Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

Tesi di Laurea Triennale in Ingegneria Informatica

Generatone di modelli NetLogo

Candidato
Aurel Pjetri

Relatore
Prof. Enrico Vicario

Co-relatore
Dott. Sandro Mehic

Anno Accademico 2015/2016

Indice

Introduzione **i**

1 Sezione teorica **1**

1.1 Simulazione delle masse 1

1.2 NetLogo 1

1.2.1 Linguaggio 2

2 Sviluppo del progetto **3**

2.1 Obiettivo e requisiti 3

2.2 Documento XML 5

2.3 Struttura Interna 6

2.3.1 Parser XML e Builder 7

2.3.2 Visitor 8

2.4 Modello NetLogo 13

2.4.1 Ambiente 13

2.4.2 Behaviors e stato iniziale 15

Bibliografia **17**

Introduzione

La simulazione delle masse negli ultimi decenni ha attirato l'attenzione di un numero crescente di gruppi di ricerca. Gli ambiti in cui questo fenomeno trova applicazioni interessanti sono molteplici: scienza della sicurezza (evacuazione di ambienti pubblici), progettazione architetturale, studio del flusso di traffico, studio sociologico dei comportamenti collettivi e anche intrattenimento.
La dinamica delle folle è di grande interesse sia sotto condizioni critiche che normali. Nella fase di progettazione di ambienti chiusi come centri commerciali, stadi e scuole dove centinaia di persone si concentrano per scopi differenti bisogna garantire una via di uscita a tutti gli utenti nonostante il numero limitato di punti di uscita. In condizioni normali si può utilizzare per studiare il comportamento delle folle in ambienti come fiere, quartieri, o intere città, in modo, ad esempio, da agevolare la viabilità nei tratti più affollati.
Lo studio delle masse è un argomento molto affascinante, ma allo stesso tempo, molto complesso. Per simulare situazioni del mondo reale è spesso richiesta la modellazione di comportamenti collettivi, come anche individuali, di folle di grandi dimensioni che si muovono in ambienti anch'essi molto grandi. La complessità quindi si sviluppa in due direzioni: logica e temporale. Le simulazioni di modelli di grandi dimensioni, infatti, sono processi che

richiedono molto tempo e grande potenza computazionale. Come è possibile ridurre il tempo di esecuzione di queste simulazioni?
Un possibile approccio (per risolvere questa criticità) è integrare la simulazione vera e propria con metodi analitici come le Catene di Markov. Si scompone il modello in porzioni di dimensioni minori e si eseguono simulazioni isolate su ognuno di questi, quindi si usano metodi matematici e analitici per mettere insieme questi risultati ed ottenere dei dati che cercano di avvicinarsi il più possibile a quelli che si sarebbero ottenuti eseguendo la simulazione sul modello completo.
Questo approccio, però, comporta una moltiplicazione delle simulazioni da eseguire su "patch" adiacenti che rischia di allungare ulteriormente il tempo necessario per una simulazione completa.
L'obiettivo perseguito in questa tesi, quindi, è ridurre i tempi di simulazione automatizzando la compilazione dei modelli NetLogo che eseguiranno le simulazioni.
In concreto il progetto consiste in un programma Java che riceve in ingresso un documento XML in cui è descritto il modello suddiviso in topologia, comportamenti possibili degli attori e stato iniziale dell'ambiente. In seguito alla costruzione degli oggetti Java necessari per la rappresentazione del modello viene eseguita la scrittura del codice NetLogo che modella l'ambiente inizialmente descritto. In questo modo si evita la scrittura da parte di un umano del codice NetLogo sicuramente meno semplice e intuitivo del XML.

Capitolo 1

Sezione teorica

«descrizione del contesto teorico della tesi»

1.1 Simulazione delle masse

1.2 NetLogo

NetLogo è un linguaggio di programmazione e un ambiente di modellazione di sistemi complessi. Adatto per la simulazione e lo studio di fenomeni naturali e sociali che si evolvono nel tempo. Gli utenti possono dare istruzioni a migliaia di agenti in modo individuale o collettivo, permettendo quindi lo studio dei loro comportamenti su più livelli, come quello microscopico dei behaviors individuali o quello macroscopico delle loro interazioni con gli altri. Nasce a scopo educativo e di ricerca, dalla fusione di **Logo** e **StarLisp**. Dal primo eredita il principio *low threshold , no ceiling*, ovvero bassa soglia di conoscenza per il suo utilizzo, rendendolo accessibile a utenti inesperti nella programmazione, ma allo stesso tempo completa programmabilità rendendolo quindi anche uno strumento utile per la ricerca. Da Logo viene ereditato

anche il concetto fondamentale di *turtle*, con la differenza che Logo permetteva il controllo di un unico agente, mentre un modello NetLogo può averne migliaia. Da StarLisp, invece, NetLogo eredita i molteplici agenti e la loro *concurrency*.

1.2.1 Linguaggio

Come Linguaggio NetLogo si evolve da Logo al quale aggiunge il concetto di agenti e di concurrency. In generale Logo è molto conoscuto per il concetto di **turtle** che ha introdotto. NetLogo generalizza questo permettendo il controllo di centinaia o migliaia di turtles che si muovono e interagiscono tra di loro.
Il mondo in cui i turtles si muovono è suddiviso in **patches** anche esse interamente programmabili, sia turtles che patches vengono chiamate collettivamente **agents**. Tutti gli agenti possono interagire tra di loro e eseguire istruzioni in modo concorrenziale. NetLogo include inoltre un terzo tipo di agente, l'**observer** il quale è unico. In generale l'observer è quello che impartisce ordini agli agenti.
Possono essere definite diverse "razze" (**breeds**) di turtles, ciascuna con variabili e behaviors caratteristici.
Una peculiarità che contraddistingue NetLogo dai suoi predecessori sono gli "agentsets", ovvero insiemi di agenti.

Capitolo 2

Sviluppo del progetto

In questo capitolo verrà descritto lo scopo di questo progetto, i requisiti necessari al suo raggiungimento e i dettagli della sua struttura interna che ne permette il funzionamento. In particolare andremo ad analizzare il class-diagram che mostra le relazioni tra le classi che lo costituiscono e discuteremo delle scelte implementative fatte.

2.1 Obiettivo e requisiti

L'obiettivo di questa tesi è creare un programma in linguaggio Java che sia in grado di ricevere in ingresso la struttura del modello descritta in linguaggio XML e di scrivere in modo automatico il codice NetLogo che possa eseguire la simulazione di interesse, rendendo, quindi, trasparente il processo di scrittura del codice.
La scelta del linguaggio XML è dettata dalla sua diffusione, con lo scopo di ridurre le conoscenze preliminari necessarie per l'utilizzo di questo strumento.

Per l'analisi del documento XML abbiamo scelto la libreria JDOM2[1] non inclusa in Java. Il Wold Wide Web Consortium, infatti, ha stabilito uno standard cross-platform e language-indipendent detto DOM (Document Object Model) per la rappresentazione di documenti strutturati (quindi XML, HTML, XHTML) come modello orientato agli oggetti. Il formato JDOM usato dalla libreria è una variazione dello standard disegnata appositamente per Java.

Per quanto riguarda la parte della scrittura del codice NetLogo, si ha che gran parte del codice che modella la simulazione rimane fissa al variare dei modelli descritti negli XML, quindi abbiamo pensato di mantenere questa in un semplice file di testo che viene letto e integrato con le informazioni contenute negli oggetti Java. Per le operazioni di lettura e scrittura su file abbiamo usato le classi BufferedReader e PrintWriter che permettono lettura e scrittura di intere linee di testo.

Lo strumento segue il seguente workflow:

- analisi del documento XML e costruzione di una rappresentazione della struttura orientata agli oggetti in JDOM attraverso l'omonima libreria
- costruzione degli oggetti Java per la rappresentazione della simulazione descritta nel modello JDOM
- visita degli oggetti Java e scrittura su file del codice NetLogo che eseguirà la simulazione

[1] http://www.jdom.org/

2.2 Documento XML

La struttura del documento XML prevista per il funzionamento del nostro strumento si attiene il più possibile allo standard di formato GraphML[2] (approvato dal W3C) in modo da evitare inconsistenze e incomprensioni, soprattutto per la parte in cui è descritta la topologia dell'ambiente, per la quale questo formato è pienamente adatto.
Il file XML è suddiviso in tre sezioni distinte:

- `Graph`
- `Behaviors`
- `System`

`Graph` rappresenta la topologia dell'ambiente in cui gli attori si muovono, ed è descritta sotto-forma di grafo con `edges` che rappresentano le strade e `nodes` che rappresentano gli incroci. Gli edges possono essere `directed` e `undirected`, per ognuno di essi vengono specificati peso e larghezza. I nodes invece possono essere di tre tipi: normal, entry o exit. Per tutti e tre i tipi vengono specificate coordinate spaziali e dimensioni fisiche dell'incrocio che esso rappresenta.
Per quanto riguarda i `Behavior` si è preferito distaccarci leggermente dallo standard, in modo da avere una struttura più comprensibile, usando gli specifici tag `<behavior \>`. Per ogni behavior viene indicata la tipologia, l'identificatore e la lista dei nodi di interesse.
La sezione `System` descrive lo stato iniziale (`state`) dell'ambiente. Per ogni nodo indica il numero di attori presenti e i loro behavior. In particolare nei nodi contrassegnati come entry o exit viene aggiunta una sezione denominata

[2] http://graphml.graphdrawing.org/

`parameters` in cui si specificano la frequenza di generazione o eliminazione degli attori e le percentuali relative ad ogni behavior. Per i nodi di ingresso viene anche indicato un limite superiore agli attori generabili.

2.3 Struttura Interna

L'unico scopo delle classi Java utilizzate è quello di rappresentare e conservare l'informazione raccolta dal documento XML, senza eseguire alcun tipo di manipolazione. Per questo motivo abbiamo cercato di mantenere la struttura delle classi il più semplice possibile, come mostrato in Figura 2.1.

Ogni behavior è caratterizzato da un identificatore, attribuitogli dal XML,

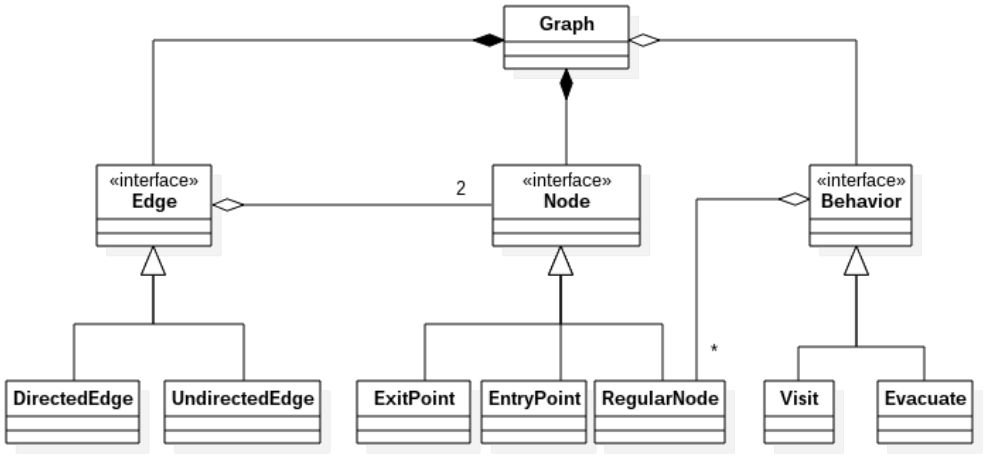

Figura 2.1: Class diagram della struttura del grafo

e da una lista di nodi regolari di interesse che devono essere visitati prima di permettere all'attore di puntare ad una uscita.

2.3.1 Parser XML e Builder

Come già accennato, abbiamo scelto il formato JDOM per la rappresentazione del file XML. La scelta di questo formato è stata dettata dalla sua struttura, che si adatta meglio ai nostri fini di sola lettura e non di manipolazione del documento.

Per la fase di costruzione delle classi abbiamo usato il pattern Builder, in

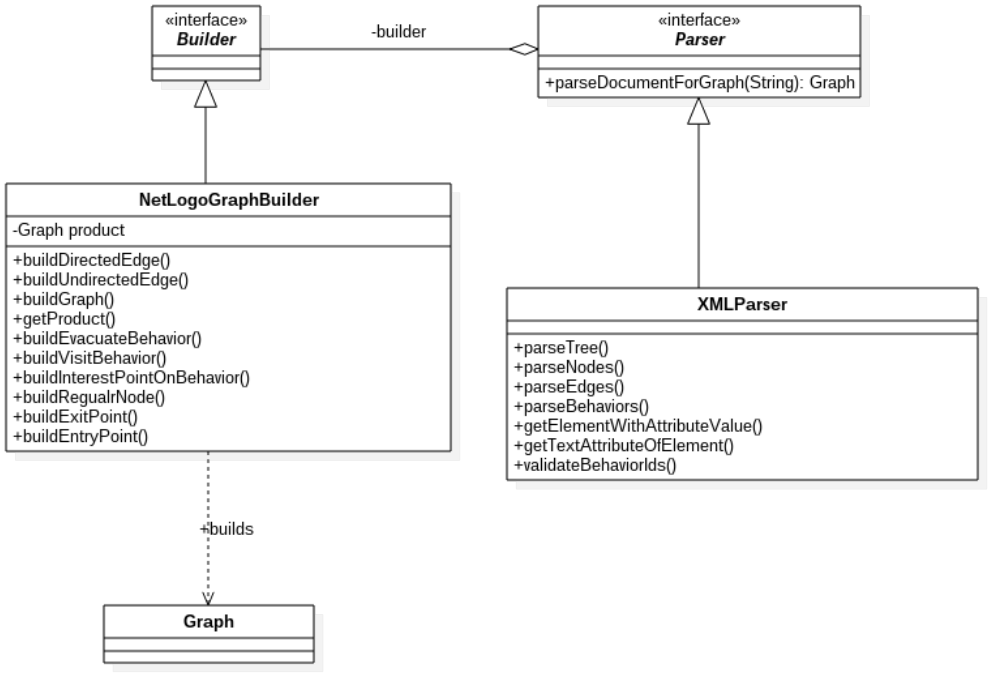

Figura 2.2: Class diagram di Parser e Builder

modo da mantenere il processo di costruzione della struttura interna dell'oggetto di tipo Grafo trasparente all'utente di questo strumento. In particolare l'interfaccia del Builder aiuta a mantenere l'algoritmo di interpretazione dell'XML separato dal processo di costruzione e rappresentazione del suo contenuto in classi Java.

Il pattern Builder, inoltre, porta il grande beneficio di migliorare la modula-

rizzazione derivante dall'incapsulamento di tutta la procedura di manipolazione della struttura interna dell'oggetto costruito, in questo modo il Client non ha necessità di conoscere la sua composizione interna.
Una conseguenza che rende il pattern ancora più adatto al nostro caso è il miglioramento del controllo sul processo di costruzione del prodotto. Al contrario di altri pattern creazionali, i quali costruiscono e restituiscono il prodotto in un'unica funzione, il Builder separa queste due operazioni mettendo a disposizione un'interfaccia più completa.
Grazie a questo noi siamo, quindi, in grado di esercitare un maggiore controllo sulla correttezza delle operazioni eseguite e delle informazioni inserite durante questa prima fase.
L'interfaccia Parser rende il progetto aperto ad estensioni future, come il supporto di linguaggi alternativi al XML.
Come si vede in Figura 2.2 il parser esercita il ruolo di Director per il builder. Il processo di analisi del documento e costruzione degli oggetti Java è descritto nel sequence diagram in Figura 2.3. XMLParser costruisce, attraverso la libreria JDOM2, il modello JDOM del documento, il quale rispecchia la struttura ad albero del documento. Quindi il parser effettua una visita del modello ad albero, legge le informazioni necessarie e impartisce i corretti comandi al builder.

2.3.2 Visitor

Per l'operazione di scrittura del codice NetLogo si ha la necessità di estrapolare dagli oggetti che costituiscono la struttura del grafo diverse informazioni. Al variare della classe concreta varieranno anche le informazioni specifiche da estrarre.

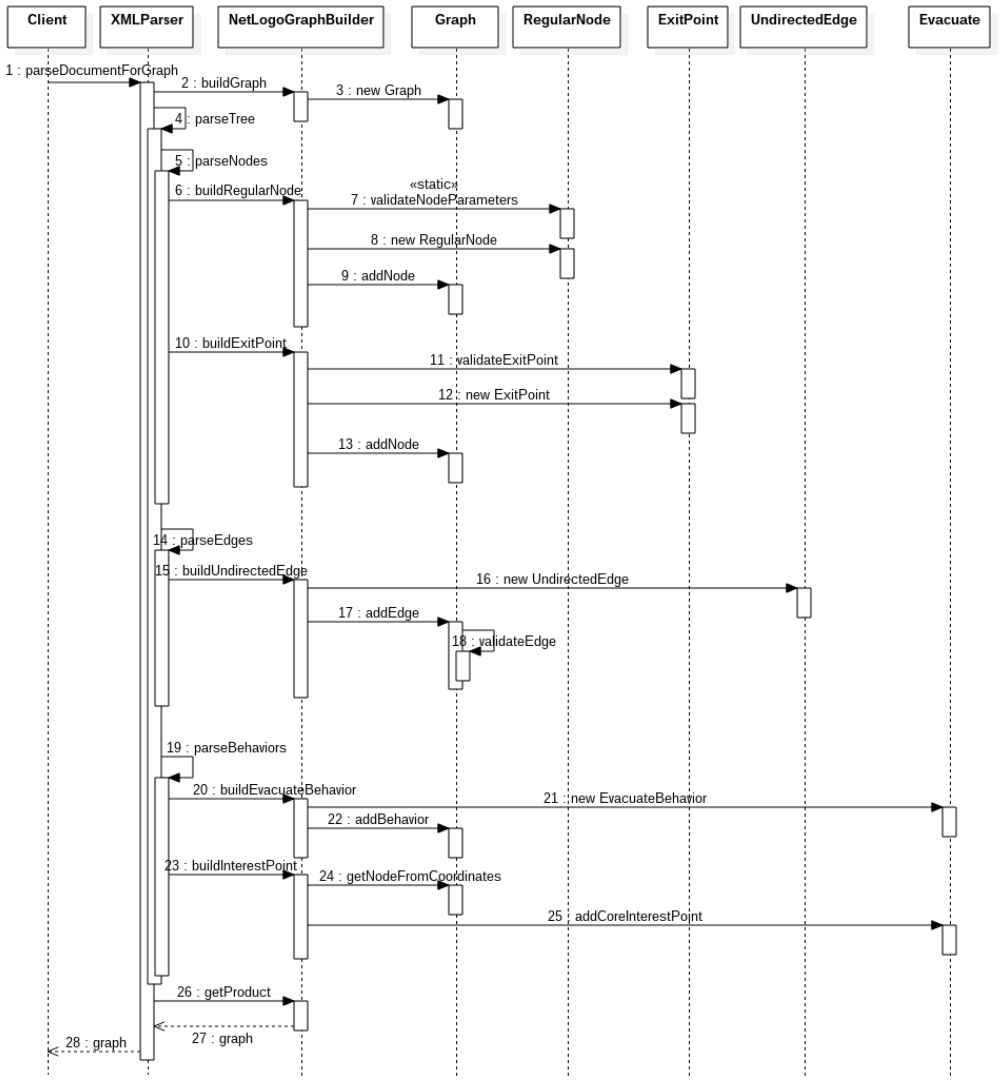

Figura 2.3: Sequence diagram della costruzione di un grafo da parte di XMLParser

Il grafo inoltre presenta una struttura di classi con interfacce diverse che devono, quindi, essere modificate per poter eseguire l'operazione necessaria.
In questo contesto, quindi, il pattern Visitor si rivela molto utile permettendo di eseguire operazioni specifiche al variare della classe concreta senza inquinare le diverse interfacce all'interno della struttura.
Un ulteriore vantaggio presentato dal Visitor è quello di concentrare le funzioni per la scrittura del codice NetLogo in classi specifiche rendendo il sistema più facile da manutenere ed eventualmente estendere.
La struttura dei modelli NetLogo finali è tale che gran parte del codice necessario rimane fissa al variare della simulazione, per questo motivo si ha che l'operazione di scrittura del codice NetLogo eseguita dai visitor è alternata a quella di lettura da appositi file di testo che contengono la parte fissa del modello. Queste parti fisse quindi vengono completate con le informazioni estrapolate dagli oggetti Java precedentemente messi in vita.
Abbiamo inoltre scelto di separare il modello NetLogo finale in due parti, mantenendo distinta la parte del modello che mette in vita l'ambiente da quella che controlla i movimenti degli attori e raccoglie le informazioni di interesse.

Si hanno quindi due classi visitor distinte per ognuno dei due modelli NetLogo da creare: NetLogoGraphVisitor e NetLogoBehaviorVisitor (Figura 2.4). Il primo effettua una visitita su nodi e archi del grafo per scrivere i comandi necessari a mettere in vita l'ambiente corretto. Il secondo, invece, effettua una visita sui behaviors in modo da definire i comportamenti che gli attori potranno avere nella simulazione e sui nodi per impostare lo stato iniziale del sistema.
In questo modo le interfacce delle classi del grafo, rimangono inalterate ma le operazioni effettuate variano al variare del visitor concreto, rendendolo un

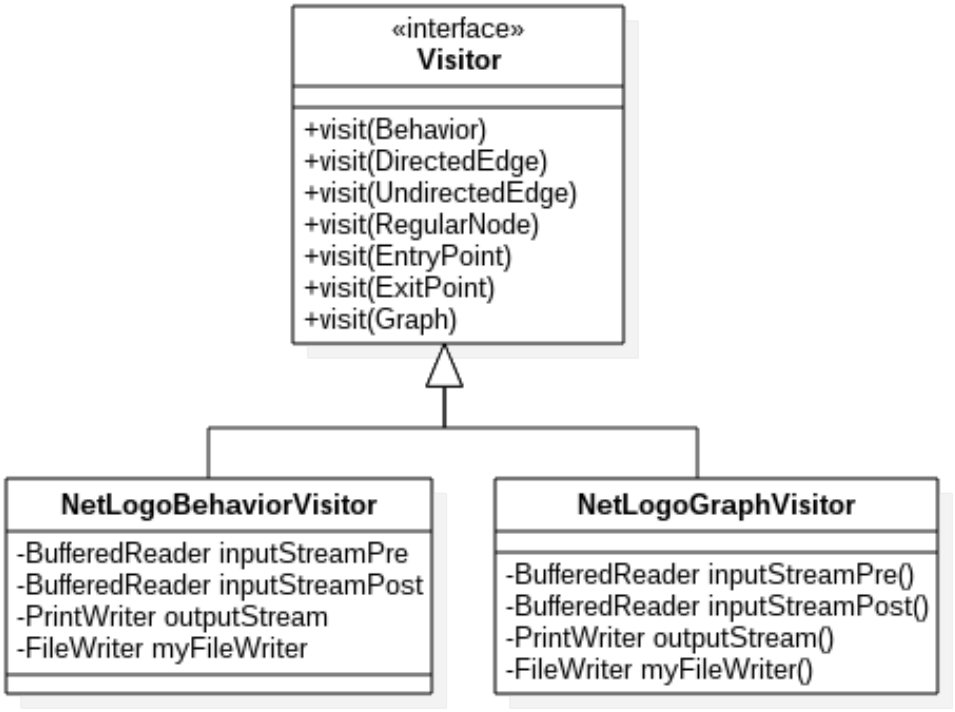

Figura 2.4: Class Diagram del Visitor

pattern ancora più adatto a questo specifico caso.

In Figura 2.5 è mostrato il funzionamento di un NetLogoGraphVisitor. L'oggetto finale ottenuto da questo visitor è un file eseguibile `.nlogo` che costruisce l'ambiente con la corretta topologia e esporta un file `.csv` necessario al secondo modello per l'acquisizione dell'ambiente.

NetLogoBehaviorVisitor, in modo analogo, eseguirà operazioni di lettura e scrittura alternate per generare un file eseguibile `.nlogo`.

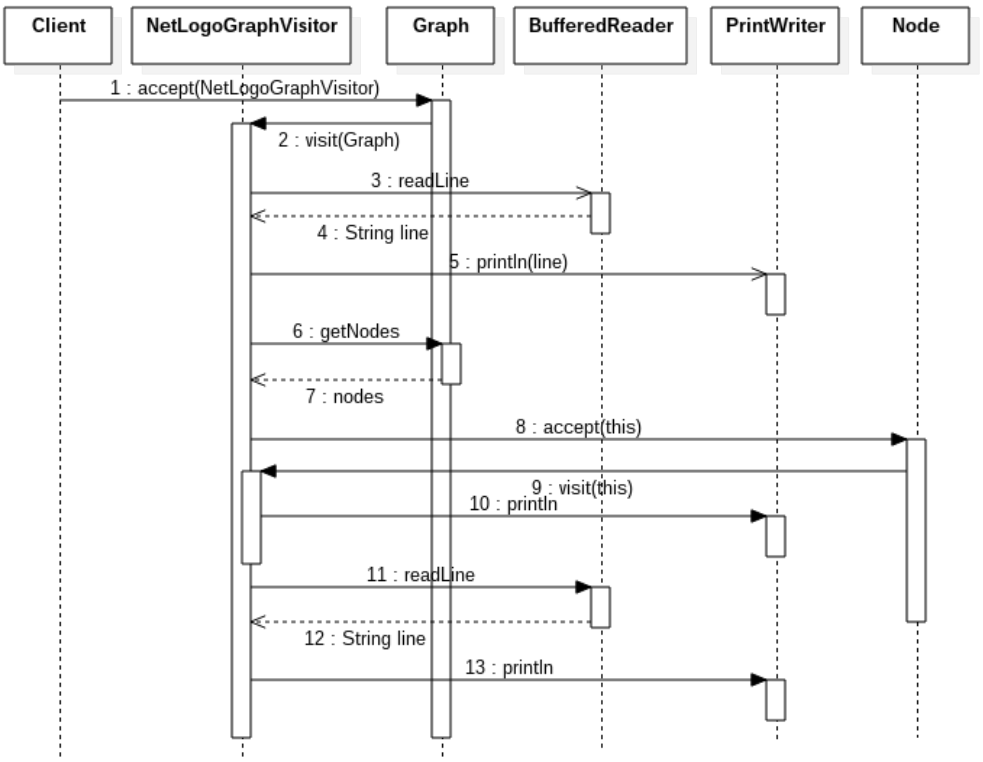

Figura 2.5: Sequence Diagram del NetLogoGraphVisitor su un grafo con un unico nodo

2.4 Modello NetLogo

Come già accennato in precedenza abbiamo scelto di dividere il modello NetLogo generato dal nostro programma in due parti con lo scopo di mantenere le funzioni necessarie per la costruzione dell'ambiente separate da quelle che modellano i movimenti degli attori e raccolgono le informazioni sulla simulazione eseguita. In questo modo il codice è più facile da gestire e manutenere.
Questa scelta porta, inoltre, il vantaggio di poter sfruttare il file csv generato dal primo modello eseguendo questo una sola volta, in questo modo si evitano operazioni ridondanti nel caso in cui si vogliano simulare le interazioni di diversi comportamenti nello stesso ambiente.

2.4.1 Ambiente

L'ambiente in cui gli attori si muoveranno è modellato sotto forma di grafo con specifiche tipologie di turtles dette `beacon` che fanno da nodi e rappresentano quindi gli incroci tra le vie percorribili.
Gli archi sono rappresentati da due tipologie diverse di turtles dette `street`, percorribili in entrambi i versi, e `directed-street`, percorribili solo in un verso.

Per i beacons di ingresso e di uscita vengono anche impostati i relativi parametri, ovvero tasso di generazione o eliminazione degli attori e percentuali relative ai behavior da attribuire ad essi. In particolare per gli entry points si imposta anche un limite massimo del numero di attori che possono essere generati.
Grazie a questi parametri il sistema è predisposto per lo studio di simulazioni in ambienti comunicanti, in cui quindi gli attori possono uscire da un

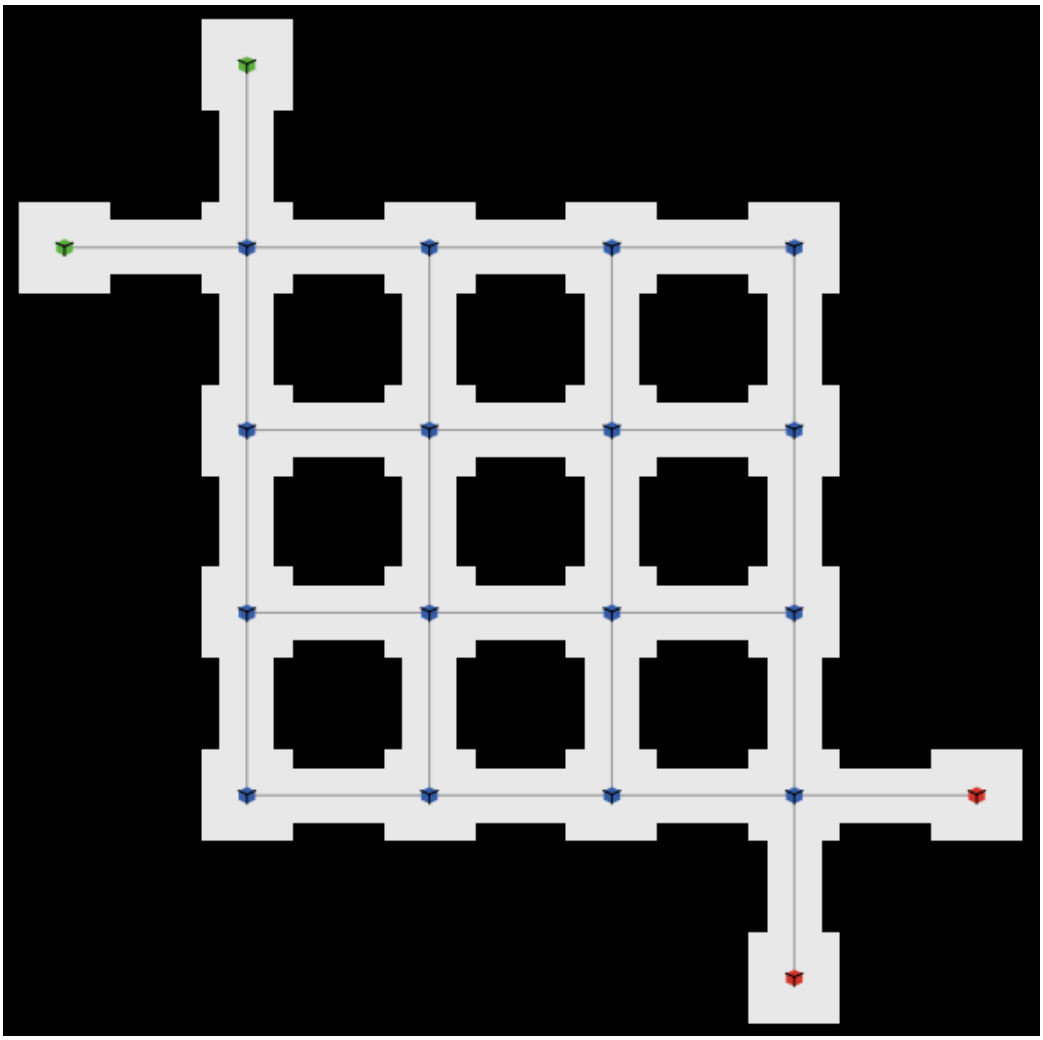

Figura 2.6: Esempio di ambiente generato. In questo caso i beacons di ingresso sono di colore verde e i beacons di uscita di colore rosso.

ambiente e entrare in un altro.

2.4.2 Behaviors e stato iniziale

Abbiamo scelto di caratterizzare i behaviors degli attori con un identificatore intero e una lista di nodi di interesse che l'attore deve raggiungere prima di uscire dall'ambiente.
Sono state inoltre inserite due possibili modalità di raggiungimento dei beacons di interesse: "`minDistance`" e "`orderedList`". Nella prima l'attore punta al nodo più vicino a quello precedentemente raggiunto tra quelli di interesse, nella seconda invece la lista viene rigidamente percorsa rispettandone l'ordine.
Lo stato iniziale del sistema è contenuto in una semplice lista `initial-state` che viene percorsa da una apposita funzione per la corretta generazione degli attori sui vari nodi.

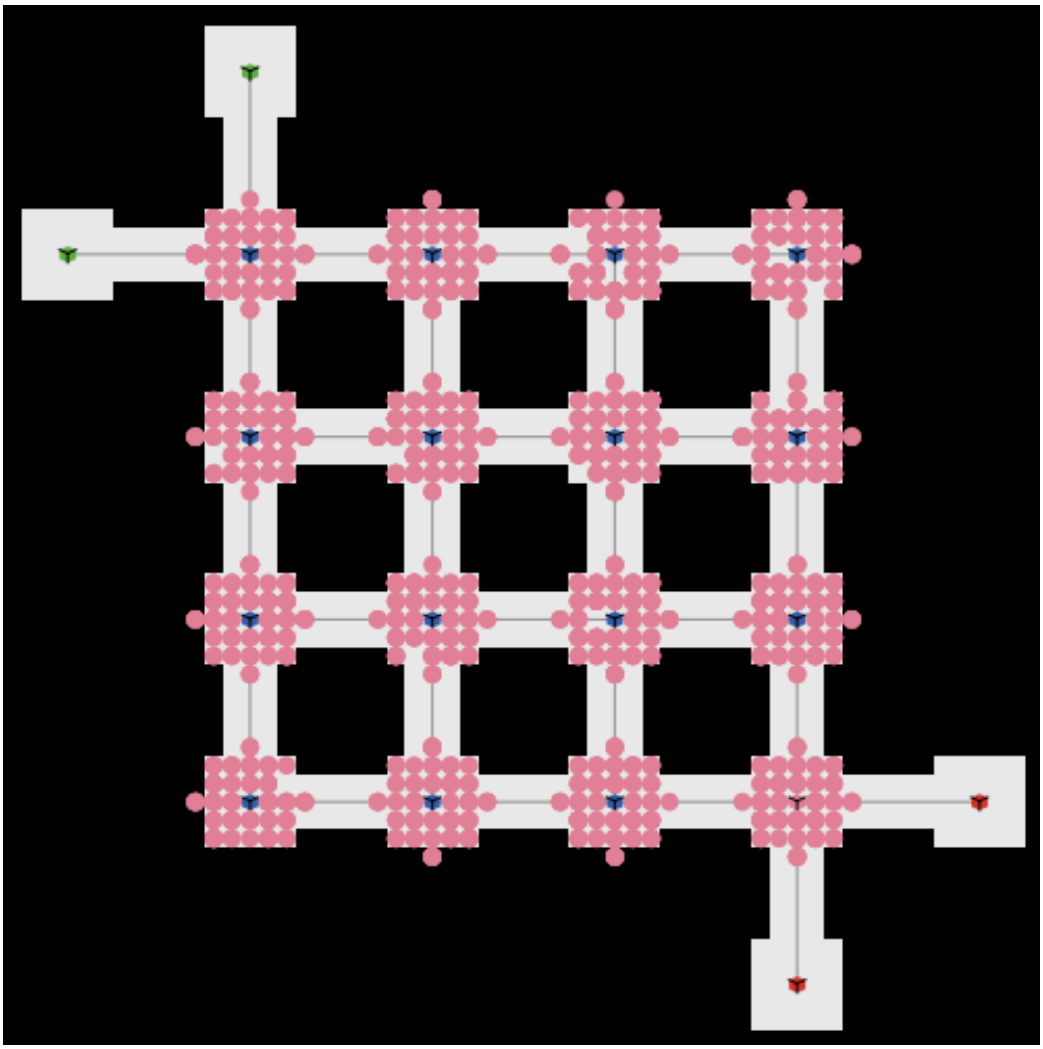

Figura 2.7: Esempio di ambiente popolato con attori nello stato iniziale. Gli attori hanno assunto il colore del beacon che hanno come obiettivo, in questo caso abbiamo un unico behavior che ha come interest point il beacon di colore rosa.

Bibliografia