Samedi 12 avril 2025

OPTION SCIENCES NUMÉRIQUES

MPI

DURÉE : 2 HEURES

Conditions particulières :

Calculatrice et documents interdits

Le sujet est composé d'un problème et d'un questionnaire à choix multiples. Le questionnaire à choix multiples devra être inséré dans votre copie.

Les arbres de preuve demandés dans ce sujet devront utiliser les règles de la déduction naturelle listées en annexe.

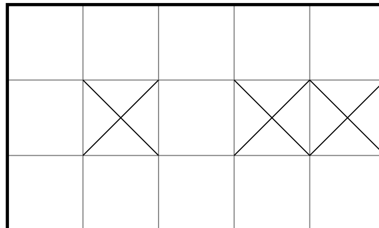
Les fonctions à produire dans ce sujet devront être rédigées en OCaml. Il est possible d'écrire des fonctions auxiliaires non explicitement demandées à condition de les documenter et de les définir avant d'en faire usage. Seules les primitives du langage OCaml ainsi que les fonctions des modules `List`, `Array` *et* `Queue` *pourront être utilisées sans restrictions. La documentation de certaines de ces fonctions est rappelée en annexe.*

Problème – Pavage d'une surface avec des dominos

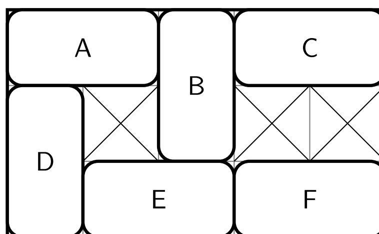
On considère une surface quadrillée composée de n rangées de p colonnes que l'on souhaite recouvrir avec des dalles de taille 1 par 2 que nous appellerons **dominos**. Les dominos ne peuvent être disposés que verticalement ou horizontalement. On fait de plus l'hypothèse que certaines cases du quadrillage peuvent comporter un **obstacle** (symbolisé par une croix) et ne sont donc pas à recouvrir.

L'objectif de ce problème est de déterminer un **pavage** de la surface, c'est-à-dire un recouvrement intégral, avec des dominos, des cases ne comportant pas d'obstacles.

Voici un exemple de surface de taille $n = 3$ par $p = 5$ comportant 3 obstacles :

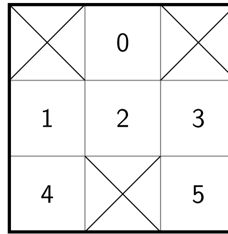


On peut paver cette surface à l'aide de 6 dominos A, B, C, D, E et F de la façon suivante :



1 Dénombrement et déduction naturelle

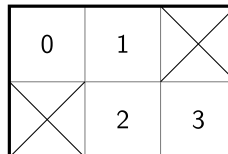
Question 1 Montrer par un raisonnement en langue française, qu'il n'existe qu'un seul pavage complet de la surface représentée ci-après et dont on a numéroté les cases.



Question 2 Une surface quadrillée avec des obstacles qui comporterait un nombre pair de cases libres est-elle toujours pavable ? En est-il de même si la surface est sans obstacle ? On justifiera ses réponses.

Question 3 Considérons une surface sans obstacles de taille 2 par n et notons u_n le nombre de façons de paver cette surface, montrer que $\forall n \in \mathbb{N}^\star,\ u_{n+2} = u_{n+1} + u_n$. On précisera les valeurs de u_1 et u_2.

On considère à présent la surface suivante dont on a numéroté les cases sans obstacles :



Notons a la variable propositionnelle indiquant si un même domino est placé sur les cases 0 et 1, b celle indiquant si un même domino est placé sur les cases 1 et 2 et c celle indiquant si un même domino est placé sur les cases 2 et 3.

Question 4 Donner une formule propositionnelle φ_1, exprimée sous forme normale disjonctive, traduisant le fait que la case numéro 1 doit être recouverte par exactement un domino.

Question 5 Donner une formule propositionnelle φ_2, exprimée sous forme normale conjonctive, traduisant le fait que la case numéro 2 doit être recouverte par exactement un domino.

Question 6 Établir un arbre de preuve pour le séquent $\varphi_1, b \vdash \neg a$.

Question 7 Établir un arbre de preuve pour le séquent $\varphi_2, b \vdash \neg c$.

Question 8 On définit la formule propositionnelle $\varphi_3 : (\neg a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c) \vee (a \wedge b \wedge \neg c)$.
Pourquoi faut-il que φ_3 soit satisfaite dans le cadre de notre problème ?

Question 9 Indiquer, au moyen d'une table de vérité, ce que l'on peut dire du séquent : $\varphi_3, b \vdash a \vee c$.

Question 10 À partir des séquents prouvés précédemment, construire un arbre de preuve du séquent $\varphi_1, \varphi_2, \varphi_3 \vdash \neg b$.
Expliquer ce que cela traduit concernant la configuration des dominos pour le problème.

Question 11 Existe-t'il un arbre de preuve du séquent $\varphi_1, \varphi_2, \varphi_3 \vdash \neg a$? On justifiera sa réponse.

2 Recherche d'un couplage maximum dans un graphe biparti

On fait l'hypothèse, **pour tout le reste du problème**, que les graphes manipulés sont connexes.

On rappelle qu'un graphe $G = (S, A)$ est **biparti** s'il existe une partition de S en deux ensembles S_1 et S_2 tels que les arcs de A ont une de leurs extrémités dans S_1 et l'autre dans S_2. On rappelle également qu'un graphe est **bicolorable** s'il est possible d'attribuer une couleur à chacun de ses sommets de façon à utiliser au plus 2 couleurs différentes et que deux sommets adjacents n'aient jamais la même couleur.

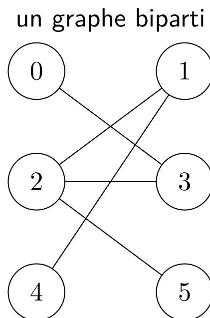
Quitte à renuméroter les sommets des graphes considérés, on choisit de représenter les graphes bipartis par le type :

```
type b_graph =
{
  n1 : int;
  n2 : int;
  voisins : int list array
}
;;
```

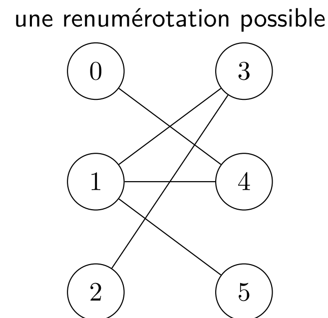
où `n1` est le cardinal de S_1, `n2` celui de S_2 et `voisins` est la représentation par listes d'adjacence du graphe.
Les sommets de S_1 étant alors **renumérotés** de 0 à $n_1 - 1$ et ceux de S_2 de n_1 à $n_1 + n_2 - 1$. La renumérotation des sommets sera représentée par un tableau de $n_1 + n_2$ entiers de sorte que la case d'indice i contienne le nouveau numéro du sommet i.

Par exemple :

un graphe biparti

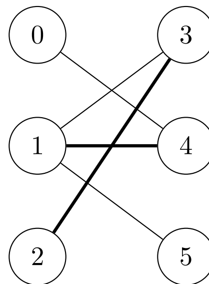


une renumérotation possible



Ainsi `{n1=3; n2=3; voisins = [|[4];[3;4;5];[3];[1;2];[0;1];[1]|]}` pourra représenter le graphe de l'exemple ci-dessus et `[|0;3;1;4;2;5|]` la renumérotation qui lui est associée.

On rappelle qu'un **couplage** d'un graphe est un ensemble d'arêtes du graphe deux à deux non incidentes (c'est-à-dire n'ayant pas de sommet en commun). Par exemple l'ensemble des arêtes {1, 4} et {2, 3} est un couplage du graphe biparti suivant :



Le **cardinal** d'un couplage est le nombre d'arêtes qui le composent. Un couplage d'un graphe est dit **maximum** lorsqu'il n'existe pas de couplage du graphe dont le cardinal lui serait strictement supérieur.

On choisit de représenter les couplages par un tableau indexé par les sommets du graphe. Le contenu de la case d'indice i contient le numéro du sommet avec lequel i est couplé, et `-1` sinon.

Question 12 Donner le tableau représentant le couplage ci-dessus.

Question 13 Écrire une fonction `cardinal : int array -> int` prenant en paramètre un couplage et retournant son cardinal.

On rappelle qu'un sommet est **libre** vis-à-vis d'un couplage s'il n'est l'extrémité d'aucune arête du couplage.

Le **graphe d'augmentation** associé à un couplage C d'un graphe biparti G est le graphe orienté obtenu à partir de G en appliquant les transformations suivantes :

1. *orientation des arêtes :*
   — les arêtes appartenant au couplage C sont orientées des sommets de S_2 vers ceux de S_1 ;
   — les arêtes n'appartenant pas au couplage C sont orientées des sommets de S_1 vers ceux de S_2 ;
2. *ajout de sommets :* on ajoute deux nouveaux sommets s et t, respectivement numérotés n_1+n_2 et n_1+n_2+1 ;
3. *ajout d'arcs :*
   — on ajoute pour chaque sommet x de S_1 et libre vis-à-vis de C, un arc allant de s vers x ;
   — on ajoute pour chaque sommet y de S_2 et libre vis-à-vis de C, un arc allant de y vers t.

Les graphes d'augmentation seront également représentés par le type `b_graph`.

Question 14 Dessiner le graphe d'augmentation associé au couplage et au graphe de l'exemple précédent.

Question 15 Écrire une fonction `graphe_augmentation : b_graph -> int array -> b_graph` construisant le graphe d'augmentation d'un graphe biparti et d'un couplage du graphe donnés. On prendra soin de retourner un graphe indépendant de celui passé en paramètre.

On appelle **chemin augmentant** tout chemin du graphe d'augmentation menant de s à t.

Question 16 Écrire une fonction `chemin_augmentant : b_graph -> int list` prenant en paramètre le graphe d'augmentation associé à un graphe et à un de ses couplages. Lorsqu'il existe un chemin menant de s à t, la fonction renverra la liste des sommets rencontrés dans l'ordre de parcours du chemin (sans s et t). Si un tel chemin n'existe pas, la fonction renverra la liste vide.

Question 17 Pourquoi un chemin augmentant comporte-t-il toujours un nombre pair de sommets ?

Question 18 Écrire une fonction `augmenter_couplage : int array -> int list -> unit` qui, étant donné un couplage et un chemin augmentant de ce couplage, remplace le couplage passé en paramètre par un couplage de cardinal strictement supérieur.

Question 19 Écrire une fonction `couplage_max : b_graph -> int array` qui, étant donné un graphe biparti, retourne un couplage maximum.

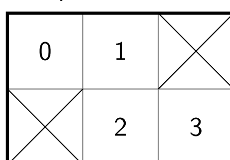
3 Retour au problème de pavage

On considère le problème de décision PAVABLE qui, à partir des dimensions d'une surface quadrillée et de la liste de ses obstacles, indique si la surface peut ou non être pavée.

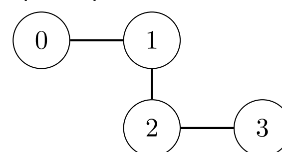
Question 20 Le problème PAVABLE est-il dans la classe NP ? On justifiera sa réponse.

On choisit de numéroter de gauche à droite et de haut en bas l'ensemble des m cases sans obstacle par les entiers consécutifs de 0 à $m-1$. On représente la surface par un graphe non orienté et non pondéré. L'ensemble des sommets est l'ensemble des numéros des cases sans obstacles et on considère que deux sommets sont adjacents si et seulement si les cases qu'ils représentent se touchent horizontalement ou verticalement. Les graphes seront représentés par leurs listes d'adjacence au moyen du type `int list array` (la case i contient la liste des voisins du sommet i). En voici un exemple :

Surface quadrillée numérotée



Graphe représentant la surface



Le graphe sera alors représenté par le tableau `[|[1];[0;2];[1;3];[2]|]`.

Pour simplifier on fera l'hypothèse, comme précisé dans la partie précédente, que l'on ne manipulera que des graphes connexes.

Question 21 Montrer qu'un graphe non orienté quelconque est biparti si et seulement si il est bicolorable.

Question 22 Montrer que le graphe représentant une surface est toujours bicolorable.

Question 23 Écrire une fonction `creer_b_graph : int list array -> b_graph * int array` prenant en paramètre un graphe biparti G représenté par listes d'adjacence. La fonction retournera une représentation de ce graphe sous le type `b_graph` ainsi que la renumérotation adoptée pour l'ensemble de ses sommets.

Question 24 Expliquer comment résoudre le problème PAVABLE et indiquer les conclusions que l'on peut en tirer concernant sa classe de complexité.

QCM – MPI

Numéro de candidat :

Document à insérer dans votre copie d'examen

Dans ce questionnaire à choix multiples, chaque question comporte une ou plusieurs bonnes réponses. Chaque réponse correcte fait gagner des points, mais chaque réponse fausse annule tous les points de la question. Les questions peuvent-être formulées au pluriel par commodité d'expression. Cela n'implique pas nécessairement qu'elles admettent plusieurs réponses correctes.

On considère la fonction, écrite en langage C, suivante :

```c
int mystere(int m, int n)
{
    int t=1;
    if (n>0)
    {
        t = mystere(m,n/2);
        t = t*t;
        if (n%2==1)
            t=m*t;
    }
    return t;
}
```

1. La fonction `mystere` :
   - ☐ effectue un nombre linéaire d'appels récursifs en `n` quand `n` est une puissance de 2 ;
   - ☐ effectue un nombre quadratique d'appels récursifs en `n` quand `n` est une puissance de 2 ;
   - ☐ permet de calculer les puissances d'un nombre ;
   - ☐ permet de calculer le produit de deux nombres.

2. L'ensemble des langages réguliers est stable :
   - ☐ par intersection finie ;
   - ☐ par union finie ;
   - ☐ par union infinie ;
   - ☐ par passage au complémentaire.

3. L'algorithme de Kosaraju permet de :
   - ☐ déterminer les composantes fortement connexes d'un graphe orienté ;
   - ☐ déterminer un plus court chemin entre tout couple de sommets d'un graphe orienté ;
   - ☐ déterminer un arbre couvrant de poids minimal d'un graphe non orienté pondéré ;
   - ☐ déterminer un automate fini acceptant le langage dénoté par une expression régulière.

4. Parmi les algorithmes suivants, indiquer ceux opérant sur des graphes.
   - ☐ l'algorithme de Kruskal ;
   - ☐ l'algorithme de Boyer-Moore ;
   - ☐ l'algorithme de Quine ;
   - ☐ l'algorithme de Dijkstra.

5. La formule propositionnelle $(\neg a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee b \vee \neg c)$:
   - ☐ est sous forme normale conjonctive ;
   - ☐ est sous forme normale disjonctive ;
   - ☐ est satisfiable ;
   - ☐ est équivalente à $(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$.

Annexe A – règles de la déduction naturelle

On se donne les règles suivantes de la déduction naturelle où φ, φ_1, φ_2 et ψ désignent des formules logiques et Γ un ensemble de formules logiques. De plus $\top$ désigne la tautologie et $\bot$ l'antilogie (ou contradiction).

Axiome	Affaiblissement
$\dfrac{}{\Gamma, \varphi \vdash \varphi}$	$\dfrac{\Gamma \vdash \varphi}{\Gamma, \Delta \vdash \varphi}$ (aff)

	Élimination	Introduction
$\top$ et $\bot$	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}$ $(\bot_e)$	$\dfrac{}{\Gamma \vdash \top}$ $(\top_i)$
$\wedge$	$\dfrac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_1}$ $(\wedge_e)$ $\quad \dfrac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_2}$ $(\wedge_e)$	$\dfrac{\Gamma \vdash \varphi_1 \quad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2}$ $(\wedge_i)$
$\vee$	$\dfrac{\Gamma \vdash \varphi_1 \vee \varphi_2 \quad \Gamma, \varphi_1 \vdash \psi \quad \Gamma, \varphi_2 \vdash \psi}{\Gamma \vdash \psi}$ $(\vee_e)$	$\dfrac{\Gamma \vdash \varphi_1}{\Gamma \vdash \varphi_1 \vee \varphi_2}$ $(\vee_i)$ $\quad \dfrac{\Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \vee \varphi_2}$ $(\vee_i)$
$\rightarrow$	$\dfrac{\Gamma \vdash \varphi \quad \Gamma \vdash \varphi \rightarrow \psi}{\Gamma \vdash \psi}$ $(\rightarrow_e)$	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \rightarrow \psi}$ $(\rightarrow_i)$
$\neg$	$\dfrac{\Gamma \vdash \varphi \quad \Gamma \vdash \neg\varphi}{\Gamma \vdash \bot}$ $(\neg_e)$	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}$ $(\neg_i)$

On s'autorisera également l'emploi des deux règles suivantes sans en fournir d'arbre de preuve :

$$\dfrac{\Gamma \vdash \neg\varphi_1 \vee \neg\varphi_2}{\Gamma \vdash \neg(\varphi_1 \wedge \varphi_2)}\ (\wedge_{\mathsf{dm}}) \qquad \dfrac{\Gamma \vdash \neg\varphi_1 \wedge \neg\varphi_2}{\Gamma \vdash \neg(\varphi_1 \vee \varphi_2)}\ (\vee_{\mathsf{dm}})$$

Annexe B – extraits de la documentation OCaml [1]

Extraits concernant le module Array pour manipuler des tableaux

`val length : 'a array -> int`

> *Return the length (number of elements) of the given array.*

`val make : int -> 'a -> 'a array`

> `make n x` *returns a fresh array of length* `n`*, initialized with* `x`*. All the elements of this new array are initially physically equal to* `x` *(in the sense of the* `==` *predicate). Consequently, if* `x` *is mutable, it is shared among all elements of the array, and modifying* `x` *through one of the array entries will modify all other entries at the same time.*

`val copy : 'a array -> 'a array`

> `copy a` *returns a copy of* `a`*, that is, a fresh array containing the same elements as* `a`*.*

`val iter : ('a -> unit) -> 'a array -> unit`

> `iter f a` *applies function* `f` *in turn to all the elements of* `a`*.*
> *It is equivalent to* `f a.(0); f a.(1); ...; f a.(length a - 1); ()`*.*

`val map : ('a -> 'b) -> 'a array -> 'b array`

> `map f a` *applies function* `f` *to all the elements of* `a`*, and builds an array with the results returned by* `f` *:*
> `[| f a.(0); f a.(1); ...; f a.(length a - 1) |]`*.*

`val for_all : ('a -> bool) -> 'a array -> bool`

> `for_all f [|a1; ...; an|]` *checks if all elements of the array satisfy the predicate* `f`*.*
> *That is, it returns* `(f a1) && (f a2) && ... && (f an)`*.*

1. Sources : https://ocaml.org/manual/5.2/api/Array.html, https://ocaml.org/manual/5.2/api/List.html et https://ocaml.org/manual/5.2/api/Queue.html

`val exists : ('a -> bool) -> 'a array -> bool`

> `exists f [|a1; ...; an|]` *checks if at least one element of the array satisfies the predicate* `f`.
> *That is, it returns* `(f a1) || (f a2) || ... || (f an)`.

Extraits du module List pour manipuler des listes

`val length : 'a list -> int`

> *Return the length (number of elements) of the given list.*

`val hd : 'a list -> 'a`

> *Return the first element of the given list.*

`val tl : 'a list -> 'a list`

> *Return the given list without its first element.*

`val rev : 'a list -> 'a list`

> *List reversal.*

`val iter : ('a -> unit) -> 'a list -> unit`

> `iter f [a1; ...; an]` *applies function* `f` *in turn to* `[a1; ...; an]`. *It is equivalent to* `f a1; f a2; ...; f an`.

`val map : ('a -> 'b) -> 'a list -> 'b list`

> `map f [a1; ...; an]` *applies function* `f` *to* `a1, ..., an`, *and builds the list* `[f a1; ...; f an]` *with the results returned by* `f`.

`val fold_left : ('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc`

> `fold_left f init [b1; ...; bn]` *is* `f (... (f (f init b1) b2) ...) bn`.

`val for_all : ('a -> bool) -> 'a list -> bool`

> `for_all f [a1; ...; an]` *checks if all elements of the list satisfy the predicate* `f`.
> *That is, it returns* `(f a1) && (f a2) && ... && (f an)` *for a non-empty list and* `true` *if the list is empty.*

`val exists : ('a -> bool) -> 'a list -> bool`

> `exists f [a1; ...; an]` *checks if at least one element of the list satisfies the predicate* `f`.
> *That is, it returns* `(f a1) || (f a2) || ... || (f an)` *for a non-empty list and* `false` *if the list is empty.*

`val mem : 'a -> 'a list -> bool`

> `mem a set is true` *if and only if* `a` *is equal to an element of* `set`.

Extraits concernant le module Queue pour manipuler des files

`val create : unit -> 'a t`

> *Return a new queue, initially empty.*

`val push : 'a -> 'a t -> unit`

> `push x q` *adds the element* `x` *at the end of the queue* `q`.

`val pop : 'a t -> 'a`

> `pop q` *removes and returns the first element in queue* `q`, *or raises* `Queue.Empty` *if the queue is empty.*

`val is_empty : 'a t -> bool`

> *Return* `true` *if the given queue is empty,* `false` *otherwise.*

Fin du sujet