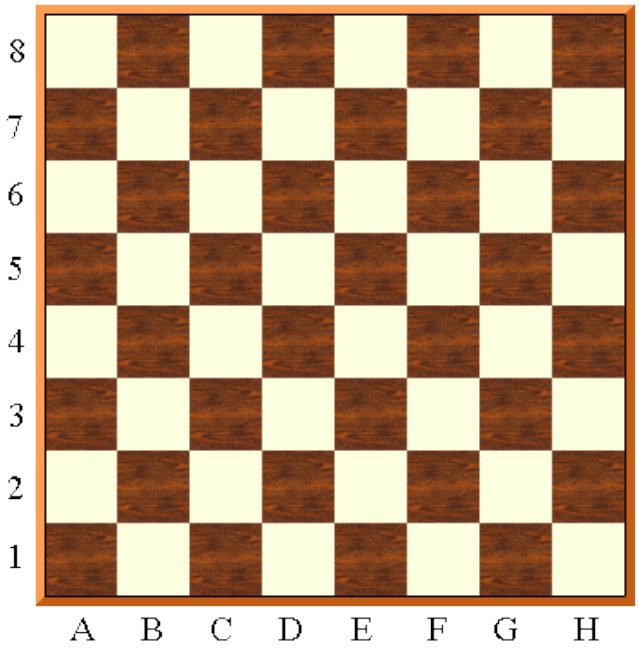
TP : Dames et labyrinthes

I. Le problème des huit dames

I.a. Rappel du problème



Question : Combien de façons existe-t-il de poser huit dames sur un échiquier sans qu'aucune ne soit en prise avec une autre ?

I.b. Fichier source

Commencez par récupérer le fichier source disponible sur le support en ligne du cours. Attention, pour cette séance, nous utiliserons successivement différents fichiers sources.

Le fichier concernant le problème des huit dames contient différentes fonctions :

- Utilisez `tracer_echiquier ()` pour tracer un échiquier vide
- Utilisez `tracer_dame i j` pour dessiner une dame sur la case (i, j)
- Utilisez `effacer_dame i j` pour effacer une dame dessinée sur la case

Il est important de remarquer que ces fonctions sont uniquement des fonctions d'affichage : il s'agit simplement de tracer un rond quelque part ou de l'effacer.

I.c. Représentation des données

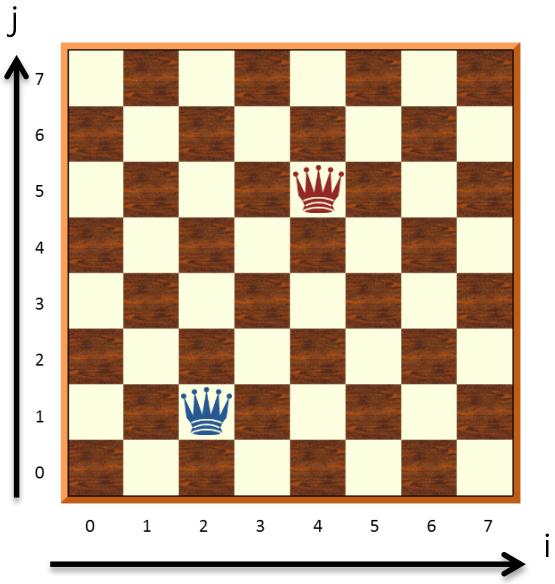
Pour représenter le contenu de l'échiquier, on utilise une variable globale nommée `echiquier`, également initialisée dans le fichier source.

On adopte la représentation suivante :

`echiquier.(i).(j) = true` si et seulement si une dame est présente sur la case (i, j)

Par exemple,

- `echiquier.(2).(2) = false`
- `echiquier.(4).(5) = false`



I.d. Premières fonctions

Pour vous approprier le modèle de données, commencez par écrire les deux fonctions suivantes.

Une fonction `ajouter_dame` :

- prenant en argument deux entiers `i` et `j`
- ne renvoyant rien
- telle que `ajouter_dame i j` ajoute une dame sur l'échiquier sur la case (i, j) et met à jour l'affichage en conséquence

Une fonction `retirer_dame` :

- prenant en argument deux entiers `i` et `j`
- ne renvoyant rien
- telle que `retirer_dame i j` retire une dame sur l'échiquier positionnée sur la case (i, j) et met à jour l'affichage en conséquence

I.e. Position valide

On suppose que quelques dames ont été posées sur l'échiquier sans être en prise.

Imaginez une fonction `position_valide` :

- prenant en argument deux entiers `i` et `j`
- renvoyant un booléen
- telle que `position_valide i j` renvoie `true` si et seulement si on peut poser une dame sur la case (i, j) sans qu'elle soit en prise avec une dame déjà posée.

I.f. Position valide : Version améliorée

Dans toute la suite, on suppose que les dames sont posées sur l'échiquier de gauche à droite :

- On pose d'abord une dame sur la première colonne
- Puis une dame sur la deuxième colonne
- Puis une dame sur la troisième colonne
- etc.

Imaginez une variante `position_valide_gauche_droite` de la fonction précédente prenant en compte cette supposition.

Quel est l'intérêt de cette nouvelle version par rapport à la précédente ?

I.g. Positions autorisées sur la colonne suivante

Imaginez une fonction `positions_valides_sur_la_colonne` :

- prenant en argument un entier `i`
- renvoyant une liste d'entiers
- telle que `positions_valides_sur_la_colonne i` renvoie la liste des positions valides sur la colonne `i`, en fonction des dames posées sur les `i` premières colonnes

Par exemple, `positions_valides_sur_la_colonne 2` renvoie la liste des positions valides sur la troisième colonne (d'indice 2) en fonction des dames posées sur les colonnes 0 et 1.

I.h. Une solution au problème des huit dames

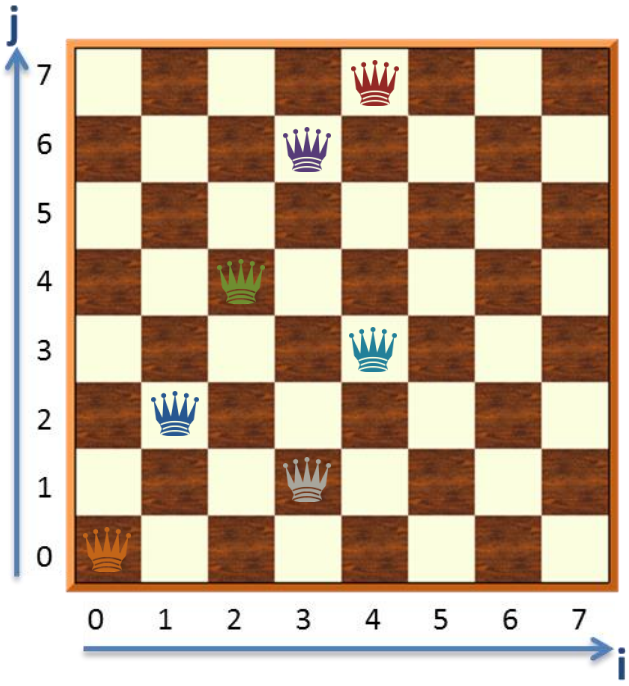
On va encore une fois utiliser une approche récursive pour trouver les solutions de ce problème.

L'idée est la suivante :

1. On suppose que des dames ont été posées sur les `i` premières colonnes
2. On récupère la liste `liste_positions` des positions valides sur la colonne `i`
3. On place une dame en fonction du premier élément de `liste_positions`
4. On reprend le même schéma en supposant que des dames ont été posées sur les (i+1) premières colonnes
5. Si l'on n'a pas trouvé de solution en utilisant le premier élément de `liste_positions`, on recommence nos essais avec le deuxième élément de cette liste.

Vous trouverez un exemple détaillé illustrant cette méthode sur la page suivante.

Exemple :



- On commence sur la colonne 0
- Cases valides restantes : (0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (0,6) (0,7)
- On pose une dame sur la case (0,0)
- On passe à la colonne 1
- Cases valides restantes : (1,2) (1,3) (1,4) (1,5) (1,6) (1,7)
- On pose une dame sur la case (1,2)
- On passe à la colonne 2
- Cases valides restantes : (2,4) (2,5) (2,6) (2,7)
- On pose une dame sur la case (2,4)
- On passe à la colonne 3
- Cases valides restantes : (3,1) (3,6) (3,7)
- On pose une dame sur la case (3,1)
- On passe à la colonne 4
- Cases valides restantes : (4,3) (4,7)
- On pose une dame sur la case (4,3)
- On passe à la colonne 5
- Il n'y a pas de cases valides sur la colonne 5 dans la configuration actuelle
- On revient à la colonne 4
- On retire la dame de la case (4,3)
- Cases valides restantes : ~~(4,3)~~ (4,7)
- On pose une dame sur la case (4,7)
- On passe à la colonne 5
- Il n'y a pas de cases valides sur la colonne 5 dans la configuration actuelle
- On revient à la colonne 4
- On retire la dame de la case (4,7)
- Cases valides restantes : ~~(4,3)~~ ~~(4,7)~~
- Aucune des cases valides sur la colonne 4 dans la configuration actuelle n'aboutit à une solution
- On revient à la colonne 3
- On retire la dame de la case (3,1)
- Cases valides restantes : ~~(3,1)~~ (3,6) (3,7)
- On pose une dame sur la case (3,6)
- On passe à la colonne 4
- etc.

En vous inspirant de l'exemple précédent, écrivez une fonction `trouver_une_solution` :

- ne prenant aucun argument
- renvoyant un booléen
- telle que `trouver_une_solution ()` renvoie `true` si et seulement si une solution a été trouvée pour le problème des huit dames (le cas échéant, l'échiquier doit représenter la solution découverte).

I.i. Résolution du problème des huit dames

Pour finir, imaginez une fonction `resoudre_huit_dames` :

- ne prenant aucun argument
- renvoyant un entier
- telle que `resoudre_huit_dames ()` cherche toutes les solutions au problème des huit dames et renvoie le nombre de solutions trouvées.

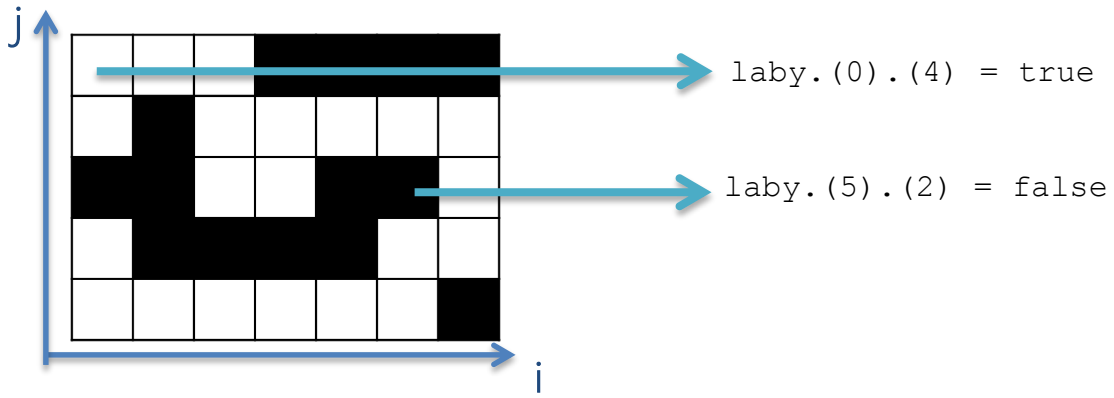
II. Labyrinthes simples

II.a. Représentation choisie

Pour se familiariser avec les labyrinthes, on va utiliser la représentation suivante :

- Un labyrinthe est représenté par un vecteur de vecteurs de booléens (`bool array array`)
- Dans chaque case, le booléen répond à la question "*Cette case est-elle libre ?*"

Attention ! Suite à vos remarques, la convention retenue n'est pas celle proposée lors du dernier cours mais son opposé. Pensez à adapter votre code en conséquence !

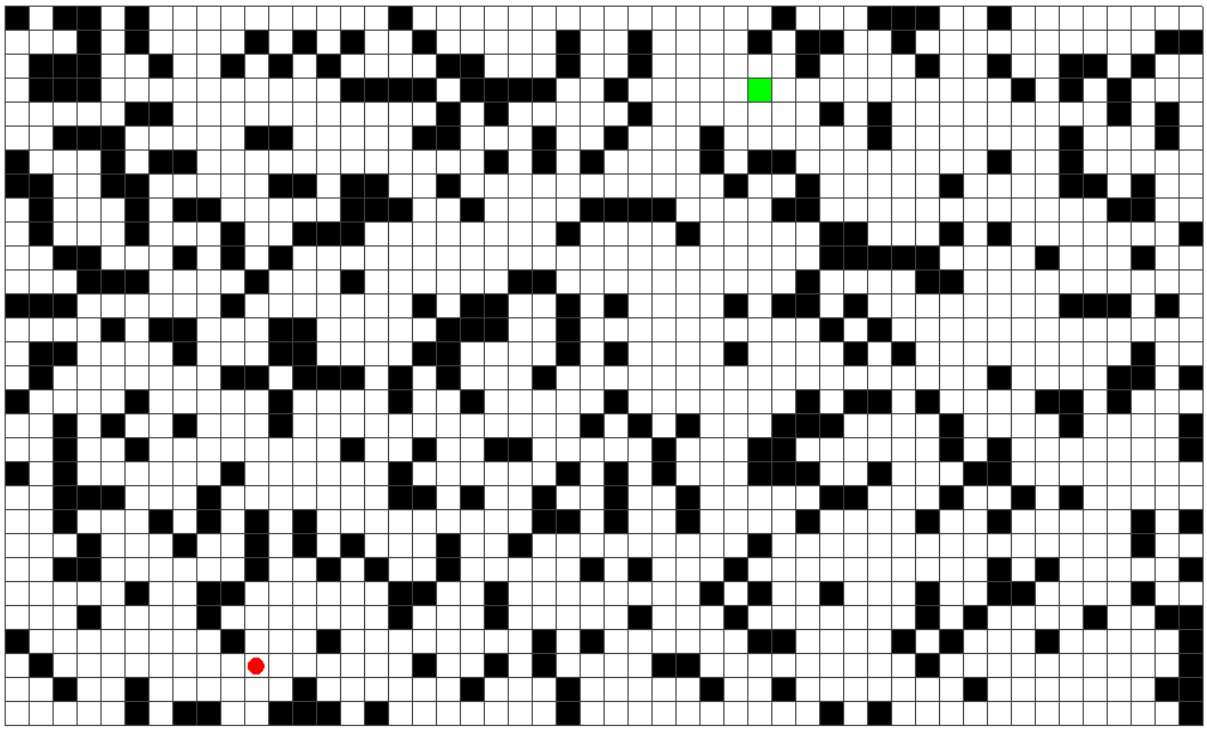


II.b. Fichier source

Récupérez le fichier source disponible sur le support en ligne du cours (*labyrinthes simples*).

Ce fichier contient de nombreuses fonctions, dont vous pourrez trouver des exemples à la fin du fichier :

- `let l = creer_laby 20 10` crée un nouveau labyrinthe de 20 cases de large et de 10 cases de haut, et enregistre ce labyrinthe dans la variable `l`.
- `largeur l` et `hauteur l` renvoient les dimensions du labyrinthe `l`
- `tracer_laby l` trace le labyrinthe représenté par `l`
- `ajouter_murs_aleatoires l 35` ne fait rien pour le moment : vous modifierez le code de cette fonction pour ajouter des murs dans notre labyrinthe
- `resoudre_laby l` vous propose de résoudre le labyrinthe représenté par `l` :
    - Sélectionnez la fenêtre graphique d'Emacs
    - Observer votre labyrinthe
    - Vous êtes le point rouge, la sortie est représentée par un carré vert
    - Appuyez sur la barre espace quand vous êtes prêts
    - Utilisez les touches Z, Q, S et D pour aller vous déplacer respectivement vers le haut, la gauche, le bas et la droite.
    - Appuyer sur la touche L pour abandonner
    - Lorsque le point rouge atteint la sortie, vous avez gagné !



Vous remarquerez néanmoins que pour l'instant, le joueur et la sortie étant toujours au même endroit, le jeu manque un peu d'intérêt.

L'explication est simple : il y a des "trous" dans le code, que vous allez devoir remplir.

II.c. Compléter le code

Pour rendre les choses plus intéressantes, vous allez ajouter des murs à votre labyrinthe.

Trouver une case libre

Pour cela, commencez par compléter le code de la fonction récursive `trouver_case_libre` :

- prenant en argument un labyrinthe (de type `bool array array`)
- renvoyant un couple d'entiers
- telle que `trouver_case_libre laby` renvoie, si elle existe, l'une des cases libres du labyrinthe `laby`.

Rappel : Pour tirer des nombres au hasard, commencez par exécuter la ligne de code `Random.self_init()` (directement dans votre programme, sans être à l'intérieur d'une fonction). Il vous suffit ensuite d'utiliser `Random.int n` pour obtenir un entier tiré au hasard entre `0` et `n-1`.

Remarque : cette fonction étant utilisée dans la fonction `resoudre_laby`, évaluez à nouveau toutes les fonctions pour constater les effets de vos modifications sur le mode résolution.

Ajouter des murs

Complétez ensuite le code de la fonction `ajouter_murs_aleatoires` :

- prenant en argument un labyrinthe (de type `bool array array`) et un pourcentage (sous la forme d'un entier compris entre 0 et 100)
- ne renvoyant rien
- telle que `ajouter_murs_aleatoires laby pourcentage_de_murs` ajoute au labyrinthe `laby` un certain nombre de murs placés aléatoirement afin d'obtenir `pourcentage_de_murs` % de murs dans le labyrinthe.

Lister les voisins d'une case

Complétez enfin le code de la fonction `voisins` :

- prenant en argument un labyrinthe et deux coordonnées `i` et `j`
- renvoyant une liste des coordonnées (c'est-à-dire de couples d'entiers)
- telle que `voisins laby i j` ajoute renvoit la liste des cases accessibles dans le labyrinthe `laby` depuis la case (i, j)

II.d. Vers des labyrinthes plus intéressants

Faites des essais en créant des labyrinthes et en les remplissant avec différents pourcentages de murs : que se passe-t-il si le pourcentage est très faible ? Et si l'on augmente trop ce pourcentage ?

Quelle solution proposeriez-vous pour obtenir de meilleurs résultats ?

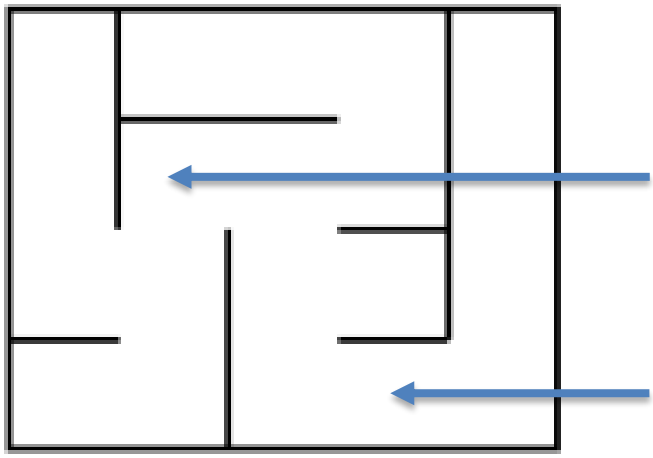
III. Labyrinthes avancés

III.a. Représentation choisie

Pour obtenir des labyrinthes à la fois plus esthétiques et plus intéressants, nous allons adopter une nouvelle représentation :

- Un labyrinthe est représenté par un vecteur de vecteurs de cases (`case array array`)
- Le type `case` est un type créé pour l'occasion afin de correspondre à nos besoins
- Dans chaque case, on stocke ainsi deux booléens qui répondend aux questions
  - *"Peut-on accéder à la case située juste à gauche ?"*
  - *"Peut-on accéder à la case située juste en dessous ?"*

Attention ! Suite à vos remarques, la convention retenue n'est pas celle proposée lors du dernier cours mais son opposé. Pensez à adapter votre code en conséquence !



III.b. Fichier source

Récupérez le fichier source disponible sur le support en ligne du cours (*labyrinthes avancés*).

Comme pour les labyrinthes simples, ce fichier contient de nombreuses fonctions, dont vous pourrez trouver des exemples à la fin du fichier :

- `let l = creer_laby 20 10` crée un nouveau labyrinthe de 20 cases de large et de 10 cases de haut, et enregistre ce labyrinthe dans la variable `l`.
- `largeur l` et `hauteur l` renvoient les dimensions du labyrinthe `l`
- `tracer_laby l` trace le labyrinthe représenté par `l`
- `ajouter_murs_aleatoires l 35` ne fait rien pour le moment : vous modifierez le code de cette fonction pour ajouter des murs dans notre labyrinthe
- `resoudre_laby l` vous propose de résoudre le labyrinthe représenté par `l`

Encore une fois, il y a des "trous" dans le code, et vous allez devoir remplir.

III.c. Compléter le code

Pour rendre les choses plus intéressantes, vous allez ajouter des murs à votre labyrinthe.

Trouver une case libre

Commencez par compléter le code de la fonction `trouver_case_mur_gauche_libre` :

- prenant en argument un labyrinthe (de type `case array array`)
- renvoyant un couple d'entiers
- telle que `trouver_case_mur_gauche_libre laby` renvoie, si elle existe, l'une des cases libres du labyrinthe `laby` ne possédant pas de mur gauche

Complétez sur le même modèle la fonction `trouver_case_mur_bas_libre`.

Remarque : cette fonction étant utilisée dans la fonction `resoudre_laby`, évaluez à nouveau toutes les fonctions pour constater les effets de vos modifications sur le mode résolution.

Ajouter des murs

Complétez ensuite le code de la fonction `ajouter_murs_aleatoires` :

- prenant en argument un labyrinthe (de type `bool array array`) et un pourcentage (sous la forme d'un entier compris entre 0 et 100)
- ne renvoyant rien
- telle que `ajouter_murs_aleatoires laby pourcentage_de_murs` ajoute au labyrinthe `laby` un certain nombre de murs placés aléatoirement afin d'obtenir `pourcentage_de_murs` % de murs dans le labyrinthe.

Lister les voisins d'une case

Complétez enfin le code de la fonction `voisins` :

- prenant en argument un labyrinthe et deux coordonnées `i` et `j`
- renvoyant une liste des coordonnées (c'est-à-dire de couples d'entiers)
- telle que `voisins laby i j` ajoute renvoit la liste des cases accessibles dans le labyrinthe `laby` depuis la case (i, j)

Des labyrinthes encore imparfaits

Prenez le temps de tester vos fonctions et tentez de résoudre les labyrinthes générés avec différentes densités de murs.

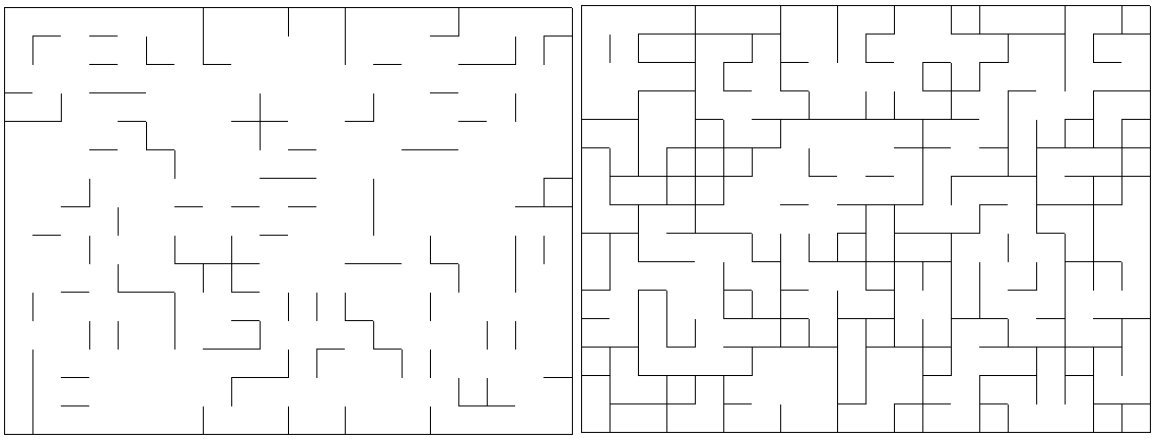
Que constatez-vous ?

IV. Labyrinthes parfaits

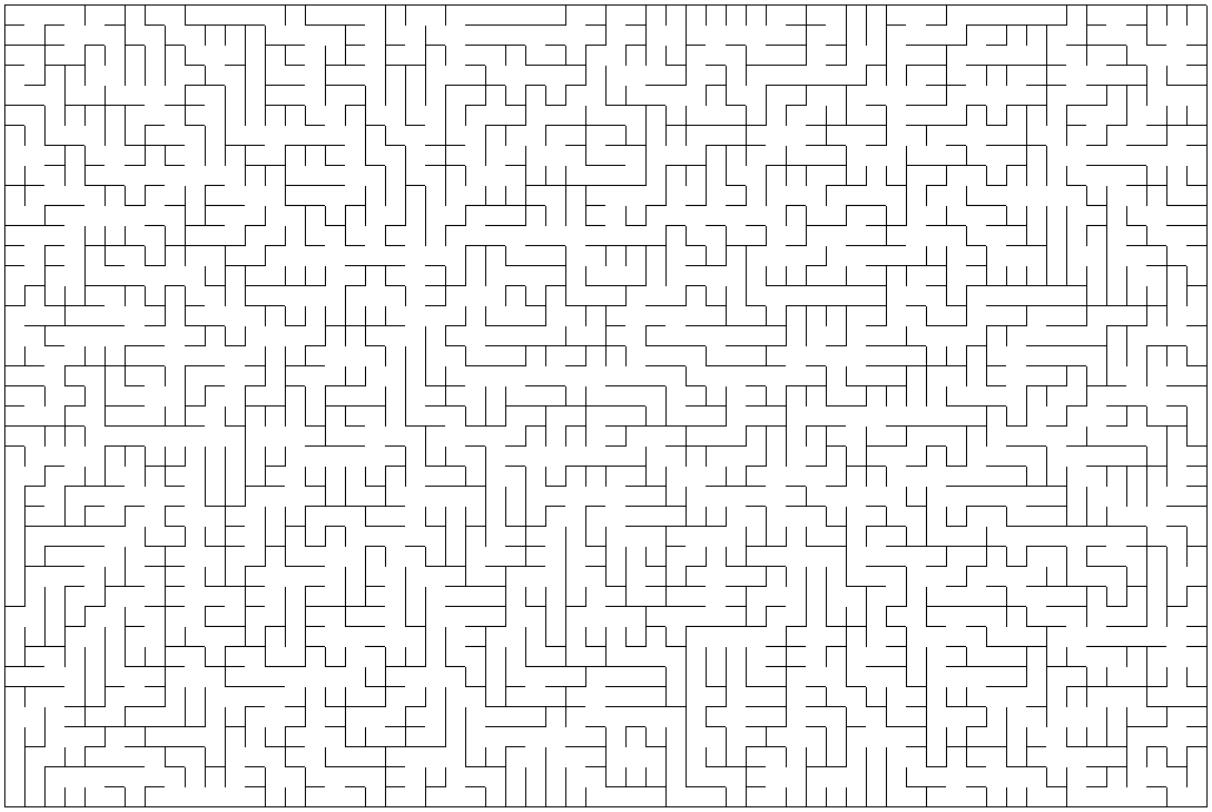
IV.a. La voie de la perfection

Les exemples ci-dessous illustrent les deux tendances que l'on peut observer en jouant sur la densité de murs dans nos labyrinthes :

- Avec une faible densité de murs, le labyrinthe est trivialement simple
- Avec une forte densité de murs, le labyrinthe est impossible à résoudre



L'objectif de cette dernière partie est de construire des labyrinthes parfaits, c'est-à-dire tel que pour tout couple de cases, il existe un unique chemin entre ces deux cases.



IV.b. Quelques petites fonctions sur les listes

Pour créer ce type de labyrinthes, nous aurons besoin de plusieurs fonctions auxiliaires, dont l'implémentation vous permettra de manipuler à nouveau les listes.

Remarque : pour chacune des fonctions demandées, il est recommandé de tester son bon fonctionnement sur quelques exemples avant de passer à la question suivante.

Insérer si nouveau

Commencez par imaginer une fonction `inserer_si_nouveau` :

- prenant en argument une liste `l1` et un élément `x`
- renvoyant une liste
- telle que `inserer_si_nouveau l1 x` renvoie une liste `l2` égale à `l1` si `x` apparaissait déjà dans `l1`, et à `x :: l1` sinon.

Remarque : Vous pouvez utiliser `List.mem x l1` pour tester l'appartenance de `x` à `l1`.

Utilisez cette fonction pour implémenter la fonction `inserer_si_nouveaux` :

- prenant en argument deux listes `l1` et `l2`
- renvoyant une liste
- telle que `inserer_si_nouveau l1 l2` renvoie une liste `l3` qui contient les éléments de `l1` ainsi que les éléments de `l2` qui n'apparaissaient pas dans `l1`.

Retirer certains éléments

Ecrivez une fonction `retirer_elements_interdits` :

- prenant en argument une liste `source` et une liste `elements_interdits`
- renvoyant une liste
- telle que `retirer_elements_interdits source elements_interdits` renvoie une liste composée des éléments de la liste `source` n'apparaissant pas dans la liste `elements_interdits`.

Identifier les éléments communs à deux listes

Ecrivez une fonction `elements_communs_a_deux_listes` :

- prenant en argument deux listes `liste1` et `liste2`
- renvoyant une liste
- telle que `elements_communs_a_deux_listes liste1 liste2` renvoie une liste composée des éléments apparaissant à la fois dans `liste1` et dans `liste2`.

Choisir un élément au hasard dans une liste

Ecrivez une fonction `element_au_hasard` :

- prenant en argument une liste `l`
- renvoyant un élément `x`
- telle que `element_au_hasard l` renvoie un élément `x` choisi au hasard parmi les éléments de `l`.

Remarque : Vous pouvez utiliser `List.length l` pour obtenir la longueur de la liste l, et `List.nth l n` pour obtenir le $(n-1)^{ème}$ élément de `l`.

IV.c. Fonctions auxiliaires sur les labyrinthes

Vous aurez également besoin d'un certain nombre de fonctions auxiliaires opérant sur nos labyrinthes pour créer des labyrinthes parfaits.

Choisir une case au hasard

Imaginez une fonction `case_au_hasard` :

- prenant en argument un labyrinthe
- renvoyant un couple de coordonées
- telle que `case_au_hasard laby` renvoie les coordonnées d'une case choisie au hasard dans le labyrinthe `laby`

Ajouter des murs partout

Imaginez une fonction `ajouter_des_murs_partout` :

- prenant en argument un labyrinthe
- ne renvoyant rien
- telle que `ajouter_des_murs_partout laby` ajoute des murs partout dans le labyrinthe `laby` (de sorte que toute case de `laby` est entourée de quatre murs)

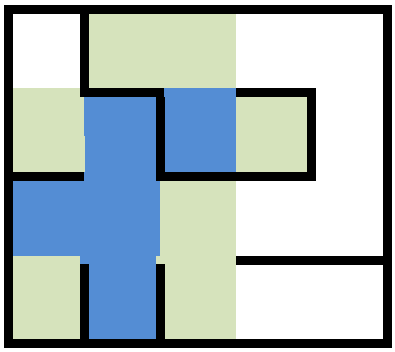
Retirer un mur

Imaginez une fonction `retirer_un_mur` :

- prenant en argument un labyrinthe, et les coordonnées de deux cases
- ne renvoyant rien
- telle que `retirer_un_mur laby (i1,j1) (i2,j2)` retire dans `laby` le mur situé entre les cases `(i1,j1)` et `(i2,j2)`

Trouver les "voisins potentiels"

Un voisin potentiel d'une case (i,j) est une case située à proximité directe de la case (i,j). Contrairement à la notion de "voisins" évoquée au III.c, la notion de "voisins potentiels" ne tient pas compte des murs entre les cases.



Il sera utile de pouvoir déterminer l'ensemble des voisins potentiels d'un ensemble de cases données.

Par exemple, sur l'illustration ci-contre, on a représenté en vert l'ensemble des voisins potentiels des cases représentées en blue.

Ecrivez une fonction `voisins_potentiels` :

- prenant en argument un labyrinthe et un couple de coordonnées
- renvoyant une liste de coordonées
- telle que `voisins_potentiels laby (i,j)` renvoie la liste des voisins potentiels de la case (i,j) dans le labyrinthe `laby`.

Ecrivez ensuite une fonction `trouver_tous_les_voisins_potentiels` :

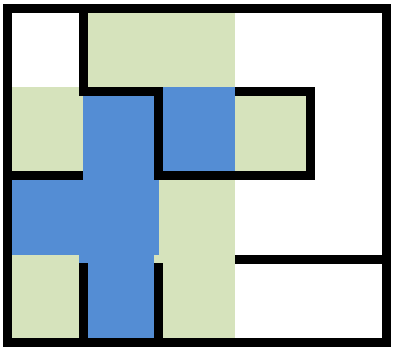
- prenant en argument un labyrinthe et une liste de couples de coordonnées
- renvoyant une liste de coordonées
- telle que `voisins_potentiels laby liste_cases` renvoie la liste des voisins potentiels des cases listées par `liste_cases`.

Indice 1 : Pensez à utiliser `inserer_si_nouveaux` !

Indice 2 : Pour récupérer la tête d'une liste de coordonnées, utilisez la syntaxe

```
let (i,j) = List.hd liste_de_coordonnees in
```

Référent d'un voisin potentiel ★



Si (i,j) est l'un des voisins potentiels d'une liste de cases l_{cases}, on appelle "référent de ce voisin potentiel" toute case de la liste l_{cases} située à proximité directe de (i,j).

Par exemple, sur l'exemple ci-contre, `(1,0)` est l'unique référent possible pour le voisin potentiel `(2,0)`.

Par contre, pour le voisin potentiel `(2,1)`, il y a deux référents possibles : `(1,1)` et `(2,2)`

Imaginez une fonction `referent_du_voisin_potentiel` :

- prenant en argument un labyrinthe, une liste de couples de coordonnées, et une paire de coordonnées (correspondant à un voisin potentiel)
- renvoyant une paire de coordonées
- telle que `referent_du_voisin_potentiel laby liste_cases (i,j)` renvoie les coordonnées d'un des référents possibles parmi `liste_cases` du voisin potentiel `(i,j)`.

Indice : Lister toutes les référents possibles et faites appel à `element_au_hasard` !

IV.d. Création de labyrinthes parfaits

Principe

Pour obtenir un labyrinthe parfait, on suit la méthode suivante :

1) On construit d'abord un labyrinthe vide
2) On ajoute des murs partout
3) On choisit une case au hasard, qu'on déclare visitée
4) Tant que toutes les cases n'ont pas été visitées :
   - On détermine les voisins potentiels des cases visitées
   - On choisit l'un de ces voisins potentiels
   - On détruit l'un des murs séparant ce voisin des cases visitées (c'est pour cela que l'on a besoin de l'un des référents de ce voisin potentiel)
5) Une fois toutes les cases visitées, il n'y a plus rien à faire, notre labyrinthe est parfait.

Implémentation

En utilisant les fonctions définies précédemment, écrivez une fonction `creer_laby_parfait` :

- prenant en argument deux entiers
- renvoyant un labyrinthe
- telle que `creer_laby_parfait l h` crée un labyrinthe parfait de largeur `l` et de hauteur `h`.

Concours de vitesse

Une fois achevée votre fonction de génération, utiliser la fonction `resoudre_laby` pour tester votre rapidité sur ces nouveaux labyrinthes !