RELAZIONE PROGETTO TOTALE

Quasi tutto il progetto si svolge all'interno della classe RC "Graph" contenente una lista di tre campi e sedici metodi.

I primi rappresentano:

- La matrice di adiacenza con cui andrò a rappresentare il grafo;
- Una lista di environment chiamata "Vertici", dove ogni environment rappresenta un vertice del grafo e contiene al suo interno le variabili nome e valore, le quali rappresentano i caratteri distintivi del vertice stesso;
- Una lista di environment chiamate "Archi", dove ogni environment rappresenta un arco del grafo e contiene al suo interno le variabili sorgente, destinazione e valore, utili per identificare univocamente l'arco.

L'utilità di conservare ogni dato riguardante vertici ed archi separatamente suddividendoli all'interno degli environment delle due liste deriva dall'esigenza di tener traccia del valore dei vertici e delle caratteristiche degli archi che sarebbero altrimenti difficilmente estrapolabili dalla matrice di adiacenza ogni volta che se ne ha bisogno.

La matrice di adiacenza è infatti una matrice quadrata dove le righe e le colonne rappresentano i vertici del grafo (e a cui, per comodità, ho anche dato i loro nomi), mentre i valori all'interno corrispondono ai valori degli archi del grafo: se il valore è 0, l'arco non esiste, altrimenti esiste ed ha il valore corrispondente.

I metodi rappresentano invece le singole richieste dettate dalla consegna del progetto.

PUNTO 1a.1

Il primo metodo creato esegue la verifica dell'esistenza di un arco tra un vertice x ed un vertice y.

Il primo passaggio consiste nel controllare l'esistenza dei due vertici, altrimenti il programma restituirebbe un errore nel momento in cui si va a cercare l'arco.

Questo controllo viene effettuato utilizzando i nomi delle righe e delle colonne della matrice di adiacenza: se il nome del vertice sorgente dell'arco figura tra le righe e quello del vertice destinazione figura tra le colonne della matrice, significa che i due vertici sono già stati creati e quindi si può procedere a verificare l'esistenza dell'arco.

Per fare ciò è sufficiente vedere se il numero corrispondente all'incrocio tra riga e colonna della matrice è diverso da zero: se lo è l'arco esiste e viene restituito, altrimenti un messaggio avvisa l'utente che l'arco non esiste.

PUNTO 1a.2

Il secondo metodo elenca tutti i vertici adiacenti ad un vertice x, cioè tutti quelli raggiungibili da x passando per un arco.

Dopo aver controllato, come nel metodo precedente, se il vertice x esiste, si crea una lista vuota che sarà riempita con i nomi dei vertici adiacenti ad x.

Per questo ho utilizzato un ciclo for che scorra le colonne della matrice e che, in caso trovi un valore diverso da 0 all'incrocio tra la riga x ed una data colonna, simbolo che esiste un arco tra i due, aggiunga alla lista il nome della colonna, e quindi del vertice raggiunto tramite l'arco.

Quando tutte le colonne sono state analizzate, se la lista è ancora vuota, un messaggio avvisa l'utente che x non ha vertici adiacenti, altrimenti viene restituita la lista completa.

PUNTO 1a.3

Il terzo metodo aggiunge un vertice al grafo, se non ne esiste già uno con lo stesso nome.

Dopo essermi assicurato che un vertice con lo stesso nome di quello sorgente non esista già, impongo che il valore del vertice debba essere maggiore o al limite uguale a 0: questo limite è posto dal fatto che il valore del vertice viene solitamente utilizzato per indicare la distanza dal vertice sorgente all'interno degli algoritmi che calcolano il cammino minimo di un grafo, dunque conterranno sempre un valore

positivo in quanto il cammino, essendo una somma dei valori degli archi al suo interno, sarà sempre rappresentato da un valore maggiore di 0.

A questo punto c'è l'aggiunta vera e propria del vertice alla sua lista, effettuata mediante la creazione di un nuovo environment, all'interno del quale vengono salvati gli attributi "nome" e "valore", e poi dall'accodamento vero e proprio alla lista sopracitata.

Similmente, anche la matrice di adiacenza viene aggiornata mediante la creazione di due nuove matrici, una di una sola riga ed una di una sola colonna, che vengono poi "unite" alla matrice principale previa assegnazione del nome del vertice sia alla singola riga che alla singola colonna.

PUNTO 1a.4

Questo metodo rimuove il vertice x, se ne esiste uno con tale nome.

Appena superato il controllo dell'esistenza di un vertice con il nome dato, ci si assicura che non ci siano archi collegati a quel vertice, ed in caso questi vengono eliminati utilizzando il metodo di rimozione degli archi al punto 1a.6.

Una volta rimossi gli archi residui, viene rimosso dalla lista dei vertici l'environment la cui posizione corrisponde al numero della riga corrispondente (l'ordine delle righe della matrice è la stessa di aggiunta dei vertici e quindi degli environment nella lista), ed allo stesso modo si rimuovono la riga e la colonna dalla matrice di adiacenza.

PUNTO 1a.5

Il quinto metodo aggiunge un arco al grafo, se non ne esiste già uno con la stessa combinazione di vertice sorgente e vertice di destinazione.

Subito dopo il solito check che i vertici esistano, si controlla sulla matrice se un arco è già presente tra i due vertici, e se il valore da dare all'arco è maggiore di 0: allo stesso modo dei vertici, gli archi sono solitamente utilizzati nei grafi per indicare una distanza, un peso od un costo, e per questo non possono essere negativi e

nemmeno nulli (anche perché se fossero nulli la matrice non li conterebbe come archi in quanto il loro valore è 0).

Una volta superate queste condizioni, viene creato l'environment dell'arco usando lo stesso procedimento dell'aggiunta del vertice, e poi viene modificata la matrice di adiacenza inserendo nella riga e nella colonna corretta il valore dell'arco.

PUNTO 1a.6

Questo metodo può essere utilizzato per eliminare l'arco tra il vertice x e il vertice y, se presente.

Passato il classico controllo dell'esistenza di entrambi i vertici, si va a vedere se l'arco effettivamente esiste, valutando se il valore all'interno della matrice di adiacenza è diverso da 0.

A questo punto si scorrono tutti gli archi cercando quello dove sia il nome del vertice sorgente che il nome del vertice di destinazione corrispondono a quelli dell'arco da eliminare e si utilizza un contatore incrementato ad ogni ciclo per memorizzare la posizione sulla lista dell'arco da rimuovere.

Finalmente, dalla lista viene rimosso l'environment nella posizione indicata dal contatore e sulla matrice il valore viene impostato a 0.

PUNTO 1a.7

Il settimo metodo restituisce il valore del vertice con nome x, se esistente.

Inizio controllando come sempre se il nome corrisponde ad un vertice esistente e, dopo averne avuta la conferma, scorro la lista dei vertici fino a trovare quello che mi interessa (e quindi il suo valore) e lo restituisco.

PUNTO 1a.8

Con questo metodo si può assegnare un determinato valore ad uno specifico vertice, sovrascrivendo quello precedente.

Si inizia controllando che il vertice esista e che il valore da inserire sia maggiore od al più uguale a 0, come nel metodo di creazione del vertice.

Successivamente, è sufficiente scorrere la lista dei vertici fino all'environment corretto ed aggiornare la variabile valore, in modo simile al metodo precedente, interrompendo poi il ciclo nell'istante in cui la modifica è stata effettuata.

PUNTO 1a.9

Per restituire il valore associato ad un arco il procedimento è molto simile alla verifica della presenza dell'arco stesso sopra descritta.

Finito il controllo dell'esistenza di entrambi i vertici ed anche dell'arco, viene utilizzata la matrice di adiacenza per restituire il valore corretto dell'arco, tramite l'incrocio tra il vertice riga ed il vertice colonna.

PUNTO 1a.10

Il decimo metodo permette di assegnare, in maniera quasi uguale all'ottavo, un valore ad un arco, cancellando quello segnato prima.

Prima di procedere con la modifica, devo controllare come sempre che i vertici esistano, che l'arco sia presente, e che il nuovo valore rispetti gli standard fissati in precedenza.

Posso a questo punto modificare il valore prima andandolo a cercare nella lista degli archi ed aggiornandolo, e poi facendo la stessa cosa con la matrice di adiacenza.

PUNTO 1b

Il metodo numero undici ha come scopo quello di restituire una rappresentazione in lista di adiacenza da creare sulla base della matrice di adiacenza preesistente.

Si inizia creando una lista di adiacenza vuota ed una variabile booleana che tiene traccia se la lista si riempie o meno (variabile che è inizialmente impostata su TRUE).

A questo punto si scorrono le righe della matrice mediante i loro nomi, e di ognuna si genera un vettore contenente i valori diversi da 0 su tutte le colonne; si calcola a quel punto la lunghezza del vettore e se essa è pari a 0, ovvero se dal vertice corrispondente alla riga non hanno origine archi, alla lista viene aggiunto un elemento nominato “”, ad indicare che il vertice esiste ma non è sorgente di nessun arco.

Se, al contrario, la lunghezza non è nulla, alla lista viene aggiunto un vettore contenente il nome delle colonne (e quindi dei vertici) che sono destinazioni di un arco, e la variabile booleana viene impostata su FALSE dato che a questo punto la lista sicuramente non è più vuota.

In entrambi i casi, all’elemento aggiunto alla lista (che sia “”, oppure il vettore contenente le destinazioni degli archi) viene dato il nome della riga corrente, quindi del vertice sorgente.

Infine, se dopo aver finito di scorrere le righe della matrice la lista risulta ancora vuota, viene stampato un messaggio che informa l’utente che non esistono archi sulla matrice stessa, altrimenti viene restituita la lista di adiacenza in una forma di questo tipo:

```
$`a`
[1] "b"

$b
[1] "c"

$c
[1] "d" "f"

$d
[1] "e" "f"

$e
[1] "f"

$f
[1] ""
```

Come si può notare, vicino al dollaro è segnato il nome del vertice sorgente e dopo l’[1] sono scritti i nomi dei vertici di destinazione, tranne per il vertice “f” che, non avendone, riporta la sopracitata dicitura “”.

PUNTO 1c

La creazione della edge list risulta invece molto più semplice della precedente poiché, dopo aver creato la lista vuota, si passa direttamente alla fase in cui essa viene riempita scorrendo la già organizzata lista degli archi e, per ogni environment, si crea un vettore composto dal vertice sorgente e dal vertice di destinazione che viene poi aggiunto alla edge list.

Se alla fine la lunghezza della lista è nulla, un messaggio avvisa che il grafo non ha archi, altrimenti viene restituito un risultato simile a questo:

```
[[1]]
[1] "a" "b"

[[2]]
[1] "b" "c"

[[3]]
[1] "c" "d"

[[4]]
[1] "d" "e"

[[5]]
[1] "e" "f"

[[6]]
[1] "d" "f"

[[7]]
[1] "c" "f"
```

PUNTO 2

Il tredicesimo metodo stampa il grafo creato mediante l'uso dei metodi precedenti, utilizzando per lo scopo una libreria denominata "networkD3".

I parametri in input del metodo sono la matrice di adiacenza, la lista dei vertici e quella degli archi: è necessario che essi siano specificati dall'utente nel momento in cui il metodo viene richiamato poiché esso verrà anche usato per la stampa dei punti successivi che, essendo salvati su variabili con nome diverso, non sarebbero altrimenti stampabili mediante l'utilizzo dello stesso metodo.

In questo caso i controlli preliminari da effettuare verificano che la matrice non sia vuota, e se lo è avvisano l'utente ed interrompono l'esecuzione, ma anche che la libreria sia già installata, altrimenti viene aggiunta e caricata.

Il metodo passa poi alla creazione di due matrici, una per i vertici ed una per gli archi, aventi entrambe una struttura particolare:

- La prima è composta di tante righe quanti sono i vertici e dalle colonne nome, valore e gruppo, dove la prima è il nome assegnato al vertice, la seconda è il suo valore e la terza può essere utile per creare gruppi separati di vertici colorati in maniera differente, ma nel nostro caso sarà ignorata ponendo il gruppo di ogni vertice al simbolico valore di 1);
- La seconda è composta di tante righe quanti sono gli archi e dalle colonne sorgente, destinazione e valore, dove le prime due sono il numero di riga e colonna che corrispondono al vertice sorgente e di destinazione, mentre la terza è la radice quadrata del valore dell'arco (ho usato la radice quadrata per ridurre il valore dell'arco, dato che esso influenza pesantemente la dimensione delle frecce del grafo e renderebbe la stampa di difficile comprensione). A causa del fatto che la libreria "networkD3" implementa Javascript, le prime due colonne sono indicizzate a partire da 0, ovvero i loro valori si riferiscono ad una matrice la cui prima riga e prima colonna hanno come indice 0 al posto di 1, che è più comunemente usato all'interno del linguaggio R.

Per riempire la prima delle due è sufficiente scorrere la lista dei vertici salvando man mano i dati al suo interno, mentre per la seconda è necessario valutare gli archi, confrontare i loro nomi con quelli delle righe e delle colonne della matrice di adiacenza e, trovati gli indici corrispondenti, sottrarre 1 al loro valori ed inserirli nella nuova matrice, aggiungendo poi la radice quadrata dei valori stessi.

Nel caso in cui non ci siano archi la seconda matrice viene invece creata con una sola riga e senza dati dentro.

Le due matrici vanno poi convertite in data frame per poter essere utilizzate dalla libreria: la conversione è eseguita direttamente dal linguaggio, ma c'è bisogno di creare due liste contenenti i nomi delle tre colonne ed i nomi delle righe (numerate da 1 fino al numero di elementi), e poi di utilizzarle per assegnare i nomi ai data frame.

Infine, per la stampa vera e propria, si usa il comando “forceNetwork”, controllando prima se la matrice di adiacenza è simmetrica, e quindi se il grafo è orientato o meno, ed eseguendo il plotting diversamente a seconda dei casi.

Per quest’ultimo passaggio i messaggi di warning sono disabilitati poiché, nel caso il primo vertice sia isolato, e quindi se non ci sono archi che si originano o terminano sul vertice di indice 0, la libreria avvisa che si è commesso un probabile errore di indexing, per il motivo spiegato precedentemente, mentre invece il risultato è corretto.

PUNTO 3

Questo metodo si occupa della visita in ampiezza di un grafo, ovvero passa, dato un vertice di partenza, su tutti gli altri vertici del grafo, andando prima su quelli vicini all’origine, poi sui vicini dei vicini e così via, tracciando in questo modo un grafo in cui la distanza tra uno qualunque dei vertici e la sorgente rappresenta anche il percorso minimo nel caso in cui i pesi degli archi siano tutti equivalenti.

Come da consegna, l’algoritmo lavora solo con grafi semplici ed orientati, quindi prima di procedere col metodo vero e proprio si verifica che il grafo sia orientato (cioè che la matrice di adiacenza sia simmetrica), che abbia almeno un arco e che il vertice sorgente esista.

A questo punto ho bisogno di creare una nuova matrice di adiacenza per la successiva stampa del grafo, quindi copio quella già esistente, insieme alle due liste di environment, su variabili temporanee, ed inizializzo quelle originali.

Tenendo poi memorizzata la distanza dall’origine in una variabile apposita (ovviamente all’inizio essa è pari a 0), il primo vertice viene aggiunto alla matrice ed alla lista dei vertici da visitare, in modo da analizzare successivamente ogni vertice collegato ad esso.

Dopo aver creato un ciclo while che vada avanti fino a quando la lista dei vertici da visitare non è vuota, si scorre la suddetta lista e, per ogni elemento che contiene, si cercano tutti gli archi che partono dal vertice ad arrivano a vertici non ancora visitati: questi ultimi vengono creati, insieme ai sopracitati archi, utilizzando il terzo

ed il quinto metodo, ed i vertici sono anche aggiunti alla lista di quelli da visitare; infine, l'elemento visitato viene tolto da quelli ancora da visitare.

In questo modo il grafo è ora costituito da tutti i vertici raggiungibili in ordine di distanza dall'origine, ma mancano ancora quelli irraggiungibili: attraverso un confronto tra i vertici sulla variabile temporanea e quelli aggiunti alla nuova lista si possono creare i mancanti, il cui valore viene fissato a 99 a simboleggiare che la distanza dal vertice di origine è infinita, dato che non sono collegati.

Se inoltre la lista degli archi rimane vuota significa che il vertice di origine è isolato oppure che è l'unico del grafo: in entrambi i casi un messaggio avvisa l'utente che non ci sono vertici raggiungibili, ma il metodo va avanti lo stesso dato che questo fatto non è considerabile come un errore ed il grafo può essere stampato ugualmente.

Infine, la matrice di adiacenza creata e le due liste vengono spostate su variabili apposite per la stampa, mentre i tre backup effettuati all'inizio del grafo originale vengono ripristinati nei loro nomi originali, dopodiché viene invocato il metodo di visualizzazione descritto sopra (i dati salvati nelle variabili temporanee devono essere ricopiati in quelle originali prima del plotting del grafo, dato che esso è considerato come un return e quindi il codice situato dopo non è eseguito).

PUNTO 4

Il quindicesimo metodo esegue l'algoritmo di Dijkstra, e trova dunque tutti i cammini minimi di un grafo pesato che partano dal vertice dato ed arrivino a qualunque altro vertice, restituendo la stampa di un grafo che li comprenda tutti.

Ancora una volta, è richiesto che l'algoritmo lavori solo con grafi orientati, quindi controllo nuovamente che quello creato lo sia, che il sorgente faccia parte della matrice di adiacenza e che esista quantomeno un arco.

Come nel metodo precedente, la matrice, gli archi e i vertici vengono spostati su variabili temporanee e poi inizializzati per essere ricreati dall'inizio, dopodiché si crea il vertice sorgente e lo si aggiunge ad una lista che conterrà tutti i vertici già aggiunti.

All’interno del ciclo while, che andrà avanti fino a che la suddetta lista non conterrà tutti i vertici, ad ogni iterazione viene inizializzato un vettore che contenga alcune info sull’arco minimo trovato, ovvero sorgente, destinazione, valore e somma del valore dell’arco e del valore del vertice sorgente.

Con un ciclo for, si cercano poi gli archi che abbiano origine su un vertice già aggiunto e destinazione su uno ancora da aggiungere, di ognuno si calcola la somma tra il valore del vertice sorgente e il valore dell’arco, si crea la destinazione se ancora non esiste oppure la si aggiorna se il valore somma risulta minore del valore della destinazione stessa (in questo modo si esegue il cosiddetto “rilassamento”, ovvero tutti i vertici vicini a quelli già aggiunti ricevono un aggiornamento del loro valore se la variabile somma è minore del loro valore attuale); infine la variabile somma è confrontata con l’ultimo valore del vettore “arcominimo” e, se risulta minore di esso, dunque se è stato trovato un arco che è minore del precedente, il nuovo arco viene memorizzato al posto di quello vecchio.

Una volta terminato il ciclo si analizza l’arco segnato come minimo: se la quarta posizione del vettore è rimasta 0, significa che non ci sono più archi da aggiungere, quindi si creano i restanti vertici, se ce ne sono (questi devono obbligatoriamente essere isolati o quantomeno irraggiungibili dal vertice sorgente), e poi si interrompe il ciclo; se invece la quarta posizione è diversa da 0, l’arco minimo viene creato e il suo vertice di destinazione è aggiunto alla lista dei vertici visitati.

Come prima, se il vertice sorgente è isolato, il metodo continua l’esecuzione ma restituisce un messaggio di avviso, le variabili vengono spostate su altre variabili per la stampa e quelle originali vengono ripristinate, ed infine viene lanciato il plotting del grafo.

PUNTO 5

L’ultimo metodo stampa il grafo derivante dall’algoritmo di Kruskal, ovvero il cammino creato scegliendo ogni volta l’arco col valore minimo che non crei cicli nel grafo stesso.

Essendo Kruskal utilizzabile solo sui grafi non orientati, l’unico controllo preliminare riguarda proprio la simmetria della matrice di adiacenza: se essa risulta asimmetrica, un messaggio avvisa l’utente e l’esecuzione si blocca.

Dopo aver copiato nuovamente la matrice di adiacenza e la lista degli archi (quella dei vertici stavolta non sarà copiata in quanto non verrà modificata), creato una seconda copia degli archi col nome “archisparsi” ed aver inizializzato le due variabili, lo step successivo consiste nell’ordinare gli archi all’interno della lista di environment, i quali sono ora messi in ordine di aggiunta, mentre alla fine saranno per valore crescente.

Questo avviene all’interno del ciclo while il quale trova ogni volta l’arco col valore minimo, elimina dagli archi non ordinati l’opposto di quello selezionato (ovvero, preso l’arco (x, y), elimina l’arco (y, x), utile solo per distinguere i grafi non orientati da quelli orientati, ma inutile in questo metodo dato che trattiamo solo i primi), controlla che non sia un cappio (ovvero un arco dove il sorgente corrisponde alla destinazione, che non ha senso nei grafi non orientati), lo aggiunge alla lista ordinata e lo rimuove dall’altra, fino a che quest’ultima rimane vuota ed il ciclo si interrompe.

A questo punto gli archi vengono nuovamente salvati all’interno della variabile “archisparsi”, usata successivamente, la matrice viene inizializzata ancora una volta e viene creata una lista che consenta di gestire, e quindi evitare, i cicli di archi.

All’interno del ciclo for principale, ogni arco viene valutato e, se non forma cicli, aggiunto alla matrice da stampare: la variabile ciclo è inizialmente FALSE e viene resa TRUE solo nel caso ci trovassimo di fronte ad un ciclo e quindi l’arco non debba essere aggiunto.

La variabile controllocicli è una lista di vettori contenente i diversi cammini trovati fino a quel momento: in ogni posizione della lista conserveremo una serie di nomi di vertici che sono tutti collegati tra loro da archi, e basterà quindi controllare, nel momento in cui si va ad aggiungere un arco, se sia il suo vertice sorgente che il suo vertice di destinazione fanno già parte della lista e sono nella stessa posizione (in questo caso avremmo un ciclo).

Questa condizione è la prima ad essere valutata nel ciclo for e, se risulta vera, si passa direttamente all’arco successivo; se invece non lo è si esegue uno dei due costrutti if: entrambi valutano se il vertice sorgente ed il vertice destinazione si trovano sulla lista ma in due posizioni diverse, infatti il primo si verifica se viene trovato prima il sorgente mentre il secondo se viene trovato prima l’altro.

In entrambi i casi, trovato un capo dell'arco si ricerca l'altro nelle posizioni successive: se questo viene trovato significa che due cammini prima separati sono uniti dall'arco, e quindi la variabile "controllocicli" viene modificata di conseguenza, altrimenti se non viene trovato va aggiunto alla lista nella stessa posizione dell'altro capo.

Se, ultimo caso, arrivassimo alla fine delle posizioni disponibili nella lista senza trovare nessuno dei due vertici da nessuna parte, viene creata una nuova posizione nella lista che li contiene entrambi.

Arrivato alla fine del ciclo for, in tutti i casi valutati in cui l'arco non forma cicli, esso viene creato insieme al suo opposto, altrimenti non vengono fatte modifiche ed il ciclo ricomincia.

In modo simile ai due metodi precedenti, la matrice di adiacenza nuova e la lista degli archi vengono trasferite su due variabili apposite per la stampa, mentre le configurazioni originali vengono ripristinate dalle variabili temporanee, ed infine viene lanciato il plotting.

CREAZIONE OGGETTO, PROVA GRAFO E TEMPO ESECUZIONE KRUSKAL

Terminata la classe "Graph", che racchiude tutto il codice "indispensabile", ho lasciato alcune righe di codice che possono essere utili nel caso si voglia creare un grafo di prova o per il test sul tempo di esecuzione dell'algoritmo di Kruskal:

```
g <- Graph$new()
```

Crea l'oggetto g della classe Graph, necessario per accedere ai metodi ed alle variabili create finora.

Le righe subito successive aggiungono una lista prefissata di vertici al grafo, quelle dopo il primo spazio vuoto aggiungono una serie di vertici per creare un grafo orientato, e quelle dopo il secondo spazio vuoto aggiungono gli opposti dei vertici aggiunti finora in modo da creare un grafo non orientato.

Infine, per calcolare il tempo di esecuzione dell'algoritmo di Kruskal ho utilizzato la funzione "proc.time", che permette di calcolare i tempi totali di esecuzione di R, e quindi salvando l'output della funzione prima dell'algoritmo e sottraendolo

dall’output della stessa funzione dopo l’algoritmo si ottiene il tempo per cui il metodo è rimasto in esecuzione, diviso in tempo reale del programma, tempo di calcolo a livello del processore e tempo di attesa effettivo prima dell’output.