Faculté de Sciences de Montpellier

Master 1 AIGLE
2016 – 2017

Projet tuteuré

Calcul et écoute de gammes musicales adaptées à une suite d'accords donnée

Rapport de projet

Étudiants:
Jean-Daniel Bargy
Mattéo Coquilhat
Vincent Iampietro
Olivier Montes
Sacha Weill

Encadrant:
Emeric Gioan

Sommaire

Introduction

L'algorithmique est une science théorique présentant de nombreux problèmes inspirés de notre quotidien. Dans le cadre de ce projet de TER, nous nous sommes penchés sur un problème d'algorithmique combinatoire dans le cadre de la musique théorique. Plus précisément, nous avons travaillé sur la réalisation d'un logiciel ayant pour fonction la recherche et la présentation d'une suite de gammes, la contrainte étant que chacune de ces gammes contienne un accord d'une liste préalablement donnée.

La problématique de ce projet repose sur deux branches distinctes, le travail de recherche et la programmation d'un logiciel.

La principale motivation du projet est la réalisation d'un algorithme optimisé répondant à une problématique précise: La résolution d'un problème de combinatoire algorithmique dans le cadre de concepts musicaux. Les enjeux de cette problématique sont nombreux.

En parallèle, ce projet nécessite la réalisation d'un logiciel ciblant les musiciens et adeptes des concepts manipulés. Il s'agit donc de réaliser un logiciel ergonomique et clair, permettant son utilisation par des utilisateurs néophytes tant en informatique qu'en musique théorique. Le logiciel devait en addition être réalisé de manière à simplifier le changement entre les différents algorithmes recherchés, et permettre l'obtention de statistique sur les performances de ces algorithmes.

Afin de présenter le travail effectué au cours de ce projet, ce rapport sera divisé comme suit:

- Dans un premier temps sera présenté le sujet ainsi que notre analyse des consignes. Les différents concepts de musique et de programmation nécessaires à la compréhension de notre travail seront définis, et le cahier des charges sera détaillés précisément, tel qu'il a été convenu avec notre encadrant.
- Une fois les consignes claires et assimilées, nous présenterons le travail de recherche effectué lors du projet. Le but étant l'optimisation d'un algorithme, nous présenterons les différentes étapes d'avancement ainsi que les diverses références ayant aidé à la création de ces algorithmes.
- Enfin, le travail de programmation réalisé sera présenté de manière concise et claire afin de présenter l'évolution du logiciel au cours de sa création.
- Nous finirons par conclure avec une présentation du logiciel fini, implémentant l'algorithme recherché.

Ce découpage du rapport est à l'image de notre mode de travail sur l'ensemble du projet, avec la séparation nette du travail de recherche et de la programmation du logiciel, et pour finir la fusion des deux branches pour répondre au mieux aux consignes présentées.

Analyse du projet

Énoncé initial

Un accord musical est vu ici comme n'importe quel ensemble de notes (3 ou 4 typiquement, 5 pour des accords colorés) parmi l'ensemble des 12 notes de la gamme chromatique. Les gammes sont vues comme des ensembles prédéfinis de notes (7 pour les gammes majeures, 5 pour les pentatoniques, 8 pour les diminuées, etc.) parmi ces mêmes 12 notes. Il s'agit de concevoir un programme qui prend en entrée une suite d'accords, et produit en sortie une (ou plusieurs) suite de gammes adaptée à la suite d'accords, de sorte que :

- *Les notes des accords fassent partie des notes de la gamme associée*
- *une même gamme soit utilisée sur le plus d'accords consécutifs possibles (ou bien que le moins de gammes possibles soient utilisées au total).*

Il peut y avoir bien sûr de nombreuses solutions possibles, et on peut aussi s'amuser à imposer plus ou moins de contraintes.

Sur le plan théorique, il s'agira de modéliser et résoudre ce problème à la fois combinatoire et algorithmique (des solutions simples seront faciles à trouver, mais on pourra aussi en concevoir de plus complexes). Sur le plan informatique, il s'agira d'implémenter proprement ces objets et solutions, et de permettre leur écoute en produisant un fichier MIDI (qui jouera simultanément les accords et les gammes associées). Sur le plan musical, l'idée est que les gammes produites puissent être utilisées pour de l'improvisation ou de la composition sur la grille d'accords donnée.

1 Définitions

Nous présentons dans cette section les éléments essentiels à la compréhension de notre problème. Ces éléments sont issus du domaine de la théorie musicale qui en font une définition que nous avons adapté et formalisé à notre manière, pour rendre plus facile la manipulation de ces objets.

1.1 Note

Dans la théorie de la musique, une *note* permet de représenter un son de manière standardisée. Une note est séparée d'une autre par un intervalle exprimé en Hertz. Le plus petit intervalle (dans le système occidental) qui sépare deux notes différentes est appelé un demi-ton. Dans notre système d'étude, une note est un élément de base des ensembles manipulés.

Gamme chromatique L'ensemble des notes sur lequel nous nous baserons est appelé la gamme chromatique. La gamme chromatique est composée de douze notes chacune séparée par un demi-ton. On se référera à la gamme chromatique par le symbole N tout au long du rapport.
N.B. La structure d'une gamme est circulaire. Ainsi, la note 11 est séparée par un demi-ton de la note 0 dans l'exemple du tableau 2.1.

Du nommage des notes Le tableau 2.1 présente la structure de la gamme chromatique (appelée aussi *échelle musicale*). Chaque note de la gamme chromatique est séparée de ses voisines par un demi-ton.

	1/2 ton		1/2 ton		1/2 ton		1/2 ton		1/2 ton		1/2 ton
0		1		2		3		4		5	
A		$A\#$		B		C		$C\#$		D	
		$B\flat$						$D\flat$			
	1/2 ton		1/2 ton		1/2 ton		1/2 ton		1/2 ton		1/2 ton
6		7		8		9		10		11	
$D\#$		E		F		$F\#$		G		$G\#$	
$E\flat$						$G\flat$				$A\flat$	

Table 2.1 – Échelle musicale de la gamme chromatique

Chaque note de la gamme est nommée selon le modèle $Base[alt]$ où $Base \in \{A, B, C, D, E, F, G\}$ (notation internationale des notes, équivalente dans l'ordre à $\{La, Si, Do, Re, Mi, Fa, Sol\}$) et $alt \in \{\#, \flat\}$.

L'altération # augmente sa base d'un demi-ton tandis que l'altération $\flat$ diminue sa base d'un demi-ton.

Ainsi, on peut associer deux noms à une même note, voir plus avec l'addition de nouvelles altérations. Pour éviter ce problème, nous considérons qu'à chaque note est associée un entier n (avec $0 \leq n \leq 11$) comme présenté dans le tableau 2.1.

Ainsi la gamme chromatique est représentée par l'ensemble des entiers de 0 à 11.

1.2 Ensemble de notes

Nous définissons intuitivement un ensemble de notes comme étant composé, sans ordre ni répétition, par les notes de la gamme chromatique.

Accords

Définition : Un *accord* est une partie de N dont le cardinal est supérieur ou égal à 3.

Un accord est généralement[1] nommé par le biais de deux paramètres :

- La note *fondamentale*, qui est un élément de N est la note sur laquelle toutes les autres se basent et se construisent
- La *structure harmonique*, la structure des notes suivant la fondamentale. On représentera cette structure par une suite d'intervalles entre une note et la note qui la précède (sachant que la note fondamentale précède la première note de la structure harmonique).

N.B. Le nom de l'accord est déterminé par ces deux paramètres par une convention musicale; ainsi, même si l'accord dont la fondamentale est Do et la structure harmonique Ré, Mi, La, Si peut être représenté par le système de description d'accord ci-dessus, aucune convention musicale ne permet de nommer un tel accord.

Exemple de composition d'un accord : Détaillons la construction d'un accord selon la définition que nous en avons faite.

- Un accord est d'abord caractérisé par le nom de sa note principale (ou fondamentale), point de départ de sa construction. Nous choisirons la note F (fa) comme note fondamentale pour notre exemple, en rappelant que la note F est représentée dans notre système par le chiffre 8.
- Ensuite, un accord est caractérisé par sa structure. Une structure est une suite d'intervalles (exprimées en demi-tons) permettant en partant de la note fondamentale de déterminer quelles notes font partie de l'accord. Pour notre exemple, l'accord de F est un accord majeur.

  Sa structure est la suivante : *4 demi-tons – 3 demi-tons*

  En partant de F donc du chiffre 8, on ajoute 4 demi-tons, ce qui donne $(8 + 4) \mod 12 = 0$.

  Puis, on ajoute 3 demi-tons au chiffre 0, pour obtenir $(0 + 3) \mod 12 = 3$.

  Par conséquent, l'accord de F majeur est l'ensemble $\{8, 0, 3\}$, ce qui en terme de notes correspond à $\{F, A, C\}$.

  N.B. Appliquer le *modulo 12* aux opérations de calcul permet de prendre en compte le caractère circulaire de la structure d'une gamme, en sachant que l'ensemble N qui est notre ensemble de référence est composé de douze notes.

[1] Il existe des accords sans nom

Dictionnaire des accords : On part du principe qu'un accord est composé d'au moins trois sons (trois notes). En effet, pour un accord à deux sons, on parle d'intervalle, à un son, de note, et à zéro, de silence.

Le dictionnaire des accords qui référence tous les accords possibles est composé de l'ensemble des parties de N dont le cardinal est supérieur ou égal à 3.

Soit DA l'ensemble représentant le dictionnaire des accords, on obtient :

$|DA| = 2^{|N|} - \sum_{p=0}^{3} \binom{|N|}{p}$, avec $|N| = 12$, donc $|DA| = 4017$ accords différents.

Gamme :

Une gamme G est une partie de N. Elle est, comme pour un accord, définie par un nom qui correspond à sa note *tonique* (ou fondamentale) et par une structure appelée *échelle musicale*. La différence entre gamme et accord réside dans la notion d'inclusion, car le plus souvent un accord est inclus dans une gamme (sauf cas particulier). En général, est vérifié : $A \subseteq G$, avec A l'ensemble des accords.

Dictionnaire des gammes : Contrairement à un accord, une gamme peut-être composée de un, ou deux éléments, mais pas de zéro.

Le dictionnaire des gammes DG référence cependant uniquement les *gammes autorisées*, c'est à dire possédant un nom et une définition connue et donnée par la théorie de la Musique (gamme majeure naturelle, mineure harmonique, mineure pentatonique, ...).

Pour résumé, DG est un ensemble donné des parties de N correspondant aux *gammes autorisées*.

Notre dictionnaire de gammes est composé des gammes suivantes :

{*Mineure harmonique, Mineure mélodique, Majeure, Pentatonique mineure, Pentatonique blues, Pentatonique majeure, Égyptienne, Bartok, Par ton, Diminuée*}

2 Cahier des charges

2.1 Description formelle du problème

Entrée Soit une suite d'accords $SA = (A_1, .., A_n)$

Sortie Une suite de gammes $SG = (G_1, .., G_n)$ telle que :

- Chaque gamme G_i de SG correspond à un accord A_i de la suite SA, c'est à dire la suite SG va de G_1 à G_n
- La gamme associée à un accord contient cet accord, c'est a dire pour tout i on a $A_i \subseteq G_i$
- Les gammes utilisées sont dans l'ensemble des gammes autorisées DG, c'est à dire pour tout i on a $G_i \in DG$
- La suite SG minimise un certain paramètre entier $f(SG)$ associé à une suite de gammes.
  Il a été convenu lors du projet que les trois paramètres à prendre en compte pour la suite SG serait :
  1. Le nombre de changements de gammes
  2. Le nombre total de gammes
  3. Le nombre de changements de notes entre deux gammes consécutives

Suite de gammes solution : Par la suite, on appellera ***suite de gammes solution***, les suites de gammes que l'on veut trouver en sortie de notre calcul et qui répondent aux mêmes caractéristiques que SG vis à vis d'une suite d'accords SA. On englobe dans cette définition les suites de gammes ayant les mêmes caractéristiques que SG mais ne minimisant aucun paramètre particulier.

Résolution : Prenons un exemple pour mettre en évidence le défi posé par le problème en termes de complexité algorithmique :

Soit une suite $\{C, G, D\}$ de trois accords en entrée.

Après calcul, l'ensemble N_1 (respectivement N_2, N_3) de gammes dans lequel est inclus l'accord C (resp. G, D) est le suivant :

- $N_1 = \{C_{maj}, G_{maj}, G_{1.3}, G_{1.4}, ...\}$

- $N_2 = \{C_{maj}, G_{maj}, G_{2.3}, G_{2.4}, ...\}$
- $N_3 = \{D_{maj}, G_{maj}, G_{3.3}, G_{3.4}, ...\}$

Le nombre de possibilités de production d'une suite de gammes en sortie est de l'ordre de $|N_1| \times |N_2| \times |N_3|$ possibilités. Plus généralement, le nombre de possibilités pour la suite de gammes à obtenir en sortie est d'environ $|N|^k$ avec k le nombre d'accords en entrée, et |N| le nombre moyen de gammes candidates par accord (les gammes dans lesquelles l'accord est inclus).

Le problème exprimé ci-dessus est solvable en exécutant un algorithme trivial qui comparera les $|N|^k$ possibilités et choisira l'une des meilleures. Notre objectif n'est donc pas seulement d'obtenir un résultat exact en sortie, mais aussi d'obtenir ce résultat avec un temps de calcul réduit et donc une complexité bien inférieure à $O(|N|^k)$.

2.2 Contraintes sur les suites de gammes

On détermine les choix effectués entre différentes gammes selon des paramètres particuliers qui sont des contraintes sur les suites de gammes, renseignés par l'utilisateur:

- Le nombre de gammes : L'utilisateur peut vouloir minimiser le nombre de gammes différentes dans la suite de gammes à générer.
- Le nombre de changements de gammes : L'utilisateur peut vouloir minimiser le nombre de changements de gammes au cours de la suite de gammes à générer.
- Le nombre de changements de notes entre deux gammes consécutives : L'utilisateur peut vouloir minimiser le nombre de changements de notes d'une gamme à la suivante dans la suite de gammes à générer.

2.3 Modélisation

Pour répondre à l'énoncé du problème, nous avons élaboré plusieurs modélisations possibles des données.
Nous avons ainsi une modélisation sous la forme d'un graphe et une autre sous la forme d'un tableau binaire. Ces deux représentations répondent à certaines contraintes sur les suites de gammes, en permettant d'effectuer un calcul de suite de gammes adéquates.
Ci-dessous sont présentés les caractéristiques de ces modélisations, ainsi que les paramètres auxquels elles apportent des solutions.

Modélisation en graphe

Le graphe est construit de la manière suivante :

- Chaque sommet est une gamme $G_{i.f(i)}$, où $f(i)$ détermine l'indice de la gamme en fonction de l'indice i, indexant l'accord dans SA, passé en paramètre.
- Chaque gamme fait partie de l'ensemble DG.
- A chaque accord A_i est associé un ensemble de gammes, appelé N_i, où i est l'index de l'accord dans SA. Les gammes associées à un accord contiennent cet accord. Plus précisément, ce sont toutes les gammes de DG contenant l'accord.
- Chaque élément de N_i contient l'accord indexé par i.
- Un arc est orienté et relie chaque élément de N_i à chaque élément de N_{i+1}.

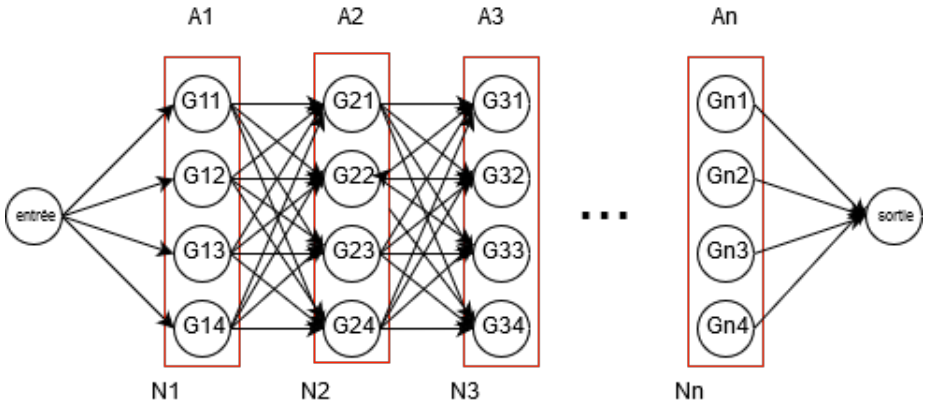


Figure 2.1 – Exemple du graphe

Le nombre de sommets associé à ce graphe est donc de $(\sum_{i=1}^{n} size(N_i)) + 2$.

Le nombre d'arcs est de $(\sum_{i=1}^{n-1} size(N_i) * size(N_{i+1})) + size(N_1) + size(N_n)$

Une fois ce graphe généré, un parcours est effectué afin de déterminer un chemin depuis le sommet "entrée" jusqu'au sommet "sortie". Ce chemin va permettre de répondre au paramètre choisi par l'utilisateur :

- **Le nombre de gammes :** il suffit de compter le nombre de gammes différentes lors du parcours du chemin.
- **Le nombre de changements de gammes :** il suffit de compter le nombre de fois que l'on passe par un arc dont le poids est différent de 0.
- **Le nombre de changements de note entre deux gammes consécutives :** il nous faut effectuer la somme du poids des arcs parcourus par le chemin.

Modélisation en tableau binaire

La modélisation en tableau binaire permet essentiellement de répondre au paramètre du nombre de changements de gammes. Nous avons comme colonnes de ce tableau la suite d'accords SA et en tant que lignes les gammes appartenant au dictionnaire de gammes DG.

Le principe est simple, à chaque fois qu'un accord A_i est contenu dans une gamme G_j, on annote d'un 1 l'intersection des deux dans le tableau binaire. A l'inverse, lorsque l'accord n'est pas contenu dans la gamme un 0 est placé à l'intersection. Un exemple vous est présenté ci-dessous :

Accords / Gammes	A_1	A_2	A_3	...	A_n
G_1	1	0	1	...	0
G_2	1	1	0	...	0
G_3	0	1	1	...	0
...	...	...	...	...	...
G_k	0	0	1	...	1

Table 2.2 – Exemple général du tableau (avec $k = size(DG)$ et $n = size(SA)$)

Comme indiqué en introduction de cette partie, ce tableau a pour objectif principal de minimiser le nombre de changements de gammes. En effet, lorsque nous parcourons ce tableau, il nous suffit de créer une liaison d'un accord à un autre tant que la même gamme contient les accords entre les deux.
Lorsqu'une même gamme ne peut pas être utilisée pour deux accords à la suite, on change de gamme en cherchant à nouveau un segment qui soit le plus long possible, afin de minimiser le nombre de changements de gammes.

L'exemple ci-dessous, présente une utilisation du tableau, les gammes sélectionnées sont celles qui ont des cases colorées en rouge :

Gammes \ Accords	A_1	A_2	A_3	A_4	A_5
G_1	1	0	1	0	0
G_2	1	1	0	0	0
G_3	0	1	1	0	0
G_4	0	0	1	1	1

Table 2.3 – Exemple de minimisation du nombre de changement de gamme

A partir de cet exemple, on peut déterminer qu'il y aura au minimum 1 changement de gamme, mais aussi au minimum 2 gammes différentes.
Pour les accords A_1 et A_2 la gamme G_2 sera préférée. Pour les accords A_3 A_4 et A_5 ce sera la gamme G_4.

Solutions Algorithmiques

Nous présentons ici le compte-rendu de la recherche, en termes d'algorithmique, sur le problème de la génération d'une ou plusieurs suites de gammes devant répondre à des contraintes, à partir d'une suite d'accords donnée.

En premier lieu, nous introduisons une structure de graphe particulière, établie à partir d'une suite d'accords, que nous appelons *GAKO* et qui sera utilisée par les algorithmes présentés plus loin.

Ensuite, en section 2, nous présentons des algorithmes de calculs brutaux pour notre problème de calcul de suites de gammes solutions. Ces algorithmes serviront de base pour vérifier les performances des solutions implémentées par la suite.

Nous détaillons dans la section 3 un algorithme issu de la littérature[1]; cet algorithme permet de calculer les plus courts chemins entre un sommet origine unique et les autres sommets d'un graphe orienté acyclique, après application d'une fonction de pondération quelconque sur les arcs.

Dans la section 4, nous montrons comment l'algorithme de la section 3, modifié notamment pour générer toutes les solutions optimales au lieu d'une seule, nous permet en étant appliqué à un *GAKO* de calculer des suites de gammes solutions en un temps et une complexité améliorés.

Puis, la section 5 s'intéresse à l'implémentation de l'algorithme résultant de la section 4, utilisé par le logiciel.

Enfin, nous présentons en section 6 une autre méthode de résolution de notre problème algorithmique, méthode basée sur les graphes d'intervalles, fruit de l'article de recherche donné par notre intervenant, et qui n'a pas été intégrée au logiciel.

1 Définition d'un GAKO

Nous définissons ici la structure de graphe particulière, déjà introduite en section *Modélisation*, que nous nommons *GAKO*. *GAKO* est l'acronyme de G*raphe* A*cyclique* K-*parties* O*rienté*.

La figure 3.1 illustre la manière dont un *GAKO* est construit.

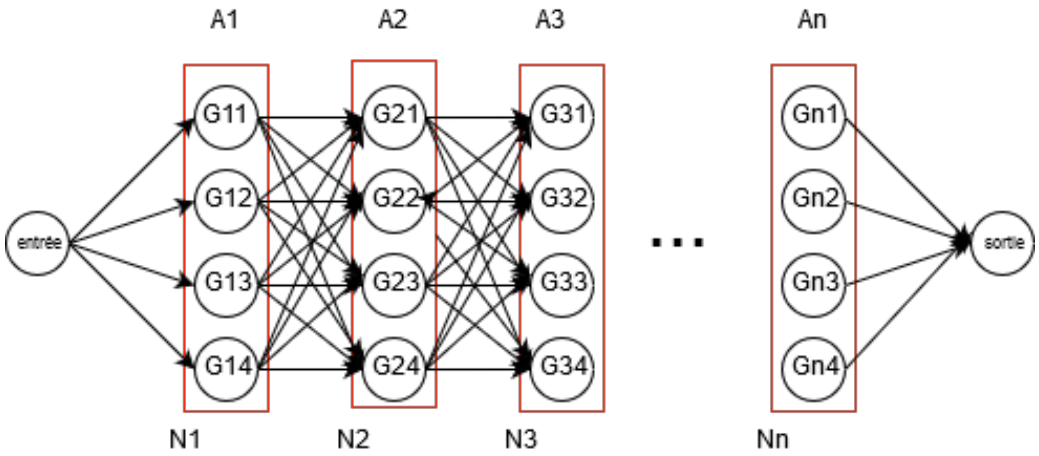


Figure 3.1 – Vue d'un *GAKO*

Soit une suite d'accords SA, on définit un $GAKO = (V_{gako}, E_{gako})$ à partir de SA.

[1] Cormen et al. "Algorithmique". In: 3ème édition. 2009. Chap. 24.2, Plus courts chemins à origine unique dans les graphes acycliques orientés, pp. 606–608.

On associe à chaque accord A_i appartenant à la suite d'accords SA un ensemble de gammes *candidates*[2] N_i, avec $1 \leq i \leq n$ et $n = |SA|$.

Pour chaque gamme $G_{i.f(i)}$ composant l'ensemble N_i, on crée un sommet dans V_{gako} et chaque sommet $v \in V_{gako}$ est étiqueté par une gamme $G_{i.f(i)}$.

On définit la fonction $G : V_{gako} \rightarrow DG$ (où DG est le dictionnaire des gammes) qui associe une gamme à chaque sommet de V_{gako}. On référera à la gamme associée à un sommet v par la fonction G par la notation $v.G$, équivalente à la notation $G(v)$.

On ajoute également à V_{gako} un sommet source i et un sommet de sortie o.

Un sommet $v \in V_{gako}$ étiqueté par une gamme $v.G \in N_i$ est relié à tous les sommets étiquetés par des gammes appartenant à N_{i+1}.

Aussi, le sommet d'entrée i est relié à tous les sommets v tels que $v.G \in N_1$, et le sommet de sortie est atteint par tous les sommets v tels que $v.G \in N_n$.

On définit E_{gako} comme suit :

$$E_{gako} = \{(u, v) : u, v \in V_{gako} \wedge u.G \in N_i \wedge v.G \in N_{i+1}\} \cup \{i, o\}$$

Propriétés du GAKO

- un $GAKO$ est un graphe k-parties, il peut être décomposé en k sous-ensembles de sommets tel que pour chaque sous-ensemble $V' \subseteq V$, tous les sommets de V' ne sont reliés entre eux par aucun arc.
- un $GAKO$ est un graphe orienté acyclique.
- Dans un $GAKO$, tous les chemins reliant le sommet i (sommet d'entrée) au sommet o (sommet de sortie) sont composés du même nombre d'arcs. De plus, il y a autant d'arcs dans une chemin reliant i à o que d'accords dans la suite d'accords SA dont on s'est servi pour construire le $GAKO$.

2 Algorithme Brut

2.1 Algorithme brut

Ici sera présentée la conception de l'algorithme de résolution du problème de manière brutale, c'est-à-dire par parcours successif de toutes les possibilités pour ensuite comparer ces possibilités et trouver les suites de gammes dont la valeur est optimale par rapport à une contrainte donnée.

Pour répondre à notre problème de calcul de suite de gammes adaptée à une suite d'accords donnée, une manière de procéder est de calculer toutes les suites de gammes solutions sans optimisation du temps de calcul. Cet algorithme exhaustif sera par la suite appelé algorithme *brut*.

L'algorithme brut sert à la fois de base pour tester le bon fonctionnement de l'architecture du projet dans son ensemble, et de point de comparaison avec la résolution du problème plus raffinée (cf. chapitre).

Génération des possibilités

L'objectif est de parcourir l'ensemble des chemins existant au sein d'un GAKO donné, la génération des différentes possibilités se fait donc de manière récursive.

Procedure Generation-chemins($\mathscr{G}$, k, *chemin_courant*, $\mathscr{L}$)

Data: $\mathscr{G}$ un GAKO, k est un entier dont la valeur est 0 au lancement de l'algorithme, chemin_courant est une liste vide

Result: Liste $\mathscr{L}$ des chemins possibles

```
1 begin
    // 𝒢.taille est la largeur de 𝒢
2   if k >= 𝒢.taille then
3     ℒ ∪ chemin_courant
4   else
5     foreach n t.q. n ∈ 𝒢[k] do
6       Generation-chemins(𝒢, k + 1, ℒ ∪ n, ℒ)
```

La complexité d'un tel algorithme est le produit de la taille de chaque k-partie de $\mathscr{G}$, soit $\prod_{k=1}^{\mathscr{G}.taille} \mathscr{G}[k].taille$

[2] gammes dans lesquelles A_i est inclus

Comparaisons des chemins

Maintenant que nous avons la liste des chemins $\mathscr{L}$ du GAKO $\mathscr{G}$, il faut les comparer pour trouver ceux qui sont les plus optimisés pour une contrainte donnée.
Chaque contrainte s'exprimera par un entier ω lié à chaque chemin $\mathscr{C}$, qu'on pourra nommer $\mathscr{C}.\omega$ (ω se calcule différemment pour chaque contrainte), on cherche donc les chemins dont le ω est le plus petit parmi $\mathscr{L}$, soit tous les $\mathscr{C}.\omega$ minimaux tels que $\mathscr{C} \in \mathscr{L}$.
Une première méthode, puisqu'on est ici dans une approche brutale, est de parcourir chaque chemin, d'en extraire leurs ω respectif en les stockant dans une liste appropriée $\mathscr{O}$, pour ensuite parcourir ladite liste afin de trouver la valeur minimale $\mathscr{O}_{min}$, et enfin parcourir de nouveau cette liste de ω pour en extraire tous les chemins de $\mathscr{L}$ où $\mathscr{C}.\omega = \mathscr{O}_{min}$.

Cette approche, bien que fonctionnelle, n'est pas efficace, même pour un algorithme brut. Une méthode d'optimisation serait de stocker le $\mathscr{O}_{min}$ au fur et à mesure de la génération des chemins possibles, afin d'éviter de parcourir les chemins où $\omega > \mathscr{O}_{min}$ avant que le chemin ne soit complet.
Il faudrait donc modifier l'algorithme de génération de chemins de cette manière :

Procedure Generation-chemins($\mathscr{G}$, k, *chemin_courant*, $\mathscr{L}$, ω , $\mathscr{O}_{min}$)

Data: $\mathscr{G}$ un GAKO, k est un entier dont la valeur est 0 au lancement de l'algorithme, chemin_courant est une liste vide
Result: Liste $\mathscr{L}$ des chemins possibles

```
1 begin
    // 𝒢.taille est la largeur de 𝒢
2   if k >= 𝒢.taille and ω <= 𝒪_min then
3     𝒪_min ← ω
4     ℒ ∪ chemin_courant
5   else
6     foreach n t.q. n ∈ 𝒢[k] do
        // modifier ω selon la contrainte concernée
7       𝒪_min ← min(ω, 𝒪_min)
8       Generation-chemins(𝒢, k + 1, ℒ ∪ n, ℒ, ω, 𝒪_min)
```

Il faut enfin filtrer $\mathscr{L}$ car il subsistera les chemins qui ont été calculé avant que le $\mathscr{O}_{min}$ ne soit trouvé.

Pour pouvoir tester la validité ainsi que l'efficacité de ses deux versions de l'algorithme brut, elles seront toutes les deux implémentées afin d'être testées et comparées.

3 Plus courts chemins à origine unique

Nous nous intéressons dans cette section à l'étude d'un algorithme issu de la littérature servant à calculer les plus courts chemins à origine unique dans un graphe acyclique orienté.

3.1 Définitions

Les termes suivants sont précisés afin d'améliorer la compréhension des algorithmes développés plus loin.

Fonction de pondération On applique à un graphe $\mathscr{G}$ une fonction de pondération $\omega : E \to \mathbb{R}$ qui associe à chaque arc un réel.

Plus court chemin On définit un chemin comme étant une suite d'arcs (ou de sommets) reliant un sommet de départ u à un sommet d'arrivée v.

Un plus court chemin entre deux sommets u et v est un chemin reliant u à v tel que la somme des pondérations des arcs composant ce chemin soit minimal.

La liberté est prise dans le rapport de référer à un plus court chemin par l'acronyme pcc.

Tri topologique *"Le tri topologique d'un graphe acyclique orienté $\mathscr{G} = (V, E)$ consiste à ordonner linéairement tous ses sommets de sorte que, si $\mathscr{G}$ contient un arc (u, v), u apparaisse avant v dans le tri. (Si le graphe contient un circuit, aucun ordre linéaire n'est possible.)"*[3]

[3] Cormen et al. "Algorithmique". In: 3ème édition. 2009. Chap. 22.4, Tri topologique, p. 566.

3.2 Algorithme des pcc à origine unique

Nous développons ici en détails l'algorithme bien connu de calcul des plus courts chemins à origine unique dans un graphe acyclique orienté. L'algorithme suivant est tiré de Cormen et al. [2] :

Algorithm 1: Plus-Courts-Chemins-GSS($\mathscr{G}$,s,ω)

Data: $\mathscr{G} = (V, E)$ un graphe orienté acyclique, s un sommet de $\mathscr{G}$ qui sera racine des plus courts chemins, ω une fonction de pondération sur les arcs.

Result: A chaque sommet $v \in V$ est associé le poids du plus court chemin de s à v (représenté par l'attribut d) ainsi que le prédécesseur de v sur le plus court chemin de s à v (représenté par l'attribut π).

```
1 begin
2     trier topologiquement les sommets de G
3     Source-Unique-Initialisation(𝒢,s)
4     for chaque sommet u pris dans l'ordre topologique do
5         for chaque sommet v ∈ Adj[u] do
6             Relâcher(u, v, ω)
```

L'algorithme 1 remplit deux tableaux :

1. Le premier tableau représente les distances entre les sommets de $\mathscr{G}$ et la racine s en empruntant un plus court chemin depuis s. Cette distance est exprimée pour un sommet par l'attribut d ($v.d$ = distance de s à v en empruntant un plus court chemin dans $\mathscr{G}$).

2. Le deuxième tableau est celui des prédécesseurs de chaque sommet sur un plus court chemin depuis s. L'attribut prédécesseur d'un sommet est noté π ($v.\pi = u$ tel que u est le prédécesseur de v sur le pcc reliant s à v). Pour expliciter la chaîne de sommets composant un pcc de s à v, il suffit de remonter récursivement chaque prédécesseur en partant de v. On construit un chemin $c = \{v_0, ..., v_n\}$ où $v_0 = s$ et $v_n = v$, tel que pour tout $v_i \in c$ on a $v_i.\pi = v_{i-1}$.

Les structures de *sous-graphe prédécesseur* et d'*arbre de plus courts chemins*, qui sont définies ci-après, sont induites par le calcul de l'algorithme 1. Ces structures sont mentionnées dans le théorème associé à l'algorithme 1 (cf. section 3.3) et seront utilisées lors du développement de notre propre théorème concernant notre algorithme modifié (cf. section 4.1).

Sous-graphe prédécesseur[2] On appelle sous-graphe prédécesseur le graphe $\mathscr{G}_\pi = (V_\pi, E_\pi)$ calculé à partir de l'attribut π, qui pour tous sommets v du graphe $\mathscr{G}$ représente son prédécesseur sur un plus court chemin depuis s.

L'ensemble V_π est l'ensemble des sommets de $\mathscr{G}$ ayant des prédécesseurs différents de NIL, auquel est ajouté l'origine s :

$$V_\pi = \{v \in V : v.\pi \neq NIL\} \cup \{s\}$$

L'ensemble d'arcs orientés E_π est l'ensemble des arcs reliant les sommets appartenant à V_π à leur prédécesseur :

$$E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\} \cup \{s\}$$

Dans l'exemple d'application de l'algorithme présenté en section 3.4, le sous-graphe prédécesseur $\mathscr{G}_\pi$ est conservé sous forme de tableau des prédécesseurs (voir plus loin le tableau 3.1).

Arbre de plus courts chemins La définition suivante est tirée du livre [4] :

"Soit $\mathscr{G} = (V, E)$ un graphe orienté pondéré par une fonction de pondération $\omega : E \to \mathbb{R}$. On suppose que $\mathscr{G}$ ne contient aucun circuit de poids strictement négatif accessible depuis le sommet origine $s \in V$, de sorte que les plus courts chemins sont bien définis. Un **arbre de plus courts chemins** de racine s est un sous-graphe orienté $\mathscr{G}' = (V', E')$, où $V' \subseteq V$ et $E' \subseteq E$, tel que

1. V' est l'ensemble des sommets accessibles à partir de s dans $\mathscr{G}$,
2. $\mathscr{G}'$ forme un arbre de racine s, et
3. pour tout $v \in V'$, le chemin simple unique de s à v dans $\mathscr{G}'$ est un plus court chemin de s à v dans $\mathscr{G}$."

Ci-dessous les procédures *Source-Unique-Initialisation* et *Relâcher* :

Procedure Source-Unique-Initialisation($\mathscr{G}$,s)

```
1 for chaque sommet v ∈ G.s do
2 |   v.d ← ∞
3 |   v.π ← NIL
4 s.d ← 0
```

La procédure Source-Unique-Initialisation sert à initialiser le tableau des distances à la racine et le tableau des prédécesseurs.

Procedure Relacher(u, v, ω)

```
1 if v.d > u.d + ω(u, v) then
2 |   v.d ← u.d + ω(u, v)
3 |   v.π ← u
```

La procédure Relacher détermine si un sommet u est un prédécesseur de v, c'est à dire si on peut atteindre v depuis s plus rapidement en passant par u.

Si v est atteignable en passant par u en une distance inférieure à $v.d$, alors u devient prédécesseur de v. Si u est un prédécesseur de v, $v.d$ prend la distance de s à u plus le poids de l'arc reliant u à v.

Complexité Théorique[2] Soit n le nombre de sommets et m le nombre d'arcs du graphe $\mathscr{G}$ passé en paramètre de l'algorithme 1.

Voilà le détail des complexités de chaque élément composant l'algorithme des pcc :

- Le tri topologique sur le graphe $\mathscr{G}$ prend un temps $\Theta(n + m)$.
- La procédure Source-Unique-Initialisation s'effectue en $\Theta(n)$ (elle passe par tous les sommets de $\mathscr{G}$).
- La boucle de la ligne 4 itère sur chaque sommet de $\mathscr{G}$ en $\Theta(n)$; la boucle de la ligne 5 entraîne l'application de la procédure Relacher, qu'on considère s'effectuer en $\Theta(1)$, une fois sur chaque arc de $\mathscr{G}$. La complexité des lignes 4-5 est de $\Theta(n + m)$.

En conclusion, la complexité théorique de l'algorithme 1 est de $\Theta(n + m)$.

3.3 Théorème sur l'algorithme 1

Le théorème suivant est tiré de l'ouvrage Cormen et al. [2] :

"Si un graphe orienté pondéré $\mathscr{G} = (V, E)$ d'origine s ne contient aucun circuit, alors, après exécution de la procédure Plus-Courts-Chemins-GSS, $v.d$ est le poids du plus court chemin de s à v pour tous les sommets $v \in V$ et le sous-graphe prédécesseur $\mathscr{G}_\pi$ est un arbre de plus courts chemins enraciné en s."

La preuve de ce théorème se trouve dans le même ouvrage.

3.4 Exemple de déroulement de l'algorithme

On va dérouler l'algorithme 1 sur le graphe $\mathscr{G}_1$ de la figure 3.2 :

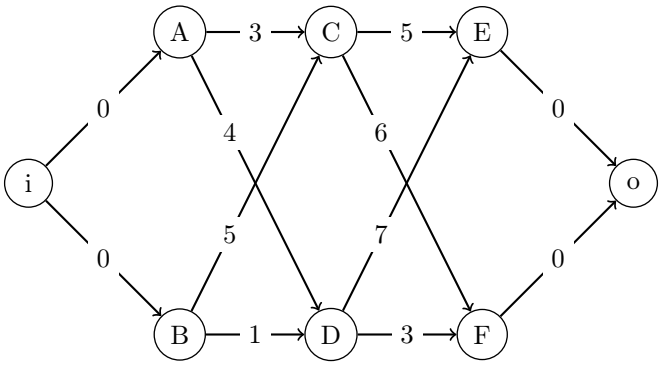


Figure 3.2 – $\mathscr{G}_1$, Graphe exemple pour l'application de l'algorithme 1

Il faut d'abord procéder à un tri topologique du graphe :

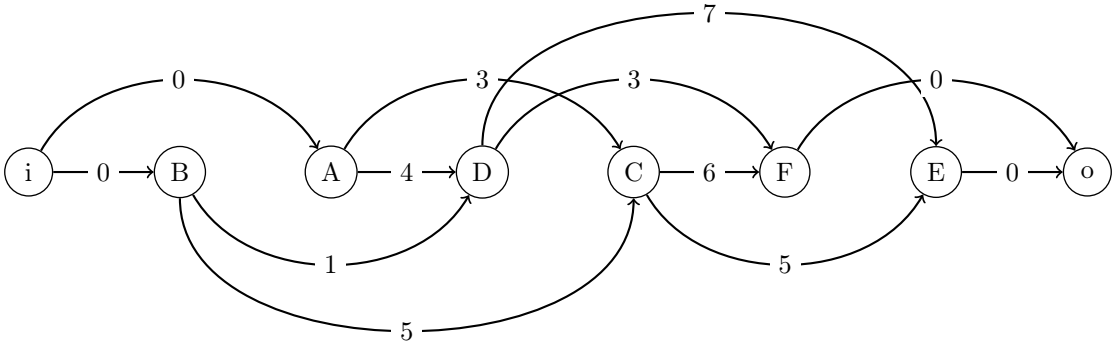


Figure 3.3 – Tri topologique des sommets de $\mathscr{G}_1$

La liste des sommets dans l'ordre topologique est la suivante : i, B, A, D, C, F, E, o

Ensuite, on va initialiser les tableaux de distances à la racine et de prédécesseurs, et lancer l'algorithme sur chaque sommet dans l'ordre du tri topologique.

La figure 3.4 montre le graphe résultat, où les labels sur les sommets représentent la taille des pcc depuis la racine i.

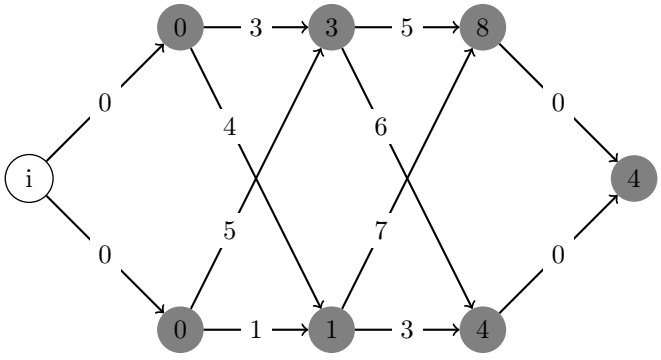


Figure 3.4 – $\mathscr{G}_1$ après application de l'algorithme 1

A la fin de l'application de l'algorithme, les tableaux de prédécesseurs et de distances à la racine sont remplis. Dans notre exemple, voilà le résultat des tableaux :

Sommets	**i**	**B**	**A**	**D**	**C**	**F**	**E**	**o**
d	0	0	0	1	3	4	8	4
π	nil	i	i	B	A	D	C	F

Table 3.1 – Calculs des distances à la racine et des prédécesseurs

Les plus courts chemins depuis i vers tous les sommets de $\mathscr{G}_1$ sont stockés dans le tableau des prédécesseurs. Par exemple, le pcc dans $\mathscr{G}_1$ de i à o est i, B, D, F, o (on remonte le tableau des prédécesseurs pour expliciter le pcc).

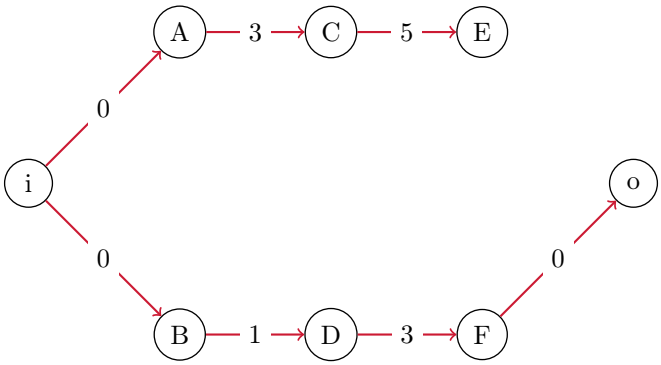


Figure 3.5 – $\mathscr{G}_{1\pi}$, Sous-graphe prédécesseur déduit par les valeurs de l'attribut π

4 Application à un cas particulier et extension de l'algorithme

En section 1, nous avons suggéré une représentation sous forme de graphe orienté des gammes associées à chaque accord (voir figure 3.1), que nous avons ensuite définie et nommée *GAKO*.

Nous allons montrer que calculer un plus court chemin entre le sommet d'entrée et le sommet de sortie d'un *GAKO* revient à calculer une suite de gammes répondant à notre problématique.

Nous utiliserons l'algorithme 1 légèrement modifié pour effectuer le calcul des pcc entre l'entrée et la sortie de ce graphe.

4.1 Calcul de plus courts chemins dans un GAKO

Nous avons défini ci-avant la structure et les propriétés du *GAKO*. Nous considérons un chemin c qui relie le sommet source i d'entrée au sommet d'arrivée o dans un tel graphe. Les caractéristiques suivantes sont déduites :

- Le nombre de sommets de c ôté des sommets i et o est égale au nombre d'accords de la suite d'accords SA en entrée.
- Pour tout sommet $v_j \in c$ tel que $c = \{i, v_0, ..., v_n, o\}$, on a $A_j \subseteq v_j.G$.

Donc n'importe quel chemin reliant i à o dans un *GAKO* est une solution au problème du calcul d'une suite de gammes correspondante, au sens de l'inclusion, vis à vis d'une suite d'accords donnée.

En se basant sur la structure particulière du *GAKO*, qui est un graphe acyclique orienté, l'algorithme 1 peut servir à calculer un plus court chemin de i à o.

La section 4.3 présente les avantages du calcul d'un pcc entre l'entrée et la sortie d'un *GAKO* pour des fonctions de pondération particulières sur les arcs.

Modification de l'algorithme 1 Lors de l'application de l'algorithme 1 sur un graphe acyclique orienté, il peut exister plusieurs pcc entre le sommet source s et les autres sommets.

Or, l'algorithme 1 ne calcule qu'un *seul* pcc entre le sommet source s et les autres sommets du graphe acyclique orienté passé en paramètre.

Nous modifions donc le pseudo-code de la procédure Relacher utilisée par l'algorithme 1 pour qu'à chaque sommet soit associé une liste de prédécesseurs (appelée Π) plutôt qu'un seul prédécesseur, et pour ainsi conserver les multiples pcc qui peuvent exister entre la racine et les autres sommets.

Le pseudo-code de la procédure Relacher modifiée est le suivant :

Procedure Relacher-Liste(u, v, ω)

```
1 if v.d > u.d + ω(u,v) then
2 |   v.d ← u.d + ω(u,v)
3 |   v.Π ← {u} // Π est une liste de predecesseurs
4 |
5 else if v.d == u.d + ω(u,v) then
6 |   v.Π ← v.Π ∪ u
```

La procédure Relacher-Liste détermine si un sommet u est un prédécesseur de v, c'est-à-dire si on peut atteindre v depuis s plus rapidement en passant par u. Si v est atteignable en passant par u en une distance équivalente à $v.d$, alors on ajoute u à la liste des prédécesseurs de v. Si v est atteignable en passant par u en une distance inférieure à $v.d$, alors une nouvelle liste des prédécesseurs contenant u remplace la liste des prédécesseurs actuels de v. Si u est un prédécesseur de v, $v.d$ prend la distance de s à u plus le poids de l'arc reliant u à v.

On appellera *Plus-Courts-Chemins-GSS-Modifie* (ou algorithme 1 modifié) l'algorithme 1 dans lequel la procédure Relacher est remplacée par la procédure Relacher-Liste.

En sortie de l'algorithme *Plus-Courts-Chemins-GSS-Modifie*, chaque sommet v du graphe passé en paramètre se voit associer le poids des plus courts chemins de s à v (représenté par l'attribut d) ainsi que la liste de prédécesseurs de v sur le ou les plus courts chemins de s à v (représenté par l'attribut Π).

Pour chaque prédécesseur u de v, il existe au moins un pcc de s à v. On explicite les pcc reliant s à v en remontant récursivement la liste des prédécesseurs de chaque sommet depuis le sommet v jusqu'au sommet s.

Lemme 1 Soit un graphe $\mathscr{G} = (V, E)$ acyclique orienté et de source s. Pour chaque sommet $v \in V$, la liste $v.\Pi$ des prédécesseurs calculée par *Plus-Courts-Chemins-GSS-Modifie* est l'union de chaque $v.\pi$ calculé par *Plus-Courts-Chemins-GSS* pour tous les tris topologiques de V.

Preuve Pour un sommet $v \in V$, la procédure Relacher met à jour $v.\pi$ si on accède plus rapidement à v depuis s en passant par u. Or si v est accédé depuis s aussi rapidement par u que par u', la procédure Relacher assignera à $v.\pi$ le sommet u ou u' selon leur ordre d'apparition dans le tri topologique des sommets. Si u apparaît dans le tri avant u' alors $v.\pi$ prendra la sommet u, et inversement sinon (conséquence de la comparaison à la ligne 1). La procédure Relacher-Liste ne tient pas compte de l'ordre du tri topologique pour l'assignation des prédécesseurs. Dans Relacher-Liste, si u et u' sont deux sommets qui permettent d'arriver en temps équivalent au sommet v, alors u puis u' seront ajoutés à la liste des prédécesseurs de v et ce que quelque soit leur ordre d'apparition dans le tri topologique (conséquence des lignes 5-6), du moment que u et u' se trouvent avant v dans le tri. Donc, pour tous $v \in V$, la liste des prédécesseurs $v.\Pi$ est bien l'union de toutes les valeurs de $v.\pi$ déterminées par les tris topologiques sur V.

Sous-graphe prédécesseur étendu Soit $\mathscr{G} = (V, E)$, un graphe acyclique orienté avec une source s. On appelle sous-graphe prédécesseur étendu le graphe $\mathscr{G}_\Pi = (V_\Pi, E_\Pi)$ calculé à partir de l'attribut Π, qui pour tous sommets v du graphe $\mathscr{G}$ représente sa liste de prédécesseurs sur des plus courts chemins depuis s. Il existe autant de pcc entre un sommet v et la source s que v a de prédécesseurs dans sa liste.

L'ensemble V_Π est l'ensemble des sommets de $\mathscr{G}$ ayant une liste de prédécesseurs différente de l'ensemble vide, auquel est ajouté l'origine s :

$$V_\Pi = \{v \in V : v.\Pi \neq \emptyset\} \cup \{s\}$$

L'ensemble d'arcs orientés E_Π est l'ensemble des arcs reliant les sommets appartenant à V_Π à leurs prédécesseurs :

$$E_\Pi = \{(u, v) \in E : v \in V_\Pi - \{s\} \wedge u \in v.\Pi\} \cup \{s\}$$

Lemme 2 Soit un graphe $\mathscr{G} = (V, E)$ acyclique orienté et de source s. Le sous-graphe prédécesseur étendu $\mathscr{G}_\Pi$ est l'union de tous les sous-graphes prédécesseurs $\mathscr{G}_\pi$ possibles.

Preuve D'après le Lemme 1, pour tous sommets $v \in V$, $v.\Pi$ est l'union de tous les $v.\pi$ possibles. En sachant qu'un sous-graphe prédécesseur $\mathscr{G}_\pi$ est construit à partir des $v.\pi$ où $v.\pi$ est un prédécesseur de v, alors un sous-graphe prédécesseur étendu $\mathscr{G}_\Pi$ est construit à partir de l'union de tous les $v.\pi$ possibles. De ce fait, un sous-graphe prédécesseur étendu $\mathscr{G}_\Pi$ est l'union de tous les sous-graphes prédécesseurs $\mathscr{G}_\pi$ possibles.

4.2 Théorème sur l'algorithme 1 modifié

On avance le théorème suivant :

Si un graphe $\mathscr{G} = (V, E)$ est un GAKO avec un sommet d'entrée i alors, après exécution de la procédure Plus-Courts-Chemins-GSS-Modifie prenant comme source i, $v.d$ est le poids des plus courts chemins de i à v pour tous les sommets $v \in V$ et le sous-graphe prédécesseur étendu $\mathscr{G}_\Pi$ est l'union de tous les arbres de plus courts chemins enraciné en i.

Preuve Démontrons le théorème exposé en section 4.2.

- Dans un premier temps, montrons que $v.d$ est bien le poids des plus courts chemins de i à v pour tous sommets $v \in V$:
  Pour un sommet v, l'attribut $v.d$ est mis à jour par la procédure Relacher-Liste. La mise à jour de l'attribut $v.d$ dans Relacher-Liste ne diffère pas de celle effectuée dans Relacher. En effet, la condition *else if* ajoutée à la ligne 5 dans Relacher-Liste ne modifie pas l'attribut $v.d$, et la première condition à la ligne 1 est identique dans son action sur $v.d$ à la première condition de la procédure Relacher. On prouve donc de la même manière que pour l'algorithme 1 que l'algorithme 1 modifié vérifie : $v.d$ est le poids des plus courts chemins de i à v pour tous sommets $v \in V$.

- Montrons maintenant que le sous-graphe prédécesseur étendu $\mathscr{G}_\Pi$ est l'union de tous les arbres de plus courts chemins enracinés en i :
  Il est prouvé dans Cormen et al. [4] qu'une fois que $v.d$ est bien le poids des plus courts chemins de i à v pour tous sommets $v \in V$, alors le sous-graphe prédécesseur $\mathscr{G}_\pi$ déduit de π est un arbre des plus courts chemins enraciné en i.

  Grâce au Lemme 2, on sait que le sous-graphe prédécesseur étendu $\mathscr{G}_\Pi$ est l'union de tous les sous-graphes prédécesseurs $\mathscr{G}_\pi$ possibles. Comme il a été prouvé plus haut que $v.d$ est bien le poids des plus courts chemins de i à v pour tous sommets $v \in V$, par déduction, le sous-graphe prédécesseur étendu $\mathscr{G}_\Pi$ est l'union des arbres de plus courts chemins enracinés en i.

Soutien par l'expérience Il est possible que la preuve présentée ci-dessus soit imprécise voir incorrecte; cependant les résultats obtenus après implémentation de la solution sont identiques aux résultats obtenus avec l'algorithme brut (cf.section 2.1 du chapitre 4) et pour un temps d'exécution bien moindre. Ce constat soutient notre théorème et confirme le gain de performance apporté par l'algorithme. Une étude comparative des performances des algorithmes développés pour la résolution de notre problème est présentée au chapitre 5 section 1.

Sous-graphe des solutions On appelle sous-graphe des solutions, le graphe $\mathscr{G}_{sols} = (V_{sols}, E_{sols})$ qui contient tous les plus courts chemins de i (sommet d'entrée) jusqu'à o (sommet de sortie), après l'application de l'algorithme 1 modifié sur un graphe $\mathscr{G} = (V, E)$ de type $GAKO$. $\mathscr{G}_{sols}$ est inclus dans $\mathscr{G}$.

L'ensemble des sommets V_{sols} contient tous les sommets se trouvant sur un pcc reliant i à o dans un $GAKO$, auxquels on ajoute les sommets i et o.

Soit $\mathscr{P}_{sols}$ l'ensemble des pcc reliant i à o dans un $GAKO$, on définit V_{sols} comme suit :

$$V_{sols} = \{v \in V : \exists p,\, p \in \mathscr{P}_{sols} \wedge v \in p\} \cup \{i, o\}$$

L'ensemble des arcs E_{sols} contient tous les arcs reliant les sommets appartenant à V_{sols} à leurs prédécesseurs sur un pcc depuis i :

$$E_{sols} = \{(u, v) \in E : v \in V_{sols} - \{i\} \wedge u \in v.\Pi\} \cup \{i\}$$

La structure de sous-graphe des solutions est intéressante, car elle ne contient que l'information qui nous est utile pour notre problème de calcul de suite de gammes. On utilisera cette structure pour afficher les suites de gammes étiquetant les pcc reliant i à o.

Une procédure pour afficher ces suites de gammes est décrite en section 5.4

On constate que le graphe prédécesseur étendu calculé par l'algorithme 1 modifié contenant tous les pcc entre le sommet i et les sommets du $GAKO$, le graphe prédécesseur contient le sous-graphe des solutions. Donc, calculer le sous-graphe prédécesseur étendu revient à calculer le sous-graphe des solutions.

4.3 Calcul de plus courts chemins et contraintes associées

La suite de gammes calculée à partir de la suite d'accords donnée en entrée doit répondre à certaines contraintes définies par l'utilisateur.

Si calculer un plus court chemin de i à o dans un $GAKO$ pour n'importe quelle pondération des arcs revient à calculer une suite de gammes adéquate, alors certaines contraintes peuvent être respectées en appliquant des fonctions de pondération spécifiques aux arcs.

Les contraintes de minimisation des changements de gammes et de minimisation des changements de notes pour la suite de gammes calculée sont modélisées par deux fonctions de pondération sur $\mathscr{G}$.

Minimisation des changements de gammes Il faut appliquer à chaque arête du graphe une fonction de pondération telle que pour un arc e reliant les sommets u, v, on ait : $\omega(e) \in \{0, 1\}$. Un arc est pondéré à 0 si les gammes consécutives reliées sont les mêmes. A défaut, il est pondéré à 1.
Le plus court chemin reliant le sommet d'entrée au sommet de sortie correspondra au chemin minimisant le nombre de changements de gammes.

Minimisation des changements de notes Il faut appliquer à chaque arête du graphe une fonction de pondération telle que pour un arc e reliant les sommets u, v, on ait : $\omega(e) = |u.G \Delta v.G|$. La fonction de pondération ainsi exprimée correspond au nombre d'éléments contenus dans l'ensemble obtenu par la différence symétrique des deux ensembles de notes que sont les gammes attribuées au sommet u et au sommet v.
Le plus court chemin reliant le sommet d'entrée au sommet de sortie correspondra au chemin minimisant le nombre de notes différentes entre deux gammes consécutives.

Convention de pondération Le sommet i ne possède que des arcs sortants pondérés à zéro, et le sommet o ne possède que des arcs entrants pondérés à zéro et ce quelle que soit la fonction de pondération associée au $GAKO$. Ces deux sommets permettent de simplifier le problème de calcul d'une suite de gammes en le ramenant au calcul d'un pcc entre une source et une arrivée unique.

4.4 Exemple de déroulement de l'algorithme 1 modifié sur un GAKO

Nous présentons dans cette section un exemple concret du déroulement de l'algorithme 1 sur un GAKO créé par la suite d'accords suivante : C F B C (Do Fa Si Do). En ne prenant en compte que les gammes majeures pour cet exemple, il a été déterminé que les gammes associées à chaque accord différent étaient les suivantes :

Accords	Gammes | Sommet correspondant		
C (Do)	C (Do) | A	F (Fa) | B	G (Sol) | C
F (Fa)	C (Do) | D	F (Fa) | E	A♯ (La♯) | F
B (Si)	E (Mi) | G	F♯ (Fa♯) | H	B (Si) | I
C (Do)	C (Do) | J	F (Fa) | K	G (Sol) | L

Après calcul de ces gammes, nous avons pu construire le GAKO ci-dessous, dont la fonction de pondération appliquée aux arcs est celle de la différence symétrique des notes entre gammes, sur lequel nous nous baserons pour développer l'algorithme.

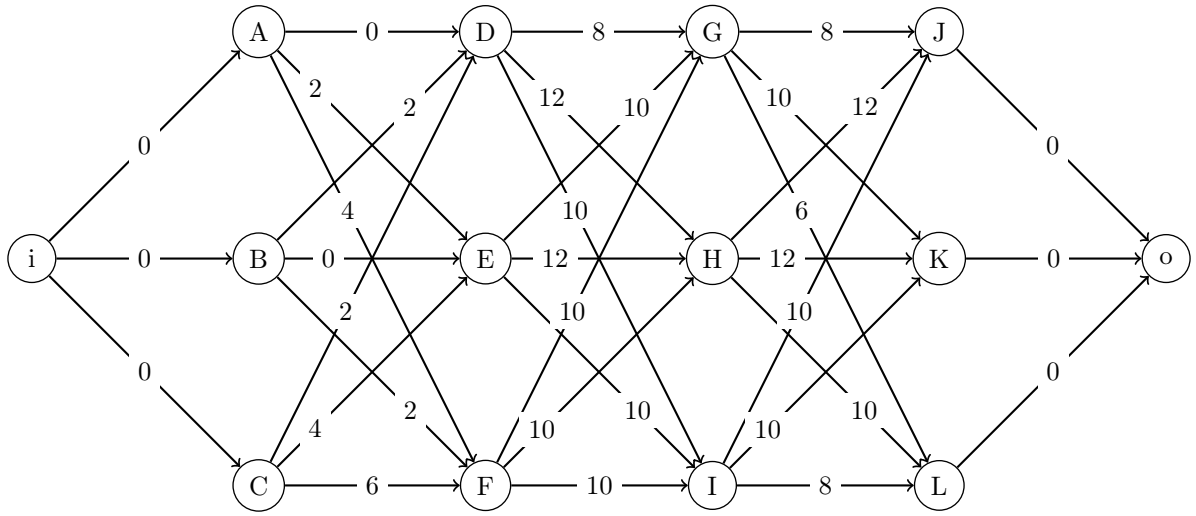


Figure 3.6 – $\mathscr{G}_1$, Graphe exemple pour l'application de l'algorithme 1 modifié basé sur la suite d'accords C F B C (Do Fa Si Do)

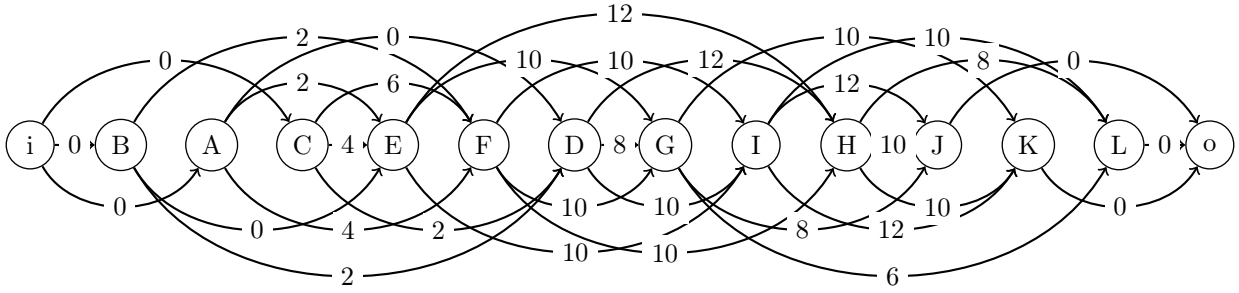


Figure 3.7 – Tri topologique des sommets de $\mathscr{G}_1$

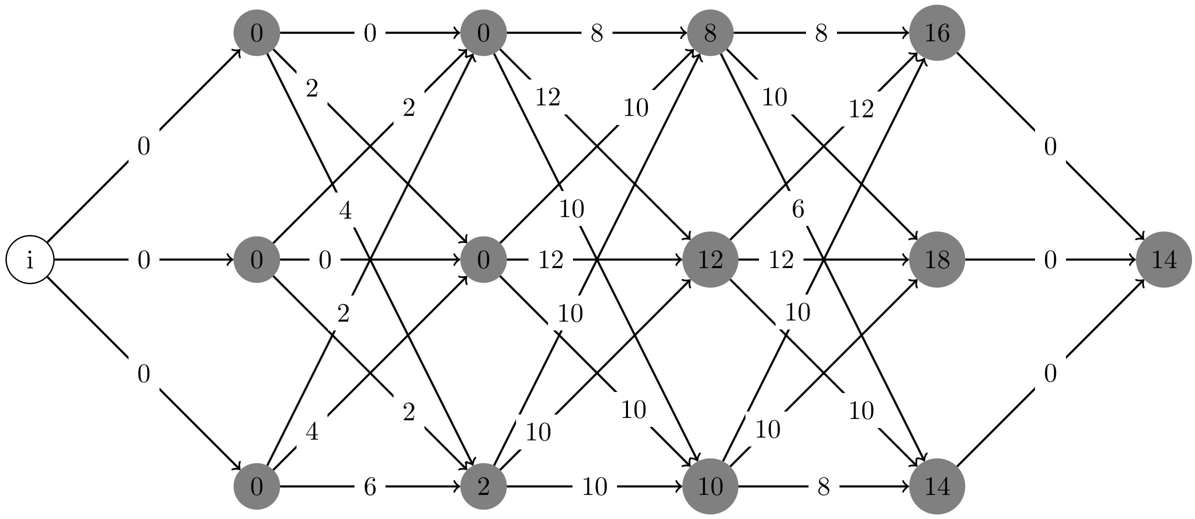


Figure 3.8 – Après application de l'algorithme

Sommets	**i**	**B**	**A**	**C**	**E**	**F**	**D**	**G**	**I**	**H**	**J**	**K**	**L**	**o**
d	0	0	0	0	0	2	0	8	10	12	16	18	14	14
Π	nil	i	i	i	B	B	A	D	D,E	D,E,F	G	G	G	L

Table 3.2 – Calculs des distances à la racine et des prédécesseurs

La solution, associée à la suite d'accords C F B C (Do Fa Si Do), minimisant les changements de notes, est donc C C E G (Do Do Mi Sol) en appliquant l'algorithme.

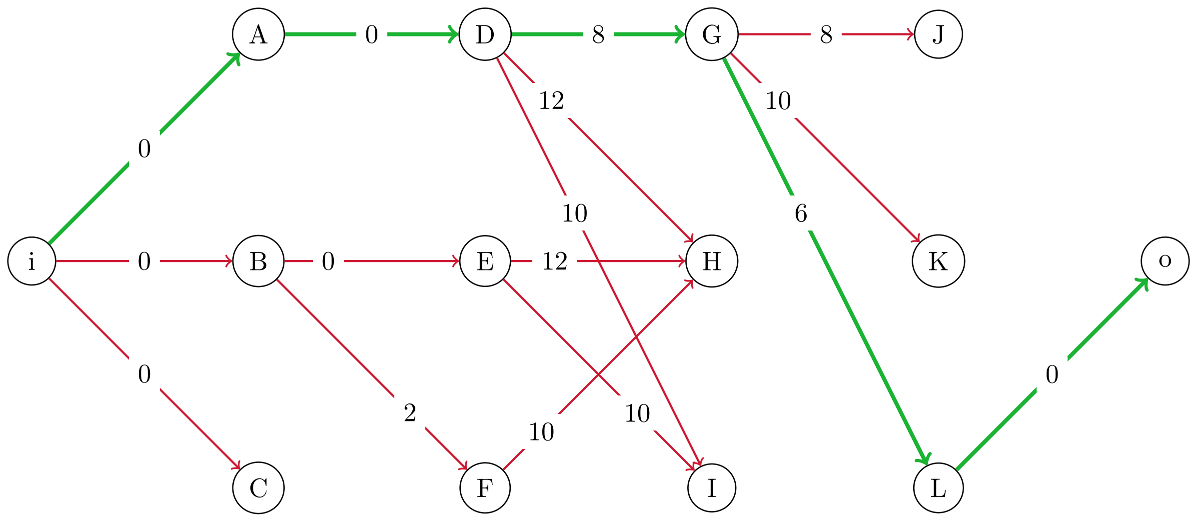


Figure 3.9 – En rouge, $\mathscr{G}_{1\Pi}$, le sous-graphe prédécesseur étendu calculé à partir de l'attribut Π; en vert, le sous-graphe des solutions $\mathscr{G}_{1Sols}$ inclus dans $\mathscr{G}_{1\Pi}$

5 Implémentation de la solution

Les algorithmes et fonctions qui suivent détaillent pas à pas la méthode d'implémentation de l'algorithme *Plus-Courts-Chemins-GSS-Modifie*.

Les algorithmes réutilisent les notations et structures de données définies par l'étude technique afin de définir une solution aisément programmable.

Afin de rendre plus claire l'utilité et l'utilisation des fonctions détaillées, on développera au fur et à mesure un exemple concret basé sur la suite d'accords C, F, B, C (Do, Fa, Si, Do), tout au long de cette section. Pour simplifier l'exemple, on considérera que le dictionnaire des gammes ne contient que les gammes "majeures".

5.1 Calcul des gammes candidates

La fonction GammesCandidates permet de calculer toutes les gammes qui incluent l'accord A donné en entrée.

L'attribut $Note\_Set$ associé aux objets Accord et Gamme correspond à l'ensemble de notes de l'accord ou de la gamme (notes qui sont représentées par des entiers).

On accède à la i-ème note de l'attribut $Note\_Set$ d'un objet Gamme ou Accord par la notation $Note\_Set(i)$.

Function GammesCandidates(A)

Data: A, un accord entré par l'utilisateur depuis l'interface graphique, et qui est maintenant représenté par un objet de la classe Accord.

Result: Retourne une liste de gammes (liste d'objets de la classe Gamme), tel que l'accord A est inclus dans chacune des gammes de la liste.

```
1  Searching_Set ← DG
   // DG référence le dictionnaire de gammes, c'est la liste de toutes les gammes
      possibles
2
3  foreach a t.q a ∈ A.Note_Set do
4      foreach g t.q g ∈ Searching_Set do
5          estDansLagamme ← false
6          i ← 0
7          while !estDansLaGamme and i < SizeOf(g.Note_Set) do
8              if a == g.Note_Set(i) then
                   // si la note a se trouve dans la gamme g
9                  estDansLaGamme ← true
10         if !estDansLaGamme then
11             Searching_Set ← Searching_Set/g
12 return Searching_Set
```

En premier lieu, la fonction récupère l'ensemble des gammes autorisées dans une liste, $Searching\_Set$, qui sera retournée en résultat de la fonction. Ensuite, pour chaque note de l'accord A, on regarde pour chaque gamme, de l'ensemble $Searching\_Set$, si elle y est incluse, le cas échéant la gamme est conservée dans l'ensemble, sinon elle y est supprimée.

Exemple sur l'accord de Do : Pour montrer le déroulement de la fonction, nous allons effectuer une trace de l'algorithme sur l'accord de Do. L'accord de Do est constitué de trois notes, le do, le mi et le sol.

Le tableau ci-dessous récapitule les tours de boucle sur les notes de l'accord :

Notes \ Gammes	**Do**	**Do♯**	**Re**	**Re♯**	**Mi**	**Fa**	**Fa♯**	**Sol**	**Sol♯**	**La**	**La♯**	**Si**
Do	✓	✗	✗	✓	✗	✓	✗	✓	✓	✗	✓	✗
Searching_Set = {Do, Re♯, Fa, Sol, Sol♯, La♯}												
Mi	✓	-	-	✗	-	✓	-	✓	✗	-	✗	-
Searching_Set = {Do, Fa, Sol}												
Sol	✓	-	-	-	-	✓	-	✓	-	-	-	-
Searching_Set = {Do, Fa, Sol}												

✓	Gamme conservée dans *Searching_Set*
✗	Gamme supprimée de *Searching_Set*
-	Gamme n'existant plus dans *Searching_Set*

Table 3.3 – Application de la fonction *GammesCandidates* sur l'accord de Do

La fonction GammesKParties renvoie une liste de listes de gammes sur laquelle nous allons nous reposer pour construire la structure de graphe (qui est celle de $\mathscr{G}$) qui sera donnée en paramètre de l'algorithme 1.

Function GammesKParties(SA)

Data: SA, une suite d'accords entrée par l'utilisateur depuis l'interface graphique, et qui est maintenant représentée par une liste d'objets de la classe Accord.

Result: Retourne un ensemble de liste de gammes $KParties$ (ensemble de listes d'objets de la classe Gamme), tel que l'accord $A_i \in SA$ est inclus dans chacune des gammes de $KParties(i)$

```
KParties ← ∅
foreach A t.q A ∈ SA do
    KParties ← KParties ∪ GammesCandidates(A)
return KParties
```

GammesKParties fait appel à GammesCandidates sur tous les accords entrés par l'utilisateur. Chaque résultat d'appel est stocké dans une liste, KParties, qui est renvoyée en fin de fonction.

La fonction est nommée GammesKParties car chaque liste de gammes composant la liste retournée va étiqueter les sommets d'une k-partie du graphe qui sera utilisée pour le calcul.

Exemple sur la suite d'accords Do Fa Si Do : Afin de présenter GammesKParties, ci-dessous, une trace est décrite sous la forme d'un tableau.

Accords \ Gammes	Do	Do♯	Re	Re♯	Mi	Fa	Fa♯	Sol	Sol♯	La	La♯	Si
Do	✓	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗
$KParties = \{\{Do, Fa, Sol\}\}$												
Fa	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓	✗
$KParties = \{\{Do, Fa, Sol\}, \{Do, Fa, La♯\}\}$												
Si	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✓
$KParties = \{\{Do, Fa, Sol\}, \{Do, Fa, La♯\}, \{Mi, Fa♯, Si\}\}$												
Do	✓	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗
$KParties = \{\{Do, Fa, Sol\}, \{Do, Fa, La♯\}, \{Mi, Fa♯, Si\}, \{Do, Fa, Sol\}\}$												

✓	Gamme contenant l'accord
✗	Gamme ne contenant pas l'accord

Table 3.4 – Application de la fonction *GammesKParties* sur la suite d'accords Do Fa Si Do

5.2 Implémentation de la structure de graphe

Du fait de la configuration particulière d'un *GAKO* (voir figure 3.1), on va se baser sur la structure de liste de listes de gammes retournée par la fonction GammesKParties pour générer le *GAKO* qui sera passé en entrée de l'algorithme *Plus-Courts-Chemins-GSS-Modifie*.

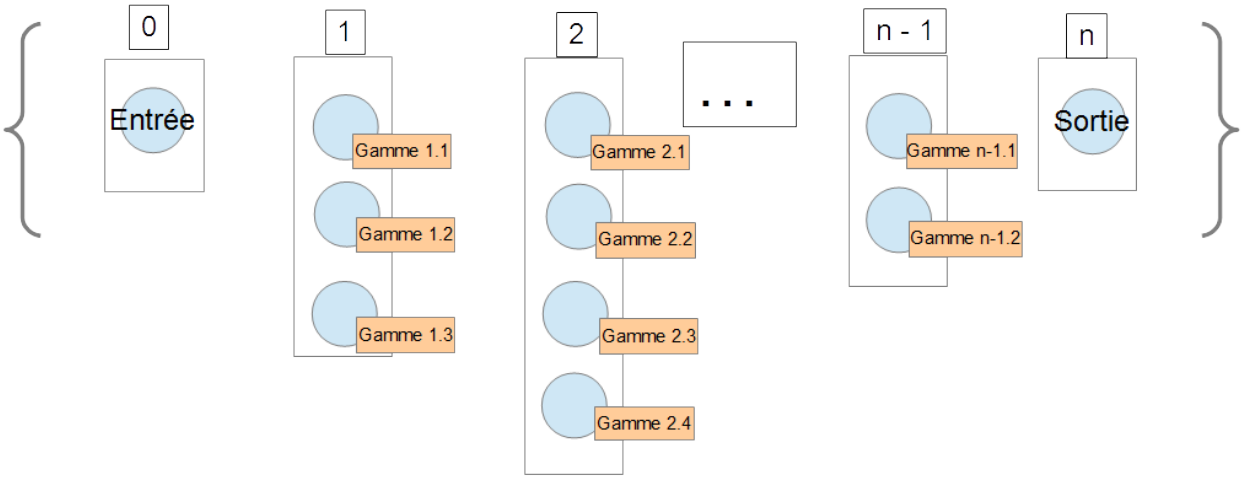


Figure 3.10 – Structure du graphe en mémoire

La figure 3.10 présente le *GAKO* créé à partir de la liste de listes de gammes. Le graphe ainsi représenté en mémoire est une liste de listes de sommets où chaque sommet de ces listes est étiqueté par une gamme.

La première liste (indice 0) de la structure de graphe contient l'unique sommet d'entrée, sommet racine dans l'application de l'algorithme 1 Algorithme des pcc à origine unique, et la dernière liste (indice n) contient l'unique sommet de sortie. Ces deux sommets sont ajoutés pour le déroulement de l'algorithme de plus court chemin et ne sont étiquetés par aucune gamme.

Sommets Afin de calculer le plus court chemin entre le sommet d'entrée et le sommet de sortie du *GAKO*, chaque sommet doit référencer son ou ses prédécesseur(s) sur un plus court chemin, depuis la racine ainsi que la longueur de ce plus court chemin. Un sommet doit également référencer tous ses successeurs dans le graphe. Nous définissons comme suit la structure d'un sommet en machine :

```
struct Sommet {

  int indiceVoisins; // indice de la liste de sommets voisins dans la structure de graphe
  int distanceRacine; // distance du plus court chemin depuis la racine
  List<struct Sommet*> predecesseurs; // liste de pointeur(s) vers sommet(s) predecesseur(s)
      sur le plus court chemin depuis la racine
  Gamme g; // gamme associee au sommet
};
```

Figure 3.11 – Structure de sommet en machine

Arcs Il n'est pas nécessaire de représenter les arcs du graphe en machine par des listes de voisins ou une matrice d'adjacence, car chaque sommet d'une k-partie du graphe est relié à tous les sommets de la k+1-partie. Ainsi, les voisins d'un sommet sont référencés dans la structure de la figure 3.11 par l'indice de la liste des sommets voisins dans la structure de graphe.

La fonction KPartiesVersGAKO, définie ci-après, crée un *GAKO* à partir d'une liste de listes de gammes.

Function KPartiesVersGAKO($KParties$)

Data: $KParties$, une liste de liste de gammes comme retournée par la fonction GammesKParties.
Result: Retourne une liste de listes de sommets *Graphe* où chaque sommet est étiqueté symétriquement par une gamme de $KParties$.

```
1  Graphe ← ∅
2  foreach k t.q k ∈ KParties do
3      liste_sommets ← ∅
4      foreach g t.q g ∈ k do
           // + 2, car l'insertion de la source apres décale les indices
5          s.indiceVoisins ← k.GetIndex() + 2
6          s.distanceRacine ← ∞
7          s.predecesseurs ← nil
8          s.g ← g
9          liste_sommets.Add(s)
10     Graphe.Add(liste_sommets)

   // Initialisation et ajout du sommet d'entree
11 s.indiceVoisins ← 1
12 s.distanceRacine ← 0
13 s.predecesseurs ← nil
14 s.g ← nil
15 Graphe.AddFirst(s)

   // Initialisation et ajout du sommet de sortie
16 s.indiceVoisins ← nil
17 s.distanceRacine ← ∞
18 s.predecesseurs ← nil
19 s.g ← nil
20 Graphe.AddLast(s)

21 return Graphe
```

La fonction KPartiesVersGAKO va instancier un à un les sommets du graphe en fonction de leur position dans l'ensemble $KParties$, en prenant garde à chaque fois d'associer la gamme correspondante.
Les sommets d'entrée et de sortie étant des sommets spéciaux ils sont rajoutés, après que la boucle sur $KParties$ soit effectuée, en bénéficiant d'une instanciation spécifique.

5.3 Tri topologique

L'algorithme *Plus-Courts-Chemins-GSS-Modifie* boucle sur les sommets du graphe en entrée selon un tri topologique de ces sommets. Dans notre cas, les sommets du *GAKO* offrent un tri topologique naturel en suivant l'ordre des k-parties.

L'algorithme 2 détaille la méthode d'obtention d'un tri topologique sur les sommets d'un *GAKO*:

Algorithm 2: Tri-Topologique($\mathscr{G}_0$)

Data: $\mathscr{G}_0$ un graphe orienté acyclique suivant la structure du graphe $\mathscr{G}$
Result: Un tri topologique des sommets de $\mathscr{G}_0$

```
1 Tri_Topologique ← ∅
2 begin
3     foreach kpartie of G_0 do
4         Ordonnancer de manière quelconque les sommets de kpartie
5         Ajouter les sommets de kpartie à la fin Tri_Topologique
6     return Tri_Topologique
```

5.4 Génération des solutions

Après avoir généré la structure de graphe, l'algorithme 1 est lancé, paramétré par une certaine fonction de pondération sur les arcs (deux fonctions possibles dans notre cas, voir 4.3).

Une fois le calcul terminé, reste à générer la/les solution(s), c'est à dire à générer la/les suite(s) de gammes étiquetant le/les plus court(s) chemin(s) depuis le sommet d'entrée vers le sommet de sortie.

Pour ce faire, nous avons conçu une fonction permettant de parcourir le sous-graphe des solutions présenté en section 4.1.

Pour générer la/les solution(s) (suite(s) de gammes éligibles), il faut parcourir en profondeur le sous-graphe des solutions en partant du sommet de sortie.

La procédure récursive détaille ce parcours et la comptabilisation de la/des suite(s) de gammes solution(s) :

Procedure GenerationSolutions(x, *solution*, $*$*liste_solutions*)

Data: x le sommet racine du parcours, *solution* une liste de gammes construite tout au long du parcours, *liste_solutions* un pointeur sur une liste de solutions. Dès qu'une solution est trouvée, elle est ajoutée à *liste_solutions*.
Result: La liste, *liste_solutions*, de toutes les solutions calculées à partir du sous-graphe des solutions.

```
1 if IsEmpty(x.Π) then
2     liste_solutions.Add(DeleteLast(Reverse(solution)))
3 else
      // On boucle sur chaque prédécesseur de x
4     foreach p t.q p ∈ x.Π do
5         GenerationSolutions(p, x.G ∪ solution, liste_solutions)
```

Dans la condition d'arrêt, l'ordre de la solution est inversé, car le parcours des prédécesseurs s'est effectué depuis la sortie vers l'entrée, et le dernier élément est supprimé car il s'agit de la gamme étiquetant le sommet de sortie (départ du parcours) ayant pour valeur *nil*.

Ci-dessous un exemple du fonctionnement de l'algorithme en se basant sur la liste des prédécesseurs obtenus dans la sous-section 4.4.

1^{er} appel de fonction : $GenerationSolutions(o, \{\}, \{\{\}\})$

2^{eme} appel de fonction : $GenerationSolutions(L, \{o\}, \{\{\}\})$

3^{eme} appel de fonction : $GenerationSolutions(G, \{o, L\}, \{\{\}\})$

4^{eme} appel de fonction : $GenerationSolutions(D, \{o, L, G\}, \{\{\}\})$

5^{eme} appel de fonction : $GenerationSolutions(A, \{o, L, G, D\}, \{\{\}\})$

6^{eme} appel de fonction : $GenerationSolutions(i, \{o, L, G, D, A\}, \{\{\}\})$

$liste\_solutions = \{\{A, D, G, L\}\}$

6 Graphe d'intervalles

Cette partie est d'intérêt théorique, elle vient d'un article de recherche à lire dans le cadre du projet et n'a pas été implémentée mais propose une solution algorithmique alternative.

6.1 Définitions

Les définitions données ci-après sont tirées de l'article *An Optimal Algorithm for Shortest Paths on Weighted Interval and Circular-Arc Graphs, with Applications.*[1]

Intervalle Un intervalle I est défini par ses deux terminaisons gauche et droite, tel que $I = [a, b]$ avec a le point de terminaison gauche de I et b le point de terminaison droit de I. Pour un intervalle I, on considère que $a \leq b$, donc un point est un intervalle (cas où $a = b$).

Graphe d'intervalles Soit l'ensemble S d'intervalles $I_1...I_n$, où $b_1 \leq b_2 \leq ... \leq b_n$ (où b_i est la terminaison droite de l'intervalle I_i), on peut représenter cet ensemble sous forme de graphe d'intervalles comme montré à la figure 3.12.

On crée un sommet pour chaque intervalle $I_i \in S$, et deux sommets sont reliés par des arêtes si les intervalles qu'ils représentent se chevauche, c'est à dire si l'intersection des deux intervalles n'est pas vide.

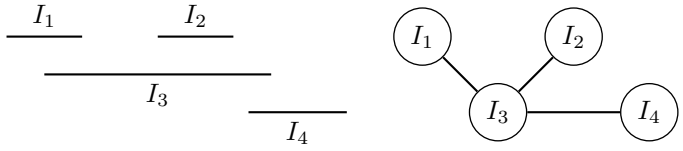


Figure 3.12 – Exemple d'ensemble d'intervalles et modélisation sous forme de graphe

Par la suite, on référera à un ensemble d'intervalles ou un graphe d'intervalles comme étant la même entité.

On exprime par la notation S_i le sous-ensemble de S contenant les intervalles $I_1, I_2, ..., I_i$. Dans un ensemble d'intervalles S, I_1 est appelé l'intervalle source.

Pondération sur les intervalles On peut associer à chaque intervalle d'un ensemble S une pondération tel que $\omega : S \to \mathbb{R}$. Pour simplifier la notation, on référera à la pondération associée à l'intervalle I_i dans l'ensemble S par ω_i. Par la suite, on considérera que ω_i est toujours supérieure ou égale à zéro (condition de fonctionnement de l'algorithme détaillé par Atallah, Chen, and Lee [1]).

Chemin d'intervalles Un chemin d'intervalles est une suite d'intervalles qui se chevauchent, et qui couvrent donc une portion continue du départ jusqu'à l'arrivée. La longueur d'un chemin d'intervalles est la somme des poids associés à chaque intervalle le composant. Un plus court chemin reliant un intervalle I à un intervalle J est un chemin dont la longueur est minimale pour aller de I à J.

Intervalle inactif Un intervalle I_k est dit inactif dans S_i (avec $k \leq i$) si et seulement si il est inutile pour trouver un plus court chemin entre la source I_1 et les autre intervalles dont la terminaison droite est supérieure ou égale à b_i.

Dans la figure 3.12, par exemple, on donne les poids suivants aux intervalles : $\omega_1 = 2$, $\omega_2 = 1$, $\omega_3 = 5$, $\omega_3 = 1$. I_2 est inactif dans S_3 car il ne permet pas d'emprunter un plus court chemin de I_1 à I_3, ou de I_1 à I_4. Un intervalle inactif dans S_i est inactif dans tous S_j avec $i \leq j$.

Longueur des plus courts chemins La notation $label_j(i)$ $(i \leq j)$ équivaut à la longueur du plus court chemin de I_1 à I_i dans S_j. Pour $j < i$, $label_j(i)$ vaut $+\infty$.

Par exemple, dans la figure 3.12, pour les mêmes pondérations que dans l'exemple précédent, $label_2(2) = +\infty$ car il n'existe pas de pcc reliant I_1 à I_2 dans S_2.

En revanche, $label_4(3) = 7$, poids du plus court chemin de I_1 à I_3 dans S_4.

On note $label_n(i)$, avec n l'indice du dernier intervalle de S, la longueur du plus court de I_1 à I_i dans S_n. Or comme $S_n = S$, $label_n(i)$ dénote la longueur du plus court chemin de I_1 à I_i dans S.

L'algorithme développé ci-après repose sur la notion d'intervalles inactifs et sur la notation $label_j(i)$ pour fonctionner.

6.2 Calcul de plus courts chemins sur un graphe d'intervalles pondérés

Dans l'article *An Optimal Algorithm for Shortest Paths on Weighted Interval and Circular-Arc Graphs, with Applications*[1], les auteurs présentent un algorithme, qu'on nommera *PCC-Intervals*, permettant de calculer les plus courts chemins entre un intervalle source I_1 et tous les autres intervalles d'un ensemble S d'intervalles.

Nous ne ferons que présenter les principaux aspects de l'algorithme dans cette section, l'étude détaillée de la solution n'étant pas notre intention.

Entrée de l'algorithme L'algorithme prend en entrée une liste S d'intervalles, triés dans l'ordre croissant de leur terminaison droite, et un intervalle source $I_1 \in S$, premier élément de la liste triée.

Sortie de l'algorithme L'algorithme calcule pour chaque intervalle I_i la longueur du plus court chemin de I_1 à I_i dans S (il calcule donc $label_n(i)$ pour tout I_i).

L'algorithme calcule également la suite d'intervalles explicitant le plus court chemin de I_1 à I_n, I_n étant le dernier élément de S (avec la terminaison droite la plus grande). Par la suite, on appellera δ_i le plus court chemin reliant I_1 à I_i dans S.

Principe de l'algorithme L'algorithme *PCC-Intervals* effectue une boucle de $i = 2$ jusqu'à n (n étant l'indice du dernier intervalle de S) en prenant en compte le sous-ensemble d'intervalles S_i à chaque tour de boucle.

Au fil de la boucle, on maintient deux ensembles d'intervalles; un premier ensemble $actifs(i)$ contenant les intervalles actifs dans S_i; un deuxième ensemble $inactifs(i)$ représentant les intervalles inactifs dans S_i. Dans les deux ensembles, les intervalles sont ordonnés de manière croissante selon la valeur de leur terminaison droite.

Au début de la boucle, on détermine si l'intervalle I_i est inactif dans S_i; si c'est le cas, on ajoute I_i à inactifs(i) et on passe tour suivant.

Si l'intervalle I_i n'est pas inactif, il est ajouté à $actifs(i)$, puis, l'algorithme détermine quel intervalle I_k avec $k \leq i$ est inactif dans S_i.

Lorsqu'un intervalle I_k est détecté, il est ajouté à $inactifs(i)$ et enlevé de $actifs(i)$.

Après délétions des intervalles inactifs, $actifs(i)$ représente le pcc reliant I_1 à I_i à dans S_i.

Soit I_j l'intervalle précédant I_i dans $actifs(i)$, on calcule la longueur du pcc de I_1 à I_i dans S_i ($label_i(i)$) en se basant sur la longueur du pcc de I_1 à I_j dans S_{i-1} ($label_{i-1}(j)$) à laquelle on ajoute le poids de l'intervalle I_i.

Pour résumé, on a $label_i(i) = label_{i-1}(j) + \omega_i$.

A la fin de la boucle, $actifs(n)$ représente le pcc reliant I_1 à I_n dans S, et pour tout intervalle $I_i \in actifs(n)$, $label_n(i)$ (longueur du pcc de I_1 à I_i dans S) est calculé.

Reste à mettre à jour les labels pour tous les intervalles $I_j \in inactifs(n)$. Pour ce faire, chaque intervalle $I_j \in inactifs(n)$ trouve un intervalle $I_i \in actifs(n)$ tel $b_i \geq a_j$.

Lorsqu'un tel I_i est trouvé, le label de I_j est mis à jour : $label_n(j) = label_n(i) + \omega_j$

Nous allons maintenant montrer comment cet algorithme nous permet, en ramenant notre problème à un problème de graphe d'intervalles, de calculer une suite de gammes répondant à des contraintes spécifiées.

6.3 Application à notre problème

Comme introduit en section 2.3, nous pouvons représenter la relation entre une suite d'accords et les gammes incluant les accords de la suite par un tableau binaire.

Comme le montre la figure 3.12, le tableau binaire prend en colonne les accords de la suite d'accords (un par colonne) et en ligne l'ensemble des gammes données par le dictionnaire des gammes (une par ligne). Un 1 est marqué dans une case du tableau lorsque l'accord de la colonne est inclus dans la gamme de la ligne correspondante. Un 0 est marqué en cas de non-inclusion.

L'exemple 3.5 présente le tableau binaire pour la suite d'accords C, F A#, B, C en prenant en compte uniquement les gammes majeures.

Gammes \ Accords	C	F	A#	B	C
C	1	1	0	0	1
D#	0	0	1	0	1
E	0	0	0	1	0
F	1	1	1	0	1
F#	0	0	0	1	0
G	1	0	0	0	1
A#	0	1	1	0	0
B	0	0	0	1	0

Table 3.5 – Tableau binaire correspondant à la suite d'accords Do, Fa, La#, Si, Do

Dans le tableau 3.5, trouver une suite de 1 qui permet de traverser la matrice de gauche à droite revient à trouver une suite de gammes solution vis à vis de la suite d'accords. Moins la suite de 1 trouvée nous amène à changer de ligne, moins la suite de gammes solution comportera de changement de gammes entre gammes consécutives.

On peut donc minimiser la contrainte du changement de gammes dans la suite de gammes solution en essayant toujours de parcourir les segments de 1 les plus longs dans la matrice.

Traduction en graphe d'intervalles On peut déduire aisément d'un tableau binaire un ensemble d'intervalles, en considérant :

- Une case contenant un 1 comme un point, sachant qu'un point est un intervalle dont les terminaisons sont égales.
- Une suite contigüe de cases contenant des 1 comme un intervalle.

La figure 3.13 montre la conversion du tableau binaire 3.5 en ensemble d'intervalles.

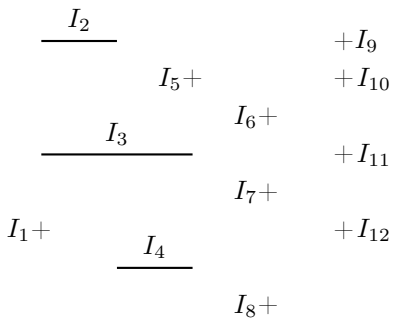


Figure 3.13 – Conversion du tableau binaire en ensembles d'intervalles

Sous forme de graphe d'intervalles, une suite de gammes solution est représentée par un chemin d'intervalles couvrant de manière continue le plan depuis l'intervalle source jusqu'à l'intervalle le plus à droite. Pour spécifier le problème, on peut ajouter deux intervalles fictifs I_d (intervalle départ) et I_a (intervalle d'arrivée) à notre graphe d'intervalles; le reflet d'une suite de gammes solution devient alors un chemin d'intervalles reliant l'intervalle I_d à l'intervalle I_a.

De plus, si on veut minimiser le nombre de changements de gammes dans notre suite de gammes solution, il suffit de trouver un chemin de I_d à I_a empruntant le moins d'intervalles possibles. Cela revient à calculer un pcc de I_d à I_a pour une fonction de pondération associant un poids de 1 à chaque intervalle, et un poids de 0 à I_d et I_a qui sont des intervalles de spécification du problème. On peut alors se baser sur l'algorithme *PCC-Intervals* pour calculer un tel pcc.

Dans le cas de deux cases adjacentes placées sur des lignes différentes et contenant un 1, on fait se chevaucher les intervalles les représentant en augmentant légèrement les terminaisons droites et gauches. En effet, pour calculer un plus court chemin dans un graphe d'intervalles pondérés, il est important que les intervalles se chevauchent.

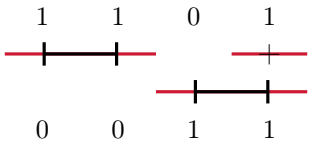


Figure 3.14 – Extension des intervalles construits à partir du tableau binaire

La figure 3.14 présente deux lignes d'une matrice binaire qui a été traduite en graphe d'intervalles. On voit en tracé noir, les intervalles obtenus directement depuis les valeurs de la matrice, et en tracé rouge les extensions ajoutées aux intervalles obtenus pour que ceux-ci puissent se chevaucher, ce qui nous permet de calculer un plus court chemin d'intervalles.

La figure représente la traduction en graphe d'intervalles du tableau binaire 3.5. Chaque intervalle a été étendu pour permettre un chevauchement et les deux intervalles I_d et I_a ont été ajoutés.

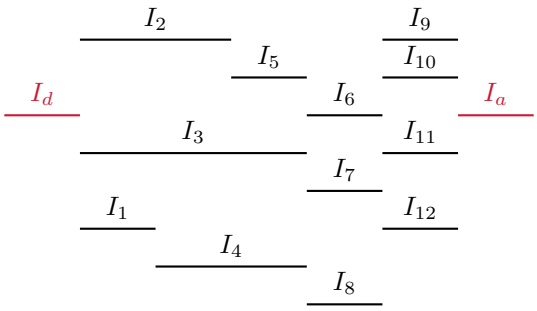


Figure 3.15 – Conversion du tableau binaire en ensemble d'intervalles avec extension, et ajout d'une source et une arrivée

On voit en figure 3.15 que les plus courts chemins pour relier I_d à I_a sont composés de 3 intervalles et ont donc une longueur de 3 (la longueur d'un chemin étant la somme des poids des intervalles le composant). On peut donner quelques exemples de plus courts chemins : $\{I_3, I_7, I_{11}\}$, $\{I_3, I_6, I_{11}\}$, $\{I_3, I_8, I_{12}\}$. On remarque que l'intervalle I_3 fait toujours partie des pcc car c'est l'intervalle le plus long (il correspond à la suite de trois 1 pour la gamme de F dans le tableau 3.5).

Rapport technique

1 Étude détaillée du logiciel

1.1 Objectif du logiciel

L'application à développer doit permettre à l'utilisateur de saisir une *grille*[1] d'accords en entrée et de lancer un calcul d'une suite de gammes qui sera disponible en sortie.

Le logiciel fournira un ensemble de fonctionnalités périphériques permettant la saisie d'une suite d'accords (ou "grille d'accords") par l'utilisateur.

Pour ce qui est de la suite de gammes obtenue en sortie, elle pourra être consultée de manière visuelle ou sonore (sous forme de fichiers *.midi*) et enregistrée de la même manière.

Choix du nom Nous avons choisi d'appeler notre logiciel *Ut*, en référence à la clef de *Ut* placée sur une portée. Nous utiliserons ce nom pour désigner le logiciel dans la suite du rapport.

1.2 Description des cas d'utilisation

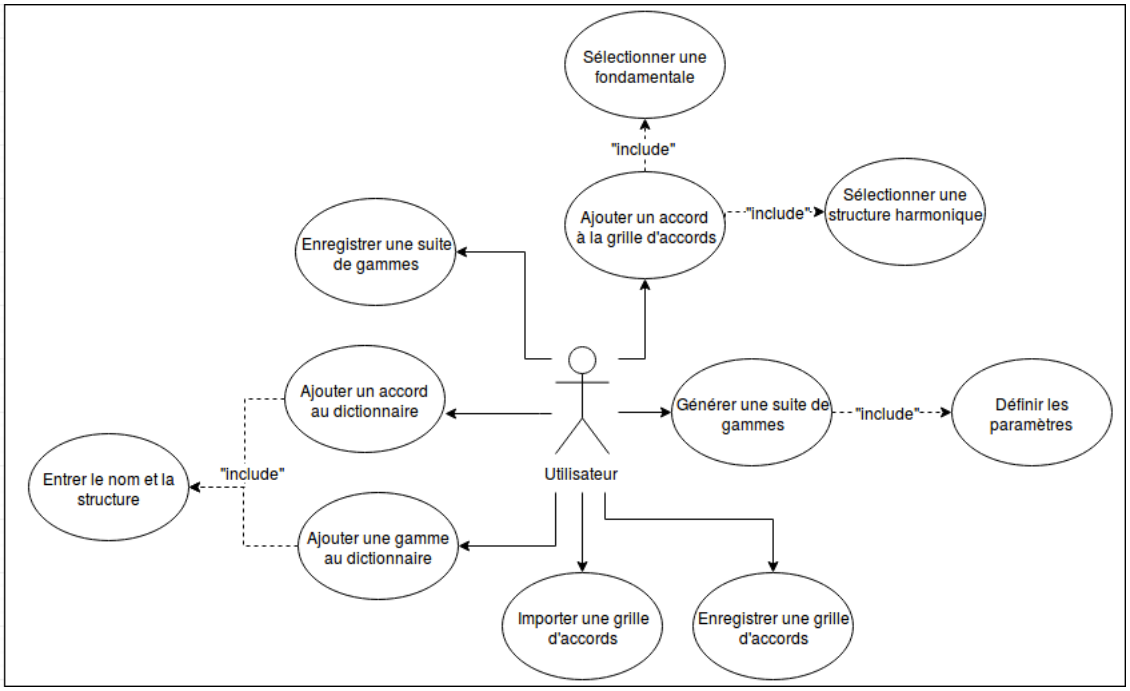


Figure 4.1 – Diagramme des cas d'utilisation

La figure 4.1 présente les fonctionnalités fournies par le logiciel. Le logiciel n'admet qu'un seul type d'utilisateur qui peut effectuer l'ensemble des actions décrites.

[1] Expression musicale pour parler de suite d'accords. Un morceau de musique peut-être représentée sous la forme d'une grille, où chaque case de la grille représente une mesure à laquelle est associée un accord.

1.3 Prototype de GUI

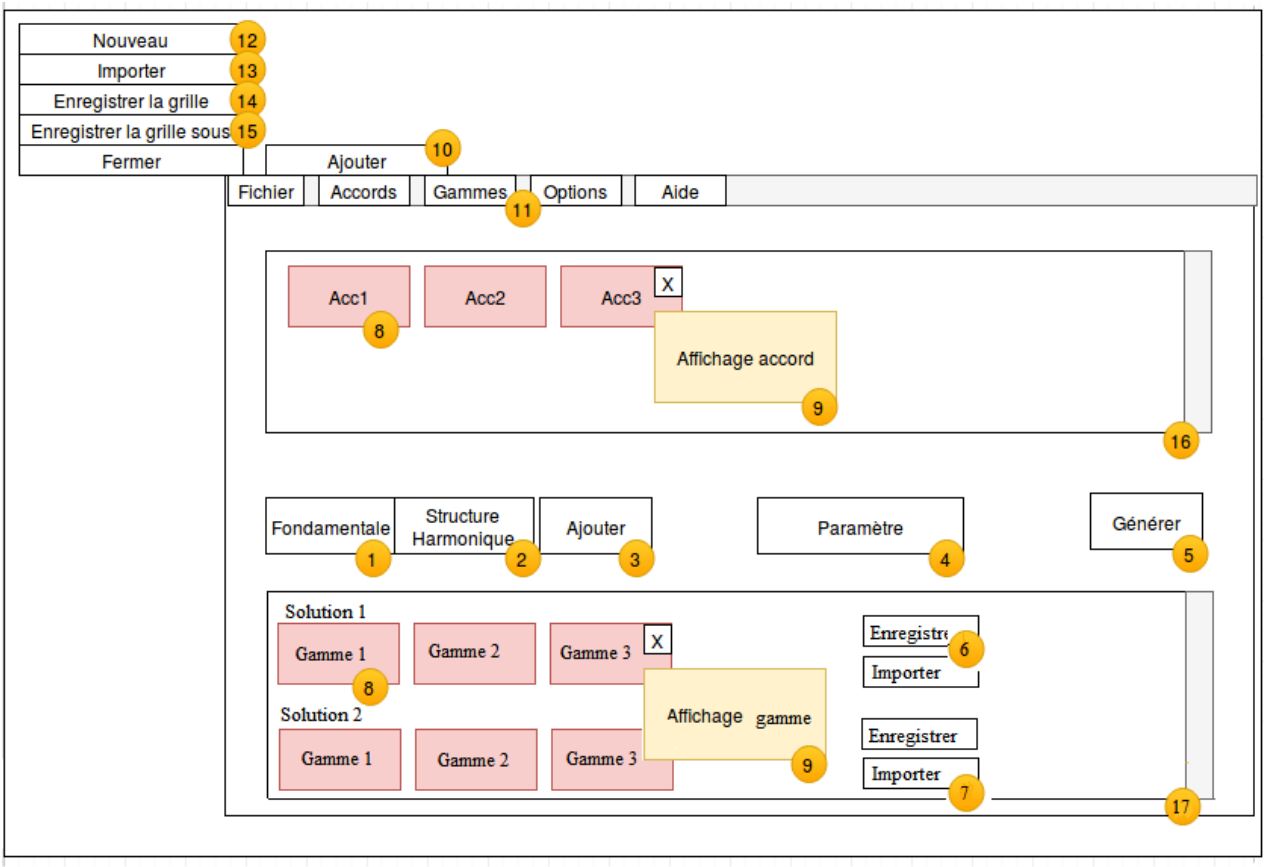


Figure 4.2 – Prévisualisation de l'interface graphique du logiciel

Au vue des fonctionnalités listées à la figure 4.1, nous avons élaboré un prototype de l'interface graphique du logiciel. Les différents composants du prototype de l'interface graphique, présenté dans la figure 4.2, sont détaillés ci-dessous.

Ajouter des accords à la grille

(1) **Fondamentale** Menu déroulant permettant de choisir la note fondamentale parmi les 12 notes de la gamme chromatique.

(2) **Structure harmonique** Menu déroulant permettant de choisir une structure harmonique parmi celles proposées par défaut et celles créées par l'utilisateur.

(3) **Ajouter** Bouton permettant d'ajouter un accord à la grille d'accords en fonction de la fondamentale et de la structure harmonique, choisies.

(4) **Paramètre** Zone permettant la sélection des contraintes à appliquer au calcul de la suite de gammes en sortie. Les paramètres pourront être les suivants :

- Option à cocher (checkbox) pour minimiser le nombre de changements de gammes lors du calcul.
- Option à cocher (checkbox) pour minimiser le nombre de changements de notes lors du calcul.
- Option à cocher (checkbox) pour minimiser le nombre total de gammes lors du calcul.
- Liste à choix multiples pour définir les gammes autorisées lors du calcul, parmi celles de bases et celles créées par l'utilisateur.

(5) **Générer** Bouton permettant de générer la suite de gammes (Lancement du calcul).

Enregistrer une suite de gammes

(6) **Titre** Enregistrer la gamme en .txt.

(7) **Enregistrer** Enregistrer la gamme sous le fichier midi.

(8) **Accords ou Gamme** Affichage des Accords déjà sélectionnés ou des gammes trouvées.

(9) **Infobulle** Infobulle donnant les informations à propos de l'accord ou de la gamme.

Barre de menu principal

(10) **Ajouter** Bouton de l'onglet *Accords* pour ajouter un accord au dictionnaire.

(11) **Gammes** Onglet gammes dont le menu déroulant propose les mêmes options que l'onglet *Accords*.

(12) **Nouveau** Bouton de l'onglet *Fichier* pour commencer un nouveau projet (effaçant l'avancement du précédent).

(13) **Importer** Bouton de l'onglet *Fichier* pour importer un projet précédemment enregistré, le projet ne sauvegarde que la suite d'accords.

(14) **Enregistrer la grille** Bouton de l'onglet *Fichier* pour enregistrer le projet courant.

(15) **Enregistrer la grille sous** Bouton de l'onglet *Fichier* pour enregistrer une instance supplémentaire du projet courant.

(16) **Grille des gammes** Espace dans lequel les suites de gammes trouvées sont représentées. Ne s'affiche qu'après avoir appuyé sur le bouton "Générer"

1.4 Cas d'utilisation détaillés

Dans cette section sont détaillés les cas d'utilisation présentés à la figure 4.1. Les différents composants du prototype de l'interface graphique, qui sont partie intégrante du cas d'utilisation détaillé, sont référencés par leur indice numérique.

Ajouter un accord à la grille d'accords

Entrée	L'utilisateur, après avoir choisi la note fondamentale (1) et la structure harmonique (2) dans les menus déroulants respectifs, clique sur le bouton *Ajouter* (3).
Sortie	L'accord choisi s'ajoute alors dans la grille d'accords (16) à la suite des accords précédents.

Générer une suite de gammes

Entrée	Après avoir choisi une liste d'accords (16) et sélectionné un ou plusieurs paramètres (4), l'utilisateur clique sur le bouton *Générer* (5).
Action	L'algorithme de création de la suite de gammes est lancé, en prenant en compte le ou les paramètres activés, avec en entrée la grille d'accords saisie.
Sortie	On retrouve en sortie la suite de gammes au format ".midi". Le titre du fichier midi apparaît dans la boîte en bas au centre de l'interface, avec la possibilité de modifier le titre du fichier (16) ou l'enregistrer (9).

Enregistrer la suite de gammes

Entrée	L'utilisateur clique sur le bouton *Enregistrer* (6).
Action	Une fenêtre d'exploration de l'arborescence du système s'ouvre, lui demandant le dossier cible dans lequel le fichier midi généré sera sauvegardé.
Sortie	Le fichier midi généré s'enregistre dans le dossier précédemment choisi par l'utilisateur.

Enregistrer une grille d'accords

Entrée	L'utilisateur clique sur l'onglet Fichier dans la barre de menu de l'interface et sélectionne le sous-onglet *Enregistrer la grille* (14) ou *Enregistrer la grille sous* (15) pour pouvoir exporter la grille d'accords dans un fichier au format *.txt* (selon le formalisme défini par le programme).
Action	Une fenêtre d'exploration du système de fichiers de la machine, sur laquelle a été lancé le programme, apparaît. L'utilisateur choisit le chemin d'accès où il veut que le fichier, représentant la grille d'accords, soit sauvegardé et il donne un nom au fichier.
Sortie	Le fichier correspondant à la grille d'accords est sauvegardé et accessible à l'adresse spécifiée par l'utilisateur.

Importer une grille d'accords

Entrée	L'utilisateur clique sur le bouton *Importer* (7).
Action	Un explorateur de fichier s'ouvre, l'utilisateur sélectionne le fichier au format .txt, contenant une grille d'accords, et valide.
Sortie	Si le document ne permet pas d'établir une grille d'accords, un message d'erreur apparaît. Sinon, la grille d'accords est affichée sur l'interface (16) et est prête à être utilisée.

Ajouter un accord au dictionnaire

Entrée	L'utilisateur clique sur le bouton *Ajouter* (10) dans l'onglet *Accords*.
Action	Une fenêtre lui permettant de créer une structure harmonique et de lui attribuer un nom s'ouvre.
Sortie	La structure harmonique précédemment créée s'ajoute au menu déroulant *Structure Harmonique* (2).

Ajouter une gamme au dictionnaire

Entrée	L'utilisateur clique sur le bouton *Ajouter* dans l'onglet *Gammes* (11).
Action	Une fenêtre lui permettant de créer une structure harmonique et de lui attribuer un nom s'ouvre.
Sortie	La structure harmonique précédemment créée s'ajoute au dictionnaire de gammes possibles à obtenir dans la suite de gammes générée par le logiciel.

2 Étude technique

2.1 structure du programme

Nous avons choisi de structurer notre logiciel en suivant le modèle MVC.

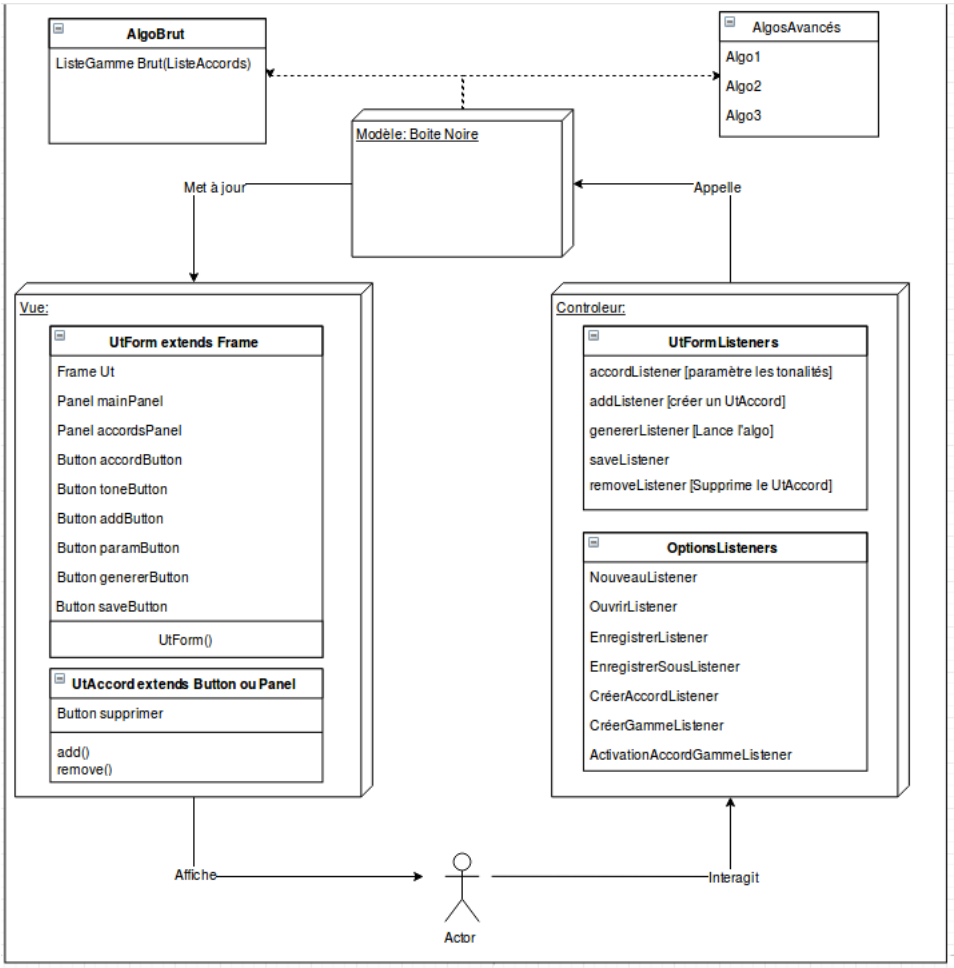


Figure 4.3 – Représentation du modèle MVC

Le choix d'utiliser le modèle MVC a été motivé par notre organisation de travail. Du fait que la recherche pour l'optimisation de l'algorithme principal et la réalisation du logiciel sont opérées en parallèle, il a fallu pouvoir tester différentes versions de l'algorithme au cours de son amélioration. Ainsi, afin de simplifier l'intégration régulière d'un nouvel algorithme dans le logiciel, nous avons opté pour le modèle MVC. Ce modèle permet de distinguer les classes selon leurs fonctionnalités:

- **Le Modèle** correspond au cœur du logiciel et à la partie algorithmique du programme. Une fois son action effectuée, il en informe la Vue afin qu'elle actualise son affichage.
- **La Vue** correspond à l'interface graphique du logiciel et à la communication du programme vers l'utilisateur.
- **Le Contrôleur** a pour rôle d'interpréter les actions de l'utilisateur et de les traiter, en faisant par exemple appel au Modèle.

Avec ce modèle de programmation, nous avons ainsi pu développer la Vue et le Contrôleur en parallèle avec les différentes versions du Modèle.

Voici le diagramme de classe de notre logiciel, découpé en 3 parties (modèle, vue et contrôleur) pour plus de clarté.

La classe Ut joue le rôle de contrôleur. Elle possède comme attribut une instance de la classe MainWindow, qui gère le coté interface graphique (la vue) du logiciel, et une instance d'une classe fille de AbstractAlgo, qui gère la partie algorithmique (le modèle). Lorsque l'utilisateur interagit avec la vue, un signal est envoyé par la vue au contrôleur, qui lance ensuite la ou les méthodes correspondantes auprès du modèle, réceptionne le résultat et demande à la vue de l'afficher.

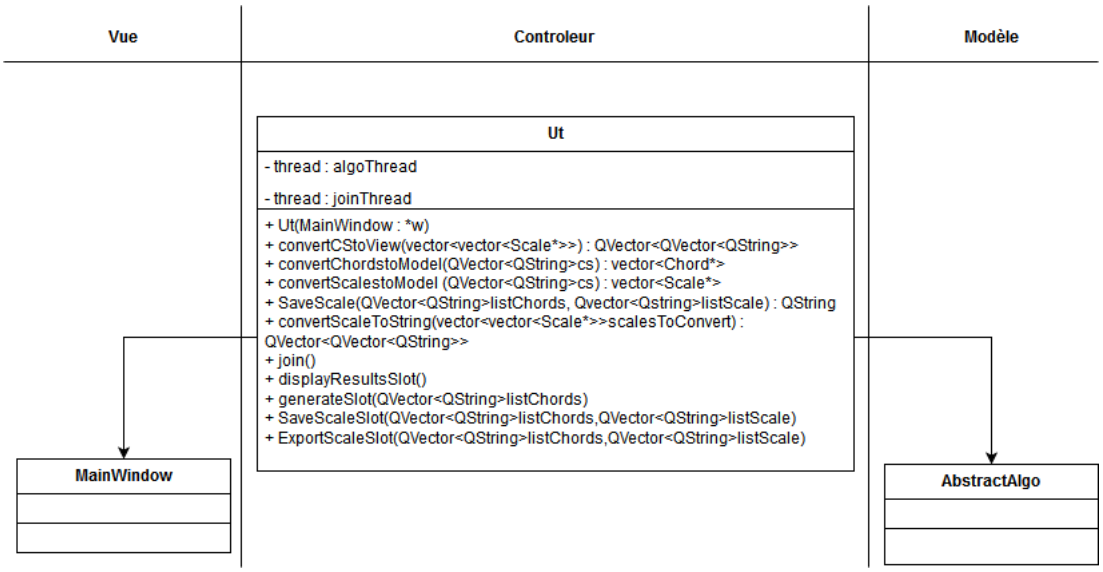


Figure 4.4 – Diagramme de classe

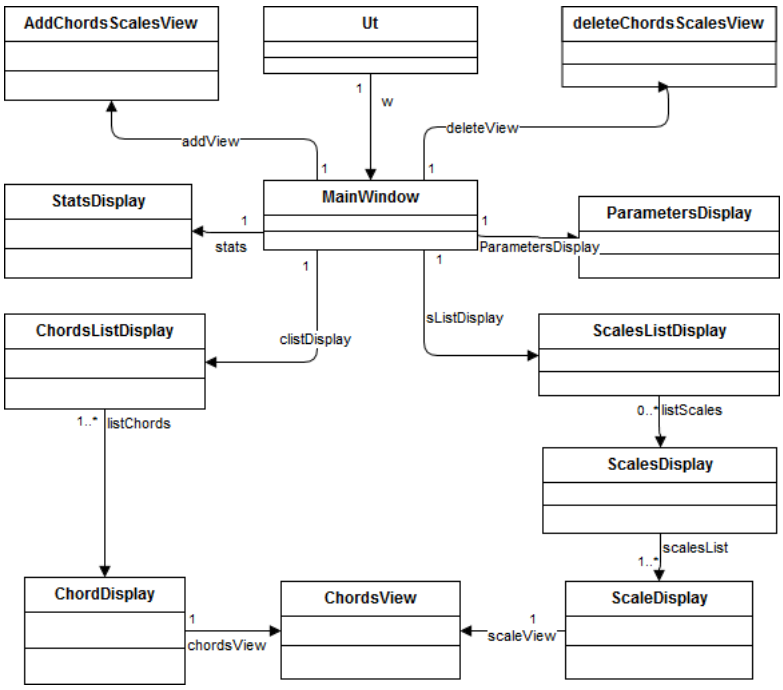


Figure 4.5 – Diagramme de classe de la vue

La vue est centrée autour de la classe MainWindow, qui possède comme attribut les différentes classe permettant d'afficher l'interface graphique. C'est elle qui gère en interne tout ce qui ne demande pas l'intervention du modèle, comme par exemple la gestion de la fenêtre permettant à l'utilisateur de choisir avec quels paramètres il veut lancer la génération des résultats. En revanche, dès que l'intervention du modèle est nécessaire, elle passe la main au contrôleur.

La classe MainWindow est la fenêtre affichée en permanence. Elle est composée d'instance de la classe ChordListDisplay qui affiche la liste des accords déjà sélectionnés par l'utilisateur, chacun représenté par une instance de la classe ChordDisplay. Lorsque l'utilisateur décide de lancer l'algorithme, une instance de la classe ScaleListDisplay s'affiche dans la MainWindow pour présenter les différents résultats. Chaque résultat est une instance de la classe ScalesDisplay, contenant une liste de ScaleDisplay qui affiche les Gammes trouvées pour chaque solution.

D'autres fenêtres sont appelées par la MainWindow selon les actions de l'utilisateurs :

- AddChordsScalesView lorsque l'utilisateur souhaite créer un nouvel accord ou une nouvelle gamme.
- DeleteChordsScalesView lorsque l'utilisateur souhaite supprimer un accord ou une gamme.

- ChordsView lorsque l'utilisateur souhaite voir a quoi ressemble un accord ou une gamme.
- ParametersDisplay lorsque l'utilisateur souhaite choisir les paramètres avec lesquels l'algorithme sera lancé.
- StatsDisplay lorsque l'utilisateur souhaite voir les performances des différents algorithmes.

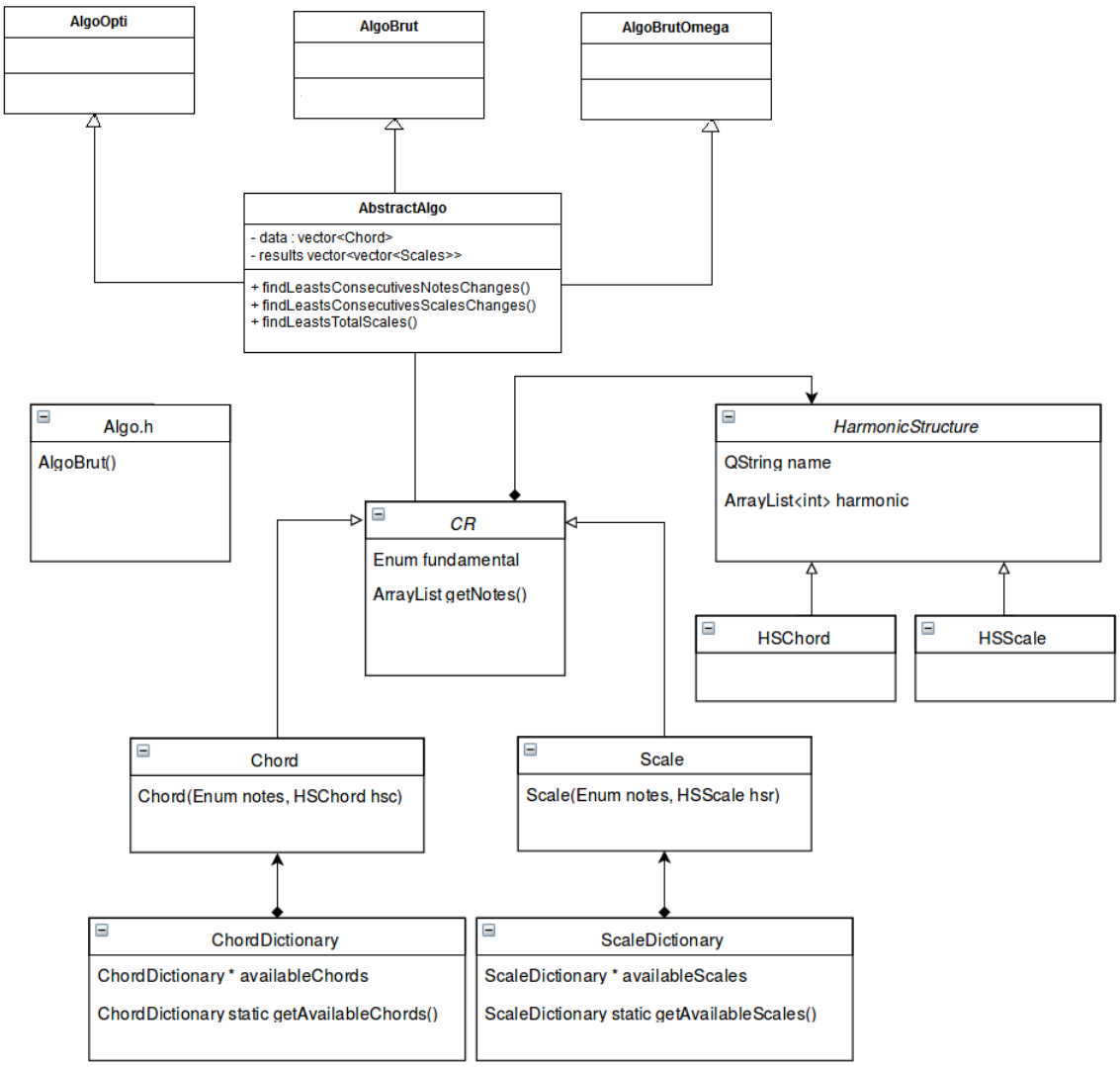


Figure 4.6 – Diagramme de classe du modèle

Le contrôleur intéragit avec le modèle par l'intermédiaire de la classe AbstractAlgo. Le rôle de celle-ci est de prendre une liste d'accords en entrée et de donner une liste de gammes, conformément aux instructions données par l'utilisateur à travers l'interface graphique du logiciel. On retrouve les classes des objets Chord (représentant les accords) et Scale (représentant les gammes). Ces objets contiennent les mêmes attributs; ils héritent donc d'une classe mère CR, mais dépendent de différents dictionnaires (ChordDictionnary et Range-Dictionnary). Ceux-ci sont manipulés en tant que singleton. Parmi les attributs des Chord et Scale, on retrouve les HarmonicStructure, des éléments comprenant un nom, les identifiants et une suite d'entiers permettant de caractériser les différents accords et les différentes gammes de l'algorithme. On distingue les HarmonicStructure associées à des accords (HSChord) et celles associées à des gammes (HSRange).

Par souci de flexibilité, les différents algorithmes pouvant être utilisés pour la génération des solutions héritent d'une classe abstraite AbstractAlgo. Ainsi le contrôleur possède comme attribut un objet de type AbstractAlgo, qui, lors du lancement de la recherche de solution, est instancié en un des trois types d'algorithme en fonction du choix de l'utilisateur. Cela permet de rajouter facilement des algorithmes sans avoir à modifier le contrôleur. Le résultat de la recherche est stocké dans l'attribut "results", récupéré ensuite par le contrôleur pour demander à la vue de l'afficher.

2.2 Structures et supports de stockage des dictionnaires d'accords et de gammes

- Les dictionnaires de gammes et d'accords engloberont respectivement l'ensemble des gammes et l'ensemble des accords autorisés. Seule la structure harmonique est stockée puisqu'on considérera que toute fondamentale, quelle que soit la structure harmonique existante, fait partie du dictionnaire des gammes ou accords autorisés.
- Chaque dictionnaire suivra le design pattern *Singleton* afin de conserver des dictionnaires uniformes à travers l'intégrité du programme.
- Chaque structure harmonique sera sérialisable selon son nom.
- Les structures harmoniques, de gammes ou d'accords, créées par l'utilisateur seront stockées dans un fichier texte afin de pouvoir les régénérer à chaque ouverture du programme.
- Les structures harmoniques définies par des conventions musicales seront stockées directement dans le code en dur et seront donc générées à chaque lancement du logiciel, cela afin d'éviter les problèmes liés à une potentielle corruption du fichier lors de la sérialisation des données, ou simplement d'éviter que l'utilisateur modifie les accords et gammes prédéfinis. La liste des structures, générée de base, est la suivante:

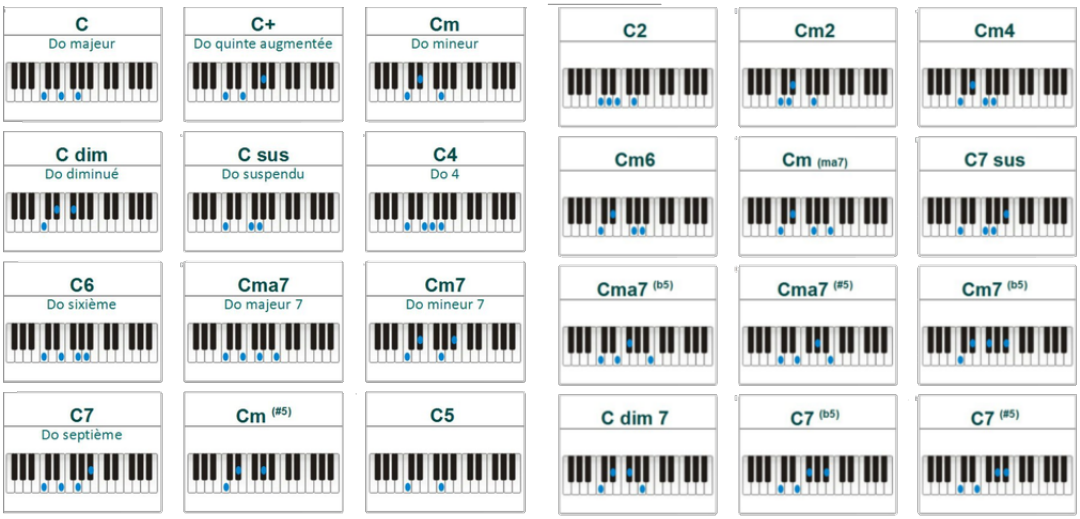


Figure 4.7 – Structures d'accords autorisées de base pour le Do (C)

2.3 Choix des outils

Le logiciel sera réalisé avec le langage C++. En effet, un des buts fondamentaux de ce projet étant l'optimisation d'un algorithme, nous avons opté pour un langage performant nous permettant ainsi d'observer avec les meilleures capacités les algorithme implémentés. Cela pris en compte, le langage C++ s'est démarqué des langages possibles par le fait qu'il permet l'utilisation de l'IDE Qt.

Afin de pouvoir observer les performances de notre algorithme, il est nécessaire d'implémenter une interface graphique simple, mais efficace. Qt permet la création rapide de tels interfaces.

Par ailleurs, il est important de pouvoir manipuler des fichiers au format MIDI, afin de juger de la qualité du résultat de notre algorithme. Qt propose à cet effet une librairie permettant de gérer les fichiers d'entrée et de sortie MIDI.

Ainsi, nous avons choisi cet IDE pour pouvoir réaliser aussi bien la partie programmation du projet que la partie graphique, sans avoir de problèmes de compatibilité que pourrait entraîner l'utilisation de différents IDE.

Ce projet impliquant l'implémentation totale d'un logiciel, nous n'utiliserons que peu de bibliothèques préfaites, en nous limitant à celles fournies par Qt.

2.4 Normes de développement

La mise en forme du code sera rédigée selon les normes utilisées en cours :

- Noms des classes et interfaces commençant avec une majuscule.

- Majuscule à chaque début de mot du nom d'une méthode sauf le premier.
- Noms des attributs et variables en minuscule.
- Chaque variable définie sur une ligne propre.
- Commentaires concernant une portion de code sur lignes concernées.
- Pas d'espace entre les parenthèses, crochets et les accolades ouvrantes.
- Espace entre les opérateurs.
- Mise à la ligne avant et après les accolades, crochets et parenthèses fermantes seules.
- Plusieurs passages à la ligne entre les définitions des fonctions, méthodes, classes et interfaces.
- Indentation à chaque portion de code imbriquée.
- Alignement des lignes à chaque coupure d'une expression.

La création du squelette du code a été réalisée en réunion afin d'éviter les soucis de versionning que peut entraîner l'utilisation de GitHub. L'écriture approfondie a ensuite été répartie selon les différentes parties du modèle MVC, la responsabilité d'une partie dépendant d'une personne. Ceci pour éviter la modification d'un même point du programme au même moment, et donc éviter la création de deux versions du programme.

Comme indiqué dans le diagramme de Gantt, plusieurs versions du logiciel seront implémentées au cours du projet.

La première version sera relativement simple, se limitant aux consignes de base de l'encadrant. Sa mise en application sera faite le plus rapidement possible afin de pouvoir tester l'efficacité des algorithmes de résolution de la problématique. D'autres versions du logiciel verront le jour, dépendamment de l'avancement des différentes équipes du projet.

3 Export des suites de gammes et d'accords

Cette section présente les différents formats d'exports des suites de gammes ou des suites d'accords entrées par l'utilisateur ou générées par le programme Ut.

3.1 Sortie au format texte

Une suite de gammes solution est exportable au format texte. Le fichier de texte de sortie présente la grille d'accords en entrée sur une ligne et la suite de gammes solution associée sur la ligne suivante.

Par exemple, pour la grille d'accords CM, FM, GM, CM, on obtient la suite de gammes CM, CM, CM, CM. Le fichier texte de sortie aura le format suivant :

Accords	CM	FM	GM	CM
Gammes	CM	CM	CM	CM

3.2 Sortie au format Music XML

Dans le cahier des charges, un des formats de sortie pour une suite de gammes solution devait être le format MIDI. Le format MIDI est un format standardisé pour la synthèse numérique de sons. Ce format est largement répandu, ce qui permet de trouver une grande variété de traitements de ces fichiers, notamment des lecteurs audio, ou des afficheurs du contenu sous forme de partition. Ce format est donc idéal pour les utilisateurs potentiels de notre logiciel.

Le format MIDI permet une très grande liberté de création et comporte de nombreux paramètres. Cependant, la grande variété des paramètres qui est offerte rend ce format difficile à manipuler directement.

Après discussion, nous avons convenu d'utiliser un autre format de sortie pour les suites de gammes que le format MIDI. Nous nous sommes orientés vers la production en sortie d'un fichier au format Music XML.

Le format Music XML permet l'échange de partition numérique via l'utilisation du langage XML. Il a été développé par l'entreprise Recordare LLC pour leur logiciel *Recordare*.

De nombreux logiciels[2] d'édition de partition numérique implémente ce format comme standard d'échange, ce qui justifie l'utilisation du format Music XML pour l'export de suites de gammes et d'accords.

[2]Voici quelques exemples de logiciels utilisant le format Music XML pour importer ou exporter des partitions numériques : *Finale*(http://www.finalemusic.com/), Muse Score (https://musescore.org), Guitar Pro (https://www.guitar-pro.com)

Nous avons décidé de produire le format Music XML en sortie et de ce fait de laisser le soin aux nombreux logiciels capables de lire ce format de le convertir en MIDI ou en WAVE (format de fichier audio).

La proposition d'une sortie au format MIDI pour notre logiciel sera proposée dans une version ultérieure.

3.3 Description du format Music XML

Nous nous sommes basés sur la spécification du format Music XML par l'entreprise Make Music[5] pour construire nos fichiers de sortie.

Le format Music XML se présente comme suit :

- Un "header" : Il comporte les balises signature du XML, ainsi que les balises nécessaires à la déclaration du format MusicXML

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE score-partwise PUBLIC
"-//Recordare//DTD MusicXML 3.0 Partwise//EN"
"http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise version="3.0">
```

- Un sommaire : La balise <part-list> permet de lister les différentes *"voix"*[3] de la partition, afin de les identifier. Dans notre cas, nous avons besoin de produire deux voix en sortie, une pour les accords et un pour les gammes.

- La définition des différentes voix: Au sein des balises <part> préalablement déclarées dans le sommaire sont définis les gammes ou accords.

Une fois les balises d'ouverture définies, la sérialisation au format MusicXML traite les gammes et les accords un à un, en commençant par les gammes. Le traitement de celles-ci se fait note par note, en définissant les notes comme des tonalités ponctuelles et finies, afin de permettre une différenciation claire des tons lors de l'écoute. Le MusicXML permet de modifier de nombreux paramètres lors de la définition des notes.

Les notes des accords sont elles définies comme des tonalités soutenues ininterrompues, ce qui les place au second plan, lors de l'écoute et permet de les dissocier des gammes qui nous intéressent davantage.

Pour finir, il s'agit de fermer les balises ouvertes, dans la première partie du fichier et de permettre à l'utilisateur d'enregistrer le fichier ainsi complété.

Une fois le fichier MusicXML acquis, l'utilisateur a la possibilité de le lire et de le convertir au format MIDI ou WAVE. Nous conseillons l'utilisation du logiciel *Muse Score* pour la lecture des fichiers au format Music XML produits par notre logiciel.

[3] Une voix correspond à une piste sur la partition. Dans une partition d'orchestre, on a par exemple un voix pour le piano, une voix pour les violons, ...

Présentation des résultats

1 Étude comparative des algorithmes

Cette partie s'intéresse aux performances des algorithmes que nous avons pu développer pour le calcul des suites de gammes solutions vis à vis d'une suite d'accords.

Ici, les performances des algorithmes sont comparées afin de montrer heuristiquement l'amélioration du temps de calcul apporté par notre algorithme le plus performant : *Plus-Courts-Chemins-GSS-Modifie* (cf. chapitre 3, section 4)

1.1 Configuration des ordinateurs de test

Les algorithmes ont été testés sur deux machines différentes, utilisant deux systèmes d'exploitation différents, *Linux* et *Windows*.

Les deux machines sont similaires au niveau de la capacité en mémoire vive, mais pour la puissance de calcul la processeur Intel de la machine Windows est plus performant que le processeur AMD de la machine Linux. Leur configuration physique correspond à la configuration de *lap top* "moyen", couramment utilisé par le grand public.

Le logiciel Ut ayant pour but d'être utilisé en tant qu'exécutable léger s'adaptant à toute puissance de calcul, les tests des algorithmes sur des machines à configuration répandue était plus pertinent.

Le tableau 5.1 présente en détails la configuration des deux machines *test* :

	Machine Windows	***Machine Linux***
Système d'exploitation	Windows 10 Famille	Linux Mint 18.1 Serena
Processeur	Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz 2.30GHz	AMD E1-2100 APU with Radeon(TM) HD Graphics CPU @ 800MHz 1GHz
Mémoire RAM	4,00 Go	4,00 Go
Architecture	x86_64 bits	x86_64 bits

Table 5.1 – Configuration des machines test

1.2 Paramètres de test

Pour les test menés sur les deux différentes machines présentées au-dessus, hormis le nombre d'accords en entrée qui est variable, les valeurs pour les paramètres des algorithmes sont les suivantes :

→ **Contrainte à minimiser** = Nombre de changements de gammes dans les suites de gammes solutions (une comparaison des performances des algorithmes vis à vis des contraintes utilisées n'est pas faite ici).

→ **Nombre de solutions retenues** = 2, l'affichage des solutions prenant beaucoup d'espace mémoire, le nombre de solutions à retenir est fixé à 2 pour ne pas exploser l'espace de pile.

→ **Dictionnaire des gammes utilisé** = toutes les gammes, l'ensemble des gammes contenues dans le dictionnaire natif du logiciel sont sélectionnées pour le lancement des tests.

Le dictionnaire natif comprend les gammes suivantes :

{Mineure harmonique, Mineure mélodique, Majeure, Pentatonique mineure, Pentatonique blues, Pentatonique majeure, Égyptienne, Bartok, Par ton, Diminuee}

1.3 Test sur Machine Windows

La figure 5.1 présente, sur une machine Windows, l'évolution des temps de calcul des algorithmes en fonction de la taille de la suite d'accords en entrée et pour les paramètres énoncés ci-avant.

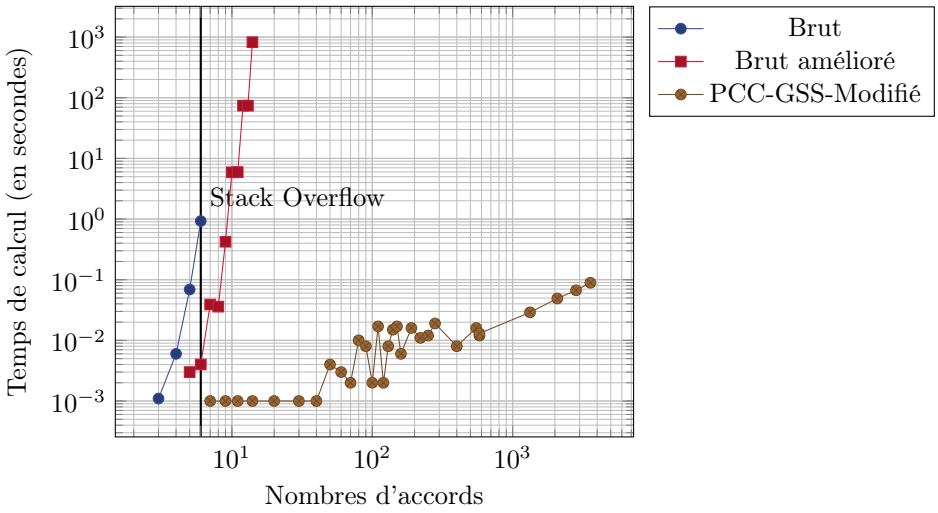


Figure 5.1 – Comparaison des performances des algorithmes sur une machine Windows

Analyse du graphique Pour l'algorithme brut, on observe que le premier facteur discriminant est la consommation mémoire. En effet, pour une entrée de 6 accords, l'algorithme brut génère un débordement de pile qui a pour effet d'arrêter le processus. Cela s'explique par la stratégie de l'algorithme brut qui consiste à générer toutes les suites de gammes solutions avant de les trier. Les temps de calcul pour l'algorithme brut dans le cas d'une entrée supérieure à 6 accords ne sont donc pas connus sur Windows. En effet, nous n'avons pas réussi à augmenter la taille de pile pour les processus sur Windows, afin de pouvoir lancer nos calculs sur l'algorithme brut. La figure détaille l'évolution de la consommation mémoire de l'algorithme brut sur un système Linux, système pour lequel la taille par défaut de la pile des processus est modulable.

Pour l'algorithme brut amélioré, la consommation mémoire n'est plus un problème puisque l'algorithme filtre les suites de gammes solutions au fur et à mesure et déleste donc la mémoire. Cependant, on observe pour des différences très faibles entre taille d'entrée, les temps de calcul augmente très rapidement. On retrouve dans les résultats de l'étude expérimentale la complexité de l'algorithme brut et brut amélioré, complexité qui est de l'ordre de $O(n^k)$, avec n la taille moyenne d'une k-partie d'un GAKO (cf.chapitre 3, section 2).

Pour l'algorithme *Plus-Courts-Chemins-GSS-Modifie*, le temps de calcul augmente lentement et la mémoire devient un problème pour une entrée de l'ordre 3000 accords, ce qui suffit largement pour notre domaine d'application, un morceau de musique étant rarement composé de 3000 accords. La dernière mesure pour l'algorithme *Plus-Courts-Chemins-GSS-Modifie* avant dépassement de la mémoire (sur Windows) a été effectuée pour une entrée de 3580 accords pour laquelle le calcul a durée 89 ms.

1.4 Test sur Machine Linux

La figure présente, sur une machine Linux, l'évolution des temps de calcul des algorithmes en fonction de la taille de la suite d'accords en entrée. Linux présente un avantage pour nos tests par rapport à Windows puisque la taille de la pile allouée aux processus est modifiable, ce qui nous a permis de lancer des calculs pour des entrées plus grandes pour l'algorithme brut et l'algorithme *Plus-Courts-Chemins-GSS-Modifie*.

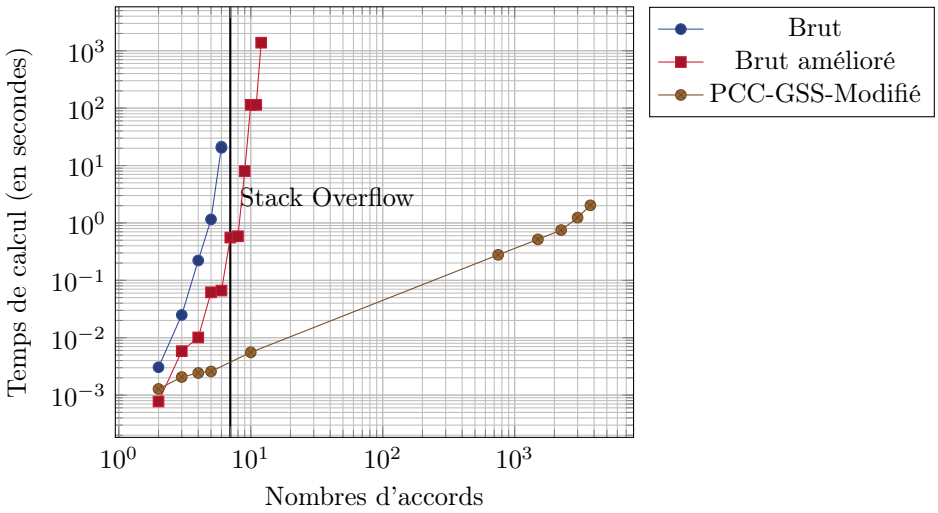


Figure 5.2 – Comparaison des performances des algorithmes sur une machine Linux

Dans la figure 5.2, on observe comme prévu les mêmes comportements que pour les tests effectués sur la machine Windows. Les temps de calcul sont plus longs sur la machine Linux qui possède un processeur moins performant.

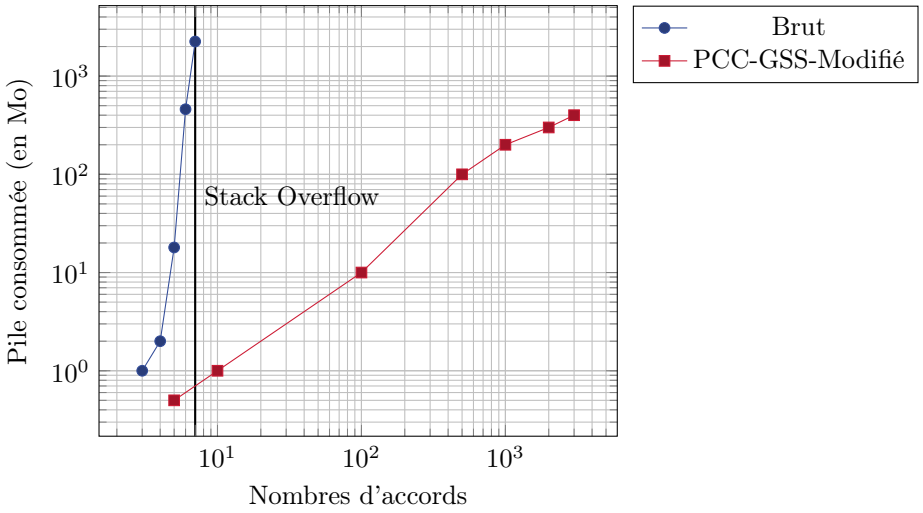


Figure 5.3 – Consommation de la pile pour l'algorithme brut et l'algorithme *Plus-Courts-Chemins-GSS-Modifie*

Pour l'algorithme brut, le débordement de pile se passe pour une taille d'entrée de 7 accords. On observe que la consommation de pile pour l'algorithme brut augmente rapidement pour un changement sensible du nombre d'accords en entrée, là où la consommation de pile pour l'algorithme *Plus-Courts-Chemins-GSS-Modifie* reste stable.

1.5 Bilan sur les algorithmes

Par l'étude expérimentale, on observe que l'algorithme *Plus-Courts-Chemins-GSS-Modifie* est plus performant que l'algorithme brut et brut amélioré, que ce soit en termes de temps de calcul ou de consommation mémoire. Notre objectif est atteint pour ce qui est du traitement de l'explosion combinatoire dont on est témoin au lancement des algorithmes brut et brut amélioré.

Cependant, l'algorithme *Plus-Courts-Chemins-GSS-Modifie* a pour défaut qu'il ne peut pas calculer de suites de gammes solutions minimisant la contrainte du nombre total de gammes utilisées. Les algorithmes brut et brut amélioré peuvent eux répondre à cette contrainte, mais on a pu observé leur manque d'efficacité même pour de petites entrées.

Un autre algorithme devra être mis au point pour répondre à cette contrainte en respectant un temps de calcul et une consommation mémoire raisonnable, c'est à dire de l'ordre de l'algorithme *Plus-Courts-Chemins-GSS-Modifie*.

2 Manuel d'utilisation

Le présent manuel décrit l'utilisation du logiciel *Ut* qui est le résultat du développement et de la recherche effectués tout au long de notre projet tuteuré. Des exécutables pour les plateformes Windows et Linux sont disponibles sur le dépôt GitHub[1] du projet.

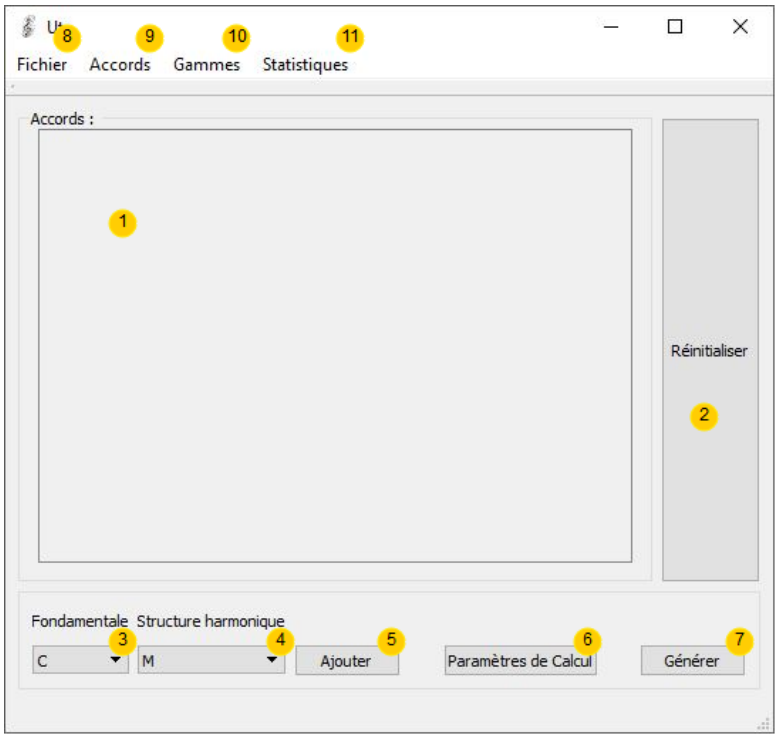


Figure 5.4 – Interface

Présentation de l'interface :

(1) **Fenêtre de visualisation** Fenêtre d'affichage des accords ajoutés.

(2) **Réinitialiser** Bouton effectuant une remise à zéro de l'affichage des accords ajoutés.

(3) **Fondamentale** Menu déroulant permettant de choisir la note fondamentale parmi les 12 notes de la gamme chromatique.(Cf. 2.1)

(4) **Structure harmonique** Menu déroulant permettant de choisir une structure harmonique parmi celles proposées par défaut et celles créées par l'utilisateur.(Cf. 2.1)

(5) **Ajouter** Bouton permettant d'ajouter un accord à la grille d'accords en fonction de la fondamentale et de la structure harmonique choisies.(Cf. 2.1)

(6) **Paramètres de calcul** Bouton donnant accès aux différents paramètres à sélectionner pour effectuer le calcul selon les critères désirés.(Cf. 2.2)

(7) **Générer** Bouton Permettant de générer la suite de gammes à partir des accords visibles en (1). (Cf. 2.3)

(8) **Fichier** Onglet permettant de :

- **Réinitialiser** : Avec l'option Nouveau.
- **Importer** une suite d'accords à partir d'un fichier texte.
- **Enregistrer** une suite d'accords, créée à partir de l'interface et visible en (1), dans un fichier texte.
- **Quitter** le logiciel.

.

(9) **Accords** Onglet permettant de créer des structure harmoniques d'accords ou de les supprimer .(Cf.2.4)

(10) **Gammes** Onglet permettant de créer des structure harmoniques de gammes ou de les supprimer.(Cf. 2.4)

(11) **Statistiques** Onglet permettant l'affichage des différentes statistiques. (Cf. 2.5)

[1] https://github.com/viampietro/gammes_musicales/runnable

2.1 Ajouter des accords

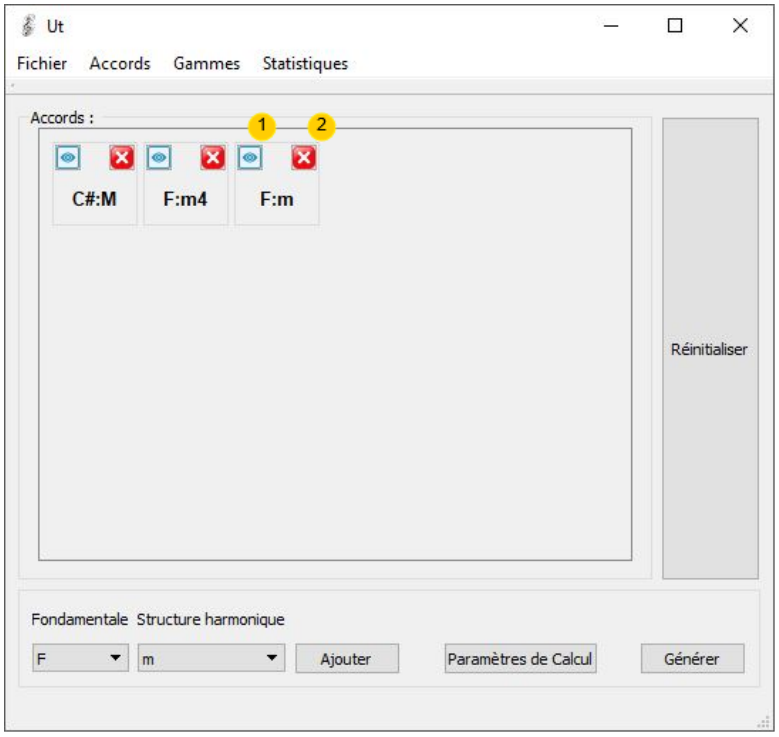
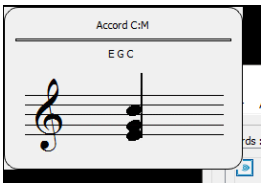


Figure 5.5 – Interface après ajout d'accords

Pour ajouter un accord, il faut simplement sélectionner sa note fondamentale dans la liste déroulante en (3) de la figure 5.4.
Suite à cette sélection, il reste à choisir la structure harmonique de cet accord dans la liste déroulant marqué par (4) sur la figure 5.4.
Pour finaliser, il ne reste plus qu'a appuyer sur le bouton **Ajouter** ((5) figure 5.4), pour pouvoir l'ajouter à la suite d'accords visible dans la **Fenêtre de visualisation** (1) de la figure 5.4.

Après ajout de plusieurs accords, le résultat est semblable à celui de la figure ci-contre 5.5:

(1) **Visualisation accord** Si l'on clique et que l'on reste appuyé sur l'œil l'on a une visualisation de l'accord.



(2) **Suppression accord** En cliquant sur l'image de croix l'on supprime l'accord de la suite.

2.2 Modifier les paramètres de calcul

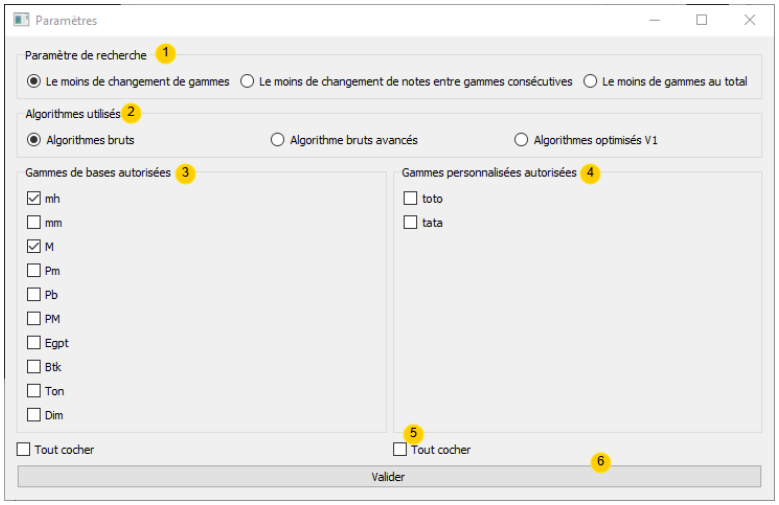


Figure 5.6 – Interface après ajout d'accords

Avant de lancer le calcul sur les accords, en cliquant sur le bouton **Paramètres de calcul** ((6) de la figure 5.4) l'on accède à une fenêtre donnant accès aux différents paramètres de calcul de suites de gammes :

(1) **Paramètres de recherche** Un seul choix possible de contrainte pour le calcul de suite de gammes.

(2) **Algorithmes utilisés** Un seul choix possible d'algorithme calculant des suites de gammes selon le paramètre sélectionné au dessus. Une modification de la fenêtre est générée par la sélection de l'**Algorithme optimisé v1**; il n'est plus possible d'utiliser la contrainte du nombre de gammes au total et une nouvelle ligne, permettant de saisir un nombre de solutions maximum à afficher, apparaît.

(3) **Gammes de bases autorisées** Les gammes, initiales du logiciel, sélectionnées seront celles utilisées par l'algorithme en plus des gammes personnalisées sélectionnées.

(4) **Gammes personnalisées autorisées** Les gammes, ajoutées par l'utilisateur, sélectionnées seront celles utilisées par l'algorithme en plus des gammes de bases sélectionnées.

(5) **Tout cocher** Si l'utilisateur coche cette case alors toutes les cases sont sélectionnées, s'il la décoche alors toutes les cases ne sont plus sélectionnées.

(6) **Valider** L'utilisateur doit valider pour prendre en compte ses modifications dans les paramètres.

2.3 Générer une liste de suite de gammes

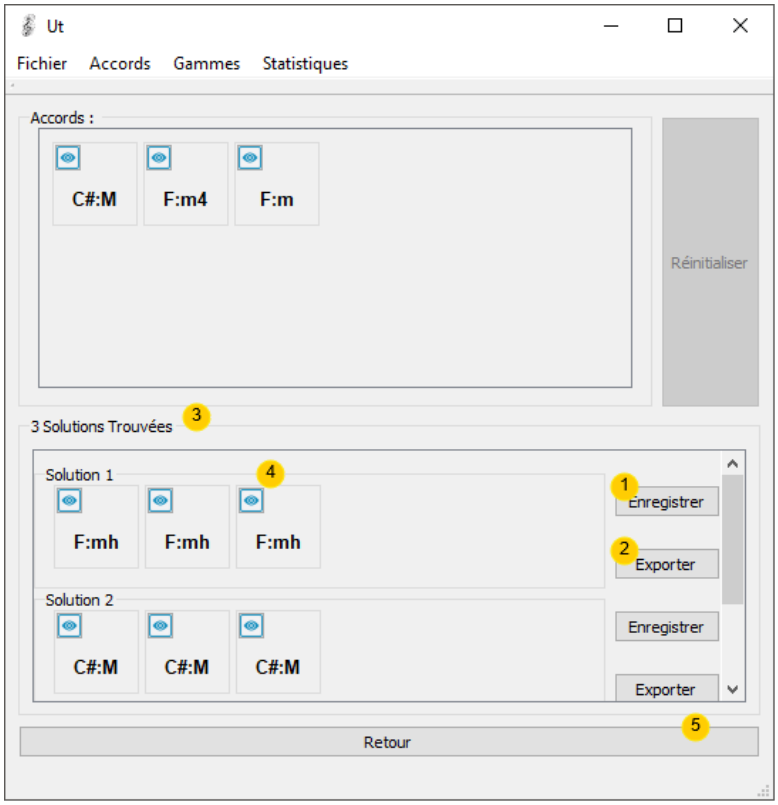


Figure 5.7 – Interface après génération des suites de gammes

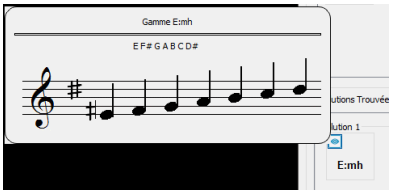
Une fois qu'une suite d'accords est entrée, l'on peut dès lors générer une liste des suites de gammes associée à cette suite d'accords, en cliquant sur le bouton **Générer** (7) sur la figure 5.4.
L'appuie sur ce bouton entraînera l'affichage des suites de gammes comme ci-contre :

(1) **Enregistrer** Enregistre, sous fichier texte, la suite de gammes.

(2) **Exporter** Exporte, au format MusicXml, la suite de gammes avec la suite d'accord associée.

(3) **Solutions trouvées** Affiche le nombre totale de suite de gammes trouvées après calcul.

(4) **Visualisation d'une gamme** Lors d'un clic prolongé, une visualisation de la gamme concernée est proposée à l'utilisateur.



(5) **Retour** Retour à l'affichage initial de l'interface avec la suite d'accord sélectionnée. (Comme sur la figure 5.5)

2.4 Création/Suppression d'une nouvelle structure harmonique d'accords ou de gammes

Deux onglets permettent de créer ou de supprimer une structure harmonique : **Accords** ((9) figure 5.4) et **Gammes** ((10) figure 5.4).

Création

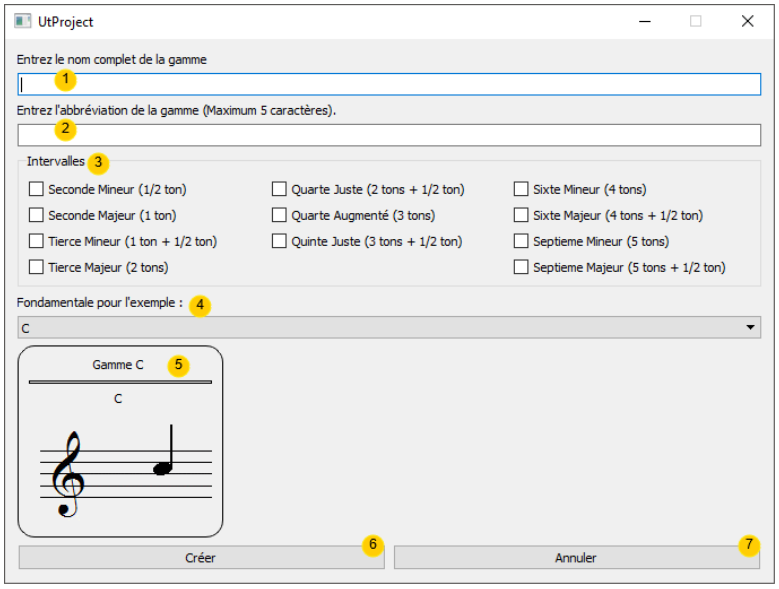


Figure 5.8 – Ajout d'une structure harmonique

En cliquant sur **Ajouter**, dans les onglets **Accords**, une nouvelle fenêtre s'ouvre :

(1) **Nom** On entre ici le nom de la gamme ou de l'accord.

(2) **Abréviation** on saisi ici l'abréviation de la gamme (non disponible pour l'accord).

(3) **Intervalles** On coche les intervalles pour la structure harmonique.

(4) **Fondamentale** Sélection de la fondamentale pou l'aspect visuel en (5).

(5) **Visualisation** Visuelle de la structure harmonique, en fonction de la fondamentale choisie en (4).

(6) **Créer** Crée la structure harmonique de la gamme ou de l'accord selon les critères sélectionnés.

(7) **Annuler** Annule la création de la structure harmonique.

Suppression

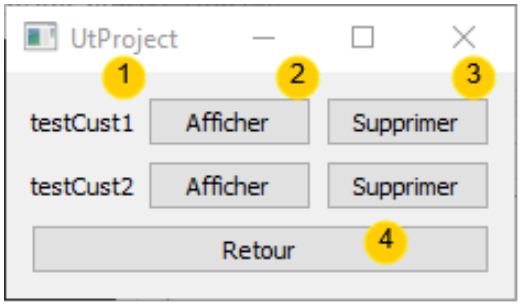


Figure 5.9 – Suppression d'une structure harmonique personnelle

En cliquant sur **Supprimer**, dans les onglets **Accords**, une nouvelle fenêtre s'ouvre :

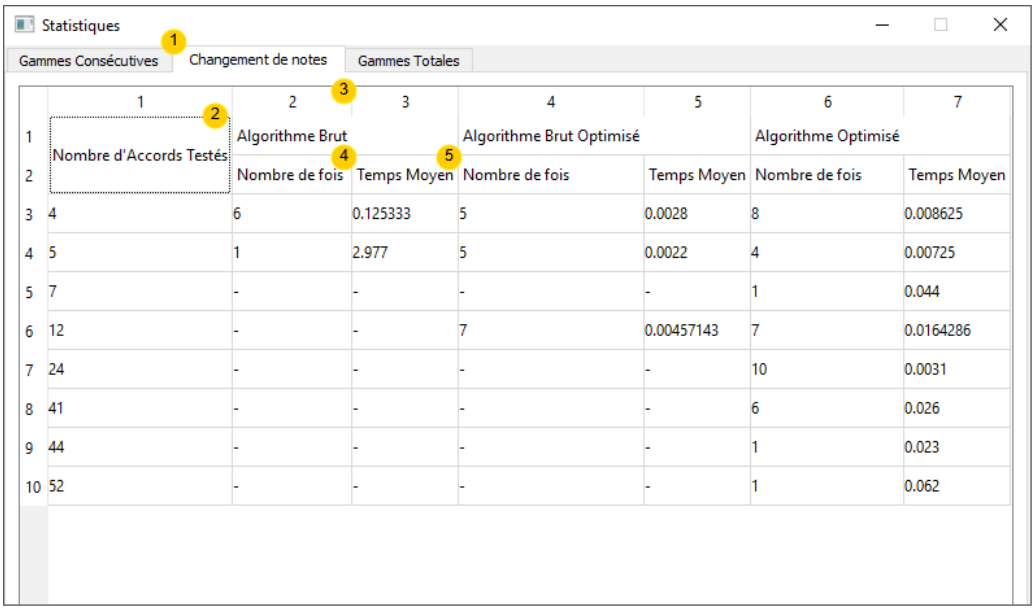
(1) **Nom** Le nom de la structure harmonique est affiché.

(2) **Afficher** Donne un visuel sur la structure harmonique comme en (4) de la figure 5.7 pour les gammes ou comme en (1) de la figure 5.5 pour les accords.

(3) **Supprimer** Supprime la structure harmonique.

(4) **Retour** Ferme la fenêtre.

2.5 Statistiques



	1	2	3	4	5	6	7
1	Nombre d'Accords Testés	Algorithme Brut		Algorithme Brut Optimisé		Algorithme Optimisé	
2		Nombre de fois	Temps Moyen	Nombre de fois	Temps Moyen	Nombre de fois	Temps Moyen
3	4	6	0.125333	5	0.0028	8	0.008625
4	5	1	2.977	5	0.0022	4	0.00725
5	7	-	-	-	-	1	0.044
6	12	-	-	7	0.00457143	7	0.0164286
7	24	-	-	-	-	10	0.0031
8	41	-	-	-	-	6	0.026
9	44	-	-	-	-	1	0.023
10	52	-	-	-	-	1	0.062

Figure 5.10 – Statistiques

En cliquant sur l'onglet **Statistiques**, une nouvelle fenêtre s'ouvre, laissant la possibilité l'utilisateur de consulter les performances du logiciel selon certains critères :

(1) **Contraintes** Choix de la contrainte dont l'on souhaite consulter les statistiques.

(2) **Nombre d'accords testés** Nombre d'accords sur lesquels sont testés les algorithmes.

(3) **Algorithmes** Les différents algorithmes utilisés pour répondre à la contrainte.

(4) **Nombre de tests** Le nombre de tests effectués selon le nombre d'accords, la contrainte et l'algorithme.

(5) **Temps moyen** Temps moyen de la génération des solutions selon les différentes contraintes.

Toutes ces informations sont mises à jours après chaque nouvelle génération de solutions.

Conclusion

Nous avons porté notre choix sur ce projet car certains d'entre-nous sont musiciens, et que d'autres étaient intéressés par l'aspect recherche qu'il offrait. Pour sa réalisation, nous avons fait le choix de vulgariser certains concepts de la musique pour les rendre plus accessibles.

Le but de ce projet était d'une part la résolution problème combinatoire, et d'autre part la production d'un logiciel fonctionnel et utilisable n'ayant pas pour unique but la recherche, mais aussi une certaine portée musicale. Nous voulions que le logiciel puisse être utilisé par des non-informaticiens dans le cadre, par exemple, de la composition d'un morceau.

En raison de cet aspect de recherche, nous avons écrit un rapport appliquant une certaine rigueur scientifique propre à la recherche (pour la partie solution algorithmique en particulier). Le but étant de comprendre les enjeux de la recherche en informatique et de s'y essayer à l'occasion de ce TER. Cela nous a donc permis d'explorer plusieurs aspects de l'informatique, allant de la recherche purement algorithmique, voire mathématique, à la production d'un logiciel opérationnel et pouvant être déployé hors du cadre de ce TER.

Les résultats de ce projet sont de notre point de vue satisfaisant au vue des résultats de l'étude comparative des algorithmes. Nous avons réussi a créer un algorithme efficace, notamment au vue des besoins du logiciel. Une autre approche aurait pu être celle présentée par l'article de recherche qui nous avait été donné à lire par notre professeur référant, M. Emeric Gioan, et présentée dans la section 6 du chapitre 3, mais nous avons choisi de ne pas l'implémenter, notre algorithme répondant au problème initial de manière suffisante.

Tout ce qui a été décrit dans ce rapport est présent dans la version rendue. Même si le public visé sont les musiciens, nous avons pris la liberté d'inclure à notre programme certains outils permettant de tester l'ensemble du travail éffectué, qui serait normalement absent d'un véritable logiciel (la possibilité de lancer la recherche avec les aglorithmes bruts, celle d'afficher les statistiques de chaque algorithme).
Nous aimerions néanmoins continuer d'améliorer notre programme afin de lui donner un aspect plus professionnel, en vue d'un possible déploiement sur le site de M Gioan. Ces améliorations pourraient consister par exemple à permettre le choix d'un paramètre secondaire de recherche permettant d'affiner la recherche de solutions et donc d'en proposer moins, et qui seraient potentiellement plus pertinentes en termes musicaux. On pourrait encore proposer une petite improvisation générée aléatoirement à l'aide des solutions trouvées, donnant ainsi un meilleur aperçu à l'utilisateur de ce que ses accords peuvent donner comme résultat.

Nous aimerions enfin remercier M Gioan pour sa disponibilité et son aide tout au long de la réalisation du logiciel et de la rédaction de notre rapport, notamment en nous guidant pour y intégrer une certaine rigueur scientifique lors de la rédaction des preuves algorithmiques.

List of Figures

Bibliography

[1] M. J. Atallah, D. Z. Chen, and D. T. Lee. "An optimal algorithm for shortest paths on weighted interval and circular-arc graphs, with applications". In: *Algorithmica* 14.5 (1995), pp. 429–441. ISSN: 1432-0541. DOI: 10.1007/BF01192049. URL: http://dx.doi.org/10.1007/BF01192049.

[2] Cormen et al. "Algorithmique". In: 3ème édition. 2009. Chap. 24.2, Plus courts chemins à origine unique dans les graphes acycliques orientés, pp. 606–608.

[3] Cormen et al. "Algorithmique". In: 3ème édition. 2009. Chap. 22.4, Tri topologique, p. 566.

[4] Cormen et al. "Algorithmique". In: 3ème édition. 2009. Chap. 24, Plus courts chemins à origine unique (Introduction), pp. 595–602.

[5] Make Music. *Music XML*. 2015. URL: http://usermanuals.musicxml.com/MusicXML/MusicXML.htm.