Informatique décisionnelle

Rapport

Hicheme BEN GAIED

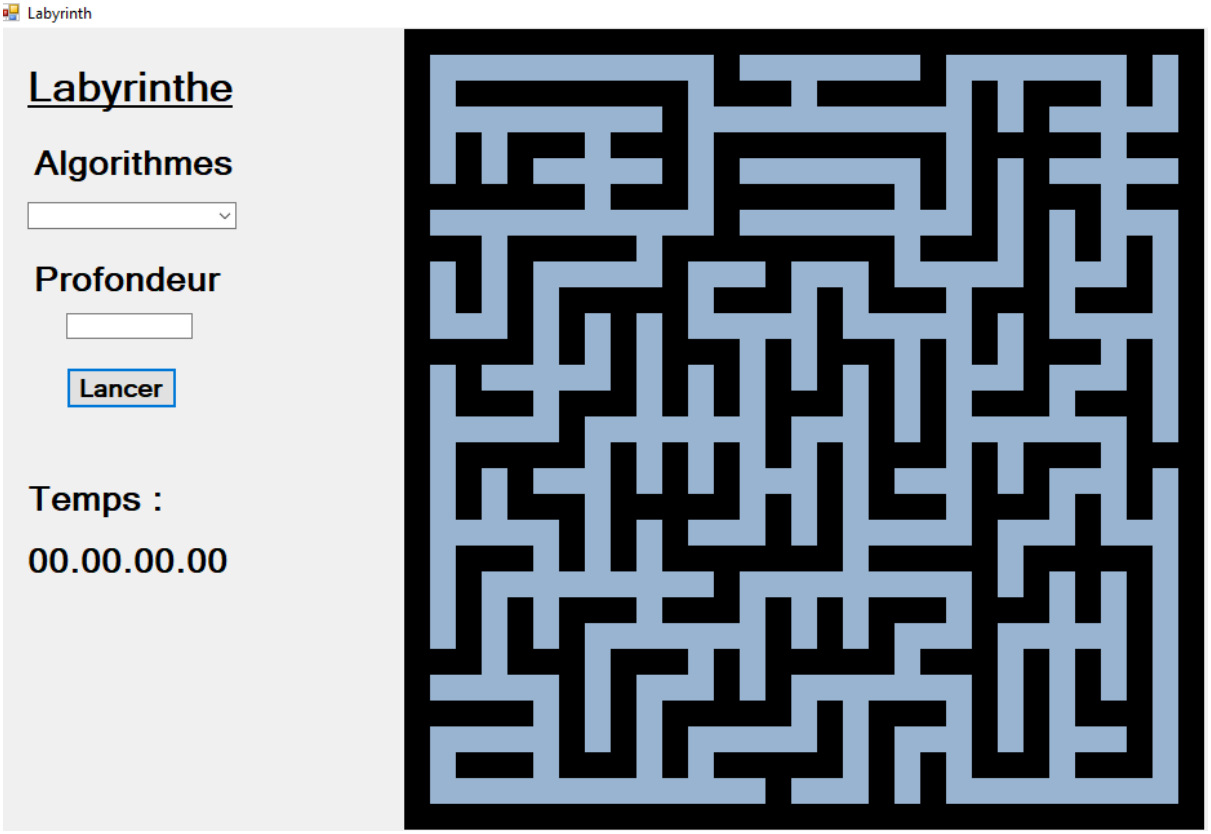
Etudiants Bloc M1 Informatique

20/11/2022

1) Projet Labyrinthe avec différentes cases.

Dans mon projet, je m'efforce de trouver un chemin à travers un labyrinthe en utilisant plusieurs algorithmes de recherche imposés. Ces algorithmes comprennent la recherche en profondeur (DFS), la recherche itérative en profondeur, Greedy Search, Hill Climbing et l'algorithme A*. Mon but est de comparer ces algorithmes et de déterminer lequel est le plus efficace pour résoudre ce labyrinthe particulier.

Voici une image qui illustre le labyrinthe utilisé dans ce projet.



1.1) DFS

L'algorithme DFS est relativement simple à mettre en œuvre et assez peu complexe. Dans mon implémentation de l'algorithme DFS(), j'ai choisi de privilégier le mouvement vers le haut en premier, puis vers la gauche, ensuite vers la droite et finalement vers le bas.
Si aucun de ces mouvements n'est possible, cela signifie qu'il n'y a aucune solution possible pour mon labyrinthe depuis le nœud de départ jusqu'au nœud goal.

Dans mon cas, si le nœud goal se trouve en dessous du nœud de départ, mon algorithme DFS() va explorer cette possibilité en dernier car il utilise un ordre de déplacement spécifié (haut, gauche, droite, bas). Dans le meilleur des cas, l'algorithme ne peut aller que vers le bas car les cases autour du nœud de départ sont des murs et donc atteindre la nœud goal du premier déplacement.
Cependant, dans le pire des cas, si l'algorithme peut aller vers le haut, à gauche ou à droite, l'exploration du chemin ne sera pas optimisée. L'algorithme peut alors également renvoyer un chemin beaucoup plus long alors que le nœud cible se trouve juste en-dessous.

1.1.1 Complexité et itération

La complexité de l'algorithme de recherche en profondeur (DFS) dans un labyrinthe de taille 31x31 dépend du nombre de nœuds qu'il doit explorer avant de trouver une solution. Comme l'algorithme explore les nœuds dans l'ordre où ils sont ajoutés à la frontière, la complexité dépend de la structure du labyrinthe et de la position du nœud goal par rapport au nœud de départ.

Dans le meilleur des cas, si le nœud goal est proche du nœud de départ et que le labyrinthe a un faible facteur de branchement(sous-noeuds), DFS pourra trouver une solution rapidement en parcourant peu de nœuds.
Dans ce cas, la complexité sera faible, de l'ordre de O(b*d), où b est le facteur de branchement de l'arbre de recherche et d est la profondeur du nœud cible le plus superficiel. Le nombre d’itération sera égale de 4x1 =4 dans le meilleur des cas.

Dans le pire des cas, si le labyrinthe a une structure complexe et que le nœud goal se trouve loin du nœud de départ, DFS pourra être obligé d'explorer chaque case du labyrinthe avant de trouver la sortie. Dans ce cas, la complexité sera de l'ordre de O(b^d), où b est le facteur de branchement de l'arbre de recherche et d est la profondeur du nœud cible le plus profond.
Pour un labyrinthe de taille 31 x 31, si le nœud cible se trouve au fond de l'arbre de recherche et que le facteur de branchement est élevé, la complexité de l'algorithme DFS peut atteindre O(b^d), où b^d représente le nombre total de nœuds de l'arbre de recherche. Si le facteur de branchement est de 4 et que la profondeur du nœud cible le plus profond est de 31, la complexité sera de l'ordre de 4^31, ce qui représente un nombre très élevé d'itérations.

1.1.2) Coût en mémoire

Le coût en mémoire de l'algorithme DFS dépend de la taille du labyrinthe et du nombre de cases explorées. Dans le meilleur des cas, l'algorithme ne devra explorer qu'une seule case avant de trouver la sortie, ce qui représenterait un coût en mémoire très faible.
Dans le pire des cas, l'algorithme devra explorer chaque case du labyrinthe avant de trouver la sortie, ce qui représenterait un coût en mémoire égal à la taille du labyrinthe.

Par exemple, si le labyrinthe a une taille de 31x31, le coût en mémoire de l'algorithme DFS sera de 31x31 = 961 cases dans le pire des cas.
Si la distance entre le noeud de départ et le noeud de fin est supérieure à la moitié de la taille du labyrinthe, alors le coût en mémoire sera encore plus élevé.

1.1.3) Vitesse

Vitesse du DFS
Pour le temps du DFS, j'ai fait une moyenne de 10 temps de mesures.
Moyenne_DFS = 0,3547s

1.2) Iterative Deepening

Non implémenté entièrement car j'ai un bug que je n'arrive pas à résoudre pour le calcul en profondeur. Mon algorithme revient bien à la case ou il doit continuer l'exploration s'il est bloqué mais il calcul mal la profondeur.

1.3) Hill Climbing

L'algorithme Hill Climbing est relativement simple à mettre en œuvre. Hill Climbing consiste en un DFS qui utilise l'heuristique. Pour calculer l'heuristique d'une case par rapport au noeud goal, j'ai utilisé la stratégie du Manhattan-Distance.
L'algorithme choisit la prochaine case de destination en fonction de l'heuristique. Hill Climbing va se déplacer à la case ayant l'heuristique la plus faible.

1.3.1 Complexité

.
Dans le pire des cas, si le labyrinthe est un mur infranchissable de tous côtés, l'algorithme devra parcourir toutes les cases du labyrinthe, ce qui donne une complexité en O(n^2), où n est la taille du labyrinthe (31x31 dans mon cas).
Dans le meilleur des cas, si le labyrinthe est entièrement vide et que la sortie est à côté de la case de départ, l'algorithme ne parcourra qu'une seule case, ce qui donne une complexité en O(1).
Dans la plupart des cas réels, la complexité de l'algorithme sera quelque part entre ces deux extrêmes. Plus il y a de murs et plus la sortie est éloignée de la case de départ, plus l'algorithme devra parcourir de cases, ce qui augmentera sa complexité.

1.3.2 Nombres d’itérations

Le nombre d'itérations de l'algorithme Hill Climbing dans le pire des cas et dans le meilleur des cas dépend de la complexité de l'algorithme, qui à son tour dépend du nombre de murs et du nombre de cases parcourues avant de trouver la sortie.
Dans le pire des cas, si le labyrinthe est un mur infranchissable de tous côtés, l'algorithme devra parcourir toutes les cases du labyrinthe avant de s'arrêter, ce qui donnera un nombre d'itérations égal au nombre de cases du labyrinthe (961 dans le cas d'un labyrinthe 31x31).
Dans le meilleur des cas, si le labyrinthe est entièrement vide et que la sortie est à côté de la case de départ, l'algorithme ne parcourra qu'une seule case et s'arrêtera immédiatement, ce qui donnera un nombre d'itérations égal à 1.

1.3.3 Coût en mémoire

Le coût en mémoire de l'algorithme Hill Climbing dépend du nombre de cases du labyrinthe et du nombre de cases parcourues avant de trouver la sortie.
Pour chaque case du labyrinthe, l'algorithme doit stocker si elle est un mur ou non et si elle a déjà été explorée ou non. Cela nécessite au minimum un booléen par case, ce qui donne un coût en mémoire minimum de 2 * n, où n est le nombre de cases du labyrinthe.
Pour un labyrinthe 31x31, cela donne un coût en mémoire minimum de 2 * 961 = 1922 booléens.
En plus de cela, l'algorithme utilise deux listes (frontier et explored) pour stocker les cases explorées et les cases à explorer. Ces listes peuvent stocker jusqu'à n éléments (un élément par case du labyrinthe), ce qui donne un coût en mémoire supplémentaire de 2 * n.
Pour un labyrinthe 31x31, cela donne un coût en mémoire supplémentaire de 2 * 961 = 1922 cases.
Enfin, l'algorithme utilise une liste intermédiaire (intermediar_list_for_sort) pour trier les cases adjacentes avant de les ajouter à la liste frontier. Cette liste peut également stocker jusqu'à n éléments, ce qui donne un coût en mémoire supplémentaire de n. Pour un labyrinthe 31x31, cela donne un coût en mémoire supplémentaire de 961 cases.
Au total, pour un labyrinthe 31x31, le coût en mémoire de l'algorithme Hill Climbing est de 1922 booléens + 1922 cases + 961 cases = 4805 cases.

1.3.4 Vitesse

Vitesse Hill Climbing
Pour le temps du Hill Climbing, j’ai fait une moyenne de 10 temps de mesures.
Moyenne_HC = 0,1331s

1.4 Greedy Search

L'algorithme Greedy Search est relativement simple à mettre en œuvre. Greedy Search tout comme Hill Climbing utilise l’heuristique pour choisir le prochain nœud. La différence entre ces 2 derniers et que Greedy Search commence à un point de départ donné et l'ajoute à la liste "explorée", qui enregistre les nœuds qui ont déjà été visités. Il ajoute ensuite tous les

voisins du point de départ à la liste "frontière", qui est une liste de nœuds qui sont considérés comme l'étape suivante.
L'algorithme trie ensuite la liste frontière dans l'ordre croissant en fonction de la valeur heuristique de chaque nœud (la méthode Manathan_Distance calcule cette valeur). La valeur heuristique est une estimation de la distance d'un nœud donné à l'objectif. Le nœud avec la valeur heuristique la plus faible est choisi comme étape suivante et est ajouté à la liste explorée. Le processus est alors répété pour le nouveau nœud courant jusqu'à ce que l'objectif soit atteint ou que la liste frontière soit vide (ce qui signifie qu'il n'y a plus de nœuds à explorer).

1.4.1 Complexité

En pire cas, la complexité de cet algorithme pour un labyrinthe 31x31 peut être de l'ordre de O(b^d), où b est le nombre de branches possibles à chaque étape (4 dans ce cas, car l'algorithme peut explorer les cases adjacentes à gauche, à droite, en haut ou en bas) et d est la distance entre le point de départ et l'objectif. Dans ce cas, plus le labyrinthe est grand et plus il y a de murs, plus le temps d'exécution de l'algorithme sera long.
Par exemple, si le labyrinthe est rempli de murs et que le point de départ et l'objectif sont situés aux extrémités opposées du labyrinthe, la complexité de l'algorithme sera de l'ordre de O(4^30) en pire cas, car il y a 30 étapes entre le point de départ et l'objectif (chaque étape consistant à explorer l'une des 4 cases adjacentes). Cela signifie que le temps d'exécution de l'algorithme sera très long dans ce cas, même pour un labyrinthe de petite taille.
En meilleur cas, la complexité de cet algorithme pour un labyrinthe 31x31 peut être de l'ordre de O(1), c'est-à-dire constante. Cela se produit lorsque le labyrinthe est complètement vide et que le point de départ et l'objectif sont adjacents. Dans ce cas, l'algorithme trouvera le chemin le plus court en un seul pas et le temps d'exécution sera indépendant de la taille du labyrinthe.

1.4.2 Nombres d’itérations

Il est difficile de donner un nombre précis d'itérations de cet algorithme en pire cas et en meilleur cas pour un labyrinthe 31x31.
En pire cas, lorsque le labyrinthe est rempli de murs et que le point de départ et l'objectif sont situés aux extrémités opposées du labyrinthe, le nombre d'itérations de l'algorithme sera très élevé, car il y a 30 étapes entre le point de départ et l'objectif (chaque étape consistant à explorer l'une des 4 cases adjacentes). Cela signifie que l'algorithme devra parcourir un très grand nombre de cases avant de trouver le chemin le plus court.
En meilleur cas, lorsque le labyrinthe est complètement vide et que le point de départ et l'objectif sont adjacents, le nombre d'itérations de l'algorithme sera très faible, car l'algorithme trouvera le chemin le plus court en un seul pas.

1.4.3 Coût en mémoire

En pire cas, lorsque le labyrinthe est rempli de murs et que le point de départ et l'objectif sont situés aux extrémités opposées du labyrinthe, le nombre de cases explorées sera très élevé, car il y a 30 étapes entre le point de départ et l'objectif (chaque étape consistant à explorer l'une des 4 cases adjacentes). Cela signifie que l'algorithme devra mémoriser un très grand nombre de cases avant de trouver le chemin le plus court.

En meilleur cas, lorsque le labyrinthe est complètement vide et que le point de départ et l'objectif sont adjacents, le nombre de cases explorées sera très faible, car l'algorithme trouvera le chemin le plus court en un seul pas.

1.4.4 Vitesse

Vitesse Greedy Search
Pour le temps du Greedy Search, j'ai fait une moyenne de 10 temps de mesures.
Moyenne_GS = 0,00911s

1.5 A*

La fonction commence par appeler Cout_Plus_Heuristic(), qui additionne le coût de la case qui est défini de manière aléatoire entre 1 et 5 et l'heuristique qui est la distance manhattan pour chaque case.
L'algorithme de recherche A * ajoute ensuite la cellule de départ à la liste frontier, qui représente l'ensemble des cellules que l'algorithme n'a pas encore entièrement explorées. L'algorithme entre alors dans une boucle qui continue jusqu'à ce que la liste frontier soit vide. À chaque itération, la cellule située à l'avant de la liste frontier est retirée et ajoutée à la liste explored, qui représente l'ensemble des cellules que l'algorithme a entièrement explorées.
Si la cellule courante est la cellule de fin, la boucle est terminée et l'algorithme se termine. Sinon, l'algorithme vérifie les cellules immédiatement au-dessus, en dessous, à gauche et à droite de la cellule courante pour voir si elles sont accessibles (c'est-à-dire qu'elles ne sont pas un mur et qu'elles ne sont pas déjà dans la liste explored). Si une cellule est accessible, elle est ajoutée à la liste frontier.
La liste frontier est alors triée par ordre croissant en fonction de la somme de la valeur coût de chaque cellule et de la valeur heuristique (heuristic_plus_cout). Si il y a des cellules avec la même valeur heuristic_plus_cout, la cellule avec la valeur heuristique la plus basse est priorisée.
Enfin, la première cellule de la liste frontier est ajoutée à la liste explored et la boucle continue. Ce processus se poursuit jusqu'à ce que la cellule de fin soit trouvée ou que la liste frontier devienne vide, indiquant que la cellule de fin n'est pas atteignable depuis la cellule de départ.

1.5.1 Complexité et nombres d'itérations

La complexité de l'algorithme A* dépend de la taille du labyrinthe et du nombre de murs (cellules non accessibles). Dans le pire des cas, où le labyrinthe est rempli de murs et où la cellule de départ et la cellule d'arrivée sont séparées par un mur, l'algorithme devra parcourir toutes les cellules du labyrinthe avant de déterminer que l'objectif n'est pas atteignable. Dans ce cas, la complexité de l'algorithme serait de l'ordre de O(n^2), où n est la taille du labyrinthe (31x31 dans mon cas).
Dans le meilleur des cas, où il y a un chemin direct entre la cellule de départ et la cellule d'arrivée sans aucun mur, l'algorithme trouvera le chemin le plus court rapidement et la complexité sera de l'ordre de O(n), où n est la longueur du chemin le plus court.

1.5.2 Coût en mémoire

Dans le pire des cas, où le labyrinthe est rempli de murs et où la cellule de départ et la cellule d'arrivée sont séparées par un mur, l'algorithme devra stocker toutes les cellules du labyrinthe dans la liste frontier avant de déterminer que l'objectif n'est pas atteignable. Dans ce cas, la complexité en mémoire de l'algorithme serait de l'ordre de O(n^2), où n est la taille du labyrinthe (31x31 dans mon cas).
Dans le meilleur des cas, où il y a un chemin direct entre la cellule de départ et la cellule d'arrivée sans aucun mur, l'algorithme trouvera le chemin le plus court rapidement et la complexité en mémoire sera de l'ordre de O(n), où n est la longueur du chemin le plus court.

1.5.3 Vitesse

Vitesse A*
Pour le temps du A*, j'ai fait une moyenne de 10 temps de mesures.
Moyenne_A* = 0,00485s

Conclusion

Suite à l'analyse des ces différents algorithmes, on détermine que l'algorithme A* est le plus propice à résoudre ce type de problème est d'algorithme en terme de vitesse, et de taille mémoire car les cases explorées sont moins importantes.
A* est suivi de Greedy Search, puis ensuite de Hill Climbing et DFS qui ferme la marche qui est le pire algorithme pour ce type de problème.

2) Algorithmes de jeux de stratégie : Le Yoté

But du jeu

Capturer tous les pions de l'adversaire.

Déroulement

Au début de la partie, tous les pions de chaque joueur sont en dehors du jeu.

À son tour de jeu on peut :

- poser un pion dans une case vide,
- ou effectuer un déplacement d'une case en ligne droite (pas en diagonale),
- ou faire une prise.

La prise se fait en sautant par-dessus un pion adjacent, comme aux dames. On ne saute qu'un seul pion à la fois, mais chaque prise permet de retirer un autre pion adverse choisi parmi ceux situés sur le plateau.

Fin du jeu

La partie se termine lorsqu'un joueur n'a plus de pion sur le plateau à partir du deuxième tour.

2.1) Fonctions pour l'IA

Pour ce projet, j'ai dû coder une IA qui respecte les règles du jeu et qui choisit toujours la meilleure option pour elle. Pour ce faire, j'ai dû développer plusieurs fonctions.

2.1.1) Fonction insérer une pièce de manière sécurisée

Cette fonction va permettre à l'IA d'insérer une pièce sur le plateau de façon à ne pas la perdre lors du prochain tour. Pour que cette pièce soit insérée, l'IA cherche une case où il n'y a pas d'ennemi autour.

2.1.2) Fonction pour prendre une pièce de manière sécurisée.

Cette fonction permet à l'IA d'attaquer son adversaire sans prendre le risque de perdre son jeton au tour suivant. Elle permet en plus de ca de prendre un jeton supplémentaire à l'ennemi après une prise. Elle vérifie que le joueur adversaire à un jeton sur le plateau. Si c'est le cas elle prend le jeton sur le plateau sinon une dans la réserve de l'adversaire.
S'il ne peut pas attaquer de manière sécurisée, il va insérer un autre jeton sur le plateau.

2.1.3)Fonction qui fait un déplacement pour ne pas être mangé au prochain tour.

Une fonction a été implémentée afin de permettre à l'IA de déplacer un jeton lorsque celui-ci est en danger d'être pris au prochain tour. Pour faire son déplacement, elle vérifie toutes les cases autour d'elle. Elle prend chaque case et regarde s'il y a des ennemis autour de cette case. Si c'est le cas, elle analyse la case suivante. Lorsqu'elle trouve une case sans danger, elle s'y rend.

2.1.4) Fonction attaquer sans se soucier d'être mangé au tour suivant.

Cette méthode permet à l'IA d'attaquer son adversaire lorsqu' elle a 100% de perdre la pièce au tour prochain. Car il se peut que l'IA se retrouve dans une situation où il est avantageux pour elle de prendre une pièce même si elle doit être prise au tour suivant.

2.1.5) Fonction déplacement sans vérification

Lorsque l'IA n'a plus de choix, elle ne peut que déplacer un jeton sans se soucier du fait qu'il sera mangé ou pas le tour prochain.

2.2)Fonction qui gère le tour de l'IA

Grâce aux fonctions développées ci-dessus, implémenter la Fonction qui gère le tour de l'IA était un jeu d'enfant.
Il suffit d'utiliser des if imbriqués afin de gérer les décisions de l'IA:

Voici comment sont imbriqués les if en pseudo-code:

```
Si l'IA a pas de jeton sur le plateau
        Alors ajouter jeton
Sinon Si IA peut attaquer sans prendre de risque
        Alors Attaquer
Sinon Si IA ne peut attaquer sans prendre de risque
        Alors ajouter jeton
Sinon Si un jeton est en danger
        Alors le déplacer pour pas se faire manger
Sinon Si aucun choix au dessus possible
        Attaquer en prenant des risques
Sinon Bouger jeton en prenant des risques.
```

Conclusion

Grâce aux méthodes codés et à l'imbrication des if, j'ai pu coder une IA qui gagne à 100% des coups car je n'ai jamais réussi à la battre.
On peut modifier le niveau de l'IA en modifiant l'ordre des if par exemple ce qui peut rendre l'IA plus forte ou moins forte.
Dans mon cas, l'IA va toujours jouer le coup le plus intéressant pour elle et désavantager ainsi le joueur.

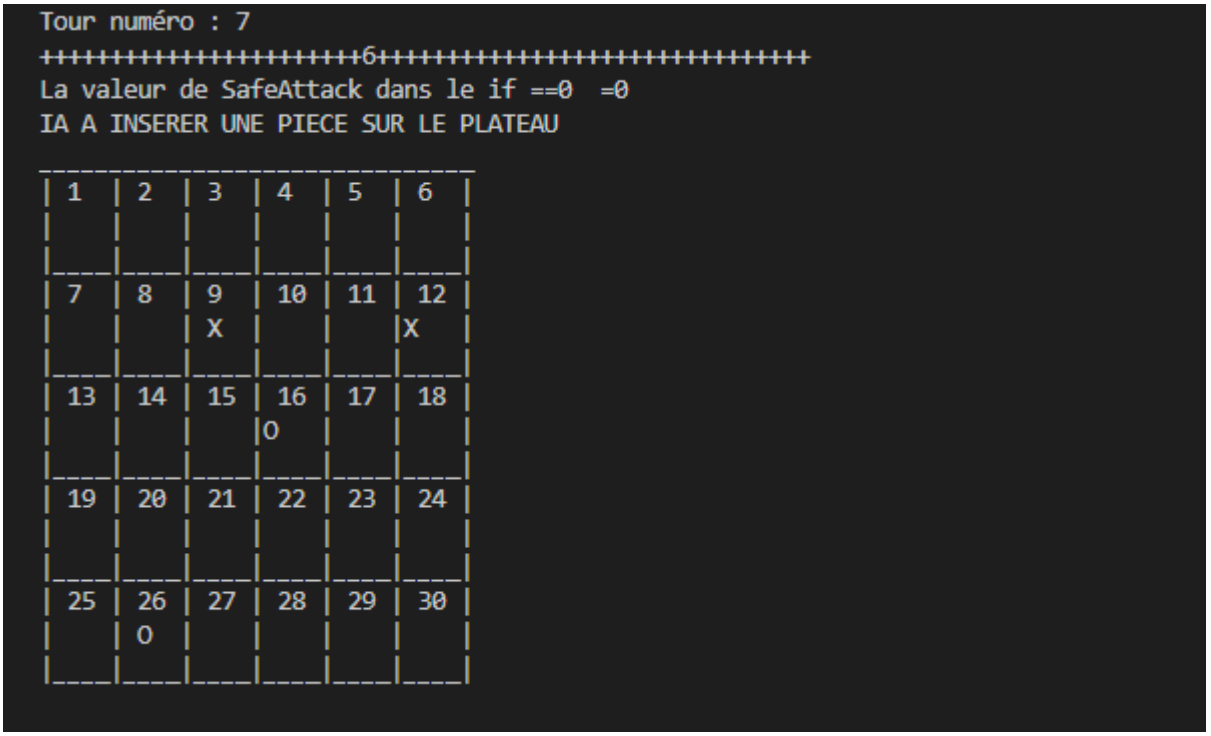


Figure : Jeu du Yoté sur console.

3) Programmation par contrainte : Gestion des chantiers

But de l'application : trouver un emploi du temps valide pour une entreprise de construction en prenant en compte les contraintes de disponibilité des matériels et des employés, et en respectant l'ordre chronologique des chantiers.

3.1) Structures de données

Structure Chantier : représente un chantier, avec un identifiant unique, une liste de matériels nécessaires, un nombre d'employés nécessaires et une date de fin.

Structure EmploiDuTemps : représente l'emploi du temps de l'entreprise, avec une liste de chantiers.

3.2) Fonctions principales

Fonction gererEmploiDuTemps : utilise l'algorithme de backtracking chronologique pour essayer chaque chantier restant dans l'ordre chronologique, en vérifiant s'il est valide pour l'emploi du temps courant et en mettant à jour l'emploi du temps et les ressources si c'est le cas. Si aucune solution valide n'a été trouvée avec les chantiers restants, elle annule les modifications faites à l'emploi du temps et aux ressources et retourne false. Si tous les chantiers ont été affectés à l'emploi du temps, elle retourne true pour indiquer qu'une solution valide a été trouvée.

Fonction estChantierValide : vérifie si un chantier est valide pour l'emploi du temps donné en prenant en compte les contraintes de disponibilité des matériels et des employés. Elle retourne true si le chantier est valide, false sinon.

Fonction mettreAJourEmploiDuTemps : met à jour l'emploi du temps et les ressources disponibles (matériels et employés) en fonction d'un chantier qui vient d'être ajouté à l'emploi du temps. Elle enlève les matériels et les employés utilisés pour le chantier du stock et de la disponibilité, et ajoute le chantier à l'emploi du temps.

3.3) Fonction main

Initialisation de l'emploi du temps et des ressources de l'entreprise (stock de matériels et nombre d'employés disponibles).

Initialisation de la liste des chantiers de l'entreprise.

Appel de la fonction gererEmploiDuTemps avec l'emploi du temps, la liste des chantiers, le stock de matériels et le nombre d'employés disponibles en paramètres.

Vérification de la solution trouvée par gererEmploiDuTemps et affichage de l'emploi du temps ou d'un message indiquant qu'aucun emploi du temps valide n'a été trouvé.

3.4) Autres fonctions

Fonction sort : utilisée dans la fonction gererEmploiDuTemps pour trier les chantiers restants par ordre chronologique. Elle prend en paramètres les itérateurs de début et de fin de la plage de chantiers à trier, ainsi qu'une fonction de comparaison qui permet de définir l'ordre de tri.

Fonction lambda : utilisée comme fonction de comparaison dans la fonction sort pour trier les chantiers par ordre chronologique. Elle retourne true si la date de fin du premier chantier est inférieure à celle du second, false sinon.

Conclusion

L'application permet de trouver un emploi du temps valide pour une entreprise de construction en prenant en compte les contraintes de disponibilité des matériels et des employés, et en respectant l'ordre chronologique des chantiers.

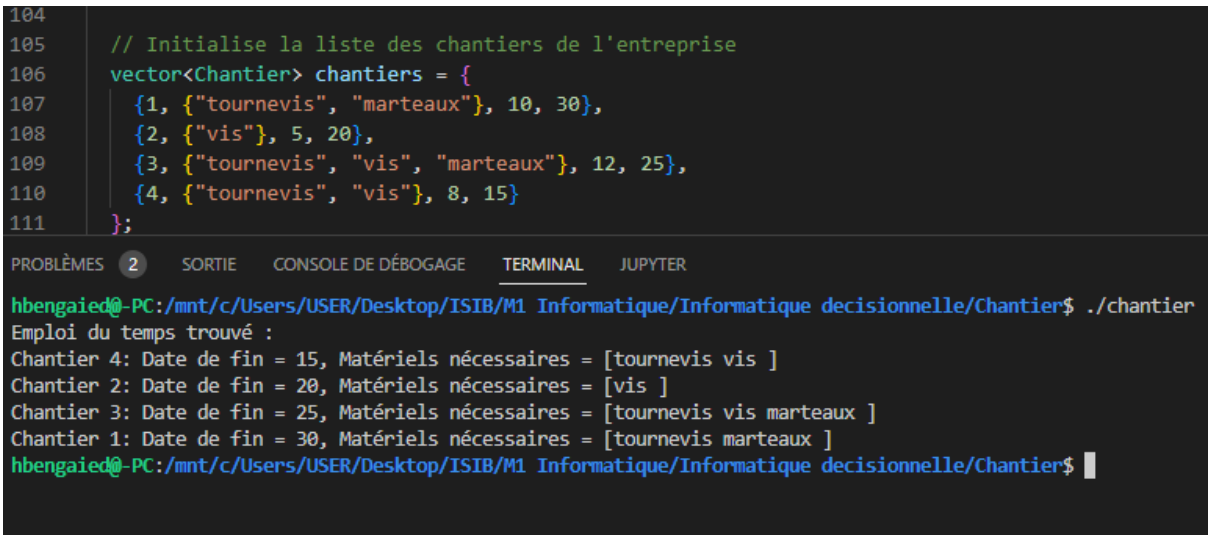


Figure 3: Exemple d'emploi du temps