Formal Concept Analysis: Dipendenze tra attributi

Implementazione nella logica classica e fuzzy

Relatore: Prof.ssa Brunella Gerla

Laureando: Andrea Barbagallo

Tesi di Laurea Triennale in Informatica

Dipartimento di Scienze Teoriche ed Applicate
Università degli studi dell'Insubria
Anno Accademico 2018/19
Varese, Italia

Indice

1 Introduzione

La seguente tesi tratta due argomenti molto importanti della *Formal Concept Analysis*, ovvero la generazione delle *functional dependencies* e del *concept lattice* sia nella logica classica che fuzzy. In particolar modo verrà approfondito l'argomento delle dipendenze funzionali, andando prima a fornire alcune definizioni (sia per quanto riguarda la logica classica che quella fuzzy) e successivamente andando a calcolarne alcune tramite degli esempi. La generazione del reticolo dei concetti (concept lattice) verrà invece trattata solo marginalmente, andando semplicemente a fornirne le definizioni fondamentali.

Dopo aver trattato il funzionamento teorico delle dipendenze funzionali ed accennato il concetto di concept lattice, si andrà ad analizzare il software *Fuzzy Explorer*, realizzato sulla base delle definizioni teoriche precedentemente discusse. Il software è stato sviluppato utilizzando *Java 8* e la piattaforma software *JavaFX* per realizzarne l'interfaccia grafica; esso permette l'analisi di un *formal concept* inserito sotto forma tabellare. È possibile caricare la tabella fornendo un file Excel (*.xlsx*) oppure inserirla manualmente dall'interfaccia principale.
Una volta inseriti i dati, il software fornisce varie funzionalità, tra cui le principali sono:

- la generazione di un reticolo dei concetti, sia in logica classica che fuzzy
- la modifica dei reticoli dei concetti generati (vedremo in che modo nei capitoli successivi)
- la generazione delle dipendenze funzionali in logica classica
- fornire il *grado di verità* delle dipendenze funzionali in logica fuzzy

Le varie funzionalità del programma saranno descritte più approfonditamente nel capitolo 4 del documento. Sempre nel capitolo 4 sarà inoltre illustrata la progettazione del software tramite alcuni diagrammi UML e verranno presentate alcune parti di codice relative prevalentemente all'implementazione della parte algoritmica di quanto visto nei capitoli precedenti.

1.1 Strumenti

Durante la realizzazione del progetto di tesi sono stati utilizzati diversi strumenti, quali librerie o software esterni.

È di fondamentale importanza citare l'utilizzo del linguaggio *Java 8* e dell'*IDE NetBeans 8.2* che è stato usato per la realizzazione del codice sorgente del software.

Per quanto riguarda l'implementazione dell'interfaccia grafica, il programma utilizza *JavaFX*, ovvero un insieme di pacchetti grafici che consentono la progettazione e la creazione di GUI basandosi sul linguaggio di programmazione *Java*.
In questo ambito è stato utilizzato anche il software esterno *Scene Builder*, in modo da facilitare la generazione dei file *FXML*, ovvero file che definiscono la posizione e le caratteristiche degli elementi dell'interfaccia.

La rappresentazione grafica dei grafi avviene invece mediante l'utilizzo della libreria *JGraph*, suddivisa in due parti:

- JGraphT: utilizzata per definire la struttura del grafo da rappresentare, generato secondo le opportune funzioni logiche.
- JGraphX: utilizzata per disegnare a schermo i nodi ed i lati del grafo, posizionandoli secondo uno stile gerarchico, tramite l'opzione *"mxHierarchicalLayout"*.

Per accedere in lettura e scrittura ai file Excel, con estensione *.xlsx*, sono state utilizzate le librerie di *Apache POI*, le quali hanno reso possibile l'inserimento di alcune funzionalità utili nel programma. Il processo di build del programma è stato inoltre automatizzato utilizzando *Apache Ant*, il quale (attraverso l'uso di specifici comandi) permette la generazione di un *Fat Jar* contenente al suo interno tutto ciò di cui il software ha bisogno per eseguire.

Concludendo, si può riportare anche LaTeX, ovvero un linguaggio di markup per la creazione di file testuali, impiegato durante la stesura della tesi.

2 Formal Concept Analysis nella Logica Classica

Gli argomenti studiati e trattati durante questa tesi fanno parte della *Formal Concept Analysis* (FCA), ovvero una branca della matematica che si occupa di studiare ed analizzare dati. Per poter parlare di FCA è necessario comprendere i concetti base degli insiemi e delle relazioni, in modo tale da poterli considerare all'interno dei concetti ben più complessi che verranno presentati successivamente.

Un insieme X è un raggruppamento di elementi di varia natura. Gli elementi di interesse per quanto riguarda la FCA sono classificabili in due categorie: *oggetti* ed *attributi*. Consideriamo quindi:

$X = \{\textit{oggetti di interesse}\}$
$Y = \{\textit{caratteristiche degli oggetti}\}$

È facilmente intuibile che i due insiemi X ed Y sono strettamente legati da una relazione. Pensiamo ad esempio agli oggetti tangibili nella realtà in cui viviamo: questi dispongono di caratteristiche differenti, a seconda del tipo di oggetto che si sta considerando. Inoltre è semplice capire oggetti diversi possono condividere una o più caratteristiche.

Matematicamente è possibile esplicitare ciò che lega gli oggetti con le loro caratteristiche come una *relazione*, ovvero un *sottoinsieme* del prodotto cartesiano tra gli insiemi X ed Y.

Esempio: Siano X e Y rispettivamente insiemi di oggetti ed attributi, del tipo:

$X = \{\textit{acqua, vino}\}$
$Y = \{\textit{trasparente}\}$

In questo caso, possiamo dire che *acqua* R *trasparente*, poiché l'acqua gode della caratteristica di essere trasparente, e che *vino* $\not R$ *trasparente*, poiché il vino non gode della medesima proprietà.

Prese per buone queste premesse, possiamo introdurre la tripla di insiemi *(X, Y, R)*, dove $R \subseteq X \times Y$ è la relazione binaria tra i due insiemi X e Y. Questa tripla si definisce ***formal context***.

Per comprendere meglio la natura di un formal context, possiamo considerare la sua rappresentazione in forma tabellare. In questo caso, ogni riga della tabella è associata ad un determinato oggetto dell'insieme X, mentre ogni colonna ad uno specifico attributo in Y. Nei campi interni della tabella, invece, indichiamo con $\times$ gli elementi in relazione, mentre con uno spazio vuoto quelli non in relazione.

Esempio: Consideriamo un esempio simile al precedente; in questo caso abbiamo:

$X = \{acqua,\ vino,\ fungo\}$ è l'insieme di oggetti;
$Y = \{liquido,\ trasparente,\ velenoso\}$ è l'insieme di attributi.

R	**liquido**	**trasparente**	**velenoso**
acqua			
vino			
fungo			

Dopo aver descritto gli insiemi nella loro forma tabellare, completiamo la rappresentazione con le relazioni oggetto-attributo presenti; otteniamo:

R	**liquido**	**trasparente**	**velenoso**
acqua	$\times$	$\times$	
vino	$\times$		
fungo			$\times$

Dalla tabella possiamo evincere le relazioni presenti tra i due insiemi, come *acqua R trasparente* o $fungo \not R\ liquido$.

Dalla forma tabellare del formal context è possibile ottenere delle informazioni riguardanti le relazioni tra i vari attributi, sulla base degli oggetti inseriti nel contesto. In questo caso si parla di ***dipendenze funzionali***.

2.1 Dipendenze Funzionali

Partendo dal contesto (X, Y, R), è possibile generare delle implicazioni tra i suoi attributi, che vadano a descriverne il comportamento. Queste implicazioni prendono il nome di *dipendenze funzionali* e costituiscono una parte importante della Formal Concept Analysis, in quanto permettono di analizzare le regole su cui il contesto si basa. Per comprendere il procedimento che permette la generazione di queste dipendenze sono necessarie alcune definizioni.

Definizione: Sia X l'insieme degli oggetti e $A \subseteq X$ un suo sottoinsieme. Si definisce:

$$A^{\uparrow} = \{y \in Y \mid \forall a \in A : (a, y) \in R\}$$

ovvero l'insieme degli attributi condivisi da tutti gli oggetti del sottoinsieme A. Notiamo che per ottenere l'insieme di attributi $A^{\uparrow}$ si parte dall'insieme A di oggetti.

Definizione: Sia Y l'insieme degli attributi e $B \subseteq Y$ un suo sottoinsieme. Si definisce:

$$B^{\downarrow} = \{x \in X \mid \forall b \in B : (x, b) \in R\}$$

ovvero l'insieme degli oggetti che godono di tutte le proprietà appartenenti al sottoinsieme B. Anche questa volta, si nota che per ottenere l'insieme di oggetti $B^{\downarrow}$ si parte dall'insieme B di attributi.

Definizione: Una ***dipendenza funzionale*** in un contesto (X, Y, R) è una relazione $A \to B$, con $A, B \subseteq Y$, dove:

- X è l'insieme degli oggetti che appartengono al contesto;
- Y è l'insieme degli attributi che appartengono al contesto;
- R è un sottoinsieme dell' insieme di coppie $(x, y) \in X \times Y$

Essa significa che $A^{\downarrow} \subseteq B^{\downarrow}$, ovvero che ogni oggetto che possiede tutti gli attributi di A possiede anche tutti gli attributi di B.

Definizione: Sia $P \subseteq Y$, P è uno ***pseudo-intent*** se $P \neq P^{\downarrow\uparrow}$ e $\forall$ pseudo-intent $Q \mid Q \subset P$ si ha $Q^{\downarrow\uparrow} \subset P$.

Definizione: Una ***implication cover*** è un sottoinsieme dal quale tutte le altre implicazioni possono essere derivate (utilizzando le regole di Armstrong, che però non saranno trattate in questo documento).

Definizione: Una ***implication base*** è la minima implicaton cover, ovvero l'insieme di implicazioni più piccolo possibile, dal quale sia possibile ricavare tutte le altre implicazioni.

2.1.1 Calcolare le dipendenze funzionali

L'obiettivo è quello di ottenere una **implication base**, grazie alla quale sarà poi eventualmente possibile ottenere tutte le altre implicazioni del contesto tramite le *regole di Armstrong* (che però non verranno trattate in questo documento).

Definizione: La ***base di Duquenne-Guigues*** è una implication base particolare, dove ogni implicazione è uno pseudo-intent così definito:

$$\{P \to (P^{\downarrow\uparrow} - P) \mid P \text{ è uno pseudo-intent}\}$$

Esempio: Riprendiamo la tabella d'esempio utilizzata precedentemente:

R	**liquido**	**trasparente**	**velenoso**
acqua	×	×	
vino	×		
fungo			×

Proviamo ora a calcolare su di essa la base di Duquenne-Guigues, utilizzando le definizioni date. Per farlo consideriamo uno per uno tutti i possibile sottoinsiemi di Y (attributi), ed andiamo a verificare se è o meno uno pseudo-intent.

Per comodità verrà utilizzata la seguente notazione:

- liq = liquido
- tra = trasparente
- vel = velenoso

Procediamo quindi per ogni sottoinsieme di oggetti P:

P	$P^{\downarrow}$	$P^{\downarrow\uparrow}$	***P è uno pseudo-intent?***
Ø	acqua, vino, fungo	Ø	No, perché $A = A^{\downarrow\uparrow}$
liq	acqua, vino	liq	No, perché $A = A^{\downarrow\uparrow}$
tra	acqua	liq, tra	Sì
vel	fungo	vel	No, perché $A = A^{\downarrow\uparrow}$
liq, tra	acqua	liq, tra	No, perché $A = A^{\downarrow\uparrow}$
liq, vel	Ø	liq, tra, vel	Sì
tra, vel	Ø	liq, tra, vel	No, perché {liq, tra} $\not\subset$ {tra, vel}
liq, tra, vel	Ø	liq, tra, vel	No, perché $A = A^{\downarrow\uparrow}$

Degli otto possibili sottoinsiemi P di Y abbiamo ricavato due pseudo-intent, ovvero i sottoinsiemi {*trasparente*} e {*liquido, velenoso*}.

A questo punto applichiamo agli pseudo-intent trovati la formula:

$$\{P \to (P^{\downarrow\uparrow} - P) \mid \quad P \text{ è uno pseudo-intent } \}$$

Ottenendo:

$$\{trasparente\} \to \{liquido\}$$

$$\{liquido, velenoso\} \to \{trasparente\}$$

La prima implicazione risulta ovvia, in quanto possiamo osservare dalla tabella che ogni volta che un oggetto è trasparente allora è anche liquido (in questo caso abbiamo un solo oggetto trasparente: l'acqua). La seconda implicazione invece, è vera anche se non è presente nessun oggetto (nel nostro contesto) sia liquido che velenoso, in quanto non esiste un controesempio che la renda falsa.

Oltre alle dipendenze funzionali, tramite il formal context in forma tabellare è possibile ricavarne una rappresentazione grafica sotto forma di grafo, chiamata chiamata ***concept lattice***.

2.2 Concept Lattice

Un concept lattice, ovvero un *reticolo dei concetti*, è un grafo i cui nodi costituiscono una coppia di insiemi. Per capire la struttura di questo reticolo è necessario considerare alcune premesse.

Definizione: Sia L un concept lattice, siano X e Y gli insiemi di oggetti ed attributi di partenza, allora $\forall A_i \in \mathcal{P}(X)$, con $0 < i \leq 2^{|A|}$, si ha che:

$$\langle A_i, A_i^{\uparrow} \rangle \in L \iff A_i \equiv A_i^{\uparrow\downarrow}$$

dove $\langle A_i, A_i^{\uparrow} \rangle$ è detto ***formal concept***, ovvero un nodo $n \in$ L. Gli insiemi $A_i, A_i^{\uparrow}$ sono detti rispettivamente *intent* ed *extent* di n e sono generati come descritto precedentemente.

Nota: $A^{\uparrow\downarrow} \equiv B^{\downarrow} \iff B \equiv A^{\uparrow}$.

I formal concepts sono legati tra loro in base ad una *relazione d'ordine* che rende la struttura del grafo *gerarchica*.

Definizione: Sia L un concept lattice e $\langle A_1, B_1 \rangle$, $\langle A_2, B_2 \rangle$ due formal concepts di L, allora la ***relazione d'ordine*** tra i due nodi è definita come:

$$(A_1, B_1) \leq (A_2, B_2) \iff (A_1 \subseteq A_2 \iff B_2 \subseteq B_1)$$

Dal punto di vista grafico, ogni volta che una di queste relazioni d'ordine è verificata, si ottiene un lato del grafo che lega i due nodi della relazione.

Definizione: Si dice ***diagramma di Hasse*** un grafo in grado di rappresentare insiemi ***parzialmente ordinati***, ovvero insiemi che godono di una relazione d'ordine. Nel nostro caso possiamo dire che i concept lattice appartengono alla famiglia dei diagrammi di Hasse, pertanto possono essere rappresentati similmente al grafo della figura successiva, il quale mostra la relazione di divisibilità tra numeri naturali.

Dalla figura si deduce la transitività delle relazioni; consideriamo ad esempio il numero 30: questo numero è divisibile per 15 ed infatti esiste un lato che collega questi due nodi, ma è anche divisibile per 5, pur non essendoci alcun lato diretto tra 30 e 5. La natura di questo grafo introduce quindi delle **relazioni transitive implicite** che, pur esistendo, non vengono rappresentate direttamente ma sono sottintese attraverso le relazioni padre-figlio.

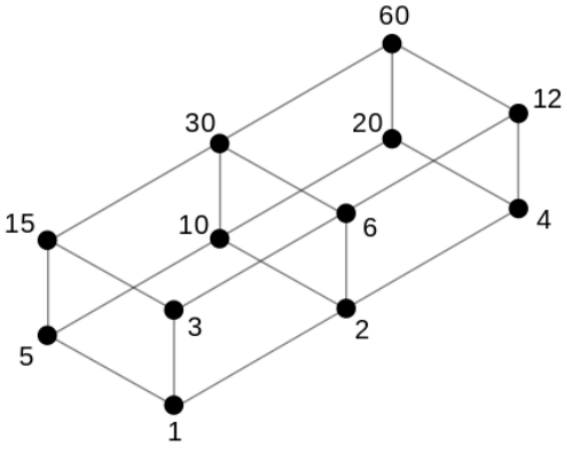


Diagramma di Hasse: divisibilità tra numeri naturali

Per ultimare la creazione del reticolo dei concetti, dobbiamo quindi tenere conto delle relazioni transitive implicite: consideriamo i nodi $N_1, N_2, N_3 \in L$ e supponiamo che $N_1 \leq N_2 \leq N_3$ siano definiti secondo la relazione d'ordine precedentemente descritta. In questo caso è evidente che $N_1 \leq N_3$ per via della proprietà transitiva della relazione; tuttavia, il reticolo dei concetti *non* prevede il lato $\overline{N_3 N_1}$ dato che esiste già il percorso $\overline{N_3 N_2}+\overline{N_2 N_1}$ che lega i due nodi N_1 ed N_3 e quest'informazione risulterebbe ridondante.

2.3 Contesti Multivalore

Nella realtà in cui viviamo è probabile trovarsi di fronte a tabelle di dati non descritte esclusivamente con i valori booleani *vero* o *falso*. È invece semplice pensare a tabelle che esprimono valori di varia natura, come ad esempio il sesso di una persona (*"Maschio", "Femmina"*) o la sua età (*10, 15, 23, ...*). In questo paragrafo analizzeremo come tradurre in logica booleana i diversi tipi di dati che possono popolare una tabella relazionale tra oggetti ed attributi.

Definizione: Si definisce ***contesto multivalore*** la quadrupla (X, Y, W, R) formata dagli insiemi *X, Y* di oggetti ed attributi, dall'insieme W dei possibili valori attribuibili alle proprietà e dall'insieme R delle relazioni che legano un oggetto x ad un attributo y con un verto valore w.

Sostanzialmente, ciò che l'insieme delle relazioni R contiene sono delle triple, scelte all'interno dell'insieme $(x, y, w) \in X \times Y \times W$; nel caso in cui una tripla $(x, y, w) \in R$, si intende che *"l'attributo y ha valore w per l'oggetto x"* e si scrive:

$$y(x) = w$$

Per descrivere le diverse tipologie di dato, si fa riferimento alla tabella che segue.

R	**Sesso**	**Corso**	**Media**
Giovanni	M	Informatica	23.5
Giulietta	F	Economia	24
Pasquale	M	Economia	22
Paolina	F	Informatica	24
Ugo	M	Medicina	25

Definizione: Si dice ***scala nominale*** la tripla $(W_m, W_m, =)$, dove W_m è il valore w per l'attributo m. Questa scala si adatta perfettamente alle rappresentazioni degli attributi *nominali*, ovvero quegli attributi che definiscono una categoria.

Nel nostro caso un esempio è la colonna *Corso*, la quale categorizza gli studenti in base al corso che frequentano. Applicando la scala a quella colonna, si ottiene:

$=$	**Informatica**	**Economia**	**Medicina**
Informatica	$\times$		
Economia		$\times$	
Medicina			$\times$

Definizione: Si dice ***scala dicotomica*** il caso particolare della scala nominale che prevede valori *mutualmente esclusivi* con valori del tipo *vero* o *falso*.

Semplicemente, nel nostro esempio potremmo considerare questa scala per la colonna *Sesso*, ottenendo:

$=$	**M**	**F**
M	$\times$	
F		$\times$

Definizione: Si dice ***scala ordinale*** la tripla $(W_m, W_m, \leq)$. Questa scala si adatta a rappresentare valori ordinabili che assumono valori all'interno dell'insieme $\mathbb{R}$ dei numeri reali.

Per quanto riguarda il nostro esempio, possiamo considerare questa scala per la colonna *Media*, ottenendo:

$\leq$	$\leq$**22**	$\leq$**23**	$\leq$**24**	$\leq$**25**
22	$\times$	$\times$	$\times$	$\times$
23		$\times$	$\times$	$\times$
23.5			$\times$	$\times$
24			$\times$	$\times$
25				$\times$

Facendo riferimento a quest'ultimo tipo di scala, possiamo introdurre una sua versione alternativa, più precisa.

Definizione: Si dice ***scala interordinale*** la contrapposizione tra le triple $(W_m, W_m, \leq)|(W_m, W_m, \geq)$, utile a rappresentare con precisione i valori ordinabili che assumono valori all'interno dell'insieme $\mathbb{R}$ dei numeri reali.

Modifichiamo la tabella della scala precedente applicando la scala interordinale.

$\leq\geq$	$\leq$**22**	$\leq$**23**	$\leq$**24**	$\leq$**25**	$\geq$**22**	$\geq$**23**	$\geq$**24**	$\geq$**25**
22	×	×	×	×	×			
23		×	×	×	×	×		
23.5			×	×	×	×		
24			×	×	×	×	×	
25				×	×	×	×	×

Una volta esaminate le tipologie di scala più importanti (esistono numerosissimi tipi di scala in grado di adattarsi perfettamente ai contesti più vari), rendiamo la tabella multivalore iniziale una tabella booleana, che quindi può essere rielaborata per generare un reticolo dei concetti o per calcolare le sue dipendenze funzionali.

R	**M**	**F**	**Informatica**	**Economia**	**Medicina**	$\leq$**22**	$\leq$**23**	$\leq$**24**	$\leq$**25**	$\geq$**22**	$\geq$**23**	$\geq$**24**	$\geq$**25**
Giovanni	×		×					×	×	×	×		
Giulietta		×		×				×	×	×	×	×	
Pasquale	×			×		×	×	×	×	×			
Paolina		×	×					×	×	×	×	×	
Ugo	×				×				×	×	×	×	×

Gli argomenti trattati in questo paragrafo non sono stati implementati tra le funzionalità del programma finale poiché si suppone che le tabelle prese in input siano già state rielaborate in modo adeguato. Le tabelle accettate dal software, infatti, prevedono solamente valori booleani oppure, come vedremo in seguito, valori compresi tra $[0, 1]$.

3 Da Logica Classica a Logica Fuzzy

Gli argomenti che abbiamo trattato nel capitolo precedente possono essere riadattatati secondo alcuni accorgimenti alla logica fuzzy, vedendo in un certo senso i casi della logica classica come dei casi particolari.

Ciò che differenzia le due logiche è il range dei valori assegnabili alle relazioni tra oggetti ed attributi. Nella logica classica, i possibili valori sono *vero* o *falso*, riconducibili anche in termini numerici a *1* o *0*.

Nella logica fuzzy i valori possibili non si limitano semplicemente a *1* o *0* ma includono tutto il range dell'insieme [0, 1].

Esempio: Consideriamo la seguente tabella in logica classica.

R	**liquido**	**trasparente**	**velenoso**
acqua	×	×	
vino	×		
fungo			×
sapone	×	×	×

È evidente che ci siano alcune situazioni in cui la logica classica possa risultare ambigua; un esempio potrebbe essere il sapone che non è considerabile né liquido né solido, ma *"denso"*. Provando però ad associare dei valori diversi alla relazione, possiamo specificare quali oggetti si avvicinano più di altri a soddisfare un certo attributo. Otteniamo, con valori puramente indicativi, una tabella di questo genere:

R	**liquido**	**trasparente**	**velenoso**
acqua	1.0	1.0	0.0
vino	1.0	0.25	0.0
fungo	0.0	0.0	1.0
sapone	0.5	0.5	0.5

3.1 Operatori della Logica Fuzzy

Prima di entrare davvero in merito alla forma dei grafi fuzzy, ci soffermiamo in modo un po' più formale sugli argomenti che questo nuovo tipo di logica tocca.

In generale il contesto in cui ci troviamo, ovvero l'insieme degli *ingredienti* che consentono la creazione di questo particolare tipo di grafo, è una **struttura** del tipo:

$$\mathbf{L} = \langle L, \wedge, \vee, \otimes, \rightarrow, 0, 1 \rangle, \quad \text{dove:}$$

- L è il contesto su cui si sta costruendo il grafo.
- $\wedge$ è detto *infima* e rappresenta l'operatore di *minimo*;
- $\vee$ è detto *suprema* e rappresenta l'operatore di *massimo*;
- $\otimes$ è l'operatore di *normalizzazione*, tale che $a \otimes b = \vee([a] + b - 1, 0)$; la notazione $[a]$ indica la *parte intera* di a, ovvero se $a < 1$, $a = 0$.
- $\rightarrow$ è l'operatore di *implicazione*, tale che $a \rightarrow b = \wedge(1 - a + b, 1)$;
- 0,1 indicano gli estremi del range di valori considerato dagli operatori.

3.2 Dipendenze Funzionali nella Logica Fuzzy

Come accade per la logica classica, anche nella logica fuzzy è possibile studiare le dipendenze tra gli attributi di un determinato contesto. Gli operatori citati nella sezione precedente vengono applicati sui *pesi* che possono essere attribuiti agli oggetti ed agli attributi, oltre che alle relazioni.

Definizione: I *pesi* prendono il nome di ***grado di verità*** e possono essere visti come delle funzioni:

$$A : X \rightarrow L$$

$$B : Y \rightarrow L$$

Definizione: Si dice ***fuzzy set*** un insieme A, tale per cui i suoi elementi a appartengono all'insieme con un grado di verità compreso tra 0 e 1. Sia $x \in X$ un oggetto, allora si indica con *A(x)* il valore con cui l'oggetto x appartiene ad A e $A(x) \in L$, ovvero appartiene all'insieme dei valori possibili.

Notazione: Sia A un fuzzy set di oggetti e sia $x \in X$ un oggetto. Se $x \in A$ con grado $a = A(x)$, allora si scrive:

$$A = \{a/x, ...\}$$

Definizione: Sia $\langle X, Y, R \rangle$ la tripla contenente rispettivamente l'insieme di oggetti, attributi e la relazione $R : X \times Y \to L$. Per ogni fuzzy set A e B, si definiscono i fuzzy set $A^{\uparrow}$ e $B^{\downarrow}$, detti ***intent*** ed ***extent***, tali che rispettivamente:

$$A^{\uparrow}(y) = \bigwedge_{x \in X} (A(x) \to R(x, y))$$
$$B^{\downarrow}(x) = \bigwedge_{y \in Y} (B(y) \to R(x, y))$$

Il significato di questi insiemi può essere interpretato come: $A^{\uparrow}(y)$ *è il grado di verità con cui l'attributo y è condiviso tra tutti gli oggetti di A.*

Definizione: Sia x un oggetto ed M il fuzzy set di attributi di x. Siano inoltre A e B due fuzzy sets di attributi, allora si indica con $||A \to B||_M$ il **grado di verità** di $A \to B$ in M, ovvero quanto $A \to B$ è vero per l'oggetto x. In questo caso si ha che:

$$||A \to B||_M = S(M, A) \to S(M, B)$$

Definizione: Sia $\langle X, Y, R\rangle$ un contesto in cui sono presenti rispettivamente il fuzzy set di oggetti, il fuzzy set di attributi e i gradi della relazione che lega i due insiemi. Si indica con $||A \to B||_{\langle X,Y,R\rangle}$ il **grado di verità** di $A \to B$ **nel contesto** $\langle X, Y, R\rangle$. In questo caso si ha che:

$$||A \to B||_{\langle X,Y,R\rangle} = S(A^{\downarrow\uparrow}, B) = \bigwedge_{y_1 \in A^{\downarrow\uparrow}, y_2 \in B} (y_1 \to y_2)$$

Esempio: Consideriamo la seguente tabella:

R	$\mathbf{y_1}$	$\mathbf{y_2}$	$\mathbf{y_3}$
$\mathbf{x_1}$	1	0.5	0.5
$\mathbf{x_2}$	1	1	1
$\mathbf{x_3}$	0	0	0.5

Consideriamo poi gli insiemi A e B e procediamo con il calcolo di $||A \to B||_{\langle X,Y,R\rangle}$:

$$A = \{1/y_1,\ 0.5/y_2,\ 0/y_3\}$$

$$B = \{0/y_1,\ 0.5/y_2,\ 1/y_3\}$$

Il primo passo è quello di calcolare $A^{\downarrow}$, secondo le regole già descritte precedentemente. Otteniamo:

$$\begin{aligned}
A^{\downarrow}(x_1) &= A(y_1) \to R(x_1, y_1) \wedge A(y_2) \to R(x_1, y_2) \wedge A(y_3) \to R(x_1, y_3) \\
&= 1 \to 1 \wedge 0.5 \to 0.5 \wedge 0 \to 0.5 = 1 \wedge 1 \wedge 1 = 1 \\
A^{\downarrow}(x_2) &= A(y_1) \to R(x_1, y_1) \wedge A(y_2) \to R(x_1, y_2) \wedge A(y_3) \to R(x_1, y_3) \\
&= 1 \to 1 \wedge 0.5 \to 1 \wedge 0 \to 1 = 1 \wedge 1 \wedge 1 = 1 \\
A^{\downarrow}(x_2) &= A(y_1) \to R(x_1, y_1) \wedge A(y_2) \to R(x_1, y_2) \wedge A(y_3) \to R(x_1, y_3) \\
&= 1 \to 0 \wedge 0.5 \to 0 \wedge 0 \to 0.5 = 0 \wedge 0.5 \wedge 1 = 0
\end{aligned}$$

Per quanto riguarda il secondo passo, dobbiamo considerare l'insieme $A^{\downarrow}$ al posto dell'insieme A per calcolare $A^{\downarrow\uparrow}$.

$$\begin{aligned}
A^{\downarrow\uparrow}(y_1) &= A^{\downarrow}(x_1) \to R(x_1, y_1) \wedge A^{\downarrow}(x_2) \to R(x_2, y_1) \wedge A^{\downarrow}(x_3) \to R(x_3, y_1) \\
&= 1 \to 1 \wedge 1 \to 1 \wedge 0 \to 0 = 1 \\
A^{\downarrow\uparrow}(y_2) &= A^{\downarrow}(x_1) \to R(x_1, y_2) \wedge A^{\downarrow}(x_2) \to R(x_2, y_2) \wedge A^{\downarrow}(x_3) \to R(x_3, y_2) \\
&= 1 \to 0.5 \wedge 1 \to 1 \wedge 0 \to 0 = 0.5 \\
A^{\downarrow\uparrow}(y_3) &= A^{\downarrow}(x_1) \to R(x_1, y_3) \wedge A^{\downarrow}(x_2) \to R(x_2, y_3) \wedge A^{\downarrow}(x_3) \to R(x_3, y_3) \\
&= 1 \to 0.5 \wedge 1 \to 1 \wedge 0 \to 0.5 = 0.5
\end{aligned}$$

Concludiamo l'esempio calcolando $S(A^{\downarrow\uparrow}, B)$.

$$\begin{aligned}
S(A^{\downarrow\uparrow}, B) &= A^{\downarrow\uparrow}(y_1) \to B(y_1) \wedge A^{\downarrow\uparrow}(y_2) \to B(y_2) \wedge A^{\downarrow\uparrow}(y_2) \to B(y_2) \\
&= 1 \to 0 \wedge 0.5 \to 0.5 \wedge 0.5 \to 1 = 0
\end{aligned}$$

3.3 Fuzzy Concept Lattice

Anche per quanto riguarda la logica fuzzy è possibile rappresentare graficamente un determinato contesto e, per farlo, sono necessarie le seguenti definizioni.

Definizione: Siano A_1 e A_2 due fuzzy set di oggetti. Se $\forall x \in X$ si ha che $A_1(x) \leq A_2(x)$ allora si scrive $A_1 \subseteq A_2$ e si dice che A_1 è ***completamente incluso*** in A_2.

Se $A_1 \subseteq A_2$ ma $\exists x \in X | A_1(x) < A_2(x)$ allora si scrive $A_1 \subset A_2$ e si dice che l'insieme A_1 è ***strettamente incluso*** in A_2.

Il discorso è valido in modo analogo anche per i fuzzy set di attributi.

Definizione: Le coppie $\langle A_i, A_i^{\uparrow} \rangle$ tali che $A_i \equiv A_i^{\uparrow\downarrow}$, si definiscono ***formal fuzzy concept*** e sono generate come descritto precedentemente. L'insieme di tutte le i coppie costituisce il ***Fuzzy Concept Lattice***.

Definizione: Sia L un Fuzzy Concept Lattice e $\langle A_1, B_1 \rangle$, $\langle A_2, B_2 \rangle$ due nodi di L, allora la ***relazione d'ordine*** tra i due nodi è definita come:

$$(A_1, B_1) \leq (A_2, B_2) \iff (A_1 \subseteq A_2 \iff B_2 \subseteq B_1)$$

Dal punto di vista grafico, ogni volta che questa relazione d'ordine è verificata, si ottiene un lato del grafo che lega i due nodi della relazione.

4 Il nostro software: FuzzyExplorer

Concretamente il nostro software tocca gran parte degli argomenti trattati nei capitoli precedenti, potendo quindi essere considerato uno strumento per facilitare l'analisi dei concetti. In questo capitolo si discutono tutte le funzionalità implementate, l'interfaccia grafica realizzata ed i risultati che il software produce.

4.1 Progettazione

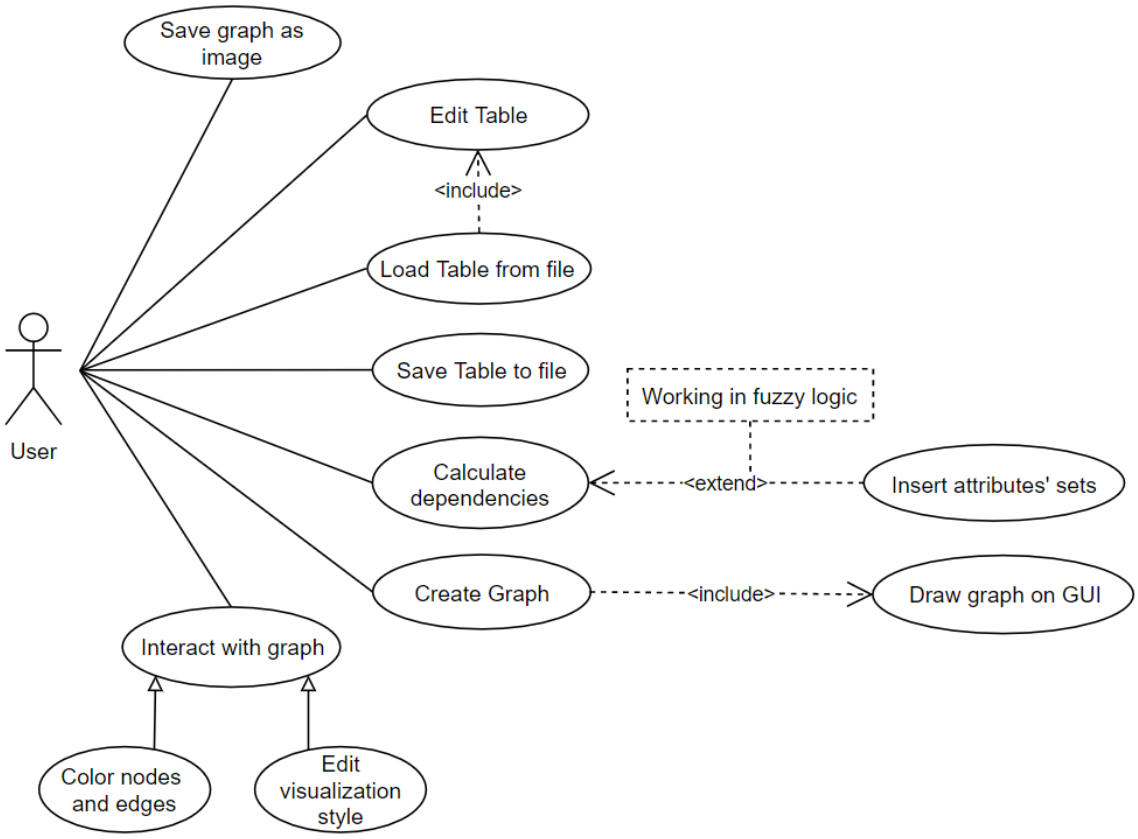
Quando si desidera realizzare un software di qualsiasi natura ci si trova immediatamente di fronte ad una delle fasi più importanti: la progettazione.

Durante lo sviluppo di questo progetto abbiamo dovuto apportare più volte modifiche al codice ed all'organizzazione delle classi che implementano le funzionalità. Sapendo fin dall'inizio che ci saremmo trovati in questa situazione, abbiamo mantenuto il codice il più **modificabile** possibile, implementando metodi e classi in grado di gestire dati **generici**.

Esempio: Un esempio è sicuramente l'accettazione dei valori calcolabili all'interno delle tabelle in logica fuzzy; in questo caso avremmo potuto implementare l'algoritmo tenendo in considerazione un sottoinsieme di valori in $[0, 1]$ come, ad esempio, $\{0, 0.5, 1\}$. Quello che però abbiamo deciso di fare è stato implementare direttamente la possibilità di accettare qualsiasi valore nel range stabilito, in modo da non dover gestire eventuali casi limite in un secondo momento.

4.1.1 UML: Use Case Diagram

La fase di progettazione ha inizio con l'analisi dei requisiti e con l'identificazione dei **casi d'uso**, ovvero delle funzionalità di cui l'utente che utilizza il prodotto software può usufruire. Grazie al diagramma dei casi d'uso è possibile rappresentare graficamente queste funzionalità, in modo da comprendere rapidamente ciò che un utente è in grado di fare. In figura è mostrato il diagramma relativo al nostro progetto.



Nel nostro caso è stato possibile identificare un solo tipo di utente, dato che non sono previsti amministratori o utenti speciali. L'utente in questione può svolgere attività di diverso tipo:

- **Edit Table**: modificare la tabella; questa funzionalità comprende, oltre che la modifica dei valori, l'aggiunta e la rimozione delle righe/colonne o la creazione di una nuova tabella.

- **Load Table from file**: caricare la tabella da un file *xlsx*; utilizzando questa funzionalità la tabella viene compilata con i valori contenuti nel file caricato, modificando automaticamente quella precedente (relazione *<include>*).

- **Save Table to file**: salvare la tabella su un file *xlsx*; utilizzando questa funzionalità la tabella viene salvata su un file nel formato accettato dal nostro software. Il file in questione potrà poi essere caricato in un secondo momento per recuperare i dati della tabella salvata.

- **Calculate depencencies**: calcolare le dipendenze funzionali tra gli attributi registrati nella tabella. Se si sta utilizzando la logica classica, verranno calcolate tutte le dipendenze presenti; altrimenti verrà richiesto all'utente l'inserimento degli insiemi fuzzy di attributi di cui si vuole calcolare il grado di verità (relazione *<extend>*).

- **Create graph**: creare il grafo, calcolando le relazioni tra oggetti ed attributi registrati nella tabella. Una volta elaborati i dati, il grafo viene visualizzato sull'interfaccia grafica (relazione *<include>*). Il grafo verrà creato in una finestra secondaria, in modo che l'utente possa continuare ad interagire con la tabella.

- **Interact with graph**: interagire con il grafo; utilizzando questa funzionalità sarà possibile colorare i nodi e i lati di interesse, modificare lo stile di visualizzazione aggiungendo tag, modificando il tipo di grafo da visualizzare o nascondendo i nodi che comprendono determinati oggetti o attributi.

- **Save graph as image**: salvare il grafo, eventualmente modificato tramite apposite interazioni, in un file *png*.

4.1.2 Suddivisione in Packages

Entriamo più nel dettaglio rispetto alla struttura del codice, discutendo come le classi presenti vengono *raggruppate* in packages. È previsto l'utilizzo di tre packages:

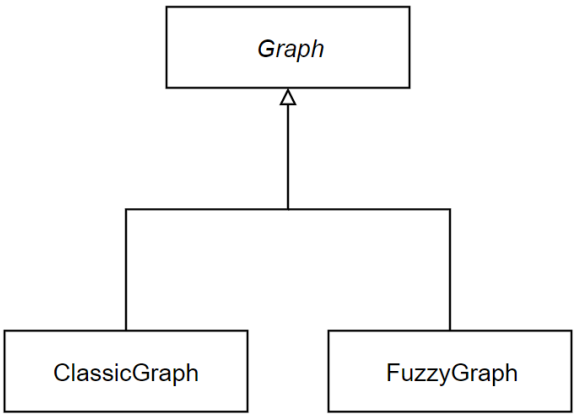
- **fuzzy.explorer**: package contenente tutte le classi relative ai grafi e al calcolo delle dipendenze funzionali. Gli algoritmi che, sfruttando le definizioni descritte nelle sezioni precedenti, implementano l'elaborazione dei dati sono contenuti in queste classi.

- **scene.controllers**: package contenente tutte le classi che operano sulla GUI e, di conseguenza, sull'interazione con l'utente. I risultati forniti dall'elaborazione dei dati, vengono visualizzati tramite le classi di questo package.

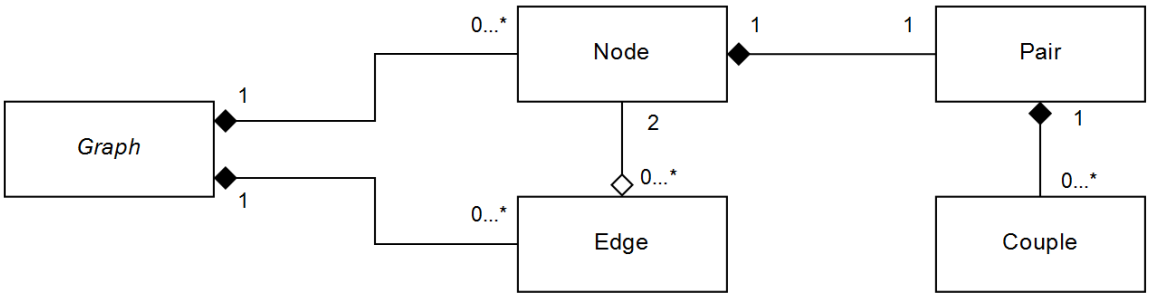
- **shared**: package contenente tutte le classi in comune tra i due package precedenti. Alcune delle classi in questione costituiscono interfacce tra le classi degli altri due packages, altre invece implementano delle funzionalità generiche, come la lettura dai file Excel.

4.1.3 UML: Class Diagrams

Uno dei diagrammi UML più significativi è sicuramente il Class Diagram, il quale mette in evidenza le relazioni tra le classi. Per mantenere l'intuitività di questi diagrammi è necessario mantenere le loro dimensioni ridotte, altrimenti il rischio è quello di ottenere un numero così elevato di collegamenti e classi da rendere la rappresentazione di difficile comprensione. Per questo motivo, in questa sezione ci limitiamo a presentare solo due sezioni del diagramma completo, ovvero quelle che possono essere utili alla comprensione della struttura dei grafi ed a come questi vengono utilizzati.



Il primo sotto-diagramma di rilievo è sicuramente quello che mostra la relazione di ereditarietà tra la **classe astratta** *Graph* e le sottoclassi *ClassicGraph*, relativa alla logica classica, e *FuzzyGraph*, per quanto riguarda la logica fuzzy. Il vantaggio di utilizzare una struttura di questo genere è la possibilità di garantire alla classe chiamante, il *GraphController*, di ottenere il grafo adatto in base al contenuto della *Table* e di utilizzare le proprietà dei grafi senza preoccuparsi effettivamente di quale grafo sta utilizzando. Il software, infatti, stabilisce automaticamente in base al contenuto della tabella il tipo di grafo da generare: se la tabella presenta solo valori booleani, il grafo disegnato sarà relativo alla logica classica, altrimenti a quella fuzzy.



Con il secondo sotto-diagramma di questa sezione viene mostrata la struttura interna del grafo, ovvero le classi con cui questo interagisce per fornire poi la visualizzazione finale rispettando le regole descritte nelle definizioni delle prime sezioni.

Il grafo contiene al suo interno una lista di *Node* (nodi) ed una lista di *Egde* (lati), gestite entrambe tramite la struttura dati Java *ArrayList*. Un nodo è composto dalla

sua etichetta, ovvero la *Pair* che rappresenta la coppia $\langle$*insieme oggetti*$\rangle$ / $\langle$*insieme attributi*$\rangle$. Ognuno di questi due insiemi è implementato da una *ArrayList<Couple>*. Una *Couple*, essendo eventualmente contenuta in un insieme fuzzy di oggetti/attributi, viene considerata come una coppia del tipo grado-nome, rispettivamente con una variabile *float* ed una *String*. Il legame tra le varie classi è rappresentato da **relazioni di composizione**, dato che l'oggetto contenuto non può mai esistere in assenza del suo contenitore (ad esempio, se il grafo venisse cancellato, un nodo non avrebbe più senso di esistere). L'unica **relazione di aggregazione** è quella tra i lati ed i nodi dato che, eliminando un lato, i nodi apparterrebbero comunque al grafo.

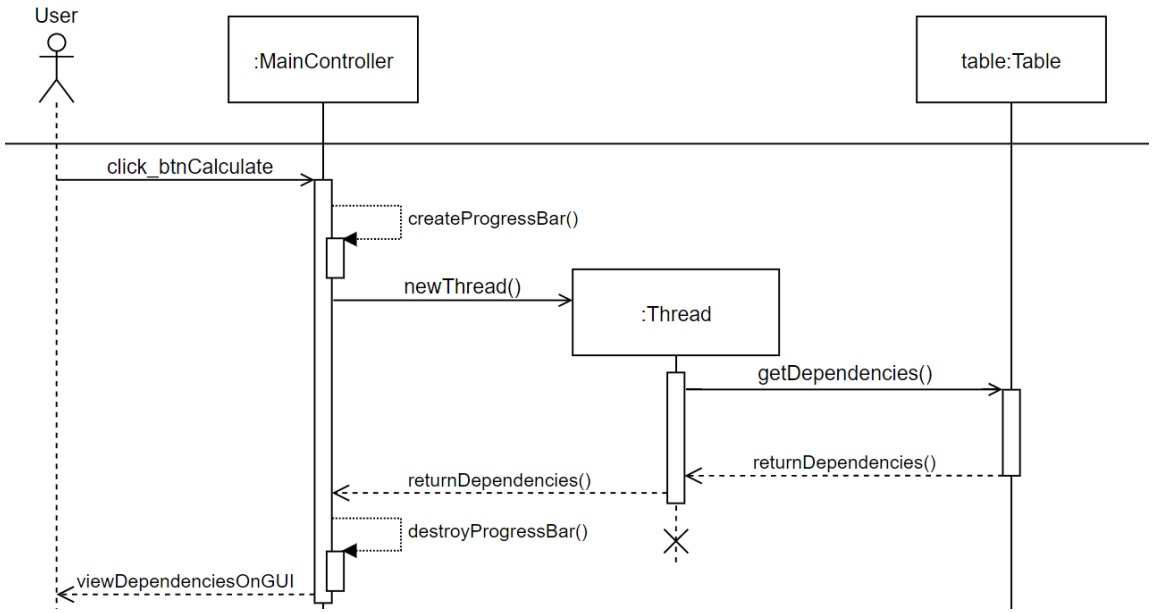
Le cardinalità descritte all'interno del diagramma sono piuttosto intuitive e non necessitano di particolari accorgimenti, se non quella relativa alla relazione di aggregazione. In quel caso abbiamo stabilito che un nodo possa anche non appartenere ad alcun lato, in modo da includere nella rappresentazione anche il caso limite in cui l'utente utilizzi una tabella con n oggetti ed m attributi ma alcuna relazione tra essi.

Esempio: Ad esempio, considerando gli insiemi:

$$X = \{sasso, candela\}$$
$$Y = \{vola, parla, ride\}$$

Otteniamo l'insieme delle relazioni $R = \varnothing$. In questo caso, l'unico nodo presente nel grafo sarà etichettato con la Pair $\langle\{\}, \{vola, parla, ride\}\rangle$ e non sarà presente alcun lato.

4.1.4 UML: Sequence Diagram



In questa sezione viene mostrato un Sequence Diagram, ovvero un diagramma che consente di rappresentare non solo l'interazione tra i componenti del software, ma anche lo scambio di messaggi relativi a quell'interazione. Il diagramma in figura si riferisce alla situazione in cui l'utente, cliccando sull'apposito bottone, si aspetta di ottenere l'elenco delle dipendenze funzionali relative alla tabella inserita.

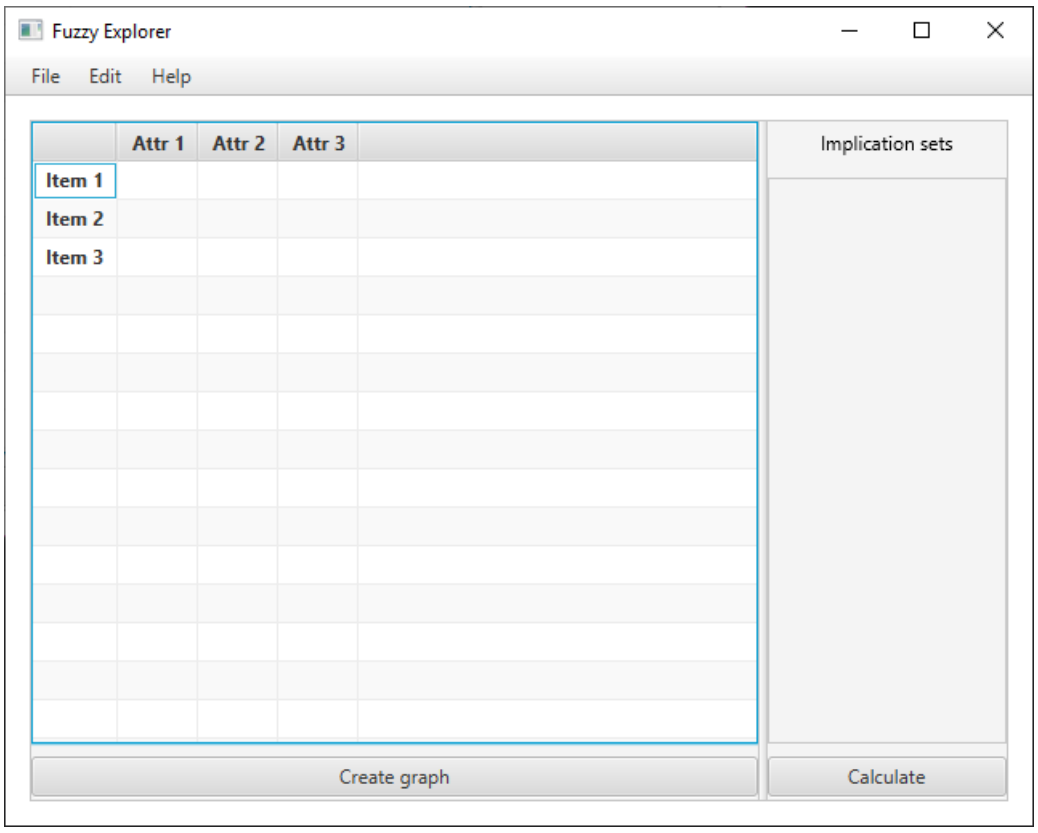
Più precisamente, l'attività descritta dal diagramma consiste nelle seguenti fasi:

1. l'utente preme il bottone *Calculate* ed attende il calcolo delle dipendenze;
2. il MainController crea una barra di caricamento in modo che l'utente, durante l'elaborazione dei dati, non abbia l'impressione che il programma si sia bloccato;
3. il MainController crea un **Thread** che, eseguendo un task, interroga la tabella per ottenere le dipendenze;
4. tramite alcuni metodi della classe Table vengono calcolate le dipendenze e restituite al thread chiamante;

5. il thread, dopo aver fornito il risultato al MainController, viene distrutto;
6. il MainController rimuove la barra di caricamento e visualizza sull'interfaccia grafica l'elenco delle dipendenze ricavate.

4.2 Interazione con la Tabella

Una delle funzionalità più importanti del programma è sicuramente l'interazione con le tabelle, ovvero con i *Formal Context*[1]; le tabelle, infatti, costituiscono il meccanismo che permette all'utente di fornire i dati al programma.

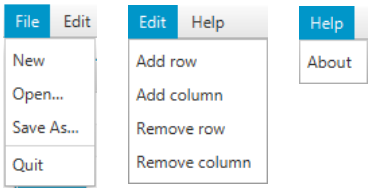


[1]**Formal Context:** definizione a pag. [3]

All'avvio del programma, l'utente viene accolto dalla schermata principale mostrata in figura. La finestra si presenta con il classico stile utilizzato dalla maggior parte dei programmi disponibili:

- In alto vi è la barra del menu, suddivisa in *File, Edit* ed *Help*.
- Al centro vi è una piccola tabella di default, composta da tre righe e tre colonne vuote per rappresentare oggetti ed attributi generici.
- Infine, nella parte bassa della finestra, sono presenti due bottoni *Create graph* e *Calculate*, i quali permettono di elaborare i dati dal punto di vista dei reticoli o delle dipendenze.

Barra del Menu: La barra del menu è suddivisa in tre sezioni, che ora saranno analizzate nel dettaglio.

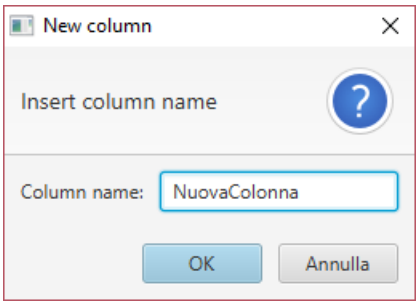


Nella sezione ***File***, appaiono le scelte:
New, per creare un nuova tabella (impostando come al solito quella di default);
Open..., per caricare una nuova tabella caricandola da un file *.xlsx*;
Save As..., per salvare la tabella su cui si sta lavorando in un file *.xlsx*;
Quit, per uscire dal programma.

Nel caso in cui si decida di aprire un nuovo progetto, oppure di chiudere il programma, il programma chiede conferma all'utente prima di completare l'operazione, in modo che questo non rischi di perdere i dati inseriti nella tabella.

Nella sezione ***Edit***, appaiono le scelte:
Add row, per aggiungere una nuova riga alla tabella;
Add column, per aggiungere una nuova colonna alla tabella;
Remove row, per rimuovere una riga dalla tabella;
Remove column, per rimuovere una colonna dalla tabella.

Cliccando su una qualsiasi tra le opzioni, viene aperta una piccola finestra come quella nell'immagine riportata, in cui può essere inserito il nome della riga o colonna da inserire/rimuovere.



Infine, nella sezione ***Help***, è possibile scegliere l'opzione *About*, la quale visualizzerà la versione del programma ed i relativi crediti.

Tabella: Ciò che rappresenta la sezione fondamentale della finestra principale è proprio la tabella. Questa, infatti, deve essere compilata correttamente prima di accedere alle finestre adibite alla visualizzazione dei risultati.

Le celle della tabella vengono compilate inserendo al loro interno una x, qualora un oggetto goda di una certa proprietà; tutte le celle che invece resteranno vuote saranno riconosciute come *assenza di una relazione* tra oggetto ed attributo.

Nel caso in cui si preferisca lavorare su un contesto in logica fuzzy, ciò che bisogna fare è semplicemente inserire all'interno della tabella dei valori nell'insieme [0,1]. L'algoritmo riconosce automaticamente il tipo di logica utilizzata in base ai valori che popolano la tabella ed agisce di conseguenza durante le fasi di creazione del grafo o di calcolo delle dipendenze tra attributi.

Il codice che implementa la tabella prevede la gestione delle eccezioni, mostrando quindi all'utente un messaggio di errore nell'eventualità in cui questo inserisca dei valori diversi da x, *spazio vuoto* o un *valore in [0,1]*. Inserendo delle x in un contesto fuzzy queste vengono considerate come *1*, mentre gli *spazi vuoti* vengono considerati come 0.

Create Graph: Cliccando sul bottone *Create graph* viene avviata l'elaborazione dei dati per generare il *Concept Lattice*, basandosi sulla compilazione della tabella. Si analizza nel dettaglio questa funzionalità nella sezione successiva.

Calculate: Cliccando sul bottone *Calculate* viene invece avviata l'elaborazione dei dati dal punto di vista delle dipendenze; anche in questo caso, si approfondisce la funzionalità in seguito.

4.2.1 Algoritmo

Le porzioni di codice che mirano all'implementazione del riconoscimento dei valori contenuti nella tabella sono mostrate qui di seguito.

```
1 public void addValue(Float value, int i){
2     // if key doesn't exist put 1 as value,
3     // otherwise sum 1 to the value linked to key.
4     mPossibleValues.merge(value, 1, Integer::sum);
5
6     values.get(i).add(value);
7 }
```

Questo metodo della classe *Table* viene chiamato più volte nel momento in cui si popola la tabella, ovvero quando si aggiungono i valori *float* al suo interno. In questa situazione viene popolata anche una *HashMap<Float, Integer>* che ha come *Key* i valori contenuti nella tabella e come *Value* i numero di volte in cui essi appaiono. Come descritto nel commento presente nel codice, la linea [4] aggiunge il nuovo valore come chiave della mappa se questo non era già presente gli assegna frequenza 1, altrimenti aggiunge 1 alla frequenza di quel valore.

L'altra situazione in cui è necessario aggiornare i valori della mappa (e quindi quelli considerati come valori presenti nella tabella) si verifica nel momento in cui l'utente modifica un valore presente nella tabella.

```
1  public void setCellValue(Float value, int i, int j) {
2      float oldValue = values.get(i).get(j);
3
4      if (oldValue == value)
5          return;
6
7      values.get(i).set(j, value);
8
9      mPossibleValues.merge(value, 1, Integer::sum);
10     if(mPossibleValues.get(oldValue) == 1)
11         mPossibleValues.remove(oldValue);
12     else
13         mPossibleValues.merge(oldValue, -1, Integer::sum);
14 }
```

L'esecuzione di questo metodo viene interrotta immediatamente se si tenta di sovrascrivere un valore della tabella con uno identico. In caso contrario, si procede con la sostituzione del valore in posizione (riga = i, colonna = j) e con il mantenimento della consistenza della mappa. Sicuramente, come nel metodo precedente, si aggiunge il nuovo valore come chiave con frequenza pari a 1 se questo non è già presente nella mappa, altrimenti si incrementa la sua frequenza. Inoltre è necessario considerare anche il vecchio valore, la cui frequenza si è ridotta di 1: se la frequenza del vecchio valore è pari ad 1, esso viene rimosso dalla mappa (dopo un decremento sarebbe arrivato a 0, quindi non è più presente nella tabella), altrimenti la sua frequenza viene decrementata.

Infine, il metodo *getter* per ottenere i valori possibili all'esterno della mappa converte l'insieme delle chiavi in un *HashSet*, in modo che sia compatibile con il resto delle strutture esterne alla classe *Table*.

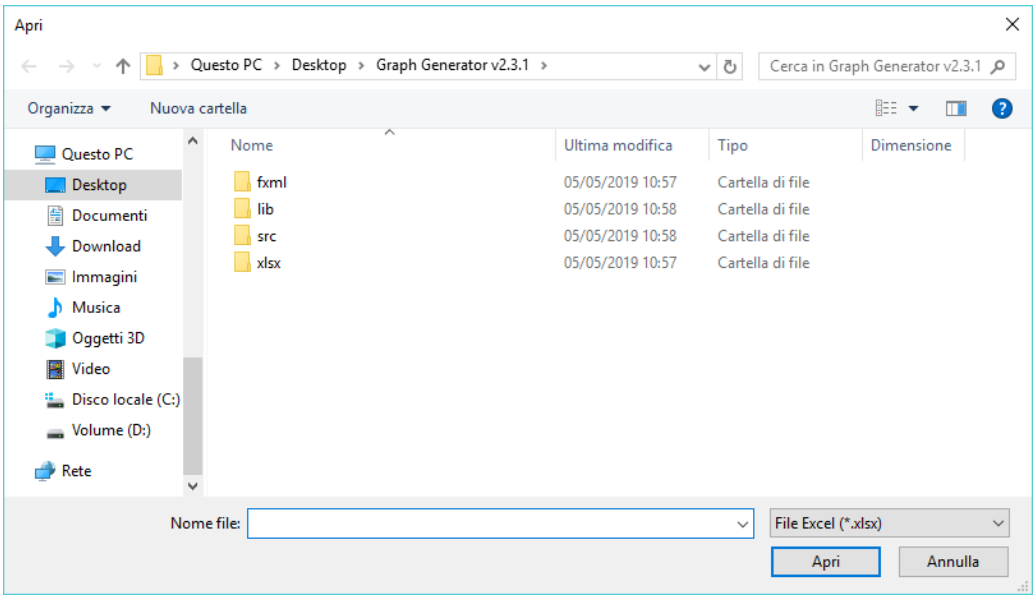
```
1  public HashSet<Float> getPossibleValues() {
2      return new HashSet<>(mPossibleValues.keySet());
3  }
```

4.3 Interazione con i File Excel

I file Excel *(.xlsx)* svolgono un ruolo fondamentale per quanto riguarda l'input e l'output delle tabelle dei dati. Il software, infatti, consente sia la lettura che la scrittura su file *.xlsx*, sfruttando la libreria *Apache POI* già citata all'inizio del documento.

4.3.1 Lettura

Accedendo all'apposita sezione nel menù ***File*** viene aperta la finestra di apertura dei file. Per garantire una maggiore usabilità da parte dell'utente, si è stabilito che l'esplorazione delle cartelle deve essere avviata come impostazione di default nella cartella in cui si trova il *.jar* del programma. Durante l'esplorazione delle cartelle, l'utente ha la possibilità di selezionare una cartella oppure un file *.xlsx*; eventuali file o collegamenti di formati differenti non vengono visualizzati.



Viene mostrata qui di seguito una sezione del metodo che implementa la lettura da file Excel, escludendo le parti meno importanti che altrimenti si troverebbero al posto dei *"..."*.

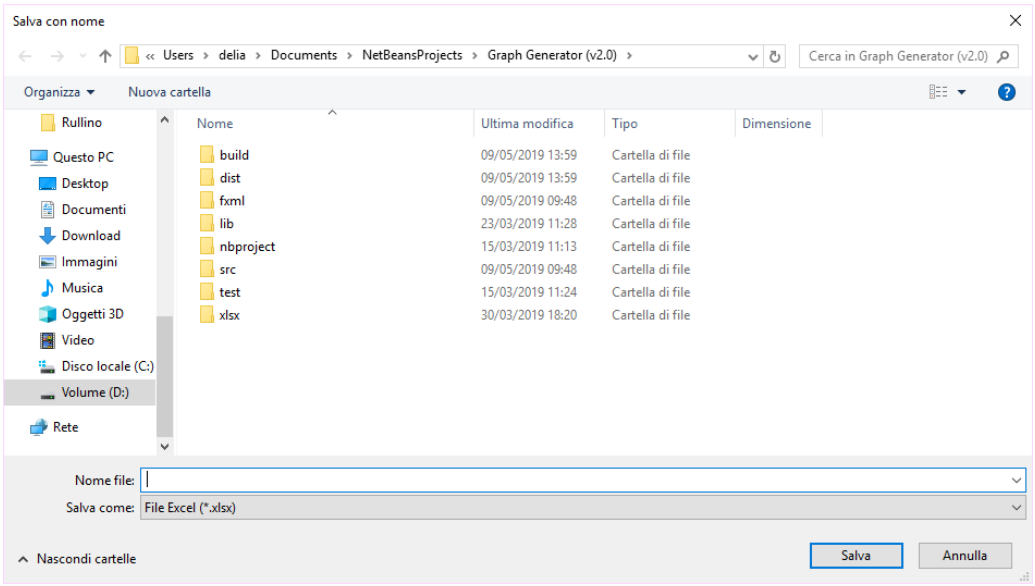
Il ciclo *while* esterno permette di percorrere la tabella Excel fintanto questa prevede una nuova riga non vuota. Per ogni riga, si considerano poi i campi relativi alle colonne attraverso il ciclo *for*:

- Per la prima riga, ovvero quella di intestazione, si procede con il riempimento della lista di attributi e del campo che rappresenta il nome della tabella.
- Per tutte le righe successive, invece, si compila la lista di oggetti (ricavati dalla prima colonna) e la lista dei valori. Se una cella contiene una $\times$, questa viene sostituita con il valore 1.0, mentre se è vuota con 0.0; in tutti gli altri casi il valore presente nella cella diventa un valore nella tabella.

```
...
boolean headerRow = true;
int rowsCount = 0;
while(rowIt.hasNext()) {
    ...
    for(int i = 0; i < columnsCount; ++i) {
        Cell cell = row.getCell(i);
        ...
        if(headerRow) {
            if(i == 0) // Nome Tabella
                table.setName(cellContent);
            else // Nuovo Attributo
                table.addAttribute(cellContent);
        }
        else {
            if(i == 0) // Nuovo Oggetto
                table.addItem(cellContent);
            else if(cellContent.equals("x"))
                table.addValue(1.0f, rowsCount-1);
            else if(cellContent.isEmpty())
                table.addValue(0.0f, rowsCount-1);
            else
                table.addValue(Float.parseFloat(cellContent),
                    rowsCount-1);
        }
    }
    ++rowsCount;
    headerRow = false;
}
...
```

4.3.2 Scrittura

Per quanto riguarda l'opzione di scrittura, selezionabile sempre attraverso il menu ***File,*** si può salvare la tabella nel suo stato corrente in un file Excel, con la solita estensione *.xlsx*. Come nel caso della lettura, l'utente inizia l'esplorazione delle cartelle per scegliere dove salvare il file all'interno della cartella contenente il *.jar*. Nella parte bassa della finestra è possibile selezionare il nome del file che, in qualsiasi caso, può avere come unica estensione accettata *.xlsx*.



Di seguito vi è una porzione del codice che implementa la scrittura su file *.xlsx*, anche questa volta privata delle parti poco significative.

```
...
for(int i = 0; i < table.getAttributesList().size(); ++i){
    cell = header.createCell(i+1);
    cell.setCellValue(table.getAttributesList().get(i));
    cell.setCellStyle(headerCellStyle);
    sheet.autoSizeColumn(i+1);
}

```

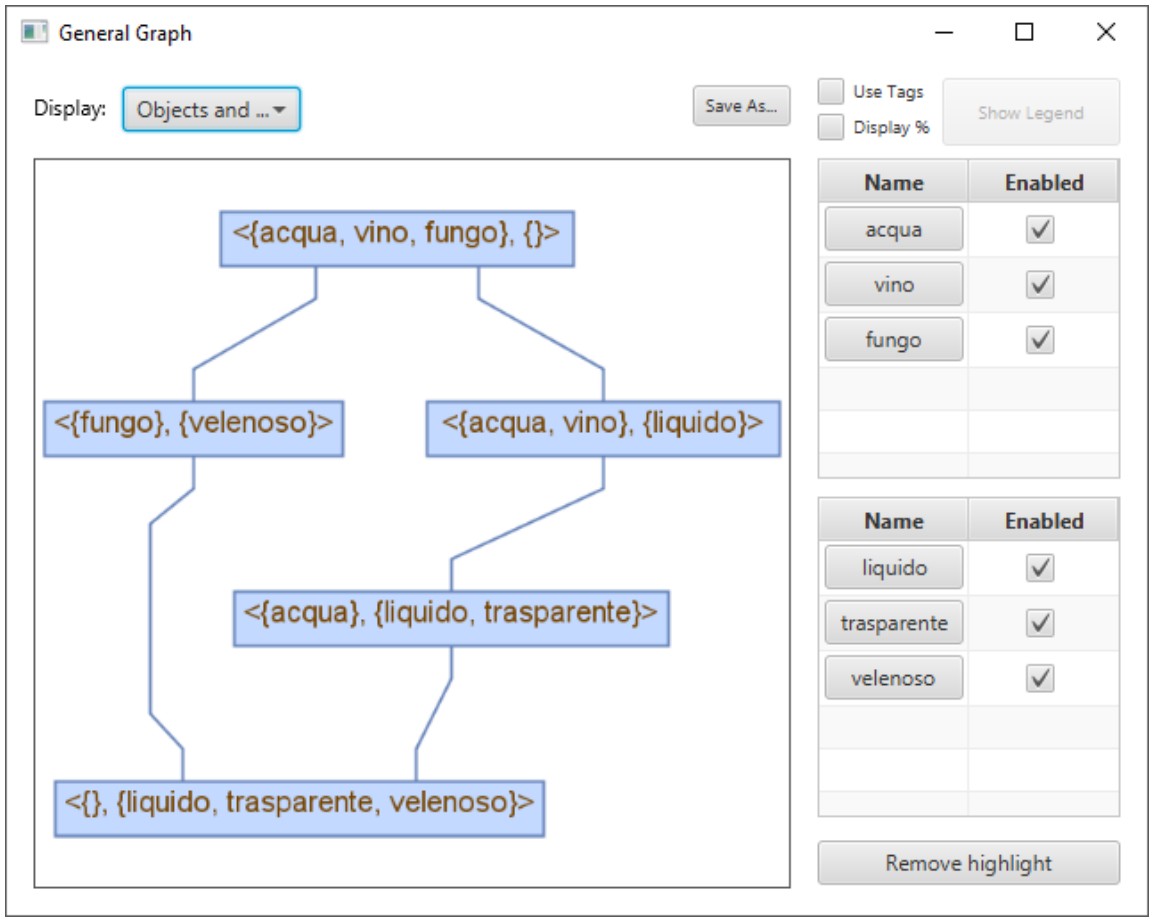
```
int r = 1;
for(int i = 0; i < table.getItemsList().size(); ++i) {
    Row row = sheet.createRow(r);

    cell = row.createCell(0);
    cell.setCellValue(table.getItemsList().get(r-1));
    cell.setCellStyle(headerCellStyle);
    sheet.autoSizeColumn(0);

    for(int j = 0; j < table.getValuesList().get(r-1).size(); ++j){
        cell = row.createCell(j+1);
        cell.setCellValue(table.getValuesList().get(r-1).get(j));
        cell.setCellStyle(dataCellStyle);
    }
    r++;
}
...
```

4.4 Generazione dei Reticoli dei Concetti

Il programma consente, partendo dalla tabella caricata nella schermata principale, di creare un concept lattice (ovvero un reticolo dei concetti), in modo da avere una rappresentazione grafica del contesto. Cliccando su *Create Graph* nella schermata principale, si aprirà una nuova finestra, che rimarrà aperta fintanto che:

- la tabella presente nella schermata principale non subisce una modifica;
- l'utente non clicca sulla [X] per chiudere esplicitamente la finestra;
- il programma non termina.

La schermata del reticolo dei concetti si presenta nel seguente modo:



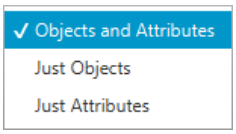
[2]

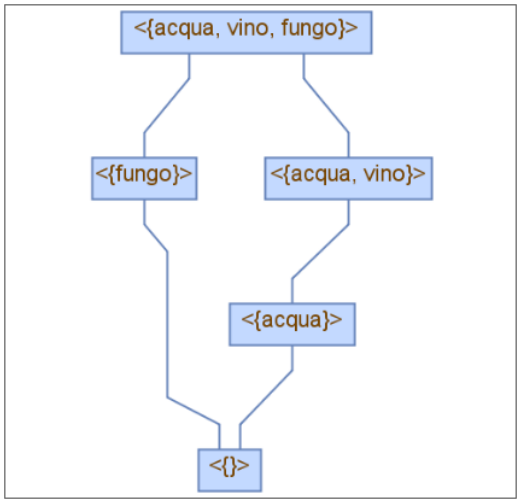
4.4.1 Funzionalità

Questa schermata presenta molteplici funzionalità, che valgono indipendentemente dal tipo di reticolo (logica Classica o Fuzzy che sia).

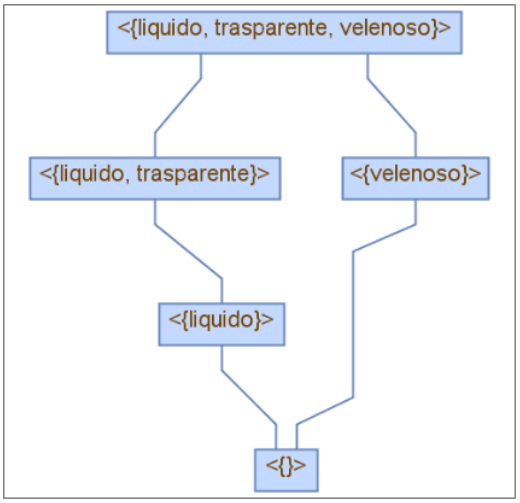
[2]Tabella a pag. [4]

Display: In alto a sinistra è presente un'etichetta e, di fianco ad essa, un menù a tendina che, una volta aperto presenta tre scelte: *Objects and Attributes, Just Objects e Just Attributes*. Qui è possibile selezionare cosa visualizzare all'interno dei nodi del reticolo, è consentito infatti visualizzare entrambi gli insiemi $\langle A_i, A_i^{\uparrow} \rangle$[3], solo l'insieme di oggetti A_i oppure solo l'insieme di attributi $A_i^{\uparrow}$.





Reticolo degli Oggetti



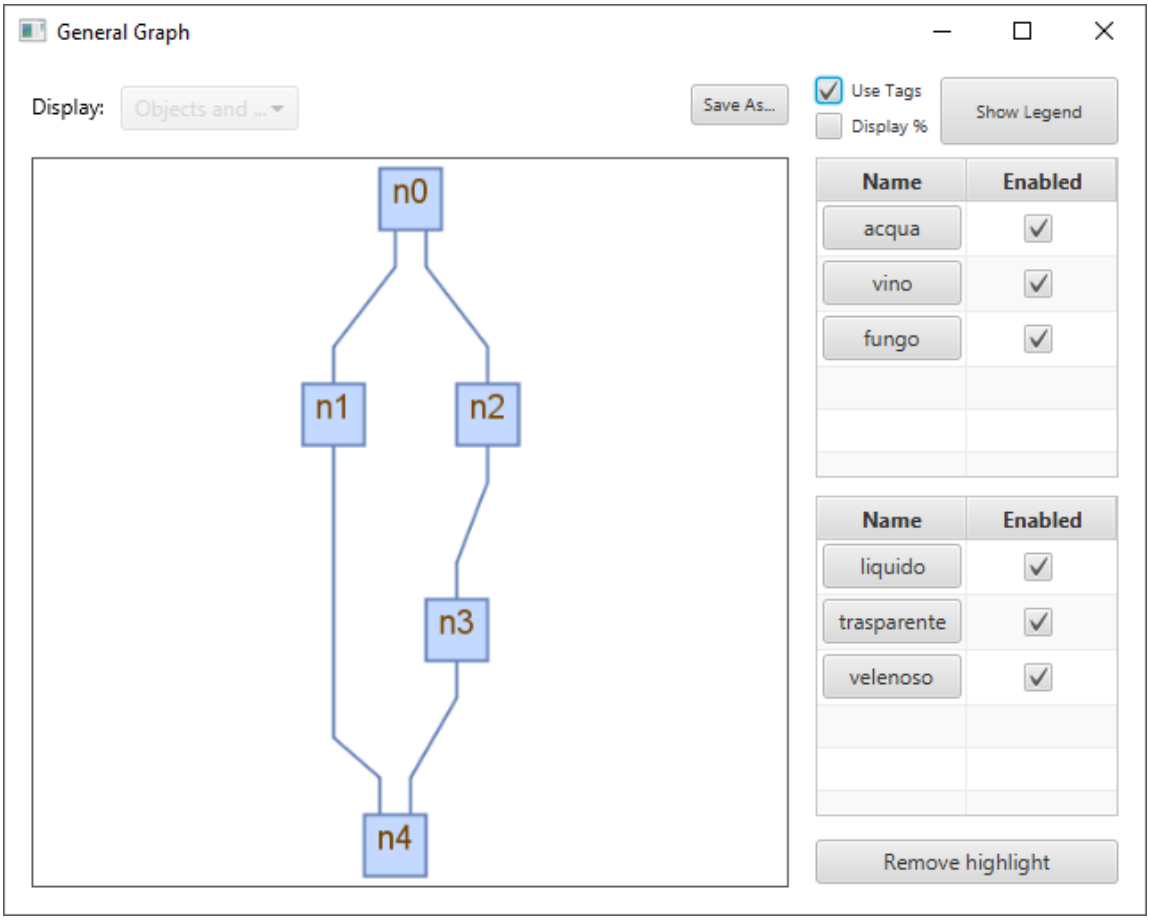
Reticolo degli Attributi

Salvataggio: Sempre nella parte alta della schermata, troviamo il pulsante *Save As...* che fornisce all'utente la possibilità di salvare l'immagine del reticolo attualmente visualizzato, in risoluzione originale e sotto il formato PNG. Questa funzionalità risulta molto utile se, per esempio, si vogliono confrontare più reticoli tra loro, in quanto il programma non permette l'apertura simultanea di più schermate del reticolo.

Tag: Può accadere che, al momento della creazione del reticolo, questo risulti eccessivamente grande e sia di conseguenza di difficile lettura. Per evitare ciò, è stata introdotta una funzionalità che permette di sostituire la normale visualizzazione dei nodi, con quella di tag che vadano a rendere il reticolo molto più leggibile.

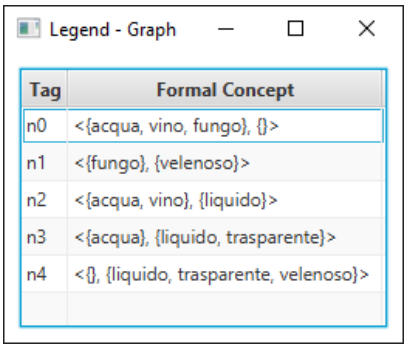
[3]**Formal Concept:** definizione a pag. [8]

È possibile attivare l'utilizzo dei tag spuntando la checkbox *Use Tags* in alto a destra. Al momento dell'attivazione, ad ogni nodo verrà assegnato un tag generato automaticamente ed il reticolo attualmente visualizzato verrà sostituito con l'equivalente versione etichettata.

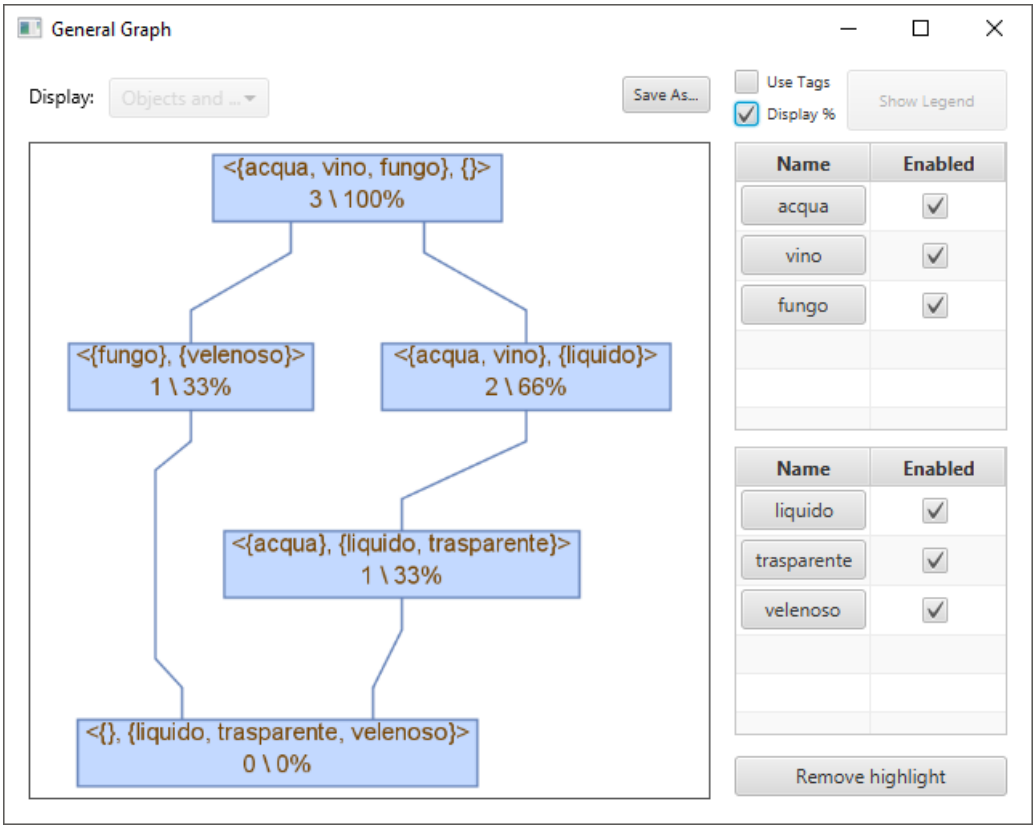


Ovviamente però, è necessaria a questo punto una legenda, che permetta all'utente di tradurre i tag assegnati ai vari nodi, nella coppia di insiemi $\langle A_i, A_i^{\uparrow} \rangle$. A tal proposito, al momento dell'applicazione dei tag, il programma genera una legenda in forma tabellare, che viene poi visualizzata in una apposita finestra (vedi figura).

La finestra è ridimensionabile a proprio piacimento e, se necessario, è possibile ridurla ad icona o chiuderla per poi riaprirla in un secondo momento. Una volta chiusa, l'utente può riaprire la legenda (o portarla in primo piano se era semplicemente ridotta ad icona) semplicemente cliccando sul pulsante *Show Legend* in alto a destra nella schermata del reticolo.

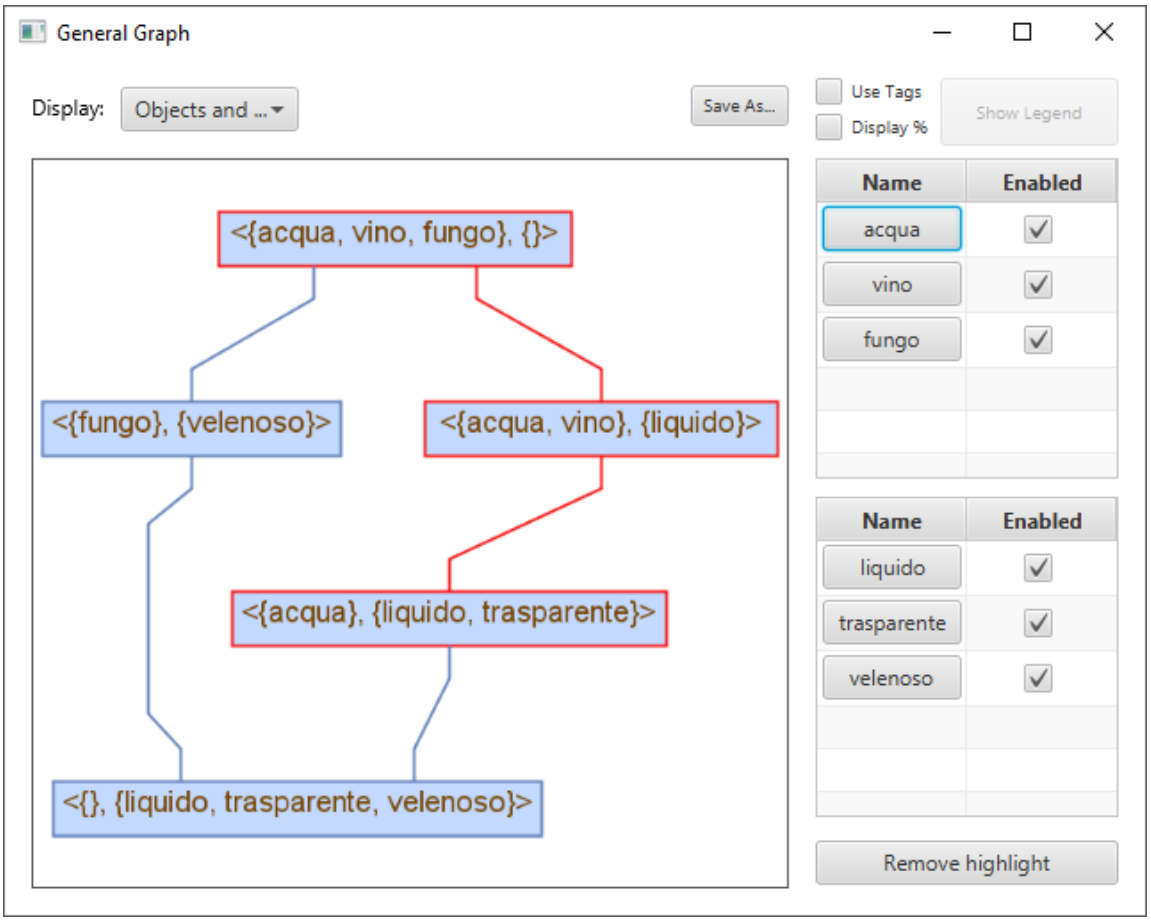


Percentuale oggetti: In alcuni casi risulta utile riuscire a capire, in modo rapido, quanti oggetti sono presenti nei vari nodi del reticolo. A tal proposito è possibile, spuntando la checkbox *Display %* visualizzare all'interno di ogni nodo quanti oggetti possiede, sia in numero che in percentuale (quanti oggetti ci sono rispetto al totale).



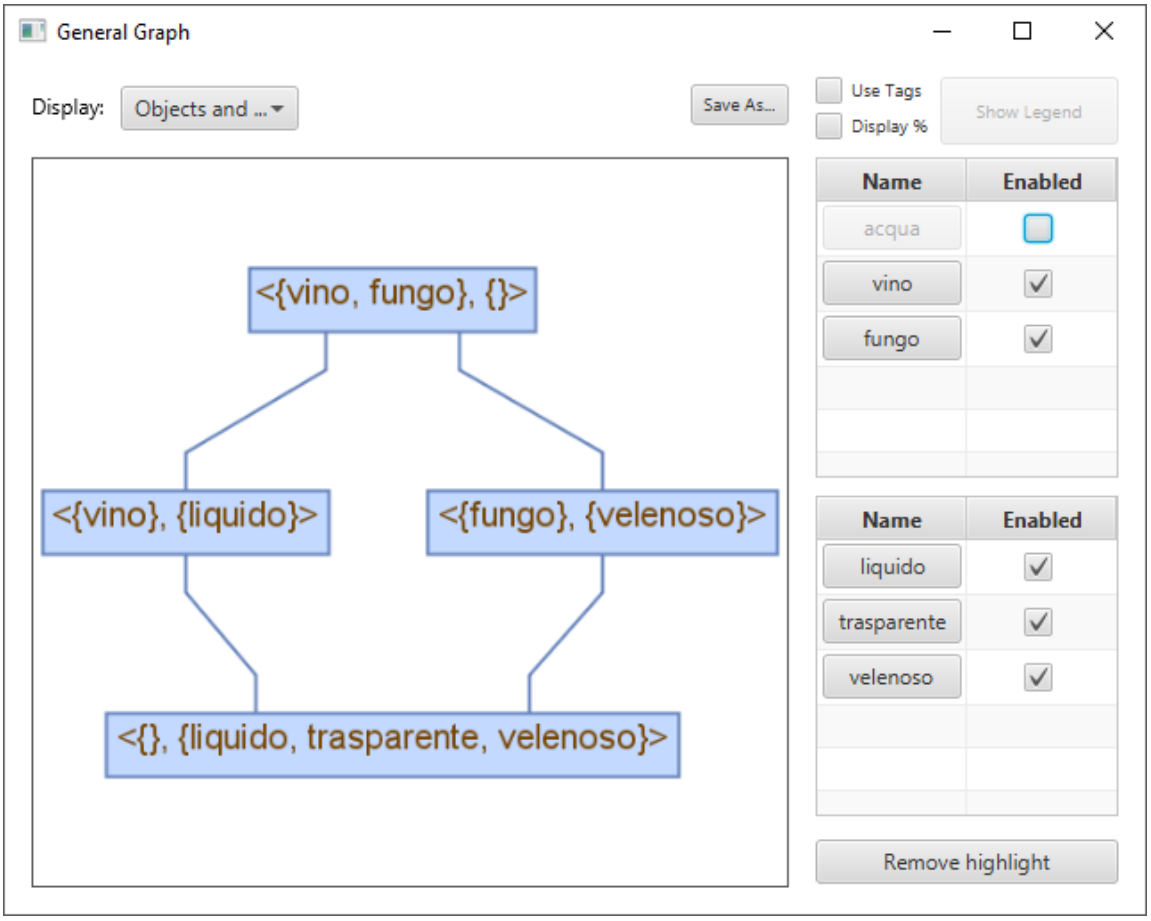
Menù degli oggetti: Per interagire con gli oggetti del reticolo è possibile utilizzare la tabella in alto a destra. Essa presenta due colonne: *Name* ed *Enabled*, che andremo ora ad analizzare.

Selezionare un oggetto: Sotto la colonna *Name* troviamo l'elenco di tutti gli oggetti presenti all'interno del nostro contesto. Facendo click su uno di questi oggetti è possibile evidenziare nel reticolo visualizzato tutti i nodi che contengono (anche solo in parte per la logica fuzzy) quel determinato oggetto. Nell'immagine di seguito è stato selezionato l'oggetto *acqua* che, come si può notare è presente solo in tre nodi del reticolo:



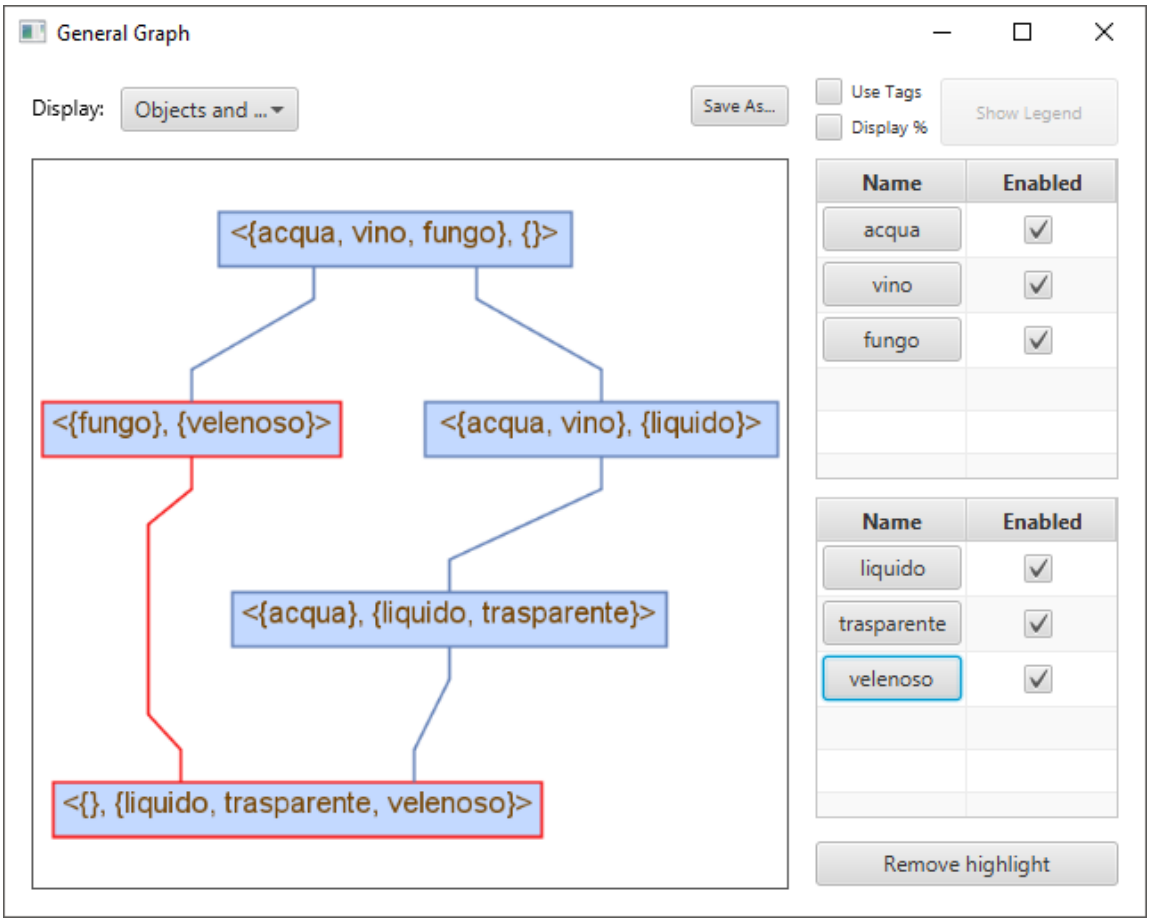
Se si desidera rimuovere l'attuale selezione, è sufficiente premere il pulsante *Remove highlight* presente in basso a destra.

Rimuovere un oggetto: Grazie alla colonna *Enabled* è possibile rimuovere temporaneamente uno o più oggetti (ma non tutti) dal contesto, rigenerando il reticolo senza di essi. Per farlo è sufficiente rimuovere la spunta dell'oggetto che si vuole rimuovere ed il reticolo verrà aggiornato automaticamente. Nella figura di seguito è possibile osservare il reticolo dopo la rimozione dell'oggetto *acqua*:



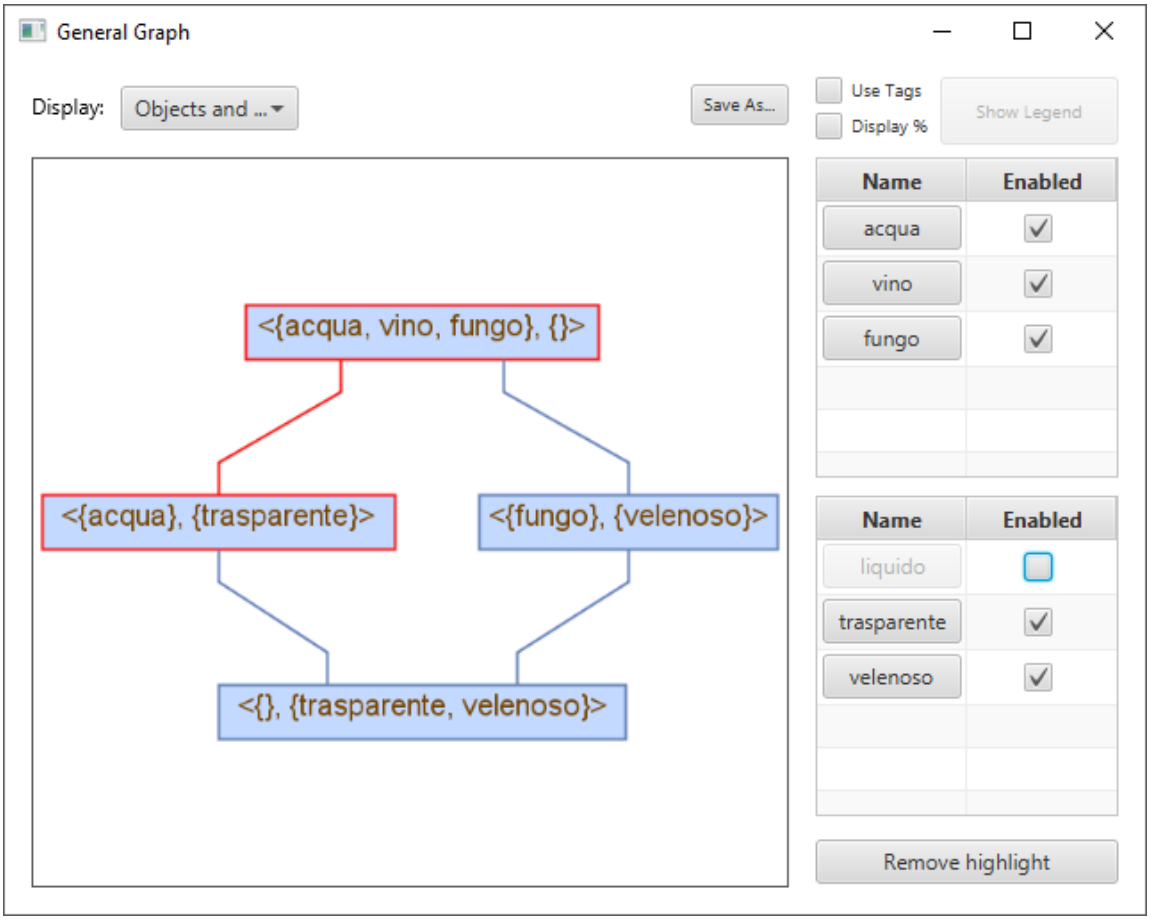
Menù degli attributi: Per interagire con gli attributi del reticolo è possibile utilizzare la tabella in basso a destra. Essa presenta due colonne: *Name* ed *Enabled*, che andremo ora ad analizzare.

Selezionare un attributo: Come per gli oggetti, sotto la colonna *Name* troviamo l'elenco di tutti gli attributi presenti all'interno del nostro contesto. Facendo click su uno di questi attributi è possibile evidenziare nel reticolo visualizzato tutti i nodi che contengono (anche solo in parte per la logica fuzzy) quel determinato attributo. Nell'immagine di seguito è stato selezionato l'attributo *velenoso* che, come si può notare è presente solo in due nodi del reticolo:



Se si desidera rimuovere l'attuale selezione, è sufficiente premere il pulsante *Remove highlight* presente in basso a destra.

Rimuovere un attributo: Grazie alla colonna *Enabled* è possibile rimuovere temporaneamente uno o più attributi (ma non tutti) dal contesto, rigenerando il reticolo senza di essi. Per farlo è sufficiente rimuovere la spunta all'attributo che si vuole rimuovere ed il reticolo verrà aggiornato automaticamente. Nella figura di seguito è possibile osservare il reticolo dopo la rimozione dell'attributo *liquido*:

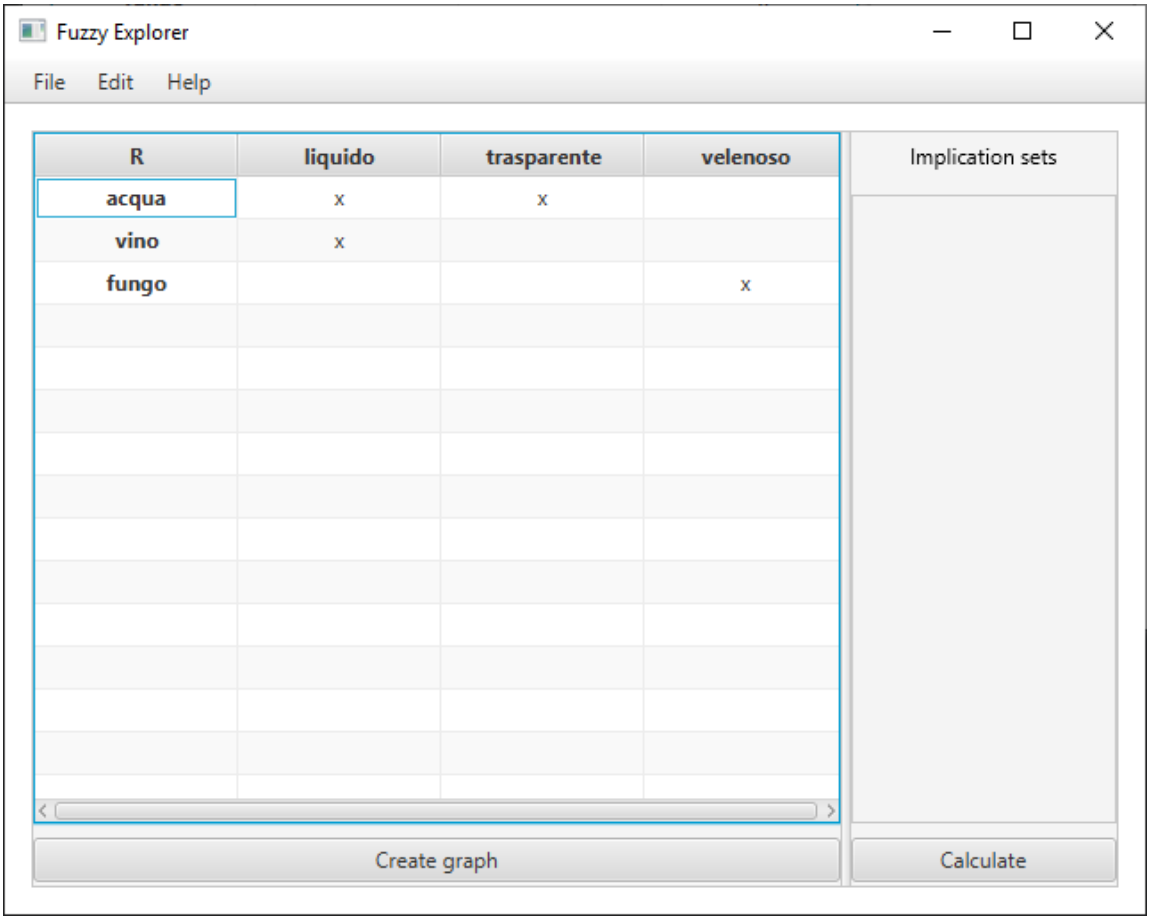


4.5 Calcolo delle Dipendenze Funzionali

Sempre partendo da una tabella caricata nella schermata principale, il programma permette, nel caso della logica classica, di calcolare la *base di Duquenne-Guigues*[4], mentre per la logica fuzzy permette all'utente di ottenere il grado di verità di un'implicazione da lui inserita.

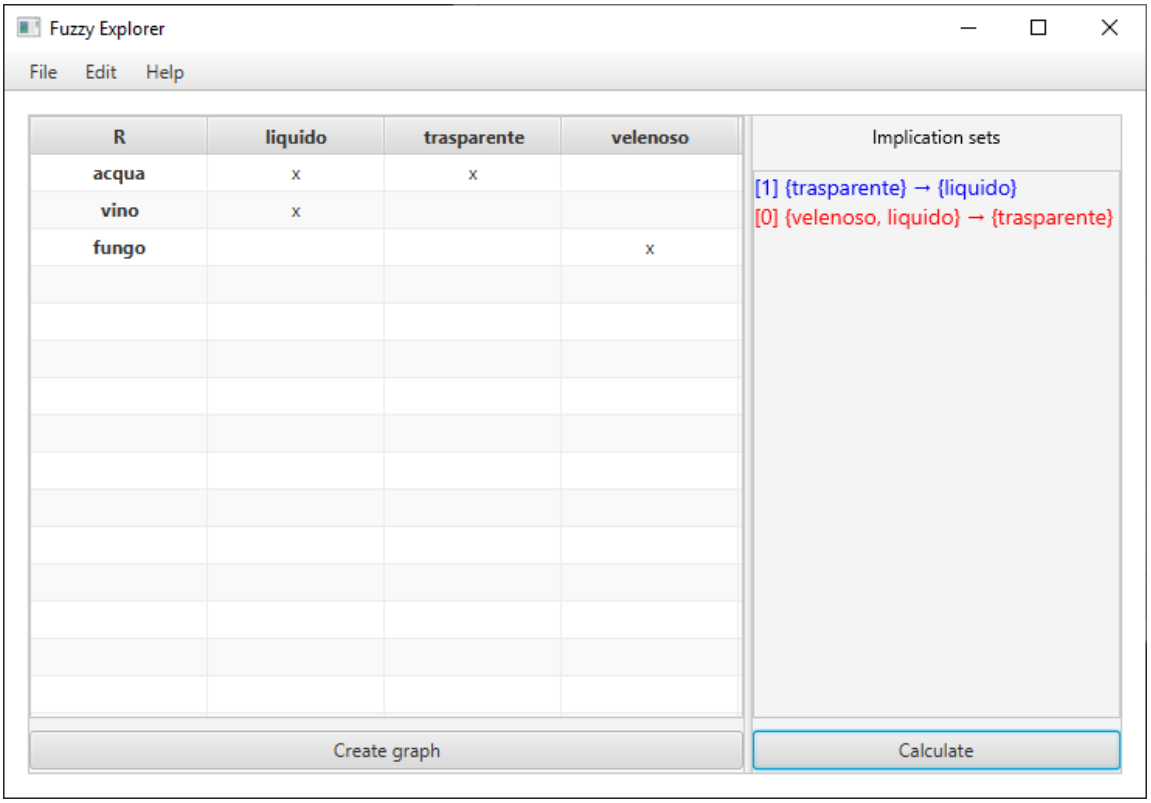
4.5.1 Logica Classica

Una volta caricata una tabella in logica classica, la schermata principale si presenterà come di seguito.



[4]**Base di Duquenne-Guigues:** definizione pag. |6|

A questo punto per calcolare la base di Duquenne-Guigues della tabella caricata è sufficiente fare click su *Calculate*, ottenendo, in questo caso, il seguente risultato:



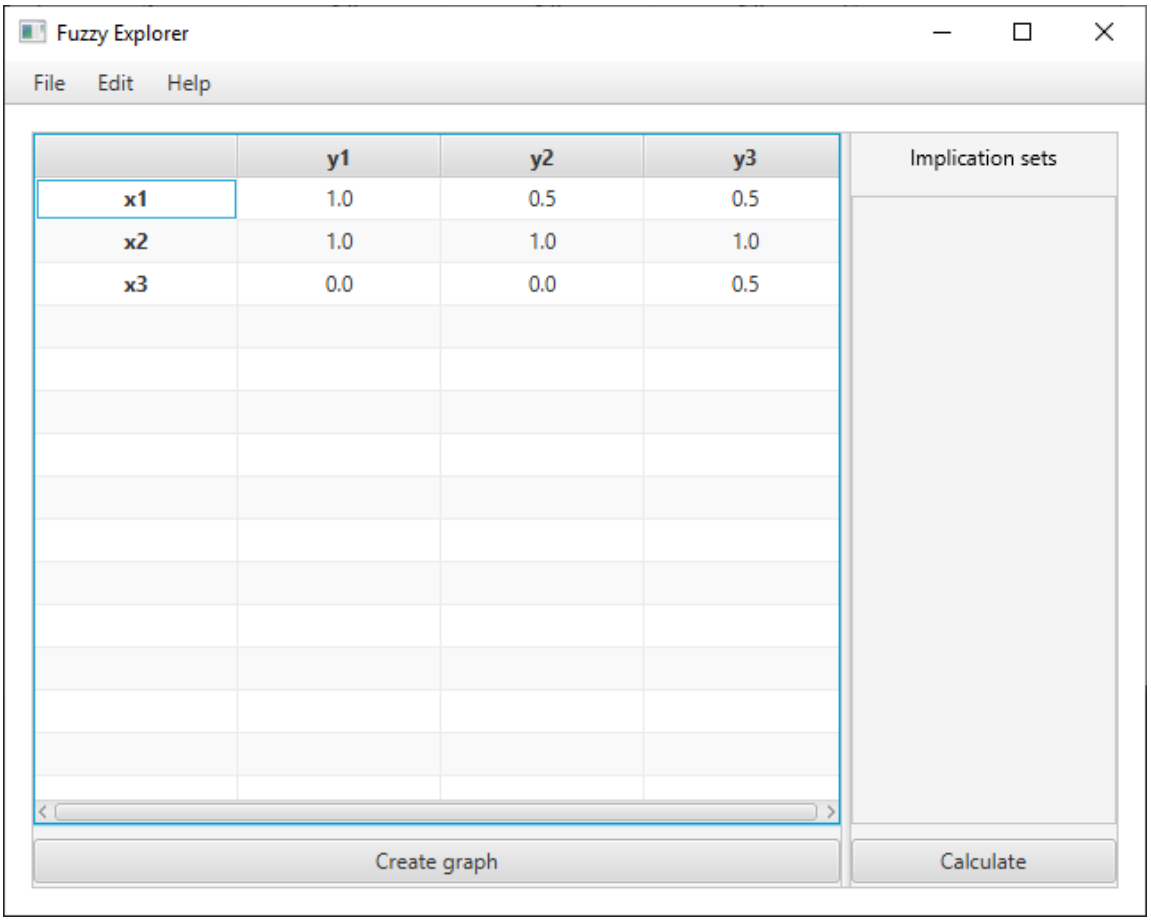
Come si può osservare dall'immagine, le implicazioni ottenute in questo caso sono due: $\{trasparente\} \rightarrow \{liquido\}$ e $\{liquido, velenoso\} \rightarrow \{trasparente\}$, che come abbiamo visto nell'esempio a pagina |7|, è esattamente la base di Duquenne-Guigues della tabella caricata.

Le implicazioni che compaiono vengono rappresentate secondo una particolare notazione che fornisce ulteriori informazioni riguardo alle stesse. Per prima cosa possiamo notare che sulla sinistra di ogni implicazione è presente, tra parentesi quadre, un numero intero al fine di indicare quante istanze di quella implicazione sono presenti all'interno del nostro contesto. Alcune implicazioni infatti, risultano nella base di Duquenne-Guigues semplicemente a causa della mancanza di un controesempio che le confuti. Queste implicazioni particolari, che non hanno neanche un'istanza

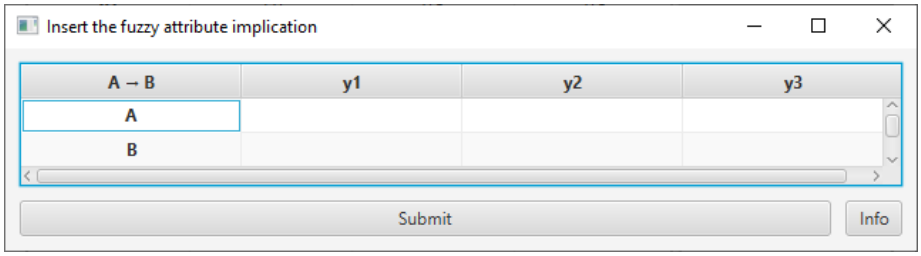
nel contesto, sono evidenziate attraverso l'uso del colore rosso che permette di distinguerle immediatamente dalle altre implicazioni. In questo caso è presente una sola implicazione di questo tipo, ovvero $\{liquido, velenoso\} \rightarrow \{trasparente\}$. È possibile rimuovere questa implicazione inserendo un controesempio, che in questo caso specifico consisterebbe in un oggetto velenoso, liquido ma non trasparente.

4.5.2 Logica Fuzzy

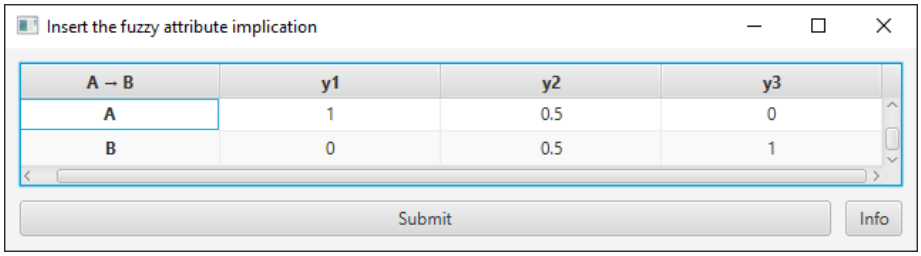
Come accennato in precedenza, nel caso in cui venga caricata una tabella contenente della logica fuzzy il programma permette di ottenere il grado di verità di una singola implicazione, inserita dall'utente. Consideriamo ora la seguente tabella:



Facendo click su *Calculate* viene richiesto all'utente di inserire i due insiemi di attributi A e B tra i quali calcolare il valore dell'implicazione $A \to B$. La schermata è la seguente:

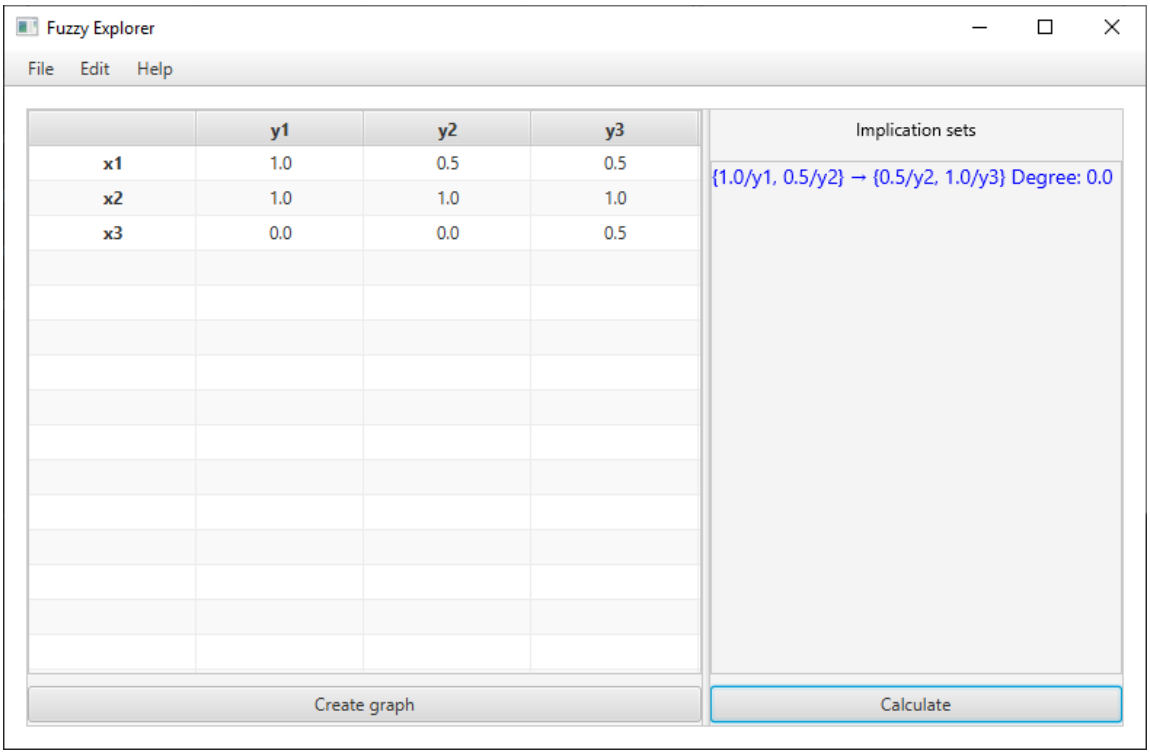


L'utente inserirà quindi il grado di verità degli attributi sui quali calcolare l'implicazione. Supponendo di dover calcolare ad esempio il grado di verità dell'implicazione $\{y1, 0.5/y2\} \to \{0.5/y2, y3\}$, bisognerà riempire la tabella nel seguente modo:



Nota: Inserire lo *0* è opzionale, infatti lasciando una cella vuota questa verrà trattata esattamente come se al suo interno ci fosse il valore *0*.

Una volta premuto *Submit* avremo quindi il seguente risultato:



Il grado di verità dell'implicazione è quindi (in questo caso) "0.0", come si può osservare dall'immagine. A questo punto è possibile calcolare altre implicazioni sulla tabella, seguendo lo stesso procedimento, che si andranno poi ad aggiungere alla lista delle implicazioni calcolate (colonna sulla destra).

4.5.3 Algoritmi

Vediamo ora le parti di codice più rilevanti riguardo la generazione delle dipendenze funzionali, in modo da comprendere gli algoritmi principali che sono dietro il funzionamento del programma.

In base ai valori inseriti all'interno della tabella nella schermata principale del programma, al momento della pressione del pulsante *Calculate* possono essere eseguiti due algoritmi differenti, rispettivamente della logica classica e logica fuzzy.

Logica Classica

Nel caso in cui nella tabella siano presenti soltanto i valori 0 e 1, il programma lo riconoscerà e comincerà a calcolare la base di Duquenne-Guigues utilizzando la tabella inserita. Per farlo viene chiamata la funzione *getDGbase()* presente nella classe *Table* il che restituisce una lista di coppie $\langle colore, stringa \rangle$, contenenti ciascuna una implicazione da visualizzare all'utente, con il rispettivo colore.

```
...
public ArrayList<SimpleEntry<Color, String>> getDGbase() {
    ...
}
...
```

All'interno di questa funzione, viene per prima cosa calcolato l'insieme delle parti relativo all'insieme degli attributi della tabella; esso viene poi ordinato in base alla dimensione degli insiemi, dal più piccolo al più grande.

```
Set<String> attr = new HashSet<>(attributes);
ArrayList<PSITableRow> pseudoIntents = new ArrayList<>();

// Gets the pseudo-intents
List<Set<String>> orderedSubsets;
orderedSubsets = new ArrayList<>(Sets.powerSet(attr));

Collections.sort(orderedSubsets, new Comparator<Set<String>>() {
    @Override
    public int compare(Set<String> o1, Set<String> o2) {
        return Integer.valueOf(o1.size()).compareTo(o2.size());
    }
});
...
```

Oltre a questo, come si piò notare nel codice soprastante, viene dichiarata anche una lista chiamata *pseudoIntents*. Essa conterrà la lista di tutti gli pseudo-intent che man mano verranno individuati dall'algoritmo, nel corso della sua esecuzione. Inoltre fa riferimento alla classe *PSITableRow*, definita all'interno della stessa classe *Table*.

Dichiarazione della classe *PSITableRow* all'interno della classe *Table*:

```
1  public class Table {
2      // A pseudo-intent table row
3      private class PSITableRow {
4          Set<String> start_attr; // P
5          Set<String> items;      // P'
6          Set<String> attr;       // P''
7
8          @Override
9          public String toString() {
10             return start_attr.toString();
11         }
12     }
13
14     ...
15 }
```

Ogni istanza di questa classe rappresenta una riga della tabella utilizzata per calcolare gli pseudo-intent, vista a pagina |7|. Nello specifico:

- *start_ attr* è l'insieme di attributi di partenza $\boldsymbol{P}$
- *items* è l'insieme di oggetti $\boldsymbol{P}^{\downarrow}$
- *attr* è l'insieme di attributi $\boldsymbol{P}^{\downarrow\uparrow}$

Una volta calcolato l'insieme delle parti degli attributi, per ogni sottoinsieme bisogna controllare se questo è uno pseudo-intent[5] o meno e, nel caso in cui lo sia, aggiungerlo alla lista.

[5]Definizione pag. |6|

Il codice è quindi il seguente:

```
1       ...
2
3       orderedSubsets.forEach(subset -> {
4           PSITableRow tmp = new PSITableRow();
5
6           tmp.start_attr = subset;                          // P
7           tmp.items = getSatisfiedItems(tmp.start_attr);    // P'
8           tmp.attr = getSatisfiedAttributes(tmp.items);     // P''
9
10          boolean isPseudoIntent = true;
11
12          if (tmp.start_attr.equals(tmp.attr))
13              isPseudoIntent = false;
14          else {
15              for (PSITableRow r: pseudoIntents) {
16                  if (isSubset(r.start_attr, tmp.start_attr)) {
17                      if (!isSubset(r.attr, tmp.start_attr)) {
18                          isPseudoIntent = false;
19                          break;
20                      }
21                  }
22              }
23          }
24
25          if (isPseudoIntent)
26              pseudoIntents.add(tmp);
27      });
28
29      ...
```

Una volta ottenuta la lista degli pseudo-intent, per restituire come risultato una lista di coppie $\langle colore, stringa \rangle$ come visto in precedenza, è necessario scorrerli e per ciascuno scegliere il colore e costruire la stringa contenente la dipendenza funzionale (seguendo la formula vista a pagina |8|).

```
    ...

    ArrayList<SimpleEntry<Color,String>> result = new ArrayList<>();

    for (PSITableRow psi: pseudoIntents) {
        int inst = getPSIColor(psi);

        String first = psi.toString().replaceAll("\\[", "{")
            .replaceAll("\\]", "}");
        String last = Sets.difference(psi.attr, psi.start_attr)
            .toString().replaceAll("\\[","{").replaceAll("\\]","}");

        result.add(new SimpleEntry<>(
        inst == 0 ? Color.RED : Color.BLUE, "[" + inst + "] "
            + first + " → " + last
        ));
    }

    return result;
}
```

Per ottenere l'insieme di oggetti $\boldsymbol{P}^{\downarrow}$ (*items*) e l'insieme di attributi $\boldsymbol{P}^{\downarrow\uparrow}$ (*attr*) vengono utilizzate rispettivamente le funzioni:

- *getSatisfiedItems(attr)*
- *getSatisfiedAttributes(items)*

La funzione *getSatisfiedItems* calcola gli oggetti che soddisfano tutti gli attributi passati come parametro (in questo caso $\boldsymbol{P}$), ovvero l'insieme $\boldsymbol{P}^{\downarrow}$.

Il codice della funzione è il seguente:

```
private Set<String> getSatisfiedItems(Set<String> columns) {
    Set<String> result = new HashSet<>();

    if (columns.isEmpty()) {
        result.addAll(items);
        return result;
    }

    ArrayList<Integer> indexes = new ArrayList<>();

```

```
11      columns.forEach(a -> indexes.add(attributes.indexOf(a)));
12
13      for (int i = 0; i < values.size(); ++i) {
14          boolean add = true;
15
16          for (Integer j: indexes) {
17              if (values.get(i).get(j) != 1.0f) {
18                  add = false;
19                  break;
20              }
21          }
22
23          if (add)
24              result.add(items.get(i));
25      }
26
27      return result;
28  }
```

La funzione *getSatisfiedAttributes* calcola invece gli attributi che soddisfano tutti gli oggetti passati come parametro (in questo caso $\boldsymbol{P}^{\downarrow}$), ovvero l'insieme $\boldsymbol{P}^{\downarrow\uparrow}$.
Il codice è il seguente:

```
1   private Set<String> getSatisfiedAttributes(Set<String> rows) {
2       Set<String> result = new HashSet<>();
3
4       result.addAll(attributes);
5
6       ArrayList<Integer> indexes = new ArrayList<>();
7       rows.forEach(r -> indexes.add(items.indexOf(r)));
8
9
10      for (Integer i: indexes) {
11          Set<String> currentAttr = new HashSet<>();
12
13          for (int j = 0; j < values.get(0).size(); ++j)
14              if (values.get(i).get(j) == 1.0f)
15                  currentAttr.add(attributes.get(j));
16
17          result = Sets.intersection(result, currentAttr);
18      }
19
20      return result;
21  }
```

Terminata l'esecuzione del metodo *getDGbase()* la schermata principale del programma si preoccuperà di visualizzare la lista di dipendenze generate (ovvero la base di Duquenne-Guigues).

È possibile trovare ulteriori informazioni riguardanti le dipendenze funzionali nella logica classica a pag. |5|.

Logica Fuzzy

Nel caso in cui nella tabella siano presenti più valori, e non semplicemente 0 e 1, il programma riconoscerà di essere in un contesto fuzzy e chiederà quindi all'utente di inserire l'implicazione della quale calcolare il grado di verità. Una volta inserita, verrà richiamata la funzione *subsethoodDegree*, che si occuperà di calcolare e restituire il grado di verità dell'implicazione. Questa funzione richiede due parametri, che sono i valori (inseriti dall'utente) degli attributi degli insiemi A e B di cui fare l'implicazione. Il codice della funzione è il seguente:

```
1  ...
2
3  public Float subsethoodDegree(ArrayList<Float> setA,
4                                ArrayList<Float> setB) {
5      Float result = 1.0f;
6
7      setA = new ArrayList<>(GraphFunctions.computeA(setA, values));
8
9      Float min;
10     for (int i = 0; i < setA.size(); ++i) {
11         Float a = setA.get(i);
12         Float b = setB.get(i);
13
14         min = GraphFunctions.imply(a, b);
15
16         if (min < result)
17             result = min;
18     }
19
20     return result;
21 }
22
23 ...
```

Per calcolare il grado di verità utilizza quanto visto a pagina e |17|, andando a calcolare quindi per prima cosa $\boldsymbol{A}^{\downarrow\uparrow}$ mediante la funzione *computeA*, e andando poi ad ottenere il valore minore tra tutti i risultati delle implicazioni tra gli attributi di $\boldsymbol{A}^{\downarrow\uparrow}$ e B. La funzione *computeA* richiede due parametri, ovvero rispettivamente i valori (float) degli attributi dell'insieme a sinistra dell'implicazione e i valori della tabella.

```
1  ...
2  public static List<Float> computeA(ArrayList<Float> attr, ArrayList<
       ArrayList<Float>> table) {
3      ...
4  }
5  ...
```

Al suo interno viene calcolato per prima cosa l'insieme $\boldsymbol{A}^{\downarrow}$, andando a scorrere tutti gli oggetti (righe della tabella) e per ciascuno aggiungere all'interno della lista *itemsValues* il valore con cui questo soddisfa l'insieme di attributi A:

```
1  ...
2
3  for(int r = 0; r < table.size(); ++r) {
4      float min = 1f;
5
6      for(int c = 0; c < attr.size(); ++c) {
7          Float res = imply(attr.get(c), table.get(r).get(c));
8          min = min(res, min);
9      }
10
11     itemsValues.add(min);
12 }
13
14 ...
```

Una volta ottenuto l'insieme $\boldsymbol{A}^{\downarrow}$, ovvero itemsValues, la funzione genera l'insieme $\boldsymbol{A}^{\downarrow\uparrow}$ scorrendo tutti gli attributi (colonne della tabella) e andando ad aggiungere alla lista *attrValues* il valore con cui ciascuno è soddisfatto dall'insieme di oggetti $\boldsymbol{A}^{\downarrow}$:

```
...

for(int c = 0; c < attr.size(); ++c) {
    float min = 1;

    for(int r = 0; r < itemsValues.size(); ++r) {
        Float res = imply(itemsValues.get(r), table.get(r).get(c));
        min = min(res, min);
    }

    attrValues.add(min);
}

return attrValues;
```

Dato che attrValues conterrà al suo interno l'insieme $\boldsymbol{A}^{\downarrow\uparrow}$, sarà sufficiente a questo punto restituirlo, terminando quindi l'esecuzione della funzione *computeA*.

5 Conclusioni

Concludiamo ora, riassumendo quanto visto all'interno di questo documento e facendo alcune considerazioni. Per prima cosa abbiamo analizzato la *Formal Concept Analysis* nella logica classica, definendo un *formal context* come una relazione binaria tra due insiemi X e Y, che può essere vista anche sotto forma tabellare. Ci siamo poi soffermati sulle *functional dependencies* nella logica classica definendo cos'è uno **pseudo-intent**, cosa rappresentano gli insiemi $\boldsymbol{A}^{\uparrow}$ e $\boldsymbol{B}^{\downarrow}$ e cosa si intende per base di Duquenne-Guigues. A tal proposito possiamo notare come queste dipendenze tra attributi permettano di ottenere ulteriori informazioni riguardanti il contesto, e le regole che i suoi dati seguono. È stata inoltre accennata una rappresentazione grafica del *formal context*, ovvero un **grafo diretto aciclico** detto *concept lattice*.

Dato che le tabelle di dati da analizzare non sono sempre espresse secondo la logica classica, sono state analizzate alcune metodologie per trasformare un contesto multivalore in una tabella contenente esclusivamente valori booleani. Questo paragrafo può risultare molto utile nel caso in cui un utente necessiti di adattare un contesto reale alla logica booleana.

Da logica classica siamo poi passati ad un altro tipo di logica, in cui i possibili valori non sono più vero o falso, ma qualsiasi valore dell'intervallo [0, 1]: la logica fuzzy. Differentemente da quanto visto per la logica classica, in quella fuzzy le implicazioni hanno un **grado di verità**, e possono essere quindi più o meno vere in base al contesto. Anche in questo caso, come per la logica classica, è possibile rappresentare il contesto tramite grafo.

Infine, abbiamo approfondito il programma *Fuzzy Explorer*, ovvero il software sviluppato durante tutto il periodo del tirocinio. Il programma riconosce automaticamente il tipo di logica (classica o fuzzy) tramite la tabella inserita e, basandosi sulle definizioni teoriche trattate, fornisce le seguenti funzionalità:

- **Logica Classica:** Permette la generazione della base di Duquenne-Guigues, visualizzando all'utente una lista di tutte le dipendenze che la compongono.

– **Logica Fuzzy:** Permette il calcolo del grado di verità di una dipendenza tra attributi, inserita dall'utente tramite l'apposita finestra.

Lo scopo di queste funzionalità è quello di evitare all'utente lunghi e tediosi calcoli manuali, permettendogli inoltre di modificare il contesto e ricalcolarne le dipendenze funzionali in modo rapido. È possibile notare che il tempo di esecuzione necessario per il calcolo delle dipendenze funzionali nella logica classica, dipende principalmente dal numero di attributi presenti nel contesto. Dato che in alcuni casi sono necessari alcuni secondi di attesa, abbiamo deciso di aggiungere una finestra di caricamento, da visualizzare all'utente durante l'esecuzione dei calcoli necessari.

5.1 Bibliografia

1. **Dmitry I. Ignatov**, Introduction to Formal Concept Analysis and Its Applications in Information Retrieval and Related Fields, National Research University Higher School of Economics (Moscow), 2017
2. **Bernard De Baets & Jan Outrata**, Computing the Lattice of All Fixpoints of a Fuzzy Closure Operator, 2010
3. **Radim Belohlavek & Jan Kenocny**, A calculus for containment of fuzzy attributes, 2017
4. **Davide Ciucci & Didier Dubois & Henri Prade**, The Structure of Oppositions in Rough Set Theory and Formal Concept Analysis - Toward a New Bridge between the Two Settings, 2015
5. **Radim Belohlavek & V. Vychodil**, Attribute dependencies for data with grades, 2016
6. **Concept Explorer**: `http://conexp.sourceforge.net/`
7. **JGraph**: `https://jgrapht.org/`