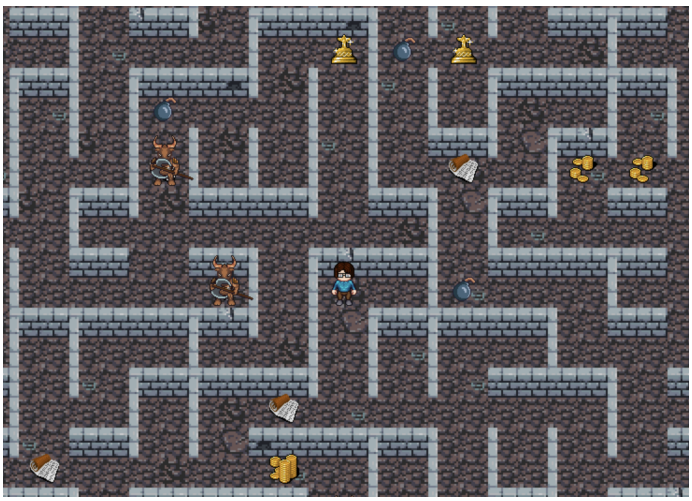
université de BORDEAUX	**Année 2024/2025** - L2 Maths-Info/CMI Algorithmique des structures de données (4TTV318U) Projet d'algorithmique et programmation	**Collège Sciences et Technologies**

Labyrinthes : génération, jeu et résolution



1 Modalités du projet

L'objectif du projet est d'implémenter un programme qui permet de gérer des labyrinthes. Il est à réaliser en binôme. Les dates limites sont précisées dans ce document.

Une **détection automatisée de plagiat** sera effectuée. En cas de plagiat (sur le web, entre groupes ou par IA comme chatGPT), les membres des groupes impliqués seront envoyés devant la commission disciplinaire de l'université.

Plusieurs rendus sont demandés sur moodle ainsi que l'URL d'un dépôt gitlab CREMI privé sur lequel vous déposerez les commits du projet, et auquel vous donnerez accès aux enseignants avec le rôle `Maintainer`. Les dates limites et les modalités de dépôt sont précisées dans ce document.

Le projet comporte **trois parties distinctes** et largement **indépendantes** qui seront décrites plus en détail dans la suite de ce document :

1. Partie I : Implémentation de plusieurs algorithmes servant à générer des labyrinthes.
2. Partie II : Implémentation du jeu à l'intérieur d'un labyrinthe.
3. Partie III : Implémentation d'algorithmes de résolution automatique d'un labyrinthe.

Les parties sont majoritairement indépendantes les unes des autres. Cela signifie que la plupart des fonctions peuvent être implémentées dans l'ordre que vous souhaitez. Néanmoins, il est nécessaire de faire le début de la partie I avant d'aborder le reste. Plus précisément, vous devez commencer par écrire les fonctions de base qui manipulent des labyrinthes et au moins une fonction de génération : la plupart des fonctions ont besoin d'un labyrinthe pour être testées.

2 Informations pratiques

2.1 Récupération des sources et mise en place d'un dépôt Gitlab au CREMI

— Commencez par cloner le dépôt suivant :

```
$ git clone git@gitlab.emi.u-bordeaux.fr:tplace/labyrinthes.git
```

— Changez l'URL d'`origin` dans dépôt que vous avez récupéré à celle de votre nouveau projet et poussez l'ensemble du projet dans votre nouveau dépôt. Vous pouvez garder `labyrinthes` comme nom de projet si vous le souhaitez.

```
$ cd labyrinthes
$ git remote set-url origin git@gitlab.emi.u-bordeaux.fr:votre_login/votre_nom_de_projet.git
$ git push
```

— Vérifiez que votre nouveau projet est bien **privé**.

— Enfin, sous gitlab, invitez avec le rôle de `Maintainer` les **trois** enseignants (noms disponibles sous Celcat).

2.2 Dates importantes

1. Urgent ! **Constitution des groupes** : le sujet est à réaliser par **groupes de deux personnes**. Sous moodle , vous devez constituer votre groupe de projet, avant le **15/11/2024**.
2. **Rendus**. Vous devez faire **deux rendus** sous Moodle :
   — Le premier avant le **20/11/2024**, implémentant les fonctions des fichiers suivants. Les trois premiers ont déjà été implémentés en grande partie en TP de C :
      — `data_dynarray.c` : gestion des tableaux dynamiques.
      — `data_queue.c` : gestion des files.
      — `maze.c` : implémentation des labyrinthes.
      — `maze_gen.c` : génération de labyrinthes. Pour ce premier rendu, vous n'avez à programmer qu'une fonction de génération de labyrinthe.
      — `maze_braiding.c` : tressage de labyrinthes.
   — Le second avant le **13/12/2024** pour le rendu final.
3. **Soutenances** : elles seront organisées **entre le 13 et le 20 décembre 2024**.

3 Ce qui est fourni

Une fois le projet récupéré (cf. section précédente), plusieurs fichiers vous sont fournis :

— Le sujet au format **pdf** (c'est le fichier que vous êtes en train de lire). Il présente une vue d'ensemble du projet et décrit à haut niveau les **algorithmes qu'on va devoir implémenter afin de générer de nouveaux labyrinthes et de les parcourir ensuite**. **Attention**, le sujet ne détaille pas les fonctions à écrire en **C**, c'est le rôle de la documentation (voir le point suivant).

— Tous les fichiers d'en-tête (**.h**) accompagnés d'une **documentation complète** au format **html**. La documentation décrit précisément toutes les fonctions à écrire. Elle est donc nécessaire à la réalisation du projet. Elle est accessible via un navigateur, depuis le répertoire du projet :

```
$ google-chrome Documentation/html/index.html
```

— Les squelettes des fichiers (**.c**) que vous devez compléter.

— Le fichier `error.h`, qui fournit des macros permettant d'afficher des messages, avec localisation du fichier, ligne et fonction d'où provient le message.

— Les fichiers `alloc.h` et `alloc.c` qui fournissent des macros d'allocation.

— Les fichiers `main.h` et `main.c`. Vous pouvez modifier les « *#define* » dans le fichier `main.h` afin de changer les paramètres de génération du labyrinthe initial de l'interface (pour les labyrinthes suivants, les paramètres sont définis directement dans l'interface).

— Un fichier **Makefile**. Attention,il ne doit pas être changé. Le programme est compilé de façon à provoquer une erreur en cas d'avertissement. Il faut donc que la compilation ne provoque aucun avertissement. Pour lancer le projet après compilation, utilisez la commande suivante :

```
$ ./projet
```

— Si besoin, vous pouvez ajouter vos propres fichiers **.c**/**.h**. Le fichier Makefile prendra en compte ces fichiers automatiquement pour la compilation.

Les fichiers sources sont répartis en plusieurs catégories correspondant chacune à une partie du projet. La catégorie de chaque fichier est indiquée par son préfixe.

— Les fichiers avec le préfixe `data` implémentent les types abstraits dont on aura besoin lors du projet. Vous avez déjà implémenté la plupart d'entre eux en cours de **C**.

— Les fichiers avec le préfixe `maze` correspondent à la partie « génération de labyrinthes » du projet. Ils sont décrits plus en détail dans la suite.

— Les fichiers avec le préfixe `game` correspondent à la partie « implémentation d'un jeu » du projet. Ils sont décrits plus en détail dans la suite.

— Les fichiers avec le préfixe `solve` correspondent à la partie « résolution automatique » du projet. Ils sont décrits plus en détail dans la suite.

— Enfin, les fichiers avec le préfixe `gui` sont déjà écrits et implémentent l'interface du projet. **Vous ne devez pas les modifier.**

Vous ne devez **pas** modifier les fichiers d'en-tête (**.h**), ni le fichier `main.c`, ni le `Makefile`, ni les fichiers qui permettent d'implémenter l'interface graphique. En effet, lorsque votre projet sera testé automatiquement, ces fichiers seront écrasés par les fichiers originaux.

3.1 L'affichage des labyrinthes

Les labyrinthes sont affichés en utilisant des *sprites*[1]. Les images correspondantes sont contenues dans le répertoire « `Sprites/` » du projet. **Il ne faut pas modifier ces fichiers** sous peine de corrompre l'affichage.

Vous pouvez cependant remplacer le fichier `player.png` si vous voulez changer l'apparence du joueur. Pour cela, vous devez générer une nouvelle feuille de sprites. Le générateur à utiliser se trouve à l'adresse suivante `http://sanderfrenken.github.io/Universal-LPC-Spritesheet-Character-Generator/`.

4 Partie I : Génération de labyrinthes

Toutes les fonctions à implémenter pour cette partie sont regroupées dans les fichiers dont le nom a pour préfixe « `maze` ». C'est la partie à commencer en premier car il est nécessaire d'avoir programmé au moins une

1. Tous les sprites utilisés dans le projet sont libres et proviennent du site `http://opengameart.org`

fonction de génération pour que l'interface puisse afficher un labyrinthe (ce qui est nécessaire afin de tester les autres fonctionnalités demandées).

> ? Il n'est pas nécessaire d'avoir *terminé* cette partie pour aborder les autres. En revanche, il est nécessaire d'écrire la majorité des fonctions permettant de manipuler un labyrinthe dans le fichier `maze.c` ainsi qu'au moins une fonction de génération dans `maze_gen.c`.
> Les autres fichiers implémentent des fonctionnalités qui ne sont pas nécessaires pour les parties II et III. On pourra donc les écrire quand on le souhaite.

4.1 Fichiers sources

Les fichiers liés à cette partie sont les suivants. Chacun d'entre eux est dédié à une fonctionnalité différente.

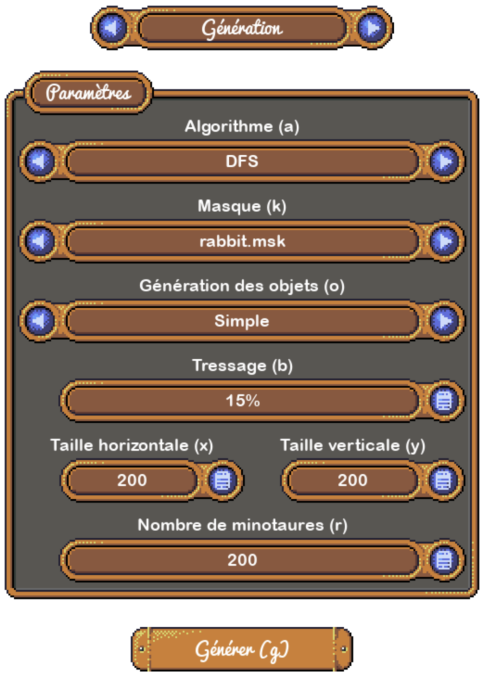


FIGURE 1 – Génération d'un labyrinthe avec le masque « `rabbit.msk` »

— `maze.h` et `maze.c`. Ce sont les premiers fichiers à implémenter avant de passer au reste du projet. Ils contiennent le type utilisé pour représenter un labyrinthe et les fonctions utilisées pour manipuler ce type.

— `maze_gen.h` et `maze_gen.c`. Contient toutes les fonctions de génération d'un labyrinthe. Les algorithmes standard sont décrits plus bas. Il est possible d'ajouter ses propres algorithmes.

— `maze_mask.h` et `maze_mask.c`. Contient toutes les fonctions qui servent à charger un masque à partir d'un fichier. Les masques permettent de donner aux labyrinthes la former que l'on souhaite (comme dans la Figure 1 ci-dessous). On peut bien sûr ne pas utiliser de masque, dans ce cas on obtient un labyrinthe rectangulaire. Il est possible de créer ses propres masques : il suffit d'ajouter un fichier « `.msk` » dans le répertoire « `Masques` ». Plusieurs exemples sont fournis dans ce répertoire.

— `maze_objects.h` et `maze_objects.c`. Contient toutes les fonctions qui servent à générer les objets à l'intérieur d'un labyrinthe déjà construit (les trésors, les bombes, les polys d'algo et les sorties). Il est possible d'ajouter ses propres algorithmes.

— `maze_braiding.h` et `maze_braiding.c`. Contient toutes les fonctions qui servent à tresser un labyrinthe déjà construit. Les algorithmes de génération produisent des labyrinthes « parfaits » (c'est-à-dire sans cycles). Le tressage consiste à créer des cycles en cassant les murs de certains culs-de-sac (voir le fichier `maze_braiding.h` pour plus de détails).

Nous allons maintenant décrire différents algorithmes standard qu'on peut utiliser pour générer un nouveau labyrinthe et qui devront être implémentés dans le projet. Tous ces algorithmes fonctionnent selon le même principe. On commence par construire une grille à l'intérieur de laquelle toutes les cellules sont *« murées » dans les quatre directions*. Nous appellerons cet objet un **proto-labyrinthe**. Une représentation graphique d'un proto-labyrinthe est donnée dans la Figure 2 ci-contre. Les fonctions qui construisent un proto-labyrinthe sont à implémenter dans le fichier `maze.c`. Une fois le proto-labyrinthe construit, on utilise un algorithme de **génération** pour « creuser ses murs » afin que toutes ses cellules soient reliées. Il y a plusieurs algorithmes de génération qui diffèrent par la stratégie qu'ils adoptent afin de creuser les murs. Ils devront être implémentés dans le fichier `maze_gen.c`. Il est également possible d'inventer ses propres algorithmes de génération et d'ajouter de nouvelles fonctions qui les implémentent dans le fichier `maze_gen.c` (référez-vous aux instructions du fichier `maze_gen.h` pour que ces nouvelles fonctions apparaissent dans l'interface).

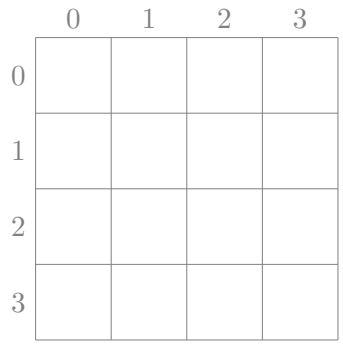


FIGURE 2 – Un proto-labyrinthe.

Utilisation des masques

Le proto-labyrinthe représenté dans la Figure 2 a été construit sans utiliser de masque : il est de forme rectangulaire. Si on se sert d'un masque, la situation est un peu plus compliquée. Dans ce cas, les cellules du proto-labyrinthe sont divisées en trois catégories :

1. Les cellules « masquées » indiquées par le masque.
2. Les cellules « non-accessibles ». Bien que non-masquées, elles sont séparées de la position initiale du joueur par des cellules masquées.
3. Les cellules « accessibles ». Celles qui n'entrent pas dans les deux premières catégories.

Ces informations sont calculées lors de la création du proto-labyrinthe (si aucun masque n'est utilisé, toutes les cellules sont accessibles). Les algorithmes de génération ne considèrent **que** les cellules accessibles. C'est-à-dire qu'ils ne peuvent casser **que** les murs qui séparent deux cellules accessibles.

Un point important est que les algorithmes doivent tous produire des labyrinthes « **parfaits** ». On dit d'un labyrinthe qu'il est parfait si celui-ci ne contient **pas de cycle**. Les algorithmes ne peuvent donc pas casser n'importe quel mur. La génération peut donc être vue comme un problème standard de théorie des graphes : le **calcul d'un arbre couvrant**. En effet, on peut voir un proto-labyrinthe comme un graphe : ses sommets sont les cellules et ses arêtes sont les murs. On donne une représentation graphique de cette idée dans la Figure 3 ci-dessous.

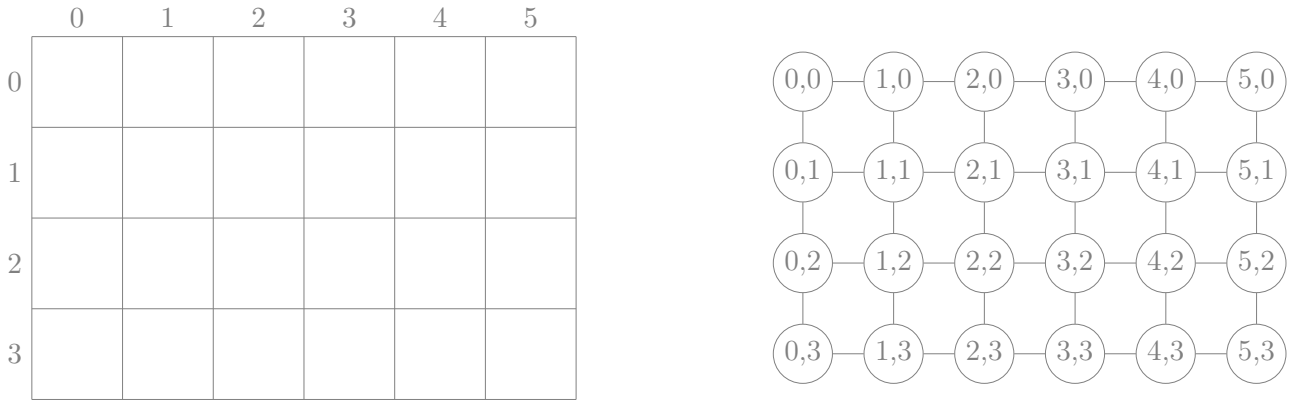


FIGURE 3 – Un proto-labyrinthe (construit sans masque) et le graphe correspondant.

De ce point de vue, générer un labyrinthe à partir d'un proto-labyrinthe revient à choisir un sous-ensemble d'arêtes (elles correspondent aux murs que l'on va casser) qui donne un arbre couvrant du graphe. On reprend le proto-labyrinthe de la Figure 3 pour donner un arbre couvrant du graphe et le labyrinthe final correspondant dans la la Figure 4 ci-dessous.

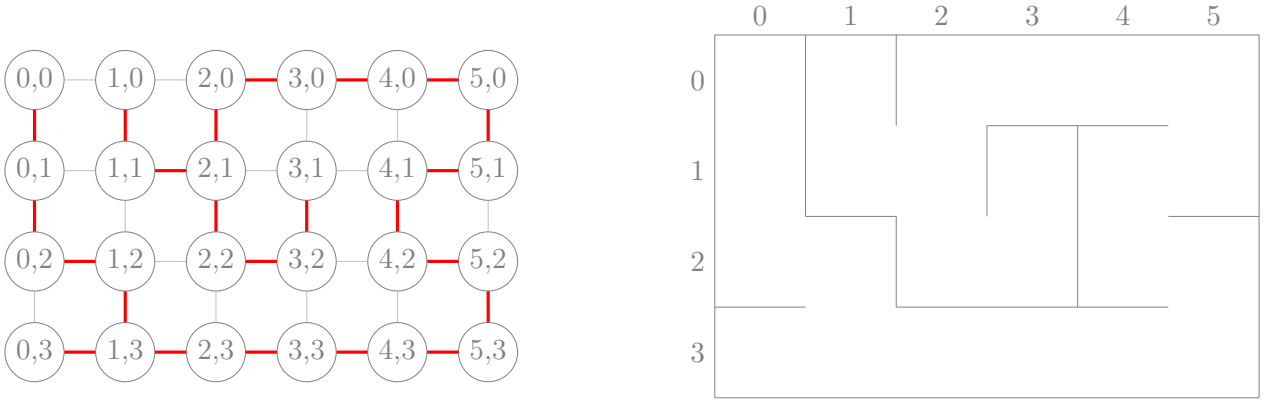


FIGURE 4 – Un arbre couvrant et le labyrinthe qu'il génère.

La plupart des algorithmes de génération sont donc inspirés par des algorithmes standard de la théorie des graphes. On va diviser en quatre catégories ceux que nous présentons ici :

1. L'approche basée sur les marches aléatoires.
2. L'approche « Hunt & Kill » basée sur les parcours de graphe.
3. L'approche basée sur les algorithmes de calcul d'un arbre couvrant (Prim et Kruskal).
4. La méthode par divisions récursives (un peu à part).

Tressage et création de cycles

On peut bien sûr construire des labyrinthes qui contiennent des cycles. Cependant, cette étape arrive **après** la génération. Dans tous les cas, on commence par générer un labyrinthe parfait. On ajoute ensuite des cycles en cassant des murs supplémentaires. Cette étape s'appelle le **tressage** et les fonctions correspondantes sont à écrire dans le fichier `maze_braiding.c`.

4.2 Marches aléatoires

Une **marche aléatoire** dans un graphe consiste à l'explorer entièrement en partant d'un sommet choisi aléatoirement. À chaque étape, on avance en choisissant *aléatoirement* un nouveau sommet parmi tous ceux qui sont adjacents au sommet courant. La marche s'arrête lorsque tous les sommets du graphe ont été explorés.

Le principe de la marche aléatoire n'interdit pas qu'on passe plus d'une fois par le même sommet. En conséquence, il est théoriquement possible qu'une marche ne s'arrête jamais (elle passe éternellement pas les mêmes sommets sans les avoir tous visités). Il faudra donc faire attention à ne l'utiliser que pour générer des labyrinthes relativement « petits ».

Il y a deux algorithmes importants basés sur le principe de la marche aléatoire. Bien qu'ils soient lents (voir la remarque ci-dessus), ils sont intéressants car ce sont les seuls qui ne sont pas biaisés : quand on les utilise, tous les labyrinthes possibles sont équiprobables.

Algorithme de Aldous-Broder. C'est l'un des algorithmes les plus simples. On part d'une cellule choisie aléatoirement et on effectue une marche aléatoire. À chaque étape, il y a deux possibilités suivant la cellule dans laquelle la marche arrive :

— Si la cellule n'a encore jamais été visitée, on casse le mur correspondant à l'arête qui nous y a menés.

— Si la cellule a déjà été visitée, on ne fait rien (le mur correspondant à l'arête qui nous y a menés est préservé si il existe encore).

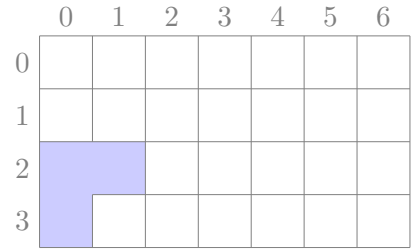
En particulier, l'algorithme à besoin d'une structure de données pour se souvenir de l'ensemble des cellules déjà visitées. On peut le décrire par le pseudo-code suivant :

```
On choisit aléatoirement une cellule cell de départ.
On ajoute cell à l'ensemble des cellules visitées.
tant que il reste une cellule non visitée faire
    On choisit aléatoirement une cellule adjacente adj à cell.
    si adj n'a pas encore été visitée alors
        On casse le mur qui mène de cell à adj et on ajoute adj à l'ensemble des cellules visitées.
    fin si
    On remplace cell par adj pour le tour de boucle suivant.
fin tant que
```

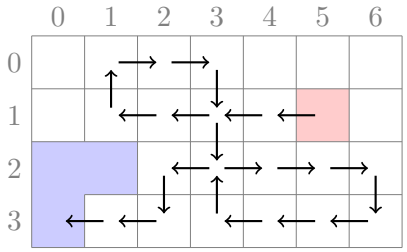
Algorithme de Wilson. Cet algorithme est un peu plus compliqué que le précédent. En particulier, il implique de faire *plusieurs marches aléatoires*. Une fois encore, on va devoir maintenir une structure de données pour se souvenir des cellules déjà visitées.

On commence par choisir une cellule **aléatoirement** qu'on ajoute à l'ensemble des cellules visitées. Ensuite, tant qu'il reste au moins une cellule non visitée, on effectue les actions suivantes (décrites dans la Figure 5) :

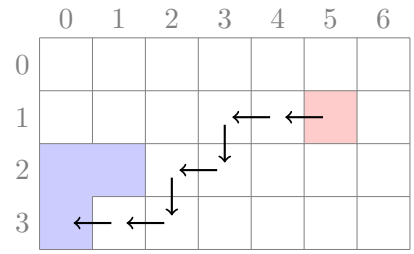
1. On choisit **aléatoirement** une cellule parmi celles qui n'ont pas encore été visitées.
2. On effectue une marche aléatoire depuis cette cellule jusqu'à trouver une cellule déjà visitée. Cette phase demande de mémoriser le chemin emprunté par la marche aléatoire pour les étapes suivantes.
3. On simplifie le chemin emprunté par la marche en supprimant tous les (éventuels) cycles qu'il contient.
4. On casse tous les murs sur le chemin qui en résulte et on ajoute ses cellules aux cellules visitées.
5. Si il reste encore des cellules non visitées, on repasse à l'étape 1.



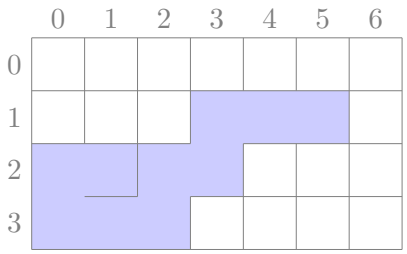
État initial (cellules visitées en bleu)



Étapes 1 et 2 : marche aléatoire à partir d'une cellule non visitée



Étape 3 : suppression des cycles



Étape 4 : on casse les murs et on met à jour les cellules visitées

FIGURE 5 – Un tour de boucle dans l'algorithme de Wilson

On peut résumer l'algorithme de Wilson par le pseudo-code suivant :

```
On choisit une cellule aléatoirement qu'on ajoute à l'ensemble des cellules visitées.
tant que il reste une cellule non visitée faire
    On choisit aléatoirement une cellule non visitée qu'on note cell.
    tant que cell n'est pas une cellule déjà visitée faire          // La marche aléatoire
        On choisit aléatoirement une cellule adjacente adj à cell.
        On enregistre cell (pour se souvenir du chemin effectué par la marche).
        On remplace cell par adj.
    fin tant que
    On élimine les cycles sur le chemin effectué par la marche aléatoire.
    On casse les murs sur le chemin et on ajoute toutes ses cellules à l'ensemble des cellules visitées.
fin tant que
```

4.3 L'approche Hunt & Kill

Malgré son nom, cette approche est certifiée 100% non violente. Elle reprend le principe d'une exploration aléatoire de tout le graphe utilisé par l'algorithme de Aldous-Broder. Il y a cependant une différence importante : ici, on interdit à l'exploration de passer deux fois par la même cellule. En d'autres termes, cette approche n'utilise pas une « vraie » marche aléatoire. On part d'une cellule choisie de manière aléatoire. Ensuite, à chaque étape, on choisit aléatoirement une cellule adjacente encore non visitée, on casse le mur qui y mène et on recommence avec la cellule adjacente choisie. Cette approche pose néanmoins un problème : il peut arriver que l'exploration se retrouve « bloquée » avant d'avoir visité tout le graphe car toutes les cellules adjacentes ont déjà été visitées, comme dans la Figure 6.

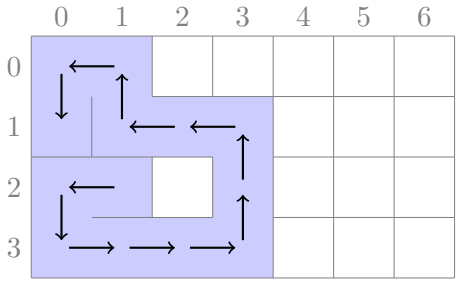


FIGURE 6 – Situation bloquée dans l'approche Hunt & Kill

On résout les situations de blocage en recherchant une cellule qui a au moins une voisine non visitée parmi celles qu'on a déjà visitées et en recommençant l'exploration depuis cette cellule.

Un algorithme basé sur l'approche Hunt & Kill a donc besoin de maintenir un ensemble de cellules visitées. On commence par choisir une cellule aléatoire qui va servir de point de départ et on l'ajoute à l'ensemble des cellules visitées. Ensuite, chaque cycle de l'algorithme est divisé en deux phases consécutives :

— **Hunt** : On recherche une cellule déjà visitée qui a **au moins une cellule voisine non visitée** (la façon dont se fait cette recherche est volontairement imprécise, nous y reviendrons plus tard). On la note *cell* pour la phase suivante.

— **Kill** : On explore à partir de *cell* jusqu'à ce qu'on soit bloqué. Tant que *cell* possède au moins une voisine non visitée, on en choisit une aléatoirement qu'on note *adj*. On casse le mur menant de *cell* à *adj*, on ajoute *adj* à l'ensemble des cellules visitées et on remplace *cell* par *adj* pour la suite de l'exploration.

Ces deux étapes sont répétées par l'algorithme tant qu'il reste des cellules non visitées. On peut donc résumer l'algorithme par le pseudo-code suivant.

```
On choisit une cellule aléatoirement et on l'ajoute à l'ensemble des cellules visitées.
tant que il reste une cellule non visitée faire
    /* Phase Hunt                                                    */
    On recherche dans les cellules visitées une cellule cell qui a une voisine non visitée.
    /* Phase Kill                                                    */
    tant que cell a une voisine non visitée faire
        On choisit aléatoirement une cellule voisine non visitée adj de cell.
        On casse le mur qui mène de cell à adj.
        On ajoute adj à l'ensemble des cellules visitées.
        On remplace cell par adj.
    fin tant que
fin tant que
```

Il nous reste maintenant à expliquer comment se déroule la phase « **Hunt** » consistant à rechercher une cellule parmi celles que l'on a déjà visitées. Il y a en réalité plusieurs façons de réaliser cette phase, chacune donnant lieu à un algorithme Hunt & Kill différent. Dans le projet, on en propose trois :

1. **Recherche linéaire** : On passe en revue linéairement *toutes les cellules du graphe* jusqu'à en trouver une qui a déjà été visitée et qui a une voisine non visitée.
2. **Recherche DFS** : On passe en revue *uniquement les cellules déjà visitées de la plus récente à la plus ancienne* jusqu'à en trouver une qui a une voisine non visitée. L'exploration effectuée en utilisant cette méthode de recherche correspond en réalité à un parcours en largeur du graphe (« DFS » pour depth first search). Pour se servir de cette méthode, il faudra maintenir un historique trié par ancienneté des cellules déjà visitées. On pourra l'implémenter avec une **pile** (l'outil classique pour un DFS).
3. **Recherche aléatoire** : On *choisit aléatoirement* dans l'ensemble des cellules visitées une cellule qui a une voisine non visitée.

Contrairement aux algorithmes basés sur les marches aléatoires, les algorithmes « Hunt & Kill » sont *biaisés* : certains labyrinthes ont plus de probabilité d'être générés que d'autres. Typiquement, ces algorithmes produisent des labyrinthes possédant de longs couloirs et peu de culs-de-sac. Cette propriété est facilement observable dans l'interface une fois les algorithmes implémentés.

4.4 Calcul d'un arbre couvrant

Cette approche consiste à adapter les algorithmes de Prim et Kruskal qui servent à calculer un *arbre couvrant de poids minimal* pour un graphe pondéré passé en entrée. Ces deux algorithmes fonctionnent en passant en revue une par une toutes les arêtes du graphe afin de sélectionner celles qui feront partie de l'arbre couvrant. Du point de vue des labyrinthes, cela veut dire que cette approche n'est *pas* basée sur une exploration du graphe comme les deux précédentes. On rappelle que dans le graphe associé à un proto-labyrinthe les arêtes correspondent aux murs. Nos algorithmes vont donc passer en revue *tous les murs* du proto-labyrinthe initial et décider pour chacun d'entre eux si celui-ci doit être détruit ou conservé. On aura donc besoin d'utiliser une structure de données qui enregistre l'ensemble des arêtes qui restent à « traiter ».

Les algorithmes de Prim et Kruskal sont conçus pour prendre un graphe *pondéré* en entrée et calculer *arbre couvrant de poids minimal*. Pour cette raison, les deux algorithmes trient les arêtes selon leur poids. Dans le contexte de la génération de labyrinthe, les arêtes n'ont pas de « poids ». À la place, on prendra les arêtes dans un ordre aléatoire.

Algorithme de Prim. On a besoin de structures de données pour représenter deux ensembles (vides au départ) : l'ensemble des cellules qui ont déjà été visitées et l'ensemble des arêtes qu'on va devoir « traiter ».

On commence par choisir une cellule aléatoirement et on l'ajoute à l'ensemble des cellules visitées. De plus, on ajoute toutes les arêtes issues de la cellule à l'ensemble des arêtes à traiter. Ensuite, tant qu'il reste au moins une arête à traiter, on effectue les actions suivantes :

— On choisit aléatoirement une arête à traiter qu'on retire de l'ensemble.

— Par construction, cette arête est issue d'une cellule déjà visitée. On regarde donc la cellule vers laquelle elle mène. Si cette cellule n'a pas encore été visitée, alors on casse le mur correspondant à l'arête, on ajoute la cellule à l'ensemble des cellules visitées et on ajoute toutes les arêtes issues de celle-ci à l'ensemble des arêtes à traiter.

Ces deux actions sont effectuées tant qu'il reste des arêtes à traiter. L'algorithme est donc décrit par le pseudo-code suivant :

```
On choisit une cellule aléatoirement et on l'ajoute à l'ensemble des cellules visitées.
On ajoute toutes les arêtes issues de cette cellule à l'ensemble des arêtes à traiter.
tant que il reste des arêtes à traiter faire
    On retire une arête choisie aléatoirement de l'ensemble des arêtes à traiter.
    On note cell la cellule vers laquelle mène cette arête.
    si cell n'a pas encore été visitée alors
        On casse le mur correspondant à l'arête.
        On ajoute cell à l'ensemble des cellules visitées.
        On ajoute toutes les arêtes issues de cell à l'ensemble des arêtes à traiter.
    fin si
fin tant que
```

Algorithme de Kruskal. On a à nouveau besoin d'une structure de données pour représenter l'ensemble des arêtes qu'on va devoir « traiter ». De plus l'algorithme demande de maintenir une *partition* de l'ensemble des cellules. Écrire cet algorithme demande donc d'implémenter le type abstrait « union-find » dans le fichier `data_ufind.c` afin de représenter cette partition [2].

L'algorithme commence par ajouter *toutes* les arêtes du graphe à l'ensemble des arêtes à traiter. De plus, on initialise la partition des cellules en une partition triviale : toutes les classes sont des singletons et chaque cellule est donc isolée des autres (au fur et à mesure que l'algorithme avance, les classes seront progressivement fusionnées). Ensuite, tant qu'il reste une arête à traiter, on effectue les actions suivantes :

— On choisit aléatoirement une arête à traiter qu'on retire de l'ensemble.

— Si les deux cellules reliées par cette arête sont dans deux classes distinctes de la partition, alors on fusionne ces classes et on casse le mur correspondant à l'arête.

Ces deux actions sont effectuées tant qu'il reste des arêtes à traiter. L'algorithme est donc décrit par le pseudo-code suivant :

```
On crée une partition de l'ensemble des cellules, dans laquelle toutes les classes sont des singletons.
On ajoute toutes les arêtes du graphe à l'ensemble des arêtes à traiter.
tant que il reste une arête à traiter faire
    On retire une arête choisie aléatoirement de l'ensemble des arêtes à traiter.
    On note cell_1 et cell_2 les deux cellules reliées par cette arête.
    si cell_1 et cell_2 sont dans deux classes distinctes de la partition alors
        On casse le mur correspondant à l'arête.
        On fusionne les classes de cell_1 et cell_2.
    fin si
fin tant que
```

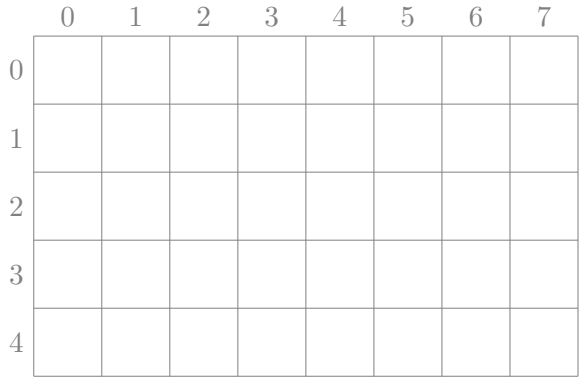
2. L'algorithme de Kruskal est la seule partie du projet qui utilise « union-find ».

4.5 Divisions récursives

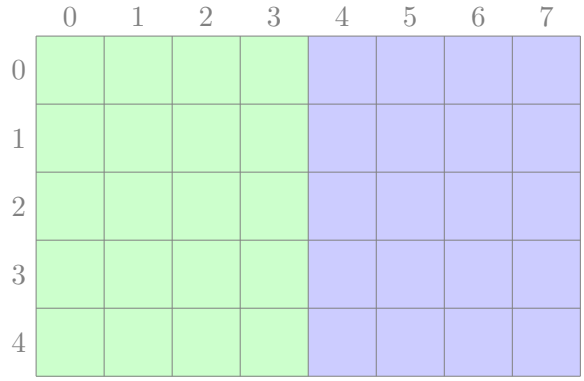
Cet algorithme est à part car il ne vient pas de la théorie des graphes : il est spécifique aux labyrinthes. Plus précisément, il exploite le fait que *les graphes associés aux proto-labyrinthes ont une forme particulière* qui permet de facilement les « couper en deux parties de taille similaire ». Pour les mêmes raisons, une autre particularité de cet algorithme est qu'*il ne prend pas en compte les masques*. Il est conçu uniquement pour prendre un proto-labyrinthe de forme rectangulaire en entrée.

Il est en réalité possible d'adapter cet algorithme pour qu'il fonctionne aussi quand on se sert des masques, mais c'est significativement plus compliqué.

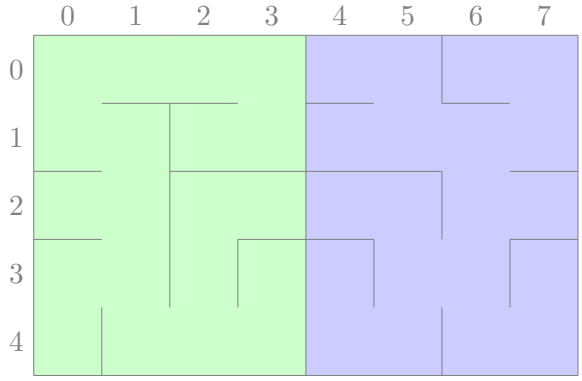
Le principe de l'algorithme est simple. Puisque que l'on part d'un proto-labyrinthe rectangulaire, il est possible de le « couper en deux » sur sa largeur ou sur sa longueur (en pratique, on coupe selon la plus grande des deux dimensions). Cette action crée deux proto-labyrinthes « plus petits » qu'on peut creuser récursivement. Il suffit ensuite de casser *un seul mur* (choisi aléatoirement) sur la ligne ou la colonne qui sépare les deux labyrinthes créés récursivement pour les relier en un seul labyrinthe. Le cas de base de la récursion se produit lorsque le proto-labyrinthe est réduit à une seule cellule. On décrit l'étape de récursion dans la Figure 7.



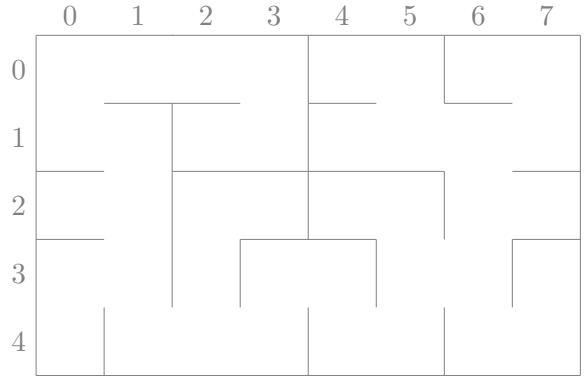
Proto-labyrinthe initial



1. On coupe le labyrinthe en deux parties



2. On creuse les deux parties en utilisant la récursion



3. On casse un mur sur la colonne séparant les deux parties

FIGURE 7 – Une étape de récursion dans l'algorithme de divisions récursives

L'algorithme est résumé par le pseudo-code suivant :

1: On note h et ℓ la hauteur et la largeur du proto-labyrinthe.
2: **si** $h = 1$ et $\ell = 1$ **alors**
3: | On ne fait rien.
4: **sinon si** $h \geqslant \ell$ **alors**
5: | On prend la ligne qui coupe le labyrinthe en deux parties égales (à 1 près).
6: | On applique récursivement l'algorithme à chacune des deux parties.
7: | On casse un seul mur choisi **aléatoirement** sur la ligne de séparation.
8: **sinon si** $h < \ell$ **alors**
9: | On prend la colonne qui coupe le labyrinthe en deux parties égales (à 1 près).
10: | On applique récursivement l'algorithme à chacune des deux parties.
11: | On casse un seul mur choisi **aléatoirement** sur la colonne de séparation.
12: **fin si**

5 Partie II : Jeu dans un labyrinthe

Cette partie consiste à implémenter toutes les fonctions qui permettent de gérer une partie à l'intérieur d'un labyrinthe qui a été construit avec l'un des algorithmes de la Partie I. Les fonctions à implémenter pour cette partie sont regroupées dans les fichiers dont le nom a pour préfixe « `game` ».

Nous allons brièvement décrire les fichiers liés à cette partie. Référez-vous à la documentation des sources pour plus de détails.

— `game.h` et `game.c`. Ce sont les premiers fichiers à implémenter pour la Partie II. Ils contiennent le type utilisé pour représenter un jeu et les fonctions utilisées pour manipuler ce type. En particulier, les fonctions qui mettent à jour le jeu selon les actions effectuées par le joueur (déplacement, utilisation d'une bombe ou utilisation d'un poly d'algo) sont toutes à implémenter dans ces fichiers.

— `game_strategies.h` et `game_strategies.c`. Dans le jeu, des minotaures se déplacent (possiblement) lorsque le joueur se déplace. Dans ce cas, les déplacements effectués par les minotaures sont calculés par une « fonction de stratégie » qui est passée en paramètre à la fonction qui gère les déplacements du joueur (implémentée dans `game.c`). Dans le projet, on va considérer plusieurs fonctions de stratégie distinctes. On peut par exemple les laisser toujours immobiles, leur faire effectuer des mouvements opposés à ceux du joueur, toujours les déplacer selon le plus court chemin qui mène au joueur, etc... Les fonctions de stratégie sont à implémenter dans ces deux fichiers. Plusieurs stratégies naturelles sont données en exemple et il est possible d'ajouter ses propres stratégies.

— `game_history.h` et `game_history.c`. Contient toutes les fonctions qui servent à gérer l'historique d'une partie. L'historique permet au joueur d'annuler les actions qu'il a effectuées et ensuite de les restaurer si il le souhaite (tant qu'il n'a pas effectué de nouvelle action).

Le type qu'on utilise pour représenter l'historique est laissé libre. C'est à vous de choisir comment l'implémenter afin de satisfaire les spécifications données dans la documentation.

6 Partie III : Résolution automatique d'un labyrinthe

Dans cette partie, on va écrire plusieurs fonctions qui permettent d'assister le joueur. Par exemple afin de trouver un chemin qui mène de sa position à la sortie du labyrinthe. Les fonctions à implémenter pour cette partie sont regroupées dans les fichiers dont le nom a pour préfixe « `solve` ». Il y a en fait deux modules de résolution automatique qu'on va décrire séparément.

6.1 Calcul de chemins simples

FIGURE 8 – Un chemin simple affiché par l'interface

Les fonctions de ce module servent uniquement à calculer des chemins **sans cycle**. Elles sont spécifiquement conçues pour résoudre le problème mentionné plus haut : trouver un chemin qui mène à la sortie (ou à un objet que l'on a choisi) depuis la position du joueur. Quand un chemin a été trouvé, l'interface est conçue pour l'afficher comme dans la Figure 8 ci-contre (l'interface peut afficher jusqu'à trois chemins différents simultanément pour les comparer).

Le calcul est réalisé en utilisant des algorithmes standard de parcours de graphe. On explore le labyrinthe jusqu'à trouver ce que l'on cherche. Ensuite, si on a trouvé cet objet, on reconstitue le chemin qui nous y a mené. Le projet inclut *trois* algorithmes de parcours à implémenter :

1. Algorithme de parcours en largeur (BFS pour « breadth first search »). Basé sur les files, il trouve toujours le chemin le plus court mais demande d'explorer beaucoup de cellules.
2. Algorithme de parcours en profondeur (DFS pour « depth first search »). Basé sur les piles, il a tendance à trouver des chemins « longs ».
3. Algorithme A^*. C'est le plus compliqué des trois. Il est paramétré par une heuristique et son implémentation utilise les tas binaires. Il a tendance à donner chemins courts et est souvent plus efficace que le parcours en largeur.

Bien que ce ne soit pas strictement nécessaire, il est recommandé d'avoir implémenté le tressage de labyrinthe afin de pouvoir comparer les résultats donnés par ces trois algorithmes. En effet, si un labyrinthe ne contient aucune cycle, il n'y a toujours qu'un seul chemin entre un point A et un point B : les trois algorithmes trouveront donc toujours le même chemin (la seule différence résidant dans le nombre de cellules qu'ils ont dû explorer pour le trouver).

Nous allons maintenant décrire le fonctionnement des trois algorithmes introduits plus haut. On commence par les deux premiers qui sont quasi-identiques en termes d'implémentation. Pour passer de l'un à l'autre il suffit de changer la structure de données utilisée : file pour l'un et pile pour l'autre.

Parcours en largeur. Cet algorithme utilise une file. De plus, puisque nos labyrinthes contiennent potentiellement des cycles, il faut utiliser une structure de données qui mémorise l'ensemble des cellules déjà visitées par le parcours (c'est nécessaire pour éviter de visiter une même cellule plusieurs fois et faire en sorte que le parcours termine). L'algorithme est résumé par le pseudo-code suivant :

```
On crée une file vide et on y enfile la cellule de départ.
tant que la file n'est pas vide faire
    On défile une cellule qu'on note cell de la file.
    si cell a déjà été visitée alors
        On passe au tour de boucle suivant.
    sinon si cell contient l'objet recherché alors
        On reconstitue le chemin menant à cell et on le retourne (ce qui met fin au parcours).
    sinon
        On ajoute cell à l'ensemble des cellules visitées.
        On enfile toutes les cellules accessibles depuis cell dans la file.
    fin si
fin tant que
```

On peut vérifier qu'un parcours en largeur explore les cellules par ordre de distance entre celles-ci et la cellule de départ. En d'autres termes, plus une cellule est proche de celle de départ, plus elle sera explorée « tôt ». Tout chemin trouvé par un parcours en largeur sera donc *optimal* : il ne peut pas exister de chemin plus court. En revanche, ce type de parcours est souvent « lent » : avant de trouver la cellule que l'on recherche, il faut toujours avoir exploré au préalable toutes les cellules qui sont plus proches du point de départ qu'elle.

Une fois l'objectif trouvé, l'algorithme demande de « reconstituer le chemin menant à cet objectif ». Pour effectuer cette opération, il faudra mémoriser de l'information supplémentaire pendant le parcours : pour chaque cellule explorée, on a besoin de la direction utilisée pour entrer dans celle-ci. On devra donc enregistrer cette information.

Parcours en profondeur. En termes d'implémentation, cet algorithme est quasi-identique au précédent. On va simplement remplacer la file par une *pile*. On rappelle que puisque nos labyrinthes contiennent potentiellement des cycles, on a également besoin d'utiliser une structure de données pour mémoriser l'ensemble des cellules déjà visitées par le parcours. L'algorithme est résumé par le pseudo-code suivant :

```
On crée une pile vide et on y empile la cellule de départ.
tant que la pile n'est pas vide faire
    On dépile une cellule qu'on note cell de la pile.
    si cell a déjà été visitée alors
        On passe au tour de boucle suivant.
    sinon si cell contient l'objet recherché alors
        On reconstitue le chemin menant à cell et on le retourne (ce qui met fin au parcours).
    sinon
        On ajoute cell à l'ensemble des cellules visitées.
        On empile toutes les cellules accessibles depuis cell dans la pile.
    fin si
fin tant que
```

Puisqu'on utilise une pile, un parcours en largeur n'abandonne jamais un chemin avant de se retrouver « bloqué ». Les chemins trouvés par ce type de parcours ne sont donc pas nécessairement optimaux. En pratique, on pourra constater que l'algorithme a plutôt tendance à produire des chemins longs. En revanche, quand un labyrinthe contient suffisamment de cycles, il est courant qu'on trouve un chemin plus rapidement avec un parcours en profondeur qu'avec un parcours en largeur.

À nouveau, l'étape finale consistant à « reconstituer le chemin menant à la cellule trouvée » demande d'avoir enregistré les directions utilisées pour entrer dans chaque cellule explorée.

Algorithme A^*. Les deux parcours précédents fonctionnent selon le même principe. Une structure de données (une file ou une pile) contient une liste de cellules à explorer. Un tour de boucle consiste à prendre une cellule dans cette structure afin de l'explorer (si ça n'a pas déjà été fait). Si cette cellule n'est pas celle que l'on cherche, on insère ses voisines dans la structure de données pour pouvoir les explorer plus tard. La seule chose qui différencie les deux algorithmes est l'ordre dans lequel les cellules sont extraites de la structure de données :

— Dans un parcours en largeur, on extrait toujours la plus ancienne des cellules à avoir été insérée.

— Dans un parcours en profondeur, on extrait toujours la plus récente des cellules à avoir été insérée.

Dans l'algorithme A^*, on essaye d'adopter une stratégie « plus intelligente » pour extraire les cellules de la structure de données. Cette stratégie est paramétrée par un nouvel objet qu'on appelle une *heuristique*.

Une *heuristique* est une fonction *rapidement calculable* qui prend une cellule en entrée et retourne une *évaluation de la distance restant à parcourir* depuis cette cellule pour atteindre l'objectif.

L'*heuristique est un paramètre de l'algorithme* A^* et on peut en définir plusieurs (des bonnes comme des mauvaises). Le projet est conçu pour qu'on puisse définir ses propres heuristiques et les sélectionner ensuite dans l'interface.

Une heuristique couramment utilisée dans ce contexte est la *distance manhattan* entre la cellule et l'objectif. La distance manhattan entre deux cellules A et B d'un labyrinthe est définie en ignorant les murs. C'est le nombre minimum de cellules qu'on doit traverser pour se rendre de A à B en se déplaçant uniquement horizontalement ou verticalement (en ignorant les murs). On peut vérifier qu'il s'agit bien d'une distance au sens mathématique du terme. On illustre cette définition dans la Figure 9 ci-dessous.

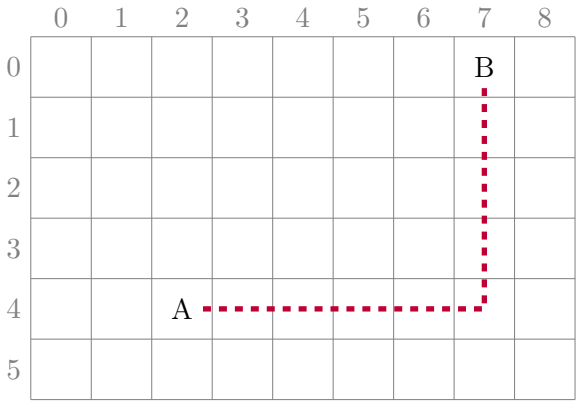


FIGURE 9 – Distance manhattan entre deux cellules d'un grille

Maintenant que nous savons ce qu'est une heuristique, nous pouvons présenter l'algorithme A^*. On note « h » l'heuristique utilisée : pour toute cellule *cell*, la valeur $h(cell)$ est un entier censé estimer le nombre de cellules qui séparent *cell* de l'objectif. Comme on l'a expliqué, A^* fonctionne selon le même principe que les parcours/profondeur. Néanmoins, la manipulation de la structure de données contenant les cellules à explorer est un peu plus compliqué. Lorsque l'on y insère une nouvelle cellule *cell*, on lui associe deux valeurs :

— Un *coût* noté $\mathbf{cost}(cell)$. C'est la longueur du chemin menant à *cell* que l'algorithme vient de trouver.

— Une *estimation* notée $\mathbf{esti}(cell)$ définie par $\mathbf{esti}(cell) = \mathbf{cost}(cell) + h(cell)$. Intuitivement, cette valeur évalue la longueur qu'aura un chemin complet du départ à l'objectif si on choisit de passer par *cell*.

Ensuite, à chaque fois qu'on extrait une cellule de la structure de données pour l'explorer, on choisit une cellule *cell* pour laquelle la valeur $\mathbf{esti}(cell)$ est *minimale*. Étant données ces contraintes, nous allons devoir utiliser une *file de priorité* pour implémenter notre structure de données. On va donc utiliser un *tas binaire*. L'algorithme est résumé par le pseudo-code suivant :

```
/* On note « h » l'heuristique utilisée par l'algorithme */
On créé un tas binaire vide et on y insère la cellule de départ scell
  avec les valeurs cost(scell) = 0 et esti(scell) = h(scell).
tant que le tas binaire n'est pas vide faire
    On extrait une cellule cell du tas binaire telle que esti(cell) est minimal.
    si cell a déjà été visitée alors
        On passe au tour de boucle suivant.
    sinon si cell contient l'objet recherché alors
        On reconstitue le chemin menant à cell et on le retourne (ce qui met fin au parcours).
    sinon
        On ajoute cell à l'ensemble des cellules visitées.
        pour chaque cellule acell accessible depuis cell faire
            On insère acell dans le tas
            avec les valeurs cost(acell) = cost(cell) + 1 et esti(acell) = cost(acell) + h(acell).
        fin pour
    fin si
fin tant que
```

6.2 Calcul de chemins complexes

Ce module permet de calculer des chemins qui contiennent des cycles. Dans le projet, on parle de « chemins complexes » et on les implémente par des séquences de chemins simples (sans cycle) consécutifs : la cellule d'arrivée d'un chemin de la séquence est aussi le point de départ du chemin suivant. Dans ce contexte, un problème typique pourrait être : « calculer un chemin complexe qui permet de récupérer *tous* les trésors avant de sortir du labyrinthe depuis la position du joueur ». La vue d'ensemble d'un tel chemin (tel que l'affiche l'interface) est représentée dans la Figure 10 (on peut aussi afficher chaque chemin simple individuellement).

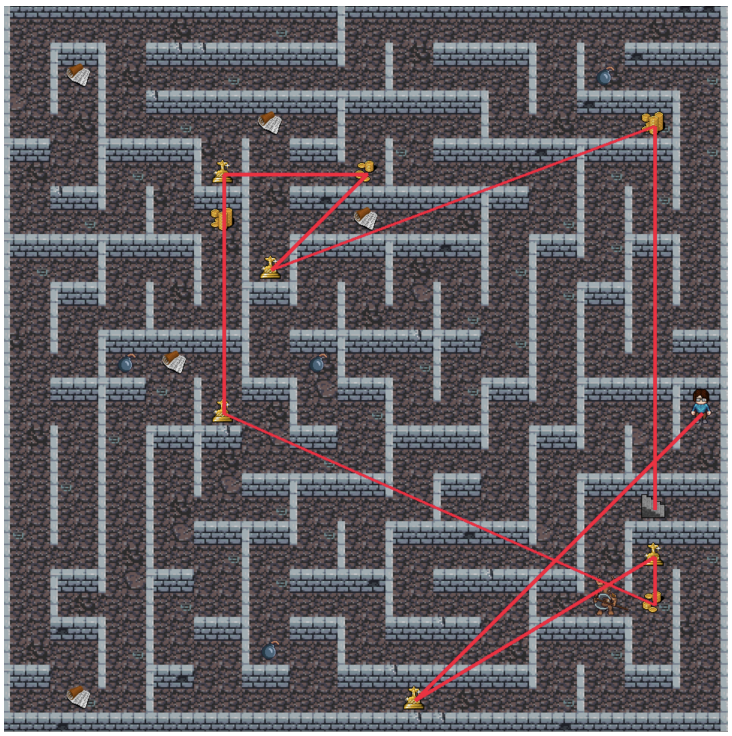


FIGURE 10 – Un chemin complexe pour récupérer tous les trésors

Le calcul de chemin complexe est *la partie la plus libre du projet*. Seul le type utilisé pour représenter un chemin complexe est imposé. Les algorithmes et leur implémentation sont libres.

La difficulté de cette partie dépend des objectifs que vous vous fixez. Par exemple, prenons le problème mentionné plu haut : « calculer un chemin complexe qui permet de récupérer *tous* les trésors avant de sortir du labyrinthe depuis la position du joueur ». Si on veut simplement calculer un tel chemin sans considérations d'optimalité, ce n'est pas très compliqué (on pourra en particulier réutiliser les fonctions écrites pour le calcul de chemins simples). En revanche, si on souhaite obtenir un chemin optimal, le problème devient *beaucoup* plus difficile.