

UNIVERSITÀ DEGLI STUDI DI TRENTO  
Facoltà di Scienze Matematiche, Fisiche e Naturali



Corso di Laurea triennale in Informatica

---

Elaborato Finale

ALGORITMI DI VISUALIZZAZIONE DI GRAFI TEMPORALI:  
UN'APPLICAZIONE IN FLEX

Relatore  
Prof. Alberto Montresor

Laureando  
Stefano Parmesan

Correlatore  
Dott. Paolo Massa

Anno Accademico 2008 - 2009

# INDICE

Introduzione.....	4
Definizioni.....	5
Stato dell'arte.....	8
Paradigmi di visualizzazione .....	8
Convenzioni di visualizzazione.....	8
Parametri di valutazione.....	10
Estetica.....	10
Vincoli.....	12
Efficienza.....	12
Social Networks.....	13
Adobe Flex.....	15
Vantaggi.....	15
Svantaggi.....	16
Adobe Air.....	17
Formati di memorizzazione.....	17
Linguaggio Dot.....	18
GraphML.....	18
Contributi: Graph Visualizer.....	20
Visualizzazione generica di un grafo.....	20
Algoritmo di affiancamento.....	21
NP-Difficoltà.....	23
Fattore di approssimazione.....	24
Costo computazionale.....	24
I grafi temporali.....	25
Eventi ed azioni.....	25
Memorizzazione.....	26
Notifica delle modifiche.....	27
Vertici ed archi.....	28
Algoritmi di visualizzazione.....	30
Algoritmi base.....	31
Random Draw.....	31
Circle Draw.....	32
Algoritmi basati su forze.....	33

Fruchterman – Reingold.....	36
Metodo del baricentro.....	38
Kamada Kawai.....	40
Algoritmi su grafi planari.....	42
Algoritmo di planarizzazione: incremental planarization.....	43
Grafì planari: rappresentazione a visibilità.....	44
Algoritmi interattivi.....	45
Un approccio agli algoritmi interattivi.....	46
Conclusioni.....	48
Bibliografia.....	51
Sitografia.....	52
Ringraziamenti.....	53

## INTRODUZIONE

Da tempo la teoria dei grafi è oggetto di studio da parte dei matematici, in quanto avente numerosi riscontri nella vita comune: lo sviluppo del mercato ha sollevato il problema dello spostamento delle merci, come pure l'avvento degli acquedotti romani per la copertura del territorio o la marcia delle truppe in guerra; tutti questi problemi hanno visto una loro rappresentazione nei grafi ed hanno suscitato l'interesse di matematici in ogni era. I grafi sono in grado di rappresentare dal punto di vista scientifico una situazione geografica, permettendo l'elaborazione di informazioni di carattere organizzativo, quali la scelta di un particolare percorso o la copertura di una determinata zona.

L'evoluzione della tecnologia ha sollevato altre necessità sui grafi: l'informatica e l'elettronica infatti sono state fin da subito molto legate a tali strutture, basti pensare alle reti di comunicazione fra computer, ma anche più semplicemente alla disposizione di componenti elettronici su di una scheda; sono stati quindi approfonditi algoritmi di planarizzazione e visualizzazione di grafi planari, ma anche problemi di tagli di reti, flussi e via dicendo.

L'interesse verso i grafi però è sempre stato forte anche per la loro capacità di rappresentare l'interazione fra persone, cosa che ha avuto ancora più centralità con la nascita del concetto di “massa popolare” e lo studio della psicoanalisi. Negli ultimi anni, in più, il forte sviluppo di internet e di piattaforme basate sull'interazione e la socializzazione di molte persone, grazie anche alla comparsa di nuove tecnologie che facilitano l'utilizzo della rete, hanno sollevato ulteriori problematiche: l'interazione fra le persone, specialmente ai nostri giorni, risulta essere una cosa estremamente dinamica. Con internet la comunicazione è globale, pressoché istantanea, e le reti di persone al mondo sono incredibilmente variabili.

Tutte queste problematiche hanno da sempre sollevato la necessità di visualizzare i grafi, portando all'idealizzazione di numerosi algoritmi per il disegno di reti; le tecnologie informatiche degli ultimi tempi e la crescente potenza di calcolo a disposizione hanno inoltre dato la possibilità di implementare nella pratica tali algoritmi, sollevando però una serie di problematiche legate alla performance, la valutazione automatica delle visualizzazioni, l'interattività e via dicendo.

L'incredibile successo che le reti sociali hanno avuto negli ultimi anni e la loro impressionante dinamicità hanno sollevato una nuova serie di problematiche riguardanti la visualizzazione di grafi variabili nel tempo. Su tali tipi di grafi, non molto studiati nel passato, non è purtroppo applicabile la maggior parte degli algoritmi di visualizzazione di reti, in quanto hanno insite differenti necessità. La ricerca nel campo della visualizzazione dinamica ed interattiva di grafi è quindi tuttora oggetto di studio.

Il campo sociale non è comunque l'unico settore in cui vengono utilizzati grafi variabili nel tempo; basti pensare ad esempio alla biologia ed in particolar modo all'epidemiologia, dove l'evoluzione e la diffusione di malattie o batteri viene studiata anche tramite grafi, che ne permettono una visione d'insieme ottimale per la ricerca di comportamenti predittivi.

In questo documento si vuole prima di tutto presentare delle nozioni base sui grafi. Successivamente si andrà a studiare i concetti che sono alla base della visualizzazione di grafi e sulla loro valutazione, per poi presentare l'evoluzione del loro utilizzo nelle reti sociali a partire dagli anni '30. Dopodiché verranno presentate delle tecnologie di programmazione web di recente diffusione e dei metodi ora in utilizzo per la memorizzazione di grafi.

Nella seconda parte si presenterà il lavoro svolto nello sviluppo di un'applicazione web per la rappresentazione di grafi temporali, GraphVisualizer, presentando innanzitutto un approccio base ai grafi temporali ed alla loro visualizzazione, per poi concentrarsi sulla realizzazione di algoritmi e la personalizzazione di archi e vertici all'interno dell'applicazione. Verranno poi presentati dei famosi algoritmi di visualizzazione che sono stati implementati, riportando del codice di spiegazione e degli esempi di applicazioni su grafi sociali e non.

## DEFINIZIONI

Per facilitare la lettura di questo documento, si riportano alcune definizioni di termini utilizzati nel testo riguardanti i grafi.

Un *grafo*  $G = (V, E)$  consiste in un insieme finito  $V$  di vertici, detti anche nodi, e di un insieme anch'esso finito  $E$  di lati o archi, ossia coppie non ordinate  $(u, v)$  di vertici. Un grafo è detto *semplice* quando non ammette lati multipli o circolari. La maggior parte degli algoritmi di visualizzazione si basa su grafi semplici, verrà quindi omessa tale informazione nella descrizione degli algoritmi per non appesantire eccessivamente la lettura.

Due vertici  $u$  e  $v$  si dicono *adiacenti* se esiste un lato  $(u, v)$  in  $E$ . I *vicini* di un vertice  $v$

sono tutti quei vertici in  $V$  che sono adiacenti a  $v$ .

Un *grafo diretto* è una specie particolare di grafo in cui l'ordine dei vertici in un lato ha importanza. Un lato  $(u, v)$  di tale grafo, detto anche *lato diretto*, si dice *uscende* da  $u$  ed *entrante* in  $v$ . Un lato diretto è spesso rappresentato da una freccia nella direzione del vertice entrante.

Un vertice con solo lati uscenti è detto *sorgente*. Un vertice con solo lati entranti è detto *pozzo*. Il *grado* di un vertice è il numero di lati uscenti da esso nel caso di un grafo diretto, il numero di suoi vicini nel caso di un grafo non diretto.

La *visualizzazione*  $\Gamma$  di un grafo  $G$  è una funzione che mappa ogni vertice  $v$  di  $G$  in un distinto punto  $\Gamma(v)$  ed ogni lato  $(u, v)$  di  $G$  in un segmento curvilineo  $\Gamma(u, v)$  avente come estremi i punti  $\Gamma(u)$  e  $\Gamma(v)$ . Una visualizzazione è detta *planare* se nessun lato interseca un altro lato.

In generale, un grafo ammette molte visualizzazioni. Esso è detto planare se ammette almeno una visualizzazione planare. I grafi planari hanno una notevole importanza, in quanto permettono una immediata visualizzazione e semplificano notevolmente alcuni concetti topologici. Inoltre è dimostrato che essi sono sparsi: un grafo planare con  $n$  vertici ha al più  $3n-6$  lati (si veda [BM76]).

Una visualizzazione planare di un grafo  $G$  suddivide lo spazio in aree, dette *facce*. La faccia con area illimitata che circonda il grafo è anche detta *faccia esterna*.

È possibile dare un ordine ai lati incidenti ad un qualunque vertice  $v$ , per esempio in base all'angolo di incidenza sul vertice ordinati in senso orario. Grazie a tale ordinamento è possibile definire una classe di equivalenza per le visualizzazioni planari di un grafo: due visualizzazioni planari sono equivalenti se ammettono lo stesso ordinamento per ogni vertice.

Le classi di equivalenza generate dalla relazione appena definita si chiamano *immersioni*. Un grafo si dice *immerso* quando possiede una determinata immersione.

Il *duale*  $G^*$  del grafo planare  $G$  è esso stesso un grafo planare, avente un vertice per ogni faccia di  $G$  ed un arco  $(f, g)$  per ogni coppia di facce  $f$  e  $g$  adiacenti in  $G$ . Se due visualizzazioni hanno la stessa immersione, avranno lo stesso grafo duale. Notiamo inoltre che il grafo duale potrebbe non essere semplice.

Un grafo si dice *connesso* se esiste un cammino da  $u$  a  $v$  per ogni coppia di vertici  $u, v$  di  $V$ . I grafi connessi, assieme ai grafi planari, sono molto importanti per gli algoritmi di visualizzazione. Il più grande sottografo connesso di un grafo  $G$  si dice *componente connessa* di  $G$ . Un vertice  $v$  di un grafo  $G$  connesso è detto *vertice di taglio* se la sua rimozione disconnette il grafo.

Un grafo connesso non avente vertici di taglio è detto *biconnesso*. Il massimo sottografo di  $G$  biconnesso è detto *blocco* o *componente biconnessa* di  $G$ . Alcuni algoritmi richiedono che il grafo su cui operano sia biconnesso. Questo non si tratta di un limite vincolante, in quanto è facile ottenere i blocchi di un grafo, eseguire l'algoritmo su di essi e ricostruire la visualizzazione: infatti un grafo è planare se tutti i suoi blocchi sono planari (si veda [BETT99]). Una coppia di vertici  $(u, v)$  di  $G$  è detta una *coppia di separazione* se la rimozione di entrambi i vertici  $u$  e  $v$  disconnette  $G$ .

Un grafo biconnesso senza coppie di separazione è detto *triconnesso*. Si dimostra che un grafo planare triconnesso ammette un'unica immersione.

Notiamo che è sempre possibile applicare un algoritmo per grafi non diretti a grafi diretti, semplicemente ignorando l'ordine dei vertici di ogni lato.

Una *numerazione topologica* di un grafo  $G$  è un assegnamento di valori ai vertici di  $G$ , tali per cui per ogni lato  $(u, v)$  il numero associato a  $v$  è maggiore del numero associato ad  $u$ . Un *ordinamento topologico* di  $G$  è una numerazione topologica tale per cui ad ogni vertice è assegnato un numero distinto tra 1 e  $|V|$ .

Una *numerazione topologica pesata* di un grafo con archi pesati  $G$  è una numerazione topologica di  $G$  tale che per ogni lato  $(u, v)$  di  $G$  il valore associato a  $v$  è maggiore o uguale al valore associato ad  $u$  più il peso dell'arco  $(u, v)$ . Tale numerazione è detta *ottimale* quando l'intervallo di valori associato ai vertici è minimo.

## STATO DELL'ARTE

### PARADIGMI DI VISUALIZZAZIONE

Gli algoritmi di visualizzazione di grafi sono molteplici, con differenti particolarità e limiti, bisogna quindi saper scegliere quale adottare in base alle proprie esigenze.

Come prima cosa, bisogna avere informazioni riguardanti il grafo che si vuole visualizzare; la cosa più importante da sapere è se stiamo trattando un grafo diretto, un albero, un grafo aciclico, ossia è essenziale conoscere la *classe* del grafo. Questo perché alcuni algoritmi funzionano solo – o sono più performanti – se eseguiti su determinate classi di grafi rispetto ad altre.

Inoltre è necessario conoscere l'ambiente nel quale questo grafo è inserito, ossia il suo significato: un albero rappresentante le dipendenze fra le classi di un determinato linguaggio di programmazione avrà, ad esempio, diverse necessità di visualizzazione rispetto ad un albero rappresentante la struttura di una pagina web. L'ambiente del grafo può indurre ad introdurre determinati vincoli al grafo, come ad esempio la necessità di raggruppare un sottoinsieme di vertici in un'area ristretta, o vincolare la posizione di speciali elementi.

Una volta capita la classe di appartenenza del grafo e studiato l'ambiente del quale fa parte, va scelto l'algoritmo adatto in base a diversi fattori, quali il tipo di convenzioni necessarie per la visualizzazione, le preferenze personali sull'estetica, il modo in cui il grafo verrà visualizzato ed ovviamente la performance.

### CONVENZIONI DI VISUALIZZAZIONE

Le convenzioni di visualizzazione sono delle determinate proprietà che la visualizzazione di un grafo deve avere per essere minimamente ammissibile. Spesso tali convenzioni dipendono dall'ambiente del grafo, e possono essere anche molto complesse. Ad esempio, il grafo di una rete sociale può richiedere che i nodi siano visualizzati in maniera tale da dare informazioni visive riguardanti la persona a cui fanno riferimento, oppure richiedere di visualizzare gli archi secondo uno schema prestabilito. Di seguito riportiamo alcune convenzioni che vengono più spesso utilizzate

in quanto semplici, generali ed assodate:

*Visualizzazione a polilinea:* ogni lato è rappresentato da una catena di segmenti. Un esempio potrebbe essere la visualizzazione di un grafo riguardante le classi di un determinato progetto e le loro interazioni.

*Visualizzazione a segmenti:* ogni lato è rappresentato da un unico segmento. I nodi sono quindi collegati direttamente, tramite una linea retta. La visualizzazione di un albero potrebbe richiedere tale convenzione, in quanto non è spesso necessario appesantire la visualizzazione con archi complessi.

*Visualizzazione ortogonale:* ogni lato è una sequenza di segmenti orizzontali o verticali, tutti ortogonali fra di loro. La rappresentazione di un circuito elettrico stampato su una scheda richiede spesso tale convenzione, in quanto è essenziale che la distanza fra due lati sia sempre maggiore di una soglia di sicurezza.

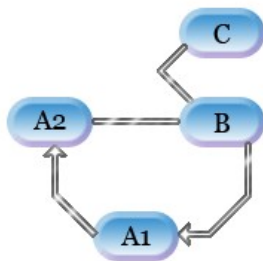


Fig. 1: Visualizzazione a polilinea

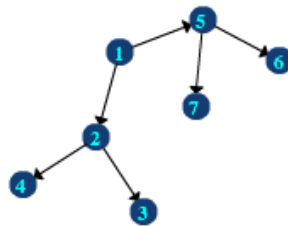


Fig. 2: Visualizzazione a segmenti

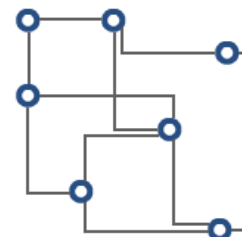


Fig. 3: Visualizzazione ortogonale

*Disposizione a griglia:* ogni vertice ed ogni piega dei lati del grafo hanno coordinate intere. Tale convenzione può essere richiesta ad esempio dalla visualizzazione topologica di una rete, e fa in modo che la distanza orizzontale e verticale fra due vertici o pieghe di lati sia sempre maggiore di un fattore numerico prefissato.

*Disposizione planare:* nessun lato del grafo si interseca. Una qualunque situazione potrebbe richiedere tale convenzione di visualizzazione, risulta infatti spesso molto leggibile. Non tutti i grafi però ammettono una tale visualizzazione.

*Disposizione crescente:* ogni lato è disegnato in modo che non sia mai decrescente. Viene spesso utilizzata per grafi diretti aciclici, in quanto permette una visualizzazione altamente intuitiva: vengono infatti a trovarsi le sorgenti del grafo nella zona bassa ed i pozzi nella zona alta della figura.

*Visualizzazione a box:* ogni vertice è rappresentato da un rettangolo, contenente varie informazioni riguardanti l'elemento rappresentato. Tale convenzione è spesso richiesta nella visualizzazione di schemi di Ingegneria del Software, dove sono molto importanti

le informazioni legate agli elementi. Una visualizzazione di questo genere ha impatto anche sui lati, in quanto essi non hanno più, dato un vertice, un unico punto di origine, bensì un intero perimetro.

*Colorazione*: si richiede che vengano utilizzati determinati colori per differenziare lati o vertici, in base alla tipologia o agli attributi associati. Tale richiesta viene spesso formulata quando sullo stesso grafo sono rappresentati lati con significati diversi, provenienti magari da differenti sondaggi sociali.

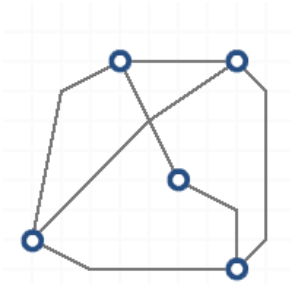


Fig. 4: Disposizione a griglia

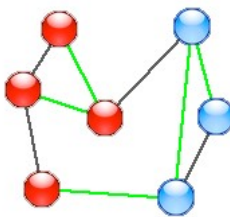


Fig. 5: Colorazione

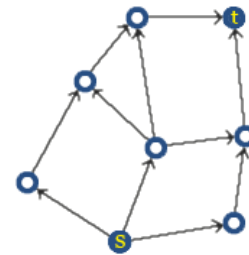


Fig. 6: Disposizione crescente

## PARAMETRI DI VALUTAZIONE

Una volta determinate le convenzioni alle quali la visualizzazione deve sottostare, è necessario scegliere un algoritmo adatto. Purtroppo però non esiste un algoritmo migliore di un altro, in quanto la scelta dipende, come abbiamo già sottolineato, dalla situazione in cui ci si trova ad operare.

Diversi algoritmi focalizzano l'attenzione su diversi fattori e diverse classi di grafi. Inoltre la performance dipende strettamente dalla tipologia del grafo su cui si sta lavorando. Bisogna quindi conoscere alcuni fattori degli algoritmi, per poter scegliere quello più appropriato alle proprie esigenze.

## ESTETICA

Gli algoritmi di visualizzazione di grafi spesso concentrano le operazioni nel tentativo di realizzare una rappresentazione facile ed intuitiva, esteticamente appagante. Per fare ciò, possono focalizzarsi su diversi obiettivi, ottenendo risultati anche totalmente differenti l'un l'altro. Spesso infatti ottimizzare un determinato fattore estetico va a discapito di altri; va quindi fatta una scelta in base alle esigenze ed al gusto personale. Tali obiettivi sono spesso alcuni dei seguenti:

*Intersezioni:* si cerca di minimizzare il numero di lati che si incrociano. Idealmente si cerca di trovare una visualizzazione planare del grafo, che però come abbiamo già visto spesso non esiste. Minimizzare il numero di intersezioni è un buon compromesso per i grafi non planari.

*Area:* si punta a minimizzare l'area del più piccolo rettangolo – o poligono convesso – che include tutto il grafo. Questo fattore è estremamente importante qualora non si possa interagire con il grafo per modificare la scala di visualizzazione, ad esempio quando si deve stampare la rappresentazione a video o su elementi fisici.

*Rapporto d'aspetto:* si minimizza il rapporto fra il lato più lungo ed il lato più corto del più piccolo rettangolo che contiene il grafo. Tale fattore è strettamente legato all'area, ma permette una rappresentazione più omogenea del grafo. Richiedendo un rapporto d'aspetto tendente all'unità, infatti, si ottiene una visualizzazione quadrata o rettangolare, in grado di utilizzare in modo corretto lo spazio a disposizione. Solitamente si cerca di ottenere un'area minima per tutti i rapporti d'aspetto inclusi in un determinato intervallo.

*Lunghezza dei lati:* si può cercare di minimizzare la lunghezza totale dei lati del grafo, come pure minimizzare il lato più lungo oppure ancora minimizzare la varianza della lunghezza dei lati; la scelta dipende dalle esigenze. Tale fattore è comodo, come per l'area, solamente qualora non sia possibile interagire con il fattore di scala della visualizzazione. Minimizzare il lato più lungo porta ad avere un limite superiore alla grandezza della rappresentazione, mentre minimizzare la varianza delle lunghezze porta ad avere una visuale più omogenea e quindi più pulita e piacevole.

*Numero di pieghe:* come per la lunghezza dei lati, si può cercare di minimizzare il numero di pieghe presenti nel grafo, oppure minimizzare il numero massimo di pieghe per ogni lato, oppure ancora minimizzare la varianza delle pieghe per ogni arco. Questo fattore è importante per quegli algoritmi che richiedono una convenzione di visualizzazione a polilinea, in quanto le catene di segmenti aiutano la visualizzazione del grafo solo quando non sono eccessivamente lunghe. Limitandone il numero o anche semplicemente rendendolo omogeneo, si riesce ad ottenere una visualizzazione non eccessivamente pesante.

*Risoluzione angolare:* si massimizza il più piccolo angolo tra due lati incidenti sullo stesso vertice. Ciò consente di non avere due lati incidenti eccessivamente vicini, ed è un fattore molto importante per le visualizzazioni a segmenti, nelle quali i vertici sono collegati da una riga retta. Si ottiene infatti una rappresentazione molto più leggibile, senza il rischio di fraintendimenti fra due archi eccessivamente vicini.

*Simmetria:* si cerca di mettere in risalto le simmetrie del grafo, se presenti. Questo tipo di fattore estetico è molto importante per i grafi rappresentanti eventi naturali, in quanto

spesso contengono delle simmetrie molto forti. Tale fattore però è spesso all'opposto di altri fattori estetici importanti, quali ad esempio le intersezioni fra i lati.

#### VINCOLI

Un altro fattore importante nella scelta di un algoritmo di visualizzazione è la possibilità o meno di poter vincolare la rappresentazione risultante sotto determinati punti di vista. Come abbiamo visto, l'ambiente del grafo può portarci a delle scelte di visualizzazione. Alcuni algoritmi prevedono nella loro esecuzione di poter definire alcuni vincoli quali ad esempio:

*Centro*: un vertice o un insieme di vertici vengono posizionati al centro della visualizzazione. Ad esempio può essere richiesto tale vincolo per la visualizzazione della rete di strade incentrata su una determinata città, o la rete dei contatti di un determinato utente in una rete sociale.

*Esterno*: un vertice o un insieme di vertici vengono posizionati all'esterno della visualizzazione. Ciò può servire qualora si preferisca avere i nodi aventi pochi vertici all'esterno, in modo da alleggerire la lettura del grafo.

*Raggruppamenti*: si richiede di raggruppare degli insiemi di vertici in modo che siano posizionati nella stessa area. Può essere utile qualora si abbia il grafo di persone suddivise in gruppi, quali ad esempio colleghi di lavoro nella stessa area all'interno di un'azienda.

*Percorsi*: si richiede che un determinato percorso sia visualizzato orizzontalmente – o verticalmente – nella visualizzazione, per focalizzarne l'attenzione. Tale percorso potrebbe essere ad esempio il percorso minimo da un nodo sorgente ad un nodo pozzo in un qualche grafo di flusso.

*Forma*: si richiede che la visualizzazione risultante abbia una determinata forma, come ad esempio un poligono regolare. Questi tipi di visualizzazione sono spesso legati al fattore estetico della simmetria.

Si veda [KMS94] per studiare altri vincoli solitamente utilizzati e per approfondire lo studio di quelli qui presentati.

#### EFFICIENZA

L'efficienza è ovviamente un fattore fondamentale nella scelta di un algoritmo; come in qualunque campo della scienza che studia gli algoritmi, vi sono metodi più o meno complessi e più o meno performanti, in base alle azioni svolte, alle strutture dati utilizzate ed alla furbizia dell'algoritmo.

Algoritmi di visualizzazione interattivi, come quelli richiesti per le reti temporali, devono essere molto efficienti in quanto richiedono dei risultati in real time, anche per grafi molto grandi. Purtroppo l'efficienza va spesso a discapito dell'estetica, di conseguenza bisogna trovare un giusto compromesso fra ciò che si vuole e ciò che detta la realtà.

## SOCIAL NETWORKS

La rappresentazione di reti sociali ha da sempre visto la sua realizzazione tramite due principali approcci, il primo basato sulle matrici di adiacenza, il secondo basato su grafi. Il primo approccio fu utilizzato per gran parte del passato, utilizzando matrici aventi come indici sulle righe e sulle colonne i medesimi elementi. In ogni cella della matrice veniva indicato in posizione  $(i, j)$  il valore di riferimento riguardante l'interazione sociale tra l'attore  $i$  e l'attore  $j$ . Tale rappresentazione però aveva il problema che non permetteva una comprensione adeguata del fenomeno che si stava studiando.

Fu Jacob L. Moreno, attorno al 1930, a rappresentare per la prima volta una rete sociale tramite “punti e linee” per trarne informazioni: indicò infatti ogni attore soggetto a studio come un punto su di un foglio di carta, e le interazioni con gli altri attori tramite delle frecce riportanti il dato di riferimento. Fu il primo ad utilizzare un grafo per rappresentare l'interazione fra persone, ottenendo così una più chiara visione d'insieme del fenomeno da lui studiato ed ottenendo quindi informazioni preziose.

Successivamente Moreno studiò in modo più sistematico tale approccio per rappresentare reti sociali, cercando metodi per migliorare i suoi disegni e per facilitare nel maggior modo possibile la lettura dei suoi sociogrammi. Fu il primo a proporre di diminuire il numero di intersezioni tra lati, di rappresentare diversi tipi di informazioni in un unico grafo utilizzando i colori per indicare il diverso tipo di interazione, di disegnare i nodi in maniera differente in base alla tipologia di attore studiato, di disporre in zone diverse nodi con particolarità diverse permettendo una più facile visione di differenziazione. Grazie a tutte queste innovazioni è stato in grado di rappresentare sociogrammi carichi di significato, in grado di utilizzare ogni forma visiva per rappresentare informazioni.

Nel 1938 altri due studiosi di reti sociali, Lundberg e Steele, rappresentarono la famosa “Lady Bountiful” rappresentando attori particolari dal punto di vista sociale come dei nuclei di dimensioni maggiori, posizionandoli al centro del disegno, ed ottenendo una notevole e chiara rappresentazione. Quest'idea di rappresentare con dimensioni differenti determinati attori, per enfatizzare il differente valore sociale, fu un altro

fattore di rappresentazione importante assieme a quelli introdotti da Moreno nello stesso periodo.

Attorno al 1950 la visualizzazione di reti sociali ebbe il primo mutamento, introducendo alla rappresentazione manuale l'adozione dei primi calcolatori. Venivano infatti effettuati dei questionari, le cui risposte venivano inserite ed elaborate da un calcolatore. Esso non era ancora in grado di rappresentare un grafo, ma fornì un notevole aiuto nel calcolo di importanti fattori, che permisero di rappresentare, seppur a mano, grafi di dimensioni ed interesse maggiori.

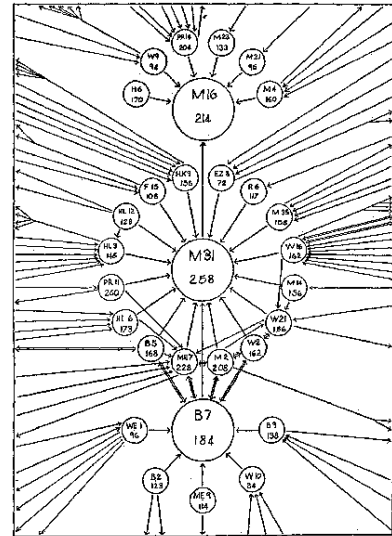


Fig. 7: *Lady Bountiful*

Nel 1970 i primi computer permisero di rappresentare reti sociali, che venivano stampate tramite plotter presenti nelle sedi universitarie o di ricerca. Tali rappresentazioni permettevano una scarsa personalizzazione ed erano solitamente di scarsa qualità, ma sollevarono per la prima volta il problema di elaborare in modo automatico un corretto posizionamento dei vertici di un grafo in modo da ottenere una rappresentazione pulita del fenomeno studiato. In questi anni vennero molto utilizzati i concetti fondati da Moreno, Lundbert e Steele, ma non prese piede l'utilizzo della differente colorazione proposta anni prima per l'impossibilità di stampare a colori.

Nel 1980 un'ulteriore evoluzione della tecnologia e lo sviluppo sempre maggiore dei computer, permise lo sviluppo dei primi software di utilizzo generale per la visualizzazione di reti sociali. Tali sistemi potevano essere eseguiti su dei computer e la rappresentazione era visualizzata a schermo. Ciò portò ad avere un sistema configurabile in base alle esigenze, che permettesse inoltre la visualizzazione a colori dei grafi. Nel 1982 iniziarono ad apparire le prime rappresentazioni in prospettiva, in grado di dare un'idea di tre dimensioni. L'interazione con la visualizzazione del grafo, una volta generata, era però ancora limitata.

Col passare del tempo sempre più applicativi di carattere generale permisero di visualizzare reti sociali, permettendo all'autore di impostare parametri di visualizzazione, interagire con il risultato e di esportare immagini condivisibili successivamente con il mondo scientifico. Fu però con l'avvento e lo sviluppo di internet nella prima metà del 1990 che si rese possibile la rappresentazione di grafi con il mondo, in grado di fornire un certo livello di interazione non solo all'autore della visualizzazione, ma anche all'utente finale. Questo grazie allo sviluppo di linguaggi e

tecnologie interattive quali VRML e MAGE ed a formati di animazioni quali GIF e MOV.

Ora, con la nascita ed il forte sviluppo di software Web 2.0, l'interazione con l'utente è diventata il punto centrale, non solo per quanto riguarda la fornitura di servizi, ma anche per la raccolta e l'elaborazione di ulteriori dati sociali. La comunità della rete è in grande fermento e tecnologie interattive quali Ajax e Flash sono un ottimo ambiente di sviluppo per quelle applicazioni sociali in grado ora più che mai di dimostrare il loro potere rappresentativo.

## **ADOBE FLEX**

Adobe Flex è un linguaggio ideato da Adobe per facilitare la realizzazione di applicativi Flash, pacchetti software interattivi basati sul web; la sua nascita è dovuta al fatto che in precedenza, per scrivere elementi in Flash veniva utilizzato un builder ad alto livello, incentrato su una barra del tempo, su oggetti visuali quali finestre, pulsanti, testi e relativi eventi di interazione. Questo approccio fu adottato in quanto Flash era principalmente un linguaggio utilizzato da grafici, e non programmatori, che lo impiegavano per realizzare degli effetti visivi per il web.

Le nuove tecnologie, il forte sviluppo di internet e lo spostamento sempre maggiore di applicativi e servizi sul web, la crescente potenza di calcolo a disposizione degli utenti e la necessità di rendere la rete più interattiva ed accattivante, hanno favorito lo sviluppo di Flash, sollevando di conseguenza la necessità di un linguaggio maggiormente programmabile: era necessario un modo differente per realizzare applicativi Flash.

Flash era, ed è tutt'ora, basato su ActionScript, un linguaggio simile a Javascript, con la capacità di supportare la programmazione ad oggetti, una forte gestione degli eventi e della rappresentazione, seppur complicata, di oggetti visuali. Ed è proprio su tale linguaggio che Flex si basa, aggiungendo però alle sue capacità di supportare la programmazione ad oggetti, una facile interazione con il web e le animazioni grafiche, nonché tutta una serie di funzionalità tipiche degli applicativi desktop quali finestre, pulsanti, tabelle e via dicendo che erano di non facile utilizzo in ActionScript.

### **VANTAGGI**

Adobe Flex è un linguaggio orientato agli oggetti ed agli eventi, con capacità di interazione con il web e l'utente ed una facile realizzazione di interfacce grafiche accattivanti e personalizzabili.

Oltre ad avere a disposizione le classi di ActionScript che permettono, tra le altre cose, di ottenere facilmente risorse da internet, Flex mette a disposizione una serie di classi di utilizzo generale per la gestione delle risorse, dalle immagini alle sorgenti di dati, quali XML o Json. Inoltre offre metodi e classi per facilitare le comuni operazioni di programmazione, quali manipolazione di stringhe, gestione di strutture dati e via dicendo.

Il vantaggio più grande di Flex però è indubbiamente la sua capacità di realizzare interfacce grafiche in modo davvero facile, veloce e personalizzabile. Queste vengono rappresentate tramite un file mxml, un particolare file xml in grado di rappresentare oggetti con attributi personalizzati, ideato dalla Macromedia, molto simile ai linguaggi XUL e XAML utilizzati rispettivamente da Mozilla e Microsoft per rappresentare interfacce grafiche. Mxml però risulta essere maggiormente personalizzabile ed elastico nelle modifiche. Permette inoltre di integrare o includere codice ActionScript, permettendo di legare in modo molto semplice gli oggetti con le loro azioni ed i loro eventi al relativo codice. Ciò permette di svincolare il modello concettuale dall'interfaccia, permettendone la modifica senza andare ad intaccare eccessivamente il codice già scritto.

La personalizzazione dello stile di visualizzazione avviene tramite un comune foglio di stile css, lo stesso usato nelle pagine web. In questo modo è possibile quindi cambiare anche drasticamente lo stile visivo dell'applicazione web realizzata, senza però modificare il posizionamento e gli attributi degli oggetti ed il codice ad essi legato.

## *SVANTAGGI*

Uno dei più grandi problemi di Flex è stato ereditato da Flash: entrambi infatti hanno problemi di performance. Essendo linguaggi orientati alla visualizzazione grafica ed alle animazioni, risultano spesso eccessivamente dispendiosi in quanto a sfruttamento della CPU.

Inoltre ActionScript non supporta ancora funzionalità multithreading, che potrebbero spesso risultare utili per effettuare in modo più performante alcune operazioni di calcolo: nonostante infatti ActionScript effettui spesso delle chiamate asincrone alla maggior parte dei metodi, in quanto sfrutta in modo pesante gli eventi, è possibile che durante certe operazioni di calcolo l'interfaccia risulti bloccata ad ogni tentativo di interazione dell'utente. Ciò accade spesso ad esempio durante il parsing di dati ottenuti dalla rete o durante la gestione di particolari strutture dati.

ActionScript in più non è esente da problemi di implementazione: non supporta ad

esempio l'override delle funzioni, come pure la definizione di metodi statici nelle interfacce di classe o l'ereditarietà multipla.

Nonostante tali problemi, Flex risulta essere un buon ambiente di sviluppo per un software per la visualizzazione di reti sociali, in quanto benché abbia problemi di performance, permette un'ottima interazione con l'utente, si rende disponibile via web ma effettua le operazioni sulle macchine degli utenti, non andando ad appesantire il server che lo ospita, ed ha a disposizione effetti grafici intriganti che permettono di sottolineare in modo efficace particolarità che possono essere interessanti per lo studio di fenomeni sociali. Infine permette con piccole modifiche di essere trasformato in codice Air.

## *ADOBE AIR*

Adobe Air è un'altra tecnologia sviluppata da Adobe, che permette di portare applicazioni web scritte in html, Flex od altri linguaggi, in ambiente Desktop. Questa caratteristica ha permesso a Flex di diffondersi maggiormente, in quanto è possibile sviluppare grazie a questa tecnologia applicazioni web e renderle disponibili anche nel classico ambiente Desktop, realizzando software polifunzionali.

Adobe Air ha infatti la caratteristica di essere cross-platform: la sua SDK<sup>1</sup> è disponibile per Linux, Mac e Windows, fornendo quindi un nuovo modo per la realizzazione di applicazioni desktop – e con Flex anche web – compatibili con ogni tipo di sistema operativo maggiormente utilizzato.

Scrivendo un unico programma è quindi possibile renderlo disponibile sia via web che tramite una classica applicazione, su sistemi windows o unix, senza variarne l'aspetto o le funzionalità. Anch'esso come Flex è basato su mxml per quanto riguarda la realizzazione di un'interfaccia grafica, ma non richiede necessariamente l'utilizzo di ActionScript per la parte di scrittura del codice.

## **FORMATI DI MEMORIZZAZIONE**

Esistono moltissimi formati per memorizzare grafi statici, la maggior parte dei quali legati strettamente ad una singola applicazione o suite. Questo perché successivamente al 1970 hanno iniziato ad essere sviluppati numerosi applicativi per la visualizzazione di grafi, senza però che nessuno emergesse rispetto agli altri. Inoltre nessuno si è mai preso il compito di formalizzare un metodo di memorizzazione univoco ed accettato da tutti.

---

<sup>1</sup> Software Development Kit, un insieme di tools per la realizzazione e l'esecuzione di applicativi.

Due formati sono ora maggiormente utilizzati, oltre alle consuete matrici di adiacenza spesso utilizzate dai software statistici, Dot e GraphML.

### *LINGUAGGIO DOT*

Dot è un semplice linguaggio che permette di memorizzare la struttura e le informazioni di un grafo. È in grado di rappresentare sia grafi che grafi diretti, e permette di associare qualunque informazione ad un lato o ad un nodo.

La sua sintassi molto semplice lo rende l'ideale per memorizzare la struttura di un grafo. Purtroppo però non ha insito il concetto di posizionamento, quindi non è adatto per la rappresentazione di visualizzazioni. In realtà, siccome permette di associare qualunque informazione agli elementi che contiene, basterebbe memorizzare per ogni vertice le coordinate del punto di appartenenza per ottenere, ma al momento non è un metodo utilizzato.

Il linguaggio Dot risulta molto leggibile per via della sua sintassi estremamente semplice e chiara. Può però portare ad avere alcuni problemi per quanto riguarda il parsing automatico, specialmente per quanto riguarda la performance.

### *GRAPHML*

Come si può intuire dal nome, GraphML è un linguaggio basato su XML. Come Dot, permette di rappresentare grafi diretti e non diretti in modo molto semplice. Inoltre supporta, proprio come Dot, campi specifici per ogni elemento, permettendo di memorizzare qualunque tipo di attributo.

Contrariamente a Dot però, risulta essere leggermente più difficile da comprendere, cosa comune a qualunque linguaggio basato su XML. Proprio perché basato su tale linguaggio strutturato permette però di realizzare parsers molto veloci e semplici da usare. Inoltre essendo XML uno standard, tali parsers sono solitamente già inclusi nella maggior parte dei linguaggi di programmazione, specialmente quelli basati sul web, rendendo l'utilizzo di GraphML veramente semplice.

Purtroppo però i file scritti in GraphML occupano spesso più spazio di un file Dot, in quanto XML ripete molte volte i termini che utilizza come tokens. D'altra parte permette di formalizzare e verificare in ogni momento la validità della sintassi, grazie ai numerosi tools di controllo XML ed ai relativi files di schema.

Due esempi di grafi memorizzati in entrambi i formati. Si può notare subito la differenza di dimensione tra i due formati, come pure la differenza nella struttura formale.

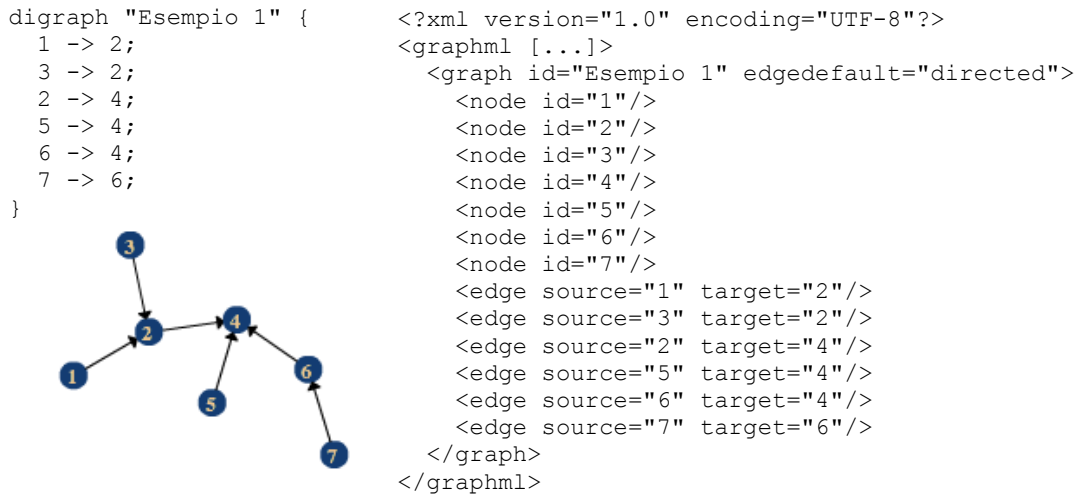


Fig. 8: rappresentazione del grafo diretto "Esempio 1".

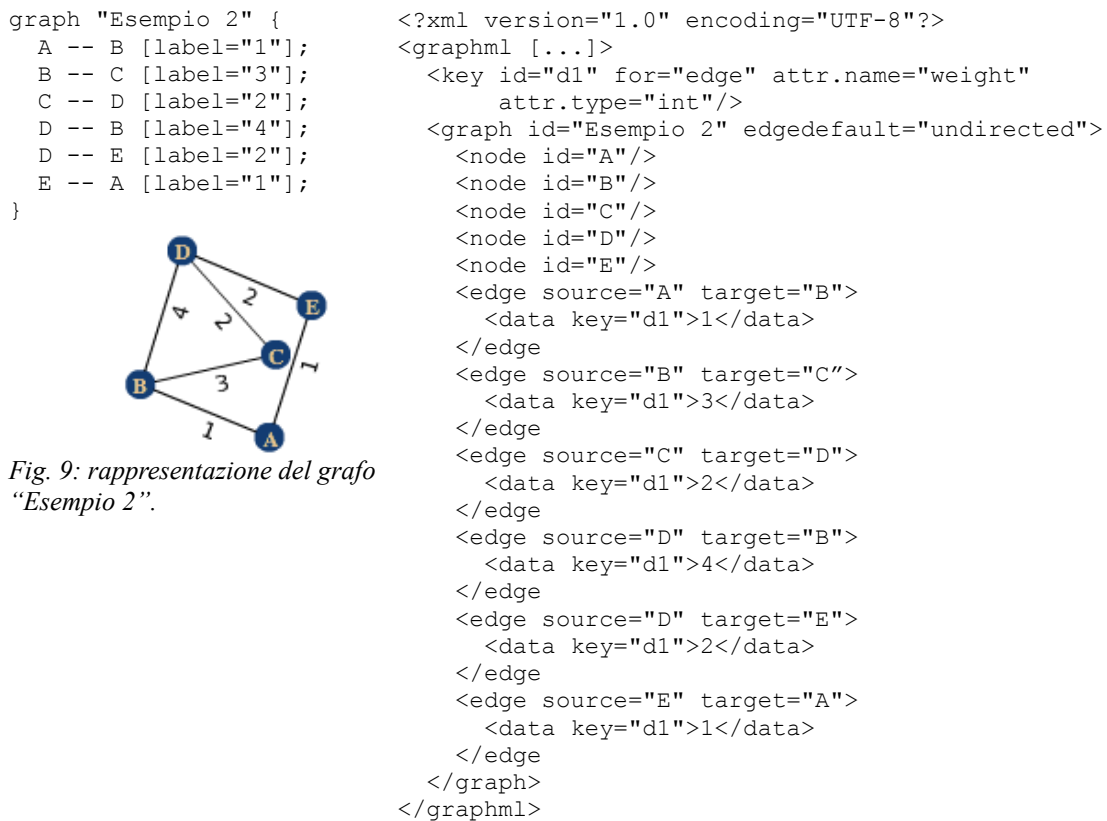


Fig. 9: rappresentazione del grafo "Esempio 2".

## CONTRIBUTI: GRAPH VISUALIZER

### VISUALIZZAZIONE GENERICA DI UN GRAFO

In generale, è possibile rappresentare graficamente un qualunque grafo. Più in particolare però, notiamo che gran parte degli algoritmi – se non tutti – lavorano su grafi connessi. Ciò è dovuto al fatto che risulta piuttosto semplice, dato un grafo qualunque, ottenere le sue componenti connesse<sup>2</sup>, eseguire l'algoritmo separatamente su ognuna di esse e ricombinare le visualizzazioni così ottenute in un'unica rappresentazione del grafo iniziale.

Ciò permette di parallelizzare la visualizzazione del grafo, ma comporta un ulteriore problema; infatti ricombinare le rappresentazioni delle componenti connesse evitando delle sovrapposizioni può risultare problematico. A questo problema sono state avanzate svariate soluzioni; in [DM06] ad esempio, si propone di aggiungere dei lati fantasma per rendere il grafo connesso, lati che poi verranno tolti nella visualizzazione finale. Tale soluzione funziona a tutti gli effetti ed ha il pregio di mantenere la qualità della rappresentazione, in quanto elimina di fatto degli elementi ad una visualizzazione ritenuta ottimale, ma comporta una crescita del tempo di elaborazione che può non essere irrilevante. In questa sezione viene presentato un algoritmo che può essere utilizzato per affiancare le rappresentazioni delle componenti connesse di un grafo.

Notiamo che più in generale può essere una buona idea effettuare un'operazione di separazione di questo genere anche per quanto riguarda un grafo connesso: basterebbe infatti localizzare i suoi vertici di taglio, eliminarli ottenendo un grafo non connesso, eseguire l'operazione sopra indicata per visualizzare le sue componenti connesse ed infine unirle in un'unica rappresentazione ripristinando i vertici di taglio. L'operazione di aggiunta dei vertici può portare ad intersezioni fra lati o più in generale al peggioramento del fattore estetico della visualizzazione risultante, ma tale operazione può comportare un notevole risparmio di elaborazione. Purtroppo non è possibile in modo automatico capire quando conviene o meno effettuare tale operazione, ma in linea generale se il grado dei vertici di taglio è basso, è probabile che si riesca ad ottenere una

---

<sup>2</sup> Tale operazione può essere facilmente eseguita tramite una visita in profondità od in ampiezza del grafo.

buona rappresentazione.

### *ALGORITMO DI AFFIANCAMENTO*

Tale algoritmo di approssimazione vuole mostrare come, data una lista di rappresentazioni di componenti connesse con le relative dimensioni, è possibile trovare una rappresentazione ottenuta affiancando tali componenti, risultando in un'area totale inferiore o uguale a 4 volte l'area ottima.

Questo algoritmo è una variante dell'euristica “First Fit Heuristic” per risolvere il problema dell'impacchettamento, che risulta essere un problema NP-difficile ([CLR90]). Consiste nel creare una griglia di rettangoli abbastanza grandi da contenere ognuno almeno una componente, facendo sempre in modo che tali rettangoli non vadano mai a sovrapporsi. Quando una componente viene aggiunta ad un rettangolo, esso viene partizionato in due rettangoli più piccoli, che potranno successivamente contenere altre componenti.

La funzione base dell'algoritmo, `place`, calcola il massimo delle larghezze e delle altezze ed inizializza la dimensione della griglia a 0. Tali dati verranno utilizzati da `expand` per creare la griglia di rettangoli. Successivamente aggiunge ogni componente connessa richiamando la funzione `add`, che restituisce le coordinate alle quali posizionare la componente.

```
place(components):
  def_width <- max(components.width)
  def_height <- max(components.height)
  rectangles <- List.empty
  size <- 0

  for each component in components:
    res <- add(component)
    component.x <- res.x
    component.y <- res.y
```

La funzione `add`, come si può intuire, aggiunge una componente connessa alla griglia, andando a cercare il primo rettangolo disponibile a contenerla. Una volta trovato lo divide in altri due rettangoli e ritorna le sue coordinate. Qualora tale rettangolo non fosse disponibile, richiama la funzione `expand` per creare altri rettangoli e si richiama ricorsivamente. La successiva chiamata troverà sicuramente in rettangolo adatto a contenere la componente connessa.

```

add(component):
  for each rect in rectangles:
    if (rect can contains component):
      split (rect, component)
      return (x: rect.x, y: rect.y)
  expand()
  return add(component)

```

La funzione `split` divide il rettangolo in due rettangoli più piccoli, in modo che non vadano a sovrapporsi. La suddivisione dei rettangoli viene effettuata dividendo il rettangolo in quattro rettangoli più piccoli, il primo dei quali delle dimensioni della componente. Risulterà quindi pieno e non utilizzabile. L'ultimo rettangolo invece verrà unito ad uno degli altri due, in questo caso a quello che risulterebbe più grande dall'unione, in modo da aumentare la probabilità che tale rettangolo venga successivamente scelto per contenere un'altra componente. Il vecchio rettangolo viene tolto dalla lista per fare posto ai due nuovi rettangoli. Questi ultimi non vanno inseriti in coda alla lista, bensì al posto del loro “padre”.

```

split(rect, component):
  r1 <- (x: rect.x + component.width, y: rect.y,
        width: rect.width - component.width, height: component.height )
  r2 <- (x: rect.x, y: rect.y + component.height,
        width: component.width, height: rect.height - component.height )

  if r1.width > r2.height):
    r1.height <- r1.height + (rect.height - component.height);
  else:
    r2.width <- r2.width + (rect.width - component.width);

  rectangles.insertBefore(rect, r2)
  rectangles.insertBefore(rect, r1)
  rectangles.remove(rect)

```

Infine `expand` crea ulteriori rettangoli qualora non ve ne siano a disposizione. Notiamo che inizialmente la lista dei rettangoli è vuota, pertanto tale funzione verrà sicuramente richiamata.

```

expand():
  size <- size + 1
  start <- (size - 1)2
  end <- (size)2
  middle <- (end - start - 1)/2 + start
  for i from start to end:
    r = ( x: if i <= middle then (size-1) * def_width
          else (end-i-1) * def_width ,
          y: if i < middle then (i-start) * def_height
          else (size-1) * def_height ,
          width: def_width, height: def_height )
    rectangles.append( r )

```

La seguente immagine illustra un esempio di funzionamento dell'algoritmo: ogni riga

rappresenta un'esecuzione della chiamata add. In prima istanza la griglia è vuota, viene richiamata la funzione expand che crea una griglia 1x1. La componente viene aggiunta. In seconda istanza la componente trova posto in uno dei rettangoli prima generati. Alla terza esecuzione però, la componente risulta eccessivamente grande; viene quindi nuovamente richiamata la funzione expand che genera altri tre rettangoli, portando la griglia ad una dimensione pari a 2x2. Nell'ultima esecuzione infine trova posto anche la quarta componente connessa.

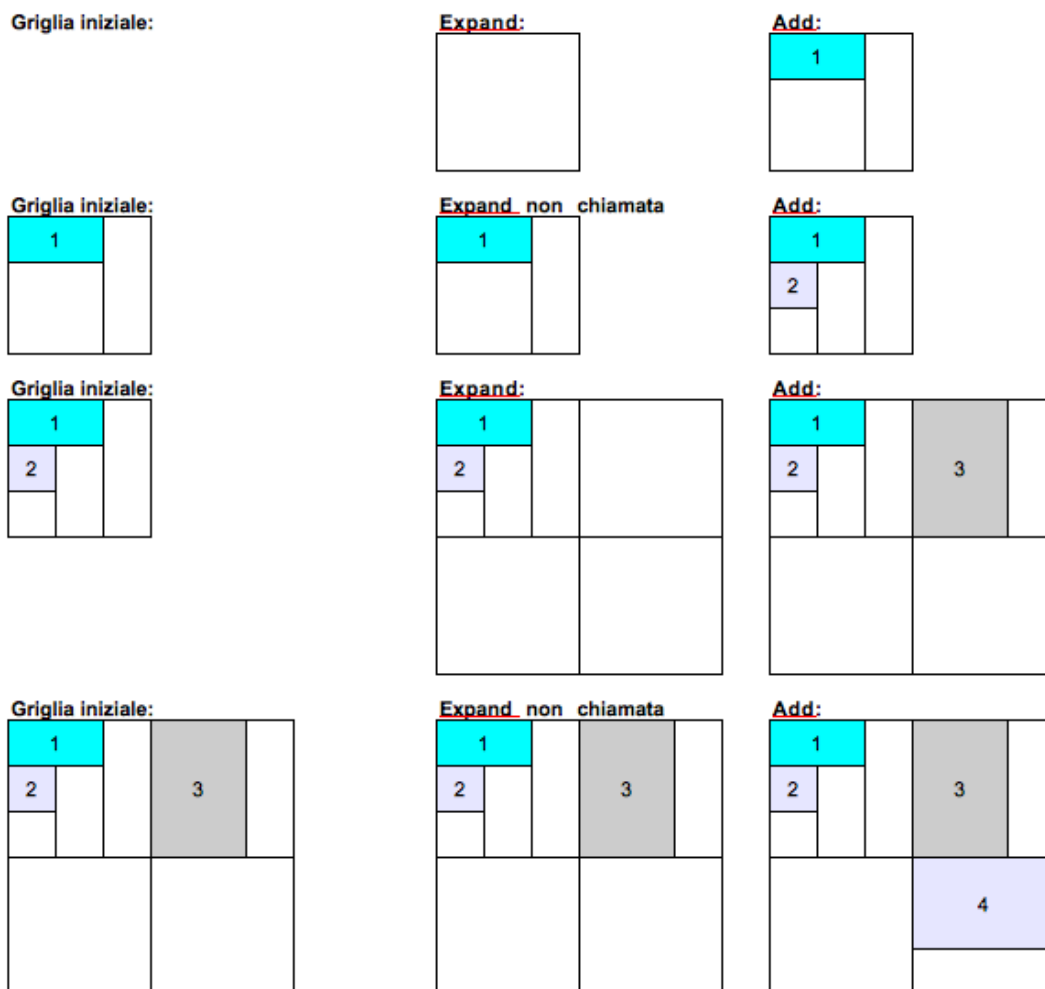


Fig. 10: Evoluzione della griglia durante l'esecuzione dell'algoritmo dell'affiancamento.

### NP-DIFFICOLTÀ

Dimostreremo che il problema dell'affiancamento è NP difficile basandoci sul fatto che lo stesso problema dell'impacchettamento risulta essere tale: se il problema dell'affiancamento non fosse NP difficile, esisterebbe un algoritmo per risolvere tale problema in tempo polinomiale. Potremmo quindi richiamare tale algoritmo con dei

rettangoli di larghezza unitaria e di altezza pari alle dimensioni degli elementi del problema dell'impacchettamento, ottenendo una soluzione a tale problema in tempo polinomiale. Siccome però tale problema è dimostrato essere NP difficile (per la prova si veda ancora [CLR90], nel quale viene dimostrata tale affermazione riducendo il problema dell'impacchettamento alla somma di sottoinsiemi) otteniamo un assurdo. La tesi di partenza è quindi falsa ed anche il problema dell'affiancamento risulta essere NP difficile.

#### FATTORE DI APPROSSIMAZIONE

Dimostriamo ora che l'algoritmo ha fattore di approssimazione pari a 4. Per fare ciò determiniamo il caso pessimo e cerchiamo di calcolarne l'area in funzione della soluzione ottima.

Il caso pessimo per l'algoritmo dell'impacchettamento risulta essere una situazione con un rettangolo di lunghezza massima e larghezza molto piccola, un rettangolo con lunghezza molto piccola e larghezza massima ed altri quadrati con dimensioni appena superiori alla metà del lato più lungo dei primi due rettangoli. In questo caso infatti l'algoritmo genera dei quadrati in grado di contenere i due rettangoli iniziali, molto grandi, nei quali però riesce a prendere posto solamente uno degli altri quadrati in input.

Definendo  $n$  come il numero di rettangoli in input e  $max$  la dimensione massima dei due rettangoli iniziali, l'area  $A$  risultante nel caso pessimo è data dalla seguente formula:

$$A = max^2(n-2)$$

La soluzione ottima  $A^*$  non può essere minore dell'area dei quadrati, da cui si ricava

$$A^* \geq (n-2)\left(\frac{max}{2}\right)^2 = (n-2)\frac{max^2}{4} \quad A \geq \frac{A}{4} \Rightarrow \frac{A}{A^*} \leq 4$$

Il rapporto tra la soluzione pessima e la soluzione ottima è proprio il fattore di approssimazione dell'algoritmo, che risulta quindi essere pari a 4.

#### COSTO COMPUTAZIONALE

L'algoritmo ha complessità massima  $O(n^2)$  dove  $n$  rappresenta il numero di componenti connesse del grafo. Questo in quanto la funzione `add`, che viene richiamata per ogni componente, ha costo  $O(n)$  nel caso pessimo. Infatti potrebbe dover scorrere tutti i rettangoli per trovarne uno adatto, ed essi sono nel caso pessimo pari a due volte il numero di componenti aggiunte fino a quel momento.

## I GRAFI TEMPORALI

I grafi temporali sono, banalmente, grafi che variano nel tempo. Vi sono svariati software per la visualizzazione di grafi, che utilizzano i più differenti algoritmi, ma sono davvero pochi quelli che permettono un'animazione temporale delle reti.

L'idea di base, dato un grafo temporale, è quella di avere una visualizzazione più o meno approfondita di come questo evolve con il passare del tempo, cercando di capirne i motivi, legandoli ad avvenimenti storici o sociali conosciuti. Il risultato potrebbe essere paragonato ad un video che rappresenta le variazioni di clima nei prossimi periodi: esso dà una visualizzazione globale di ciò che accade – o sta per accadere – permettendo di raccogliere utili informazioni sull'evoluzione dello stato nel tempo.

Purtroppo la maggior parte degli algoritmi di visualizzazione di grafi sono stati ideati per grafi statici, senza possibilità di variazioni una volta impostati i valori iniziali. Con delle modifiche è però possibile adattare alcuni di questi algoritmi, in modo da poter ottenere dei risultati accettabili anche su grafi temporali.

Questo tipo di grafi infatti ha una serie di problematiche differenti dai grafi statici; è essenziale ad esempio, variando da una visualizzazione ad un'altra dello stesso grafo, non rompere la rappresentazione mentale che l'utente si è fatto del grafo. Ciò significa che non è ammissibile spostare continuamente dei nodi in zone diverse al variare del tempo, in quanto ciò disturba l'osservazione dell'evoluzione del grafo e fa perdere di vista la consistenza della visualizzazione, facendo cadere nel caos e nell'inutilità la rappresentazione. È altresì importante evidenziare le modifiche maggiori una volta giunti ad un nuovo stato, come pure da non sottovalutare è il modo in cui tali grafi vengono memorizzati e gestiti.

### *EVENTI ED AZIONI*

Un grafo temporale può variare in diversi modi: possono essere aggiunti o rimossi vertici o lati, oppure possono essere modificati i dati associati ad essi. Ad ognuno di questi eventi va associata una particolare azione.

Nel caso di rimozione di vertici o lati, si può semplicemente decidere di far scomparire l'elemento rimosso tramite un'animazione. Qualora ci fosse la necessità di attirare l'attenzione dell'utente, si può pensare a differenti animazioni, ad esempio simulando un'esplosione variando il colore o la dimensione dell'elemento prima di farlo scomparire. Oppure ancora si può pensare di continuare a visualizzare l'elemento rimosso, aggiungendo però un fattore di trasparenza o modificandone la visualizzazione, utilizzando ad esempio un tratteggio in sostituzione ad una linea continua.

Per quanto riguarda l'aggiunta di nuovi elementi alla visualizzazione, vi è da decidere la posizione di partenza. Nel caso di un lato la scelta è piuttosto banale, in quanto esso va a collegare due nodi già esistenti. Nel caso però dell'aggiunta di un vertice, vi sono diverse possibilità: si può decidere di posizionare i nuovi vertici nell'origine – il punto  $(0, 0)$  della visualizzazione – oppure si può pensare di posizionare in modo casuale in nodo all'interno del disegno. Una scelta più interessante potrebbe essere quella di posizionare un nuovo vertice in prossimità di un suo genitore, in modo da simulare la sua “nascita”. In questo caso per genitore si intende un vertice esistente in precedenza con il quale il nuovo nodo è collegato. In tutti questi casi comunque, si ritiene necessario eseguire nuovamente l'algoritmo utilizzato per visualizzare il grafo di partenza in modo da posizionare correttamente i nuovi elementi. Affinché quest'operazione lavori in modo appropriato, è importante che l'algoritmo non vada a perturbare eccessivamente il grafo corrente, bensì parta dallo stato in cui ci si trova per arrivare ad una nuova situazione di quiete.

Qualora invece venisse cambiato un attributo associato ad un elemento del grafo, ci troviamo a dover segnalare all'utente la variazione senza disturbarlo eccessivamente dalle sue operazioni: non sempre infatti tale variazione risulta essere importante. Il metodo migliore in questo caso risulta essere la notifica delle modifiche spiegata più nel dettaglio nella sezione successiva.

## *MEMORIZZAZIONE*

Come abbiamo visto, un grafo temporale varia nel tempo. Un grafo di questo tipo può essere rappresentato in due modi differenti, a slice o ad eventi.

La *memorizzazione a slice*, o a fette, è una memorizzazione piuttosto semplice. Viene memorizzato lo stato del grafo in determinati istanti di tempo, come se si effettuassero delle fotografie del grafo mentre questo si evolve. Si andrà quindi a memorizzare dei semplici grafi statici in file differenti, riportando poi in un file riassuntivo la mappatura tra file e data di rilevamento. Se il grafo risulta essere piuttosto grande o più semplicemente poco mutevole, potrebbe essere una buona idea quella di non memorizzare l'intero grafo in un istante di tempo, bensì solamente le variazioni rispetto allo stato precedente. Ciò comporta un notevole risparmio di spazio e di tempo di lettura, anche se richiede di scorrere tutti gli stati precedenti ad un momento desiderato per ottenere la visualizzazione corretta.

La *memorizzazione ad eventi* invece consiste nel memorizzare, come dice il nome stesso, gli eventi che caratterizzano l'evoluzione del grafo nel tempo. Verrà quindi memorizzato lo stato iniziale del grafo e successivamente, per ogni aggiunta, rimozione

o modifica di un qualunque elemento del grafo, verrà indicato l'evento ed il relativo offset dall'inizio dell'animazione. Tale memorizzazione permette di conoscere lo stato di evoluzione del grafo in ogni istante di tempo desiderato, diversamente dalla memorizzazione a slice che risulta vincolato ai momenti rilevati. La memorizzazione ad eventi risulta però più complessa da gestire, quindi viene spesso preferito il primo tipo di visualizzazione.

Il sistema sviluppato adotta una memorizzazione a slice, utilizzando un file xml per rappresentare il grafo temporale, contenente indicazioni sul grafo e le associazioni tra istanti di tempo e files in cui è memorizzata la relativa rappresentazione del grafo in quell'istante. Un esempio di tale file è il seguente, che rappresenta il grafo temporale del grado di fiducia degli utenti di Advogato durante la sua prima fase di sviluppo. I grafi relativi sono memorizzati in formato Dot.

```
<?xml version="1.0" encoding="utf-8"?>
<temporal_graph edges="TrustletEdge">
  <graph date="2000-02-25">advogato-graph-2000-02-25.dot</graph>
  <graph date="2000-07-18">advogato-graph-2000-07-18.dot</graph>
  <graph date="2000-08-11">advogato-graph-2000-08-11.dot</graph>
  <graph date="2000-09-28">advogato-graph-2000-09-28.dot</graph>
</temporal_graph>
```

## NOTIFICA DELLE MODIFICHE

Un modo innovativo ed interessante per notificare all'utente le modifiche di un qualunque elemento del grafo durante l'animazione temporale, è quello di utilizzare lo sfondo sul quale tali elementi vengono visualizzati per attirare l'attenzione su di essi.

L'idea di fondo è quella di colorare l'area attorno all'elemento aggiunto o modificato in maniera più o meno intensa in base all'entità della modifica. Il modo più semplice per fare ciò è quello di modificare due fattori, l'ampiezza dell'area e la sua trasparenza:

*L'ampiezza* dell'area può essere più o meno grande in base all'entità della modifica avvenuta, in base a quanto importante risulta essere tale

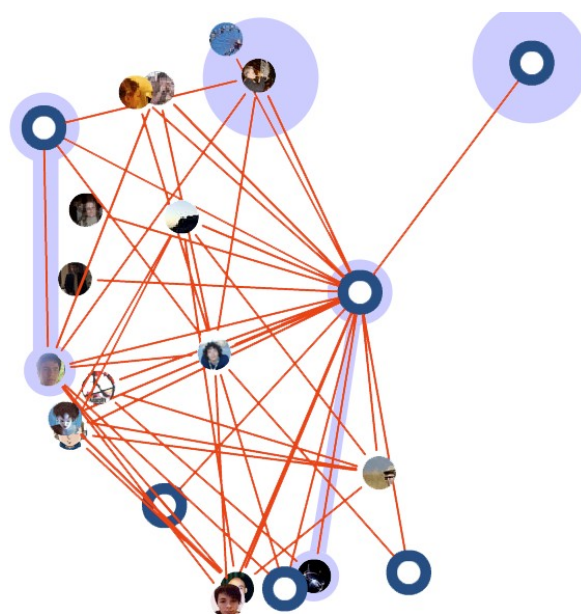


Fig. 11: L'ampiezza può variare in base alla tipologia di cambiamento...

cambiamento. Maggiore è l'area colorata, maggiore sarà l'attenzione che è in grado di catturare sull'utente. Ad esempio per quanto riguarda un vertice, l'area potrebbe essere legata ad un attributo del vertice stesso, oppure al suo grado. Un'area grande potrebbe includere anche vertici vicini, per sottolineare che la modifica appena avvenuta potrà comportare modifiche ai nodi circostanti, che spesso sono direttamente legati all'elemento in questione. Per quanto riguarda i lati, l'ampiezza dell'area potrebbe essere legata ad un suo attributo nel caso di un grafo pesato, o

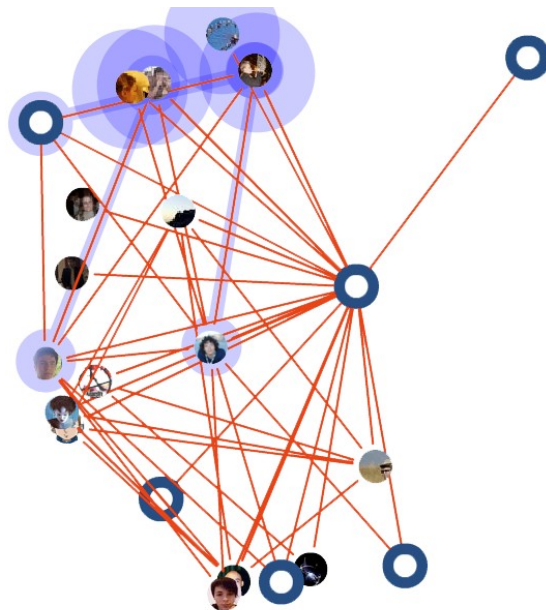


Fig. 12: ... mentre la trasparenza può indicare più modifiche nella stessa area.

in base alla sua lunghezza qualora abbia un significato particolare nella rappresentazione corrente. È buona norma colorare anche l'area circostante i vertici collegati dal lato in questione, per sottolineare il fatto che il loro stato è cambiato per via dell'aggiunta del nuovo lato.

La *trasparenza* dell'area può essere utilizzata per evidenziare maggiormente più modifiche che interessano la stessa zona. Tenendo ad esempio costante la trasparenza per ogni area visualizzata, due modifiche differenti potrebbero sovrapporsi in una zona della rappresentazione formando un'area più scura che indica una modifica cumulativa. L'area circostante un nuovo vertice legato a svariati altri lati risulterebbe quindi molto più accentuata rispetto all'area circostante un nuovo vertice singolo, attirando quindi l'attenzione dell'utente in maniera molto differente.

Grazie a questo tipo di notifica, è possibile osservare l'evoluzione di un grafo nel suo insieme focalizzando l'attenzione sulle zone di maggior modifica, senza entrare nel dettaglio delle singole variazioni.

## VERTICI ED ARCHI

Nel sistema sviluppato, si è cercato di generalizzare la rappresentazione di nodi ed archi, in maniera da rendere il più possibile elastica la scelta degli elementi da visualizzare nei diversi grafi. Sono quindi state realizzate delle classi base generiche in grado di fornire gli elementi base di gestione, permettendo così di estendere tali classi

concentrandosi solamente sulla parte di visualizzazione ed estetica. Più nel dettaglio sono stati realizzati i seguenti elementi:

*Generic Vertex*: è il vertice generale, che oltre a fornire tutti i metodi base per ogni tipo di vertice, viene rappresentato in maniera piuttosto semplice ed accettabile per qualunque tipo di grafo.

*Labeled Vertex*: si tratta di un vertice che eredita da *Generic Vertex*. Come lui possiede una rappresentazione semplice, ma vi aggiunge un piccolo testo che ne visualizza il nome. Qualora però esso fosse troppo lungo hanno luogo problemi di visualizzazione.

*Lastfm Vertex*: si tratta di un vertice utilizzato per rappresentare una rete sociale basata su Last.fm, in grado di visualizzare un piccolo avatar dell'utente di riferimento. Richiede che al vertice sia associato un attributo "image", in caso contrario il nodo viene visualizzato come un normale *Generic Vertex*.

*Generic Edge*: trattasi dell'arco generale. Come per il *Generic Vertex*, esso effettua le operazioni di base che riguardano tutti i lati. In base al tipo di grafo viene rappresentato come una freccia o come un semplice segmento retto.

*Labeled Edge*: sottoclasse di *Generic Edge*, questo lato ha la caratteristica di indicare il relativo peso nel suo punto medio, ruotando il testo in funzione della pendenza del lato stesso. Il testo visualizzato è il valore dell'attributo "value" associato al lato.

*Lastfm Edge*: si tratta di un lato utilizzato in coppia con *Lastfm Vertex* per rappresentare reti sociali di Last.fm. Viene visualizzato più o meno largo in base all'affinità musicale tra i due utenti che lega. Tale informazione dev'essere memorizzata come valore dell'attributo "tastemeter".

*Trustlet Edge*: si tratta di un lato che associa ad ogni livello di fiducia che un utente ha rispetto ad un altro utente un determinato valore numerico compreso tra 0 ed 1. Tale valore numerico viene poi utilizzato come livello di trasparenza del lato, in modo da essere più marcato per livelli di fiducia alti ed andando via via a scalare. Tali livelli di fiducia sono "Master", "Journeyer", "Apprentice" ed "Observer" e sono generati dal sistema sociale TrustLet.

Altri tipi di vertici o nodi sono stati realizzati, in base al tipo di grafo che è stato provato con l'applicazione. La tipologia di elemento da utilizzare in un determinato grafo temporale viene indicata nel file xml che viene passato all'applicazione.

Ovviamente è possibile realizzare altri elementi semplicemente creando una classe che erediti *GenericVertex* o *GenericEdge*.

## ALGORITMI DI VISUALIZZAZIONE

Si è cercato nel sistema sviluppato di generalizzare il più possibile l'applicazione di algoritmi al grafo da visualizzare. È stato innanzitutto scelto di definire algoritmi che operino solamente su componenti connesse di grafi. Il sistema, una volta ottenuto il grafo, lo divide automaticamente nelle sue componenti, esegue sequenzialmente l'algoritmo selezionato su ognuna di esse e poi utilizza l'algoritmo dell'affiancamento spiegato in precedenza per posizzarle in modo da evitare sovrapposizioni.

Differenti algoritmi operano in differenti modi: alcuni ad esempio calcolano le posizioni dei vertici una sola volta, mentre altri richiedono di iterare più volte, partendo ogni volta dallo stato dell'esecuzione precedente. Per generalizzare questo concetto, è richiesto che ogni algoritmo ritorni, una volta terminato, una funzione da eseguire successivamente sul grafo, permettendo in questo modo l'iterazione degli algoritmi e l'animazione dei nodi. Per terminare tale iterazione, o semplicemente per non iterare, basta che l'algoritmo ritorni una funzione nulla.

La funzione generica per visualizzare un grafo dato un algoritmo risulta quindi essere la seguente:

```
draw(graph, algorithm):
  for c in graph.components:
    f = algorithm(c)

  animate(graph);

  if f is not null:
    callLater(draw(f));
```

dove indichiamo con *animate* la funzione che sposta i vertici nelle coordinate elaborate dall'esecuzione dell'algoritmo e con *callLater* una funzione che richiama il suo argomento una volta che l'animazione è terminata.

Ogni algoritmo quindi, come abbiamo detto, deve prendere in input una componente connessa ed effettuare delle operazioni su di essa, impostando le nuove coordinate per i suoi vertici. In generale però un algoritmo ha bisogno di svariati parametri, oltre ad avere o meno necessità di iterazione. Per far sì che ci si possa concentrare maggiormente sulla singola esecuzione dell'algoritmo senza pensare a questi fattori, ogni algoritmo è solitamente composto da tre metodi differenti, solo uno dei quali risulta essere pubblico.

Il primo metodo è quello pubblico che viene inizialmente utilizzato per rappresentare il grafo ed avviare l'animazione. Prende quindi in input una componente connessa e ritorna eventualmente una funzione da eseguire successivamente. Di fatto tale metodo

serve solamente per incapsulare quello seguente, che svolge il compito di iterare:

```
algorithm(component):
    return algorithm_iterate(component, #iterations)
```

Il secondo metodo serve ad iterare<sup>3</sup>; esso svolge principalmente tre funzioni, la prima è quella di verificare lo stato dell'iterazione, eventualmente terminandola qualora si rendesse necessario, la seconda è quella di richiamare il terzo metodo – quello volto ad eseguire a tutti gli effetti le operazioni sulla componente connessa – la terza è quella di costruire la funzione da ritornare una volta terminata l'elaborazione.

```
algorithm_iterate(component, iter):
    if iter = 0:
        return null

    algorithm_elaborate(component, params)

    return ( function(c): algorithm_iterate(c, iter-1) )
```

Come possiamo vedere, tale metodo termina le iterazioni senza elaborare alcunché qualora sia stato raggiunto il numero di iterazioni desiderato. Altrimenti esegue l'elaborazione sulla componente connessa, per poi ritornare una funzione che richiama l'iterazione stessa, indicando però un numero decrementato di iterazioni rimanenti.

Questo schema è comune a tutti gli algoritmi implementati, a meno di qualche piccola modifica sui primi due metodi. La funzione *algorithm\_elaborate* invece varia da algoritmo ad algoritmo, permettendo al programmatore di concentrarsi sull'elaborazione singola senza pensare a tutto ciò che riguarda la gestione del grafo.

## ALGORITMI BASE

Gli algoritmi di base sono molto semplici, non sono spesso in grado di realizzare una visualizzazione apprezzabile di un grafo, ma risultano essere la base iniziale per molti algoritmi adattativi, ossia algoritmi che necessitano di un punto di partenza sul quale appoggiarsi per effettuare le dovute elaborazioni. Per tali algoritmi un buon stato di partenza può migliorare di gran lunga l'esecuzione nel tempo, di conseguenza è bene saper scegliere quale algoritmo di base utilizzare.

### RANDOM DRAW

L'algoritmo di visualizzazione casuale non fa altro che posizionare in modo randomizzato i vertici della componente connessa su cui opera. Prende come parametri

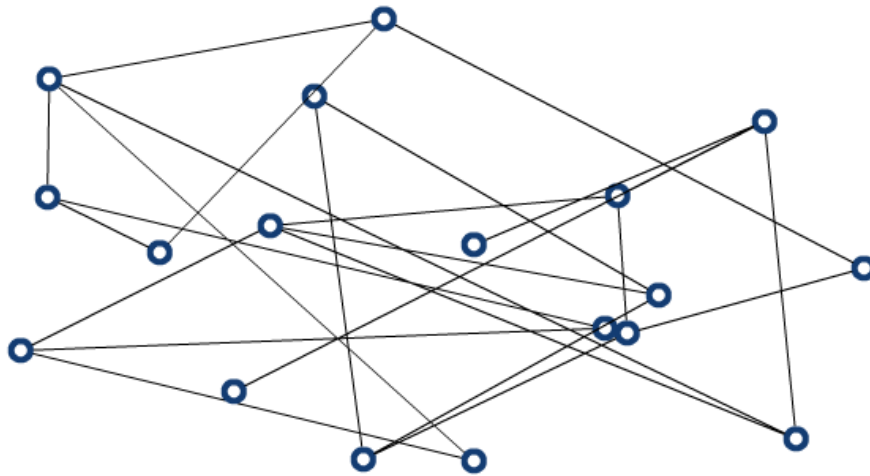
---

<sup>3</sup> Notiamo che qualora l'iterazione non fosse necessaria, basterebbe che il primo metodo richiamasse il secondo con *#iterations* pari a 1.

la larghezza e l'altezza del rettangolo entro il quale può disporre i vertici, anche se potrebbe essere una futura implementazione quella di determinare tale area in base al numero di vertici presenti nella componente.

Nonostante tale algoritmo sia effettivamente banale, è bene fare un'osservazione: un algoritmo come questo non è al momento adatto alla visualizzazione di grafi temporali. Si deve pensare infatti che esso verrà richiamato anche durante l'evoluzione del grafo, comportando lo spostamento di tutti i vertici e portando quindi ad annullare ogni speranza di comprensione dell'animazione. Per risolvere questo problema si può pensare di spostare in modo casuale solamente i nuovi vertici, lasciando quelli già presenti in precedenza nella medesime coordinate. In questo modo l'utente vedrà comparire i nuovi vertici in posizioni casuali, lasciando però intatta l'idea mentale che si era fatto sui nodi preesistenti.

La funzione di elaborazione risulta essere piuttosto semplice, quindi non andremo a studiarla. Inoltre random draw non fa utilizzo di iterazione, di conseguenza il metodo *RandomDraw\_iterate* verrà richiamato con parametro *#iterations* pari a 1.



*Fig. 13: Random Draw applicata all'interazione tra dei monaci durante la sua permanenza come visitatore.*

#### CIRCLE DRAW

Un altro semplice algoritmo di visualizzazione consiste nel posizionare i vertici in circolo, in modo che siano equidistanti fra di loro. Bensì sia un algoritmo piuttosto banale, si tratta già di un sistema che cerca di ottimizzare un fattore estetico, in questo caso la simmetria. Questa disposizione di vertici veniva utilizzata da Moreno nei primi anni del 1930 quando non aveva una particolare idea di come posizionare i nodi nel suo

sociogramma. Essendo infatti piuttosto ordinata e lineare, permette di avere un'idea generale del grafo rappresentato, permettendo poi di scegliere un algoritmo più complesso per migliorarne la visualizzazione.

Come per RandomView, anche questo algoritmo non necessita di iterazioni. La funzione di elaborazione utilizza semplici funzioni matematiche per calcolare il posizionamento dei vertici:

```
circleDraw_elaborate(component, width, height):
  card = component.|V|
  offset = 2π / card
  β = -π
  for v in component.V:
    v.toX = cos(β) * width/2
    v.toY = sin(β) * height/2
    β = β+offset
```

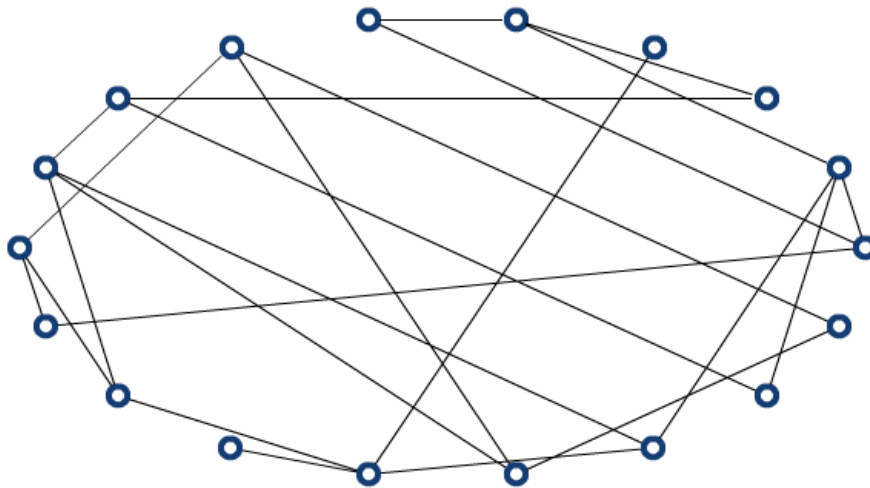


Fig. 14: Circle Draw sul medesimo grafo dell'illustrazione precedente.

Come possiamo notare da questa illustrazione, l'algoritmo CircleDraw riesce a dare una visione d'insieme più piacevole rispetto a RandomDraw.

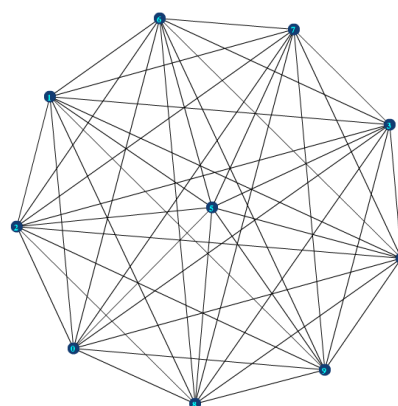
## ALGORITMI BASATI SU FORZE

Gli algoritmi basati sulle forze fanno parte di una classe di tecniche di visualizzazione che si adattano molto bene ai grafi temporali per il modo in cui sono strutturati. Essi trattano gli elementi del grafo – statico – come degli oggetti con proprietà fisiche, calcolando le forze di interazione che gli uni esercitano sugli altri e cercando una situazione il più possibile equilibrata. Per fare ciò possono effettuare un'unica elaborazione oppure possono iterare più volte fino ad ottenere un'energia complessiva del sistema inferiore ad una soglia prefissata.

Sono molto adatti ai grafi temporali per il seguente motivo: per calcolare le forze interne al sistema essi necessitano che i vertici del grafo risiedano già in una posizione spaziale. Per far ciò, quando lavorano su di un grafo statico, vengono solitamente eseguiti degli algoritmi base quali RandomDraw o CircleDraw per ottenere uno stato iniziale dal quale iniziare ad elaborare le forze. Successivamente, qualora ce ne sia bisogno, viene utilizzato lo stato finale di un'elaborazione come punto di partenza della successiva, andando così a formare grafo in movimento che con iterazioni successive si avvicina sempre più ad una situazione di quiete. Nel caso di grafi temporali, basta quindi che lo stato iniziale dell'elaborazione della visualizzazione del grafo al tempo  $t$  si basi sullo stato finale dell'elaborazione al tempo  $t-1$ , senza effettuare l'inizializzazione tramite un algoritmo base, ottenendo come risultato che le modifiche apportate modificano l'energia del sistema portandolo ad evolvere ad un nuovo stato di quiete. Quando avviene l'inserimento delle modifiche nel grafo non si va a modificare la posizione dei nodi o dei lati già esistenti, non andando quindi ad intaccare la rappresentazione mentale che l'utente potrebbe avere del grafo. Inoltre la perturbazione del sistema non comporta semplicemente la modifica delle forze in gioco, rimettendo in moto il sistema di animazioni del grafo che lo porta ad un nuovo stato di quiete.

Tali algoritmi hanno per contro il fatto che spesso sono piuttosto lenti, soprattutto per via del fatto che possono essere necessarie numerose iterazioni per raggiungere uno stato di quiete accettabile.

Gli algoritmi basati sulle forze enfatizzano solitamente le simmetrie presenti nei grafi, andando però a discapito di altri fattori estetici importanti, primo di tutti il numero di intersezioni dei lati: è pressoché impossibile infatti ottenere una rappresentazione planare di un grafo anche piuttosto semplice con tali algoritmi, in quanto non si interessano di questi fattori.



*Fig. 15: Un grafo completo disegnato con un algoritmo basato sulle forze con stato iniziale casuale. La simmetria è resa alla perfezione.*

Un argomento dibattuto richiede di trovare un momento nel quale terminare l'iterazione dell'algoritmo, ove presente. La scelta più facile indica di fermarsi quando l'energia interna è inferiore ad una data soglia, ma ci si può fermare anche qualora la variazione di energia da uno stato ad un altro risulta essere eccessivamente bassa, indicando che ormai i vertici non subiscono troppe variazioni di posizione. Come però insegna la ricerca locale stocastica, il fatto che tra due stati vi sia poca variazione non significa che in stati evolutivi futuri non vi sia un drastico cambiamento di stato. Si punta quindi a fermare l'iterazione qualora essa non porti ad

avere alcuna modifica affatto, cosa peraltro probabile in uno stato di coordinate discrete e di approssimazioni fra numeri reali e relative rappresentazioni in un computer.

Un metodo per determinare la soglia di terminazione dell'algoritmo consiste nell'utilizzare un metodo di Cooling Schedule, nel quale viene inizialmente impostata una soglia alta, che va via via "raffreddata" generando delle visualizzazioni successive sempre più adatte alla visualizzazione. Questo metodo è l'ideale per ottenere in tempi brevi una visualizzazione accettabile, andando ad affinarla con il passare del tempo.

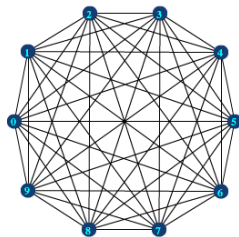
Un altro fattore interessante potrebbe essere lo studio degli algoritmi al variare del tipo di distanza utilizzata; molti degli algoritmi basati sulle forze, e non solo, utilizzano la distanza euclidea per effettuare i propri calcoli. Può essere interessante studiare il comportamento di tali algoritmi utilizzando altre definizioni di distanze, quali ad esempio la Manhattan Distance. Effettuando alcuni veloci test pare che utilizzando proprio tale distanza si ottengono rappresentazioni più compatte e meno circolari rispetto alla stessa visualizzazione effettuata con la distanza euclidea. Ciò è probabilmente dovuto al fatto che archi non orientati orizzontalmente o verticalmente attirano i due vertici con una forza maggiore, per via della definizione di distanza. I vertici vengono quindi a trovarsi più vicini, se non in quelle situazioni in cui essi sono sulla stessa retta orizzontale o verticale. Si ha quindi una specie di rappresentazione a rombo. È inoltre interessante notare l'effetto che la Manhattan Distance ha sulla rappresentazione di un grafo completo. Esso infatti mantiene la sua simmetria, portando però verso il centro i due vertici più esterni orizzontalmente.



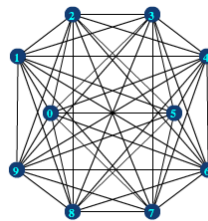
*Fig. 16: Grafo con distanza euclidea.*



*Fig. 17: Grafo con Manhattan Distance*



*Fig. 18: Grafo completo con distanza euclidea.*



*Fig. 19: Grafo completo con Manhattan Distance.*

## FRUCHTERMAN – REINGOLD

Quello di Fruchterman e Reingold è un algoritmo del 1991, uno dei più famosi algoritmi basati sulle forze. Essi hanno pensato di interpretare ogni vertice del grafo come un elemento carico elettricamente ed ogni arco come una molla collegata a due cariche, ottenendo un sistema nel quale i vertici si respingono in quanto aventi la stessa carica, ma con delle forze di attrazione / repulsione dovute alle molle che li collegano.

In questa situazione la forza esercitata su un nodo  $v$  risulta essere la somma vettoriale delle forze di repulsione dovute a tutti gli altri nodi e la somma delle forze attrattive – o repulsive, in base allo stato della molla – dovute ai vertici collegati a  $v$ .

Lo stato iniziale del grafo viene costruito eseguendo una `RandomDraw`; successivamente vengono calcolate le forze su ogni nodo, effettuando poi per ognuno di essi uno spostamento proporzionale alla forza esercitata su di esso. Tale operazione viene effettuata iterativamente, fino ad ottenere una situazione di quiete.

L'energia tra due vertici collegati da una molla segue la legge di Hooke, ossia è proporzionale alla differenza tra la distanza tra i due vertici e la lunghezza naturale della molla. La componente lungo l'asse  $x$  della forza elastica esercitata su  $v$  risulta quindi essere:

$$\sum_{(u,v) \in E} k_1 (d(p_u, p_v) - l_m) \frac{x_v - x_u}{d(p_u, p_v)}$$

dove indichiamo con  $d$  la distanza euclidea fra due punti;  $l_m$  rappresenta la lunghezza a riposo della molla, ossia la lunghezza tale per cui non viene esercitata alcuna forza tra le particelle collegate da essa;  $k_1$  rappresenta invece la rigidità della molla, più cresce e più due nodi collegati tenderanno ad avere una distanza pari ad  $l_m$ . La componente lungo l'asse  $y$  è pressoché identica.

La forza esercitata su  $v$  dagli altri vertici segue invece una legge con proporzionalità inversa quadratica:

$$\sum_{(u,v) \in V \times V} \frac{k_2}{d(p_u, p_v)^2} \frac{x_v - x_u}{d(p_u, p_v)}$$

Anche qui indichiamo con  $d$  la distanza euclidea, mentre  $k_2$  rappresenta la forza repulsiva tra i vertici.

Sono state proposte altre euristiche per quanto riguarda il calcolo delle forze d'interazione. In [Ead84] ad esempio viene proposta la seguente formula logaritmica per il calcolo della forza elastica dovuta agli archi:

$$k_1 \log\left(\frac{d(p_u, p_v)}{l_m}\right) \frac{x_v - x_u}{d(p_u, p_v)}$$

Tali euristiche però sono state spesso accantonate, in quanto non si riesce a trovare un motivo sufficientemente valido per adottarle in sostituzione a quelle proposte da Fruchterman e Reingold.

Una volta implementato l'algoritmo appena descritto, è possibile modificare i parametri  $l_m$ ,  $k_1$  e  $k_2$  in base al grafo ed al suo significato. Ad esempio  $l_m$  rappresenta la distanza desiderata fra due nodi collegati, se il legame di collegamento è forte, conviene impostare tale parametro ad un valore piuttosto piccolo, in modo che venga enfatizzato il legame tra due nodi con il fatto che verranno posizionati vicini.

Effettuando alcune prove con questo algoritmo è stato rilevato che a volte potrebbe portare ad una situazione in cui lo stato di quiete non viene raggiunto, tutt'altro i nodi vengono allontanati sempre più portando ad un'esecuzione indefinita dell'algoritmo. Questo comportamento è dovuto al fatto che due nodi eccessivamente distanti collegati da un arco potrebbero esercitare l'un l'altro una forza elastica eccessivamente grande, tanto da farli scambiare la posizione e superarsi, causando al passo successivo una forza ancora maggiore e così via. Tale problema è stato riscontrato anche in altre ricerche, quali ad esempio [DM06]. Per risolverlo basterebbe introdurre un fattore di lunghezza massima delle molle, cosa per altro del tutto legittima dal punto di vista fisico.

Qui di seguito si può trovare un'immagine rappresentante l'evoluzione di un grafo riguardante l'interazione fra gli utenti di un social network in fase di sviluppo, su cui viene eseguito l'algoritmo di Fruchterman e Reingold. Come possiamo vedere, i nodi assumono via via una posizione più omogenea e simmetrica. Il grafo potrebbe comunque risultare caotico, proprio in quanto l'algoritmo non si concentra sull'eliminare le intersezioni fra i lati del grafo. In questo caso comunque un fattore repulsivo maggiore avrebbe fatto allontanare maggiormente i nodi fra di loro, facilitando ulteriormente la lettura del grafo.

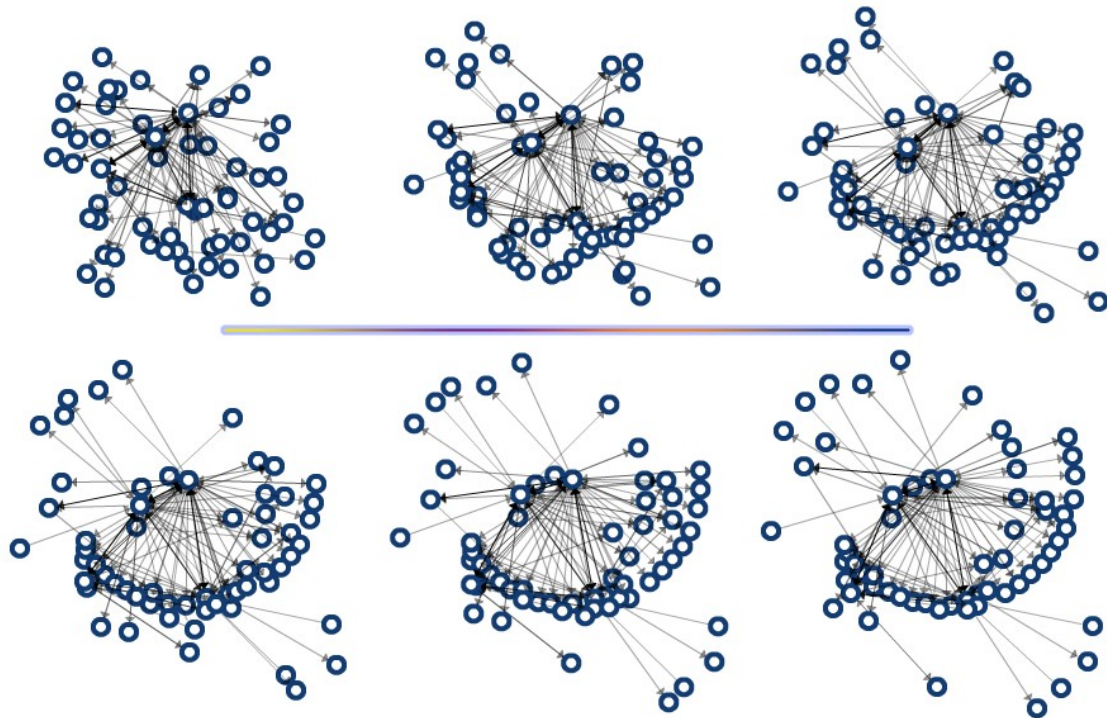


Fig. 20: L'evoluzione di un grafo sul quale è stato applicato l'algoritmo di Fruchterman e Reingold, partendo da una disposizione casuale di larghezza e altezza 200 pixels e con parametri dell'algoritmo impostati a  $l_m = 100$ ,  $k_1 = 3$  e  $k_2 = 500$ .

#### METODO DEL BARICENTRO

Il metodo del baricentro, uno dei primi algoritmi di visualizzazione, fu ideato da W. T. Tutte negli anni '60. Esso può essere visto come una variante dell'algoritmo di Fruchterman e Reingold, anche se quest'ultimo è stato ideato circa trent'anni dopo.

In tale algoritmo viene utilizzata una lunghezza naturale delle molle nulla, una costante di rigidità unitaria e non viene aggiunta forza elettrica fra vertici. Con  $l_m = 0$ ,  $k_1 = 1$  e  $k_2 = 0$  la forza esercitata su un vertice  $v$  risulta quindi essere

Ovviamente l'equilibrio in questa situazione risulta essere la soluzione banale in cui tutti i vertici convergono nello stesso punto. Per evitare ciò, si vincolano certi vertici, almeno

$$\sum_{(u,v) \in E} d(p_u, p_v)$$

tre, ad una posizione fissata, lasciando gli altri liberi di muoversi. Solitamente i nodi fissi vengono posti lungo i vertici di un poligono convesso.

Banalmente, le coordinate dei vertici liberi possono essere calcolate risolvendo le

equazioni  $F(v) = 0$  per ogni vertice libero  $v$ . Supponendo quindi che  $N_0$  rappresenti i vicini fissi di  $v$  e che  $N_1$  ne rappresenti i vicini liberi, si ottiene per le coordinate  $x$  e similmente per le coordinate  $y$ . In tale formula abbiamo indicato con  $\text{deg}(v)$  il grado

$$\sum_{(u,v) \in E} (x_u - x_v) = 0 \quad \Rightarrow \quad \text{deg}(v)x_v - \sum_{u \in N_1(v)} x_u = \sum_{w \in N_0(v)} x_w^*$$

del vertice  $v$  e con  $(x^*, y^*)$  le coordinate di un vertice fisso.

Notiamo che tali equazioni sono lineari ed in egual numero alle variabili libere. Ponendole quindi a sistema e risolvendolo con un normale algoritmo di risoluzione di sistemi lineari, si ottiene la soluzione ottima.

Tale soluzione pone ogni vertice libero nel baricentro dei suoi vicini, da qui il nome dell'algoritmo. Ciò comporta però un problema piuttosto importante, ossia che ogni vertice con un solo vicino andrà a sovrapporsi con esso, portando ad avere una visualizzazione non corretta del grafo. Tale problema può essere risolto scegliendo proprio tali vertici come fissi sul poligono convesso, facendo sì che non possano essere mossi e che quindi fungano da punto di ancoraggio per gli altri vertici. Un'altra soluzione sarebbe quella di modificare la formula base utilizzata per calcolare la forza, aggiungendo un fattore di repulsione che quindi porterebbe a non far sovrapporre i vertici.

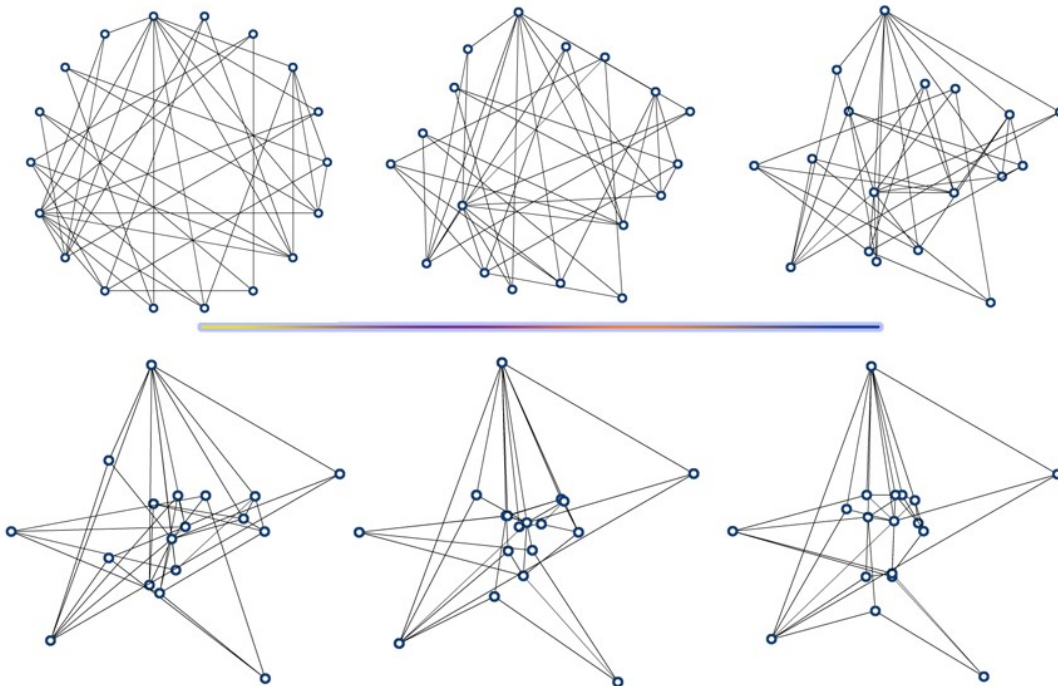


Fig. 21: Esempio di metodo del baricentro applicato al grafo di interazione tra monaci. Si nota il problema della sovrapposizione dei nodi dovuta all'assenza di forza respingente.

Per contro Tutte dimostrò che tale algoritmo eseguito su un grafo planare triconnesso  $G$  utilizzando i vertici di una qualunque faccia di un'immersione di  $G$  come vertici fissi, disposti su un poligono convesso, fornisce come risultato una rappresentazione planare e convessa di  $G$ .

Varianti dell'algoritmo del baricentro sono state studiate, utilizzando anche formule differenti per il calcolo della forza attrattiva tra i vertici; ad esempio è stato utilizzato il quadrato della distanza come energia fra due vertici.

Oltre al problema dei vertici sovrapposti, è stato dimostrato ([EG96]) che esistono dei grafi sui quali il metodo del baricentro genera una rappresentazione con area esponenziale, come nell'esempio qui a fianco. Come possiamo vedere la visualizzazione esalta comunque la simmetria del grafo, ma la grande differenza di distanze fra il nucleo centrale ed i nodi fissi all'esterno rende difficile lettura la visualizzazione. Il grafo è lo stesso di pagina 34, ossia un grafo completo con 10 vertici. È palese la differente qualità delle rappresentazioni.

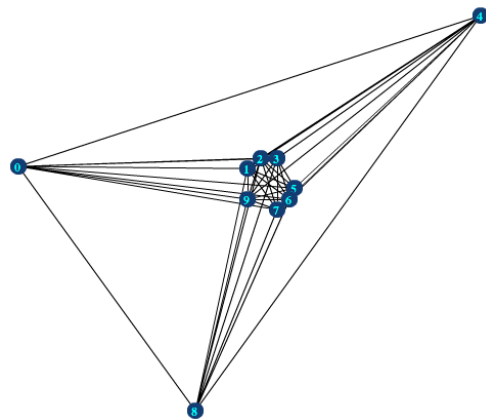


Fig. 22: Visualizzazione tramite metodo del baricentro con area esponenziale.

#### KAMADA KAWAI

L'algoritmo ideato da Kamada e Kawai nel 1989 propone di utilizzare la distanza teorica fra i nodi, legandola alla classica distanza euclidea. In un grafo connesso  $G = (V, E)$  la distanza teorica fra due nodi  $u$  e  $v$ , definita come  $\delta(u, v)$ , è data dal numero di lati del percorso più breve tra  $u$  e  $v$ .

L'obiettivo di questo algoritmo è quello di trovare una visualizzazione nella quale la distanza euclidea è proporzionale il più possibile alla distanza teorica, ottenendo una visualizzazione in grado di rappresentare correttamente la distanza fra tutti i vertici. Per fare ciò la forza fra due vertici deve essere proporzionale alla differenza fra la distanza euclidea e la distanza teorica.

Kamada e Kawai ebbero una visione energetica di questa intuizione, definendo l'energia potenziale di una molla tra  $u$  e  $v$  come l'integrale della forza che la molla esercita, ossia

$$\frac{1}{2} k_1 (d(p_u, p_v) - \delta(p_u, p_v))^2$$

Kamada scelse di utilizzare un valore di  $k_1$  in modo tale che una molla tra due vertici con una piccola distanza teoretica fosse più forte. Più precisamente scelse  $k_1 = k / \delta(u,v)^2$  per una costante  $k$ . Di conseguenza l'energia tra  $u$  e  $v$ , risulta

$$\eta = \frac{k}{2} \left( \frac{d(p_u, p_v)}{\delta(u, v)} - 1 \right)^2$$

L'energia totale del sistema viene calcolata effettuando la somma di tale energia per ogni coppia di vertici  $u, v$  con  $u$  diverso da  $v$ . L'algoritmo cerca una posizione  $p_v$  per ogni vertice in modo da minimizzare tale energia. Il minimo si ottiene quando le derivate parziali di  $\eta$ , rispetto alle variabili  $x_v$  ed  $y_v$ , sono pari a zero. Ciò porta ad avere  $2|V|$  equazioni

$$\frac{\partial \eta}{\partial x_v} = 0, \quad \frac{\partial \eta}{\partial y_v} = 0, \quad v \in V.$$

che purtroppo non sono lineari. Può però essere utilizzato un metodo iterativo per risolverle, muovendo ad ogni passo un vertice in posizione tale da minimizzare l'energia, mentre gli altri vertici rimangono fissi. Come vertice da muovere viene scelto quello con la maggior forza esercitata su di esso, ossia tale per cui è massima

$$\sqrt{\left(\frac{\partial \eta}{\partial x_v}\right)^2 + \left(\frac{\partial \eta}{\partial y_v}\right)^2}$$

L'algoritmo di Kamada Kawai risulta essere più complesso da codificare, ma spesso fornisce risultati migliori. È infatti uno degli algoritmi basati sulle forze più utilizzato, assieme a quello di Fruchterman e Reingold.

Sono state studiate le differenze fra tali due algoritmi, notando che mentre l'algoritmo Kamada Kawai genera visualizzazioni focalizzate sulla distanza, quello di Fruchterman e Reingold focalizza l'attenzione sulla connettività. Ciò dev'essere preso in considerazione al momento della scelta dell'algoritmo da utilizzare.

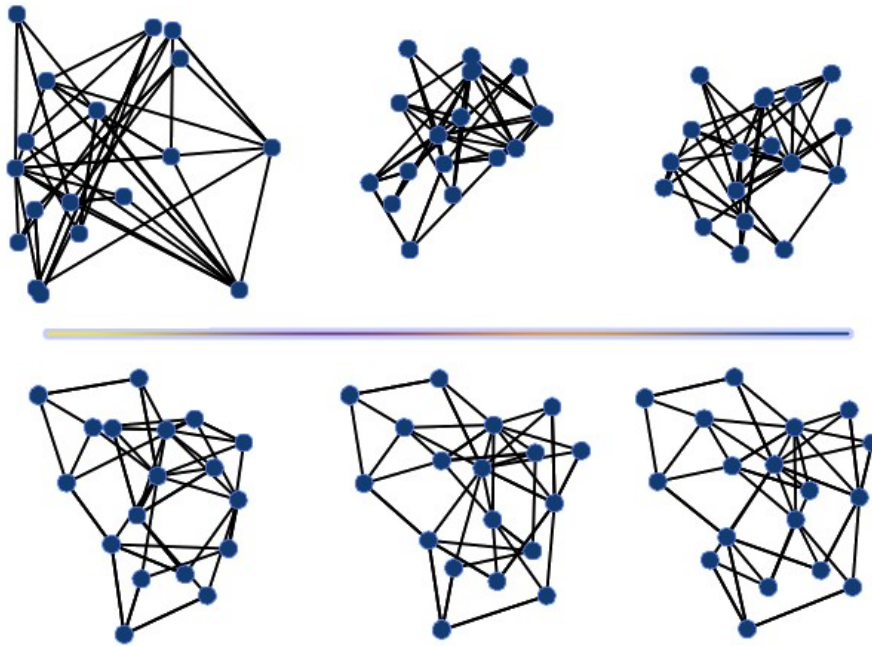


Fig. 23: Esempio di esecuzione dell'algoritmo di Kamada Kawai con stato di partenza generato da una *RandomView*. Il risultato migliora enormemente con iterazioni successive.

### ALGORITMI SU GRAFI PLANARI

Sono stati ideati degli algoritmi basati su grafi planari molto performanti e di piacevole risultato. Purtroppo però come abbiamo visto sono poche le occasioni in cui possono essere applicati. Si potrebbe pensare, dato un grafo, di verificare se sia o meno planare, andando poi ad utilizzare un algoritmo o un altro in base al risultato ottenuto. È possibile effettuare un test di planarità relativamente semplice basato sul concetto di Divide et Impera in tempo  $O(n^3)$ , oppure una versione molto più sofisticata nel tempo ottimale  $O(n)$  ideata da Hopcroft e Tarjan nel 1974 ([HT74]).

Un'altra soluzione potrebbe invece essere quella di rendere planare un grafo, semplicemente aggiungendo dei vertici fittizi per ogni intersezione – cercando comunque di minimizzarle – per poi applicare gli algoritmi relativi ai grafi planari. Tale approccio ha alcuni problemi, come ad esempio il fatto che la planarizzazione di un grafo potrebbe aggiungere  $O(n^4)$  nuovi vertici, facendo aumentare enormemente la complessità dell'algoritmo di visualizzazione, ma spesso è applicabile in quanto i grafi che portano ad una situazione di questo tipo sono piuttosto rari. Il problema della minimizzazione delle intersezioni è NP-completo, quindi l'unico modo per trovare una soluzione con numero di intersezioni accettabile in tempo polinomiale è quello di

utilizzare algoritmi approssimati.

#### ALGORITMO DI PLANARIZZAZIONE: INCREMENTAL PLANARIZATION

Un algoritmo piuttosto semplice per planarizzare un grafo è detto a planarizzazione incrementale, e consiste nel trovare il massimo sottografo planare di  $G$ , aggiungendogli poi lati andando a minimizzare il più possibile il numero di intersezioni con quelli già presenti. In particolare opera in tre passaggi:

1. computa il massimo sottografo planare di  $G$ , suddividendo i vertici tra “planari” e “non planari”. Per fare ciò parte da un grafo  $G'$  avente gli stessi vertici di  $G$  ma senza archi. Dopodiché valuta un arco alla volta, aggiungendolo e marcandolo come “planare” se  $G'$  dopo la sua aggiunta è ancora planare, rifiutandolo marcandolo come “non planare” se invece interseca un qualche lato.
2. Costruisce un'immersione planare di  $G'$  ed il relativo duale.
3. Aggiunge a  $G'$  i lati non planari, uno alla volta, ad ogni passo minimizzando il numero di intersezioni. Per effettuare questa operazione, aggiungendo un lato  $(u, v)$ , cerca nel duale di  $G'$  il percorso più breve tra una faccia incidente a  $u$  ed una faccia incidente a  $v$ , dopodiché aggiunge  $(u, v)$  a  $G'$  facendolo “passare” dalle facce appartenenti al percorso minimo. Il numero di intersezioni sarà pari a  $L-1$  dove  $L$  è la lunghezza del cammino minimo. Dopo ogni inserimento, va ricalcolato il duale di  $G'$ .

Questo algoritmo non si limita in realtà a trovare una planarizzazione del grafo iniziale, bensì ne realizza anche una visualizzazione con numero di intersezioni abbastanza piccolo. Può essere comunque usato come input per un qualunque algoritmo per grafi planari, ricordandosi di aggiungere un vertice per ogni intersezione ottenuta al passo 3. Dopo l'applicazione dell'algoritmo ovviamente questi vertici vengono tolti.

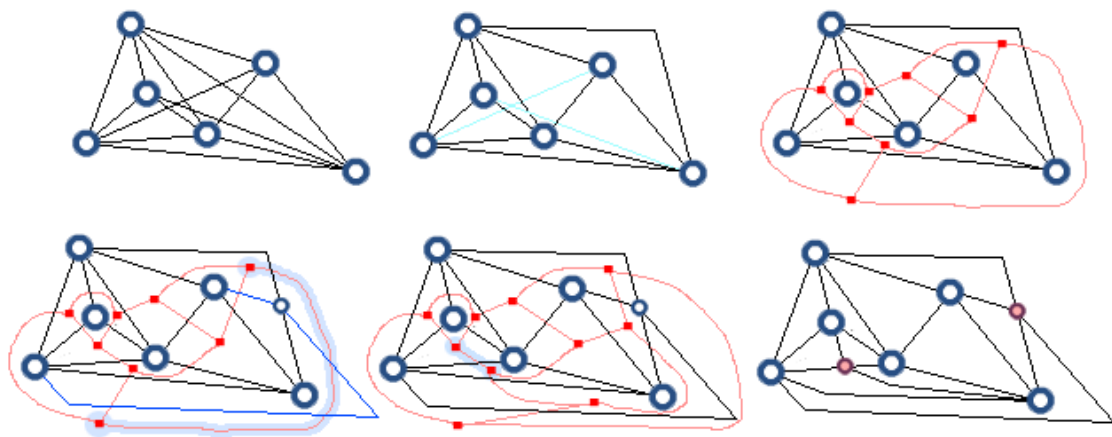


Fig. 24: Esempio di esecuzione dell'algoritmo di planarizzazione.

L'immagine rappresenta l'esecuzione di Incremental Planarization su un grafo di esempio: la prima immagine mostra lo stato iniziale, quella subito a destra il sottografo planare  $G'$ , con i lati segnati come “non planari” in azzurro (punto 1). Ancora a destra vediamo  $G'$  con il relativo duale in rosso, con i relativi vertici rappresentati da dei quadrati (punto 2). Nella riga successiva, le prime due immagini rappresentano l'aggiunta degli archi non planari di  $G$  (punto 3). Il percorso più breve da una faccia all'altra è evidenziato in azzurro e rappresenta il percorso che il nuovo arco dovrà fare per avere il minor numero di intersezioni. Nella prima delle due immagini sono stati inoltre indicati il nuovo arco ed il nuovo vertice fittizio, mentre nella seconda tali elementi sono stati omessi per rendere più leggibile il disegno. L'ultima figura rappresenta il grafo ultimato, con i nuovi vertici fittizi più piccoli, che andranno poi eliminati per lasciare il posto ad un'intersezione.

#### GRAFI PLANARI: RAPPRESENTAZIONE A VISIBILITÀ

Dato un grafo planare, si cerca di realizzare una rappresentazione a visibilità, nella quale ogni vertice è rappresentato da un segmento orizzontale ed ogni lato da un segmento verticale, in modo che non vi siano sovrapposizioni fra i segmenti e tale per cui un segmento verticale che fa riferimento ad un lato  $(u,v)$  abbia come punto iniziale e punto finale i segmenti di  $u$  e  $v$ .

Per costruire tale rappresentazione a visibilità, viene utilizzata la seguente procedura:

1. Viene costruito il duale planare  $G^*$  di  $G$ .
2. Vengono assegnati dei pesi unitari agli archi di  $G$  e viene computata una numerazione topologica pesata ottimale  $Y$  di  $G$ .
3. Vengono assegnati dei pesi unitari agli archi di  $G^*$  e viene computata una numerazione topologica pesata ottimale  $X$  di  $G^*$ .
4. Per ogni vertice  $v$  di  $G$ , viene tracciato un segmento orizzontale avente coordinata  $y$  pari a  $Y(v)$  e coordinate  $x$  pari a  $X(\text{left}(v))$  e  $X(\text{right}(v) - 1)$ .
5. Per ogni lato  $e$  di  $G$ , viene tracciato un segmento verticale avente coordinata  $x$  pari a  $X(\text{left}(e))$  e coordinate  $y$  date da  $Y(\text{orig}(e))$  e  $Y(\text{dest}(e))$ .

Un metodo, forse il più semplice, per costruire una numerazione topologica pesata ottimale di un grafo  $G$  consiste nell'assegnare ad ogni vertice un valore pari al numero di lati sul più lungo cammino diretto fino ad esso.

Una volta costruita la rappresentazione a visibilità del grafo  $G$ , la sua visualizzazione risulta essere banale.

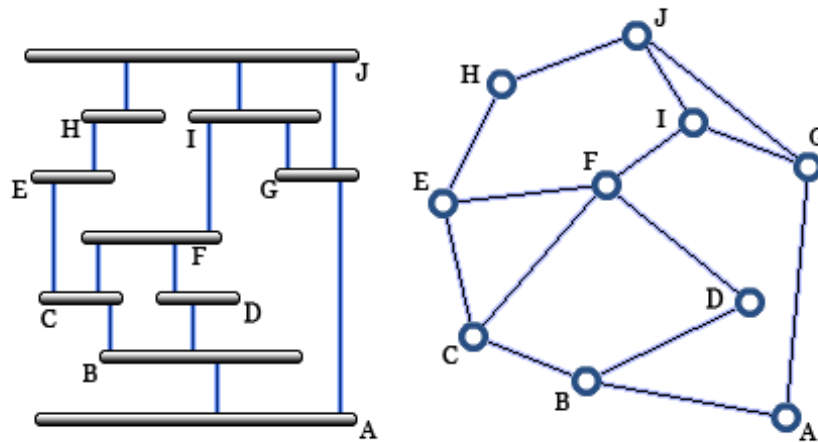


Fig. 25: Una rappresentazione a visibilità ed il relativo grafo planare.

### ALGORITMI INTERATTIVI

Quelli interattivi sono algoritmi particolari volti all'interazione con l'utente o un qualunque sistema esterno. Sono nati principalmente per permettere all'utente di modificare la posizione di vertici e lati durante l'operazione di disegno, proprio per questo motivo però sono ottimi per la visualizzazione di grafi temporali. Infatti il passaggio da un istante temporale all'altro può essere visto dal sistema come un insieme di modifiche inserite dall'utente. Inoltre tali algoritmi sono stati pensati proprio per rendere robusta la visualizzazione alle modifiche effettuate in fase di elaborazione, rendendoli di fatto ottimali per i grafi temporali.

Tali algoritmi operano sotto diversi tipi di scenari, che differiscono tra di loro per il grado di libertà dato all'utente e per performance:

*Scenario con pieno controllo:* l'utente ha il pieno controllo della posizione degli elementi, può quindi spostarli o ruotarli, inoltre può aggiungerne di nuovi o eliminarne alcuni. Il sistema non fa altro che accettare l'interazione, limitando eventualmente i movimenti per non superare una determinata area o per riposizionare in maniera più consona gli elementi. Le prestazioni sono ottime, in quanto il sistema non fa praticamente nulla.

*Scenario con ripartenza:* l'utente può effettuare modifiche sugli elementi aggiungendoli o togliendoli, ma senza poterne modificare la posizione. Questa infatti viene ricalcolata ogni volta che viene applicata una modifica, utilizzando un normale algoritmo di visualizzazione. Tale scenario, oltre a comportare un grande costo computazionale

dovuto all'esecuzione dell'algoritmo ad ogni modifica del grafo, comporta la distruzione della mappa mentale dell'utente, rendendo il grafo relativamente illeggibile.

*Scenario a coordinate relative:* come nello scenario con ripartenza, l'utente ha la possibilità di aggiungere o rimuovere elementi, ma in questo caso dopo una modifica non viene eseguito nuovamente l'algoritmo di visualizzazione, bensì vengono effettuate le modifiche preservando il più possibile la visualizzazione corrente. Gli elementi già esistenti possono subire lievi modifiche per rendere la rappresentazione migliore.

*Scenario senza cambiamenti:* in questo approccio le coordinate degli elementi già posizionati non vengono affatto cambiate quando l'utente apporta delle modifiche. L'immagine mentale dell'utente viene preservata totalmente.

Per quanto riguarda i grafi temporali, solamente gli ultimi due scenari risultano accettabili, in quanto il primo risulterebbe inutile per la visualizzazione di tali grafi, mentre il secondo renderebbe incomprensibile ogni animazione temporale.

#### UN APPROCCIO AGLI ALGORITMI INTERATTIVI

Come abbiamo visto, nello scenario a coordinate relative l'utente può interagire con il sistema aggiungendo od eliminando elementi dal grafo. Quando una di queste operazioni avviene, il sistema deve reagire di conseguenza, effettuando la modifica ed aggiustando gli elementi già esistenti in modo limitato.

Nel caso in cui venga eliminato un elemento non vi è alcun problema, in quanto la visualizzazione non può che migliorare. Nel caso invece di un'aggiunta bisogna effettuare due passaggi:

- L'algoritmo deve decidere dove posizionare il nuovo elemento, facendo sì che la rappresentazione aggiornata risulti il più chiara possibile. Per fare ciò ci si permette anche di fare una scelta non ottima sulla posizione, proprio perché viene data priorità alla visualizzazione corrente.
- Si modifica lievemente la posizione degli elementi esistenti, aggiustando così la visualizzazione, che altrimenti diventerebbe caotica dopo poche aggiunte.



Fig. 26: Un esempio di comportamento di grafo interattivo: a partire dal primo grafo ed aggiungendo il lato (1,3), un algoritmo di visualizzazione classico disegnerebbe il triangolo in seconda figura. Un algoritmo interattivo invece, per preservare la visualizzazione, realizzerebbe la terza.

Per il primo punto, la ricerca della posizione dipende dall'elemento inserito. Una buona idea potrebbe essere quella di realizzare il duale  $G^*$  del grafo  $G$ , opportunamente planarizzato qualora non lo fosse. Per inserire un nuovo vertice  $v$ , si cerca in  $G^*$  la faccia che si trova a minor distanza fra le facce che circondano i vicini di  $v$ . In questo caso si minimizza il numero di intersezioni fra i lati esistenti ed i lati aggiunti con l'inserimento di  $v$ . In modo simile, aggiungendo un lato  $(u,v)$  al grafo  $G$ , si cerca il percorso più breve in  $G^*$  tra una faccia attorno ad  $u$  ed un'altra faccia attorno a  $v$ .

Per quanto riguarda il secondo punto, un algoritmo basato sulle forze potrebbe ridisporre gli elementi attorno al grafo in maniera ordinata, allontanando lati eccessivamente vicini ed avvicinando nodi collegati da archi eccessivamente lunghi. In una situazione in cui le modifiche apportate non sono molte, un algoritmo di questo tipo porterebbe velocemente ad una situazione di equilibrio.

In uno scenario senza cambiamenti, il sistema non modifica mai la posizione degli elementi già esistenti, andando solamente ad effettuare le modifiche apportate dall'utente. Per fare ciò basta eseguire un algoritmo simile a quello adottato nello scenario a coordinate relative, senza però andare ad effettuare l'aggiustamento degli elementi già esistenti. In questo caso però la scelta del posizionamento assume un ruolo di primaria importanza, in quanto non vi è una successiva fase di rifinitura del risultato.

Esempi di implementazione di tali algoritmi, in grado di realizzare visualizzazioni ortogonali di grafi con vertici di grado massimo pari a 4 si possono trovare in [BETT99].

## CONCLUSIONI

Gran parte degli algoritmi presentati in questo documento sono stati implementati in GraphVisualizer, un software web basato su tecnologia Flex 3. Tale software permette di caricare grafi temporali memorizzati a slice su più file in formato Dot. Una volta elaborato il grafo, ne visualizza alcune informazioni base quali il numero di vertici, di archi e di componenti connesse.

Il sistema permette di scegliere l'algoritmo da applicare al grafo caricato, permettendo inoltre di impostare i parametri peculiari dell'algoritmo scelto e dando la possibilità quindi di effettuare varie prove, fornendo un comodo metodo di valutazione e studio degli algoritmi implementati. Sono supportati algoritmi iterativi e non, grazie all'implementazione degli stessi tramite il metodo di ritorno di funzione spiegato in questo documento.

Il grafo viene visualizzato richiamando l'algoritmo scelto su ogni sua componente connessa in maniera separata. Le componenti vengono poi unite in una visualizzazione unica tramite l'algoritmo dell'affiancamento. Durante il cambio di tempo, l'algoritmo di affiancamento viene richiamato nuovamente, causando un possibile movimento delle componenti. Ciò può causare disturbo nella visualizzazione, anche se non si tratta di nodi singoli ma di blocchi di nodi. Si può pensare di risolvere il problema non richiamando l'algoritmo, ma semplicemente riposizionando le componenti connesse in funzione della variazione di dimensioni delle altre.

Per poter facilitare la visualizzazione è stato implementato un sistema che permette la navigazione e lo zoom del grafo in maniera comoda ed intuitiva, grazie alla possibilità di impostare l'ingrandimento a dimensioni reali o a dimensioni di schermo intero, permettendo di spostare la finestra corrente lungo il grafo e così via. Ciò permette di avere visualizzazioni anche complesse, riuscendo comunque ad interagire e raccogliere informazioni da esse.

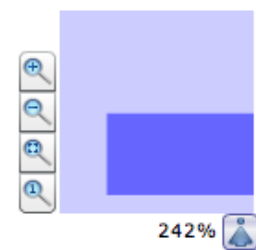


Fig. 27: Il tool di ingrandimento e navigazione.

Per ottenere informazioni sugli elementi del grafo, vertici e lati, è presente un box informativo a scomparsa, che visualizza le informazioni dell'elemento selezionato. I dati visualizzati sono generalmente il nome e le caratteristiche base, ma possono essere impostati in fase di definizione di nuovi elementi, ereditando e modificando un metodo della classe generica.

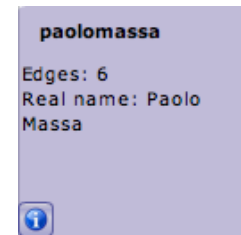


Fig. 28: Il box informativo

Il codice è scritto infatti in modo generale, permettendo ad un programmatore ActionScript di realizzare nuovi tipi di vertici o lati ereditando da quelli già esistenti. Tali elementi possono poi essere utilizzati specificandoli nel file di memorizzazione del grafo. Lo stesso vale per la realizzazione di parser per nuovi tipi di dati o per la scrittura di nuovi algoritmi, per i quali si può utilizzare una versione generica già realizzata o degli esempi di studio.

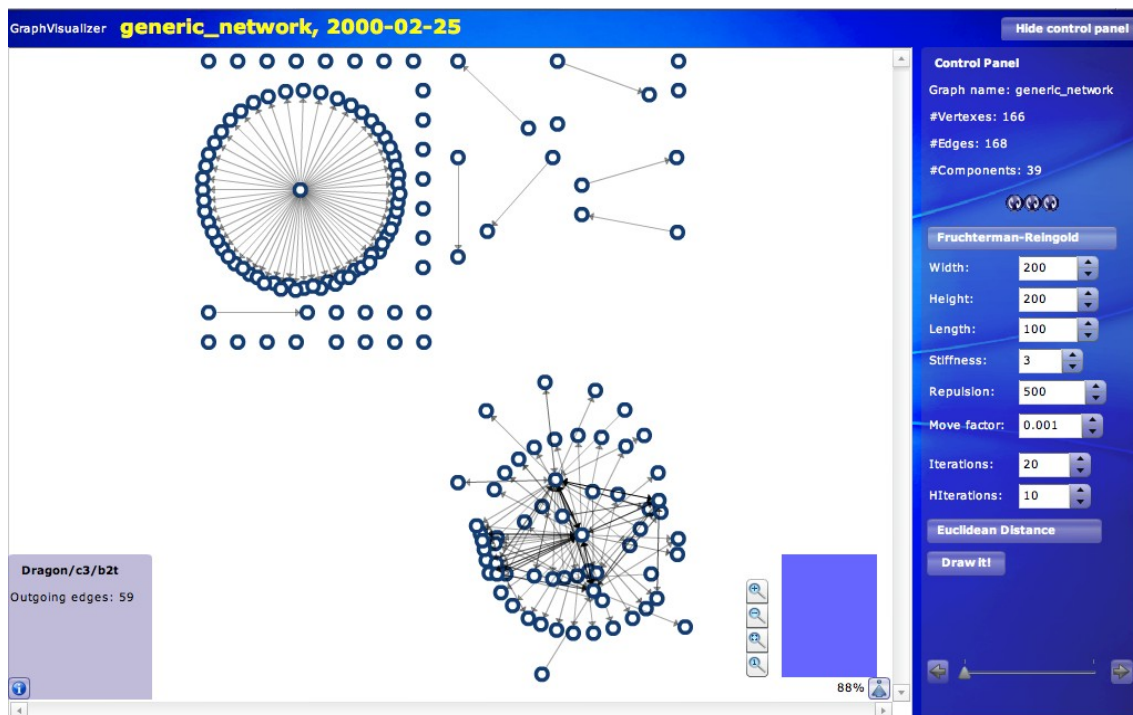


Fig. 29: Screenshot dell'applicazione; il grafo visualizzato rappresenta l'interazione fra utenti di un MediaWiki in fase di testing, l'algoritmo utilizzato è quello di Fruchterman e Reingold basato sulle forze, sono visibili i parametri di visualizzazione ed il risultato dell'applicazione dell'algoritmo di affiancamento.

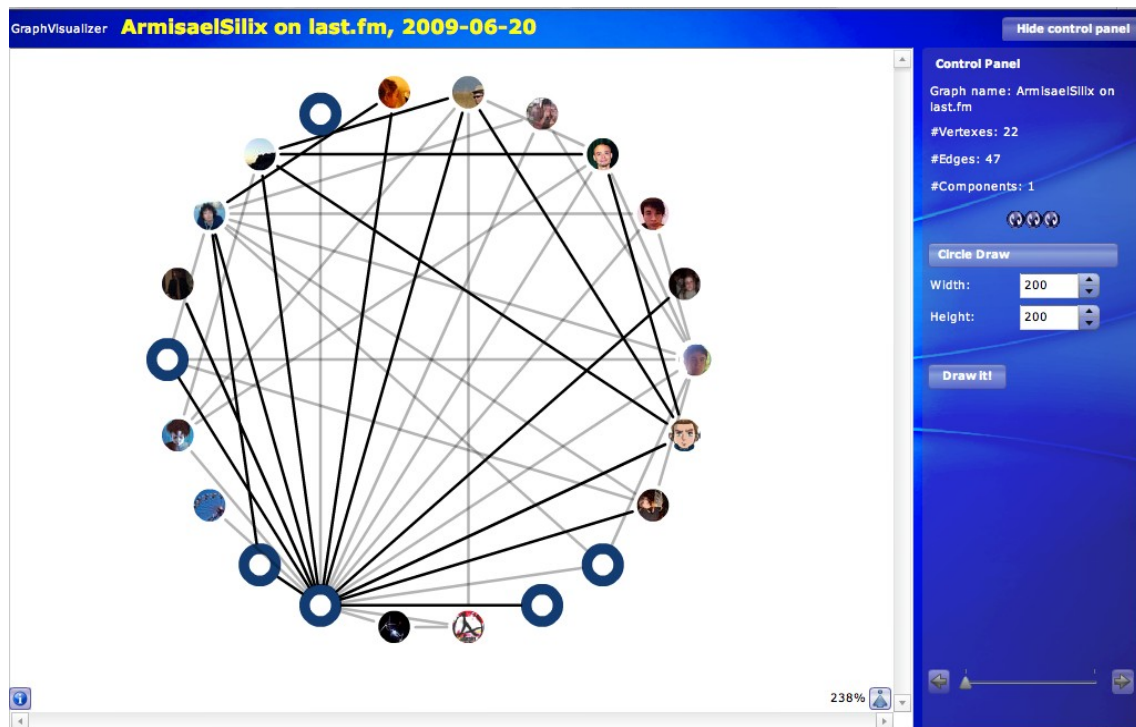


Fig. 30: Un esempio di utilizzo di vertici ed archi non generici. In questo caso i vertici visualizzano la foto dell'utente sul social network di Last.fm, mentre l'arco indica l'affinità musicale fra due amici.

## BIBLIOGRAFIA

- [BETT99] G. Battista, P. Eades, R. Tamassia e I. G. Tollis, *Graph Drawing: algorithms for the visualization of graphs*, 1999
- [BM76] J. A. Bondy e U. S. R. Murty, *Graph Theory with applications*, 1976
- [CLR90] Thomas H. Cormen, Charles E. Leiserson e Ronald L. Rivest, *Introduction to Algorithms*, 1990
- [DM06] Skye Bender-deMoll, Daniel A. MvFarland, *The art and Science of Dynamic Network Visualization*, 2006
- [Ead84] P. Eades, *A Heuristic for Graph Drawing*, 1984
- [EG96] P. Eades e P. Garvan, *Drawing stressed planar graphs in three dimensions*, 1996
- [FR91] T. Fruchterman, E. Reingold, *Graph drawing by Force-directed Placement*, 1991
- [Free00] Linton C. Freeman, *Visualizing Social Networks*, 2000
- [GZ05] Peter A. Gloor, Yan Zhao, *Visualizing time in social networks with TeCFlow*, 2005
- [HT74] J. Hopcroft e R. E. Tarjan, *Efficient planarity testing*, 1974
- [KMS94] C. Kosak, J. Marks e S. Shieber, *Automating the layout of Network Diagrams with Specified Visual Organization*, 1994
- [LS38] G. A. Lundberg, M. Steele, *Social attraction-patterns in a village*, 1938
- [Mor32] Jacob L. Moreno, *Application of the Group Method to Classification*, 1932
- [Mor53] Jacob L. Moreno, *Who Shall Survive?*, 1953

- [Tut60] W. T. Tutte, *Convex representation of graphs*, 1960
- [Tut63] W. T. Tutte, *How to draw a graph*,
- [WF94] S. Wasserman, K. Faust, *Social Network Analysis: Methods and Applications*, 1994

## SITOGRAFIA

Joss – Journal of Social Structure, <http://www.cmu.edu/joss/>  
*Il giornale elettronico dell'INSNA, "International Network for Social Network Analysis" per la diffusione dello stato dell'arte nella ricerca interdisciplinare sulle strutture sociali.*

SoNIA – Social Network Image Animator, <http://www.stanford.edu/group/sonia/>  
*Un software Java per la visualizzazione di sociogrammi temporali.*

Advogato, <http://www.advogato.org/>  
*Un sistema di sviluppo software opensource collaborativo.*

Last.fm, <http://www.lastfm.it/>  
*Un social network basato sulla musica in grado di capire i gusti degli utenti e proporre loro musica che non conoscono ma che sulla base di studi potrebbe essere di loro gradimento.*

TrustLet, <http://www.trustlet.org/> e <http://www.trustlet.org/datasets/advogato/>  
*Un ambiente collaborativo per la ricerca di metriche di fiducia su reti sociali e il relativo dataset sulla rete di Advogato.*

Ucinet IV Dataset, <http://vlado.fmf.uni-lj.si/pub/networks/data/Ucinet/UciData.htm>  
*Un insieme di dataset riguardanti interazioni sociali.*

## **RINGRAZIAMENTI**

I miei ringraziamenti vanno al Professor Alberto Montresor, al Dottor Paolo Massa e al Dottor Maurizio Napolitano, per avermi saputo guidare e consigliare nella stesura di questo documento. Inoltre ringrazio sentitamente Fabrizia per avermi supportato e guidato durante tutto il periodo di stesura. I miei ringraziamenti vanno anche ai ragazzi partecipanti a WebValley 2009 ed in particolare ai miei compagni di stanza, per avermi supportato tre settimane ed avermi distratto dalla frenesia del momento. Infine ringrazio la Fondazione Bruno Kessler, in particolare tutto il gruppo SoNet e Mpba, per avermi reso ciò che sono.