UNIVERSITÀ DEGLI STUDI DI CAMERINO

FACOLTÀ DI SCIENZE E TECNOLOGIE

CORSO DI LAUREA IN MATEMATICA

DIPARTIMENTO DI MATEMATICA E INFORMATICA

TEORIA COMPUTAZIONALE DEI NODI

CODIFICA E RAPPRESENTAZIONE GRAFICA

Tesi Sperimentale di Laurea in Geometria

Relatore

Prof. Riccardo Piergallini

Laureando

Mauro Rinaldelli

Anno Accademico 2003-2004

Alla mia famiglia,

presente e futura...

Ringraziamenti

Per la realizzazione di questa tesi un ringraziamento particolare va al mio relatore Prof. Riccardo Piergallini per la disponibilità e l'aiuto che mi ha dato.

Desidero inoltre ringraziare enormemente i miei due fratelli Bruno e Giampaolo, i miei genitori e zia Claudia per il loro altrettanto grande supporto, sia morale che pratico, sotto forma di preziosi consigli, paziente rilettura ed aiuto nelle traduzioni. Lo stesso ringraziamento va alla mia ragazza, Giada, alla quale va anche tutto il mio amore. Grazie al resto della famiglia tutta, zii, cugini ed in particolare alle mie nonne, Lina e Mimì, per il loro amorevole sostegno a distanza. Un ringraziamento speciale per essermi stati sempre vicini ai miei amici più "vecchi" Andrea ed Alessandra, fratelli mancati ma solo tecnicamente, e alle mie due nuove amiche, Helga ed Emilia, per il loro incoraggiamento.

Con questa mia ultima fatica universitaria si chiude formalmente un capitolo importante del mio cammino. In questo lungo periodo tante sono state le persone con le quali ho condiviso diversi momenti e che vorrei ringraziare. Un riferimento fisso sono stati e sono i miei amici di sempre Claudio "Pelò", Graziano, Giacomo & Michela, Paolo "Ponzio" & Lucia, Emanuele, Vittorio, Adolfo, Lamberto "Mayel" & Federica, Oliviero, Lorenzo & Gessica, Sergio "Fra Filì", Marco, Gianluca "il Prof.", e poi ancora Nicolò, Jacopo, Francesca & Laura, Laura e Daniele "Passò".

Un ringraziamento particolare al "Padre Nostro", apparentemente dietro le quinte ma sempre ben presente, e a quanti hanno pregato per me. Il merito di questa tesi è anche vostro.

Grazie anche a tutti i miei compagni della pallavolo, sia attuali che passati, per avermi "distratto" dall'Università. Grazie a tutti.

Concludo ringraziando il Prof. Luciano Misici per alcuni suoi consigli e per la sua disponibilità, in questi anni, in particolare per avermi "sbrogliato" alcune pratiche burocratiche che mi avrebbero ingiustamente costretto a partire militare.

Indice

1 Introduzione **1**

1.1 Disegno topologico al computer 3

1.2 Contributo della tesi 5

1.3 Panoramica sulla tesi 6

2 Teoria dei nodi **9**

2.1 Introduzione e definizioni 10

2.2 Notazioni 20

2.2.1 I tangle di Conway 21

2.2.2 I codici di Gauss e di Dowker-Thistlethwaite 25

2.3 Invarianti topologici dei nodi 28

2.3.1 Linking number 29

2.3.2 Crossing Number 31

2.3.3 Stick number 31

2.3.4 Invarianti polinomiali 33

3 La teoria dei nodi al computer **43**

3.1 Un programma ideale 43

3.1.1 Caratteristiche tecniche 44
3.1.2 Caratteristiche grafiche 45
3.2 Caratterizzazione dei programmi di grafica topologica 45
3.2.1 Geomview 46
3.2.2 JavaView 48
3.2.3 KnotPlot 51
3.2.4 SnapPea 52
3.2.5 Knotscape 52

4 Il mio programma: KnotExplorer 55
4.1 Scelte preliminari 56
4.2 Codifica 56
4.2.1 Codifica Interna 57
4.2.2 Codifica Esterna 64
4.3 Immissione dei dati 69
4.4 Conversione dalla codifica esterna alla codifica interna 70
4.5 La decodifica: rappresentazione grafica iniziale del diagramma . 75
4.6 Energia dei nodi 81

5 Conclusioni e sviluppi futuri 91

A Il codice del KnotExplorer 95
A.1 La classe Arco 95
A.2 La classe Componente 97
A.3 La classe Connessione 98
A.4 La classe Controls 103

A.5 La classe Diagramma 106
A.6 La classe Direzione 108
A.7 La classe Energia 110
A.8 La classe Grafico 110
A.9 La classe Immissione 128
A.10 La classe Incrocio 138
A.11 La classe KnotExplorer 143
A.12 La classe ListaArchi 147
A.13 La classe ListaComponenti 148
A.14 La classe ListaConnessioni 149
A.15 La classe ListaIncroci 149
A.16 La classe ListaPonti 150
A.17 La classe ListaPunti 151
A.18 La classe ListaPuntiOrdinata 153
A.19 La classe ListaVertici 155
A.20 La classe Mobius 156
A.21 La classe Ottimizzazione 169
A.22 La classe Piano 170
A.23 La classe Ponte 171
A.24 La classe Punto 171
A.25 La classe RunControls 173
A.26 La classe Vertice 174

Bibliografia **178**

Capitolo 1

Introduzione

Una questione fondamentale nella matematica, affrontata anche da molti punti di vista, è sempre stata quella di determinare quando due oggetti devono essere considerati equivalenti. Un ramo della matematica che abbonda di interrogativi di questo tipo è la topologia, dove molte forme di equivalenza sono basate su idee pseudo-geometriche, come l'adiacenza o la concatenazione. Due oggetti possono avere la stessa "forma", intesa in senso topologico, anche se a prima vista possono sembrare completamente differenti. Ad esempio, consideriamo gli oggetti in figura 1.1, dove le due curve rappresentano due corde chiuse nello

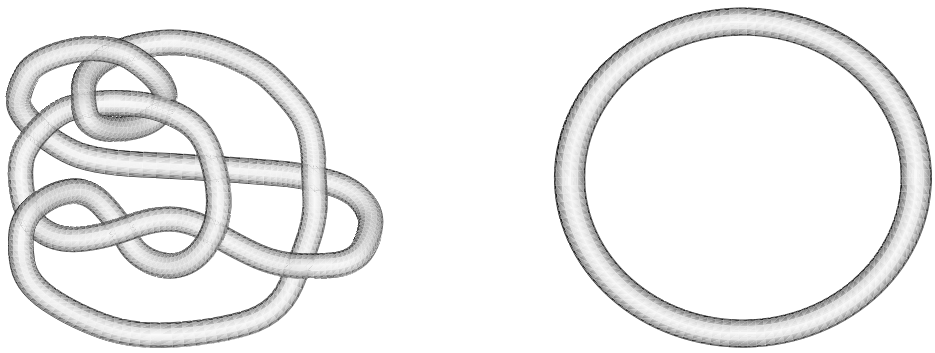


Figura 1.1: Due oggetti aventi la stessa forma.

spazio tridimensionale. Le due figure sono equivalenti? Qui, l'equivalenza è intesa in maniera particolare: due curve di questo tipo si dicono equivalenti se e solo se una può essere deformata con continuità fino a farla coincidere con l'altra, senza tagli, auto-intersezioni o punti singolari. Anche questo caso apparentemente semplice, costituito da due curve chiuse nello spazio, si è dimostrato un problema piuttosto difficile da risolvere; nel secolo scorso, la Teoria dei Nodi ha sviluppato una serie di tecniche che hanno dato una notevole svolta a tali problemi di equivalenza. La questione è stata risolta teoricamente, poiché si è dimostrato che il problema è determinabile [Mat03], invece, nella pratica, la soluzione risulta difficilmente applicabile (soprattutto in senso computazionale).

Supponiamo che le due figure siano equivalenti nel senso descritto sopra (ed è così), come possiamo dimostrare la loro equivalenza, mostrando una deformazione che porta una curva sull'altra? Nella teoria dei nodi questo è un problema computazionalmente difficile, che diventa ancora più complesso se facciamo una piccola modifica alla prima figura: proviamo ad esempio ad invertire un incrocio (figura 1.2). Anche se la modifica apportata è apparentemente

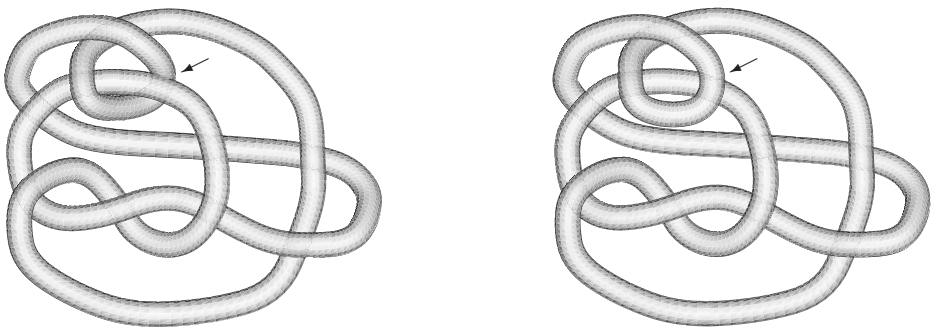


Figura 1.2: Due oggetti aventi forma diversa.

piccola, essa risulta determinante in termini di equivalenza, infatti, ora non è più possibile trovare una trasformazione continua dalla corda "intrecciata" alla circonferenza. Questo dimostra la complessità del problema globale.

1.1 Disegno topologico al computer

Il discorso precedente suggerisce di utilizzare l'aiuto del computer per trattare i nodi, soprattutto attraverso un buon programma che consenta di risolvere le difficoltà computazionali del problema.

Negli ultimi anni sono stati sviluppati diversi software interattivi in grado di generare e visualizzare figure geometriche, sia in ambiente 2D, che 3D e persino 4D. Gran parte di questi lavori sono usciti dal *Geometry Center*, fondato dalla "U.S. National Science Foundation", associato all'Università del Minnesota, negli Stati Uniti. Fortunatamente per i ricercatori, molti di questi software sono gratuiti e spesso ne è reso disponibile il codice sorgente.

Un primo tentativo di produrre un programma degno di nota è stato intrapreso da David Dobkin e il suo gruppo, presso l'Università di Princeton nel New Jersey. Il loro lavoro ha dato origine al *Salem* [DNT90, DNT91], un software per visualizzare oggetti matematici in 3D. Tutti questi oggetti sono costruiti come insiemi di punti (vertici) e di poligoni. Da queste "primitive", il Salem è in grado di costruire oggetti di alto livello, come poliedri, tori con un numero arbitrario di buchi, involucri convessi per rivestire un insieme di punti, e nodi torici. Al Salem è stata associata un'interfaccia per lavorare nel sistema operativo UNIX, che consente di creare oggetti di forma arbitraria. Il programma fornisce diversi comandi per gestire l'oggetto rappresentato ed in

particolare i cambiamenti di vista (le trasformazioni di rotazione e traslazione), alcuni dei quali sono gestibili direttamente con il mouse. Forse, l'aspetto più interessante del Salem è la sua capacità di navigare attraverso lo spazio ellittico o in quello iperbolico.

Nel 1991 al Geometry Center si è incominciato lo sviluppo di un nuovo software, il *GeomView* [PLM93], che in qualche modo può essere considerato il successore del Salem. Come il predecessore, infatti, può visualizzare oggetti sia nello spazio iperbolico che in quello sferico; inoltre ne riproduce quasi tutte le caratteristiche, introducendone di nuove.

Sempre al Geometry Center, circa due anni dopo, è nato un nuovo progetto da un'idea del matematico Kenneth Brakke, suo autore. Si tratta del *Surface Evolver* [Bra92, Bra03], che ha come direzione principale lo studio di superfici formate da tensioni superficiali ed altre energie, il tutto soggetto ad ulteriori costrizioni. Il Surface Evolver è in grado di triangolarizzare[1] le superfici, inoltre la sua peculiarità principale è quella di "evolvere" le superfici di partenza fino a far assumere loro la configurazione che minimizza le energie coinvolte. La capacità di triangolarizzare le superfici consente al programma di manipolare qualsiasi tipo di oggetti topologici. Il Surface Evolver è stato scritto principalmente per lavorare con una o due dimensioni, tuttavia è in grado utilizzare anche dimensioni superiori, con alcune restrizioni sulle sue capacità disponibili. Il programma ha ottenuto molto successo, tanto da venir utilizzato persino per la realizzazione di alcuni video.

Sia il Geomview che il Surface Evolver sono gratuitamente disponibili e

[1] La triangolazione è un processo secondo cui una superficie viene scomposta nell'unione di superfici più piccole di forma triangolare. E' un'operazione sempre possibile.

sono utilizzati da parte di molti ricercatori.

Questi software hanno posto le basi per lo sviluppo di programmi capaci di combinare grafica computerizzata e calcolo numerico, improntati allo studio della topologia e delle proprietà topologiche degli oggetti. Nell'ultimo decennio sono stati realizzati numerosi software anche per essere applicati alla teoria dei nodi, come ad esempio lo SnapPea, il KnotPlot o il Knotscape (dei quali parleremo nel capitolo 3), ognuno con le proprie possibilità e i propri limiti, anche se "il programma" per analizzare i nodi, con la "P" maiuscola, non è stato ancora sviluppato.

1.2 Contributo della tesi

Questa tesi di ricerca ha come obiettivo la realizzazione di un software che rappresenti graficamente i nodi e ne agevoli la comprensione e lo studio.

Abbiamo voluto creare qualcosa di nuovo all'interno di un software che lavori in 2D, quindi che rappresenti i nodi attraverso le loro proiezioni su un piano, e le novità introdotte riguardano in particolare:

1. il sistema di codifica;
2. il calcolo dell'energia;
3. la possibilità di effettuare i movimenti di Reidemeister in modo interattivo.

Nel primo punto abbiamo introdotto due nuovi tipi di codifica, l'uno è stato pensato per l'interfaccia con l'utente, in modo da rendergli semplice la codifica

specifica di un determinato nodo e l'inserimento dei dati; l'altro invece è interno al programma e risulta più complesso del tipo precedente perchè necessita di un maggior numero di informazioni che consentono di creare in modo più semplice procedure più efficienti. Anche quest'ultimo tipo di codifica può essere usato per l'inserimento dei dati.

Nel secondo punto (quello del calcolo dell'energia) abbiamo composto e minimizzato un funzionale di energia, avente come termine principale l'energia di Möbius, discretizzata da Denise Kim e Rob Kusner in [KK93]. Dato che l'energia di Möbius è stato creato per essere utilizzato nelle tre dimensioni, abbiamo dovuto provvedere ad adattarlo e migliorarlo con altri termini.

Il terzo punto non è stato ancora sviluppato, potrà quindi essere preso in esame per prossimi approfondimenti.

1.3 Panoramica sulla tesi

Nel capitolo 2 introdurremo le basi della teoria dei nodi, necessarie per comprendere le questioni del lavoro svolto nella tesi. Daremo le definizioni principali e ci interesseremo dei problemi relativi all'equivalenza tra nodi e alla loro classificazione, trattando le notazioni e gli invarianti più interessanti.

Nel capitolo 3 accosteremo nodi e computer con l'intento di utilizzare il secondo per studiare i primi. Analizzeremo le proprietà necessarie ad un buon programma affinchè sia adatto a farci acquisire una migliore conoscenza dei nodi e della loro teoria, poi prenderemo in esame i programmi già esistenti e ne valuteremo le qualità e i limiti.

Il capitolo 4 costituisce la parte principale della tesi e contiene gli studi

fatti per la realizzazione del nostro programma. Presenteremo inizialmente le due codifiche adottate, con le rispettive caratteristiche, e dimostreremo due teoremi per verificare la loro coerenza. Mostreremo come è possibile passare da una codifica ad un altra, ed in seguito vedremo come il programma esegue l'operazione inversa, ossia la decodifica, fino a dare una prima rappresentazione del diagramma del nodo. Infine analizzeremo i termini dell'energia utilizzata dal programma per trovare una configurazione ottimale del diagramma.

Nel capitolo 5 porremo le conclusioni del lavoro svolto e daremo dei suggerimenti per possibili sviluppi futuri.

Capitolo 2

Teoria dei nodi

La teoria dei nodi è un ramo della topologia che studia le immersioni di uno spazio topologico dentro un altro. Diciamo che una trasformazione dello spazio X è una ***immersione*** nello spazio Y, se il risultato di tale trasformazione è una deformazione continua del primo spazio in un sottoinsieme del secondo spazio, con inversa anch'essa continua.

L'aspetto più semplice – non banale – di questa teoria si occupa del modo in cui si possono immergere una o più circonferenze disgiunte nello spazio a tre dimensioni.

Nelle pagine che seguono scopriremo che cosa si intende in matematica per "nodo", che cos'è un "diagramma" di un nodo e che cosa vuol dire che due nodi sono "equivalenti" o "isotopi". Inoltre, ci addentreremo nei due problemi di maggior interesse affrontati in questa teoria:

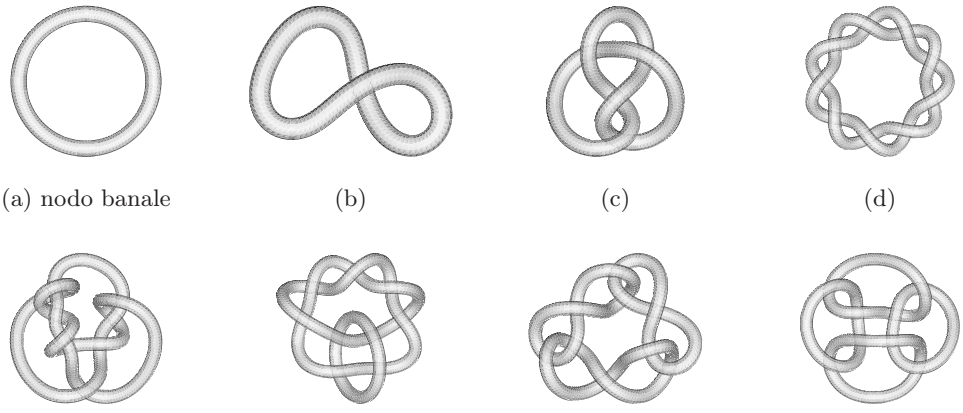
1. determinare se due nodi assegnati siano equivalenti;

2. classificare i nodi.

Svolgeremo questi argomenti seguendo prevalentemente le trattazioni intuitive usate in [Sch98] e [Kos88].

2.1 Introduzione e definizioni

Ai fini di questa tesi, definiamo il ***nodo*** come l'immersione di *una* circonferenza nello spazio Euclideo tridimensionale $\mathbb{R}^3$. Vogliamo far notare che in matematica un nodo è qualcosa di diverso dall'idea usuale, che ci fa pensare ad un pezzo di corda intrecciata con le estremità libere: i nodi studiati in questa teoria hanno (quasi) sempre le estremità chiuse, come in un cappio. Possiamo far riferimento alle figure 2.1 e 2.2 per renderci meglio conto di cosa stiamo parlando. Osservando accuratamente i nodi raffigurati nella figura 2.1 possiamo



(a) nodo banale (b) (c) (d)

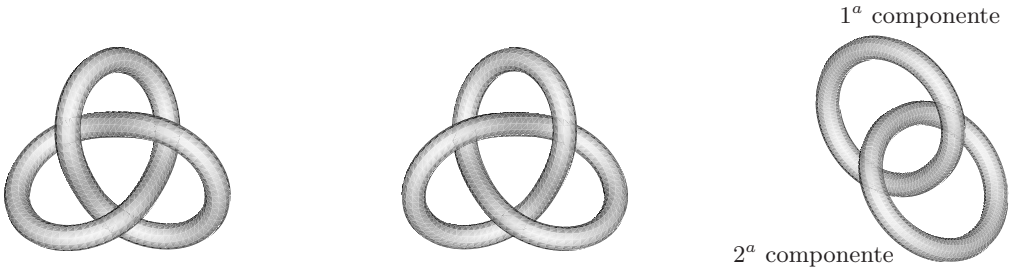
(e) (f) (g) (h)

Figura 2.1: Alcuni nodi.

vedere che non si può ottenere il nodo (c) dal nodo (a) senza tagliare la corda: ciò accade perchè il nodo (c) è "intrecciato" mentre (a) non lo è. Un nodo viene detto "non intrecciato" o "sciolto" se esiste un movimento continuo nello

spazio $\mathbb{R}^3$ che lo faccia coincidere col nodo (a) della figura 2.1, il quale viene solitamente chiamato ***nodo banale***. Formalmente, diremo che un nodo K è ***intrecciato*** se non esiste un omeomorfismo[1] $h : \mathbb{R}^3 \to \mathbb{R}^3$ tale che $K = h(S^1)$, dove $S^1 = \{(x, y, 0) \in \mathbb{R}^3 \mid x^2 + y^2 = 1\}$ è la circonferenza standard nel piano xy. Pertanto i nodi (a) e (b) della figura 2.1 non sono intrecciati, mentre tutti gli altri lo sono.

Un intreccio di due o più nodi è chiamato ***link***, e ciascuno dei nodi che costituiscono il link viene chiamato ***componente*** del link. Per semplificare la terminologia della tesi ci riferiremo spesso ai link chiamandoli comunemente nodi, mentre per riferirci specificatamente ad un nodo costituito da una sola componente useremo il termine "nodo proprio", seguendo [Conway70]. In figura 2.2 possiamo vedere il nodo a trifoglio, sia destro che sinistro, ed anche il più semplice dei link: il link di Hopf. Osservando i primi due, si intuisce una



(a) trifoglio sinistro (b) trifoglio destro (c) link di Hopf

Figura 2.2: Il nodo più semplice, nelle sue due forme, e il più semplice dei link.

certa "parentela": questi nodi sono uno l'immagine speculare dell'altro, riflessa rispetto ad un piano verticale. A parte questa proprietà, non c'è modo di far

[1] Un ***omeomorfismo*** è un'applicazione continua e biunivoca (cioè uno-a-uno) con inversa continua, tra due spazi topologici.

coincidere l'uno sull'altro con un movimento continuo nello spazio, quindi le due figure non rappresentano lo stesso nodo. Secondo la terminologia utilizzata nella teoria dei nodi possiamo comunque dire che il trifoglio destro e il trifoglio sinistro sono equivalenti. Infatti, secondo la definizione, due nodi (o link) si dicono ***equivalenti*** se esiste un omeomorfismo su $\mathbb{R}^3$ che trasforma un nodo nell'altro. Possiamo finalmente affermare che anche i nodi (a) e (b) della figura 2.1 sono equivalenti, così come lo sono i due nodi della figura 1.1 a pagina 1 (anche se ad occhio è più difficile da stabilire).

La riflessione rispetto ad un piano, necessaria per trasformare il trifoglio destro nel trifoglio sinistro (o viceversa) è un'operazione che "non conserva l'orientazione", ossia trasforma un "triedro destro" in un "triedro sinistro" (vedere figura 2.3) e non può essere realizzata con un movimento continuo

Figura 2.3: I due triedri, destro e sinistro.

nello spazio $\mathbb{R}^3$. Un omeomorfismo, però, non è necessariamente una trasformazione che può essere ottenuta con un movimento continuo dello spazio a partire dall'identità, quindi non è detto che conservi l'orientazione. Si dice che un omeomorfismo $h : \mathbb{R}^3 \to \mathbb{R}^3$ conserva l'***orientazione*** se h trasforma un triedro destro in un altro triedro destro[2].

[2]Questa definizione non è rigorosa, infatti non è detto che un omeomorfismo di $\mathbb{R}^3$ trasformi un triedro in un altro triedro. La nozione ha senso per un omeomorfismo lineare e può essere estesa

Il fatto che l'equivalenza sia definita sulla base degli omeomorfismi fa sì che un nodo e la sua immagine speculare siano considerati equivalenti. Esiste invece un tipo di equivalenza che fa distinzione anche tra le immagini speculari: l'*isotopia*.

Siano X e Y due spazi topologici, diciamo che la funzione H è un'*omotopia di X in Y* se è tale che $H : X \times [0,1] \to Y$ ed è continua. In pratica, l'omotopia è una famiglia "continua" di funzioni continue $(h_t : X \to Y)_{t\in[0,1]}$, definite in modo che $h_t(x) = H(x,t)$.

Due nodi si dicono *isotopicamente equivalenti* o, più semplicemente, ***isotopi*** se esiste un'omotopia $H : \mathbb{R}^3 \times [0,1] \to \mathbb{R}^3$, che trasforma un nodo nell'altro, tale che $h_t : \mathbb{R}^3 \to \mathbb{R}^3$ è un omeomorfismo per ogni $t \in [0,1]$, ovvero, se esiste una famiglia "continua" di omeomorfismi che trasforma un nodo nell'altro; questo ci consente di affermare che esiste un omeomorfismo su $\mathbb{R}^3$ che trasforma un nodo nell'altro conservando l'orientazione. Dire che questa condizione è valida, è equivalente a dire che un nodo può essere "armoniosamente" deformato nello spazio fino ad assumere la stessa forma dell'altro. Non è consentito ad esempio tagliare il nodo, né farlo passare attraverso se stesso. Pertanto possiamo dire che tra i due trifogli della figura 2.2 c'è equivalenza, ma non c'è isotopia: infatti non c'è modo di trasportare il trifoglio sinistro su quello destro senza fare una delle operazioni sopra "vietate", a meno di operare una riflessione.

---

ad un omeomorfismo differenziabile (con inversa differenziabile) richiedendo che in ogni punto il differenziale (che è un omeomorfismo lineare) conservi l'orientazione. Si può dare una definizione generale di omeomorfismo che conserva l'orientazione, ma si tratta di una definizione molto tecnica che appesantirebbe la nostra trattazione.

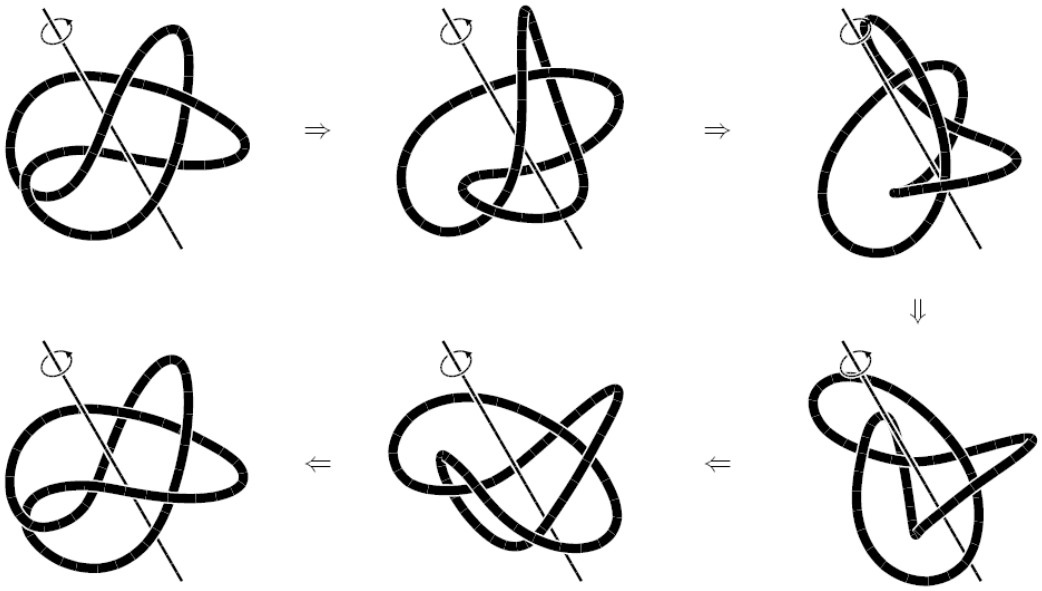


Figura 2.4: Il più semplice dei nodi achirali. La rotazione di 180° rispetto all'asse indicato, trasforma il nodo nella sua immagine speculare.

I nodi con questa caratteristica sono conosciuti come nodi ***achirali***, mentre quelli come il trifoglio vengono chiamati ***chirali***. A differenza del trifoglio, esistono alcuni nodi con la peculiarità di essere isotopi alla loro immagine speculare e ne è un esempio il nodo in figura 2.4.

Un nodo si dice ***docile*** se è equivalente ad un nodo poligonale, cioè ad un nodo ottenibile come unione finita di intervalli rettilinei; in caso contrario sarà chiamato ***selvaggio*** e ne possiamo vedere un esempio in figura 2.5 (un classico esempio di nodo selvaggio è una composizione infinita di trifogli).

Pur avendo chiaro il concetto di equivalenza, nella pratica rimane difficile stabilire se due nodi sono equivalenti e gran parte della teoria è rivolta a sviluppare tecniche che aiutino a risolvere questo problema.

Un modo alternativo per accertare l'equivalenza tra nodi è quello di esami-

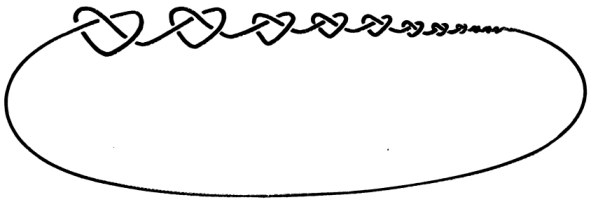


Figura 2.5: Un nodo selvaggio.

nare le rispettive proiezioni su di un piano: una tale proiezione è chiamata ***diagramma*** del nodo ed è sicuramente il modo più immediato per visualizzarlo. In figura 2.6 possiamo vedere un nodo ed un suo diagramma.

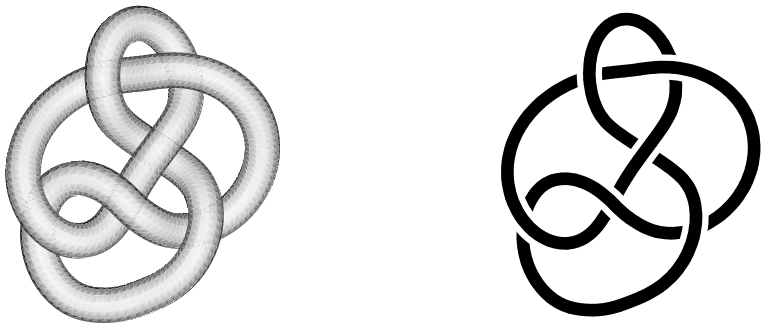


Figura 2.6: Un nodo ed un suo diagramma.

Risulta subito evidente che nel proiettare un nodo su di un piano verranno a formarsi delle intersezioni particolari, in corrispondenza dei punti dove un tratto di corda passa sopra (o sotto) ad un altro tratto. Queste intersezioni, che chiameremo *punti di incrocio* o più semplicemente ***incroci***, possono presentarsi in più forme, come spiegato in figura 2.7. E' comunque sempre possibile, con lievi perturbazioni del nodo, fare in modo che i punti d'incrocio che compongono il diagramma assumano tutti una forma simile a quella mostrata in (a), sempre nella figura 2.7.

(a) punto doppio (b) punto triplo (non ammissibile) (c) punto doppio non ammissibile

Figura 2.7: Incroci ammissibili nelle proiezioni planari

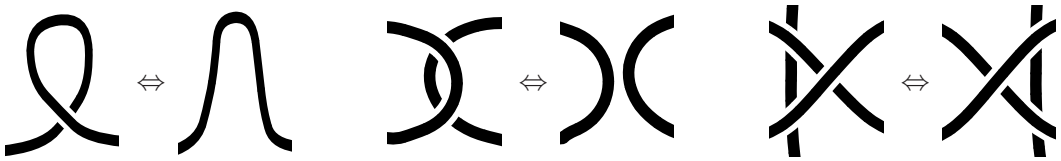
Oltre agli incroci, in un diagramma possiamo distinguere anche i "ponti" e gli "archi", a seconda di come consideriamo i tratti di curva che collegano gli incroci: un ***ponte*** è un pezzo di curva che va da un sottopassaggio fino al successivo, mentre un ***arco*** è un tratto di curva compreso tra due incroci consecutivi. In figura 2.8 possiamo vedere il trifoglio suddiviso in ponti e in archi; si noti che i ponti sono tanti quanti gli incroci, mentre gli archi sono esattamente il doppio.

ponti archi

Figura 2.8: Il diagramma del trifoglio con i ponti e gli archi evidenziati ed etichettati.

Già nel 1935 Reidemeister dimostrò che due nodi sono equivalenti se e solo se la proiezione di uno può essere convertita nella proiezione dell'altro attraverso l'uso di tre semplici movimenti, a meno di isotopie del diagramma

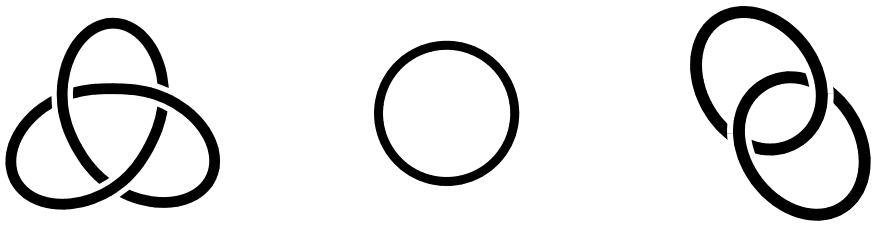
nel piano. I movimenti di Reidemeister sono mostrati in figura 2.9: ogni illustrazione in figura rappresenta una classe di movimenti.



Tipo I Tipo II Tipo III

Figura 2.9: I tre tipi di movimenti di Reidemeister.

Osservando i movimenti di Reidemeister risulta evidente che ogni nodo ha infinite proiezioni possibili sul piano. Per questo motivo, spesso si sceglie come diagramma rappresentante uno di quelli che mostrano il minor numero di incroci tra tutte le possibili proiezioni (in generale accade che ce ne siano più di uno). A questi diagrammi viene dato il nome di ***proiezione minima*** del nodo e il numero di incroci che vi si possono contare prende il titolo di ***crossing number*** del nodo stesso.

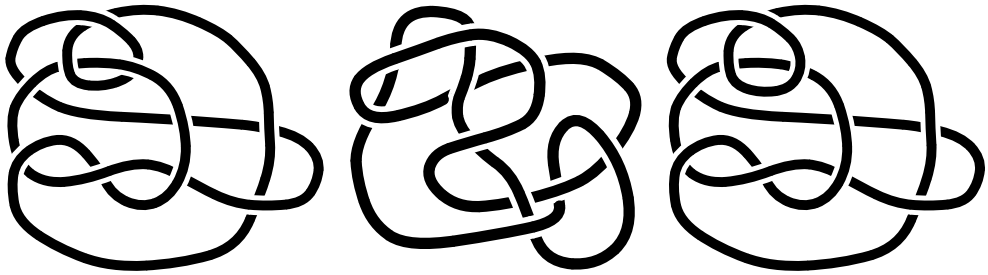


(a) trifoglio (cn=3) (b) nodo banale (cn=0) (c) link di Hopf (cn=2)

Figura 2.10: Diagrammi dei nodi più semplici.

Pensiamo nuovamente al trifoglio di figura 2.2; dovrebbe essere intuitiva-

mente chiaro che la sua proiezione minima è unica e il suo crossing number è pari a 3 (figura 2.10). Inoltre, il trifoglio è l'unico nodo avente crossing number uguale a 3. E' ancora più facile vedere che il nodo banale è l'unico nodo ad avere crossing number pari a 0 e che la sua proiezione minima è unica. In figura 2.11 possiamo vedere alcuni esempi del trifoglio e del nodo banale in proiezione non minima, con 10 incroci ciascuno. Questi diagrammi ci danno qualche indicazione sul fatto che il minimo crossing number di un nodo non sia generalmente facile da determinare.

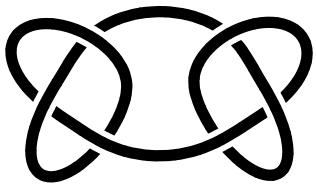


(a) nodo banale (cn=0) (b) link di Hopf (cn=2) (c) trifoglio (cn=3)

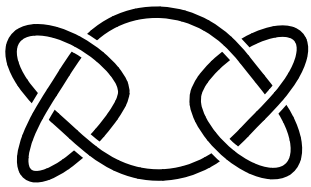
Figura 2.11: Esempi di proiezioni non minime.

Come abbiamo già avuto modo di dire nell'introduzione al capitolo, un grande problema affrontato in questa teoria è quello di trovare una o più caratteristiche peculiari dei nodi che ci consentano di farne una classificazione adeguata. In questo senso, il crossing number fornisce un'intuitiva misura di complessità di un nodo e fino a pochi anni fa era consuetudine classificare i nodi in base al loro crossing number. A questo proposito è necessario prendere in considerazione solo i nodi che siano ***primi***: questi sono i nodi che non possono

essere suddivisi in due o più nodi di semplicità superiore, non banali. I nodi che non sono primi vengono detti ***composti***. Per creare un nodo composto è sufficiente allacciare due nodi in sequenza su uno stesso pezzo di corda, come mostrato in figura 2.12. Questa operazione viene chiamata somma connessa tra nodi, e rigorosamente è così definita: dati due nodi K_1 e K_2, e fissato un verso di percorrenza su di essi, la loro ***somma connessa*** $K_1 \sharp K_2$ è il nodo ottenuto tagliando ognuno dei nodi K_1 e K_2, ed attaccando per le estremità le due stringhe così ottenute, in modo che i versi di percorrenza risultino compatibili.



nodo Quadrato



nodo della Nonna

Figura 2.12: Esempi di nodi composti, somma connessa tra trifogli.

Nel libro di Rolfsen "*Knots and Links*" [Rol76] possiamo trovare un catalogo di tutti i nodi primi fino a 10 incroci e tutti i link fino a 9 incroci, in proiezione minima, completi di diagramma. Questo catalogo, attualmente il più considerato, dipende da due importanti lavori precedenti: quello di Alexander e Briggs [AB27] e quello di Conway [Con70]. Alexander e Briggs produssero il primo catalogo completo di nodi fino a 9 incroci, nel quale ciascun nodo era con certezza distinto dagli altri appartenenti allo stesso catalogo. Conway estese questo a 11 incroci nel 1970, includendo i link fino a 5 componenti.

Numero di componenti	Crossing Number								
	2	3	4	5	6	7	8	9	10
1	0	1	1	2	3	7	21	49	165
2	1	0	1	1	3	8	16	61	184
3	0	0	0	0	3	1	10	21	73
4	0	0	0	0	0	0	3	1	21
5	0	0	0	0	0	0	0	0	3

Tabella 2.1: Numero di nodi primi e link fino a dieci incroci.

2.2 Notazioni

Per indicare un nodo, spesso si ricorrerà alla notazione usata da Alexander, Briggs, Conway e Rolfsen per classificare i nodi nelle loro opere [AB27], [Con70] e [Rol76]. Secondo questa notazione, un nodo viene identificato da tre numeri, scritti in questa forma:

$$K_n^c$$

Nella precedente, K indica il numero di incroci visibili nella proiezione minima, cioè il crossing number del nodo, c il numero di componenti (omesso se uguale a 1), mentre n non è un valore proprio del nodo, ma è un semplice indice che richiama il catalogo costruito dai quattro autori sopracitati (più precisamente da Rolfsen in [Rol76]) e distingue due nodi diversi, aventi però stesso crossing number e stesso numero di componenti, secondo il lavoro svolto in quella classificazione. Nel comporre il catalogo non si è fatta distinzione tra un nodo e la sua immagine speculare, pertanto con 3_1 viene indicato l'unico nodo con tre incroci, il trifoglio. Esiste anche un solo nodo proprio avente quattro incroci, denotato con 4_1, mentre con cinque incroci ne esistono due, 5_1 e 5_2, tre con sei incroci, 6_1, 6_2 e 6_3 e così via. Un esempio di nodo a più

componenti è il link di Hopf, indicato da 2_1^2.

2.2.1 I tangle di Conway

Nel 1970 John Conway portò una boccata d'aria nuova alla teoria dei nodi, inventando un sistema di notazione efficace che permise un'efficiente catalogazione di un numero sempre maggiore di nodi. Il sistema di Conway si basa sul concetto di ***tangle*** (in italiano 'intreccio") , ovvero sezioni di diagramma di un nodo consistenti in due stringhe separate con le estremità aperte, come mostrato nella figura 2.13. Conway prendeva in considerazione principalmente tangle a due stringhe (quattro entrate) tuttavia senza escludere la possibilità di lavorare con tangle a più stringhe.

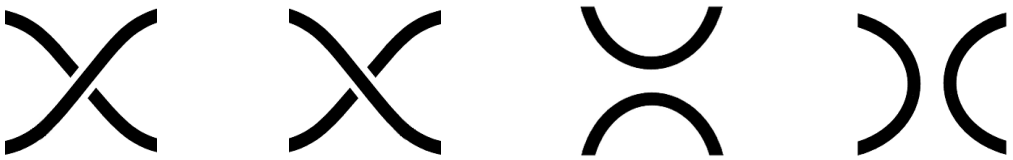


Figura 2.13: Tangle fondamentali di Conway.

I tangle primari possono essere combinati usando operazioni algebriche come nella figura 2.14. Con il simbolo "Ⓛ", Conway indica l'orientamen-

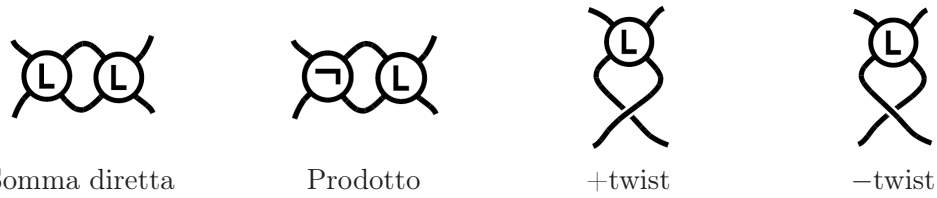


Figura 2.14: Tangle fondamentali di Conway.

to del tangle inserito. Queste operazioni producono tangle più complessi da

tangle più elementari. Per esempio, due tangle di tipo +1 possono essere sommati a formare un tangle di tipo +2. Iterando questo processo, facendo uso esclusivamente dei tangle di tipo +1 o −1, si dà origine ai ***tangle interi***, alcuni dei quali sono mostrati nella figura 2.15. I tangle integrali possono

Figura 2.15: Tangle integrali.

essere combinati usando l'operatore prodotto come nella figura 2.16. Si noti

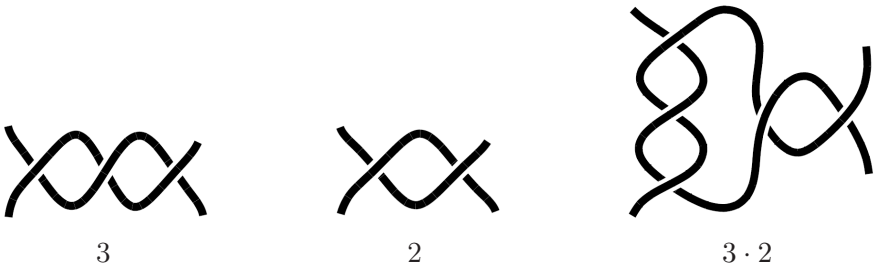
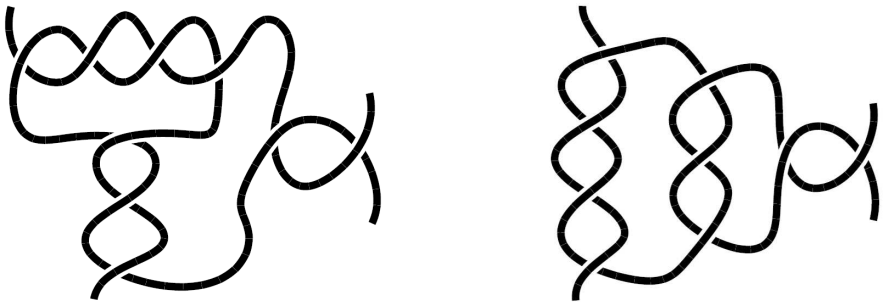


Figura 2.16: Costruzione del prodotto tra i tangle +3 e +2.

che l'operatore prodotto è simile in tutto all'operatore somma tranne che per il primo fattore, il quale viene ruotato e riflesso. Il prodotto di Conway è associativo a sinistra, cioè $abcd = (((ab)c)d)$. La figura 2.17 mostra come si ottengano tangle diversi se si considera il prodotto associativo a destra. I tangle formati in questo modo (prodotti associati a sinistra di tangle integrali) sono noti come ***tangle razionali***.

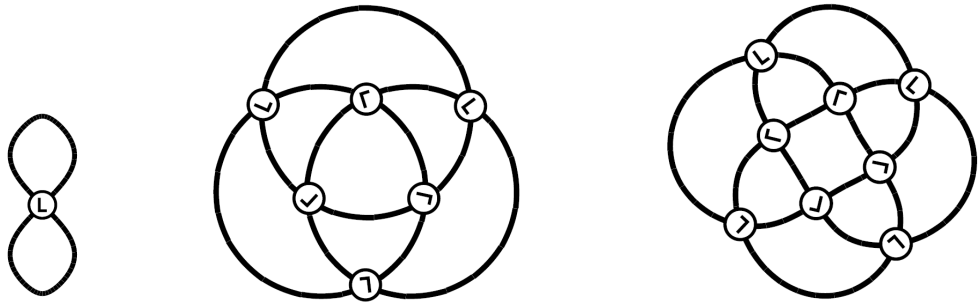
Infine, per produrre un diagramma di un nodo, le estremità libere del tangle devono essere in qualche modo collegate. Conway fornì diversi esempi di quel-



$4 \cdot 3 \cdot 2 \equiv (4 \cdot 3) \cdot 2$ $\qquad$ $4 \cdot (3 \cdot 2)$

Figura 2.17: Il prodotto tra tangle è associativo a sinistra ma non a destra.

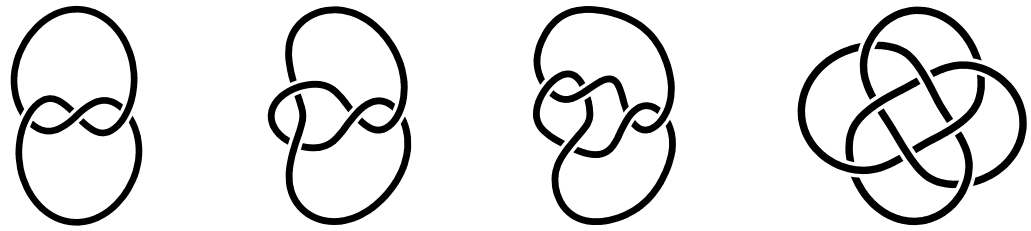
li che lui definì *poliedri primari* come modelli nei quali inserire i vari tangle. Alcuni di questi vengono mostrati in figura 2.18, ridisegnati da Scharein in [Sch98] ma topologicamente equivalenti agli originali di Conway. Il più sem-



$1^\star$ $\qquad$ $6^\star$ $\qquad$ $8^\star$

Figura 2.18: Alcuni poliedri di base.

plice di questi è il poliedro $1^\star$, che collega tra di loro le due estremità superiori (Nord-Est e Nord-Ovest) e le due estremità inferiori (Sud-Est e Sud-Ovest) per formare ciò che viene chiamato il numeratore del tangle. Per esempio, inserendo un tangle di tipo $+3$ nel poliedro $1^\star$ si ottiene il nodo di sinistra in figura 2.19, facilmente riconoscibile come trifoglio. Per convenzione, Conway



3 (trifoglio) $2 \cdot 2$ (4_1) $2 \cdot 1 \cdot 2$ (5_1^2) $8^\star$ (8_{18})

Figura 2.19: Alcuni nodi con la loro notazione secondo Conway. I primi tre sono costruiti per mezzo del poliedro $1^\star$

afferma che nel caso in cui venga usato il poliedro $1^\star$ la notazione per il nodo è la stessa del tangle inserito, cosicché il trifoglio può essere identificato dal numero 3.

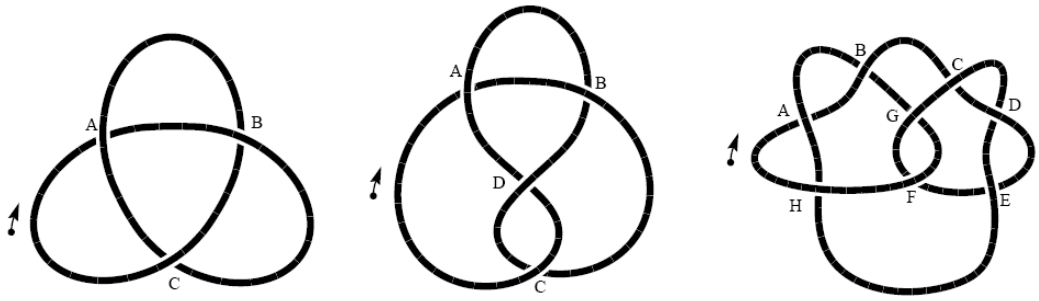
La costruzione di nodi con poliedri primari più complessi del tipo $1^\star$ richiede che alcuni dei tangle inseriti vengano riflessi o ruotati in maniera simile a quanto fatto nell'operazione prodotto mostrata in figura 2.14; infatti, proprio come in questo caso i simboli "Ⓛ" guidano l'inserimento dei tangle. Ad esempio, se si prova ad inserire dei tangle di tipo $+1$ in ciascun punto d'inserzione del poliedro primario $8^\star$ si otterrà il nodo di destra della figura 2.19. Sebbene in questo caso la notazione di Conway per il nodo sia semplicemente $8^\star$, per la maggior parte dei nodi la notazione è alquanto più complessa[3]. In questa analisi del calcolo di Conway è stato necessario omettere molti dettagli ed applicazioni, vedi in particolare i metodi da lui usati per eliminare i duplicati nella sua tavola. Il sistema di notazione è designato per far sì che operazioni complesse sui diagrammi dei nodi corrispondano a una semplice manipolazione delle stringhe del tangle. Di notevole importanza teorica, inoltre, è la curiosa relazione scoperta

[3]Per maggiori dettagli si consiglia di consultare [Con70].

da Conway tra i ***nodi razionali*** e i numeri razionali (scritti come frazioni finite continue). I nodi razionali sono i nodi ottenuti dall'inserimento di un tangle razionale nel poliedro $1^{\star}$. Non approfondiremo ulteriormente tale relazione vista la scarsa incidenza nei contenuti della tesi; per chi fosse interessato può consultare direttamente [Con70]. Forse il più grande contributo del trattato di Conway è il concetto stesso di tangle.

2.2.2 I codici di Gauss e di Dowker-Thistlethwaite

Il ***Codice di Gauss*** per il diagramma di un nodo proprio sul piano consiste in una sequenza di lettere che codificano gli incroci. Per un nodo a n incroci, il Codice di Gauss ha $2n$ lettere, dal momento che ciascuna lettera ricorre due volte. Ai fini della codificazione, bisogna per prima cosa etichettare tutti gli incroci sul piano con le lettere A, B, C, ... come mostrato per tre nodi diversi nella figura 2.20, successivamente si sceglierà un'orientazione, un punto di



ABCABC ABCDBADC ABCDEFGCDEHABGFH

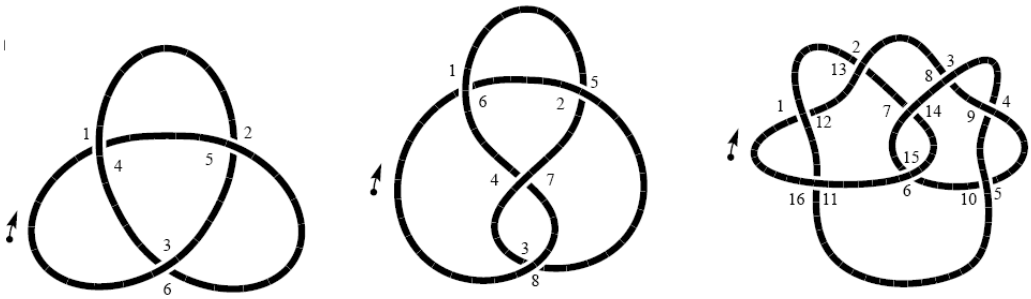
Figura 2.20: Alcuni nodi con il loro codice di Gauss.

partenza nel diagramma e si procederà per tutto il nodo riportando l'etichetta di ciascun incrocio incontrato lungo il tragitto. Una volta ritornati al punto di partenza si sarà ottenuta una sequenza di $2n$ etichette, ciascuna ricorrente

due volte, che codifica completamente il diagramma del nodo (fino alle sue immagini speculari). L'esempio del trifoglio dà la sequenza ABCABC. Durante questa procedura non si tiene nota della natura soprapassante o sottopassante degli incroci, e ciò limita l'applicazione del codice di Gauss a diagrammi alternanti; per ***diagramma alternante*** intendiamo un diagramma dove la corda passa, ad esempio, sotto, sopra, sotto ... a mano a mano che si procede lungo il nodo. Una volta raggiunti nodi a otto incroci, cominciamo ad incontrare nodi non-alternanti in proiezione minima, si rende quindi necessario incrementare in qualche modo il Codice di Gauss.

Una semplice variazione del Codice di Gauss, che possa adattarsi al caso dei non-alternanti, venne sviluppata da Hugh Dowker e si è dimostrata preziosissima nell'enumerazione di Dowker e Thistlethwaite dei nodi fino a 13 incroci ([DT83] e [Thi85]). Il ***codice di Dowker-Thistlethwaite*** (o DT-code) per la proiezione di un nodo, viene ottenuta in modo simile al Codice di Gauss: anche qui si sceglie un punto di partenza sul diagramma e si procede in una direzione intorno al nodo. Questa volta, però, l'operazione di identificare gli incroci non può essere fatta preventivamente, ma deve essere eseguita mano a mano che si percorre il nodo: ciascun incrocio va identificato, nell'ordine, con i numeri $1, 2, 3, \ldots 2n$ per un nodo a n incroci e riceve quindi due etichette distinte, una per ogni passaggio sull'incrocio stesso. Non è difficile vedere che i due numeri assegnati a ciascun incrocio sono uno pari ed uno dispari. Questa procedura definisce una funzione che potremmo definire a "parità inversa" $p(i)$ degli interi $1, 2, 3, \ldots 2n$ in loro stessi, tale che gli incroci i e $p(i)$ si proiettano sullo stesso punto del piano e $p(p(i)) = i$. Considerato che ciascun incrocio riceve una

etichetta pari e una dispari, la funzione p(i) è completamente specificata dalla sequenza $p(1), p(3), \ldots p(2n-1)$, che saranno, ovviamente, tutti numeri pari. Se dotiamo ciascun termine di questa sequenza di un segno + o un segno − a seconda che l'incrocio $p(i)$ sia soprapassante o sottopassante, otterremo il DT-code del nodo. Nella fig. 2.21 sono contenuti diversi esempi; ovviamente

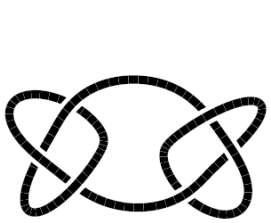


[4 6 2] [6 8 2 4] [12 8 10 −14 4 16 2 −6]

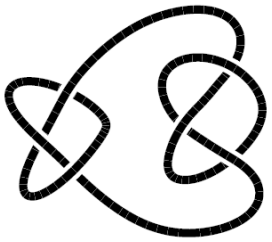
Figura 2.21: Alcuni nodi con il loro codice di Dowker-Thistlethwaite.

il codice può dipendere dal punto di partenza e dalla direzione (indicata nella figura da un punto con una freccia). Invertendo, per esempio, l'orientazione sul trifoglio di fig. 2.21 si ottiene il codice $[-4\ -6\ -2]$, che in questo caso inverte semplicemente i segni. In questo esempio, il codice non dipende dal punto di partenza a causa della simmetria nel diagramma del trifoglio; invece, nel caso del nodo sulla destra, sempre in fig. 2.21, tutte le 16 possibilità di scelta di punto di partenza e orientazione danno origine a codici diversi. Questo nodo, inoltre, ha più di un diagramma avente minimo numero di incroci e ciascuno di questi produce altri codici distinti. Nonostante la maggior parte dei nodi (diagrammi) abbia molti codici DT, uno di questi sarà minimo in senso stretto e potrà essere scelto come "nome" standard per il nodo (diagramma).

Non è immediatamente evidente che un DT-code specifica unicamente un nodo dato, ed infatti già Adams [Ada94] aveva capito che questo codice non è in grado di distinguere nodi composti, come mostrato in fig. 2.22. Per



Nodo della Nonna



Nodo Quadrato in una proiezione alternativa

Figura 2.22: Nodi composti diversi aventi lo stesso DT-code [10 6 8 4 12 2].

approfondimenti sull'argomento, come pure sul problema di riconoscere quando un codice dato effettivamente specifica un nodo, si può fare riferimento allo studio di Dowker e Thistlethwaite [DT83], dove viene fornito un algoritmo per filtrare i casi non validi.

Sia il codice di Gauss che quello di Dowker-Thistlethwaite si sono dimostrati limitati per identificare il diagramma di un nodo, principalmente perchè non è possibile usarli per i link, ma anche perchè all'interno dei soli nodi propri spesso falliscono, come per il caso della figura 2.22. Per questi motivi adotteremo un'altro codice da utilizzare nel nostro programma, che si dimostrerà più efficiente.

2.3 Invarianti topologici dei nodi

Gran parte della teoria dei nodi è coinvolta nel tentativo di fornire dei buoni metodi per determinare se due nodi, esibiti in due rappresentazioni differenti,

siano in realtà lo stesso. In questo senso ci vengono in aiuto gli invarianti topologici. Un ***invariante topologico*** di un nodo è una sua proprietà intrinseca, che non cambia cioè se il nodo è rappresentato in maniera differente e resta valida anche per tutti gli altri nodi equivalenti al primo. La conseguenza importante che ci interessa dell'invarianza topologica è che due nodi che hanno proprietà diverse per uno stesso invariante non possono essere equivalenti.

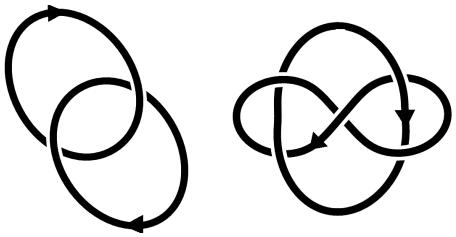
Un esempio banale di invariante topologico per i nodi è il numero di componenti di un link. Due link aventi un numero diverso di componenti non possono ovviamente essere equivalenti; questo può essere provato facilmente facendo notare che i movimenti di Reidemeister non cambiano il numero di componenti. Probabilmente il numero di componenti è il più debole invariante dei nodi, mentre quelli che vedremo in seguito sono sicuramente più potenti e interessanti. Tuttavia, un invariante che sia in grado di distinguere *tutti* i nodi e che non sia troppo difficile da valutare, ancora non è stato scoperto.

2.3.1 Linking number

Per link con due o più componenti, un semplice invariante topologico più potente del numero di componenti è il ***linking number*** o numero di allacciamento. Questo può essere definito in diversi modi ([Rol76, pag.132] ne dà otto definizioni equivalenti), ma il più semplice, in termini di diagramma del nodo, prende in considerazione gli incroci in cui una componente passa sopra un'altra (le auto-intersezioni vengono ignorate). Ad ogni incrocio viene assegnato un valore che può essere -1 o +1, a seconda che sia, rispettivamente, *sinistrorso* o *destrorso*, come mostrato in figura 2.23. Si noti che, per poter dare un segno

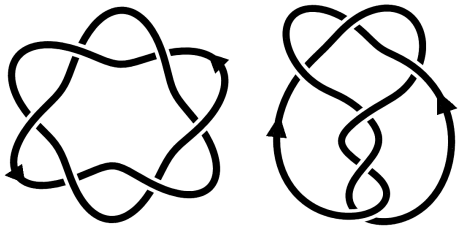
sinistrorso, -1 destrorso, +1

Figura 2.23: Tipi di incroci orientati.



(a) $L(2_1^2) = 1$ (b) $L(5_1^2) = 0$

Figura 2.24: Due nodi distinti dal loro linking number.



(a) $L(6_1^2) = 3$ (b) $L(6_2^2) = 3$

Figura 2.25: Due nodi non equivalenti aventi stesso linking number.

all'incrocio, ciascuna componente deve essere orientata[4]. Il linking number $L(A, B)$ tra le due componenti A e B in un link è definito come la metà della somma dei valori associati agli incroci tra i due componenti. Se cambiamo l'orientazione di una componente cambia anche il segno del linking number. Anche in questo caso, con l'aiuto dei movimenti di Reidemeister non è difficile dimostrare che il linking number è un invariante topologico per i nodi. Con questo invariante possiamo dimostrare che i due nodi della figura 2.24 sono diversi. Tuttavia, la figura 2.25 ci mostra che l'invariante non è completo: i due nodi rappresentati sono differenti, ma il loro linking number è il medesimo.

[4]Un modo semplice per individuare il segno dell'incrocio è il seguente: porre la mano destra, di taglio, sulla componente che passa sopra, con il mignolo sulla freccia, e spostare la mano seguendo l'angolo minore (convesso) fino a sovrapporsi all'altra freccia. Se il movimento è nel verso della chiusura del pugno, allora l'incrocio è destrorso, e quindi positivo, altrimenti è sinistrorso e negativo.

2.3.2 Crossing Number

Il ***crossing number*** è dato dal numero di incroci che si possono contare sul diagramma di un nodo in proiezione minima. E' un invariante molto importante perchè, pur non essendo completo, in quanto non è in grado di distinguere alcuni nodi diversi tra loro, fornisce una immediata misura della complessità di un nodo. Questa particolare caratteristica ha suscitato la formulazione di un primo catalogo dei nodi da parte di Rolfsen in [Rol76]. Del crossing number abbiamo già parlato a pagina 17.

2.3.3 Stick number

Letteralmente "numero di bastoncini", lo ***stick number*** è un altro invariante polinomiale per nodi semplici. Corrisponde al minor numero di "bastoncini" necessari per costruire il nodo. Più formalmente, lo stick number $s(K)$ di un nodo K è il più piccolo n per il quale esista un poligono di n lati in $\mathbb{R}^3$ che rappresenti K.

Seppure apparentemente semplice, ancora si sa poco su questo tipo di invariante. Infatti, per alcuni nodi ancora non è stato dimostrato che il poligono *minimo* ad essi associato sia effettivamente tale, cioè non se ne possa trovare un altro con un numero inferiore di lati che rappresenti ancora il nodo.

Quanto finora accertato sullo stick number, può essere riassunto così:

- Il trifoglio (3_1) è l'unico nodo proprio che può essere costruito con sei stick, mentre il nodo 4_1 è l'unico nodo proprio con stick number pari a sette. Nella tabella 2.2 possiamo vedere lo stick number di altri nodi; in

realtà, i nodi per i quali è noto lo stick number non sono molti di più di quelli che si vedono in tabella.

- La somma connessa di N trifogli ha stick number pari a $2N + 4$.

- Per un nodo K con crossing number $c(K)$ lo stick number $s(K)$ è limitato superiormente e inferiormente da:

$$\frac{5 + \sqrt{8c(K) + 9}}{2} \leq s(K) \leq 2c(K)$$

  come dimostrato in [Neg91]. Il limite inferiore è piuttosto debole, infatti non vieta che nodi con dieci incroci possano avere stick number pari a otto, il che è decisamente inammissibile come visto dai risultati sperimentali di [Mei96] e [Sch98].

- I nodi razionali hanno un limite superiore più basso. McCabe [Ran96] ha dimostrato che per questi nodi vale $s(K) \leq c(K) + 4$.

- I nodi torici del tipo $(m, m-1)$ hanno stick number pari a $2m$ [ABGW97].

Gran parte del lavoro fatto per determinare lo stick number di alcuni nodi, necessariamente sperimentale, è stato condotto da Meissen [Mei96] presso l'Università di Iowa, negli Stati Uniti. Meissen arrivò ai suoi risultati manipolando manualmente ciascun nodo, attraverso il programma di elaborazione e visualizzazione di nodi creato da Hunt, il KED [Hun96]. Indubbiamente si tratta di un lavoro che richiede molta pazienza, ma attualmente sembra essere il metodo conosciuto più efficace per trovare lo stick number di un nodo.

nodo	stecche
3_1	$6^\star$
4_1	$7^\star$
5_1	$8^\star$
5_2	$8^\star$
6_1	$8^\star$
6_2	$8^\star$

nodo	stecche
6_3	$8^\star$
7_1	9
7_2	9
7_3	9
7_4	9
7_5	9

nodo	stecche
7_6	9
7_7	9
8_1	10
8_{19}	$8^\star$
8_{20}	$8^\star$
8_{21}	9

Tabella 2.2: Stick number di alcuni nodi, dai risultati sperimentali di Meissen. Il simbolo * indica che il valore è dimostrato essere il migliore possibile (quindi è lo stick number vero e proprio del nodo).

2.3.4 Invarianti polinomiali

Il nostro esame degli invarianti polinomiali dei nodi segue l'eccellente analisi contenuta nella trattazione di Wu [Wu92], dove egli presenta un trattamento omogeneo di diversi nodi polinomiali in termini di ***relazioni a matassa*** (tradotto dall'inglese *skein relation*). Il concetto di relazione a matassa venne inventata da Conway [Con70] e consiste in una relazione tra funzionali definiti su un insieme di diagrammi di nodi che differiscono solo localmente. Essenzialmente l'idea consiste nel "ritagliare" un tangle (per la definizione di tangle vedi pagina 21) da un diagramma di un nodo e sostituirlo con un altro. I tangle in questione compaiono nella figura 2.26 e sono stati già discussi a proposito della notazione di Conway; qui però le stringhe nei tangle sono orientate. Altra

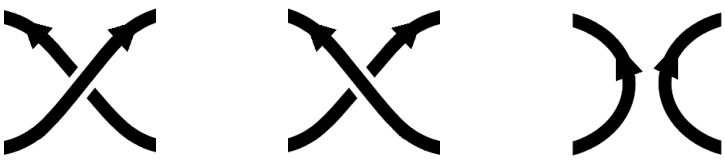


Figura 2.26: Tangle presenti nelle relazioni a matassa.

piccola differenza dai tangle incontrati in precedenza è che L_0 assomiglia al tangle ∞ di Conway. La figura 2.27 mostra esempi di diagrammi di un nodo collegati attraverso una relazione a matassa che differiscono solo localmente nelle regioni ritagliate.

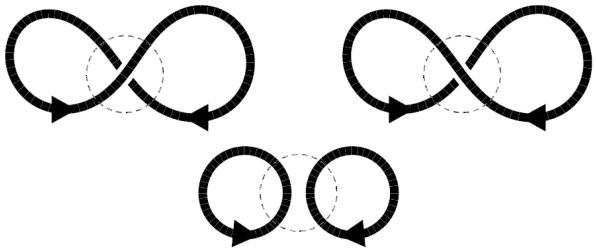


Figura 2.27: Esempio di diagrammi collegati da relazioni a matassa.

Un *invariante polinomiale* per i nodi è un'espressione di un certo diagramma di un nodo attraverso i polinomi di Laurent[5], invariante per movimenti di Reidemeister. Alexander derivò il polinomio che porta il suo nome attraverso un metodo completamente indipendente dalle relazioni a matassa [Ale28]. Le tecniche che egli usò esulano dagli scopi di questa tesi, quindi non ci addentreremo ulteriormente nella questione. Conway riformulò il polinomio di Alexander attraverso una relazione a matassa del tipo

$$\begin{aligned} \nabla_{L_+}(z) - \nabla_{L_-}(z) &= z\nabla_{L_0}(z) \\ \nabla_{\text{unknot}}(z) &= 1 \end{aligned}$$

dove L_+, L_- e L_0 rappresentano tre diagrammi di nodo che differiscono solo localmente nella maniera mostrata nella figura 2.27, mentre i termini $\nabla(z)$ rappresentano i polinomi di Conway e la formula sopraindicata ci mostra come

[5]Polinomi del tipo $a_{-n}t^{-n} + a_{-(n-1)}t^{-(n-1)} + \cdots + a_{-1}t^{-1} + a_0 + a_1t + \cdots + a_nt^n$ dove gli a_i appartengono ad un campo $\mathbb{F}$.

questi sono collegati gli uni agli altri. Calcolare il polinomio per un nodo dato comporta l'applicazione ripetuta di una relazione a matassa, finchè non si rimanga con un insieme di nodi banali (o altri nodi i cui polinomi siano stati precedentemente determinati). Ad ogni passo, è consentita una qualsiasi sequenza di movimenti di Reidemeister. Il risultato di tale procedura è un albero binario, noto come albero a matassa, un esempio del quale è contenuto nella figura 2.28 per il nodo 6_3. Le foglie di questo albero sono tutti nodi banali aventi polinomio noto $\nabla(z) = 1$. Operando a ritroso verso la radice dell'albero attraverso l'applicazione della formula a matassa, ci permette di calcolare il polinomio del nodo originale. Se osserviamo le figure (q), (r) e (u), notiamo che tra le prime due viene eseguita una trasformazione $0 \mapsto -$, mentre tra (q) ed (u) la trasformazione è del tipo $0 \mapsto +$. Questo ci consente di calcolare il polinomio di Conway di (q) dai polinomi delle altre due figure in questo modo:

$$\nabla_{L_0}(z) = \frac{\nabla_{L_+}(z) - \nabla_{L_-}(z)}{z} = \frac{1-1}{z} = 0$$

e risalendo l'albero fino alla radice, otteniamo il polinomio del nodo.

Non è difficile mostrare, come da [LM88], che il calcolo del polinomio termini in un numero finito di passi. Per poter affermare che il polinomio è ben definito, rimane da provare che il polinomio risultante sia unico, cioè non dipenda dalla forma esatta dell'albero a matassa scelto. Guardando la figura 2.28 risulta probabilmente ovvio che per lo stesso nodo siano possibili più alberi a matassa, dal momento che a ciascun passo di solito le scelte sono molteplici: infatti è possibile scegliere l'incrocio al quale andrà applicata la relazione a matassa per giungere al passo successivo. Quello che invece non è ovvio è che il polinomio sia effettivamente un invariante per il nodo. In effetti,

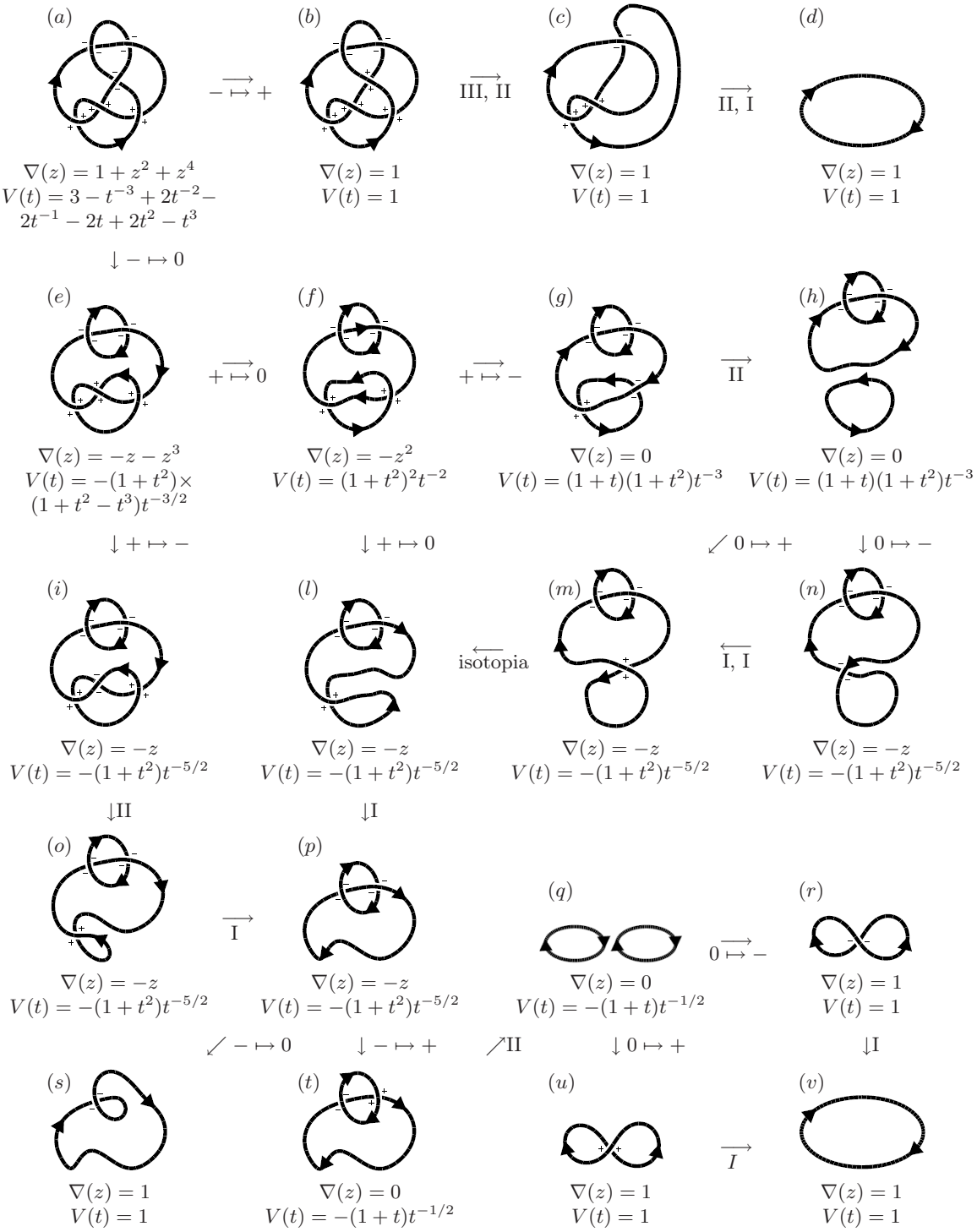


Figura 2.28: Albero a matassa per il calcolo del polinomio di Conway e di Jones per il nodo 6_3.

il polinomio è allo stesso tempo unico ed un invariante ben definito per il nodo, come dimostrato da Kauffman in [Kau91].

Come abbiamo già detto, il polinomio di Conway è una riformulazione del polinomi originario di Alexander, $\triangle(t)$. I due polinomi sono effettivamente equivalenti e collegati dalla relazione:

$$\nabla(z) = \triangle(t), \quad z = \frac{1}{\sqrt{t}} - \sqrt{t}$$

Questa differenza di normalizzazione è comune nel campo dei nodi polinomiali, dove gli stessi polinomi compaiono spesso in forme diverse. Sebbene il polinomio di Conway sia computazionalmente equivalente al polinomio di Alexander, l'approccio di Conway è importante poiché rivela che il polinomio può essere calcolato attraverso tecniche combinatorie sul diagramma di un nodo (le relazioni a matassa) piuttosto che per mezzo delle molto più astruse nozioni topologiche usate da Alexander. Tuttavia è ironico che dall'approccio di Alexander sia possibile derivare un algoritmo efficiente per il calcolo del polinomio di Alexander-Conway che abbia tempo di esecuzione polinomiale rispetto al numero di incroci originario del diagramma del nodo: questo contrasta con l'andamento esponenziale che ci aspetteremmo nel calcolare la stessa quantità usando l'albero a matassa.

Il polinomio di Alexander-Conway risulta essere un invariante ragionevolmente potente dei nodi. Dei 249 nodi propri primari con dieci o meno incroci, 175 si distinguono per il loro polinomio di Alexander-Conway, 68 condividono il polinomio con un altro nodo, mentre i restanti 6 finiscono in due gruppi di tre nodi aventi stesso polinomio. Il polinomio non distingue le immagini speculari: entrambe le versioni del trifoglio (figura 2.2) hanno un polinomio

di Conway pari a $1 + z^2$. Anche alcuni semplici nodi composti, come il nodo Quadrato e il nodo della Nonna (figura 2.12) hanno lo stesso polinomio di Conway $1 + 2z^2 + z^4$. La ragione per la quale questi due nodi non vengono distinti è legata al fatto che, in primo luogo, i due trifogli hanno lo stesso polinomio, in secondo luogo, sussiste un'interessante proprietà relativa a come il polinomio si comporta sotto la somma diretta di due nodi. Se K e L sono due nodi (non necessariamente primari) con polinomi di Conway $\nabla(K)$ e $\nabla(L)$, allora il polinomio di Conway del nodo composto $K\sharp L$ ($\sharp$ è la somma connessa di due nodi; vedi pag. 19) è dato da

$$\nabla_{K\sharp L}(z) = \nabla_K(z)\nabla_L(z)$$

così possiamo notare che l'insuccesso nel distinguere il nodo Quadrato da quello della Nonna risulta dall'impossibilità di distinguere le immagini speculari.

Ancora più scoraggiante dell'incapacità di distinguere le immagini speculari è il fatto che per ogni nodo dato esiste un numero infinito di nodi non equivalenti con lo stesso polinomio di Alexander-Conway [Whi37]. In particolare, per quanto riguarda il polinomio, esiste una quantità infinita di nodi diversi che sono indistinguibili dal nodo banale. Il primo di questi, un nodo a 11 incroci, compare nella figura 2.29.

Per buona parte della storia della teoria dei nodi, il polinomio di Alexander è stato il solo polinomio dei nodi. Fu uno shock per molti quando Vaughan Jones [Jon85] scoprì nel 1985 un nuovo e potente invariante polinomiale per i

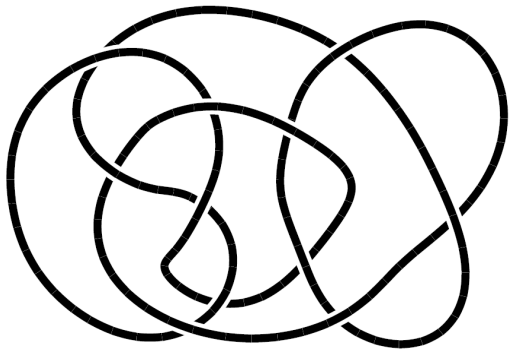


Figura 2.29: Il primo di una lunga serie (infinita) di nodi non banali avente stesso polinomio di Alexander-Conway del nodo banale. È un nodo ad 11 incroci.

nodi a più componenti, definito dalla relazione a matassa

$$\begin{aligned} \frac{1}{t}V_{L_+}(t) - tV_{L_-}(t) &= \left[\sqrt{t} - \frac{1}{\sqrt{t}}\right]V_{L_0}(t) \\ V_{\text{unknot}}(t) &= 1 \end{aligned}$$

Il polinomio di Jones è di gran lunga più discriminante del polinomi di Alexander-Conway. Per esempio, è in grado di distinguere le due versioni del trifoglio; i trifogli sinistro e destro hanno, come polinomi di Jones, rispettivamente $-t^{-4} + t^{-3} + t^{-1}$ e $t + t^3 - t^4$. Questi due polinomi illustrano una caratteristica generale del polinomio di Jones. Se il polinomio di Jones di un nodo dato è noto, allora il polinomio di Jones della sua immagine speculare può essere trovato operando la sostituzione $t \to \frac{1}{t}$. Nel caso del trifoglio e di molti altri nodi questa proprietà fornisce una facile prova della chiralità del nodo. Tuttavia, il metodo non è perfetto; il nodo 9_{42} di figura 2.30 ha come polinomio di Jones $-1 + t^{-3} - t^{-2} + t^{-1} + t - t^2 + t^3$ che resta invariato dalla sostituzione $t \to \frac{1}{t}$ anche se il nodo 9_{42} è chirale [HW92]. Il polinomio di Jones condivide la proprietà del polinomio di Alexander-Conway relativa ai

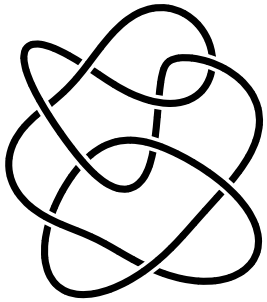


Figura 2.30: Un nodo, il 9_{42}, la cui chiralità non è rilevata dal polinomio di Jones.

nodi composti, secondo cui

$$V_{K\sharp L}(z) = V_K(z)V_L(z)$$

In questo caso risulta che il nodo Quadrato avente polinomio $3 - t^{-3} + t^{-2} - t^{-1} - t + t^2 - t^3$, viene distinto da ambedue le versioni del nodo della Nonna (i polinomi sono $t^{-8} - 2t^{-7} + t^{-6} - 2t^{-5} + 2t^{-4} + t^{-2}$ per la somma di due trifogli sinistri e $t^2 + 2^4 - 2^5 + t^6 - 2t^7 + t^8$ per la somma di due trifogli destri).

La scoperta di Jones ha avuto un effetto profondo sulla comunità della teoria dei nodi, facendo sì che i ricercatori cominciassero a cercare nuovi invarianti polinomiali. E' notevole come sei ricercatori che lavoravano indipendentemente in quattro gruppi arrivarono allo stesso polinomio quasi simultaneamente (dopo appropriate normalizzazioni) che è una generalizzazione sia del polinomio di Alexander-Conway che di quello di Jones [FYH$^+$85]. Il nuovo polinomio, chiamato HOMFLY secondo le iniziali dei suoi scopritori, può essere ottenuto da quello di Conway o da quello di Jones operando le seguenti sostituzioni:

$$\nabla(z) = P(1, z), \quad V(t) = P\left(t, \sqrt{t} - \frac{1}{\sqrt{t}}\right)$$

Il polinomio HOMFLY è definito dalla relazione

$$\begin{aligned} \frac{1}{t}P_{L_+}(t,z) - tP_{L_-}(t,z) &= zP_{L_0}(t,z) \\ P_{\text{unknot}}(t,z) &= 1 \end{aligned}$$

Questo polinomio, a differenza degli altri due, è in grado di distinguere molti nodi, pur non essendo anch'esso un invariante completo. La figura 2.31 mostra tre nodi "pretzel" non equivalenti con identici polinomi HOMFLY (e quindi Alexander-Conway e Jones). Esistono altri polinomi di nodi oltre ai tre di-

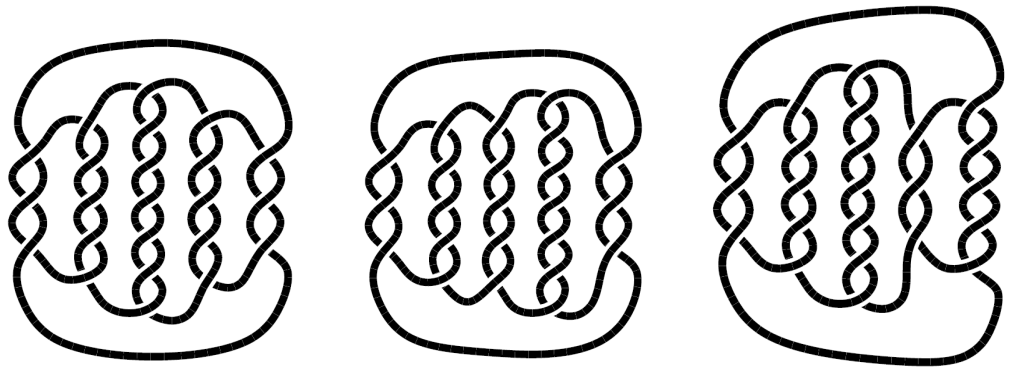


Figura 2.31: Tre nodi "pretzel" differenti, con la loro notazione di Conway, aventi stesso polinomio HOMFLY pari a $t^{-24}(-2-2t^2+2t^4+2t^6+t^8-z^2-t^2z^2+2t^4z^2+6t^6z^2+12t^8z^2+15t^{10}z^2+12t^{12}z^2+6t^{14}z^2+t^{16}z^2+t^2z^4+5t^4z^4+13t^6z^4+23t^8z^4+29t^{10}z^4+27t^{12}z^4+19t^{14}z^4+10t^{16}z^4+4t^{18}z^4+t^{20}z^4)$.

scussi in questa tesi, in particolare il polinomio di Kauffman e il polinomio di Akutsu-Wadati. Questi polinomi sono ottenuti in maniera diversa da quelli visti, dal momento che usano una versione modificata dell'albero a matassa. Il polinomio di Kauffman sembra essere grossomodo equivalente al polinomio HOMFLY per l'identificazione di un nodo; è in grado di distinguere i nodi contenuti nella figura 2.31. D'altro canto, esistono nodi che il polinomio HOMFLY

può distinguere e il polinomio di Kauffman no. Ancora una volta, il saggio di Wu [Wu92] e il libro di Kauffman [Kau91] sono due buoni riferimenti.

Capitolo 3

La teoria dei nodi al computer

In questo capitolo parleremo brevemente dei software reperibili in rete al momento della compilazione di questa tesi, che abbiano a che fare in qualche maniera con la topologia, la visualizzazione di curve e superfici, il calcolo di oggetti matematici ed in particolare con i nodi.

Dapprima individueremo le caratteristiche che desidereremmo trovare nel nostro programma ideale, distinguendo le qualità di calcolo da quelle grafiche; in seguito forniremo delle informazioni più dettagliate sui programmi che abbiamo ritenuto più interessanti.

3.1 Un programma ideale

Alla luce di quanto è stato descritto nel precedente capitolo possiamo esporre le caratteristiche che un buon programma dovrebbe avere per facilitare lo studio e l'approfondimento della teoria dei nodi.

Per fare questo analizzeremo innanzitutto ciò che il programma dovrebbe calcolare, poi quello che il programma ci dovrebbe consentire di visualizzare e di "fare" graficamente sul nodo.

3.1.1 Caratteristiche tecniche

Caratteristica prioritaria del programma deve essere la *codifica* del nodo, una codifica che sia *completa* e allo stesso tempo la più semplice possibile.

Per "completezza" di una codifica si intende l'esistenza di una biunivocità che deve sussistere tra il nodo e la sua codifica specifica. In una situazione pratica di ricerca di una codifica risulta semplice far sì che ad un nodo corrisponda una codifica specifica unica, mentre è assai difficile l'inverso: che ad una codifica specifica corrisponda un unico nodo! Ne sono di esempio la codifica di Gauss e il codice di Dowker-Thistlethwaite di cui abbiamo trattato nel capitolo 2.

Per avere la certezza della biunivocità è necessario aumentare il numero delle informazioni inserite nella codifica; tuttavia, meno informazioni dovremo inserire, più semplice sarà la codifica stessa (mantenendo la completezza!).

Nello scegliere una codifica idonea bisogna tenere conto preventivamente delle operazioni che dovranno essere effettuate su di questa e che il programma utilizzerà, ad esempio:

- inversione di una componente;
- inversione di un incrocio;
- movimenti di Reidemeister;
- trasformazioni di singoli tangle;
- trasformazione speculare;

e ulteriori operazioni di calcolo:

- calcolo degli invarianti (polinomiali, linking number, crossing number, stick number);
- minimizzazione dell'energia.

In ogni caso, la codifica dovrà dare la certezza di una certa versatilità.

3.1.2 Caratteristiche grafiche

Il programma deve consentirci di visualizzare il nodo nelle dimensioni 2D e 3D; in particolare, nella 2D ciò che viene rappresentato è una proiezione del nodo che nella teoria viene definita "diagramma". A sua volta la visualizzazione deve consentirci diverse forme di interattività, ad esempio:

1. creazione di un nodo;
2. rotazione e traslazione del nodo;
3. editazione degli incroci;
4. movimenti di Reidemeister.

Quest'ultima è di maggiore interesse e complessità rispetto alle altre.

Infine, per completare il programma, sarebbe opportuno che fosse in grado di esportare la visualizzazione in altri formati grafici, piu precisamente in PS, EPS, GIF, JPG e nel caso di animazioni in GIF o PPM.

3.2 Caratterizzazione dei programmi di grafica topologica

Effettuando in internet una ricerca con le parole chiave "knot" (nodo) e "software" ci accorgiamo che l'idea di accostare alla teoria dei nodi le capacità te-

cniche di calcolo e le grandi potenzialità di visualizzazione messe a disposizione dai sistemi moderni è già stata accolta da diversi matematici.

Sulla rete globale si trovano come al solito le soluzioni più disparate: si va dal semplice programma per la sola visualizzazione di figure geometriche, ai più potenti software per il calcolo matematico di superfici, passando per il comodo vademecum dei nodi da portare sempre con sè sul proprio "palmare", che insegna i movimenti da far compiere alla corda per ottenere l'intreccio giusto al momento giusto. Ovviamente i programmi come quest'ultimo non fanno al caso nostro.

Vediamo, tra tutti i software analizzati, le caratteristiche di quelli che abbiamo ritenuto più interessanti.

3.2.1 Geomview

Il *Geomview* è un programma interattivo di visualizzazione e manipolazione di superfici per piattaforme Unix. È stato realizzato dal *Geometry Center* dell'Università di Minnesota nel 1991, allo scopo di fornire un software interattivo che fosse particolarmente appropriato per la ricerca e l'educazione matematica. Può essere utilizzato come un programma di visualizzazione a sé stante per oggetti statici, oppure come motore grafico per altri programmi che producono geometrie in variazione dinamica. Viene usato principalmente per grafici 3D, ma può anche visualizzare dati in 2D e 4D. Inoltre consente di rappresentare oggetti nello spazio iperbolico o sferico così come in quello Euclideo. Non è possibile invece visualizzare volumi veri e propri, a meno di utilizzare dei package appositi.

Geomview può essere usato per manipolare la visualizzazione di dati prove-

nienti da altri programmi che sono in fase di esecuzione simultaneamente. Appena gli altri programmi modificano i dati, l'immagine nel Geomview riflette i cambiamenti.

I programmi generatori di oggetti che usano il Geomview per visualizzarli sono chiamati *moduli esterni*. I moduli esterni possono controllare quasi tutti gli aspetti del Geomview. L'idea è che svariati aspetti della visualizzazione e delle parti interattive del software geometrico sono indipendenti dal contenuto geometrico e possono essere raccolti insieme in un singolo pezzo di software, che può essere usato in una grande varietà di situazioni.

L'autore del modulo esterno si puó quindi concentrare nell'implementazione dell'algoritmo desiderato e lasciare l'aspetto della visualizzazione al Geomview.

Il Geomview si presenta con una serie di semplici moduli esterni, e il manuale spiega come scriverne uno nuovo. Il modulo Animator, ad esempio, consente di creare l'animazione di una sequenza di immagini.

Il Geomview consente di controllare indipendentemente più oggetti e "telecamere". Fornisce il controllo interattivo per:

- il movimento;
- l'aspetto (incluse illuminazione, ombreggiatura e materiale);
- la selezione di un oggetto, una faccia o un vertice;
- catturare "istantanee" in diversi formati, Postscript o Renderman RIB;
- aggiungere o eliminare oggetti.

Tutto questo è possibile realizzarlo sia attraverso l'uso diretto del mouse, sia tramite pannello di controllo, sia utilizzando degli shortcuts[1] da tastiera.

Nel Geomview possono essere caricati oggetti di tipo OOGL, che sta per Object Oriented Graphics Library; questa libreria è quella sin cui è scritto il Geomview stesso.

Il Geomview può essere usato come dispositivo di output per i grafici creati con il Mathematica o con il Maple; questo consente di vedere i suddetti grafici in maniera molto più interattiva.

3.2.2 JavaView

Il *JavaView* è composto da un visualizzatore 3D e da una libreria software numerica scritta in Java. JavaView consente di aggiungere figure 3D interattive in un qualsiasi documento HTML e di eseguire esperimenti online.

Il JavaView è stato sviluppato per risolvere i seguenti compiti:

- visualizzazione di geometrie in 3D ed esperimenti numerici online;
- pubblicazione di esperimenti matematici su giornali elettronici online;
- sviluppo di visualizzazione ed algoritmi numerici in una libreria "open class";
- specifici algoritmi e formati di files per preparare i modelli geometrici alla pubblicazione online;
- semplice integrazione in altri software attraverso le API del JavaView.

[1] Combinazioni di tasti che producono un effetto. Letteralmente "scorciatoie".

La prima versione del JavaView è stata rilasciata nel Novembre del 1999, dopo oltre un anno di utilizzo in progetti di ricerca, dalla Technische Universität di Berlino.

Il modo più semplice di usare il JavaView è quello di visualizzare online dei modelli geometrici interattivi, mentre la risoluzione matematica di un problema attraverso un esperimento intero può costituire un applet più complesso. JavaView fornisce molti *tools*[2] ed algoritmi accessibili dal menu per la creazione di propri esperimenti matematici.

L'interfaccia del JavaView è composta da un visualizzatore 3D e da vari pannelli di controllo per esaminare l'oggetto geometrico, modificarne le caratteristiche del materiale, cambiare le impostazioni di visualizzazione e della telecamera, o guidare un'animazione.

JavaView si presenta con una serie di applet per mostrare dei modelli di geometrie già calcolati, oppure per svolgere degli esperimenti matematici, come ad esempio la risoluzione di una equazione differenziale ordinaria.

Il JavaView opera in 2D, 3D, e dimensioni superiori, ed implementa algoritmi raffinati da geometria differenziale discreta.

Per elaborare i file, il JavaView usa un proprio formato chiamato JVX. Il formato JVX è basato sul linguaggio XML, un nuovo tipo di linguaggio usato per vari tipi di dati in internet. Il formato JVX è usato anche per scambiare dati tra il JavaView ed altri software, come il Maple e Polymake. Ogni file JVX può essere equipaggiato opzionalmente con informazioni sull'autore o con informazioni dettagliate circa le proprietà matematiche del modello.

[2]Programmi di utilità

Attualmente, JavaView utilizza le seguenti estensioni per i propri file, ognuna delle quali è basata sul linguaggio XML:

JVX formato dei file che individuano geometrie e scene, anche con geometrie multiple.

JVR configurazione dei dati del JavaView stesso, come tipo di linguaggio, posizione delle finestre e contenuto dei menu.

JVD impostazioni di visualizzazione e posizionamento delle telecamere.

Il JavaView è in grado di lavorare anche con molti altri tipi di formati. Può importare ed esportare anche i formati BYU, FE, MGS, MPL, OBJ, OFF, STL, WRL (o VRML), e può solamente importare i file BD e DXF (alcuni di questi solo parzialmente).

Si possono salvare anche dei file immagine, in formato PS, EPS, GIF o PPM, inoltre, se il progetto attivo contiene un'animazione, questa può essere salvata come GIF o PPM.

Un'importante caratteristica del JavaView è quella di potersi integrare come visualizzatore o come motore grafico in altri applicativi commerciali come Mathematica e Maple, con comunicazioni bidirezionali tra i due software. In particolare, la Wolfram Research (produttrice del Mathematica) fornisce un package denominato J/Link che non solo consente al JavaView di visualizzare i grafici creati col Mathematica, ma dà anche la possibilità di utilizzare il Mathematica come motore di calcolo per eventi catturati dal JavaView. Ad esempio, selezionare un vertice di una figura geometrica visualizzata nel display

del JavaView può essere usato come comando di input per dei calcoli con il Mathematica.

3.2.3 KnotPlot

Il *KnotPlot* è un programma per la visualizzazione interattiva e l'elaborazione di nodi in 3D e 4D. È stato ideato dal canadese Robert Glenn Scharein con l'utilizzo delle librerie grafiche OpenGL e del VisualC++, e dalle sue caratteristiche si intuisce un particolare interesse dell'autore alla grafica computerizzata. Il programma gira su molti tipi di sistemi operativi: Windows 95/98/NT/ME/2000/XP, Linux, Silicon Graphics IRIX, SunOS (Solaris) e MacOSX.

Il KnotPlot fornisce molti nodi già pronti da caricare, catalogati secondo il libro di Rolfsen *Knots and Links*. Alternativamente, i nodi possono essere creati dall'utente, secondo il sistema di notazione sviluppato da Conway (attraverso il "tangle calculator"). Si possono costruire nodi, catene, nodi torici, nodi di Lissajous, ma anche grafi, stringhe aperte, intrecci e nodi "cablati".

Il nodo può essere disegnato anche a mano libera con l'aiuto del mouse, usando alternativamente il tasto destro o il sinistro per indicare gli incroci superiori o inferiori.

I nodi possono essere trasformati in altri nodi e manipolati.

Il KnotPlot dà la possibilità di "rilassare" un nodo, tramite l'applicazione di un sistema di forze.

KnotPlot può esportare le superfici in diversi formati, inclusi OBJ, AutoCAD DXF, POVRay e METAFONT. Inoltre si possono estrarre immagini in diversi formati Postscript (EPS, PPM, RAW).

Dei programmi esistenti e analizzati possiamo dire senza ombra di dubbio che il KnotPlot è quello che produce graficamente il risultato più soddisfacente, ed è per questo che gran parte delle figure della presente tesi sono state ottenute con tale software.

3.2.4 SnapPea

Lo *SnapPea* è un programma nato per la creazione e lo studio delle 3-varietà iperboliche; successivamente sono state introdotte alcune funzionalità per lo studio dei nodi. E' stato ideato e realizzato da Jeff Weeks per i sistemi Unix, poi adattato da altre persone anche per Windows e MacOS9. Il nucleo del programma è scritto in linguaggio C, ed è gratuitamente disponibile e compilabile su tutte le piattaforme.

L'interfaccia piuttosto essenziale consente di disegnare il diagramma di un nodo con l'aiuto del mouse, specificando i soprapassaggi e i sottopassaggi, inserendo anche più componenti. Principalmente, però, lo SnapPea si distingue per i numerosi invarianti che è in grado di calcolare, come il dominio di Dirichlet, la caratteristica di Eulero, la chiralità, il gruppo fondamentale del nodo associato alla 3-varietà, etc...

Per quanto riguarda strettamente la teoria dei nodi, risulta poco pratico e difficile da utilizzare.

3.2.5 Knotscape

Il *Knotscape* è un programma interattivo per lo studio dei nodi, ideato nel gennaio del 1999 da due matematici molto famosi proprio per il loro contributo alla teoria dei nodi: Jim Hoste e Morwen Thistlethwaite. Il programma è

stato progettato in origine per consentire un facile accesso alle tabelle dei nodi, ma ben presto il relativo scopo si è espanso per includere il calcolo di molti invarianti del nodo. Gli utenti possono inserire un nodo in molte maniere diverse: come una treccia[3], oppure attraverso il codice di Dowker-Thistlethwaite (o quello di Gauss), o selezionandolo dalle tabelle[4] dei nodi, oppure ancora disegnandolo con un mouse.

Una volta che il nodo è stato inserito nel Knotscape, è possibile inoltre:

- trovare il nodo nelle tabelle se è un nodo primo (per nodi aventi fino a 16 incroci), altrimenti i nodi primi che lo compongono nel caso di somma connessa;
- disegnarne un'immagine piacevole;
- calcolare i suoi polinomi di HOMFLY, di Kauffman, di Jones, o di Alexander;
- calcolare le rappresentazioni del relativo gruppo fondamentale nel gruppo simmetrico su cinque lettere;
- calcolare alcuni invarianti iperbolici, grazie all'inserimento di una parte del codice dello SnapPea.

Gli eseguibili di questo programma sono disponibili solamente per Linux, Macintosh e Solaris.

[3]Secondo la relativa *Teoria delle trecce*, inventata nel 1925 da Emil Artin [Art47]
[4]Secondo la classificazione di Rolfsen, reperibile nell'appendice di [Rol76]

Capitolo 4

Il mio programma: KnotExplorer

Il risultato perseguito in questa tesi è la realizzazione di un software che rappresenta graficamente i nodi e ne agevola la comprensione e lo studio.

Nel fare questo si è posta particolarmente l'attenzione nella scelta di:

- un linguaggio di programmazione idoneo per la maggior parte dei sistemi operativi;
- una codifica che fosse allo stesso tempo completa e semplice e che consentisse di realizzare facilmente procedure efficienti;
- un funzionale di energia la cui minimizzazione consentisse al nodo di disporsi in una configurazione ottimale.

In questo capitolo illustreremo le parti essenziali del programma realizzato, il suo funzionamento e giustificheremo le scelte effettuate durante la sua creazione.

4.1 Scelte preliminari

Per realizzare un programma, il primo passo da fare consiste nella scelta dell'"ambiente di sviluppo". Questa scelta deve essere ben ponderata, altrimenti si corre il rischio di lasciare fuori dalla possibilità di utilizzare il programma buona parte di potenziali utenti, tra matematici e ricercatori.

Infatti, l'attuale panorama informatico è costituito da diversi sistemi operativi, tra i quali Windows è di gran lunga il più utilizzato specialmente dagli utenti comuni di PC, mentre SunOs, Linux e Silicon Graphics sono preferiti dai ricercatori matematici.

Per non escludere nessun tipo di utente abbiamo ritenuto conveniente utilizzare un linguaggio ad oggetti di ultima generazione: il Java, che con la sua "Java Virtual Machine" crea un ambiente di sviluppo virtuale in grado di funzionare su un qualsiasi sistema operativo.

Questa scelta è altresì sostenuta dal fatto che Java è attualmente il linguaggio più diffuso nel campo della ricerca poichè fornisce tutti gli strumenti ad essa adatti, inoltre la sua caratteristica di essere multipiattaforma consente un più facile scambio di "librerie" e programmi, cosicchè i progetti con esso creati possono essere resi disponibili a chiunque.

4.2 Codifica

In vista della creazione del "nostro" software abbiamo deciso di adottare due tipi di codifica, entrambi utilizzabili per l'inserimento dei dati, ma solamente uno si articola all'interno del programma come chiave di funzionamento delle

procedure. Per una questione di praticità chiameremo quest'ultima *codifica interna* e l'altra *codifica esterna*.

4.2.1 Codifica Interna

La codifica interna trae origine dal numero degli incroci e degli archi (vedi definizioni a pag.16) che possono essere individuati in un diagramma orientato di uno specifico nodo e dalla loro combinazione. In particolare, ciò che costituisce oggetto della codifica non è propriamente il nodo, bensì il determinato diagramma preso in esame.

A questo punto possiamo chiarire come avviene la codifica: prendendo in considerazione il diagramma da codificare è necessario dapprima individuarne tutti gli archi ed associare a ciascuno di essi un'etichetta differente (ad esempio un nome, un numero o una lettera), poi considerare tutti gli incroci e per ciascuno di essi inserire in una tabella cinque informazioni che determinano univocamente l'incrocio: le prime quattro relative agli archi che vi convergono, l'ultima invece è un numero che identifica l'orientazione dell'incrocio sul piano.

Per capire meglio come vanno inserite le informazioni è bene prendere in esame un singolo incrocio. In esso confluiscono quattro archi orientati: due di essi passeranno sopra agli altri due, e per ciascuna coppia ci sarà un arco che entra e uno che esce. Una volta che si è scelto un incrocio, agli archi interessati viene attribuito un identificativo locale, in questo modo (vedi figura 4.1):

overIncomingArc arco (arc) che passa sopra (over) e che entra (incoming) nell'incrocio;

overOutgoingArc arco che passa sopra ed esce (outgoing) dall'incrocio;

underIncomingArc arco che passa sotto (under) ed entra nell'incrocio;

underOutgoingArc arco che passa sotto ed esce dall'incrocio.

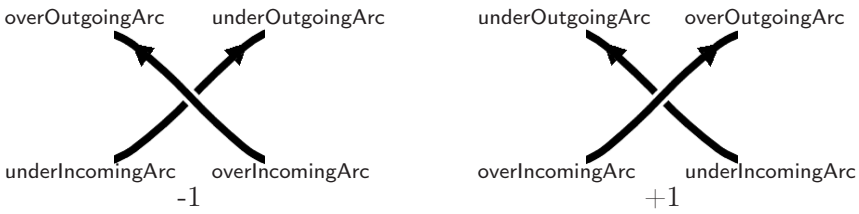


Figura 4.1: Gli indici degli archi sui due tipi di incroci orientati.

Questi identificativi sono necessari per rilevare le prime quattro informazioni relative agli archi, ognuna delle quali è costituita dall'etichetta dell'arco corrispondente all'identificativo locale. Per svolgere questa operazione è importante seguire rigorosamente l'ordine delineato precedentemente nell'elencare gli identificativi. Come si può capire dalla figura 4.1, per dare la giusta identità all'arco le componenti dovranno essere orientate.

Infine, la quinta informazione sarà costituita dal segno dell'incrocio che potrà essere ± 1 come già spiegato a pag.30.

Per chiarirci le idee, vediamo un esempio pratico: proviamo a codificare il nodo in figura 4.2. La prima cosa da fare è quella di dare un'orientazione

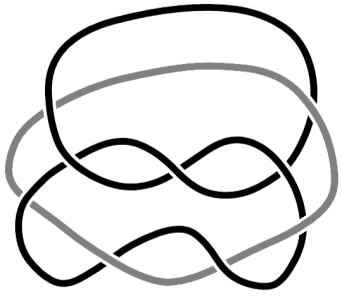


Figura 4.2: Il diagramma di un nodo da codificare.

alle componenti del nodo: questo ci permetterà in seguito di stabilire correttamente le identità relative degli archi e i segni degli degli incroci. Scegliamo indifferentemente una delle due possibili direzioni di percorrenza per ciascuna componente; la scelta non comprometterà la buona riuscita della codifica, a patto che si mantenga sempre la stessa orientazione durante tutta l'operazione. Decidiamo di adottare le direzioni stabilite dalle frecce come in figura 4.3.

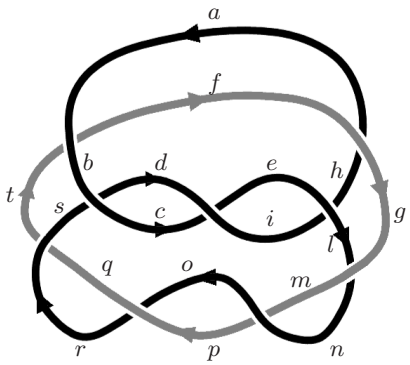
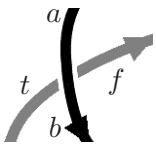


Figura 4.3: Il diagramma orientato ed etichettato.



a	b	t	f	$+1$

Figura 4.4: Un incrocio del diagramma e la sua codifica.

Osservando accuratamente il diagramma possiamo vedere che è costituito da 18 archi, ad ognuno dei quali dobbiamo attribuire una specifica etichetta; così, se utilizziamo ad esempio le lettere dell'alfabeto, uno di essi sarà a, un altro b, e così via, fino ad ottenere quello mostrato in figura 4.3. Dopo aver etichettato gli archi prendiamo in considerazione gli incroci, facendo attenzione non all'ordine, ma semplicemente a non tralasciarne nessuno. Esaminiamo quindi l'incrocio ingrandito in figura 4.4: la sua codifica sarà composta, come già detto, dalle etichette dei quattro archi confluenti, secondo l'ordine overIncomingArc, overOutgoingArc, underIncomingArc, underOutgoingArc, e dal

segno dell'incrocio stesso; in questo modo si avrà la seguente tabella parziale:

a	b	t	f	$+1$

. Una volta che tale procedura sarà stata eseguita anche per gli incroci rimanenti, otterremo una tabella completa, risultato della codifica interna del diagramma (tabella 4.1).

Sopra		Sotto		Segno
In	Out	In	Out	
a	b	t	f	$+1$
f	g	h	a	$+1$
e	l	i	h	$+1$
d	i	c	e	$+1$
b	c	s	d	$+1$
r	s	q	t	$+1$
p	q	o	r	$+1$
n	o	m	p	$+1$
g	m	l	n	$+1$

Tabella 4.1: La codifica interna del diagramma in figura 4.2.

Ora che abbiamo capito come funziona la codifica sorge spontaneo domandarsi se questa sia completa o meno, ovvero se ci sia biunivocità tra l'insieme dei diagrammi e l'insieme delle codifiche specifiche. Dimostreremo che la relazione vale sull'insieme delle codifiche specifiche *ammissibili*, cioè quelle codifiche per le quali esiste un diagramma associato. Osserviamo che una particolarità fondamentale di queste mappe è che nella tabella associata ciascun arco comparirà solamente due volte, una come entrante e una come uscente (è indifferente che passi sopra o sotto).

Teorema 4.1. *Tra l'insieme dei diagrammi dei nodi, a meno di isotopie nel piano, e quello delle codifiche specifiche interne ammissibili, sussiste una relazione di biunivocità.*

Dimostrazione: dato un diagramma, l'esistenza ed unicità della codifica

specifica deriva da come essa è stata definita. È interessante, invece, vedere l'inverso.

Supponiamo di avere una codifica specifica ammissibile: quello che ci domandiamo è se risulti sempre possibile ricavare un diagramma associato e se questo diagramma sia unico. Proviamo ad ottenere il diagramma: dalla codifica possiamo disegnare separatamente ciascun incrocio, fino ad ottenere una situazione ben rappresentata dalla figura 4.5.

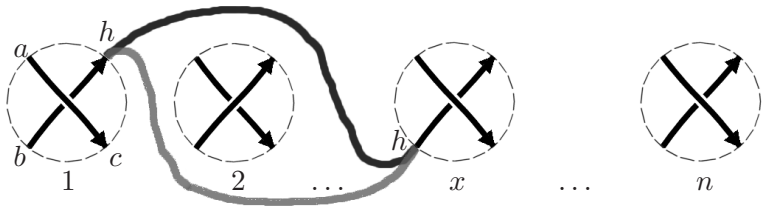


Figura 4.5: Gli incroci della codifica messi in sequenza.

Voglio portare l'incrocio x vicino all'incrocio 1, per collegare l'arco h senza "intoppi". Posso sempre farlo, seguendo indifferentemente il percorso nero o quello grigio che risulteranno omotopi. Con questo procedimento posso arrivare facilmente a congiungere tutti gli incroci, che andranno a comporre un grafo ad albero con alcune estremità libere che rappresentano gli archi da connettere, come in figura 4.6.

È chiaro che potrò formare più tipi di alberi, a seconda dell'incrocio che scelgo come iniziale e della direzione in cui mi muovo ogniqualvolta raggiungo un certo incrocio; tuttavia, una volta deciso quale arco utilizzare per "uscire" dall'intersezione, la scelta dell'incrocio successivo sarà univocamente determinata. Ad esempio, tornando alla figura 4.5, si è usciti dall'incrocio 1 attraverso

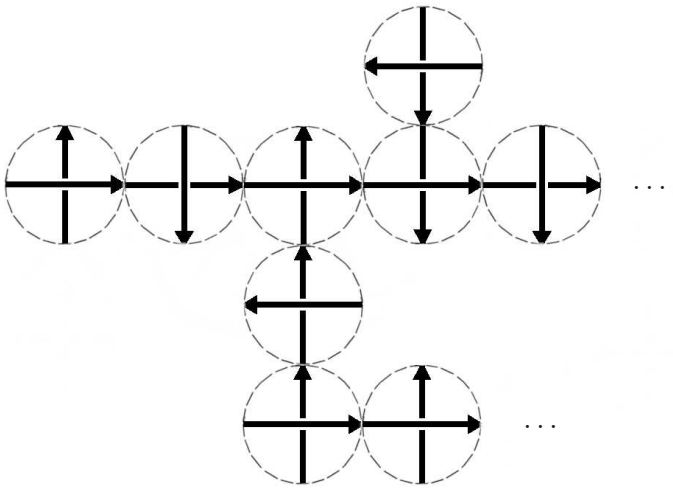


Figura 4.6: Gli incroci della codifica collegati fino a formare un grafo ad albero.

l'arco h ed a questo andrà unito l'*unico* incrocio che possiede lo stesso arco h come entrante.

A questo punto rimane da collegare gli archi esterni e, per addentrarci meglio nel problema, operiamo questa costruzione: circondiamo l'albero con una curva chiusa, in maniera da farle intersecare tutti i semiarchi esterni, come in figura 4.7. Possiamo pensare di "rigirare" il contenuto della curva fino a portar fuori tutto quello che era dentro, e dentro tutto quello che era fuori, una specie dell'operazione che si fa quando si rigira un guanto! Questa operazione viene chiamata solitamente *cambio del punto all'infinito*, poichè il "punto", che prima consideravamo all'infinito, diventa ora interno alla curva, mentre un punto che precedentemente all'operazione era interno alla curva, ora diventerà il nuovo infinito. Topologicamente, il problema di connettere i semiarchi esternamente alla curva nel primo caso è equivalente al problema di collegare i semiarchi internamente alla curva nel secondo. Lavoreremo dunque

in questa ultima condizione.

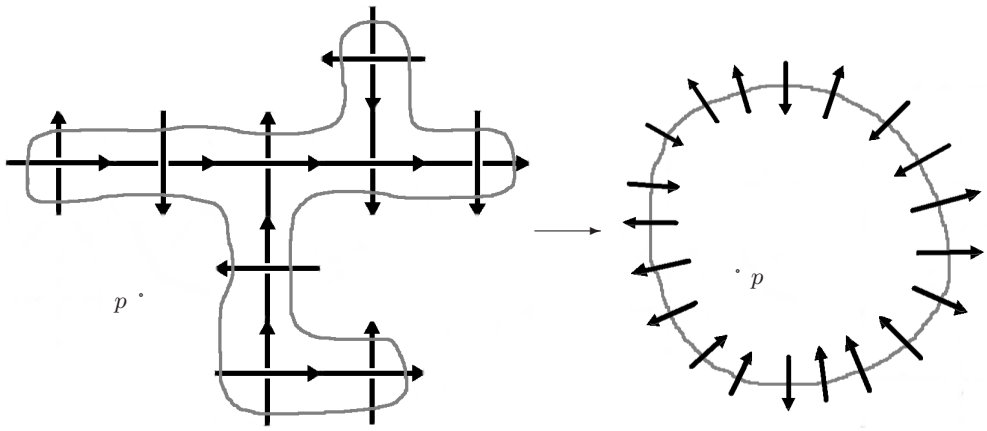


Figura 4.7: L'albero viene circondato da una curva, che poi viene invertita in modo che gli archi non devono più essere collegati esternamente, ma internamente. Il punto p, che prima era esterno alla curva, a seguito dell'inversione diventa un punto interno.

Rimodelliamo la curva fino a farla diventare simile ad una circonferenza (questo solamente per una più chiara analisi) e proviamo ad unire le coppie di archi. A questo punto, a meno di isotopie del piano, c'è un solo modo di congiungere gli archi, dato che sono tutte coppie, quindi si potranno verificare due casi:

1. si creano nuove intersezioni;

2. non si creano nuove intersezioni.

Il primo caso non lo possiamo accettare, perchè vorrebbe dire che il diagramma non è stato codificato in maniera corretta. Infatti, provando a estrarre la codifica dal diagramma che abbiamo ottenuto dovremmo tenere conto dei nuovi incroci che si sono formati, andando a cambiare di fatto la tabella.

Il secondo caso, invece, è quello che ci auguriamo, ed è proprio quello che

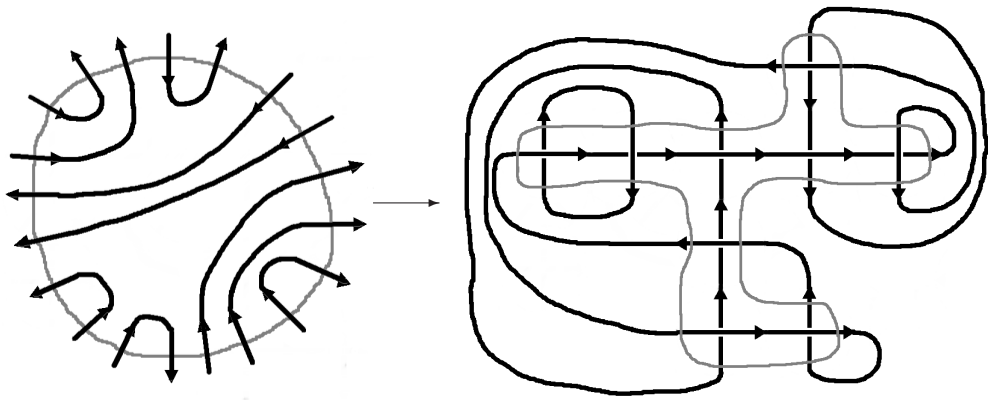


Figura 4.8: Gi archi sono stati collegati internamente, poi la curva è stata nuovamente invertita.

si verifica dal momento in cui andiamo a prendere in considerazione esclusivamente le codifiche specifiche ammissibili. □

4.2.2 Codifica Esterna

Per l'interfaccia con l'utente è stata utilizzata una codifica esterna, che ha come caratteristica principale quella di essere più semplice da applicare rispetto alla codifica interna, poichè necessita di sole quattro informazioni per ciascun incrocio.

Se per la codifica interna si usavano gli archi, per questa codifica si usano i ponti. Ricordiamo che un ponte è un pezzo di corda compreso tra due sottopassaggi consecutivi, inoltre facciamo notare che gli archi in numero sono il doppio degli incroci, mentre i ponti saranno tanti quanti gli incroci stessi.

Detto questo, il primo passo da fare è quello di individuare i ponti sul diagramma ed etichettarli, con lo stesso procedimento usato per gli archi nella

codifica interna. Al passo successivo bisogna prendere in esame ciascun incrocio e codificarlo; con una notazione simile alla codifica precedente possiamo identificare (figura 4.9):

overBridge ponte (bridge) che passa sopra (over);

underIncomingBridge ponte che passa sotto (under) ed entra (incoming) nell'incrocio;

underOutgoingBridge ponte che passa sotto ed esce (outgoing) dall'incrocio;

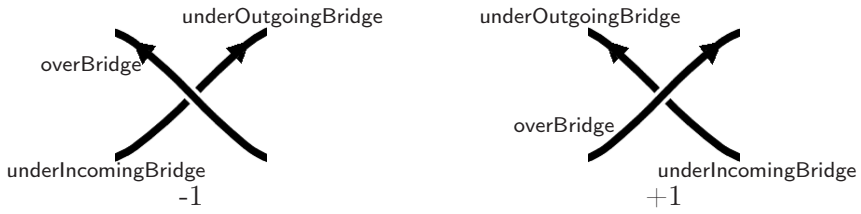


Figura 4.9: Gli indici dei ponti sui due tipi di incroci orientati.

e questo sarà anche l'ordine in cui andranno inseriti i dati nella tabella parziale relativa all'incrocio. La quarta informazione sarà, come per l'altra codifica, il segno dell'incrocio. Pertanto, in un esempio pratico, se supponiamo di voler codificare nuovamente il diagramma della figura 4.2, dopo aver etichettato i ponti (in maniera tale da ottenere la situazione in figura 4.10) la codifica del singolo incrocio ingrandito in figura 4.11 sarà $\boxed{a \mid h \mid b \mid +1}$. Si può vedere che i ponti coinvolti sono stati inseriti, come già detto, secondo l'ordine overBridge, underIncomingBridge, underOutgoingBridge, mentre +1 è il segno dell'incrocio. Ripetendo l'operazione su tutti gli incroci si otterrà la tabella

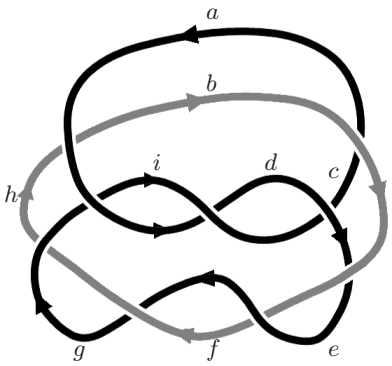
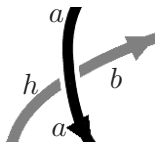


Figura 4.10: Il diagramma orientato ed etichettato con i ponti.



a	h	b	$+1$

Figura 4.11: Un incrocio del diagramma e la sua codifica.

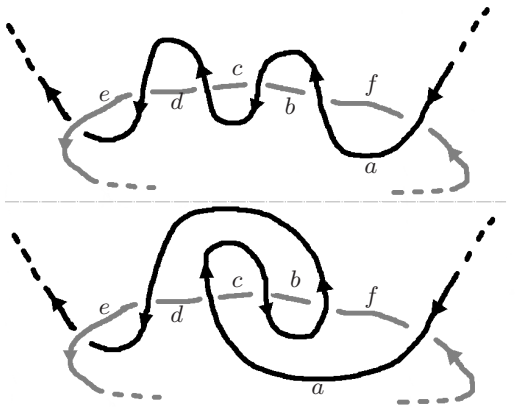
4.2 a meno dell'ordine delle righe: infatti, anche in questa codifica, come per la precedente, non conta l'ordine con cui vengono considerati gli incroci, ma solamente che vengano esaminati tutti.

Sopra	Sotto		Segno
	In	Out	
a	h	b	$+1$
b	c	a	$+1$
a	g	i	$+1$
i	a	d	$+1$
d	i	c	$+1$
g	f	h	$+1$
f	e	g	$+1$
e	b	f	$+1$
b	d	e	$+1$

Tabella 4.2: La codifica esterna del diagramma in figura 4.10.

Vogliamo ora capire se anche per questa codifica vale la biunivocità tra l'insieme dei diagrammi e l'insieme delle codifiche specifiche. Questa volta, la risposta è negativa anche se prendiamo in considerazione l'insieme delle codifiche specifiche ammissibili, come fatto per la codifica interna, ma per tornare

ad avere biunivocità dobbiamo operare un'ulteriore restrizione sull'insieme dei diagrammi: tra di questi, infatti, ce ne saranno alcuni ai quali è associata la stessa codifica esterna (pur avendo codifiche interne differenti). Questi diagrammi saranno comunque rappresentazioni diverse dello stesso nodo, quindi il fatto che la codifica esterna non li distingua non è da considerarsi un problema, dato che il nostro scopo iniziale era proprio quello di identificare il nodo e non il singolo diagramma. Per capire bene dove sia il problema, osserviamo la figura 4.12: le due parti di diagramma sono evidentemente differenti, tuttavia,



...			
a	f	b	$+1$
a	b	c	-1
a	c	d	$+1$
a	d	e	-1
...			

Figura 4.12: Due diagrammi diversi con stessa codifica esterna (parziale).

in entrambi i casi, il ponte a passa tutto sopra agli altri tratti di corda (anche perchè altrimenti cambierebbe nome), così possiamo facilmente affermare che i due nodi rappresentati sono isotopi in $\mathbb{R}^3$.

Per avere un insieme dei diagrammi che sia biunivoco con le codifiche esterne specifiche, basterà dunque quozientare, nel primo insieme, tutti quei diagrammi aventi soprapassaggi plurimi e tra di loro isotopi in $\mathbb{R}^3$.

Teorema 4.2. *Tra l'insieme dei diagrammi dei nodi, a meno di isotopie nel piano e quozientati secondo la relazione spiegata sopra, e l'insieme delle codifiche specifiche esterne ammissibili, sussiste una relazione di biunivocità.*

Dimostrazione

Per dimostrare questa biunivocità, possiamo seguire gli stessi passi della dimostrazione del teorema 4.1, modificando i passi non più validi in seguito alla modifica delle ipotesi.

Dato un diagramma, ne esiste sicuramente un'unica codifica specifica; questo è abbastanza intuitivo.

Supponiamo, viceversa, di avere una codifica specifica esterna ammissibile e proviamo ad ottenerne il diagramma corrispondente. Se ci rifacciamo al teorema 4.1, la dimostrazione, in questo caso, fallisce in due passi: il primo al momento della creazione del grafo ad albero, e il secondo quando dobbiamo collegare i semiarchi esterni. Infatti, in quei due interventi, punto forte della dimostrazione era che il modo di collegare gli archi fosse unico, cosa che non è più vera quando trattiamo con i ponti. Proviamo ad esempio a comporre l'albero partendo da un incrocio e da un ponte qualsiasi: a questo punto dobbiamo cercare un altro incrocio da attaccare al primo, che contenga anch'esso il ponte di partenza. Nel caso che il suddetto ponte compia dei soprapassaggi, avrò più di una scelta su quale incrocio avvicinare al primo, scelta che potrebbe compromettere il risultato finale. Infatti, una volta ottenuto l'albero, dovrò collegare i ponti esterni senza creare nuove intersezioni, il che risulterà impossibile per alcune composizioni dell'albero, quindi dovremo ritornare al momento della sua formazione e cambiare scelta. Di certo invece, per quanto detto precedentemente, una volta che avrò ottenuto un diagramma corretto,

esso rappresenterà perfettamente il nodo.

Corollario 4.1. *Il tipo topologico del nodo è univocamente determinato.*

Questo è garantito dalle scelte fatte nelle ipotesi del teorema: prendere in considerazione le codifiche specifiche ammissibili ci assicura che esiste un diagramma di un nodo associato; inoltre, i possibili diagrammi associati differiscono solamente per i modi in cui un ponte con più soprapassaggi può essere "ricombinato" (vedi figura 4.12), modi che sono tutti isotopi tra loro in $\mathbb{R}^3$. Così, possiamo affermare che il tipo di isotopia del nodo è univocamente determinata.

4.3 Immissione dei dati

Una volta avviato il KnotExplorer, il primo passo da fare è quello di caricare i dati relativi alla codifica che abbiamo utilizzato per identificare il nodo.

I dati vanno inseriti attraverso un normale file di testo, composto mettendo la codifica di un incrocio per ogni riga, dove gli archi (o i ponti) vanno separati da uno spazio, e il numero rappresentante il segno dell'incrocio deve essere 1 se è positivo (con il segno + davanti non funziona!) e -1 se negativo.

Il programma procede innanzitutto all'interpretazione del file: riconosce la codifica come interna o esterna, poi la interpreta istanziando gli incroci e gli archi (o i ponti) necessari e costituendo, contemporaneamente, i loro legami. In seguito, se la codifica interpretata è interna il KnotExplorer esegue direttamente la rappresentazione grafica del nodo, se invece è esterna deve prima effettuare la conversione da una codifica all'altra.

4.4 Conversione dalla codifica esterna alla codifica interna

Si tratta di individuare gli archi dall'insieme dei ponti della codifica esterna, per ricavare la codifica interna degli incroci. Ciascun ponte, infatti, è costituito da uno o più archi, a seconda di quanti soprapassaggi compie: dobbiamo suddividere il ponte negli archi che lo compongono, e per ciascun arco dobbiamo capire da quale incrocio esce ed in quale entra, inoltre se passa sopra o sotto. Questo è possibile attraverso un procedimento suddivisibile in diverse fasi e vedremo che la difficoltà sta nello stabilire l'ordine corretto di successione degli incroci aventi lo stesso ponte come soprapassante.

La prima fase è quella che ci predispone a comporre la codifica interna: prepariamo una "tabella di corrispondenza", cioè una tabella che abbia tante righe quanti sono gli incroci della codifica esterna e cinque colonne, che accoglieranno i valori della codifica interna. La chiamiamo "di corrispondenza" perchè facciamo in modo che ad ogni incrocio corrisponda la stessa riga nelle due codifiche.

A questo punto possiamo già riempire la 5ª colonna della tabella della codifica interna con gli stessi valori di quelli della 4ª colonna della codifica esterna, dove sono contenuti i segni degli incroci, i quali non cambiano con la conversione. Per avere un'idea ben chiara facciamo riferimento alla tabella 4.2, contenente la codifica esterna del nodo in figura 4.10, e proviamo a ricavare una codifica interna corrispondente operando la conversione. La tabella di corrispondenza sarà composta da nove righe, pari al numero degli incroci, e la quinta colonna andrà riempita con tutti segni positivi.

La seconda fase consiste nel convertire i ponti più "corti": sono quelli che non compiono neppure un soprapassaggio e sono presenti nella intera tabella della codifica esterna solamente due volte, una come ponte entrante (UIB[1]) ed una come uscente (UOB). Ciascuno di questi ponti è sostituito interamente da un arco nella codifica interna, in corrispondenza degli stessi incroci, come arco sottopassante entrante (UIA) ed arco sottopassante uscente (UOA), rispettivamente. Possiamo quindi aggiungerli alla tabella.

Proseguendo con l'esempio precedente, nella tabella 4.2 di pagina 66 troviamo i ponti c ed h in questa condizione; possiamo chiamare gli archi correlati c_1 ed h_1, per distinguerli dai ponti, ed inserirli nella nuova tabella alle posizioni corrispondenti (tab. 4.3).

Nella fase successiva convertiamo i ponti che eseguono un solo soprapassaggio. Per ognuno di questi ponti è ancora facile stabilire l'ordine in cui sono attraversati gli incroci che lo contengono: dato che il primo è l'incrocio da cui esce e l'ultimo è quello in cui arriva, che passano entrambi sotto, l'unico incrocio di cui è soprapassante deve essere necessariamente intermedio tra i due.

A questo punto istanziamo due nuovi archi (possiamo chiamarli, ad esempio, con lo stesso nome del ponte associato, seguito dagli indici 1 e 2, rispettivamente) e li inseriamo nella codifica interna. Un arco prende il posto di UOA del primo incrocio e OIA dell'incrocio intermedio, e l'altro arco occupa il posto di OOA nell'incrocio intermedio e UIA dell'ultimo incrocio.

[1]D'ora in avanti useremo le seguenti abbreviazioni: UIB per underIncomingBridge, UOB per underOutgoingBridge, OB per overBridge; UIA per underIncomingArc, UOA per underOutgoingArc, OIA per overIncomingArc e OOA per overOutgoingArc. Per i significati si vedano le spiegazioni delle due codifiche ai paragrafi 4.1 e 4.2.2.

Sopra	Sotto		Segno
	In	Out	
a	h	b	$+1$
b	c	a	$+1$
a	g	i	$+1$
i	a	d	$+1$
d	i	c	$+1$
g	f	h	$+1$
f	e	g	$+1$
e	b	f	$+1$
b	d	e	$+1$

$\longrightarrow$

Sopra		Sotto		Segno
OIA	OOA	UIA	UOA	
		h_1		$+1$
		c_1		$+1$
		g_2	i_1	$+1$
i_1	i_2		d_1	$+1$
d_1	d_2	i_2	c_1	$+1$
g_1	g_2	f_2	h_1	$+1$
f_1	f_2	e_2	g_1	$+1$
e_1	e_2		f_1	$+1$
		d_2	e_1	$+1$

Tabella 4.3: Conversione della codifica esterna in tabella 4.2. Un passo intermedio.

E' questo il caso dei ponti d, e, f, g e i nell'esempio precedente, sostituiti dagli archi d_1, d_2, e_1, e_2, f_1, f_2, g_1, g_2, i_1 e i_2 nella tabella 4.3.

Nella quarta fase ci occupiamo di convertire tutti i ponti restanti, aventi quindi più di un soprapassaggio. Il difficile è trovare l'ordine giusto in cui vanno presi gli incroci che vedono il ponte passargli sopra, per poter assegnare, a ciascun arco che lo costituisce, un incrocio di partenza ed uno di arrivo che diano come risultato una codifica ammissibile; infatti, non tutti gli ordini danno un diagramma valido (vedi par. 4.2.2). Per fare questo abbiamo ideato un metodo che ci è piaciuto chiamare "delle superfici", dove per "superficie" intendiamo una parte del piano racchiusa dagli archi di un diagramma.

Il metodo delle superfici sostanzialmente consiste in questo: se prendiamo in considerazione un incrocio, possiamo notare che esso separa quattro superfici (non necessariamente distinte). Uscendo da questo incrocio attraverso uno qualsiasi degli archi, possiamo provare a "chiudere" una delle due superfici separate dall'arco, andando ripetutamente all'incrocio successivo e "girando" sempre da una stessa parte.

Se facciamo questa operazione partendo da una codifica ad archi, vediamo che è molto semplice chiudere una superficie, assegnato un incrocio di partenza ed un arco con cui uscire da esso: ogni volta prendo l'incrocio al capo opposto dell'arco, su di questo "giro" a sinistra o a destra, e ripeto l'operazione finchè non ritorno all'incrocio di partenza. Con i ponti, invece, non è altrettanto facile: quando esco da un incrocio attraverso un ponte, non c'è sempre un unico incrocio che può essere collegato all'altro capo del ponte stesso, ma posso avere più di una scelta (come si è intuito, questo accade quando ho più di un soprapassaggio). Non tutte le scelte fatte, però, portano alla chiusura della superficie, ma solo quelle che rendono la codifica ammissibile. Per accorgermi se non ho più speranza di chiudere la superficie, e quindi se l'incrocio che ho scelto come successivo non va in quella posizione, devo tenere conto di quali incroci incontro durante la procedura: quando ripasso da uno stesso incrocio, può essere o quello di partenza, o uno degli altri. Se non è quello di partenza, allora devo ritornare al momento della scelta ed effettuarne un'altra; se invece è quello di partenza, ho ancora due possibilità: o sono entrato nell'incrocio attraverso l'arco giusto e nella direzione giusta, allora la superficie si è chiusa e la procedura passa a tentare di chiudere un'altra superficie; oppure, una tra l'arco e la direzione di ingresso sono errate, e anche in questo caso devo ritornare al momento della scelta ed effettuarne un'altra. Quando ho chiuso tutte le superfici, allora il nodo è correttamente convertito: questo significa che conosco per ogni ponte la sequenza corretta degli incroci che attraversa e posso passare a completare la tabella. Per concludere l'operazione, devo istanziare $n+1$ archi per ciascun ponte avente n soprapassaggi, e poi assegnare ad ognuno

di essi due incroci, uno di partenza ed uno di arrivo, seguendo la sequenza.

Vediamo un esempio pratico per chiarirci le idee: in figura 4.13 mostriamo alcuni tentativi di chiudere la superficie compresa tra i ponti a e b, presi entrambi in direzione uscente, dell'incrocio $(a, h, b, +1)$, percorrendo il ponte b e scegliendo di girare a destra. Nei primi due disegni della figura, i tentativi

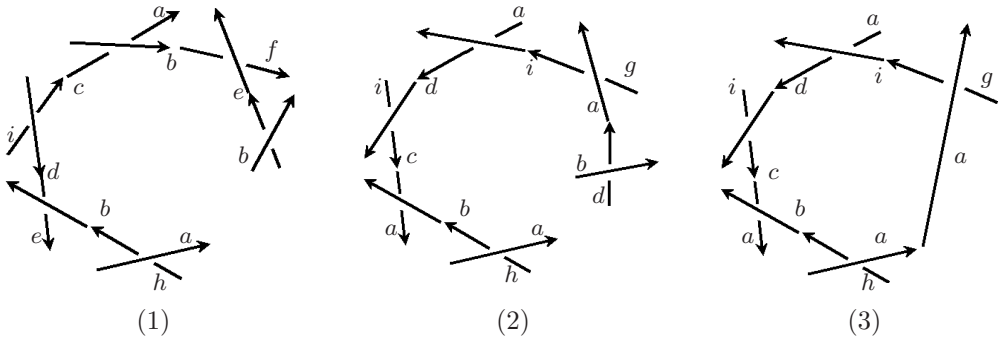


Figura 4.13: Alcuni tentativi di chiudere una superficie dell'incrocio $(a, h, b, +1)$. Nelle figure (1) e (2) il tentativo fallisce, mentre in (3) va a buon fine.

falliscono e dobbiamo ogni volta ritornare ad un ponte di cui non conosciamo la sequenza e cambiare la scelta effettuata. Anche se in figura vediamo tutti nomi di ponti, gli unici due non ancora convertiti sono solo a e b; per tutti gli altri, in realtà si è seguita la sequenza certa, espressa dagli archi nella tabella 4.3.

Per puro caso, questa sola superficie ci è già sufficiente per conoscere la sequenza finale sia del ponte a che del ponte b. Per il ponte a avremo: $(b, c, a, +1)$, $(a, h, b, +1)$, $(a, g, i, +1)$, $(i, a, d, +1)$; per il ponte b avremo: $(a, h, b, +1)$, $(b, c, a, +1)$, $(b, d, e, +1)$, $(e, b, f, +1)$. A questo punto possiamo istanziare tre archi per ciascuno di questi ponti, e completare la conversione (tab. 4.4).

Sopra	Sotto		Segno
	In	Out	
a	h	b	$+1$
b	c	a	$+1$
a	g	i	$+1$
i	a	d	$+1$
d	i	c	$+1$
g	f	h	$+1$
f	e	g	$+1$
e	b	f	$+1$
b	d	e	$+1$

$\longrightarrow$

Sopra		Sotto		Segno
OIA	OOA	UIA	UOA	
a_1	a_2	h_1	b_1	$+1$
b_1	b_2	c_1	a_1	$+1$
a_2	a_3	g_2	i_1	$+1$
i_1	i_2	a_3	d_1	$+1$
d_1	d_2	i_2	c_1	$+1$
g_1	g_2	f_2	h_1	$+1$
f_1	f_2	e_2	g_1	$+1$
e_1	e_2	b_3	f_1	$+1$
b_2	b_3	d_2	e_1	$+1$

Tabella 4.4: Conversione della codifica esterna in tabella 4.2. Un passo intermedio.

4.5 La decodifica: rappresentazione grafica iniziale del diagramma

Se proviamo a ricostruire il diagramma partendo dalla tabella di una delle codifiche precedenti, ci accorgiamo che è un'operazione non proprio semplice e che ci può portare a risultati apparentemente diversi; di seguito capiremo come e perchè questo accade.

Per affrontare la questione, vediamo come il nostro programma ricrea in video la figura iniziale del diagramma (chiariamo, per chi non avesse dimestichezza con la programmazione, che tale rappresentazione di partenza sarà ben lontana da quella che ci aspettiamo come finale, avendo conosciuto i diagrammi del capitolo 2). La procedura che determina le posizioni dei vari incroci sul piano costruisce, come primo passo, un grafo ad albero, come spiegato a pag. 61 (nella dimostrazione del teorema 4.1), a partire dalla codifica del diagramma, secondo la classica visita in profondità della teoria sui grafi. Nel dettaglio della procedura si può osservare che come radice dell'albero viene scelto il primo

incrocio della lista, si prosegue verso il successivo attraverso uno dei due archi in direzione uscente, poi si cerca sempre, se possibile, di "girare a sinistra", cioè tra gli archi che ci portano agli incroci non ancora presi in considerazione per completare l'albero, viene scelto sempre il primo a sinistra (indifferentemente che sia entrante o uscente). In verità, la struttura del nostro albero è lievemente più complessa dell'albero classico: mentre normalmente un vertice[2] padre può avere un numero qualsiasi di figli, tra i quali non c'è preferenza, il nostro vertice padre può avere da zero a tre figli (eccezion fatta par la radice che ne può avere anche quattro), tra i quali è importante la posizione; è per questo, e per una maggiore comodità nello sviluppo di quanto segue, che i discendenti di un certo vertice potranno avere la qualifica di figlio sinistro, figlio centrale o figlio destro (per la radice esiste anche il figlio posteriore).

Dall'albero "logico", costituito da padri e figli, si passa all'albero "pratico", assegnando a ciascun vertice una posizione sul piano. Immaginiamo quindi di avere a disposizione una griglia con coordinate unitarie e di piazzare su alcuni dei suoi punti i nostri vertici: questo modello rappresenta bene la nostra situazione, infatti ciascuno dei nostri incroci ha quattro archi uscenti e possiamo fare in modo che questi si orientino secondo le direzioni cardinali principali (Nord, Est, Sud, Ovest).

Si parte ancora una volta dalla radice dell'albero, alla quale viene assegnata la posizione (0,0), mentre al vertice successivo va la posizione (1,0). Da qui

[2]Nella terminologia corretta delle strutture gerarchiche, ogni elemento costitutivo dell'albero viene chiamato *nodo* e può assumere la qualifica di nodo *padre* o *figlio* a seconda della sua posizione all'interno della gerarchia (si pensi all'albero genealogico di una famiglia). In questa trattazione, utilizzando il termine "nodo" si potrebbe creare un'evidente confusione tra l'elemento dell'albero e l'oggetto del diagramma ed è per questo che preferiamo utilizzare il termine "*vertice*" per indicare il nodo dell'albero.

in avanti, ci muoveremo cercando di sistemare prima il figlio sinistro, poi il centrale, infine il destro (se presenti); dal punto attuale, il primo andrebbe a finire in (1,1), a sinistra *rispetto alla nostra direzione di provenienza*, il secondo in (2,0), lungo la direzione di provenienza, ed il terzo in (1,-1), alla nostra destra relativa (vedi figura 4.14).

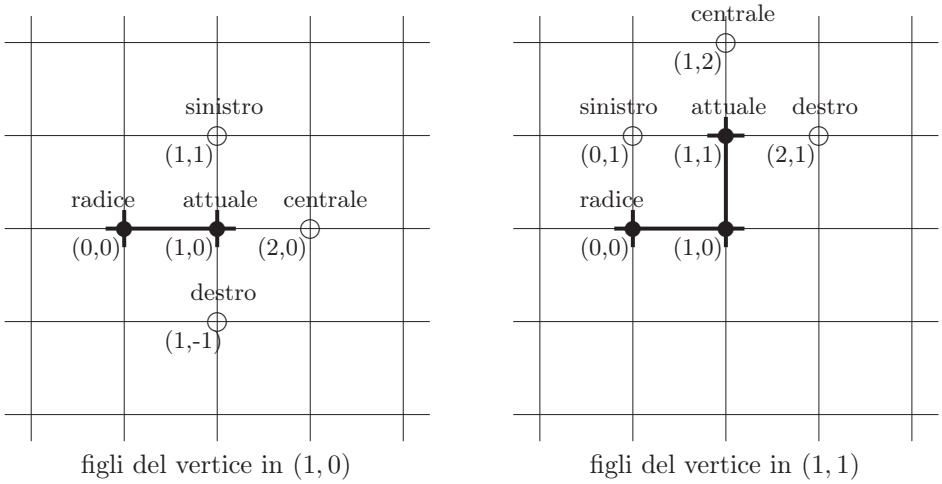


Figura 4.14: Posizionamento dei vertici dell'albero sulla griglia.

Proseguendo da (1,1), invece, i tre figli assumerebbero, nell'ordine: (0,1), (1,2), (2,1). Guardando la figura 4.15 si intuisce facilmente che un ulteriore figlio sinistro del vertice in posizione (0,1), nell'esempio sopracitato, si sovrapporrebbe alla radice del nostro albero. Per ovviare a questo inconveniente, prima di posizionare il nuovo vertice viene inserita una nuova riga (o colonna) nella griglia, così si è certi di avere il punto libero (in realtà viene operata una traslazione di tutti i punti interessati, con l'effetto di avere una riga libera nella griglia, proprio come se vi fosse stata inserita: nel nostro esempio vengono traslati verso il basso tutti i punti aventi ordinata minore o uguale a zero,

figura 4.15).

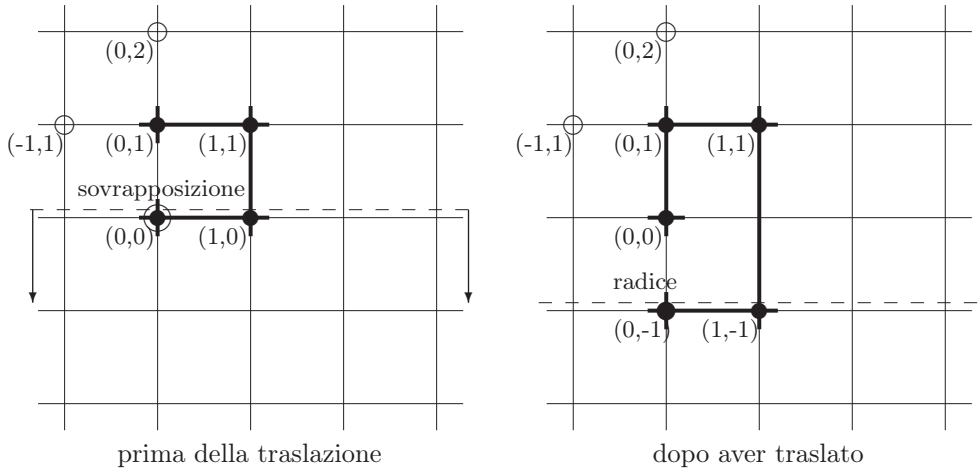


Figura 4.15: Per evitare possibili sovrapposizioni si effettua una traslazione all'inserimento di ogni nuovo punto.

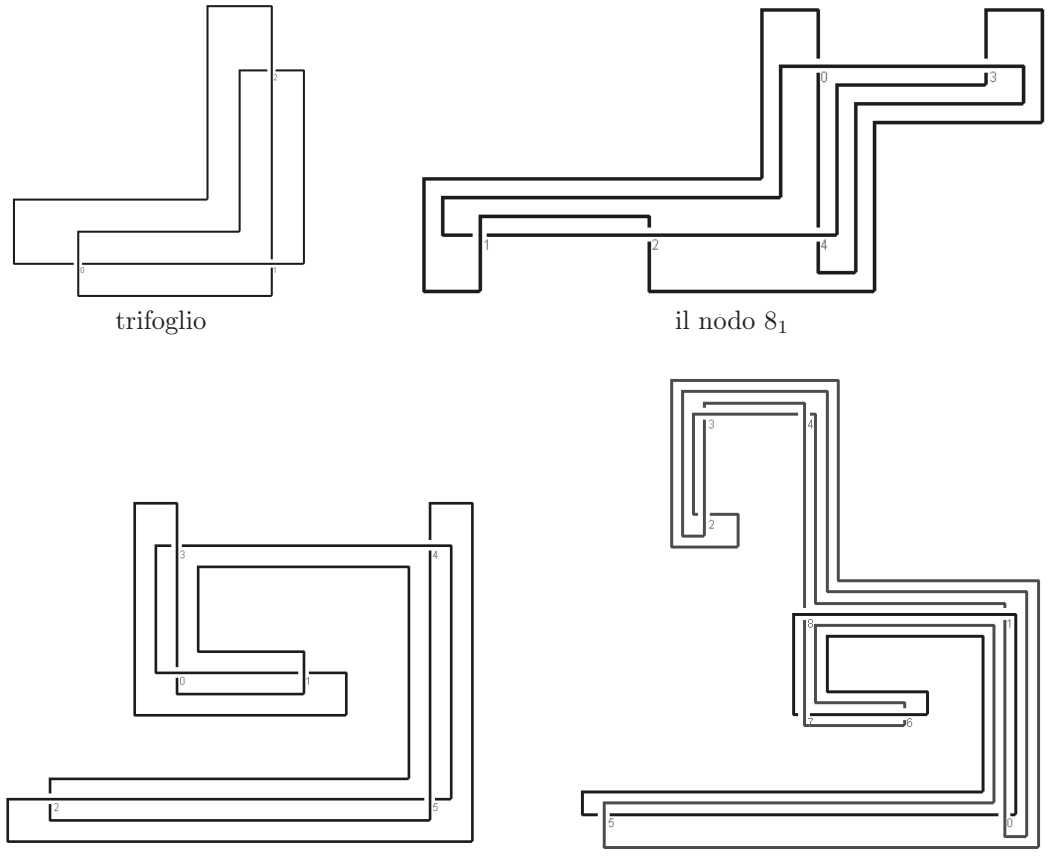
Di questo passo avremo ben presto disegnato tutto l'albero (e quindi tutti gli incroci) e per completare il diagramma dovremo collegare, nel modo giusto, gli archi liberi. Il problema, ora, è inserire nel grafico una linea che unisca le due parti di ciascun arco spezzato, senza incrociare le altre linee già presenti (nè quelle future!). L'idea è più o meno la stessa utilizzata per la dimostrazione del teorema 4.1: immaginiamo di circondare l'albero con una curva chiusa, in modo tale da intersecare le estremità libere degli incroci, ed inseriamo in una lista "circolare"[3] le etichette degli archi spezzati che incontriamo lungo il percorso. Nella lista, ciascun arco comparirà esattamente in due posizioni, che potranno essere consecutive o meno. Questa condizione ci consente di dire con certezza quando due semiarchi possono essere uniti senza intersecarne

[3]Per lista *circolare* intendiamo una lista come se fosse disposta su una circonferenza, dove cioè l'ultimo elemento viene idealmente "incollato" al primo in maniera che ciascun termine ne abbia uno che lo precede ed un altro che lo segue.

nessun altro: infatti, se nella lista ho le due etichette dello stesso arco inserite consecutivamente, allora le posso congiungere sul grafico senza disagio. Se ora le cancello dalla lista, posso controllare se si sono formate altre coppie adiacenti, unirle graficamente ed eliminarle dalla lista stessa, e così via finchè non ho completato tutte le connessioni. La presenza ad ogni passo di almeno una coppia di etichette adiacenti è garantita, indirettamente, sempre dal teorema 4.1 e si evince dalla dimostrazione (qualora non dovessero comparire coppie consecutive significherebbe che la codifica di partenza è errata). A livello di codice del programma, rimane comunque il problema di assegnare a ciascun arco i punti da percorrere graficamente per congiungere le due estremità libere. Questo viene eseguito facendo percorrere alla connessione lo stesso tragitto che compie l'albero per andare dall'incrocio di partenza dell'arco a quello di arrivo, parallelamente. In figura 4.16 ci sono alcuni esempi di diagrammi decodificati e rappresentati da un grafico iniziale.

Per ottenere un diagramma più "leggibile", che ci consenta di interpretare in pochi istanti il nodo corrispondente, bisognerà poi ottimizzare la configurazione del diagramma, applicando un funzionale di energia lungo la corda e minimizzandolo (parleremo di questo nel prossimo paragrafo 4.6).

All'inizio del paragrafo avevamo detto che la decodifica del nodo ci può portare a diagrammi apparentemente diversi. Ciò si verifica al momento della costruzione dell'albero, in base alla scelta del vertice che ne costituirà la radice. Cambiando vertice, infatti, al momento di congiungere gli archi esterni, l'algoritmo di ricostruzione del diagramma potrebbe trovare più "conveniente" far passare l'arco da una parte dell'albero piuttosto che dall'altra. Questo è



trifoglio

il nodo 8_1

il nodo della Nonna

un link più complesso

Figura 4.16: Alcuni nodi decodificati e rappresentati dal programma.

possibile perchè l'algoritmo dipende dalla lista delle etichette degli archi: se ripensiamo alla curva chiusa che abbiamo utilizzato per circondare l'albero e prendiamo in considerazione due semiarchi da congiungere, questi potranno essere uniti percorrendo la curva nell'uno o nell'altro verso; il nostro algoritmo sceglie sempre la direzione del tratto di curva che taglia meno archi. Il diagramma rappresentato nei vari casi, quindi, è effettivamente diverso, ma rappresenta lo stesso nodo.

Il programma ha nella sua interfaccia una "combo box", cioè un pulsante di selezione, che ci permette di scegliere il vertice da prendere come radice dell'albero. Variando questa scelta troveremo con molta probabilità la configurazione che più ci aggrada.

4.6 Energia dei nodi

I funzionali di energia sono stati associati ai nodi fin dalle origini della loro teoria, quando Gauss, impegnato a condurre esperimenti riguardo la mutua induzione di spire cariche poste nello spazio, ebbe la necessità di dare un rigore matematico agli oggetti fisici dei suoi esperimenti, ossia fili conduttori, in modo da poterli studiare. Questo ha poi suscitato un notevole interesse per lo studio di fenomeni fisici, come l'osservazione dell'evoluzione di un sistema costituito da un filo conduttore caricato uniformemente, annodato e infine chiuso agli estremi. L'idea di definire una funzione che associ ad ogni configurazione del filo in un certo istante un numero reale è giustificata dal fatto che, se la funzione definita ha buone proprietà, allora dal suo studio possiamo dedurre importanti informazioni sull'evoluzione del sistema stesso. Nell'esempio del filo condut-

tore, poichè presumibilmente il sistema tenderà a raggiungere una posizione di equilibrio, ci aspettiamo che in corrispondenza a tale configurazione la funzione assuma un valore minimo rispetto ai valori che assume nelle posizioni vicine, e ci aspettiamo che nel momento in cui il filo dovesse autointersecarsi (evento impossibile nella realtà) la funzione "esploda" all'infinito.

Nel nostro caso applicheremo al diagramma un funzionale di energia esclusivamente per ottenerne una configurazione ottimale, che ci consenta di distinguere bene sia gli incroci che gli archi. Nel tentativo di ottenere un modello simile al filo conduttore, immaginiamo di disporre lungo la corda intrecciata una serie di perline, che possiamo pensare come cariche elettriche, e costruiamo un funzionale di energia che regoli i rapporti tra queste. Per avere un'idea concreta di ciò che stiamo facendo, come primo tentativo prendiamo in considerazione il classico potenziale elettrostatico di Coulomb: se sulla nostra corda abbiamo disposto n perline caricate elettronicamente con $q_1, \ldots, q_n$ cariche, nei punti $p_1, \ldots, p_n$ dello spazio, il potenziale di Coulomb è dato da

$$F_c = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{q_i q_j}{d_{ij}}$$

dove $d_{ij} = |p_i - p_j|$ è la distanza tra i punti p_i e p_j. Minimizzare questo funzionale significherebbe portare la corda in una particolare posizione di equilibrio; se poniamo che le cariche siano unitarie, rimane come termine delle sommatorie il solo inverso della distanza che nel processo di minimizzazione tende a far allontanare il più possibile le cariche tra di loro, infatti, la distanza di due sferette che tende a zero fa tendere all'infinito l'energia, quindi durante la minimizzazione ci si muove nella direzione opposta.

Sommare più funzioni diverse in un unico funzionale di energia dà alla

corda un comportamento specifico. Per definire un'energia che ci permetta di ottenere una configurazione ottimale del diagramma, dobbiamo tenere conto delle sue particolari caratteristiche: stiamo lavorando in uno spazio a due dimensioni che rappresenta la proiezione di uno spazio tridimensionale, quindi i punti con cui lavoriamo, che rappresentano le posizioni delle perline, hanno due sole coordinate spaziali e quindi dei movimenti più limitati; il diagramma ha dei punti "speciali", che sono gli incroci, i quali andranno trattati diversamente dagli altri. Vediamo i termini che abbiamo inserito nel nostro funzionale, indicandone anche la derivata che ci tornerà utile nel calcolare il gradiente per la minimizzazione dell'energia stessa:

Energia di Möbius Questo è il termine principale della nostra energia ed è un riadattamento dell'energia di Möbius introdotta da O'Hara in [O'H91] e da Freedman, He e Wang in [FHW94], e rivista da Kim e Kusner in [KK93]. L'effetto della funzione è di allontanare le perline tra di loro, mantenendo una sorta di controllo tra una perlina e quelle adiacenti ad essa, in maniera che queste si dispongano uniformemente lungo la corda e non si "disperdano" troppo. L'*energia di Mobius per una coppia di punti* k, h $E_M(k,h)$ è:

$$E_M(k,h) = \frac{1}{(d_{k,h})^\alpha}\left[\overline{adj}_k \cdot \overline{adj}_h\right]$$

dove $d_{k,h}$ è la distanza dei punti k e h, α è un numero maggiore o uguale a 2, mentre $\overline{adj}_k$ è la media delle distanze di k dai punti adiacenti a k stesso. Se poniamo che k abbia, ad esempio, a_k punti adiacenti $(j_1, \ldots, j_{a_k}) \in A_k$,

essendo Ak l'insieme dei punti adiacenti a k, allora avremo:

$$d_{k,h} = \sqrt{(x_k - x_h)^2 + (y_k - y_h)^2}$$

$$\overline{adj}_k = \frac{1}{a_k}\Big[d_{j_1,k} + \ldots + d_{j_{a_k},k}\Big]$$

In particolare, per un incrocio del diagramma avremo $a_k = 4$, mentre per un generico punto lungo la corda avremo $a_k = 2$.

Possiamo definire l'*energia di Mobius del punto k* $E_M(k)$ come la somma delle energie delle coppie di punti formate da k stesso con tutti gli altri punti del diagramma,

$$E_M(k) = \sum_{h \neq k} E_M(k,h) = \sum_{h \neq k} \frac{1}{(d_{k,h})^\alpha}\Big[\overline{adj}_k \cdot \overline{adj}_h\Big]$$

e l'*energia di Mobius totale* E_M come la somma dei contributi di energia dei singoli punti, divisa per due (dato che lo stesso contributo di energia, che coinvolge due punti, compare appunto due volte, una per ogni punto):

$$E_M = \frac{1}{2}\sum_k E_M(k) = \frac{1}{2}\sum_k \sum_{h \neq k} E_M(k,h) = \sum_{k=1}^{n-1} \sum_{h=k+1}^{n} E_M(k,h)$$

dove n è il numero totale delle perline del diagramma.

Se n è il numero totale di perline, allora il nostro funzionale avrà $2n$ variabili, corrispondenti alle due coordinate x e y di ciascun punto che determina la posizione della perlina stessa. Possiamo rappresentare le $2n$ variabili con una $2n$-upla $(z_1, \ldots, z_{2n})$; in questo modo, a ciascuna $2n$-upla corrisponderà una configurazione dei punti del diagramma sul piano. Il gradiente del funzionale sarà un vettore di $2n$ componenti, dove ciascuna componente è la derivata parziale del funzionale secondo la variabile corrispondente.

Per l'energia di Mobius, il contributo totale al gradiente per la componente corrispondente a x_k è dato da:

$$\frac{\partial E_M}{\partial x_k} = \sum_{h\neq k} \overline{adj}_h \left\{ \frac{1}{a_k (d_{k,h})^\alpha} \sum_{j\in A_k} \left[\frac{(x_k - x_j)}{d_{k,j}} \right] - \frac{\alpha\ \overline{adj}_k (x_k - x_h)}{(d_{k,h})^{\alpha+2}} \right\} + \\ + \sum_{j\in A_k} \sum_{h\neq j} \frac{\overline{adj}_h}{a_j (d_{j,h})^\alpha} \frac{(x_k - x_j)}{(d_{k,j})}$$

mentre quello per la componente corrispondente a y_k è:

$$\frac{\partial E_M}{\partial y_k} = \sum_{h\neq k} \overline{adj}_h \left\{ \frac{1}{a_k (d_{k,h})^\alpha} \sum_{j\in A_k} \left[\frac{(y_k - y_j)}{d_{k,j}} \right] - \frac{\alpha\ \overline{adj}_k (y_k - y_h)}{(d_{k,h})^{\alpha+2}} \right\} + \\ + \sum_{j\in A_k} \sum_{h\neq j} \frac{\overline{adj}_h}{a_j (d_{j,h})^\alpha} \frac{(y_k - y_j)}{(d_{k,j})}$$

e si ottiene dal precedente scambiando le x con le y.

Incroci ortogonali Questo termine deriva dal prodotto scalare tra vettori ed ha il suo minimo quando gli archi convergenti in ciascun incrocio si dispongono ortogonalmente. Sia i il punto associato ad un incrocio, di coordinate x_i, y_i, e siano $j_0, \ldots, j_3$ i quattro punti dove sono posizionate le perline adiacenti all'incrocio, ordinati in senso orario o antiorario (e non, ad esempio, Nord-Sud-Est-Ovest). Allora, se j_l e j_m sono due punti consecutivi secondo l'ordine precedentemente scelto, l'energia $E_{orto}(j_l, j_m)$ prodotta tra di loro è:

$$E_{orto}(j_l, j_m) = \left[\frac{(x_{j_l} - x_i)(x_{j_m} - x_i) + (y_{j_l} - y_i)(y_{j_m} - y_i)}{d_{i,j_l} d_{i,j_m}} \right]^2$$

e l'energia totale di questo termine, E_{orto} è data dalla sommatoria degli elementi precedenti, al variare degli incroci e delle j. Ciascuna delle

$E_{orto}(j_l, j_m)$ dà un contributo al gradiente per le componenti corrispondenti a $x_k, y_k, x_{j_l}, y_{j_l}, x_{j_m}, y_{j_m}$: posti

$$N = \left[(x_{j_l} - x_i)(x_{j_m} - x_i) + (y_{j_l} - y_i)(y_{j_m} - y_i)\right] \tag{4.1}$$

$$D = d_{i,j_l} d_{i,j_m} \tag{4.2}$$

in modo che

$$E_{orto}(j_l, j_m) = \left[\frac{N}{D}\right]^2$$

otteniamo

$$\frac{\partial E_{orto}(j_l, j_m)}{\partial x_i} = 2\left[\frac{N}{D}\right] \frac{1}{(d_{i,j_l} d_{i,j_m})^2} \left\{ (2x_i - x_{j_l} - x_{j_m}) \cdot D + \right.$$
$$\left. - N \cdot \left[\frac{(x_i - x_{j_l}) d_{i,j_m}}{d_{i,j_l}} + \frac{(x_i - x_{j_m}) d_{i,j_l}}{d_{i,j_m}} \right] \right\}$$

$$\frac{\partial E_{orto}(j_l, j_m)}{\partial x_{j_l}} = 2\left[\frac{N}{D}\right] \frac{1}{d_{i,j_m} (d_{i,j_l})^2} \left\{ (x_{j_m} - x_i) d_{i,j_m} - N \cdot \left[\frac{(x_{j_l} - x_i)}{d_{i,j_l}} \right] \right\}$$

ed i rimanenti si ricavano da questi o invertendo le x con le y, o per simmetria.

Curvatura bassa I Questo termine opera direttamente sugli angoli compresi tra segmenti di corda consecutivi, e con esso diamo preferenza alle configurazioni che portano i tratti di corda ad avere la curvatura più bassa possibile. Se prendiamo in considerazione la figura 4.17, pensando a k, j_1, j_2 come punti della corda di cui abbiamo le coordinate, possiamo ricavare l'ampiezza dell'angolo γ come

$$\gamma = \arccos \frac{x_a x_b + y_a y_b}{|\overline{a}| \cdot |\overline{b}|} =$$
$$= \arccos \frac{(x_{j_1} - x_k)(x_{j_2} - x_k) + (y_{j_1} - y_k)(y_{j_2} - y_k)}{d_{k,j_1} d_{k,j_2}}$$

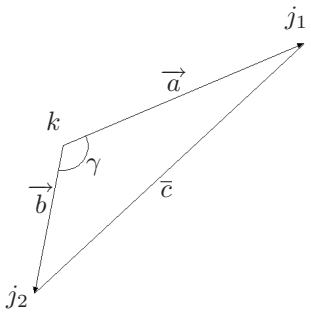


Figura 4.17

Dire che vogliamo avere curvatura bassa equivale a dire che l'angolo γ deve tendere a π, e si ottiene lo stesso risultato imponendo che l'argomento del coseno tenda a -1. Pertanto, l'energia di bassa curvatura I per il punto k, $E_{Cb_I}(k)$ è pari a:

$$E_{Cb_I}(k) = \left[\frac{(x_{j_1}-x_k)(x_{j_2}-x_k)+(y_{j_1}-y_k)(y_{j_2}-y_k)}{d_{k,j_1}d_{k,j_2}}+1\right]^2$$

Posti anche qui, come nel termine di ortogonalità degli incroci, N e D uguali, rispettivamente, al numeratore e al denominatore dell'argomento dell'arcocoseno, ciascuna delle $E_{Cb_I}(k)$ dà un contributo al gradiente per le componenti corrispondenti a $x_k, y_k, x_{j_1}, y_{j_1}, x_{j_2}, y_{j_2}$ pari a:

$$\frac{\partial E_{Cb_I}(k)}{\partial x_k} = 2\left[\frac{N}{D}+1\right]\frac{1}{(d_{k,j_1}d_{k,j_2})^2}\left\{(2x_k-x_{j_1}-x_{j_2})\cdot D + \right.$$
$$\left. - N\cdot\left[\frac{(x_k-x_{j_1})d_{i,j_2}}{d_{k,j_1}}+\frac{(x_k-x_{j_2})d_{k,j_1}}{d_{k,j_2}}\right]\right\}$$
$$\frac{\partial E_{Cb_I}(k)}{\partial x_{j_1}} = 2\left[\frac{N}{D}+1\right]\frac{1}{d_{k,j_2}(d_{k,j_1})^2}\left\{(x_{j_2}-x_k)d_{k,j_2}-N\cdot\left[\frac{(x_{j_1}-x_k)}{d_{k,j_1}}\right]\right\}$$

ed anche qui i rimanenti si ricavano da questi o invertendo le x con le y, o per simmetria.

Hooke E' il classico termine attrattivo dell'energia di Hooke ed ha l'effetto di una molla che collega le coppie consecutive di perline. Il suo valore per il punto k è:

$$E_{Hooke}(k) = Hd_{k,j}^{1+\beta}$$

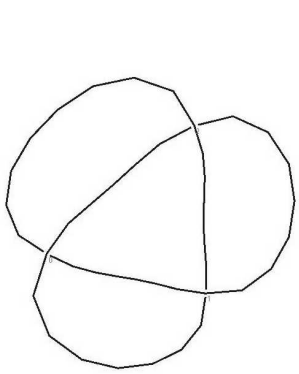
dove H è un coefficiente che rappresenta l'elasticità della molla, $d_{k,j}$ è la solita distanza tra i punti k e j, con j adiacente a k, e β è un esponente variabile che può andare da zero in su. Il contributo al gradiente per questo termine è:

$$\frac{\partial E_{Hooke}(k)}{\partial x_k} = H(\beta+1)d_{k,j}^{\beta-1}(x_k - x_j)$$

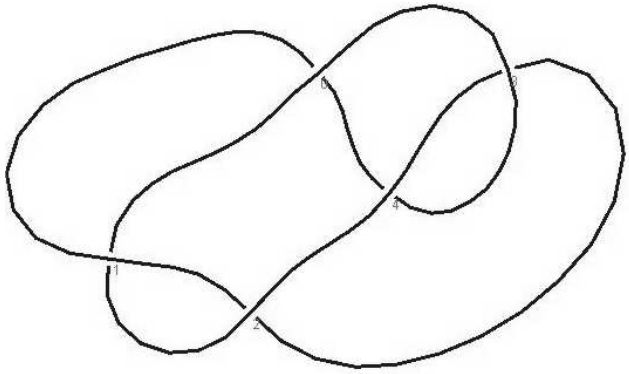
Il programma è fornito di un pannello di controllo che presenta al suo interno una checkbox[4] per ciascun termine dell'energia, precedentemente trattati. L'energia complessiva del diagramma è data dalla somma dei termini che hanno la checkbox attiva, e durante l'ottimizzazione è possibile aggiungerne o toglierne alcuni, semplicemente cliccando sulla checkbox corrispondente.

In figura 4.18 mostriamo i diagrammi che avevamo visto rappresentati da un disegno iniziale in figura 4.16, dopo che è stato minimizzata la loro energia. Allo stato attuale è facile rendersi conto che manca ancora al funzionale quel termine che rende ideale la configurazione del nodo, ma data la situazione iniziale particolarmente "intricata", possiamo al momento ritenerci soddisfatti.

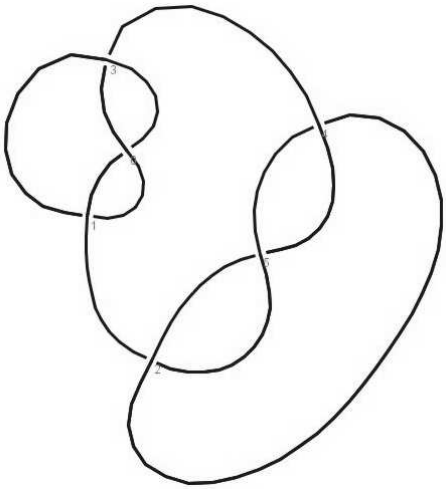
[4]Pulsante di selezione on/off.



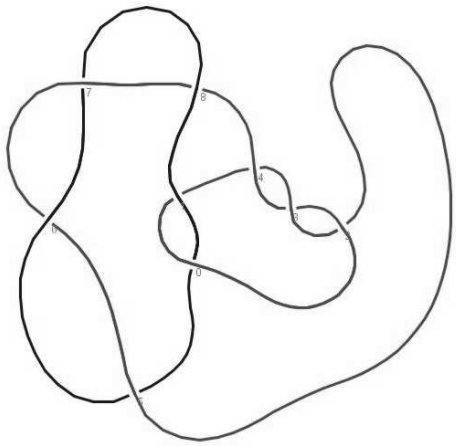
trifoglio



il nodo 8_1



il nodo della Nonna



un link più complesso

Figura 4.18: Alcuni nodi decodificati e ottimizzati dal programma.

Capitolo 5

Conclusioni e sviluppi futuri

Le novità principali introdotte in questa tesi sono i due tipi di codifica del nodo ed il software realizzato che le utilizza come scheletro per gestire il diagramma del nodo stesso, dando al programma un'impronta di carattere fortemente topologico.

Entrambe le codifiche hanno, per motivi diversi, la loro importanza: per la codifica esterna sottolineiamo la semplicità, data dalle (relativamente) poche informazioni necessarie ad identificare l'intero diagramma. La codifica interna ha invece la grande prerogativa di essere notevolmente "malleabile" all'interno del programma: ha il pregio, cioè, di poter essere gestita e modificata facilmente, il che consente di creare procedure anche particolarmente complesse senza dover ricorrere a sforzi eccessivi. Per entrambe abbiamo dimostrato la coerenza ed abbiamo sviluppato un algoritmo di conversione, dalla prima alla seconda, che consente all'utente del software di poter usufruire delle qualità della codifica interna, senza rinunciare alla naturalezza di quella esterna.

Il software realizzato è in grado di leggere e interpretare la codifica, sia interna che esterna, di un qualsiasi diagramma, passatagli attraverso un comune

file di testo contenente la codifica di un incrocio per ogni riga. E' stato scritto in linguaggio Java, che è dotato di molti punti a suo favore:

- è un linguaggio ad oggetti di ultima generazione;
- è in grado di funzionare su tutti i sistemi operativi;
- è molto diffuso;
- si presenta con una numerosa collezione di librerie, adatte ai nostri scopi;
- è in continua evoluzione.

Nella versione attuale, il programma:

- legge il file di testo;
- identifica il tipo di codifica utilizzata;
- interpreta la codifica, generando internamente gli incroci e gli archi (o i ponti) relativi;
- qualora la codifica utilizzata sia quella esterna, viene convertita in interna;
- genera il grafico iniziale del nodo, utilizzando la costruzione di un grafo ad albero avente per radice il nodo scelto dall'utente;
- ottimizza il grafico, applicando ripetutamente il metodo del gradiente per la minimizzazione dell'energia associata.

Il grafico finale, in verità, non è ancora di piena soddisfazione e necessita di ulteriori affinamenti.

Il programma realizzato è fondato su basi molto solide e, a mio giudizio, ben fatte, e può sicuramente costituire un punto di partenza per progetti futuri. Una volta terminata l'ottimizzazione del diagramma, è possibile cominciare ad implementare l'interattività vera e propria, come le operazioni sugli incroci, sulle componenti, o creare una procedura che consenta di semplificare un nodo operando i movimenti di Reidemeister. Ancora, si possono aggiungere procedure per la creazione di un nodo attraverso l'uso del mouse, per il calcolo degli invarianti, per il confronto tra nodi. Per concludere, si suggerisce l'implementazione della rappresentazione 3D, sempre utile e chiara, e dell'esportazione delle immagini ottenute in altri formati grafici, come ad esempio il pratico ".eps"[1].

Personalmente mi auguro che questo lavoro venga portato avanti.

Alla fine, non siamo riusciti a raggiungere tutti gli obiettivi che ci eravamo prefissi, ma la mole di lavoro svolto è stata veramente notevole, compresi i numerosi "intoppi - passi indietro - si ricomincia", non descritti nella tesi, inevitabili quando si svolge un lavoro di ricerca e sperimentazione. Rimane comunque la soddisfazione per il lavoro svolto e la sensazione di aver portato a termine qualcosa di grande: la laurea!

[1] Encapsulated PostScript.

Appendice A

Il codice del KnotExplorer

A.1 La classe Arco

```
public class Arco{
    public String label;
    public Incrocio incrocioIniziale, incrocioFinale;
    public Componente componente;

    public Arco(){
        incrocioIniziale=null;
        incrocioFinale=null;
    }

    public Arco(String i){
        new Arco();
        label=i;
    }

    public Arco(Incrocio diPartenza, Incrocio diArrivo){
        incrocioIniziale=diPartenza;
        incrocioFinale=diArrivo;
    }

    public void esceDa(Incrocio diPartenza){
        incrocioIniziale=diPartenza;
    }

    public void arrivaIn(Incrocio diArrivo){
        incrocioFinale=diArrivo;
    }

/**
 * Restituisce l'arco successivo
 */
    public Arco successivo(){
        if (incrocioFinale.underIncomingArc == this)
```

```
            return incrocioFinale.underOutgoingArc;
        if (incrocioFinale.overIncomingArc == this)
            return incrocioFinale.overOutgoingArc;
        System.out.println("ERRORE: Arco non trovato!");
        return new Arco();
    }

/**
 * Restituisce l'arco precedente
 */
    public Arco precedente(){
        if (incrocioIniziale.underOutgoingArc == this)
            return incrocioIniziale.underIncomingArc;
        if (incrocioIniziale.overOutgoingArc == this)
            return incrocioIniziale.overIncomingArc;
        System.out.println("ERRORE: Arco non trovato!");
        return new Arco();
    }

/**
 * Se la direzione è USCENTE mi restituisce l'incrocioFinale,
 * se è ENTRANTE l'incrocio iniziale.
 */
    public Incrocio getIncrocioOpposto(Direzione direzione){
        if (direzione.isEntrante()) return incrocioIniziale;
        else return incrocioFinale;
    }

    public Incrocio getIncrocio(Direzione direzione){
        if (direzione.isEntrante()) return incrocioFinale;
        else return incrocioIniziale;
    }

    /*nel caso in cui "label" sia di tipo int
    public String toString()
    {
        Integer intero = new Integer(label);
        return intero.toString();
    }*/

    public boolean finisceSotto(){
        return (incrocioFinale.underIncomingArc == this);
    }

    public boolean iniziaSotto(){
        return (incrocioIniziale.underOutgoingArc == this);
    }

    public void setLabel(String s){
        label=s;
    }
```

```
    public String toString(){
        return label;
    }
}
```

A.2 La classe Componente

```
public class Componente {

    private int label;//
    private boolean chiusa;
    private Arco arcoIniziale;

    public Componente(){}

    public Componente(Arco arco){
        arcoIniziale = arco;
    }

    public void setArcoIniziale(Arco arco){
        arcoIniziale = arco;
    }

    public Arco getArcoIniziale(){
        return arcoIniziale;
    }

    public void chiudi(){ chiusa = true; }

    public void apri(){ chiusa = false;}
    //public boolean isClosed() {}

/**
 * Restituisce un vettore contenente la lista degli archi appartenenti
 * alla componente
 */
    public ListaArchi archi(){
        ListaArchi listaArchi = new ListaArchi();
        if (arcoIniziale == null)
            {
                System.out.println("La componente è vuota!");
                return listaArchi;
            }
        listaArchi.addArco(arcoIniziale);
        Arco arcoPercorso = new Arco();
        arcoPercorso = arcoIniziale.successivo();
        while(arcoPercorso!=arcoIniziale)
            {
             listaArchi.addArco(arcoPercorso);
             arcoPercorso = arcoPercorso.successivo();
```

```
            }
        return listaArchi;
    }

/**
 * Restituisce un vettore contenente la lista degli incroci che incontro
 * percorrendo la componente, sia soprapassaggi che sottopassaggi.
 */
    public ListaIncroci incroci(){
        ListaIncroci listaIncroci = new ListaIncroci();
        if (arcoIniziale == null)
            {
                System.out.println("La componente è vuota!");
                return listaIncroci;
            }
        listaIncroci.addIncrocio(arcoIniziale.incrocioIniziale);
        Arco arcoPercorso = new Arco();
        arcoPercorso = arcoIniziale.successivo();
        while(arcoPercorso!=arcoIniziale)
            {
             listaIncroci.addIncrocio(arcoPercorso.incrocioIniziale);
             arcoPercorso = arcoPercorso.successivo();
            }
        return listaIncroci;
    }
}
```

A.3 La classe Connessione

```
import java.awt.*; import java.awt.geom.Line2D; import
java.util.Random;

public class Connessione{
    public Arco arco;
    public Punto puntoIniziale, puntoFinale;
    public ListaPunti puntiIntermedi;

    Connessione(){
        puntiIntermedi = new ListaPunti();
    }

    Connessione(Arco a){
        arco = a;
        puntiIntermedi = new ListaPunti();
    }

    Connessione(ListaPunti pti){
        puntiIntermedi = pti;
    }

    public void setPuntoIniziale(Punto p){
```

```
    puntoIniziale=p;
}

public void setPuntoFinale(Punto p){
    puntoFinale=p;
}

public Punto getPuntoIniziale(){
    return puntoIniziale;
}

public Punto getPuntoFinale(){
    return puntoFinale;
}

public void addPunto(Punto p){
    puntiIntermedi.addPunto(p);
}

public void addPunto(Punto[] p){
    for(int i=0; i<p.length; i++)
    puntiIntermedi.addPunto(p[i]);
}

public void suddividi(double passo){
    double distanza;
    int n = puntiIntermedi.size();
    Punto p1, p2, q;
    int nInt;  //numero di intervalli = n°punti da inserire
    double passoX, passoY;

    //se abbiamo solo i punti iniziale e finale
    if (puntiIntermedi.isEmpty()){
        distanza = puntoIniziale.distanzaDa(puntoFinale);
        nInt = (int)(distanza/passo);

        passoX = (puntoFinale.x-puntoIniziale.x)/(nInt+1);
        passoY = (puntoFinale.y-puntoIniziale.y)/(nInt+1);

        for (int k=1; k<=nInt; k++){
            q = new Punto(puntoIniziale.x+k*passoX,
                                puntoIniziale.y+k*passoY);
            puntiIntermedi.addPunto(q);
        }

        return;
    }

    //suddividiamo i tratti intermedi
    for (int i=n-1; i>0; i--){
        p1 = puntiIntermedi.getPunto(i);
```

```
            p2 = puntiIntermedi.getPunto(i-1);
            distanza = p1.distanzaDa(p2);
            nInt = (int)(distanza/passo);

            passoX = (p1.x-p2.x)/(nInt+1);
            passoY = (p1.y-p2.y)/(nInt+1);

            for (int k=1; k<=nInt; k++){
                q = new Punto(p1.x-k*passoX, p1.y-k*passoY);
                puntiIntermedi.addPunto(q, i);
            }
        }

        //suddividiamo il primo tratto
        p1 = puntiIntermedi.getPunto(0);
        distanza = puntoIniziale.distanzaDa(p1);
        nInt = (int)(distanza/passo);

        passoX = (p1.x-puntoIniziale.x)/(nInt+1);
        passoY = (p1.y-puntoIniziale.y)/(nInt+1);

        for (int k=1; k<=nInt; k++){
                q = new Punto(p1.x-k*passoX, p1.y-k*passoY);
                puntiIntermedi.addPunto(q, 0);
        }

        //suddividiamo l'ultimo tratto
        p1 = puntiIntermedi.getPunto(puntiIntermedi.size()-1);
        distanza = puntoFinale.distanzaDa(p1);
        nInt = (int)(distanza/passo);

        passoX = (puntoFinale.x-p1.x)/(nInt+1);
        passoY = (puntoFinale.y-p1.y)/(nInt+1);

        for (int k=1; k<=nInt; k++){
                q = new Punto(p1.x+k*passoX, p1.y+k*passoY);
                puntiIntermedi.addPunto(q);
        }
    }

    //restituisce il n° di punti inseriti
    public int suddividi(int k1, int k2, double passo){
        int r=0;

        if (k1<k2){
            Punto p1 = puntiIntermedi.getPunto(k1);
            Punto p2 = puntiIntermedi.getPunto(k2);
            double distanza = p1.distanzaDa(p2);
            //nInt = numero di intervalli = n°punti da inserire
            int nInt = (int)(distanza/passo);
            double passoX, passoY;
```

```
            nInt = (int)(distanza/passo);

            passoX = (p2.x-p1.x)/(nInt+1);
            passoY = (p2.y-p1.y)/(nInt+1);
            Punto q;

            for (int k=1; k<=nInt; k++){
                q = new Punto(p2.x-k*passoX, p2.y-k*passoY);
                puntiIntermedi.addPunto(q, k2);
                r++;
            }
        }
        return r;
    }

    public void suddividiPuntiCost(int nPunti){
        double distanza;
        int n = puntiIntermedi.size();
        Punto p1, p2, q;
        double passoX, passoY;

        if (n<=nPunti){

            //se abbiamo solo i punti iniziale e finale
            if (puntiIntermedi.isEmpty())
            {
                passoX = (puntoFinale.x-puntoIniziale.x)/(nPunti+1);
                passoY = (puntoFinale.y-puntoIniziale.y)/(nPunti+1);

                for (int k=1; k<=nPunti; k++){
                    q = new Punto(puntoIniziale.x+k*passoX,
                                                puntoIniziale.y+k*passoY);
                    puntiIntermedi.addPunto(q);
                }
            }
            else{
                int puntiDaInserire = nPunti - n;

                //trattoLungo=indice del punto che precede il tratto più lungo
                int trattoLungo=0;
                //inserisco ogni volta un punto nel tratto più lungo
                for (int i=0; i<puntiDaInserire; i++){
                    double distanzaMax = 0, distanzaAtt;
                    for (int j=0; j<puntiIntermedi.size()-1; j++){
                        p1 = puntiIntermedi.getPunto(j);
                        p2 = puntiIntermedi.getPunto(j+1);
                        distanzaAtt = p1.distanzaDa(p2);
                        if (distanzaAtt>distanzaMax){
                            distanzaMax = distanzaAtt;
                            trattoLungo = j;
```

```
                    }
                }

                //inserisco un nuovo punto a metà del tratto più lungo
                p1 = puntiIntermedi.getPunto(trattoLungo);
                p2 = puntiIntermedi.getPunto(trattoLungo+1);
                passoX = (p1.x-p2.x)/2;
                passoY = (p1.y-p2.y)/2;

                q = new Punto(p1.x-passoX, p1.y-passoY);
                puntiIntermedi.addPunto(q, trattoLungo+1);
            }
        }
    }
}

public void disegna(Graphics2D g2){
    //System.out.println("connessione.disegna");
    if (puntiIntermedi.isEmpty()){
            g2.setColor(Color.lightGray);
        g2.draw(new Line2D.Double(puntoIniziale.x, puntoIniziale.y,
                                  puntoFinale.x, puntoFinale.y));
    }
    else{
        Punto p1, p2;
        p1 = puntiIntermedi.getPunto(0);
        p1.disegna(g2);
        g2.setColor(new Color(new Random().nextInt()));
        g2.draw(new Line2D.Double(puntoIniziale.x, puntoIniziale.y,
                                                        p1.x, p1.y));
        for (int i=1; i<puntiIntermedi.size(); i++){
            p2 = puntiIntermedi.getPunto(i);
            p2.disegna(g2);
            g2.draw(new Line2D.Double(p1.x, p1.y, p2.x, p2.y));
            p1 = p2;
        }
        g2.draw(new Line2D.Double(p1.x, p1.y, puntoFinale.x, puntoFinale.y));
    }
}

public void inverti(){
    Punto temp = puntoIniziale;
    puntoIniziale = puntoFinale;
    puntoFinale = temp;

    int size = puntiIntermedi.size();

    ListaPunti tempLista = new ListaPunti(size);

    for (int i=size-1; i>=0; --i) {
        tempLista.addPunto(puntiIntermedi.getPunto(i));
```

```
        }

        puntiIntermedi = tempLista;
    }

    public String toString(){
        return "connessione di "+arco.toString();
    }

    public void scrivi(){
        System.out.println("Punti della "+this+":");
        System.out.print(puntoIniziale+", ");
        for (int i=0; i<puntiIntermedi.size(); i++){
            System.out.print(puntiIntermedi.getPunto(i)+", ");
        }
        System.out.println(puntoFinale);
    }
}
```

A.4 La classe Controls

```
import java.awt.*; import javax.swing.*; import
java.awt.event.ItemListener; import
javax.swing.border.TitledBorder; import
javax.swing.border.EtchedBorder;

public class Controls extends JPanel{

    private JPanel nodoPanel,energiaPanel;
    private JLabel nomeNodo = new JLabel(" ");

    public String[] hookeNames = {"1", "2", "3", "4", "5", "10"};

    public JComboBox incrociCombo, mobiusCombo, hookeCombo;

    private final int nCheckBox = 7; //numero delle CheckBox
    public JCheckBox mobiusCB, incrociOrtogonaliCB, curvaturaMinimaICB;
    public JCheckBox curvaturaMinimaIIICB, inversoDistanzaCB, hookeCB;

    public JCheckBox curvaturaMinimaIICB;
    public RunControls runControls;

    private Font font = new Font("serif", Font.PLAIN, 11);
    private Grafico grafico;

    public Controls(ItemListener myListener) {
        setSize(300,300);

        GridBagLayout myLayout = new GridBagLayout();
        setLayout(myLayout);
```

```
nodoPanel = new JPanel();
nodoPanel.setLayout(new GridBagLayout());
nodoPanel.setBorder(new TitledBorder(new EtchedBorder(), "Nodo"));

KnotExplorer.addToGridBag(this,nodoPanel,0,0,1,1,1,1);

energiaPanel = new JPanel();
energiaPanel.setLayout(new GridBagLayout());
energiaPanel.setBorder(new TitledBorder(new EtchedBorder(),"Energia"));
this.setMinimumSize(energiaPanel.getSize());
KnotExplorer.addToGridBag(this,energiaPanel,0,1,1,1,1,1);

mobiusCB = createCheckBox(energiaPanel,"Mobius", true, 0, myListener);
incrociOrtogonaliCB = createCheckBox(energiaPanel,
                           "Incroci Ortogonali",true, 1, myListener);
curvaturaMinimaICB = createCheckBox(energiaPanel,
                        "Curvatura Minima Tipo I", false, 2, myListener);
curvaturaMinimaIICB = createCheckBox(energiaPanel,
                        "Curvatura Minima Tipo II", false, 3, myListener);
curvaturaMinimaIIICB = createCheckBox(energiaPanel,
                        "Curvatura Minima Tipo III", false, 4, myListener);
inversoDistanzaCB = createCheckBox(energiaPanel,
                        "Inverso della distanza", false, 5, myListener);

hookeCombo = new JComboBox();
hookeCombo.setPreferredSize(new Dimension(45, 18));
hookeCombo.setLightWeightPopupEnabled(true);
hookeCombo.setFont(font);
hookeCombo.setEditable(true);
for (int i = 0; i < hookeNames.length; i++) {
    hookeCombo.addItem(hookeNames[i]);
}

JPanel hookeP = new JPanel(new FlowLayout(FlowLayout.LEFT, 0, 0));
hookeCB = new JCheckBox("Hooke", false);
hookeCB.setFont(font);
hookeCB.addItemListener(myListener);
hookeP.add(hookeCB);
hookeP.add(hookeCombo);
KnotExplorer.addToGridBag(energiaPanel, hookeP, 0, 6, 1, 1, 1, 1);

JLabel incrocioIniziale = new JLabel("Incrocio iniziale: ");
incrocioIniziale.setFont(font);
incrociCombo = new JComboBox();
incrociCombo.setSize(new Dimension(30, 16));
incrociCombo.setLightWeightPopupEnabled(true);
incrociCombo.setFont(font);
incrociCombo.addItemListener(myListener);
KnotExplorer.addToGridBag(nodoPanel, nomeNodo, 0, 0, 1, 1, 0.0, 0.0,
                                        GridBagConstraints.LINE_START);
KnotExplorer.addToGridBag(nodoPanel, incrocioIniziale, 0, 1, 1, 1, 0.0, 0.0,
```

```
                                            GridBagConstraints.LINE_START);
    KnotExplorer.addToGridBag(nodoPanel, incrociCombo, 1, 1, 1, 1, 10.0, 0.0,
                                            GridBagConstraints.LINE_END);
    KnotExplorer.addToGridBag(nodoPanel, new JLabel(" "), 0, 2, 1, 1, 0.0, 0.0);

    Mobius.setTermini(status());
}

private JCheckBox createCheckBox(String s, boolean b, int y, ItemListener listener) {
    JCheckBox cb = new JCheckBox(s, b);
    cb.setFont(font);
    cb.setHorizontalAlignment(JCheckBox.LEFT);
    cb.addItemListener(listener);
    KnotExplorer.addToGridBag(this, cb, 0, y, 1, 1, 1.0, 1.0);
    return cb;
}

private JCheckBox createCheckBox(JPanel p, String s, boolean b, int y,
                                                        ItemListener listener) {
    JCheckBox cb = new JCheckBox(s, b);
    cb.setFont(font);
    cb.setHorizontalAlignment(JCheckBox.LEFT);
    cb.addItemListener(listener);
    KnotExplorer.addToGridBag(p, cb, 0, y, 1, 1, 1.0, 1.0);
    return cb;
}


public void loadIncrociCombo(ListaIncroci li){
    incrociCombo.removeAllItems();
    for (int i = 0; i < li.size(); i++) {
        incrociCombo.addItem(li.getIncrocio(i).toString());
    }
}

public boolean[] status(){
    boolean[] b = new boolean[nCheckBox];

    b[Mobius.REPULSIONE_ATTRAZIONE] = mobiusCB.isSelected();
    b[Mobius.INCROCI_ORTOGONALI] = incrociOrtogonaliCB.isSelected();
    b[Mobius.CURVATURA_MINIMA_I] = curvaturaMinimaICB.isSelected();
    b[Mobius.CURVATURA_MINIMA_II] = curvaturaMinimaIICB.isSelected();
    b[Mobius.CURVATURA_MINIMA_III] = curvaturaMinimaIIICB.isSelected();
    b[Mobius.INVERSO_DISTANZA] = inversoDistanzaCB.isSelected();
    b[Mobius.HOOKE] = hookeCB.isSelected();

    return b;
}


public Dimension getPreferredSize() {
```

```
        return new Dimension(135,260);
    }

    public void setnomeNodo(String s){
        nodoPanel.setBorder(new TitledBorder(new EtchedBorder(), "Nodo:  "+s));
    }
}
```

A.5 La classe Diagramma

```
import java.awt.*; import javax.swing.*; import
java.awt.event.ItemListener; import
javax.swing.border.TitledBorder; import
javax.swing.border.EtchedBorder;

public class Controls extends JPanel{

    private JPanel nodoPanel,energiaPanel;
    private JLabel nomeNodo = new JLabel(" ");

    public String[] hookeNames = {"1", "2", "3", "4", "5", "10"};

    public JComboBox incrociCombo, mobiusCombo, hookeCombo;

    private final int nCheckBox = 7; //numero delle CheckBox
    public JCheckBox mobiusCB, incrociOrtogonaliCB, curvaturaMinimaICB;
    public JCheckBox curvaturaMinimaIIICB, inversoDistanzaCB, hookeCB;

    public JCheckBox curvaturaMinimaIICB;
    public RunControls runControls;

    private Font font = new Font("serif", Font.PLAIN, 11);
    private Grafico grafico;

    public Controls(ItemListener myListener) {
        setSize(300,300);

        GridBagLayout myLayout = new GridBagLayout();
        setLayout(myLayout);

        nodoPanel = new JPanel();
        nodoPanel.setLayout(new GridBagLayout());
        nodoPanel.setBorder(new TitledBorder(new EtchedBorder(), "Nodo"));

        KnotExplorer.addToGridBag(this,nodoPanel,0,0,1,1,1,1);

        energiaPanel = new JPanel();
        energiaPanel.setLayout(new GridBagLayout());
        energiaPanel.setBorder(new TitledBorder(new EtchedBorder(),"Energia"));
        this.setMinimumSize(energiaPanel.getSize());
        KnotExplorer.addToGridBag(this,energiaPanel,0,1,1,1,1,1);
```

```
    mobiusCB = createCheckBox(energiaPanel,"Mobius", true, 0, myListener);
    incrociOrtogonaliCB = createCheckBox(energiaPanel,
                              "Incroci Ortogonali",true, 1, myListener);
    curvaturaMinimaICB = createCheckBox(energiaPanel,
                            "Curvatura Minima Tipo I", false, 2, myListener);
    curvaturaMinimaIICB = createCheckBox(energiaPanel,
                            "Curvatura Minima Tipo II", false, 3, myListener);
    curvaturaMinimaIIICB = createCheckBox(energiaPanel,
                            "Curvatura Minima Tipo III", false, 4, myListener);
    inversoDistanzaCB = createCheckBox(energiaPanel,
                            "Inverso della distanza", false, 5, myListener);

    hookeCombo = new JComboBox();
    hookeCombo.setPreferredSize(new Dimension(45, 18));
    hookeCombo.setLightWeightPopupEnabled(true);
    hookeCombo.setFont(font);
    hookeCombo.setEditable(true);
    for (int i = 0; i < hookeNames.length; i++) {
        hookeCombo.addItem(hookeNames[i]);
    }

    JPanel hookeP = new JPanel(new FlowLayout(FlowLayout.LEFT, 0, 0));
    hookeCB = new JCheckBox("Hooke", false);
    hookeCB.setFont(font);
    hookeCB.addItemListener(myListener);
    hookeP.add(hookeCB);
    hookeP.add(hookeCombo);
    KnotExplorer.addToGridBag(energiaPanel, hookeP, 0, 6, 1, 1, 1, 1);

    JLabel incrocioIniziale = new JLabel("Incrocio iniziale: ");
    incrocioIniziale.setFont(font);
    incrociCombo = new JComboBox();
    incrociCombo.setSize(new Dimension(30, 16));
    incrociCombo.setLightWeightPopupEnabled(true);
    incrociCombo.setFont(font);
    incrociCombo.addItemListener(myListener);
    KnotExplorer.addToGridBag(nodoPanel, nomeNodo, 0, 0, 1, 1, 0.0, 0.0,
                                          GridBagConstraints.LINE_START);
    KnotExplorer.addToGridBag(nodoPanel, incrocioIniziale, 0, 1, 1, 1, 0.0, 0.0,
                                          GridBagConstraints.LINE_START);
    KnotExplorer.addToGridBag(nodoPanel, incrociCombo, 1, 1, 1, 1, 10.0, 0.0,
                                          GridBagConstraints.LINE_END);
    KnotExplorer.addToGridBag(nodoPanel, new JLabel(" "), 0, 2, 1, 1, 0.0, 0.0);

    Mobius.setTermini(status());
}

private JCheckBox createCheckBox(String s, boolean b, int y, ItemListener listener) {
    JCheckBox cb = new JCheckBox(s, b);
    cb.setFont(font);
```

```
        cb.setHorizontalAlignment(JCheckBox.LEFT);
        cb.addItemListener(listener);
        KnotExplorer.addToGridBag(this, cb, 0, y, 1, 1, 1.0, 1.0);
        return cb;
    }

    private JCheckBox createCheckBox(JPanel p, String s, boolean b, int y,
                                                          ItemListener listener) {
        JCheckBox cb = new JCheckBox(s, b);
        cb.setFont(font);
        cb.setHorizontalAlignment(JCheckBox.LEFT);
        cb.addItemListener(listener);
        KnotExplorer.addToGridBag(p, cb, 0, y, 1, 1, 1.0, 1.0);
        return cb;
    }


    public void loadIncrociCombo(ListaIncroci li){
        incrociCombo.removeAllItems();
        for (int i = 0; i < li.size(); i++) {
            incrociCombo.addItem(li.getIncrocio(i).toString());
        }
    }

    public boolean[] status(){
        boolean[] b = new boolean[nCheckBox];

        b[Mobius.REPULSIONE_ATTRAZIONE] = mobiusCB.isSelected();
        b[Mobius.INCROCI_ORTOGONALI] = incrociOrtogonaliCB.isSelected();
        b[Mobius.CURVATURA_MINIMA_I] = curvaturaMinimaICB.isSelected();
        b[Mobius.CURVATURA_MINIMA_II] = curvaturaMinimaIICB.isSelected();
        b[Mobius.CURVATURA_MINIMA_III] = curvaturaMinimaIIICB.isSelected();
        b[Mobius.INVERSO_DISTANZA] = inversoDistanzaCB.isSelected();
        b[Mobius.HOOKE] = hookeCB.isSelected();

        return b;
    }


    public Dimension getPreferredSize() {
        return new Dimension(135,260);
    }

    public void setnomeNodo(String s){
        nodoPanel.setBorder(new TitledBorder(new EtchedBorder(), "Nodo:  "+s));
    }
}
```

A.6 La classe Direzione

```
public class Direzione {
```

```
private int segno;

//devo immaginare di stare in un incrocio. La direzione di un arco sarà
//ENTRANTE se l'arco entra nell'incrocio, cioè se è un IncomingArc
public static final int ENTRANTE = 1;
public static final int USCENTE = -1;
public static final int ASSENTE = 0;

Direzione(int i){
    segno=i;
}

Direzione(Direzione d){
    segno = d.segno;
}

public int get(){
    return segno;
}

public boolean isEntrante(){
    if (segno==ENTRANTE) return true;
    else return false;
}

public void set(int i){
    segno = i;
}

public void setEntrante(){
    segno = ENTRANTE;
}

public void setUscente(){
    segno = USCENTE;
}

public void inverti(){
    segno = segno*(-1);
}

public boolean ugualeA(Direzione d){
    if (segno==d.segno) return true;
    else return false;
}

public String toString(){
    if (segno==1) return "ENTRANTE";
    if (segno==-1) return "USCENTE";
    if (segno==0) return "ASSENTE";
    return "non definita";
```

```
    }
}
```

A.7 La classe Energia

```
import java.util.*;
//import pal.math.*;

public abstract class Energia //implements MultivariateFunction,
                                                        MFWithGradient {
    public HashMap vertici, connessioni;
    public double[] argomenti;
    public double defaultLowerBound = -500, defaultUpperBound = 500;

    public abstract double evaluate(double[] args);

    public abstract double evaluate(double[] args, double[] gradient);

    public abstract void computeGradient(double[] args, double[] gradient);

    public int getNumArguments(){
        return argomenti.length;
    }

    public double getLowerBound(int n){
        return defaultLowerBound;
    }

    public double getUpperBound(int n){
        return defaultUpperBound;
    }

    public abstract double[] creaArgomenti();

    public abstract void applicaArgomenti();
}
```

A.8 La classe Grafico

```
import java.awt.*; import java.awt.geom.GeneralPath; import
java.util.*;
//import pal.math.*;

public class Grafico{
    public Diagramma diagramma;
    public HashMap vertici; //è la mappa incrocio->vertice corrispondente
    public HashMap connessioni; //è la mappa arco->connessione corrispondente

    private AlberoDiIncroci albero;
    private ListaPuntiOrdinata punti;
```

```
private static Color colors[] = { Color.blue, Color.red, Color.green,
            Color.magenta, Color.orange, Color.pink,
            Color.yellow, Color.lightGray, Color.cyan };

//caratteristiche del grafico, da tenere sotto controllo nelle procedure
private static double xMin=0, yMin=0, xMax=0, yMax=0;//le dimensioni del grafico
public double medSeg=1; //lunghezza media dei segmenti
private static final int minPunti = 6;//minimo punti intermedi di una connessione
private static BasicStroke knotStroke = new BasicStroke(3.0f, BasicStroke.CAP_BUTT,
                                                        BasicStroke.JOIN_ROUND);
//private static BasicStroke knotStroke = new BasicStroke();
private int firstNode = 0;

public Grafico(Diagramma d){
    diagramma = d;
    inizializza();
}

public void inizializza(){
    xMin=0; yMin=0; xMax=0; yMax=0;

    Incrocio root = diagramma.incroci.getIncrocio(firstNode);
    albero = new AlberoDiIncroci(root);
    punti = new ListaPuntiOrdinata();
    Vertice radice = albero.radice;
    associaPunto(radice, new Punto(0,0));

    posizionaFigli(radice);

    connetti();

    System.out.println("PRIMA DI SCALARE:");
    double larghezza = xMax-xMin;
    double altezza = yMax-yMin;

    System.out.println("xMin: "+xMin);
    System.out.println("xMax: "+xMax);
    System.out.println("yMin: "+yMin);
    System.out.println("yMax: "+yMax);

    System.out.println("altezza "+altezza);
    System.out.println("larghezza "+larghezza);

    scala(1,-1);

    System.out.println("DOPO AVER SCALATO:");
    larghezza = xMax-xMin;
    altezza = yMax-yMin;

    System.out.println("xMin: "+xMin);
    System.out.println("xMax: "+xMax);
```

```
        System.out.println("yMin: "+yMin);
        System.out.println("yMax: "+yMax);

        System.out.println("altezza "+altezza);
        System.out.println("larghezza "+larghezza);
    }

    public void associaPunto(Vertice q, Punto p){
        q.setPosizione(p);
        //punti.scrivi();

        if (!(punti.isEmpty())) sposta(q);
        punti.addPunto(p);
    }

    public void sposta(Vertice q){
        int direzione = q.provenienza();

        switch (direzione){
            case 0: {punti.spostaDestra(q);//System.out.println("spostaDestra");
                     break;}
            case 1: {punti.spostaSu(q);//System.out.println("spostaSu");
                     break;}
            case 2: {punti.spostaSinistra(q);//System.out.println("spostaSinistra");
                     break;}
            case 3: {punti.spostaGiù(q);//System.out.println("spostaGiù");
                     break;}
            case -1: {System.out.println("ERRORE su Grafico.sposta! il vertice è "+q);}
        }
    }

/**
 * Dato un vertice, stabilisce
 * le posizioni dei figli
 */
    public void posizionaFigli(Vertice q){

        Punto[] stella = q.stella();
        int n = stella.length;

        Vertice figlio;

        Ciclo_for:
        for (int k=0; k<n; k++){
            figlio = q.getFiglio(k);
            if (figlio==null) continue Ciclo_for;

            associaPunto(figlio, stella[k]);
            posizionaFigli(figlio);
        }
```

```
}


public void scala(double s){
    punti.espandi(s);
    Connessione c;

    Iterator i = connessioni.values().iterator();
    while(i.hasNext()){
        c = (Connessione) i.next();
        c.puntiIntermedi.espandi(s);
    }
    if (s>=0){
        xMin *= s;
        yMin *= s;
        xMax *= s;
        yMax *= s;
        medSeg *= s;
    }
    else {
        double temp = xMin;
        xMin = xMax*s;
        xMax = temp*s;
        temp = yMin;
        yMin = yMax*s;
        yMax = temp*s;
        medSeg *= -s;
    }
}

public void scala(double sx, double sy){
    punti.espandi(sx, sy);
    Connessione c;

    Iterator i = connessioni.values().iterator();
    while(i.hasNext()){
        c = (Connessione) i.next();
        c.puntiIntermedi.espandi(sx, sy);
    }

    if (sx>=0) {
        xMin *= sx;
        xMax *= sx;
    }
    else {
        double temp = xMin;
        xMin = xMax*sx;
        xMax = temp*sx;
    }

    if (sy>=0) {
```

```
            yMin *= sy;
            yMax *= sy;
        }
        else {
            double temp = yMin;
            yMin = yMax*sy;
            yMax = temp*sy;
        }
        medSeg *= Math.min(Math.abs(sx), Math.abs(sy));
    }

    public void disegnaVertici(Graphics2D g2){
        Vertice radice = albero.radice;
        radice.disegna(g2);

        disegnaFigli(g2, radice);
    }

    public void disegnaConnessioni(Graphics2D g2){
        for (int i=0; i<diagramma.archi.size(); i++){
            Arco a = diagramma.archi.getArco(i);
            Connessione c = getConnessione(a);

            c.disegna(g2);
        }
    }

    public void disegna(Graphics2D g2, int w, int h){

        //controllo le dimensioni del grafico, lo scalo e lo centro
        double larghezza = xMax-xMin;
        double altezza = yMax-yMin;

        double margine = 20; //margini esterni al grafico

        scala(Math.min(((double)w-2*margine)/larghezza,
                                            ((double)h-2*margine)/altezza));
        //dopo la scalatura, cambiano anche i margini Min e Max

        //coordinate del centro:
        double xc = xMin + (xMax-xMin)/2d;
        double yc = yMin + (yMax-yMin)/2d;

        g2.translate(w/2 - xc, h/2 - yc);//c'è un +15 per la barra dei menu?

        disegnaDiagramma(g2);
    }

    public void disegnaDiagramma(Graphics2D g2){

        g2.setStroke(knotStroke);
```

```
int ncomp = diagramma.componenti.size();//ncomp=numero componenti
Componente c;
Arco a, b = null;
Connessione conn;
GeneralPath gp = new GeneralPath();
double distDaIncrocio = 6;

for (int i=0; i<ncomp; i++){
    c = diagramma.componenti.getComponente(i);
    a = c.getArcoIniziale();

    //partiamo da un sottopassaggio:
    while (!a.iniziaSotto()) a = a.successivo();

    conn = getConnessione(a);
    b = a;
    Punto p, pi, p1;
    int jj=0;

    do {
        //punto iniziale
        pi = conn.getPuntoIniziale();
        p = pi; //se la connessione NON ha punti Intermedi questa
                //procedura NON funziona!
        if (conn.puntiIntermedi.isEmpty())
                                System.out.println("cosa non va?");
        if (b.iniziaSotto()) {
            p = conn.puntiIntermedi.getPunto(0);
            p1 = pi.puntoToPuntoDist(p, distDaIncrocio);
            gp.moveTo((float)p1.x, (float)p1.y);//
        }
        else {
            gp.lineTo((float)pi.x, (float)pi.y);
        }

        //punti intermedi
        for (int k=0; k<conn.puntiIntermedi.size(); k++) {
            p = conn.puntiIntermedi.getPunto(k);
            gp.lineTo((float)p.x, (float)p.y);
            //p.disegna(g2);//disegna i punti intermedi
        }

        //punto finale
        pi = conn.getPuntoFinale();

        if (b.finisceSotto()) {
            p1 = pi.puntoToPuntoDist(p, distDaIncrocio);
            gp.lineTo((float)p1.x, (float)p1.y);//
            g2.setColor(colors[i%colors.length]);
            g2.draw(gp);
```

```
                    gp = new GeneralPath();
                }
                else {
                    gp.lineTo((float)pi.x, (float)pi.y);
                    g2.setColor(colors[i%colors.length]);
                    g2.draw(gp);
                }

                b = b.successivo();
                conn = getConnessione(b);
            }
            while(b!=a);
        }

        disegnaVertici(g2);
    }

    public synchronized void changeFirstNode(int i){
        firstNode = i;
    }

/**
 * Dato un vertice, ne disegna i figli
 */
    public void disegnaFigli(Graphics2D g2, Vertice q){
        int n;

        if (q.padre==null) n=4;
        else n=3;

        Vertice figlio;

        Ciclo_for:
        for (int k=0; k<n; k++){
            figlio = q.getFiglio(k);
            if (figlio==null) continue Ciclo_for;

            figlio.disegna(g2);
            disegnaFigli(g2, figlio);
        }
    }

/**
 * Assegna alle connessioni, relative agli archi dei vertici, i punti di passaggio
 */
    public void connetti(){
        albero.connettiArchiInterni();
        albero.connettiArchiEsterni();
    }

    public void suddividi(double passo) {
```

```
        Iterator i = connessioni.values().iterator();
        while (i.hasNext()) {
            Connessione c = (Connessione)i.next();
            c.suddividi(passo);
        }

    }

    public void suddividiPuntiCost(int nPunti){
        Iterator i = connessioni.values().iterator();
        while (i.hasNext()) {
            Connessione c = (Connessione)i.next();
            c.suddividiPuntiCost(nPunti);
        }
    }

    public Connessione getConnessione(Arco a){
        return (Connessione)connessioni.get(a);
    }

    public static void azzeraLimiti(){
        xMin=0;
        yMin=0;
        xMax=0;
        yMax=0;
    }

    public static void assegnaLimiti(Punto p){
        xMin=p.x;
        yMin=p.y;
        xMax=p.x;
        yMax=p.y;
    }

    public static void controllaMargini(Punto p){
        xMin = Math.min(xMin, p.x);
        xMax = Math.max(xMax, p.x);
        yMin = Math.min(yMin, p.y);
        yMax = Math.max(yMax, p.y);
    }

    public void tagliaCuci(){
        //Calcolo la lunghezza media di un segmento, in base alla
        //lunghezza totale della curva
        double ltot = 0; //lunghezza totale della curva
        double nseg = 0; //numero dei segmenti
        Punto p1, p2, p;

        Iterator i = connessioni.values().iterator();

        while (i.hasNext()){
```

```
    Connessione c = (Connessione)i.next();

    //segmenti "esterni":
    if (c.puntiIntermedi.isEmpty()) {
        nseg++;
        ltot += (c.puntoIniziale).distanzaDa(c.puntoFinale);
    }
    else {
        nseg+=2;
        ltot += (c.puntoIniziale).distanzaDa(c.puntiIntermedi.getPunto(0));
        ltot += (c.puntoFinale).distanzaDa(c.puntiIntermedi.getPunto(
                                            c.puntiIntermedi.size()-1) );
    }

    //segmenti "interni"
    for (int k=1; k<c.puntiIntermedi.size(); k++){
        p2 = c.puntiIntermedi.getPunto(k);
        p1 = c.puntiIntermedi.getPunto(k-1);
        ltot += p1.distanzaDa(p2);
        nseg++;
    }
}

medSeg = ltot/nseg;

//Analizzo i segmenti: spezzerò quelli più lunghi di 2*medSeg e unirò
//quelli più corti di medSeg/2. Controllo anche i margini
double lmax = 2*medSeg;
double lmin = medSeg/2;

i = connessioni.values().iterator();

while (i.hasNext()){
    Connessione c = (Connessione)i.next();

    if (!c.puntiIntermedi.isEmpty())
                        controllaMargini(c.puntiIntermedi.getPunto(0));

    for (int k=1; k<c.puntiIntermedi.size(); k++){
        p2 = c.puntiIntermedi.getPunto(k);
        p1 = c.puntiIntermedi.getPunto(k-1);

        if (p1.distanzaDa(p2)<lmin && c.puntiIntermedi.size()>minPunti){
            //p1 = p1.puntoIntermedio(p2);//è meglio toglierla?
            c.puntiIntermedi.remove(k);
            k--;
        }
        else {
            if (p1.distanzaDa(p2)>lmax){
                //p = p1.puntoIntermedio(p2);
                //c.puntiIntermedi.addPunto(p, k);
```

```
                    int puntiInseriti = c.suddividi(k-1, k, medSeg);
                    k+=puntiInseriti;
                }
                controllaMargini(p2);
            }
        }
    }
}

public void ottimizzaUnPasso(){
    Mobius mobius = new Mobius(vertici, connessioni);
    double[] argomenti = mobius.creaArgomenti();
    double[] gradiente = new double[mobius.getNumArguments()];

    mobius.computeGradient(argomenti, gradiente);
    double trasl;
    int limite = (int)medSeg;

    //applica il gradiente
    for(int i=0; i<argomenti.length; i++){
        trasl = 2*gradiente[i];
        if (trasl<limite && trasl>-limite) argomenti[i]-=trasl;
        else argomenti[i]-= limite*(trasl/Math.abs(trasl));
        //mobius.applicaArgomenti();
        //piano.repaint();

        try{
                //rimane in attesa 300ms
                Thread.sleep(0);
        }
        catch (InterruptedException e){}

    }

    System.out.println("arguments= "+mobius.getNumArguments());

    mobius.applicaArgomenti();
    tagliaCuci();
}

public void ottimizza(int niterate){
    suddividi(4*medSeg); //suddivisione a passo costante

    Mobius mobius = new Mobius(vertici, connessioni);
    double[] argomenti = mobius.creaArgomenti();
    double[] gradiente = new double[mobius.getNumArguments()];
    double trasl;
    int limite;

    for(int num=0; num<niterate; num++){
        mobius.computeGradient(argomenti, gradiente);
```

```
            limite = (int)medSeg;
            //applica il gradiente
            for(int i=0; i<argomenti.length; i++){
                trasl = 2*gradiente[i];
                if (trasl<limite && trasl>-limite) argomenti[i]-=trasl;
                else argomenti[i]-= limite*(trasl/Math.abs(trasl));
            }

            System.out.println("num= "+num);
            System.out.println("arguments= "+mobius.getNumArguments());

            mobius.applicaArgomenti();
            tagliaCuci();
            argomenti = mobius.creaArgomenti();
            gradiente = new double[mobius.getNumArguments()];
        }
    }

    private class AlberoDiIncroci{
        private ListaArchi archiEsterni, archiInterni;
        private Jordan jordan;
        private Vertice radice;

        public AlberoDiIncroci(Incrocio root){
            radice = new Vertice(root);
            vertici = new HashMap();
            connessioni = new HashMap();
            archiEsterni = new ListaArchi();
            archiInterni = new ListaArchi();

            jordan = new Jordan();

            vertici.put(root, radice);
            creaAlbero();
        }

        public void creaAlbero(){
            Incrocio root = radice.incrocio;

            Arco tragitto;

            Direzione direzione = new Direzione(Direzione.ENTRANTE);
            if (root.segno==+1) tragitto = root.overIncomingArc;
            else tragitto = root.underIncomingArc;

            visitaGrafo(radice, tragitto, direzione);
            jordan.eliminaUltimo();
        }

        private void visitaGrafo(Vertice vertice, Arco provenienza, Direzione direzione1){
            int n;
```

```
        if (vertice.padre==null) n=4;
        else n=3;

        Direzione direzione = new Direzione(direzione1);
        Arco tragitto = provenienza;
        Incrocio x = vertice.incrocio;

        for(int k=0; k<n; k++){
            jordan.add(vertice);

            tragitto = x.giraSinistra(tragitto, direzione);

            if(!(connessioni.containsKey(tragitto))){
                Connessione connessione = new Connessione(tragitto);
                connessioni.put(tragitto, connessione);
            }

            Incrocio child = tragitto.getIncrocioOpposto(direzione);

            if (!(vertici.containsKey(child)))  {
                Vertice figlio = new Vertice(child);
                figlio.setPadre(vertice);
                System.out.println("l'incrocio "+figlio+" è figlio di "+vertice);
                vertice.setFiglio(figlio, k);
                vertici.put(child, figlio);
                Direzione nuovaDirezione = new Direzione(direzione);
                nuovaDirezione.inverti();
                visitaGrafo(figlio, tragitto, nuovaDirezione);
                archiInterni.addArco(tragitto);
            }
            else {
                archiEsterni.addArco(tragitto);
                jordan.add(connessioni.get(tragitto));
            }
        }
        jordan.add(vertice);
    }

/**
 * Assegna alle connessioni, relative agli archi interni dei vertici, i punti
 * iniziale e finale, inserendo k=minPunti interni.
 */
    public void connettiArchiInterni(){
        for (int i=0; i<archiInterni.size(); i++){
            Arco a = archiInterni.getArco(i);
            Connessione c = (Connessione)connessioni.get(a);
            Vertice n;
            Punto p1, p2, p;
            double passoX, passoY;
```

```
            n = (Vertice)vertici.get(a.incrocioIniziale);
            p1 = n.posizione;
            c.setPuntoIniziale(p1);

            n = (Vertice)vertici.get(a.incrocioFinale);
            p2 = n.posizione;
            c.setPuntoFinale(p2);

            passoX = (p2.x-p1.x)/(minPunti+1);
            passoY = (p2.y-p1.y)/(minPunti+1);

            for(int k=1; k<=minPunti; k++){
                p = new Punto(p1.x+k*passoX, p1.y+k*passoY);
                c.puntiIntermedi.addPunto(p);

            }
        }
    }

/**
 * Assegna alle connessioni, relative agli archi esterni dei vertici, i punti
 * iniziale e finale e quelli Intermedi, seguendo la curva di Jordan
 */
    public void connettiArchiEsterni(){
        jordan.connetti();
    }

    private class Jordan extends ArrayList{
        public Jordan(){
            super();
        }

        public void connetti(){
            Jordan mj23 = (Jordan)this.clone();
            livelli(1, mj23);
        }

        private int livelli(int n, Jordan air){
            int maxLivelli = n;

            HashMap listeVertici = new HashMap();
            ListaVertici ln = new ListaVertici();
            Connessione conPrec = null; //connessione precedente
            Connessione conCor; // connessione corrente
            int conPrecIndex=-1; //indice della connessione precedente
            int connessioniRimanenti = 0;
            int totVertici = air.totVertici();


            for(int i=0; i<air.size(); i++){
                if (air.get(i) instanceof Connessione){
```

```
            conCor = (Connessione)air.get(i);
            connessioniRimanenti++;
            if (conCor==conPrec && ln.size()<=totVertici/2){
                listeVertici.put(conCor, ln);
                ln = new ListaVertici();
                air.remove(i);
                air.remove(conPrecIndex);
                i-=2;
                connessioniRimanenti-=2;
            }
            else{
                conPrec = conCor;
                conPrecIndex = i;
                ln.clear();
            }
        }
        else {
            ln.addVertice((Vertice)air.get(i));
        }
    }

    //controllo se l'ultima connessione è uguale alla prima
    for(int i=0; i<conPrecIndex && connessioniRimanenti>1; i++) {
        if (air.get(i) instanceof Connessione){
            conCor = (Connessione)air.get(i);
            if (conCor==conPrec){
                listeVertici.put(conCor, ln);
                air.remove(conPrecIndex);
                air.remove(i);
                connessioniRimanenti-=2;
            }
            i=air.size();
        }
        else{
            ln.addVertice((Vertice)air.get(i));
        }
    }

    //l'& funziona bit a bit: dà 1 se sono entrambi dispari
    if ((connessioniRimanenti&1)==1){
        System.out.println("ERRORE: il diagramma è IMPOSSIBILE!!! " +
                                            "C'è un arco spaiato!");
        return 0;
    }
    else if (connessioniRimanenti>0) maxLivelli = livelli(n+1, air);

    double distanza = 1d/3d;

    distanza = (distanza/maxLivelli)*n;
    //tengo traccia della distanza minima tra connessioni,
    //mi servirà come passo iniziale tra punti
```

```
    medSeg = Math.min(medSeg, distanza);

    Iterator i = listeVertici.entrySet().iterator();
    Map.Entry map;

    //creo la connessione, tramite il metodo offset, per ogni connessione
    //del livello corrente
    while (i.hasNext())
    {
        map = (Map.Entry)i.next();
        //System.out.println("mapGetEntry"+(Connessione)map.getKey());
        ln = (ListaVertici)map.getValue();
        //ln.scrivi();
        conCor = (Connessione)map.getKey();
        conCor.setPuntoIniziale(ln.getVertice(0).posizione);
        conCor.setPuntoFinale(ln.getVertice(ln.size()-1).posizione);
        offset(conCor, ln, distanza);

        //controllo se la connessione ha il verso giusto,
        //altrimenti la inverto
        Vertice verticeIniziale = (Vertice) vertici.get(conCor.arco.incrocioIniziale);
        if (conCor.puntoIniziale != verticeIniziale.posizione) conCor.inverti();
    }

    return maxLivelli;
}

public void offset(Connessione c, ListaVertici ln, double d){
    int k=0;
    Vertice n1, n2, n3;
    n1 = ln.getVertice(k);
    k++;
    n2 = ln.getVertice(k);
    k++;

    //punto di partenza
    if (n1==n2)//due occorrenze consecutive dello stesso vertice
        {
            n2 = ln.getVertice(k);
            k++;
            if (n1==n2)//tre occorrenze
            {
                n2 = ln.getVertice(k);
                k++;
                c.addPunto(puntoDestro(n1, n2, d));
                c.addPunto(puntoSinistro(n2, n1, n2, d));
            }
            else {
                c.addPunto(puntoPosteriore(n1, n2, d));
                c.addPunto(puntoSinistro(n2, n1, n2, d)[1]);
            }
```

```
            }
        c.addPunto(puntoSinistro(n1, n2, d));

        //punti successivi
        for (; k<ln.size();k++) {
            n3 = ln.getVertice(k);

            while (n2==n3){
                try {
                    n3 = ln.getVertice(k+1);
                    k++;
                }
                catch (IndexOutOfBoundsException e) {
                    //se sono qui ho almeno 2 ripetizioni a fine vettore
                    k = ln.size()-2;
                    n1 = ln.getVertice(k);
                    n2 = ln.getVertice(k-1);

                    if (n1==n2)
                        {
                            n2 = ln.getVertice(k-2);
                            c.addPunto(puntoSinistro(n2, n1, n2, d));
                            c.addPunto(puntoSinistro(n1, n2, d));
                        }
                    else
                        {
                            c.addPunto(puntoSinistro(n2, n1, n2, d)[0]);
                            c.addPunto(puntoPosteriore(n1, n2, d));
                        }

                    return;
                }
            }

            c.addPunto(puntoSinistro(n1, n2, n3, d));
            n1 = n2;
            n2 = n3;
        }

        c.addPunto(puntoDestro(n2, n1, d));
    }

    private int totVertici(){
        int count=0;
        for (int k=0; k<size(); k++){
            if (this.get(k) instanceof Vertice) count++;
        }
        return count;
    }

    public int getFirstConnectionIndex(){
```

```
            for (int k=0; k<size(); k++)
                {
                    if (this.get(k) instanceof Connessione) return k;
                }
            return -1;
        }

        public int getLastConnectionIndex() {
            for (int k=size()-1; k>-1; k--) {
                if (this.get(k) instanceof Connessione) return k;
            }
            return -1;
        }

//teniamo traccia nelle procedure "punto*" delle coordinate min e max calcolate
        public Punto puntoSinistro(Vertice centrale, Vertice successivo, double d){
            Punto c = centrale.posizione;
            Punto s = successivo.posizione;
            Punto r = new Punto();

            r.x = c.x - d*segno(s.y-c.y);
            r.y = c.y + d*segno(s.x-c.x);

            xMin = Math.min(xMin, r.x);
            xMax = Math.max(xMax, r.x);
            yMin = Math.min(yMin, r.y);
            yMax = Math.max(yMax, r.y);

            return r;
        }

        public Punto[] puntoSinistro(Vertice precedente, Vertice centrale,
                                                Vertice successivo, double d){
            Punto p = precedente.posizione;
            Punto c = centrale.posizione;
            Punto s = successivo.posizione;
            Punto[] punti;
            double xd, yd;

            if (p.ugualeA(s)){
                punti = new Punto[2];

                xd = c.x + d*segno(c.x-p.x) - d*segno(c.y-p.y);
                yd = c.y + d*segno(c.y-p.y) + d*segno(c.x-p.x);
                punti[0] = new Punto(xd, yd);

                xMin = Math.min(xMin, xd);
                xMax = Math.max(xMax, xd);
                yMin = Math.min(yMin, yd);
                yMax = Math.max(yMax, yd);
```

```
            xd = c.x + d*segno(c.x-p.x) + d*segno(c.y-p.y);
            yd = c.y + d*segno(c.y-p.y) - d*segno(c.x-p.x);
            punti[1] = new Punto(xd, yd);

            xMin = Math.min(xMin, xd);
            xMax = Math.max(xMax, xd);
            yMin = Math.min(yMin, yd);
            yMax = Math.max(yMax, yd);
        }
        else{
            punti = new Punto[1];
            xd = c.x - d*segno(s.y-p.y);
            yd = c.y + d*segno(s.x-p.x);

            punti[0] = new Punto(xd, yd);

            xMin = Math.min(xMin, xd);
            xMax = Math.max(xMax, xd);
            yMin = Math.min(yMin, yd);
            yMax = Math.max(yMax, yd);
        }

        return punti;
    }

    public Punto puntoDestro(Vertice centrale, Vertice successivo, double d){
        Punto c = centrale.posizione;
        Punto s = successivo.posizione;
        Punto r = new Punto();

        r.x = c.x + d*segno(s.y-c.y);
        r.y = c.y - d*segno(s.x-c.x);

        xMin = Math.min(xMin, r.x);
        xMax = Math.max(xMax, r.x);
        yMin = Math.min(yMin, r.y);
        yMax = Math.max(yMax, r.y);

        return r;
    }

    public Punto puntoPosteriore(Vertice centrale, Vertice successivo, double d){
        Punto c = centrale.posizione;
        Punto s = successivo.posizione;
        Punto r = new Punto();

        r.x = c.x - d*segno(s.x-c.x);
        r.y = c.y - d*segno(s.y-c.y);

        xMin = Math.min(xMin, r.x);
        xMax = Math.max(xMax, r.x);
```

```
                yMin = Math.min(yMin, r.y);
                yMax = Math.max(yMax, r.y);

                return r;
            }



            protected int segno(double i){
                if (i==0) return 0;
                if (i>0) return 1;
                return -1;
            }

            public void scrivi(){
                System.out.println("Curva di Jordan:");
                for (int i=0; i<size(); i++){
                    Object obj = this.get(i);
                    if (obj instanceof Vertice){
                        Vertice n = (Vertice)obj;
                        System.out.println("Vertice "+n);
                    }
                    else{
                        Connessione a =(Connessione)obj;
                        System.out.println("Arco "+a);
                    }
                }
            }



            public void eliminaUltimo() {
                super.remove(this.size()-1);
            }
        }//Fine Classe Jordan

    }//Fine Classe Albero

}// Fine Grafico
```

A.9 La classe Immissione

```
import java.io.*; import java.util.*;

public class Immissione{

    public Immissione(){}

    public void daFile(Diagramma diagramma, String file){
        try {
            StreamTokenizer in = new StreamTokenizer(new InputStreamReader(
```

```
                                                    new FileInputStream(file)));
            in.slashSlashComments(true);
            in.slashStarComments(true);

            int nColonne = 0;
            int nextElem;

            in.eolIsSignificant(true);
            //controllo se ho un file a Ponti (4 colonne) o Archi (5 colonne)
            while (( (nextElem=in.nextToken() ) != StreamTokenizer.TT_EOL) &&
                                    (nextElem != StreamTokenizer.TT_EOF)) nColonne++;
            in.eolIsSignificant(false);

            in = new StreamTokenizer(new InputStreamReader(new FileInputStream(file)));

            if (nColonne==4) leggiPonti(in, diagramma);
            else leggiArchi(in, diagramma);

        } catch (IOException errore) { System.out.println("Errore nel file"); }
    }

    public void leggiArchi(StreamTokenizer in, Diagramma diagramma) {

        int nextElem;
        String chiave;
        int counter = 0;
        int npunto = 0;
        String[] chiavi = new String[4];
        HashMap mappa = new HashMap();

        try {
            while ((nextElem = in.nextToken()) != StreamTokenizer.TT_EOF) {
                if (nextElem == StreamTokenizer.TT_WORD) chiave = in.sval; //Stringa
                    else    chiave = (new Integer((int)in.nval)).toString();

                if (!mappa.containsKey(chiave))
                    {   mappa.put(chiave, new Arco(chiave)); }

                chiavi[counter] = chiave;
                counter++;

                if ( counter == 4){
                    // Carico l'incrocio
                    in.nextToken();
                    int segno = (int)in.nval;
                    Arco overInArc = (Arco)mappa.get(chiavi[0]);
                    Arco overOutArc = (Arco)mappa.get(chiavi[1]);
                    Arco underInArc = (Arco)mappa.get(chiavi[2]);
                    Arco underOutArc = (Arco)mappa.get(chiavi[3]);

                    Incrocio cross = new Incrocio(overInArc, overOutArc,
```

```
                                                underInArc, underOutArc, segno);

                cross.setLabel(npunto);

                System.out.println ("Nuovo incrocio n°"+cross+": "+overInArc+
                    ", "+overOutArc+", "+underInArc+", "+underOutArc+", "+segno);

                diagramma.addIncrocio(cross);
                npunto += 1;
                counter = 0;
            }
        }
    } catch (IOException errore) { System.out.println("Errore nel file"); }
}

public void leggiPonti(StreamTokenizer in, Diagramma diagramma) {

    int nextElem;
    String chiave;
    int counter = 0;
    int npunto = 0;
    String[] chiavi = new String[3];
    HashMap mappa = new HashMap();

    try {
        while ((nextElem = in.nextToken()) != StreamTokenizer.TT_EOF) {
            if (nextElem == StreamTokenizer.TT_WORD) chiave = in.sval; //Stringa
            else    chiave = (new Integer((int)in.nval)).toString();

            if (!mappa.containsKey(chiave))
                                { mappa.put(chiave, new Ponte(chiave)); }

            chiavi[counter] = chiave;
            counter++;

            if ( counter == 3){
                // Carico l'incrocio
                in.nextToken();
                int segno = (int)in.nval;
                Ponte overBridge = (Ponte)mappa.get(chiavi[0]);
                Ponte underInBridge = (Ponte)mappa.get(chiavi[1]);
                Ponte underOutBridge = (Ponte)mappa.get(chiavi[2]);

                Incrocio cross = new Incrocio(overBridge, overBridge,
                                        underInBridge, underOutBridge, segno);

                cross.setLabel(npunto);

                System.out.println ("Nuovo incrocio n°"+cross+": "+overBridge+
                            ", "+underInBridge+", "+underOutBridge+", "+segno);
```

```
                diagramma.addIncrocio(cross);
                npunto += 1;
                counter = 0;
            }
        }
    } catch (IOException errore) { System.out.println("Errore nel file"); }

    new Conversione(diagramma.incroci);
    diagramma.estraiArchi();
}

private class Conversione{

    private HashMap mappaIncroci;//mappa Ponte-->IncrociDelPonte
    private ListaPonti pontiDaConvertire;

    public Conversione(ListaIncroci vecchiIncroci){
        ListaIncroci li = vecchiIncroci;
        int lung = li.size();

        //mappaIncroci: creo una HashMap Ponte-->IncrociDelPonte
        //ogni lista contiene i soprapassaggi del Ponte, + gli incroci
        //iniziale e finale
        mappaIncroci = new HashMap();
        pontiDaConvertire = new ListaPonti(lung);//i ponti sono tanti quanti incroci

        Incrocio x;
        IncrociDelPonte incrociDelPonte;
        Ponte bridge, bridgeIn, bridgeOut;
        /* Analizzo tutti gli incroci della lista, memorizzando x ogni Ponte
            quali incroci lo vedono come soprapassante */
        for (int k=0; k<lung; k++)
        {
            x = li.getIncrocio(k);

            bridge = (Ponte)x.overIncomingArc;
            bridgeIn = (Ponte)x.underIncomingArc;
            bridgeOut = (Ponte)x.underOutgoingArc;
            //nota: leggo anche i Ponti underIncoming e underOutgoing per far sì
            //alla fine di avere una ListaPonti (lp) completa, e per avere
            //nella mappaIncroci una lista IncrociDelPonte per ogni Ponte.

            if (pontiDaConvertire.addIfAbsent(bridge)){
                incrociDelPonte = new IncrociDelPonte();
                mappaIncroci.put(bridge, incrociDelPonte);
            }
            if (pontiDaConvertire.addIfAbsent(bridgeIn)){
                incrociDelPonte = new IncrociDelPonte();
                mappaIncroci.put(bridgeIn, incrociDelPonte);
            }
            if (pontiDaConvertire.addIfAbsent(bridgeOut)){
```

```
            incrociDelPonte = new IncrociDelPonte();
            mappaIncroci.put(bridgeOut, incrociDelPonte);
        }

        //inserisco in incrociDelPonte corrispondenti ai nostri bridge,
        //l'incrocio analizzato
        incrociDelPonte = (IncrociDelPonte)mappaIncroci.get(bridge);
        incrociDelPonte.addIncrocio(x);

        incrociDelPonte = (IncrociDelPonte)mappaIncroci.get(bridgeIn);
        incrociDelPonte.setFinale(x);

        incrociDelPonte = (IncrociDelPonte)mappaIncroci.get(bridgeOut);
        incrociDelPonte.setIniziale(x);
    }

    /* ora eseguo una prima lettura di mappaIncroci, andando a controllare
        per ogni Ponte quanti soprapassaggi fa. Se ne fa 0 o 1 so già
        trasformare i Ponti in Archi secondo la corretta sequenza, mentre per
        2 soprapassaggi o più dovrò fare una seconda analisi.
    */
    Ciclo_for:
    for (int k=0; k<pontiDaConvertire.size(); k++){
        System.out.println(k);
        bridge = pontiDaConvertire.getPonte(k);
        incrociDelPonte = (IncrociDelPonte)mappaIncroci.get(bridge);

        if (incrociDelPonte.size()>3)   continue Ciclo_for;

        if (incrociDelPonte.size()==2){
            Arco a = new Arco(bridge.label);

            x = incrociDelPonte.getIniziale();
            x.setUnderOutgoing(a);

            x =incrociDelPonte.getFinale();
            x.setUnderIncoming(a);

            pontiDaConvertire.remove(k);
            k--;

            continue Ciclo_for;
        }

        if (incrociDelPonte.size()==3){
            Arco a = new Arco(bridge.label);
            Arco b = new Arco(bridge.label);

            x = incrociDelPonte.getIniziale();
            x.setUnderOutgoing(a);
```

```
                x = incrociDelPonte.getIncrocio(0);
                x.setOverIncoming(a);
                x.setOverOutgoing(b);

                x =incrociDelPonte.getFinale();
                x.setUnderIncoming(b);

                pontiDaConvertire.remove(k);
                k--;

                continue Ciclo_for;
            }
            //tutti i ponti dovrebbero avere almeno gli incroci iniziale e finale,
            //quindi abbiamo considerato tutti i casi.
        }

        for (int k=0; k<li.size(); k++) {
            Incrocio i = li.getIncrocio(k);
        }

        ListaIncroci incrociPercorsi;

        //convertiamo i ponti rimasti, quelli con più di 1 soprapassaggio (size>3)
        while(!pontiDaConvertire.isEmpty()) {
            bridge = pontiDaConvertire.getPonte(0);
            incrociDelPonte = (IncrociDelPonte)mappaIncroci.get(bridge);
            x = incrociDelPonte.getIncrocio();

            incrociPercorsi = new ListaIncroci();
            incrociPercorsi.addIncrocio(x);
            Direzione dBridge = x.direzione(bridge);

            superficie(x, bridge, dBridge, x, bridge, dBridge, incrociPercorsi);
        }

    }


//superficie(incrociopercorso x, arconell'incrocio arc, direzione di arc in x,
            //incrociodistop, arcodistop s, direzione di s, listaincrocipercorsi)
    private boolean superficie(Incrocio xProvenienza, Arco tragitto, Direzione dT,
                               Incrocio xFinale, Arco arcoFinale, Direzione dF,
                                                    ListaIncroci incrociPercorsi) {
        boolean chiusa = false;
        Incrocio x = null;
        Arco successivo;
        Direzione dSucc;

        if (tragitto instanceof Ponte){
            IncrociDelPonte idp = (IncrociDelPonte)mappaIncroci.get(tragitto);
```

```
//trovo tra gli IncrociDelPonte quello che chiude la superficie
for (int i=0; i<idp.nSoprapassaggi(); i++){
    x = idp.getIncrocio(i); //AndRemove??
    dSucc = new Direzione(dT);
    dSucc.inverti(); //la direzione va invertita perchè andiamo
                     //all'altro capo dell'arco
    successivo = x.giraSinistra(tragitto, dSucc);

    //condizioni di fine procedura
    if (incrociPercorsi.contains(x)){
        if (x==xFinale && successivo==arcoFinale && dSucc.ugualeA(dF))
            chiusa = true;
        else
            chiusa = false;
    }
    //nessuna condizione di termine è valida, allora reitero
    else {
        incrociPercorsi.addIncrocio(x);
        chiusa = superficie(x, successivo, dSucc, xFinale,
                            arcoFinale, dF, incrociPercorsi);
    }

    //se la superficie si è chiusa sono a posto, altrimenti devo
    //ritentare con un altro incrocio tra gli IncrociDelPonte
    if (chiusa) i = idp.nSoprapassaggi();
    else {}
}

//se la superficie ancora non è chiusa, provo gli incroci estremi
Prova_Estremi:
if (!chiusa) {
    dSucc = new Direzione(dT);
    dSucc.inverti();

    if (dSucc.isEntrante() && idp.haFinale()) x = idp.getFinale();
    else if (!dT.isEntrante() && idp.haIniziale())x = idp.getIniziale();
            else break Prova_Estremi;

    successivo = x.giraSinistra(tragitto, dSucc);

    //condizioni di fine procedura
    if (incrociPercorsi.contains(x)){
        if (x==xFinale && successivo==arcoFinale && dSucc.ugualeA(dF))
            chiusa = true;
        else
            chiusa = false;
    }
    //nessuna condizione di termine è valida, allora reitero
    else {
        incrociPercorsi.addIncrocio(x);
        chiusa = superficie(x, successivo, dSucc, xFinale,
```

```
                                        arcoFinale, dF, incrociPercorsi);
                }
            }

            //se la procedura è chiusa, allora posso creare un arco che andrà
            //da xProvenienza a x
            if (chiusa) {
                Arco nuovoArco = new Arco(tragitto.label);
                xProvenienza.cambiaArco(tragitto, dT, nuovoArco);
                dT.inverti();
                x.cambiaArco(tragitto, dT, nuovoArco);
                dT.inverti();

                //se l'incrocio non contiene più il ponte, lo rimuovo da idp
                if (!xProvenienza.contiene(tragitto)) {idp.rimuovi(xProvenienza);}
                if (!x.contiene(tragitto)) {idp.rimuovi(x);}

                //se idp è vuoto, rimuovo il ponte da pontiDaConvertire
                if (idp.isEmpty())
                    pontiDaConvertire.remove(pontiDaConvertire.indexOf(tragitto));
            }
        }
        else {
            x = tragitto.getIncrocioOpposto(dT);
            dSucc = new Direzione(dT);
            dSucc.inverti();
            successivo = x.giraSinistra(tragitto, dSucc);

            //condizioni di fine procedura
                if (incrociPercorsi.contains(x)){
                    if (x==xFinale && successivo==arcoFinale && dSucc.ugualeA(dF))
                        chiusa = true;
                    else
                        chiusa = false;
                }
                //nessuna condizione di termine è valida, allora reitero
                else {
                    incrociPercorsi.addIncrocio(x);
                    chiusa = superficie(x, successivo, dSucc, xFinale,
                                        arcoFinale, dF, incrociPercorsi);
                }
        }

        //se la superficie non si è chiusa devo eliminare da incrociPercorsi
        //l'arco aggiunto
        if (!chiusa){
            incrociPercorsi.remove(incrociPercorsi.size()-1);
        }

        return chiusa;
    }
```

```
}

private class IncrociDelPonte extends ListaIncroci  {
    private final Incrocio NULL = new Incrocio();

    public IncrociDelPonte(){
        super(3);
        addIncrocio(NULL);
        addIncrocio(NULL);
    }

    public void rimuoviIniziale(){
        remove(0);
        addIncrocio(NULL, 0);
    }

    public void rimuoviFinale(){
        remove(1);
        addIncrocio(NULL, 1);
    }

    public void rimuovi(Incrocio x) {
        if (x==getIniziale()) rimuoviIniziale();
        else    if (x==getFinale()) rimuoviFinale();
                else remove(indexOf(x));
    }

    public void setIniziale(Incrocio i) {
        remove(0);
        addIncrocio(i, 0);
    }

    public void setFinale(Incrocio i){
        remove(1);
        addIncrocio(i, 1);
    }

    public Incrocio getIniziale(){
        return super.getIncrocio(0);
    }

    public Incrocio getFinale() {
        return super.getIncrocio(1);
    }

    public boolean haIniziale() {
        if (getIniziale()==NULL) return false;
        else return true;
    }

    public boolean haFinale(){
```

```
    if (getFinale()==NULL) return false;
    else return true;
}

public boolean isEmpty(){
    if (size()>0) return false;
    else return true;
}

public boolean haSoprapassaggi(){
    if (nSoprapassaggi()>0) return true;
    else return false;
}

public int nSoprapassaggi() {
    int count = 0;
    if (haIniziale()) count++;
    if (haFinale()) count++;

    return size()-count;
}

public int size(){
    int count = 0;
    if (!haIniziale()) count+=1;
    if (!haFinale()) count+=1;
    return super.size() - count;
}

public Incrocio getIncrocio(int index){
    return super.getIncrocio(index+2);
}

public Incrocio getIncrocio(){
    Incrocio x = null;
    if (haIniziale()) x = getIniziale();
    else    if (haFinale()) x = getFinale();
            else if (!isEmpty()) x = getIncrocio(0);
    return x;
}

public Incrocio getAndRemoveIncrocio(){
    Incrocio x = null;
    if (haIniziale()) {
        x=getIniziale();
        rimuoviIniziale();
    }
    else    if (haFinale()){
                x=getFinale();
                rimuoviFinale();
            }
```

```
                    else    if (!isEmpty()){
                                //va bene?? o ci vuole 0?
                                x=super.getAndRemoveIncrocio(2);
                            }
            return x;
        }
    }
}
```

A.10 La classe Incrocio

```
public class Incrocio{
    public Arco overIncomingArc, overOutgoingArc, underIncomingArc, underOutgoingArc;
    public int segno;
    private String label;

/**
 * Creazione di un incrocio. Vengono passate informazioni anche agli archi
 * coinvolti.
 */
    public Incrocio(Arco overIncomingArc1, Arco overOutgoingArc1,
                    Arco underIncomingArc1, Arco underOutgoingArc1, int segno1) {
        overIncomingArc=overIncomingArc1;
        overOutgoingArc=overOutgoingArc1;
        underIncomingArc=underIncomingArc1;
        underOutgoingArc=underOutgoingArc1;
        segno=segno1;

        overOutgoingArc.esceDa(this);
        overIncomingArc.arrivaIn(this);
        underOutgoingArc.esceDa(this);
        underIncomingArc.arrivaIn(this);
    }

    public Incrocio(){
        overIncomingArc=null;
        overOutgoingArc=null;
        underIncomingArc=null;
        underOutgoingArc=null;
        segno=0;
    }

    public Incrocio(String e){
        overIncomingArc=null;
        overOutgoingArc=null;
        underIncomingArc=null;
        underOutgoingArc=null;
        segno=0;

        label=e;
    }
```

```
    public void setOverIncoming(Arco a) {
        overIncomingArc = a;
        overIncomingArc.arrivaIn(this);
    }

    public void setOverOutgoing(Arco a) {
        overOutgoingArc = a;
        overOutgoingArc.esceDa(this);
    }

    public void setUnderIncoming(Arco a){
        underIncomingArc = a;
        underIncomingArc.arrivaIn(this);
    }

    public void setUnderOutgoing(Arco a){
        underOutgoingArc = a;
        underOutgoingArc.esceDa(this);
    }

    public void setLabel(String l){
        label = l;
    }

    public void setLabel(int l){
        label = String.valueOf(l);
    }

    public void setSegno(int segno1){
        segno=segno1;
    }

    public void cambiaArco(Arco old, Direzione d, Arco niu){
        if (d.isEntrante()){
            if (old==underIncomingArc) setUnderIncoming(niu);
            else if (old==overIncomingArc) setOverIncoming(niu);
                    else System.out.println("Errore in Incrocio.cambiaArco");
        }
        else {
            if (old==underOutgoingArc) setUnderOutgoing(niu);
            else if (old==overOutgoingArc) setOverOutgoing(niu);
                    else System.out.println("Errore in Incrocio.cambiaArco");
        }
    }

/**
 * Quando chiamo questa procedura significa che la componente che nell'incrocio
 * passa sotto, ha invertito la direzione di percorrenza.
 */
    public void invertiSotto(){
```

```
        Arco temp = new Arco();
        temp = underIncomingArc;
        setUnderIncoming(underOutgoingArc);
        setUnderOutgoing(temp);
        segno = segno * (-1);
    }

/**
 * Quando chiamo questa procedura significa che la componente che nell'incrocio
 * passa sopra, ha invertito la direzione di percorrenza.
 */
    public void invertiSopra(){
        Arco temp = new Arco();
        temp = overIncomingArc;
        setOverIncoming(overOutgoingArc);
        setOverOutgoing(temp);
        segno = segno * (-1);
    }

/**
 * Quando chiamo questa procedura significa che sia la componente che nell'incrocio
 * passa sotto, sia quella che passa sopra, hanno invertito la direzione di percorrenza.
 */
    public void invertiEntrambe(){
        invertiSopra();
        invertiSotto();
    }

/**
 * Restituisce True se la componente di "overIncomingArc" e di "underIncomingArc"
 * sono la stessa
 */
    public boolean autoIncrocio(){
        return (overIncomingArc.componente == underIncomingArc.componente);
    }

/**
 * L'arco e la direzione immessa vengono sostituiti, come se nell'incrocio
 * girassimo a sinistra. La direzione mi dice se l'arco entra in questo
 * incrocio o ne esce.
 */
    public Arco giraSinistra(Arco tragitto, Direzione direzione){
        if (segno==+1){
            if (direzione.isEntrante()) {
                if (tragitto==overIncomingArc){
                    direzione.setUscente();
                    return underOutgoingArc;
                }
                if (tragitto==underIncomingArc) return overIncomingArc;
                System.out.println("ERRORE: qui l'Arco non entra!");
                return tragitto;
```

```
        }
        else  {
            if (tragitto==underOutgoingArc) {return overOutgoingArc;}
            if (tragitto==overOutgoingArc){
                direzione.setEntrante();
                return underIncomingArc;
            }
            System.out.println("ERRORE: da qui l'Arco non esce!");
            return tragitto;
        }
    }

    if (segno==-1){
        if (direzione.isEntrante()){
            if (tragitto==overIncomingArc) {return underIncomingArc;}
            if (tragitto==underIncomingArc){
                direzione.setUscente();
                return overOutgoingArc;
            }
            System.out.println("ERRORE: qui l'Arco non entra!");
            return tragitto;
        }
        else  {
            if (tragitto==underOutgoingArc){
                direzione.setEntrante();
                return overIncomingArc;
            }
            if (tragitto==overOutgoingArc)  {return underOutgoingArc;}
            System.out.println("ERRORE: da qui l'Arco non esce!");
            return tragitto;
        }
    }

    System.out.println("ERRORE: questo incrocio non ha segno!");
    return tragitto;
}

public Arco giraDestra(Arco tragitto, Direzione direzione)
{
    if (segno==+1){
        if (direzione.isEntrante()){
            if (tragitto==overIncomingArc)
                {return underIncomingArc;}
            if (tragitto==underIncomingArc){
                direzione.setUscente();
                return overOutgoingArc;
            }
            System.out.println("ERRORE: qui l'Arco non entra!");
            return tragitto;
        }
        else  {
```

```
            if (tragitto==underOutgoingArc){
                direzione.setEntrante();
                return overIncomingArc;
            }
            if (tragitto==overOutgoingArc)
                {return underOutgoingArc;}
            System.out.println("ERRORE: da qui l'Arco non esce!");
            return tragitto;
        }
    }

    if (segno==-1){
        if (direzione.isEntrante()) {
            if (tragitto==overIncomingArc){
                direzione.setUscente();
                return underOutgoingArc;
            }
            if (tragitto==underIncomingArc)
                {return overIncomingArc;}
            System.out.println("ERRORE: qui l'Arco non entra!");
            return tragitto;
        }
        else  {
            if (tragitto==underOutgoingArc)
                {return overOutgoingArc;}
            if (tragitto==overOutgoingArc){
                direzione.setEntrante();
                return underIncomingArc;
                }
            System.out.println("ERRORE: da qui l'Arco non esce!");
            return tragitto;
        }
    }

    System.out.println("ERRORE: questo incrocio non ha segno!");
    return tragitto;
}

/**
 *  Dato un arco restituisce la direzione che l'arco
 */
public Direzione direzione(Arco a){
    Direzione d;
    if (a==overOutgoingArc || a==underOutgoingArc)
        d = new Direzione(Direzione.USCENTE);
    else    if (a==overIncomingArc || a==underIncomingArc)
                d = new Direzione(Direzione.ENTRANTE);
            else d = new Direzione(Direzione.ASSENTE);
    return d;
}
```

```
    public boolean contiene(Arco a){
        if (a==overIncomingArc || a==overOutgoingArc || a==underIncomingArc ||
                                                a==underOutgoingArc) return true;
        else return false;
    }

    public String toString(){
        return label;
    }
}
```

A.11 La classe KnotExplorer

```
import javax.swing.*; import java.awt.event.*; import java.awt.*;

class KnotExplorer extends JFrame implements ItemListener{
    private Grafico grafico = null;
    private Piano piano;
    private Controls controls;
    private RunControls runControls;

    public static String file;
    public static LogBox log;

        KnotExplorer(){
            //creazione della finestra
            super("KnotExplorer");
            setLocation(0, 0);
            setSize(780, 560);
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            getContentPane().setLayout(new BorderLayout());

            MenuBar menu = creaMenu();
            setMenuBar(menu);

            piano = new Piano();
            piano.setSize(this.getSize());
            getContentPane().add(piano, BorderLayout.CENTER);

            //creo il pannello di controllo
            JPanel controllo = new JPanel(new GridBagLayout());

            log = new LogBox("");
            JScrollPane areaScrollPane = new JScrollPane(log);
            areaScrollPane.setVerticalScrollBarPolicy(
                            JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
            areaScrollPane.setBorder(
                BorderFactory.createCompoundBorder(
                    BorderFactory.createCompoundBorder(
                                    BorderFactory.createTitledBorder("Log "),
                                    BorderFactory.createEmptyBorder(5,5,5,5)),
```

```
                areaScrollPane.getBorder()));

        controls = new Controls(this);
        addToGridBag(controllo,controls,0,0,1,1,0,0);
        addToGridBag(controllo,areaScrollPane,0,1,1,1,30,100,
                                              GridBagConstraints.SOUTH);

        runControls = new RunControls(grafico,piano);
        addToGridBag(controls,runControls,0,3,1,1,0,0);
        GridBagConstraints c = new GridBagConstraints();

        getContentPane().add(controllo, BorderLayout.EAST);
        setVisible(true);
    }

public String file(){
    JFrame padre = this;
    FileDialog aprifile = new FileDialog(padre, "Apri File", FileDialog.LOAD);
    aprifile.setFile("*.txt");

    aprifile.show();
    String nomefile = aprifile.getFile();
    String nomedir  = aprifile.getDirectory();
    controls.setnomeNodo(nomefile);

    if (nomefile == null)  { // Abbandonata operazione
        return "";
    }
    return nomedir+nomefile;
}

public JFrame getJFrame(){
    return (JFrame)this;
}

static void addToGridBag(JPanel panel, Component comp,
        int x, int y, int w, int h, double weightx, double weighty) {

    GridBagLayout gbl = (GridBagLayout) panel.getLayout();
    GridBagConstraints c = new GridBagConstraints();
    c.fill = GridBagConstraints.BOTH;
    c.anchor = GridBagConstraints.PAGE_START;
    c.gridx = x;
    c.gridy = y;
    c.gridwidth = w;
    c.gridheight = h;
    c.weightx = weightx;
    c.weighty = weighty;
    panel.add(comp);
    gbl.setConstraints(comp, c);
}
```

```
static void addToGridBag(JPanel panel, Component comp,
        int x, int y, int w, int h, double weightx, double weighty,
        int anchor_type) {

    GridBagLayout gbl = (GridBagLayout) panel.getLayout();
    GridBagConstraints c = new GridBagConstraints();
    c.fill = GridBagConstraints.BOTH;
    c.anchor = anchor_type;
    c.gridx = x;
    c.gridy = y;
    c.gridwidth = w;
    c.gridheight = h;
    c.weightx = weightx;
    c.weighty = weighty;
    panel.add(comp);
    gbl.setConstraints(comp, c);
}

static void printLog(String s){
    log.print(s);
}

static void printlnLog(String s){
    log.print(s+ '\n');

}

public void itemStateChanged(ItemEvent e) {
    Object obj = e.getSource();

    if (obj instanceof JComboBox){
        JComboBox incrociCombo = (JComboBox) e.getSource();
        try {
            grafico.changeFirstNode(Integer.valueOf(
                            (String)incrociCombo.getSelectedItem()).intValue());
        } catch (NumberFormatException ex){grafico.changeFirstNode(0);}
        grafico.inizializza();
        piano.repaint();
    }
    else if (obj instanceof JCheckBox){
        runControls.ottimizzazione.stateChanged(controls.status());
    }
}

public static void main(String args[])  {
    KnotExplorer ke = new KnotExplorer();
}

protected MenuBar creaMenu(){
    MenuBar mb = new MenuBar();
```

```
    MenuItem apri=new MenuItem("Apri");
    MenuItem esci=new MenuItem("Esci");

    Menu file= new Menu("File");

    file.add(apri);
    file.addSeparator();
    file.add(esci);

    file.addActionListener(new FileActionListener());

    mb.add(file);

    MenuItem ottimizza = new MenuItem("Ottimizza");

    Menu strumenti = new Menu("Strumenti");

    strumenti.add(ottimizza);

    strumenti.addActionListener(new StrumentiActionListener());

    //mb.add(strumenti);

    return mb;
}

protected final class FileActionListener implements ActionListener{

    public void actionPerformed (ActionEvent e){
       if (e.getActionCommand().compareTo("Esci")==0)
             System.exit(0);

       if (e.getActionCommand().compareTo("Apri")==0){
            Immissione imm = new Immissione();
            Diagramma  diagramma = new Diagramma();
            String newFile = file();
            if (newFile != ""){
                file = newFile;
                imm.daFile(diagramma, file);
                diagramma.analizzaComponenti();
                Grafico.azzeraLimiti();
                grafico = new Grafico(diagramma);
                piano.setGrafico(grafico);
                runControls.setGrafico(grafico);
                controls.loadIncrociCombo(grafico.diagramma.incroci);
                piano.repaint();
            }
        }

    }
 }//FINE CLASSE FileActionListener
```

```
    protected final class StrumentiActionListener implements ActionListener {

        public void actionPerformed (ActionEvent e){
            if (e.getActionCommand().compareTo("Ottimizza")==0){

                String risultato = JOptionPane.showInputDialog(getJFrame(),
                                                    "Quante iterazioni?", "100");
                grafico.ottimizza((Integer.valueOf(risultato)).intValue());
            }
        }
    }//FINE CLASSE StrumentiActionListener

    class LogBox extends JTextArea implements Runnable {

        private Thread thread;

        public LogBox(String s){
            super(s);
            setEditable(false);
            setRows(5);
            setColumns(15);
            //log.setFont(new Font("Serif", Font.ITALIC, 10));
            setLineWrap(true);
            setWrapStyleWord(true);
        }

        public synchronized void print(String s){
            super.insert(s+'\n',0);
        }

        public void run(){

        }
    }
}
```

A.12 La classe ListaArchi

```
import java.util.ArrayList;

public class ListaArchi extends ArrayList{

    public ListaArchi(){
        super();
    }

    public boolean addArco(Arco o){
        return super.add(o); //sicuramente TRUE
    }
```

```
/**
 * Controlla che l'oggetto inserito non sia già presente. Se lo è,
 * ListaPonti resta invariata e viene restituito FALSE,
 * altrimenti il Ponte viene inserito nella lista e il risultato à TRUE
 */
    public boolean addIfAbsent(Arco a){
        if (!(this.contains(a))) return super.add(a); //sicuramente TRUE
        else return false;
    }

    public Arco getArco(){
        return (Arco)super.get(0);
    }

    public Arco getArco(int index){
        return (Arco)super.get(index);
    }

    public void scrivi(){
        System.out.println("Lista di archi:");
        for (int i=0; i<size(); i++){
            System.out.println(getArco(i));
        }
    }
}
```

A.13 La classe ListaComponenti

```
import java.util.ArrayList;

public class ListaArchi extends ArrayList{

    public ListaArchi(){
        super();
    }

    public boolean addArco(Arco o){
        return super.add(o); //sicuramente TRUE
    }

/**
 * Controlla che l'oggetto inserito non sia già presente. Se lo è,
 * ListaPonti resta invariata e viene restituito FALSE,
 * altrimenti il Ponte viene inserito nella lista e il risultato à TRUE
 */
    public boolean addIfAbsent(Arco a){
        if (!(this.contains(a))) return super.add(a); //sicuramente TRUE
        else return false;
    }

    public Arco getArco(){
```

```
        return (Arco)super.get(0);
    }

    public Arco getArco(int index){
        return (Arco)super.get(index);
    }

    public void scrivi(){
        System.out.println("Lista di archi:");
        for (int i=0; i<size(); i++){
            System.out.println(getArco(i));
        }
    }
}
```

A.14 La classe ListaConnessioni

```
import java.util.ArrayList;

public class ListaConnessioni extends ArrayList{

    public ListaConnessioni(){
        super();
    }

    public boolean addConnessione(Connessione o){
        return super.add(o); //sicuramente TRUE
    }

    public Connessione getConnessione(){
        return (Connessione)super.get(0);
    }

    public Connessione getConnessione(int index){
        return (Connessione)super.get(index);
    }

    public void scrivi(){
        System.out.println("Lista di connessioni:");
        for (int i=0; i<size(); i++)
        {
            System.out.println(getConnessione(i));
        }
    }
}
```

A.15 La classe ListaIncroci

```
import java.util.ArrayList;
```

```
public class ListaIncroci extends ArrayList{

    public ListaIncroci(){
        super();
    }

    public ListaIncroci(int dimensioneIniziale){
        super(dimensioneIniziale);
    }

    public boolean addIncrocio(Incrocio o){
        return super.add(o); //sicuramente TRUE
    }

    public void addIncrocio(Incrocio x, int i){
        super.add(i, x);
    }


/**
 * Controlla che l'oggetto inserito non sia già presente. Se lo è,
 * ListaIncroci resta invariata e viene restituito FALSE,
 * altrimenti l'incrocio viene inserito nella lista e il risultato à TRUE
 */
    public boolean addIfAbsent(Incrocio i){
        if (!(this.contains(i))) return super.add(i); //sicuramente TRUE
        else return false;
    }

    public Incrocio getIncrocio(){
        return (Incrocio)super.get(0);
    }

    public Incrocio getAndRemoveIncrocio(int index){
        return (Incrocio)remove(index);
    }

    public Incrocio getIncrocio(int index){
        return (Incrocio)super.get(index);
    }
}
```

A.16 La classe ListaPonti

```
import java.util.ArrayList;

public class ListaPonti extends ArrayList{

    public ListaPonti(){
        super();
    }
```

```
    public ListaPonti(int dimensioneIniziale){
        super(dimensioneIniziale);
    }

/**
 * Controlla che l'oggetto inserito non sia già presente. Se lo è,
 * ListaPonti resta invariata e viene restituito FALSE,
 * altrimenti il Ponte viene inserito nella lista e il risultato à TRUE
 */
    public boolean addIfAbsent(Ponte p) {
        if (!(this.contains(p))) return super.add(p); //sicuramente TRUE
        else return false;
    }

    public Ponte getPonte() {
        return (Ponte)super.get(0);
    }

    public Ponte getPonte(int index)    {
        return (Ponte)super.get(index);
    }
}
```

A.17 La classe ListaPunti

```
import java.util.ArrayList; import java.util.Iterator;

class ListaPunti extends ArrayList{

    public ListaPunti() {
        super();
    }

    public ListaPunti(int i){
        super(i);
    }

    public boolean addPunto(Punto p){
        return super.add(p);
    }

    public void addPunto(Punto p, int i){
        super.add(i, p);
    }

    public Punto getPunto(int indice){
        return (Punto)super.get(indice);
    }

/**
```

```
 * Moltiplica per c tutte le coordinate
 */
    public void espandi(double c){
        Punto p;
        for (int i=0; i<this.size(); i++){
            p = getPunto(i);
            p.x*=c;
            p.y*=c;
        }
    }

/**
 * Moltiplica per cx e cy rispettivamente le coordinate x e y
 */
    public void espandi(double cx, double cy){
        Punto p;
        for (int i=0; i<this.size(); i++){
            p = getPunto(i);
            p.x*=cx;
            p.y*=cy;
        }
    }

/**
 * Aggiunge alla ListaPunti tutti i punti della connessione, escluso puntoFinale
 */
    public void addConnessione(Connessione c){
        addPunto(c.puntoIniziale);
        ListaPunti pi = c.puntiIntermedi;

        Iterator i = pi.iterator();

        while(i.hasNext()) this.addPunto( (Punto)i.next() );
    }

/**
 * Da questa lista punti genera un array di double contenente in sequenza
 * le coordinate di ciascun punto: le x nei pari (partendo da 0), le y nei
 * dispari
 */
    public double[] toDouble(){
        int size = this.size()*2;

        double[] coords = new double[size*2];
        Punto p;

        for(int i=0; i<size; i++){
            p = getPunto(i);
            coords[i*2] = p.x;
            coords[i*2+1] = p.y;
        }
```

```
        return coords;
    }


    public void scrivi(){
        System.out.println("listaPunti");
        for (int i=0; i<size(); i++){
            System.out.println(getPunto(i));
        }
    }
}
```

A.18 La classe ListaPuntiOrdinata

```
class ListaPuntiOrdinata extends ListaPunti{

    public ListaPuntiOrdinata(){
        super();
    }

/**
 * Prova ad aggiungere il punto nela lista. Se è già presente restituisce
 * FALSE, altrimenti lo aggiunge nella corretta posizione e restituisce TRUE.
 */
    public boolean addPunto(Punto p){
        int posizione = whereIs(p);

        if (posizione <= 0) {
            super.add(-1*posizione, p);
            return true;
        }
        else return false;
    }

/**
 * Mi dice che posizione occupa il Punto all'interno della lista.
 * Se invece il punto non è presente, mi restituisce un numero negativo
 * che, se reso positivo, mi indica la posizione che occuperebbe il punto
 * se fosse presente.
 */
    public int whereIs(Punto p) {
        if (this.isEmpty()) return 0;//?
        int nonTrovato;

        int inizio = 0;
        int fine = this.size()-1;
        int centro;

        do  {
            centro = (inizio+fine)/2;
```

```
        if (p.x!=getPunto(centro).x){
            if (p.x<getPunto(centro).x)
                {fine = centro-1;}
            if (p.x>getPunto(centro).x)
                {inizio = centro+1;}
        }
        else{
            if (p.y<getPunto(centro).y)
                {fine = centro-1;}
            if (p.y>getPunto(centro).y)
                {inizio = centro+1;}
        }
    }
    while (inizio<=fine && !(p.equals( getPunto(centro) )));

    if (p.equals( getPunto(centro) )) return centro;

    nonTrovato = -1*inizio;
    return nonTrovato;
}

public Punto getPunto(int indice){
    return (Punto)super.get(indice);
}


public void spostaSu(Nodo q) {spostaSu(q.posizione);}

public void spostaSu(Punto p) {spostaSu((int)p.y);}

public void spostaSu(int yy){
    Punto p;
    for (int i=0; i<this.size(); i++){
        p = getPunto(i);
        if (p.y>=yy) p.spostaSu();
    }
}

public void spostaGiù(Nodo q) {spostaGiù(q.posizione);}

public void spostaGiù(Punto p){spostaGiù((int)p.y);}

public void spostaGiù(int yy){
    Punto p;
    for (int i=0; i<this.size(); i++){
        p = getPunto(i);
        if (p.y<=yy) p.spostaGiù();
    }
}
```

```
    public void spostaDestra(Nodo q) {spostaDestra(q.posizione);}

    public void spostaDestra(Punto p) {spostaDestra((int)p.x);}

    public void spostaDestra(int xx){
        Punto p;
        for (int i=0; i<this.size(); i++){
            p = getPunto(i);
            if (p.x>=xx) p.spostaDestra();
        }
    }

    public void spostaSinistra(Nodo q) {spostaSinistra(q.posizione);}

    public void spostaSinistra(Punto p){spostaSinistra((int)p.x);}

    public void spostaSinistra(int xx){
        Punto p;
        for (int i=0; i<this.size(); i++){
            p = getPunto(i);
            if (p.x<=xx) p.spostaSinistra();
        }
    }

//restituisce in [0] un numero che, se moltiplicato per la lista, garantisce
//comunque che i punti sono ancora entro i limiti (0,0) --> (w,h)
//in [1] e [2] ho le coordinate del "centro" della lista.
    public double[] getExpanderAndCenter(int w, int h){
        double[] ec = new double[3];
        double xMin=0.d, xMax=0.d, yMin=0.d, yMax=0.d;

        xMin = getPunto(0).x;
        xMax = getPunto(size()-1).x;

        for (int i=0; i<size(); i++){

        }

        return ec;
    }
}
```

A.19 La classe ListaVertici

```
import java.util.ArrayList;

public class ListaVertici extends ArrayList{

    public ListaVertici(){
        super();
    }
```

```
    public boolean addVertice(Vertice o){
        return super.add(o); //sicuramente TRUE
    }

    public void addListaVertici(ListaVertici ln){
        int n = ln.size();
        for (int i=0; i<n; i++) {
            addVertice(ln.getVertice(i));
        }
    }

    public Vertice getVertice(){
        return (Vertice)super.get(0);
    }

    public Vertice getVertice(int index){
        return (Vertice)super.get(index);
    }

    public void scrivi(){
        System.out.println("Lista di nodi:");
        for (int i=0; i<size(); i++)
        {
            System.out.println(getVertice(i));
        }
    }
}
```

A.20 La classe Mobius

```
import java.util.*;

public class Mobius extends Energia{

    private ListaPunti[] listaAdiacenze;
    private HashMap indici; //mappa punto-->indice di punto.x in argomenti[]

    private static boolean repulsioneAttrazione;
    private static boolean incrociOrtogonali;
    private static boolean curvaturaMinimaI;
    private static boolean curvaturaMinimaII;
    private static boolean curvaturaMinimaIII;
    private static boolean inversoDistanza;
    private static boolean isHooke;
    private static int alpha = 10;
    private static int alphaMobius = 2;
    private static double coeffDistIncr = 1;
    private static double coeffT3 = .05;
    private static double coeffHooke = 1;
    private static double espHooke = 0;
```

```
    public static final int REPULSIONE_ATTRAZIONE = 0;
    public static final int INCROCI_ORTOGONALI = 1;
    public static final int CURVATURA_MINIMA_I = 2;
    public static final int CURVATURA_MINIMA_II = 3;
    public static final int CURVATURA_MINIMA_III = 4;
    public static final int INVERSO_DISTANZA = 5;
    public static final int HOOKE = 6;

    public Mobius(HashMap n, HashMap c){
        vertici = n;
        connessioni = c;
    }

/**
 * E' fondamentale che come args[] venga passato l'array argomenti[]
 * ottenuto tramite il metodo costruisciArgomenti.
 * Si mantiene args[] solo per coerenza con l'interfaccia della libreria
 * di minimizzazione.
 */
    public double evaluate(double[] args){
        int n = args.length/2;
        double e=0;

        for (int i=0; i<n-1; i++)
            for (int j=i+1; j<n;j++){
                e += energia(i,j);
            }

        return e;
    }

    public double evaluate(double[] args, double[] gradient){
        computeGradient(args, gradient);
        return evaluate(args);
    }

//ATTENZIONE:la funzione che calcola l'energia NON contiene tutti i termini e NON è aggiornata!
    private double energia(int i, int j){

        double risultato, mediaAdiacenze=0;
        int ind;
        risultato = (1/Math.pow(distanza(i,j),2));

        //calcolo la media delle lunghezze dei segmenti che convergono in i
        for (int k=0; k<listaAdiacenze[i].size(); k++){
            ind = ( (Integer)indici.get(listaAdiacenze[i].getPunto(k)) ).intValue();
            mediaAdiacenze+= distanza(i, ind);
        }
        mediaAdiacenze = mediaAdiacenze/listaAdiacenze[i].size();
        risultato *= mediaAdiacenze;
```

```
    //calcolo la media delle lunghezze dei segmenti che convergono in i
    mediaAdiacenze = 0;
    for (int k=0; k<listaAdiacenze[j].size(); k++){
        ind = ( (Integer)indici.get(listaAdiacenze[j].getPunto(k)) ).intValue();
        mediaAdiacenze+= distanza(j, ind);
    }
    mediaAdiacenze = mediaAdiacenze/listaAdiacenze[j].size();
    risultato *= mediaAdiacenze;

    return risultato;
}

public void computeGradient(double[] args, double[] gradiente){
    int nVar = getNumArguments();
    int n = nVar/2;
    int k, h, j, ak, aj, ind, ind2;
    double sumAdKder, termine1, termine2, termine3, termineDist, hooke;
    double xk, yk, xh, yh, xj, yj, coeffT1, espT1;
    boolean isIncrocio;

    double[][] d = new double[n][n]; //contiene le distanze tra punti
    double[][] dDerX = new double[n][n];//contiene le derivate in x
                                        //del primo indice, delle distanze
    double[][] dDerY = new double[n][n];//contiene le derivate in y
                                        //del primo indice, delle distanze
    double[] adj = new double[n];//contiene le medie delle distanze degli adiacenti

     //variabili per termine90 e 180
    double termine90, termine180, num, den, r, xj2, yj2;

    for (k=0; k<n; k++){
        xk = argomenti[k*2];
        yk = argomenti[k*2+1];

        //inizializzo i gradienti
        gradiente[k*2]=0;
        gradiente[k*2+1]=0;

        //calcolo d, dDerX, dDerY
        for (h=k+1; h<n; h++){
            xh = argomenti[h*2];
            yh = argomenti[h*2+1];
            d[k][h] = distanza(k,h);
            d[h][k] = d[k][h];
            dDerX[k][h] = (xk-xh)/d[k][h];
            dDerX[h][k] = -dDerX[k][h];
            dDerY[k][h] = (yk-yh)/d[k][h];
            dDerY[h][k] = -dDerY[k][h];
        }
```

```
    //calcolo adj
    ak = listaAdiacenze[k].size(); //numero adiacenti k
    adj[k]=0;
    for (j=0; j<ak; j++){
        ind = ( (Integer)indici.get(listaAdiacenze[k].getPunto(j)) ).intValue();
        adj[k] += d[k][ind];
    }
    adj[k] = adj[k]/ak;
}

//derivata parziale di tutti i punti
for (k=0; k<n; k++){

    ak = listaAdiacenze[k].size(); //numero adiacenti k
    xk = argomenti[k*2];
    yk = argomenti[k*2+1];

    if (ak==4) isIncrocio = true;
    else isIncrocio = false;

//derivata parziale della xk

    //calcolo il termineDist
    if (inversoDistanza){
        for(h=0; h<n; h++) if (h!=k){
            termineDist = -( (double)alpha * dDerX[k][h] )/Math.pow(d[k][h],alpha);
            gradiente[k*2] += termineDist;
        }
    }

    //derivata parziale della media delle adiacenze di xk
    termine3=0;
    sumAdKder = 0;
    hooke = 0;
    for (j=0; j<ak; j++){
        aj = listaAdiacenze[j].size();
        //ind è l'indice di aj
        ind = ( (Integer)indici.get(listaAdiacenze[k].getPunto(j)) ).intValue();
        xj = argomenti[ind*2];
        yj = argomenti[ind*2+1];

        //calcolo il termine2
        if (repulsioneAttrazione){
            //termine2=0;
            if (isIncrocio){
                for(h=0; h<n; h++) if (h!=ind){
                    if (listaAdiacenze[h].size()==4) espT1=alphaMobius+2;
                    else espT1=alphaMobius;
                    gradiente[k*2] += (dDerX[k][ind]/aj)*(adj[h]/Math.pow(d[ind][h],
                                                                          espT1));
                }
```

```
    }
    else
        for(h=0; h<n; h++) if (h!=ind){
            gradiente[k*2] += (dDerX[k][ind]/aj)*(adj[h]/Math.pow(d[ind][h],
                                                                alphaMobius));
        }
}

//se sono in un vertice, calcolo il termine90
if (isIncrocio && incrociOrtogonali){
    ind2 = ( (Integer)indici.get(
                    listaAdiacenze[k].getPunto((j+1)%4)) ).intValue();
    xj2 = argomenti[ind2*2];
    yj2 = argomenti[ind2*2+1];

    num = (xj-xk)*(xj2-xk)+(yj-yk)*(yj2-yk);
    den = d[ind][k]*d[ind2][k];
    r = num/den;

    //termine90 di k per i due punti j e j2
    termine90 = 2*r*(1d/Math.pow(d[ind][k]*d[ind2][k],2))*((2*xk-xj-xj2)*den-
                    num*(dDerX[k][ind]*d[ind2][k]+dDerX[k][ind2]*d[ind][k]));
    gradiente[k*2] += termine90;

    //termine90 per j
    termine90 = 2*r*(1d/(d[ind2][k]*Math.pow(d[ind][k],2)))*
                                ((xj2-xk)*d[ind][k]-dDerX[ind][k]*num);
    gradiente[ind*2] += termine90;

    //termine90 per j2
    termine90 = 2*r*(1d/(d[ind][k]*Math.pow(d[ind2][k],2)))*
                                ((xj-xk)*d[ind2][k]-dDerX[ind2][k]*num);
    gradiente[ind2*2] += termine90;
}

//se non sono in un vertice, calcolo il termine180
if (!isIncrocio && curvaturaMinimaII){
    ind2 = ( (Integer)indici.get(
                    listaAdiacenze[k].getPunto((j+1)%2)) ).intValue();
    xj2 = argomenti[ind2*2];
    yj2 = argomenti[ind2*2+1];

    //termine180 di k per i due punti j e j2
    termine180 = (dDerX[k][ind]+dDerX[k][ind2]);
    gradiente[k*2] += termine180;

    //termine180 per j
    termine180 = dDerX[ind][k]-dDerX[ind][ind2];
    gradiente[ind*2] += termine180;

    //termine180 per j2
```

```
            termine180 = dDerX[ind2][k]-dDerX[ind2][ind];
            gradiente[ind2*2] += termine180;
        }

        //se non sono in un vertice, calcolo il termine180 di tipo II
        if (!isIncrocio && j==0 && curvaturaMinimaI){
            ind2 = ( (Integer)indici.get(
                            listaAdiacenze[k].getPunto((j+1)%2)) ).intValue();
            xj2 = argomenti[ind2*2];
            yj2 = argomenti[ind2*2+1];

            num = (xj-xk)*(xj2-xk)+(yj-yk)*(yj2-yk);
            den = d[ind][k]*d[ind2][k];
            r = num/den+1;

            //termine180 di k per i due punti j e j2
            termine180 = 2*r*(1d/Math.pow(d[ind][k]*d[ind2][k],2))*((2*xk-xj-xj2)*den-
                            num*(dDerX[k][ind]*d[ind2][k]+dDerX[k][ind2]*d[ind][k]));
            gradiente[k*2] += termine180;

            //termine180 per j
            termine180 = 2*r*(1d/(d[ind2][k]*Math.pow(d[ind][k],2)))*
                                        ((xj2-xk)*d[ind][k]-dDerX[ind][k]*num);
            gradiente[ind*2] += termine180;

            //termine180 per j2
            termine180 = 2*r*(1d/(d[ind][k]*Math.pow(d[ind2][k],2)))
                                        *((xj-xk)*d[ind2][k]-dDerX[ind2][k]*num);
            gradiente[ind2*2] += termine180;
        }

        //calcolo il termine di Hooke
        if (isHooke){
            hooke = coeffHooke*(1+espHooke)*Math.pow(d[k][ind],espHooke)*
                                                                dDerX[k][ind];
            gradiente[k*2] += hooke;
        }

        termine3 += xj;

        //somma delle distanze dagli adiacenti derivate
        sumAdKder += dDerX[k][ind];
    }
    //System.out.println("termine2 "+termine2);

    //calcolo il termine3
    if (curvaturaMinimaIII){
        termine3 -= (ak*xk);
        termine3 *= coeffT3;
        gradiente[k*2] -= termine3;
    }
```

```
    //calcolo il termine1
    if (repulsioneAttrazione){
        termine1=0;
        if (isIncrocio){
            for(h=0; h<n; h++) if (h!=k){
                if (listaAdiacenze[h].size()==4) espT1=alphaMobius+1;
                else espT1=alphaMobius;
                termine1 += adj[h]*( (-espT1*dDerX[k][h]*
                                    adj[k]/Math.pow(d[k][h],espT1+1d))
                                + (sumAdKder/(ak*Math.pow(d[k][h],espT1))) );
            }
        }
        else
            for(h=0; h<n; h++) if (h!=k){
                termine1 += adj[h]*(-alphaMobius*dDerX[k][h]*
                                    adj[k]/Math.pow(d[k][h],alphaMobius+1d)
                                +sumAdKder/(ak*Math.pow(d[k][h],alphaMobius)));
            }

        gradiente[k*2] += termine1;
    }

//derivata parziale della yk

    //calcolo il termineDist
    if (inversoDistanza){
        for(h=0; h<n; h++) if (h!=k){
            termineDist = -( (double)alpha*dDerY[k][h] )/Math.pow(d[k][h],alpha);
            gradiente[k*2+1] += termineDist;
        }
    }

    //derivata parziale della media delle adiacenze di yk
    termine3=0;
    sumAdKder = 0;
    for (j=0; j<ak; j++){
        aj = listaAdiacenze[j].size();
        ind = ( (Integer)indici.get(listaAdiacenze[k].getPunto(j)) ).intValue();
        xj = argomenti[ind*2];
        yj = argomenti[ind*2+1];

        //calcolo il termine2
        if (repulsioneAttrazione){
            //termine2=0;
            if (isIncrocio){
                for(h=0; h<n; h++) if (h!=ind){
                    if (listaAdiacenze[h].size()==4) espT1=alphaMobius+2;
                    else espT1=alphaMobius;
                    gradiente[k*2+1] += (dDerY[k][ind]/aj)*
                                        (adj[h]/Math.pow(d[ind][h], espT1));
```

```
            }
        }
        else
            for(h=0; h<n; h++) if (h!=ind){
                gradiente[k*2+1] += (dDerY[k][ind]/aj)*(
                                    adj[h]/Math.pow(d[ind][h], alphaMobius));
            }
        //gradiente[k*2+1] += termine2;
    }

    //se sono in un vertice, calcolo il termine90
    if (isIncrocio && incrociOrtogonali){
        ind2 = ( (Integer)indici.get(
                        listaAdiacenze[k].getPunto((j+1)%4)) ).intValue();
        xj2 = argomenti[ind2*2];
        yj2 = argomenti[ind2*2+1];

        num = (xj-xk)*(xj2-xk)+(yj-yk)*(yj2-yk);
        den = d[ind][k]*d[ind2][k];
        r = num/den;

        //termine90 di k per i due punti j e j2
        termine90 = 2*r*(1d/Math.pow(d[ind][k]*d[ind2][k],2))*((2*yk-yj-yj2)*den-
                        num*(dDerY[k][ind]*d[ind2][k]+dDerY[k][ind2]*d[ind][k]));
        gradiente[k*2+1] += termine90;

        //termine90 per j
        termine90 = 2*r*(1d/(d[ind2][k]*Math.pow(d[ind][k],2)))*
                                    ((yj2-yk)*d[ind][k]-dDerY[ind][k]*num);
        gradiente[ind*2+1] += termine90;

        //termine90 per j2
        termine90 = 2*r*(1d/(d[ind][k]*Math.pow(d[ind2][k],2)))*
                                    ((yj-yk)*d[ind2][k]-dDerY[ind2][k]*num);
        gradiente[ind2*2+1] += termine90;
    }

    //se non sono in un vertice, calcolo il termine180
    if (!isIncrocio && curvaturaMinimaII){
        ind2 = ( (Integer)indici.get(
                        listaAdiacenze[k].getPunto((j+1)%2)) ).intValue();
        xj2 = argomenti[ind2*2];
        yj2 = argomenti[ind2*2+1];

        //termine180 di k per i due punti j e j2
        termine180 = dDerY[k][ind]+dDerY[k][ind2];
        gradiente[k*2+1] += termine180;

        //termine180 per j
        termine180 = dDerY[ind][k]-dDerY[ind][ind2];
        gradiente[ind*2+1] += termine180;
```

```
                    //termine180 per j2
                    termine180 = dDerY[ind2][k]-dDerY[ind2][ind];
                    gradiente[ind2*2+1] += termine180;
                }

                //se non sono in un vertice, calcolo il termine180
                if (!isIncrocio && j==0 && curvaturaMinimaI){
                    ind2 = ( (Integer)indici.get(
                                    listaAdiacenze[k].getPunto((j+1)%2)) ).intValue();
                    xj2 = argomenti[ind2*2];
                    yj2 = argomenti[ind2*2+1];

                    num = (xj-xk)*(xj2-xk)+(yj-yk)*(yj2-yk);
                    den = d[ind][k]*d[ind2][k];
                    r = num/den+1;

                    //termine180 di k per i due punti j e j2
                    termine180 = 2*r*(1d/Math.pow(d[ind][k]*d[ind2][k],2))*((2*yk-yj-yj2)*den-
                                    num*(dDerY[k][ind]*d[ind2][k]+dDerY[k][ind2]*d[ind][k]));
                    gradiente[k*2+1] += termine180;

                    //termine180 per j
                    termine180 = 2*r*(1d/(d[ind2][k]*Math.pow(d[ind][k],2)))*
                                        ((yj2-yk)*d[ind][k]-dDerY[ind][k]*num);
                    gradiente[ind*2+1] += termine180;

                    //termine180 per j2
                    termine180 = 2*r*(1d/(d[ind][k]*Math.pow(d[ind2][k],2)))*
                                        ((yj-yk)*d[ind2][k]-dDerY[ind2][k]*num);
                    gradiente[ind2*2+1] += termine180;
                }

//              calcolo il termine di Hooke
                if (isHooke){
                    hooke = coeffHooke*(1+espHooke)*Math.pow(d[k][ind],espHooke)*
                                                                    dDerY[k][ind];
                    gradiente[k*2+1] += hooke;
                }

                termine3 += yj;

                //somma delle distanze dagli adiacenti derivate
                sumAdKder += dDerY[k][ind];
            }

            //calcolo il termine3
            if (curvaturaMinimaIII){
                termine3 -= (ak*yk);
                termine3 *= coeffT3;
                gradiente[k*2+1] -= termine3;
```

```
            }

//          calcolo il termine1
            if (repulsioneAttrazione){
                termine1=0;
                if (isIncrocio){
                    for(h=0; h<n; h++) if (h!=k){
                        if (listaAdiacenze[h].size()==4) espT1=alphaMobius+1;
                        else espT1=alphaMobius;
                        termine1 += adj[h]*(-espT1*dDerY[k][h]*
                                        adj[k]/Math.pow(d[k][h],espT1+1)
                                    +sumAdKder/(ak*Math.pow(d[k][h],espT1)));
                    }
                }
                else
                    for(h=0; h<n; h++) if (h!=k){
                        termine1 += adj[h]*(-alphaMobius*dDerY[k][h]*
                                        adj[k]/Math.pow(d[k][h],alphaMobius+1)
                                    +sumAdKder/(ak*Math.pow(d[k][h],alphaMobius)));
                    }

                gradiente[k*2+1] += termine1;
            }
        }

    }

    //calcola la media delle lunghezze dei segmenti che convergono in i
    private double mediaAdiacenze(int i){
        double media = 0;
        int indice;//l'indice dei vari punti adiacenti ad i

        for (int k=0; k<listaAdiacenze[i].size(); k++){
            indice = ( (Integer)indici.get(listaAdiacenze[i].getPunto(k)) ).intValue();
            media+= distanza(i, indice);
        }
        media = media/listaAdiacenze[i].size();
        return media;
    }

    private double distanza(int i, int j){
        double d = Math.sqrt((argomenti[i*2]-argomenti[j*2])*(argomenti[i*2]-argomenti[j*2])+
                    (argomenti[i*2+1]-argomenti[j*2+1])*(argomenti[i*2+1]-argomenti[j*2+1]));
        return d;
    }

//  derivata della distanza rispetto al primo punto
    private double distanzaDerivata(int i, int j, int XoY){
        //se XoY=0 allora è come se derivassi in xi, se XoY=1 è come se derivassi in yi
        double d = ((argomenti[i*2+XoY]-argomenti[j*2+XoY])/
                        (Math.sqrt(Math.pow((argomenti[i*2]-argomenti[j*2]),2)+
```

```
                    Math.pow((argomenti[i*2+1]-argomenti[j*2+1]),2))));
    return d;
}

public double[] creaArgomenti(){
    int nPunti=0;//n°punti in totale
    nPunti += vertici.size();
    Iterator i = connessioni.values().iterator();
    while (i.hasNext()){
        nPunti += ( (Connessione)i.next() ).puntiIntermedi.size();
    }

    argomenti = new double[nPunti*2];
    listaAdiacenze = new ListaPunti[nPunti];

    indici = new HashMap(nPunti);

    int k=0; //indice del punto in argomenti[]

    //inserisco i vertici e le rispettive adiacenze (in ordine orario o antiorario)
    i = vertici.values().iterator();

    Vertice n;
    Punto p1,p2;

    while(i.hasNext()){
        n = (Vertice)i.next();
        p1 = n.posizione;
        if (!indici.containsKey(p1)){
            indici.put(p1, new Integer(k)); //inserimento vertice
            listaAdiacenze[k] = adiacenzeVertice(n);//adiacenti
            argomenti[k*2] = p1.x;
            argomenti[k*2+1] = p1.y;
            k++;
        }
    }


    i = connessioni.values().iterator();

    Connessione c;

    while(i.hasNext()){
        c = (Connessione)i.next();

        p1 = c.puntoIniziale;
        p2 = c.puntoFinale;

        if (c.puntiIntermedi.size()>0){
            //inserisco i punti intermedi e le adiacenze
            p2 = c.puntiIntermedi.getPunto(0);
```

```
            indici.put(p2, new Integer(k));
            listaAdiacenze[k] = new ListaPunti();
            argomenti[k*2] = p2.x;
            argomenti[k*2+1] = p2.y;
            listaAdiacenze[k].addPunto(c.puntoIniziale);
            k++;
            p1=p2;

            for (int j=1; j<c.puntiIntermedi.size(); j++){
                p2 = c.puntiIntermedi.getPunto(j);
                indici.put(p2, new Integer(k));
                listaAdiacenze[k] = new ListaPunti();
                argomenti[k*2] = p2.x;
                argomenti[k*2+1] = p2.y;
                listaAdiacenze[k].addPunto(p1);
                listaAdiacenze[( (Integer)indici.get(p1) ).intValue()].addPunto(p2);
                k++;
                p1=p2;
            }

            p2 = c.puntoFinale;
            listaAdiacenze[( (Integer)indici.get(p1) ).intValue()].addPunto(p2);
        }
    }

    return argomenti;
}


public void applicaArgomenti(){
    Iterator i = indici.entrySet().iterator();
    Map.Entry map;
    int k;
    Punto p;

    //applico il primo punto e ne assegno i margini
    i.hasNext();
    map = (Map.Entry)i.next();
    k = ( (Integer)map.getValue() ).intValue();
    p = (Punto)map.getKey();
    p.x = argomenti[k*2];
    p.y = argomenti[k*2+1];


    Grafico.assegnaLimiti(p);

    while (i.hasNext()){
        map = (Map.Entry)i.next();
        k = ( (Integer)map.getValue() ).intValue();
        p = (Punto)map.getKey();
        p.x = argomenti[k*2];
```

```
            p.y = argomenti[k*2+1];

            //controllo i margini del grafico
            Grafico.controllaMargini(p);
        }
    }

    /**
     *
     * @author Mauro
     *
     * Questa procedura restituisce i quattro punti adiacenti al nodo, andandoli a prendere
     * nelle connessioni uscenti, in ordine OO UO OI UI (attenzione: può essere orario o antiorario).
     */
        private ListaPunti adiacenzeVertice(Vertice n){
            ListaPunti lp = new ListaPunti(4);
            Incrocio x = n.incrocio;
            Punto p;
            Connessione c;

//          aggiungo OO
            c = (Connessione)connessioni.get(x.overOutgoingArc);
            if (c.puntiIntermedi.size()==0) p = c.getPuntoFinale();
            else p = c.puntiIntermedi.getPunto(0);
            lp.addPunto(p);

//          aggiungo UO
            c = (Connessione)connessioni.get(x.underOutgoingArc);
            if (c.puntiIntermedi.size()==0) p = c.getPuntoFinale();
            else p = c.puntiIntermedi.getPunto(0);
            lp.addPunto(p);

//          aggiungo OI
            c = (Connessione)connessioni.get(x.overIncomingArc);
            if (c.puntiIntermedi.size()==0) p = c.getPuntoIniziale();
            else p = c.puntiIntermedi.getPunto(c.puntiIntermedi.size()-1);
            lp.addPunto(p);

//          aggiungo UI
            c = (Connessione)connessioni.get(x.underIncomingArc);
            if (c.puntiIntermedi.size()==0) p = c.getPuntoIniziale();
            else p = c.puntiIntermedi.getPunto(c.puntiIntermedi.size()-1);
            lp.addPunto(p);

            return lp;
        }

    public static void setTermini(boolean[] status){
        repulsioneAttrazione = status[REPULSIONE_ATTRAZIONE];
        incrociOrtogonali = status[INCROCI_ORTOGONALI];
        curvaturaMinimaI = status[CURVATURA_MINIMA_I];
```

```
        curvaturaMinimaII = status[CURVATURA_MINIMA_II];
        curvaturaMinimaIII = status[CURVATURA_MINIMA_III];
        inversoDistanza = status[INVERSO_DISTANZA];
        isHooke = status[HOOKE];
    }

    public void scriviAdiacenze(){
        int n = listaAdiacenze.length;
        for (int i=0; i<n; i++){
            System.out.println("lista adiacenze di "+i);
            listaAdiacenze[i].scrivi();
        }
    }
}
```

A.21 La classe Ottimizzazione

```
public class Ottimizzazione implements Runnable{

    private Thread thread;
    private boolean play = false;
    private int iterata;

    public Piano piano;
    public Grafico grafico;

    public Ottimizzazione(Grafico g, Piano p){
        grafico = g;
        piano = p;
    }

    public void setGrafico(Grafico g){
        grafico=g;
    }

    public synchronized void stateChanged(boolean[] status){
        Mobius.setTermini(status);
        System.out.println("cambio!");
    }

    public void start() {
        if (thread != null) {
            return;
        }
        play=true;
        iterata = 0;

        thread = new Thread(this);
        thread.setPriority(Thread.MIN_PRIORITY);
        thread.setName("Ottimizzazione");
        thread.start();
```

```
    }


    public synchronized void stop() {
        if (thread != null) {
            play=false;
        }
    }

    public void run(){
        grafico.suddividi(4*grafico.medSeg);

        while(play){
            synchronized(this){
                if (play){
                    System.out.println("num= "+iterata++);
                    KnotExplorer.printLog("Iterazione: "+iterata);
                    grafico.ottimizzaUnPasso();
                    piano.repaint();
                }
            }
        }
        thread = null;
    }
}
```

A.22 La classe Piano

```
import java.awt.*; import javax.swing.*;

public class Piano extends JPanel{
    private Graphics2D g = null;
    private Grafico grafico = null;

    public Piano()  {
        setOpaque(true);
        setBackground(Color.white);
        setVisible(true);
    }

    public void paintComponent(Graphics g){
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        Dimension d = getSize();
        g2.setBackground(getBackground());
        g2.clearRect(0, 0, d.width, d.height);
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                                        RenderingHints.VALUE_ANTIALIAS_ON);
        if (grafico!=null) grafico.disegna(g2, d.width, d.height);
```

```
    }

    public void setGrafico(Grafico g1){
        grafico = g1;
    }
}
```

A.23 La classe Ponte

```
public class Ponte extends Arco{

    public Ponte()  {
        incrocioIniziale=null;
        incrocioFinale=null;
    }

    public Ponte(String i)  {
        new Ponte();
        label=i;
    }

    public Ponte(Incrocio diPartenza, Incrocio diArrivo){
        incrocioIniziale=diPartenza;
        incrocioFinale=diArrivo;
    }
}
```

A.24 La classe Punto

```
import java.awt.*; import java.awt.geom.Ellipse2D;

class Punto{
    public double x, y;

    public Punto(){
        x=0;
        y=0;
    }

    public Punto(double ascissa, double ordinata){
        x = ascissa;
        y = ordinata;
    }

    public Punto(int ascissa, int ordinata) {
        x = ascissa;
        y = ordinata;
    }

    public void spostaSu(){
```

```
        y +=1;
    }

    public void spostaGiù(){
        y -=1;
    }

    public void spostaDestra(){
        x +=1;
    }

    public void spostaSinistra(){
        x -=1;
    }

    public boolean equals (Object obj){
        if (!(obj instanceof Punto)) return false;
        Punto p = (Punto)obj;
        return (x==p.x && y==p.y);
    }

    public boolean ugualeA(Punto p){
        return (x==p.x && y==p.y);
    }

    public double distanzaDa(Punto p){
        return Math.sqrt((p.x-this.x)*(p.x-this.x)+(p.y-this.y)*(p.y-this.y));
    }

    public Punto puntoIntermedio(Punto finale){
        double xp, yp;
        xp = x+(finale.x-x)/2;
        yp = y+(finale.y-y)/2;
        return new Punto(xp, yp);
    }

/**
 * Restituisce un punto compreso tra questo punto e "finale", a una distanza
 * "dist" da questo punto.
 *
 * @param finale
 * @param dist
 * @return
 */

    public Punto puntoToPuntoDist(Punto finale, double dist){
        double xp, yp, distanza, perc;
        distanza = this.distanzaDa(finale);//distanza tra i punti
        if (distanza<dist) perc = 1;
        else perc = dist/distanza;
        xp = x+perc*(finale.x-x);
```

```
        yp = y+perc*(finale.y-y);
        return new Punto(xp, yp);
    }

    public void disegna(Graphics2D g2){
        g2.setColor(Color.red);
        g2.draw(new Ellipse2D.Double(x - 1, y - 1, 2, 2));
    }

    public String toString(){
        return "("+x+","+y+")";
    }
}
```

A.25 La classe RunControls

```
import java.awt.*; import java.awt.event.ActionEvent; import
java.awt.event.ActionListener; import javax.swing.*;

public class RunControls extends JPanel implements ActionListener{

    private ImageIcon stopIcon, startIcon;
    private Grafico grafico;

    public Ottimizzazione ottimizzazione;
    public JButton startStopB;
    public JToolBar toolbar;


    public RunControls(Grafico g,Piano p){
        setBackground(Color.gray);
        setLayout(new BorderLayout());

        stopIcon = new ImageIcon(getClass().getResource("/images/stop.gif"));
        startIcon = new ImageIcon(getClass().getResource("/images/start.gif"));

        toolbar = new JToolBar();
        toolbar.setPreferredSize(new Dimension(26, 26));
        toolbar.setFloatable(false);

        startStopB = addTool("start.gif", "Start Animation", this);

        add(toolbar);
        ottimizzazione = new Ottimizzazione(g,p);
    }

    public void setGrafico(Grafico g){
        grafico=g;
        ottimizzazione.setGrafico(g);
    }
```

```
    public JButton addTool(String name, String toolTip, ActionListener al) {
        JButton b = null;
        b = (JButton) toolbar.add(new JButton(startIcon));
        b.setSelected(true);
        b.setToolTipText(toolTip);
        b.addActionListener(al);
        return b;
    }

    public void actionPerformed(ActionEvent e) {
        Object obj = e.getSource();

        if (obj instanceof JButton) {
            JButton b = (JButton) obj;
            b.setSelected(!b.isSelected());
            if (b.getIcon() == null) {
                b.setBackground(b.isSelected() ? Color.green : Color.lightGray);
            }
        }
        if (obj.equals(startStopB)) {
            if (KnotExplorer.file != null && KnotExplorer.file != ""){
                if (startStopB.getToolTipText().equals("Stop Animation")) {
                    startStopB.setIcon(startIcon);
                    startStopB.setToolTipText("Start Animation");
                    KnotExplorer.printLog("Stop ");
                    ottimizzazione.stop();
                } else {
                    startStopB.setIcon(stopIcon);
                    startStopB.setToolTipText("Stop Animation");
                    KnotExplorer.printLog("Start ");
                    ottimizzazione.start();
                }
            }
        }
    }
}
```

A.26 La classe Vertice

```
import java.awt.*;

public class Vertice{
    public Vertice padre, figlioSinistro, figlioCentrale, figlioDestro, figlioPosteriore;
    public Incrocio incrocio;
    public Punto posizione;

    public Vertice(){}

    public Vertice(Incrocio x)  {
        incrocio = x;
    }
```

```
    Vertice(Vertice parent){
        padre = parent;
    }

    public void setPadre(Vertice parent){
        padre = parent;
    }

    public void setIncrocio(Incrocio x) {
        incrocio = x;
    }

    public void setFiglio(Vertice child, int way)   {
        switch (way){
            case 0: {figlioSinistro = child; break;}
            case 1: {figlioCentrale = child; break;}
            case 2: {figlioDestro = child; break;}
            case 3: {figlioPosteriore = child; break;}
        }
    }

    public Vertice getFiglio(int way){
        switch (way){
            case 0: {return figlioSinistro;}
            case 1: {return figlioCentrale;}
            case 2: {return figlioDestro;}
            case 3: {return figlioPosteriore;}
        }
        System.out.println("ERRORE!");
        return new Vertice();
    }

    public void setPosizione(Punto p){
        posizione = p;
    }

    public void setPosizione(int x, int y){
        posizione.x = x;
        posizione.y = y;
    }

//immagino un riferimento statico, ad esempio guardando lo schermo.
    public int provenienza(){
        //provengo da sinistra (mi sposto a destra) = 0
        if (padre.posizione.y==posizione.y && padre.posizione.x<posizione.x)
            return 0;

        //provengo dal basso (mi sposto in alto) = 1
        if (padre.posizione.x==posizione.x && padre.posizione.y<posizione.y)
            return 1;
```

```
        //provengo da destra (mi sposto a sinistra) = 2
        if (padre.posizione.y==posizione.y && padre.posizione.x>posizione.x)
            return 2;

        //provengo dall'alto (mi sposto in basso) = 3
        if (padre.posizione.x==posizione.x && padre.posizione.y>posizione.y)
            return 3;

        return -1;
    }

// per usare questa procedura, questo nodo e l'eventuale padre devono avere
// una posizione ben definita
    public Punto[] stella() {
        if (padre==null){
            Punto[] punti = new Punto[4];
            if (posizione==null) setPosizione(new Punto(0,0));
            punti[0] = new Punto(posizione.x+1, posizione.y);
            punti[1] = new Punto(posizione.x, posizione.y-1);
            punti[2] = new Punto(posizione.x-1, posizione.y);
            punti[3] = new Punto(posizione.x, posizione.y+1);
            return punti;
        }

        Punto[] punti = new Punto[3];

        if (padre.posizione.x==posizione.x) {
            if (padre.posizione.y<posizione.y){
                punti[0] = new Punto(posizione.x-1, posizione.y);
                punti[1] = new Punto(posizione.x, posizione.y+1);
                punti[2] = new Punto(posizione.x+1, posizione.y);
            }
            else{
                punti[0] = new Punto(posizione.x+1, posizione.y);
                punti[1] = new Punto(posizione.x, posizione.y-1);
                punti[2] = new Punto(posizione.x-1, posizione.y);
            }

        }

        if (padre.posizione.y==posizione.y) {
            if (padre.posizione.x<posizione.x){
                punti[0] = new Punto(posizione.x, posizione.y+1);
                punti[1] = new Punto(posizione.x+1, posizione.y);
                punti[2] = new Punto(posizione.x, posizione.y-1);
            }
            else{
                punti[0] = new Punto(posizione.x, posizione.y-1);
                punti[1] = new Punto(posizione.x-1, posizione.y);
                punti[2] = new Punto(posizione.x, posizione.y+1);
```

```
            }

        }
        return punti;
    }//era meglio se utilizzavo this.provenienza()!!!!!

    void disegna(Graphics2D g2) {
        g2.setColor(new Color(0, 200, 0));
        g2.drawString(incrocio.toString(), (int)posizione.x + 3, (int)posizione.y + 11 + 3);
    }

    public String toString(){
        return incrocio.toString();
    }
}
```

Bibliografia

[AB27] J. W. Alexander e G. B. Briggs, *On types of knotted curves*, Annals of Mathematics 28 (1927), 562–586.

[ABGW97] Colin C. Adams, Bevin M. Brennan, Deborah L. Greilsheimer e Alexander K. Woo, *Stick numbers and composition of knots and links*, Journal of Knot Theory and Its Ramifications 6 (1997), 149–161.

[Ada94] Colin Adams, *The Knot Book*, W. H. Freeman, New York (1994).

[Ale28] J. W. Alexander, *Topological invariants of knots and links*, Transactions of the American Mathematical Society 30 (1928), 275–306.

[Art47] Emil Artin, *Theory of braids*, Annals of Mathematics 48 (1947), 101–126.

[Bra92] Kenneth Brakke, *The surface evolver*, Experimental Mathematics (1992), 141–165.

[Bra03] Kenneth Brakke, *Surface Evolver Manual* (2003).

[Con70] John Conway, *An enumeration of knots and links, and some of their algebraic properties*, Computational Problems in Abstract Algebra, In John Leech editor (1970), 329–358.

[DNT90] David Dobkin, Stephen North e Nathaniel Thurston, *A viewer for mathematical structures and surfaces in 3d*, Computer Graphics 24 (1990).

[DNT91] David Dobkin, Stephen North e Nathaniel Thurston, *Salem User's Guide*, Princeton University (1991).

[DT83] C. H. Dowker e Morwen B. Thistlethwaite, *Classification of knot pojections*, Topology and its Applications 16 (1983), 19–31.

[FHW94] M. H. Freedman, Z. X. He e Z. Wang, *Möbius energy of knots and unknots*, Annals of Mathematics 139 (Gennaio 1994), 1–50.

[FYH+85] P. Freyd, D. Yetter, J. Hoste, W. B. R. Lickorish, K. Millet e A. Ocneanu, *A new polynomial invariant of knots and links*, Bulletin of the American Mathematical Society 12 (Aprile 1985), 239–246.

[Hun96] Kenny Hunt, *KED* (1996), Software di editazione di nodi sviluppato presso l'Universita di Iowa.

[HW92] Shawn R. Henry e Jeffrey R. Weeks, *Symmetry groups of hyperbolic knots and links*, Journal of Knot Theory and Its Ramifications 1 (1992), 185–201.

[Jon85] Vaughan F. R. Jones, *A polynomial invariant for links via von*

neumann algebras, Bulletin of the American Mathematical Society 12 (1985), 103–112.

[Kau91] Louis H. Kauffman, *Knots and Physics*, Knots and Everything, World Scientific 1, Singapore (1991).

[KK93] Denise Kim e Rob Kusner, *Torus Knots Extremizing the Möbius Energy*, Experimental Mathematics 2 (1993), 1–9.

[Kos88] Czes Kosniowski, *Introduzione alla topologia algebrica*, Zanichelli Editore S.p.A. (1988).

[LM88] W. B. R. Lickorish e K. C. Millet, *The new polynomial invariants of knots and link*, Mathematics Magazine 61 (1988), 3–23.

[Mei96] Monica Meissen, *Edge numbers of knots*, PhD thesis, Department of Mathematics, The University of Iowa (1996).

[Mat03] Sergei Matveev, *Algorithmic Topology and Classification of 3-Manifolds*, Algorithms and Computation in Mathematics 9, Springer (2003).

[Neg91] S. Negami, *Ramsey theorems for knots, links, and spatial graphs*, Transactions of the American Mathematical Society 324 (1991), 527–541.

[O'H91] J. O'Hara, *Energy of a knot*, Topology 30 (1991), 241–247.

[PLM93] Mark Phillips, Silvio Levy e Tamara Munzner, *Geomview - an interactive geometry viewer*, Notices of the American Mathematical Society (1993).

[Ran96] Richard Randell, 1996 Private communication.

[Rol76] Dale Rolfsen, *Knots and Links*, Publish or Perish (1976).

[Sch98] Robert G. Scharein, *Interactive Topological Drawing*, PhD thesis, Department of Computer Science, The University of British Columbia (1998).

[Thi85] Morwen B. Thistlethwaite, *Knot tabulations and related topics*, Aspects of Topology in Memory of Hugh Dowker, London Mathematical Society Lecture Note Series 93, In I. M. James and E. H. Kronheimer editors, Cambridge University Press (1985), 1–76.

[Whi37] J. H. C. Whitehead, *On doubled knots*, J. London Math. Soc. 12 (1937), 63–71.

[Wu92] F. Y. Wu, *Knot theory and statistical mechanics*, Reviews of Modern Physics 64 (1992), 1099–1131.