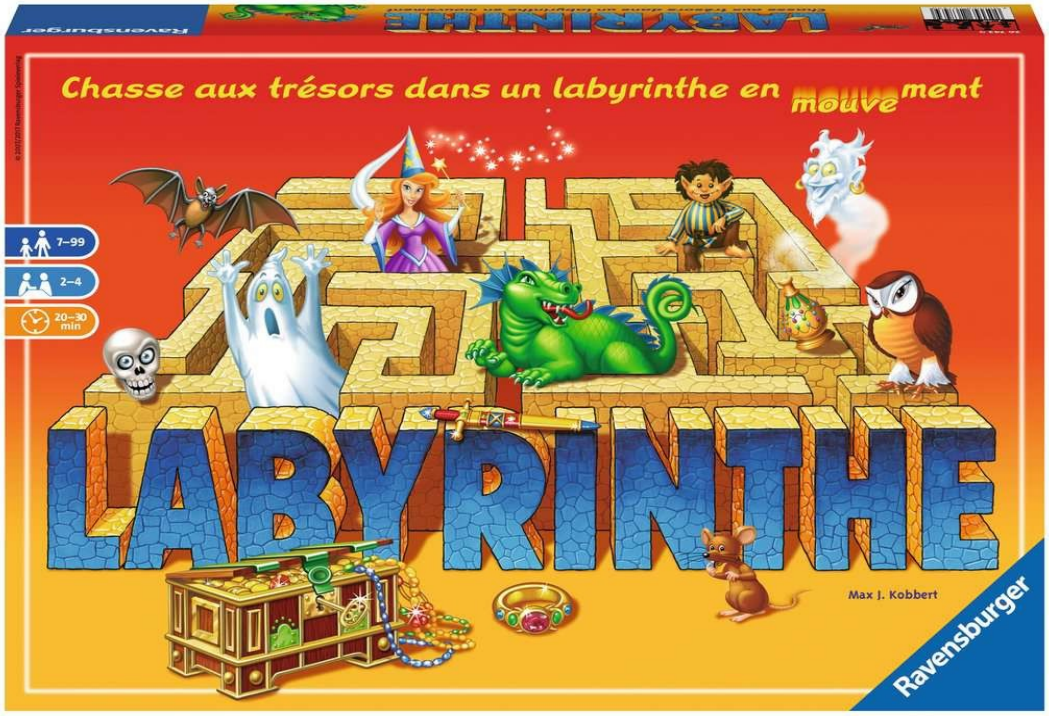
Rapport d’analyse

HERPOUX Mattéo

PLEUTIN Benjamin

Le jeu du Labyrinthe en C :



Sommaire

I. Introduction

Ce document est le rapport d'analyse d'un projet de programmation en C dont le but est de développer un programme permettant de jouer à un jeu de société: le Labyrinthe.

On devra pouvoir faire des parties de Labyrinthe à partir d'un programme en C, en suivant les règles du Labyrinthe classique. Ici, on développera néanmoins une IA, dans le but de pouvoir jouer avec 2, 3, ou 4 joueurs même si l'on est seul.e à jouer.

Ce rapport comprend une analyse exhaustive des besoins représentée par du texte, où on part du problème et on réflechis à des idées de solutions qui pourraient être apportées audits problèmes, mais aussi où on change le problème, parfois une simplification pour faire apparaître les solution plus clairement, et parfois une complexification pour faire intervenir des nouvelles manières de résoudre les problèmes.

Ensuite viens l'analyse descendante, où on se sert de l'analyse des besoins pour échafauder un plan de fonctions qu'on utilisera pour répondre aux différents besoins. On a le nom des fonctions, leurs entrées et sorties supposées, et les relations entre les différentes fonctions.

Logiquement, on écrit après la conception préliminaire, qui consiste simplement à récupérer dans l'analyse descendante le nom des fonctions, leurs entrées et leurs sorties et à les lister de façon à ce qu'on puisse les examiner plus lisiblement.

Enfin, nous avons la conception détaillée, qui, pour chaque fonction de la conception préliminaire, donne le pseudo-code associé. Ce travail facilitera grandement l'implémentation, car on sait presque exactement quel rôle devra jouer chaque fonction.

II. Spécifications

1. Analyse des besoins

Identifier les joueurs

- Qui sont-ils?
    - Donner la possibilité aux joueurs de choisir leur pseudo
    - Donner la possibilité aux joueurs de choisir leur couleur
        - Changer la couleur d'une écriture.
- Combien d'humains / Combien d'IA ?
    - Programmer une IA
        - Faire en sorte qu'elle puisse jouer comme un joueur humain, qu'elle ne soit ni stupide ni infaillible
            - Regarder si de là où elle est, l'IA peut atteindre son trésor
            - Elle ne bougera une case que si ça lui ouvre un chemin pour déplacer son pion d'au moins une case, et si ce n'est pas possible, chercher à embêter les autres joueurs
                - Regarder si, quand on bouge une ligne à proximité de l'IA, ça lui libère un chemin
                - Boucher le chemin, si c'est possible d'un autre joueur
                    - Regarder, si en bougeant une ligne, le chemin du joueur sera bouché
                - Un autre joueur a-t-il un chemin ouvert pour son trésor?(qui est connu on le rappelle)
- Assigner un pion à chaque joueur(pseudo, couleur...)
- Répartir équitablement et aléatoirement la liste des trésors entre chaque joueurs
    - Trier un tableau de façon aléatoire

Créer le plateau

- Placer aléatoirement les cases (chemin, trésor)
    - Définir tous les différents types de cases
        - Définir une case
        - Créer les trésors
- Permettre à 12 lignes uniquement de se déplacer

Jouer un tour

- Afficher le plateau
    - Afficher le plateau avec ses cases fixes
    - Afficher les flèches autour du plateau avec leur numéro
    - Afficher une ligne de cases mouvantes
        - Afficher une case

                - Afficher un trésor sur une case
                - Afficher une case de sorte à ce qu'on reconnaisse quel type de case c'est, où sont les murs
                - Afficher un pion sur une case
        - Afficher la case temporaire à coté du plateau
        - Si on peut, décaler la ligne d'un cran
            - Regarder si la ligne qu'on veut décaler est décalable
            - Regarder si la ligne qu'on veut décaler à été décalée juste avant dans le sens inverse
                - Garder en mémoire la dernière ligne décalée et dans quel sens elle l'a été
            - Insérer la case temporaire dans le plateau pour remplir la ligne
            - La case à la fin de la ligne devient la nouvelle case temporaire

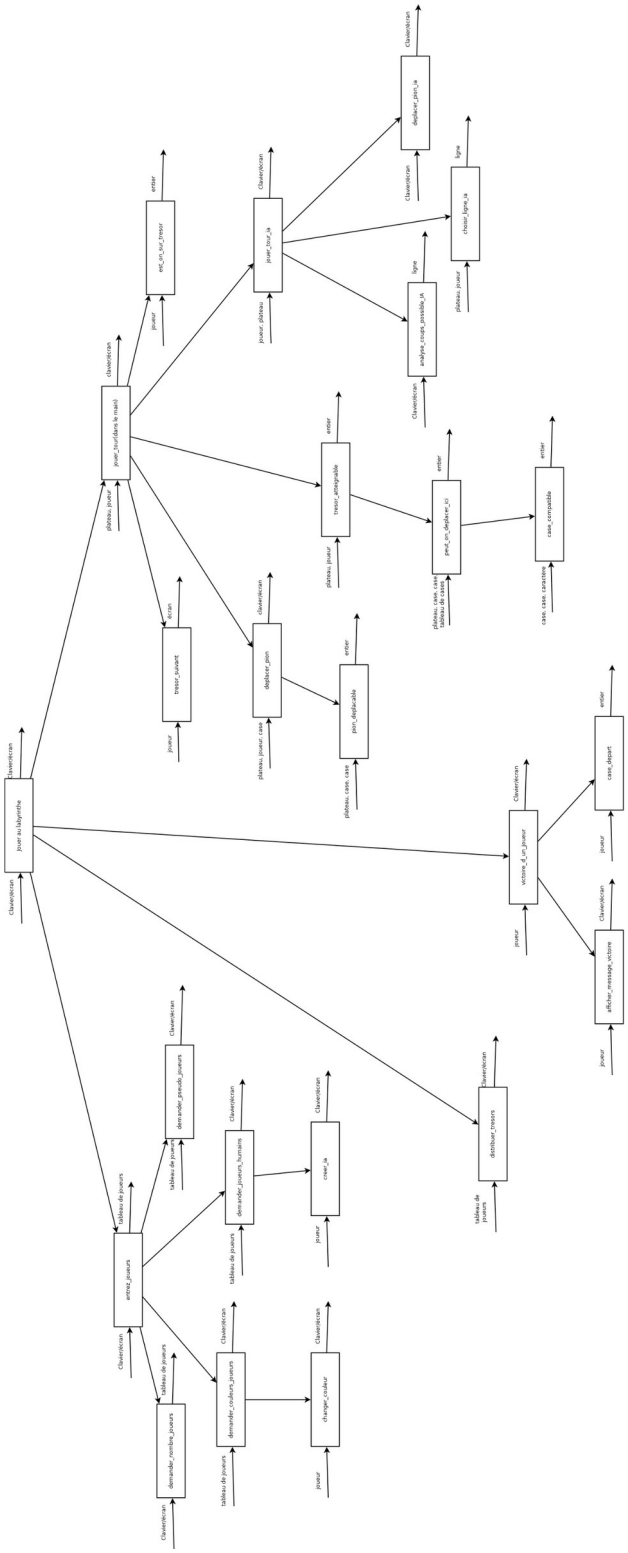
            - Si un ou plusieurs pions sont sur la case qui sort, alors ils sont placés sur la case qui vient de rentrer
    - Permettre ou non au pion de se déplacer
        - Le pion peut-il se déplacer à l'endroit voulu par le joueur?
            - Existe-il un chemin?
                - Lier toutes les cases compatibles pour créer un chemin
    - Afficher la manipulation effectuée par le joueur({nom du joueur} a déplacé son pion en...)
    - Regarder si on est sur une case trésor
    - Passer au trésor suivant quand on atteint
        - Arrivé au dernier trésor, le nouvel objectif est le point de départ(un des 4 coins de l'écran)

Stopper la partie lorsqu'un joueur a gagné

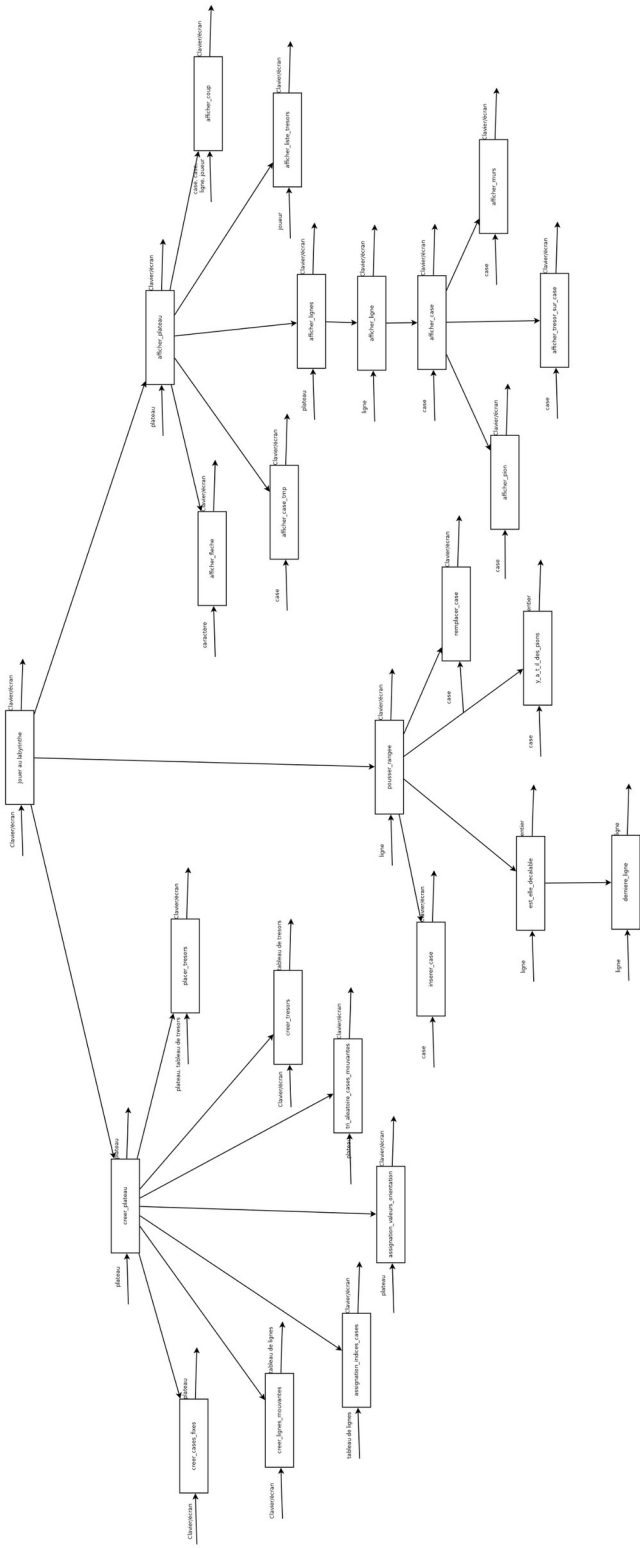
- Regarder si un joueur a gagné à la fin du tour
    - Regarder si la pile de trésors d'un joueur est vide et si il est placé sur sa case de départ
- Afficher un message en conséquence
    - Regarder quel joueur a gagné

2. Analyse descendante

Première partie, qui contient les fonctions conçernant les joueurs, et les fonctions qui seront utilisées pour jouer un tour. On peux supposer, même si elle n’est pas dans la conception préliminaire ou détaillée, que jouer_tour pourras être aussi une fonction plus tard.



Seconde partie, qui regroupe les fonctions conçernant l’IA, le plateau en général (sa création, sa gestion, son fonctionnement) et les fonctions grâce auxquelles on pourras utiliser la particularité du jeu du Labyrinthe : pousser les rangées.



III. Conception préliminaire

1. Fonctions

- tableau de joueurs entrer_joueurs()
- demander_joueurs_humains(tableau de joueurs)
- tableau de joueurs demander_nombre_joueurs()
- demander_couleur_joueurs(tableau de joueurs)
- demander_pseudo_joueur(tableau de joueurs)
- changer_couleur(joueur)
- joueur creer_ia(joueur)
- victoire_dun_joueur(joueur)
- afficher_message_victoire(joueur)
- entier case_depart(joueur)
- jouer_tour_ia(joueur, plateau)
- ligne analyse_coups_possibles_IA() (En cours de conception)
- ligne choisir_ligne_ia(plateau, joueur)
- deplacer_pion_ia()
- plateau creer_plateau()
- tableau de trésors creer_tresors()
- placer_tresors(plateau, tableau de trésors)
- tri_aleatoire_cases_mouvantes(plateau)
- assignation_indices_cases(tableau de lignes)
- tableau de ligne creer_lignes_mouvantes()
- plateau creer_casesfixes()
- assigner_valeurs_orientations(plateau)
- entier tresor_atteignable(plateau, joueur)
- entier est_on_sur_tresor(joueur)
- tresor_suivant(joueur)
- deplacer_pion(joueur, plateau, case)
- entier pion_deplacable(plateau, case, case)
- entier peut_on_deplacer_ici(plateau, case, case, tableau de cases)
- entier case_compatible(case, case, caractère)
- afficher_plateau(plateau)
- afficher_fleche(caractère)
- afficher_case_tmp(case)
- afficher_lignes(plateau)
- afficher_ligne(ligne)
- afficher_case(case)
- afficher_liste_tresors(joueur)

- afficher_tresor_sur_case(case)
- afficher_murs(case)
- afficher_pion(case)
- afficher_coup(case, case, ligne, joueur)
- entier est_elle_decalable(ligne)
- pousser_rangee(ligne)
- ligne derniere_ligne(ligne)
- inserer_case(case)
- remplacer_case(case)
- entier yatil_des_pions(case)

2. Structures :

- plateau
- trésor
- joueur
- ligne
- case

IV. Conception détaillée

1. Fonctions

- tableau de joueurs entrer_joueurs()
    - on crée un tableau de joueur et on l'initialise avec demander_nombre_joueurs()
    - demander_joueurs_humains(le tableau)
    - demander_couleur_joueurs(le tableau)
    - demander_pseudo_joueur(le tableau)
- demander_joueurs_humains(tableau de joueurs)
    - afficher "Combien d'IA?"
    - si le nombre d'IA est supérieur a demander_nombre_joueurs()-1
        - afficher "Nombre d'IA pas cohérent avec le nombre de joueurs"
    - sinon
        - pour chaque ia
            - creer_ia(le joueur qui va devenir une ia)
- tableau de joueurs demander_nombre_joueurs()
    - afficher "Combien de joueurs sur cette partie?"(nbr)
    - si demander_nombre_joueurs() égal à 2, 3 ou 4
        - créer des struct joueurs en conséquence
    - sinon
        - afficher "Il n'y a pas un nombre adéquat de joueur"

- demander_couleur_joueurs(tableau de joueurs)
    - pour chaque joueur
        - afficher "Quelle est la couleur du joueur j.i?"(afficher les couleurs 1:Rouge, 2:Jaune...)
        - le paramètre couleur du joueur prends l'entrée clavier
- demander_pseudo_joueur(tableau de joueurs)
    - pour chaque joueur
        - afficher "Quel est le pseudo du joueur j.i?"
        - le paramètre nom du joueur prends l'entrée clavier
- changer_couleur(joueur)
    - voir https://stackoverflow.com/questions/3219393/stdlib-and-colored-output-in-c
- joueur creer_ia(joueur)
    - le paramètre ia du joueur passe a 1(fonction courte mais plus lisible dans le code que juste "j.ia=1")
- victoire_dun_joueur(joueur)
    - si case_depart(j) égal à 1(si le joueur a trouvé tous ses trésors et qu'il est retourné à son point de départ)
        - afficher_message_victoire(le joueur)
        - on sort de la boucle du main avec les jouer_tour
- afficher_message_victoire(joueur)
    - afficher "Le joueur j.nom a remporté la partie!"
- entier case_depart(joueur)
    - si le joueur a trouvé tous ses trésors et qu'il est retourné à son point de départ
        - retourne 1
    - sinon
        - retourne 0
- jouer_tour_ia(joueur, plateau)
    - on retourne les coups possibles avec analyse_coups_possibles_IA()
    - pour pousse la rangée choisie par l'ia avec les 2 fonctions pousser_rangee(choisir_ligne_ia(le plateau, le joueur))
    - l'ia deplace le pion avec deplacer_pion_ia()
    - on afficher un bilan de ce que l'ia viens de faire avec afficher_coup()
- tableau de lignes analyse_coups_possibles_IA(plateau, joueur)
    - On fait un tableau regroupant tous les coups possibles
    - (Faut trouver un moyen d'associer chaque indice du tableau à un coup, nouvelle struct?)
    - si le trésor est atteint dans une des 48 premières possibilités alors on joue ce coup
    - sinon on cherche les combinaisons de 2 coups qui permettraint d'atteindre le tresor
    - sinon on pousse une rangée au hasard qui libère un chemin et on déplace le pion sur ce chemin

- analyse_bloquage_possible_IA(plateau, joueur, autres joueurs(tableau de joueurs), analyse_coups_possibles_IA(le plateau, le joueur))
    - Même principe que analyse_coups_possibles_IA pour les 48 prochains coups possibles des autres joueurs, afin de savoir s'ils pourront accéder à leur trésor au prochain coup.
    - On va ensuite analyser chaque coup retenu par analyse_coups_possibles_IA et incrémenter chaque case dans le cas où le coup d'un joueur passe de 1 à 0. Le coup ayant été le plus de fois incrémentée est celle qui bloquera le plus de coups possibles chez les autres joueurs tout en permettant à l'IA d'atteindre son trésor

- ligne choisir_ligne_ia(plateau, joueur)
    - pour les 12 lignes:
        - si, quand on bouge la ligne, le prochain tresor est accessible
            - on retourne cette ligne
- deplacer_pion_ia()
    - si le prochain tresor est atteignable
        - deplacer_pion() sur le tresor
        - on passe au trésor suivant avec tresor_suivant()
    - sinon on reste ou on est

- plateau creer_plateau()
    - on crée un plateau égal à creer_cases_fixes()
    - on cree un tableau de 12 lignes égal à creer_lignes_mouvantes()
    - on assigne des indice hauteur et largeur à chaque cases pour les situer dans le plateau et on leur donne un numéro unique avec assignation_indices_cases(le tableau)
    - on place aléatoirement les cases mouvantes sur le plateau tri_aleatoire_cases_mouvantes(p)
    - on crée un tableau de trésors égal à creer_tresors()
    - on disperse aléatoirement les trésors sur le plateu avec placer_tresors(p, liste_tresors)

- tableau de trésors creer_tresors()
    - on crée un tableau de 24 trésors
    - pour chaque trésor:
        - on lui assigne son numéro
        - on lui assigne son nom (on prendra les noms dans un fichier)
    - on retourne le tableau
- distribuer_tresors(tableau de joueurs, tableau de trésors)
    - on distribue aléatoirement et équitablement les trésors entre les joueurs(paramètre tresors_restants)

- placer_tresors(plateau, le tableau de trésors)
    - creer un tableau de int de taille 34, le remplir avec 0,1,2,3,4...33
    - le trier aléatoirement
    - pour les 24 tresors:
        - si la case dont le numéro correspond à l'indice n'est pas un coin
            - on associe a la case qui a le numéro correspondant le tresor de la liste
        - sinon
            - on associe le tresor a une case dont le numéro est supérieur a 24(qui n'est pas un coin)

- tri_aleatoire_cases_mouvantes(plateau)
    - creer un tableau de int de taille 34, le remplir avec 0,1,2,3,4...33
    - trier aléatoirement ce tableau
    - pour les 12 cases de types a:
        - si le num de la case est 0(la case tmp), le paramètre type de la case_tmp est ax,x étant une orientation aléatoirement choisie entre les 4, sinon:
            - pour la case dont le numéro correspond a l'indice dans le tableau d'entier: son type devient ax, x étant une orientation aléatoirement choisie entre les 4
    - pour les 6 cases de type c:
        - si le num de la case est 0(la case tmp), le paramètre type de la case_tmp est cx,x étant une orientation aléatoirement choisie entre les 4, sinon:
            - pour la case dont le numéro correspond a l'indice dans le tableau d'entier: son type devient cx, x étant une orientation aléatoirement choisie entre les 4
    - pour les case de type b:
        - si le num de la case est 0(la case tmp), le paramètre type de la case_tmp est bx,x étant une orientation aléatoirement choisie entre les 4, sinon:
            - pour la case dont le numéro correspond a l'indice dans le tableau d'entier: son type devient bx, x étant une orientation aléatoirement choisie entre les 4

- assignation_indices_cases(tableau de lignes)
    - en s'aidant de la position des lignes par rapport au plateau on et de maths, on parvient à assigner des indices hauteur et largeur a chaque case de telle sorte que la première case aie comme indices 0,0 et la dernière(en bas a droite) 7,7.
    - calculs non détaillés ici car conception détaillée

- tableau de lignes creer_lignes_mouvantes()
    - on crée un tableau de 12 lignes
    - pour chaque ligne
        - on lui assigne son numéro(de 1 a 12)(paramètre numero)
        - on créé un tableau de 8 cases et on l'assigne a la ligne
        - pour chaque case de la ligne
            - on assigne au paramètre numligne de la case le numéro de la ligne
    - on renvoie le tableau

- plateau creer_casesfixes()
    - on crée un plateau
    - on s'aide des indices pour creer les cases fixes du plateau en suivant le modèle(on leur assigne le bon type et la bonne place)
    - retourne le plateau
- void assigner_valeurs_orientations(p)
    - pour chaque case du plateau:
        - on assigne la position des murs en fonction du type de case, pour chaque type de case possible(12 types)
- entier tresor_atteignable(plateau, joueur)
    - si peut_on_deplacer_ici(le plateau, la case sur laquelle est le joueur, la case sur laquelle est le trésor que doit atteindre le joueur) est égal à 1:
        - retourne 1
    - sinon
        - retourne 0

- entier est_on_sur_tresor(joueur)
    - si le joueur est sur le trésor qu'il doit atteindre
        - retourne 1
    - sinon
        - retourne 0

- tresor_suivant(joueur)
    - on décale d'un cran vers la gauche le tableau de trésors, donc le trésor qu'on vient d'atteindre, qui était premier, disparaît, et le deuxième devient le premier, etc
    - le dernier tresor qui est ducoup en double devient la fin du tableau

- deplacer_pion(joueur, plateau, case)
    - si pion_deplacable(le plateau, la case sur laquelle est le joueur, la case passée en paramètre(qui est la case où l'on veut aller)) est égal à 1
        - le paramètre case du joueur devient la case où il devait aller
    - sinon
        - afficher "Le pion ne peux pas être déplacé ici"

- entier pion_deplacable(plateau, case, case)
    - si le pion n'est pas entre 4 murs(utiliser case_compatible) ET si peut_on_deplacer_ici(p, c1, c2) égal à 1
        - retourne 1
    - sinon
        - retourne 0

- entier peut_on_deplacer_ici(plateau, la case où on est, la case où on veut aller, un tableau a taille variable de cases qui contient toutes les cases où il est possible d'aller a partir de la où on est)
    - (cette ligne est une boucle)si on est à la première itération, pour les 4 orientation, sinon, pour seulement 3(pas celle d'ou l'on vient):
        - si case_compatible(le plateau, la case ou on est, (la case a coté de c1(suivant l'orientation)), le tableau de cases) égal à 1:
            - récursion: peut_on_deplacer_ici(le plateau, la case a coté, c2, tabcase)
    - si c2 présente dans le tableau de cases
        - retourne 1
    - sinon
        - retourne 0
- entier case_compatible(première case, deuxième case, l'orientation(où est la case 1 par rapport a la 2))
    - si il y a un chemin sur la première case sur l'orientation demandée ET pareil pour la deuxième:
        - retourne 1(ça veut dire qu'on peut passer entre les 2 cases)
    - sinon
        - retourne 0
- afficher_plateau(plateau)
    - on affiche le plateau dans son état actuel: la case temporaire(afficher_case_tmp), les flèches pour les lignes(afficher_fleche), les 49 cases et ce qu'il y a dessus(murs, pions, trésors)
- afficher_fleche(emplacement de la flèche(par où elle pointe))
    - on affiche une flèche en fonction de l'emplacement
- afficher_case_tmp(case)
    - afficher "Case pour pousser:"
    - afficher_case(la case)
- afficher_lignes(plateau)
    - pour les 7 premières lignes * afficher_ligne(la ligne)
- void afficher_ligne(ligne)
    - afficher_fleche(l'orientation selon le numéro de la ligne)
    - pour chaque case visible de la ligne
        - afficher_case(la case])
    - afficher_fleche(l'orientation inverse)
- afficher_case(case)
    - afficher_murs(la case)
    - afficher_pion(la case)
    - afficher_tresor_sur_case(la case)

- afficher_liste_tresors(joueur)
    - pour tous les trésors restants
        - afficher "numéro du trésor" et "nom du trésor"
- afficher_tresor_sur_case(case)
    - afficher le numéro du trésor qu'il y a sur la case si il y en a un
- afficher_murs(case)
    - afficher les murs en fonction du type de c avec des "|" et "-"
- afficher_pion(case)
    - changer la couleur du pion en fonction du joueur
    - afficher "I" de la bonne couleur
- afficher_coup(case où on était, case où on est allé, ligne qu'on a poussé, joueur)
    - afficher "Le joueur x a poussé la ligne y et a déplacé son pion de la case c1 a la case c2"
- entier est_elle_decalable(ligne)
    - Si la ligne n'est pas l'inverse de celle d'on vient de décaler au tour d'avant(ligne_tmp)
        - on retourne 1
    - Sinon on retourne 0
- void pousser_rangee(ligne l)
    - Si est_elle_decalable(l)==1
        - Pour chaque case
            - on décale(il y a 8 cases sur une ligne mais la dernière le fait pas partie du plateau) de la même manière que les trésors
        - la première case(qui est égale a la seconde) prends la valeur de la case tmp
        - la 8eme case devient la case tmp
- ligne derniere_ligne(ligne)
    - la ligne_tmp prends la valeur de la ligne en paramètre(paramètre numéro)
- inserer_case(case)
    - la case prend la valeur de case_tmp
- remplacer_case(case)
    - case_tmp prend la valeur de la case
- entier yatil_des_pions(case)
    - Si il y a 0 joueurs sur la case:
        - on renvoit 0
    - Sinon:
        - on renvoit 1

2. Structures

- plateau
  - les 12 lignes
  - la case temporaire
- joueur
  - le nom
  - la couleur
  - si c'est une ia(un int)
  - la liste des tresors qu'il lui reste a atteindre
- tresor
  - le nom
  - son numero
  - la case où il est
- ligne
  - le numéro de la ligne
  - le tableau de cases de la ligne
- case
  - le trésor sur la case(rien si il n'y en a pas)
  - le types de case et son orientation(chaine(an, be...))
  - si il y a des murs au 4 orientations(4 entiers(booléens) h, b, d, g)
  - le pion sur la case (PERSONNE si il n'y en a pas)
  - numero_ligne le numéro de la ligne auquelle la case appartient (tableau de 3 int, -1, -1, -1 si case non mouvante)
  - numero le numéro de case/34
  - les indices de la case dans le plateau

V. Conclusion

Maintenant que la conception détaillée est finie, on va pouvoir se lancer dans la prochaîne étape du cycle en V, ce qui sera probablement la plus longue étape du projet : l'implémentation.

Cependant, après avoir fini l'analyse descendante et commencé la conception détaillée, nous avons été amennés à créer de nouvelles fonctions, en supprimer certaines, et en modifier d'autres, et donc à « remonter » et modifier l'analyse descendante.

Cela s'explique par le fait que plus on descend profondément dans le projet, mieux on entrevoit les problématiques de près, et plus on est apte à les résoudre efficacement. Il n'est donc pas exclu de penser qu'en commençant l'implémentation, nous devrons changer des choses dans nos plans, et ne pas faire exactement notre conception détaillée, c'est même fort probable.