Filière MP - ENS de Cachan, Lyon, Rennes et Paris - Session 2016
Page de garde du rapport de TIPE

NOM : SEGUIN	Prénoms : Béranger Fabrice
Classe : MP*	
Lycée : Fabert	Numéro de candidat : 10404
Ville : Metz (57)	

Concours auxquels vous êtes admissible, dans la banque MP inter-ENS (les indiquer par une croix) :

ENS Cachan	MP - Option MP		MP - Option MPI	X
	Informatique			
ENS Lyon	MP - Option MP		MP - Option MPI	X
	Informatique - Option M		Informatique - Option P	
ENS Rennes	MP - Option MP		MP - Option MPI	X
	Informatique			
ENS Paris	MP - Option MP		MP - Option MPI	X
	Informatique			

Matière dominante du TIPE (la sélectionner d'une croix inscrite dans la case correspondante) :

Informatique	X	Mathématiques		Physique	

Titre du TIPE :
Approche algorithmique du contrepoint rigoureux

Nombre de pages (à indiquer dans les cases ci-dessous) :

Texte	7	Illustration	9	Bibliographie	1

Résumé ou descriptif succinct du TIPE (6 lignes, maximum) :
Il s'agit de développer puis d'optimiser des algorithmes pour écrire du contrepoint rigoureux en tenant compte des règles et des recommandations au mieux.

À Metz

Le 06/06/2016

Signature du (de la) candidat(e)

BSeguin

Signature du professeur responsable de la classe préparatoire dans la discipline

Cachet de l'établissement

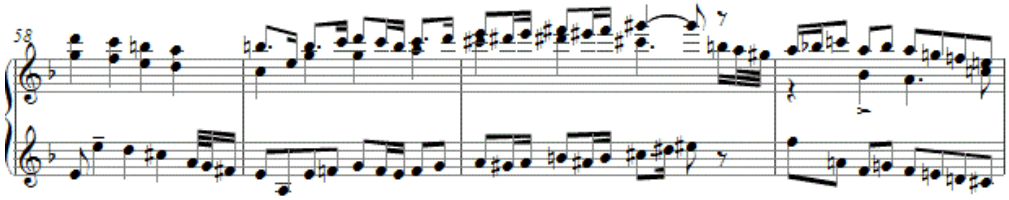
LYCÉE FABERT
12, rue Saint Vincent
57045 METZ Cedex 1
Tél. 03 87 30 03 03
Fax 03 87 31 29 55

LYCÉE FABERT - METZ
LE PROVISEUR ADJOINT

Approche algorithmique du contrepoint rigoureux

Introduction

Le contrepoint est une forme d'écriture musicale populaire à l'époque baroque (Bach). Le principe est que les lignes mélodiques évoluent indépendamment tout en restant consonantes. Cette conception horizontale s'oppose à l'harmonie, qui conçoit la musique verticalement. L'écriture contrapuntique est très rigoureuse, avec de nombreuses règles et recommandations.



Contrepoint fleuri à 3 voix (Bach, l'Art de la Fugue*)*

L'objectif de ce TIPE est d'étudier d'un point de vue informatique le problème de l'écriture en temps raisonnable de contrepoint qui tienne compte des règles et recommandations, en maximisant un certain score esthétique. Le contrepoint est un choix naturel, puisque les règles limitent le risque que la musique écrite ne soit trop désagréable.

J'ai choisi pour les règles le synthétique *Traité de Contrepoint*[1] d'André Gedalge.
Les programmes seront écrits en langage Python.
Toutes les mesures effectuées sont regroupées dans l'**annexe 10**.

I. Analyse du problème

Représentation des notes

La représentation traditionnelle des notes (degré et altération) n'est pas pratique ici. J'ai choisi de représenter une note par le nombre (algébrique) de demi-tons qui la séparent de la tonique médium. Cela permet plusieurs choses :

- La tonalité effective importe peu. Je nommerai les notes comme en Do majeur
- On distingue facilement les intervalles : Si-Fa vaut $5-(-1)=6$ demi-tons, Sol-Do vaut $7-0=7$ demi-tons
- Les altérations ne sont pas une exception : pour écrire Fa#, il suffit d'écrire 6

Par ailleurs, le module `MIDIUtil`[2] créant les fichiers MIDI utilise la même représentation.

Les voix ne contiendront que des rondes, ce seront donc de simples tableau d'entiers. Cette représentation par voix est en accord avec l'approche horizontale propre au contrepoint.

Étude théorique

Lorsque je calculerai des complexités, *n* désignera toujours le nombre de mesures.

Voici les règles retenues pour le contrepoint de première espèce :

- On écrira du contrepoint à deux voix, en rondes
- **Dans chaque voix (horizontal) :**
  - Les mouvements de triton (6 demi-tons) et de septième sont interdits
  - * Rester dans la gamme

1 Disponible ici : http://www.musimem.com/Contrepoint-Gedalge.pdf
2 Disponible ici : http://www.emergentmusics.org/midiutil

- ^ Favoriser largement le mouvement conjoint

- **Superposition (vertical) :**
  - Commencer par une consonance parfaite
  - Finir par une 6te maj ou une 3ce min, suivie d'un unisson ou d'une 8ve
  - N'employer que des consonances inférieures à la 12e
  - Ne pas employer plus de trois fois de suite la même consonance imparfaite
  - Les consonances parfaites ne doivent pas être approchées par mouvement direct ni être doubles.
  - La fausse relation de triton est interdite
  - * Pas de croisements entre les voix
  - ^ Les consonances doivent être imparfaites de préférence
  - ^ Le mouvement contraire est préféré à l'oblique, lui-même préféré au direct

* → modification par rapport à Gedalge
^ → recommandation
Les règles sans intérêt ici ont été supprimées

La vérification d'un contrepoint se fait en temps linéaire.

Les règles sont en fait toutes *locales*. On peut donc faire des vérifications par morceaux, ce qui sera important pour diminuer la complexité.

Le langage ainsi défini est même reconnaissable par un automate fini. En considérant comme alphabet les doublets (i, j) correspondant à (note inférieure, note supérieure) avec $-16 < i < 16$ et $i \leq j < i + 19$, on peut en effet écrire chaque règle comme un automate et considérer l'intersection des langages correspondants. L'automate obtenu est certainement gigantesque, mais l'écriture de l'automate de chaque règle est assez facile.

Si on oublie le défi posé par la maximisation du score, le problème est assez semblable à la résolution d'un Sudoku : il faut remplir un tableau en respectant des règles. Des méthodes de *backtracking s*ont donc envisageables : le programme s'appelle récursivement avec chaque possibilité pour une certaine case, en revenant en arrière lorsque la grille n'est plus correcte, et en renvoyant le résultat lorsqu'elle est pleine et correcte.

Comme le Sudoku, on peut aussi s'inspirer du comportement humain en prenant des décisions rapides réduisant le nombre de possibilités pour limiter les appels récursifs. Cela suppose de comprendre comment l'humain fait pour écrire du contrepoint. Je l'ai pratiqué et j'en suis venu aux conclusions suivantes :

- Il est très rare qu'on ait à revenir en arrière. Dans le pire des cas, j'ai dû revenir en arrière d'une mesure pour éviter une inélégance
- Si une note de la gamme est suffisamment proche de la précédente, qu'elle forme avec elle un intervalle mélodique licite et qu'elle forme avec la voix inférieure un intervalle harmonique licite, il faut l'essayer préférablement, en favorisant le mouvement contraire.

On a une complexité théorique au pire exponentielle. Pour la recherche d'**une** solution, la complexité est en fait linéaire en moyenne, puisque les retours en arrière sont rares. Le temps exponentiel est atteint par exemple lorsqu'il n'y a aucune solution, ou lorsqu'il n'y en a qu'une (cas du Sudoku).

Pour la recherche de **la** solution maximisant le score, il faut tester chaque possibilité, ce qui fait une complexité spatiale et temporelle véritablement exponentielle. Le score n'étant qu'indicatif (car la beauté musicale est subjective), je ne cherche pas des solutions absolument maximales, mais des solutions quasi-maximales obtenues rapidement avec des heuristiques peu coûteuses.

Vérification, score d'un contrepoint

J'écris une fonction `voix_valide(voix, debut, fin)` qui calcule le score d'une voix seule, en renvoyant `-1` si et seulement si elle est invalide, et qui ne vérifie la validité de la voix qu'entre les positions `debut` et `fin-1`. Le score d'une voix est la somme des valeurs de ses intervalles mélodiques (stockées dans `val_melo`).

Lorsque la voix `v` est valide entre `a` et `c-1`, on a la relation de Chasles :

`voix_valide(v, a, b) + voix_valide(v, b, c)= voix_valide(v, a, c)`

[ANNEXE 1]

Cette fonction est clairement linéaire, en O(`fin-debut`) (1,5µs par mesure). Les valeurs de `val_melo` ont été choisies arbitrairement pour favoriser le mouvement conjoint.

Quant au score d'un contrepoint, c'est la somme de quatre scores :

- Le score mélodique : c'est la somme des scores des voix, multipliée par `coeff_melo`.
- Le score harmonique : c'est la somme des valeurs des intervalles harmoniques (stockées dans `val_harmo`), multipliée par `coeff_harmo`.
- Le score de mouvement : en affectant 3 aux mouvements inverses, 1 aux obliques et 0 aux directs, c'est la somme des scores des mouvements, multipliée par `coeff_mvt`.
- Le score de diversité : on découpe la voix en paquets de `nb_mesures` mesures consécutives, on calcule pour chacun le nombre de notes distinctes puis on multiplie la somme de ces nombres par `coeff_div`.

Les coefficients sont choisis pour que les scores aient un ordre de grandeur comparable.

J'en déduis une fonction `contrepoint_valide(voix1, voix2, chk1, chk2, debut, fin)` en O(`fin-debut`) (8,1µs par mesure), ayant le même comportement que `voix_valide` (Chasles, `-1` ssi incorrect). Elle ne calcule pas le score de diversité, sans quoi la relation de Chasles ne serait pas respectée (le calcul du score de diversité étant impossible par morceaux).

J'écris également des fonctions `score_div` et `score` pour calculer le score total.

[ANNEXE 2]

Solutions naïves

Le problème est d'écrire la voix supérieure à partir de la voix inférieure (cantus firmus).

Cela commence avec la fonction `candidats2(voix1, voix2)` qui renvoie des notes susceptibles de compléter correctement la mesure suivante de `voix2`. J'implémente pour cela les critères vus précédemment.

[ANNEXE 3]

Pour trouver *une* solution quelconque, il suffit d'écrire l'algorithme de backtracking. Si une solution existe, il est certain qu'on en trouvera. Il ne faut par contre pas calculer le score complet à chaque fois, sinon la complexité est $O(\sum_{i=0}^{n} k) = O(n^2)$. On utilise donc la relation de Chasles sur `contrepoint_valide` pour mettre à jour le score en temps constant à chaque fois.

[ANNEXE 4]

La complexité en moyenne est linéaire (45µs par mesure). Ceci est compliqué à prouver rigoureusement, ne serait-ce que parce qu'on ne sait pas s'il y a une solution, mais si on fait l'hypothèse (fausse mais presque toujours vérifiée en pratique) que chaque contrepoint non encore faux peut être complété correctement, c'est clair.

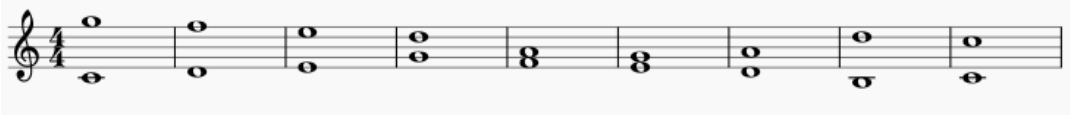
En mesurant le nombre de retours en arrière sur des exemples, je n'en ai en fait mesuré

aucun, ce qui signifie que l'hypothèse précédente est raisonnable (on n'est jamais coincé suite à un choix). Le *backtracking* est donc peu utile dans un cas comme celui-ci où les solutions sont nombreuses.

Cherchons désormais à maximiser le score. L'algorithme naïf pour obtenir une solution dont le score est parfaitement maximal est simple à écrire.

[ANNEXE 5]

Sa complexité (spatiale et temporelle) est clairement exponentielle, des mesures me permettent de l'estimer comme $\Theta(a^n)$ où $a \approx 4,2$.



Contrepoint (9 mesures) généré en 8,7s, de score 406 (maximal pour ce cantus firmus)

II. Optimisations et améliorations

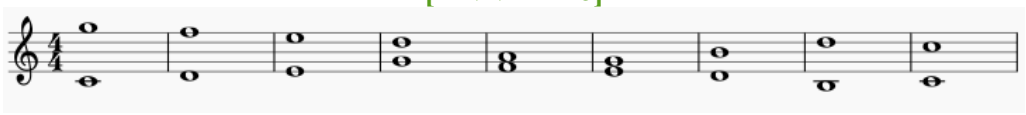
Algorithme glouton, résultats

Pour diminuer le temps de génération tout en essayant de maximiser le score localement, je mets en place l'heuristique suivante, qu'on appelle un algorithme glouton :

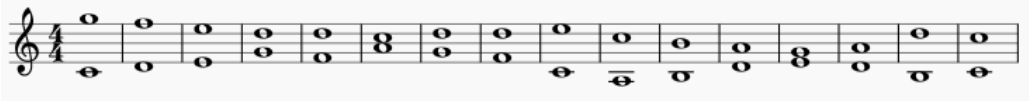
- À chaque étape, on a une liste `L` de `voix2` incomplètes, ne contenant initialement que `[]`.
- Pour chaque `voix2` de `L`, on génère toutes les possibilités pour les `nb_mesures` suivantes.
- De toutes les `voix2` ainsi obtenues, on ne garde que les `nb_glout_garde` meilleures, qu'on met dans la liste `L`, puis on recommence jusqu'à ce que les `voix2` soient complètes, et on garde alors la meilleure.

Plutôt que de calculer les scores à partir de rien, on les met à jour à chaque note, puis on met le score de diversité à jour après la génération de `nb_mesures`. Chaque étape se fait donc en temps constant, et le nombre d'étapes est de l'ordre de `n/nb_mesures`. L'algorithme, lorsqu'il trouve une solution, la trouve donc en temps linéaire, et maximise localement le score à chaque étape.

[ANNEXE 6]



Contrepoint (9 mesures) généré en 45ms, de score 406 (le même !)



Contrepoint (16 mesures) généré en 231ms, de score 748

Des mesures effectuées (avec `nb_mesures=5` et `nb_glout_garde=10`) montrent que le temps pris par l'algorithme est bien linéaire (41ms par mesure avec ces paramètres plutôt élevés).

Pour $n \leq 14$, on sait calculer le score maximal en temps raisonnable et on peut donc le comparer au score obtenu par l'algorithme glouton. Les résultats sont encourageants, avec un score plus faible d'au plus $0,55\%$. Pour de grandes valeurs de `n`, il est clair qu'on peut être bien plus éloignés du score maximal puisque les optimisations locales sont de moins en moins signifiantes, mais le résultat reste satisfaisant (comme on le verra avec le temps réel).

Pour des valeurs de `n` faibles devant `nb_mesures`, la complexité semble évidemment

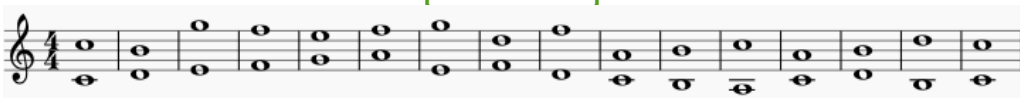
exponentielle, puisque c'est comme si on appliquait l'algorithme naïf.

Autres formes de contrepoint

Contrepoint renversable :

Il s'agit d'un contrepoint qui reste correct si on inverse les mélodies inférieures et supérieures. Les règles étant symétriques, à l'exception de celles concernant la quinte (qui renversée devient une quarte), il suffit d'exclure les quintes de `int_harmo` pour que les contrepoints obtenus soient renversables. Cependant, dans l'optique de maximiser le score dans les deux configurations, il faut recalculer le score harmonique qui, pour le renversement, n'a rien à voir.

[ANNEXE 7]



Contrepoint renversable (16 mesures) généré en 36ms, de score 723

Génération aléatoire :

Voici quelques choix et remarques pour la génération aléatoire de cantus firmi :

- Un cantus firmus commencera et finira par la tonique ;
- Les cantus firmi s'étendront entre le Do (-12) et le Do (+12), soit 25 notes possibles ;
- Pour qu'on puisse finir par une 6te maj ou une 3^{ce} min puis par une 8^{ve} approchée par mouvement contraire, il est nécessaire qu'on ait Si/Ré ou Ré/Si à l'avant dernière mesure, et le cantus firmus doit donc se finir soit par Si-Do, soit par Ré-Do.

On choisit d'utiliser des chaînes de Markov pour générer le cantus firmus : à chaque fois qu'on est sur une note, il y a une certaine probabilité d'aller vers chaque autre note, ce qui permet étape par étape de générer un cantus firmus. On rajoute deux particularités à ce processus :

- Si on vient de suivre l'arête Si-Do ou Ré-Do, on arrête la génération avec une probabilité ½, et on la continue normalement dans le cas contraire.
- Lorsqu'on vient de suivre une arête, on ne la reprend pas dans l'autre sens.

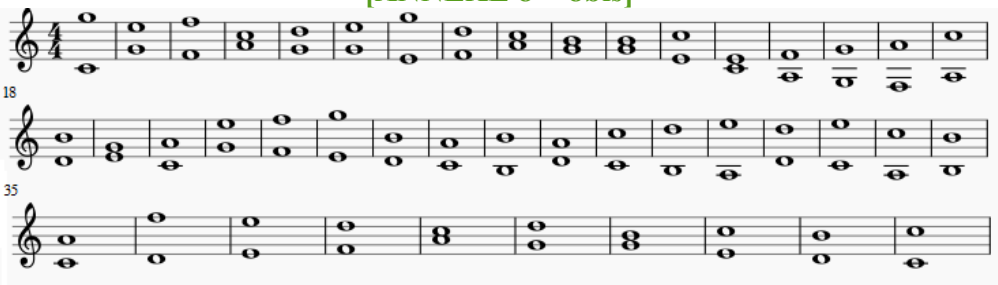
Cela rend plus variées les notes employées et rend aléatoire la longueur du cantus firmus.

Les probabilités sont stockées dans le fichier `transi.txt` sous forme d'une matrice M de taille 25×25. Le coefficient M_{ij} est un entier tel que la probabilité de transition de la note $i-12$ à la note $j-12$ soit $\frac{M_{ij}}{\sum_{k=0}^{24} M_{ik}}$.

J'obtiens les coefficients de la matrice M par analyse d'exemples : j'ai écrit un programme qui lit des exemples et incrémente $M_{i+12,j+12}$ à chaque transition de i à j. En lisant suffisamment d'exemples, cela permet d'écrire des cantus firmi proches de ce qu'un humain écrirait (par exemple, le mouvement conjoint est naturellement privilégié, puisqu'il l'est dans les exemples).

Tous ces programmes (analyse, génération) ont une complexité linéaire en la taille du cantus firmus et sont assez simples à écrire.

[ANNEXE 8 + 8bis]



Contrepoint sur un cantus firmus aléatoire (44 mesures) généré en 1,43s, de score 2127

Temps réel :

La génération en temps réel (on joue directement ce qu'on génère, à l'infini) repose sur des principes assez similaires, mais il faut coordonner la génération du cantus firmus, la réalisation du contrepoint et la lecture tout en gardant en tête la question du score.

On ne peut plus appliquer le principe de l'algorithme glouton comme précédemment, puisqu'une fois qu'on a joué des mesures, on est obligés de compléter celles-ci. Cela revient à avoir `nb_glout_garde = 1`.

Le principe est le suivant :

- On commence par les trois mesures (`[0, 2, 4]`, `[12, 11, 12]`)
- On répète ensuite indéfiniment ces étapes :
  - On génère les `nb_mesures` suivantes du cantus firmus avec les chaînes de Markov
  - On génère toutes les `voix2` possibles, puis on en conserve une de score maximal
  - On joue cela puis on recommence (en gardant en mémoire les trois dernières mesures)

Cela nécessite de modifier `contrepoint_valide` pour qu'elle ne fasse plus de vérifications exceptionnelles sur le début du contrepoint et sur les deux dernières mesures, le contrepoint généré étant potentiellement infini.

[ANNEXE 9]

Des essais montrent que la génération se fait suffisamment vite pour qu'on puisse jouer du contrepoint de première espèce en temps réel à 400bpm, et la qualité reste suffisante.

III. Autres solutions et développements possibles

Pistes à explorer

Les récents développements du *machine learning* montrent que les machines sont de plus en plus performantes pour réaliser des tâches typiquement humaines : avec l'aide de réseaux de neurones, on peut aujourd'hui générer des tableaux dans le style des grands maîtres ou battre l'humain au jeu de Go[3].

Des méthodes d'analyse et de reproduction du style de grands contrapuntistes par des réseaux neuronaux pourraient donner de la musique de grande qualité. Cependant, ces algorithmes sont compliqués et requièrent (surtout pour le *deep learning*) une quantité gigantesque de données et d'exemples. On pourrait par exemple entraîner un réseau pour trouver les valeurs optimales des coefficients affectés aux différents scores.

Projets similaires et travaux universitaires

Quelques recherches m'ont permis de découvrir des initiatives similaires :

- L'application Android *FuX* (gratuite) de D. Herremans : génère un flot ininterrompu de contrepoint généré en continu, dans le style d'un compositeur qu'on peut choisir. Elle repose sur l'algorithme VNS (« Variable Neighbourhood Search algorithm », cf. infra). Les rythmes sont variés, mais assez chaotiques à l'écoute. Le réglage du compositeur est par ailleurs décevant.
- *melody.py* de A. Nisnevich[4] : génère des cantus firmi de dix mesures puis trouve par la force brute un contrepoint convenable par dessus. Les temps sont de l'ordre du dixième de

3 Voir *https://twitter.com/deepforger* et *https://fr.wikipedia.org/wiki/AlphaGo* ;*voir aussi https://www.youtube.com/watch?v=trWrEWfhTVg et/ou https://www.youtube.com/watch?v=RgUcQceqC_Y*

4 On peut le tester depuis un navigateur Internet en allant sur http://melodypy.com/

secondes, comme les miens. L'auteur a rajouté des règles arbitraires très intéressantes pour rendre les mélodies plus cohérentes. Les résultats sont corrects.

- *Contrapunctus* de Bai Li : semble reposer sur des principes assez similaires à mon programme, mais a l'avantage de générer des rythmes variés. Les résultats sont bons quoique robotiques.
- *Hyperscore* du MIT : un programme d'aide à la composition. Les pages 37 à 45 de la publication issue de ce projet (cf. bibliographie) mentionnent la génération de contrepoint dans le style de Palestrina à l'aide de chaînes de Markov. On y retrouve un historique très intéressant, où l'on apprend qu'il est utile d'ajouter des règles supplémentaires, par exemple « si la mélodie contient un saut, il faut que toutes les notes sautées apparaissent à un moment ». Le travail est très détaillé et donne de bonnes idées d'améliorations.

Par ailleurs, Dorien Herremans de l'université d'Anvers (à l'origine de l'application FuX) publie régulièrement sur son profil *academia.edu* des travaux de recherche sur le sujet, tels que :

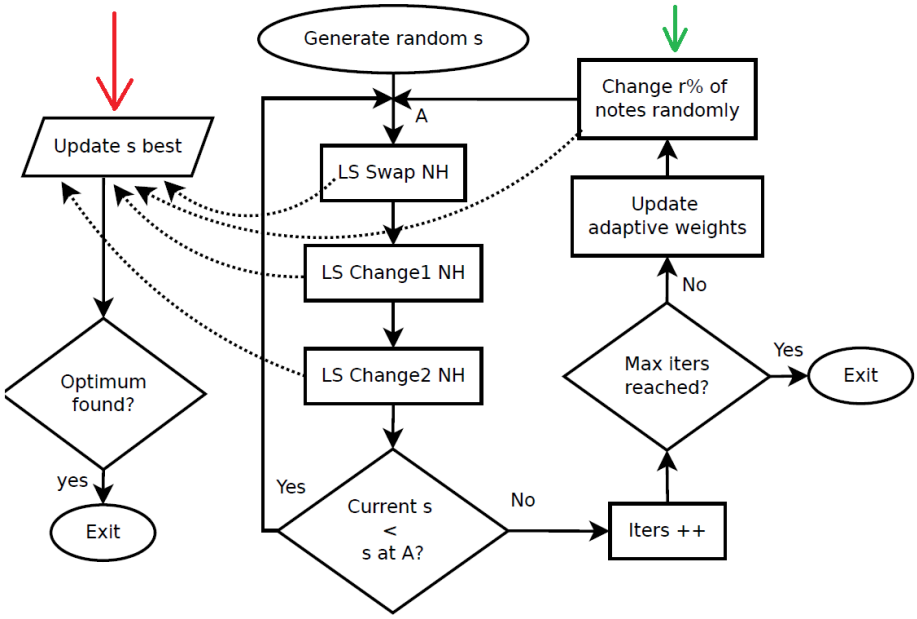
- « *Generating structured music using quality metrics based on Markov models* », qui utilise de l'analyse d'exemples et des chaînes de Markov pour générer une structure cohérente
- « *Classification and generation of composer-specific music using global feature models and VNS* » qui cherche à analyser le style de compositeurs pour les imiter
- « *FuX, an Android app that generates counterpoint* » (cf. bibliographie), où on trouve notamment une explication de l'algorithme VNS, dont ce schéma :

"VNS systematically changes the neighborhood in two phases: firstly, descent to find a local optimum and finally, a perturbation phase to get out of the corresponding valley."

"It starts from an initial solution s and iteratively makes small improvements (or moves) to the solution in order to find a better one."

Name	Description
Swap	Swap two notes
Change1	Change one note
Change2	Change two notes

LS = Local Search
NH = Neighbourhood
(toutes les solutions obtenues en faisant le changement indiqué)



Cette recherche m'a appris plusieurs choses :

- Mon approche (introduction d'un score, backtracking, chaînes de Markov) n'est pas exotique du tout, mais est celle de tous ceux qui ont réfléchi à la question.
- Ce qui manque à mon programme est surtout de la variété dans les rythmes, des altérations et peut-être des règles supplémentaires.

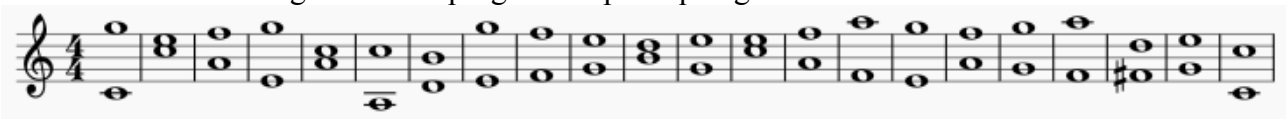
Conclusion

Il est possible pour un algorithme assez simple de générer du contrepoint en temps raisonnable, les règles empêchant les inélégances ; le principal défaut est surtout le manque de variété et de cohérence, qu'on peut probablement résoudre en utilisant des méthodes plus poussées d'analyse et de génération, notamment pour le rythme, et en donnant à la musique une structure respectant la grammaire contrapuntique, avec des répétitions, des modulations et des accidents.

Le principe est cependant critiquable, comme le montre un parallèle avec la langue : c'est

comme si on avait un programme générant des phrases grammaticalement impeccables, mais sans contenu sémantique.

Pour finir, voici un contrepoint sur le cantus firmus de *Jesus bleibet meine Freude* de Bach, obtenu en modifiant légèrement le programme pour qu'il gère l'altération :



22 mesures, généré en 500,42ms, score de 1027

Références et bibliographie

Gedalge A., *Traité de Contrepoint*, vers **1920**
Farbood M., *Hyperscore: A New Approach to Interactive, Computer-Generated Music*, MIT, **2001**
Herremans D., Sörensen K., *FuX, an Android app that generates counterpoint*, University of Antwerp, **2013**
Documentation de *melody.py* : https://github.com/AlexNisnevich/melody.py
Le projet *Contrapunctus* : https://code.google.com/archive/p/contrapunctus/

Pour écrire les MIDI → *MIDIUtil* : http://www.emergentmusics.org/midiutil
Pour lire les MIDI → *playsmf.exe* : http://osspack32.googlecode.com/files/playsmf.exe

ANNEXE 1 : Score d'une voix

```
gamme = [0, 2, 4, 5, 7, 9, 11]  # notes de la gamme majeure

# Intervalles mélodiques acceptables, dans l'ordre de préférence
int_melo = [2, 1, -2, -1, 0, 3, -3, 4, -4, 5, -5, 7, -7, -8, 8, -9, 9]


# Valeurs des intervalles mélodiques, dans l'ordre croissant de demi-tons
val_melo = [8, 12, 13, 4, 4, 3, 0, 2, 2, 1]


def voix_valide(voix, debut=0, fin=-1):
    """ * Vérifie une voix entre debut et fin (jusqu'au bout si -1)
        * Renvoie son score (-1 ssi la voix n'est pas valide)

        * La fonction est telle que pour 0 <= a < b < c <= n, on ait :
          voix_valide(v,a,b) + voix_valide(v,b,c) = voix_valide(v,a,c)
          lorsque la voix est valide entre a et c

        * On fait toujours les vérifications avec les mesures précédentes,
          et non suivantes, de manière à ce que qu'on puisse rajouter
          des mesures à la fin de la voix.
    """
    n = len(voix)
    if debut < 0:
        debut = 0
    if fin == -1 or fin > n:
        fin = n
    if debut >= fin:
        return 0  # rien à vérifier

    score = 0
    for i in range(debut, fin):
        if (voix[i] % 12) not in gamme:
            return -1
        elif i >= 2 and voix[i-2] % 12 == voix[i-1] % 12 == voix[i] % 12:
            return -1  # même note trois fois de suite
        elif i >= 1:
            if (voix[i]-voix[i-1]) not in int_melo:
                return -1  # intervalle mélodique interdit
            else:
                score += val_melo[abs(voix[i]-voix[i-1]) % 12]
    return score
```

ANNEXE 2 : Score d'un contrepoint

```
tierces = [3, 4, 15, 16]  # tierces
sixtes = [8, 9]  # sixtes
consonances_parfaites = [12, 0, 7, 19]  # consonances parfaites

# Intervalles harmoniques acceptables, dans l'ordre de préférence
int_harmo = [4, 3, 9, 8, 7, 16, 15, 12, 19]

# Int harmo acceptés resp à la 1ère, l'avant-dernière, la dernière mesure
int_harmo_debut = consonances_parfaites
int_harmo_penul = [9, 3, 15]
int_harmo_fin = [12, 0]

# Valeurs des intervalles harmoniques, dans l'ordre croissant de demi-tons
val_harmo = [0, 0, 0, 7, 7, 0, 0, 3, 6, 7, 0, 0, 3, 0, 0, 5, 5, 0, 0, 1]
```

```python
# Coefficients des scores mélo, harmo, de mouvement, de diversité
coeff_melo = 1    # pour chaque voix
coeff_harmo = 2   # pour chaque mesure
coeff_mvt = 5     # *3 pour inverse, *1 pour oblique, *0 pour direct
coeff_div = 1     # score de "diversité"  = nombre de notes distinctes

nb_mesures = 5


def signe(n):
    """ Renvoie 1 si n > 0, -1 si n < 0 et 0 si n == 0 """
    if n < 0:
        return -1
    elif n == 0:
        return 0
    else:
        return 1


def score_div(voix):
    """ Calcule le score de diversité d'une voix """
    s = 0
    for i in range(0, len(voix), nb_mesures):
        frac = voix[i:i+nb_mesures]
        s += len(set(frac)) * nb_mesures
    return int(coeff_div * s)


def contrepoint_valide(voix1, voix2, chk1=True, chk2=True, debut=0, fin=-1):
    """ * Vérifie un contrepoint entre debut et fin (jusqu'au bout si -1)
        * Renvoie son score (-1 ssi le contrepoint n'est pas valide)

        * On vérifie la validité de la voix inférieure ssi chk1 = True
        * On vérifie la validité de la voix supérieure ssi chk2 = True

        * La fonction est telle que pour 0 <= a < b < c <= n, on ait :
          contrepoint_valide(v,a,b) + contrepoint_valide(v,b,c)
                                                = contrepoint_valide(v,a,c)
          lorsque le contrepoint est valide entre a et c

        * On fait toujours les vérifications avec les mesures précédentes,
          et non suivantes, de manière à ce que qu'on puisse rajouter
          des mesures à la fin de la voix.
    """
    ntotal = max(len(voix1), len(voix2))
    n = min(len(voix1), len(voix2))
    if debut < 0:
        debut = 0
    if fin == -1 or fin > n:
        fin = n
    if ntotal < 3:
        return -1  # cantus firmus trop court
    elif debut >= fin:
        return 0  # rien à vérifier

    score_melo, score_harmo, score_mvt = 0, 0, 0

    # Vérifications de voix :
    if chk1:  # vérification de la voix inférieure
        score_melo1 = voix_valide(voix1, debut, fin)
        if score_melo1 == -1:
            return -1
        else:
            score_melo += score_melo1
```

```
    del score_melo1

if chk2:  # vérification de la voix supérieure
    score_melo2 = voix_valide(voix2, debut, fin)
    if score_melo == -1:
        return -1
    else:
        score_melo += score_melo2
    del score_melo2

# Intervalles harmoniques entre les voix, puis mélodiques de chaque voix :
harmo = {k: voix2[k]-voix1[k] for k in range(max(debut-3, 0), fin)}
melo1 = {k: voix1[k+1]-voix1[k] for k in range(max(debut-2, 0), fin-1)}
melo2 = {k: voix2[k+1]-voix2[k] for k in range(max(debut-2, 0), fin-1)}

if debut == 0 and harmo[0] not in int_harmo_debut:
    return -1  # première mesure

if fin >= ntotal-1 and harmo[ntotal-2] not in int_harmo_penul:
    return -1  # avant-dernière mesure

if fin == ntotal and harmo[ntotal-1] not in int_harmo_fin:
    return -1  # dernière mesure

# Vérifications harmoniques (sauf la première et les 2 dernières mesures) :
for i in range(max(1, debut), min(fin, ntotal-2)):
    if harmo[i] not in int_harmo:
        return -1  # intervalles interdits

    if harmo[i-1] % 12 == harmo[i] % 12 in consonances_parfaites:
        return -1  # deux consonances parfaites de suite

for i in range(debut, fin):
    score_harmo += val_harmo[harmo[i]]  # score harmonique

# Vérifications relatives aux répétitions
for i in range(debut, fin):
    if i >= 1:
        if 6 in ((voix2[i]-voix1[i-1]) % 12, (voix1[i]-voix2[i-1]) % 12):
            return -1  # fausse relation de triton

    if i >= 2:
        if ((voix1[i] in [voix1[i-1], voix2[i-1]] and
             voix1[i] in [voix1[i-2], voix2[i-2]]) or
            (voix2[i] in [voix1[i-1], voix2[i-1]] and
             voix2[i] in [voix1[i-2], voix2[i-2]])):
            return -1  # note répétée trois fois

    if i >= 3:
        if (harmo[i-3] in tierces and harmo[i-2] in tierces and
           harmo[i-1] in tierces and harmo[i] in tierces):
            return -1  # quatre tierces de suite
        elif (harmo[i-3] in sixtes and harmo[i-2] in sixtes and
              harmo[i-1] in sixtes and harmo[i] in sixtes):
            return -1  # quatre sixtes de suite

# Vérifications de mouvement
for i in range(max(debut-1, 1), fin-1):
    mvt = signe(melo1[i-1])*signe(melo2[i-1])
    if mvt == 1 and harmo[i] in consonances_parfaites:
        return -1  # consonance parfaite directe
    elif mvt == -1:
        score_mvt += 3  # mouvement contraire
    elif mvt == 0:
```

```
            score_mvt += 1  # mouvement oblique

    return (coeff_melo * score_melo + coeff_harmo * score_harmo +
            coeff_mvt * score_mvt)


def score(voix1, voix2, chk1=True, chk2=True):
    """ * Renvoie le score total d'un contrepoint.

        * On vérifie la validité de la voix inférieure ssi chk1 = True
        * On vérifie la validité de la voix supérieure ssi chk2 = True
    """
    s = contrepoint_valide(voix1, voix2, chk1, chk2)
    if s == -1:
        return -1
    else:
        if chk1:
            s += score_div(voix1)
        if chk2:
            s += score_div(voix2)
        return s
```

ANNEXE 3 : Candidats

```
def candidats2(voix1, voix2):
    """ Prend un contrepoint inachevé et renvoie des candidats potentiels pour
        compléter la voix2
    """
    res = []
    n1 = len(voix1)
    n2 = len(voix2)
    if n1 > n2:
        if n2 == 0:
            res = [voix1[0]+i for i in int_harmo_debut]
        elif n2 == n1-2:
            res = [voix1[n2]+i for i in int_harmo_penul]
        elif n2 == n1-1:
            res = [voix1[n2]+i for i in int_harmo_fin]
        else:
            sens = 1 if (voix1[n2] > voix1[n2-1]) else -1
            for i in int_melo:
                candidat = voix2[n2-1] - i*sens
                if candidat-voix1[n2] in int_harmo:
                    res.append(candidat)
    return [c for c in res if c % 12 in gamme]
```

ANNEXE 4 : Backtracking intelligent

```
def ecrire2(voix1, voix2=[]):
    """ Reçoit une voix1 complète et renvoie une voix2 convenable. """
    if contrepoint_valide(voix1, voix2, False, True, len(voix2)-1) == -1:
        return None
    if len(voix2) >= len(voix1):
        return voix2
    else:
        for c in candidats2(voix1, voix2):
            res = ecrire2(voix1, voix2+[c])
            if res is not None:
                return res
        return None
```

ANNEXE 5 : Algorithme naïf pour un score maximal

```
def ecrire2_all(voix1, voix2=[]):
    """ Prend une voix1 complète, une liste L de voix2 incomplètes, et renvoie
        toutes les possibilités de voix2 obtenues grâce à candidats.
    """
    if contrepoint_valide(voix1, voix2, False, True, len(voix2)-1) == -1:
        yield from []
    if len(voix2) >= len(voix1):
        yield voix2
    else:
        for c in candidats2(voix1, voix2):
            yield from ecrire2_all(voix1, voix2+[c])


def ecrire2(voix1):
    """ Reçoit une voix1 complète et renvoie la voix2 convenable
        de score maximal. """
    L = sorted(list(ecrire2_all(voix1)),
               key=lambda e: -score(voix1, e, False, True))
    if len(L) == 0:
        print("Pas de solution !...")
    else:
        return L[0]
```

ANNEXE 6 : Algorithme glouton

```
nb_mesures = 5  # nb de mesures générées à chaque étape de l'algo glouton
nb_glout_garde = 10  # nb de versions gardées à chaque étape de l'algo glouton


def ecrire2_next(voix1, L, nb):
    """ Reçoit une voix1 complète, une liste L de (voix2, score) incomplètes,
    et renvoie les (voix, score) obtenues après génération de nb mesures.
    """
    for v in L:
        if len(v[0]) == len(voix1) or nb == 0:
            yield (v[0], v[1] + score_div(v[0][-nb_mesures:]))
        else:
            for c in candidats2(voix1, v[0]):
                subscore = contrepoint_valide(voix1, v[0]+[c], False, True,
                                              len(v[0]))
                if subscore == -1:
                    yield from []
                else:
                    yield from ecrire2_next(voix1,
                                            [(v[0]+[c], v[1]+subscore)], nb-1)


def ecrire2(voix1):
    """ Reçoit une voix1 complète et renvoie la voix2 obtenue par l'algorithme
        glouton.
    """
    L = [([], 0)]
    while len(L[0][0]) < len(voix1):
        L = sorted(list(ecrire2_next(voix1, L, nb_mesures)),
                   key=lambda e: -e[1])[0:nb_glout_garde]
    return L[0][0]
```

ANNEXE 7 : Contrepoint renversable

```
consonances_parfaites = [12, 0]  # consonances parfaites

# Intervalles harmoniques acceptables, dans l'ordre de préférence
int_harmo = [9, 8, 16, 15, 12]

# Int harmo acceptés resp à la 1ère, l'avant-dernière, la dernière mesure
int_harmo_debut = [12]
int_harmo_penul = [9, 15]
int_harmo_fin = [12, 0]


def transpose(voix, d):
    """ Transpose la voix de d demitons """
    return [i+d for i in voix]


def contrepoint_valide(voix1, voix2, chk1=True, chk2=True, debut=0, fin=-1):

      """ ... (cf. annexe 2) """

    # Vérifications harmoniques (sauf la première et les 2 dernières mesures) :
    for i in range(max(1, debut), min(fin, ntotal-2)):
        if harmo[i] not in int_harmo:
            return -1  # intervalles interdits

        if harmo[i-1] % 12 == harmo[i] % 12 in consonances_parfaites:
            return -1  # deux consonances parfaites de suite

        score_harmo += (val_harmo[harmo[i]] + val_harmo[24-harmo[i]])//2

      """ ... (cf. annexe 2) """
```

ANNEXE 8 : Génération aléatoire

```
from random import randint

# Couples de notes successives conduisant à la fin du cantus firmus
vers_fin = [(-1, 0), (2, 0), (11, 12), (-10, -12)]


def matrice_markov():
    """ Lit le fichier transi.txt et renvoie la matrice de transition de Markov
        correspondante
    """
    f = open('transi.txt', 'r')
    L = f.readlines()
    f.close()
    M = []
    for i in L:
        r = []
        nI = i.strip().split(";")
        for j in nI:
            r.append(int(j))
        M.append(r)
    return M


def case_suivante(M, i):
    """ Prend une matrice de Markov M et un indice de ligne i et renvoie, selon
        les coefficients de la ligne i, une case aléatoire à partir de i
    """
    proba_cumulees = []
    somme = 0
```

```
    for j in M[i]:
        somme += j
        proba_cumulees.append(somme)
    aleatoire = randint(1, somme)
    k = 0
    while k < len(M[i]) and proba_cumulees[k] < aleatoire:
        k += 1
    return k


def generer_cantus(M):
    """ Prend une matrice de transition de Markov M et renvoie un cantus_firmus
        généré avec celle-ci
    """
    cantus_firmus = [0]
    etat_actuel = 0
    etat_precedent = 0
    continuer = True
    while continuer:
        if ((etat_precedent, etat_actuel) in vers_fin and
           len(cantus_firmus) >= 5 and randint(0, 1) == 1):
            continuer = False
        else:
            suivante = etat_precedent
            while suivante == etat_precedent:
                suivante = case_suivante(M, etat_actuel + 12) - 12
            (etat_precedent, etat_actuel) = (etat_actuel, suivante)
            cantus_firmus.append(etat_actuel)
    return cantus_firmus
```

ANNEXE 8bis : Analyse d'exemples

```
f = open('transi.txt', 'w')
l = "0;"*24+"0"
r = (l+"\n")*24+l
f.write(r)
f.close()

ex = open('exemples.txt', 'r')
E = ex.readlines()
ex.close()

for e in E:
    en = e.split(',')
    exemple = []
    for j in en:
        exemple.append(int(j))

    f = open('transi.txt', 'r')
    L = f.readlines()
    f.close()
    M = []
    for i in L:
        r = []
        nI = i.strip().split(";")
        for j in nI:
            r.append(int(j))
        M.append(r)

    for i in range(len(exemple)-1):
        M[exemple[i]+12][exemple[i+1]+12] += 1

    f = open('transi.txt', 'w')
```

```
    r = ""
    for i in range(24):
        for j in range(24):
            r += str(M[i][j])+";"
        r += str(M[i][24])+"\n"
    for j in range(24):
        r += str(M[24][j])+";"
    r += str(M[24][24])

    f.write(r)
    f.close()
```

ANNEXE 9 : Temps réel

(note : `play` est une fonction jouant un nombre indéterminé de voix simultanément)

```
from random import randint

nb_mesures = 5


def contrepoint_valide(voix1, voix2, chk1=True, chk2=True, debut=0, fin=-1):
    """ ... """

    # Intervalles harmoniques entre les voix, puis mélodiques de chaque voix :
    harmo = {k: voix2[k]-voix1[k] for k in range(max(debut-3, 0), fin)}
    melo1 = {k: voix1[k+1]-voix1[k] for k in range(max(debut-2, 0), fin-1)}
    melo2 = {k: voix2[k+1]-voix2[k] for k in range(max(debut-2, 0), fin-1)}

      """ /!\ Plus de vérifications exceptionnelles /!\ """

    # Vérifications harmoniques :
    for i in range(debut, fin):
            """ ... """

    # Vérifications relatives aux répétitions
    for i in range(debut, fin):
            """ ... """

      """ ... """

def ecrire2_next(voix1, v, nb):
    """ Reçoit une voix1 complète, une voix2 incomplète,
    et renvoie les voix2 obtenues après génération de nb mesures.
    """
    if len(v) == len(voix1) or nb == 0:
        yield v
    else:
        for c in candidats2(voix1, v):
            yield from ecrire2_next(voix1, v+[c], nb-1)


def ecrire2TR(trois_prec_cf, trois_prec_mel, nextcf):
    """ Reçoit une voix1 complète et renvoie une voix2 correspondante avec un
    score maximisé par un algorithme glouton. """
    voix1 = trois_prec_cf + nextcf
    L = sorted(list(ecrire2_next(voix1, trois_prec_mel, nb_mesures)),
               key=lambda e: -score(voix1, e))
    return L[0]


M = matrice_markov()
trois_prec_cf = [0, 2, 4]
trois_prec_mel = [12, 11, 12]
```

```
play(trois_prec_cf, trois_prec_mel)
tps_debut = clock()
wait = 3*4*60/tempo
note_courante_cf = 4
note_prec_cf = 2
while True:
    nextcf = []
    for i in range(nb_mesures):
        note_suivante_cf = note_prec_cf
        while note_suivante_cf == note_prec_cf:
            note_suivante_cf = case_suivante(M, note_courante_cf + 12) - 12
        note_prec_cf = note_courante_cf
        note_courante_cf = note_suivante_cf
        nextcf.append(note_courante_cf)
    nextmel = ecrire2TR(trois_prec_cf,
                        trois_prec_mel, nextcf)[-nb_mesures:]
    trois_prec_cf = nextcf[-3:]
    trois_prec_mel = nextmel[-3:]
    note_courante_cf = nextcf[-1]
    print(nextcf, nextmel)
    while clock()-tps_debut < wait:
        pass
    play(nextcf, nextmel)
    tps_debut = clock()
    wait = nb_mesures*4*60/tempo
```

ANNEXE 10 : Mesures effectuées

Mesures de temps :

Les cantus firmi utilisés sont les `[0, 7, 9, 5, 4, -1]*N+[0]`, où j'ai fait varier `N`.

Pour les mesures sur `contrepoint_valide`, j'ai pris comme voix supérieure `[12, 11, 12, 14, 12, 14]*N+[0]`.

Tous les temps sont données en millisecondes.

Taille	7	13	19	25	31	61	301	601	3001	6001	60001	600001
voix_valide	0,21	0,21	0,35	0,44	0,38	0,33	0,7	1,06	4,45	9,23	90,1	923
contrepoint_valide	0,35	0,31	0,4	0,52	0,64	0,7	2,52	4,86	23,6	46,6	481	4883
Génération naïve d'une solution	0,5	0,74	0,88	1,44	1,59	2,77	12,9	27,4	*N'a pas pu être mesuré*			
Génération naïve de la solution maximale	301	*N'a pas pu être mesuré*										
Algorithme glouton	15,8	11,42	236	401	519	1182	6640	14425	96108	254777	*N'a pas pu être mesuré*	

Comparaison entre l'algorithme naïf et l'algorithme glouton :

Cantus firmus		**[0, 2, 0]**	**[0, 2, -1, 0]**	**[0, 2, 4, -1, 0]**	**[0, 2, 4, 2, -1, 0]**	**[0, 2, 4, 5, 2, -1, 0]**	**[0, 2, 4, 5, 4, 2, -1, 0]**	**[0, 2, 4, 5, 7, 4, 2, -1, 0]**	**[0, 2, 4, 5, 7, 5, 4, 2, -1, 0]**
Taille		**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Naïf (maximal)	**Temps**	1,71	4,18	17,1	78,3	356,3	1566	6833	29391
	Score	111	154	207	266	296	366	405	475
Glouton	**Temps**	1,29	1,8	6,6	11,79	18,5	22,8	37,17	74,7
	Score	111	154	207	265	296	364	405	475