POLITECNICO DI MILANO

Scuola di Ingegneria dell'Informazione

POLO TERRITORIALE DI COMO

Corso di Laurea Specialistica in Ingegneria Informatica

Un'applicazione cloud-oriented per l'analisi armonica e improvvisativa nel Jazz

Relatore:
Prof. Augusto Sarti

Tesi di Laurea di:
Fausto Savatteri
Matr. 755120

Anno Accademico 2012-2013

A mia madre

SOMMARIO

Il Jazz è indubbiamente un genere tanto affascinante quanto complesso. In un contesto moderno in cui lo sviluppo tecnologico offre servizi e piattaforme con funzionalità sempre più avanzate, anche la musica sta rapidamente attuando la propria rivoluzione. Nel settore di ricerca specifico della *computer music* numerosi sono gli studi di interesse relativi ai sistemi di analisi musicale e le applicazioni interattive. Oggi vi è una crescita esponenziale di applicativi per smartphone e tablet. L'obbiettivo di questa tesi è quello di creare un applicativo per l'automazione dell'analisi armonica e l'improvvisazione melodica in ambito jazzistico (Jazz Chord Progressions). Il musicista Jazz, in particolar modo, sviluppa progressivamente un *dizionario* e utilizza un modello musicologico basato su regole e vincoli. Bisogna dunque affrontare l'analisi dell'improvvisazione jazz con un modello generativo simile a quello di un musicista reale. Prendendo spunto dal software *open-source ImproVisor* (sviluppato da *Keller*, *Harvey Mudd College*, dal 2005 ad oggi), si utilizza una grammatica probabilistica per l'analisi armonica e la generazione melodica (algoritmo di *Clustering* e catene di *Markov*) in modo tale da definire le caratteristiche e le regole implicite di un linguaggio musicale e di uno stile specifico. Ciò che si vuole fornire al fruitore finale è uno strumento valido dal punto di vista didattico ma al contempo interattivo e di facile utilizzo. A tale scopo la scelta ricade sull'implementazione di un tool per Android. Un'applicazione *Cloud Oriented* che possa contenere al suo interno tutte le funzionalità in modo semplice e con un occhio di riguardo agli ostacoli e ai limiti (dimensionali e performanti) dei dispositivi mobili. Il vantaggio principale nel creare un'applicazione Android è sicuramente la *portabilità*: un'applicazione mobile avrà maggiore possibilità di essere usata dagli utenti, perché, potenzialmente, potrà essere utilizzata in qualsiasi luogo e in qualsiasi momento. Un altro vantaggio è quello di offrire all'utente un'esperienza diversa e forse anche più completa rispetto a quella, generalmente soltanto informativa, di un sito web o di un libro didattico. Molteplici dunque i casi d'uso che fanno sì che chiunque (professionista e non) possa trarre beneficio dall'analisi o dall'improvvisazione musicale generata dal tool per imparare o per condividere delle informazioni.

Indice

Capitolo 1 - Introduzione

Il Jazz è indubbiamente un genere tanto affascinante quanto complesso. In un contesto moderno in cui lo sviluppo tecnologico offre servizi e piattaforme con funzionalità sempre più avanzate, anche la musica, in maniera vertiginosa, sta attuando la propria rivoluzione. Nel settore specifico di ricerca della *computer music* numerosi sono gli studi di interesse relativi ai sistemi di analisi musicale. Numerosissime soprattutto le applicazioni interattive. Oggi ad esempio vi è una crescita esponenziale di applicativi per smartphone e tablet che vengono implementati sia per uso professionale (controller audio, sequencer, strumenti di misurazione, processing audio, etc.) che per uso ricreativo (controller Midi, videogiochi, piano e strumenti virtuali per la musica elettronica). L'obbiettivo prefissato di questa tesi è quello di creare un applicativo per l'automazione dell'analisi armonica e l'improvvisazione melodica in ambito jazzistico (Jazz Chord Progressions). Il musicista Jazz, in particolar modo, sviluppa progressivamente un *dizionario* e utilizza un modello musicologico basato su regole e vincoli. Bisogna dunque affrontare l'analisi dell'improvvisazione jazz con un modello generativo simile a quello di un musicista reale. Prendendo spunto dal lavoro di Keller[1][5], si utilizzano e modellano i tools di RoadMap ed ImproVisor. La creazione di una grammatica probabilistica e l'utilizzo di alcuni algoritmi per l'analisi e la generazione melodica (*Parsing[18]* e *Clustering [2][31]* fra tutti) aiutano a definire le caratteristiche e le regole implicite di un linguaggio musicale e di uno stile specifico. Ciò che si vuole fornire al fruitore finale è uno strumento valido dal punto di vista didattico ma al contempo interattivo e di facile utilizzo. A tale scopo la scelta ricade sull'implementazione di un tool per *Android [32]*. Un'applicazione *Cloud Oriented* che possa contenere al suo interno tutte le funzionalità prefissate in modo semplice e con un occhio di riguardo agli ostacoli e ai limiti (dimensionali e performanti) dei dispositivi mobili.
Il vantaggio principale di creare un'applicazione *Android* è sicuramente la

portabilità: un'applicazione per smartphone e tablet avrà maggiore possibilità di essere usata dagli utenti, perché, potenzialmente, potrà essere utilizzata in qualsiasi luogo e in qualsiasi momento.
Un altro vantaggio è quello di offrire all'utente un'esperienza diversa e forse anche più completa rispetto a quella, generalmente soltanto informativa, di un sito web o di un libro didattico. Molteplici dunque i casi d'uso che fanno sì che chiunque (professionista e non) possa trarre beneficio dall'analisi e dall'improvvisazione musicale generata dal tool per imparare e condividere informazioni. Molteplici però saranno anche gli ostacoli a cui bisogna far fronte. Per fare improvvisare una macchina infatti, dobbiamo innanzitutto capire come lo faccia nella pratica un musicista jazz. Un'operazione non del tutto scontata. La musica Jazz è il genere musicale più strutturato che esista, proprio perché pieno di vincoli e regole. Il musicista professionista dunque trae beneficio da queste regole e riesce talvolta anche a stravolgerle componendo istantaneamente delle melodie armoniche, di senso compiuto, istantanee e melodiche, senza mai risultare banali né tantomeno ripetitive. Alla base di tutto ciò vi è ovviamente una pratica e una costanza considerevole, ogni musicista jazz studia dapprima le regole basilari della teoria musicale, le regole armoniche, le scale modali e le diverse posizioni dovute alla trasposizione nelle 12 tonalità. Mettere in pratica quanto appreso dalla teoria significa spesso fare riferimento ai così detti "*standards*" del genere (contenuti nei *Real Books*) da cui si estrapolano informazioni essenziali per l'arricchimento di un dizionario personale fatto di strutture armoniche e di *patterns (o licks)* generativi che possano andar bene per una data progressione armonica. Quest'ultima è una successione di accordi la cui struttura è determinata dal tipo di accordi che la contengono. Ogni accordo infatti ha una funzione diversa dipendendo dal contesto in cui ci si trova. L'improvvisazione è spesso definita come composizione istantanea, il musicista riesce ad avere pieno controllo e sincronia tra ciò che pensa e ciò che esegue. Un'idea musicale viene dunque composta e suonata allo stesso tempo. La macchina dovrà carpire le regole, generare delle melodie armoniche e possibilmente emulare ciò che realmente un musicista jazz fa

durante un'esecuzione. E' necessario dunque un background, un apprendimento basilare fatto di scale, note e regole che prende forma all'interno di una grammatica probabilistica. Una grammatica di simboli dapprima astratta, non deterministica, in cui il peso associato a ciascun simbolo porta alla generazione di un albero probabilistico fatto di simboli terminali e simboli non terminali. Il raggiungimento di un nodo terminale significa la generazione di note e melodie astratte simili tra loro e appartenenti ad un certo stile.

I Musicisti interessati all'improvvisazione Jazz traggono beneficio dall'analisi armonica (Chords Progression) considerando che su di essa si basa gran parte dello studio basilare per capire ed interiorizzare realmente la struttura portante del brano, la tonalità (o le tonalità) di impianto e, non ultimo, trovare analogie con altri brani precedentemente analizzati. Il primo approccio è quello di suddividere la struttura armonica di un brano soffermandosi sul ruolo di ciascun accordo e sulle scale appartenenti ad esso, in modo tale da avere una visione più dettagliata e da poter prevedere quali note possano essere incluse nella generazione di una eventuale linea improvvisativa. Nel *Capitolo 2* si descriverà dunque l'analisi armonica all'interno del tool di Roadmap (Keller [1] [5]) che automatizza il processo di analisi per una data progressione di accordi. A tal proposito vi è un approccio per rappresentare degli idiomi conosciuto come "*bricks*". Tale nomenclatura deriva dall'approccio dei "Lego Bricks" di Conrad Cork [8][9] ed esteso successivamente da John Elliott [9]. Si definirà infine una grammatica semi-automatica per analizzare le sequenze di accordi. I più noti standard Jazz spesso si sviluppano su più tonalità ed i *chorus* si ripetono ciclicamente rispettando alcune regole e strutture note (es.Blues: 12 battute) o più genericamente suddividendo il *chorus* con una struttura del tipo "AABA" (A tipicamente è la Strofa, B il ritornello o una parte che presenta una modulazione armonica). Provvedere ad *un'automazione* può sicuramente aiutare il musicista ad imparare, capire ed interiorizzare intere Jazz Chord Progressions (JCP) piuttosto che accordi specifici o coppie di accordi isolati tra loro. Altrettanto importante sarà dare un suggerimento idiomatico mirato

ad identificare e apprezzare quelle che sono le progressioni più comuni, le cadenze risolutive del brano, così come i più tipici *turnaround [10]*. Ci si serve di un algoritmo modificato di Parsing: *Cocke-Younger-Kasami (CYK) parsing algorithm [*, seguito da uno step per minimizzare il costo associato al cammino, entrambi basati su algoritmi di dynamic programming.
Nel *Capitolo 3* invece verrano descritti gli strumenti per la generazione di melodie improvvisative. Le Grammatiche sono alla base delle tecniche di generazione melodica. Al fine di catturare le "gestures" idiomatiche di un solista specifico si estende una rappresentazione grammaticale (precedentemente introdotta da Keller e Morrison nel 2007) con una tecnica per rappresentare un contorno melodico. I contorni più rappresentativi (ovvero le melodie più coerenti) sono estratti da un corpus utilizzando il *clustering* e il sequenziamento tra contorni avviene utilizzando *catene di Markov* che sono codificate nella grammatica per aumentare la coesione e il legame tra le melodie.
Il *Capitolo 4* descrive le tecniche e le metodologie attuate per effettuare un *Porting* degli algoritmi precedentemente citati sul sistema di sviluppo *Android*. Verranno apportate modifiche alle classi al fine di estrapolare le informazioni necessarie ed automatizzare il processo. Verrà inoltre creata una struttura Client-Server socket per gestire lo scambio dei dati tra il Server (contenente gli algoritmi di RoadMap e di improvvisazione) e il Client (ovvero l'applicativo Android). Il *Capitolo 5* descrive l'implementazione *Android* per la creazione dell'applicazione (l'architettura, il supporto MIDI per il playalong di leadsheet musicali, la trasposizione, la visualizzazione in notazione musicali, le tabelle per l'interfaccia GUI). Infine, nel Capitolo 6, verranno tratte le conclusioni e descritti gli sviluppi futuri.

Capitolo 2 - RoadMap Armonica

RoadMap: Automatizzare l'Analisi Armonica di un brano (Jazz Chord Progressions)

Questo capitolo descrive i metodi, le tecniche e gli algoritmi utilizzati nella progettazione e implementazione del tool di *RoadMap [1][5]*. I Musicisti interessati all'improvvisazione Jazz traggono beneficio dall'analisi armonica (Chord Progressions) considerando che su di essa si basa gran parte dello studio basilare per capire ed interiorizzare realmente la struttura portante del brano, la tonalità (o le tonalità) di impianto e, non ultimo, trovare analogie con altri brani precedentemente analizzati. Si descriverà dunque l'analisi automatizzata all'interno del tool *RoadMap*. A tal proposito vi è un approccio per rappresentare idiomi conosciuto come "bricks".

Tale nomenclatura deriva dall'approccio dei "Lego Bricks" di Conrad Cork [8][9] ed esteso successivamente da John Elliott [10]. Si definirà infine una grammatica semi-automatica per analizzare le sequenze di accordi.

2.1 Introduzione al RoadMap: musical background

Così come accennato in apertura di capitolo, alla base dell'improvvisazione vi deve essere una conoscenza della struttura del brano (Jazz Chord Progressions) e delle tonalità di impianto che ne fanno parte. I più noti standard Jazz spesso si sviluppano su più tonalità ed i *chorus* si ripetono ciclicamente rispettando alcune regole e strutture note (es.Blues: 12 battute) o più genericamente suddividendo il *chorus* con una struttura del tipo "AABA" (A tipicamente è la Strofa, B il ritornello o una parte che presenta una modulazione armonica). Dunque, senza una coscienza approfondita l'improvvisazione potrebbe risultare arida, inarmonica e, perché no, anche poco piacevole all'ascolto. Un primo approccio improvvisativo si focalizza sulle tonalità di impianto, senza per forza dover pensare al singolo accordo. Tale approccio conviene soprattutto in presenza di brani con molti accordi, piuttosto che per brani veloci, con un ritmo incalzante ed un numero considerevole di battiti per minuto (es.200bpm *fast swing*). Il nostro interesse è allora quello di provvedere ad **un'automazione** che aiuti il musicista ad imparare, capire ed interiorizzare intere Jazz Chord Progressions(JCP) invece di accordi specifici o coppie di accordi isolate tra loro.
Altrettanto importante è dare un suggerimento idiomatico mirato ad identificare e apprezzare quelle che sono le progressioni più comuni, le cadenze risolutive del brano, così come i più tipici *turnaround*.
Pensando in termini di idiomi, piuttosto che in termini di brani completi (*tunes*), tali suggerimenti possono offrire un aiuto considerevole ed intelligente, mirato all'interiorizzazione e all'analogia di strutture armoniche in diverse tonalità.
Infine, un altro obbiettivo dell'automazione - nel lungo termine - è l'efficenza risultante dall'analisi armonica computerizzata che risulterà più robusta e fedele rispetto alla soggettività di un'analisi umana.

2.1.1 Grammatica ed algoritmo di analisi: Ulrich (1977)

Negli ultimi anni diversi sono stati gli approcci grammaticali costruiti per rappresentare sequenze armoniche comuni alle progressioni jazz. Ulrich [12] presentò degli algoritmi per analizzare ed individuare tonalità maggiori ed accordi partendo con le note negli accordi espresse come il numero di semitoni dalla nota più bassa. Fu dunque il primo a stabilire l'insieme di tonalità alle quali ogni accordo può prendere parte. Allora, incominciando l'analisi con piccoli segmenti di accordi presi singolarmente, il suo algoritmo - iterativamente - combinava quelli tra loro adiacenti in una segmentazione più grande tale da poter giustificare la presenza di una sola tonalità d'impianto. L'algoritmo si ripeteva finché nessun'altra segmentazione era possibile. Constatando la differenza di risultati a seconda della combinazione di coppie di accordi, fu inoltre stabilito un ordine, da sinistra verso destra, ignorando altre possibilità. Infine, un altro algoritmo, una volta che le tonalità del brano erano stabilite, derivava la funzione degli accordi all'interno del brano (*i gradi armonici*).

2.1.2 RoadMap a confronto con altri studi

Il *Roadmap* può essere visto come un completamento del precedente algoritmo. Seppur utilizzando lo stesso approccio, esso utilizza un algoritmo di ***Parsing*** *(CYK Parsing algorithm)* per identificare le progressioni idiomatiche di accordi da cui deduce le tonalità di appartenenza. Il RoadMap non accetta a priori il solo ordine *"left to right"* ma definisce una programmazione dinamica basata sui costi. Ciò favorisce la determinazione di un'analisi coerente per una certa sequenza. Vi è dunque anche un approccio *"looking-forward"* che, dipendendo dal contesto, riesce ad ottimizzare - grazie ai costi e alla grammatica - l'efficienza e il risultato dell'algoritmo.
L'intero approccio è basato sulla grammatica, a differenza di Ulrich che utilizzava la grammatica solo per il *parsing* dei simboli dell'accordo, non per le progressioni di accordi. Steedman [13] propose una grammatica per generare sequenze di accordi Blues, basata sui nomi degli accordi piuttosto che sulle note. Nel 1996 semplificò la presentazione in una grammatica libera dal contesto.
Chemillier [14] esaminò l'approccio di Steedman in profondità e, con maggiore

accuratezza, ne apportò dei miglioramenti. Pachet [3][15] osservando i precedenti approcci basati su grammatiche, ne descrisse i casi d'uso ed i problemi che con più frequenza si presentavano. Descrisse un approccio basato su ***regole*** per analizzare le progressioni, con l'auspicio di fornire un'estensione al lavoro operato da Ulrich, così come descritto in precedenza [vedi 2.1].

2.1.3 Il concetto di "bricks" nel RoadMap

Il RoadMap presenta molte analogie con il lavoro di Pachet, condividendone alcuni aspetti. L'ontologia di Pachet, ad esempio, include anche quelle *forme globali* della struttura definite nel gergo jazzistico: *forme blues, AABA, ABA*. L'analisi di forme globali non è uno degli obbiettivi del RoadMap dato che l'obbiettivo primario è quello di catturare i "*bricks*": il nome che identifica progressioni idiomatiche locali. Essi non hanno una lunghezza fissa, tendenzialmente possono essere di 8 battute, o anche più corti. L'ontologia di Pachet presenta solo 5 diversi idiomi: *Turn Around 1, Turn Around 2, II-V, II-V-I* e *cadenze dominanti* di risoluzione.

Roadmap invece, offre un dizionario estendibile dall'utente, includendo anche diversi nomi per un idioma e un set di idiomi di default decisamente più ampio rispetto a Pachet. Basandosi sull'influenza di Cork [1998, 2008] e gli studi di Elliot [2009], il dizionario di base di RoadMap offre una vasta gamma di *turnarounds*: *Plain Old Turnaround (POT), Suspended POT, Ladybird Turnaround, Foggy Turnaround, Whoopee Turnaround, etc.* (alcuni nomi derivano da brani celebri in cui effettivamente risiedono). Allo stesso modo vasta è anche la gamma di cadenze: *Straight Cadence, Extended Cadence, Long Cadence, Starlight Cadence, etc.* e dei corrispondenti ***approcci*** (ovvero la parte di una cadenza prima che risolva), così come le cadenze estese con ***dropbacks*** ed ***overruns*****.** Le prime sono quelle cadenze che raggiungono accordi *sopratonici***,** che rappresentano cioè la seconda nota di una scala diatonica. Ad esempio nella scala *maggiore di Do* la sopratonica è la *nota Re*. L'accordo di sopratonica allora sarà composto dalle note: *Re, Fa e La*. Mentre per gli *overruns* si intende ad esempio un accordo di IV grado che risolve sul I grado.

Molti di questi *bricks* sono definiti gerarchicamente. Così come Pachet, il RoadMap è capace di riconoscere il carattere ciclico dei chorus jazz, quando gli accordi finali di un chorus risolvono sull'accordo iniziale dello stesso. E' inclusa nella sua definizione anche la sostituzione di accordi, gestendo le varie sostituzioni grazie ad una fase di *pre-processamento*. In alcuni casi anche usando il *"brick dictionary"* che permette multiple variazioni su uno specifico tipo di brick.

2.1.4 Introduzione al CYK Parsing

RoadMap sfrutta il *Parsing* per le sequenze di accordi cercando di ottenere il miglior "*parse*", la migliore analisi per un brano sotto forma di unità idiomatiche. Un problema significante al quale bisogna prestare attenzione è la risoluzione di ambiguità, dovute al fatto che le sub-sequenze di accordi all'interno di un brano potrebbero essere interpretate in diversi modi. La soluzione del tool RoadMap è quella di utilizzare un algoritmo modificato di Parsing: *Cocke-Younger-Kasami (CYK) parsing algorithm*, seguito da uno step per minimizzare il costo associato al cammino, entrambi basati su una programmazione dinamica in JAVA. Si assume inoltre che l'algoritmo di Pachet, essendo base di quello di Ulrich, risolva le ambiguità accettando l'ordine *left-to-right*, che invece RoadMap non accetta.
RoadMap inoltre identifica i ***joins*** (ad esempio le transizioni tra i bricks, che verranno discusse con maggiore dettaglio successivamente). RoadMap dunque, considerando il panorama generale e gli studi finora effettuati nell'ambito dell'informatica musicale, rappresenta il primo tool che automatizzi l'identificazione dei *joins*.
La presentazione e l'ergonomia fa si che vi sia una sostanziale differenza tra RoadMap e Pachet. Quest'ultimo utilizza una rappresentazione ad albero a differenza di una visualizzazione di "*roadmap*" basata sui *bricks* (nel caso del RoadMap). La gerarchia che prende risalto in una struttura ad albero può allo stesso modo essere messa in evidenza nel secondo caso, dato che la suddivisione dei *bricks* è a sua volta, interattivamente, decomposta in *sub-bricks* ad un livello gerarchico più basso.
Considerando ulteriori approcci e studi inerenti all'analisi armonica, si può citare lo studio effettuato da *Scholz, Dantas e Ramalho* [15]. Anch'essi hanno focalizzato l'attenzione su alcuni aspetti comuni come la sovrapposizione e l'ambiguità; va detto però che, non avendo elaborato un approccio basato sull'algoritmo di Parsing, la

definizione di regole risulta più semplicistica e in forma ridotta. Indipendentemente dall'implementazione dell'algoritmo CYK fatta nel RoadMap, *Wilding* [16] ha usato lo stesso algoritmo per costruire un analizzatore per l'analisi dei numeri Romani, sfruttando il formalismo della "grammatica categorica" di *Steddman*, a differenza dell'identificazione degli idiomi specifici. Infine *Choi* [17] ha provveduto all'implementazione di un algoritmo per l'analisi armonica trattando la segmentazione di tonalità come un'ottimizzazione del problema, derivando le liste dei numeri Romani per funzioni di accordi, evidenziando gli approcci e le cadenze mediante l'utilizzo di frecce. Nel caso di RoadMap le unità sono maggiormente combinate in *bricks* idiomatici che ne indicano la funzione.

2.2 Un Approccio per Analizzare le Progressioni di Accordi

Un promettente approccio per l'analisi delle progressioni jazz fu proposto da Cork [8] [9] sotto il nome di metodo "*Lego Brick*". Cork suggerì che, considerando diversi brani, le progressioni di accordi possono essere strutturate e più facilmente ricordate, dividendo la progressione in mattoni idiomatici. Un *brick* non ha lunghezza fissa, ma tipicamente ha la durata di una, due, quattro oppure otto battute. Per esempio, una cadenza fa parte di un *brick* (sebbene come discusso in precedenza i *bricks* non sono limitati a rappresentare solo cadenze). Un'ulteriore discussione dei *pro,* relativi all'utilizzo dell'analisi mediante *bricks,* in contrapposizione a metodi tradizionali, sarebbe ridondante e controproducente, dato che si tratta comunque di argomenti e metodologie ad oggi innovative. Ad ogni modo, il tool di RoadMap, grazie alla sua implementazione *Open-Source* è sicuramente da considerarsi una risorsa innovativa e condivisibile con ampi margini di miglioramento.

Alcuni *bricks* possono avere un orientamento armonico maggiore, minore o ancora di dominante, tali orientamenti sono definiti all'interno del dizionario. Sebbene il dizionario dei *bricks* permetta l'introduzione di uno spazio temporale relativo (e.s., una progressione II-V della durata di una battuta o due battute), nella definizione dei *bricks*, l'algoritmo non sfrutta correntemente tale spazio. Ciò potrebbe dunque essere un interessante topic per il miglioramento futuro.

La *Figura 2.1* presenta quattro esempi di *bricks* idiomatici, così come sono stati identificati dal tool di analisi. La nomenclatura attribuita a ciascuno dei *bricks* può

essere, come si evince, sia convenzionale che colloquiale. Ad esempio, il nome "Rainy Turnaround" nel secondo brick di *Figura 2.1* fu introdotto da Elliot [10] perchè presenta un tipo di progressione famosa nel brano: "Here's That Rainy Day".

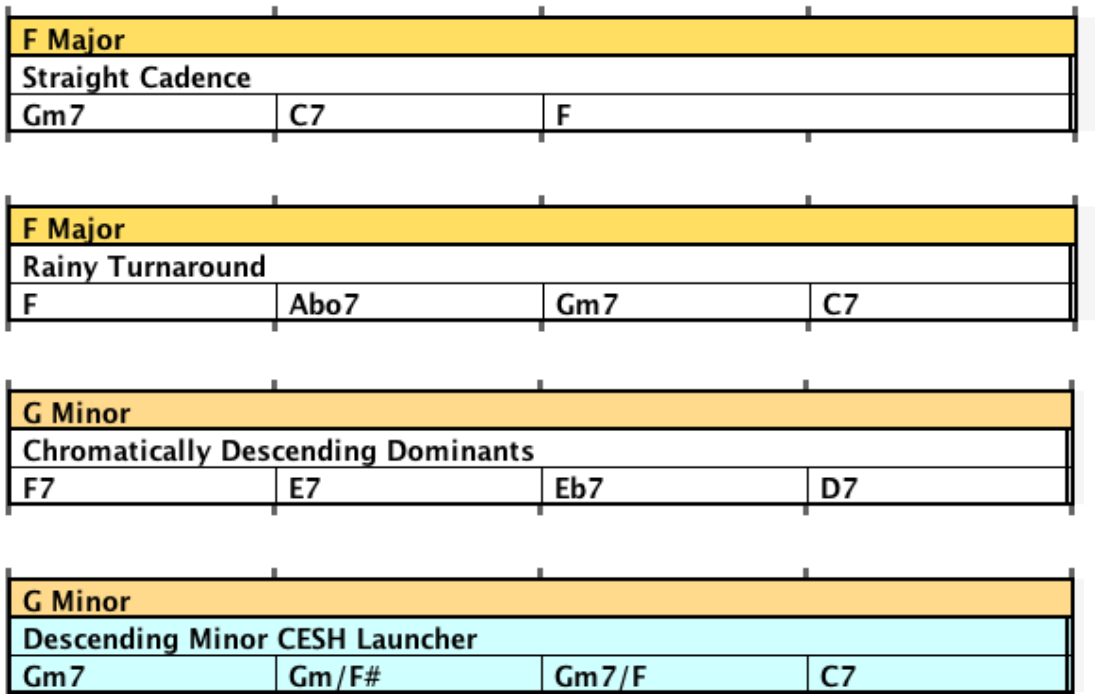

Figura 2.1 - Quattro esempi di bricks idiomatici. La terza riga di ciascun brick rappresenta l'input della sequenza di accordi, la riga di mezzo indica il nome (convenzionale o colloquiale) assegnato al brick, infine la riga in alto si riferisce alla tonalità di impianto.

In alcuni casi, la tonalità d'impianto di un *brick* è dedotta dal contesto piuttosto che dall'analisi di un *brick* isolata. Per esempio, nel terzo *brick*, la discendenza cromatica di 4 accordi di dominante : F7 | E7 | Eb7| D7| potrebbe far cadere in inganno o creare ambiguità, essendo il D7 la dominante sia della tonalità maggiore di G che di quella minore. Tuttavia, se si guarda alla prosecuzione della progressione, il tool di analisi correttamente assegnerà la tonalità minore di G dedotta dal contesto successivo. La *Tabella 2.1* riassume le convenzioni nella nomenclature di accordi che oggi si usano maggiormente nel jazz e che, a sua volta, utilizza il tool di RoadMap.
Esistono diversi tipi di bricks, includendo cadenze, approcci, turnarounds, launcher, etc. La maggior parte di questi termini sono diffusi anche al di là dell'analisi puramente jazzistica, per quanto riguarda la rimanente parte, la cui nomenclatura è

più concettuale, il loro significato dovrebbe essere chiaro dipendendo dal contesto in cui essi si trovano. Il termine "*launcher*" è forse il meno familiare tra questi. Esso denota una cadenza nella quale la risoluzione avviene in una nuova sezione dalla forma separata, ma come conseguenza diretta della precedente. Tale termine fu introdotto da Cork il quale suggerì che una cadenza lanciasse (*lanuches*) verso una nuova frase appartenente ad una nuova sezione (ad esempio il passaggio da una sezione A del *chorus* alla B). Elliot provvide ad una definizione più esaustiva di questi e altri tipi di *bricks*.

2.3 Joins

Ovviamente, ogni *brick* può essere trasposto in una delle dodici tonalità. Cork suggerì che, oltre ai *bricks*, esistono transizioni virtuali chiamate "*joins*" (giunture) tra i bricks. Sebbene alcuni di essi potrebbero essere interpretati come cambi tonali, Corks preferì descriverli come un "momento magico" nella progressione, dove vi è un'inusuale ma affascinante transizione da un sound stabile (es. cadenze di risoluzione canoniche), quali potrebbero essere un accordo di sesta maggiore o di settima, ad un sound instabile, quali accordi di settima minore o di dominante. L'instabilità del sound serve appunto da transizione e può portare ad una risoluzione verso un diverso sound stabile in un'altra tonalità, ma Cork preferì pensarlo come un "sound di transizione" piuttosto che un vero e proprio cambio tonale a cui attingere. Ciò dunque definisce il tipo di join. La *Figura 2.2* si riferisce a joins che sono evidenziati da etichette (tags) al di sotto delle tre tracce. Vengono mostrati tre tipi di joins: *New Horizon, Bauble e Homer*. Il primo di questi deriva dal famoso brano *How High The Moon*, dove vi è una modulazione tonale un tono sotto passando per una transizione da maggiore a minore della stessa fondamentale. Il nome "Bauble" deriva dal brano *"Baubles, Bangles and Beads"* ed è spesso ricorrente in molti brani quando vi è una transizione dal I grado al III minore. Infine, il terzo *join*, "Homer", è usato come risoluzione ciclica per tornare alla stessa tonalità attraverso una progressione II-V in chiave, nel caso specifico in G minore.

Characters	*Meaning*	*Example Spelling*
b	Flat	Bb = B flat
#	Sharp	F# = F sharp
m	Minor	Cm = {C, E♭, G}
M	Major	C = {C, E, G}
7 (no M or m)	Dominant seventh	C7 = {C, E, G, B♭}
m7	Minor seventh	Cm7 = {C, E♭, G, B♭}
M7	Major seventh	CM7 = {C, E, G, B}
o7	Diminished seventh	Co7 = {C, E♭, G♭, A}
m69	Minor triad with added sixth and ninth.	Cm69 = {C, E♭, G, A, D}
m7b5	Minor seventh with flat 5 (also known as half-diminished)	Cm7b5 = {C, E♭, G♭, B♭}
/	The pitch after the slash is the bass note.	Gm/F# = {F♯, G, B♭, D}

Tabella 2.1: La nomenclatura utilizzata per identificare gli accordi

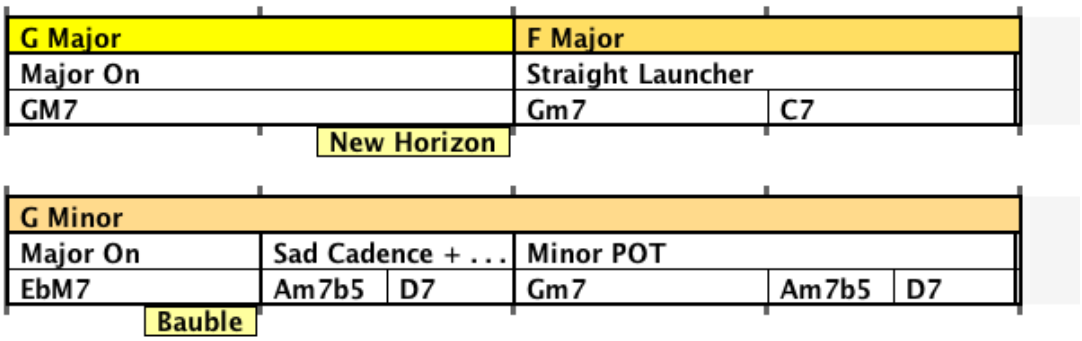

Figura 2.2: Tre esempi di joins, indicati dal tag al di sotto dei bricks (New Horizon, Bauble e Homer), così come appaiono all'interno del Tool di RoadMap.

Ci sono dodici diversi *joins*, corrispondenti ai possibili intervalli cromatici tra le classi dei pitches. Questi sono mostrati in *Figura 2.2***.** Il lavoro effettuato da Elliott sicuramente estende quello di Cork, rilevando e classificando una varietà di *bricks* più ampia che occorre negli standards jazz. Approssimativamente vi sono 21 tipi di *bricks* identificati da Cork; 34 in più quelli identificati da Elliott. Inoltre, il contributo di quest'ultimo provvede anche ad una organizzazione chiara e corretta dal punto di vista tassonomico. Elliott fornisce *241 esempi* in forma di road maps che rappresentano altrettanti brani con progressioni di accordi suddivise in *bricks e joins*. A questo punto diventa logico utilizzare la terminologia definita da Elliott all'interno del tool di RoadMap.

2.4 Dichiarazione del Problema

Le tecniche presentate in questo paragrafo hanno come obbiettivo quello di risolvere due problemi per l'*automazione dell'analisi*:

1. Data una progressione di accordi, determinare un mapping dalle istanze dell'accordo nella progressione alle tonalità.
2. Data una progressione di accordi, analizzare la progressione come una serie di idiomi armonici chiamati *bricks* e, ove vi è rilevanza, fare il *joins* tra i *bricks*.

L'approccio utilizzato nel metodo risolve dapprima il secondo problema, dopodiché utilizza l'informazione della tonalità della progressione di accordi per risolvere il primo. Dunque vi è un approccio opposto rispetto a quello di Ulrich, il quale provava a stabilire prima la tonalità e poi identificare la funzione degli accordi all'interno della tonalità. Il repertorio, il dizionario di possibili bricks è predefinito in una tabella chiamata **brick dictionary**, proprio perchè provvede ad associare ad il nome del brick la sua componente. Affinché un dato nome abbia più di una realizzazione in una sequenza di accordi, tale dizionario funge da grammatica (Chomsky 1956 [7]). Le componenti dei bricks possono essere sia accordi che altri bricks. Ripetendo l'espansione, ciascun brick potrà eventualmente essere tradotto in una sequenza di accordi.

2.4.1 Esempio di Analisi

Si prenda come esempio uno standard jazz quale "Bye Bye Black Bird". La progressione di accordi del brano è mostrata in *Figura 2.3*. L'input è visualizzato come una *"lead sheet"*, ovvero una notazione di griglie che comunemente viene condivisa tra i musicisti jazz. Le barre verticali nella tabella stanno ad indicare le divisioni tra le varie battute. Uno *slash* (/) a sua volta indica la ripetizione esatta degli argomenti espressi nella battuta precedente. Infine, come espresso precedentemente nella **Tabella 1**, viene utilizzata una convenzione per esprimere il nome degli accordi.

L'output dell'analisi si può apprezzare invece in *Figura 2.5*. La griglia (*lead sheet*) di *Figura 2.3* include i produttori di sezione (*section makers*), utili e a volte anche necessari per ridurre le ambiguità nell'analisi. Nell'esempio in questione comunque il loro apporto è minimo. Come si può notare nella road map in *Figura 2.4*, sia le informazioni tonali (nella riga superiore: *upper track*) che le informazioni idiomatiche del brick (*middle track*) sono ottenute dalla leadsheet in ingresso (*bottom track*). Si noti come i bricks il cui nome finisce con il simbolo "+..." (così come il primo brick della riga 2) stanno ad indicare il fatto che l'ultimo accordo del brick è condiviso con il primo accordo del prossimo brick. In questo esempio , l'accordo di Gm7/F fa parte sia dell'etichetta del brick "Descending Minor CESH" che di "Two Goes Straight Cadence". Ciò prende il nome di **pivot chord**, ovvero un accordo condiviso. L'analizzatore correttamente identifica la sequenza Gm7 | Gm/F# | Gm7/F presente nelle righe 2 e 3 come un'elaborazione contrapuntale di armonia statica (Contrapuntal Elaboration of Static Harmony - CESH), volendo utilizzare la terminologia di Coker (1997). Così come descritto precedentemente, POT indica sta per Plain Old Turnaround, un termine utilizzato da *Aebersold* [11] ed anche nella terminologia di Cork. SPOT (Superended POT), significa invece che un accordo di **III grado** occorre quando il **I grado** iniziale occorre nel POT.

```
(section) FM7 | Gm7 C7 | F6 | / | F6 | Abo7 | Gm7 | C7 |
(section) Gm7 | Gm/F# | Gm7/F | C7 | Gm7 | C7 | F6 | / |
(section) F7 | E7 | Eb7 | D7 | Gm7 Gm/F# | Gm7/F Gm/A | Gm7 | C7 |
(section) FM6 | Gm7 C7 | F6 | Am7b5 D7 | Gm7 | C7 | F6 | Gm7 C7 |
```

Figura 2.3: Notazione Lead Sheet dello standard Bye Bye Blackbird.

La definizione dei *bricks* è spesso basata su osservazioni empiriche di quegli idiomi comuni utilizzati dai compositori jazz e dai brani più celebri. Nell'analisi, i *bricks* sono definiti in un dizionario, il risultato finale dell'analisi è fortemente dipendente dal contenuto del dizionario. Si utilizza nel tool una base costituita da un insieme di bricks definiti da Elliott, con piccole aggiunte e modifiche.

```
(defbrick Perfect-Cadence Major Cadence C
          (chord G7 1)
          (chord C 1))

(defbrick Straight-Approach Major Approach C
          (chord Dm7 1)
          (chord G7 1))

(defbrick Straight-Cadence Major Cadence C
          (brick Straight-Approach C 2)
          (chord C 2))

(defbrick POT Major Turnaround C
          (chord C 1)
          (chord Am7 1)
          (brick Straight-Approach C 2))

(defbrick Extended-Approach Major Approach C
          (chord Am7 C 1)
          (brick Straight-Approach C 2))

(defbrick Extended-Cadence Major Cadence C
          (brick Extended-Approach C 3)
          (chord C 1))

(defbrick Minor-On(main) Minor On C
          (chord Cm 1))
```

Figura 2.4: Rappresentazione del brick nel dizionario, Ogni definizione è una espressione di simboli che produce un nome, una sua variante opzionale (tra parentesi), il modo, il tipo del brick, la tonalità risultante e una lista dei bricks costituenti.

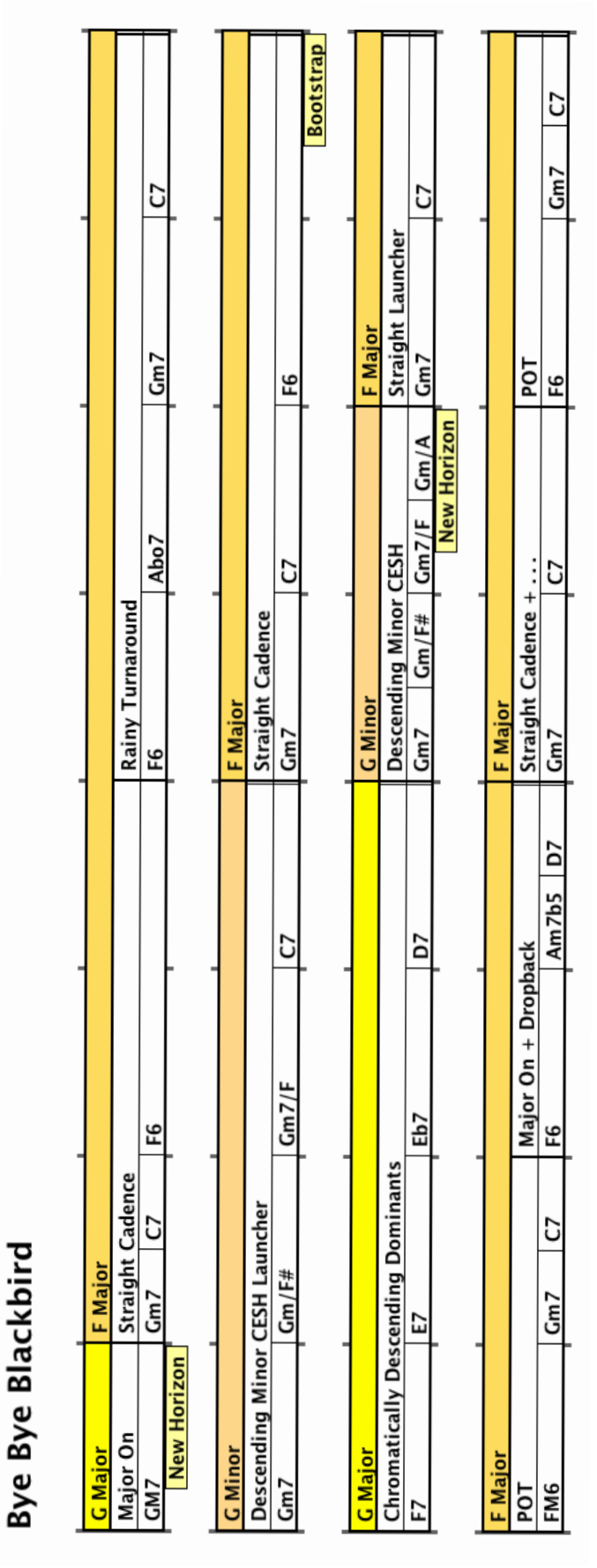

Figura 2.5: Road map del brano Bye Bye Blackbird

2.5 La grammatica del brick dictionary

Il *brick dictionary* può essere visto come una grammatica (Chomsky 1956 [7]) nel seguente significato: i simboli terminali della grammatica sono rappresentati dai simboli dell'accordo. I simboli non-terminali della grammatica sono i nomi associati ai bricks, accompagnati da una cera informazione che li qualifichi, come ad esempio la tonalità o la funzione. Ogni entry del dizionario è effettivamente una regola di produzione nella grammatica. La *Figura 2.4* mostra un paio di esempi di regole a partire dal dizionario in forma testuale seguendo: il nome del brick, la tonalità (il modo, es.: maggiore o minore), la categoria del *brick* (definibile anche dall'utente) e la relativa funzione, rappresenta una sequenza di oggetti ciascuno dei quali può essere un accordo o il nome di un altro *brick*. Entrambi specificano la durata relativa (sebbene l'informazione non sia attualmente sfruttata) e, nel caso del *brick*, deve specificare la tonalità a cui appartiene. Solo una regola è inclusa per tutte le 12 possibili toniche. L'algoritmo di parsing si fa carico della trasposizione della regola alla tonalità rilevante di interesse. Un dizionario dunque capace di analizzare la maggior parte degli standard jazz facenti parte del vasto repertorio di oltre 500 brani. Una forma di grammatica non-deterministica ottenuta grazie alla definizione multipla di certi *bricks*, ciascuno con una definizione diversa seguita da un processo di ottimizzazione che elimina le ambiguità nella spiegazione.

2.6 Analysis Technique - Tecniche di Analisi

In questo paragrafo vengono illustrate le tecniche di automazione di analisi dei bricks di accordi per ottenere una profonda analisi della struttura della progressione di accordi nel brano jazz. Nel modello, un brick può essere una sequenza di soli accordi o di altri bricks. Questa generalizzazione permette la possibilità di una classificazione gerarchica, nella quale bricks più complessi vengono generati grazie a quelli più semplici. Un accordo singolo differisce da un brick perché ad esso non vi è associata alcun tipo di funzione, mentre un brick, ad esempio, può fungere da cadenza, in aggiunta agli accordi esistenti. Allora un brick è generalmente rappresentato da una sequenza di oggetti che possono essere solamente accordi, bricks più piccoli, o entrambi. Si può parlare di "meta-brick" per denotare un brick composto da altri

bricks. Le *Figure 2.6* (a,b,c) illustrano come un brick possa essere decomposto gerarchicamente. Il *brick* di inizio è l'incipit di "Autumn Leaves". Viene quindi suddiviso in due cadenze che possono maggiormente essere decomposte nei rispettivi approcci e blocchi di risoluzione. Si continua la procedure finchè nessun'altra suddivisione è possibile e definendo tale operazione come "flattening" del brick. L'analisi essenzialmente inverte il processo di segmentazione, iniziando con accordi e raggruppandoli in bricks sempre più grezzi. Uno dei problemi principali è che talvolta più di una combinazione è possibile. Si trova allora la migliore tra le possibili in modo da descrivere correttamente il brano in analisi. L'analisi è effettuata in due fasi: la prima analizza col parsing le progressioni di accordi in *bricks* e restituisce la migliore sequenza di brick percepita per la progressione. La seconda fase usa invece la sequenza di brick per determinare la key map e i joins. Quest'ultima fase potrebbe anche modificare alcuni bricks precedentemente trovati.

2.7 Regole Grammaticali

La grammatica dei bricks è basata sull'idea di avere accordi come simboli terminali e bricks come simboli non-terminali. Si utilizza l'algoritmo CYK di parsing (Cocke-Younger-Kasami) [18] in quanto si rivela essere un buono strumento di analisi per i bricks grazie alla sua fessibilità nel parsing di grammatiche ambigue. La conditio sine qua non da applicare all'algoritmo CYK è che la grammatica deve essere espressa nella

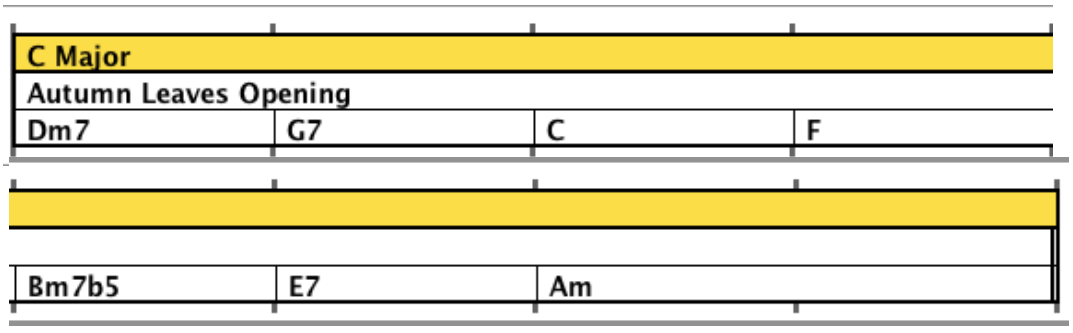

Figura 2.6 (a): Esempio di un blocco di gerarchia con successiva decomposizione di bricks: brick composto da 8 accordi.

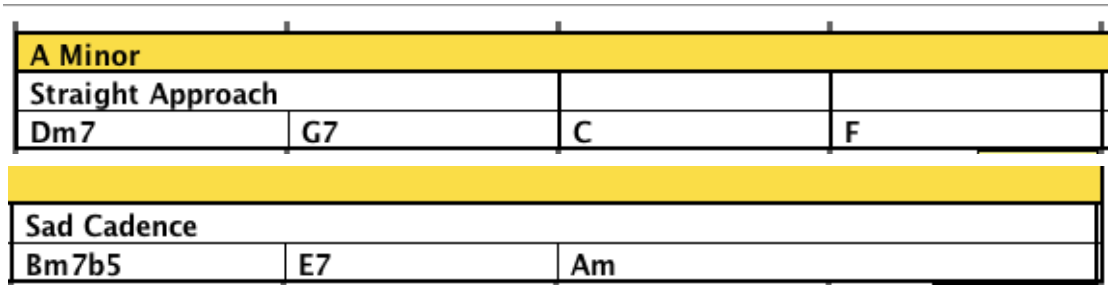

Figura 2.6 (b) : 2 bricks da 4 accordi intesi come cadenze.

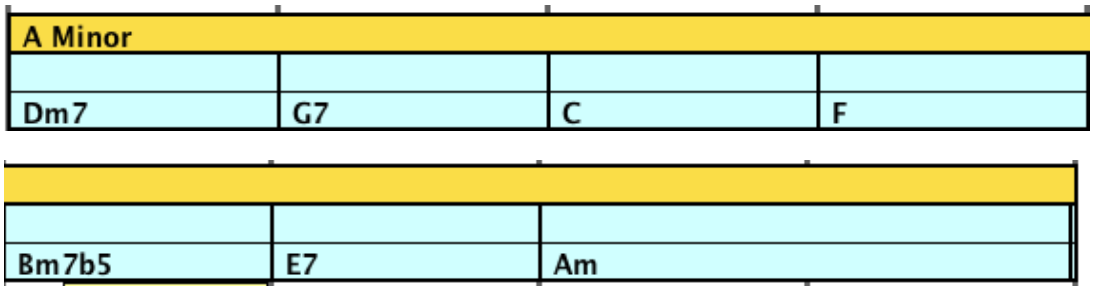

Figura 2.6 (c): sequenza di accordi dopo "flattening"

forma normale di Chomsky (CNF), nella quale ogni regola di produzione deve essere una dei due tipi:

1. Binaria: A → B C, un simbolo non-terminale produce due simboli non-terminali.

2. Unaria: A → α, un simbolo non-terminale produce un simbolo terminale.

Per convenienza nel gestire le sostituzioni di accordi e le sequenze di progressioni jazz, si permette un terzo tipo di produzione:

3. Unaria: A → B, un simbolo non-terminale produce un differente simbolo non-terminale.

E' facile estendere l'algoritmo CYK affinché accetti anche il terzo caso.
Le regole nel dizionario di brick dell'utente possono avere più di due oggetti dal lato destro, se necessario, si può processare tali regole per introdurre simboli addizionali non-terminali.

2.8 Parsing Algorithm

L'algoritmo standard di Parsing CYK determina se c'è o meno un'analisi "parse" di una stringa in ingresso da un singolo simbolo che risulta essere quello iniziale (start symbol). All'interno del tool di RoadMap invece l'obbiettivo è leggermente diverso. Non serve avere un simbolo iniziale per generare nessun tipo di brano. Ciò che si vuole ottenere è invece analizzare una sequenza di non-terminali, *bricks* o singoli accordi. Il processo è descritto con la costruzione di un albero. Il CYK produce una foresta (un insieme di alberi) e la fase di minimizzazione dei costi combina gli alberi selettivamente per produrre un albero singolo. Gli alberi sono costituiti da nodi che possono avere zero, uno o due figli.

Nella fase iniziale di parsing, il primo accordo terminale è convertito in un nodo che rappresenta l'accordo e posizionato sulla entry diagonale (i,i) della tabella di parse. Se avvengono sostituzioni per l'accordo, i nodi per le sostituzioni vengono aggiunti alla diagonale. La logica di tale procedimento risiede nel fatto che per evitare un'esplosione combinatoria della dimensione del dizionario, non si mantengono le regole del dizionario per ogni possibile sostituzione. Infatti ci si affida agli accordi rappresentativi che accolgono l'insieme di possibili sostituzioni. Per esempio, sebbene una regola nel dizionario potrebbe includere un accordo di G7 sul lato destro della regola (la produzione effettiva), vi sono molti altri accordi che potrebbero essere sostituiti al G7, incluso G9, G13, G7b9, G7#9, G7b9#11, etc. Provvedendo allora a liste separate per tipi di sostituzioni tra accordi comuni si mantiene la dimensione ad una grandezza ragionevole. Un altro modo di gestire il problema sarebbe quello di utilizzare un singolo non-terminale per tutti i membri della famiglia di sostituzioni. Si può pensare che, dal punto di vista del musicista, usare un sistema per lavorare con un membro rappresentativo piuttosto che con un simbolo astratto possa risultare più intuitivo. Lo sviluppo della sequenza di *bricks* analizzata da una serie di accordi segue una procedura del tipo:

1. Per una sequenza di accordi in ingresso C1, C2, ..., Cn di lunghezza n, esiste una tabella T di dimensione $n \times n$. Ogni entry sulla diagonale (i,i) di T è inizializzata con un singolo nodo dell'albero che rappresenta l'accordo i-esimo Ci.

2. Il Parser applica l'algoritmo CYK per riempire le entries della tabella sulla diagonale (es. le entries (i,j) dove i < j) con alberi. Ogni albero rappresenta un'analisi di tutte le sottosequenze contigue degli accordi Ci, ..., Cj. Ogni figlio di un dato nodo è una radice di uno dei sotto-alberi che combinati formano il nodo dell'albero. L'algoritmo iterativamente calcola tutte le entries T(i, i+1) per ogni i, poi T(i , i+2) , etc., fino ad arrivare a T (1, n). Il calcolo della tabella T(i,j), per i<j è ottenuto mediante un'iterazione sistematica attraverso coppie in ingresso T(i,k) T (k+1, j) per ogni j, dove $i \leq k \leq j$. Per ogni coppia di questo tipo il parser esamina tutte le le regole di produzione di TIPO 1 (il nome deriva dalla classificazione del tipo di regole di produzione definita nella sezione precedente) per vedere se la coppia di non-terminali nel corpo delle produzioni corrisponda a quelle nella coppia correntemente selezionata. Sono necessarie considerazioni extra rispetto all'algoritmo standard di CYK per connettere sequenze sul lato destro con il corretto "relative key difference" e per derivare l'attributo tonale della parte sinistra come non-terminale. (Essendo le regole grammaticali definite per una tonalità, vi è il bisogno di trasporle per tutte le dodici tonalità. La trasposizione relativa è determinata quando il primo elemento della produzione (lato destro) è confrontato (match) con il nodo nella tabella. Per esempio, una regola per un approccio al *brick* Dm7 G7 nella chiave di C confronterebbe la sequenza dell'accordo Fm7 Bb7 nella tonalità di Eb trasponendo Dm7 di tre semitoni fino a raggiungere Fm7. A questo punto G7 viene a sua volta trasposto di tre semitoni per diventare Bb7 ed ottenere il corretto risultato).
3. Un algoritmo per trovare il cammino determina la combinazione a costo-minimo delle analisi nella tabella che descrive interamente il brano.
4. Un processing aggiuntivo ritocca il risultato in maniera da essere descritto successivamente.

Brick	*Cost*
Approach	45
Cadence	25
CESH	10
Deceptive Cadence	60
Dropback	30
Ending	30
Invisible	2,000
Misc	40
Off	1,000
Off-On	1,010
On	550
On-Off	1,005
On-Off+	100
Opening	25
Overrun	30
Pullback	45
Turnaround	20

Tabella 2.2: Assegnazione Tipica del Costo per diversi Tipi di Bricks.

2.9 Gestione delle regole di Sovrapposizione

Un ulteriore fattore a rischio per le sequenze di accordi da tenere in considerazione è l'overlap (sovrapposizione) tra patterns. Per esempio, l'accordo finale di un brick potrebbe attuare da pivot per l'accordo successivo, l'iniziale della battuta seguente. In *Figura 2.5* si è visto l'esempio nella seconda riga, all'interno del brano *"Bye Bye Blackbrid"*. per tenere conto di tali possibilità di sovrapposizione, ogni volta che un nuovo simbolo non-terminale viene messo in T(i, j), un non-terminale quasi identico viene a sua volta posizionato in T(i, j-1). Quest'ultimo avrà un accordo finale di durata pari a zero, dovendo considerare la presenza dell'accordo nel brick senza cambiare la durata complessiva del brano.

2.10 Trovare un Costo Minimo per la Tabella di Parsing

Per determinare la qualità di una data analisi basata sull'identificazione dei bricks delle fasi precedenti, il parser assegna un costo a ciascun tipo di brick, favorendo la sequenza con il più basso costo totale. I costi individuali di ciascun brick sono determinati euristicamente da alcuni fattori. La base del costo deriva dal tipo di Cadenza a cui è assegnata un costo pari a 30. Il costo è poi leggermente regolato nel caso di un overlap o di una sostituzione di accordo. La Tabella 2.2 i costi utilizzati per l'analisi sebbene l'utente ha piena libertà di specificare costi diversi rispetto a quelli indicati nella tabella.
I costi nella Tabella 2.2 sono stati determinati partendo da un'idea generale di tipi preferiti di bricks e successivamente modificati dal designer trial-and-error. I bricks più generici e gli accordi più isolati sono posti due ordini di grandezza più distanti - dal costo - rispetto ai bricks preferiti per l'analisi. Ad esempio, su bricks che rappresentano un accordo di tonica singolo, costano molto meno rispetto ai non-tonici, sicuramente meno stabili. Un brick rappresentante un turnaround è in qualche modo meno costoso di una cadenza se si vuole considerarlo come un insieme di cadenze, un'analisi leggermente meno informativa che dovrebbe processare lo stesso numero di bricks. Il costo di un approccio (la prima parte di una possibile cadenza incompleta come ad esempio un approccio II - V in una cadenza II- V - I) è inizializzato per essere maggiore dato che si da priorità a quei bricks che risolvino il più possibile nella forma di una cadenza. L'algoritmo di minimizzazione del costo è riassunto in *Figura 2.7*. Esso è simile all'algoritmo di Floyd [19] per trovare i cammini a costo minimo in un grafo etichettato. Considerando l'ordine in cui l'algoritmo aggiorna la tabella dei minimi, i suoi ingressi nella fase di aggiornamento non saranno mai vuoti. Dopo la minimizzazione dei costi, la cella in alto a destra della tabella di costi minimi (nodi dell'albero) avrà un livello superiore (tree node) contenendo il più basso costo possibile che rappresenti correttamente il brano. Tale nodo esisterà sempre. Nel peggiore dei casi ci sarà sempre una soluzione finita di assegnazione dei costi per l'analisi nella forma degli accordi originali. L'albero finito viene dunque riassemblato in una sequenza di blocchi che saranno la base della road map.

```
 mins = new Array<Node>[cykTable.length]
[cykTable.length]
    for row = 0 to mins.length – 1:
        for col = row to mins.length – 1:
          mins[row][col] = the lowest cost Node in
                cykTable[row][col]


    for i = mins.length – 1 to 0 by -1:
      for j = i + 1 to mins.length - 1:

       for k = i + 1 to j:

  if mins[i][k-1].cost + mins[k][j].cost < mins[i]
[j].cost:
        mins[i][j] = new Node(mins[i][k-1], mins[k]
[j])
            end if
```

Figura 2.7: Algoritmo per trovare il parse a costo minimo. La variabile CYKTable rappresenta l'array di tutti i possibili nodi nella lista dopo aver eseguito l'algoritmo CYK.

2.11 Post-Porcessing e Analisi Tonale

Dopo che il Parser restituisce una sequenza di blocchi, i risultati sono ridefiniti attraverso una serie di post-processing a più steps per trovare le tonalità delle sezioni, i launchers ed i joins.

Sebbene ogni brick (contando accordi isolati come bricks di un accordo) ha la sua tonalità memorizzato alla fine dell'analisi, la maggior parte dei brani restano in tonalità per più di un brick consecutivo, necessitando dunque un'accurata tracciabilità della sommaria progressione tonale del brano. Per ottenere questo rilevamento, la tonalità, il modo e la durata di ciascuna porzione del brano determinano una serie di "key spans" (archi tonali, periodi tonali). Quanto ai modi il tool utilizza tre modi

generalizzati: maggiore, minore e dominante (anche conosciuto come misolidio), sacrificando dunque altri modi più specifici per favorire la semplicità.
Per determinare la lunghezza e le caratteristiche di un key span, la lista dei bricks in un brano viene scansionata "backward" dalla fine all'inizio, confrontando la tonalità e il modo del brick attuale con la tonica del key span che segue immediatamente. Se il brick in questione può essere assorbito nel key span seguente, viene effettuato un processo di "merging" e la scansione prosegue. Altrimenti, se siamo in presenza di un nuovo key span lo si inizializza utilizzando la tonalità attuale del brick, il suo modo e la sua durata. La scansione continua fino a quando non si raggiunge l'inizio del brano.
Il metodo del confronto tonale differisce leggermente dipendendo dalla presenza di un "vero" brick piuttosto che un accordo isolato trattato come brick. Per un brick reale si effettua un semplice controllo per vedere se la tonalità e il modo del brick siano uguali a quelli del key span. Se si, il brick viene assorbito nel key span attuale.
Si consideri anche un caso speciale in cui il brick è un approccio che risolve sul key span. Ciò fu preso in questione da Cork che aggiunse il concetto di "cadenze di sorpresa" (suprise cadences) per progettare cadenze che risolvono su un modo differente rispetto a quello dell'approccio che porta ad essa. Analizzando un accordo singolo per possibile raddoppio in un arco tonale richiede un'infrastruttura addizionale, perché l'accordo in questione potrebbe appartenere a diverse tonalità, dipendendo dal grado della scala. Ogni accordo è controllato rispetto una lista di accordi comunemente usati in una data tonalità e modo. Queste liste sono mantenute per i modi minori, maggiori e dominanti nella tonalità di Do (C) e trasposti quindi appropriatamente per le altre tonalità.

2.12 Launchers VS Joins

Per allocare i launchers, ogni blocco finale di una sezione o frase è confrontato con quello iniziale della sezione immediatamente successiva, per testare se il primo blocco è un approccio che potenzialmente può risolvere sul secondo. L'ultimo blocco nel brano è confrontato con il primo dato che i chorus nel jazz sono quasi sempre ripetuti per permettere di ripetere, dopo il tema principale, le improvvisazioni estese. Dunque la fine del chorus rappresenta una transizione che ritorna all'inizio.

Uno “Straight Launcher” è semplicemente una cadenza perfetta (streight cadence) divisa su una frase o sezione armonica. La *Figura 2.8* mette in relazione una cadenza e un launcher. Si noti che la doppia barra nel mezzo della parte bassa, indica una break di frase. In questo caso, ci sono due bricks piuttosto che uno, così come nella metà alta. Il processo di trovare launchers e joins è implementato attraverso un’iterazione attraverso tutti i blocchi. Esiste anche una seconda soluzione alternativa: trovare launchers che risolvono in modo “speciale”. La componente di un brick come ad esempio Yardbird Cadence [Fm7 Bb7 C], se esaminata evidenzia la stessa risoluzione di un approccio straight in tonalità diversa, Mib (E-flat). Grazie al post-processing ogni blocco è confrontato col successivo per identificare eventuali approcci risolutivi speciali e convertirli in launchers. La *Figura 2.9* mostra un esempio di launcher Yardbird.[1]

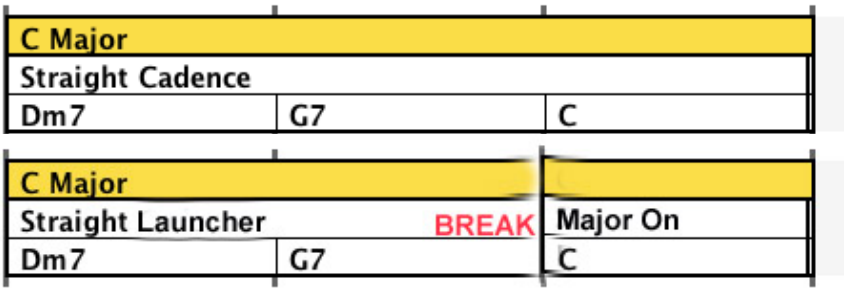

Figura 2.8: Cadenza e Launcher a confronto. Nel secondo caso (launcher) si noti il break di frase (o sezione) in corrispondenza della barra verticale. La cadenza è dunque divisa in un launcher ed un “On” brick rappresentante la risoluzione della progressione.

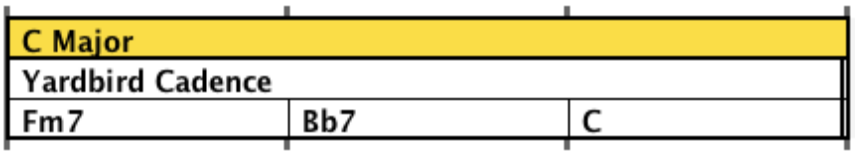

Figura 2.9: Risoluzione speciale per un launcher Yardbird. La risoluzione che ci si aspetterebbe dopo Bb7 è Eb ma, dato che risolve in C, eccola definita come cadenza Yardbird (Charlie Parker).

[1] La terminologia fu introdotta da Cork dopo l’analisi del brano “Yardbird Suite” di Charlie Parker, sebbene sia anche conosciuta in musica come una cadenza “back door” o una cadenza plagale minore.

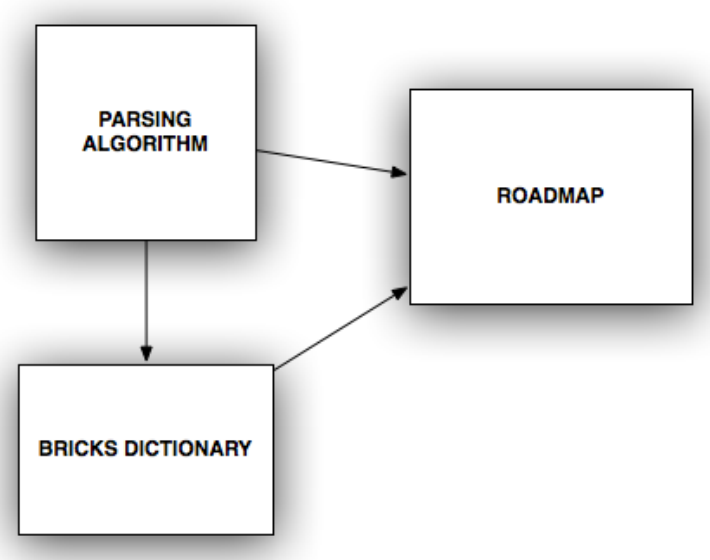

Figura 2.10: Diagramma UML tra i packages "RoadMap", "BrickDictionary" e "Parsing".

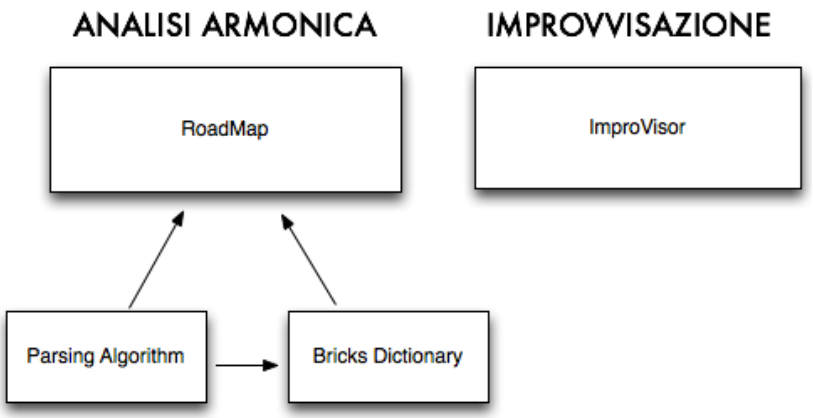

Figura 2.11: RoadMap per l'analisi armonica, ImproVisor per la generazione di melodie e pattern improvvisativi.

Le *Figure 2.10 e 2.11* evidenziano la relazione tra le classi all'interno del tool di RoadMap. Viene dato un primo sguardo al tool di improvvisazione, di cui si discuterà nel *Capitolo 3*.

Capitolo 3 - Modellazione Improvvisativa

Una Macchina di Apprendimento per Grammatiche Jazz

Nell'ambito di uno strumento software educativo in grado di generare nuovi patterns jazz utilizzando una grammatica probabilistica (Keller [1][5]), questo capitolo descrive l'apprendimento automatico di tali grammatiche. Ciò viene effettuato imparando da un corpus di trascrizioni, di solito di un singolo esecutore, ed i metodi implementati fanno si che la macchina possa improvvisare con alcuni tra gli stili più rappresentativi. Al fine di catturare le "gestures" idiomatiche di un solista specifico si estende una rappresentazione grammaticale (precedentemente introdotta da Keller e Morrison nel 2007) con una tecnica per rappresentare un *contorno* melodico. I contorni più rappresentativi (chiamati successivamente melodie astratte) sono estratti

da un corpus utilizzando il *clustering*. Il sequenziamento tra contorni avviene utilizzando *catene di Markov* che sono codificate nella grammatica.
Questo capitolo definisce in primo luogo gli elementi di base per i contorni di tipici assoli jazz, definiti come *slopes* (pendenze), mostra poi come questi pendii (slope ascendente/discendente) possono essere incorporati in una grammatica in cui le note sono scelte in base alle categorie tonali rilevanti dello stile jazzistico . Si mostrerà che i contorni melodici possono essere rappresentati con precisione utilizzando *slopes* imparati da un corpus.

3.1 ImproVisor: introduzione e lavori correlati

Le Grammatiche sono alla base delle tecniche di generazione melodica. L'utilizzo delle grammatiche per creare strutture musicali è stato già oggetto di progetti di ricerca da parte di numerosi studiosi (*Winograd 1968, Roads 1979, Bell e Kippen 1992; Cope 1992; McCormack 1996; Pachet 1999 [3] [23]; Papadopoulos e Wiggins 1999* solo per citarne alcuni). *Dubnov* [20] utilizzò una macchina probabilistica e statistica di metodi per l'apprendimento e il rilevamento di stili musicali. *Eck e Lapalme* [21] studiarono la composizione automatica e l'improvvisazione con le reti neurali, Cruz-Alcazar e Vidal-Ruiz [22] svilupparono un metodo per imparare le grammatiche e modellare uno stile musicale. Quest'ultimo applicò tre algoritmi di inferenza grammaticali per la composizione automatica di: melodie Gregoriane, melodie appartenenti allo stile di Joplin e di Bach. Le prime diedero sicuramente i migliori risultati. Vidal-Ruiz classificarono il 20 per cento delle melodie composte come *"very good melodies"*, ovvero melodie verosimili e coerenti, definite come *"un pezzo originale dello stile attuale, senza essere una copia o senza contenere frammenti evidenti da campioni pre-impostati"*.
Una parte importante della rappresentazione grammaticale implica una formalizzazione del concetto di contorno melodico. *Kim et al.* [4] usarono contorni per la classificazione musicale, *Chang e Jiau* [24] studiarono il contorno musicale con applicazioni volte ad estrarre le figure e i temi che si ripetevano all'interno di un

brano. Inoltre, *De Roure e Blackburn* [25] proposero dei contorni melodici di pitch per un'analisi musicale basata sul contenuto.
All'interno del tool *ImproVisor (Keller)* si utilizza un *clustering* come mezzo di organizzazione e astrazione di grande varietà, un mezzo che raccoglie dei frammenti melodici simili tra loro e che, grazie alle catene di *Markov*, riesce a legarli in maniera coesa e a rappresentare le possibili transizioni tra i frammenti astratti. *Kang, Ku, e Kim* utilizzarono un algoritmo di clustering grafico per l'estrazione dei temi melodici. *Jones* [26] invece descrisse gli usi sia delle catene di Markov che delle grammatiche per la composizione musicale. *Verbeurgt, Dinolfo, e Fayer* [27], tra gli altri, usarono modelli di Markov come composizione e apprendimento di probabilità di transizione tra patterns. Ames [6] così come Simone Bollini [29] introdussero catene di Markov a lunghezza variabile per la rappresentazione di pattern musicali, ottenendo sicuramente interessanti risultati dal punto di vista armonico, un po' meno dal punto di vista ritmico.

3.1.1 L'improvvisazione Jazz

Idealmente, l'improvvisazione jazz rappresenta la creazione di nuove melodie, una composizione istantanea nel momento stesso in cui le melodie vengono eseguite. È ovvio che questo processo, intrinseco nel musicista, preveda una precedente creazione grazie alla pratica costante delle idee che man mano formano il vocabolario che, ciascun musicista, ha costruito prima della performance. Pertanto uno degli scopi è quello di costruire degli strumenti software che facilitino la costruzione e la registrazione di tali idee. Tale strumento può anche essere utilizzato per trascrivere e analizzare idee esistenti. Il presente capitolo infatti mostra che, una volta trascritto, un assolo può essere immagazzinato in un database da utilizzare per la creazione di una grammatica, che sarà quindi utilizzato per fornire improvvisazioni su qualsiasi progressione di accordi e non solo per quelli del corpus già esistenti. Anche se un dato performer jazz potrebbe non essere consapevole di come lui (o lei) improvvisi, sembra ragionevole affermare che le idee istantanee che costituiscono l'improvvisazione derivino dall'insieme dei pattern e dagli studi precedentemente acquisiti sotto forma di modelli o, più in generale, sotto forma di grammatica. È ovvio

dunque che un insieme finito di pattern può essere descritto da una grammatica ad hoc. Allo stesso tempo però una grammatica troppo ad hoc, invece, tenderebbe a generare melodie prevedibili, risultando poco interessante. E 'importante, quindi, che le idee melodiche siano astratte per permettere la sostituzione di alcuni elementi con altri, in modo che si favorisca la produzione di nuovi risultati. Se l'astrazione è troppo grossolana, tuttavia, la melodia può perdere coerenza.

3.2 Generazione di Licks

Nella generazione di un assolo jazz lo scopo è quello di creare novità e al contempo seguire alcuni orientamenti strutturali e armonici. Sebbene l'astrazione melodica abbia evidenti usi nell'analisi, essa risulta molto utile anche nella generazione di patterns. Ad esempio, si può generare una melodia da producendo prima un'astrazione di tale melodia, successivamente *istanziando* quell'astrazione ad una melodia eseguibile. Tale approccio viene elaborato così come descritto nel paragrafo successivo.

Symbol	*Color*	*Meaning*
C	black	**Chord** tones of the current chord
L	green	**Color** tones (chord extensions)
H	——	Either a chord tone or a color tone ("**Helpful**" tones)
A	blue	Tones that chromatically **Approach** one of the above
——	red	Tones that are neither C, L, H, nor A
X	——	Arbitrary tone
R	——	Rest

Tabella 3.1: Simboli terminali della grammatica

3.2.1 Categorie di Note

L'approccio grammaticale per la creazione di melodie jazz mira a trovare un equilibrio tra la novità e la coerenza aumentando le categorie di note espresse da Keller e Morrison, che corrispondono ai concetti fondamentali di note e tensioni della musica jazz. Queste categorie sono istanziate probabilisticamente ed anche nel rispetto di altri vincoli, come ad esempio le considerazioni sul range al momento della generazione. Un esempio tipico può essere quello di un chitarrista che improvvisa favorendo le posizioni sulla tastiera tra loro vicine. Un legame e una coesione imprescindibile se si vuole ottenere, oltre ad un "sound" interessante, anche una semplicità di esecuzione. Ogni categoria, come indicato nella tabella 1, ha un simbolo terminale corrispondente nella grammatica e quattro di loro sono identificati da colorazioni diverse (sulla testa della nota sul pentagramma) per fornire all'utente una maggiore chiarezza. Una grammatica genera un elenco di simboli terminali dai quali vengono poi generata una melodia. Ogni nota corrisponde ad un simbolo terminale, ma il simbolo terminale specifica solo la categoria della nota - come descritto in precedenza - e la sua durata. La nota attuale mantiene la durata, ma il pitch è generato probabilisticamente e con l'attenzione rivolta all'intervallo tra essa e la nota precedente. Per esempio, in alfabeto terminale, **C4** rappresenta un nota appartenente all'accordo di durata di un quarto, **L4/3** è un tono di colore (nota di colore[2]) di durata un quarto e una terzina, **A8** rappresenta un tono di approccio di durata di un ottavo, **H4** una nota puntata di un quarto che può essere una nota appartenente all'accordo o una nota di colore, **R2** un nota della durata di 1/2 da considerarsi resto (ovvero non appartenente a nessuna delle categorie precedentemente definite.

3.2.2 Melodie Astratte

E' ragionevole considerare una sequenza di simboli terminali nella grammatica come una *melodia astratta*, nel senso che le melodie multiple si adattano alla sequenza

---

[2] Per nota di colore si intende una nota che non appartiene all'accordo, ma che è complementare ad esso ed ha una qualità sonora. Le note di colore vengono definite anche come *"tensioni"* (o "*estensioni*") se si guarda alla loro relazione rispetto alla tonica dell'accordo.

quando le categorie delle note sono istanziate ai pitch corrispondenti. Un altro vantaggio di queste astrazioni melodiche è che possono essere istanziate su qualsiasi

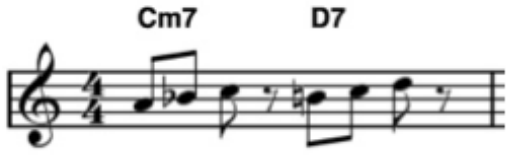

Fig. 3.1: una realizzazione di espressioni a simboli (S-expression: ▲1 2 H8 H8). Interpretazione: gruppo di 3 note da 1/8 che sono note dell'accordo o di colore. Ogni nota è separata in pitch dalla precedente di almeno un semitono e al più un tono.

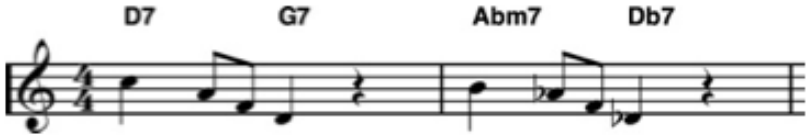

Fig. 3.2: 2 esempi di realizzazioni (S-expression: ▲-3 -4 C4 H8 H8 C4). L'interpretazione dell'espressione è di una serie discendente di una nota dell'accordo con durate 1/4, 2 note ausiliarie "helpful notes" da 1/8 e infine una nota dell'accordo da 1/4. La separazione minima è di tre semitoni (ovvero un tono e mezzo sotto) fino ad un massimo di 4 semitoni (ovvero due toni). Si ricorda che per "Helpful" si indica sia una nota dell'accordo che una nota di colore.

progressione di accordi, anche per gli accordi di diversi tipi, come *maggiore, minore, diminuita, dominante, ecc.* Anche se le singole categorie di note possono essere utilizzate per generare melodie jazz più che convincenti, al fine di catturare stili specifici, è necessario introdurre uno o più meccanismi per garantire una maggiore coerenza tra le singole note. Quindi si estendono le singole categorie di nota con dei "macros" in grado di catturare sequenze di note in certi patterns. Il lavoro si concentra su un singolo concetto macro, chiamato ***slope*** (pendenza). Ciascun *slope* ha due parametri numerici, seguiti da una sequenza di uno o più simboli terminali. I parametri numerici indicano l'aumento minimo e massimo tra note successive nella sequenza. Ovvero un range. I numeri negativi indicando una caduta, una pendenza discendente piuttosto che ascendente. Nelle regole grammaticali gli *slopes* sono trattati come simboli terminali che compaiono in conseguenza di una produzione. Nel generare una melodia, i simboli terminali all'interno di una pendenza vengono

convertiti in note specifiche, come prima.

Le espressioni di simboli (*S- expressions McCarthy 1960*) vengono utilizzate nella notazione della grammatica per fornire il raggruppamento di note in una sequenza e per la gerarchia, quando necessario . Per semplicità la parola “slope” da ora in avanti viene indicata con il simbolo ▲. Le Figure da 3.1 a 3.4 forniscono alcuni esempi di S-expressions per gli slopes, includendo nella didascalia l'interpretazione di tali espressioni. Non è sempre possibile obbedire i vincoli sia della pendenza che della categoria a cui la nota appartiene, pertanto il tool deve dare priorità ad uno piuttosto che ad un altro. Gli slopes sono unidirezionali e quindi non di per sé sufficienti a rappresentare tutti gli idiomi di interesse. Ad esempio, si consideri il linguaggio ***bebop*** di una melodia data in cui un tono appartenente ad un accordo viene avvicinato da note sopra e sotto ad esso vicine[3]. Esso richiede due slopes per rappresentare il tipo di contorno, così come mostrato in **Figura 3.3** . La notazione degli accordi seguirà la notazione jazz Lead Sheet (spartiti jazz) per le abbreviazioni, come indicato nella **Tabella 2** . Oltre agli idiomi brevi, il tool è in grado di catturare le selezioni più grandi, come in Figura 5.5, tratto dall’assolo di *Red Garland* su *Bye Bye Blackbird (Davis 1961)* . Note come il G# - nella prima misura della melodia originale nella Figura 5.5 - iniziano un segmento ascendente, così che hanno solo un intervallo da cui scegliere una pendenza minima e massima. In tali casi, estendere i limiti di semitoni di un mezzo tono per ogni step tra il limite inferiore e superiore porta a risultati migliori. Di conseguenza l’espressione (▲ -9-9 A16) diventerà (▲ -8-10 A16) prima di creare un'istanza per una melodia astratta.

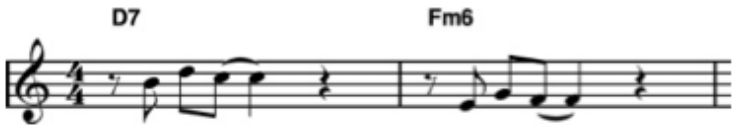

Fig. 3.3: Due esempi di realizzazione (S-expression: (R8 L8 (▲ 3 5 H8) (▲-2 -1 C4)). L’interpretazione di questa espressione è un resto, seguito da una nota di colore da 1/8, seguita da una nota ausiliaria (dell’accordo o di colore) della durata di 1/8 che può essere da 3 a 5 semitoni sopra la precedente. Nella battuta successiva si avrà invece una nota dell’accordo da 1/4 che scenderà di un tono o di un semitono. Questo è dunque un esempio di “ecnlosure” (Baker 1998).

[3] Tale approccio, spesso cromatico, fu definito da Baker (1998) come *enclosure*

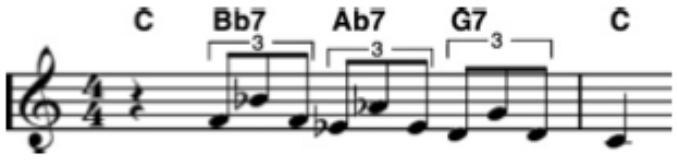

Fig. 3.4: Un esempio di realizzazione (S-expression: (R4 (▲5 5 C8/3 C8/3) (▲-5 -5 C8/3) (▲-5 -5 C8/3) (▲-2 -2 C8/3) (▲5 5 C8/3) (▲-5 -5 C8/3) (▲-1 -1 C8/3) (▲5 5 C8/3) (▲-5 -5 C8/3) (▲-2 -2 C8/3))
Interpretazione: si tratta di un idioma standard nel jazz, contenente terzine discendenti diatonicamente per note dell'accordo, con specifici intervalli.

Symbol	*Meaning*
M	major
m	minor
m7	minor seventh
7	dominant seventh, if by itself
6	added sixth

Tabella 3.2: I simboli per identificare gli accordi

Poiché le note dell'accordo (es. C4) svolgono il ruolo più importante nel dare forma alla melodia, essi avranno priorità rispetto ai limiti di pendenza, ma non per le categorie di nota diverse dalle note dell'accordo (es. H4). Al di sopra della melodia vi è una serie di segmenti che mostrano il profilo qualitativo della linea originale. La didascalia della *Figura 3.5* mostra la melodia astratta derivata. Le *Figure 3.6 e 3.7* illustrano due melodie generate da quella melodia astratta . Al contrario, la Figura 5.8 mostra cosa succede quando si usano solo le categorie di nota e si ignorano le informazioni di contorno codificato come pendenze, mentre la Figura 9 mostra tali informazioni utilizzando gli *slopes*, ma non considerando le categorie di note . Nell'ultimo esempio, vengono generate più note fuori armonia (note cerchiate). Se non c'è nessun tono del tipo dato disponibile all'interno dei limiti di pitch, si usa una tabella di probabilità per scegliere un'altra nota. Ciò potrebbe comportare un'estensione dei limiti di Pitch (di solito quando si è alla ricerca di note dell'accordo) o selezionando una nota di tipo diverso all'interno dei limiti (di solito per tutte le note tranne quelle dell'accordo). Lo scopo di tali espedienti è quello di evitare l'introduzione di un sistema di vincoli nell'implementazione.

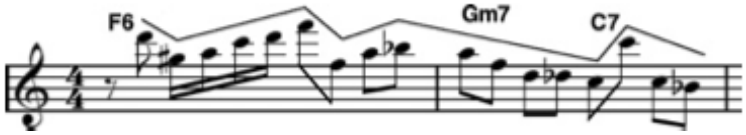

Fig.3.5: Una linea melodica originale evidenziata dal contorno al di sopra delle note. Una melodia astratta (rappresentazione dello slope) può essere estrapolata grazie alla S-expression: (R8 C8 (▲-6 -6 A16) (▲1 3 C16 C16 C16 C8) (▲-12 -12 C8) (▲2 4 C8 L8) (▲-4 -1 L8 C8 C8 A8 C8) (▲12 12 C8) (▲-12 -2 C8 C8)).

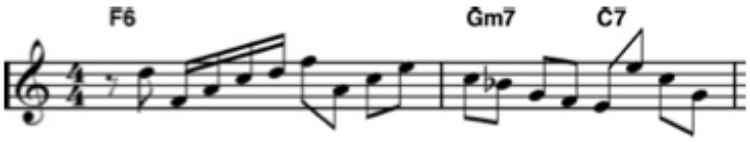

Fig.3.6: Una linea melodica generata dalla S-expression di Figura 3.5, usando gli slopes.

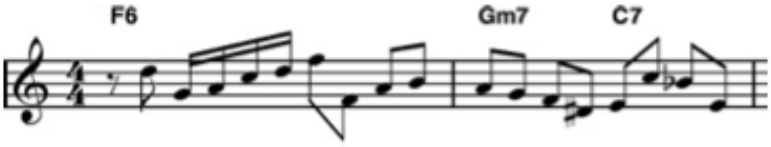

Fig.3.7: Un'altra linea melodica generata dalla S-expression di Figura 3.5, usando gli slopes.

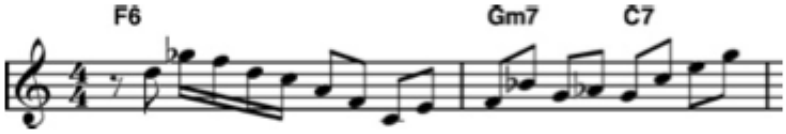

Fig.3.8: Una variante rispetto alle precedenti, questa volta la melodia generata dalla S-expression di Figura 3.5 utilizza soltanto le categorie di note e non più gli slopes.

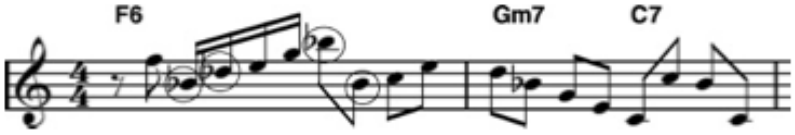

Fig.3.9: Una melodia generata dalla S-expression di Figura 3.5 che utilizza soltanto gli slopes e non le categorie di note. Le note cerchiate non sono né note dell'accordo, né note di colore.

3.3 Grammar Learning

All'interno del tool, degli algoritmi di inferenza grammaticali tentano di definire le regole di una grammatica di simboli per una lingua sconosciuta; ciò viene implementato attraverso l'analisi di un set di dati di *training*. I dati possono contenere entrambi i campioni positivi (stringhe nel linguaggio) ed i campioni negativi (le stringhe che non devono essere accettate). Nel dettaglio si usano solo set di campioni positivi, ad esempio, le prestazioni di un artista che si sta tentare di imitare. La grammatica, poi, dovrebbe generare stringhe corrispondenti alle melodie simili a quelle nei corpi di formazione e idealmente non dovrebbe generare nulla che differisca notevolmente da quest'ultime. Esistono diversi metodi per estrarre le regole grammaticali dai dati di *training*, compresa l'estrazione per frasi, con una frase definita come una sezione di un solo che inizia dopo una pausa (ad esempio in levare) e termina a sua volta con una pausa. Data la lunghezza variabile delle frasi e la difficoltà di ricombinare le frasi in un assolo di una dimensione specifica, ecco che si è optato per dividere le melodie in *finestre di tempo* di lunghezza predefinita. Dopo aver scelto due parametri, *il numero di battiti per finestra* e il *numero di battiti da cui far scorrere la finestra* (che è necessariamente minore o uguale alla dimensione della finestra), si raccolgono tutti i frammenti melodici di una certa lunghezza in un corpus, associando un simbolo terminale della grammatica per ogni melodia astratta della lunghezza data. Si ottiene dunque, insieme a frammenti tra 1 e 8 battiti. Nel caso di un tempo 4/4 i frammenti di 4 battiti raggiungono il miglior bilanciamento tra originalità e continuità, dato che sono i più comuni.

3.3.1 Grammar.Java e LickGen.java

Le due classi *Grammar.java e LickGen.java* implementano i metodi e le funzionalità necessarie alla generazione di melodie astratte. La prima definisce una grammatica di simboli non-deterministica che viene generata inserendo, all'interno di oggetti PolyList di array, tutte le "regole possibili" e, successivamente, implementandole con una distribuzione cumulativa di probabilità dando un peso a ciascuna di esse e scegliendo in modo "random" un percorso da seguire tra le regole possibili.

```
//      Oggetti per la modellazione di regole
      // Tutte le regole applicabili (e i loro pesi
corrispondenti)
      // vengono istanziate in questi array.
      Vector<Polylist> ruleArray = new Vector<Polylist>(5);
      Vector<Polylist> baseArray = new Vector<Polylist>(5);
      Vector<Double> ruleWeights = new Vector<Double>(5);
      Vector<Double> baseWeights = new Vector<Double>(5);
```

Fig. 3.10: Gli oggetti Polylist dove vengono istanziate tutte le regole.

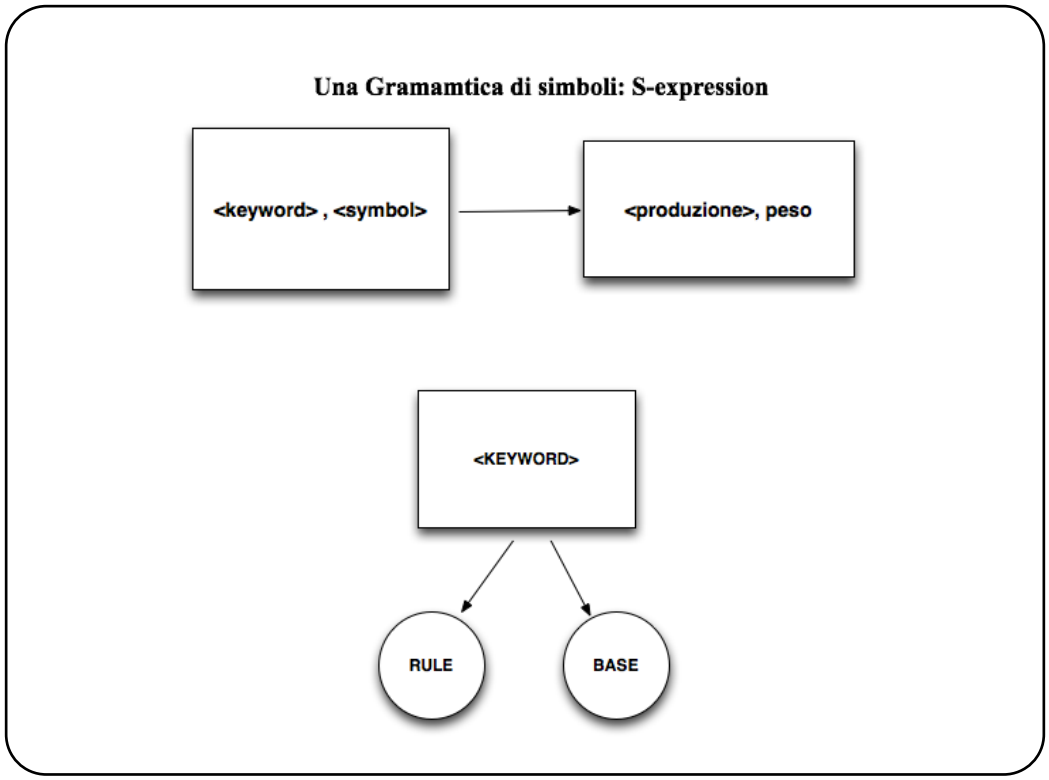

Fig.3.11 La grammatica di simboli probabilistica (S-expressions)
<keword><symbol> → <production> <weight>

La Grammatica viene poi caricata nella classe ***LickGen.java*** nella quale vengono generati i patterns (o *licks*) improvvisativi. La grammatica viene definita con un'espressione di simboli come mostrato in *Fig.3.11*: ogni simbolo associato ad una keyword identificativa genererà una produzione. A quest'ultima verrà associato un peso, ovvero maggiore sarà il peso e maggiore sarà probabilità che quel simbolo sia presente nella generazione melodica finale. Una *Keyword* può essere o una regola o una base (*RULE o BASE*). *<symbol>* può essere una stringa o una Polylist di simboli.

Una produzione <*production*> è tipicamente una PolyList di simboli (oppure se è una RULE, si continua con la riduzione). Il peso associato a ciascuna produzione viene espresso da: <*weight*> , un double che esprime *l'importanza* della regola. Le regole più importanti allora saranno quelle scelte più spesso. La *keyword BASE* stoppa tutte le valutazioni e le sostituzioni di variabili. Se un simbolo va bene sia come RULE che come BASE, verrà data priorità alla BASE. Tale approccio consente di ridurre i calcoli e di trovare i casi base con un modo di risoluzione semplice. Dopo aver generato la grammatica ***LickGen*** caricherà al suo interno questi simboli e inizierà a "filtrarli", fino al raggiungimento di simboli terminali che genereranno una melodia astratta. Vi sono metodi per la generazione di *figurazioni ritmiche* che vengono poi riempirle con le note secondo una certa grammatica (tra questi *fillMelody (), fillRythm()*, etc.) ed anche classi e metodi per la scelta delle note opportune secondo le regole prestabilite (ad esempio ***NoteChooser.java*****,** dove viene fatta una distinzione tra le note appartenenti all'accordo, note della scala, note di colore, ecc. così come precedentemente definito in apertura di capitolo - *Tabella 1*).

```
// Stampa i valori terminali.
      // Se l'utente ha provveduto con una lista di terminali usali,
       altrimenti:
      // assumiamo che i valori terminali siano stringhe che
iniziano una lettera in "minuscolo"

      if( !terminals.isEmpty() )
        {
        //System.out.println("pop = " + pop);

        while( (pop.first() instanceof String &&
                terminals.contains((String)(pop.first()))) )
          {
          if( pop.first().equals("slope") )
            {
            terminalString = terminalString.cons(pop);

           }
          else
            {
            // prendi il primo elemento, assumi che pop sia un
singleton
            terminalString = terminalString.cons(pop.first());
            }
```

Fig.3.12 Inizio riduzione della grammatica

3.3.2 Markov Chains

Per migliorare la continuità tra i frammenti melodici e per aumentarne la coesione, una volta che abbiamo raccolto le melodie astratte (classe LickGen.java) esse andranno a comporre i nostri assoli e dobbiamo combinarle tra di loro grazie ad una catena di Markov [4] [30] [31] implementata all'interno della grammatica. Le catene di Markov rappresentano un sistema con una sequenza di stati, con probabilità condizionali per modellare le transizioni tra stati successivi. Poiché le grammatiche sono già probabilistiche, le catene di Markov si possono inserire all'interno della grammatica più o meno naturalmente. Una catena di Markov ad **n-grammi** (*n-gram Markov Chain*) lavora come un predittore a memoria uno, ovvero usa una probabilità condizionata dal precedente $n - 1$ stato. L'insieme delle melodie astratte servono per rappresentare gli stati della catena di Markov. Data una melodia di partenza, si aggiunge la seguente frase sulla base di un elenco di probabilità di transizione dalla prima battuta.

3.4 Clustering

Per aumentare la varietà nella generazione di idee e patterns melodici, si *raggruppano* melodie astratte simili con **l'algoritmo di clustering K-means** [31]. Esistono poi delle statistiche che rappresentano i cluster che seguono altri cluster nel corpus e costruiscono la tabella di probabilità come conseguenza, utilizzando i raggruppamenti (clusters) come stati della catena di Markov. Per comporre nuovi soli, dapprima si genera una sequenza di clusters dalla grammatica, successivamente si scelgono casualmente quelli più rappresentativi (*representatives*) dai clusters, nuovamente usando le regole grammaticali per specificare la distribuzione dei *representatives*.

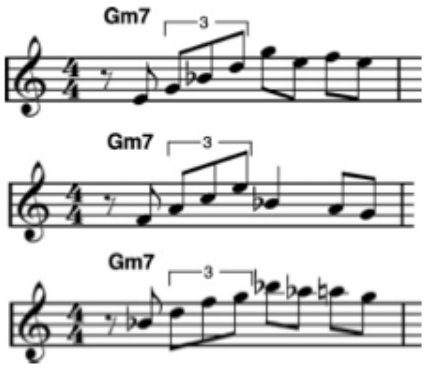

Fig. 3.13: Tre melodie simili che vengono istanziate dai representatives dentro un cluster (Corpus di solo di Charlie Parker)

Gli algoritmi di clustering rappresentano i dati come punti in un piano n-dimensionale e raggruppano i punti attraverso una certa distanza metrica. L'analisi di cluster del tool ImproVisor è basata su una misura di distanza euclidea su sette parametri:

1. Numero di note nella melodia astratta.
2. Posizione della prima nota che inizia dentro la battuta.
3. Durata totale delle pause.
4. Pendenza media massima di gruppi di note ascendenti/discendenti.
5. Verifica se la nota inizia in battere o levare all'interno della battuta.
6. Ordine del contorno (quante volte cambia direzione)
7. Consonanza (intervalli consonanti, es.: di quarta, di quinta. In contrapposizione a quelli dissonanti).

La *pendenza massima media* (quarto parametro nella lista) è ottenuta segmentando la melodia astratta in sequenze di note che sono tutti ascendenti o discendenti, calcolando: le pendenze (slopes) tra ciascuna coppia di note adiacenti all'interno di ogni sequenza, prendendo il valore assoluto di ciascuna pendenza, trovando il valore massimo assoluto all'interno di ogni sequenza e poi la media dei valori massimi assoluti di tali sequenze all'interno della melodia astratta.

Il valore di *consonanza* (parametro 7) è assegnato a una battuta in base alle categorie di nota. Per ogni nota si aggiunge, al valore di consonanza, un coefficiente per la categoria della nota moltiplicato per la durata della nota. Per esempio, coefficienti tipici sono 0,8 per una nota appartenente all'accordo, 0,6 per una nota d'approccio, 0.4 per una nota di colore, e 0,1 per le altre note. Dato un parametro k per il numero di raggruppamenti (clusters), l'algoritmo k-means seleziona k punti come centri di cluster (centroidi) e quindi inizia un processo iterativo dato dai seguenti due passaggi:

1. Assegna ogni data point al centro di cluster più vicino.
2. Ricalcola i nuovi centroidi.

Questi passaggi vengono ripetuti per un certo numero di iterazioni o fino ad ottenere abbastanza pochi dati points come cluster di scambio tra le iterazioni. In linea di principio, l'utente può impostare il valore di k. Tuttavia, attualmente si usa come

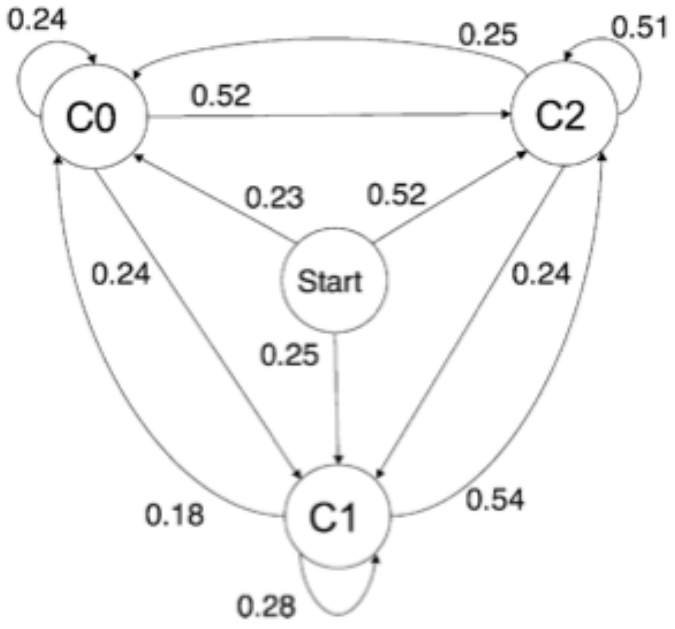

Fig. 3.14 Markov Chains: esempio di probabilità di transizione dopo il clustering.

Production Rule	*Probability*
Start(Z) → C0(Z)	0.23
Start(Z) → C1(Z)	0.25
Start(Z) → C2(Z)	0.52
C0(0) → ()	1
C1(0) → ()	1
C2(0) → ()	1
C0(Z) → Q0 C0(Z-1)	0.24
C0(Z) → Q0 C1(Z-1)	0.24
C0(Z) → Q0 C2(Z-1)	0.52
C1(Z) → Q1 C0(Z-1)	0.18
C1(Z) → Q1 C1(Z-1)	0.28
C1(Z) → Q1 C2(Z-1)	0.54
C2(Z) → Q2 C0(Z-1)	0.25
C2(Z) → Q2 C1(Z-1)	0.24
C2(Z) → Q2 C2(Z-1)	0.51
Q0 → ((Δ 0 0 R2 R4 R8 C16/3) (Δ 1 1 A16/3 L16/3)	1
Q1 → ((Δ 0 0 C8) (Δ -9 -9 C8) (Δ 2 3 C8 G4+8 R4))	1
Q2 → ((Δ 0 0 C4/3) (Δ 1 2 L4/3 A4/3) (Δ -7 -1 C4/3 G4 C8/3))	1

Tabella 3.3: Grammatica Probabilista che Implementa una Markov Chain

valore di k il numero di frammenti melodici divisi per 10 come un valore nominale empirico. La Figura 5.10 mostra tre melodie rappresentative della durata di una battuta. Esse sono estratte da un corpus di assoli di Charlie Parker. L'algoritmo prima estrae queste melodie e quindi raggruppati insieme le corrispondenti melodie astratte. Dunque quando si definisce la grammatica viene invocato il Clustering in concomitanza con l'implementazione delle Markov Chains.

3.4.1 Probabilità di Transizione

Le probabilità di transizione della catena di Markov sono basate vedendo i clusters come stati. Nei dati di training le melodie reali sono mappate in melodie astratte che vengono quindi a loro volta mappate in clusters così come descritto in precedenza. A questo punto si ricostruiscono i dati di *training* e il conteggio (count), per ogni coppia

di cluster A, B quante volte una melodia dal gruppo B segue una melodia dal gruppo A. La *normalizzazione* di questi conteggi ci dà le probabilità di transizione tra i clusters. La Figura 11 illustra il risultato per i clusters derivati da un piccolo corpus.

3.4.2 - Rappresentazione della Grammatica

Nella grammatica si codifica la transizione delle probabilità come conseguenza del clustering. In effetti, le probabilità di transizione generano una sequenza di raggruppamenti e quindi una melodia astratta è scelta da ogni cluster per essere istanziata ad una vera e propria melodia.
La Tabella 3.3 mostra una semplice grammatica probabilistica corrispondente alle transizioni nella Figura 11. Si noti che ogni simbolo non terminale ha un argomento che indica il numero di battute da riempire. Questo è utilizzato solo per controllare l'espansione della grammatica e riempire una certa quantità di spazio temporale. Tale numero è rappresentato dal numero di battute specificato come argomento per l'inizio della produzione (START). I simboli *non-terminali* **Start, C0, C1, C2** rappresentano gli stati della catena di Markov. Le regole vengono espresse con **Q0, Q1, Q2** e sulla loro destra (produzione grammaticale) mostrano le scelte derivate dai raggruppamenti (clusters). Per brevità, si includono solo una delle melodie astratte rappresentative per ciascun Q0, Q1 e Q2 (soprattutto considerando che in genere, in un file grammaticale, definito come *.grammar*, il numero di tali regole è superiore alle 200 unità). Per facilitarne la leggibilità si trascrive la notazione dell'implementazione da S-expressions a regole grammaticali più convenzionali.
La grammatica presente in Tabella 5.3 è per una catena di Markov del primo ordine. Per una catena di ordine n, dove $n \geq 2$, gli stati sarebbero stati etichettati da sequenze di n-1 indici di cluster. Questi corrispondono alle sequenze di cluster che possono effettivamente verificarsi nel corpus.

3.4.3 - Produzione finale della Grammatica

La stringa terminale prodotta dalla grammatica è una *melodia astratta,* creata concatenando insieme, per ogni cluster, una melodia astratta rappresentativa. I rappresentanti (*rappresentatives*) sono attualmente scelti per una distribuzione

uniforme. Viene quindi generate una vera e propria melodia selezionando in modo casuale una nota iniziale della categoria di nota specifica e poi riempiendo il resto dallo slope e dai vincoli delle note di categoria.

3.4.4 - Riassunto dell'algoritmo: analisi e sintesi

Il seguente paragrafo riassume l'approccio che l'algoritmo utilizza a partire dalla fase di **analisi**, creando una grammatica da un corpus di assoli su progressioni di accordi, seguita dalla **sintesi**, che crea nuovi assoli su progressioni di accordi possibilmente diverse tra loro. Ci saranno molte sintesi effettuate per una determinata fase di analisi, dato che gli utenti avranno il modo di utilizzare principalmente grammatiche sintetizzate da sé o per altri.

- Parte di ***analisi***, risultati in una grammatica:

1. Dividere il corpus di trascrizioni in frammenti melodici (finestre di tempo), in genere la lunghezza è di una battuta.
2. Tradurre ogni frammento in una melodia astratta fatta di slopes, categorie di note e ritmi.
3. Eseguire un algoritmo di clustering sulle melodie astratte, raggruppando tutte le melodie astratte nei clusters, tipicamente si hanno - in media - 10 melodie astratte per cluster.
4. Riesaminare ogni trascrizione nei corpus per determinare la sequenze in cui i clusters appaiono.
5. Raccogliere le statistiche *n-gram* sui clusters, tipicamente per *n* compreso tra 2 e 4 (scelto dall'utente).
6. Creare una grammatica probabilistica che generi sequenze di clusters basati sui dati *n-gram*.

- La parte di ***sintesi melodica*** che usa una grammatica può essere riassunta come segue:

1. Dato una leadsheet (spartito) con una progressione di accordi ma nessuna melodia, l'utente seleziona una sezione della melodia su cui generare un solo.

2. Le regole grammaticali si espandono, utilizzando le probabilità di produzione, finché viene generata una sequenza di cluster corrispondente al numero di battute desiderato.

3. Da ogni cluster in sequenza scegliere a caso una melodia astratta e concatenare le melodie astratte.

4. Tradurre le melodie astratte in musica per selezionare probabilisticamente le note che rispettano i vincoli di *slopes* e di *note di categoria* come progettato per la melodia astratta.

5. (Facoltativo): A discrezione dell'utente, la melodia rumori elettrici può essere corretta automaticamente, tirando le note non classificate come accordo, colore, o un approccio in linea con l'armonia per una correzione di mezzo tono. Ciò corregge eventuali conflitti tra le specifiche qualità delle note astratte e le caratteristiche di pendenza.

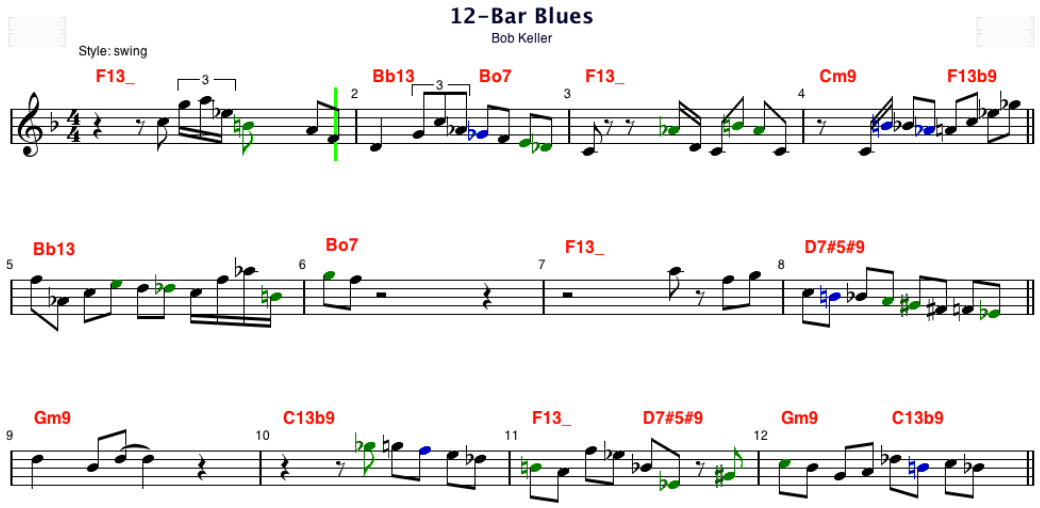

Figura 3.15: Una improvvisazione generata (sfruttando la grammatica di Charlie Parker) su una progressione blues di 12 battute.

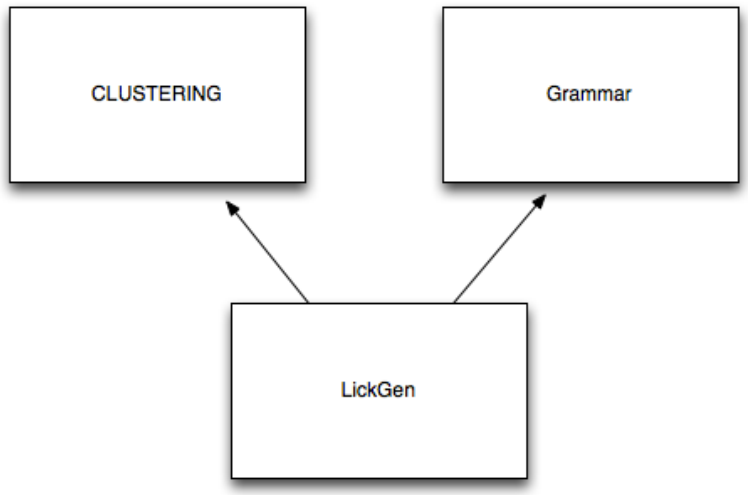

Figura 3.16: Diagramma delle classi - LickGen eredita la grammatica dalla classe Grammar.java. Le melodie astratte vengono poi raggruppate e selezionate dall'algoritmo di Clustering.

Figura 3.17: Diagramma UML tra le classi Grammar.java, LickGen.java e Cluster.java (per semplicità sono stati omessi nel diagramma molti dei metodi implementati all'interno delle classi).

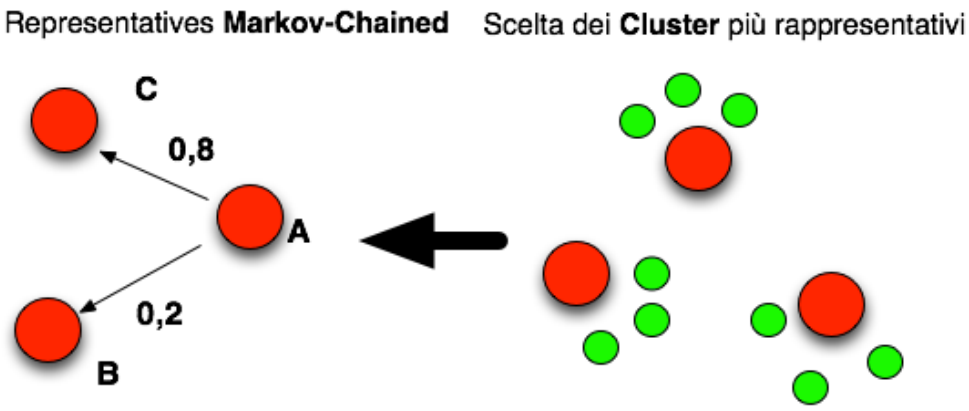

Fig.3.18: La relazione tra il Clustering e le catene di Markov

Capitolo 4 - Piattaforma Android

Android: Architettura e componenti applicativi

Questo capitolo fornisce una panoramica sulla programmazione *Android [32]*. Viene offerta un'introduzione, compatta ed efficace, al fine di comprendere meglio l'implementazione che verrà discussa più avanti. Dopo aver descritto le caratteristiche, gli algoritmi per l'analisi (RoadMap), per l'improvvisazione melodica (ImproVisor) di progressioni di accordi Jazz, inizia dunque, così come preannunciato nell'introduzione, l'obbiettivo di raggruppare le funzionalità di cui sopra in un'unica applicazione mobile. Quest'ultima dovrà mantenere l'interattività e l'utilità dal punto di vista didattico ma al contempo aggiungere dei vantaggi, primo fra tutti quello della portabilità. Occorre soffermarsi innanzitutto sui problemi che si incontreranno nel voler programmare in Android.
Dopo un primo sguardo al sistema in questione, si proseguirà con una descrizione dell'architettura e delle componenti strutturali principali dello stesso.
Verranno indicate le caratteristiche e definiti i ruoli di ciascuna di essa; partendo dal concetto di *Activity*, di *Intent*, di *AndroidManifest*, si tratterà delle interfacce utente e il loro utilizzo durante la creazione di un progetto Android. Un'occhio di riguardo verrà dedicato alle configurazioni necessarie prima di poter programmare e al ruolo di due classi significativa importanza *(R.java* e *R.class*). L'implementazione sarà discussa nel Capitolo 5, iniziando con la creazione di un *Server/Client* che possa avviare una comunicazione per la richiesta e l'invio dei dati. A questo punto lo scoglio più grosso da superare resta quello di stabilire un formato comune di scambio, dato che *Android* non supporta le librerie grafiche *java.awt e java.swing*. Esse dovranno essere ricreate in toto.

4.1 Introduzione ad Android

Android è una piattaforma per dispositivi mobili gestiti da *Google* e sostenuti dalla Open Handset Alliance. La piattaforma Android comprende un framework applicativo, un sistema operativo, una Java Virtual Machine chiamata *Dalvik*, un browser web basato su WebKit, database SQLite, servizi middleware e applicazioni chiave .L'*Android SDK* fornisce gli strumenti e le API necessarie per iniziare lo sviluppo di applicazioni su piattaforma *Android* utilizzando il linguaggio di programmazione Java. Tuttavia, le API Java supportate da *Android* non sono standard come le Java ME API(JSR), quindi, al fine di sviluppare un'applicazione *Android* (e tanto più effettuare un *Porting* da parti di Software java esistenti), lo sviluppatore deve imparare e far fronte alle diversità strutturali e alle nuove API da gestire. *Android* è basato sul *kernel di Linux* e utilizza una macchina virtuale personalizzata , comunemente indicata come *Dalvik* VM. Sebbene il kernel di Android sia basato su Linux , gli sviluppatori possono utilizzare solo la sintassi di programmazione Java per sviluppare applicazioni Android. Applicazioni *Java Micro Edition (Java ME)* non sono compatibili con *Android* a causa del nuovo formato dei codici di byte utilizzati dalla macchina virtuale *Dalvik*; dunque gli sviluppatori devono riscrivere le applicazioni *Android* per far si che queste possano funzionare su dispositivi moderni (es. Galaxy, tablet) o altri dispositivi che supportano standard di Java ME.

4.1.1 L'Architettura Android

L'architettura di Android è sostanzialmente divisa in diversi componenti: Application Framework, Librerie, Runtime Android e kernel Linux. Come accennato in apertura di capitolo, Android ha al suo interno una macchina virtuale chiamata *Dalvik* e ogni applicazione Android verrà eseguita sul proprio processo, cioè, utilizzando la propria istanza di Dalvik JVM. La Dalvik JVM esegue file chiamati eseguibili (Dalvik, File DEX), una sorta di codice Java byte ottimizzato per la piattaforma Android. Il kernel Linux è basato sulla versione 2.6. Infine, il codice Android può utilizzare la versione 1.5 del linguaggio Java (Tiger), quindi tutte le caratteristiche di Java Tiger sono disponibili per Android. Significa che è possibile utilizzare i *Generics*, migliorati per i loop, *auto boxe* e altre nuove funzionalità del linguaggio nello sviluppo di software Android.

4.2 Programmare in Android (Application Development)

Le principali componenti di un'applicazione Android sono: *Activity, IntentReceiver, Service e ContentProvider*. A seconda del tipo di applicazione che potrebbe svilupparsi, non tutti i componenti saranno presenti in un'applicazione. Riassumendo, essi sono definiti come segue:

1. Un'*Activity* ha i principali metodi di callback ed è collegata (rappresentata) ad una schermata vuota in modo logico (un'applicazione può avere diverse classi di attività, dunque diverse Activities);
2. Una *Intent* rappresenta un'azione e viene descritta come <azione> ed elementi XML <category>. Vi sono poi gli *IntentReceiver* e *IntentFilter* (elemento <intent-filter>) che si prenderanno cura della trasformazione da intento ad azione e garantiranno che sarà processata l'operazione prefissata;
3. AndroidManifest.xml: i componenti di cui sopra sono controllati tramite il file di configurazione AndroidManifest.xml, che è una sorta di descrittore per l'applicazione Android. Così come indicato in *Fig.4.1* al suo interno infatti vi sono le indicazioni relative alle varie Activities, la versione del codice (riga 4,5), i

permessi (righe 11-14), i targets a cui si riferisce (righe 8-9, target minimi e massimi relativi ai prerequisiti minimi di sistema che gli smartphone/tablet dovranno avere), il nome dell'applicazione (riga 19).

4.2.1 Interfaccia Utente (User Interface)

Per la creazione di interfacce Android utilizza principalmente due componenti: *android.view.View* e *android.view.ViewGroup*.

La classe *View* rappresenta un'area grafica sullo schermo del dispositivo e per mezzo di essa si può scoprire la larghezza e l'altezza, impostare la messa a fuoco, controllare lo scorrimento e la manipolazione chiave per l'area corrispondente. *View* dunque viene estesa da molti oggetti e diversi *Widget Android*.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="com.fausto.fausto"
4      android:versionCode="1"
5      android:versionName="1.0" >
6
7      <uses-sdk
8          android:minSdkVersion="8"
9          android:targetSdkVersion="18" />
10
11     <uses-permission android:name="android.permission.INTERNET"/>
12     <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
13     <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
14     <uses-permission android:name="android.net.conn.CONNECTIVITY_CHANGE" />
15
16     <application
17         android:allowBackup="true"
18         android:icon="@drawable/ic_launcher"
19         android:label="@string/app_name"
20         android:theme="@style/AppTheme" >
21         <activity
```

Fig. 4.1: un primo esempio di AndroidManifest.xml per l'applicazione.

```
1  package com.fausto.fausto;
2
3  import java.io.BufferedReader;
25
26 public class MainActivity extends Activity {
27     Socket socket;
28     BufferedWriter out;
29     DataInputStream in;
30     DataOutputStream dos, dos2;
31
32     private Socket s, s2;
33     private BufferedReader input, input2;
34     private OutputStream output, output2;
35     private static final int id=1;
36
37     Button btn, btn2;
38
39     Thread t1;
40
41     String riga_letta;
42
43     @Override
44     protected void onCreate(Bundle savedInstanceState) {
45         super.onCreate(savedInstanceState);
46         setContentView(R.layout.activity_main);
47
48         btn = (Button) findViewById(R.id.button_Open);
```

Fig. 4.2: un primo esempio di attività chiamata "MainActivity" per l'applicazione. Da notare l'estensione alla classe "Activity". Il metodo "protected void onCreate(Bundle savedInstanceState) {..." dichiarato nella riga 44 è il metodo principale di un'applicazione Android. La definizione dell'istanza di riga 45 conferma l'ereditarietà alla casse Activity. Infine, nella riga 48 viene definito un primo oggetto (grazie al Widget della classe Button). L'id (in forma esadecimale) viene collegato al bottone grafico definito nel layout grazie alla classe "R.id.button_Open" di cui si discuterà più avanti.

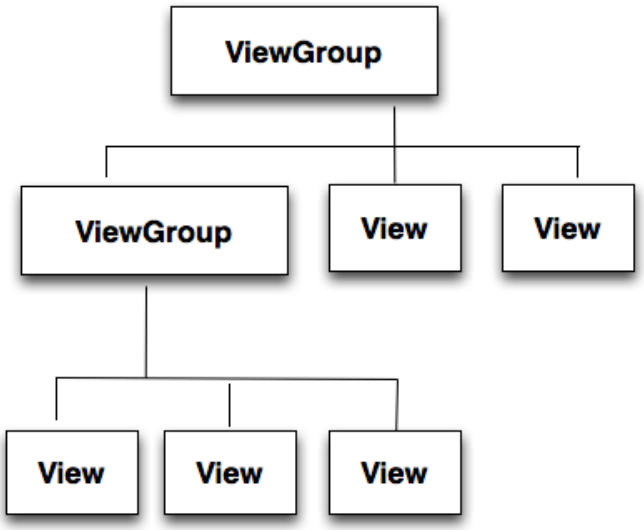

Fig. 4.2: Diagramma delle classi ViewGroup e View

	Widgets	
AbsListView	*AbsSeekBar*	*AppWidgetHostView*
Button	*Galleria*	*GridView*
FrameLayout	*LinearLayout*	*ListView*
ImageSwitcher	*ScrollView*	*TabHost*
TableLayout	*RadioButton*	*RelativeLayout...*

Tabella 4.1: Lista di alcuni tra i widgets più utilizzati

Il *ViewGroup* è un contenitore di visualizzazione, cioè, si tratta di una classe contenitore e aiuta a definire le caratteristiche del layout della GUI .
Alcuni esempi di layout sono *AbsoluteLayout , FrameLayout , LinearLayout e RelativeLayout* (Layout sia orizzontale che verticale).
La Tabella 4.1 mostra una lista dei widgets più importanti che potranno essere importanti all'interno delle proprie Activity.

4.3 Struttura del progetto

I File XML svolgono un ruolo importante nello sviluppo di applicazioni Android, soprattutto per quanto riguarda la configurazione e la dichiarazione delle risorse.
A titolo di esempio, nella programmazione di applicazioni Android, per creare la GUI, la mappatura e l'azione di navigazione e linkage generalmente l'autore utilizza alcuni file XML e riferiscono ogni componente utilizzando il suo ID.
In *Fig.4.4* abbiamo un esempio di struttura di progetto (molto basica) in vista di un progetto Android.

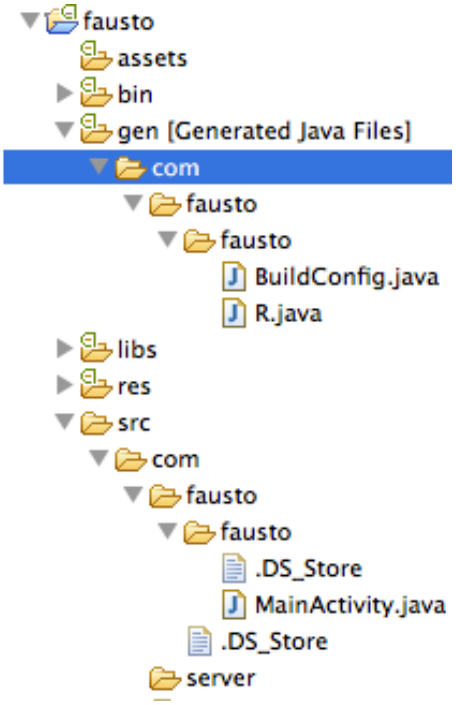

Fig.4.4: Esempio di Progetto Android all'interno dell'IDE Eclipse

Riassumendo:

1. La cartella *"src"* contiene tutte le classi create esplicitamente dallo sviluppatore. Si noti come una Classe di attività di base è presente in questa cartella, all'interno del pacchetto *com.fausto.fausto*;
2. La cartella di generazione *"gen"* ha diverse classi che vengono create automaticamente da *Android* e non dovrebbero mai essere modificate manualmente dallo sviluppatore;
3. Il file *android.jar* è incluso nel classpath e ha il framework di Android;

4. La cartella "*res"* contiene tutte le risorse utilizzate dall'applicazione, come immagini (la sottocartella *drawable* ha il file dell'icona di lancio dell'applicazione), layout (la sottocartella di layout contente il file *main.xml* che definisce il layout GUI), valori statici, tipo di risorse *bundle* (come i valori presenti nella sottocartella *values* all'interno del file *string.xml*);
5. *AndroidManifest.xml* - come citato prima, è il file di configurazione principale per l'applicazione Android.

4.3.1 R.java ed R.class

Uno degli errori tipici che, in fase di compilazione, ci viene restituito quando si inizia a programmare in Android è il seguente:

"R cannot be resolved to a variable"

In ogni applicazione android ci sono 2 classi disponibili con il nome R.
La prima classe fa parte del nucleo del sistema Android o Android SDK, che può essere letta come *android.R* Possiamo vedere questa classe nel file di *R.class* che è disponibile nel file di *android.jar* e viene automaticamente incluso nel progetto per plugin ADT (Android Developer Tools).[4]
La seconda classe è parte della nostra applicazione, che può essere letta come nome_pacchetto.R (ad esempio basandoci sull'esempio di Fig.4.4: *com.fausto.fausto.R*). Possiamo vedere questa classe nel file di *R.java* che è disponibile nella directory *gen*. Il file R.java è visibile solo dopo il successo di compilazione del progetto, dato che, al momento della compilazione, all'interno del file viene associata una stringa esadecimale per ogni elemento dell'applicazione. R.java dunque è un file generato automaticamente quando si crea un'applicazione Android. Esso contiene identificatori unici (numeri normalmente 32bit) per gli elementi di ogni categoria (*drawable, string, layout, color, ecc.*) delle risorse

[4] ADT (Android Developer Tools) è un plugin per Eclipse che fornisce una suite di strumenti che si integrano con l'IDE Eclipse. Esso consente di accedere a molte funzioni che consentono di sviluppare applicazioni Android in fretta. ADT fornisce l'accesso GUI per molti degli strumenti SDK della riga di comando, nonché serve come strumento di progettazione dell'interfaccia utente per la prototipazione rapida, progettazione e costruzione dell'interfaccia utente dell'applicazione.

(elementi sotto la directory *res*) disponibili nell'applicazione Android. Lo scopo principale del file di R.java è la rapida accessibilità delle risorse nel progetto. Se una risorsa ha cancellato o aggiunto al progetto, il file R.java sarà aggiornato automaticamente, tale processo viene fatto da plugin ADT in Eclipse.
Il plugin ADT in Eclipse darà un avviso se si tenta di modificare il file R.java.

4.3.2 La creazione di R.java

Nel file *R.java*, ciascuna categoria della risorsa verrà creata come una classe. In ogni classe di risorsa verranno creati tutti i rispettivi elementi come membri statici, ovvero come costanti. Quindi, non si dovrà cambiare questi valori perché identificati unicamente. Essi sono anche dei membri finali (*final static...*), così da potervi accedere con i loro nomi della classe; è il caso di *R.drawable.ic_launcher, R.layout.main, ecc*. Le *figure 4.5-4.6* evidenziando la creazione delle classi per ciascuna categoria di risorse.

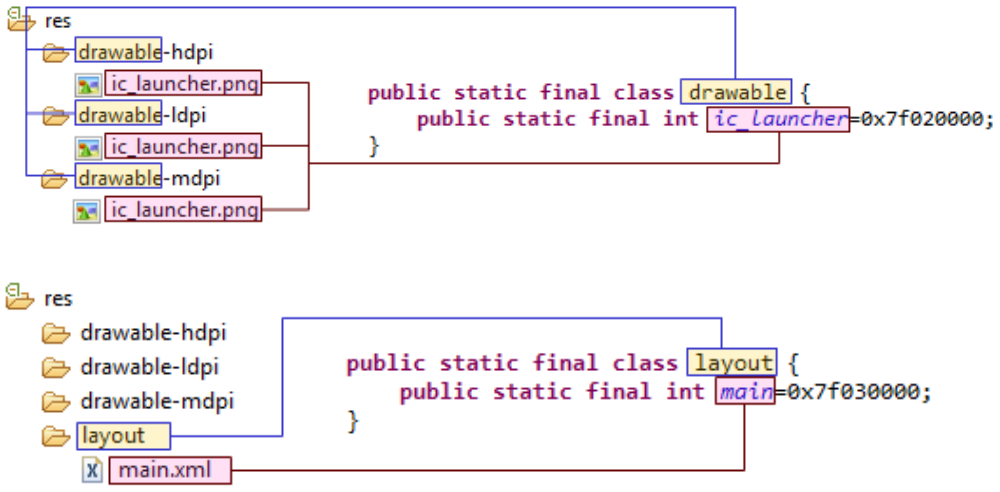

Fig.4.5: Creazione di due stringhe esadecimali per l'icona di lancio dell'applicazione "ic_launcher" e la classe di layout "main.xml".

Esiste poi, all'interno della directory *res,* una cartella di valori (*values*) che non è una categoria di risorse. Alcuni tipi di risorse, come stringhe, colori, stili, array, ecc. vengono raggruppate e nominate all'interno di *values*.

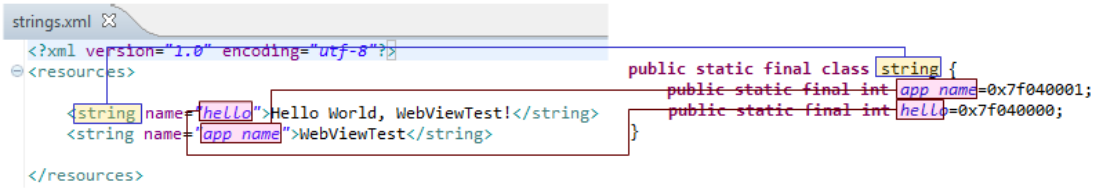

Fig.4.6: Creazione di due valori esadecimali per due stringhe "app_name" ed "hello"

4.4 Configurazione

Per sviluppare delle applicazioni eseguibili sui sistemi *Android* bisogna innanzitutto installare e configurare l'apposito kit di sviluppo (SDK) contenente al suo interno un emulatore virtuale, la documentazione e le librerie (*http://developer.android.com/sdk/*). Attualmente la versione Android disponibile è la 4.4.

La scelta dell'IDE ricade sull'ambiente integrato *Eclipse*. E' di recente pubblicazione anche un IDE apposito per la programmazione android (Android Studio). Tuttavia l'utilizzo di Eclipse è preferibile consigliato nel caso specifico dato che, specialmente in fase di compilazione, vi è un controllo sicuramente più affidabile ed un occhio di riguardo per le classi Java.

Come già anticipato, il kit di sviluppo contiene anche un utile emulatore in modo tale da consentire allo sviluppatore il test delle applicazioni su PC, prima che le stesse vengano installate su dispositivo mobile (o tablet) col formato eseguibile "*.apk*".

Non resta dunque da imparare il concetto di *Android Virtual Device (AVD)*, cioè di dispositivo *virtuale Android* (Fig. 4.7 (a)). La *Fig.4.7(b)* mostra un primo layout grafico dell'applicazione.

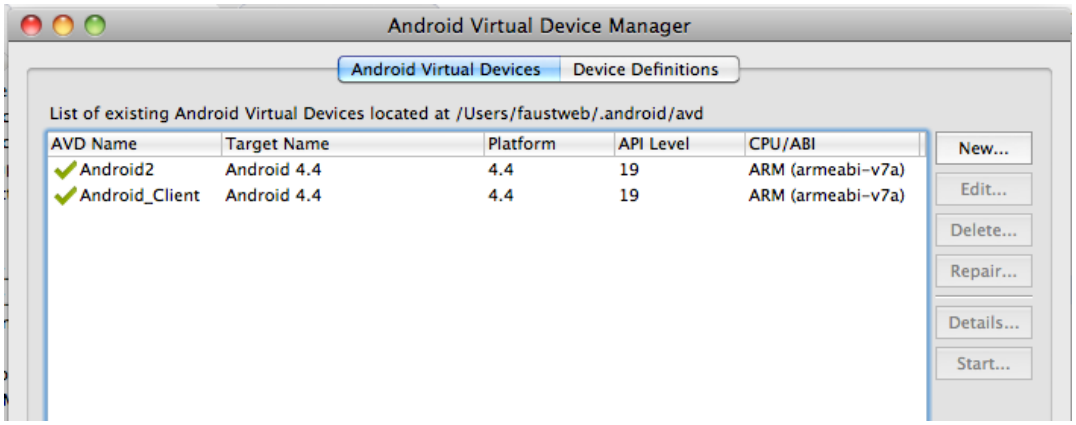

Fig. 4.7 (a): Finestra di gestione dei dispositivi virtuali di emulazione.

Fig. 4.7(b): Ecco come si presenta l'emulatore AVD all'avvio (sinistra). Sulla destra invece un primo layout grafico per l'applicazione.

Capitolo 5 - Implementazione dell'applicazione

Come già discusso in chiusura del precedente capitolo, inizia adesso l'implementazione dell'applicazione. Il primo paragrafo descriverà l'implementazione del sistema Client-Server per lo scambio dei dati d'interesse (estrapolati dai tools di RoadMap e ImproVisor) e la gestione degli stessi sotto una prima forma testuale. A seguire la descrizione di come verranno ricostruire le interfacce grafiche e le funzionalità associate ad esse (bottoni, finestre, tabelle, menu, visualizzazione in notazione musicale, semiografia, ecc.) soprattutto per la visualizzazione a schermo della leadsheet con improvvisazione. Creare un'applicazione mobile pensando di implementare la stessa grafica sarebbe impossibile e anche quando contro producente, considerando la mole di lavoro nel dover gestire il numero elevato di classi e metodi da tenere in considerazione. La piattaforma Android non accetta le librerie *java.awt* e *java.swing* (se non qualche font in esse contenute). L'importanza nel costruire un'applicazione efficiente risiede anche nel saperne ottimizzare le interfacce grafiche. Basti pensare al fatto che i frameworks in questione devono offrire un servizio per una piattaforme totalmente diversa (Desktop Vs Smartphone). In questo capitolo dunque si descriveranno le metodologie e gli oggetti utilizzati al fine di dare una forma conclusiva all'applicazione che possa essere la base di eventuali sviluppi futuri (Capitolo 6).

5.1 Implementazione Client/Server

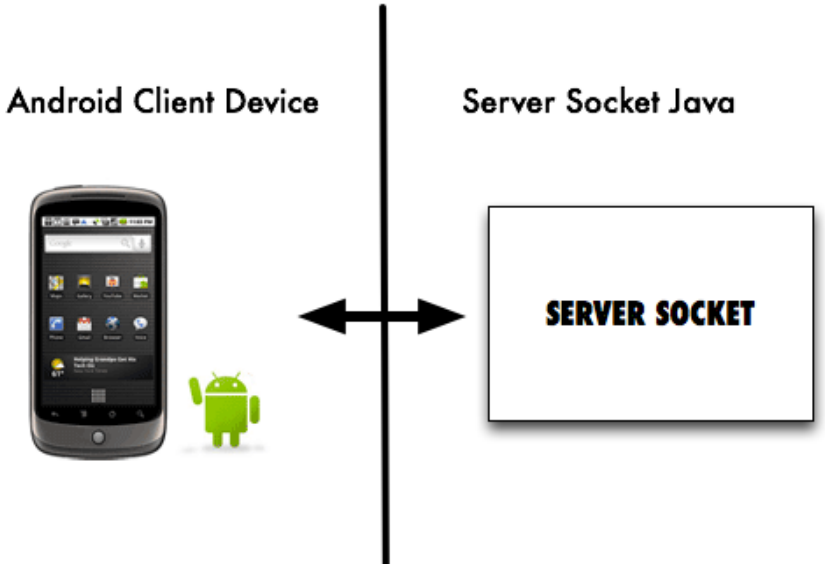

Fig.5.1: Client - Server per l'invio e la ricezione dei dati

Così come mostrato in *Figura 5.1,* lo scambio dei dati avviene mediante un Server Socket. Viene implementato in Java un programma in grado di gestire la connessione *Client-Server*. I metodi della classe *ServerSocket* servono a gestire e creare la connessione con il Client. I costruttori della classe generano in generale una *Socket* servendosi di una porta per l'ascolto (un secondo argomento del costruttore può essere l'indirizzo IP specifico). La costruzione del Server può essere riassunta in 4 fasi:

1. Si crea un oggetto *ServerSocket* in ascolto su una determinata porta. La Classe Server implementerà l'oggetto, creando all'interno del *main* un nuovo oggetto Server. Nel costruttore della classe Server si crea un oggetto in ascolto sulla porta "*4444"*.
2. Da parte del Client vi potrebbero essere richieste di analizzare un brano (una *leadsheet*) o di creare un'improvvisazione in qualsiasi momento, allora non dovrà essere definita una *deadline* in cui la connessione del Server sarà chiusa. Per ovviare al problema bisognerà creare un ciclo infinito in cui il Server sia sempre in ascolto sulla porta *"4444"*, ed ogni volta che riceve una richiesta crea i canali di comunicazione per poter inviare i dati al Client. Quando riceve una richiesta da un Client, una nuova istanza di Socket verrà creata per quel Client.

3. Una volta accettata la connessione viene creata una nuova istanza dell'oggetto *Connect*. Il suo utilizzo serve per creare i canali di comunicazione tra il Server e il Client.
4. Una volta conclusa la comunicazione si chiudono i canali di comunicazione della connessione. La *Fig.5.2* fa riferimento alle fasi appena descritte.

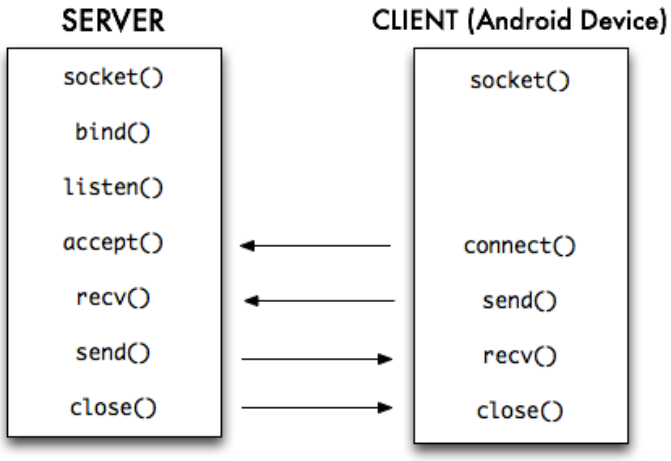

Fig.5.2: Comunicazione Client - Server

A sua volta, il *Client*, crea un nuovo oggetto *socket()* così come i canali di comunicazione. Inizia da qui la comunicazione con il Server e, una volta completata, vengono disallocate le risorse impiegate chiudendo la connessione (*Fig.5.2*). Si noti come, all'interno della Activity di Android, l'inizializzazione del Socket avviene in background. Una scelta apposita per far sì che le richieste vengano nascoste all'utente durante l'esecuzione dell'applicazione *(Fig. 5.3)*.

```
239    class loadingConnectionAndData extends AsyncTask<Integer, String, String>{
240
241        @Override
242        protected String doInBackground(Integer... arg0) {
243
244            /* Inizializzazione SOcket*/
245            try {
246
247                s = new Socket("192.168.0.24", 4444);
248                output = s.getOutputStream();
249                dos = new DataOutputStream(output);
250                Log.i("prova", "connesso");
```

Fig.5.3: Inizializzazione Socket in modalità background

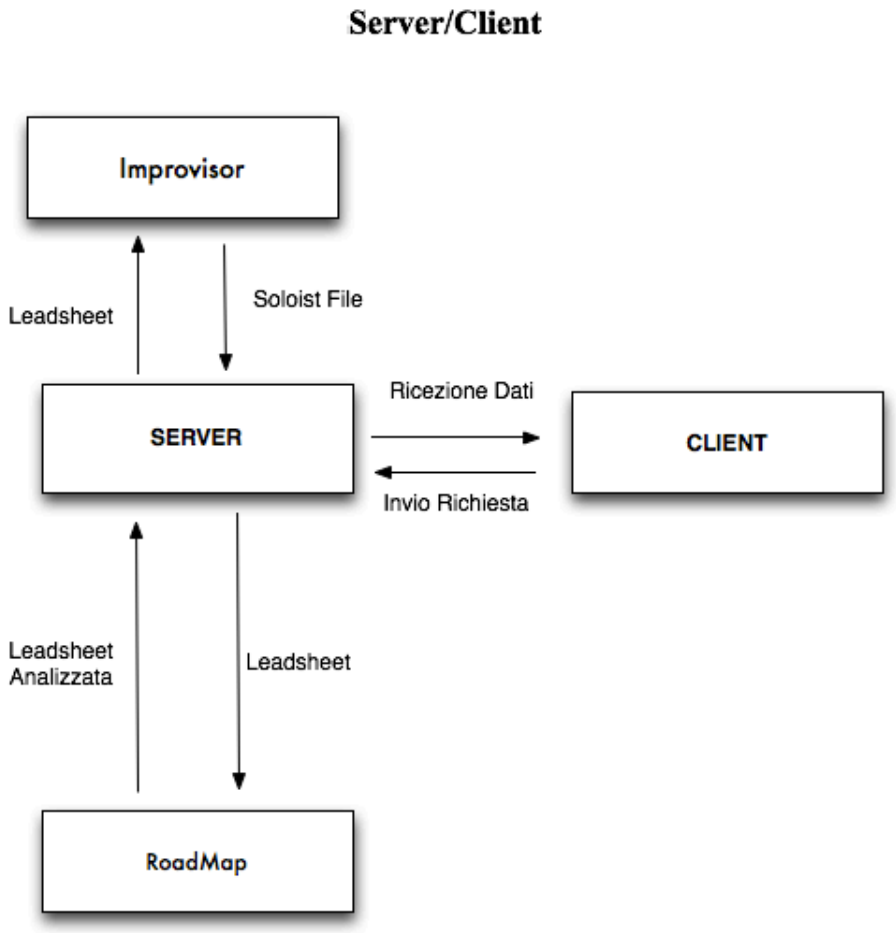

Fig.5.4: Server Client per analisi e improvvisazione leadsheet

Entrando maggiormente nel dettaglio della questione, vogliamo che il Client invii al Server delle richieste specifiche e non dei semplici messaggi. Nella fattispecie, quando verrà richiesta l'analisi sarà invocato dal Server il tool di *RoadMap* che, prendendo in ingresso una *leadhseet*, restituirà una *leadsheet analizzata* così come descritto nel *Capitolo 2*. Quando invece verrà effettuata la richiesta di improvvisare su una data leadsheet, l'invocazione del Server sarà relativa al tool *ImproVisor* così come descritto nel *Capitolo 3* di *Modellazione Improvvisativa*. Si faccia riferimento alla *Fig.5.4* per avere uno sguardo più dettagliato della comunicazione da parte del Server.

4.5.1 La ricezione dei dati sotto forma di stringhe

Dopo aver descritto le funzionalità del Server ed aver introdotto le caratteristiche del sistema Android, ci si sofferma adesso sulla modalità di scambio dei dati.

I. Così come discusso in precedenza, programmare in Android significa anche poter gestire le librerie JAVA, seppur in questo caso il Porting preveda la modellazione ad hoc di un sistema automatizzato. Un sistema che di suo utilizza già numerose classi e metodi, ciascuno con le sue istanze, implementazioni ed ereditarietà diverse. La parte grafica va totalmente ricostruita e le librerie quali *java.awt e java.swing* non sono supportate. Scambiare una *leadsheet* che abbia la possibilità di essere visualizzata in un dispositivo mobile (o tablet) significa dunque dover ricostruire da zero l'intera interfaccia proprio perché diverso è il dispositivo a cui si farà riferimento (Desktop VS Phone). Grazie alle modifiche apportate, un primo approccio ci consente di scambiare le *leadsheet* sotto forma di messaggi di testo, delle stringhe dunque che rispecchiano in effetti quanto espresso finora. Verranno utilizzate le stesse diciture e nomenclature sia per gli accordi che per le note. Prendendo come esempio lo standard jazz "*All Of Me*", la progressione di accordi che ne costituisce la struttura viene espressa sotto forma testuale e salvata in un file **.ls** contenente stringhe *(Fig.5.5)*.

```
...(section (style swing))  C6 | / | E7 | / | A7 | / | A7 Dm | / |
(phrase (style))  E7 | / | Am | / |
(section (style))   D7 | / | G7 | / |
(section (style))  C6 | / | E7 | / | A7 | / | A7 Dm | / |
(phrase (style))   F6 | Fm | C6 Em7b5/Bb | A7 |
(section (style))  Dm7b5 | G7 | C6 Ebo7 | Dm7 G7 |
```

Fig.5.5: La progressione di accordi del brano "All of Me" in forma testuale

Se il Client invia la richiesta di generazione melodica del brano, esso verrà restituito sempre in forma testuale, ma con al suo interno le informazioni restituite dal tool di ImproVisor (ovvero la generazione del solo sulla progressione armonica) così come evidenziato dalla *Fig.5.6*.

... (part (type melody) (title) (composer) (instrument 11)
(volume 85) (key 0) (stave treble))
a8 g8 e8/3 g8/3 a8/3 b8 d+16 c+16 a8 g8
f#+4 r4 d+8 a8 c+8 e8
r1+8
b8 d+8 f#+8 d+8 b8 g#4
r8 g16 d16 a8 c+8 d+8 f+8 e+8 d#+8
f+8 d#+8 f+8 d#+8 c#+8 e+16 c#+16 a8 c+8
a8 f#4+8 r4+8+16 f16
r2+4 g8 e8
g8 f8 e8 c#8 c8 a8 g#8 e8
g8 e16/3 d16/3 b16/3 g#8 f#8 e8 d8 b8/3 d+8/3 e+8/3
g#+8 d#+8 e+8 a+4+8 r1+4+8 ...

Fig.5.6: Una leadsheet testuale che rappresenta l'assolo generato per il brano "All of Me" La nomenclatura delle note, così come la loro durata è descritta all'interno del Capitolo 3.

5.2 Creazione dello scheletro dell'applicazione

Una volta che lo scambio di dati è gestito dal programma di Client/Server (4.5.1) si inizia a implementare le interfacce grafiche utilizzando all'interno delle Activities, novi "*Intent*" e "*Alert Dialog*". La *Figura 5.7*, evidenzia sul lato sinistro un primo layout grafico ottenuto gestendo opportunamente gli oggetti Widgets quali *Button* e *TextView* (es. `btnRoadMap = (Button) findViewById(R.id.btnAlert);` `btnRoadMap.setOnClickListener(this);`). L'utente, cliccando su uno di essi vedrà comparire delle *Alert Dialog* come quella mostrata nel lato destro di *Fig. 5.1*.

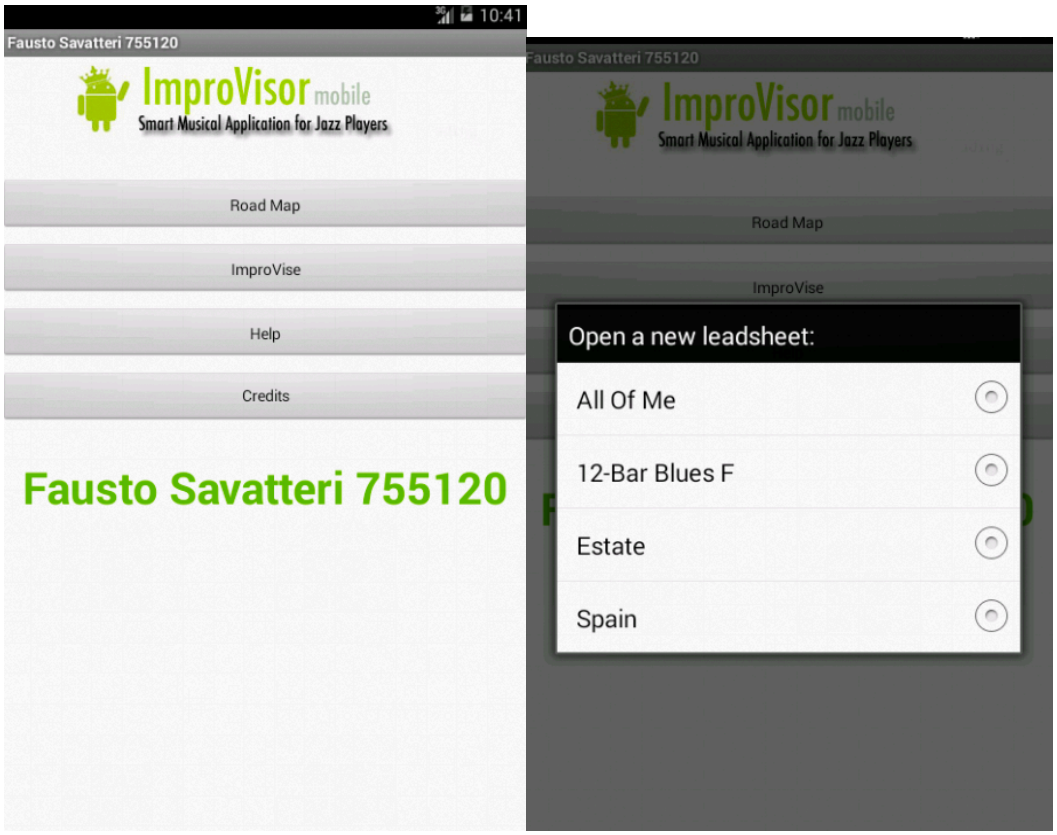

Fig. 5.7: ecco un primo Layout grafico dell'applicazione

La creazione di un'*Alert Dialog* permette di poter scegliere una leadsheet da un elenco di files. Quest'ultima avrà la possibilità di essere analizzata (invocazione del RoadMap) o arricchita con la generazione di una melodia improvvisativa (ImproVisor). La *Fig.5.8* mostra uno schema riassuntivo e mette in evidenza - così come già discusso in precedenza - la relazione che vi è tra il Server e il Client. L'attenzione si focalizza adesso sul lato Client. Si iniziano ad evincere le caratteristiche che fanno parte dell'applicazione stessa e prendono forma all'interno della piattaforma *Android*. Dopo la scelta iniziale di un file *leadsheet* da un elenco (implementazione di *Alert Dialog*), viene dunque inviata una richiesta al Server (istanza socket() con l'apertura di una connessione per iniziare la comunicazione) che ne restituisce le informazioni aggiornate (RoadMap o ImproVisor). Quest'ultime, grazie all'uso di opportuni "*Intent*" verranno visualizzate come nuove finestre GUI personalizzate sia per il RoadMap che per la generazione melodica improvvisativa.

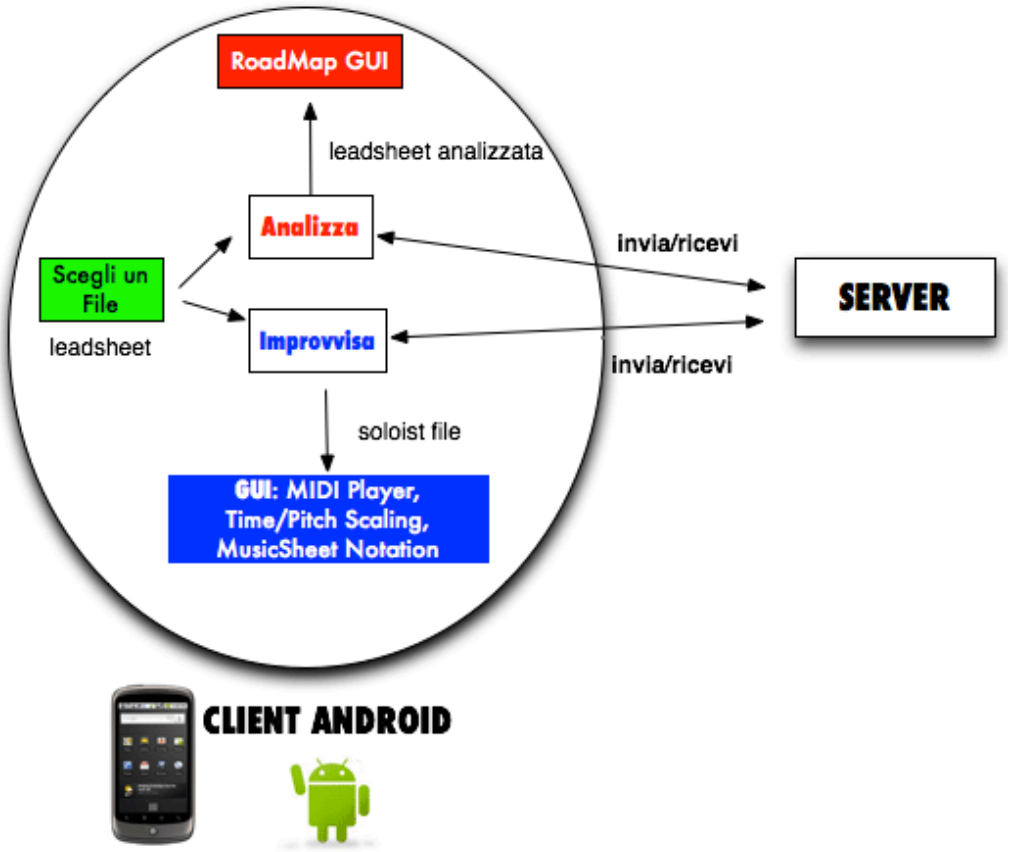

Fig. 5.8: Schema a blocchi riassuntivo: il ruolo del Client (Android Device)

5.2 Il RoadMap all'interno dell'applicazione

Nel caso del RoadMap verrà visualizzata una leadsheet come quella descritta all'interno del Capitolo 2 (*Fig.5.9*). Oltre al RoadMap sono presenti anche le *features* per il salvataggio del file in locale (*Fig.5.10*) e la condivisione dello stesso come un file immagine (Social Networks, E-Mail, ecc.). Inserita anche la possibilità di rotazione dell'immagine per favorire una migliore leggibilità sul dispositivo. Infatti, così come precedentemente discusso, la RoadMap di una progressione di accordi presenti in un brano Jazz, non ha una lunghezza prestabilita ma dipende dal brano in questione. Un *Blues* di 12 battute avrà, allo stato attuale, una visualizzazione sicuramente più ampia rispetto ad esempio ad un brano come *All of Me* (che presenta invece una struttura con più accordi). Se specialmente si utilizza il dispositivo in modalità verticale, la visualizzazione sarà più leggibile nel primo caso piuttosto che nel secondo. Per ovviare al problema basta roteare il dispositivo e, così facendo, anche i brani più complessi (in termini di RoadMap) verranno visualizzati

orizzontalmente sfruttando maggiormente le dimensioni dello schermo del dispositivo. Avere a disposizione

Fig.5.9: Il RoadMap di un blues di 12 battute visualizzato all'interno del dispositivo

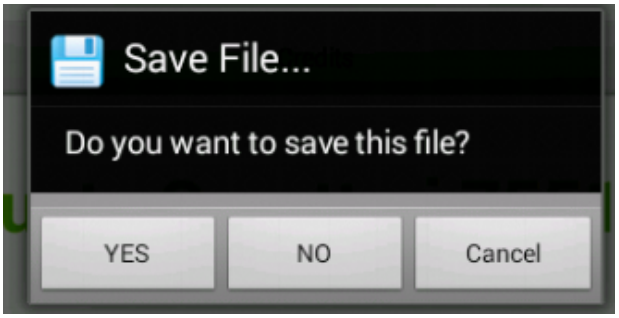

Fig.5.10: Il salvataggio di una Leadsheet analizzata

una leadsheet analizzata può essere un grande vantaggio in termini di portabilità, semplicità e condivisione. Rendere interattiva e "tascabile" l'analisi armonica di un brano, frutto normalmente di anni di studi da parte di un musicista, offre uno strumento didattico compatto e semplice, ma al contempo ricco di informazioni. Diversi possono essere i casi d'uso che si contrappongono agli strumenti didattici più tradizionali (Libri accademici, DVD, lezioni private, ecc.). La *Fig.5.11* fornisce un

caso d'uso possibile per l'applicazione Android e, nello specifico, la condivisone tra due musicisti di un brano analizzato data una relativa progressione di accordi Jazz (JCP).

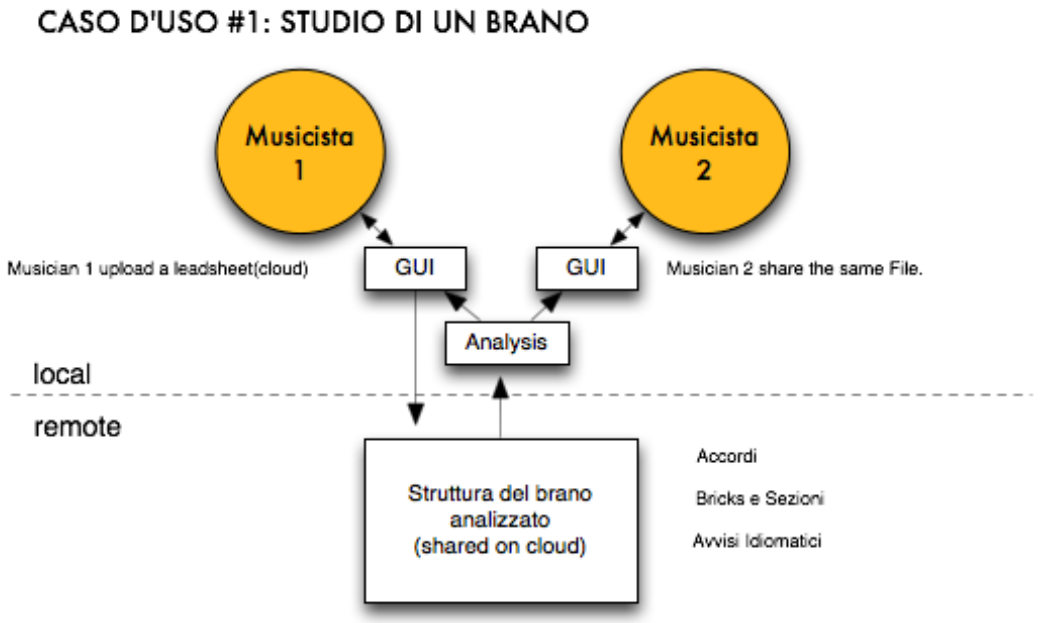

Fig.5.11: Condivisione tra due musicisti di una RoadMap.

5.3 Generazione improvvisativa all'interno dell'applicazione

Analogamente a quanto visto nel precedente paragrafo, la parte relativa all'improvvisazione restituirà, dopo la richiesta da parte del Client, una nuova finestra grafica contenente la leadsheet arricchita con la melodia improvvisativa. A tal proposito entra in atto l'implementazione di un *midi player* per la riproduzione del solo. In un primo prototipo sono state utilizzate le classi *android.media.MediaPlayer*. La *Fig.5.12* mostra una prima interfaccia grafica per la finestra di ImproVisor. Vi sono 4 semplici bottoni che permettono l'ascolto del "tema" e quindi della base relativa al file scelto. L'utente, premendo il tasto "Play Solo" potrà invece ascoltare l'assolo generato dal tool di ImproVisor grazie agli algoritmi e alla grammatica probabilistica di cui si è discusso nel *Capitolo 3*. Il beneficio tratto dall'ascolto mediante un lettore MIDI può essere sicuramente un vantaggio che tuttavia rimane limitato. Un secondo prototipo dell'applicazione invece, arricchisce notevolmente la finestra di ImproVisor, grazie all'apporto di nuove features essenziali per la riproduzione di un file musicale e la visualizzazione dello stesso sotto forma di Midi Musical LeadSheet. L'applicativo

risulta adesso più efficace e interattivo. Oltre al *Play* e allo *Stop* vengono aggiunte le funzioni di *scorrimento* all'interno del brano (avanti e indietro di una battuta) di *time scaling* e *pitch scaling*. Ma soprattuto, dopo la generazione di un solo, l'utente potrà fare un vero e proprio *Play-Along* del brano. In aggiunta all'ascolto è adesso possibile visualizzare la melodia generata su una tastiera virtuale (*Virtual Piano*) e su uno spartito contenente le note facenti parte del solo. Esse rispettano le regole [5] della notazione musicale. Si faccia riferimento alle *Fig.5.13* e *5.14* che mostrano il secondo prototipo dell'applicativo.

Fig.5.12: Un primo prototipo per la riproduzione dei files Midi rappresentanti la generazione melodica di improvvisazione.

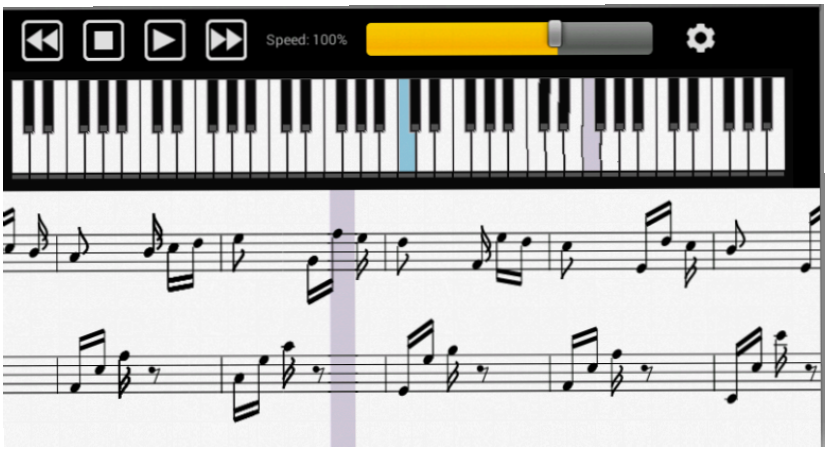

Fig.5.13: Nuova finestra di riproduzione per ImproVisor. Oltre ai pulsanti basilari aggiunte le funzioni di time e pitch scaling, visualizzazione su un Piano Virtuale e su Spartito.

[5] La semiografia (o notazione musicale) è la rappresentazione grafica della musica.

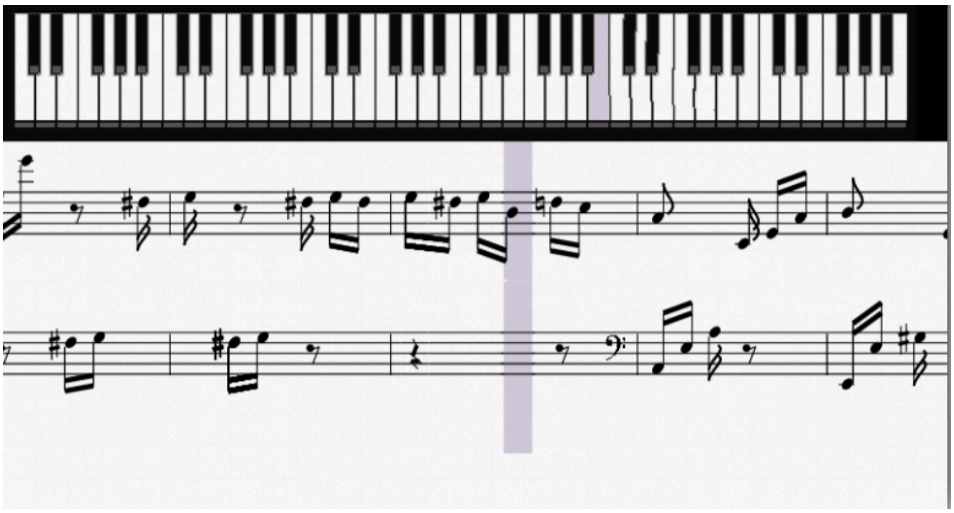

Fig.5.14: Il Play Along di un solo. Visualizzazione a schermo intero all'interno dell'applicazione.

All'utente viene offerta la possibilità di salvare il file come immagine ed eventualmente condividerlo (o stamparlo). Il Salvataggio, ancor più che nel caso della RoadMap, risulta essere di estrema utilità. L'utente ad esempio potrebbe avere delle perplessità relative ad un brano, la cui progressione di accordi o la velocità dello stesso abbia come conseguenza quella di ostacolare il musicista nell'esecuzione e limitarlo nell'improvvisazione. Il salvataggio di una *leadsheet* "improvvisata" è da considerarsi una *feature* essenziale se si vuole offrire ai fruitori dell'applicazione un servizio interattivo (*play along*) e al contempo un supporto didattico da cui poter trarre spunti e suggerimenti innovativi *(Fig.5.15)*.

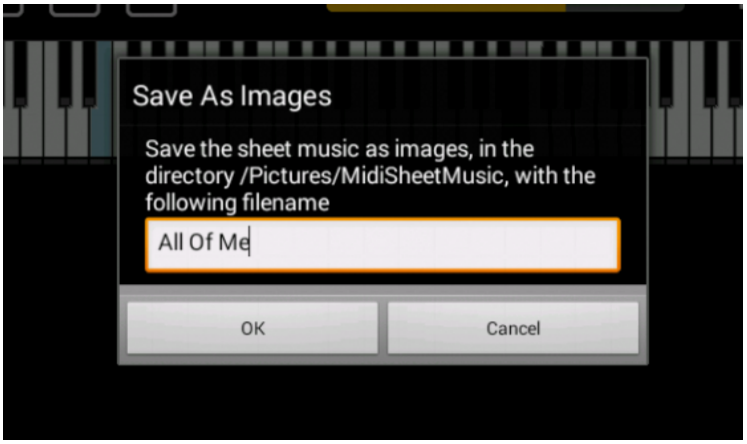

Fig.5.15: Il Salvataggio di una leadsheet come file immagine

Capitolo 6 - Conclusioni e Sviluppi Futuri

In questo progetto di tesi sono state presentate le metodologie e le tecniche volte a realizzare l'automazione dell'analisi musicale per sequenze di accordi. Prendendo spunto dal software *open-source ImproVisor* (sviluppato da *Keller*, *Harvey Mudd College*, dal 2005 ad oggi), sono stati introdotti dei *bricks* idiomatici che stabiliscono le tonalità basandosi su di un *brick dictionary*. Le problematiche di ambiguità e sovrapposizione (*overlapping*) sono state affrontate assegnando dei costi relativi ad ogni tipo di *brick*. E' stato mostrato come le tecniche implementate siano in grado di determinare i *bricks* e unirli (*joins*) in una melodia, come suggerito da Cork. Le informazioni tonali definite, insieme ai bricks, sono utilizzate per determinare le modulazioni e le tonalità del brano. Tutte le informazioni sono presentate in una struttura chiamata RoadMap. Tali informazioni sono ritenute molto utili sia per gli improvvisatori che per i musicisti di accompagnamento. Si consideri ad esempio il brano *"All The Things You Are"* famoso, oltre che per la sua bellezza, anche per avere al suo interno sei tonalità distinte. Avere a disposizione un tool che riesca ad automatizzare il processo anche per strutture complesse è sicuramente un vantaggio, specie se il tutto diventa portatile e tascabile. Oltre a fornire una rappresentazione dell'analisi, compresa la capacità di "rompere" grossi bricks in sottoparti costituenti, vi è anche la possibilità di utilizzare il tool di improvvisazione, includendo anche la riproduzione interattiva con l'ascolto del brano (*play along*) e le numerose funzionalità di *time scaling, pitch scaling*, *salvataggio* dei files analizzati e *semiografia* delle note.

La capacità del tool nel generare assoli che suonino simili ad un'artista reale, partendo dai dati di training, ma allo stesso tempo distinti da un particolare assolo, dimostra l'efficacia del metodo di astrazione. La combinazione di contorni e categorie di note sembra bilanciare la somiglianza e l'innovazione sufficientemente bene da poter caratterizzare le melodie come *melodie jazz*. Inoltre, il *clustering* sembra essere un algoritmo adatto per raggruppare le melodie astratte simili. Le catene di Markov sono efficaci nella strutturazione degli assoli, tuttavia, un'ulteriore struttura globale sarebbe auspicabile per fornire maggiore coerenza tra gli assoli. Un'implementazione possibile potrebbe essere quella di catene di Markov a lunghezza variabile (Variable

Lenghts Markov Models: VLMM). Vista la mole di lavoro e il numero di classi in questione, bisognerebbe riconsiderare gran parte della struttura di Clustering, ove attualmente le Markov risiedono. Si ricordi infatti quanto espresso all'interno del Capitolo 3, ovvero che il Clustering si basa su 7 parametri ed utilizza oggetti *Polylist* per costruire le n-grams. Le catene di Markov a lunghezza variabile sono già state implementate in altri studi nell'ambito della *computer music.* Sono stati ottenuti anche risultati molto soddisfacenti, fra tutti, l'algoritmo *DCTW: Markov Lenght Predictor* che tuttavia, non farebbe a caso nostro. Trattandosi di un semplice predittore, la sua struttura risulterebbe totalmente diversa rispetto a quella implementata nel Clustering. Ad esempio non sarebbe adatto a gestire oggetti Polylist, ma solo semplici stringhe. Una prova più convincente del metodo sarebbe un esperimento per determinare se gli ascoltatori, soprattutto jazzisti, possano trovare delle differenze tra un assolo generato da una grammatica di apprendimento e un assolo composto umanamente. Entra in atto il concetto di psicoacustica e soprattutto la soggettività nel giudizio individuale. D'altronde, anche in musica, non è detto che la difficoltà o il numero di note di un assolo determinino la qualità o la bellezza dello stesso. Parlando di sviluppi futuri si potrebbe lavorare ancora molto a livello di grammatica ed armonia, consentendo una maggiore espressività e avvicinandosi ulteriormente all'esecuzione di un musicista reale. Sono stati introdotti gli *slopes* per rappresentare i contorni, ma è chiaro che l'idea generale di macro strutture diverse per ogni *slope* potrebbe rivelarsi molto utile. Esaminando ad esempio informazioni più specifiche sulle note, si potrebbero ottenere nuovi risultati e definire anche il tipo di intervalli. La più grande debolezza nella generazione di assoli, quando si è in presenza di un brano molto lungo, è forse la mancanza di una struttura globale. Il famoso vibrafonista *Gary Burton*, considerato una figura di prim'ordine nel campo dell'insegnamento, spiega come improvvisare significhi creare una storia, raccontarla - anche in maniera semplice - ma seguendo un ordine e uno sviluppo della storia, in modo tale da catturare l'attenzione degli ascoltatori. Una valutazione conclusiva dell'efficacia di n-grammi per una struttura globale può essere fatta dato un numero sufficientemente ampio come insieme di training. Un altro approccio che potrebbe essere esplorato è di costruire la struttura (ad alto livello) di un assolo generato, basandosi su un particolare insieme di training. In conclusione, raccogliere le informazioni e le metodologie descritte in

un’applicazione *Android* crea sicuramente dei vantaggi nell’utilizzo e nella portabilità della stessa. Tale progetto di tesi si prefigge come obbiettivo quello di fornire uno strumento didattico utile ed allo stesso tempo interattivo, prerequisito fondamentale nella creazione di un’applicazione *Android*. Allo stato attuale può sicuramente essere considerato una base da cui attingere nell’eventualità di sviluppi futuri. Infatti, l’implementazione di nuove *features cloud oriented,* come ad esempio la condivisione e la sincronizzazione tra più dispositivi della stessa risorsa (*leadsheet*), potrebbe ampliare notevolmente i casi d’uso. Basti pensare al famoso Real Book, il cui nome fa riferimento alla raccolta di diversi spartiti all’interno di un libro, un’enciclopedia dei più noti standard jazz trascritti e riuniti, in modo tale da avere una base per lo studio e l’improvvisazione. La creazione di un “Real Book interattivo” contenente - oltre al tema ed alla “*griglia”* di accordi - espressioni idiomatiche e suggerimenti per una migliore comprensione, potrebbe essere uno strumento didattico di notevole innovazione.

Bibliografia

[1] Keller, R. 2005. “Leadsheet Notation.” Available on- line at www.cs.hmc.edu/ ~keller/jazz/improvisor/ LeadsheetNotation.pdf. Last accessed 20 May 2013.

[2] Kang, Y.-K., K.-I Ku, and Y.-S. Kim. 2001. “Extracting Theme Melodies by Using a Graphical Clustering Algorithm for Content-Based Music Information Retrieval.” Proceedings of the Fifth East Euro- pean Conference on Advances in Databases and Information Systems. New York: Springer-Verlag, pp. 84–97.

[3] Pachet, F. 1999. “Surprising Harmonies.” International Journal on Computing Anticipatory Systems 4:1–20.

[4] Kim, Y. E., et al. 2000. “Analysis of a Contour-Based Representation for Melody.” Proceedings of the Inter- national Symposium on Music Information Retrieval. Amherst, New York: University of Massachusetts. Disponibile online al seguente indirizzo: http://ciir.cs.umass.edu/music2000/indexnoframes.html.

[5] ImproVisor 2007: Jazz Improvisation Advisor for the Improviser, http:// www.cs.hmc.edu/~keller/jazz/improvisor/

[6] Ames, C. 1989. “The Markov Process as a Compositional Model: A Survey and Tutorial.” Leonardo 22(2):175–187.

[7] Chomsky, N. 1956. “Three Models for the Description of Language.” IRE Transactions on Information Theory 2(3):113–124.

[8] Cork, C. 1988. Harmony by LEGO Bricks: A New Approach to the Use of Harmony in Jazz Impro- visation. Leicester, United Kingdom: Tadley Ewing Publications.

[9] Cork, C. 2008. The New Guide to Harmony with LEGO Bricks. London: Tadley Ewing Publications.

[10]Elliott, J. 2009. Insights in Jazz: An Inside View of Jazz Standard Chord Progressions. Available online at dropback.co.uk. Last accessed 20 May 2013.

[11] Aebersold, J. 1979. Turnarounds, Cycles, and II/V7’s. New Albany, Indiana: Jamey Aebersold Jazz.

[12] Ulrich, W. 1977. “The Analysis and Synthesis of Jazz by Computer.” In International Joint Conference on Artificial Intelligence, pp. 865–872.

[13] Steedman, M. 1984. “A Generative Grammar for Jazz Chord Sequences.” Music Perception 2:52–77.

[14] Chemillier, M. 2004. “Toward a Formal Study of Jazz Chord Sequences Generated by Steedman’s Grammar.” Soft Computing 8(9):617–622.

[15] Scholz, R., V. Dantas, and G. Ramalho. 2005. “Automating Functional Harmonic Analysis: The Funchal System.” In Proceedings of the Seventh IEEE International Symposium on Multimedia, pp. 759–764.

[16] Wilding, M. 2008. “Automatic Harmonic Analysis of Jazz Chord Progressions Using a Musical Categorical Grammar. ” M.Sc. thesis, University of Edinburgh, School of Informatics.

[17] Choi, A. 2011. “Jazz Harmonic Analysis as Optimal Tonality Segmentation.” Computer Music Journal 35(2):49–66.

[18] Sipser, M. 2005. Introduction to the Theory of Computa- tion. 2nd ed. Independence, Kentucky: Cengage.

[19] Floyd, R.W. 1962. “Algorithm 97: Shortest Path.” Com- munications of the ACM 5(6):345.

[20] Dubnov, S., et al. 2003. “Using Machine-Learning Methods for Musical Style Modeling.” Computer 36(10):73–80.

[21] Eck, D., and J. Lapalme. 2008. “Learning Musical Structure Directly from Sequences of Music.” Technical report 1300.Montreal:Universite´deMontre´al,De´partement d’informatique et de recherche ope´rationnelle.

[22] Cruz-Alcazar, P., and E. Vidal-Ruiz. 1998. “Learning Regular Grammars to Model Musical Style: Comparing Different Coding Schemes.” Proceedings of the Fourth International Conference on Grammatical Inference.

[23] F. Pachet, *Playing with Virtual Musicians: the Continuator in practice*, IEEE Multimedia 9:3, pp. 77-82. 2000

[24] Chang, C.-W., and H. C. Jiau. 2003. “Extracting Significant Repeating Figures in Music by Using Quantized Melody Contour.” Proceedings of the Eighth IEEE International Symposium on Computers and Communication. New

[25] De Roure, D. C., and S. G. Blackburn. 2000. “Content- Based Navigation of Music Using Melodic Pitch Contours.” Multimedia Systems 8(3):1–11.

[26] Jones, K. 1981. “Compositional Applications of Stochastic Processes.” Computer Music Journal 5(2):45–61.

[27] Verbeurgt, K., et al. 2004. “Extracting Patterns in Music for Composition via Markov Chains.”Proceedings of the 17th International Conference on Innovations in Applied Artificial Intelligence. New York: Springer-Verlag, pp. 1123–1132.

[28] Simone Bollini, 2011 “The Virtual Jazz Player: real-time modelling of improvisational styles using Variable-Length Markov Models incorporating musicological rules” Politecnico di Milano.

[29] Yuval Marom. *Improvising Jazz with Markov Chains*. Technical report, Department of Computer Science, The University of Western Australia (1997).

[30] David M. Franz. *Markov Chains as Tools for Jazz Improvisation Analysis*. Thesis, Virginia Polytechnic Institute and State University 1998.

[31] Hartigan, J. A., and M. A. Wong. 1979. “Algorithm AS 136: A k-Means Clustering Algorithm.” Applied Statistics 28(1):100–108.

[32] Official Android Developer by Google Inc. 2008: Packages http:// developer.android.com/reference/packages.html. Components: http:// developer.android.com/guide/components/index.html Training:http:// developer.android.com/training/index.html