Rapport de projet

Ce projet consiste à créer un labyrinthe bi-dimensionnels et dirigés avec le langage OCAML. L'archive contient deux programmes ainsi qu'un exécutable pour chaque.

Labyrinthe

Structures de données

type coord → type produit, couple d'entiers
type salle → quadruplet de booléens
type chemin → liste de coordonnées
type 'a quadtree → structure arborescente dont les feuilles sont de type 'a et chaque nœud admet 4 sous – arbres (ne, no, se, so)
type labyrinthe → type produit, représenté par un côté et un salle - quadtree

Guide d'utilisation

Le joueur est représenté par le personnage en bleu. Le joueur se déplace en utilisant les touches 'z' pour aller au nord, 'd' pour aller vers l'est, 's' au sud et 'q' pour l'ouest. Le joueur dispose aussi des touches 'i' pour activer un indice qui montre les premières directions vers la sortie et 'a' s'il veut voir le chemin complet. Si le joueur réussit à trouver la sortie, son personnage devient vert et il peut utiliser 'e' pour fermer la fenêtre. S'il n'y a pas de passage possible vers la sortie, il peut toujours se déplacer mais il devient gris. Pour fermer la fenêtre à tout moment il peut utiliser la touche 'p'.

Explications

Nous commençons tout d'abord par dessiner le labyrinthe à l'aide d'un salle-quadtree que nous créons puis nous le parcourons pour dessiner chaque salle. Ces salles sont générées aléatoirement. En même temps, nous stockons dans la variable *labin* le labyrinthe créé et dans *liste_lab* nous stockons la liste des couples correspondant aux coordonnées de chaque salle ainsi que cette salle en question. Les coordonnées correspondent au point en bas à gauche de la salle. Nous utilisons les fonctions suivantes :
- *pow_of_2* : teste si un entier est une puissance de 2, renvoie vrai ou faux.
- *g_Room* : créer une salle aléatoire.
- *g_Tree* : créer un salle-quadtree par rapport au côté du labyrinthe.
- *count* : compte le nombre de feuille d'un 'a-quadtree.
- *draw_room* : dessine une salle avec les passages possibles en vert et les passages non-possibles en rouge, aux coordonnées données.
- *draw_laby* : dessine le labyrinthe à partir d'un salle-quadtree est des coordonnées.

Ensuite, nous calculons le chemin vers la sortie. Pour cela, nous allons mémoriser les salles déjà parcourues dans (coord * bool) - quadtree, avec les coordonnées étant ceux de la position du personnage, c'est-à-dire le milieu de chaque salle. Les fonctions sont :
- *g_bool* : génère un bool-quadtree initialisé à 'false' de la taille du labyrinthe.

- *g_coor* : génère à partir d'un bool-quadtree, un (coord * bool) – quadtree. Chaque feuille correspond à une salle du labyrinthe.

Par la suite, nous aurons besoin de parcourir la liste stockée dans *liste_lab* pour savoir comment est structuré la salle. Par conséquent, nous avons besoin d'une fonction qui convertit les coordonnées du personnage en coordonnées d'une salle. Cette fonction est *test*.
Du coup, la fonction *cherche*, comme indique son nom, cherche dans *liste_lab* la salle qui correspond à la coordonnée donnée.
Maintenant que l'on peut récupérer la salle avec les coordonnées du personnage, nous avons besoin de mettre à jour l'arbre. La fonction *update* prend les coordonnées ainsi que l'arbre et renvoie un arbre avec la feuille concernée changé à 'true'.
La fonction *visited* renvoie vrai si la feuille a été visitée, c'est-à-dire qu'elle est à 'true', et faux sinon.
Avant de passer à la fonction qui calcule le chemin réussi, nous avons la fonction *goal* qui teste si le personnage se situe à une sortie ou non, elle renvoie donc un booléen et la fonction *neighbours* qui renvoie une liste de coordonnées correspondant aux salles voisines ayant un passage vert.

À présent, nous avons la fonction *path*, correspondant à la fonction *cheminReussiDepuis* dans le sujet. Nous regardons les salles voisines du personnage et nous continuons à avancer jusqu'à trouver la sortie. Si on ne trouve pas, nous sommes donc bloqués, nous ajoutons cette salle à celles qui ont été visité et nous revenons à la salle précédente. Si à la fin, il n'y a pas de chemin réussi, la fonction renvoie une liste vide, sinon elle renvoie la liste de coordonnées vers cette sortie.

Nous avons maintenant besoin de représenter tout ça dans notre labyrinthe. Pour cela nous avons *perso* qui dessine un personnage, *delete* qui le supprime, *draw_path* qui prend une liste de coordonnées et qui trace le chemin vers la sortie et *draw_indice* qui fait comme *draw_path* mais seulement pour les 4 premières coordonnées, ce qui montre un indice pour le joueur et finalement, *clean_list* qui prend une liste et qui ne renvoie rien.

La fonction *bouton* renvoie un couple des nouvelles coordonnées calculé pour le personnage ou trace le chemin réussi calculé selon la touche que le joueur active.

Notre dernière fonction est *move*. C'est elle qui appelle la fonction qui dessine le personnage avec la couleur qui correspond. Pour exemple, si le personnage est bloqué, il devient gris, s'il a gagné il devient vert. Cette fonction est bien sur infinie.

Tout à la fin de notre programme, nous avons quelques variables qui permettent une meilleure visibilité, puis nous avons l'appelle à la fonction *move*.

Complexité

pow_of_2 : O (log n) , n → l'entier à tester.
g_Tree : Θ (log c) , c → côté du labyrinthe (pour nous le quadtree est équilibré)
count : Θ (log c) , c → côté du labyrinthe
*draw_room :*O (1)
draw_laby : Θ (log c) , c → côté du labyrinthe
g_bool : Θ (log c) , c → côté du labyrinthe
g_coor : Θ (log c) , c → côté du labyrinthe
test : O (1)
cherche : O (n), n→ longueur de la liste
update : O (n log c) , c → côté du labyrinthe et n → nombre de salles (on teste chaque salle et on parcours le quadtree en profondeur)
visited : O (n log c) , c → côté du labyrinthe et n → nombre de salles (on teste chaque salle et on parcourt le quadtree en profondeur)
goal : O (n), n → longueur de la liste (à cause de *cherche*).
neighbours : O (1) → meilleur des cas, O (n^2) → en moyenne et pire des cas (soit 0 comparaison soit 1 + 2 + 3 + …. + n = $\sum_{k=0}^{n} k = \frac{n(n+1)}{2} \approx O(n^2)$ n = 4 dans notre code)
suivre : O (1)
path : O (n) + O (n log c) + O (n^2) = O (n^2) (parcours de la liste avec *cherche* + teste avec *visited* et mis à jour avec *update* + *neighbours*)
*perso :*O (1)
grenouille : O (1)
delete : O (1)
draw_path : O (n), n→ longueur de la liste
draw_indice : O (n), n→ longueur du chemin d'indice (4)
clean_list : O (n), n→ longueur de la liste
random : O (1) → meilleur des cas, O (n^2) → en moyenne et pire des cas (soit 0 comparaison soit 1 + 2 + 3 + …. + n = $\sum_{k=0}^{n} k = \frac{n(n+1)}{2} \approx O(n^2)$ n = 4 dans notre code)
*coor_monstre :*O (n^2) (cela dépend de *path*)
bouton : - si la touche choisie est 'p' ou toutes autres touches non définies → O (1)
- pour les boutons 'z', 's', 'q', 'd'→ O (n) + O (n) + Θ (log c) = O (n) (teste pour chaque bouton + complexité du *clean_list* + complexité du *draw_laby*)
- pour les touches 'a', 'i' → O (n) + O (n^2) + O (n) = O (n^2) (test pour chaque bouton + calcul de chemin + dessin de chemin)
move : O (n^2) (cela dépend de *path*)

Difficultés rencontrées

Notre plus gros problème a été le calcul du chemin réussi. On a compris que l'utilisation d'une fonction qui trouve les voisins possibles d'une salle donnée est essentiel pour le calcul de chemin. Et donc nous avons trouvé l'algo.

Jeu

Création d'un jeu

Nous avons essayé de créer un jeu avec l'implantation d'un second personnage qui représente une créature qui a pour but d'attraper le joueur.

Explications

Ce dernier utilise la fonction *path* pour calculer le chemin vers le personnage. Nous utilisons la fonction *suivre* qui teste si la créature a réussi à attraper le joueur.
La fonction *coor_monstre* renvoie les nouvelles coordonnées de la créature en appelant *path.* S'il n'a pas de passage possible, nous utilisons la fonction *random* qui renvoie aléatoirement une coordonnée d'une salle voisine avec passage possible.

Maintenant, nous ajoutons ce second personnage dans la fonction *move*. Nous avons créé une fonction *grenouille* qui dessine ce second personnage.
Par contre, ce programme contient encore quelques problèmes que nous n'avons pas réussi à rectifier à temps. Nous voulions avoir un exécutable qui fonctionne correctement mais quand même montrer notre tentative à créer un jeu.

Pour exemple, parfois quand le personnage bouge, la grenouille disparaît puis réapparaît plus tard. Parfois le personnage devient gris alors qu'il n'est pas bloqué. Et aussi, en ajoutant des yeux aux personnages, les yeux changeaient de place quand le personnage se déplaçait. (donc nous avons décidé de dessiner les yeux correctement seulement pour un labyrinthe de côté 8).