TP n°3 - Algorithme A* en OCaml

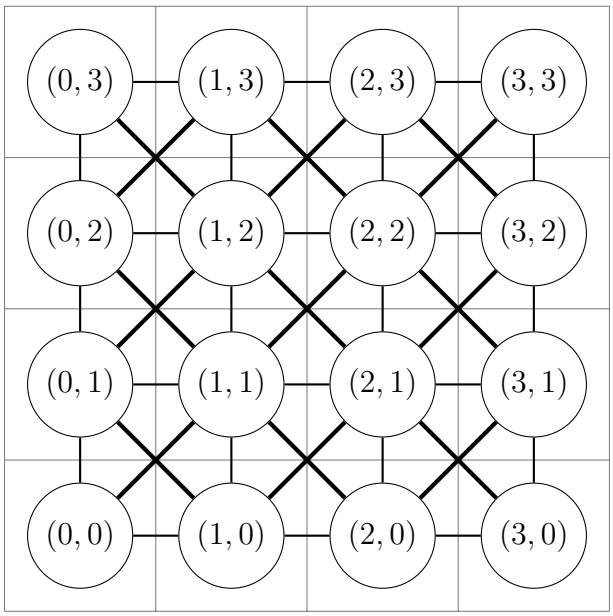
Notions abordées

- représentation de graphes orientés en OCaml
- algorithme de Dijkstra
- algorithme A*, heuristique, heuristique admissible, heuristique monotone
- preuve de correction par invariant

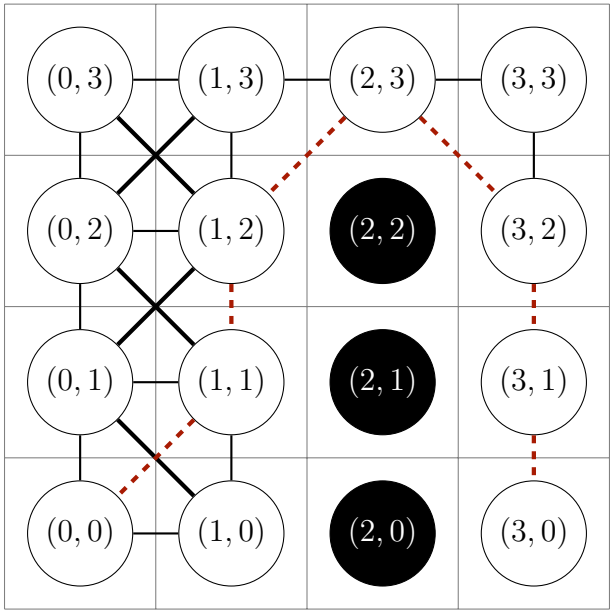
Exercice 1 : Implémentation en OCaml de l'algorithme A*

L'algorithme A* est un algorithme de recherche de plus court chemin dans un graphe. Cet algorithme peut être vu comme une variante de l'algorithme de Dijkstra dans lequel on modifie le critère de sélection du prochain sommet à visiter.

Graphes étudiés. Afin d'illustrer cet algorithme, on étudie une famille de graphes particuliers. Pour une largeur $w \in \mathbb{N}^*$ et une hauteur $h \in \mathbb{N}^*$, on considère le graphe non orienté pondéré $G = (S, A, c)$ qui représente une grille à h lignes et w colonnes. Il est défini par $S = [\![0, w[\![\times [\![0, h[\![$, $A = \{\{(x, y), (x', y')\} \mid (x, y) \in S, (x', y') \in S, (x, y) \neq (x', y'), |x - x'| \leqslant 1, |y - y'| \leqslant 1\}$ et la pondération $c : A \to \mathbb{R}^+$, $\{(x, y), (x', y')\} \mapsto \sqrt{(x - x')^2 + (y - y')^2}$. Finalement le graphe est celui des cases d'une portion rectangulaire du plan dans lequel une case est reliées aux 8 cases adjacentes (8 au plus, au bord les cases peuvent avoir moins de voisins), et où la distance entre deux telles cases est la distance euclidienne entre leurs centres.
On représente ci-dessous le graphe pour $(w, h) = (4, 4)$. Les arêtes de coût $\sqrt{2}$ (les "diagonales") sont tracées en trait épais, tandis que les arêtes de coût 1 (les "verticales" et les "horizontales") sont tracées en trait fin.



Afin d'illustrer les différences entre l'algorithme A^* et l'algorithme de Dijkstra, on suppose de plus que certaines cases, certains sommets du graphe donc, sont en fait inaccessibles, c'est-à-dire qu'il n'existe pas d'arête incidentes à ces sommets. De tels sommets peuvent représenter, par exemple, un mur. Dans le graphe ci-dessous par exemple, les sommets $(2,0)$, $(2,1)$ et $(2,2)$ forment un mur (de tels sommets seront représentés en blanc sur noir et pas en noir sur blanc). De plus on a représenté le plus court chemin entre les sommets $(0,3)$ et $(3,3)$ au moyen d'une ligne rouge pointillée.



Dans la suite on appellera **monde** la donnée d'un tel graphe, qui est donc caractérisé par sa largeur, sa hauteur et son ensemble de cases inaccessibles.

`compagnon.ml`. Le fichier `compagnon.ml` accompagnant ce TP fournit les outils suivants.

- Un module `Floatb` permettant la manipulation de nombres de $\mathbb{R}\cup\{+\infty\}$. Un élément `Floatb.F(f)` où `f` est de type `float` représente le réel encodé par `f` tandis que l'élément `Floatb.Infini` de type `Floatb.t` représente $+\infty$. On trouvera dans ce module des fonctions pour afficher, comparer et additionner de tels nombres.
- Des définitions de types. Le type `world` pour représenter un monde, défini par sa largeur (`w`), sa hauteur (`h`) et la liste des cases non accessibles (`obstacles`).

```
type world =
  {
    w         : int;
    h         : int;
    obstacles : cell list
  }
```

Le type `cell` pour représenter les cases du monde.

```
type cell = int * int            (* x, y *)
```

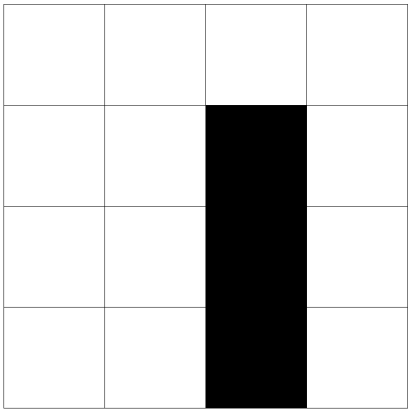
De plus, en vue de stocker les cases du monde dans un tableau dans la suite du TP, on se propose d'utiliser la linéarisation classique des couples d'entiers en un entier au moyen de la fonction $(x,y)\mapsto xw+y$, où w est la largeur du monde. On définit alors le type `cell_l` pour les cellules identifiées par un entier et non par un couple d'entiers.

```
type cell_l = int  (* représente une version linéarisée des cellules *)
```

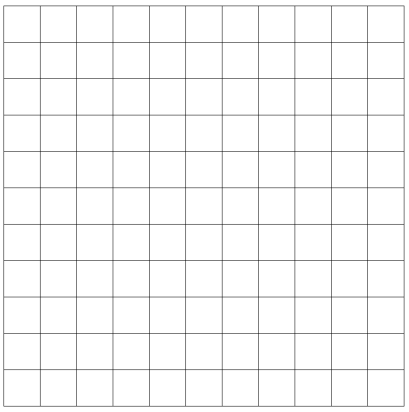
On fournit les fonctions de linéarisation et délinéarisation : `lin : world -> cell -> cell_l` et `delin : world -> cell_l -> cell` réciproques l'une de l'autre.

- Des exemples de mondes prédéfinis : `w_ex_0` à `w_ex_3` (représentés ci-après).
- Une fonction `voisins : world -> cell -> (cell * float) list` prenant en arguments un monde, une case s et retournant la liste des cases adjacentes à s ainsi que leur distance à s (1 ou $\sqrt{2}$).
- Une fonction `draw_world : algo_step option -> world -> unit` permettant l'affichage graphique d'un monde. Cette fonction utilise le module `Graphics`. On trouvera ci-dessous des exemples de rendus graphiques de cette fonction. Cette fonction accepte en premier argument une structure de type `algo_step` qui a vocation à être utilisée pendant l'exécution de l'algorithme de Dijkstra ou A* afin de passer à la fonction diverses grandeurs manipulées par ces algorithmes. On ne détaille pas ici les champs de cette structure, on trouvera, en temps voulu, dans le fichier `compagnon.ml` une description de ces différents champs. Dans le cas où l'on souhaite simplement afficher un monde, sans trace de l'algorithme, il suffit de passer `None` comme premier argument à cette fonction.

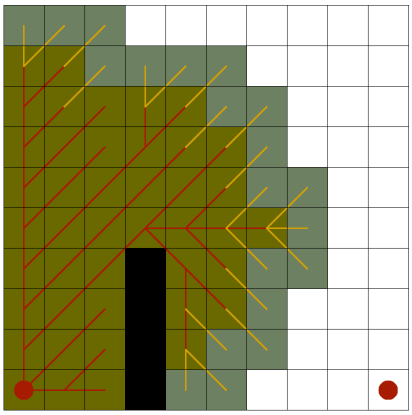
Exemples fournis. On fournit ci-dessous l'affichage graphique des quatre mondes d'exemples fournis dans `compagnon.ml`. L'affichage du monde `w_ex_2` comprend aussi une représentation d'un état lors d'une exécution de l'algorithme de Dijkstra. Cet affichage a été produit en appelant la fonction `draw_world` avec un premier argument différent de `None`.



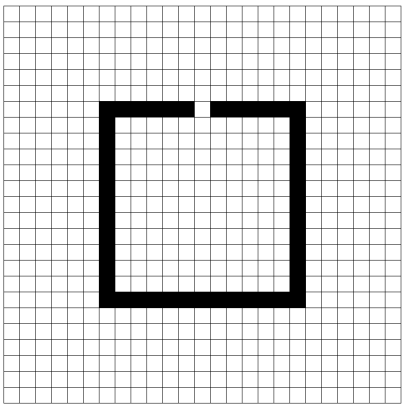
Monde `w_ex_0`



Monde `w_ex_1`



Monde `w_ex_2`



Monde `w_ex_3`

1. Rappels : l'algorithme de Dijkstra

On redonne ci-dessous l'algorithme de Dijkstra.

Algorithme 1 : Dijkstra

Entrée : Un graphe pondéré $G = (S, A, c)$ où $c : A \to \mathbb{R}^+$, une source $s \in S$
Sortie : Une table indexée par S donnant la distance depuis s de chaque sommet de G

```
1  Initialiser μ[u] à +∞ pour chaque u ∈ S ;
2  Initialiser π[u] à ?? pour chaque u ∈ S ;
3  μ[s] ← 0 ;                          // La source est à distance 0 d'elle-même
4  π[s] ← s ;                          // La source est la racine de l'arborescence
5  todo ← {s} ;                        // La source est le premier sommet à traiter
6  Procedure Relâcher((u, v)) :
       Entrée : u et v deux sommets voisins dans G ;
7      si μ[u] + c((u, v)) < μ[v] alors
8          μ[v] ← μ[u] + c((u, v)) ;
9          π[v] ← u ;
10         Ajouter v à todo ;
11 tant que todo ≠ ∅ faire
12     Soit u ∈ todo minimisant μ[u] ;
13     pour tout v ∈ succ(u) faire
14         Relâcher(u, v)
15     Ôter u de todo ;
16 retourner μ
```

On se propose d'implémenter l'algorithme de Dijkstra. Les différentes grandeurs entrant en jeu dans l'algorithme devront être implémentées au moyen des types suivants :

- le tableau μ sera un tableau de type `Floatb.t array` de dimension wh ;
- le tableau π sera un tableau de type `int option array` de dimension wh ;
- l'ensemble todo sera implémenté au moyen d'une référence vers une liste : `cell_l list ref` ;
- on maintiendra de plus une liste finis des sommets déjà traités, implémentée au moyen d'une référence vers une liste : `cell_l list ref`. Cette liste, bien qu'inutile pour l'algorithme de Dijkstra, sera utile pour la représentation graphique.

L'implémentation proposée ici ne met pas en œuvre de file de priorité pour gérer todo. On décide en effet de se concentrer ici sur l'algorithme plutôt que sur les structures de données.

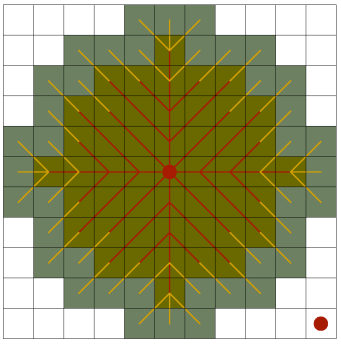
Q. 1 L'algorithme de Dijkstra est ici présenté pour calculer distances de la sources à tous les sommets. Rappeler comment adapter cet algorithme pour calculer seulement la distance de la source s à un sommet donné t.

Q. 2 Implémenter une fonction `dijkstra : world -> cell -> cell -> Floatb.t` prenant en arguments un monde, un sommet de départ et un sommet d'arrivée et calculant à l'aide de l'algorithme de Dijkstra la distance entre les deux ♣. On pourra s'aider des deux fonctions intermédiaires :

- `find_min : int list -> Floatb.t array -> int`, telle que pour `l` à valeurs parmi les indices valides pour `mu`, l'appel `(find_min l mu)` calcule un entier `i` de `l` tel que `mu.(i)` soit minimal ;
- `relax` implémentant l'algorithme Relâcher ci-dessus.

♣. Une telle formulation sous-entend qu'il faut renvoyer $+\infty$ si les sommets ne sont pas reliés dans le graphe, mais cela n'arrivera pas ici car les mondes sont connexes.

Limites de l'algorithme de Dijkstra. Considérons le monde `w_ex_0` et recherchons un chemin du centre du monde (qui est donc la source) vers le coin inférieur droit (qui est donc l'objectif). L'algorithme de Dijkstra va traiter les sommets dans l'ordre induit par les valeurs contenues dans μ, soit par distance au centre croissante, comme le montre le schéma ci-dessous. Il est pourtant évident à l'œil humain qu'il serait avantageux de chercher un chemin optimal en direction du coin.



2. L'algorithme A*

L'algorithme A* est un algorithme de calcul de plus court chemin entre un sommet source $s \in S$ et un sommet objectif $o \in S$. Cet algorithme évite l'écueil de l'algorithme de Dijkstra mis en avant dans la section précédente grâce à une fonction heuristique qui donne une estimation de la distance restant à parcourir pour atteindre l'objectif.

Q. 3 Proposer une fonction d'heuristique **admissible** pour les graphes considérés dans ce TP. En donner une implémentation (qui pourra prendre en paramètre le monde si besoin). *Indication : penser aux oiseaux.*

On redonne ci-dessous l'algorithme $A^\star$.

Algorithme 2 : A*

Entrée : Un graphe pondéré $G = (S, A, c)$ où $c : A \to \mathbb{R}^+$, une source $s \in S$, un objectif $o \in S$, une heuristique $h : S \to \mathbb{R}^+$

Sortie : La distance de s à o

Initialiser $\mu[u]$ à $+\infty$ pour $u \in S$;
Initialiser $\eta[u]$ à $+\infty$ pour $u \in S$;
Initialiser $\pi[u]$ à ? ? pour $u \in S$;
$\mu[s] \leftarrow 0$; // La source est à distance 0 d'elle-même
$\eta[s] \leftarrow \mu[s] + h(s)$; // Estimation de la distance source objectif
$\pi[s] \leftarrow s$; // La source est la racine de l'arborescence
todo $\leftarrow \{s\}$; // La racine est le premier sommet à traiter
Procedure Relâcher$((u, v))$ **:**
    Entrée : u et v deux sommets voisins dans G ;
    si $\mu[u] + c((u, v)) < \mu[v]$ **alors**
        $\mu[v] \leftarrow \mu[u] + c((u, v))$;
        $\eta[v] \leftarrow \mu[v] + h(v)$;
        $\pi[v] \leftarrow u$;
        Ajouter v à todo ;
tant que todo $\neq \emptyset$ **faire**
    Soit $u \in$ todo minimisant $\eta[u]$;
    si $u = o$ **alors**
        retourner $\mu[o]$; // Objectif atteint
    pour tout $v \in$ voisins(u) **faire**
        Relâcher(u, v)
    Ôter u de todo ;
retourner ∞ ; // Objectif inaccessible

On fait les mêmes choix d'implémentation pour l'algorithme A* que pour l'algorithme de Dijkstra, ainsi η sera lui aussi implémenté par un tableau de type `Floatb.t array` de dimension wh. Il est fortement recommandé de commencer par copier-coller le code.

Q. 4 Implémenter une fonction `a_star` prenant en arguments un monde, une heuristique, un point de départ et un point d'arrivée et calculant, au moyen de l'algorithme A* défini par cette heuristique, la distance entre entre ces deux points dans le monde.

Q. 5 *Question bonus.* On peut aussi penser à une heuristique admissible qui considère non pas le chemin en ligne droite, mais les chemins qui suivent un segment de droite de pente 1 (*i.e.* une "diagonale"), puis un segment de pente 0 ou infini (*i.e.* un morceau plat, horizontal ou vertical). Coder cette heuristique et comparer les exécutions de l'algorithme A* avec ces deux heuristiques.

Q. 6 *Question bonus.* La meilleure heuristique♣ est en fait ... la distance exacte à l'objectif. Pour s'en convaincre on peut lancer l'algorithme A* en utilisant l'algorithme de Dijkstra heuristique.

♣. meilleure si on met de côté le coût du calcul de l'heuristique. ..