

Dans les trois exemples qui vont suivre, « X » est un agent, la case rouge représente la case cible de l’agent X.

### Etape 1 : Déplacement simple

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| X |  |  |  |  | X |  |  |  |  | X |  |  |  |  | X |
|  |  |  |  |  |  |  |  | Comme expliqué précédemment, L’agent « X » se déplace d’abord sur la bonne colonne pour ensuite se déplacer sur la bonne ligne. |  |  |  |  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  | X |

### Etape 2 : Déviation

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| X |  | O |  |  | X | O |  |  |  | O |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  | X  L’agent « X » se déplace de la même façon (colonne puis ligne). Par contre cette fois un autre agent se trouve sur son chemin. A l’étape 2 l’agent n’a que deux solutions car devant lui se trouve l’agent et derrière lui il se trouve la case d’où il vient et ne peut donc pas y retourner. Une fois la déviation faite alors l’algorithme normal reprend. |  |  |  |  | X |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  | X |

### Etape 3 : Amélioration du déplacement

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | X |  |  |  | Comme expliqué précédemment, Nous avons décidé d’améliorer le déplacement des agents en se déplaçant soit vers la bonne ligne soit vers la bonne colonne. Ici l’agent se trouve à une distance de 1 de la bonne ligne et 3 de la bonne colonne. L’agent va donc se déplacer sur la bonne ligne puis sur la bonne colonne. | X |  |  |  |  | X |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  | X |

## Limite

Après avoir implémenté cette approche naïve, nous pouvons relever quelques limites. En effet, les agents marchent indépendamment les uns des autres, il n’y a aucune communication entre eux. Ce qui pose un problème de taille : si un agent est bien placé il ne bougera plus ce qui fait que si un agent est entouré d’agents bien positionnés, il sera coincé et ne pourra pas atteindre son objectif.

# Approche multi-agent avec dialogue

## Principe général

L’étape suivante est d’ajouter un système de messagerie entre les agents afin que ceux-ci puissent dialoguer entre eux. Afin de se concentrer un maximum sur le principe de l’algorithme en lui-même et de ne pas trop perdre de temps sur l’implémentation, nous continuons d’utiliser Madkit qui intègre un system de messagerie avec possibilité d’attente de réponse.

### Système d’agression

Afin de se satisfaire, c’est-à-dire d’atteindre sa case « but », un agent peut se déplacer sur une case vide ou **agresser** un agent voisin. Ce nouveau comportement prend appui sur le système de messagerie de notre programme.

Ainsi, un agent cherche à se satisfaire. Pour se faire, il cherche la case voisine optimale à sa satisfaction. Si cette case est libre, il s’y déplace, et sinon, il agresse l’agent qui s’y trouve en lui faisant passer une contrainte. Après une agression, l’agent attend une réponse de l’agent agressé avant de continuer son processus. S’il ne reçoit pas de réponse après un certain temps, il réessaye de se satisfaire. Sinon, c’est qu’il a reçu une confirmation de l’agressé qu’il s’est déplacé, l’agresseur peut alors se déplacer à sa place.

Chaque agent se faisant agresser à l’obligation de fuir. Pour se faire, il cherche à son tour une place pour fuir. Si cette case est libre, il s’y déplace et notifie l’agent agresseur qu’il peut maintenant se déplacer, ou il agresse l’agent qui le gêne dans sa fuite en lui faisant également passer une contrainte.

### Heuristique de choix d’une contrainte

A chaque agression, l’agresseur choisi une contrainte à faire passer, que l’agressé devra obligatoirement respecter (c’est-à-dire ne pas se déplacer sur la contrainte).

Dans le cas où l’on attaque pour se satisfaire, la contrainte est simplement le but de l’agresseur.

Dans le cas d’une attaque pour fuir, la contrainte est la case où l’on veut fuir, donc la case sur laquelle se trouve l’agressé.

### Heuristique de choix d’une place

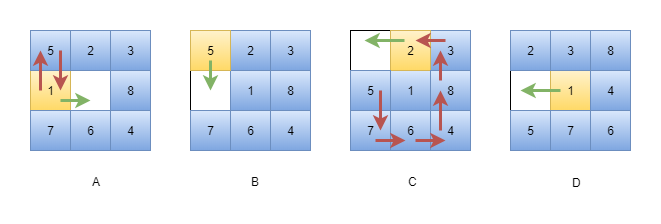
Le choix d’une place pour se satisfaire ou fuir se base sur 2 types de gradients. Le premier pondère chaque case en fonction de sa distance à une case blanche alors que le deuxième pondère la distance de chaque case par rapport au but d’un agent.

Ainsi, pour se satisfaire, on utilise le gradient de but, et on prend la case adjacente la plus proche du but (donc avec le coefficient de gradient le plus petit). S’il y a plusieurs cases équivalentes, on les départage en utilisant le deuxième gradient.

Puis, pour choisir une place pour fuir, on utilise le gradient des cases vides, afin de choisir la case adjacente la plus proche d’une case vide. Il est également interdit de fuir sur une place qui a été passée en contrainte par l’agresseur. Le choix de cette place pour fuir (ou agresser si nécessaire) est également pondéré par l’agresseur. En effet, on ne peut choisir de fuir sur la place de notre agresseur (donc l’agresser en retour) que si c’est notre seul choix possible (ex : la seule autre case possible est la contrainte de mon agresseur). Cette dernière contrainte est nécessaire afin d’éviter que deux agents ne se bloquent mutuellement sans fin.

Pour éviter de trop faire se déplacer les agents ayant atteint leur but, les deux gradients sont également pondérés par les cases avec le but satisfait (on augmente de 1 le coefficient du gradient de ces cases, ainsi elles seront moins privilégiées lors du choix d’une place.

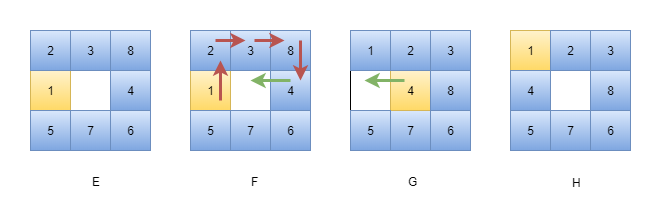
## Exemple



1. 1 agresse 5 pour se satisfaire, 5 agresse 1 en retour car 2 est satisfait.

Donc 1 fuit vers la case vide.

1. 5 se déplace vers la case vide après la fuite de 1.
2. 1 réagresse 5, et 5 ne fuit pas vers la case vide, car c’est le but de 1 (contrainte). Donc 5 attaque 7 et par réaction en chaine, 2 fuit vers la case vide.
3. 1 se déplace vers la case vide après la fuite de 5.



1. ------
2. 1 agresse 2, ensuite par réaction, 3 agresse 8 plutôt que de se déplacer sur la case vide, car c’est son but. 8 attaque 4 et 4 fuit sur la case vide. Alors 1 ensuite se déplacer sur son but et se satisfaire.
3. 4 se déplace sur la case vide pour se satisfaire.
4. ------

## Résultats

Notre implémentation de résolution de taquin en SMA nous permet de résoudre des problèmes de taille 5 par 5 avec deux cases blanches en 1 minute environ. Lorsque que l’on réduit le nombre de case vide à une, celui-ci à beaucoup plus de mal à se résoudre.

Nous pensons que ce problème est dû à un soucis de coordination entre les agents. En effet, ils cherchent tous à se satisfaire en même temps. Nous n’avons pas eu le temps d’expérimenter de solution pour ce problème, mais attribuer un poids de priorité à chaque agent pourrait peut-être améliorer notre SMA.

Enfin, nous avons également essayé avec des taquins beaucoup plus grands, les résultats sont similaires, c’est-à-dire qu’avec 8% de cases blanches (=2/25), nous obtenions une résolution assez facile, mais néanmoins avec une augmentation du temps de résolution. Ceci s’explique par le fait que le nombre d’agents à satisfaire est logiquement plus grand.

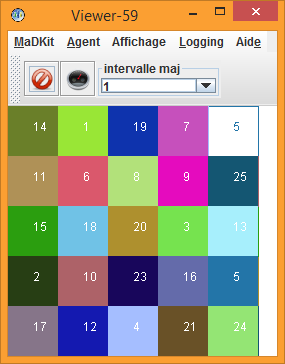


Figure : capture d’écran de notre agent d’affichage de la résolution

# Conclusion

L’utilisation de système multi agent dans ce genre de problème permet de décomposer la problématique en plusieurs but bien identifiables et plusieurs règles de logique facilement applicables. On obtient alors un system qui bien configurer est capable de résoudre des problèmes dont la solution optimale serait beaucoup trop compliquée à implémenter.

Dans le cas du taquin, cette solution est très souple. En effet, si l’on modifie l’environnement, celui-ci sera toujours capable de se résoudre. On peut par exemple faire varier le nombre de cases blanches ou ajouter des zones noires ou aucun agent ne peut se déplacer. Pour ce dernier cas, il suffit de mettre très élevé le coefficient de gradient des cases noires afin que les agents ne les choisissent pas dans leurs déplacements.

Enfin, afin d’améliorer les performances de notre algorithme, il y a sûrement un grand nombre de boucles à optimiser, mais par manque de temps nous ne nous somme pas penché sur cela, étant donné que le but de ce cours est avant tout la découverte de ce système de résolution.