Capitolo 5

Similarità e plagio musicale

In questo capitolo ci occuperemo del problema della similarità di brani musicali. Questo aspetto è alla base di un problema più complesso che è quello del plagio musicale. La tematica del plagio musicale è assai controversa perché coinvolge aspetti economici non di poco conto. Esistono vari esempi di cause legali relativi a plagi musicali di enorme impatto mediatico ed economico. Il principale ostacolo, che ritroviamo anche in altri problemi musicali, è quello della non esistenza di una metrica formale che permetta di dire quanto due composizioni musicali siano simili. Spesso un tale giudizio è soggettivo. Data la complessità del problema, focalizzaremo la nostra attenzione solo sull'aspetto melodico; tuttavia questo aspetto è quello che viene principalmente considerato anche nelle reali cause legali. Utilizzeremo delle rappresentazioni specifiche basate su stringhe di caratteri: nella prima parte del capitolo rivedremo alcune nozioni e algoritmi relativi alla manipolazione di stringhe.

5.1 Stringhe e distanze

Una stringa è una sequenza di caratteri presi da un dato alfabeto. Ad esempio, dato l'alfabeto $\Sigma = \{a, b, c, d, e\}$, un esempio di stringa è $s = abcbddead$. La lunghezza della stringa è il numero di caratteri di cui è costituita; nell'esempio la stringa s ha lunghezza 9. Con un opportuno alfabeto è possibile usare delle stringhe per rappresentare la musica. Un confronto delle stringhe che rappresentano due melodie può rivelare eventuali similarità.

5.1.1 Edit distance

Date due stringhe di simboli la edit distance è definita come il minimo numero di operazioni di inserimento, cancellazione e sostituzione che devono essere effettuate per trasformare una delle stringhe nell'altra. Più formalmente consideriamo un alfabeto Σ, ad esempio tutte le lettere. Una stringa definita su Σ è una sequenza di simboli appartenenti a Σ, ad esempio $\alpha = torta$, oppure $\beta = parlare$.

L'operazione di cancellazione elimina un simbolo: una stringa $\alpha = \alpha_1 x \alpha_2$ viene trasformata in $\alpha = \alpha_1 \alpha_2$, eliminando il simbolo x. Ad esempio $\alpha = torta$, cancellando il carattere o, diventa $\alpha = trta$. Il simbolo da cancellare può trovarsi in qualsiasi punto della stringa.

L'operazione di inserimento inserisce un simbolo: la stringa $\alpha = \alpha_1 \alpha_2$ viene trasformata in $\alpha = \alpha_1 x \alpha_2$, inserendo il simbolo x. Ad esempio $\alpha = torta$ diventa $\alpha = tortar$. Il simbolo può essere inserito in qualsiasi punto della stringa.

L'operazione di sostituzione cambia un simbolo: una stringa $\alpha = \alpha_1 x \alpha_2$ viene trasformata in $\alpha = \alpha_1 y \alpha_2$, cambiando x in y. Ad esempio $\alpha = torta$ diventa $\alpha = porta$. Anche in questo caso il simbolo da cambiare può trovarsi in qualsiasi punto della stringa.

Denotando con $ed(\alpha(i), \beta(j))$ la edit distance fra i primi i caratteri di α e i primi j caratteri β si ha che se una delle due stringhe è nulla allora la distanza è pari alla lunghezza della stringa non nulla altrimenti la edit distance (minima) è data dal minimo fra

$$ed(\alpha(i), \beta(j)) = \min \begin{cases} ed(\alpha(i-1), \beta(j) + 1, & \text{(ins. o canc. in } \alpha) \\ ed(\alpha(i), \beta(j-1) + 1, & \text{(ins. o canc. in } \beta) \\ \begin{cases} ed(\alpha(i-1), \beta(j-1) & \text{se } \alpha_i = \beta_i \text{ (nessuna)} \\ ed(\alpha(i-1), \beta(j-1) + 1 & \text{se } \alpha_i \neq \beta_i \text{ (sost.)} \end{cases} \end{cases} \tag{5.1}$$

È facile implementare un algoritmo che calcola la edit distance. Inoltre si può facilmente tenere traccia di quali operazioni sono state utilizzate. Basta usare due matrici di dimensione $(n+1) \times (m+1)$ dove n e m sono le lunghezze delle stringhe. La prima matrice conterrà il valore della distanza minima mentre la seconda servirà a memorizzare quale delle tre operazioni deve essere utilizzata per ottenere la distanza minima. Le righe e le colonne corrispondono ai singoli caratteri, nell'ordine in cui compaiono nelle strighe, e quindi corrispondono alle sottostringhe, inclusa la sottostringa vuota, a partire dall'inizio delle stringhe. Ad esempio per le parole usate prima avremmo delle matrici come quelle riportate sotto:

Distanze

	ϵ	p	a	r	l	a	r	e
ϵ	0	1	2	3	4	5	6	7
t	1							
o	2							
r	3							
t	4							
a	5							

Operazioni

	ϵ	p	a	r	l	a	r	e
ϵ	-	i_p	i_a	i_r	il	i_a	i_r	i_e
t	i_t							
o	i_o							
r	i_r							
t	i_t							
a	i_a							

La prima colonna e la prima riga delle matrici possono essere facilmente riempite in quanto corrispondono ai casi in cui una delle due stringhe è vuota, pertanto la distanza minima è semplicemente la lunghezza della stringa non vuota e le operazioni da utilizzare sono ovviamente tutte inserimenti, se si parte dalla stringa vuota, o tutte cancellazioni, se si parte dalla stringa non vuota (o anche un misto fra cancellazioni e inserimenti). Le restanti celle delle matrici possono essere calcolate sfruttando l'equazione 5.1. Nella matrice delle distanza si memorizza il valore

della distanza mentre nella matrice delle operazioni si memorizza quale dei 3 casi dell'equazione ha determinato il minimo.

L'algoritmo 1 calcola le due matrici.

Algorithm 1 Edit distance

```
public static int computeED(String m1, String m2) {
  int distance = 0;
  int len1=m1.length();
  int len2=m2.length();
  int[][] d = new int[len1+1][len2+1];
  int[][] s = new int[len1+1][len2+1];
  for (int i=0; i<=len1; i++) {
    for (int j=0; j<=len2; j++) {
      d[i][j]=0;
      s[i][j]=0;
    }
  }
  s[0][0]='-';
  for (int i=1; i<=len1; i++) {
    d[i][0]=i;
    s[i][0]='D';
  }
  for (int j=1; j<=len2; j++) {
    d[0][j]=j;
    s[0][j]='D';
  }
  int i,j;
  for (i=1; i<=len1; i++) {
    for (j=1; j<=len2; j++) {
      if (m1.charAt(i).equals(m2.charAt(j))) {
      d[i][j]=d[i-1][j-1];
      s[i][j]='E';
  } else {
      if (d[i-1][j-1] <= d[i-1][j] && d[i-1][j-1] <= d[i][j-1]) {
        d[i][j]=1+d[i-1][j-1];
        s[i][j]='S';
      }
      else if (d[i-1][j] <= d[i][j-1]) {
        d[i][j]=1+d[i-1][j];
        s[i][j]='^';
      }
      else {
        d[i][j]=1+d[i][j-1];
        s[i][j]='<';
      }
      d[i][j]=1+Math.min(d[i-1][j-1], Math.min(d[i-1][j],d[i][j-1]));
    }
  }
}
```

5.1.2 Ricerca di sottostringhe (string matching)

Un algoritmo di string matching permette di vedere se determinati pattern di caratteri sono presenti in una stringa. Più formalmente il problema consiste nel determinare le occorrenze di un pattern di lunghezza p in un testo di lunghezza n, con $p < n$. La ricerca di una stringa all'interno di un'altra stringa può essere utilizzata come strumento per individuare frammenti melodici in comune. L'algoritmo 2 è l'algoritmo più semplice che usa la forza bruta controllando la presenza della sottostringa a partire da ogni possibile posizione.

Algorithm 2 Ricerca sottostringa, forza bruta

```
stringSearchNaive(String pattern, String str) {
  int p=pattern.length();
  int n=str.length();
  for (int i = 0; i <= n - p; i++) {
    int j;
    for (j = 0; j < p; j++)
       if (text[i+j] != pattern[j]) break;
    if (j == p) return i;  // pattern found at offset i.
  }
  return n; // pattern not found.
```

La complessità è $O(nm)$ ma il for interno, nel caso in cui non ci sia un match, si interrompe al primo carattere diverso e questo succede quasi subito se le stringhe sono distribuite uniformemente. Le prestazioni sono relativamente buone in pratica. Ovviamente il caso pessimo è $\Theta(nm)$. Ad esempio se la stringa $\alpha = aaaaaaaaaaaaaaaaaaaaaaab$ e la stringa $\beta = aaaaaaaaaaaaa.....aaaaaaaaaaa$ il for interno si accorgerà del mismtach solo all'ultimo carattere e il controllo verrà ripetuto per tutte le possibili posizioni iniziali. Esistono algoritmi più efficienti. Ad esempio l'algoritmo KMP (Knuth-Morris-Pratt) che opera una breve pre-computazione sulla stringa da cercare per poi sfruttare le informazioni dedotte per rendere più efficiente la ricerca. L'algoritmo KMP ha complessità $O(m)$.

La pre-computazione sul pattern si basa sulla seguente osservazione. Se un possibile match fallisce nella posizione q-esima del pattern sappiamo che i primi $q-1$ caratteri del pattern sono uguali a quelli della stringa (a partire dalla posizione corrente). Il pattern è sempre lo stesso quindi possiamo pre-calcolare la prossima posizione da analizzare possibilmente "saltando" delle posizioni. Facciamo un esempio concreto. Supponiamo che il pattern sia "abcde" e che stiamo controllando la sua occorrenza a partire dalla posizione 10 dalla quale la stringa ha i seguenti caratteri "abcda". Il mismatch verrà scoperto solo quando si controllerà il quinto carattere del pattern. Tuttavia possiamo ora evitare di controllare a partire dalle posizioni 11,12,13 e 14 visto che nessuna di esse contiene una "a". La prossima posizione "utile" per un possibile match è la 15. Un altro esempio. Pattern "abcabe", stringa "abcabc...". Il mismatch in questo caso verrà rilevato al sesto carattere. Inutile controllare le successive due posizione: il prossimo possibile match è dopo 3 posizioni. Dovrebbe essere chiaro che "il prossimo possibile match" dipende dai caratteri del pattern. La pre-computazione serve a stabilire l'indice del prossimo possibile match. Bisogna calcolare una tabella che per ogni indice i, $i = 1, ..., len(pattern)$, ci dice l'indice $next[i]$ del prossimo possibile match. Si noti che l'indicizzazione parte da 1 e non da 0 (scelta arbitraria).

In pratica la tabella *next* contiene informazioni su come il pattern si confronti con se stesso e più precisamente su come la parte iniziale del pattern si confronti con ogni suo prefisso. Sia q il numero di caratteri per i quali c'è stato un match con il testo. Nell'esempio precedente

pattern=abcabe, $q = 6$ (al sesto carattere c'è stato un mismatch). Ci interessa sapere quale è il minimo spostamento (shift) del pattern tale che un suo prefisso corrisponda ad un suffisso dei primi $q-1$ caratteri del pattern. Tale scorrimento fornisce la prima posizione utile per un possibile occorrenza del pattern. E questa informazione ci serve per tutti i possibili q.

Algorithm 3 Algoritmo KMP, pre-computazione sul pattern

```
KMP-precomputation(String pattern) {
  m=pattern.length;
  int next[m];
  next[1]=0;
  k=0;
  for (q=2; q<m; i++) {
    while ((k>0) and (pattern[k+1] <> pattern[q])) {
      k=next[k];
    }
    if (pattern[k+1] == pattern[i]) then k=k+1;
    next[q]=k;
  }
  return next[];
}
```

Quindi la domanda alla quale dobbiamo dare una risposta con la pre-computazione è:

> Dato il prefisso $pattern[1..q]$, quale è il più piccolo intero s, tale che per un $k < q$ si ha che $P[1..k] = T[s+q-k..s+q]$, cioè quale è il più piccolo shift del pattern tale che un suo prefisso, di lunghezza $k < q$, sia uguale ad un suffisso che finisce nella posizione q?

L'algoritmo 3 calcola questa informazione per tutti i possibili q e la memorizza nella tabella *next*.

Algorithm 4 Algoritmo KMP

```
stringSearchKMP(String pattern, String str) {
  n = str.length;
  m = pattern.length;
  next = KMP-precomputation(pattern);
  q = 0;                                        //#chars match
  for (i=1; i<n; i++) {
    while (q>0) and (pattern[q+i] <> str[i])
      q = next[q];                              //next char does not match
    if (pattern[q+1] == str[i])
      q = q+1;                                  //next char match
    if (q == m)
      print "Occorrenza alla posizione i-m";
      q = next[q];                              //Look for next match
  }
}
```

L'algoritmo 4 sfrutta la tabella *next* per evitare i confronti inutili: ogni qualvolta c'è un mismatch si evita di riconfrontare i simboli che già si conoscono.

5.1.3 Ricerca approssimata

Dato il pattern si potrebbe essere interessati a cercare occorrenze approssimate. Per fare questo si sfrutta la edit distance. Il problema dello string matching approssimato può essere definito nel seguente modo. Dato un pattern $p_1, \ldots, p_m$ e un testo $t_1, \ldots, t_n$, con $m < n$, trovare la sottostringa $t_i, \ldots, t_j$ tale che la edit distance $ed((p_1, \ldots, p_m), (t_i, \ldots, t_j))$ è quella minima fra tutte le possibili sottostringhe.

Quindi un approccio semplice sarebbe quello di considerare tutte le possibili sottostringhe, cioè tutti i possibili indici i, j, con $1 \leq i < j \leq n$, che sono $O(n^2)$ e per ognuno di essi calcolare la edit distance con il pattern. Quindi la complessità totale sarebbe $O(mn^3)$.

Esistono algoritmi più efficienti.

5.2 Similarità basata sulla distanza testuale

Vediamo adesso degli algoritmi per individuare la similarità fra due (o più) melodie. Mettendo insieme una rappresentazione testuale e la edit distance si può immediatamente avere un semplice algoritmo per la similarità: la edit distance è una misura di similarità. Se cambiano poche note nella melodia anche la stringa della rappresentazione cambierà di poco e conseguentemente l'edit distance sarà piccola. Se invece due melodie sono molto diverse fra loro lo saranno anche le stringhe che le rappresentano e pertanto l'edit distance sarà più grande.

Come facciamo a valutare la "bontà" di un tale algoritmo? Procederemo con degli esempi, valutando il comportamento dell'algoritmo.

Per poter utilizzare gli algoritmi di edit distance abbiamo bisogno di rappresentare la musica con stringhe di testo: una melodia sarà specificata da una stringa. Che cosa codifichiamo? La quantità di informazioni che codifichiamo nella stringa e anche il modo in cui le codifichiamo giocano un ruolo fondamentale. Per procedere consideriamo un caso concreto. La Figura 5.1.

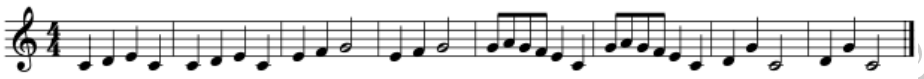

Figura 5.1: Melodia di Fra Martino

Consideriamo una prima, semplice rappresentazione che codifica ogni nota con due o tre simboli, una o due lettere che corrispondono al nome della nota e all'eventuale alterazione musicale (♯ o ♭) e un numero che specifica l'ottava; inoltre useremo un carattere di separazione, −, per rendere la stringa più facilmente leggibile. La melodia della Figura 5.1 è rapprresentata dalla seguente stringa: "C4 D4 E4 C4 C4 D4 E4 C4 E4 F4 G4 E4 F4 G4 G4 A4 G4 F4 E4 C4 G4 A4 G4 F4 E4 C4 D4 G4 C4 D4 G4 C4".

In questo esempio semplice non ci sono alterazioni e tutte le note sono nell'ottava numero 4 (quella centrale) della tastiera del pianoforte.

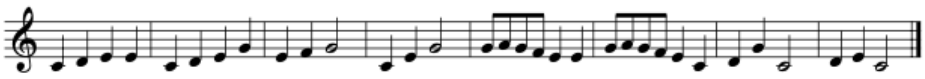

Figura 5.2: Melodia simile a Fra' Martino campanaro, in Do maggiore.

La Figura 5.2 mostra una variazione della melodia in Figura 5.1. La stringa che la descrive è "C4 D4 E4 C4 C4 D4 E4 E4 F4 G4 C4 D4 E4 F4 G4 G4 A4 G4 F4 E4 C4 G4 A4 G4 F4 E4 C4 D4 G4 C4 D4 E4 D4 C4". La edit distance fra queste due stringhe è 5. Una distanza così bassa, considerando la lunghezza delle stringhe confrontate, ci dice che sostanzialmente le due melodie

sono uguali; infatti le modifiche fatte per ottenere la seconda melodia, a parte la trasposizione, sono minimali.

5.2.1 Problema della trasposizione

La Figura 5.3 mostra la melodia della Figura 5.2 trasposta nella tonalità di Re. La stringa che la descrive è "D4 E4 F♯4 D4 D4 E4 F♯4 F♯4 G4 A4 D4 E4 F♯4 G4 A4 A4 B4 A4 G4 F♯4 D4 A4 B4 A4 G4 F♯4 D4 E4 A4 D4 E4 F♯4 E4 D4". La edit distance fra le due stringhe è di 24. Considerando che le lunghezze delle 2 stringhe sono 31 e 34 caratteri, una edit distance così grande indicherebbe una differenza sostanziale fra le 2 melodie. Un valore così alto è però dovuto al fatto che le due melodie hanno tonalità diverse. Abbiamo visto prima che considerando la stessa tonalità la distanza fra le due stringhe è 5.

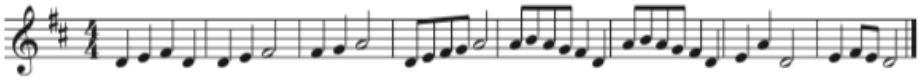

Figura 5.3: Melodia simile a Fra' Martino campanaro, in Re maggiore.

Vediamo un altro esempio, questa volta su un un caso reale di plagio, quello che ha visto contrapposto Al Bano a Michael Jackson.

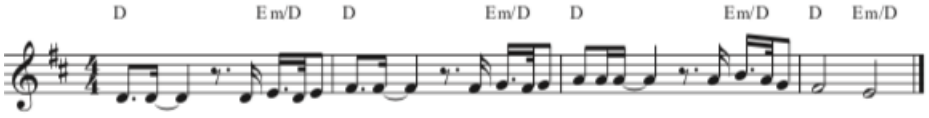

Figura 5.4: Will you be there, M. Jackson

La Figura 5.4 mostra la partitura della melodia del brano *Will you be there* di Jackson, la cui tonalità originale è Re maggiore. La stringa che rappresenta tale melodia è "D4 D4 D4 E4 D4 E4 F♯4 F♯4 F♯4 G4 F♯4 G4 A4 A4 A4 A4 B4 A4 G4 F♯4 E4"

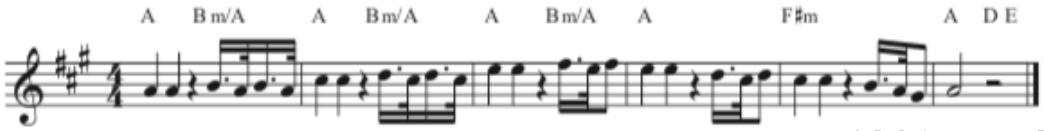

Figura 5.5: I cigni di Balaka, Al Bano

La Figura 5.5 mostra la partitura della melodia del brano *I cigni di Balaka* di Al Bano, la cui tonalità originale è La maggiore. La stringa che rappresenta tale melodia è "A4 A4 B4 A4 B4 A4 C♯5 C♯5 D5 C♯5 D5 C♯5 E5 E5 F♯5 E5 F♯5 E5 E5 D5 C♯5 D5 C♯5 C♯5 B4 A4 G♯4 A4". La distanza fra queste due stringhe è 27, pari alla lunghezza della stringa più lunga. Quindi queste 2 stringhe sono completamente diverse. Abbiamo già detto che occore confrontare i brani nella stessa tonalità.

La Figura 5.6 mostra il brano di Al Bano trasposto nella stessa tonalità, Re maggiore, del brano di Jackson. La stringa che rappresenta tale melodia del brano trasposto è: "D4 D4 E4 D4 E4 D4 F♯4 F♯4 G4 F♯4 G4 F♯4 A4 A4 B4 A4 B4 A4 A4 G4 F♯4 G4 F♯4 F♯4 E4 D4 C♯4 D4". La distanza fra le due stringhe ora è 11.

In modo analogo si potrebbe usare una rappresentazione testuale che codifica solo gli accordi. Il brano di Jackson diventerebbe "D Em/D D Em/D D Em/D D Em/D" mentre quello di Al Bano "D Em/D D Em/D D Em/D D Bm D G A". La distanza testuale fra queste 2 stringhe è

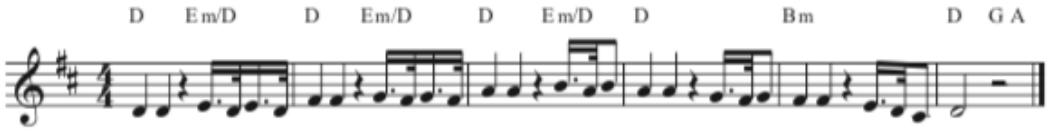

Figura 5.6: I cigni di Balaka trasposto in Re

4. Se consideriamo che gli ultimi 2 accordi del secondo brano sono solo di collegamento e quindi potremmo anche eliminarli, la seconda stringa diventerebbe "D Em/D D Em/D D Em/D D Bm D" e la distanza testuale sarebbe 2. Se i brani fossero stati considerati nelle tonalità originali la distanza sarebbe stata quella massima.

5.2.2 Altre rappresentazioni

La rappresentazione testuale che abbiamo considerato è forse la più semplice possibile in quanto l'unica informazione che viene inserita nella stringa è la sequenza delle note. Tutto ciò che va oltre tale sequenza, come ad esempio la durata di ogni singola nota, viene ignorato. Non dovrebbe sorprendere che una codifica così semplice funziona bene in alcuni casi ma non in tutti.

Ad esempio, una codifica più complessa potrebbe inserire anche informazioni riguardo la durata delle note oppure codificare altri aspetti della melodia. Vediamo alcune possibilità. Un problema comune a tutte le rappresentazioni è quello di poter usare più caratteri per rappresentare una singolo elemento della partitura mentre per usare gli algoritmi su stringhe dovremmo usare un solo carattere. In realtà allargando l'alfabeto è possibile usare un solo carattere ma si perde in leggibilità (umana, non per il computer). Dunque per comododitá i nostri "caratteri" saranno in realtà delle stringhe loro stessi, stringhe che chiameremo *meta*caratteri. Dal punto di vista logico però ogni sottostringa che rappresenta una singola nota deve essere considerata un solo carattere. Quindi, ad esempio, "A♭4", che rappresenta il La bemolle nella quarta ottava è un solo "(meta)carattere". Inoltre useremo lo spazio per seperare questi metacaratteri.

Rappresentazione NOTE**.** Questa è la rappresentazione che abbiamo già visto: ogni metacarattere è composta da una o o più lettere che ci dicono il nome della nota e eventuali alterazioni (♯ e ♭) e un numero che indica l'ottava in cui si trova. Magari possiamo aggiungere un carattere per rappresentare la pausa. Useremo la lettera "*p*". Come esempio, consideriamo la melodia riportata nella Figura 5.7. La sua rappresentazione NOTE è: *p* C5 B4 D5 C5 B4 Bb4 G♯4 A4 A4 A4 G♯4 B♭4 A4 A♭4 G4 E4 F4 F4 E4 F4 G4 A4 B♭4 D5 C5 B4 D5 C5 G4 A4 B4 C5 D5 C♯5 D5 E5 D5 B4 A4 G4 C5 B4 D5 C5 B4 B♭4 G♯4 A4 A4 A4 G♯4 B♭4 A4 G♭4 G4.

Rappresentazione PITCH**.** La rappresentazione NOTE funziona molto male quando due melodie simili sono presentate in tonalità diverse. Per aggirare questo problema possiamo pensare alla seguente rappresentazione che chiameremo PITCH e che da importanza agli intervalli fra note successive. Al posto di rappresentare le singole nota rappresentermo gli intervalli fra due noye successive misurandolo in semitoni. Anche in questa caso potremmo facilmente includere le pause rappresentadole con la lettera *p* La melodia della Figura 5.7 sarebbe rappresentata da: *p* -1 3 -2 -1 -1 -2 1 0 0 -1 2 -1 -1 -1 -3 1 0 -1 1 2 2 1 4 -2 -1 3 -2 -5 2 2 1 2 -1 1 2 -2 -4 -1 -2 5 -1 3 -2 -1 -1 -2 1 0 0 -1 2 -1 -3 1.

Rappresentazione PITCH-DA**.** Le precedenti rappresentazioni ignorano completamente le durate delle note. Vediamo ora una rappresentazione che invece include anche la durata. Definiamo l'alfabeto delle durate come $D = \{ w, h, q, e, s\}$, nel quale abbiamo messo dei simboli per

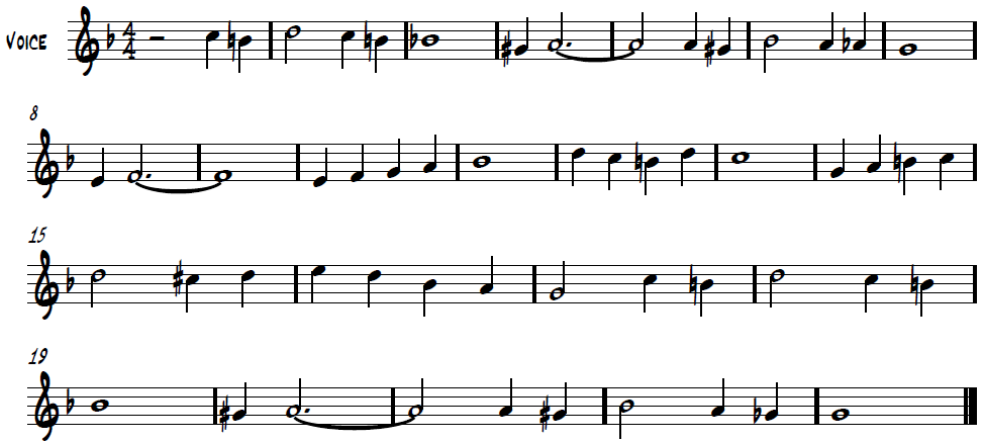

Figura 5.7: Frammento della melodia del brano *"Near you"*, di Francis Craig.

rappresentare le durate: w = *whole* (4/4), h = *half* (2/4), q = *quarter* (1/4), e = *eight* (1/8), e s = *sixteenth* (1/16). Possiamo aggiungere tale simbolo subito dopo l'intervallo, unendo così tale informazione con quella della rappresentazione PITCH. La rappresentazione PITCH-DA per la melodia della Figura 5.7 è: ph q -1q +3h -2q -1q -1w -2q +1h. 0h 0q -1q +2h -1q -1q -1w -3q +1h. 0w -1q +1q +2q +2q +1w +4q -2q -1q +3q -2w -5q +2q +2q +1q +2h -1q +1q +2q -2q -4q -1q -2h +5q -1q +3h -2q -1q -1w -2q +1h. 0h 0q -1q +2h -1q -3q +1w.

Rappresentazione PITCH-ND**.** L'utilizzo di durate "assolute" può creare un problema simile a quello della traspozione. Un brano che è semplicemente scritto con le note che durano tutte il doppio (o la metà) di un altro brano verrà considerato completamente diverso mentre in realtà sono uguali. Per ovviare a questo problema possiamo pensare a rappresentare le dirate in termine di un "massimo comun divisore". Più in dettaglio, possiamo pensare di avere un simbolo, "b", che rappresenta una durata di base e poi rappresentare le durate in termini di ripetizioni di questo simbolo in proporzione alla reale durata. Per stabilire il valore del simbolo di ripetizione (cioè quale durata rappresenta) possiamo analizzare preliminarmente la melodia da rappresentare e scegliere la durata più grande che permetta di rappresentare tutte le durate delle note e della pause della melodia come multipli di tale durata (appunto un massim comun divisore).

Per la melodia della Figura 5.7 il simbolo di ripetizione rappresenterebbe la durata di una semiminima e la stringa è: pbb b -1b +3bb -2b -1b -1bbbb -2b +1bbbbb 0b -1b +2bb -1b -1b -1bbbb -3b -2bbbbbbb -1b +1b +2b +2b +1bbbb +4b -2b -1b +3b -2bbbb -5b +2b +2b +1b +2bb -1b +1b +2b -2b -4b -1b -2bb +5b -1b +3bb -2b -1b -1bbbb -2b -1bbbbb 0b -1b +2bb -1b -3b +1bbbb.

Rappresentazione PITCH-NDT La rappresentazione PITCH-ND presenta però un problema: se nella melodia è presente una nota o pausa di una durata molto breve (si pensi ad esempio a una acciaccatura) o comunque se il "massimo comun divisore" risulta essere estremamente piccolo rispetto alla quasi totalità delle durte da rappresentare, la stringa risultante diventerebbe eccessivamente lunga. Per risolvere questo problema si potrebbe scegliere un "massimo comun divisore" più grande ignorando note di durata poco frequente.

Esercizi

1. Quale è la edit distance fra `ababca` e `cbaca`? Indicare anche la sequenza di operazioni che permettono di trasformare una stringa nell'altra usando un numero di operazioni pari alla edit distance.

2. La edit distance può essere usata come metrica di similitudine fra due stringhe. Rappresentando la musica con delle stringhe si può sfruttare la edit distance per fornire una metrica di similitudine fra due melodie. Tuttavia per rappresentare la musica è comodo usare più di un carattere per ogni nota. Come si può adattare la edit distance per gestire questo aspetto?

3. Definire una rappresentazione testuale che permetta di descrivere una melodia specificando le note; in particolare la rappresentazione deve considerare l'altezza delle note e la durata permettendo di rappresentare le durate di semibrevi, minime, semiminime, ... fino alle semibiscrome.

4. Si consideri la seguente rappresentazione testuale per descrivere delle melodie: una stringa di metacaratteri in cui ogni metacarattere ha una lettera per il nome della nota e un numero per l'ottava. Qualsi sono i vantaggi e gli svantaggi di una tale rappresentazione?

5. Definire una rappresentazione testuale che evita il problema della trasposizione.

6. Descrivere la rappresentazione testuale di una melodia che abbiamo denominato NOTE.

7. Descrivere la rappresentazione testuale di una melodia che abbiamo denominato PITCH.

8. Descrivere la rappresentazione testuale di una melodia che abbiamo denominato PITCH-DA.

9. Descrivere la rappresentazione testuale di una melodia che abbiamo denominato PITCH-ND.

10. Descrivere la rappresentazione testuale di una melodia che abbiamo denominato PITCH-NDT.