Rapport de projet de RICHAUME Anthéa et LEFEVRE Sebastien

ORush

I. Choix techniques

Voici les structures de données utilisée pour l'élaboration de Orush :

Un bateau (type boat) est représenté par un record, avec 5 valeurs :
- son identifiant
- sa longueur
- son orientation
- les coordonnées de sa case la plus haute et la plus à gauche, x et y.

Ces coordonnées sont mutables pour y accéder plus facilement et afin de pouvoir directement modifier ces valeurs lors du déplacement du bateau via une copie du state.

L'état du port (type state) est représenté par une liste de boat.

Un déplacement (type move) est représenté par un tuple de char, le premier étant l'identifiant de bateau à déplacer et le deuxième le caractère '<' (pour reculer) ou'>' (pour avancer).

II. Contraintes techniques

- Contraintes sur le port :

La fonction initialisant le port est **input_state** , elle permet, à partir d'un fichier, de convertir les lignes en bateaux pour créer un state initial. Afin d'avoir des données cohérentes et respecter les contraintes de l'environnement, il faut gérer les différents cas d'erreurs lors de ces lectures.
C'est pour cela que nous avons décidé d'utiliser un mécanisme de **try with** :
- **Invalid_argument** : on lève cette exception lorsque les contraintes du port ne sont pas respectées, notamment lorsque le bateau sort du port avec une certaine position et une certaine orientation …
- **End_of_file** : afin d'arrêter la lecture du fichier et de renvoyer le state construit.

Lors d'un ajout de bateau, nous utilisons **grid_of_state** sur l'état courant afin de pouvoir simplifier les vérifications des conditions d'ajout notamment lorsqu'il y a chevauchement avec d'autres bateaux déjà présents sur certaines cases. Si il y a chevauchement, l'exception **Invalid_argument** sera levée et arrêtera, lors du **input_state**, la lecture du fichier.

- Contraintes sur les moves :

La fonction **check_solution** permet de vérifier si une solution est correcte (solution:string) sur un state initial donné et renvoie true si le bateau A arrive à l'état gagnant, false sinon. Elle applique chaque move de la solution en appelant **apply_move**. Cette fonction applique le move sur le state passé en paramètre. Afin de ne pas modifier le state appelant, on effectue le changement de position sur une copie de celui-ci, que l'on retournera.

Si le move n'est pas conforme (lors d'un chevauchement ou de sortie de port) l'exception **Cannot_move** est levée, check_solution retournera donc false. Afin de savoir si l'état final est gagnant, on vérifie si le bateau A est en position de sortie après tous les moves effectués. Si le bateau A n'était pas présent, check_solution retournera également false.

III. Recherche de solution

L'algorithme de recherche de solution consiste à parcourir en largeur les possibilités de déplacement en utilisant une file de nœuds via le module Queue (une file de type FIFO afin de pouvoir récupérer les nœuds visités encore non explorés) et en utilisant une table de hachage pour marquer les nœuds visités. Les nœuds sont des tuples composés d'un state courant et d'une chaîne de moves représentant l'ensemble des mouvements pour atteindre cet état.

solve_state :
On commence l'algorithme avec le state de départ récupéré via le input_state et une chaîne vide qui représentera l'ensemble des mouvements. On crée ensuite la variable visite, la table de hachage qui contiendra en clé les states représentés sous chaîne de caractères et en valeurs l'ensemble mouvements.

Tant que la file n'est pas vide, c'est-à-dire tant que l'on a pas trouvé la solution, on défile la tête de la file et on teste si le state courant est en position gagnante via la fonction **win** définie dans Moves.

Sinon, on récupère tous les moves possibles à partir du state courant avec la fonction **all_possible_moves** qui itère sur la liste des bateaux et appelle pour chaque bateau la fonction **move_possible** qui teste tous les déplacements possibles (avancer ou reculer) pour ce bateau dans un state donné. La fonction **can_move** renvoie true ou false si un déplacement est possible ou non.

On récupère ensuite la liste de tous les states possibles avec **all_reachable_states**. Cette fonction applique les moves possibles (calculés précédemment) avec **apply_move**.

La dernière étape de chaque appel récursif consiste à l'ajout de tous les nœuds n'ayant pas déjà été visités. Pour ce faire, il faut ajouter les states (sous forme de chaîne de caractère : **string_of_state**) non visités trouvés par all_reachable_states dans la table de hachage. Il faut également ajouter en fin de file ces nœuds correspondants. Ces nœuds sont composés de chaque state non visités et des moves qui leur sont associés (concaténation de l'ensemble des moves du nœud courant avec le move qui a permis d'arriver à ce state).

La fonction s'arrête lorsque le state courant est en position gagnante et retourne l'ensemble des mouvements permettant d'y arriver. La solution peut alors être retournée.

D'autres versions ont été faites. Nous avions fait une liste en référence à la place d'une table de hachage, mais cette implémentation était moins performante que la version finale. Il semblerait que les structures utilisées dans les tables soient bien plus performantes.
Afin d'optimiser au mieux l'ancienne version (sans file) nous ajoutions à cette liste de visite les states non visités en tête. Cela permettait d'éviter de reparcourir en entier cette liste. Dans l'ancienne implémentation on vérifiait si un state avait déjà été visité si tous ces bateaux concordent avec un state de la liste des visites. Ceci n'était pas du tout performant, c'est pour cela que nous avons trouvé une autre manière de comparer les states en les transformant en string.