Compléments d'informatique

Projet 2 : Reconnaissance de croquis

27 octobre 2022

Dans ce projet [1], on vous propose d'implémenter une programme de reconnaissance de croquis ("sketch" en anglais), un croquis étant un dessin réalisé rapidement à main levée (à la souris ou au stylet). Par reconnaître un croquis, on veut dire identifier l'objet ou la chose que le dessinateur a voulu représenter par le croquis. Ce type d'algorithme de reconnaissance est étudié dans le domaine de l'intelligence artificielle. Du point de vue de la programmation, l'objectif est de vous apprendre à organiser un programme réparti en plusieurs fichiers, en vous basant sur la plupart des concepts vus au cours (organisation de programmes, type opaque, pointeurs de fonctions, etc.), ainsi que de vous faire faire un peu d'algorithmique pour arriver à un code efficace.

La section 1 décrit le principe général de l'algorithme de reconnaissance qu'on vous demande d'implémenter. La section 2 détaille ensuite la structure de code proposée et les différentes fonctions qu'il vous faudra implémenter dans chaque fichier. Des conseils sur l'implémentation vous sont donnés à la section 3. Vous serez également libre d'améliorer l'algorithme et vos implémentations seront mises en compétition dans un challenge décrit dans la section 4.

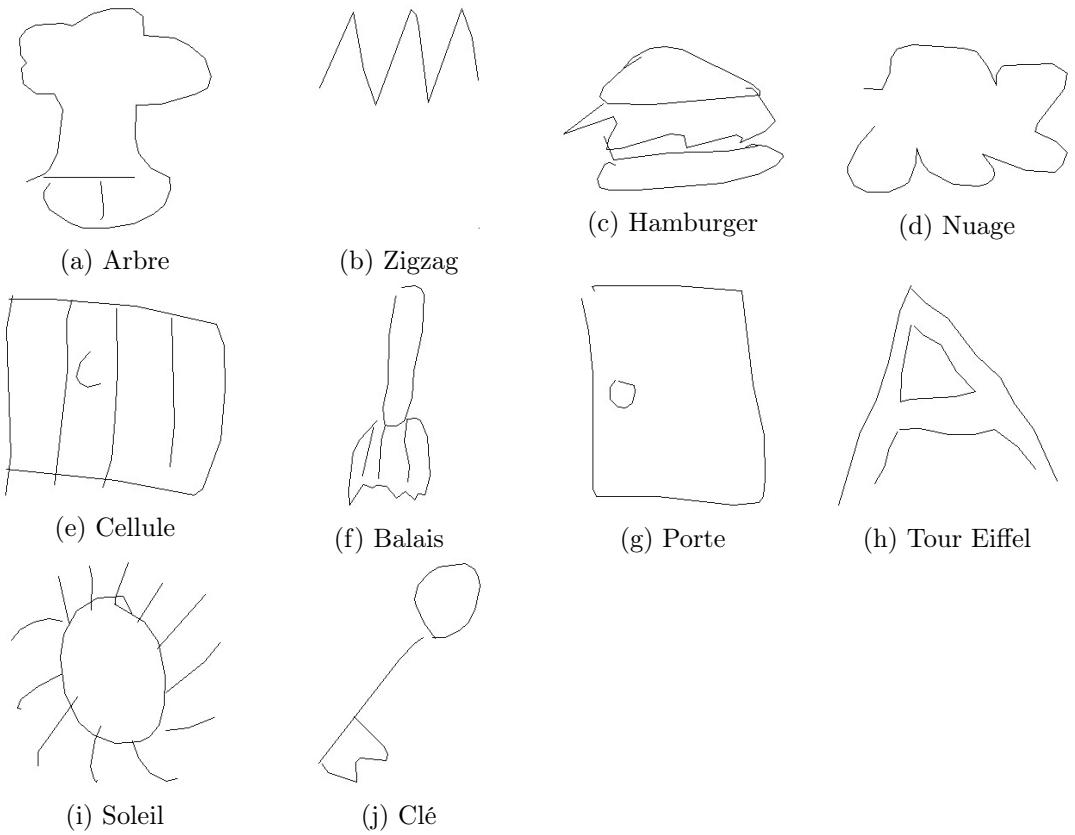
Ce projet est à réaliser par **groupe de deux étudiants maximum**. La date limite de remise du projet sur Gradescope est indiquée sur Ecampus.

1 Description de l'approche

Objectif général. On appellera dans la suite le **label** d'un croquis le nom de l'objet représenté par ce croquis. La figure 1 reprend une série de croquis avec leur label. L'objectif de l'algorithme de reconnaissance sera de prédire le label d'un nouveau croquis.

L'approche proposée est une méthode de reconnaissance très simple et générale appelée les k plus proches voisins ("k-nearest neighbors" ou kNN). L'idée de cette méthode est de

1. Ce projet s'inspire d'un projet proposé par Stephanie Valentine à l'Université de Nebraska-Lincoln.



(a) Arbre (b) Zigzag (c) Hamburger (d) Nuage

(e) Cellule (f) Balais (g) Porte (h) Tour Eiffel

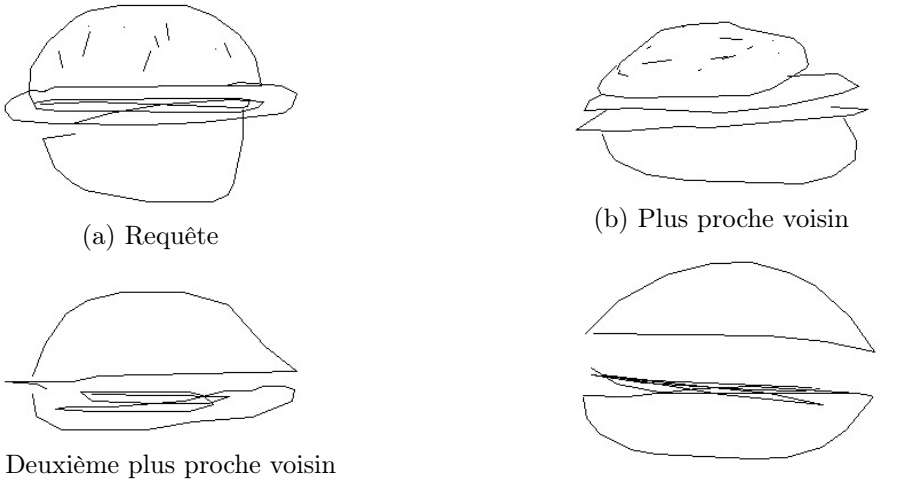
(i) Soleil (j) Clé

FIGURE 1 – Exemples de croquis avec leur label

constituer une base de données de croquis la plus large possible contenant des croquis avec leur label associé. En présence d'un croquis (appelé la requête) dont on veut déterminer le label, on identifie ensuite dans la base de données les k croquis les plus proches de ce croquis, selon une mesure de distance à préciser, et on prédit pour ce croquis le label qui apparaît le plus fréquemment parmi ses k plus proches voisins. Ce processus est illustré sur un exemple à la figure 2. La valeur de k est un paramètre de la méthode. On pourrait se contenter d'utiliser seulement le croquis le plus proche mais on obtient généralement de meilleurs résultats en utilisant une valeur de k plus élevée (mais pas trop).

Pour implémenter cette idée, il est nécessaire de choisir une représentation d'un croquis et ensuite de définir une mesure de distance entre deux croquis, qui servira à déterminer les plus proches voisins.

Représentation d'un croquis (*sketch*) Un croquis est représenté par un ensemble de *strokes*. Un *stroke* est une polyligne représentant un coup de crayon. Une polyligne est une



(a) Requête

(b) Plus proche voisin

(c) Deuxième plus proche voisin

(d) Troisième plus proche voisin

FIGURE 2 – Requête et plus proches voisins

succession de points (dans le plan 2D) formant des segments de droite. L'ordre des *strokes* et des points à l'intérieur d'un *stroke* suit l'ordre chronologique de tracé du croquis. Par exemple la porte de la figure 1 est constituée de deux *strokes* (un pour le panneau de porte et un pour la poignée) et le Zigzag d'un seul.

Mesure de distance. Il existe énormément de possibilités pour calculer la distance entre deux croquis. Dans ce projet, on vous propose d'utiliser la distance de Hausdorff. Si on considére un croquis comme un ensemble P de points (x, y) dans le plan, la distance de Hausdorff entre deux croquis P_1 et P_2 se calcule comme suit :

$$d_H(P_1, P_2) = \max\{\max_{(x_1,y_1)\in P_1}\{\min_{(x_2,y_2)\in P_2}\{d_E(x_1, y_1, x_2, y_2)\}\}, \max_{(x_2,y_2)\in P_2}\{\min_{(x_1,y_1)\in P2}\{d_E(x_1, y_1, x_2, y_2)\}\}\}, \tag{1}$$

où d_E est la distance euclidéenne entre deux points :

$$d_E(x_1, y_1, x_2, y_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Si on définit la distance d'un point à un croquis par la plus petite distance entre ce point et n'importe quel point du croquis, la distance de Hausdorff consiste à calculer la plus grande distance entre un point du premier croquis et le deuxième croquis et la plus grande distance entre un point du deuxième croquis et le premier croquis et ensuite à prendre la plus grande des deux comme distance entre les croquis. Deux croquis strictement identiques auront ainsi une distance de Hausdorff de 0 et ils seront d'autant plus distants l'un de l'autre qu'il existera un point dans l'un des croquis très éloigné de tous les points de l'autre croquis.

Nos croquis étant représentés par des polylignes, il n'est pas possible d'énumérer tous les points qui les composent pour calculer (1) (il y en a une infinité). Il est néanmoins possible de calculer la distance de Hausdorff entre deux croquis exactement. Soit deux croquis $\mathcal{S}_1$ et $\mathcal{S}_2$. Notons par S_1 et S_2 l'ensemble des segments de droite qui les composent et par P_1 et P_2 l'ensemble des points correspondant aux extrémités de ces segments. Il est possible de montrer que la distance de Hausdorff entre les deux croquis peut se calculer par l'expression suivante :

$$d_H(\mathcal{S}_1, \mathcal{S}_2) = \max\{\max_{(x_1,y_1)\in P_1}\{\min_{s_2\in S_2}\{d_s((x_1,y_1),s_2)\}\}, \max_{(x_2,y_2)\in P_2}\{\min_{s_1\in S_1}\{d_s((x_2,y_2),s_1)\}\}\}, \tag{2}$$

où $d_s((x,y),s)$ mesure la distance (euclidienne) minimale entre le point (x,y) et un point du segment s. Cette dernière distance peut se calculer efficacement de manière analytique (*Astuce :* distinguer le cas où la projection perpendiculaire du point sur le droite formée par le segment se trouve sur le segment du cas où il se trouve en dehors).

Compression des croquis. La complexité de l'algorithme de reconnaissance va dépendre beaucoup des calculs de distances entre croquis, qui eux-mêmes vont dépendre du nombre de points et du nombre de segments, vu l'équation (2). Pour accélérer les temps de calcul, on se propose de compresser les croquis, c'est-à-dire de réduire le nombre de points des différents *strokes* (polylignes) constituant les croquis. Evidemment, l'algorithme de compression devra réaliser cette compression en maintenant la polyligne compressée aussi proche que possible de la polyligne originale, au sens de la distance de Hausdorff.

On se propose d'implémenter pour cela l'algorithme de Ramer-Douglas-Peucker [2], qui se base sur l'idée du diviser-pour-régner vue au cours. L'algorithme prend en entrée une polyligne et un seuil de distance d_{max} et renvoie une nouvelle polyligne, constituée d'un sous-ensemble des points de la polyligne de départ, dont la distance de hausdorff à la polyligne initiale ne dépasse pas d_{max}. L'algorithme fonctionne de la manière suivante sur une polyligne démarrant au point p_s et se terminant au point p_e :

- On détermine le point p^* entre les extrémités p_s et p_e qui est le plus distant du segment défini par p_s et p_e (au sens de la distance d_s ci-dessus).
- Si la distance entre p^* et le segment est inférieure ou égale à d_{max}, on ne garde que p_s et p_e dans la polyligne compressée.
- Sinon, on applique l'algorithme récursivement pour compresser les deux sous-polylignes entre p_s et p^* et entre p^* et p_e.

Vu le principe de l'algorithme, il est facile de se convaincre que les extrémités de la polyligne et chaque nouveau point p^* menant à un appel récursif seront conservés dans la polyligne compressée. Notons qu'on n'a pas de garantie que l'algorithme fournira la plus courte, en termes de nombre de points, polyligne d'une distance de Hausdorff inférieure à d_{max} de la polyligne de départ mais l'algorithme devrait néanmoins être relativement efficace.

2. https://en.wikipedia.org/wiki/Ramer–Douglas–Peucker_algorithm

Evaluation des performances. Etant donné un seuil de compression, une mesure de distance entre croquis et une valeur de k, il est intéressant d'évaluer les performances du système de reconnaissance (par exemple, pour déterminer les valeurs de ces paramètres qui permettent d'obtenir les meilleures performances). Pour cela, on peut utiliser une base de données de croquis de test (différents des croquis de l'ensemble de référence), dont les labels sont connus. Pour chacun des croquis de cet ensemble, on calcule les k plus proches voisins dans la base de données de référence selon la mesure de distance choisie et on compare le label le plus fréquent parmi les k voisins au label correct du croquis de test. On peut ensuite quantifier les performances de l'algorithme de reconnaissance par le pourcentage de croquis de la base de données de test dont le label est correctement prédit. On appelle ce pourcentage la **précision** (accuracy) de l'algorithme.

2 Implémentation

L'implémentation est basée sur quatre modules :

— Un module constitué des fichiers `PointLine.c` et `PointLine.h` implémentant les structures et les fonctions nécessaires à la manipulation de points et de polylignes.

— Un module constitué des fichiers `Sketch.c` et `Sketch.h` déclarant la structure de croquis et les fonctions nécessaires à leur manipulation.

— Un module constitué des fichiers `Recognizer.c` et `Recognizer.h` implémentant les fonctions de reconnaissance à proprement parler.

— Un module constitué des fichiers `Dataset.c` et `Dataset.h` gérant des bases de données de croquis.

Nous vous fournissons une implémentation complète des fichiers `Dataset.c` et `Dataset.h` et vous êtes responsables de l'implémentation des autres modules en respectant les consignes données ci-dessous et dans le fichiers d'entête de ces modules.

2.1 Fichiers `PointLine.h` et `PointLine.c`

Ce module définit des fontions de manipulation de points et de polylignes. Les structures de points et de polylines sont définies de manière non opaques dans le fichier `PointLine.h` fourni. Ces objets doivent être passés par valeur, plutôt que par pointeur, entre les fonctions.

Fonctions de l'interface. Les fonctions à implémenter dans le fichier `PointLine.c` sont les suivantes :

`double plDistance(Point p1, Point p2)` : renvoie la distance euclidienne entre les points `p1` et `p2`.

`double plDistanceToSegment(Point p, Point p1, Point p2)` : renvoie la distance entre le point `p` et le segment de droite défini par les points `p1` et `p2` (c'est-à-dire la distance d_s

expliquée plus haut).

`double plDistanceToPolyline(Point p, PolyLine polyLine, double distanceMax)` : calcule la distance minimale entre un point `p` et tous les segments d'une polyligne. Si cette distance est inférieure à `distanceMax`, la fonction peut se contenter de renvoyer `distanceMax`. Cet argument a pour objectif d'éviter des calculs inutiles si l'information que la distance minimale est inférieure à `distanceMax` est suffisante.

`PolyLine plCompressPolyline(PolyLine polyLine, double dMax)` : compresse la polyligne en argument en utilisant l'algorithme décrit ci-dessus avec le seuil `dMax` donné en argument.

2.2 Fichiers `Sketch.h` et `Sketch.c`

Ce module définit le type de données abstrait `Sketch` utilisé pour définir un croquis (pour rappel, un ensemble de polylignes). La structure est opaque et doit être définie par vous-même.

Fonctions de l'interface. Les fonctions à implémenter dans le fichier `Sketch.c` sont les suivantes :

`Sketch *sketchCreate(int nbPoints, Point *points, bool *strokeStarts)` : crée un objet de type sketch où, `points` est un tableau de taille `nbPoints` de tous les points déterminant les segments du croquis. Le passage d'un *stroke* (ou polyligne) à l'autre est déterminé par les valeurs `true` dans le tableau `strokeStarts`, c'est-à-dire que si `strokeStarts[i]` vaut `true`, alors le point `points[i]` est le début d'un *stroke* (`strokeStarts[0]` vaut toujours `true`).

`void sketchFree(Sketch *sk)` : libère la mémoire prise par le croquis.

`int sketchGetNbStrokes(Sketch *sk)` : renvoie le nombre de *strokes* dans le croquis.

`PolyLine sketchGetStroke(Sketch *sk, int i)` : renvoie la polyligne correspondant au `i`ème *stroke* du croquis.

`PolyLine sketchGetNbPoints(Sketch *sk)` : renvoie le nombre total de points définissant le croquis.

`Sketch *sketchCompress(Sketch *sk, double dMax)` : compresse le croquis, c'est-à-dire toutes ses polylignes, avec le seuil de distance `dMax`. La fonction doit créer et renvoyer un nouveau croquis.

`double sketchDistanceHausdorff(Sketch *sk1, Sketch *sk2)` : calcule la distance de Hausdorff d_H entre deux croquis selon la procédure décrite plus haut.

`double sketchDistanceCustom(Sketch *sk1, Sketch *sk2)` : calcule la distance de votre choix entre deux croquis. Voir la section 4.

2.3 Fichiers `Recognizer.h` et `Recognizer.c`

Ce module contient l'implémentation des fonctions permettant de calculer les plus proches voisins, de prédire le label d'un sketch et d'évaluer les performances du système de reconnaissance. Une structure de type `kNN` est définie pour vous dans le fichier `Recognizer.h` pour récupérer la liste des plus proches voisins. La structure est non opaque mais elle doit être manipulée par pointeur par les différentes fonctions.

Fonctions de l'interface. Les fonctions suivantes sont à implémenter dans le fichier `Recognizer.c` :

`void recFreekNN(kNN *knn)` : libére une structure de type `kNN`.

`kNN *recNearestNeighbors(Sketch *sk, Dataset *rs, int k, double (*distance)(Sketch *, Sketch *))` : calcule les `k` plus proches voisins d'un croquis `sk` à partir de l'ensemble de référence `rs` en utilisant la fonction de calcul de distance fournie en argument par pointeur. La structure renvoyée doit contenir la valeur de `k` utilisée, un pointeur vers l'ensemble de référence, les indices des `k` voisins dans le tableau `neighbors` et leurs distances respectives au croquis `sk` dans le tableau `distances`. Les croquis doivent être triés dans les tableaux `neighbors` et `distances` par ordre de distances croissantes.

`char *recGetMajorityLabel(kNN *knn)` : renvoie une chaîne de caractère contenant le label le plus fréquent parmi les `k` voisins.

`float recEvalkNN(Dataset *referenceSet, Dataset *testSet, int k, double (*distance)(Sketch *, Sketch *), FILE *out)` : calcule le pourcentage de croquis dans la base de données `testSet` dont le label est correctement prédit en utilisant la base de référence `referenceSet`, la valeur de `k` et la mesure de distance données en argument. L'argument `out` est un fichier de sortie permettant d'afficher des informations de votre choix sur l'état d'avancement du test et donc de faire patienter l'utilisateur. S'il est à `NULL` aucun résultat ne doit être affiché.

2.4 Fichier `Dataset.c` et `Dataset.h`

Ces fichiers contiennent des fonctions vous permettant de charger des croquis à partir d'un fichier. Un type de données `Dataset` est défini, de manière opaque, pour gérer une base de données de croquis avec leurs labels. L'implémentation de ces fonctions vous est fournie.

Fonctions de l'interface. Les fonctions fournies par l'interface sont les suivantes :

`Dataset* dsLoad(const char* filepath, FILE* out)` : charge une base de données de croquis à partir du nom de fichier donné en argument. Affiche des messages sur la sortie `out`. Si `out` est à `NULL`, aucun message n'est affiché.

`void dsFree(Dataset *dataset)` : libère l'espace mémoire pris par une base de données.

`int dsGetNbSketches(Dataset *dataset)` : renvoie le nombre de croquis dans la base de données.

`Sketch *dsGetSketch(Dataset *dataset, int i)` : renvoie le `i`ème croquis dans la base de données.

`int dsGetNbLabelNames(Dataset *dataset)` : renvoie le nombre de labels différents dans la base de données.

`int dsGetLabel(Dataset *dataset, int i)` : renvoie un entier entre 0 et $N_l - 1$ (où N_l est le nombre de labels dans la base de données) encodant le label du `i`ème croquis de la base de données.

`char *dsGetLabelName(Dataset *dataset, int i)` : renvoie une chaîne de caractère correspondant au `i`ème label (avec `i` compris entre 0 et $N_l - 1$).

`void dsCompress(Dataset *dataset, double dMax)` : compresse tous les croquis de la valeur de seuil `dMax`. Les anciens croquis sont libérés de la mémoire suite à l'opération.

2.5 Autres fichiers

Un fichier `main.c` vous est fourni qui permet de créer un exécutable appelé `NNSketch` pour tester votre code. Lancez cet exécutable sans arguments en ligne pour voir les options proposées. Vous pouvez tester l'algorithme de reconnaissance sur un croquis en particulier de la base de données de test (option `-i`) ou bien calculer la précision sur l'ensemble de test complet. Lorsqu'un seul croquis est testé, des images de ce croquis et de ses plus proches

voisins sont générés au format `ppm` dans un répertoire de sortie qui peut être changé dans les arguments (option `-o`). Pour générer ces images, une librairie graphique (fichiers `easyppm.c` et `easyppm.h`) est incluse lors de la compilation.

Trois fichiers de croquis vous sont fournis :

— `testset.txt` : la base de données de test.

— `trainingsetverylarge.txt` : la base de données de référence contenant 1000 exemples par label.

— `trainingset.txt` : un sous-ensemble de la base de données de référence contenant 100 exemples par label.

Ces croquis ont été obtenu de la base de données `Quick, Draw!` [3] de google. Nous nous sommes restreints à un sous-ensemble de 10 labels différents (tree, zigzag, hamburger, cloud, jail, broom, door, the Eiffel Tower, sun, et key, représentés à la figure 1). Pour permettre la comparaison entre croquis, les coordonnées (x, y) (entières) des points ont été ramenées entre 0 et 255. Ces croquis ont déjà été traités avec l'algorithme de compression (`dMax`$= 2$) pour réduire la taille des fichiers.

3 Conseils d'implémentation

Commencez par implémenter les fonctions du fichier `PointLine.c`. La plupart des fonctions sont simples à implémenter. `plDistanceToSegment` demandera de vous rappeler un peu de géométrie. Pour `plCompressPolyline`, nous vous conseillons de marquer d'abord par un algorithme récursif les points de la polyligne qui sont à garder et ensuite à construire la nouvelle polyligne à partir de cette information.

Passez ensuite au fichier `Sketch.c`. Réfléchissez à la structure `Sketch` en prenant en compte les fonctions que vous aurez à implémenter. Les fonctions de ce fichier ne devraient pas vous poser trop de problème. Pour la fonction `sketchDistanceHausdorff`, utilisez autant que possible les fonctions de `PointLine.h` et essayez d'obtenir l'implémentation la plus efficace possible (notamment en exploitant l'argument `distanceMax` de la fonction `plDistanceToPolyline`).

Terminez par l'implémentation de `Recognizer.c`. La fonction la plus importante de ce fichier est `recNearestNeighbors`. Essayez à nouveau de proposer l'implémentation la plus efficace possible de cette fonction, notamment en terme de mémoire. Une des difficultés de cette fonction est la gestion des `k` plus proches voisins, pour laquelle plusieurs solutions existent.

Comme toujours, nous vous conseillons d'écrire des fichiers de tests intermédiaires pour tester vos fonctions au fur et à mesure, plutôt que d'attendre d'avoir tout écrit pour faire des tests avec notre fichier main.c ou via Gradescope.

3. `https://quickdraw.withgoogle.com`

4 Compétition

Pour ajouter un peu de fun au projet, nous allons vous mettre en compétition. En plus de la mesure de distance de Hausdorff, nous vous laissons la possibilité de mettre au point une mesure de distance de votre cru qui vous semblerait plus appropriée que la distance de Hausdorff pour faire de la reconnaissance. Vous pouvez l'implémenter dans la fonction `sketchDistanceCustom`. Cette mesure de distance sera testée sur Gradescope sur des croquis (et potentiellement des labels) différents de ceux qui vous sont fournis et vous serez classés selon la précision que vous obtiendrez sur ces nouveaux croquis. Un timer devra être utilisé pour éviter que les tests ne durent trop longtemps. Il sera donc important que cette mesure de distance soit aussi efficace que possible.

5 Soumission

Le projet doit être soumis via Gradescope. Seuls les fichiers suivant doivent être soumis :

— `PointLine.c`
— `Sketch.c`
— `Recognizer.c`
— Le fichier `rapport.txt` complété avec vos réponses.

Dans ce fichier `rapport.txt` vous sont demandés les contributions de chacun au projet, des analyses de complexités pour certaines fonctions, ainsi que les valeurs de `k` et du paramètre compression `dMax` que vous souhaitez que nous utilisions pour tester votre mesure de distance pour la compétition.

Vos fichiers seront compilés et testés en utilisant le fichier `Makefile` fourni via la commande `make all` qui utilise les flags de compilation habituels (`-pedantic -Wall -Wextra -Wmissing-prototypes`), qui ne devront déclencher aucun avertissement lors de la compilation, sous peine d'affecter négativement la cote. L'efficacité de votre code et votre performance à la compétition (facultative) interviendront pour maximum 2 points de la cote finale du projet (sur 20).

Toutes les soumissions seront soumises à un programme de détection de plagiat. En cas de plagiat avéré, l'étudiant (ou le groupe) se verra affecter une cote nulle à l'ensemble des projets.

Bon travail !