ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

AUTO-ORGANIZZAZIONE DELLE INFORMAZIONI IN UNA RETE TRAMITE ALGORITMO NATURE-INSPIRED

Elaborata nel corso di: Sistemi MultiAgente LM

Progetto svolto da:
DOMENICONI GIACOMO, FABBRI MANUEL

ANNO ACCADEMICO 2010–2011

Indice

Introduzione

Questo progetto si svolge nell'ambito dell'auto-organizzazione della conoscenza in una rete formata da nodi mobili (intesa come possibilità di modificare la struttura della rete). L'obiettivo che ci si pone è di aggregare i dati a seconda di un concetto di tipo, in modo da crearne un'organizzazione che diminuisce l'entropia del sistema e facilitarne dunque la ricerca.

Si vuole perciò realizzare un middleware, nel quale i nodi, comandati opportunamente da agenti software, interagiscono fra di loro e, seguendo un algoritmo di ispirazione naturale, si scambiano le informazioni in loro possesso, per realizzare globalmente il comportamento voluto nel sistema.

Nel primo capitolo viene descritto il problema, presentata la sua analisi e mostrati alcuni riferimenti teorici al concetto di agente e auto-organizzazione in un sistema multi-agente.

Nel secondo capitolo viene presentato il progetto della soluzione con una descrizione delle scelte progettuali riguardo i singoli agenti e i risultati che complessivamente producono sul funzionamento del sistema.

Nel terzo capitolo viene fornita la spiegazione dell'implementazione relativa al codice Java scritto per realizzare le varie parti del sistema e come mandarlo in esecuzione.

Nel quarto capitolo vengono valutati i risultati relativi alle varie funzionalità, con considerazioni su possibili sviluppi futuri..

Capitolo 1

Analisi del problema

In questo capitolo si presenta il problema, si elencano i requisiti, per poi procedere alla sua analisi, nell'ottica dei *sistemi multi-agente*, fornendo infine un esempio di scenario applicativo.

1.1 Descrizione del problema

Il problema introdotto si colloca nell'ampio spazio dell'*organizzazione della conoscenza*. In particolare, il contesto riguarda una rete di nodi *distribuiti* e *mobili*, nel quale ogni nodo può comunicare direttamente con il proprio vicino, con comunicazioni punto a punto. L'obiettivo del sistema è ordinare le informazioni dimodoché tutte quelle di un certo tipo siano aggregate in un nodo o in nodi vicini, avendo inoltre la possibilità di indirizzare una certa tipologia in un determinato nodo, facilitandone così la ricerca e l'accesso. Il sistema, inoltre, si pone l'obiettivo di adattarsi dinamicamente ai cambiamenti delle informazioni presenti, senza interventi dell'utente: si parla quindi di *auto-organizzazione*.

1.1.1 Funzionalità

Le funzionalità che si vogliono garantire sono:

- *Organizzazione* delle informazioni nella rete, ovvero aggregarle, secondo un concetto di tipo, in un nodo o in un cluster di nodi vicini.
- *Ricerca*: ottenere (meta-)informazioni sulla presenza e sulla posizione di un certo tipo di informazioni nella rete.
- *Spostamento* delle informazioni volute da un nodo all'altro, vale a dire attirarne un certo tipo su di un nodo.

1.2 Requisiti

I requisiti verso cui ci si trova di fronte sono i seguenti:

- *Situatedness*: è necessario tenere presente la situazione in cui lavora il sistema, che non è predicibile; l'ambiente diventa dunque parte del sistema. I dati subiscono spostamenti a seconda della situazione rappresentata dall'ambiente.

- *Openess*: non si conosce a priori il numero di componenti che faranno parte del sistema; esso è soggetto a cambiamenti in misura e in struttura. Il sistema è formato da dispositivi con un certo grado di mobilità e deve garantire il suo funzionamento anche in assenza di alcuni componenti.

- *Località nel controllo*, l'autonomia è diffusa fra i componenti, i quali sono entità autonome che possiedono un proprio flusso di controllo.

- *Località nelle interazioni*, le quali avvengono su base locale. I dispositivi sono consapevoli del contesto su base locale, fra nodi vicini.

1.3 Analisi del problema

Il problema presenta tutti i requisiti che, per essere risolti, richiedono l'utilizzo di un sistema multi-agente (d'ora in avanti *MAS*). Prima di tutto ci si trova in una rete, con tutti i problemi che ciò comporta, ovvero l'impossibilità di avere un controllo e un tempo *globale*. Ogni nodo è disaccoppiato temporalmente (non tutti i nodi possono essere contemporaneamente presenti) e spazialmente (mobilità, distanza spaziale) da tutti gli altri. Le azioni di un nodo, sebbene siano ristrette in un set di azioni ammissibili (collegarsi/lasciare la rete, inserire informazioni), non sono predicibili, né nella quantità, né nella tipologia, né tantomeno riguardo l'istante temporale in cui avverranno.

Ogni nodo dovrà dunque avere un *flusso di controllo* proprio che gli permetterà di svolgere tutte queste azioni dove e quando vuole. Ciò che il sistema, che andremo a progettare, deve regolare sono le *interazioni* (punto a punto) fra questi nodi dimodoché si abbia complessivamente il risultato voluto. Ogni nodo dovrà quindi essere provvisto di un agente che si occupi di mediare le interazioni e di realizzare il comportamento cercato.

In una rete di questo tipo, ogni agente può essere consapevole solo dell'ambiente circostante (i suoi vicini direttamente collegati), senza una visione globale del sistema. Ci saranno, quindi, una moltitudine di interazioni locali fra agenti vicini, il cui andamento è influenzato e influenza solo l'ambiente

circostante nelle quali esse avvengono. Il comportamento complessivo sarà dunque *emergente* a seguito di tante interazioni locali.

Appaiono, infine, due fronti principali sui quali agire: il primo (*esterno*) è rappresentato dalle interazioni fra gli agenti, che, per essere regolato, necessita di un protocollo, mentre il secondo (*interno*) è rappresentato dal comportamento di ogni agente, che dovrà essere risolto in un algoritmo.

1.3.1 Il concetto di agente

Prima di procedere oltre è necessario esporre le proprietà definitorie degli agenti:

- *Autonomia*: c'è incapsulamento del controllo, solo il flusso dei dati oltrepassa l'agente; l'interazione avviene mediante scambio di informazioni, non c'è interfaccia o invocazione.

- *Azione*: evento collegato a chi agisce e che produce effetti osservabili. La *proattività* è il far accadere qualcosa, agendo.

- *Contesto*: è strettamente accoppiato con l'azione, in quanto l'agente è situato in un ambiente.

- *Reattività*: sia computazionale (controparte della proattività), sia ai cambiamenti. Vi dev'essere un efficace bilanciamento fra percezione (condizione necessaria per la reazione ai cambiamenti dell'ambiente) e rappresentazione.

- *Socialità*: l'autonomia richiede il concetto di socialità; agenti autonomi interagiscono in un MAS.

- *Interazione*: informazione e conoscenza sono scambiate fra gli agenti, direttamente o mediante l'ambiente.

- *Goal*: l'autonomia ha un senso ad alto livello nella realizzazione di un goal (stato del mondo da raggiungere) o di un task (tipo di attività da svolgere).

Tra queste, vi sono *4 qualità chiave* che definiscono il concetto di *weak agent*: autonomia, proattività, reattività, socialità. La mobilità è una caratteristica utile all'autonomia, ma non è necessaria.

1.3.2 Auto-organizzazione e emergenza nei MAS

Il concetto di *emergenza* fa riferimento a un *fenomeno che caratterizza i sistemi auto-organizzati ovvero può essere: crescita di una struttura, di un modello o di una proprietà coerente, in maniera non lineare* [1]. L'*intelligenza collettiva* è la capacità, che prescinde dal comportamento individuale, di un sistema di risolvere un problema; non si può prescindere dall'*osservatore*, che considera intelligente un certo comportamento, e dal *punto di vista*, che deve essere dunque sistemico. Questo tipo di intelligenza emerge da gruppi di *semplici* agenti. Si parla di *emergenza* in quanto non c'è un'entità centrale e la coordinazione fra le entità è priva di un progetto specifico; a partire da regole locali, il sistema assume una configurazione che dal punto di vista dell'osservatore è significativa. Gli agenti risultano entità semplici, perché è mediante la loro interazione che emergono i comportamenti intelligenti; vi sono ripetute interazioni locali, principalmente di tipo indiretto (tramite l'ambiente) e ciascuna entità risponde a delle regole semplici.

Proprietà dei sistemi di *intelligenza collettiva*:

- Computazione *distribuita*
- *Interazioni* dirette e indirette (prevalenti, tramite la modifica dell'ambiente)
- Agenti equipaggiati con capacità computazionali *semplici*
- *Robustezza*: il sistema resiste alla perdita di entità e mantiene le funzionalità (graceful degradation)
- *Adattatività* ai cambiamenti dell'ambiente

Queste proprietà, presenti in molti modelli naturali, risultano essere importanti per i sistemi artificiali.

L'*auto-organizzazione*, invece, fa riferimento a sistemi che esibiscono in maniera osservabile un'organizzazione, guidata da forze interne, e a seguito di modifiche di condizioni esterne, vanno a formare una nuova struttura. Essa, come fenomeno emergente, è un insieme di meccanismi dinamici per i quali, da interazioni fra componenti di livello (di astrazione) inferiore, appaiono strutture a livello globale. Non c'è una comunicazione globale o centralizzata fra questi componenti e non c'è un coordinatore o una gerarchia di flusso informativo. Gli effetti producono una diminuzione di *entropia*, ovvero si nota (dal punto di vista dell'osservatore) la creazione di *pattern temporali* (oscillazioni regolari del fenomeno) e *spaziali* (ordine dal punto di vista dello spazio). Questi

sistemi presentano molti stati di equilibrio e sono difficili da gestire; a seguito della modifica di alcuni parametri *critici* si possono avere comportamenti molto diversi.

L'auto-organizzazione come sorgente dell'emergenza è presente in molti ambiti naturali: la radice sta nel modello di comportamento degli insetti sociali per l'organizzazione delle colonie. Un esempio è la *stigmergia*, una forma di comunicazione indiretta, che avviene tramite l'ambiente. Gli agenti modificano l'ambiente e altri agenti reagiscono a tali cambiamenti. L'azione di un agente, in un certo tempo, è anche funzione dello stato dell'ambiente, che a sua volta dipende da azioni precedenti di agenti.

Un'applicazione di questi sistemi si ritrova nell'*ant colony optimization*, una tecnica meta-euristica basata su una popolazione di agenti, usata per problemi di ottimizzazione. Si ha un modello probabilistico e parametrico del *feromone*, che guida gli agenti a compiere percorsi stocastici seguendo con maggior probabilità il percorso ove la scia dove il feromone è più intensa. Altre applicazioni possono riguardare il *clustering*, ovvero raggruppare dati sulla base di criteri di somiglianza.

Il paradigma ad agenti può essere qullo giusto per modellare sistemi auto-organizzanti. I requisiti sono infatti simili: autonomia e comportamento incapsulato, azioni e percezioni locali, ambiente distribuito che supporta le interazioni, supporto per concetti di organizzazione e cooperazione. I MAS sono il riferimento corrente sia per la modellazione sia per l'ingegneria dell'auto-organizzazione.

I *fattori principali* nel design dei sistemi artificiali auto-organizzanti sono due:

1. Come progettare il comportamento individuale che collettivamente produce la proprietà emergente prevista? Molti dei sistemi auto-organizzanti sono ispirati a sistemi naturali (reverse engineering); servono comunque design pattern per procedere con un approccio scientifico nella realizzazione del sistema. I design pattern forniscono una soluzione riutilizzabile a prolemi ricorrenti in un dominio specifico.

2. Come valutare una soluzione specifica e garantirne la qualità? Si opera mediante un processo di ingegneria iterativo che prevede:

   - *Modellazione*: identificare le varie entità e i ruoli, formando un'architettura astratta.
   - *Simulazione*: prevedere dinamiche globali del sistema (le incognite sono i parametri degli agenti).
   - *Verifica*: il modello da verificare è spesso espresso in un transition system (con linguaggio formale) che permette model checking.

- *Tuning*: se i risultati non corrispondono ai requisiti, si aggiustano i parametri e si ritorna alla simulazione.

1.3.3 Modelli di coordinazione

L'origine della coordinazione (e della sua necessità) sta nella computazione parallela. Ci sono modelli e linguaggi di coordinazione:

- *Space-based*: basati su tuple, per sistemi aperti. La coordinazione è basata sulla presenza o sull'assenza della conoscenza. L'accesso è associativo sulla struttura della conoscenza stessa. `Linda` fornisce un supporto di questo tipo, dalla quale sono derivati altri modelli per la coordinazione distribuita, come `TuCSoN`.

- Modelli di coordinazione a *ispirazione naturale*: fortemente situati e connessi con l'ambiente. È l'antenato del modello space-based. Si estraggono pattern da sistemi naturali:

  - *Field-based*: particelle coordinate da un campo (elettromagnetico, di feromone, eccetera). Un esempio di questo modello è fornito da `TOTA` [5].
  - *Stigmergic-coordination*: coordinazione mediata dall'ambiente che porta all'emergenza dell'auto-organizzazione.

1.4 Scenario applicativo

Questo tipo di premesse sono comuni a diversi scenari applicativi, dove questo problema ha luogo. Si prenda un esempio fra tutti: una *redazione di giornale* è organizzata con una rete di nodi, sui quali lavorano i giornalisti, utenti del sistema. Ognuno di essi lavora su di una postazione e può inserire nella rete un articolo, catalogato secondo un concetto di tipo (ad esempio, cronaca, politica estera, spettacolo). Il sistema si occupa continuamente di gestire le informazioni esistenti e quelle che vengono inserite man mano. Ad esempio il giornalista può voler avere, nella postazione che occupa, tutti gli articoli appartenenti a un certa categoria, per cui il sistema farà in modo di organizzarsi per soddisfare l'esigenza dell'utente. Le caratteristiche del sistema devono poi essere tali da soddisfare le esigenze di mobilità e di apertura riguardo le postazioni, che possono aggiungersi, rimuoversi o essere spostate nella rete.

Ci si trova nella necessità di dover progettare un **middleware** che sia sempre attivo sulla rete, e che in maniera continuata mantenga le informazioni organizzate, a fronte di cambiamenti topologici o di inserimento di nuovi dati.

Capitolo 2

Progetto

In questo capitolo vengono presentate le scelte progettuali attuate, a fronte dell'analisi del problema fatta in precedenza. Prendendo spunto dallo stato dell'arte riguardo gli algoritmi di ispirazione naturale, si mette in campo una specifica soluzione.

2.1 Architettura del sistema

Andiamo dunque a progettare il sistema che soddisfa i requisiti e risolve le problematiche evidenziate nella fase di analisi; dovrà essere un vero e proprio **middleware**.

Come approccio alla progettazione, si decide di seguire il design pattern *Model-View-Controller*, per tener separati dati, interazione con l'utente e controllo.

2.1.1 Architettura TuCSoN

Descriviamo in breve parte dell'architettura che viene fornita da `TuCSoN`, dalla quale si prende spunto per lo sviluppo del sistema.

La sua infrastruttura fornisce servizi abilitando la coordinazione di agenti indipendenti distribuiti/concorrenti, fornendo tuple centre (spazi di informazione reattivi e condivisi) distribuiti sull'infrastruttura di nodi. Gli agenti inseriscono, prelevano e leggono le informazioni nella forma di tuple.

I *tuple centres* sono spazi di tuple programmabili, con comportamento reattivo, che può essere programmato dinamicamente. I componenti software accedono associativamente al tuple centre scrivendo, leggendo e prelevando tuple mediante semplici operazioni di comunicazione (`out, rd, in, inp, rdp`). A differenza degli altri spazi di tuple, il comportamento del *tuple centre* in risposta a eventi di comunicazione può essere fatto su misura alle esigenze

applicative definendo un insieme di *specification tuple* espresse in linguaggio `ReSpecT`.

La comunicazione è *generativa*, ovvero gli agenti comunicano creando e recuperando associativamente le tuple, la cui esistenza, una volta create, è indipendente da quella degli agenti. Questo rende possibile ottenere una proprietà di disaccoppiamento, temporale e spaziale, soddisfacendo il requisito di apertura nel sistema software.

Vengono inoltre forniti due tool:

- *CLI-Agent tool*: interfaccia shell per agenti umani.
- *Inspector tool*: tool per monitorare la comunicazione nel tuple centre e lo stato di coordinazione e per fare debug del comportamento del tuple centre.

`TuCSoN` dunque ci fornisce un'infrastruttura utile, che però nel nostro caso sarà sotto-utilizzata. Infatti i *tuple centers* verranno in seguito utilizzati come semplici *tuple spaces*.

2.2 Model: la rete

Considerando come obiettivo primario del sistema l'emergenza di una auto-organizzazione delle informazioni distribuite in una rete, la prima scelta progettuale è stata come realizzare la rete stessa, facente parte del *model*. In particolare, si è scelto di utilizzare spazi di tuple come contenitori delle informazioni e le tuple stesse come i dati da organizzare (tuple *information*); verrà quindi assegnato a ogni agente un proprio spazio di tuple (d'ora in avanti *ts*) in cui memorizzare localmente le informazioni, e sarà l'algoritmo comportamentale di ogni agente a decidere come movimentare le singole tuple (cioè se mantenerle nel proprio *ts* oppure passarle a un proprio vicino).

La rete del sistema è un insieme di collegamenti punto-punto, in cui ogni nodo ha un certo numero di vicini coi quali potrà comunicare. Per poter implementare ciò, ogni agente memorizza (tramite una tupla *neighbors*) quali sono i propri vicini, e tramite questa conoscenza potrà accedere ai loro *ts* per comunicare con essi e scambiare le informazioni. Il *ts* di ogni nodo è strettamente legato al nome dell'agente, in questo modo è possibile risalire ed accedere a un *ts* conoscendo il nome del nodo cui appartiene.

2.3 Controller: gli agenti

In questa sezione verranno descritte le scelte progettuali riguardanti gli agenti e i loro comportamenti. Per quanto riguarda l'implementazione degli agenti,

si è utilizzata la libreria di `TuCSoN` (`alice.tucson.api`) estendendo la classe astratta `Agent`. Ogni agente ha un body formato da una fase di inizializzazione, in cui l'agente setta i propri parametri e il proprio *ts*, e un comportamento ciclico, nel quale realizza tutte le proprie funzionalità. L'interazione con l'utente avviene mediante una interfaccia grafica, con la quale si possono monitorare lo stato dei nodi e richiedere funzionalità all'agente.

Come detto, il sistema deve garantire un comportamento auto-organizzativo al macro-livello, generato da scambi di tuple tra nodi vicini. Questi scambi sono guidati da una scia di feromone (in quanto ogni categoria di informazione ha il proprio feromone), che ogni agente rilascia nel proprio *ts* tramite una tupla *pheromone*; gli agenti calcolano il proprio feromone basandosi sul numero di tuple *information* nel proprio *ts* e in base al feromone dei vicini. Il feromone della rete forma quindi un campo di forza che ha un picco nel nodo con concentrazione massima di tuple, guidando così gli spostamenti locali delle stesse verso tale nodo.

Ogni spostamento di tuple viene fatto dall'agente possessore della tupla stessa, togliendola dal proprio *ts* e inserendola in quello del vicino prescelto. Tutti gli agenti possiedono un contatore delle proprie tuple (per poter calcolare il feromone); ciò avviene tramite una tupla *nTuple*. È così possibile modificare direttamente il contatore di un proprio vicino ogniqualvolta si sposta un tupla verso il suo *ts*.

2.3.1 Conteggio delle tuple

In primo luogo si è dovuta affrontare la problematica riguardante il conteggio delle tuple nel *ts* dell'agente; come detto in precedenza, il contatore (uno per ogni categoria di informazione) non può essere privato all'agente. Esso viene quindi implementato con una tupla contenente il valore del contatore e la categoria cui riferisce. Mantenere l'integrità di questi dati risulta così uno sforzo maggiore, ma in questo modo si rende possibile l'aggiornamento di tale valore da parte dei nodi vicini, quando questi ultimi spostano una tupla verso il *ts* dell'agente.

2.3.2 Gestione delle categorie

Un punto cardine del sistema è la divisione delle informazioni in categorie, le quali sono create localmente dagli agenti e devono poi essere propagate in tutta la rete, in quanto ogni nodo, pur non avendo tuple di una certa categoria, dovrà calcolarne e propagarne il feromone. Ogni agente ha una tupla *categories* contenente la lista di tutte le categorie a lui note, a cui corrisponderà anche una variabile locale nell'agente. La tupla *categories* deve fornire anche il supporto

alla propagazione di nuove categorie e ciò avviene tramite un *flag* booleano: all'aggiunta di una nuova categoria in un agente, esso modifica sia la propria lista di categorie (nella variabile locale e nella tupla) sia quella dei vicini, aggiungendo la nuova nella tupla *categories* del vicino e settando il relativo *flag* a `true`. Ogni agente controlla ciclicamente il *flag* della propria tupla *categories*: nel caso sia a `true` aggiorna la propria lista di categorie locale (ri-settando il *flag* della tupla a `false` una volta finito) e propaga la nuova categoria ai propri vicini che ancora non l'hanno nella propria lista. In questo modo, iterativamente di vicino in vicino, ogni nuova categoria viene propagata in tutta la rete.

2.3.3 Neighborhood

Come descritto in precedenza, ogni nodo è a conoscenza di tutti i propri vicini, coi quali comunica accendo ai relativi *ts*; ogni agente ha quindi una propria lista *neighbors* composta dai nomi dei nodi con cui è collegato e tramite i quali può accedere ai relativi *ts*. Una caratteristica fondamentale dei sistemi MAS è l'*apertura*, e quindi dovrà essere possibile aggiungere o rimuovere nodi e/o collegamenti dinamicamente. Queste funzionalità saranno utilizzabili dall'utente tramite l'apposita sezione della GUI di ogni agente.

Si crea un semplice algoritmo di aggiornamento delle liste dei vicini da parte di ogni nodo, similarmente a come avviene per l'aggiornamento delle categorie: ogni agente, come detto, mantiene salvati i nomi dei propri vicini sia localmente in un `ArrayList`, sia in una tupla *neighbors*, la quale contiene la lista stessa più un *flag* booleano. All'aggiunta (o, analogamente, alla rimozione) di un collegamento tramite la GUI, l'agente aggiorna entrambe le variabili (mantenendo il proprio *flag* a `false`) e in seguito modifica la tupla del nuovo vicino aggiungendosi alla lista del suo vicinato e settando a `true` il suo *flag*. In questo modo il nuovo vicino, controllando a ogni ciclo il *flag* della tupla, potrà accorgersi della presenza di nuovi collegamenti e cambiare anche il proprio `ArrayList` locale.

Nella fase di aggiunta di un nuovo vicino, dato che ogni agente non ha conoscenza del sistema, il controllo del nome del nuovo vicino è relativo solo alla conoscenza locale, ovvero: sono esclusi i nomi del nodo stesso, o di nodi coi quali un collegamento è già esistente. Non viene invece controllato l'inserimento di un collegamento a un nome di nodo non esistente nella rete.

2.3.4 Feromone

Il concetto su cui il sistema si basa [1.3.2] è quello di un modello probabilistico e parametrico basato sul feromone, che viene calcolato da ogni agente sulla

base della propria conoscenza locale. Prendendo spunto dalla *stigmergia* e dalla *coordinazione field-based*, il modello deve generare al macro-livello un campo di forza capace di guidare lo spostamento delle tuple per far creare autonomamente i cluster di tuple di stesse categorie.

Si è quindi resa necessaria la creazione di un algoritmo di aggiornamento del feromone per gli agenti. In primo luogo si è separato il valore del feromone per ogni categoria, creando così una lista di feromoni locale e diverse tuple *pheromone* contenenti il valore e la categoria cui si riferisce. Tale valore dipende sia dal numero di tuple contenute nel proprio *ts* sia dal valore di feromone dei propri vicini, in particolare da quello con valore massimo. Si è dunque arrivati alla formula definitiva con cui calcolare il feromone distinguendo due distinti casi:

1. *Il proprio contatore delle tuple è maggiore del massimo valore di feromone trovato nel vicinato*: in questa situazione l'agente ha il cluster maggiore di tuple della categoria del suo vicinato, e quindi deve attirare a sé le tuple dagli altri nodi. Il feromone assume semplicemente il valore del contatore delle tuple nel proprio *ts*:

$$pheromone = nTuple \tag{2.1}$$

2. *Nel vicinato si trova un valore di feromone maggiore del proprio contatore delle tuple*: si deve alimentare la propagazione del feromone in modo da continuare la creazione del gradiente. In questo caso si tiene in considerazione il proprio valore del contatore aggiungendoci la differenza con il valore di feromone massimo trovato, il tutto moltiplicato per un parametro β:

$$pheromone = nTuple + (maxPheromone - nTuple) * \beta \tag{2.2}$$

   Il parametro β è una variabile compresa tra 0 e 1, che identifica la propagazione del feromone: maggiore è β maggiore sarà la propagazione del feromone (e viceversa). Estremizzando, con $\beta = 1$ si avrebbe in tutti i nodi lo stesso valore di feromone indicante il valore delle tuple nel nodo con maggior concentrazione, mentre con $\beta = 0$ non si avrebbe propagazione e ogni nodo avrebbe come valore di feromone il proprio numero di tuple.

Ciclicamente l'agente deve aggiornare il proprio valore di feromone (per ogni categoria) e quindi confrontare il proprio valore del contatore delle tuple con il massimo valore di feromone dei vicini, aggiornando così il proprio feromone e riscrivendolo nel *ts*.

2.3.5 Spostamento delle tuple

Ogni ciclo del comportamento dell'agente corrisponde alla scelta dell'invio o meno di una tupla verso un nuovo *ts* e, nel caso, a quale vicino inviarla. Tali azioni sono guidate da diverse fasi:

- Scelta della *categoria* della tupla da inviare: in questa fase l'agente sceglie la categoria da cui prelevare la tupla (tramite il matching degli spazi di tuple). In particolare, è stato creato un algoritmo probabilistico che seleziona una categoria, tra quelle aventi almeno una tupla all'interno del *ts*, assumendo la probabilità di selezione inversamente proporzionale al valore del contatore delle tuple riferite alle categorie. L'algoritmo segue l'idea di abbassamento dell'entropia; mantenendo sempre probabilistiche le scelte, si dà maggiore probabilità all'invio ad altri *ts* delle tuple con aggregazione più debole all'interno del nodo. Avendo k categorie, La probabilità di selezionare la categoria i-esima è dunque:

$$p_i = \frac{\frac{1}{nTuple_i}}{\sum_{j=1}^{K} \frac{1}{nTuple_j}} \tag{2.3}$$

- Scelta del *vicino* a cui inviare la tupla: anche in questo caso la scelta avviene tramite un algoritmo probabilistico. Nell'insieme dei vicini viene considerato anche il nodo stesso, per dar la possibilità di mantere la tupla nel proprio *ts*. Questa scelta avviene proporzionalmente al valore del feromone della categoria selezionata per i *ts* dei vicini e per il valore del contatore delle tuple per il nodo stesso. In questa fase si è aggiunto il secondo fattore parametrico per il comportamento degli agenti, chiamato α. Questo parametro viene utilizzato quando il nodo che sta compiendo la scelta ha il valore del contatore delle tuple (della categoria prescelta) maggiore del feromone di tutti i vicini. Ciò vuol dire che esso possiede l'agglomerato numericamente maggiore di tuple della categoria e ha senso che abbia più possibilità di mantenere le tuple e non di inviarle. Si considera perciò la probabilità di mantenere la tupla non con $p = nTuple$ (come nei casi normali), ma $p = ntuple * \alpha$. Aumentando α, si aumenta quindi la probabilità di mantenere le tuple nei propri *ts*, se si ha l'agglomerato maggiore del vicinato.

- *Invio* della tupla ad altri *ts*: nel caso nella fase precedente si sia deciso di inviare ad un determinato vicino una tupla, allora si preleva casualmente una tupla nel proprio *ts* della categoria prescelta e la si inserisce nel *ts* del vicino, cambiando poi entrambe le tuple *nTuple* per aggiornarne i rispettivi contatori.

2.3.6 Ricerca delle informazioni

Una funzionalità richiesta è quella della ricerca delle informazioni all'interno della rete: ogni utente, tramite la GUI, può ricercare qual è il nodo con maggiore concentrazione di una determinata categoria.

La ricerca è fatta tramite una tupla *search* che ha un campo indicante la categoria che si vuole ricercare e un campo *path* in cui viene costruito iterativamente il percorso seguito dalla tupla. L'agente che vuole iniziare la ricerca confronta il proprio feromone con quello dei propri vicini; nel caso abbia il valore maggiore, la ricerca termina, in quanto è esso stesso ad avere la maggiore aggregazione di tuple. In caso contrario, viene scritta la tupla *search* nel *ts* del vicino con feromone maggiore, aggiungendo il proprio nome nel campo *path*. Ogni agente, a ogni ciclo, controlla la presenza di eventuali tuple *search* e esegue la stessa procedura appena descritta, inviando la tupla al vicino con feromone maggiore e aggiungendo il proprio nome al campo *path*, creando così una lista indicante il percorso eseguito della tupla. Quando la tupla arriva al nodo con maggiore feromone la ricerca termina; inizia così un procedimento inverso che porterà il risultato al nodo che ha dato il via alla ricerca. Tale procedimento viene eseguito similarmente alla fase precedente, ma tramite una tupla *return search*, contenente la categoria, il percorso da seguire per il ritorno al nodo iniziale, il valore del feromone massimo (ovvero quello del nodo finale raggiunto) e il percorso completo (dal nodo iniziale a quello finale). Ogni nodo, partendo da quello finale, cancella sé stesso dall'ultima posizione della lista *path* e invia la tupla *return search* al nuovo ultimo nodo della lista. In questo modo si ripercorre a ritroso il percorso fatto dalla tupla *search*, mantenendo intatti i valori del feromone massimo e del percorso finale, fino al raggiungimento del nodo che ha richiesto inialmente la ricerca. Quest'ultimo può così comunicare, tramite la GUI, qual è il nodo con aggregazione maggiore, il relativo valore di feromone e il percorso ottimo per arrivarci, in quanto viene seguita la scia di feromone.

Un'osservazione importante su questa funzione è che la ricerca è impostata senza variabili probabilistiche: viene sempre seguita la scalata a gradiente del feromone cercando e seguendo sempre il vicino con feromone massimo. Ciò porta inevitabilmente al raggiungimento di un *ottimo locale*, non globale.

2.3.7 Attrazione delle tuple

Altra importante funzionalità del sistema è la possibilità di aggregare determinate categorie in certi nodi della rete. Ciò è stato implementato con una sezione nelle GUI degli agenti, tramite la quale l'utente può decidere di attrarre verso di sé le tuple di una categoria a scelta.

Per *attrarre* le tuple, concettualmente è sufficiente aumentare il feromone del nodo in questione in modo che diventi il più alto nella rete. Per innalzare il valore di feromone correttamente, esso viene calcolato tramite un semplice algoritmo. L'idea è quella di ricercare il nodo con maggiore feromone della rete tramite la ricerca descritta in [2.3.6] e aumentare il proprio valore di feromone di:

$$attractVal = maxPheromoneOnNetwork * pathLength \tag{2.4}$$

dove `maxPheromoneOnNetwork` è il valore del feromone massimo trovato e `pathLength` è il numero di nodi che separano il nodo con aggregazione massima da quello che vuole attirare le tuple. In questo modo, il nodo attrattore avrà un valore di feromone tale da essere non solo il più alto, ma da creare una discesa a gradiente fino al nodo che possiede l'aggragazione maggiore delle tuple.

Ottenuto il valore `attractVal`, l'agente modifica la (2.2) aggiungendoci tale valore e avendo perciò:

$$pheromone = nTuple + (maxPheromone - nTuple) * \beta + attractVal \tag{2.5}$$

2.4 View: l'interfaccia grafica

Ogni agente è dotato di una GUI con cui comunicare con l'utente. Essa è formata da varie parti:

- *Node summary*: contiene le informazioni relative alle informazioni contenute nel proprio *ts* per ogni istante, indicando per ogni categoria di informazione la quantità di tuple e il relativo valore del feromone.
- *Insert tuple*: in questa sezione è possibile inserire nuove tuple o nuove categorie nel proprio *ts*.
- *Search tuple*: tramite questa funzione l'utente può ricercare il nodo della rete che contiene il cluster maggiore di tuple di una data categoria; verrà indicato anche il percorso di nodi da seguire per arrivare fino al nodo indicato.
- *Parameters*: questa sezione contiene la parte più strettamente legata all'algoritmo comportamentale dell'agente, in cui si possono modificare i valori α e β e attirare verso di sé le tuple di una categoria scelta.
- *Neighbors*: contiene le informazione relative alla rete conosciuta, ovvero ai propri vicini, indicandone la lista e dando la possibilità di aggiungere nuovi collegamenti o eliminare quelli esistenti.

È necessario gestire l'interazione tra agente e GUI, lasciando l'interfaccia sempre reattiva all'interazione dell'utente e non alterando il flusso di controllo dell'agente. Per questi motivi si è scelto di attuare il *design pattern Observer*, rendendo la GUI osservabile dagli agenti, notificando loro gli eventi a ogni interazione dell'utente; ogni agente avrà quindi una coda di eventi che viene controllata a ogni ciclo e dalla quale vengono prelevati e processati (se presenti) gli eventi.

Alla chiusura della GUI corrisponde lo spegnimento del'agente; esso deve quindi informare ogni suo vicino della propria prossima scomparsa dalla rete. Ciò viene fatto andando a modificare le tuple *neighbors* dei vicini: l'agente in chiusura andrà a rimuovere il suo nome dalle liste dei collegamenti attivi. Per evitare la *perdita delle infomazioni*, il nodo, prima di spegnersi, invia tutte le proprie tuple ai suoi vicini, decidendo, per ogni categoria, di inviare le tuple al vicino con maggior feromone di tale tipo.

Capitolo 3

Implementazione

In questo capitolo si descrive l'implementazione del progetto, spiegando le varie parti da cui è composto il sistema, dalla creazione dei nodi, della rete, all'agente e il suo comportamento. Verranno riportate le classi `Java` implementate e le signature dei metodi significativi.

3.1 Model: la rete

La rete fisica sopra la quale viene mandato in esecuzione il *middleware* ad agenti, può essere simulata tramite la classe `Network`. Essa è formata da un insieme di nodi (classe `Node`), sui quali sono mandati in esecuzione gli agenti, e di collegamenti. Questi ultimi sono riportati al livello degli agenti mediante le tuple *neighbors* presenti nei vari *ts*. La classe `Network` è utile in quanto può istanziare velocemente una certa topologia di rete con i relativi agenti e categorie, senza dover lanciarli singolarmente e aggiungere a mano le connessioni a run-time, semplificandone dunque la simulazione.

3.1.1 La classe Node

Un nodo è rappresentato da un nome (`name`) e da una lista di nomi (`links`) che rappresentano i collegamenti con i propri vicini.

```
String name;
ArrayList<String> links;
```

Nel costruttore gli viene assegnato il nome (`name`), che deve essere univoco nella rete (questa responsabilità è lasciata a colui che crea il nodo):

```
public Node(String name);
```

Vi sono poi i seguenti metodi, rispettivamente per settare un (nuovo) nome, per ottenere il nome, per aggiungere o rimuovere una connessione e per ottenere la lista dei vicini con cui il nodo è connesso:

```
public void setName(String name);
public String getName();
public void addConnection(String node);
public void removeConnection(String node);
public ArrayList<String> getConnections();
```

3.1.2 La classe Network

Questa classe descrive la rete e le categorie iniziali, rappresentate rispettivamente da due liste private alla classe:

```
ArrayList<Node> nodes;
ArrayList<String> categories;
```

Nel costruttore vengono inoltre inizializzate le categorie di informazioni da passare agli agenti, passategli come parametro:

```
public Network(ArrayList<String> categories) {
        nodes=new ArrayList<Node>();
        this.categories=new ArrayList<String>(categories);
}
```

Vi sono inoltre dei metodi rispettivamente per aggiungere un nodo alla rete, ottenere un nodo (dato l'indice con cui è memorizzato nella lista `nodes`, aggiungere un collegamento fra due nodi (dati i nomi) e lanciare gli agenti.

```
public void addNode(Node node);
public Node getNode(int i);
public void addLink(String nodeName1, String nodeName2);
public void launchAgents();
```

Quest'ultimo metodo è quello che si occupa di lanciare un agente per ogni nodo, passando all'agente il nome del nodo, le connessioni che ha con gli altri nodi e le categorie di informazioni presenti all'inizio, mediante la chiamata del metodo `spawn`:

```
new NodeAgent(name, node.getConnections(), categories).spawn();
```

3.1.3 Le tuple

Come detto, le tuple che rappresentano le informazioni da organizzare sono così formate:

```
information(String category, String content)
```

dove `category` è la categoria di appartenenza e `content` rappresenta il contenuto informativo.

Riepiloghiamo ora le tuple create per il funzionamento del sistema, che formano le *meta-informazioni*:

```
nTuples(String category, int number)
pheromone(String category, double value)
```

Esse rappresentano rispettivamente il numero di tuple appartenenti a una certa categoria e il valore di feromone associato.

```
categories(String categories, boolean flag)
neighbors(String neighbors, boolean flag)
```

Queste due tuple conservano le liste delle categorie e dei vicini noti ad ogni nodo. Sia la stringa `categories`, sia la stringa `neighbors` sono scritte nel formato che risulta dal metodo `.toString()` della struttura dati `ArrayList<String>`.

Infine vi sono le due tuple per la gestione della funzionalità di ricerca informazioni:

```
search(String category, String path)
return search(String category, String path, double
  maxPheromone, String finalPath)
```

Anche qui, le stringhe `path` e `finalPath` sono scritte nel formato che risulta dal metodo `.toString()` della struttura dati `ArrayList<String>`.

3.2 Controller: gli agenti

Ogni agente viene lanciato in esecuzione su di un nodo; esso è rappresentato da `NodeAgent`. Dopodiché, egli ha una parte di interazione con l'utente, che avviene mediante una sua interfaccia grafica `AgentGUI`, osservata dall'agente. Queste due entità interagiscono tra loro mediante un meccanismo ad eventi, ognuno rappresentato da una classe `Event`.

3.2.1 La classe NodeAgent

Questa è la classe che implementa l'agente, la principale (e la più corposa) del sistema. Il comportamento dell'agente è realizzato nel `body`; dopo una fase di inizializzazione, si passa al comportamento ciclico, che rimane attivo fin tanto che l'agente è presente nel sistema. Come detto, questa classe estende quella fornita da `TuCSoN` (`Agent`). Introduciamo i campi privati utilizzati in questa

classe:

String name	Nome dell'agente
ArrayList<String> categories	Contiene le categorie delle tuple
ArrayList<String> neighbors	Contiene i nomi dei vicini
ArrayList<Integer> nTuplesCategories	Contiene il numero di tuple per ogni categoria
ArrayList<Double> pheromoneCategories	Contiene il valore di feromone locale per ogni categoria
ArrayList<Double> attractCategories	Modificatore del feromone per la categoria da attrarre nel nodo
double alfa	Parametro per modulare la probabilità di tenere o inviare un certo tipo di tupla
double beta	Parametro per modulare la propagazione del feromone
AgentGUI gui	Interfaccia grafica
TupleCentreId myTc	Identificatore del proprio tuple centre
BlockingQueue<Event> eventQueue	Coda di eventi relativa all'agente
Util util	Classe con metodi di utilità
boolean active	Attiva o disattiva l'attività del nodo
String categoryAttractTuples	Rappresenta l'eventuale categoria di tuple da attrarre nel nodo

Nel costruttore di `NodeAgent` vengono poi inizializzati tutti questi campi, inoltre l'agente si registra come osservatore dell'interfaccia grafica, con il metodo `register` della GUI, dimodoché gli vengano notificati i suoi eventi. Il comportamento di `NodeAgent` viene realizzato nel metodo:

```
protected void body();
```

nel quale, dopo una fase di inizializzazione, con il metodo `inizialization`, vengono svolte un insieme di operazioni cicliche, ovvero:

1. Lettura e gestione di eventuali *eventi* provenienti dalla GUI.
2. Gestione di eventuali tuple per la *ricerca* di informazioni.
3. Aggiornamento dei propri *vicini*
4. Aggiornamento delle proprie *categorie* di informazioni e, se serve, propagazione l'aggiornamento delle categorie anche ai vicini.
5. Scelta della categoria di tuple da *spedire* (se possibile).
6. Aggiornamento del valore di *feromone* per ogni categoria nota
7. Scelta del *vicino* a cui inviare una tupla della categoria scelta oppure di tenersela nel proprio `ts`.

8. Eventualmente, *invio* della tupla al vicino.

L'inizializzazione di un agente è implementata nel metodo `inizialization()`, nel quale si scrivono nel proprio `ts` le tuple iniziali e si attende che tutti i vicini siano giunti alla fine di questa fase. Per ogni categoria si scrive un certo numero di tuple *information* (usando un numero `Random`, da 0 a 100) e le relative *nTuple* e *pheromone* con i valori iniziali. Il metodo `waitForNeighbors` blocca il proseguimento dell'agente fino a che i suoi vicini non sono giunti al termine delle fasi di inizializzazione. Ciò è necessario per lanciare il sistema una volta che la topologia di rete descritta in `Network` è stata creata; diversi agenti, infatti, potrebbero dover scrivere molte più tuple di altri (in quanto possono essere in numero casuale), perciò prima di fare interagire i vari nodi, si attende che ognuno sia giunto alla stessa condizione di partenza. Qui inoltre si fa uso del metodo`initializeCategories`, che serve per recuperare dai vicini eventuali categorie non note; ciò è utile nel caso in cui i nodi vengano lanciati con categorie diverse, oppure quando un nuovo nodo si aggiunge a rete già avviata e porta con sé nuove categorie. Nel caso vengano trovate di nuove, mediante il metodo `updateNeighborsCategories` si propaga l'aggiornamento ai vicini.

Il punto 1 viene svolto nel metodo `manageEvents`. Esso si occupa di gestire eventuali eventi (classe `Event`) provenienti dall'interfaccia grafica. L'agente ha una coda nella quale possono essere accodati eventi (mediante il metodo `Notify`), che egli può recuperare utilizzando `fetchEventIfPresent`.

Ad ogni ciclo vengono gestiti tutti gli eventi presenti, creati a seguito di azioni dell'utente, che possono essere i seguenti:

- *insert category*: questo evento notifica l'inserimento di una nuova categoria nel nodo. In questo caso si aggiorna la propria conoscenza e si propaga la categoria ai vicini mediante il metodo `updateNeighborsCategories`.

- *insert tuple*: notifica l'inserimento di una nuova tupla nel nodo, appartenente a una nota categoria. In questo caso si aggiorna il contatore *nTuple*.

- *add link*: notifica l'aggiunta di un nuovo collegamento ad un nodo. Qui si fa uso del metodo `addLink`, che si occupa di rendere effettivo il collegamento col nuovo vicino. Si controlla che il nome inserito sia valido (ovvero che non sia quello del nodo stesso o che non sia di un nodo già presente tra i vicini), dopodiché si aggiorna la propria tupla *neighbors* e quella del vicino. Si procede poi al recupero e alla diffusione di eventuali nuove categorie mediante i metodi sopracitati `initializeCategories` e `updateNeighborsCategories`.

- *remove link*: notifica la rimozione di un collegamento con un vicino. Si utilizza il metodo `removeLink` per rimuovere il nome del vicino dalla propria tupla *neighbors* e da quella del vicino da cui ci si sta scollegando.

- *search tuples*: notifica la ricerca di una categoria di tuple. Si utilizza il metodo `chooseNeighborWithMaxPheromone` per vedere chi, nel vicinato, ha il massimo valore di feromone per la categoria data, dopodiché si confronta col proprio valore (metodo `comparePheromones`. Se si ha il feromone maggiore, la ricerca si ferma qui, altrimenti viene inviata la tupla *search* al vicino con feromone maggiore.

- *attract tuples*: notifica l'attrazione di una categoria di tuple nel proprio nodo. Il procedimento inizialmente è analogo a quello della ricerca, dopodiché, se serve, si aumenta il proprio feromone del valore necessario per attirare le tuple.

- *stop attract tuples*: notifica lo stop nell'attrazione di una categoria di tuple iniziata in precedenza, risettando il valore di feromone col normale calcolo.

- *plus/minus alfa, plus/minus beta*: col primo, si aumenta o si diminuisce di 0.5, fino a un minimo di zero, il valore del parametro *alfa*, mentre col secondo si aumenta o si diminuisce di 0.1 (rimanendo fra 0 e 1) il valore di *beta*.

- *close*: notifica lo spegnimento del nodo. Come prima cosa si rimuovono tutti i collegamenti con i vicini, mediante il metodo `removeLink`, dopodiché si inviano tutte le tuple in proprio possesso al vicino che ha il maggior valore di feromone (che si trova col metodo `bestNeighborForCategory`), per ogni categoria. Infine si disattiva la GUI e si ferma l'attività del nodo.

Il punto 2 viene svolto nel metodo `manageSearch`, che si occupa di gestire la presenza, nel proprio `ts`, di eventuali tuple *search* o *return search*, in modo da soddisfare l'algoritmo di ricerca. Ad ogni ciclo, vengono smaltite tutte quelle presenti nel nodo.

Il punto 3 viene svolto nel metodo `updateMyNeighbors`, che legge nel proprio `ts` la tupla *neighbors* e, se qualche altro nodo ha aggiunto sé stesso in tale tupla, esso viene aggiunto al proprio `ArrayList` dei vicini, aggiornando poi la GUI.

Il punto 4 serve per mantenere aggiornate le categorie, mediante i metodi `updateMyCategories` e `updateNeighborsCategories`. Il primo serve per aggiornare la propria conoscenza sulle categorie, se qualche altro vicino ha

propagato una sua categoria nella tupla *categories* dell'agente, il secondo per continuare a propagare il cambiamento.

Il punto 5 viene svolto in `chooseCategory`, che è un metodo per la scelta probabilistica della categoria da cui sarà presa la tupla da poter inviare a un vicino (secondo il modello del feromone). La probabilità di essere scelta è inversamente proporzionale al numero di tuple presenti di tale categoria. Se non vi sono tuple, quindi se non si può scegliere alcuna categoria, si salteranno i punti 7 e 8.

Il punto 6 serve per mantenere aggiornati i valori dei feromoni relativi ad ogni categoria di tuple. Ciò viene fatto nel metodo `updatePheromone`, come spiegato in [2.3.4], per poi aggiornare la GUI.

Nel punto 7 si sceglie o un vicino al quale inviare la tupla della categoria scelta al punto 6 oppure di mantenere tale tupla all'interno del proprio `ts` (in tal caso si salta il punto 8); il metodo usato è `chooseNeighbor` e la probabilità di ogni vicino di essere scelto è basata sul valore di feromone relativo alla categoria a cui appartiene la tupla. Il funzionamento è spiegato in [2.3.5].

Infine, al punto 8, si spedisce effettivamente la tupla *information* della categoria scelta al punto 5 al vicino scelto al punto 7.

Si ritorna poi al punto 1, per ripetere il ciclo.

3.2.2 La classe Util

In questa classe vi sono implementati metodi di utilità per l'agente, che vengono utilizzati in varie parti del codice. Il primo metodo è:

```
public ArrayList<String> convertArrayToArrayList(String[]
  array)
```

Serve per avere, da un `array` di stringhe, un `ArrayList` di stringhe.

Il secondo è:

```
public String[] splitStrings(String strToSplit)
```

Questo metodo, a partire da una stringa (scritta secondo il formato di un `ArrayList.toString()`, quindi, supponendo che l'`ArrayList` abbia tre elementi, con `toString()` diverrebbe: `[el1, el2, el3]`) ritorna un array di stringhe con un elemento per ogni cella.

3.3 Model: gli eventi

Gli eventi sono rappresentati dalla classe `Event` e da altre classi che estendono la stessa e ne rappresentano tipologie più specifiche. Gli eventi vengono generati dalla GUI e accodati nell'apposita coda dell'agente. Ogni evento ha due

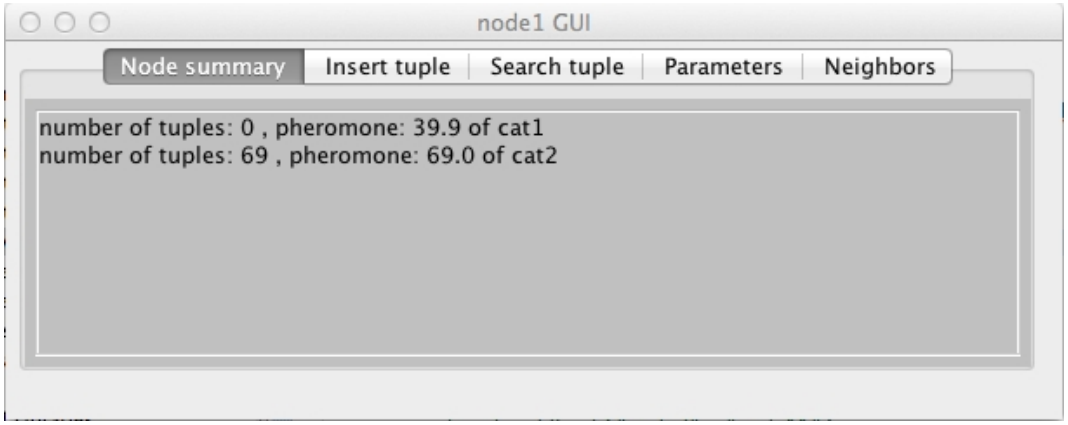

Figura 3.1: AgentGUI, sezione Node summary

campi privati che rappresentano la descrizione e la sorgente che l'ha generata (che deve implementare l'interfaccia `ObservableComponent`:

```
String descr;
ObservableComponent source;
```

Le classi effettivamente utilizzate sono estensioni di `Event` e sono le seguenti:

- *EventCategory*: ha un ulteriore campo che memorizza una categoria, e viene utilizzato per rappresentare gli eventi *insert category*, *search tuples*, *attract tuples* e *stop attract tuples*.
- *EventInsertTuple*: ha due ulteriori campi che memorizzano la categoria e il contenuto di una tupla e viene utilizzata per rappresentare l'evento *insert tuple*.
- *EventLink*: ha un ulteriore campo per memorizzare il nome del vicino; si utilizza per rappresentare gli eventi *add link* e *remove link*.

3.4 View: l'interfaccia grafica

L'interfaccia grafica degli agenti è stata implementata tramite la classe `AgentGUI`; tramite la GUI l'utente può osservare e interagire con l'agente. Essa è stata divisa in varie sezioni, navigabili tramite un `JTabbedPane`, come mostrate di seguito:

Come detto nel cap.[2.4], la comunicazione tra GUI e agente avviene tramite l'implementazione del *design pattern Observer*; in particolare la clas-

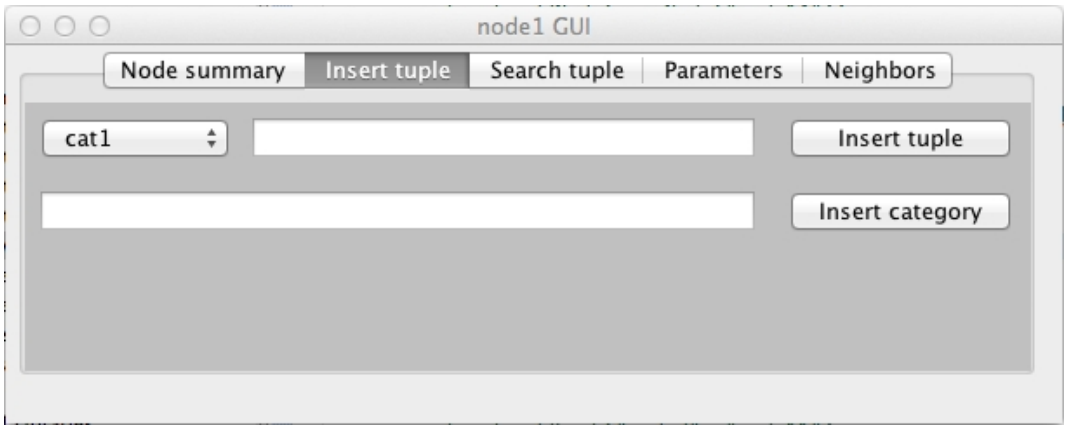

Figura 3.2: AgentGUI, sezione Insert tuple

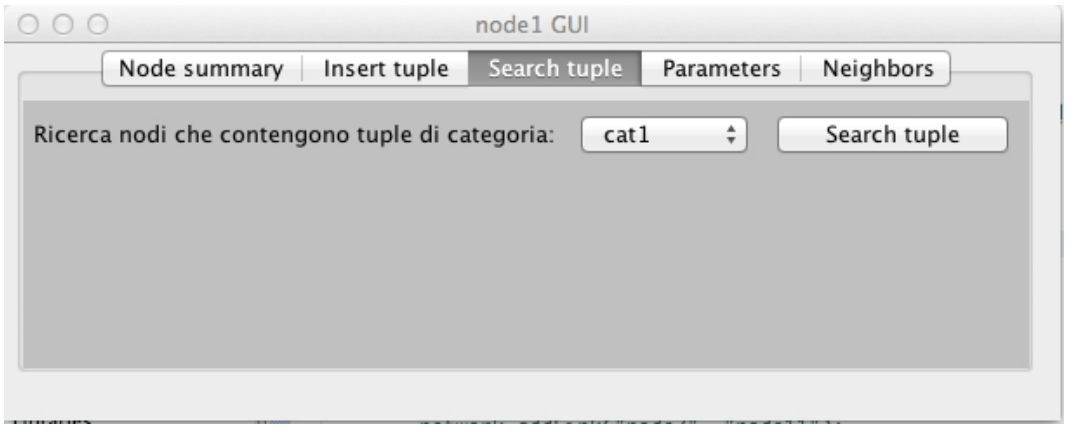

Figura 3.3: AgentGUI, sezione Search tuple

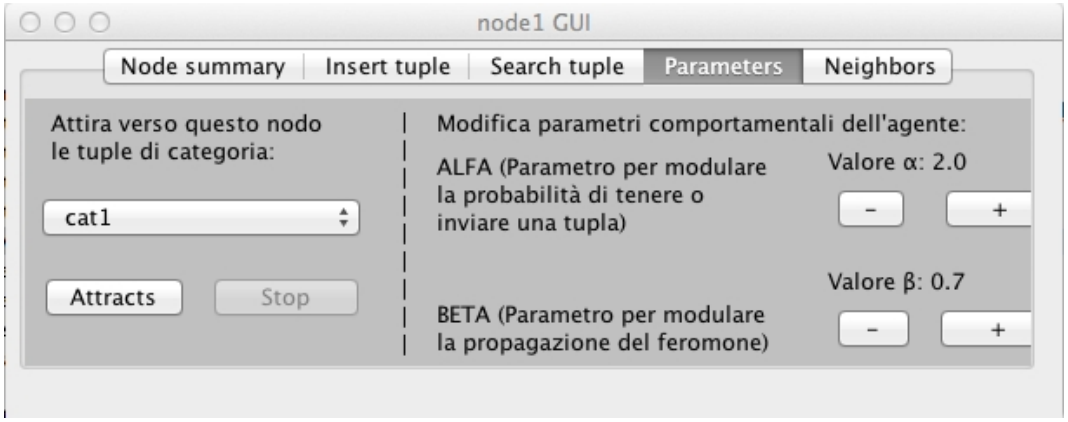

Figura 3.4: AgentGUI, sezione Parameters

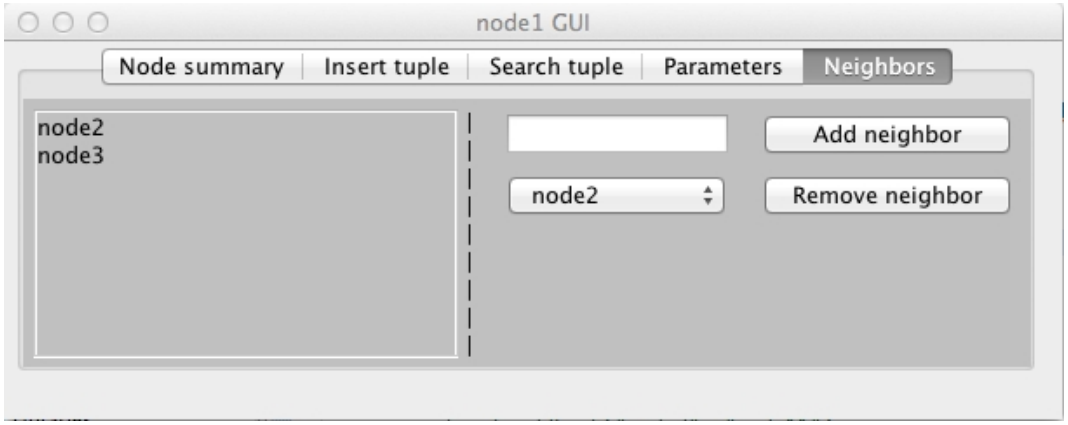

Figura 3.5: AgentGUI, sezione Neighbors

se `AgentGUI` implementa l'interfaccia `ObservableComponent`, che aggiunge i seguenti metodi:

```
void register(NodeAgent agent);
void notifyEventToObservers(Event ev);
```

Tramite il metodo `register` l'agente si registra come osservatore della GUI, la quale potrà poi notificare ai suoi osservatori ogni evento tramite il metodo `notifyEventToObservers`. Ad ogni pulsante della GUI è associato a un metodo che notifica all'ascoltatore il relativo evento.

Per quanto riguarda ciò che l'agente vuole visualizzare nella GUI, sono stati implementati alcuni metodi tramite i quali la GUI viene aggiornata:

```
public void updateSummary()
```

Viene utilizzato per aggiornare la sezione `Node Summary`, nella quale vengono visualizzati, per ogni categoria, il valore del contatore delle tuple e del feromone.

```
public void updateNodeNeighbors()
```

Aggiorna semplicemente la lista dei vicini nella sezione `Neighbors`.

```
public void updateComboBoxCategories()
```

Viene usato ogniqualvolta l'agente riscontra una nuova categoria; questo metodo aggiorna tutti i `JComboBox` utilizzati nelle varie sezioni.

I metodi `setResultSearchText`, `insertTupleResult`, `neighborsResult`, `updateParameters` sono utilizzati per visualizzare il risultato di un 'operazione avvenuta in risposta ad un'azione dell'utente sulla GUI.

3.5 Esecuzione

Innanzitutto bisogna avviare il servizio di `TuCSoN` mediante shell, con il comando:

```
# java -cp tucson.jar alice.tucson.service.Node
```

Poi, per lanciare il sistema ci sono due modi:

1. Lanciare la classe `NetworkMain`, che si occupa di creare la rete della classe `Network` e di mandarla in esecuzione. Essa ha un metodo `main` dove si creano le categorie iniziali, si istanzia la `Network`, si aggiungono i nodi con il metodo `addNode`, le connessioni con il metodo `addLink` e infine si lancia il metodo `launchAgents`. Questa classe può essere anche lanciata (con una rete di esempio: 4 agenti collegati come un quadrato) mediante il file *Network-4agents.jar*, facendoci doppio click sopra, oppure con il comando:

```
# java -jar Network-4agents.jar
```

2. Lanciare la classe `NodeAgentMain`, la quale manda in esecuzione un singolo `NodeAgent`. Essa dispone di un metodo `main`, che permette di lanciare un agente separatamente da tutto il resto e connettersi a una rete già esistente. Gli si può passare come parametro il nome dell'agente. In sua assenza, il nome viene scelto connettendo la stringa node con un numero casuale fra 20 e 1020. L'agente si suppone inizialmente connesso a nessun vicino. Questa classe può essere anche lanciata mediante il file *NodeAgent.jar*, facendoci doppio click sopra, oppure con il comando:

   ```
   # java -jar NodeAgent.jar [name]
   ```

Il sistema è inteso come un middleware da mantenere sempre attivo e il servizio fornito da `TuCSoN` è da riavviare, nel caso si voglia rilanciare il sistema.

Capitolo 4

Valutazione dei risultati

In questo capitolo si analizzano i comportamenti emergenti dal sistema durante la sua esecuzione e si valutano i risultati.

4.1 Aggregazione delle tuple

Per giudicare se questa funzionalità sia stata ottenuta, si è scelto di monitorare l'evoluzione di una rete composta da 4 nodi collegati come un quadrato. Memorizzando, per ogni ciclo compiuto dagli agenti, il numero di tuple per ogni categoria contenute nei vari *ts*, si verifica che per ogni categoria, dopo un certo tempo, le tuple risultano aggregate in un nodo della rete. Graficando l'evoluzione dei vari *nTuple* si è ottenuto il risultato in figura [4.1], in cui chiaramente si può constatare lo spostamento delle tuple fra i nodi, fino ad arrivare ad un completa aggregazione di quelle di *category1* e *category2* nell'agente *node2*, mentre la *category3* viene aggregata nel *node1*.

4.1.1 Entropia

Un parametro utile per giudicare l'effettiva emergenza di una auto-organizzazione della rete è l'*entropia (H)*, che fornisce una misura del disordine (o caos) di un sistema. In particolare, considerati gli obiettivi del sistema, si è calcolata l'entropia divisa per categorie (H_i con $i = 0..k$ categorie). In questo modo più alto è il valore di H_i, maggiore è il disordine delle tuple della categoria i-esima nella rete; mentre un valore di $H_i = 0$ significa che la rispettiva categoria è completamente organizzata nella rete.

Ora si denota con n_ij il valore di *nTuple* della categoria i nel nodo j; supponendo di avere una rete composta da N nodi e definendo: $NTtot_i = \sum_{j=0}^{N} n_{ij}$ (ovvero il totale delle tuple della categoria i-esima presenti nella rete),

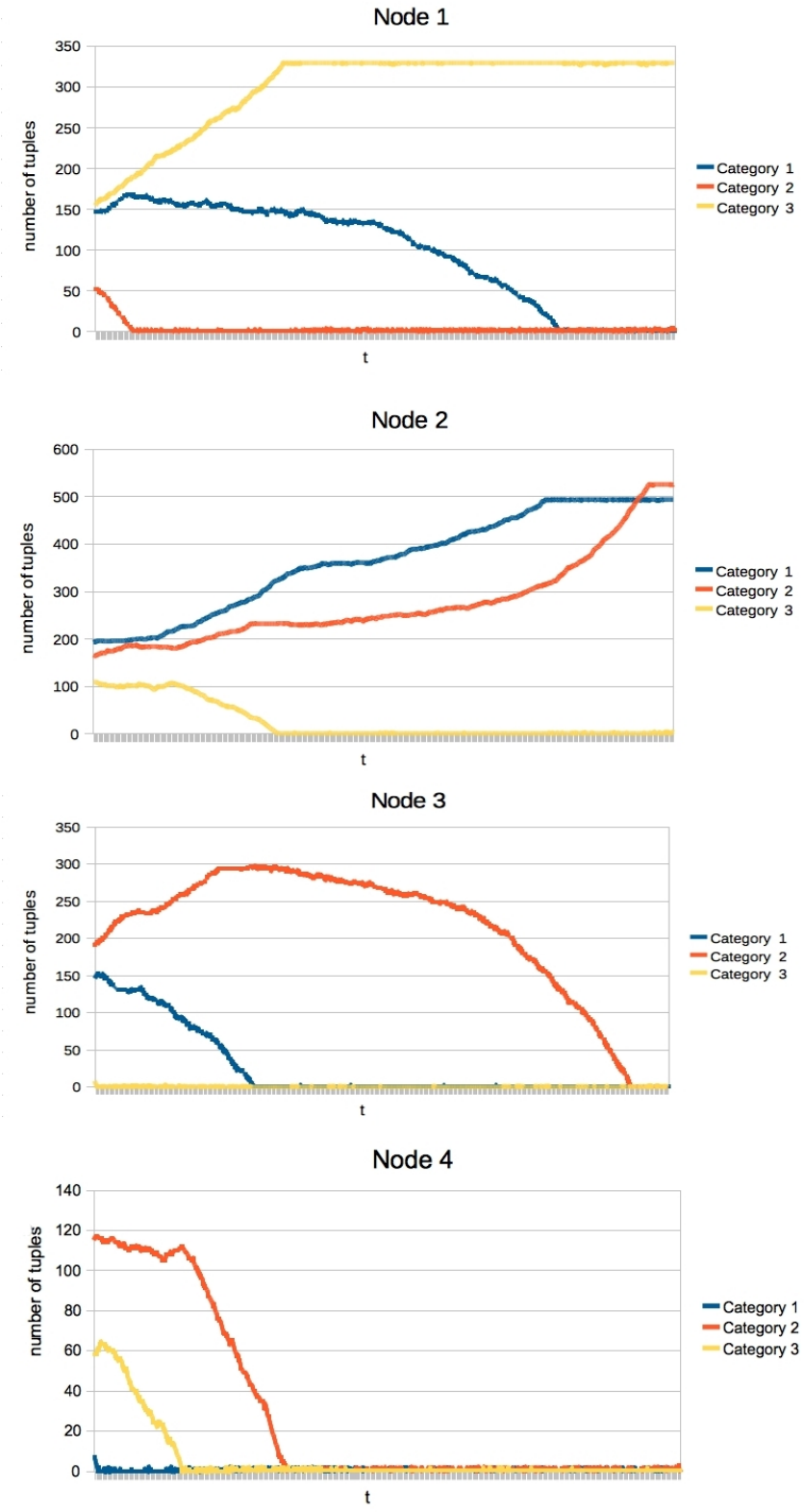

Figura 4.1: Evoluzione del numero di tuple nei 4 nodi della rete

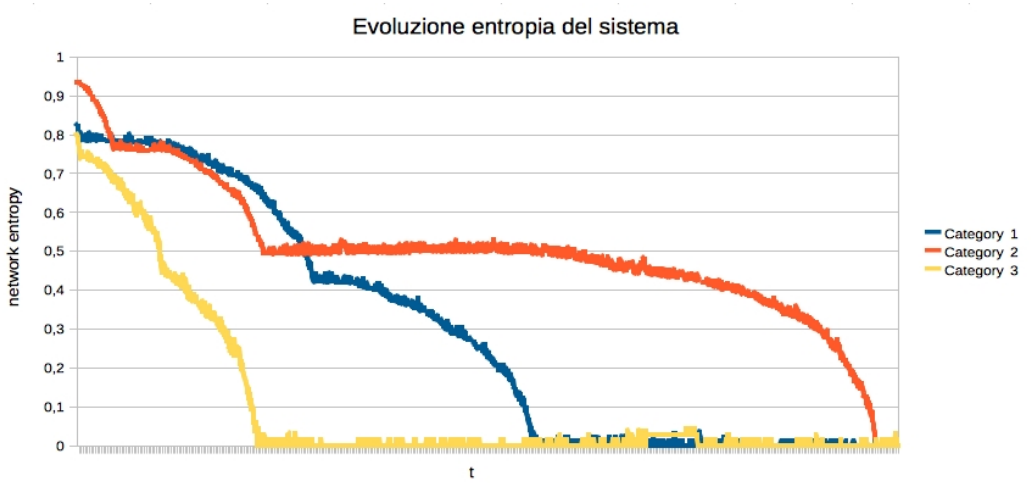

Figura 4.2: Evoluzione dell'entropia delle varie categorie

si calcolca l'entropia del sistema relativa alla categoria i come:

$$H_i = K * \sum_{j=0}^{N} \frac{n_{ij}}{NTtot} \log_2 \frac{NTtot}{n_{ij}} \tag{4.1}$$

K è una variabile utilizzata per modulare tra 0 e 1 il valore dell'entropia: $K = \frac{1}{\log_2 N}$. In questo modo l'entropia massima del sistema si avrà per $H = 1$, mentre con $H = 0$ si avrà il completo ordinamento della rete.

In base a questo parametro è facile constatare come l'entropia del sistema diminuisca col passare del tempo, graficando l'andamento di H nella rete a quadrato descritta in [4.1]; il risultato ottenuto è la figura [4.2], che mostra chiaramente come l'entropia inizale del sistema sia alta (in particolare per la *category2* è prossima a 1) e con l'andare del tempo diminuisce. Confrontando questo grafico con i precedenti movimenti delle tuple [Fig.4.1] è semplice notare come l'entropia sia strattamente dipendente dallo spostamento delle tuple verso i cluster di categorie formatisi.

4.2 Creazione di un campo di forza del feromone

Come descritto nel capitolo 2, il feromone di ogni nodo è calcolato tramite percezioni locali dell'ambiente; si deve quindi verificare che effettivamente il modello creato è in grado di formare *globalmente* un campo di forza che guidi gli

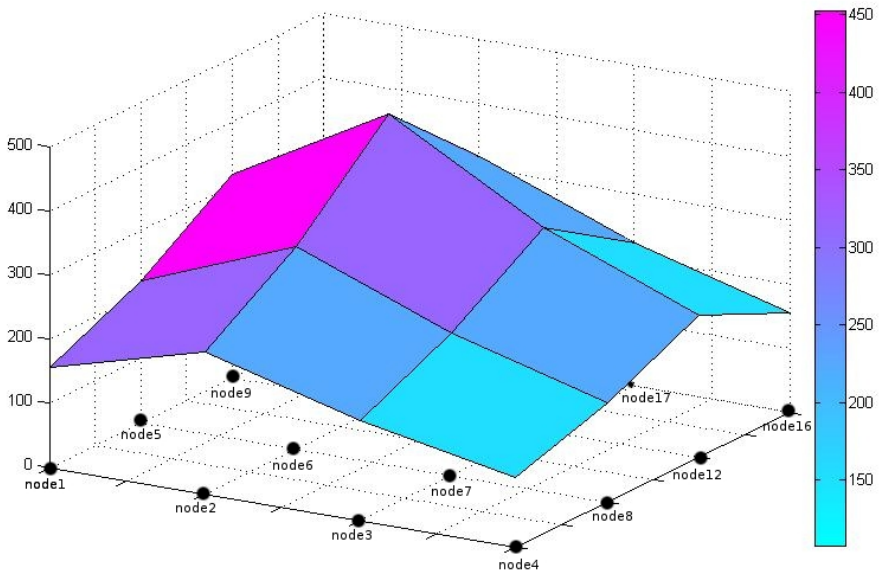

Figura 4.3: Feromone nella rete per la *category1*

spostamenti delle tuple e le ricerche verso il nodo con maggiore concentrazione di tuple di un certo tipo.

Anche in questo caso è risultata utile la creazione di un grafico che mostra (in un dato istante in cui la rete è quasi totalmente già organizzata) il valore del feronomone di ogni nodo della rete. Nella figura [4.3] viene mostrato il campo di forza generato dai valori del feromone nei vari nodi di una rete a griglia 4x4; si può effettivamente notare come, tramite una scalata a gradiente, ogni nodo indirizza le proprie tuple verso quello con maggior concentrazione di feromone (e quindi di numero di tuple maggiore).

4.3 Ricerca di una categoria di tuple

L'efficacia della ricerca di una categoria dipende da alcuni fattori: il sistema deve infatti trovarsi in una situazione di relativa stabilità e l'aggregato di tuple che si sta cercando deve essere un minimo numeroso. Ciò per evitare che, mentre la tupla *search* segue la scia di feromone, quest'ultima cambi repentinamente, e il percorso risulti così non più relativo ad un ottimo locale. Negli altri casi di normale funzionamento, la ricerca porta ad un risultato utile/significativo.

4.4 Attrazione di una categoria di tuple

L'attrazione si basa inizialmente su di una ricerca, quindi i presupposti sulla validità di questa funzione sono gli stessi. Inoltre l'efficacia dell'attrazione varia a seconda degli scenari nei quali la si usa; essa risulterà più veloce quanto più sarà vicino il nodo (o i nodi) dai quali si stanno attraendo (in termini di numero di salti). Inoltre, per come è stato implementato l'algoritmo di invio delle tuple, sarà più difficile attrarre una categoria da un nodo se quest'ultimo ha (relativamente) molte tuple della categoria da attrarre e poche tuple di altre categorie. In questa situazione, il nodo ha più probabilità di liberarsi delle categorie meno numerose, piuttosto che di quella più numerosa, rallentando così il processo di attrazione. I casi d'uso, dunque, presentano una buona dose di variabilità: il caso ottimo risulta decisamente più veloce di quelli peggiori.

4.5 Sviluppi futuri

Il progetto consiste in un sistema funzionante e che (come analizzato in precedenza) ha raggiunto gli obiettivi preposti; ci sono tuttavia diversi sviluppi che protrebbero essere implementati per migliorarne la qualità. Ne elenchiamo ora brevemente alcuni che potrebbero essere aggiunti:

- Miglioramento dell'algoritmo comportamentale dell'agente, aggiungendo nuovi parametri (ad esempio introducendo feedback positivi o negativi per favorire situazioni migliori e sfavorire quelle peggiori, già utilizzato negli algoritmi *ACO*), o rendendo i parametri già presenti dinamici e autonomi, variando cioè il comportamento dell'agente dinamicamente (e autonomamente) durante l'esecuzione dello stesso, in base a vari fattori. Ad esempio, pensando allo scenario della redazione giornalistica, si potrebbe pensare di aumentare il feromone di un nodo che immette nel sistema tuple di una certa categoria, immaginando quindi che l'interesse di quell'agente si rivolga principalmente a quella tipologia di informazioni e quindi cercare di portare tale categoria di tuple al nodo stesso.

- Assegnamento delle informazioni a più categorie, o creazione di gerarchie di categorie.

- Sfruttamento delle potenzialità dei *tuple center* di `TuCSoN`, rendendoli reattivi ai cambiamenti e deresponsabilizzando l'agente.

- Creazione di un linguaggio di definizione della rete, tramite il quale inizializzare la rete.

Bibliografia

[1] Goldstein J. (1999) - *Emergence as a Construct: History and Issues*

[2] Omicini A. and Denti E. (2001) - *From tuple spaces to tuple centres*

[3] Omicini A. and Zambonelli F. (1999) - *Coordination for Internet application development*

[4] Ricci A., Viroli M. and Omicini A. (2005) - *Environment-based coordination through coordination artifacts*

[5] Mamei M. and Zambonelli F. (2008) - *Programming Pervasive and Mobile Computing Applications: the TOTA Approach*

[6] Casadei M., Mirko Viroli M. and Gardelli L. (2009) - *On the collective sort problem for distributed tuple space*