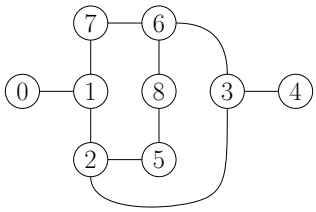
Backtracking

1 Chemins hamiltoniens

Définition 1.1

Soit $G = (V, E)$ un graphe non-orienté à n sommets.
- Un *chemin hamiltonien* de G est un chemin élémentaire de longueur $n - 1$. Autrement dit, c'est un chemin passant une et une seule fois par chaque sommet du graphe.
- Un *cycle hamiltonien* de G est un cycle élémentaire de longueur n. Autrement dit, c'est un cycle $x = x_0, x_1, \ldots, x_{n-1}, x_n = x$ où tous les x_i sont distincts, à part x_0 et x_n.

Déterminer si un graphe possède un chemin, ou un cycle, hamiltonien est un problème « difficile » (NP-complet, nous le verrons l'année prochaine). Cependant, il se prête bien au *backtracking*, et avec la bonne heuristique on peut traiter des graphes relativement gros, hors cas pathologiques.



Le graphe G_0.

1. Combien de chemins hamiltoniens le graphe G_0 possède-t-il ? Combien de cycles hamiltoniens ?

Pour représenter les graphes, on utilisera le type suivant :

```
type graphe = {nb_sommets : int; voisins : int -> int list}
```

On supposera toujours les sommets d'un graphe `g` numérotés de 0 à `g.nb_sommets - 1`.

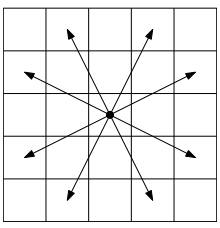
2. Écrire une fonction `hamiltonien_depuis` qui prend en entrée un graphe `g` et un indice de sommet (valide) `x0` et renvoie :
   - `None` s'il n'existe pas de chemin hamiltonien dans G commençant au sommet x_0 ;
   - `Some ordre`, où `ordre` est un `int array` de longueur n codant un chemin hamiltonien à partir de x_0, s'il en existe un. On codera le chemin de la façon suivante : si le chemin est $x_0, x_1, \ldots, x_{n-1}$, alors la case x_i du tableau ordre contiendra l'entier i.

   On pourra :
   - utiliser une exception `Trouve of int array`, qu'on lèvera dès que l'on trouve un chemin hamiltonien ;
   - initialiser `ordre` à `[|-1; -1; ...; -1|]` et le remplir au fur et à mesure. Le tableau `ordre` fera alors double emploi, puisqu'il indiquera si le sommet i est, ou non, encore disponible (suivant que `ordre.(i)` vaut `-1` ou pas).

```
hamiltonien_depuis : graphe -> int -> int array option
```

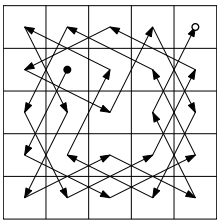
2 Parcours du cavalier

Aux échecs, le cavalier est une pièce qui peut se déplacer de deux case suivant un axe et une suivant l'autre :



Les huit déplacements possibles pour un cavalier.

Le problème du *parcours du cavalier* consiste à faire parcourir à un cavalier toutes les cases d'un échiquier $n \times m$ sans jamais repasser deux fois par la même. On peut éventuellement imposer de plus qu'on finisse sur la case dont on est parti.



Un parcours du cavalier sur un échiquier 5×5.

1. Écrire une fonction `graphe_cavalier` prenant en entrée deux entiers n et m et codant le problème du cavalier sur un échiquier $n \times m$ sous forme d'un graphe, dans lequel on pourra chercher un chemin hamiltonien. On numérotera les cases du graphes de 0 à $m-1$ sur la première ligne, m à $2m-1$ sur la deuxième ligne et ainsi de suite.

```
graphe_cavalier : int -> int -> graphe
```

2. Écrire une fonction `affiche_parcours_cavalier` prenant en entrée deux entiers n et m, ainsi qu'un couple (x, y) indiquant les coordonnées d'une case, et affichant le parcours sous la forme ci-dessous :

```
# affiche_parcours_cavalier 5 6 (0, 1);;
 11   0  17  24   9   6
 18  25  10   7  22  29
  1  12  23  16   5   8
 26  19  14   3  28  21
 13   2  27  20  15   4
```

   S'il n'existe pas de parcours depuis la case demandée, on l'indiquera par un message : c'est par exemple le cas dans un échiquier 5×5 si l'on part de la case $(0, 1)$.

```
affiche_parcours_cavalier : int -> int -> (int * int) -> unit
```

3. Jusqu'à quelle valeur de n la recherche d'un parcours sur un échiquier $n \times n$ (à partir de la case $(0, 0)$, disons) prend-elle un temps raisonnable ?

3 Heuristique

Pour améliorer la recherche, on propose d'utiliser l'heuristique suivante : on ordonne les enfants d'un nœud (dans l'arbre de recherche sous-jacent) par nombre croissant de voisins non visités. Autrement dit, si l'on se trouve sur un sommet du graphe et que l'on souhaite étendre le chemin, on commence par essayer le voisin ayant le moins de voisins non visités.

Écrire une fonction `hamiltonien_opt_depuis` implémentant cette heuristique. Cette fonction aura comme effet secondaire d'afficher le nombre de nœuds de l'arbre de recherche explorés.

```
hamiltonien_opt_depuis : graphe -> int -> int array option
```

1. Utiliser cette heuristique pour trouver un parcours du cavalier sur un échiquier 200×200 (*afficher* ce parcours n'est pas une très bonne idée...). Que constate-t-on ?

   Cette heuristique très simple permet de trouver un chemin hamiltonien en temps linéaire dans de nombreuses classes de graphes (possédant un tel chemin, bien sûr). Les graphes du cavalier sont l'une de ces classes...

4 Tableaux auto-référents

Un tableau t d'entiers de taille n (indices allant de 0 à $n-1$) est dit *auto-référent* si, pour chaque i entre 0 et $n-1$, le nombre d'occurrences de i dans t est égal à t_i. Par exemple :

indices	0	1	2	3
`t`	2	0	2	0
occurrences	2	0	2	0

1. Écrire une fonction énumérant toutes les solutions de taille n.

```
auto_referents : int -> int array list
```

2. Votre fonction marche-t-elle pour $n = 8$? 10 ? 20 ? 50 ? 80 ? Faire en sorte que ce soit le cas...
3. Que peut-on conjecturer ? Démontrer votre conjecture. emphJe n'ai pas vraiment essayé, mais ça n'a pas l'air évident.