Projet de Programmation Fonctionnelle 2018-2019 : Mondrian

Table des matières

1 Le jeu

Le but de ce projet est d'implémenter en OCaml un jeu de puzzle consistant à colorier, avec les couleurs rouge et bleu, un plan partitionné en rectangles [1].

Au début du jeu, tous les rectangles sont blancs, et les lignes partitionnant le plan peuvent être de couleur noire, rouge, bleu ou violet (*cf.* Figure 1). Le joueur doit choisir une couleur de remplissage pour chaque rectangle, en respectant les contraintes suivantes. Pour chaque ligne de séparation, on considère l'ensemble des rectangles dont le bord se superpose à celle-ci :

— Si la ligne est rouge ou bleue, cet ensemble doit contenir strictement plus de rectangles de même couleur que la ligne, que de rectangles de l'autre couleur.

1. Le nom du projet est en l'honneur du célèbre peintre abstrait Piet Mondrian, dont une partie des œuvres structurent le plan de la toile en rectangles colorés par les trois couleurs primaires rouge, bleu et jaune. Voir `https://fr.wikipedia.org/wiki/Piet_Mondrian`.

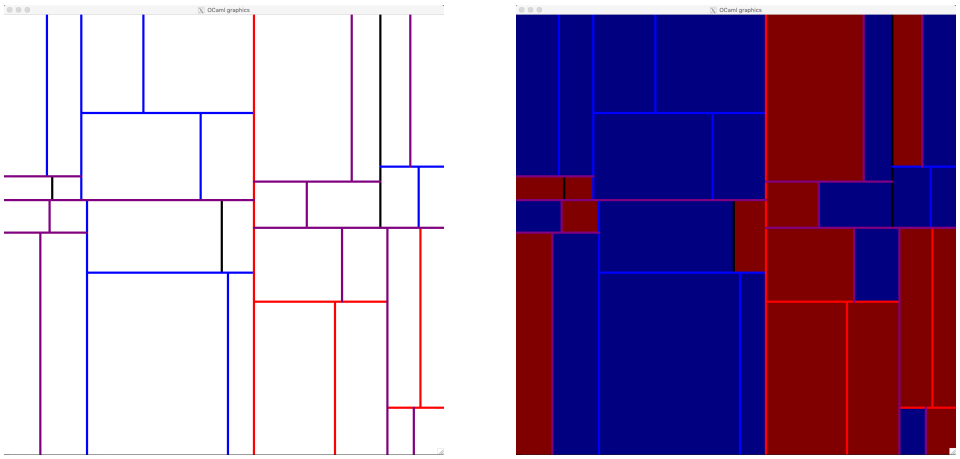


FIGURE 1 – Exemple d'une configuration initiale (à gauche) et finale (à droite) du jeu

— Si la ligne est violette, l'ensemble doit contenir autant de rectangles d'une couleur que de l'autre.

— Si la ligne est noire, il n'y a aucune contrainte sur ces rectangles.

La Figure 1 montre, à droite, un exemple de configuration initiale, à gauche, un exemple de remplissage respectant ces contraintes.

2 Le projet minimal

Le travail minimal que nous vous demandons d'effectuer est le suivant :

Vous devrez écrire et nous présenter un programme OCaml permettant :

1. **de générer aléatoirement une configuration initiale du jeu, sous la forme d'une partition binaire de l'espace (Section 3) ;**
2. **de jouer manuellement, l'utilisateur choisissant les couleurs des rectangles non encore remplis, le programme lui annonçant la fin du jeu dès que le coloriage est complet et correct ;**
3. **de générer aléatoirement une configuration initiale du jeu admettant une unique solution. Cette dernière fonctionnalité doit utiliser un solveur SAT fourni sur Moodle, permettant de tester, dans la plupart des cas de manière efficace, si une solution est unique.**

Cette liste de fonctionnalités n'est évidemment pas exhaustive : toute extension approuvée par votre chargé de TP sera la bienvenue, quelque idée d'extension facultative est donnée en Section 8. Votre programme devra pouvoir gérer de manière efficace un nombre raisonnable de régions (32 dans l'exemple ci-dessus).

3 Partition binaire de l'espace (arbre BSP)

La modélisation d'une partition d'un plan en rectangles doit être raisonnablement concise (sans informations inutiles) : elle doit en outre permettre de reconnaître rapidement si un point appartient à un rectangle (nécessaire au déroulement du jeu), et d'afficher facilement une partition à l'écran.

La représentation que nous avons choisie, facilement modélisable par un ensemble des types OCaml, est celle des *arbres BSP* (“Binary Space Partitioning”). Il s’agit d’un système très efficace permettant de partitionner un espace en zone convexes (dans notre cas, des rectangles) qui a été utilisé dans les années 90 dans les moteurs graphiques de plusieurs jeux video 3D (*Doom*, *Unreal*, *Quake*, etc).

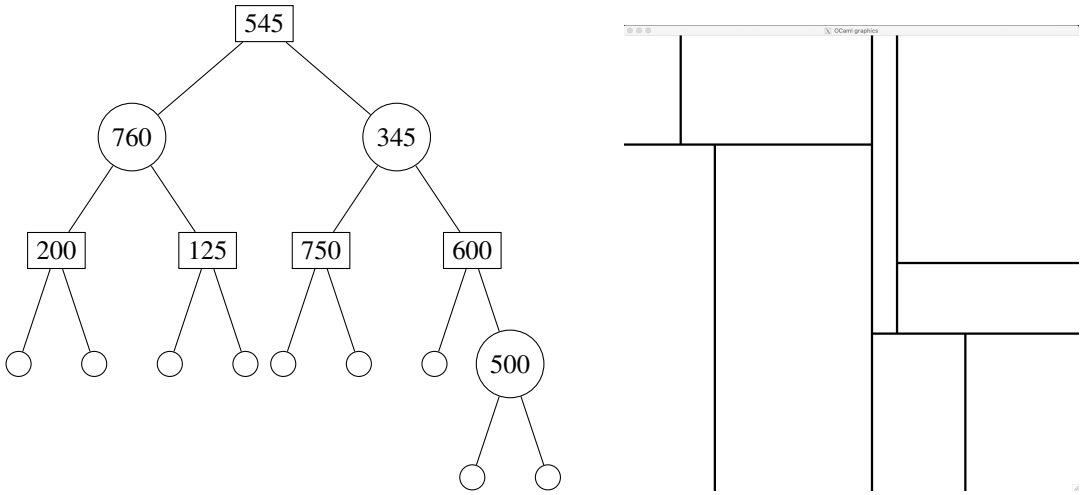


FIGURE 2 – Exemple d’un arbre BSP (gauche) et de la partition associé sur un plan 1000×1000 (droite).

Les BSP considérés ici sont des arbres *alternés*, c’est-à-dire contenant deux sortes de nœuds : les noeuds de hauteur paire sont d’une sorte, les noeuds de hauteur impaire sont de l’autre sorte. Un nœud de l’une ou l’autre des deux sortes peut être soit binaire, à deux fils et étiqueté par un entier, soit d’arité nulle, sans descendants et étiqueté par une couleur – ou par une valeur indiquant une absence de couleur. Les nœuds binaires de chaque sorte vérifient dans un BSP la propriété des ABR :

— Pour chaque sous-arbre d’un BSP dont la racine est un nœud binaire, si cette racine est étiquetée par n, tous les noeuds binaires de même sorte dans son fils gauche (resp. droit) sont étiquetés par des entiers strictement inférieur (resp. supérieur) à n.

Chaque sous-arbre d’un BSP est associé à une région rectangulaire du plan. L’étiquette d’un noeud binaire de hauteur paire (respectivement impaire) indique l’abscisse (resp. l’ordonnée) d’une ligne verticale (respectivement, horizontale) découpant cette region en deux nouvelles régions du plan associées à chacun de ses fils. Les feuilles d’un BSP sont des nœuds d’arité nulle, associés aux rectangles de la configuration représentée par cet arbre, et étiquetés par la couleur de remplissage de ces rectangles – ou indiquant leur non-remplissage éventuel.

La figure 2 donne un exemple de BSP (sans l’étiquetage de ses feuilles) et de la partition qui lui est associée : la racine indique que le plan est divisée en deux régions par une ligne verticale d’abscisse 545 ; l’étiquette de la racine de son fils droit indique que la région droite est elle-même divisée en deux sous-régions par une ligne horizontale d’ordonnée 345, la region supérieure est elle-même divisée en deux régions par une ligne verticale d’abscisse 600, etc.

4 Modélisation des BSP en OCaml

Vous devez utiliser les types définis ci-dessous sans les modifier, afin de nous permettre de tester votre code avant et pendant la soutenance.

Un BSP sera représenté à l’aide des types suivants :

```
type label = { coord : int; colored : bool; }
type bsp = R of color option | L of label * bsp * bsp
```

Les points considérés ici seront des coordonnées de pixels dans une fenêtre du module `Graphics`. Les deux sortes de noeuds d'un BSP sont représentés à l'aide de l'unique type `bsp`.

— Le constructeur `L` permet de réprésenter à la fois les nœuds binaires correspondant à une ligne de séparation verticale dont l'abscisse est précisée par le champ `coord` d'une valeur de type `label`, et les nœuds binaires correspondant à une ligne de séparation horizontale dont l'ordonnée est précisée par le champ `coord` d'une valeur de type `label`.

— Le constructeur `R` permet de construire une feuille de hauteur paire ou impaire, étiqueté ou non par une couleur.

Le champ `colored` du type `label` indique si la ligne séparant deux régions doit être coloriée en fonction de la couleur de remplissage des rectangles avec lesquelles elle est en contact, ou si elle doit rester noire.

Voici par exemple la valeur de type `bsd` correspondant à l'arbre de la Figure 2. On suppose qu'aucun rectangle n'est colorié – *ie* que tous les nœuds terminaux sont étiquetés par `None` – et que toutes les lignes de séparation sont noires :

```
L ({coord = 545; colored = false},
   L({coord = 760; colored = false},
     L({coord = 200; colored = false},
       R None,
       R None
      ),
     L({coord = 125; colored = false},
       R None,
       R None
      )
     ),
   L({coord = 345; colored = false},
      L({coord = 750; colored = false},
       R None,
       R None
      ),
     L({coord = 600; colored = false},
       R None,
       L({coord = 500; colored = false},
         R None,
         R None
         )
      )
     )
  )
```

Remarque. L'usage d'un unique type pour les deux sortes de nœuds est destiné à vous permettre de factoriser au maximum votre code, sans avoir à dédoubler les fonctions prenant en argument un sous-arbre d'un abr et effectuant un traitement qui ne dépend pas de la parité de sa hauteur. Lorsque le traitement est très similaire, mais diffère d'une parité à l'autre, ce dédoublement reste en général évitable en fournissant un argument booléen supplémentaire aux fonctions, cet argument leur indiquant la parité de hauteur du sous-arbre qu'elles manipulent.

5 Fonctions principales du traitement

Les éléments qui suivent ne sont que des suggestions. Vous êtes libres d'en implémenter des variantes, ou de proposer vos propres méthodes, à condition qu'elles soient correctement présentées et justifiées pendant la soutenance.

5.1 Génération aléatoire de configurations finales

Une fonctionnalité à mettre en œuvre est celle de la génération aléatoire d'une configuration finale du jeu, avec ou sans unicité de la solution de la configuration initiale associée (*cf.* les points 1 et 3 des fonctionnalités minimales décrites à la Section 2).

Nous vous suggérons d'implémenter explicitement ces deux variantes – non unicité éventuelle, unicité garantie. La programmation de la seconde est un peu plus subtile. Nous vous donnerons plus de détails sur son écriture à la Section 7, et nous vous conseillons de ne vous en occuper qu'après avoir réussi à programmer la première :

— `random_bsp_naive`

— `random_bsp_unique`

Les deux fonctions devront génèrer aléatoirement, sous la forme d'un BSP, une configuration du jeu *finale*, c'est-à-dire dans laquelle chaque rectangle a effectivement reçu une couleur de remplissage (rouge ou bleu), et dans laquelle les lignes de séparation noires seront choisies de manière aléatoire. Chaque fonction recevra en arguments les dimensions de la zone rectangulaire globale de la configuration à générer, la profondeur maximale du BSP associé, et éventuellement d'autres informations.

La fonction `random_bsp_unique` devra effecter la génération aléatoire d'une configuration finale de manière à ce que la configuration initiale associée ait une unique solution, le choix des lignes de séparation noires étant minimal pour garantir cette unicité. Autrement dit :

— Le coloriage de la configuration initiale, selon les règles de la Section 1, est le seul compatible avec la couleur des lignes non noires dans la configuration finale, il n'en existe aucun autre.

— Si une seule des lignes non noires de la configuration initiale est rendue noire, cette propriété d'unicité est perdue.

5.2 Extraction des couleurs des lignes de séparation

Chaque ligne de séparation d'une configuration finale est associée à une couleur parmi rouge, bleu ou violet, uniquement déterminée par les couleurs de remplissage des rectangles avec lesquelles elle est en contact – selon l'étiquetage des nœuds binaires correspondants, cette couleur sera ou bien affichée, ou bien remplacée par du noir dans la configuration initiale proposée au joueur. Le calcul des couleurs de ces lignes peut par exemple être effectué à l'aide de deux fonctions auxiliaires

— `rectangles_from_bsp`

— `lines_from_bsp`

renvoyant respectivement, la liste des rectangles d'un BSP avec leurs couleurs, et la liste des lignes de séparation d'un BSP avec leurs couleurs.

5.3 Rendus de configurations à l'écran

Une autre fonctionnalité fondamentale est celle, en cours de jeu, de l'affichage sur le canevas graphique de la configuration courante du joueur :

— `draw_current_bsp`

Les rectangles seront remplis suivant l'état de cette configuration courante. Les segments seront, bien sûr, colorés comme ceux de la configuration initiale choisie en début de jeu.

5.4 Vérification de la configuration finale du joueur

Lorsque le joueur termine le remplissage des rectangles de la configuration initiale, il faut vérifier que celui-ci est compatible avec les couleurs de lignes de la solution. Ce test peut être implémenté à l'aide d'une fonction :

— `check_current`

Attention, s'il n'y a pas unicité de la solution, il est bien possible que le remplissage proposé par le joueur soit différent de celui de la configuration finale choisie en début de jeu.

6 Implémentation de la boucle d'interaction

Les éléments qui suivent sont imposés : votre programme doit implémenter au minimum les fonctionnalités décrites.

Sans tenir compte de ses extensions éventuelles, votre programme devra effectuer au moins les opérations suivantes :

0. Créer aléatoirement une configuration finale, puis afficher dans une fenêtre graphique la configuration initiale correspondante.

L'utilisateur doit pouvoir choisir le niveau de difficulté du puzzle généré : dimensions et profondeur égales, le niveau de difficulté le plus élevé est un puzzle généré par `random_bsp_unique` (donc à solution unique, et à visibilité minimale du coloriage des lignes, voir Section 7).

1. Attendre une action de l'utilisateur – idéalement à l'aide de la souris seule – en mettant à jour l'affichage après chaque action traitée, en lui permettant :

— de colorier un rectangle en rouge, bleu ou de remettre sa couleur en blanc ;
si la configuration est finale et gagnante, le féliciter et aller en **(2)**, sinon, revenir en **(1)** ;

— d'afficher la solution de la grille (voir Section 5.3) ;
aller en **(2)**.

2. Proposer à l'utilisateur de quitter le jeu ou de revenir en **(0)** en jouant un nouveau puzzle.

7 Unicité de la solution par un SAT-solveur

Il est plus probable de générer aléatoirement un BSP à solution unique si l'on choisit de rendre visibles les couleurs de toutes ses lignes de séparation. Une idée pour programmer `random_bsp_unique` est donc de partir d'un BSP dont toutes les lignes sont de couleur visible, de vérifier l'unicité de sa solution, puis de masquer ces couleurs tant qu'il reste des lignes dont la couleur peut être omise sans perdre cette propriété d'unicité. Pour cela, nous avons besoin d'une fonction `check_unicity` vérifiant qu'un BSP a un seul coloriage possible compatible avec le coloriage de ses lignes de séparation.

Vérifier la compatibilité de *tous* les coloriages possibles d'une configuration avec le coloriage de ses lignes est un traitement de complexité exponentielle en le nombre de rectangles. Nous avons donc besoin d'une heuristique permettant de traiter plus efficacement ce problème dans la plupart de cas. Nous vous proposons de le faire en traduisant le problème d'unicité du coloriage d'un BSD `a` en une formule du calcul propositionnel P sous forme normale conjonctive (voir Section 7.1), telle que P soit satisfaisable si et seulement si `a` n'a pas un coloriage unique. La satisfaisabilité de la formule sera testée par un SAT-solveur.

7.1 Un SAT-solveur en OCaml

Le fichier `sat_solver.ml` déposé sur `Moodle` vous propose un outil permettant de résoudre automatiquement les puzzles de coloriage. La fonction principale de ce fichier est `solve`, prenant en argument une formule du calcul propositionnel sous forme normale conjonctive, et renvoyant : ou bien une affectation qui satisfait cette formule ; ou bien l'indication du fait qu'elle n'est pas satisfaisable [2].

7.2 Formes normales conjonctives

Un *littéral* est une variable propositionnelle ou la négation d'une variable propositionnelle. Une *clause disjonctive* est une disjonction de littéraux. Une *forme normale conjonctive* est une conjonction de clauses disjonctives. Par exemple, la formule suivante est une forme normale conjonctive (fnc), formée de trois clauses contenant chacune deux littéraux :

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2 \vee x_3)$$

Une fnc peut être représentée en OCaml par une liste de listes de couples, chaque liste de couples représentant une clause, chaque couple représentant un littéral. La partie gauche de chaque couple est un booléen indiquant, lorsque sa valeur est `false`, que la variable propositionnelle est précédée d'une négation dans la clause correspondante. Par exemple, la formule ci-dessus pourra être représentée par une valeur de la forme

```
[[(true, x1); (false, x2)]; [(false, x1); (false, x3)]; [(true, x2); (true, x3)]]
```

Appliquée à une telle liste, la fonction `solve` produira une valeur de la forme

```
Some [(false, x3); (true, x1); (true, x2)]
```

indiquant des valeurs pour `x1`, `x2` et `x3` satisfaisant la formule ci-dessus (`x3` à `false`, `x1` et `x2` à `true`). Appliquée à une liste représentant une formule insatisfaisable, `solve` renvoie `None`.

Les listes passées à la fonction `solve` seront toujours de cette forme, mais le choix pour les parties droites de couples du type le mieux adapté dépend, en général, du problème à résoudre. Si l'on souhaite par exemple résoudre une grille de de Sudoku [3], un choix naturel est celui des triplets d'entiers, une valeur de la forme `(i, j, k)` s'interprétant par : "la case de ligne `i` et de colonne `j` de la grille contient la valeur `k`". Afin d'éviter de lier la fonction `solve` à un choix de type particulier, le fichier `sat_solver.ml` fournit ce qu'on appelle un *foncteur*.

7.3 Utilisation du foncteur `Sat_solver.Make`

Les notions de *module* (un ensemble de déclarations de types et de définitions de fonctions regroupées sous un même nom, *eg.* `List` ou `Graphics`), de *foncteur* (un module paramétrique, intuitivement, l'équivalent d'une "fonction prenant en argument un module et renvoyant un module") et de *signature* (une spécification du type des éléments d'un module ou d'un paramètre de foncteur) font partie des notions les plus avancées de ce cours. Elles vous seront présentées en cours de semestre, et ne sont pas indispensables pour commencer la rédaction de ce projet.

Le foncteur `Sat_solver.Make` peut prendre en argument tout module contenant la déclaration d'un certain type `t` ainsi que l'implémentation d'une certaine fonction `compare` de type `t -> t -> int`, *ie.* tout module dont la signature est :

2. Ce fichier est une version non optimisée du solveur SAT-MICRO de S. Conchon, J. Kanig et S. Lescuyer. Voir `https://hal.inria.fr/inria-00202831`. Pour plus de détails sur l'algorithme implémenté, *cf.* `https://fr.wikipedia.org/wiki/Algorithme_DPLL`.

3. Voir `https://fr.wikipedia.org/wiki/Sudoku`.

```
module type VARIABLES = sig
  type t
  val compare : t -> t -> int
end
```

Le choix de `t` est libre. La fonction `compare` doit spécifier un certain ordre sur les valeurs de type `t`, en respectant la convention suivante : `compare de a b` doit renvoyer `0` si `a` et `b` sont égaux, une valeur positive si `a` est plus grand que `b`, une valeur négative sinon.

Supposons par exemple que, dans la modélisation d'un problème donné, le type idéal pour représenter les variables propositionnelles soit le type `int`. On définit tout d'abord un module `Variables` respectant la signature `VARIABLES` :

```
module Variables = struct
  type t = int
  let compare x y = if x > y then 1 else if x = y then 0 else -1
end;;
```

On définit ensuite un nouveau module `Sat`, en passant le module `Variables` au foncteur `Sat_solver.Make` :

```
module Sat = Sat_solver.Make(Variables);;
```

Le module construit implémente alors la signature :

```
sig
  type literal = bool * int
  val solve : literal list list -> literal list option
end
```

Autrement dit, la fonction `Sat.solve` accepte en argument des listes de type `((bool * int) list) list`, et renvoie une valeur de type `((bool * int) list) option`. Le type `int` à droite de chaque occurrence de `bool` est `Variables.t`, c'est-à-dire le type `t` choisi dans le module `Variables`. On peut par exemple écrire

```
Sat.solve
  [[(true, 1); (false, 2)]; [(false, 1);(false, 3)]; [(true, 2); (true, 2)]]
```

et obtenir le résultat suivant :

```
Some [(false, 3); (true, 1); (true, 2)]
```

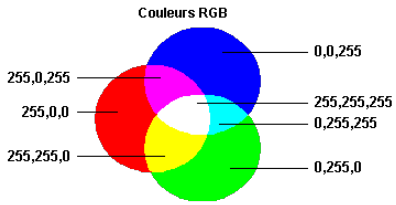
Plus généralement, quel que soit le choix de `Variables.t`, la fonction `Sat.solve` accepte des valeurs de type `(bool * Variables.t) list list`, et renvoie des valeurs de type `(bool * Variables.t) list option`. Dans le cadre de ce projet, vous devez choisir pour `Variables.t` la définition qui vous semble la mieux adaptée au problème posé.

8 Extensions

Une extension possible du projet consisterait à jouer avec d'autres couleurs que le rouge et le bleu. Par exemple, pour se rapprocher des vraies toiles de Mondrian, on pourrait autoriser les trois couleurs primaires, en coloriant les lignes par les couleurs primaires ou secondaires, selon la couleur dominante des rectangles en contact avec ces lignes (voir Figure 3 gauche).

Une autre extension pourrait être de partitionner le plan avec des régions plus générales que des rectangles, comme par exemple en Figure 3 à droite. [4]

4. Le projet de PF de 2016/2017 utilisait les diagrammes de Voronoi pour décrire une partition de l'espace en régions convexes. Pour éviter tous suspicion de plagiat, nous ne permettons pas d'utiliser cette technique dans le cadre de ce projet.



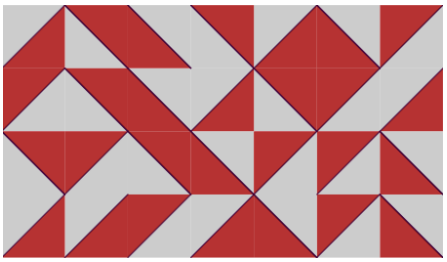


FIGURE 3 – Couleurs primaires et secondaires (à gauche) et exemple partition non rectangulaire (à droite)

Enfin, d'autres extensions pourraient enrichir la boucle d'interaction, par exemple en conservant l'historique des derniers coups, ou en permettant de sauvegarder (ou d'ouvrir) une partie dans un fichier.

9 Modalités de rendu

Le projet est à réaliser par groupes de 2 personnes au plus :

les projets réalisés à 3 personnes ou plus ne seront pas acceptés.

Votre rendu consistera en :

— un code-source écrit en OCaml, compilable et utilisable tel quel sous Linux et/ou sur les ordinateurs de l'UFR ; tout commentaire utile dans le code est le bienvenu ;

— un fichier texte nommé README contenant vos noms et indiquant brièvement comment compiler votre code-source et, si nécessaire, comment utiliser l'exécutable produit ;

— tout autre fichier nécessaire à la compilation et à l'exécution : fichiers d'exemples, `Makefile`, script permettant de compiler votre code-source, etc.

Tous ces éléments seront placés dans une unique archive compressée en `.tar.gz`. L'archive devra *obligatoirement* s'appeler *`nom1-nom2`*`.tar.gz`, et s'extraire dans un répertoire *`nom1-nom2/`*, où *`nom1`* et *`nom2`* sont les noms des deux personnes constituant le groupe. Par exemple, si vous vous appelez Pierre Corneille et Jean Racine, votre archive devra s'appeler `corneille-racine.tar.gz` et s'extraire dans un répertoire `corneille-racine/`.

La date limite de soumission est le vendredi 4 janvier à 23h59.

La soumission se fera via la page Moodle du cours.

Les projets rendus au delà de ce délai ne seront pas acceptés.

Attention, il s'agit d'un projet, pas d'un TP — vous devez donc nous fournir un programme non seulement bien écrit, mais faisant au moins quelque chose. Il vaut mieux un projet incomplet mais bien écrit et dans lequel une partie des choses fonctionnent, plutôt que d'essayer de tout faire, et de nous rendre un programme mal rédigé et/ou dans lequel rien ne fonctionne.

Enfin, de manière évidente,

votre code doit être strictement personnel.

Il ne doit s'inspirer ni de code trouvé sur le web, ni de celui d'un autre groupe, ni d'une quelconque "aide" trouvée sur un forum [5]. Inversement, il relève de votre responsabilité de faire en sorte que votre code reste inaccessible aux autres groupes. Par exemple, si vous vous servez d'un dépot `git`, ce dépot doit être privé.

5. Un plagiat avéré en projet constitue une fraude aux examens, passible de sanctions discplinaires assez lourdes. Une telle situation s'est déjà produite dans le cadre de ce cours, et a entraîné l'expulsion de l'étudiant concerné de l'Université Paris Diderot.