

Il problema del confronto fra stringhe "simili"

da <http://escher07.altervista.org>

In questo documento vediamo una implementazione di una tecnica per trovare all'interno di un dato archivio, gruppi di record contenenti informazioni simili per errore, che corrispondono cioè allo stesso soggetto fisico. Viene illustrata una procedura per individuare tali gruppi e quindi poter operare le correzioni necessarie a rendere più consistente (nel senso di più aderente alla realtà) l'archivio stesso.

Generalità

Capita spesso il problema di confrontare due stringhe, nel senso che specie durante il passaggio da un software gestionale ad un altro è buona norma "ripulire" per quanto possibile gli archivi che possono avere delle anagrafiche siffatte:

CODICE	DESCRIZIONE
1	BANCA DEL MONTE
2	B.CA DEL MONTE
3	B. DEL MONTE
4	BANCA DEL MONTE

... per indicare lo stesso soggetto. Ora, come individuare le descrizioni "simili" per creare un nuovo archivio corretto (in cui vi è un solo codice cliente per l'unico soggetto fisico, "Banca del Monte" ?

Una soluzione in proposito può venire dalla cosiddetta "distanza di Levenshtein" che è una funzione che date in input due stringhe restituisce il numero di operazioni elementari (inserimenti/cancellazioni o sostituzioni di caratteri) necessarie per passare dalla prima alla seconda e viceversa. Ad esempio detta $L(A,B)$ tale funzione si ha che:

$A = "abc"$

$B = "ax"$

$L = 2$

In quanto occorre un inserimento/cancellazione (in B, con cui $B \rightarrow "abx"$ o una cancellazione in a con cui $A \rightarrow "ab"$) ed una sostituzione (" $abx" \rightarrow "abc"$ nel primo caso, " $ab" \rightarrow "ax"$ nel secondo) per passare da una stringa all'altra.

Come implementare la funzione "distanza di Levenshtein"

L'algoritmo è abbastanza semplice con una serie di considerazioni. Definiamo una funzione distanza:

$d(i,j)$ = distanza fra stringa A troncata al i-esimo carattere e la stringa B troncata al j-esimo.

...e utilizziamo una forma ricorsiva per arrivare a $L(A,B) = d(n,m)$ dove n ed m sono le lunghezze di A e B basata su una matrice $(n+1) * (m+1)$. Riprendendo l'esempio precedente possiamo comporre una tabella come questa:

		0	i=1	2	3
	-	-	a	b	c
j=0	-	0	1	2	3
1	a	1	0	1	2
2	x	2	1	1	2

L'indice i varia in orizzontale, j in verticale. Gli incrementi di uno solo fra i e j sono inserimenti (dualmente le diminuzioni equivalgono a cancellazioni), gli incrementi di entrambi equivalgono a sostituzioni. Cominciamo a trovare i valori delle prime righe e colonne:

$d(0,0) = \text{confronto fra } A = "" \text{ e } B = "" \Rightarrow 0$
 $d(1,0) = \text{confronto fra } A = "a" \text{ e } B = "" \Rightarrow 1$
 $d(2,0) = \text{confronto fra } A = "ab" \text{ e } B = "" \Rightarrow 2$
 $d(3,0) = \text{confronto fra } A = "abc" \text{ e } B = "" \Rightarrow 3$
 $d(0,1) = \text{confronto fra } A = "" \text{ e } B = "a" \Rightarrow 1$
 $d(0,2) = \text{confronto fra } A = "" \text{ e } B = "ax" \Rightarrow 2$

Considerare che esiste la seguente corrispondenza:

$(i,j) \Leftrightarrow$ pattern dato dai primi i caratteri di A e dai primi j di B

Ci permette di riempire la tabella dalla quale otteniamo $L=d(3,2)=2$. Ora, come spiegare il tutto ad un calcolatore? Le assegnazioni precedenti sono meccaniche : sono infatti sempre il numero progressivo dei caratteri considerati di una delle due stringhe perché l'altra è nulla. Si tratta ora di esprimere i successivi $d(,)$ in termini di questi. Facciamo un esempio:

$(0,0) = \text{"null : null"} \quad d=0$
 $(1,0) = \text{"a : null"} \quad d=1$
 $(0,1) = \text{"null : a"} \quad d=1$
 $(1,1) = \text{"a : a"} \quad d=0$

$d(1,1) = d(0,0) + 1/0$ a seconda che il carattere 1 delle due le stringhe sia diverso o uguale;

$d(1,1) = d(1,0) + 1$, perché (1,0) + 1 carattere dà la stringa di (1,1)

$d(1,1) = d(0,1) + 1$, perché (0,1) + 1 carattere dà la stringa di (1,1)

di questi 3 possibili valori dobbiamo prendere il minimo perché la distanza di Levenstein è definita come il numero minimo di operazioni elementari. Generalizzando il tutto in termini di $i,j,i-1,j-1$ ecco il nostro algoritmo che ad esempio in C# si presenterà così:

```
int StringDistance(string str1, string str2)
{
    int n, m, i, j, cost, Dins, Dcan, Dsos, Dmin;
    string c1, c2;
    n = str1.Length;
    m = str2.Length;
    int [,] D = new int[n+1,m+1];
    for (i=0; i <= n; i++) D[i, 0] = i;
    for (j=0; j <= m; j++) D[0, j] = j;
    for (i = 1; i <= n; i++)
```

```

{
  for (j = 1; j <= m; j++)
  {
    c1 = str1.Substring(i-1, 1);
    c2 = str2.Substring(j-1, 1);
    if (c1 == c2)
    {
      cost = 0;
    }
    else
    {
      cost = 1;
    }
    Dins = D[i - 1, j] + 1;
    Dcan = D[i, j - 1] + 1;
    Dsos = D[i - 1, j - 1] + cost;
    //
    // Calcolo del Minimo
    //
    if (Dins < Dcan)
    {
      Dmin = Dins;
    }
    else
    {
      Dmin = Dcan;
    }
    if (Dsos < Dmin)
    {
      Dmin = Dsos;
    }
    D[i, j] = Dmin;
  }
}
return D[n, m];
}

```

E ora ?

La distanza di Levenshtein si calcola fra due elementi. Questo vuol dire che tramite tale funzione se N sono gli elementi del nostro archivio anagrafico \mathcal{A} , tenendo conto che $L(A,A)=0$ e $L(A,B)=L(B,A)$ l'applicazione di L in $\mathcal{A} \times \mathcal{A}$ determina la presenza di :

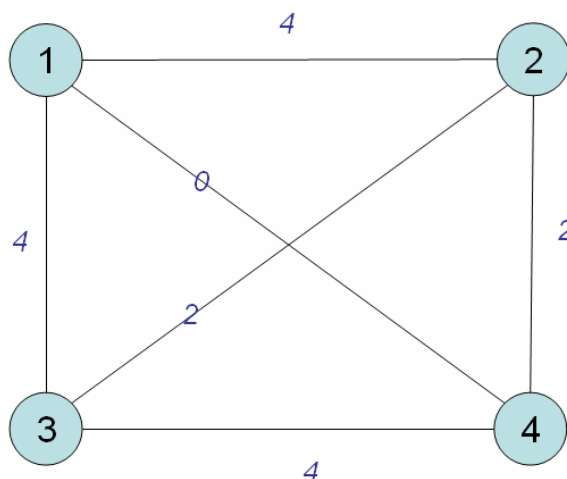
$$N * (N-1) / 2$$

Coppie di chiavi primarie con associato un valore di distanza. Nel nostro esempio avremmo una cosa del genere:

CODICE	CODICE	DISTANZA
1	2	2
1	3	4
1	4	0
2	3	2
2	4	2
3	4	4

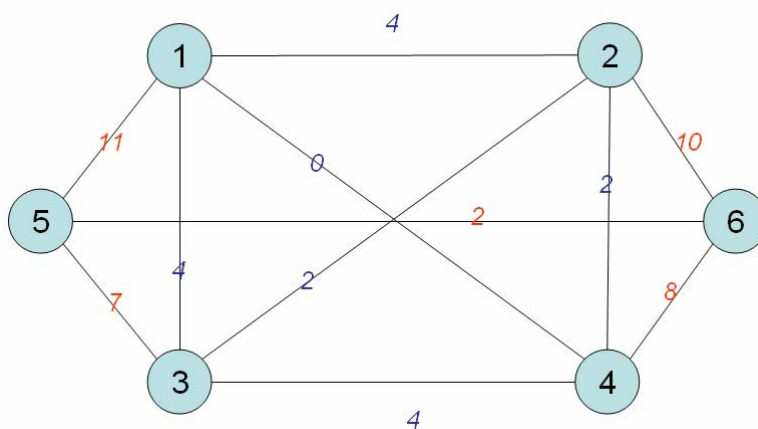
Tabella 1

Ovvero in pratica una struttura a grafo con gli archi pesati (i pesi sono le distanze) come la seguente:

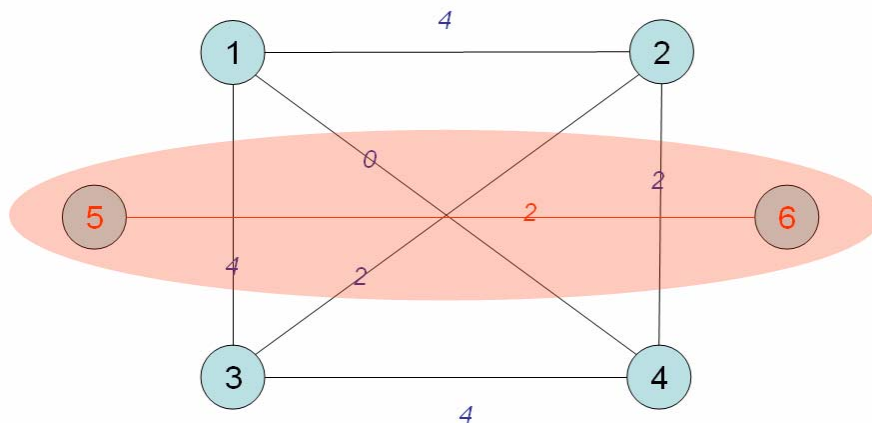


Tutto questo è molto bello, ma ricordiamoci che lo scopo iniziale era trovare un modo di raggruppare "punti" (cioè documenti, ovvero occurenze di CODICE nel nostro esempio) simili. Per cui sapere che tutto è connesso con tutto non dà assolutamente una mano, anzi.

Per capire come procedere poniamoci in un caso più reale, ovvero ipotizziamo di inserire nel nostro archivio altri due record corrispondenti ad un nuovo soggetto fisico (es. "Mario Rossi Spa" oltre alla nostra Banca). In questo caso si aggiungeranno al grafo dei nodi in corrispondenza dei quali i rami avranno pesi elevati tranne che fra loro stessi come in questa immagine:



In pratica, se mettiamo una soglia al valore della distanza, ovvero cancelliamo i rami il cui peso è oltre un certo valore (es. 5) il grafo precedente si spezza, ovvero rimangono solo dei gruppi ("contesti") di punti legati da archi, il cui contenuto è simile. La seguente immagine chiarisce un po' le cose:



Mettendo un valore limite alla distanza il grafo precedente dà luogo alla seguente tabella:

Contesto	Codice	Distanza
0	1	2
0	2	2
0	3	4
0	4	0
1	5	2
1	6	2

Tabella 2

La distanza è indicativa e ci ritorneremo più avanti.

A parte questo, sappiamo dall'analisi che ci sono buone probabilità che i codici 1,2,3,4 rappresentino lo stesso soggetto e così i 5,6. Questo dà una mano significativa in problemi di questo tipo, specie se i dati sono molti.

Implementazione della tabella dei contesti

Supponiamo di aver riempito la Tabella 1. In pratica si tratta di ciclare su se stesso l'archivio \mathcal{A} eliminando le coppie con chiavi uguali e quelle già presenti a chiavi invertite calcolando per ogni coppia il valore di $L(A,B)$ delle corrispondenti descrizioni A,B.

Il punto è trasformare la Tabella 1 nella Tabella 2 cosa che possiamo fare con una procedura come questa:

```
private void TrovaContesti()
{
    bool RigheVuote = true;
    int Contesto = -1;
    string currCntx1, currCntx2;
    while (RigheVuote == true)
    {
        Contesto++;
        AssegnaContesti(Contesto, "root");
    }
}
```

```

RigheVuote = false;
foreach (DataRow row in TbContesti1.Rows)
{
    currCntx1 = row["Contesto 1"].ToString();
    currCntx2 = row["Contesto 2"].ToString();
    if ((currCntx1 == "") && (currCntx2 == ""))
    {
        RigheVuote = true;
    }
}
}
}

```

Dove TbContesti1=Tabella1. In pratica questa procedura scorre la tabella, cercando di assegnare un contesto con un certo indice (0,1,2 ...). Il tutto dura finchè c'è almeno una riga a cui non è stato assegnato alcun contesto. Materialmente l'assegnazione avviene tramite la seguente procedura:

```

private void AssegnaContesti(int IDContesto, string DocPadre)
{
    string CurrCont1, CurrCont2;
    string CurrDoc1, CurrDoc2;
    foreach (DataRow row in TbContesti1.Rows)
    {
        CurrCont1 = row["Contesto 1"].ToString();
        CurrDoc1 = row["Documento 1"].ToString();
        CurrCont2 = row["Contesto 2"].ToString();
        CurrDoc2 = row["Documento 2"].ToString();
        //
        if ((CurrCont1 == "") || (CurrCont2 == ""))
        {
            if (DocPadre == "root")
            {
                if (CurrCont1 == "")
                {
                    DocPadre = CurrDoc1;
                }
                if (!(CurrCont1 == "") && (CurrCont1 == ""))
                {
                    DocPadre = CurrDoc2;
                }
            }
            if (DocPadre != "root")
            {
                if ((CurrCont1 == "") && (CurrCont2 == "") && ((DocPadre
== CurrDoc1) || (DocPadre == CurrDoc2)))
                {
                    CurrCont1 = IDContesto.ToString();
                    CurrCont2 = IDContesto.ToString();
                    row["Contesto 1"] = CurrCont1;
                    row["Contesto 2"] = CurrCont2;
                    AssegnaContesti(IDContesto, CurrDoc1);
                    AssegnaContesti(IDContesto, CurrDoc2);
                }
            }
        }
    }
}
}
}

```

Anche questa spazzola la tabella in questione utilizzando queste considerazioni:

- Se il documento padre (ovvero quello del quale si cercano altri documenti appartenenti allo stesso contesto) è "root" il documento padre diventa il primo non appartenente a contesto della prima riga non targata.
- Le righe, ovvero le coppie di documenti, vengono aggiunte nella loro interezza ad un contesto. Cioè basta che uno dei due documenti sia uguale al padre che tutti e due vengono aggiunti.
- L'idea di base è prendere la prima riga non targata, targare i due membri con lo stesso contesto e poi cercare le altre righe associabili allo stesso contesto (ovvero che contengono o il primo o il secondo documento) in modo ricorsivo.

Con questo sistema non si arriva proprio ad una tabella come la Tabella 2 ma ad un qualcosa che ha le righe formate dai seguenti campi:

- ID Documento 1
- ID Documento 2
- Contesto 1
- Contesto 2
- Distanza

Questa sarà trasformata nella forma desiderata semplicemente trasformando l'orientamento da orizzontale a verticale, in modo da avere solo un documento per riga. Ovvero con un'istruzione tipo:

```
SELECT ID Documento 1, Contesto 1, AVG(Distanza)
FROM TbContestil
GROUP BY Documento 1, Contesto 1
UNION
SELECT ID Documento 2, Contesto 2, AVG(Distanza)
FROM TbContestil
GROUP BY Documento 2, Contesto 2
```

In questo modo la Distanza passa da valore di coppia (in quanto il valore su una coppia viene assegnato a tutte e due le righe generate) a valore per documento (in quanto ogni documento compare una sola volta ed in un solo contesto nella Tabella 2). Che significato ha questo valore? E' in pratica un'approssimazione della distanza dal centro ideale del contesto in cui il codice/documento in questione si trova, ottenuto come media della distanze sulle coppie che coinvolgono il punto (documento) in questione all'interno del contesto di cui fa parte. La cosa che si capisce considerando questa tabella :

CODICE	CODICE	DISTANZA
1	2	2
1	3	4
1	4	0
2	3	2
2	4	2
3	4	4

... e come sarà aggregata. Essa, innanzitutto, contiene tutti codici dello stesso contesto perché tutte le righe contengono il codice 1,2 o codici (es. 3,4) che compaiono in righe dove sono 1 o 2.

Trasformando la tabella in questione in "verticale" il documento 1 si trova associato a 3 distanze, corrispondenti a quelli degli altri elementi del gruppo di cui fa parte, ovvero 2,4,0 la cui media (6/3) è 2.0. Procedendo allo stesso modo anche per gli altri arriviamo ad un risultato come questo:

CODICE	DISTANZA
1	2.0
2	4.0
3	4.0
4	2.0

Che illustra quanto si diceva.

Note sull'implementazione

Il codice con cui praticamente si realizza l'assegnazione dei contesti è di certo molto ottimizzabile, basta pensare al fatto che ad ogni passaggio si analizza tutta la tabella mentre basterebbe scorrerne solo una parte. L'ottimizzazione sarebbe oltremodo necessaria specie per archivi con molti record.

Nel caso specifico in cui l'algoritmo è stato utilizzato già questa realizzazione consentiva tempi più che accettabili, sicché non mi sono impegnato troppo in tale direzione. Visto poi il modo in cui questo codice è nato (ovvero non come software autonomo ma come parte di un modulo per la gestione delle liste elettorali e delle sottoscrizioni, sviluppato per un comune nell'ambito di un gestionale a tecnologia web) non è stato possibile dare un esempio funzionante da poter scaricare, ma solo prendere i pezzi essenziali e metterli su un documento, spiegando soprattutto le logiche.

A voi quindi la palla ☺.