Linguaggio R
Ripasso

Matteo Re

April 27, 2017

Obiettivo: ripasso pre incontro progetto esame
Argomenti trattati: controllo flusso, cicli e funzioni

1 Obiettivi

- Ripasso controllo di flusso (in particolare if)
- Ripasso cicli
- Ripasso costruzione funzioni
- Introduzione al progetto per l'esame

In questa breve dispensa **non** verranno riprese le strutture dati di R. Per esse fare riferimento alle slide disponibili nella home page del corso. Le strutture dati coinvolte nel progetto d'esame sono le seguenti:

- Matrici
- Vettori
- Occasionalmente dataframe

2 Ripasso controllo di flusso

Un pc non è in grado di prendere decisioni autonome. Quelle che definiamo genericamente "decisioni" sono operazioni che si basano sulla verifica di una determinata condizione (la condizione può essere molte cose ma, comunemente, si tratta di verificare se una determinata variabile ha un valore che risponde ad un criterio fissato a priori, ad esempio se la somma dei valori di una colonna di una matrice è maggiore di un certo numero o quanti degli elementi di un vettore sono minori di un certo numero).
In molti linguaggi di programmazione (e anche in R) c'è una parola chiave che permette di:

- Verificare se la condizione è vera o falsa (nel senso che il risultato di questa verifica **deve** restituire sempre **uno ed un solo** valore logico TRUE o FALSE
- eseguire un blocco di istruzioni (può essere anche contenere una sola istruzione ma non ci sono vincoli riguardo al numero di istruzioni da eseguire. Questo è il motivo per cui si parla genericamente di un **blocco** di istruzioni) solo nel caso in cui il valore logico ottenuto al punto precedente sia TRUE. Questo blocco è **obbligatorio**.
- eseguire un blocco di istruzioni (può essere anche contenere una sola istruzione ma non ci sono vincoli riguardo al numero di istruzioni da eseguire. Questo è il motivo per cui si parla genericamente di un **blocco** di istruzioni) solo nel caso in cui il valore logico ottenuto al punto precedente sia FALSE. Questo blocco è **opzionale**.

La struttura della sintassi di if (ossia il modo in cui utilizziamo if in R) risponde alla struttura logica appena scritta in forma di elenco ed è la seguente:

```
if(testlogico){
    istruzioni da eseguire se testlogico restituisce TRUE
}else{
    istruzioni da eseguire se testlogico restituisce FALSE
}
```

Quando scegliamo di **non** utilizzare il blocco di istrtuzioni per il caso FALSE allora dobbiamo rimuovere tutto da *else* in poi:

```
if(testlogico){
    istruzioni da eseguire se testlogico restituisce TRUE
}
```

Ora vediamo degli esempi che mettono in evidenza alcuni punti fondamentali dell'utiliddo fi if. Troverete degli esempi di codice. Alcuni di essi restituiscono dei warning o degli errori (servono a mettere in evidenza errori comuni).

2.1 Esempio 1.1:

```
a <- 5
if(a<3){
    print("a      minore di 3.")
}
```

Se provate ad eseguire il codice di questo esempio ... non succede niente. E questo non è un errore. R non fa nulla perchè il test logico restituisce FALSE e noi abbiamo specificato solo il blocco di istruzioni da eseguire quando il test logico (che in questo esempio $a < 3$) restituisce TRUE.

Provate a rieseguire l'esempio dopo aver assegnato ad a il valore 2. Questa volta R scriverà che a è minore di 3. In definitiva ... quando R non fa niente questo non indica necessariamente un errore. Tutto dipende da cosa gli abbiamo chiesto di fare.

2.2 Esempio 1.2 (uso improprio di if):

Supponiamo di avere un vettore composto da 10 elementi ognuno corrispondente ad un numero reale campionato dalla distribuzione uniforme (possiamo ottenere un vettore così utilizzando la funzione R *runif*). Cosa succede se effettuiamo un test logico su questo vettore? Ad esempio:

```
a <- runif(10)
a > 0.5
```

R *vettorizza*. Questo vuol dire che, dato che l'elemento a monte del simbolo > è un vettore, R applica il test non solo al primo elemento del vettore ma (separatamente) ad ogni elemento presente nel vettore. Questo genera un numero di riswultati pari al numero di elementi presenti nel vettore a. E, quindi, il risultato non è più un singolo valore logico ma un vettore di valori logici.
A volte ci dimentichiamo della vettorizzazione e scriviamo qualcosa di simile:

```
> a <- runif(10)
> a
 [1] 0.4882013 0.4963132 0.8465271 0.9013782 0.2309184 0.8182265 0.5275784
 [8] 0.9092602 0.7747877 0.3006986

> if(a>0.5){
+ print("a    maggiore di 0.5")
+ }else{
+ print("a    minore o uguale a 0.5")
+ }

[1] "a    minore o uguale a 0.5"

Warning message:
In if (a > 0.5) { :
  la condizione la lunghezza > 1 e solo il promo elemento verra' utilizzato
```

In questo caso R:

- Emette un risultato
- Avvisa (mediante il Warning) che if è progettato per lavorare con un solo valore logico (si aspetta che tutto ciò che sta tra le parentesi tonde produca un solo valore logico) e che, quindi, userà solo il primo valore del vettore

a). E qui vediamo che il risultato fornito da R è corretto. Scrive che a (in realtà non tutto il vettore ma solo il primo valore) è minore di 0.5. In effetti il valore del primo elemento di a è 0.4882013.

In definitiva ... qualsiasi test logico in grdo di fornire un solo valore logico è adatto all'utilizzo in if.
Ricordate che il risultato di un test logico si può invertire scrivendo !(testlogico) all'interno delle parentesi tonde di if. Nel blocco di codice (o nei blocchi di codice se scriviamo codice anche per il caso negativo (FALSE), possiamo mettere ciò che vogliamo. Non è obbligatorio che nei blocchi di codice sia presente la variabile o le variabili presenti nel test logico presente tra le parentesi tonde.

2.3 Esempio 1.3:

Commentare il seguente codice R.

```
a <- 0

if(a>0){
    b <- 1/a
}else{
    b <- "Infinito"
}

b
```

Quale è il valore di b dopo che il codice è stato eseguito?

2.4 Esempio 1.4:

Il seguente codice è funzionalmente equivalente a quello dell'esercizio precedente? Motivate la risposta. Quale è il valore di b dopo che il codice è stato eseguito?

```
a <- 0
b <- "Infinito"

if(a>0){
    b <- 1/a
}

b
```

3 Ripasso cicli

I icli in R hanno sostanzialmente una struttura sintattica molto simile a *if*, ossia un meccanismo di controllo e (questa è una differenza importante) **un solo** blocco di codice. Lo scopo del mecanismo di controllo, a diffenenza del caso di **if**, non è quella di decidere se l'unico blocco di codice tra parentesi graffe va eseguito ma, piuttosto, decidere *quante volte* dovrà essere eseguito. E qui iniziamo a distinguere due casi principali:

- Il blocco di codice tra parentesi graffe viene eseguito un numero di volte noto a priori (ad esempio tante volte quanti sono gli elementi presenti in un vettore)
- Il blocco di codice tra parentesi graffe viene eseguito un numero di volte non noto a priori. Il ciclo si ripete finchè il valore restituito da un determinato test logico è TRUE. In questo caso possono verificarsi problemi.In particolare se il contenuto del blocco di codice da eseguire più volte **non influisce** sul risultato del test logico che costituisce l'elemento principale del meccanismo di controllo e, al momento dell'ingresso nel ciclo, il risultato del test logico è TRUE allora ci troveremo bloccati per sempre nel ciclo. In questo caso si parla di *ciclo infinito*.

In questo ripasso vedremo in particolare due tipi di ciclo che sono esempi del primo (for) e del secondo caso (while).. Partiamo con il ciclo for.

4 Il ciclo for

Questo tipo di ciclo serve per effttuare un determinata operazione "per ogni elemento di un insieme dato". Detto così può suonare criptico. Per chiarire il tutto basta pensare che il meccanismo di controllo di un ciclo for è basato sull'utilizzo di un *iteratore* ossia di ua variabile che assume tutti i valori presenti in un insieme. Uno alla volta. Quindi si dice che *itera* attraverso il contenuto dell'insieme. Questo ha, dal punto di vista della programmazione, diverse applicazioni.
Ad esempio se so a priori che voglio ripetere il codice contenuto nel blocco di istruzioni 10 volte allora posso usare R per dire di ripetere il codice

```
for(i in c(1:10) ){
     ... qui metteremo le istruzioni che vogliamo ripetere 10 volte
}
```

Noi sappiamo che

```
c(1:10)
```

restituisce un vettore contenente tutti i numeri interi da 1 a 10. Il primo valore che assume la variabile i è 1 ... e il blocco di codice viene eseguito una prima

volta. Poi i assume il valore 2, e il blocco di codice viene eseguito una seconda volta. E il tutto procede così fino a quando i assume l'ultimo valore nell'insieme (ossia nel vettore) che poi è il valore 10.
Non è obbligatorio utilizzare nel blocco di codice il valore di i (il valore *corrente* dell'iteratore. Ma possiamo farlo se questo è utile. In questo caso la prima volta che il blocco di codice viene eseguito i vale 1, la seconda 2 e così via. Potremmo sfruttare questo meccanismo per stampare i quadrati di tutti i valori assunti dall'iteratore i.

```
> for(i in c(1:5)){
+       print(i^2)
+ }

[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

Qui notiamo un fatto importante: non si tratta di una vettorizzazione. Infatti non otteniamo un unico vettore composto da 5 elementi ma 5 vettori ognuno composto da 1 solo elemento. E perchè? Perchè l'istruzione di stampa del quadrato viene eseguita *separatamente* 5 volte ... una per ogni volta che i cambia valore. Al contrario:

```
> a<-c(1:5)
> a
[1] 1 2 3 4 5

> a^2
[1]  1  4  9 16 25
```

Se eleviamo al quadrato un vettore (in questo esempio composto da 5 elementi) otterremo un vettore di 5 elementi contenenti i quadrati dei valori del vettore originario (nel nostro ultimo esempio il vettore a). Tuttavia, per dimostrare che non siamo obbligati ad usare il valore dell'iteratore nel blocco di codice ripetuto possiamo chiedere a R di fare qualcosa che non implichi l'utilizzo della variabile i. Ad esempio:

```
> for(i in c(1:5)){
+     print("Ciao")
+ }

[1] "Ciao"
[1] "Ciao"
[1] "Ciao"
[1] "Ciao"
[1] "Ciao"
```

Il tutto funziona comunque. Come abbiamo detto in precedenza questo tipo di ciclo non è pericoloso ... nel senso che non è in grado di bloccare R entrando in un ciclo senza uscita (un ciclo infinito). Ora passiamo all'esempio di ciclo del secondo tipo, il ciclo *while*.
Ora spendiamo due parole sull'iteratore. E' possibile usarlo per "attraversare" diversi tipi di insiemi. Ad esempio nel prossimo esempio lo useremo per calcolare la media di tutte le righe di una matrice. Facciamo attenzione all'utilizzo di **dim()** sulla matrice per ottenere un vettore contenente tutti i valori che vanno da 1 al numero di righe della matrice.

```
> m <- matrix(runif(28), nrow=7)
> m
             [,1]       [,2]       [,3]       [,4]
[1,] 0.385278794 0.1361316 0.40890931 0.60881909
[2,] 0.305008054 0.1357846 0.91546769 0.97737884
[3,] 0.008419127 0.4251276 0.34553198 0.91832350
[4,] 0.339603662 0.7100697 0.81789095 0.52650640
[5,] 0.079050406 0.5734088 0.01679654 0.01052112
[6,] 0.889200523 0.2316997 0.83365913 0.98897193
[7,] 0.410919182 0.5546843 0.57876197 0.34434455

> dim(m)
[1] 7 4

> for(i in c(1:dim(m)[1])){
+     print(mean(m[i,]))
+ }
[1] 0.3847847
[1] 0.5834098
[1] 0.4243506
[1] 0.5985177
[1] 0.1699442
[1] 0.7358828
[1] 0.4721775
```

E' possibile fare cose complesse con un ciclo for. Tutto dipende da cosa inseriamo nel blocco di codice da ripetere.

5 Il ciclo while

Il meccanismo di controllo di un ciclo while (sempre rappresentato dal contenuto delle parentesi tonde come nel caso di if e del ciclo for) è diverso da quello del ciclo for. In questo caso il blocco di codice viene ripetuto fino a quando il risultato del *test logico* contenuto tra le parentesi tonde è TRUE. Questo ha la conseguenza che, se il valore del test logico è FALSE in partenza allora il blocco di codice non verrà eseguito mai. Come nel caso del ptorrimo esempio. In esso

il ciclo esegue il codice *se e solo se* il valore di a è minore di 5. Ma noi prima del ciclo, impostiamo il valore di a a 10. Quindi il blocco di codice non sarà mai eseguito.

```
> a<-10
> while(a<5){
+     print("Ciao")
+     a <- a-1
+ }
>
```

Tuttavia se cambiamo il significato del test logico (invece di usare $a < 5$ usiamo $a > 5$ allora esistono le condizioni per poter entrare nel ciclo, come dimostra ils eguente esempio:

```
> a<-10
> while(a>5){
+     print("Ciao")
+     a <- a-1
+ }
[1] "Ciao"
[1] "Ciao"
[1] "Ciao"
[1] "Ciao"
[1] "Ciao"
```

Tuttavia è necessario dare attenzione ad un particolare: nel blocco di codice da eseguire **deve** essere presente qualcosa che fa variare il valore di a (la variabile coinvolta nel test logico utilizzato nel meccanismo di controllo del ciclo while). Altrimenti riusciamo ad entrare nel ciclo ma non potremo più uscirne. Fate una prova. Eseguite lo stesso ciclo dell'ultimo esempio dopo aver levato la riga

```
a <- a-1
```

(che decrementa il valore di a ad ogni esecuzione del blocco di codice). In questo caso R non si ferma e non scrive nessun risultato. Il motivo è che R aspetta di aver finito l'esecuzione del ciclo prima di stampare i risultati ottenuti. Ma in questo caso non finisce mai. L'unico modo per bloccare il tutto (e ottenere parte dell'output) è quello di premere il pulsante STOP (quello che interrompe la computazione corrente).

NB: sia nel caso degli esempi sul ciclo for che nel caso degli esempi sul ciclo while non ho focalizzato l'attenzione sul contenuto del blocco di codice da ripetere ma, piuttosto, sulla natura del tipo di ciclo.

6 Funzioni in R

Possiamo pensare alle funzioni come blocchi di codice specializzati per:

- Accettare in input variabili
- Restituire valori ottenuti manipolando le variabili ricevute in inpu
- Essere riutilizzabili facilmente

Le funzioni definite dall'utente in R si creano grazie alla funzione **function()**. Prima di presentare l'utilizzo della funzione function() proviamo a riesaminare le caratteristiche di una generica funzione creata dall'utente in R. Essa deve essere in grado di accettare variabili in input e quindi la funzione function() deve permettere di specificare quante variabili saranno necessarie per utilizzare la nostra funzione e definire dei nomi con cui riferirci ad esse. Dato che lo scopo finale quando creiamo una funzione è quello di riutilizzare spesso del codice che troviamo utile laprima domanda da porsi è la seguente: dove lo scriviamo il codice da eseguire? E potrà essere composto da più di una riga di codice (ovviamente si). Quindi è lecito aspettarsi la presenza di un blocco di codice (quello che verrà eseguito ogni volta che utilizzeremo la funzione). Infine la funzione dovrà essere in grado di restituire un valore (es. un singolo numero, un vettore, una lista ecc.) a chi ha invocato la funzione. Per ottenere questo risultato dovremo utilizzare la funzione **return(valoredarestituire)** all'interno del blocco di codice. Ma attenzione! L'utilizzo di return() causa l'uscita immediata dalla funzione. Quindi va usata solo quando siamo sicuri di non avere altre istruzioni R da eseguire. Ad esempio quando abbiamo completato il calcolo del valore da restituire e non prima. Infine, se vogliamo essere in grado di riutilizzare una funzione essa deve avere come minimo un nome con cui riferirsi ad essa (per poterla invocare).
Lo schema generale di funzione creata dall'utente in R è il seguente:

```
nomemiafunzione <- function(argomenti separati da virgole){

      ... qui ci sar\'a il codice R che permetter\'a di manipolare le
          variabili fornite in input (il loro numero \'e uguale a quel-
          lo degli argomenti) e mettere il risultato del calcolo in una
          variabile (ad es di nome risultato)

      return(risultato)
}
```

Una volta definita la nuova funzione sarà possibile utilizzarla per effettuare calcoli (a volte molto complessi). Vediamo un esempio. Creeremo una semplicissima funzione avente nome *perdue* che, come potete immaginare, accetta in input un valore, lo moltiplica per due, e restituisce il risultato.

```
> perdue <- function(valoreinput){
+     return(valoreinput * 2)
+ }

> perdue
```

```
function(valoreinput){
    return(valoreinput * 2)
}

> perdue(3)
[1] 6
```

Dato che il calcolo effettuato è davvero semplice (una moltiplicazione per una costante) il calcolo è stato effettuato direttamente tra le parentesi tonde di return(). Se il calcolo fosse stato più complesso (e avesse richiesto più righe di codice) questo non sarebbe stato possibile.
Se avessimo avuto bisogno di ulteriori variabili (magari per salvare risultati intermedi del calcolo) avremmo potuto crearle nel blocco istruzioni (quello racchiuso tra graffe) e avremmo potuto usarle. Le variabili create dentro ad una funzione esistono solo al suo interno. Inoltre è bene ricordare che è meglio evitare di creare nuove variabili con i nomi degli argomenti. Le variabili associate agli argomenti esistono comunque all'interno della funzione ma il loro scopo è quello di prelevare variabili dall'esterno e renderle disponibili all'interno della funzione (nell'esempio appena visto questo è il casdo della variabile valoreinput). Quindi, onde evitare confusione e sovrascritture non volute di dati di input) vale la regola che possiamo dare alle variabili create all'interno della funzione tutti i nomi che vogliamo **ma non quelli già assegnati agli argomenti**.
Notiamo infine che se scrivo il nome della funzione appena creata *senza* le parentesi tonde , R mostrerà il suo codice sorgente (quello che abbiamo creato noi usando la funzione function()). Per riutilizzare una funzione scriveremo il suo nome apriremo la parentesi tonda, e forniremo tutti gli argomenti richiesti *sempre separati da virgole*. Poi chiudiamo la parentesi tonda e premiamo invio. Il codice verrà eseguito e R fornirà il risultato che potremo visualizzare o salvare in una variabile per futuri utilizzi.
Proviamo ad estendere la nostra funzione in modo da poter passare due valori (in modo che faccia sempre una moltiplicazione ma non per una costante, bensì per un valore fornito dall'utente della funzione).

```
> moltiplica <- function(valoreinput1, valoreinput2){
+   return(valoreinput1 * valoreinput2)
+ }

> moltiplica(5,2)
[1] 10
```

Come potete vedere il risultato calcolato dalla nuova funzione moltiplica() è corretto. Non ci sono limiti a quello che potete fare nel blocco di istruzioni della funzione. Tuttavia, a parte rari casi, è insensato non elaborare i dati ricevuti in input. Altrimenti una funzione a cosa servirebbe? Ma ci sono eccezioni. Immaginate una funzione che scriva la data odierna. Avrebbe senso passargli dei valori di input? Il PC lo sa che giorno è oggi. Quindi non avrebbe bisogno di informazioni aggiuntive. Per convincervi provate a usare la funzione date(). Un

altro esempio che ci è più familiare è la funzione ls().
Un punto importante è tenere a mente che alla funzione return() è possibile passare qualsiasi cosa. POtreste fare calcoli estremamente complessi e costruire una lista di nome listarisultati e metterci dentro matrici, dataframe (anche più di uno) vettori e tutto quello che vi viene in mente. Poi, se scriviamo return(listarisultati) la lista, e tutto il suo contenuto, viene restituito all'esterno.
Riassumendo. Ci sono due flussi di informazioni. Gli argomenti portano i dati dall'esterno all'interno della funzione. La funzione return() restituisce i risultati all'esterno. Tra di essi c'è il blocco di istruzioni che può contenere tutto il codice che volete.

Ultima domanda importante da porsi. Dove le salviamo le funzioni? E come le rendiamo nuovamente disponibili? Di solito si crea un file di testo (ad esempio miefunzioni.txt) contenenti tutte le funzioni che ci sembrano utili. Poi ci si posiziona nella cartella che contiene il file (verificate utilizzando la funzione dir()) e si scrive:

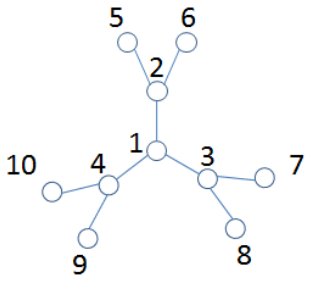
```
> source("miefunzioni.txt")
```

Se tutto va bene non succede nulla (apparentemente). Nel senso che R non da errori. E le vostre funzioni (nel file potete metterne più di una) sono tornate disponibili. Per verificare provate a usare ls() (che mostra anche le funzioni. Si lo so che sembra strano ma ls oltre alle normali variabili lista anche le funzioni disponibili).

7 INTRODUZIONE PROGETTO ESAME:

In questa verrà presentata una tecnica di apprendimanto automatico che contense di fare predizioni riguardanti caratteristiche di farmaci e, più in generale, molecole.
Inizieremo con la spiegazione della tecnica dei cammini aleatori su grafo. In seguito vedremo come applicarla in un'analisi bioinformatica riguardante una collezione di molecole approvate per l'utilizzo come farmaci da FDA.

7.1 Cammini aleatori su grafo



Un grafo (o rete) una struttura che, matematicamente, viene utilizzata per descrivere un insieme di oggetti e le relazioni tra di essi. Gli oggetti possono essere qualsiasi cosa, ad esempio persone in contatto in una rete sociale, le pagine web su Internet ... molecole di n set di farmaci approvati da FDA (si, proprio quelle che abbiamo visto nei lab precedenti). Sempre dal punto di vista matematico un grafo è un insieme di vertici (nodi) connessi da archi (collegamenti tra nodi) e si indica con $G = (V, E)$. Questo indica che il grafo G è composto da due cose: un insieme di nodi V ed un insieme di archi E. Si usa la E perchè in inglese arco (nel senso delle reti e dei grafi) si dice *edge* e indica il collegamento tra due nodi.
Nel grafo di esempio di cui trovate l'immagine all'inizio di questa sezione ci sono 10 nodi e 9 archi (collegamenti tra nodi).
In R V (l'insieme di nodi) e E (l'insieme degli archi) sono strutture dati di tipo diverso. Per V una volta che assegno ad ogni nodo un identificativo univoco (ad es. da 1 a 10) sono in grado di accedere ad ogni nodo senza ambiguità. Quindi, è come dire che la struttura dati che ospita le proprietà dei nodi è **unidimensionale** e, quindi, è un **vettore**.
Al contrario la struttura dati R per l'insieme di archi (collegamenti tra nodi) E non può essere un vettore. Vediamo perchè. E non esprime caratteristiche di singoli oggetti ma relazioni *tra coppie di oggetti* (coppie di nodi in questo caso). Quindi se, ad esempio, in V (il set dei nodi) ci sono 10 nodi allora le dimensioni di E saranno 10×10 ossia tutte le relazioni tra tutte le possibili coppie di nodi. In definitiva E ha non una ma **due** dimensioni ... e quindi non è un vettore ma una matrice.
Nella matrice E abbiamo un numero di righe pari al numero dei nodi presenti in V ed un numero di colonne pari al numero di nodi presenti in V. Quindi E è sempre una matrice quadrata. Se all'incrocio tra la riga 2 e la colonna 6 troviamo un numero diverso da 0 (ad esempio un 1) questo indica che tra il nodo 2 ed il nodo 6 c'è un collegamento. Ma attenzione: se c'è un collegamento tra il nodo 2 ed il nodo 6 allora **deve** essere vero anche il contrario ... e quindi troveremo **lo stesso valore** all'incrocio della riga 6 con la colonna 2. Questo ecquivale a dire che il collegamento lo vediamo sia se guardiamo dal nodo 2 che

Figure 1: Creazione matrice incidenza I (rappresenta E).

```
> I <- matrix(rep(0,100), nrow=10)
> I[1,2]=1
> I[2,1]=1
> I[1,3]=1
> I[3,1]=1
> I[1,4]=1
> I[4,1]=1
> I[2,5]=1
> I[5,2]=1
> I[2,6]=1
> I[6,2]=1
> I[3,7]=1
> I[7,3]=1
> I[3,8]=1
> I[8,3]=1
> I[4,9]=1
> I[9,4]=1
> I[4,10]=1
> I[10,4]=1
> sum(I)
[1] 18
>
```

dal nodo 6. Questo comporta che il grafo disegnato all'inizio di questa sezione (che ha 9 collegamenti tra nodi) sarà descritto da una matrice di 10×10 elementi e, di questi 100 elementi, solo 9×2 (18) elementi saranno diversi da 0.
Facendo un passo avanti con l'immaginazione potremmo dire che, invece di avere collegamenti tutti con lo stesso peso (nel nostro esempio 1) ogni collegamento può avere un suo peso (un suo numero) che, se più alto, indica un collegamento più forte tra i nodi uniti dall'arco. Questo è esattamente il caso della nostra matrice di similarità tra farmaci. Essa può essere vista come l'insieme dei pesi dei collegamenti di un grafo (una rete) che descrive la similarità tra tutti i possibili farmaci di un insieme.

7.1.1 Cammini aleatori su grafo ... in R

Proviamo a costruire in R la matrice E del grafo diregnato a pagina 12. Sappiamo che richiede 100 elementi (Figura 1).
Se ispezioniamo il contenuto della matrice troviamo quanto segue: Se osserviamo le posizioni degli 1 nella matrice (in termini di riga e colonna) notiamo che

Figure 2: Contenuto matrice I.

```
> I
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    0    1    1    1    0    0    0    0    0     0
 [2,]    1    0    0    0    1    1    0    0    0     0
 [3,]    1    0    0    0    0    0    1    1    0     0
 [4,]    1    0    0    0    0    0    0    0    1     1
 [5,]    0    1    0    0    0    0    0    0    0     0
 [6,]    0    1    0    0    0    0    0    0    0     0
 [7,]    0    0    1    0    0    0    0    0    0     0
 [8,]    0    0    1    0    0    0    0    0    0     0
 [9,]    0    0    0    1    0    0    0    0    0     0
[10,]    0    0    0    1    0    0    0    0    0     0
```

questa matrice rappresenta effettivamente il grafo usato come esempio.
L'obiettivo del prossimo passo dell'esempio è quello di trasformare in ina matrice probabilistica. Una **matrice di transizione** (che chiameremo Q). Ma prima di descrivere come fare (una sola istruzione R), cerchiamo di capire perchè è utile trasformare la matrice del grafo (della rete di nodi) in questo modo. Immaginatevi di essere su un nodo della rete. E di volervi spostare su un altro nodo. Ovviamente potete farlo se e solo se il nodo in cui volete andare è collegato con il nodo in cui siete ora. Guardiamo la matrice che rappresenta la rete (quella che descrive i collegamenti tra nodi in Figura 2). Supponiamo di essere sul nodo 1 ... e quindi guardiamo la **riga** 1. Su quella riga troviamo solo 3 valori diversi da 0 poichè il nodo 1 è collegato solo con altri 3 nodi. Per sapere quali nodi dobbiamo vedere **in quali colonne** si trovano i valori diversi da 0. Stando al contenuto della figura i nodi collegati al nodo 1 sono il 2, il 3 ed il 4.
Per far sì che la matrice I diventi probabilistica dobbiamo manipolarla in modo che i numeri sulla riga di ogni nodo esprimano non la semplice presenza o assenza di un collegamento con gli altri nodi ma la **probabilità** di spostarci sui nodi connessi al nodo della riga corrente.
Dato che i nodi con valore maggiore di 0 sulla riga del nodo 1 sono solo 3 e che tutti quanti gli archi hanno valore 1 la somma di tutti i valori sulla riga di nodo 1 è 3. Per convertire i pesi in probabilità è sufficiente dividere tutti i valori della prima riga per la somma della stessa. In questo modo sulla riga del nodo 1 avremo tre valori uguali (ognuno pari a $1/3$) ad indicare che, quando raggiungiamo il nodo 1 avremo $P = 1/3$ di spostarci su nodo 2, $P = 1/3$ di spostarci su nodo 3 e $P = 1/3$ di spostarci su nodo 4. La matrice delle transizioni è come una mappa: una volta che so dove sono so dove posso spostarmi e con quale probabilità.

Figure 3: Creazione matrice transizioni.

```
> library(NetPreProc)
> Prob.norm(I)
           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]      [,8]      [,9]     [,10]
 [1,] 0.0000000 0.3333333 0.3333333 0.3333333 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
 [2,] 0.3333333 0.0000000 0.0000000 0.0000000 0.3333333 0.3333333 0.0000000 0.0000000 0.0000000 0.0000000
 [3,] 0.3333333 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.3333333 0.3333333 0.0000000 0.0000000
 [4,] 0.3333333 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.3333333 0.3333333
 [5,] 0.0000000 1.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
 [6,] 0.0000000 1.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
 [7,] 0.0000000 0.0000000 1.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
 [8,] 0.0000000 0.0000000 1.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
 [9,] 0.0000000 0.0000000 0.0000000 1.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[10,] 0.0000000 0.0000000 0.0000000 1.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
> Q <- Prob.norm(I)
> 
```

Per costruire la matrice delle transizioni Q (quella in Figura 3) in R dovremo utilizzare la funzione **Prob.norm()** presente nel package **NetPreProc**. Essa acceetta un unico argomenti in input (la matrice I) e restituisce un unico oggetto (la matrice di transizione Q).

I cammini aleatori (Random walk) sui grafi assomigliano ad un processo di diffusione. Tutto quello che serve per utilizzarli in R sono:

- una matrice di transizione della rete (come la matrice Q)
- un **vettore** contenente un numero di elementi pari al numero di nodi nella rete. In questo vettore imposteremo i valori in modo da rappresentare le **masse di probabilità** prima di iniziare a camminare sulla rete. Detto così sembra spaventoso (cosa saranno queste masse di probabilità?). In realtà è più semplice di quello che sembra. Il nostro grafo di esempio ha 10 nodi. Supponiamo che a tempo 0 abbiamo deciso che possiamo solo trovarci in nodo 1. Allora il primo elemento del vettore di 10 elementi va impostato a 1 e tutti gli altri vanno lasciati a 0. Come nel caso delle righe della matrice di transizione anche per questo vettore vale la regola che la somma dei suoi elementi deve essere pari a 1 (probabilità totale). Questo implica che, se in questo vettore i cammini possono iniziare con uguale probabilità dal nodo 1 oppure dal nodo 2, allora dovremo impostare sia il primo (nodo 1) elemento che il secondo (nodo 2) elemento al valore di 1/2 e lasciare tutti gli altri elementi a 0.

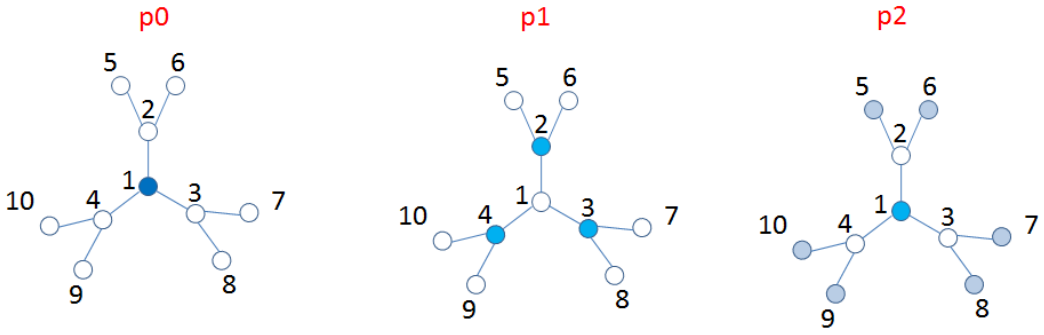
Una volta in possesso di questi due elementi (la matrice delle transizioni ed il vettore delle probabilità iniziali) per calcolare la probabilità di trovarci in uno qualsiasi dei nodi della rete dopo un passo è sufficiente moltiplicare il vettore per la matrice. Così facendo otterremo il vettore delle probabilità al passo 1.

Figure 4: Calcolo probabilità dopo uno e due passi.

```
> probvector <- rep(0,10)
> probvector[1]<-1
> probvector
 [1] 1 0 0 0 0 0 0 0 0 0
>
> p0 <- probvector
> p0 %*% Q
     [,1]      [,2]      [,3]      [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    0 0.3333333 0.3333333 0.3333333    0    0    0    0    0     0
> p1 <- p0 %*% Q
> p1 %*% Q
          [,1] [,2] [,3] [,4]      [,5]      [,6]      [,7]      [,8]      [,9]      [,10]
[1,] 0.3333333    0    0    0 0.1111111 0.1111111 0.1111111 0.1111111 0.1111111 0.1111111
> p2 <- p1 %*% Q
> sum(p0)
[1] 1
> sum(p1)
[1] 1
> sum(p2)
[1] 1
>
```

Per ottenere le probabilità dopo due passi è sufficiente moltiplicare il vettore delle probabilità dopo un passo per la matrice di transizione. E così via per i massi successivi al secondo. La matrice delle transizioni non cambia mai. Sono solo le masse di probabilità a **camminare** sul grafo (e quindi a **ridistribuirsi sui nodi**). Dato che la matrice delle transizioni è probabilistica ad ogni passo percorriamo tutte le possibili vie che partono da ogni nodo. E costuiamo (passo per passo) tutti i possibili percorsi. Usando la matrice probabilistica delle transizioni è come se campionassimo tutti i possibili cammini casuali su una rete. E' per questo che questa tecnica viene detta **cammini aleatori**.
Una domanda importante da porsi è cosa ha influenza sul modo in cui le probabilità si ristribuiscono e la risposta è che i due fattori più importanti sono il vettore delle probabilità iniziale e (più importante ancora) la **struttura della rete** ossia le connessioni esistenti tra i nodi.
In R il calcolo delel probabilità al passo successivo (che restituisce un vettore avente un elemento per ogni nodo) si fa come è mostrato in Figura 4.
Cerchiamo di capire se i risultati ottenuti hanno senso. Teniamo presente che abbiamo impostato il vettore delel probabilità iniziali mettendo tutta la massa di probabilità sul nodo 1 (il nodo centrale del grafo di esempio).
Se confrontiamo il contenuto di Figura 5 con il contenuto di Figura 4 noteremo

Figure 5: Probabilità nodi iniziali, dopo uno e due passi.



che nel vettore delle probabilità iniziali (p0) tutta la probabilità non si è anocra diffusa (è tutta su nodo 1). Dopo un passo (vettore probabilità p1) la massa di probabilità si è distribuita equamente sui nodi connessi a nodo 1 (ossia sui nodi 2, 3 e 4). Dopo un ulteriore passo (vettore probabilità p2) tutti i nodi connessi ai nodi 2, 3 e 4 ricevono un pò di probabilità (quanta? 1/3 di 1/3.) ma tra essi uno dei nodi (il nodo 1) riceve questo contributo da tre nodi (il 2, il 3 ed il 4) e quindi, in totale, dopo due passi la probabilità totale di essere di nuovo sul nodo 1 è 1/3.
La tecnica che abbiamo appena visto è molto comoda per fissare dei nodi di partenza in una rete ed esplorarla. Non c'è limite al numero di passi che possiamo far fare ad R sulla rete ... possiamo anche decidere di effettuare 10000 passi. E tutto ciò che dovremo fare per ogni passo è solo la moltiplicazione vettore per matrice avendo cura di usare come vettore quello delle probabilità correnti (quello ottenuto dal passo effettuato più di recente).

7.2 Riposizionamento di farmaci

Di sicuro vi starete chiedendo cosa c'entrano i cammini aleatori (Random walks) con il riposizionamento di farmaci.
Una volta che abbiamo costruito una matrice di similarità tra farmaci essa può essere vista come una rete tra farmaci e, inoltre, da essa è possibile ottenere una matrice delle transizioni. Se, oltre alla rete, abbiamo anche informazioni sulle caratteristiche terapeutiche dei farmaci esse possono essere utilizzate per fissare le probabilità iniziali ed iniziare ad esplorare la rete di farmaci.
Tutto il lavoro sperimentale, in questo approccio, risuede nell'effettuare test che possono essere utilizzati per costruire una misura di similarità tra tutte le

possibili coppie di farmaci. Informazioni possibili da cui partire sono le seguenti (questo è solo un esempio ... ne esistono molte altre):

- Similarità tra strutture molecolari (in questo laboratorio siamo partiti da questo tipo di informazione)
- Effetto della somministrazione dei farmaci (ad es. misura tasso di crescita) ad un pool di mutanti di un determinato parassita. Per ogni farmaco ottengo un vettore di valori (uno per ogni mutante) e il calcolo della similarità tra coppie di farmaci si riduce ad un calcolo di quanto sono simili i loro vettori.
- Affinità dei farmaci per un pool di molecole (proteine che rappresentano potenziali target). Anche qui otteniamo per ogni farmaco un vettore contenente l'affinità per ogni proteina. E anche in questo caso la similarità tra farmaci si calcola confrontando i vettori.

Questi sono solo tre possibili esempi. In realtà esistono casi più complessi in cui abbiamo non una ma più reti di similarità tra farmaci (ognuna derivante da un diverso tipo di evidenza sperimentale). In questo caso basta ricordare che le reti (i grafi) sono rappresentati da matrici. Fino a quando le diverse reti si riferiscono allo stesso set di farmaci sarà sempre possibile integrare queste reti (ad esempio calcolando la matrice media ottenuta da tutte le matrici disponibili) e poi procedere con la costruzione della matrice delel transizioni e utilizzare la tecnica dei Random walk.
Il razionale per questo tipo di esperimento (riposizionamento di farmaci) è il seguente: se ho una rete che esprime la similarità tra i farmaci di un set dato e so che alcuni di essi hanno una proprietà terapeutica utile allora *esplorando la rete* potrò trovare i farmaci più simili ad essi e i farmaci trovati potrebbero avere proprietà terapeutiche simili e, quindi, essere utilizzati nel trattamento della medesima patologia per cui i farmaci (i nodi) da cui abbiamo iniziato l'esplorazione della rete sono indicati.

7.3 Progetto d'esame

Nel progetto d'esame dovrete effettuare, utilizzando R, dei test di riposizionamento di farmaci basati su analisi di reti di similarità farmacologica. Maggiori informazioni (ed esempi step by step) saranno forniti nelle slide della lezione del corso dedicata alla presentazione dei progetti d'esame.
L'esame richiede la realizzazione di un progetto e la risposta ad una serie di domande. La lista completa di domande, ognuna corredata da spiegazione, sarà fornita durante l'ultima lezione del corso.