Università degli Studi di Perugia

Dipartimento di Ingegneria

Corso di Laurea in Ingegneria Informatica ed Elettronica

Progetto e realizzazione di un algoritmo per la visualizzazione di insiemi di punti con posizioni assegnate

Laureando

Francesco Mancinelli

Relatore

Prof. Emilio Di Giacomo

Anno Accademico 2022-2023

Ringraziamenti

Prima di tutto vorrei ringraziare l'Università degli studi di Perugia, ed in particolare il mio relatore, il professore Emilio Di Giacomo, per le precise indicazioni, una presenza che mi ha accompagnato in questo momento cruciale per la mia carriera universitaria e per l'estrema disponibilità in ogni momento.

Un grazie speciale va alla mia famiglia, che mi ha sostenuto e incoraggiato in questo percorso difficile e in tutti gli altri momenti della mia vita. Senza il loro affetto e il loro aiuto, non sarei riuscito a raggiungere questo traguardo. Grazie anche per il sostegno incondizionato di tutti i miei sogni e tutti i miei desideri. Ringrazio di cuore i miei genitori, Roberta e Simone, che mi hanno sempre spinto a dare il meglio di me, che hanno creduto nelle mie capacità e che mi sono stati sempre vicini in ogni momento. Sono orgoglioso di averli come esempio e come guida. Non posso dimenticare mio fratello, Alessandro, con cui ho condiviso tante esperienze, che mi ha dimostrato il suo amore, che mi ha fatto divertire e sorridere. Non mi scorderò mai dei preziosi consigli che mi hai dato. Sei un fratello fantastico e sono fiero di te. Un ringraziamento speciale va anche al mio cane Tobia, che mi ha regalato il suo amore incondizionato, il suo affetto sincero e la sua allegria. Sei stato una fonte di ispirazione e di forza per me.

Ringrazio con affetto i miei nonni, che sono stati un esempio speciale da seguire, pieni di esperienze e consigli saggi, amore e benevolenza. Sono stati sempre presenti e mi hanno trasmesso i valori importanti della vita. In particolare, vorrei ringraziare nonno Enzo e nonna Oriana per i momenti divertenti, le pasque in vostra compagnia e la disponibilità illimitata che mi avete dato. Siete dei nonni meravigliosi e vi voglio bene. Ringrazio con amore mia nonna Lidia, che mi ha insegnato la responsabilità, che mi ha raccontato le sue storie colme di consigli e suggerimenti, che mi ha ospitato per le cene di Natale. Sei una nonna speciale e ti sono grato. Ringrazio anche mio nonno in cielo, Francesco. Sono sicuro che sei stato un grande uomo; molti parenti mi paragonano a te, quindi non posso non nutrire grande rispetto e grande approvazione. Un grazie sincero va a tutti i miei parenti, che mi hanno dato il loro supporto, che mi hanno invitato per i pranzi e le cene in compagnia, che mi hanno mostrato il loro affetto.

Un ringraziamento speciale va alla mia fidanzata, Chiara, che è stata una fonte inesauribile di stima, forza, passione e sostegno. Gli anni universitari passati con lei sono stati molto più leggeri e felici. Ti ringrazio anche per le serate passate insieme, per le uscite e per i bacini. Ti sono grato di tutto cuore. È doveroso ringraziare il mio amico d'infanzia Yuri. Abbiamo condiviso tante avventure, da piccoli, le giornate a giocare a Mario e Luigi o a Garfield, fino a ora, quindi alle uscite con gli altri e alle serate in compagnia. Non mi scorderò mai delle discussioni sui nostri racconti, delle lunghe camminate fino alla Rocca e dello sfiancante giro in bici per evadere dalla società. Sei un amico fantastico e ti sono grato. Un grazie sincero va al mio amico Eduard, amico leale che mi ha aiutato nel momento del bisogno. I bei momenti nostalgici passati insieme vanno dall'autobus, alla palestra, fino alle uscite a Santa Maria e Foligno. Un grazie anche alla sua fidanzata Rosatea, che è stata anche lei una grande amica. Siete una coppia fantastica e vi voglio bene. Non posso dimenticare il mio amico Mattia, conosciuto come Zibo, considerato un grande amico per me. Insieme a lui non può mancare il

divertimento, lo spasso e le risate a crepapelle. Non mi scorderò mai delle nostre avventure principali fatte insieme al gruppo: la famosissima penna sotto la sedia, l'insegnante di matematica, il commissariato e il 13 aprile. Sei un amico fantastico e ti rispetto molto.

Vorrei ringraziare tutti i miei amici di Foligno, senza di loro la scuola superiore sarebbe stata molto più noiosa e difficoltosa. Non si scordano le serate che abbiamo passato insieme in giro per Foligno, i giorni che non andavamo a scuola e le feste in discoteca che abbiamo fatto. In particolare, ringrazio Lorenzo, per il supporto che mi forniva, le pazze giornate a casa sua, i drink e le notti passate assieme, per la sua sincerità. Ringrazio Niccolò, un amico molto fedele, sincero e responsabile, dai primi giorni giocando a basket ed a ping-pong fino agli ultimi, con le serate in giro per tutte le città ed i paesini. Ringrazio Antonio, amico affidabile e sincero, per le serate in compagnia ed i continui discorsi su chi è migliore tra Icardi e Lukaku con Zibo. Ringrazio Anna, Gaia e Valentina, tre amiche molto unite tra loro, simpatiche, che riescono sempre a tirarti su di morale. È doveroso ringraziare tutti i miei amici di Santa Maria degli Angeli, per tutte le giornate giocate a calcio, le partite di Yu-Gi-Oh e le serate passate in compagnia. In particolare, vorrei ringraziare il mio amico Ilsi e suo fratello Florence, complici, insieme al sottoscritto, di serate divertentissime insieme al gruppo. Ringrazio Francesco, chiamato Dem, per tutte le skill che mi ha fatto imparare sulla mia pelle e le lunghe passeggiate per il nostro paese. Ringrazio Dickson, Major e Ugo, per le partite a calcio nella piazza vicino la basilica.

Un grazie sincero va anche a tutti i miei amici dell'università, per i continui aiuti nello studio e per i momenti esilaranti passati insieme. In particolare, ringrazio Tommaso, un amico che ti aiuta nel momento del bisogno, per l'aiuto nello studio e per la risoluzione di problemi software. Ringrazio Alessandro, detto "il Biondo", per i continui viaggi fatti insieme attraversando Collestrada, le battute sui grilli e per tutta la nuova musica che mi ha fatto ascoltare. Ringrazio Mattia, detto "il Maestro" per le continue battute sul suo pollame, per gli aiuti sovraumani sullo studio e per la disponibilità che lo contraddistingue. Ringrazio Marilù ed Emanuele, quest'ultimo detto "il Fresco", per le divertenti battute fatte insieme e per i momenti indimenticabili insieme al gruppo, soprattutto al McLoud. Ringrazio Alessandro, un amico assai pazzo, per le continue barzellette e battute a freddo. Ringrazio Bilal, detto "Billy" e Sebastiano, per le indimenticabili serate su Discord passate insieme alla vecchia compagnia a giocare, soprattutto ad Among Us, Krunker e allo psicologo. Ringrazio Maria, amica affidabile e simpaticissima che sa incrociare perfettamente il tempo dedicato allo studio a quello dedicato all'amicizia. Ringrazio Federico, per le continue battute sulla Svezia e sul regalo di Aldo, oltre ciò, amico molto sincero e affidabile. Ringrazio Francesco, detto "Jimbo", per la sua amicizia, la sua sincerità e per la sua dedizione allo studio. Ringrazio Matteo, detto "il King", per le continue battute sulle sue bambine, per gli aiuti nel lato hardware e per le uscite insieme al gruppo. Ringrazio Stefan, per la sua simpatia e le sue continue e divertenti battute sul Maestro. Ringrazio William per l'ispirazione che mi dava riguardo al mio racconto e per i continui e divertenti meme che ci inviamo su Instagram. Ringrazio Stefano per le continue battute su Foligno e per essere un amico molto simpatico e divertente. Ringrazio Marta per le poche ma divertenti serate passate assieme. Ringrazio Matteo, detto "Meth sei la mia Meth" per il supporto che dava nello studio e per i pranzi passati assieme.

Indice

Introduzione

L’obiettivo di questa tesi è la creazione di un algoritmo per risolvere il problema del Constrained Set Visualization, ovvero la visualizzazione di insiemi con posizione dei punti vincolate. Dato un insieme di punti nel piano, ciascuno appartenente ad uno o più insiemi, si vuole visualizzare i diversi insiemi di punti connettendo con delle curve o racchiudendo in una regione chiusa tutti i punti di ciascun insieme. I punti in input potrebbero essere, ad esempio, ristoranti di una città che potrebbero essere raggruppati in base al tipo di cucina offerta (ristoranti gourmet, pizzerie, ristoranti orientali, ecc.). Si noti che ogni punto può appartenere a più insiemi; ad esempio, un ristorante potrebbe essere una pizzeria ma anche offrire piatti di cucina tipica. L'obiettivo della visualizzazione è mostrare in maniera chiara e leggibile i diversi insiemi. A questo proposito le curve (o le regioni) che rappresentano gli insiemi dovrebbero, ad esempio, avere pochi incroci e pochi piegamenti. In questa tesi è stata progettata e realizzata una pipeline algoritmica per risolvere il problema precedente. La pipeline si compone di due algoritmi: il primo determina le connessioni tra i punti definendo un opportuno grafo ed un suo linear layout; il secondo algoritmo, partendo da tale linear layout, e utilizzando una tecnica di point-set embedding realizza la visualizzazione vera e propria. La tesi si articola in cinque capitoli:

Il primo capitolo discute in modo più dettagliato del problema affrontato. Parla dello stato dell’arte in merito al problema descritto e discute di alcune tecniche di visualizzazione, più o meno complesse, per far capire l’intento che si vuole raggiungere.

Il secondo capitolo parlerà dell’effettiva pipeline algoritmica creata. La prima sezione parlerà della composizione generale della pipeline, poi le sezioni successive discuteranno i singoli passi della pipeline. Ci si soffermerà prima sulla progettazione di ogni step, poi sulla implementazione del codice corrispondente.

Nel terzo capitolo si descrive la tecnologia utilizzata per implementare un semplice sistema di visualizzazione che, oltre ad implementare la pipeline algoritmica, poi prevede una interfaccia che premette di mostrare all’utente la visualizzazione prodotta. Nella descrizione dell’interfaccia vengono descritti sia gli elementi visualizzati che la costruzione della stessa.

Il penultimo capitolo descrive un’attività sperimentale svolta per valutare l’efficacia e l’efficienza degli algoritmi implementati. Vengono descritti gli obiettivi sperimentali e si descrive in modo dettagliato il setting sperimentale. Infine, nell’ultima sezione verranno discussi i risultati ottenuti dall’esperimento, con le regole che sono state descritte nella sezione precedente.

L’ultimo capitolo tratterà delle conclusioni della tesi e di possibili sviluppi futuri.

Capitolo 1. Constrained Set Visualization

1.1 Problema affrontato

L’esplorazione di grandi spazi informativi richiede spesso il raggruppamento degli elementi in base alle loro proprietà. Ad esempio, gli scienziati affrontano l’analisi di grandi reti sociali raggruppando le singole persone in comunità e studiando poi come questi gruppi interagiscano tra loro. Un’analisi simile è un compito chiave in molti altri ambiti, dall’identificazione di parole correlate in linguistica allo studio delle relazioni tra luoghi geografici. Gli insiemi o le collezioni di oggetti si presentano naturalmente in molti contesti. Possiamo parlare di diversi tipi di ristoranti, modelli di automobili, unioni di Paesi e così via. Un insieme di dati può avere una serie di insiemi di questo tipo: i Paesi si trovano in un continente, fanno eventualmente parte dell'UE, della NATO, dell'ONU e/o di vari sindacati, oppure sono raggruppati in base ad altre caratteristiche come le emissioni di anidride carbonica, il prodotto interno lordo, l'utilizzo di Internet. Se vogliamo analizzare come questi gruppi si relazionano tra loro, siamo tipicamente interessati a trovare intersezioni (quali Paesi dell'UE hanno basse emissioni di anidride carbonica), unioni (quali Paesi hanno un PIL elevato o fanno parte della NATO) e differenze (quali Paesi europei non fanno parte dell’UE) [5]. La rappresentazione visiva degli insiemi e dei loro elementi può portare gli analisti a identificare efficacemente le proprietà di un elemento e le sue relazioni con quelli circostanti. La visualizzazione degli insiemi e dei loro elementi è un tema ricorrente nella visualizzazione delle informazioni (Figura 1).

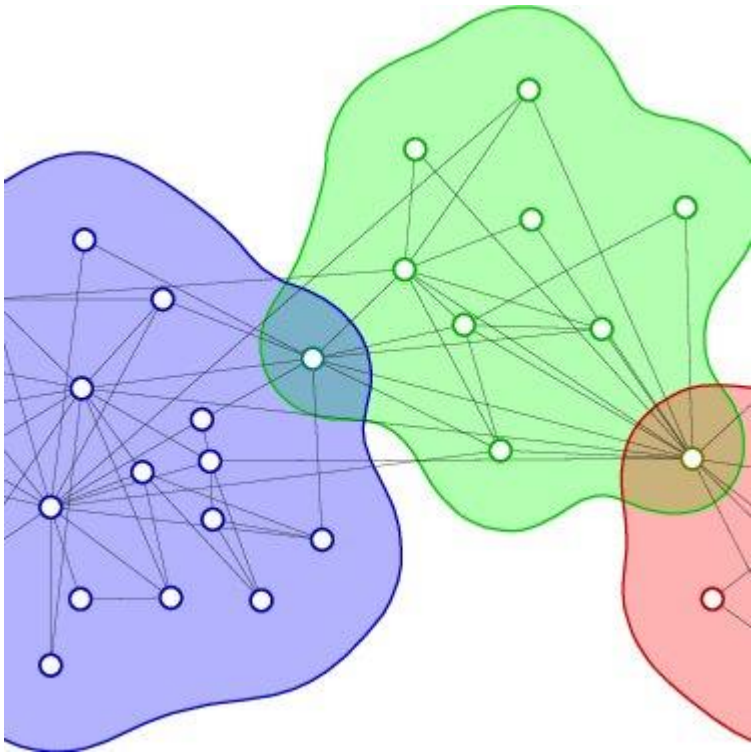

Figura 1: i diversi colori mostrano l'appartenenza a un insieme. Come si può notare un punto può appartenere anche a più insiemi.

Gli approcci meno recenti utilizzano forme molto rade per la rappresentazione degli insiemi, come LineSets, Kelp Diagrams e KelpFusion. Questi metodi cercano di ridurre il disordine visivo riducendo la quantità di "inchiostro" necessaria per collegare tutti gli elementi di un insieme. Sebbene i risultati ottenuti siano gradevoli alla vista, non utilizzano la quantità ottimale di “inchiostro” [3]. Questo problema viene chiamato constrained set visualization, ai punti vengono assegnati dei vincoli per la loro posizione. In questo caso l’input è un insieme di punti fissati nel piano che possono appartenere a diversi gruppi di insiemi. I punti possono appartenere ad uno o più insiemi, quindi vengono colorati in base agli insiemi a cui appartengono. Si deve disegnare una visualizzazione di questi insiemi che sia conforme alle regole di appartenenza e che sia facilmente visualizzabile dall’utente.

Naturalmente per rendere la visualizzazione più leggibile agli occhi del lettore si devono evitare, o quanto meno limitare alcuni elementi disturbanti, quali incroci, angoli ed aree occupate inutilmente. Ciò può essere migliorato creando una struttura concettuale a monte della visualizzazione che riordina l’input per creare una combinazione di insiemi vantaggiosa al fine di eliminare o limitare gli elementi di disturbo descritti sopra. Ci si è serviti di un linear layout come struttura concettuale che, se planare, cioè se non presenta incroci, diventa un book embedding (Figura 6). Il linear layout preso in considerazione è sostanzialmente una visualizzazione astratta degli elementi. Come abbiamo discusso prima, l’insieme dei punti in input contiene elementi vincolati per posizione; quindi, non si può lavorare nel riordinamento dei punti. Si può lavorare nel posizionamento di certi elementi (linee) che fanno capire qual è l’insieme che si sta prendendo in considerazione. Banalmente basta unire i punti dalle posizioni vincolate con queste linee (anch’esse colorate in base all’insieme che prendono in considerazione) in modo tale da rendere la visualizzazione semplice. La struttura concettuale considerata per questa pipeline algoritmica sarà descritta approfonditamente in seguito (sezione 2.2).

1.2 Stato dell’Arte

Esistono molti modi per rappresentare gli insiemi e i loro elementi. Tra le rappresentazioni più antiche (e probabilmente le più familiari per molti) ci sono i diagrammi di Venn e la loro generalizzazione, i diagrammi di Eulero. Queste rappresentazioni visive trasmettono in modo naturale ed efficace l’intersezione di un piccolo numero di insiemi attraverso semplici regioni sovrapposte. Quando si ha a che fare con un numero maggiore di insiemi e intersezioni più complesse, l’uso di forme semplici come le ellissi non è più possibile. Le soluzioni più comuni propongono di utilizzare forme più complesse per racchiudere gli elementi degli insiemi, ad esempio i Bubble Sets (Figura 2). Tuttavia, quando si rappresenta l’appartenenza a un insieme di luoghi segnati su una mappa, ad esempio, o altre visualizzazioni in cui non c’è libertà di riorganizzare gli elementi, queste tecniche più semplici generano molta occlusione.

Delle soluzioni hanno proposto geometrie di contenimento più minimali:

- I LineSet, che riducono la geometria a una singola linea continua (Figura 2);
- I Kelp Diagrams, che utilizzano uno sparse spanning graph, essenzialmente un minimal spanning tree con alcuni spigoli aggiuntivi scelti con cura (Figura 2);
- I Bubble Sets, annunciati prima.

Tutte le tecniche che si riportano cercano di ridurre il disordine visivo e di chiarire le regioni in intersezione in modi diversi [1].

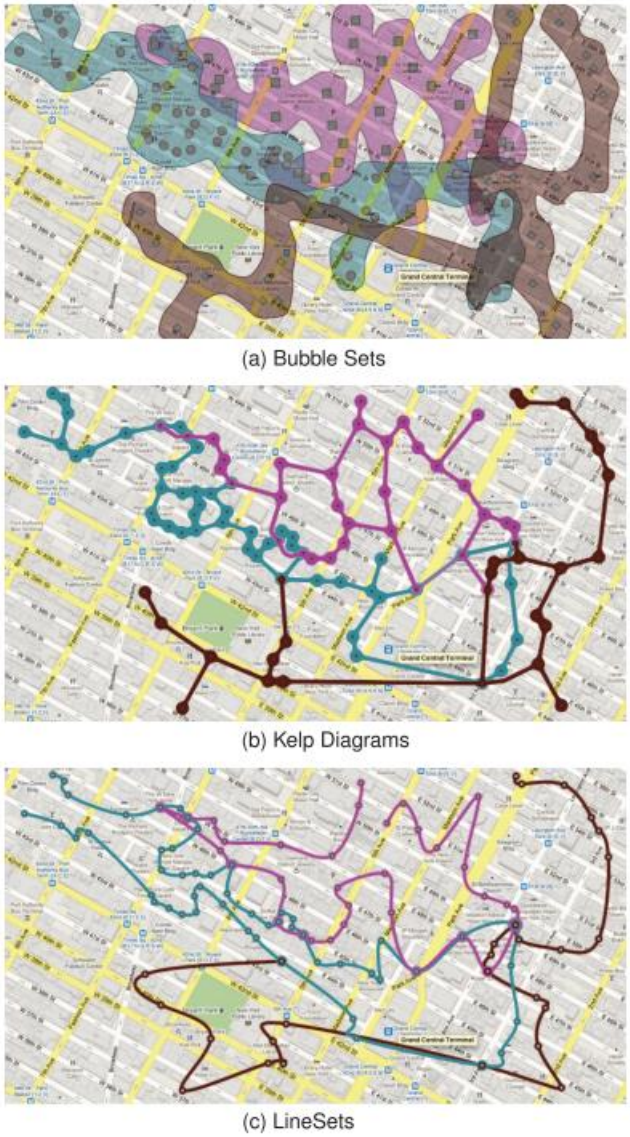

Figura 2: visualizzazioni dei vari metodi discussi precedentemente. (a) Immagine generata utilizzando l'implementazione di Bubble Sets. (b) Immagine dell'implementazione dei diagrammi di Kelp. (c) Immagine generata utilizzando l'implementazione di LineSet.

1.2.1 Diagrammi di Venn o di Eulero

Un diagramma di Venn (detto anche diagramma di Eulero-Venn) (Figura 2) è un diagramma che mostra tutte le possibili relazioni logiche tra una collezione finita di insiemi differenti. Questo metodo è stato proposto nel 1880 dal matematico inglese John Venn. Un diagramma di Venn è composto da multiple curve chiuse (di solito cerchi) che si sovrappongono. I punti all'interno di una curva etichettata *S* rappresentano elementi dell'insieme *S*, mentre i punti all'esterno rappresentano gli elementi che non fanno parte di *S*. Così, per esempio, l'insieme di tutti gli elementi che sono membri di entrambi gli insiemi *S* e *T* ($S \cap T$) è visivamente rappresentato dall'area dove si sovrappongono le regioni *S* e *T*. Nei diagrammi di Venn, a differenza dei diagrammi di Eulero, le curve si sovrappongono in tutti i modi possibili, mostrando tutte le 2^n possibili relazioni tra gli insiemi; occorre infatti anche considerare la zona esterna a tutte le regioni [2]. I diagrammi di Venn o di Eulero sono modi popolari per rappresentare visivamente le intersezioni di insiemi. In questi diagrammi, le curve chiuse corrispondono a insiemi e le sovrapposizioni tra le curve indicano le intersezioni. Diversi lavori hanno esplorato il problema del disegno automatico dei diagrammi di Eulero per trasmettere la topologia astratta degli insiemi, come ad esempio Simonetto e Auber e Stapleton et al. Altri approcci hanno studiato la possibilità di aggiungere etichette o glifi ai diagrammi di Eulero per rappresentare i singoli membri dell'insieme, cioè i dati contenuti nell'insieme. Come discusso da Simonetto et al., ciò richiede che le regioni dell'insieme e le aree di intersezione tra di esse siano sufficientemente grandi da racchiudere tali etichette. La leggibilità delle regioni altamente sovrapposte diventa rapidamente un problema. In uno studio controllato, Henry e Dwyer hanno scoperto che è vantaggioso mostrare le intersezioni utilizzando regioni di insiemi semplici e un contenimento rigoroso, abilitato dalla duplicazione o dalla suddivisione degli insiemi [1].

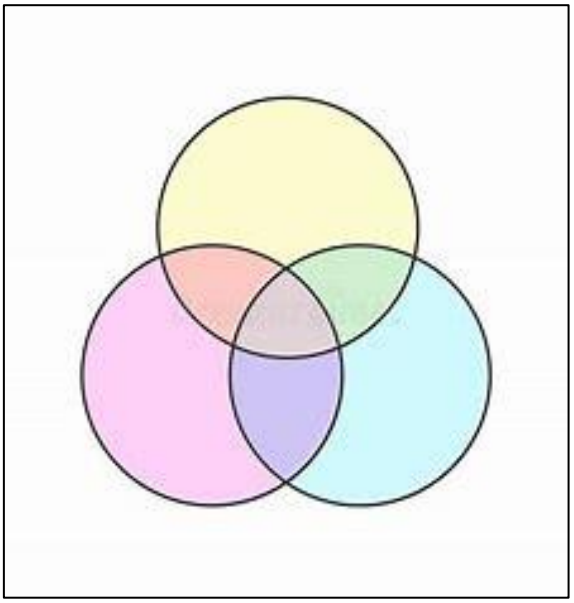

Figura 3: diagramma di Venn o di Eulero [2].

1.2.2 LineSets e diagrammi di Kelp

È una tecnica che tenta di migliorare la leggibilità delle intersezioni di insiemi complessi e di ridurre al minimo il disordine visivo complessivo, riducendo le regioni degli insiemi a semplici linee curve tracciate attraverso

gli elementi dell'insieme. L'uso di una singola linea continua induce un ordine sequenziale in cui scorrere gli elementi di un insieme, trasmettendo un diverso concetto di raggruppamento. I diagrammi di Kelp incorporano i classici grafi "bolla e bastone" o di chiavi ad albero sui punti membri di un insieme. Le regioni che si sovrappongono vengono disegnate simultaneamente con un contenimento strettamente annidato [1].

1.2.3 Implementazioni avanzate: Kelp Fusion

La tecnica Kelp Fusion è una tecnica ibrida che utilizza un mix di scafi e di linee e genera confini adattati per i gruppi di elementi in una determinata posizione. Questa tecnica ibrida è interessante da esplorare perché si colloca tra due estremi desiderabili. Kelp Fusion introduce l'uso di un grafo di prossimità, il cosiddetto grafo del percorso più breve. Nel contesto della scienza dell'informazione geografica, i grafi a percorso più breve sono utilizzati per delineare regioni imprecise, ricostruendo il confine di una regione sulla base di punti che probabilmente si trovano all'interno della regione prevista. I grafi a percorso più breve si adattano a insiemi di punti di densità variabile e mirano a catturare la forma e i cluster di un insieme di punti. In altre parole, l'uso dei grafi a percorso più breve consente a KelpFusion di riempire le facce quando i punti sono spazialmente vicini [1].

1.2.4 Implementazioni avanzate: Pivot Paths

Pivot Paths (Figura 4) è una visualizzazione recente interattiva per l'esplorazione di risorse informative sfaccettate. Durante il lavoro e il tempo libero, interagiamo con molti spazi informativi contenenti sfaccettature e relazioni, come autori, parole chiave e citazioni di pubblicazioni accademiche, o attori e generi di film. Per navigare in queste risorse interconnesse, di solito si selezionano elementi da elenchi che comportano bruschi cambiamenti da un sottoinsieme di dati a un altro. Può essere difficile vedere come le sfaccettature e gli elementi siano in relazione e comprendere l'effetto delle interazioni. Pivot Paths, invece, espone le relazioni tra le sfaccettature come percorsi visivi che invitano l'osservatore a "passeggiare" attraverso uno spazio informativo. Le operazioni di pivot vengono effettuate attraverso interazioni leggere che innescano transizioni graduali. L'interfaccia è stata progettata per la navigazione casuale delle collezioni in un modo esteticamente gradevole che incoraggia l'esplorazione e la scoperta casuale. Il progetto Pivot Paths si basa su ricerche precedenti riguardanti la ricerca di informazioni attraverso l'interazione e la visualizzazione. L'obiettivo è quello di combinare l'interrogazione e la navigazione per soddisfare le esigenze informative in modo più completo. Questa idea è strettamente collegata all'idea di "flaneur dell'informazione", una figura che esplora gli spazi informativi utilizzando le visualizzazioni per scoprire informazioni in modo casuale. Le interfacce tradizionali per l'accesso a grandi collezioni di artefatti digitali o fisici si basano principalmente sui metadati per la ricerca, la navigazione e il filtraggio. Tuttavia, mentre le interfacce a faccette offrono modi potenti per

filtrare le risorse, le relazioni tra le faccette e le risorse sono spesso trascurate. Pivot Paths cerca di migliorare questo aspetto e rappresentare in modo più completo le relazioni tra le faccette e le risorse [6].

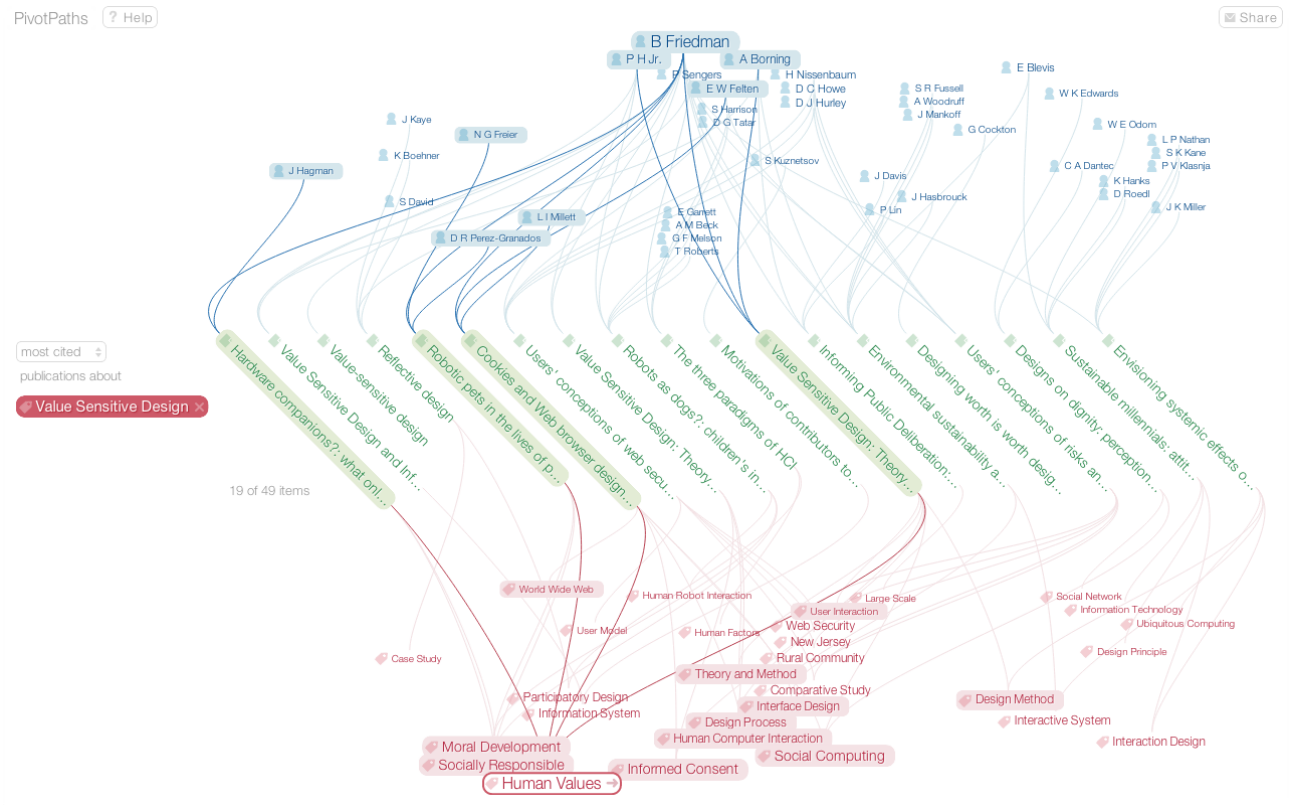

Figura 4: rappresentazione tramite Pivot Paths [6].

Capitolo 2. Pipeline Algoritmica

2.1 Descrizione della Pipeline

La pipeline si compone di due algoritmi:

- Algoritmo per la realizzazione del linear layout;
- Algoritmo per la costruzione geometrica.

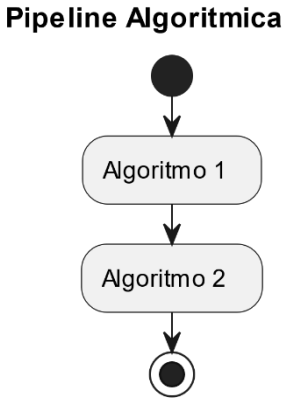

Figura 5: pipeline algoritmica che mostra come i due algoritmi comunicano tra loro.

Il primo algoritmo riceve come input l'insieme dei punti e produce in output un linear layout di un grafo. Un linear layout di un grafo può essere visto come un disegno del grafo in cui i vertici sono disposti lungo una retta ℓ (ad esempio orizzontale) e gli archi sono partizionati in diversi insiemi ognuno disegnato su un semipiano avente la retta ℓ come origine. I diversi semipiani vengono chiamati *pagine*. Tipicamente, gli archi assegnati alla stessa pagina devono rispettare determinati vincoli, ad esempio non devono incrociarsi. In questo caso si parla di *book embedding*. In alcuni casi si permette che gli archi siano divisi in più "pezzi" ognuno dei quali può essere assegnato ad una pagina diversa. Si parla in questo caso di *linear layout topologico* e *book embedding topologico*. Il nostro algoritmo mira a calcolare un *book embedding* topologico su due pagine che viene poi utilizzato dal secondo algoritmo per produrre la vera e propria visualizzazione utilizzando una tecnica di *point-set embedding*. Nel problema del *point-set embedding* si ha in input un grafo ed un insieme di punti e si vuole ottenere un disegno di un grafo in cui i vertici sono rappresentati dai punti dati e il disegno soddisfa determinate proprietà (planarità, numero limitato di piegamenti lungo gli archi, ecc.). Diverse tecniche di point-set embedding utilizzano un book embedding topologico su due pagine come struttura per produrre il disegno desiderato. In particolare, l'assenza di incroci nel book embedding topologico implica l'assenza di incroci nel disegno finale. Come già detto, l'Algoritmo 1 costruisce un grafo cercando di realizzarne un book embedding topologico su due pagine al fine di garantire una visualizzazione senza incroci; ciò però non è sempre possibile

in quanto l'appartenenza dei punti di input ai vari insiemi da visualizzare può essere tale che, indipendentemente dal modo in cui gli archi vengono aggiunti, si ottiene un grafo non planare e che quindi non ammette un *book embedding* topologico su due pagine. Per questo motivo, l'Algoritmo 1 produce in generale linear layout topologici che non sono *book embedding* topologici.

Nella teoria dei grafi, il book embedding (Figura 6) è una generalizzazione dell'embedding planare di un grafo all'embedding in un libro, un insieme di semipiani aventi tutti la stessa linea di confine [4]. Di solito, i vertici del grafo devono giacere su questa linea di confine, chiamata spina dorsale, e gli spigoli devono rimanere

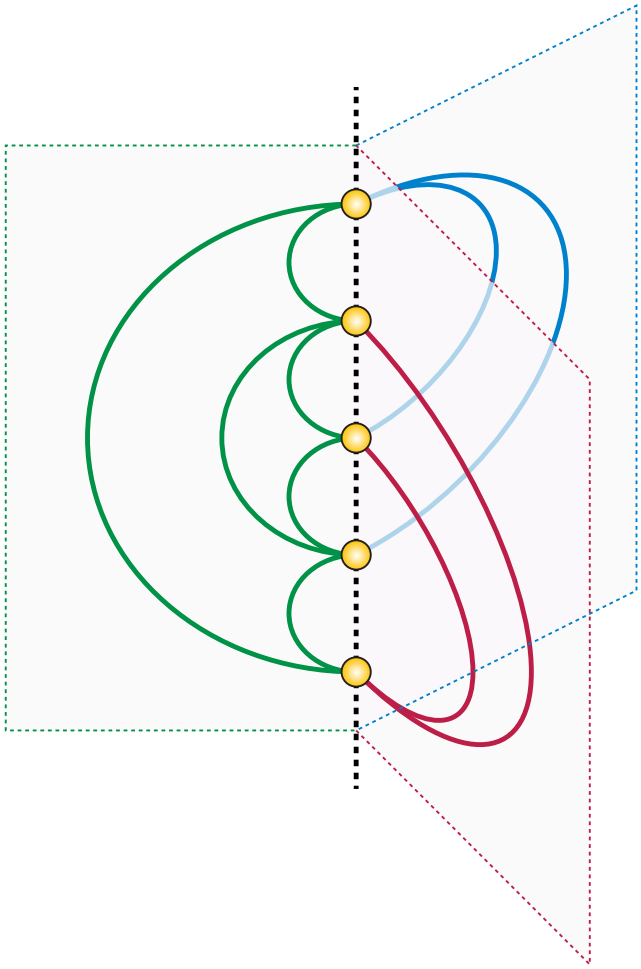

Figura 6: esempio di book embedding di spessore tre. Ha tre pagine: il semipiano che contiene archi rossi, il semipiano che contiene archi blu e il semipiano che contiene archi verdi.

all'interno di un singolo semipiano. Lo spessore del libro di un grafo è il minor numero possibile di semipiani per un qualsiasi incorporamento del grafo, nel nostro caso lo spessore del libro è due, perché esistono solo due semipiani, la pagina sopra e la pagina sotto. Lo spessore del libro è anche chiamato numero di pagina, numero di stack. Ogni grafo con n vertici ha uno spessore del libro al massimo $\lceil \frac{n}{2} \rceil$, e questa formula fornisce l'esatto spessore del libro per i grafi completi. Il book embedding trova applicazione anche nel disegno di grafi, dove due degli stili di visualizzazione standard per i grafi, i diagrammi ad arco e i layout circolari, possono essere

costruiti utilizzando i book embedding. Altre applicazioni dei book embedding includono l'algebra astratta e la teoria dei nodi.
La struttura ottenuta, cioè il linear layout, soltanto nel caso planare sarà un book embedding. L'algoritmo di costruzione di questa struttura sarà approfondito nella parte 2.2. Il secondo algoritmo riceve in input la struttura linear layout e produce in output le coordinate di tutti i punti e degli archi con relativo colore, in modo tale da rendere molto semplice la visualizzazione della struttura creata (Figura 5).

2.2 Architettura del codice generale

Il codice che si è scritto è composto da tre package e due classi all'infuori dei package:

- Package *view*, cartella utilizzata per raccogliere le classi utili alla visualizzazione della finestra grafica;
- Package *algorithm*, cartella utilizzata per raccogliere le classi utilizzate per il calcolo della pipeline algoritmica;
- Package *utilities*, cartella utilizzata per raccogliere le classi utilizzate come supporto;
- Classe *Main*, ha lo scopo di far partire il codice scritto visualizzando anche la finestra grafica;
- Classe *Experiment*, l'unico scopo è di far eseguire il codice molte volte per effettuare sperimentazioni, non visualizzerà la finestra grafica.

Ogni package comunica con gli altri attraverso delle interfacce. Esse hanno lo stesso nome del package con la "I" davanti. Queste interfacce implementano delle classi dentro ai rispettivi package chiamate come le cartelle che le racchiudono. Si proseguirà con una breve discussione di ogni classe dentro i package, iniziando dal package *view*:

- *MainGUI*, classe principale di questa cartella che ha il compito di disegnare e controllare le geometrie nel *JPanel*, implementa molte classi, ad esempio *ActionListener* e *MouseMotionListener*;
- *GeneralGUI*, classe specializzata soltanto nella creazione del *JFrame*, del *Container* e dell'implementazione del *KeyListener*.

Nel package *algorithm*:

- *BookEmbedding*, classe principale per la creazione della struttura linear layout, approfondita nei punti seguenti;
- *Drawing*, classe principale per il calcolo delle geometrie del disegno che si vuole creare, anch'essa approfondita nei punti seguenti;
- *Edge*, classe di supporto che definisce cos'è un arco;
- *Node*, classe di supporto che definisce cos'è un nodo;
- *GraphicEdge*, classe di supporto che definisce cos'è un arco grafico;
- *GraphicNode*, classe di supporto che definisce cos'è un nodo grafico;
- *Bend*, classe di supporto che definisce cos'è un angolo.

Nel package *utilities*:

- *AbsolutePath*, questa classe ha lo scopo di calcolare il percorso assoluto della cartella contenente il codice;
- *Assets*, classe che calcola il percorso relativo del file e lo unisce al percorso assoluto della cartella per creare il percorso assoluto del file;
- *ReadTextFile*, classe che permette la lettura di un file;
- *WriteTextFile*, classe che permette la scrittura di un file;

2.3 Algoritmo per la Realizzazione del Linear Layout

Come riportato sopra, il primo macro-algoritmo è l'algoritmo per la realizzazione del linear layout. Esso riceve in input l'insieme dei punti che si vogliono inserire e produrrà in output un linear layout che, se planare, è un book embedding. Si può individuare una sotto-pipeline di operazioni e di algoritmi per la costruzione del linear layout (Figura 7):

1. Algoritmo di ordinamento delle coordinate;
2. Operazione di creazione e di inserimento di tutti i nodi astratti che comporranno il linear layout;
3. Algoritmo di inserimento di tutti gli archi astratti che collegheranno i nodi con stesso colore;
4. Ottimizzazione del linear layout creato.

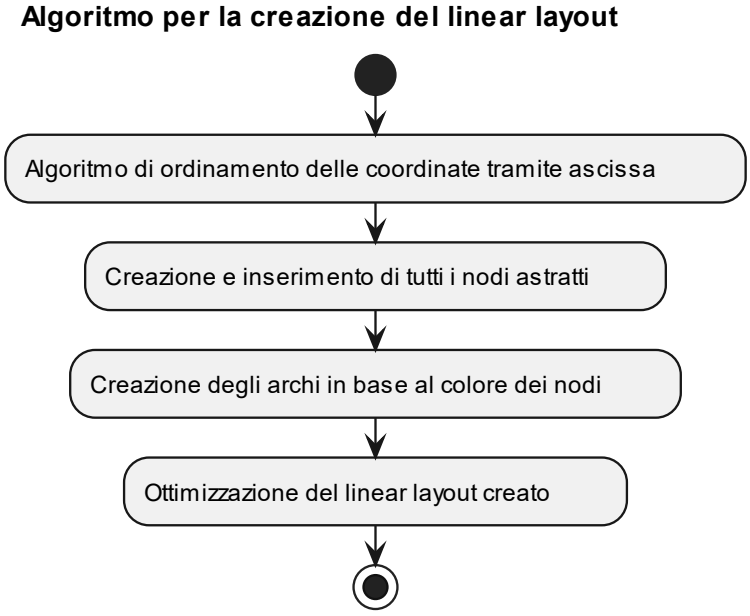

Figura 7: sotto-pipeline algoritmica per la realizzazione del linear layout.

La prima operazione che si deve fare è l'ordinamento delle coordinate dei punti tramite l'ascissa. Per l'ordinamento di questi punti si è deciso di utilizzare l'algoritmo di ordinamento Bubble Sort [7], data la sua semplicità. Per quanto riguarda la terza operazione, cioè l'idealizzazione dell'algoritmo di inserimento degli archi astratti che collegheranno i nodi dello stesso colore, si è deciso di utilizzare questo algoritmo: i colori degli archi vengono enumerati e scanditi. Per il primo colore della lista, gli archi verranno inseriti nella pagina

in alto e collegheranno tutti i nodi che hanno quel determinato colore. Per i colori restanti, se esiste un arco posizionato nella pagina in alto che crea un'intersezione, sarà sicuramente colpa di un nodo interno, cioè il nodo di destinazione dell'arco in mezzo. In questo caso si effettuerà un giro del nodo passando nella pagina in basso. Si creeranno intersezioni solo nel caso in cui un terzo colore farà il giro ad un nodo precedentemente usato come nodo di intorno per il secondo colore (Figura 8).

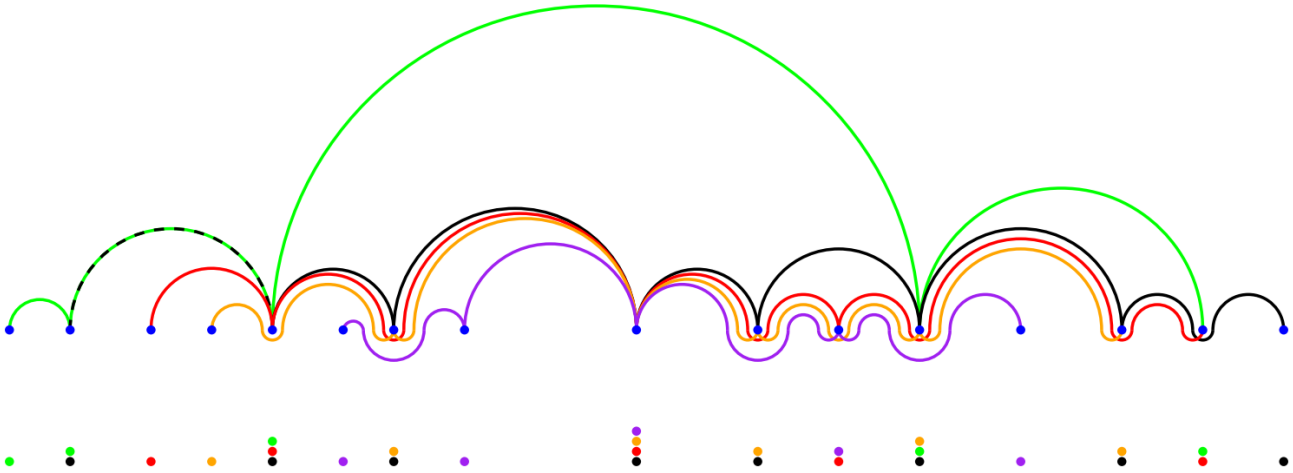

Figura 8: esempio dell'algoritmo utilizzato per costruire gli archi astratti tra i nodi dello stesso colore. In questo esempio vengono utilizzati 16 nodi e 5 colori diversi.

Nella quarta operazione si procederà a ottimizzare il linear layout, rendendolo molto meno ingarbugliato. Se in un arco situato nella pagina in basso non ha nessun arco situato nella pagina bassa che lo sovrasta, l'arco non farà più il giro del nodo, ma passerà semplicemente nella pagina in basso (Figura 9).

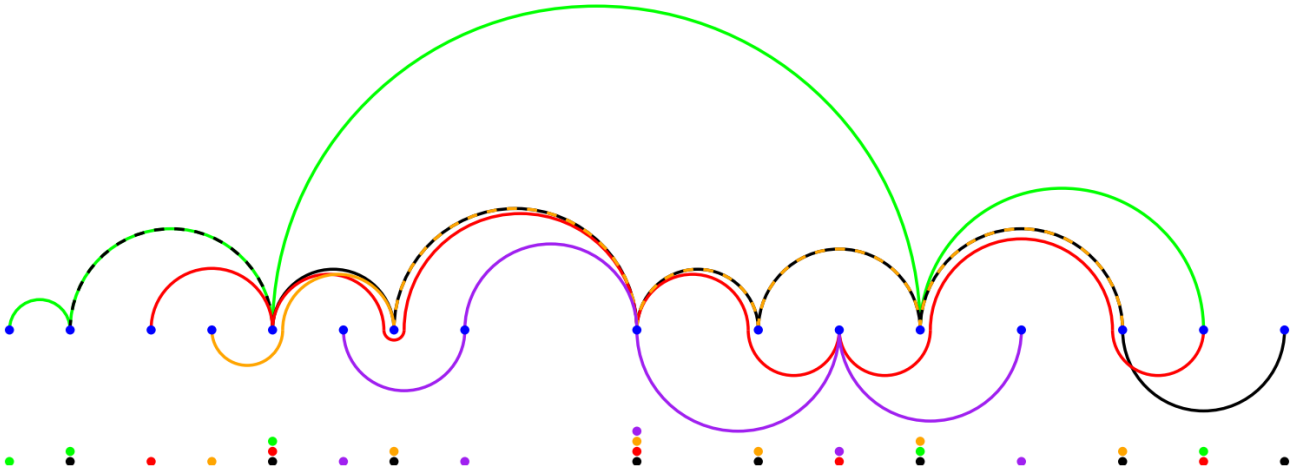

Figura 9: esempio dell'algoritmo utilizzato per raffinare il linear layout e renderlo facilmente visualizzabile. In questo esempio vengono utilizzati 16 nodi e 5 colori diversi.

2.3.1 Realizzazione

Per effettuare la sotto-pipeline algoritmica fissata nella sottosezione precedente, si utilizzerà il costruttore della classe *BookEmbedding* (abuso di notazione, infatti verrà costruito un linear layout che, solo nel caso planare sarà un book embedding). Si riceverà in input a questo costruttore due *LinkedList* di array di interi:

- Una *LinkedList* conterrà tutte le coordinate dei punti che abbiamo sul piano;
- L'altra *LinkedList* conterrà tutti i colori che avranno i punti.

Per il riordinamento tramite l'ascissa si usa un algoritmo di sorting chiamato Selection Sort [7], e viene implementato come segue: due cicli scandiscono i nodi, il ciclo più annidato parte dall'indice del ciclo meno annidato. Un *if* all'interno dei due cicli decide se l'ascissa dell'elemento puntato dal primo ciclo è maggiore dell'ascissa dell'elemento puntato del secondo ciclo. In caso affermativo vengono scambiati i due elementi. In caso negativo non succede nulla. A termine dei due cicli i punti saranno ordinati tramite l'ascissa dal punto con ascissa minore al punto con ascissa maggiore.

Per la seconda operazione vengono creati tutti oggetti *Node* e verranno inseriti nel linear layout grazie al metodo *insertNodeInBE*. Ogni oggetto *Node* avrà un riferimento al nodo che lo precede e gli verranno assegnati i suoi colori grazie alla *LinkedList* che contiene tutti i colori di tutti i nodi.

È arrivato il momento di effettuare la terza operazione, cuore centrale del primo macro-algoritmo insieme all'ottimizzazione, effettuata dal metodo *edgeColors*. Innanzitutto, vengono create n LinkedList, dove n è il numero di colori differenti. Viene inizializzata anche una *LinkedList* chiamata *colorsInsered*, che conterrà i colori che sono già stati considerati. Tre cicli annidati vengono scanditi, i due più esterni servono per scandire ogni nodo di ogni colore, invece, quello più interno controlla se il colore j-esimo del nodo i-esimo è già stato inserito in *colorsInsered*, dato j il numero di colori ed i il numero di nodi. Ogni nodo verrà inserito in una (o più) *LinkedList*, in base al colore (o ai colori) che ha. Se si trova un nuovo colore si allocherà spazio per una nuova *LinkedList* destinata al nuovo colore e verrà aggiunto il colore in *colorsInsered*. Infine, verranno scandite tutte queste *LinkedList* da un ciclo, ed un altro ciclo annidato penserà a scandire tutti i nodi della *LinkedList* k-esima, dato k il numero di LinkedList create. Tramite il metodo *control*, che riceve in input l'ultimo nodo ad avere un arco, la pagina in cui dovrà essere inserito l'arco e la *LinkedList* attualmente scansionata, si capirà se ci sono degli archi che intralciano il percorso dell'arco che stiamo per creare. Se non ce ne sono verrà creato l'arco utilizzando il metodo *insertEdgeInBE*, che inserirà l'arco che va dall'ultimo nodo ad avere un arco al nodo corrente. Ovviamente il nodo corrente diventerà l'ultimo nodo ad avere un arco. Se, invece, ci sono degli archi intralcianti si utilizzerà il metodo *aroundTheNode* per creare un arco che attraverserà la spina dorsale per due volte: con un arco in alto fino ad un nodo fittizio creato subito prima del nodo che intralcia, con un arco in basso fino ad un altro nodo fittizio creato subito dopo il nodo che intralcia, e con un terzo arco che parte dal nodo fittizio e arriva a destinazione, nel nodo analizzato. Così facendo si crea un arco che va dall'ultimo nodo ad avere un arco al nodo corrente, attraversando per due nodi fittizi per aggirare il nodo d'intralcio. Ovviamente il nodo corrente diventerà l'ultimo nodo ad avere un arco.

Infine, l'operazione di ottimizzazione del linear layout, effettuata dal metodo *optimization*. Inizia eliminando gli archi che hanno lo stesso nodo di partenza e di arrivo e gli archi doppi, sia nella pagina in alto, sia nella pagina in basso. Lo fa un iterator sulle due *LinkedList* che contengono gli archi nella pagina alta e gli archi nella pagina bassa. Con un *if* si controlla se il nodo di arrivo è lo stesso del nodo di partenza dell'arco e, in caso affermativo, si procede con l'eliminazione dell'arco. Per gli archi doppi si costruisce un ciclo annidato che scandisce gli archi per la seconda volta e controlla che due archi abbiano:

- lo stesso colore;
- stesso nodo di partenza;
- stesso nodo di arrivo.

Inoltre, devono controllare che non si parli dello stesso arco. Finita questa parte si procede all'ottimizzazione vera e propria che riguarda principalmente la pagina in basso. Se, come sottolineato prima, non esistono archi che sovrastano altri archi nella pagina in basso, allora si possono ottimizzare fondendoli. Un ciclo scandisce solo gli archi della pagina sotto e seleziona gli archi che partecipano ad un attraversamento della spina dorsale. Vengono individuati gli altri due archi che partecipano allo stesso attraversamento e, se esiste un arco della pagina in basso che sovrasta verso il basso gli archi selezionati, il codice passa al successivo. Ma in caso non esistessero si procede alla fusione di questi archi per creare un arco diretto, quindi senza attraversamenti, che vive nella pagina in basso. Il metodo che permette la fusione degli archi è chiamato *mergeDown*. In poche parole, elimina gli archi di attraversamento dalla *LinkedList* della pagina in basso ed inserisce al loro posto il nuovo arco diretto.

2.3.2 Complessità dell'algoritmo

Per quanto riguarda la complessità del primo macro-algoritmo dobbiamo prendere in considerazione solo l'istruzione dominante di tutto il codice. Considerando che l'ordinamento delle coordinate ha una complessità di $\Theta(n^2)$ e la seconda operazione ha una complessità di $\Theta(n)$, dove n è il numero di nodi, escludiamo che l'istruzione dominante si trovi in questa parte di codice. Il metodo *edgeColors* ha una complessità pari a $\Theta(m^2 \cdot n^3)$, dove m è il numero di colori e n è il numero di nodi. Invece la complessità dell'ottimizzazione è pari a $\Theta(a^2 \cdot n \cdot m)$, dove m è il numero di colori e n è il numero di nodi ed a è il numero di archi nella pagina bassa. Nel caso peggiore possibile, possiamo affermare che il numero di archi è pari al numero dei nodi moltiplicato per il numero di colori. Quindi la complessità diventa $\Theta(n^3 \cdot m^3)$, cioè la parte con complessità più alta di tutto il pezzo di codice preso in considerazione, contenente l'istruzione dominante.

2.4 Algoritmo per la Costruzione Geometrica

Come riportato sopra, il secondo macro-algoritmo è l'algoritmo per la costruzione geometrica. Esso riceve in input il linear layout costruito nell'algoritmo precedente e produrrà in output le realizzazioni grafiche che servono per visualizzare correttamente il linear layout, in particolare due *LinkedList*, una per i nodi, l'altra per gli archi, che verranno passate all'interfaccia di disegno. Si può individuare una sotto-pipeline di operazioni e di sotto-algoritmi per la costruzione geometrica (Figura 10):

1. Operazione di assegnazione delle coordinate ai nodi;
2. Creazione delle liste di adiacenza;
3. Calcolo della pendenza;
4. Algoritmo di calcolo degli angoli;
5. Algoritmo di taglio degli archi;

6. Cancellazione dei nodi falsi.

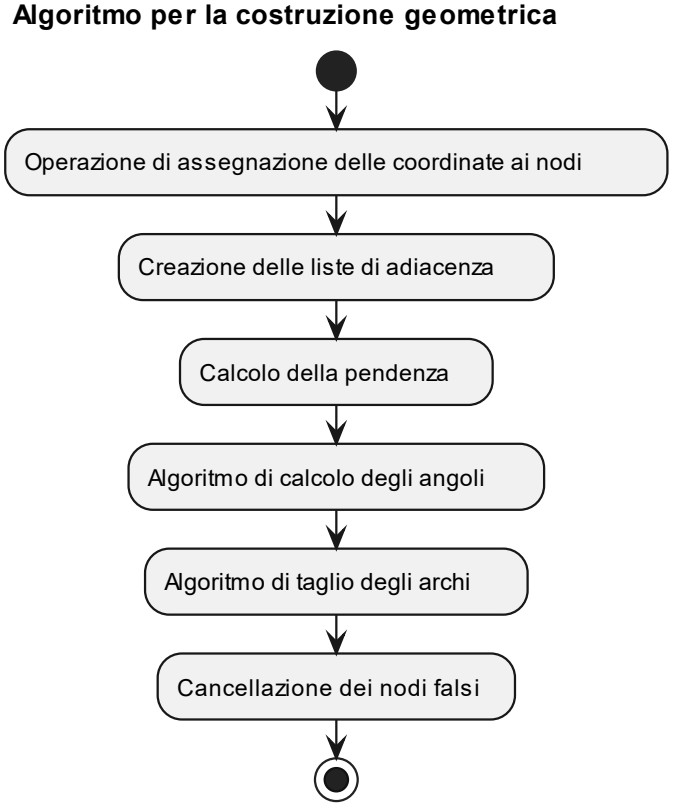

Figura 10: sotto-pipeline algoritmica dell'algoritmo per la costruzione geometrica.

L'operazione di assegnazione delle coordinate assegna:

- ai nodi reali, le coordinate reali date in input al sistema;
- ai nodi fittizi, le coordinate calcolate in base alla loro posizione.

La creazione delle liste di adiacenza è utile per visualizzare correttamente il grafo creato. La rappresentazione con liste di adiacenza di un grafo $G = (V, E)$, V insieme dei vertici ed E insieme degli archi, consiste di un array Adj di $|V|$ liste, una per ogni vertice di G. Per ogni $u \in V$, la lista di adiacenza $Adj[u]$ contiene tutti i vertici v tale che $u, v \in E$. Nel nostro caso G è orientato, quindi l'arco u, v, con u, v vertici del grafo, è rappresentato inserendo v nella lista di adiacenza di u (Figura 11) [8].

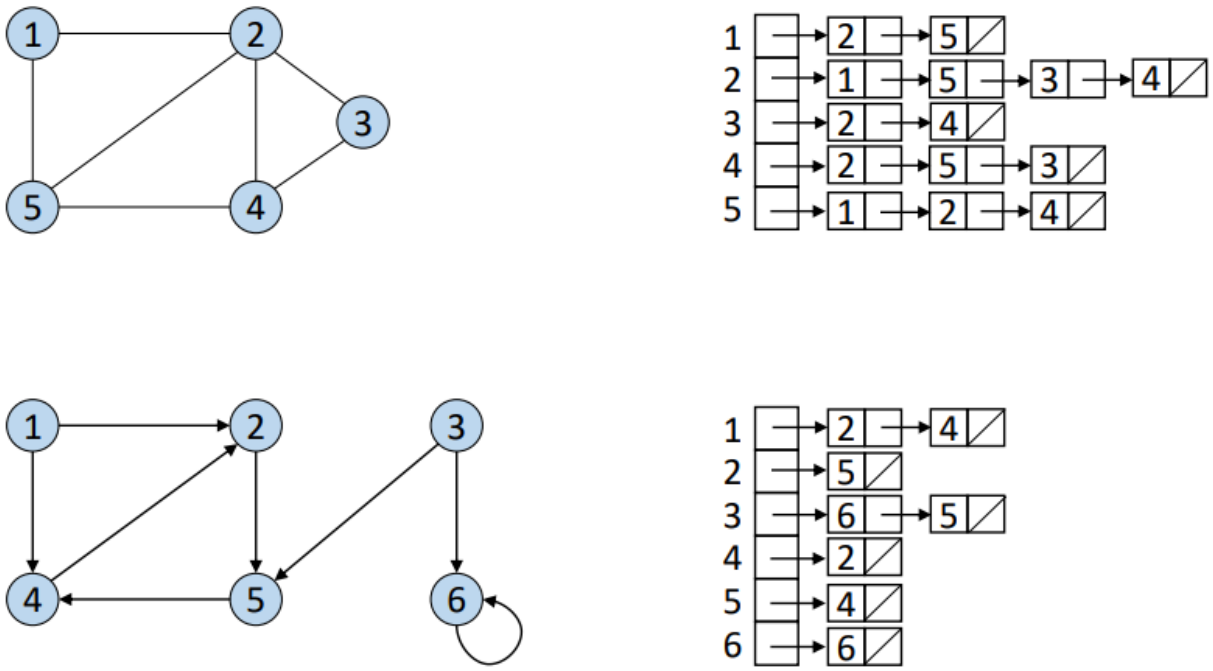

Figura 11: esempio di liste di adiacenza [8].

La terza e la quarta operazione saranno utili a realizzare effettivamente la geometria del grafo. Lo scopo è quello di ricondursi ad una tecnica base della mappatura di un grafo vincolato dalle posizioni dei nodi. La tecnica è descritta di seguito. Sia $G = (V, E)$ un qualsiasi grafo piano con un linear layout C generico, dove V è l'insieme dei nodi ed E l'insieme degli archi. Assumiamo che i vertici di V siano numerati da v_1 a v_n come nel linear layout C, dove n è il numero dei nodi. Il vertice con l'indice più piccolo è scelto in modo tale che il bordo sia su v_1 e su v_n. Sia S un qualsiasi insieme di punti $p_1, p_2, \dots, p_n$, con $p_i = (x_i, y_i)$, con i da 1 ad n. Si supponga che il piano sia ruotato in modo tale da rendere le coordinate x dei punti tutte diverse. Si supponga inoltre che i punti siano ordinati in modo crescente per coordinate x crescenti. Ora possiamo mappare il linear layout $C = (v_1, v_2, \dots, v_n)$ ai punti $p_1, p_2, \dots, p_n$. Tutti gli spigoli di C possono essere tracciati come una linea retta in modo che si estendano monotonamente in direzione x. L'idea è quella di scegliere i segmenti in modo tale che le loro pendenze siano uguali e siano a forma di cono (Figura 12). La pendenza è determinata dalla pendenza massima degli spigoli rettilinei sul percorso di C. Più precisamente, la pendenza per un possibile segmento rettilineo è calcolata da $\sigma' = max_i \frac{|y_{i+1} - y_i|}{x_{i+1} - x_i}$. Per garantire che i segmenti non interferiscano con altri segmenti o punti, dobbiamo aumentare leggermente il valore σ'. Infatti, assegniamo il valore $\sigma = 2 - \sigma'$ come pendenza di una retta passante per p_1 e $-\sigma$ come pendenza di un'altra retta passante per p_n. L'intersezione delle due linee fornisce la posizione della curva. Gli spigoli rimanenti vengono disegnati ciascuno con una sola curva, in modo che tutti i segmenti di sinistra degli spigoli abbiano la stessa pendenza σ e i segmenti di destra abbiano la pendenza $-\sigma$. Correranno in parallelo. Questo è più che altro per ragioni estetiche e per semplificare le argomentazioni per evitare alcuni possibili incroci. Gli spigoli interni a C sono disegnati sopra, e gli spigoli esterni sono disegnati sotto. Si noti che, con questa tecnica, alcuni dei segmenti adiacenti allo stesso vertice

potrebbero sovrapporsi. Abbiamo ideato uno schema di perturbazione che risolve queste sovrapposizioni: Sia la distanza minima tra due segmenti paralleli non sovrapposti, sia L la distanza massima tra due segmenti paralleli non sovrapposti. Sia L la lunghezza massima di un segmento e $maxdeg$ il grado massimo. Per ogni vertice v, ordiniamo i segmenti adiacenti che si sovrappongono in base alla loro lunghezza in modo decrescente. Per ogni segmento s che è l'i-esimo segmento di sovrapposizione adiacente a v, ruotiamo s di $i \cdot \frac{\epsilon}{L \cdot maxdeg}$ verso il basso. I nuovi punti di intersezione dei segmenti forniscono nuove posizioni per le curve. In questo modo ci assicuriamo che i segmenti che si sovrapponevano precedentemente si distribuiscono e si evitano nuove intersezioni, dato che gli angoli di rotazione sono mantenuti sufficientemente piccoli [9].

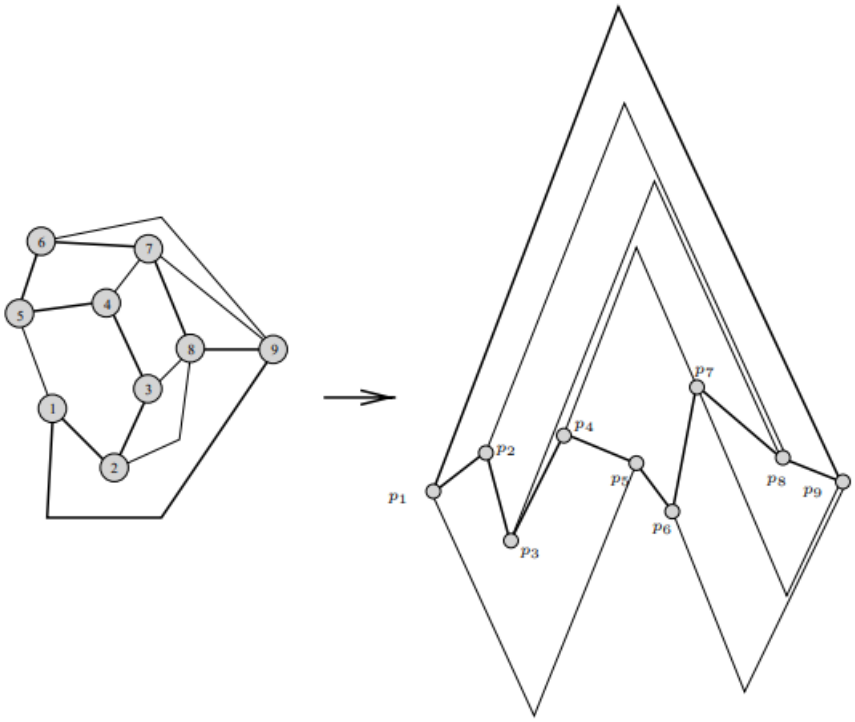

Figura 12: esempio della tecnica utilizzata usando come linear layout un ciclo Hamiltoniano [9].

Il quinto algoritmo della sotto-pipeline algoritmica riguarda l'ottimizzazione dell'area del disegno. Si noti che l'area può essere molto più grande dell'area R occupata dall'insieme di punti. Più precisamente, assumiamo che il rettangolo minimo R sia un quadrato di larghezza W e δ sia la distanza minima in direzione x tra due punti qualsiasi. È evidente che il valore assoluto della pendenza σ dei segmenti del bordo (v_n, v_1) è al massimo $2 \cdot W \cdot \frac{W}{\delta}$ mentre la larghezza rimane la stessa. Questo significa che, se assumiamo coordinate intere ($\delta = 1$), otteniamo un'area di $O(W^3)$ per il disegno. Si noti che, se si consentissero due curve per ogni spigolo, si potrebbero facilmente disegnare l'arco in modo ortogonale e mantenere l'area proporzionale all'area della carena convessa dell'insieme di punti. In questo caso, lo schema di perturbazione non funziona più e potremmo dover ingrandire le dimensioni dei vertici e assegnare degli offset ai segmenti adiacenti (Figura 13) [9].

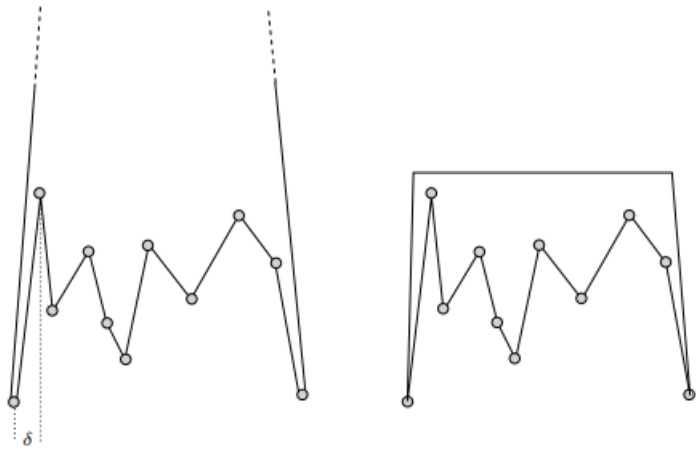

Figura 13: come risparmiare area tagliando l'arco ed inserendo due spigoli al posto di uno [9].

L'ultima operazione fa riferimento alla cancellazione dei nodi falsi, in modo tale da non essere visualizzati dall'interfaccia di disegno.

2.4.1 Realizzazione

Per effettuare la sotto-pipeline algoritmica discussa prima, si utilizzerà il costruttore della classe Drawing, che riceverà in input un oggetto *BookEmbedding* e una *LinkedList* di array di interi:

- L'oggetto *BookEmbedding* rappresenta il linear layout costruito nel macro-algoritmo discusso precedentemente, che conterrà la lista di tutti i nodi, sia veri che fittizi (*nodesList*), e la lista di tutti gli archi, divisa in pagina sopra (*pageUp*) e pagina sotto (*pageDown*);
- La *LinkedList* di array di interi contiene le coordinate dei punti reali (*coordinates*).

Come primo step, si costruisce un ciclo *while* che scandirà tutti i nodi passati come parametro da *BookEmbedding*. Ai nodi reali verrà assegnata la posizione contenuta in *coordinates*, invece, ai nodi fittizi verranno calcolate le posizioni relative grazie al metodo *addDummyNodes*. Grazie ad un doppio ciclo for, si individuano i nodi fittizi consecutivi e si calcola, in base al predecessore e al successore della sequenza di questi nodi falsi, le coordinate che hanno nel piano, cioè equi distanziati tra loro e sulla retta che unisce i due nodi reali detti prima (predecessore e successore).

Per la creazione delle liste di adiacenza, grazie a un altro ciclo *while*, si aggiungono gli archi ai loro rispettivi nodi di partenza utilizzando la *LinkedList* links per ogni nodo. Questo lavoro si esegue prima per la pagina in alto, poi per la pagina in basso. Quindi si salvano in links, gli archi che partono dal nodo corrispondente.

Per il calcolo della pendenza si procede come segue: si scorre tutta la lista di nodi e si calcola la pendenza che c'è tra il nodo i-esimo e il nodo $(i + 1)$-esimo, dove i è un intero che parte da 1 fino al numero di nodi. Nel caso particolare in cui la pendenza è parallela all'asse delle ordinate, si trascura.

Si procede poi al codice dell'algoritmo per la creazione e il calcolo degli angoli grazie al metodo *createBends*, che riceve come parametro la pendenza massima. Innanzitutto, si ordinano le rispettive liste di adiacenza di ogni nodo, dall'arco con nodo di destinazione più vicino, all'arco con nodo di destinazione più lontano. Quindi,

per ogni nodo, un ciclo scandisce tutta la lista di adiacenza, per convenzione gli archi con nodi adiacenti sono stati messi nella pagina in alto; l'arco verrà inserito nella lista finale degli archi da passare al motore grafico senza angoli. Per quanto riguarda gli altri archi, viene invocato il metodo *calculateBend* che riceverà come parametro le coordinate dei due punti dell'arco e la pendenza massima, e produrrà un'oggetto *Bend* che rappresenta l'angolo dell'arco da creare. L'angolo creato viene inserito in una lista chiamata *bendsList* che conterrà tutti gli angoli di tutti gli archi creati. Questo procedimento viene eseguito sia per gli archi della pagina in alto, sia per gli archi della pagina in basso.

Si è deciso di implementare l'algoritmo per il taglio degli archi troppo lunghi nel seguente modo: il metodo *cutBends* risolve il problema scandendo la lista di tutti gli angoli creati. Un secondo metodo chiamato *createSecondLevelBends* calcola i due nuovi oggetti *Bend* che rappresenteranno gli angoli utilizzando l'arco selezionato. Questo metodo valuta anche se è saggio tagliare l'arco in questione. Se lo ritiene idoneo produce in output i due *Bend* e il codice restante del metodo *cutBends* elimina il vecchio arco per sostituirlo con l'arco tagliato. Se non lo ritiene idoneo non produce nulla in output ed il metodo *cutBends* non fa nulla.

L'ultima operazione semplicemente rimuove i nodi fittizi dalla lista dei nodi scandendola e controllando che l'attributo *isReal* contenuto dentro l'oggetto *Node* non sia falso.

2.4.2 Correttezza e complessità dell'algoritmo

Dimostriamo che la geometria ottenuta sia planare. Il seguente caso di verifica dimostra la planarità del disegno: gli spigoli interni ed esterni a C, con C che indica il linear layout, non si incrociano, poiché sono separati dal percorso poligonale $C - e$, detto e arco generico di C.

Il caso di due spigoli e_1 ed e_2 all'interno di C viene illustrato in modo più dettagliato. Sia $e_1 = (v_i, v_l)$ e $e_2 = (v_j, v_k)$. È chiaro che $i \leq j < k \leq l$ è valido per la planarità. Ora, poiché i segmenti di sinistra corrono parallelamente e i segmenti di destra e i quattro punti finali si trovano in quest'ordine sulla poligonale x-monotona, non ci sono segmenti che si incrociano. Lo stesso vale per gli spigoli esterni. Le pendenze dei segmenti degli spigoli non in C sono state scelte sufficientemente grandi da evitare che gli spigoli in C possano interferire con gli spigoli non in C [9]. Per quanto riguarda la complessità, la rappresentazione con liste di adiacenza ha un'occupazione di memoria pari a $\Theta(n + m)$, essendo $n = |V|$ e $m = |E|$. L'array ha dimensione $\Theta(n)$. Le liste di adiacenza hanno complessivamente un numero di elementi pari a $\sum_{v \in V} outdeg(v) = m$ se il grafo è orientato [8]. La pipeline algoritmica risulta lenta nella quarta e quinta operazione (Figura 9) in quanto per le altre operazioni la complessità è una $\Omega(n^2)$, dove n è la cardinalità dei nodi. Nella quarta operazione, invece, la complessità è di $\Theta(n \cdot a^2)$, dato a la cardinalità degli archi, visto che l'istruzione dominante del sotto-algoritmo è nell'ordinamento degli archi per ogni nodo. Nella quinta operazione, l'istruzione dominante è situata nella verifica di idoneità nel tagliare l'arco. La complessità è $\Theta(b \cdot n)$, dove b sono il numero di angoli creati. Sapendo che un arco ha al più un solo angolo prima di questo metodo, allora la complessità è $\Theta(a \cdot n)$. La complessità totale dell'interna pipeline algoritmica è $\Theta(n \cdot a^2)$. Un ulteriore ragionamento: nel caso peggiore in cui tutti i nodi fanno parte di tutti gli insiemi, la cardinalità degli archi è

uguale alla cardinalità dei nodi moltiplicata per il numero di insiemi; quindi, la complessità nel caso peggiore è $\Theta(n^3 \cdot m^2)$, dove m è il numero di colori (numero di insiemi).

Capitolo 3. Interfaccia di Visualizzazione

3.1 Tecnologia utilizzata

Per favorire un'ampia compatibilità su diverse piattaforme hardware e software, l'applicativo implementerà la tecnologia JFC/Swing. L'applicazione è stata ampiamente testata su piattaforme Windows e *NIX-Like.

3.2 Interfaccia

Terminata la compilazione dell'applicativo, l'utente si troverà davanti una finestra *JFrame* con all'interno un *JPanel* che permette di visualizzare il disegno creato. L'interfaccia è contenuta tutta all'interno del package *view*, in particolare nella classe *MainGUI* troviamo il codice che ha lo scopo di visualizzare il disegno. *MainGUI* è una classe che estende *JPanel* e implementa le seguenti classi: *ActionListener*, *MouseWheelListener*, *MouseMotionListener* e *MouseListener*. Queste implementazioni hanno due scopi: disegnare effettivamente nel *JPanel* e ricevere la posizione del mouse e della rotellina in diversi momenti. Il metodo che effettivamente disegna la rappresentazione grafica è *paintComponent*, che riceve come parametro un oggetto Graphics ed invoca tre altri metodi privati:

- *drawZoomButtons*, che raffigura i rettangoli di contorno dei tre pulsanti in basso;
- *drawPoints*, che raffigura i punti della struttura da visualizzare, descritto in modo approfondito nella sottosezione 3.2.1;
- *drawLines* che raffigura le linee della struttura da visualizzare, descritto in modo approfondito nella sottosezione 3.2.2.

Gli elementi sullo schermo, oltre al disegno sono diversi, abbiamo tre pulsanti in basso a destra della finestra. Sono i pulsanti che regolano lo zoom in, lo zoom out e un pulsante reset. Cliccando su zoom in si effettuerà uno zoom in avanti della visualizzazione. Cliccando su zoom out si effettuerà uno zoom in indietro della visualizzazione. Con il pulsante reset, invece, si azzerano tutte le modifiche fatte allo zoom. Oltre a ciò, il pulsante reset avrà un'altra funzione spiegata più avanti. Altri elementi che si notano in basso a sinistra della finestra sono degli oggetti *CheckBox* incapsulati da oggetti grafici *Rectangle2D* (Figura 14). Questi *CheckBox* hanno lo scopo di far visualizzare all'utente solo gli insiemi che vorrà. Ogni insieme è associato ad un colore, cliccando sulla *CheckBox* di un colore, la visualizzazione di quel determinato colore scomparirà, permettendo una migliore facilità di visualizzazione del disegno. Esiste un *CheckBox* per ogni colore all'interno della rappresentazione grafica, utilizzando un array di oggetti *CheckBox*, la dimensione dell'array è la cardinalità degli insiemi presi in considerazione dall'applicativo. Se nella *CheckBox* di un determinato colore c'è la spunta significa che il colore è visibile, naturalmente se la spunta non c'è il colore non è più visibile. Inoltre, per una migliore maneggevolezza nell'osservazione della rappresentazione grafica, si può utilizzare il mouse, trascinando verso una direzione, per spostarsi all'interno del piano virtuale creato dalla finestra applicativa. L'altro utilizzo del pulsante reset è quello di azzerato le coordinate di spostamento, ritornando alla posizione

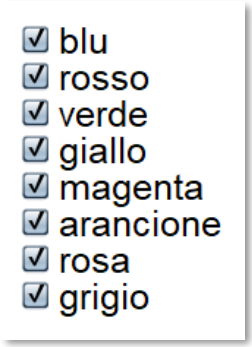

Figura 14: rettangolo visualizzato all'interno della finestra, esempio fatto con otto insiemi.

iniziale. L'interfaccia, per disegnare i componenti geometrici, riceve in input dal package *algorithm*, in particolare dalla classe *Drawing*, due *LinkedList* che contengono rispettivamente i nodi e gli archi che si devono disegnare.

3.2.1 Nodi

Ricevuta la lista di nodi da disegnare, il metodo *paintComponent* richiama il metodo privato *drawPoints*. Questo metodo riceve come parametro lo stesso oggetto *Graphics* ricevuto come parametro da *paintComponent*. All'interno del metodo privato si inizializza un ciclo for che scandisce tutti i nodi contenuti all'interno della lista passata. Dentro il ciclo *for* si inizializza un altro ciclo *for* annidato per scandire l'array di colori di ogni nodo. Con l'aiuto di una *LinkedList* contenente tutti oggetti *Color*, si assegnano i rispettivi colori al nodo, e per farli vedere tutti, si divide il nodo in spicchi, ognuno contenente il proprio colore (Figura 15-16-17). Tutti i nodi verranno rappresentati da oggetti *Arc2D* che permettono questa divisione in spicchi colorati con i colori dei nodi corrispondenti, posizionati esattamente nella posizione destinata al nodo.

3.2.2 Archi

Ricevuta la lista di archi da disegnare, il metodo *paintComponent* richiama il metodo privato *drawLines*. Questo metodo riceve come parametro lo stesso oggetto *Graphics* ricevuto come parametro da *paintComponent*. All'interno del metodo privato viene inizializzato un ciclo *for*, che scandirà tutti gli archi contenuti nella lista in input. Una lista contenente tutti oggetti *Color* servirà da appoggio nell'assegnare ogni colore al proprio arco. Questa lista di appoggio è diversa da quella usata precedentemente con i nodi; infatti, questa lista fa cambiare i propri colori grazie ad un altro metodo annidato dentro *paintComponent*, il metodo *clickingInCheckBox*. Se la *CheckBox* i-esima rappresentata dal colore i-esimo è vera, al colore i-esimo non succederà nulla, dove i è un intero da 1 al numero dei colori presi in considerazione. In caso fosse falsa, allora si procede con un'operazione bit a bit per rendere il colore trasparente e permettere a tutti gli archi di sparire dalla vista. Gli archi verranno rappresentati da oggetti *Line2D* che prendono come parametro le quattro coordinate dell'arco: ascissa e ordinata del nodo di partenza e ascissa e ordinata del nodo di destinazione. Oltre

a ciò, per le linee sovrapposte, verranno tratteggiate proprio per essere viste entrambe dall'utente. Lo spessore delle linee viene aumentato in base allo zoom per farle vedere meglio (Figura 15-16-17).

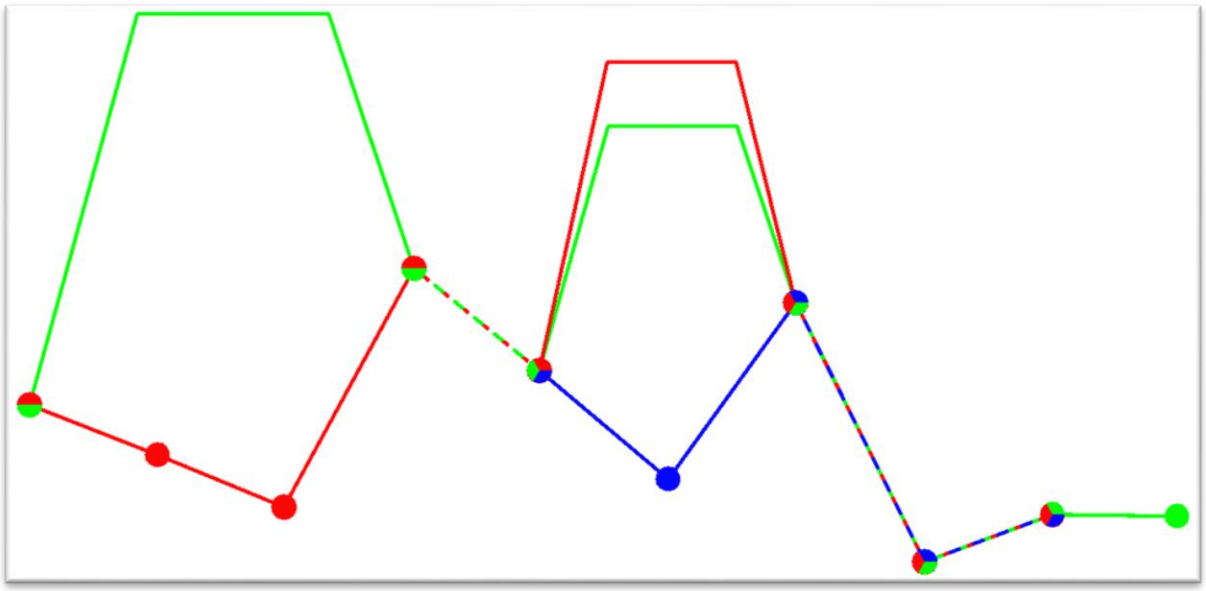

Figura 15: esempio dell'applicazione scritta utilizzando 10 nodi e 3 colori.

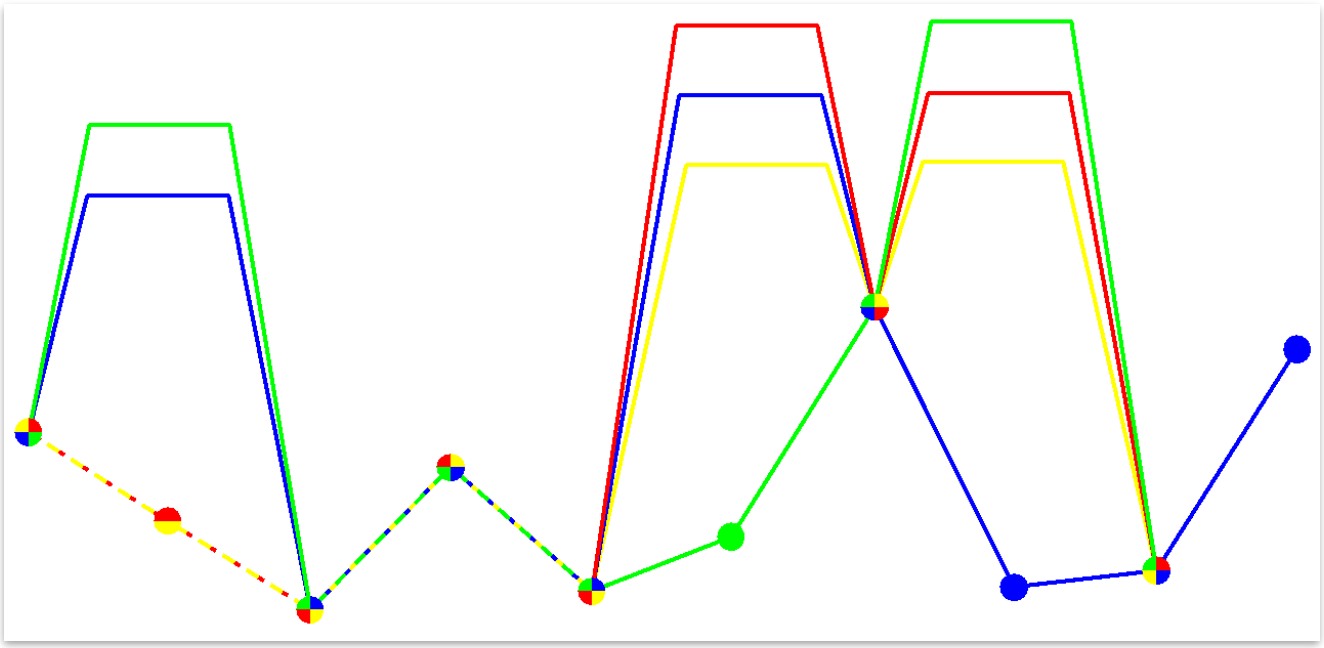

Figura 16: esempio dell'applicazione scritta utilizzando 10 nodi e 4 colori.

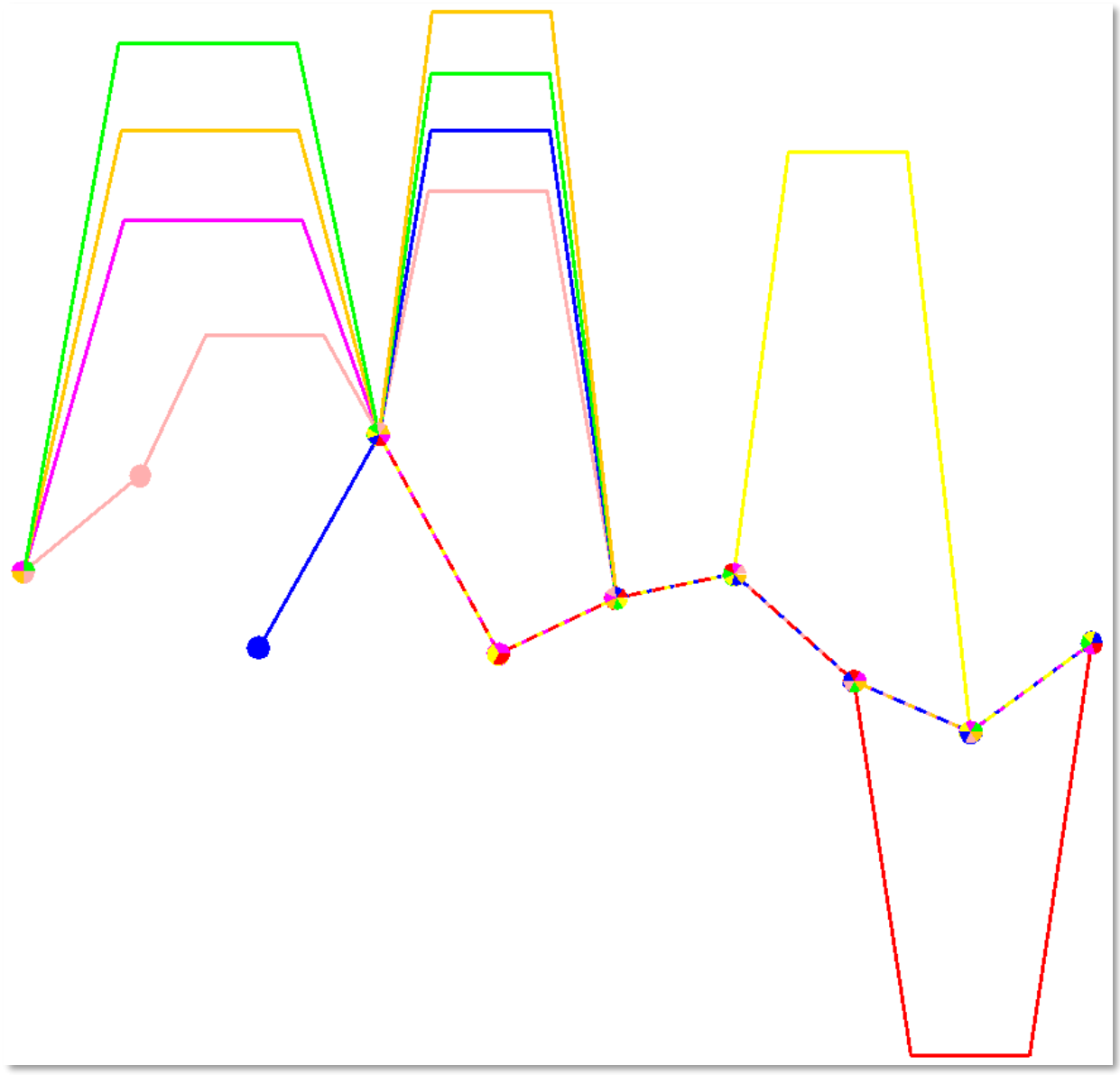

Figura 17: esempio dell'applicazione scritta utilizzando 10 nodi e 7 colori.

Capitolo 4. Esperimenti

4.1 Obbiettivo esperimento

È stato condotta un'analisi sperimentale sull'applicazione descritta al fine di visionare i risultati che si sono ottenuti e far capire le prestazioni della pipeline algoritmica creata. Questo confronto fra algoritmi è utile per raccogliere i punti di forza degli algoritmi scelti e confrontare le prestazioni. Oltre a ciò, è stato condotto anche un esempio d'uso per far capire al meglio il funzionamento dell'algoritmo anche fuori dai contesti puramente teorici, analizzando come si comporta la pipeline algoritmica creata sul campo. Nell'esempio, si sono utilizzati dei dati reali o raccolti sul campo per testare gli algoritmi in condizioni realistiche. In questo modo, si è potuto osservare come si comporta la pipeline algoritmica in diversi scenari. Come scenario e dominio si è utilizzato il contesto delle mappe geografiche per il constrained set visualization di diversi punti vincoli dalla posizione. È un esempio che riguarda gli hotel, i ristoranti e i punti più visitati nella zona di Perugia centro. Nelle prossime sezioni verrà prima definito come è stata condotta l'analisi sperimentale, poi verranno discussi i dati sperimentali ottenuti, il tutto osservando i grafici che si sono creati utilizzando i dati sperimentali. Nell'appendice verrà visualizzata la tabella completa contenente i dati sperimentali utilizzati (Appendice).

4.2 Setting sperimentale

Ho usato le seguenti metriche per misurare le prestazioni:

- Tempo impiegato nella compilazione dell'applicazione (espresso in millisecondi). Questo tempo è stato ottenuto con la semplice sottrazione $t_T = t_I - t_F$, dove t_I è il tempo preso a valle dell'intera pipeline algoritmica, t_F è il tempo preso a monte della pipeline algoritmica e t_T è il tempo che è stato registrato nella tabella, che coincide con l'esatto tempo di esecuzione della pipeline algoritmica;
- Area in più occupata dal mio algoritmo per il tracciamento di archi e l'individuazione degli angoli. Ci si è serviti semplicemente della formula $\frac{a}{a_n}$, dove a è l'area occupata dall'intero disegno e a_n è l'area occupata solo dai nodi;
- Numero di *Bend* creati;
- Numero di *Bend* creati rapportato al numero di archi creato. Questi dati sono stati calcolati nel modo seguente: $\frac{n_a}{a}$, dove n_a è il numero di *Bend* creati e a è il numero di archi generati. Nella formula a comprende sia gli archi generati nella pagina sotto, sia gli archi generati nella pagina sopra e sia gli archi che attraversano la spina dorsale del linear layout;
- Numero di incroci capitati durante l'esecuzione dell'applicazione.

Queste metriche sono state misurate per i seguenti singoli esperimenti cambiando il numero dei nodi: da 10 a 100 nodi con passo di 10, da 100 a 1000 nodi con passo di 100, da 1000 a 10000 nodi con passo di 1000. Per ogni singolo esperimento, inoltre, sono state misurate le metriche cambiando il numero di insiemi (colori) presi

in considerazione, che vanno dai 2 ai 10. La prova sull'applicazione è stata valutata utilizzando le considerazioni elencate seguentemente: i nodi che sono stati creati hanno posizione casuale, il range in cui possono essere creati vanno da 0 pixel ai $1000 \cdot n$, dove n è il numero di nodi. Per quanto riguarda il numero di insiemi per ogni nodo, è stato scelto in modo puramente casuale utilizzando la classe *Random* offerta da Java. Anche gli insiemi specifici per ogni nodo sono puramente casuali. Ci si è serviti della classe *Experiment* come *main* per simulare tutti i casi descritti sopra. Oltre ad essa, la classe *ReadTextFile* si è rilevata molto utile nella sperimentazione. È stata creata allo scopo di salvare e ogni dato sperimentale ottenuto in un file CSV (Experiments.csv) per la raccolta di essi. Infine, i dati sperimentali sono stati trattati per ottenere la tabella (Appendice) e, di conseguenza, anche i grafici riportati nella sezione successiva.

4.3 Risultati Esperimento

4.3.1 Analisi Sperimentale

Nel complesso l'analisi ha raccolto informazioni molto utili e sottolinea molti vantaggi della pipeline algoritmica utilizzata. Prima di tutto la sua planarità: la pipeline, con un numero di colori abbastanza consistente, riesce comunque ad avere pochi incroci (Figura 27). Un altro vantaggio molto grande è il fatto che crea molti pochi Bend rapportato al numero di archi (Figura 25-26), ciò comporta minor difficoltà di lettura. Durante la sperimentazione, purtroppo, ci si è accorti che le prestazioni dell'algoritmo degradavano molto velocemente. Si è riusciti a effettuare le sperimentazioni fino a 4000 nodi. I grafici riguardanti i nodi da 1000 a 4000 con passo di 1000 non sono stati presi in considerazione, tranne per l'andamento dei tempi:

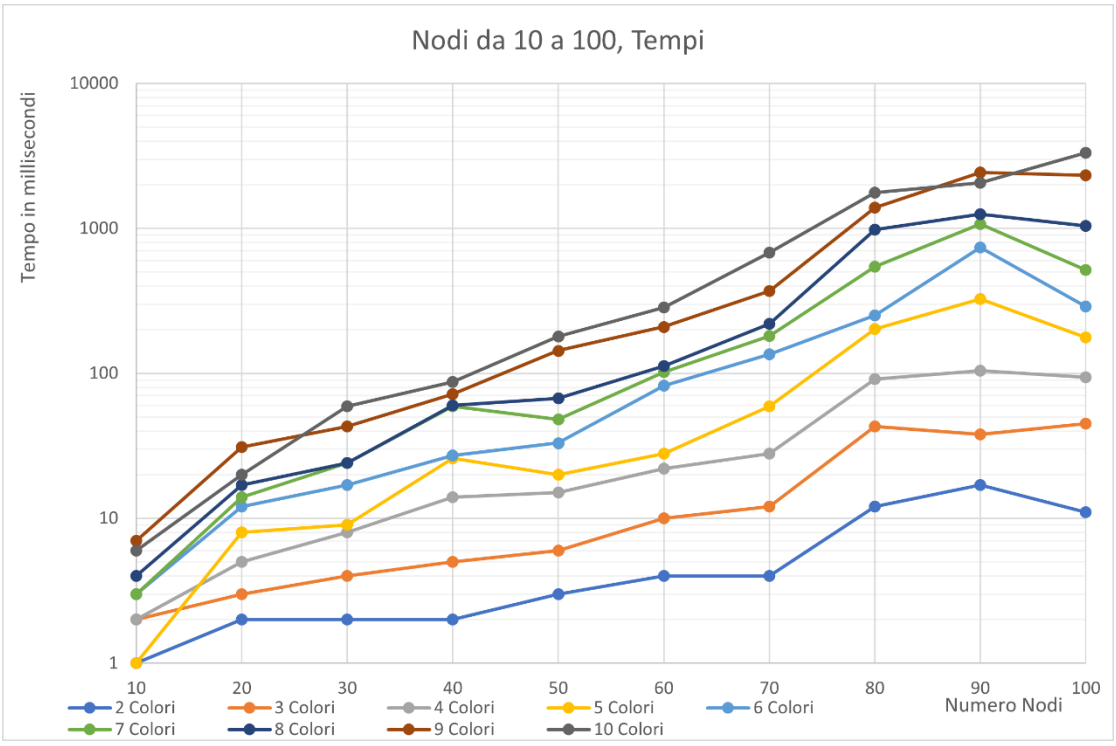

Figura 18: grafico che mostra l'andamento dei tempi da 10 a 100 nodi con passo di 10.

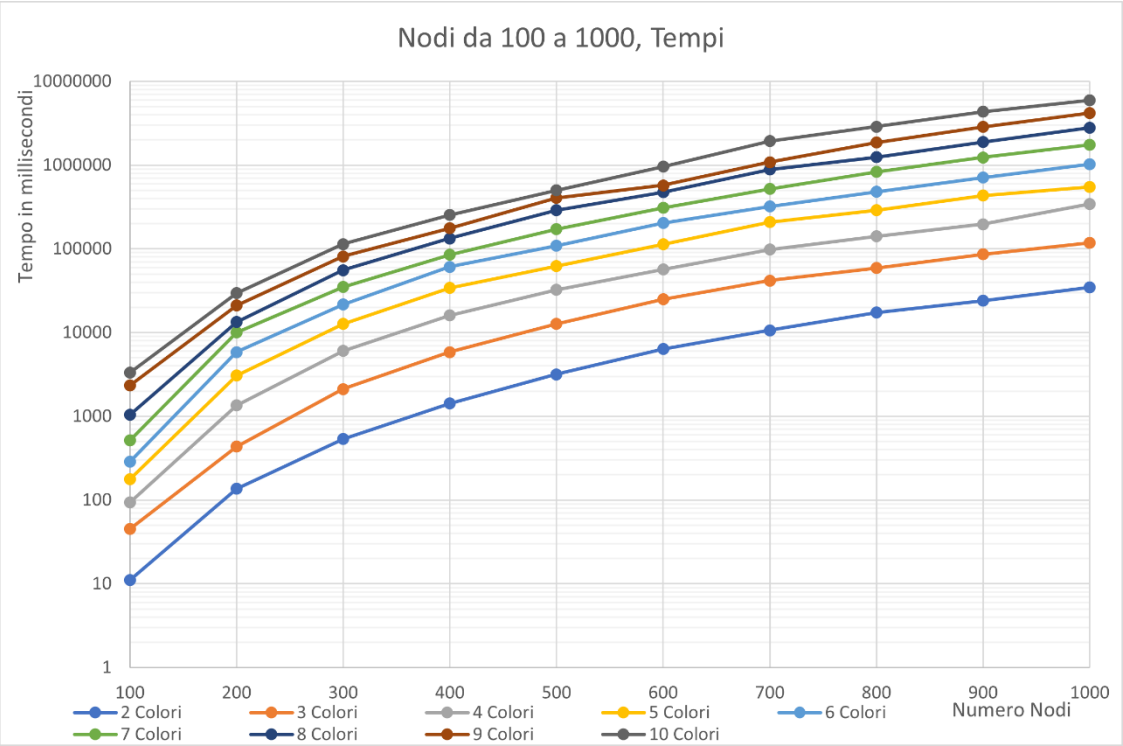

Figura 19: grafico che mostra l'andamento dei tempi da 100 a 1000 nodi con passo di 100.

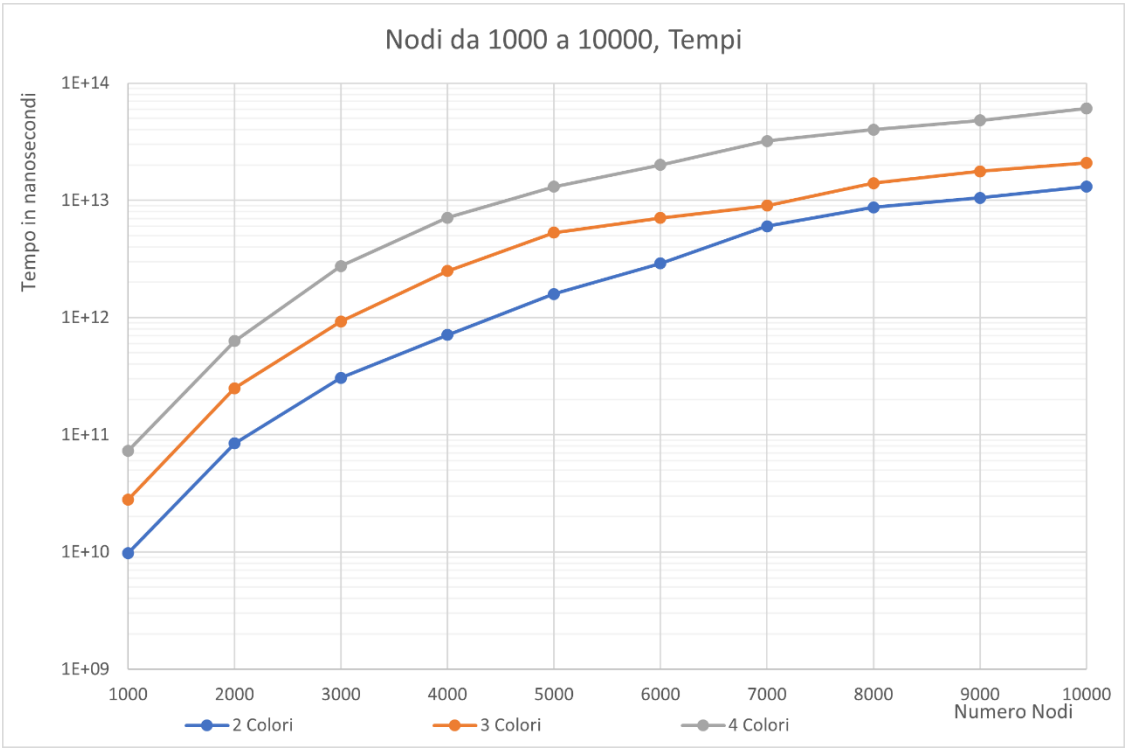

Figura 20: grafico che mostra l'andamento dei tempi da 1000 a 4000 nodi con passo di 1000.

Questi grafici visti in precedenza sono i grafici che mostrano le prestazioni della pipeline algoritmica in diversi scenari: il primo grafico (Figura 18) mostra le prestazioni dell'algoritmo da 10 a 100 nodi con passo di 10, il secondo grafico (Figura 19) mostra le prestazioni dell'algoritmo da 100 a 1000 nodi con passo di 100, il terzo

grafico (Figura 20) mostra le prestazioni dell'algoritmo da 1000 a 4000 nodi con passo di 1000. Come si può notare dalle tre figure, al crescere dei nodi le prestazioni degradano in maniera esponenziale, essendo l'ordinata di ogni grafico non lineare, ma logaritmica. Inoltre, anche al crescere del numero di colori le prestazioni degradano: più colori ci sono, peggiori sono i tempi impiegati dall'algoritmo. Si mostra addirittura che il numero di colori influenza più significativamente rispetto al numero dei nodi. L'aggiunta di 10, 100 o 1000 nodi in più non è paragonabile all'aggiunta di un solo colore in più. Nel grafico da 10 a 100, le linee rappresentate dai colori sono più incerte nella loro andatura, ma questo è solo perché i tempi sono relativamente bassi, usando delle posizioni dei nodi e il numero di colori tutti casuali, può succedere che ci sia una formazione di colori favorevole ai tempi, addirittura da paragonarsi al passo precedente. Infatti, quando i tempi aumentano, il fenomeno descritto non si verifica più anche con formazioni ottime, perchè l'algoritmo ne risente poco rispetto allo scorrere dei passi. Inoltre, la crescita dei tempi aumentando il numero di colori cresce in modo uniforme. Ad esempio, se prendiamo la linea dei 2 colori del secondo grafico (Figura 19), vediamo che all'aumentare dei nodi, il valore cresce di circa tre ordini di grandezza; se lo confrontiamo con la linea dei 2 colori del terzo grafico (Figura 20), vediamo che anch'essa cresca di circa tre ordini. Ciò non vale per il primo grafico (Figura 18), dove i tempi sono relativamente bassi.

Si può procedere con la visualizzazione dell'area occupata:

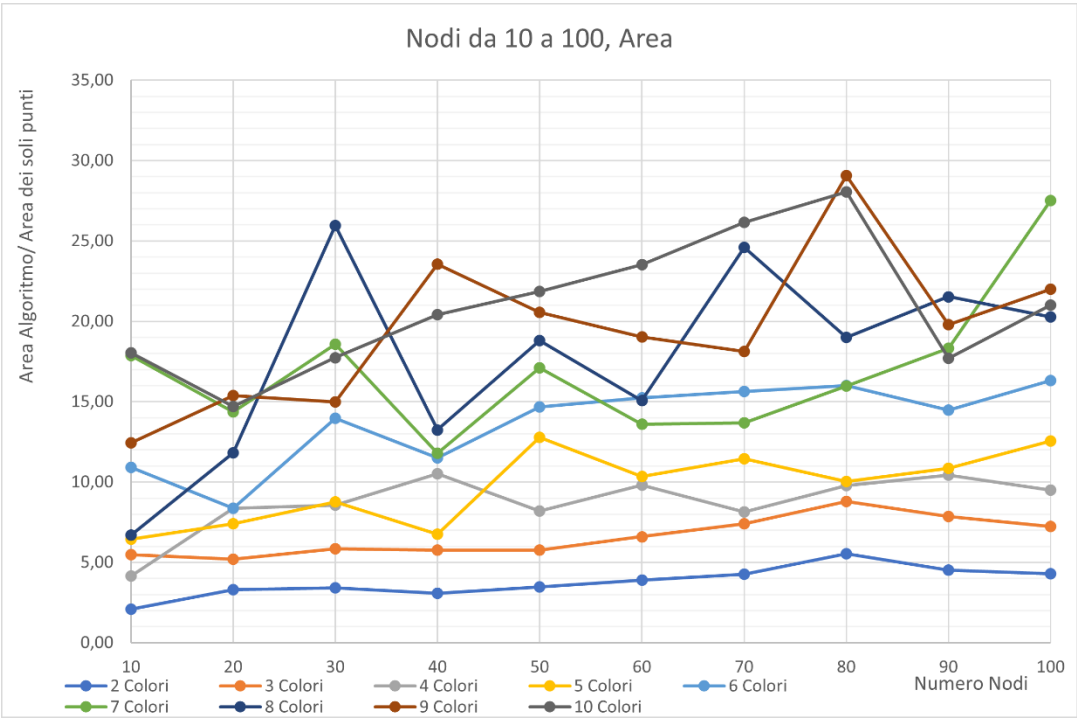

Figura 21: grafico che mostra l'andamento del rapporto tra area occupata del disegno e area già occupata da 10 a 100 nodi con passo di 10.

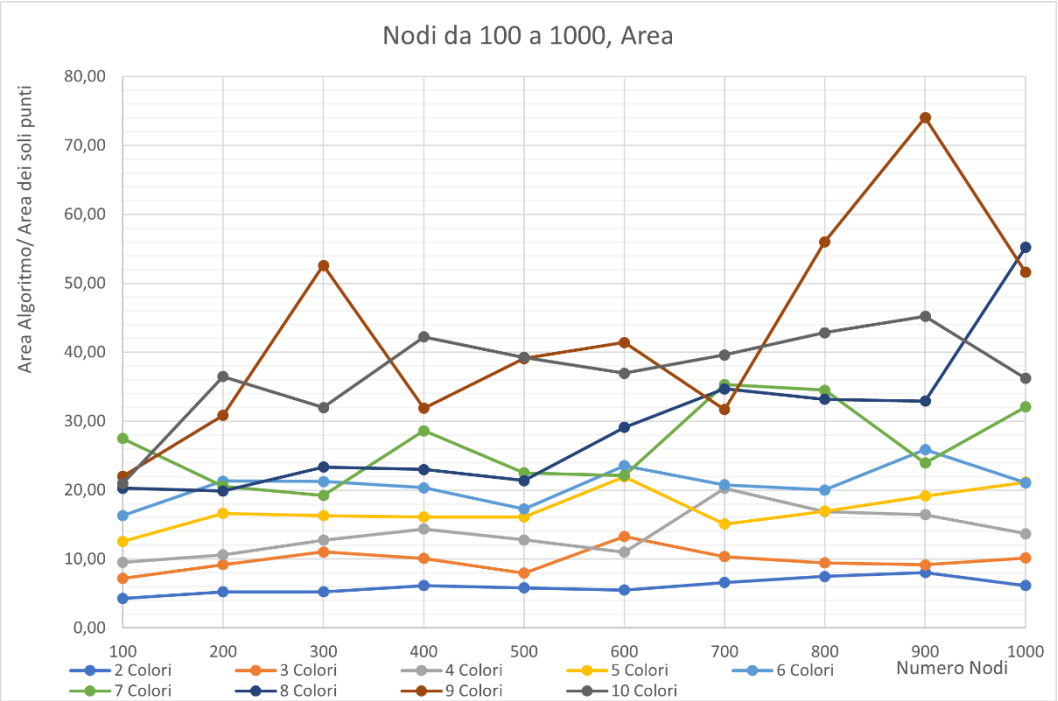

Figura 22: grafico che mostra l'andamento del rapporto tra area occupata del disegno e area già occupata da 100 a 1000 nodi con passo di 100.

I due grafici (Figura 21-22) sopra elencati mostrano l'andamento del rapporto tra area occupata dal disegno e l'area già occupata dai punti. Nell'ascissa troviamo il numero di nodi, nell'ordinata troviamo il rapporto descritto e come parametro abbiamo il numero di colori. Come si può notare all'aumentare del numero dei colori, l'area occupata aumenta. Il primo grafico (Figura 21) è più incerto del grafico situato sotto di lui (Figura 22) perché la poca distanza tra i passi viene influenzata significativamente dalla posizione e dall'ordine favorevole o sfavorevole dei punti e dei colori per ogni punto. Comunque, però, si può notare che questo rapporto, intuitivamente, non scende mai sotto l'1; ma può arrivare nel caso peggiore anche a circa 75 volte di più.

Si passa alla prossima metrica, cioè il numero di *Bend* creati:

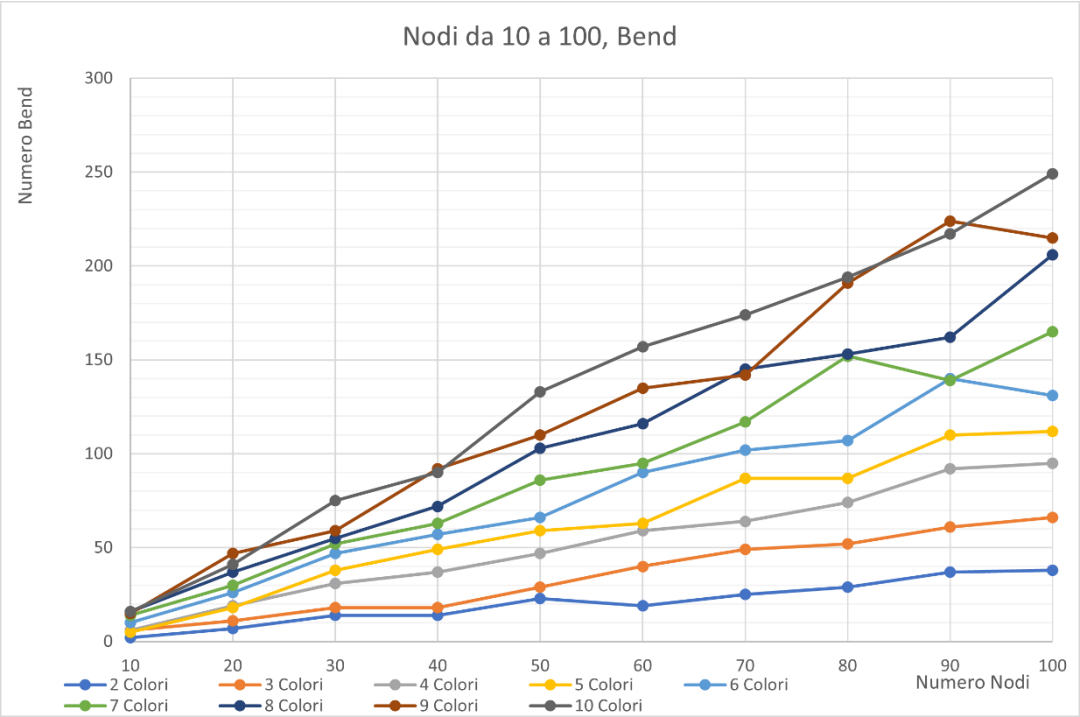

Figura 23: grafico che mostra l'andamento del numero di *Bend* da 10 a 100 nodi con passo di 10.

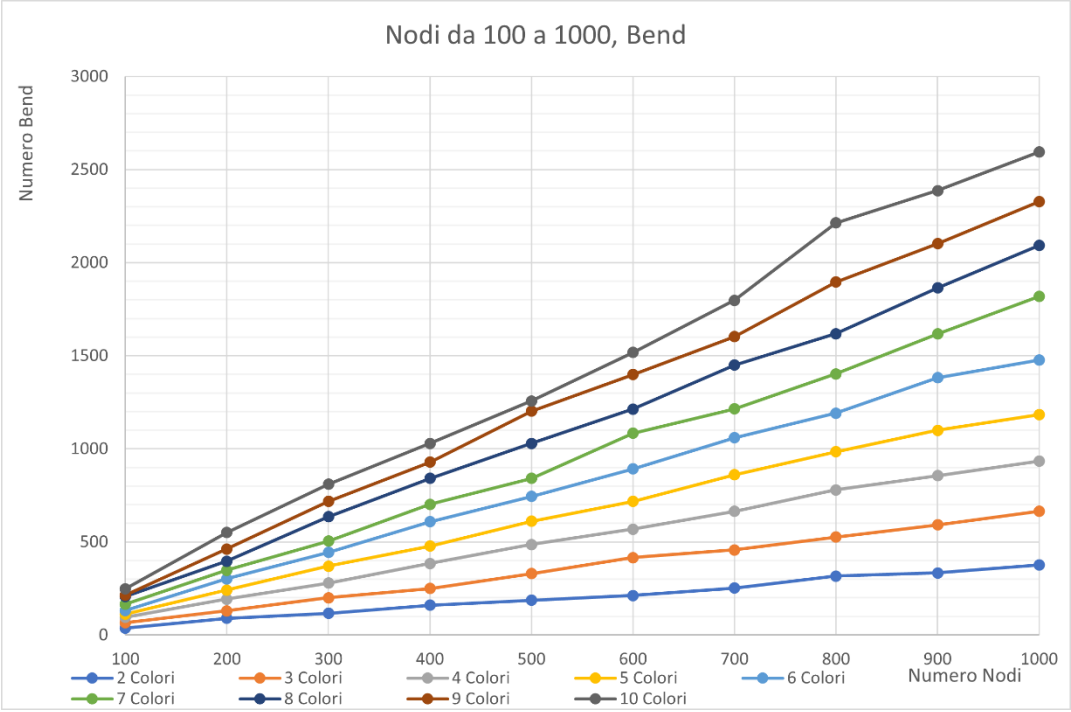

Figura 24: grafico che mostra l'andamento del numero di *Bend* da 100 a 1000 nodi con passo di 100.

Per quanto riguarda questa metrica, i risultati che si sono ottenuti sono conformi alle aspettative. L'andamento che segue è lineare. Aumentando il numero di colori, il numero di *Bend* aumenta significativamente rispetto all'aumento del numero. Fatto importante da notare è che, la crescita del numero degli *Bend* ad ogni passo viene incrementata sempre di più, proporzionalmente rispetto al numero di colori utilizzati. Come succedeva

per i tempi, nel primo grafico (Figura 23), il numero di *Bend* non è costante, varia rispetto al caso. Questo non succede per il secondo grafico (Figura 24), dove il numero di *Bend* tra un passo e l'altro è più significativo.

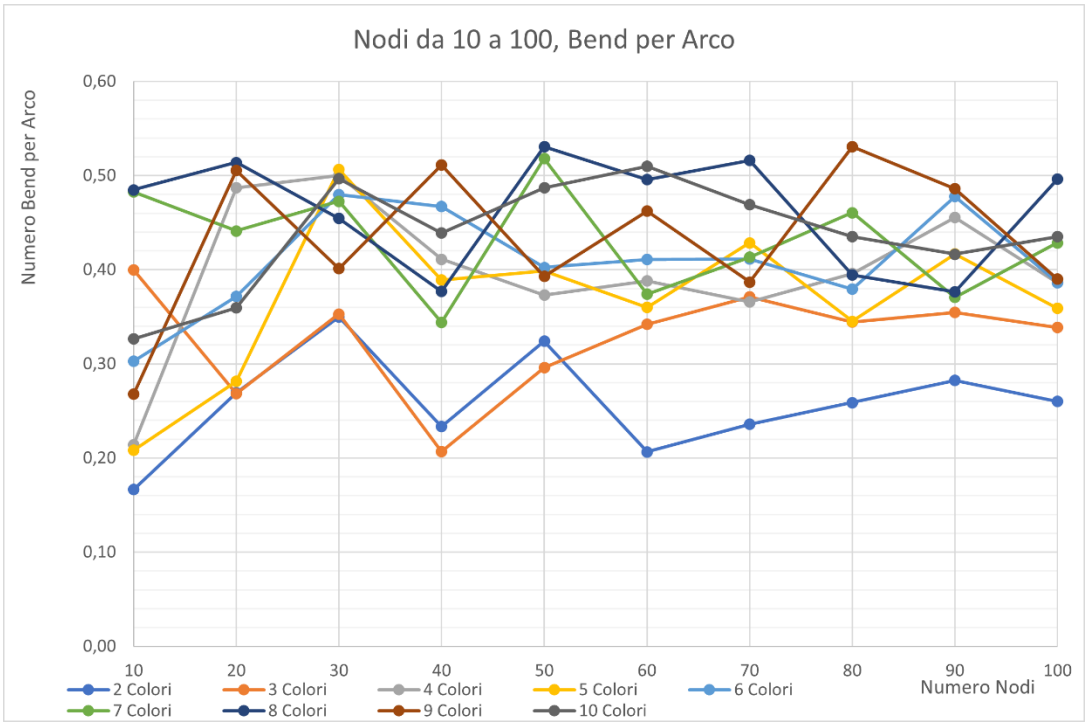

Figura 25: grafico che mostra l'andamento del rapporto tra il numero di *Bend* ed il numero di archi da 10 a 100 nodi con passo di 10.

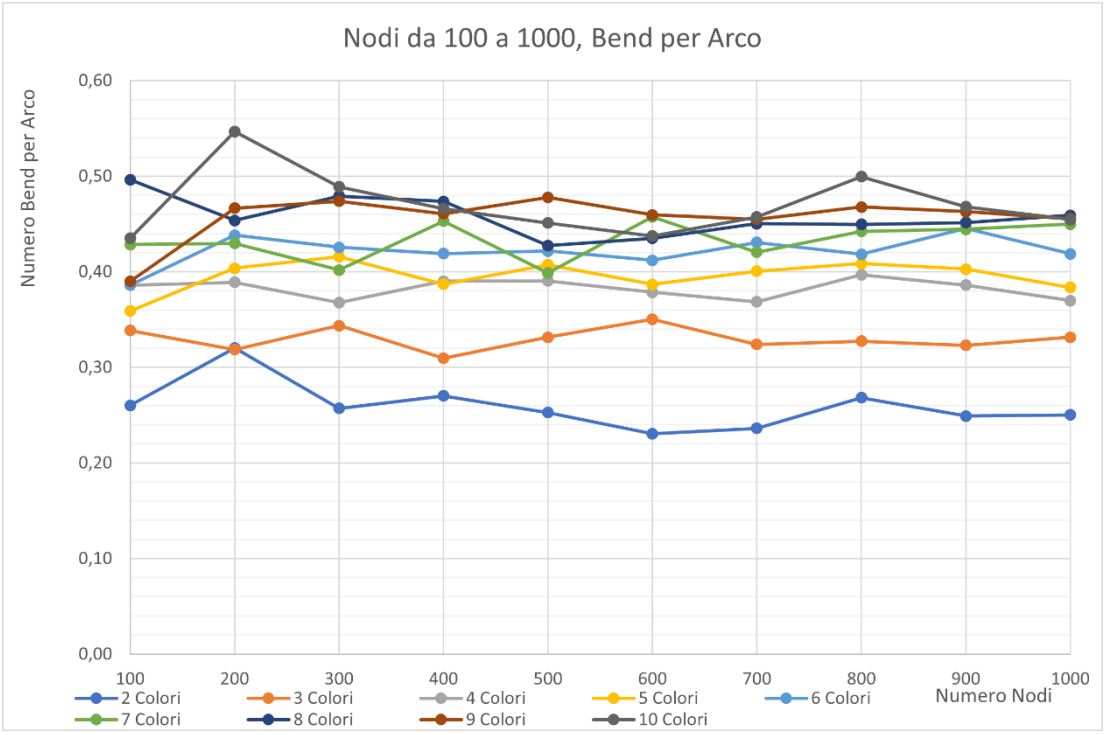

Figura 25: grafico che mostra l'andamento del rapporto tra il numero di *Bend* ed il numero di archi da 100 a 1000 nodi con passo di 100.

I due grafici sopra elencati (Figura 25-26) mostrano l'andamento del rapporto tra il numero di *Bend* ed il numero di archi creati. Nel primo grafico (Figura 25) non si riconosce bene l'andamento che ha, sempre per il problema della casualità delle posizioni dei punti e della casualità delle posizioni dei colori. Nel secondo grafico (Figura 26), invece, si nota che l'andamento dei due grafici è un andamento costante. Infatti, l'aumento del numero dei nodi non influenza in alcun modo il rapporto descritto. Se i nodi aumentano, sia il numero degli archi, che il numero degli *Bend* aumenta, lasciando il rapporto quasi invariato. Il numero dei colori, al contrario, cambia significativamente l'andamento del grafico, aumentando il rapporto. Si noti che l'aumento da parte dei colori del rapporto è logaritmico, infatti per meno colori, ad esempio 2 e 3, il rapporto cresce circa di 1; per tanti colori, ad esempio 7 e 8, il rapporto cresce di poco. Il range in cui varia esso è compreso tra 1 e 7 circa.

Verranno riportati sotto i due grafici riguardanti l'ultima metrica, il numero di incroci:

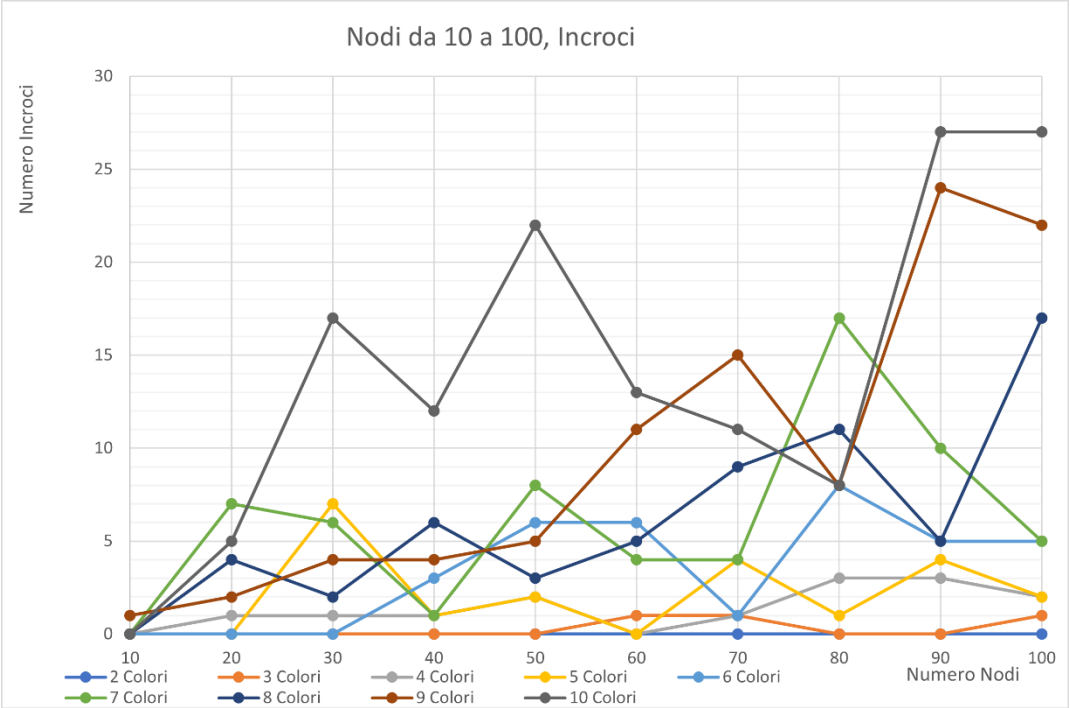

Figura 26: grafico che mostra l'andamento del numero di incroci creati tra gli archi di diverso colore da 10 a 100 nodi con passo di 10.

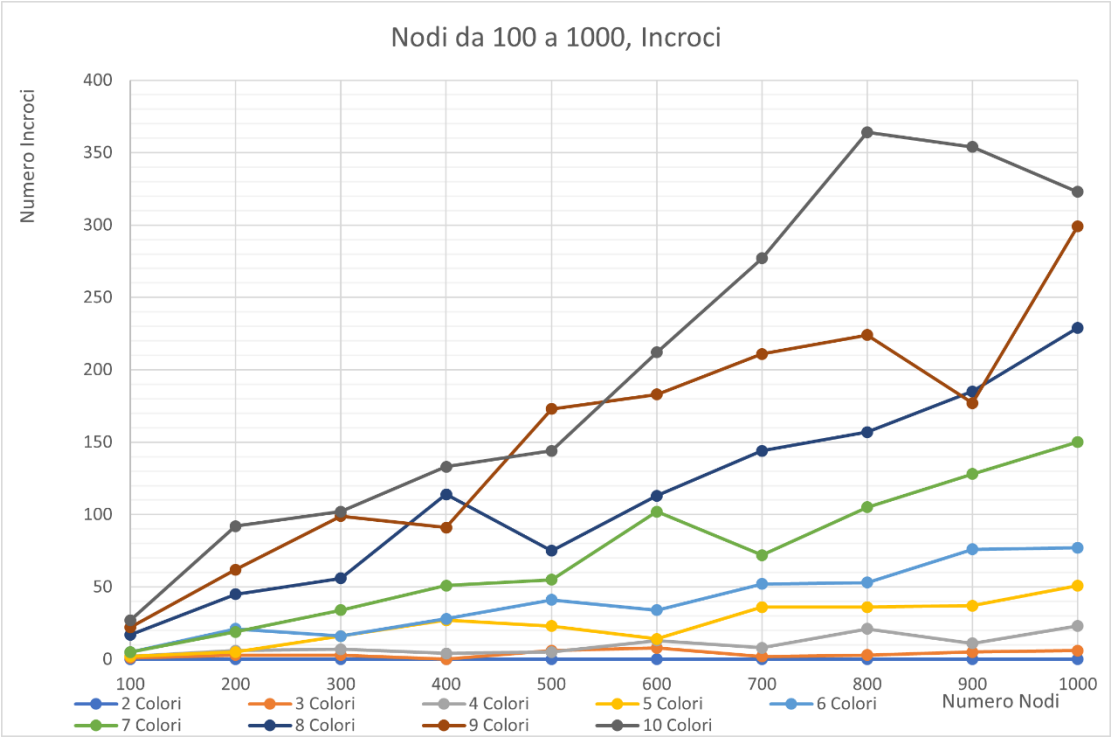

Figura 27: grafico che mostra l'andamento del numero di incroci creati tra gli archi di diverso colore da 100 a 1000 nodi con passo di 100.

L'andamento dei due grafici (Figura 27-28) è circa lineare. Quindi, all'aumentare del numero dei nodi, il numero di incroci aumenta. Questi grafici rispettano le nostre aspettative. Si noti che, anche all'aumentare del numero di colori, gli incroci aumentano. Naturalmente, aumentando il numero di colori, le chance di incrocio di due archi di colori differenti aumenta in modo significativo. Il range creato dai due grafici va da 0 incroci, nel caso più semplice di 2 e 3 colori per qualsiasi numero di nodi, a circa 370 incroci nel caso di 800 nodi con 10 colori. L'andamento del secondo grafico (Figura 28) risulta più chiara in quanto la distanza dei passi è più accentuata rispetto a quella del primo grafico (Figura 27). Alcune linee risentono molto della casualità degli esperimenti fatti, come ad esempio il punto di intersezione tra la linea di 10 colori e 800 nodi. Esso ha più incroci del caso di 1000 nodi con stesso numero di colori.

4.3.2 Esempio d'Uso

Per quanto riguarda l'esperimento pratico si è scelto lo studio delle relazioni tra luoghi geografici. Nella foto seguente (Figura 29) si può notare una piantina del centro di Perugia vista dall'alto. Sono stati segnati diversi punti con tre rispettivi colori:

- I punti rossi rappresentano alcuni ristoranti con recensioni più alte della zona centro di Perugia;
- I punti blu rappresentano alcuni l'hotel e il B&B con recensioni più alte della zona centro di Perugia;
- I punti verdi rappresentano alcuni dei luoghi con recensioni più alte della zona centro di Perugia.

La pipeline algoritmica è stata applicata all'insieme di punti visualizzato (Figura 30). Il punto con doppio colore (Figura 29-30) rappresenta una località che esegue sia attività di hotel, che attività di ristorazione.

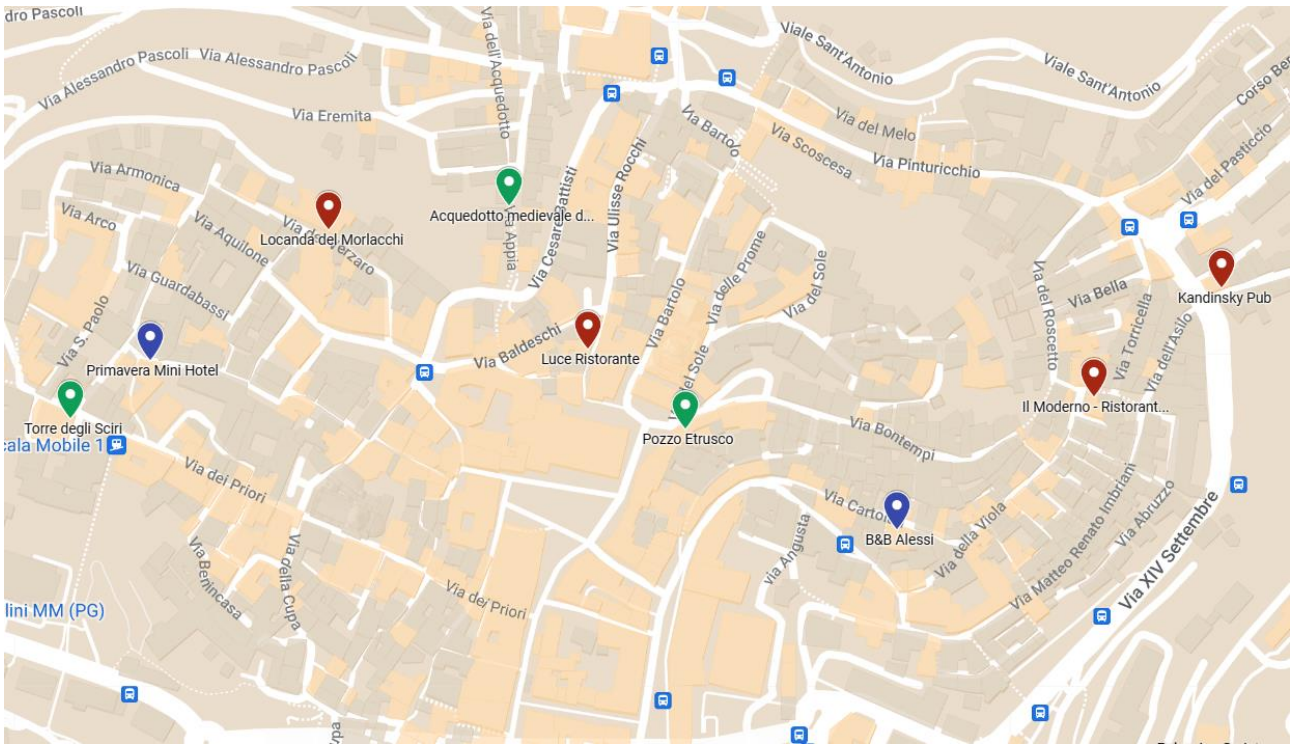

Figura 29: piantina della città di Perugia centro vista dall'alto. I punti rossi rappresentano alcuni dei ristoranti con recensioni più alte. I punti verdi rappresentano alcuni dei luoghi più visitati di Perugia. I punti blu rappresentano alcuni degli hotel e B&B con recensioni più alte di Perugia [10].

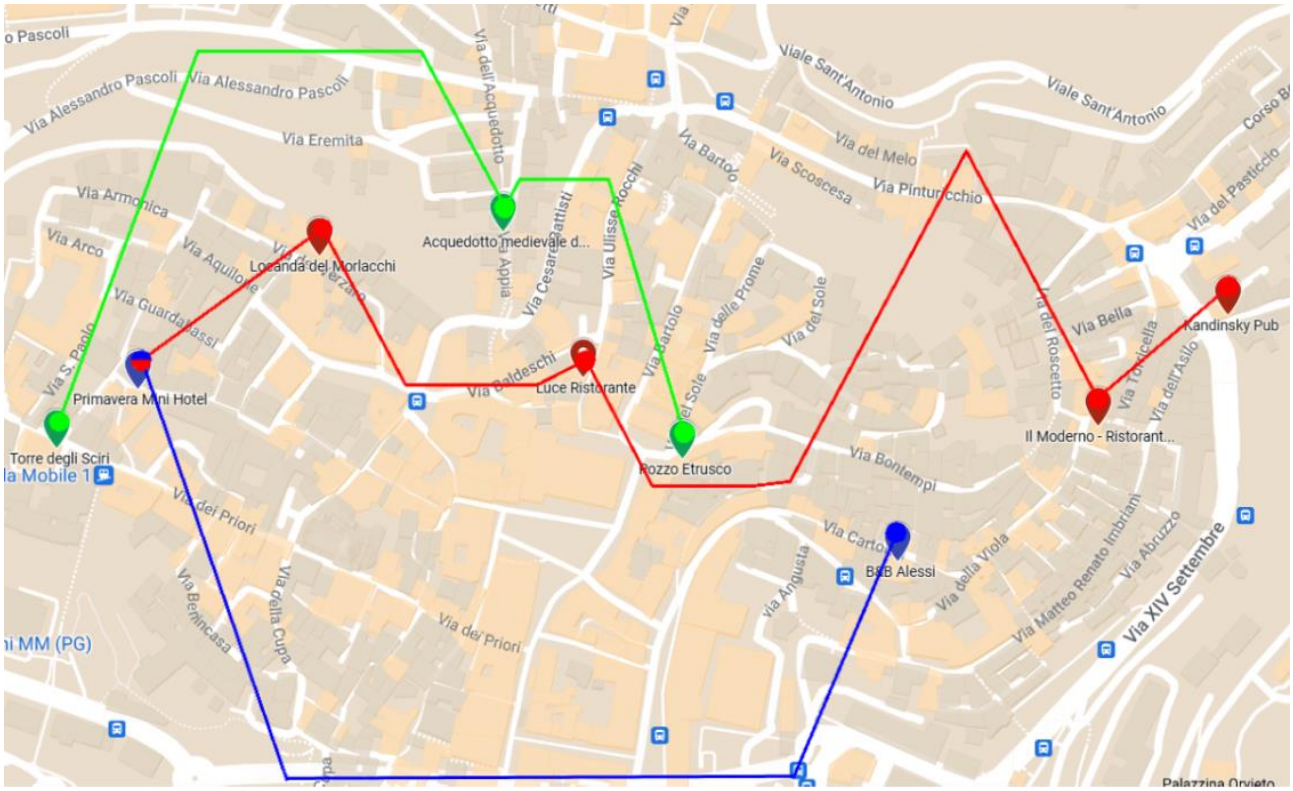

Figura 30: Questa foto mostra la pipeline algoritmica descritta applicata a questo contesto geografico [10].

Capitolo 5. Conclusioni e Sviluppi Futuri

5.1 Conclusioni

L'applicazione che è stata descritta mostra una tecnica utilizzata per il problema della Constrained Set Visualization. L'obbiettivo richiesto dall'applicazione era la visualizzazione di un linear layout senza l'aggiunta di eventuali incroci che non siano già stati definiti dal linear layout. I risultati raggiunti mostrati nella sezione 4.3 fanno intendere che la pipeline algoritmica utilizzata ha una complessità molto alta, ma riesce nell'intento richiesto. Queste conclusioni sono state raggiunte osservando le prestazioni della pipeline algoritmica, i grafici delle prestazioni elencati nella sezione 4.3 (Figura 18-19-20) e i grafici che mostrano gli incroci che sono stati creati (Figura 27-28). I limiti negli esperimenti del programma che sono stati raggiunti si fermano a 4000 nodi con 10 colori. Le difficoltà di realizzazione dell'algoritmo spaziano dalla creazione del linear layout, in particolare l'aggiunta degli archi e la successiva ottimizzazione, fino alla costruzione geometrica, in particolare l'aggiunta dei nodi fittizi e il taglio degli archi.

5.2 Sviluppi futuri

Si possono apportare molte migliorie all'applicazione in generale. Sicuramente la più importante riguarda l'efficienza della pipeline algoritmica; alcune sezioni di codice possono essere implementate in maniera più efficiente. Oltre alle migliorie riguardanti il codice, si possono apportare le seguenti migliorie a livello di progettazione:

- Creazione di un algoritmo a monte rispetto la pipeline algoritmica creata che decide l'ordine dei punti in modo tale da minimizzare gli incroci del linear layout;
- Creazione di un algoritmo per il taglio degli archi più utile di quello realizzato, utilizzando una struttura ad albero per l'individuazione degli archi più piccoli e degli archi più grandi e il successivo taglio di essi in modo tale da non aumentare di molto l'area occupata;
- Per quanto riguarda la visualizzazione dell'interfaccia, il disegno degli archi tramite curve di Bezier, in modo tale da diminuire l'area occupata dal grafico creato.

Bibliografia

[1] Problema SetVisualization. Disponibile su KelpFusion_A_Hybrid_Set_Visualization_Techniqhe.

[2] Diagrammi Di Venn. Disponibile su Diagramma di Venn - Wikipedia.

[3] Problema SetVisualization, tecniche complesse. Disponibile su Set Visualisation - Algorithms - Utrecht University (uu.nl).

[4] Book embedding. Disponibile su Book embedding - Wikipedia.

[5] SetVisualizazion. Disponibile su Set Visualization — giCentre.

[6] PivotPaths. Disponibile su Marian Dörk – PivotPaths (mariandoerk.de).

[7] Selection Sort. Disponibile su Emilio Di Giacomo, 07-Ordinamento, Apr 2023.

[8] Liste di adiacenza. Disponibile su Emilio Di Giacomo, 16-Grafi, Apr 2023.

[9] Tecnica base della mappatura di un grafo vincolato dalle posizioni dei nodi, Disponibile su KaufmannWiese2002.6.1.

[10] Applicazione web utilizzata per l'esperimento pratico. Disponibile su Google Earth.

Appendice

Nodi	Colori	Tempo di Esecuzione (in ms)	Area Occupata	Numero Bend	Numero Bend per Arco	Incroci
10	2	1	2,08	2	0,17	0
10	3	2	5,48	6	0,40	0
10	4	2	4,16	6	0,21	0
10	5	1	6,44	5	0,21	0
10	6	3	10,90	10	0,30	0
10	7	3	17,86	14	0,48	0
10	8	4	6,69	16	0,48	0
10	9	7	12,43	15	0,27	1
10	10	6	18,04	16	0,33	0
20	2	2	3,31	7	0,27	0
20	3	3	5,19	11	0,27	0
20	4	5	8,36	19	0,49	1
20	5	8	7,40	18	0,28	0
20	6	12	8,36	26	0,37	0
20	7	14	14,36	30	0,44	7
20	8	17	11,81	37	0,51	4
20	9	31	15,38	47	0,51	2
20	10	20	14,70	41	0,36	5
30	2	2	3,41	14	0,35	0
30	3	4	5,85	18	0,35	0
30	4	8	8,57	31	0,50	1
30	5	9	8,77	38	0,51	7
30	6	17	13,95	47	0,48	0
30	7	24	18,57	52	0,47	6
30	8	24	25,95	55	0,45	2
30	9	43	14,97	59	0,40	4
30	10	59	17,74	75	0,50	17
40	2	2	3,09	14	0,23	0
40	3	5	5,77	18	0,21	0
40	4	14	10,52	37	0,41	1
40	5	26	6,75	49	0,39	1
40	6	27	11,50	57	0,47	3
40	7	59	11,79	63	0,34	1
40	8	60	13,24	72	0,38	6
40	9	72	23,55	92	0,51	4
40	10	87	20,40	90	0,44	12
50	2	3	3,48	23	0,32	0
50	3	6	5,77	29	0,30	0
50	4	15	8,18	47	0,37	2
50	5	20	12,78	59	0,40	2
50	6	33	14,66	66	0,40	6
50	7	48	17,10	86	0,52	8
50	8	67	18,80	103	0,53	3
50	9	143	20,55	110	0,39	5
50	10	179	21,86	133	0,49	22
60	2	4	3,90	19	0,21	0
60	3	10	6,60	40	0,34	1
60	4	22	9,80	59	0,39	0
60	5	28	10,35	63	0,36	0

60	6	82	15,23	90	0,41	6
60	7	102	13,59	95	0,37	4
60	8	112	15,06	116	0,50	5
60	9	209	19,02	135	0,46	11
60	10	285	23,53	157	0,51	13
70	2	4	4,25	25	0,24	0
70	3	12	7,39	49	0,37	1
70	4	28	8,15	64	0,37	1
70	5	59	11,46	87	0,43	4
70	6	135	15,63	102	0,41	1
70	7	181	13,70	117	0,41	4
70	8	219	24,59	145	0,52	9
70	9	369	18,12	142	0,39	15
70	10	679	26,15	174	0,47	11
80	2	12	5,53	29	0,26	0
80	3	43	8,79	52	0,34	0
80	4	91	9,77	74	0,40	3
80	5	202	10,03	87	0,35	1
80	6	251	16,01	107	0,38	8
80	7	545	15,98	152	0,46	17
80	8	976	19,00	153	0,39	11
80	9	1394	29,06	191	0,53	8
80	10	1762	28,04	194	0,43	8
90	2	17	4,52	37	0,28	0
90	3	38	7,87	61	0,35	0
90	4	104	10,44	92	0,46	3
90	5	326	10,85	110	0,42	4
90	6	737	14,46	140	0,48	5
90	7	1068	18,31	139	0,37	10
90	8	1253	21,53	162	0,38	5
90	9	2434	19,79	224	0,49	24
90	10	2056	17,69	217	0,42	27
100	2	11	4,30	38	0,26	0
100	3	45	7,23	66	0,34	1
100	4	94	9,50	95	0,39	2
100	5	177	12,56	112	0,36	2
100	6	289	16,33	131	0,39	5
100	7	518	27,50	165	0,43	5
100	8	1040	20,28	206	0,50	17
100	9	2326	22,00	215	0,39	22
100	10	3323	21,00	249	0,44	27
200	2	137	5,25	90	0,32	0
200	3	435	9,19	130	0,32	3
200	4	1350	10,63	193	0,39	6
200	5	3074	16,63	241	0,40	5
200	6	5852	21,31	302	0,44	21
200	7	9994	20,57	348	0,43	19
200	8	13370	19,87	397	0,45	45
200	9	21089	30,85	462	0,47	62
200	10	29465	36,47	551	0,55	92

300	2	537	5,26	116	0,26	0
300	3	2114	11,02	200	0,34	3
300	4	6014	12,76	279	0,37	7
300	5	12636	16,31	370	0,42	16
300	6	21658	21,24	444	0,43	16
300	7	35161	19,24	504	0,40	34
300	8	55602	23,35	635	0,48	56
300	9	81181	52,61	718	0,47	99
300	10	113714	31,98	810	0,49	102
400	2	1426	6,13	160	0,27	0
400	3	5836	10,09	250	0,31	0
400	4	15993	14,34	384	0,39	4
400	5	34163	16,09	478	0,39	27
400	6	60937	20,35	609	0,42	28
400	7	84639	28,61	702	0,45	51
400	8	132973	23,00	841	0,47	114
400	9	175371	31,90	929	0,46	91
400	10	254784	42,23	1029	0,47	133
500	2	3173	5,82	187	0,25	0
500	3	12658	7,98	330	0,33	6
500	4	32542	12,77	486	0,39	5
500	5	62261	16,10	611	0,41	23
500	6	109125	17,27	744	0,42	41
500	7	171725	22,49	842	0,40	55
500	8	288846	21,36	1029	0,43	75
500	9	404986	39,08	1203	0,48	173
500	10	498245	39,23	1257	0,45	144
600	2	6364	5,51	212	0,23	0
600	3	25001	13,26	416	0,35	8
600	4	56917	11,00	569	0,38	13
600	5	113491	21,96	718	0,39	14
600	6	202756	23,56	892	0,41	34
600	7	308648	22,13	1084	0,46	102
600	8	474885	29,08	1214	0,44	113
600	9	572392	41,40	1399	0,46	183
600	10	958364	36,96	1518	0,44	212
700	2	10637	6,61	252	0,24	0
700	3	41803	10,36	457	0,32	2
700	4	97652	20,26	664	0,37	8
700	5	208630	15,07	861	0,40	36
700	6	321270	20,76	1059	0,43	52
700	7	518389	35,31	1215	0,42	72
700	8	887939	34,69	1450	0,45	144
700	9	1082248	31,68	1602	0,45	211
700	10	1933984	39,62	1798	0,46	277
800	2	17292	7,47	317	0,27	0
800	3	59197	9,44	527	0,33	3
800	4	140559	16,86	780	0,40	21
800	5	289028	16,93	984	0,41	36
800	6	481680	20,01	1192	0,42	53

800	7	828550	34,49	1403	0,44	105
800	8	1241990	33,17	1618	0,45	157
800	9	1868511	56,03	1896	0,47	224
800	10	2882334	42,84	2214	0,50	364
900	2	23970	8,05	334	0,25	0
900	3	86013	9,15	592	0,32	5
900	4	196775	16,41	856	0,39	11
900	5	434420	19,13	1100	0,40	37
900	6	709071	25,88	1382	0,45	76
900	7	1234367	23,92	1617	0,44	128
900	8	1880489	32,90	1864	0,45	185
900	9	2851900	74,05	2102	0,46	177
900	10	4335197	45,20	2387	0,47	354
1000	2	34457	6,15	377	0,25	0
1000	3	118329	10,16	666	0,33	6
1000	4	344773	13,68	934	0,37	23
1000	5	550080	21,15	1185	0,38	51
1000	6	1021880	21,06	1476	0,42	77
1000	7	1737402	32,06	1818	0,45	150
1000	8	2803095	55,25	2094	0,46	229
1000	9	4158404	51,58	2328	0,46	299
1000	10	5978247	36,22	2594	0,45	323
2000	2	297323	6,57	744	0,25	0
2000	3	1071208	12,29	1330	0,33	8
2000	4	2848675	18,60	1885	0,38	49
2000	5	5853508	22,71	2388	0,40	102
2000	6	11803382	23,89	2993	0,42	166
2000	7	19590974	33,46	3567	0,44	253
2000	8	53065716	36,93	4111	0,43	159
2000	9	84420533	41,88	4672	0,45	169
2000	10	155775351	46,83	5232	0,46	175
3000	2	1095253	8,70	1173	0,26	0
3000	3	4512605	11,37	1999	0,33	16
3000	4	17929957	4,56	3521	0,44	992
3000	5	31347309	5,91	5006	0,50	1515
3000	6	54764661	4,63	6180	0,59	2011
3000	7	98182013	4,37	7482	0,67	2563
3000	8	221599365	4,05	8784	0,75	3115
3000	9	525016717	8,67	10086	0,83	3667
3000	10	928434069	5,35	11388	0,91	4219
4000	2	3020137	9,75	1501	0,25	0
4000	3	11906890	2,67	2938	0,60	702
4000	4	61095263	5,47	4643	0,33	1277
4000	5	171422869	6,84	6620	0,92	2060
4000	6	369287984	5,47	8191	0,96	2699
4000	7	816467200	6,44	9897	0,44	3374
4000	8	2009248250	6,44	11603	0,91	4049
4000	9	3925920785	3,33	13309	0,39	4725
4000	10	10018582865	3,26	15015,8	0,86	5400