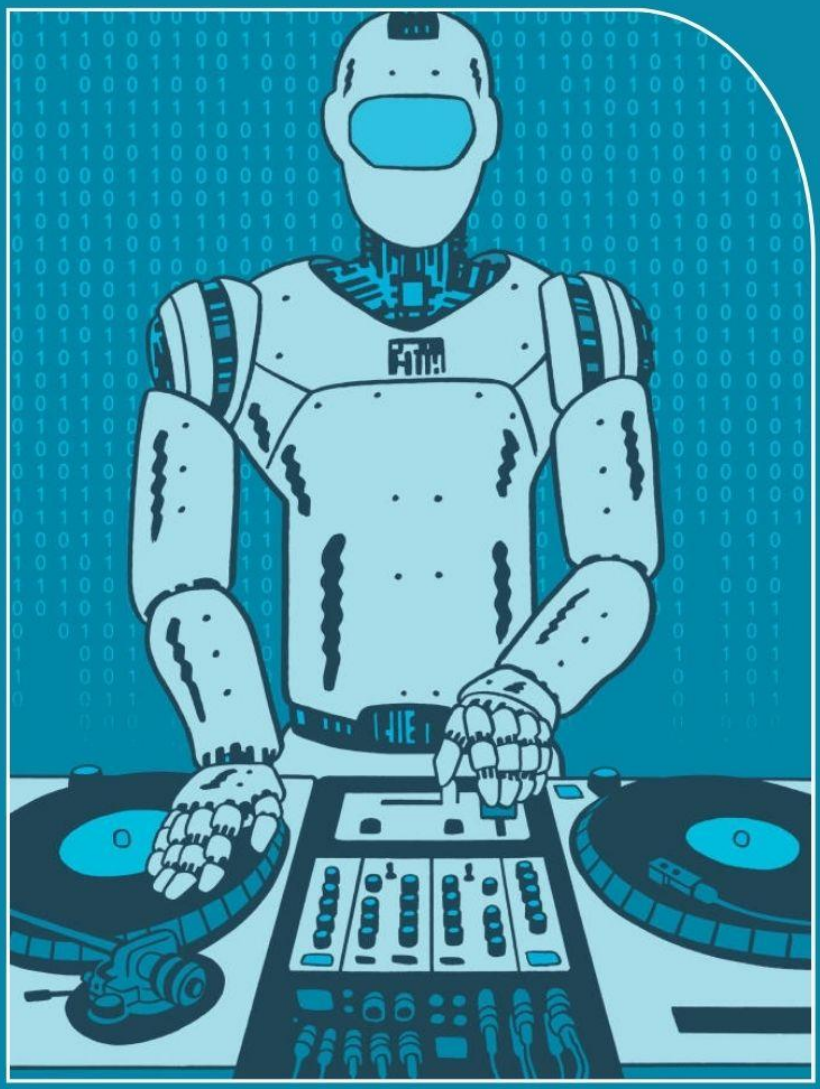
ENSAM
ÉCOLE NATIONALE SUPÉRIEURE D'ARTS ET MÉTIERS
UNIVERSITÉ HASSAN II DE CASABLANCA



Mini – Projet

Module : Mathématiques pour l'Ingénieur

Automate pour la génération de musique

par Gilles AKPINFA

Pr. LAZAIZ
Pr. KAMOUS

2023 - 2024

Table des matières

Introduction

La génération de musique algorithmique représente une intersection fascinante entre les arts et les sciences, combinant la créativité musicale avec les principes rigoureux des mathématiques et de l'informatique. Depuis des siècles, les musiciens et les compositeurs ont cherché à systématiser la composition musicale, et avec l'avènement de l'informatique moderne, cette exploration a pris une nouvelle dimension. La musique algorithmique, où les éléments musicaux sont générés par des algorithmes plutôt que composés directement par un humain, permet de créer de vastes variations musicales, parfois impossibles à réaliser manuellement. Les automates finis, en particulier, offrent un cadre puissant pour modéliser et générer des séquences musicales de manière structurée et prédictive, rendant cette approche particulièrement adaptée à l'expérimentation et à l'innovation musicale.

Ce mini-projet vise à concevoir et à implémenter un automate capable de générer des séquences musicales basées sur des règles définies. L'objectif principal est de démontrer comment les concepts des automates finis peuvent être appliqués à la création musicale, en explorant les possibilités offertes par la musique algorithmique. En développant cet automate, nous cherchons à :

- **Modéliser des motifs musicaux** en utilisant une approche orientée objet.
- **Développer des règles harmoniques et rythmiques** pour guider la génération musicale.
- **Implémenter des déclencheurs** internes et externes pour gérer les transitions et variations musicales.
- **Combiner créativité et technique** pour produire des séquences musicales variées et cohérentes.

I. Phase Initiale : Modélisation et Préparation de l'Automate

1. Modélisation orientée objet de l'automate

a) *Concepts de base des automates finis*

Les automates finis sont des modèles mathématiques utilisés pour représenter des systèmes avec un nombre fini d'états. Un automate fini Aut = (A, Q, I, T, E) est composé de :

- **Un ensemble d'états (Q)** : un ensemble fini d'états de l'automate.
- **Un alphabet (Σ)** : l'alphabet utilisé pour la construction de mots.
- **Un ensemble d'états initiaux (I)** : Un sous-ensemble de Q formant les états dits "initiaux", ou états "de départ"
- **Un ensemble d'états finaux (F)** : un sous-ensemble de Q formant les états dits "terminaux", ou états "finals", ou états "d'acceptation
- **Un ensemble de transition E** : un ensemble de triplets appelés "transitions" $E \subseteq Q \times A \times Q$

Ces concepts fondamentaux permettent de structurer un automate fini de manière à modéliser des processus complexes, tels que la génération de séquences musicales, en respectant des règles définies.

b) *Définition des classes et objets*

Pour modéliser un automate de manière orientée objet, nous définissons plusieurs classes représentant les éléments de base de l'automate :

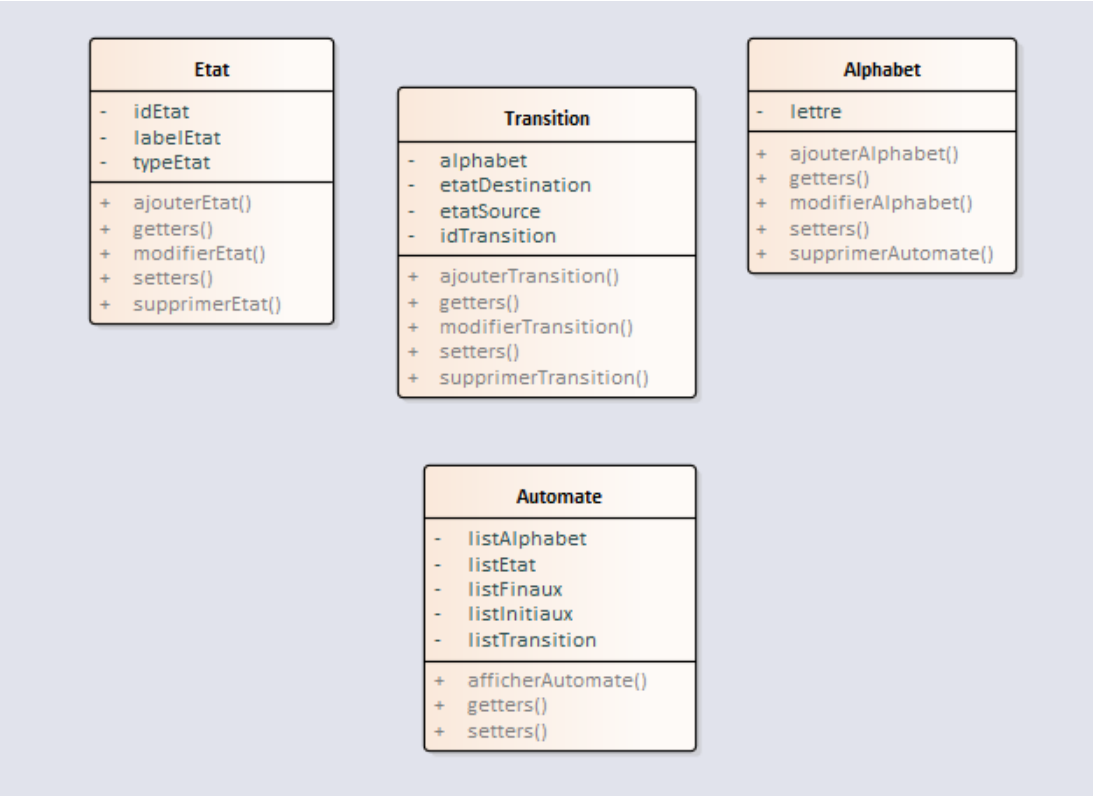


Figure 1 : Vue d'ensembles des classes

Remarques :

Chaque classe possède un constructeur et les attributs de chaque classe possèdent un getter et un setter.

c) *Fonction de lecture d'un automate*

La fonction **lire_automate** permet de lire et de créer un automate à partir de données fournies. Elle suit plusieurs étapes clés pour transformer les données d'entrée en un objet automate utilisable.

i. *Extraction des données*

Nous commençons par extraire les différentes parties des données fournies.

```
def lire_automate(data):
    alphabet_data, etats_data, etats_initiaux_data, etats_finaux_data, transitions_data = data
```

ii. *Création des objets Alphabet et Etat*

Chaque symbole de l'alphabet est transformé en un objet **Alphabet** et chaque état est créé avec son type déterminé (initial, final, ou intermédiaire).

```
# Création des objets Alphabet
alphabets = [Alphabet(lettre) for lettre in alphabet_data]

# Création des objets Etat
etats = []
for idEtat in etats_data:
    etat_types = []
    if idEtat in etats_initiaux_data:
        etat_types.append("initial")
    if idEtat in etats_finaux_data:
        etat_types.append("final")
    if not etat_types:
        etat_types.append("intermediaire")
    etats.append(Etat(idEtat, f"Etat_{idEtat}", " + ".join(etat_types)))
```

iii. *Définition des états initiaux et finaux*

Nous définissons les listes des états initiaux et finaux à partir des objets état.

```
# Définir les états initiaux et finaux
etats_initiaux = [etat for etat in etats if etat.idEtat in etats_initiaux_data]
etats_finaux = [etat for etat in etats if etat.idEtat in etats_finaux_data]
```

iv. *Création des objets Transition*

Chaque transition est transformée en un objet **Transition**.

```python
# Création des objets Transition
transitions = []
for idx, (src, lettre, dst) in enumerate(transitions_data):
    etatSource = next((etat for etat in etats if etat.idEtat == src), None)
    etatDestination = next((etat for etat in etats if etat.idEtat == dst), None)
    alphabet = next((alpha for alpha in alphabets if alpha.lettre == lettre), None)
    transitions.append(Transition(idx + 1, etatSource, etatDestination, alphabet))
```

v. Création de l'objet Automate

Enfin, nous créons l'automate avec les listes des états, des alphabets, des états initiaux, des états finaux, et des transitions.

```python
# Création de l'automate
automate = Automate(etats, alphabets, etats_initiaux, etats_finaux, transitions)

return automate
```

Soit l'automate suivant à lire :

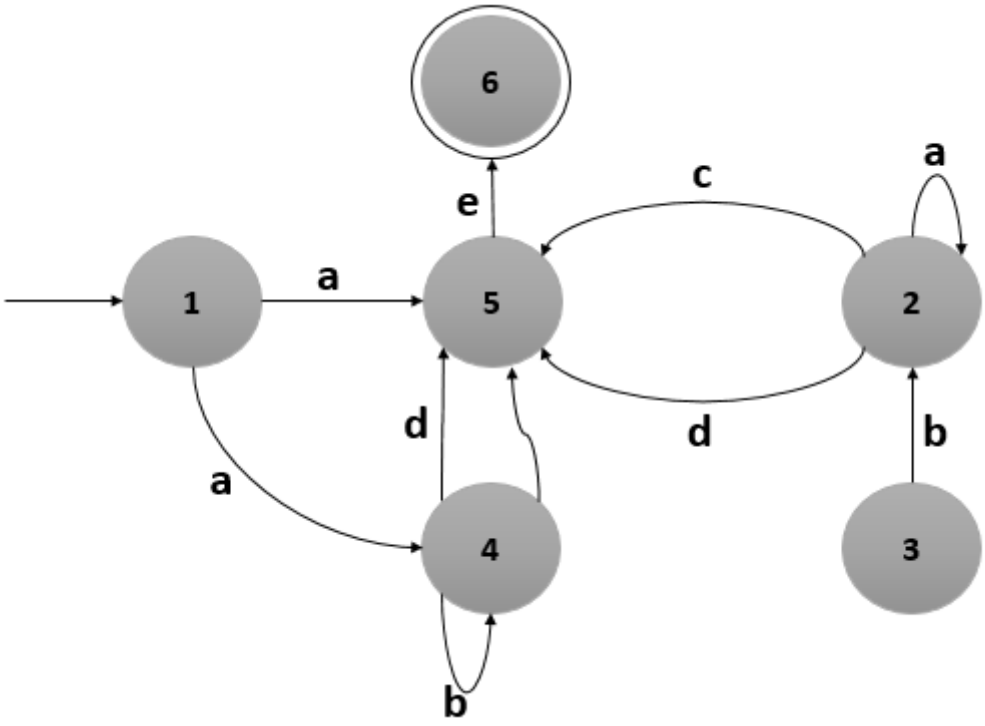


Figure 2 Exemple d'automate à lire

```python
#Fonction lire automate
# Exemple d'utilisation
data = [
    ['a', 'b', 'c', 'd', 'e'],
    [1, 2, 3, 4, 5, 6],
    [1],
    [6],
    [
        (1, 'a', 4),
        (1, 'a', 5),
        (2, 'a', 2),
        (2, 'c', 5),
        (2, 'd', 5),
        (3, 'b', 2),
        (4, 'b', 4),
        (4, 'c', 5),
        (4, 'd', 5),
        (5, 'e', 6)
    ]
]
automate = lire_automate(data)
print("Lecture de l'automate")
automate.afficherAutomate()
```

```
[Running] python -u "c:\Users\Gilles AKPINFA\Desktop\Automate_Musique\main.py"
Lecture de l'automate
Etats:
  ID: 1, Label: Etat_1, Type: initial
  ID: 2, Label: Etat_2, Type: intermediaire
  ID: 3, Label: Etat_3, Type: intermediaire
  ID: 4, Label: Etat_4, Type: intermediaire
  ID: 5, Label: Etat_5, Type: intermediaire
  ID: 6, Label: Etat_6, Type: final
Etats initiaux:
  ID: 1, Label: Etat_1, Type: initial
Etats finaux:
  ID: 6, Label: Etat_6, Type: final
Transitions:
  1 --> a --> 4
  1 --> a --> 5
  2 --> a --> 2
  2 --> c --> 5
  2 --> d --> 5
  3 --> b --> 2
  4 --> b --> 4
  4 --> c --> 5
  4 --> d --> 5
  5 --> e --> 6
```

2. Fonctionnalités principales du code

a) *Rendre un automate déterministe s'il ne l'est pas : Algorithme de déterminisation*

i. *Vérifions d'abord si l'automate est déterministe*

Pour qu'un automate fini soit déterministe :

- Chacun de ses états doit avoir au plus une transition pour chaque symbole de l'alphabet.
- Il ne doit y avoir qu'un seul état initial.

Codons d'abord une fonction qui permet de vérifier si un automate est déterministe. Pour cela, nous allons utiliser la définition précédente.

Le code python est le suivant :

```python
def est_deterministe(automate):
    """Vérifie si un automate est déterministe."""
    # Vérifier qu'il n'y a qu'un seul état initial
    if len(automate.listInitiaux) != 1:
        return False

    # Vérifier que chaque état a au plus une transition pour chaque symbole de l'alphabet
    transitions_par_etat = {}
    for transition in automate.listTransition:
        if (transition.etatSource.idEtat, transition.alphabet.lettre) in transitions_par_etat:
            return False
        transitions_par_etat[(transition.etatSource.idEtat, transition.alphabet.lettre)] = transition.etatDestination.idEtat

    return True
```

ii. *Détermination de l'automate non déterministe*

Pour déterminiser un automate, nous allons utiliser l'algorithme de déterminisation. Voici ces différentes étapes.

1. **État initial composite :** Nous commençons par créer un état initial composite qui est un ensemble des états initiaux de l'automate original.

```python
def determiniser(automate):
    # Créer un mapping pour les états composites
    etats_composites = {}
    nouvelles_transitions = []

    # Initialiser l'état initial du nouvel automate
    etats_initiaux = [etat for etat in automate.listEtat if etat.typeEtat == "initial"]
    etat_initial_composite = frozenset(etats_initiaux)
    etats_composites[etat_initial_composite] = Etat(0, f"{{{','.join(str(etat.idEtat) for etat in etat_initial_composite)}}}", "initial")
```

2. **Transitions par ensemble d'états :** Pour chaque état composite et chaque symbole de l'alphabet, nous déterminons les états suivants en utilisant les transitions de tous les états composant l'état courant.

```
# Utiliser une liste pour traiter les nouveaux états trouvés
a_traiter = [etat_initial_composite]
traites = set()

while a_traiter:
    courant = a_traiter.pop()
    if courant in traites:
        continue
    traites.add(courant)

    for alphabet in automate.listAlphabet:
        etats_suivants = set()
        for etat in courant:
            for transition in automate.listTransition:
                if transition.etatSource == etat and transition.alphabet == alphabet:
                    etats_suivants.add(transition.etatDestination)

        if not etats_suivants:
            continue
```

3. **Nouveaux états et transitions :** Les nouveaux ensembles d'états générés sont ajoutés comme des nouveaux états, et les transitions entre ces états sont enregistrées.

```
if not etats_suivants:
    continue

etat_composite_suivant = frozenset(etats_suivants)
if etat_composite_suivant not in etats_composites:
    nouvel_id = len(etats_composites)
    type_etat = "intermediaire"
    if any(etat.typeEtat == "final" for etat in etats_suivants):
        type_etat = "final"
    if any(etat.typeEtat == "initial" for etat in etats_suivants):
        type_etat += "+initial"
    label = f"{{{','.join(str(etat.idEtat) for etat in etat_composite_suivant)}}}"
    etats_composites[etat_composite_suivant] = Etat(nouvel_id, label, type_etat)
    a_traiter.append(etat_composite_suivant)

nouvelles_transitions.append(Transition(len(nouvelles_transitions) + 1, etats_composites[courant], etats_composites[etat_composite_suivant], alphabet))
```

4. **Répétition jusqu'à convergence** : Le processus continue jusqu'à ce que tous les nouveaux ensembles d'états soient traités.

```
# Créer les nouvelles listes d'états et de transitions
nouveaux_etats = list(etats_composites.values())
etat_initial = etats_composites[etat_initial_composite]
etats_finaux = [etat for etat in nouveaux_etats if "final" in etat.typeEtat]

nouvel_automate = Automate(nouveaux_etats, automate.listAlphabet, [etat_initial], etats_finaux, nouvelles_transitions)

return nouvel_automate
```

Exemple de fonctionnement

Soit l’automate de la figure… à déterminer :

```python
# Vérification si l'automate est déterministe
if est_deterministe(automate):
    print("L'automate est deterministe.")
else:
    print("L'automate n'est pas deterministe.")

# Déterminiser l'automate
automate_deterministe = determiniser(automate)
print("Automate determinise:")
automate_deterministe.afficherAutomate()
```

Résultat obtenu est le suivant :

```
[Running] python -u "c:\Users\Gilles AKPINFA\Desktop\Automate_Musique\main.py"
Automate determinise:
Etats:
  ID: 0, Label: {1}, Type: initial
  ID: 1, Label: {5,4}, Type: intermediaire
  ID: 2, Label: {4}, Type: intermediaire
  ID: 3, Label: {5}, Type: intermediaire
  ID: 4, Label: {6}, Type: final
Etats initiaux:
  ID: 0, Label: {1}, Type: initial
Etats finaux:
  ID: 4, Label: {6}, Type: final
Transitions:
  0 --> a --> 1
  1 --> b --> 2
  1 --> c --> 3
  1 --> d --> 3
  1 --> e --> 4
  3 --> e --> 4
  2 --> b --> 2
  2 --> c --> 3
  2 --> d --> 3
```

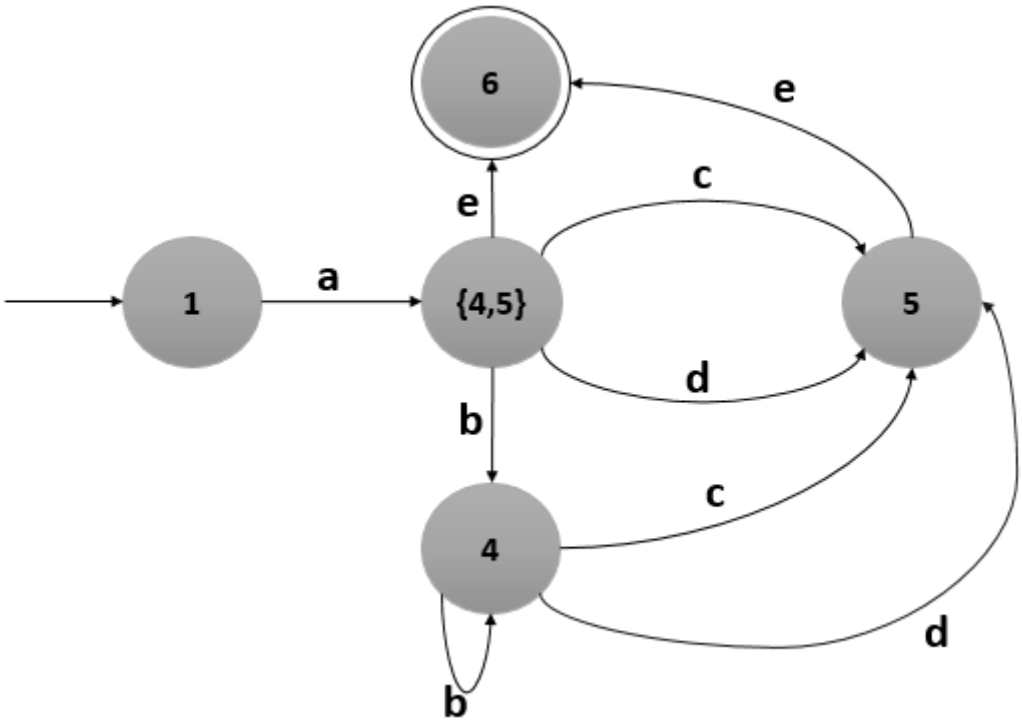


Figure 3 Résultat obtenu après la déterminisation de l'automate de la figure 1

b) Rendre un automate complet s'il ne l'est pas : Algorithme de complétion

Un automate fini est dit complet si, pour chaque état et chaque symbole de l'alphabet, il existe une transition sortante.

Un automate non complet peut être transformé en automate complet en ajoutant des transitions manquantes vers un état puits. Dans cette section, nous décrivons l'algorithme de complétion et fournissons les fonctions nécessaires pour vérifier la complétion et compléter un automate si nécessaire.

i. Vérification de la Complétion

Pour chaque état et chaque symbole de l'alphabet, nous vérifions s'il existe au moins une transition sortante correspondante. Si une transition est manquante, l'automate n'est pas complet.

Code pour la Vérification de la Complétion

```python
def estComplet(automate):
    """Vérifie si l'automate est complet."""
    alphabet = automate.listAlphabet
    etats = automate.listEtat
    transitions = automate.listTransition

    for etat in etats:
        for lettre in alphabet:
            if not any(t.etatSource == etat and t.alphabet == lettre for t in transitions):
                return False
    return True
```

ii. Algorithme et code de complétion

1) **Vérifier la complétion** : Nous commençons par vérifier si l'automate est déjà complet. Si c'est le cas, nous n'effectuons aucune modification et retournons l'automate tel quel.

```python
def completerAutomate(automate):
    """Rend un automate complet s'il ne l'est pas déjà."""
    if estComplet(automate):
        print("L'automate est déjà complet.")
        return automate
```

2) **Créer un état "puits"** : Nous créons un nouvel état "puits" et des transitions de cet état vers lui-même pour chaque symbole de l'alphabet.

```
alphabet = automate.listAlphabet
etats = automate.listEtat
transitions = automate.listTransition

# Créer un état "puits"
id_puits = max(etat.idEtat for etat in etats) + 1
etat_puits = Etat(id_puits, "Puits", "intermediaire")
transitions_puits = []

for lettre in alphabet:
    transitions_puits.append(Transition(len(transitions) + 1, etat_puits, etat_puits, lettre))
```

3) **Ajouter les transitions manquantes** : Pour chaque état de l'automate, nous ajoutons une transition vers l'état "puits" pour chaque symbole de l'alphabet manquant.

```
for lettre in alphabet:
    transitions_puits.append(Transition(len(transitions) + 1, etat_puits, etat_puits, lettre))

# Ajouter les transitions manquantes pour chaque état
for etat in etats:
    for lettre in alphabet:
        if not any(t.etatSource == etat and t.alphabet == lettre for t in transitions):
            transitions.append(Transition(len(transitions) + 1, etat, etat_puits, lettre))
```

4) **Ajouter les transitions de l'état "puits"** : Nous ajoutons les transitions de l'état "puits" à lui-même pour chaque symbole de l'alphabet.

```
# Ajouter les transitions de l'état "puits"
transitions.extend(transitions_puits)

# Ajouter l'état "puits" à la liste des états
etats.append(etat_puits)
```

5) **Mettre à jour l'automate** : Nous ajoutons l'état "puits" à la liste des états et mettons à jour les listes d'états et de transitions de l'automate.

```
# Mise à jour des listes d'états et de transitions de l'automate
automate.listEtat = etats
automate.listTransition = transitions

return automate
```

Exemple d’utilisation

```
Automate determinise:
Etats:
  ID: 0, Label: {1}, Type: initial
  ID: 1, Label: {5,4}, Type: intermediaire
  ID: 2, Label: {4}, Type: intermediaire
  ID: 3, Label: {5}, Type: intermediaire
  ID: 4, Label: {6}, Type: final
Etats initiaux:
  ID: 0, Label: {1}, Type: initial
Etats finaux:
  ID: 4, Label: {6}, Type: final
Transitions:
  0 --> a --> 1
  1 --> b --> 2
  1 --> c --> 3
  1 --> d --> 3
  1 --> e --> 4
  3 --> e --> 4
  2 --> b --> 2
  2 --> c --> 3
  2 --> d --> 3
```

```
Automate apres completion:
Etats:
  ID: 0, Label: {1}, Type: initial
  ID: 1, Label: {5,4}, Type: intermediaire
  ID: 2, Label: {4}, Type: intermediaire
  ID: 3, Label: {5}, Type: intermediaire
  ID: 4, Label: {6}, Type: final
  ID: 5, Label: Puits, Type: intermediaire
Etats initiaux:
  ID: 0, Label: {1}, Type: initial
Etats finaux:
  ID: 4, Label: {6}, Type: final
Transitions:
  0 --> a --> 1
  1 --> b --> 2
  1 --> c --> 3
  1 --> d --> 3
  1 --> e --> 4
  3 --> e --> 4
  2 --> b --> 2
  2 --> c --> 3
  2 --> d --> 3
  0 --> b --> 5
  0 --> c --> 5
  0 --> d --> 5
  0 --> e --> 5
  1 --> a --> 5
  2 --> a --> 5
  2 --> e --> 5
  3 --> a --> 5
  3 --> b --> 5
  3 --> c --> 5
  3 --> d --> 5
  4 --> a --> 5
  4 --> b --> 5
  4 --> c --> 5
  4 --> d --> 5
  4 --> e --> 5
  5 --> a --> 5
  5 --> b --> 5
  5 --> c --> 5
  5 --> d --> 5
  5 --> e --> 5
PS C:\Users\Gilles AKPINFA\Desktop\Automa
```

c) *Minimiser un automate qui ne l'est pas*

Algorithme et code

Détermination des états accessibles

Ce bloc détermine les états accessibles à partir de l'état initial. Les états inaccessibles sont éliminés, ce qui simplifie l'automate en supprimant les états et transitions inutiles.

```python
# Fonction pour minimiser l'automate
def minimiser_automate(automate):
    # Étape 1: Déterminer les états accessibles
    def etats_accessibles(etat_initial, transitions):
        accessibles = set()
        a_traiter = [etat_initial]
        while a_traiter:
            etat = a_traiter.pop()
            if etat not in accessibles:
                accessibles.add(etat)
                for transition in transitions:
                    if transition.etatSource == etat and transition.etatDestination not in accessibles:
                        a_traiter.append(transition.etatDestination)
        return accessibles

    accessibles = etats_accessibles(automate.listInitiaux[0], automate.listTransition)
    etats = [etat for etat in automate.listEtat if etat in accessibles]
    transitions = [t for t in automate.listTransition if t.etatSource in accessibles and t.etatDestination in accessibles]
    etats_finaux = [etat for etat in automate.listFinaux if etat in accessibles]
```

Détermination des états équivalents

- Les états sont partitionnés en groupes d'états équivalents, où deux états sont considérés équivalents s'ils ne peuvent pas être distingués par aucune séquence d'entrées.
- L'algorithme affine progressivement les partitions jusqu'à ce qu'elles deviennent stables (aucun changement supplémentaire).

```python
# Étape 2: Déterminer les états équivalents
def etats_equivalents(etats, etats_finaux, alphabet, transitions):
    non_finaux = [etat for etat in etats if etat not in etats_finaux]
    partitions = [set(etats_finaux), set(non_finaux)]

    def trouver_partitions(etat, partitions):
        for i, part in enumerate(partitions):
            if etat in part:
                return i
        return -1
```

```
    stable = False
    while not stable:
        stable = True
        nouvelle_partition = []
        for part in partitions:
            sous_partitions = {}
            for etat in part:
                signature = []
                for lettre in alphabet:
                    dest = next((t.etatDestination for t in transitions if t.etatSource == etat and t.alphabet == lettre), None)
                    signature.append(trouver_partitions(dest, partitions))
                signature = tuple(signature)
                if signature not in sous_partitions:
                    sous_partitions[signature] = set()
                sous_partitions[signature].add(etat)
            if len(sous_partitions) > 1:
                stable = False
                for sous_part in sous_partitions.values():
                    nouvelle_partition.append(sous_part)
            else:
                nouvelle_partition.append(part)
        partitions = nouvelle_partition
    return partitions

partitions = etats_equivalents(etats, etats_finaux, automate.listAlphabet, transitions)
etat_partition = {etat: i for i, part in enumerate(partitions) for etat in part}
```

Construction de l’automate minimal

- Les partitions d'états équivalents sont utilisées pour créer un nouvel automate avec un nombre minimal d'états.
- Les nouvelles transitions sont définies en fonction des partitions.

```
# Étape 3: Construire l'automate minimal
etats_minimaux = [Etat(i, f"Etat_{i}", "intermediaire") for i in range(len(partitions))]
etat_initial_minimal = etats_minimaux[etat_partition[automate.listInitiaux[0]]]
etats_finaux_minimaux = [etats_minimaux[i] for i, part in enumerate(partitions) if any(etat in etats_finaux for etat in part)]
transitions_minimales = []

for transition in transitions:
    nouvelle_src = etats_minimaux[etat_partition[transition.etatSource]]
    nouvelle_dst = etats_minimaux[etat_partition[transition.etatDestination]]
    nouvelles_transitions = [t for t in transitions_minimales if t.etatSource == nouvelle_src and t.alphabet == transition.alphabet]
    if not nouvelles_transitions:
        transitions_minimales.append(Transition(len(transitions_minimales) + 1, nouvelle_src, nouvelle_dst, transition.alphabet))

nouvel_automate = Automate(
    etats_minimaux,
    automate.listAlphabet,
    [etat_initial_minimal],
    etats_finaux_minimaux,
    transitions_minimales
)

return nouvel_automate
```

Exemple d’utilisation

```
Lecture de l'automate
Etats:
  ID: 1, Label: Etat_1, Type: initial
  ID: 2, Label: Etat_2, Type: intermediaire
  ID: 3, Label: Etat_3, Type: intermediaire
  ID: 4, Label: Etat_4, Type: intermediaire
  ID: 5, Label: Etat_5, Type: intermediaire
  ID: 6, Label: Etat_6, Type: final
Etats initiaux:
  ID: 1, Label: Etat_1, Type: initial
Etats finaux:
  ID: 6, Label: Etat_6, Type: final
Transitions:
  1 --> a --> 4
  1 --> a --> 5
  2 --> a --> 2
  2 --> c --> 5
  2 --> d --> 5
  3 --> b --> 2
  4 --> b --> 4
  4 --> c --> 5
  4 --> d --> 5
  5 --> e --> 6
```

```
Automate apres minimisation:
Etats:
  ID: 0, Label: Etat_0, Type: intermediaire
  ID: 1, Label: Etat_1, Type: intermediaire
  ID: 2, Label: Etat_2, Type: intermediaire
  ID: 3, Label: Etat_3, Type: intermediaire
Etats initiaux:
  ID: 2, Label: Etat_2, Type: intermediaire
Etats finaux:
  ID: 0, Label: Etat_0, Type: intermediaire
Transitions:
  2 --> a --> 1
  1 --> b --> 1
  1 --> c --> 3
  1 --> d --> 3
  3 --> e --> 0
```

Tableau 1 Résultat obtenu après minimisation de l'automate

d) Affichage du graphe de transition de l'automate sous forme graphique

Fonctions > dessin_aut.py > afficher_graphe_transition

```python
#Importation de la bibliothèque graphviz
from graphviz import Digraph

#Définition de la fonction afficher_graphe_transition :
#La fonction afficher_graphe_transition prend en paramètre un objet automate et initialise un objet Digraph nommé dot.
def afficher_graphe_transition(automate):
    dot = Digraph()

    # Ajouter les noeuds
    for etat in automate.listEtat:
        shape = 'doublecircle' if etat in automate.listFinaux else 'circle'
        # Ajouter une flèche de début à l'état initial
        if etat in automate.listInitiaux:
            dot.node(str(etat.idEtat), shape=shape, style='filled', fillcolor='yellow')
        else:
            dot.node(str(etat.idEtat), shape=shape)

    # Ajouter les transitions
    for transition in automate.listTransition:
        dot.edge(str(transition.etatSource.idEtat), str(transition.etatDestination.idEtat), label=transition.alphabet.lettre)

    # Afficher le graphe
    dot.render('automate', format='png', view=True)
```

Exemple d'affichage

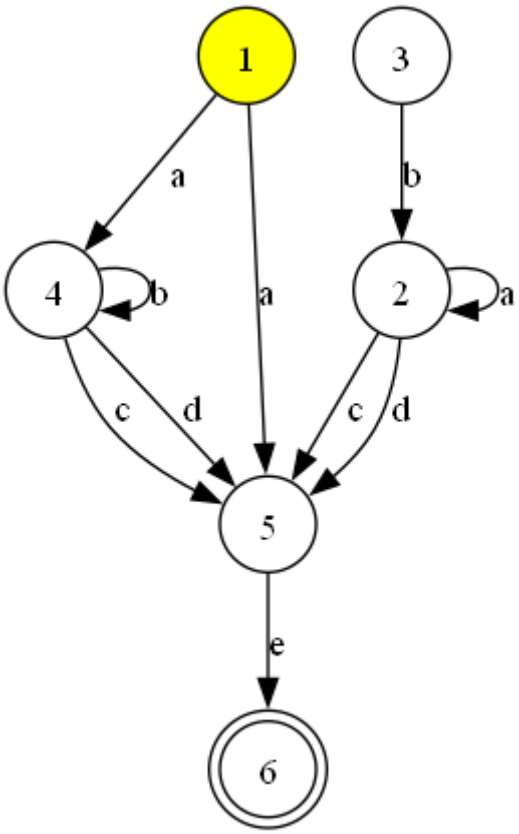


Figure 4 : Affichage graphique d'un automate

II. Phase applicative

1. Idée

Pour cette phase, mon idée est de générer de la musique automatiquement en utilisant les concepts des automates pour créer de la musique à partir du texte fourni par l'utilisateur. Le système convertit chaque caractère du texte en notes musicales. Voici les étapes de réalisation et les résultats obtenus durant la phase application.

2. Étapes de Réalisation

a) Définition des Règles Harmoniques et Rythmiques

Les caractères du texte sont mappés à des notes. Cette étape implique la création d'un dictionnaire (**char_to_music**) qui associe chaque caractère à une note et à une durée.

Fonctions > char.py > ...

```python
char_to_music = {
    'a': ('do', 1),
    'b': ('re', 1),
    'c': ('mi', 1),
    'd': ('fa', 1),
    'e': ('sol', 1),
    'f': ('la', 1),
    'g': ('si', 1),
    'h': ('do', 1),
    'i': ('re', 1),
    'j': ('mi', 1),
    'k': ('fa', 1),
    'l': ('sol', 1),
    'm': ('la', 1),
    'n': ('si', 1),
    'o': ('do', 2),
    'p': ('re', 2),
    'q': ('mi', 2),
    'r': ('fa', 2),
    's': ('sol', 2),
    't': ('la', 2),
    'u': ('si', 2),
    'v': ('do', 2),
    'w': ('re', 2),
    'x': ('mi', 2),
    'y': ('fa', 2),
    'z': ('sol', 2),
}
```

b) *Classe Automate pour la Génération de Musique*

La nouvelle version de la classe **Automate** inclut des fonctionnalités supplémentaires pour la génération de séquences musicales à partir du texte, ainsi que pour la gestion des états et des transitions dans ce contexte spécifique. Voici les ajouts et modifications effectués :

Attributs et Initialisation

- Ajout de l'attribut **etat_courant** pour suivre l'état actuel de l'automate.

Méthodes

1. **transition** : Cette méthode effectue une transition d'état en fonction d'un signal (ici, une note musicale).
2. **etat_actuel** : Retourne l'état actuel de l'automate.
3. **generer_sequence** : Génère une séquence musicale en fonction du texte fourni.

Voici ce qui a changé dans le code de base :

```python
# Class Automate
from Fonctions.char import char_to_music

class Automate:
    def __init__(self, listEtat, listAlphabet, listInitiaux, listFinaux, listTransition):
        self._listEtat = listEtat
        self._listAlphabet = listAlphabet
        self._listInitiaux = listInitiaux
        self._listFinaux = listFinaux
        self._listTransition = listTransition
        self._etat_courant = listInitiaux[0] if listInitiaux else None

```

```python
def generer_sequence(self, texte):
    sequence = []
    for char in texte:
        if char in char_to_music:
            accord, rythme = char_to_music[char]
            etat_actuel = self._etat_courant
            self.transition(accord)  # Utilise l'accord comme signal de transition
            sequence.append((etat_actuel, accord, rythme))
    return sequence

def etat_actuel(self):
    return self._etat_courant

def transition(self, lettre):
    for t in self._listTransition:
        if t.etatSource == self._etat_courant and t.alphabet.lettre == lettre:
            self._etat_courant = t.etatDestination
            break
```

c) *Génération de Fichiers MIDI*

Une fonction **generer_midi** a été ajoutée pour convertir la séquence musicale en un fichier MIDI.

Fonctions > musique.py > ...

```python
from midiutil import MIDIFile
from tkinter import messagebox
import pygame
import time

def generer_midi(sequence, fichier_sortie):
    # Création d'un objet MIDIFile
    midi = MIDIFile(1)  # Un seul piste
    piste = 0
    temps = 0  # En temps de battements
    canal = 0
    volume = 100

    # Ajouter une piste de musique
    midi.addTrackName(piste, temps, "Automate Track")
    midi.addTempo(piste, temps, 120)

    # Ajouter les notes de la séquence à la piste
    note_map = {
        'do': 60,  # Note C4
        're': 62,  # Note D4
        'mi': 64,  # Note E4
        'fa': 65,  # Note F4
        'sol': 67, # Note G4
        'la': 69,  # Note A4
        'si': 71   # Note B4
    }

```

```python
    for etat, note, rythme in sequence:
        note_number = note_map[note]
        duree = rythme
        midi.addNote(piste, canal, note_number, temps, duree, volume)
        temps += duree

    # Écrire le fichier MIDI sur le disque
    with open(fichier_sortie, "wb") as sortie:
        midi.writeFile(sortie)
```

d) *Création de l'interface utilisateur*

Une interface utilisateur (UI) a été créée en utilisant Tkinter. L'interface permet aux utilisateurs d'entrer du texte, de générer de la musique et de lire la musique directement.

```python
import tkinter as tk
from tkinter import messagebox
from midiutil import MIDIFile
import pygame
import time
```

```python
def generer_musique():
    texte = entree_texte.get()
    if not texte:
        messagebox.showwarning("Avertissement", "Veuillez entrer du texte.")
        return

    sequence = automate.generer_sequence(texte)
    if not sequence:
        messagebox.showerror("Erreur", "Aucun motif musical généré.")
        return

    nom_fichier = "musique_generee.mid"
    generer_midi(sequence, nom_fichier)
    messagebox.showinfo("Succès", f"Fichier MIDI généré: {nom_fichier}")

def jouer_musique():
    nom_fichier = "musique_generee.mid"
    try:
        pygame.mixer.init()
        pygame.mixer.music.load(nom_fichier)
        pygame.mixer.music.play()
        while pygame.mixer.music.get_busy():
            time.sleep(1)
    except Exception as e:
        messagebox.showerror("Erreur", f"Impossible de lire le fichier MIDI: {str(e)}")
```

```python
# Interface utilisateur avec tkinter
fenetre = tk.Tk()
fenetre.title("Générateur de Musique Automatique")

fenetre.geometry("400x300")  # Définir la taille de la fenêtre

# Ajouter des cadres pour organiser les éléments
cadre_instruction = tk.Frame(fenetre, padx=10, pady=10)
cadre_instruction.pack(pady=(20, 10))

cadre_texte = tk.Frame(fenetre, padx=10, pady=10)
cadre_texte.pack(pady=(0, 20))

cadre_boutons = tk.Frame(fenetre, padx=10, pady=10)
cadre_boutons.pack(pady=(0, 10))

# Label d'instruction
label_instruction = tk.Label(cadre_instruction, text="Entrez du texte pour générer de la musique:", font=("Helvetica", 12))
label_instruction.pack()

# Entrée de texte
entree_texte = tk.Entry(cadre_texte, width=50, font=("Helvetica", 12))
entree_texte.pack()

# Boutons de génération et de lecture
bouton_generer = tk.Button(cadre_boutons, text="Générer Musique", command=generer_musique, bg="#4CAF50", fg="white", font=("Helvetica", 12), width=15)
bouton_generer.grid(row=0, column=0, padx=10, pady=5)

bouton_jouer = tk.Button(cadre_boutons, text="Jouer Musique", command=jouer_musique, bg="#008CBA", fg="white", font=("Helvetica", 12), width=15)
bouton_jouer.grid(row=0, column=1, padx=10, pady=5)

fenetre.mainloop()
```

e) *Résultats Obtenus*

1. **Génération de Séquences Musicales** : L'automate génère des séquences musicales en fonction du texte fourni par l'utilisateur. Chaque caractère est converti en une note et une durée.
2. **Création de Fichiers MIDI** : Les séquences générées sont converties en fichiers MIDI, ce qui permet de sauvegarder et de partager la musique.
3. **Lecture de Musique** : L'interface utilisateur permet de lire les fichiers MIDI générés directement, offrant une expérience utilisateur interactive et instantanée.

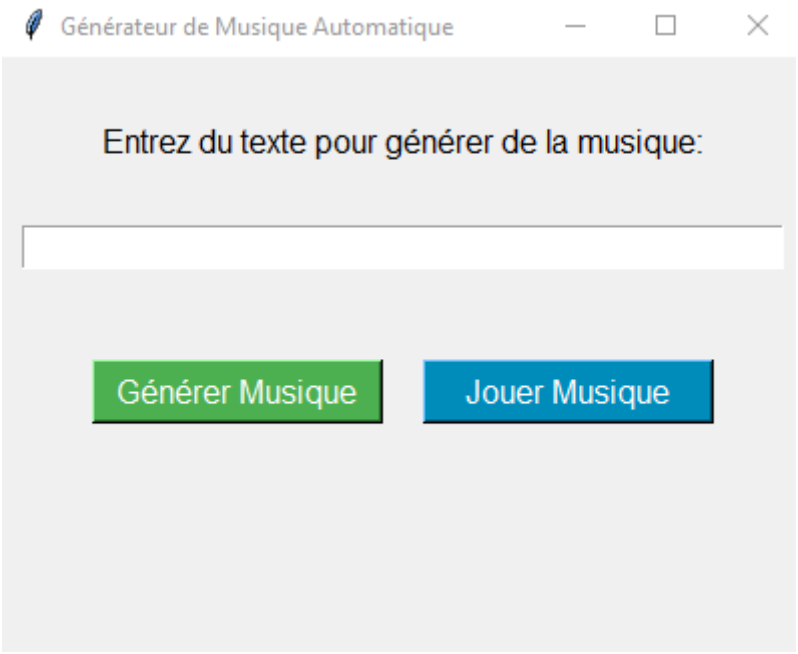


Figure 5 : Interface Utilisateur

III. Conclusion et perspectives

1. Bilan du projet

Le projet de génération de musique algorithmique via un automate a démontré comment les concepts théoriques des automates finis peuvent être appliqués à la création musicale. J'ai réussi à :

- Modéliser des motifs musicaux en utilisant une approche orientée objet.
- Développer des règles harmoniques et rythmiques pour guider la génération musicale.

2. Limites Rencontrées

Malgré les succès, plusieurs limites ont été rencontrées :

- **Complexité des Règles Musicales** : La création de règles harmoniques et rythmiques précises est complexe et nécessite une expertise musicale approfondie.
- **Qualité Artistique** : Bien que l'automate puisse générer des séquences musicales, la qualité artistique et l'émotion véhiculée par la musique générée sont encore loin de rivaliser avec celles composées par des humains.
- **Déclencheurs Internes et Externes** : Des déclencheurs sophistiqués pour gérer les transitions et variations musicales en réponse à des conditions internes et des événements externes n'ont pas été implémentés dans cette version.

3. Perspectives et Améliorations Futures

Pour améliorer et étendre ce projet, plusieurs pistes peuvent être envisagées :

- **Affinement des Règles Musicales** : Collaborer avec des musiciens professionnels pour affiner et enrichir les règles harmoniques et rythmiques, augmentant ainsi la qualité musicale des séquences générées.
- **Déclencheurs Internes et Externes** : Implémenter des déclencheurs internes et externes pour permettre une gestion plus dynamique et interactive des transitions musicales.
- **Expérience Utilisateur** : Développer une interface utilisateur conviviale permettant aux utilisateurs de créer, modifier et écouter des séquences musicales générées de manière intuitive.
- **Incorporation de l'Apprentissage Machine** : Utiliser des techniques d'apprentissage machine pour apprendre des modèles musicaux à partir de grandes bases de données de musique existante, ce qui pourrait améliorer la diversité et la qualité des séquences générées.

IV. Références Bibliographiques et Webographies

1. Références Bibliographiques

- *Informatique Théorique et Applications*, Marc Chemillier
- *Monoïde libre et musique, première partie : les musiciens ont-ils besoin des mathématiques ?***,** Marc Chemillier
- *Rapport du projet d'Informatique Théorique II***,** Jérémy Turon & Salomé Coavoux
- *Learning Python,* par Mark Lutz

2. Webographies

- **Music FX - Google AI Test Kitchen,** Music FX
- Python Documentation
- Graphviz Documentation
- Midiutil Documentation