Università degli Studi di Salerno

Dipartimento di Informatica

Tesi di Laurea di I livello in
Informatica

Simulazione basata su agenti con JULIA

Relatore
Prof. Vittorio Scarano

Candidato
Nicola Amoriello
0512104742

Anno Accademico 2018-2019

A Filomena e Lucia

Ringraziamenti

Desidero ringraziare...
il professore Vittorio Scarano per avermi permesso di poter lavorare a questo progetto all'interno dell'ISISLab...
Alessia Antelmi per avermi aiutato nei momenti di difficoltà incontrati durante lo sviluppo di questa tesi...
tutti i miei amici per il supporto costante in questo percorso durato tre anni...
i ragazzi del team DeltaCappello per aver condiviso con me questa fantastica avventura...
la mia famiglia, sia gli zii e la nonna, che si sono sempre interessati al mio percorso di studi, sia ai miei genitori e fratelli, che mi hanno permesso di studiare e mi hanno sempre sostenuto...

Questa tesi è stata sviluppata in

Abstract

In questo lavoro è stata trattata la tematica della simulazione basata su agenti. Questa, anche conosciuta come ABM (Agent Based Simulation), è una classe di modelli computazionali finalizzati alla simulazione al computer di azioni e interazioni di agenti autonomi al fine di valutare i loro effetti sul sistema nel suo complesso.

Una Agent-Based Simulation è caratterizzata da tre componenti fondamentali: i) agenti, entità autonome e con proprie caratteristiche e comportamenti; ii) ambiente, detto anche campo o field, nel quale gli agenti interagiscono tra loro; iii) un insieme di regole che definisce le azioni e le reazioni che si manifestano durante il corso della simulazione.

Le applicazioni di ABM sono molteplici e particolarmente utili: solitamente sono utilizzate per studiare fenomeni troppo complessi per essere riprodotti fedelmente e/o dispendiosi in termini di costo e impegno. Ad esempio, lo studio dei comportamenti di una popolazione delimitata in una certa area (città, regione, nazione,...); la simulazione di reazioni tra molecole in una soluzione chimica o anche la gestione di sistemi distribuiti che possono essere rappresentati come nodi che richiedono e offrono servizi tra loro.

E' in questo contesto che nasce *ABM.jl*, un engine open source che mette a disposizione gli strumenti fondamentali per poter implementare una simulazione agent based. Tale tool è stato completamente sviluppato in *Julia*, un nuovo promettente linguaggio di programmazione, nato dalla mente di alcuni programmatori e studiosi del *MIT* di Boston, che promette prestazioni molto elevate (comparabili quasi a quelle del *C*) e al contempo codice snello e veloce da implementare.

In particolare, primo obiettivo del mio lavoro di tesi, è stato implementare due simulazioni ben note per valutare le performance di *ABM.jl*: *AntsForaging* e *Boids*.

La prima si occupa di rappresentare il comportamento di una colonia di formiche occupata nella ricerca di cibo; la seconda mostra il comportamento di un gruppo di volatili in una sezione di cielo, i quali seguono traiettorie di volo differenti in base alla presenza di simili nelle vicinanze. L'implementazione è stata effettuata sfruttando le peculiarità del linguaggio *Julia* e adottando un paradigma object-oriented. Di ogni modulo di simulazione ne è stata progettata l'architettura secondo i principi di leggibilità e manutenibilità del codice. Ogni simulazione, infatti, è strutturata in due componenti principali, ognuna delle quali si occupa di modellare: i) la logica dell'agente e ii) la definizione dell'ambiente. Una terza componente, interna alla simulazione, si occuperà, invece, della fase di benchmark della simulazione stessa.

Secondo obiettivo del mio lavoro è stato il design e l'implementazione di una interfaccia grafica che permettesse la visualizzazione, in tempo reale, dello stato della simulazione.

Test di benchmark sono stati effettuati considerando i principali competitor in ambito di simulation engine, tra cui *MASON*, *NetLogo* e *RustAB*. Nonostante tali test abbiano evidenziato come *ABM.jl* sia uno strumento che in termini di prestazioni ha un ampio margine di miglioramento, questa libreria rappresenta un'importante opportunità per i ricercatori che usano abitualmente *Julia* come linguaggio di programmazione, in quanto questi potranno avere la possibilità di lavorare su simulazioni basate ad agenti senza dover necessariamente imparare un diverso linguaggio.

QUESTA PAGINA È STATA LASCIATA INTENZIONALMENTE IN BIANCO.

Indice

1 Introduzione **1**

1.1 Contesto applicativo 1

1.1.1 ABM 2

1.1.2 Julia 3

1.2 Motivazioni ed Obiettivi 5

1.3 Struttura della tesi 6

2 ABM.jl **7**

2.1 Architettura 8

2.2 GUI 19

3 Simulazione Boids **22**

3.1 La logica 22

3.2 L'architettura 23

3.3 L'implementazione 24

3.3.1 L'agente 24

3.3.2 Script per il run 26

3.3.3 Esecuzione grafica 28

3.4 Prestazioni 30

3.4.1 Benchmark 30

3.4.2 Risultati 31

4 Simulazione Ant Foraging **34**

4.1 La logica 34

4.2 L'architettura 36

4.3 L'implementazione 36
4.3.1 L'agente 36
4.3.2 Script per il run 38
4.3.3 Esecuzione grafica 41
4.4 Prestazioni 45
4.4.1 Benchmark 45
4.4.2 Risultati 46

5 Conclusioni **49**

Bibliografia **51**

Capitolo 1

Introduzione

1.1 Contesto applicativo

L'utilizzo di simulazioni basate su agenti, anche note come "Agent Based Model" (ABM), è andata col tempo ad aumentare, passando da un primo periodo, negli anni '90, dove era un concetto conosciuto ed utilizzato solo nell'ambito informatico, per arrivare ai giorni nostri, nei quali altre branche di studi non direttamente connesse alle scienze informatiche, come la biologia e le scienze sociali, utilizzano quotidianamente questo modello computazionale.

La motivazione dell'utilizzo sempre maggiore delle simulazioni basate su agenti va ricercato nella crescente complessità delle reti con cui può essere modellato un sistema. I costi organizzativi, temporali ed economici per lo studio analitico di tali sistemi risulterebbe troppo elevato; inoltre, potrebbe rivelarsi rischiosa la stessa sperimentazione (basti pensare alla simulazione della diffusione di un virus).

Questo strumento permette di fatto alle comunità scientifiche di studiare e analizzare il comportamento dinamico di un sistema reale tramite il comportamento di un sistema artificiale. L'utilizzo di questo tipo di simulazioni spazia in diversi ambiti scientifici, tecnologici e sociali, dalla biologia alla chimica, dall'economia all'intelligenza artificiale, dalle scienze sociali all'ingegneria dei sistemi.

1.1.1 ABM

I modelli basati sull'agente (in sigla, ABM, acronimo di agent-based model) sono una classe di modelli computazionali finalizzati alla simulazione al computer di azioni e interazioni di agenti autonomi (tanto individuali quanto collettivi, come organizzazioni e gruppi) al fine di valutare i loro effetti sul sistema nel suo complesso [1].

Un tipico modello agent based ha solitamente tre elementi fondamentali che lo caratterizzano:

- **Gli agenti:** sono le entità che costituiscono la simulazione. Ognuno è identificabile come individuo a sé stante, ma è un'insieme di essi a formare un sistema multi-agente. Dispongono di un set di proprietà e comportamenti che li caratterizza e che permetto loro di interfacciarsi con il resto dell'ambiente di simulazione.
- **Il campo (field):** è l'ambiente, cioè lo spazio d'interazione tra agenti e altri agenti e/o agenti e oggetti del campo.
- **Le regole:** definiscono come e quando vengono svolte le interazioni tra gli agenti.

Uno dei paradigmi alla base dell'approccio basato sulla modellizzazione ad agente è costituito dal fatto che solo le informazioni locali sono disponibili agli agenti.

Il sistema basato su gli agenti è un sistema decentralizzato, non esiste un'entità centrale che diffonde globalmente le medesime informazioni a tutti gli agenti o ne controlla il comportamento con l'obiettivo di aumentare la performance del sistema nel suo complesso.

Gli agenti interagiscono tra loro, ma non interagiscono contemporaneamente tutti insieme. L'interazione tra di loro avviene, tipicamente, con un sotto-gruppo di simili entità prossime tra loro (neighbours); in modo analogo anche l'interazione tra questi e l'ambiente è localizzata: non è detto che gli agenti interagiscano con ogni parte dell'ambiente.

Quindi, le informazioni ottenute dagli agenti sono ottenute dal sotto-gruppo di stessi con cui interagiscono e dalla porzione di ambiente in cui si trovano. Se il modello non è statico, il sotto-gruppo e l'ambiente con cui un agente interagisce varia nel tempo [2].

L'agente di un ABM interagisce con altri suoi simili e con l'ambiente. Questo riveste un ruolo fondamentale nella simulazione e viene rappresentato in modo da riprodurre il contesto nel quale si verifica l'evento reale.

Solitamente mantiene le informazioni posizionali degli agenti nel tempo, così da poter tenere traccia di tutti gli spostamenti effettuati durante l'esecuzione della simulazione.

Possono essere presenti anche oggetti nel campo. Rappresentano specifiche zone dell'ambiente che hanno caratteristiche diverse tra di loro a seconda della propria locazione.

Gli agenti hanno la possibilità di interagire con essi in modo da manipolarli, produrli, distruggerli o essere influenzati da essi e cambiare comportamento.

1.1.2 Julia

Julia è un linguaggio di programmazione, multi-paradigma, open source pubblicato a partire dal 2012 (anche se le basi furono poste nel 2009) da un gruppo di programmatori impegnati professionalmente nel campo del calcolo numerico pensato con lo scopo di fornire uno strumento ad alte prestazioni dedicato in maniera particolare al mondo della programmazione scientifica [3].

E' stato scritto in *C++* e *Scheme* e si basa su un compilatore JIT(Just in Time). Il codice sorgente viene inizialmente trasformato in una rappresentazione intermedia portabile e ottimizzabile, detta bytecode. Successivamente il codice bytecode viene installato sul sistema di destinazione. Quando il codice viene eseguito, il compilatore dell'ambiente di esecuzione lo traduce in codice macchina nativo. La traduzione in codice macchina può avvenire per file o per funzione: le funzioni possono essere compilate solo quando stanno per essere eseguite, da qui il nome just-in-time, ovvero "appena in tempo". [4]

Fin dalla sua nascita il linguaggio *Julia* ha voluto puntare sulle prestazioni. Gli sviluppatori si sono infatti impegnati per portare le performance a livelli

equiparabili a quelli di linguaggi famosi per le loro alte prestazioni come *C* e *Fortran*.

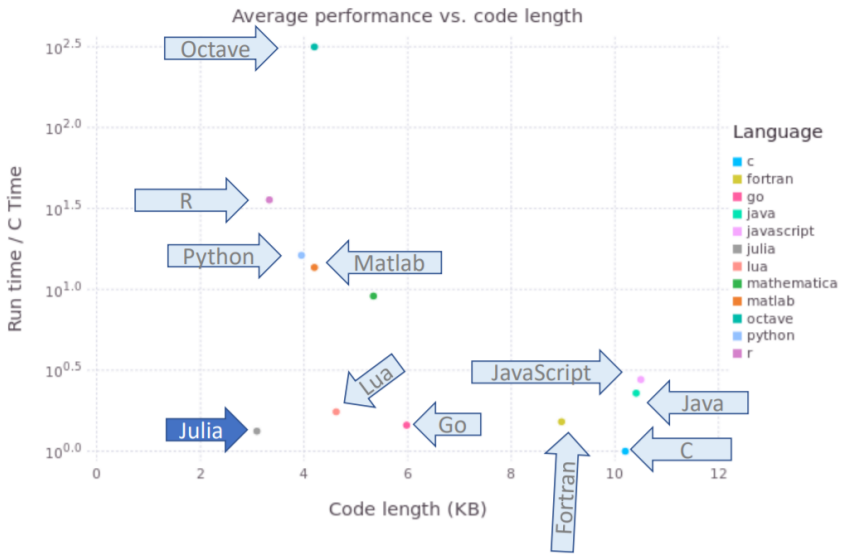

Figura 1.1: Prestazioni di *Julia* e altri linguaggi a confronto con quelle del *C*.

Come viene esposto sul sito ufficiale [5] le principali caratteristiche di *Julia* sono:

- **Velocità**
  E' stato progettato sin dall'inizio per prestazioni elevate. I programmi *Julia* vengono compilati in un efficiente codice nativo per più piattaforme tramite *LLVM*.

- **Dinamicità**
  Utilizza la tipizzazione dinamica, sembra un linguaggio di scripting e ha un buon supporto per l'uso interattivo.

- **Tipizzazione opzionale**
  Grazie alla tipizzazione dinamica non è presente l'obbligo di specificare il tipo delle variabili quando queste vengono dichiarate, ma indicare il tipo spesso può essere utile per chiarire e/o consolidare i programmi, oltre a

ricevere maggiori performance evitando la fase di scelta dinamica fatta dal compilatore.

- **Multi-paradigma**
  Permette facilmente di utilizzare pattern tipici della programmazione object-oriented e della programmazione funzionale

- **Facile da usare**
  Ha una sintassi di alto livello che lo rende un linguaggio accessibile per i programmatori di qualsiasi livello o esperienza.

- **Open source**
  Julia è gratuito per tutti e il codice sorgente completo è pubblicamente visibile su *GitHub*. Inoltre è supportato da una comunity costantemente attiva.

1.2 Motivazioni ed Obiettivi

Il crescente utilizzo e la conseguente affermata utilità delle simulazioni agent based ha suscitato negli scienziati la richiesta di strumenti che permettano uno sviluppo e un'esecuzione di queste in tempi sempre più ristretti e con fasi d'implementazioni maggiormente fruibili a chi non è del settore informatico.

Poiché con il passare del tempo gli studi si sono indirizzati su fenomeni di portata sempre più vasta, si è sentito il bisogno di fornire tool che riescano a sostenere una così grande mole di lavoro, senza perdere efficienza e mantenendo comunque il dispendio di risorse al di sotto di una certa soglia.

L'obiettivo che è stato prefissato all'inizio di questo lavoro di tesi è quello di implementare due simulazioni ben note per valutare le performance di *ABM.jl*: *AntsForaging* e *Boids*; in grado di mostrare le potenzialità dell'engine e del linguaggio.

Scritti in *Julia*, così come il motore di simulazione *ABM.jl*, il quale all'inizio del lavoro risulta sprovvisto di moduli agent based funzionanti, questi sono stati creati per dare al mondo scientifico simulazioni implementate con un

nuovo tool, le quali possano offrire maggiori vantaggi rispetto a ciò che è già presente.

1.3 Struttura della tesi

Nei prossimi capitoli saranno proposti in dettaglio tutti gli argomenti ai quali si è accennato. Questo lavoro di tesi si compone di cinque capitoli, incluso il presente. In dettaglio la suddivisione è la seguente:

- **Capitolo 2: ABM.jl**

  Illustra nello specifico l'engine *ABM.jl*, che è stato utilizzato per sviluppare le simulazioni basate su agenti presentate in questo lavoro. In dettaglio, descrive l'architettura del progetto e si sofferma sulle componenti chiavi di questo.

- **Capitolo 3: Simulazione Boids**

  Illustra *Boids.jl*, spiegando la natura della simulazione, l'architettura adottata, le scelte fatte in fase di implementazione e concludendo con benchmark delle prestazioni.

- **Capitolo 4: Simulazione Ant Foraging**

  Illustra la simulazione *AntForaging.jl*, spiegando il concetto che c'è alla base, l'architettura adottata, le scelte fatte in fase di implementazione e concludendo con una vista sulle prestazioni di questa messa a confronto con altre sviluppate partendo da tool differenti.

- **Capitolo 5: Conclusioni**

  In questo capitolo vi sono alcune osservazioni e critiche sullo stato corrente dell'engine di simulazione *ABM.jl* e vengono presentati spunti da cui partire per lavori futuri.

Capitolo 2

ABM.jl

In questo capitolo viene illustrato il tool utilizzato per sviluppare le simulazioni che sono state oggetto di studio di questo lavoro.

ABM.jl è un motore di simulazione basato su agenti scritto nel linguaggio di programmazione *Julia* [6].

Si tratta di un tool che mette a disposizione gli elementi fondamentali presenti in qualsiasi simulazione agent based, così da permettere allo sviluppatore di concentrarsi esclusivamente sulla logica di business, riducendo considerevolmente il tempo per sviluppare la logica sottostante a quella applicativa.

E' inoltre free e open-source, disponibili su *GitHub* al link che segue: `https://github.com/spagnuolocarmine/ABM.jl`.

2.1 Architettura

In questa sezione viene illustrata la struttura dell'engine con la relativa suddivisione in "pacchetti" e la descrizione delle principali strutture e funzioni.

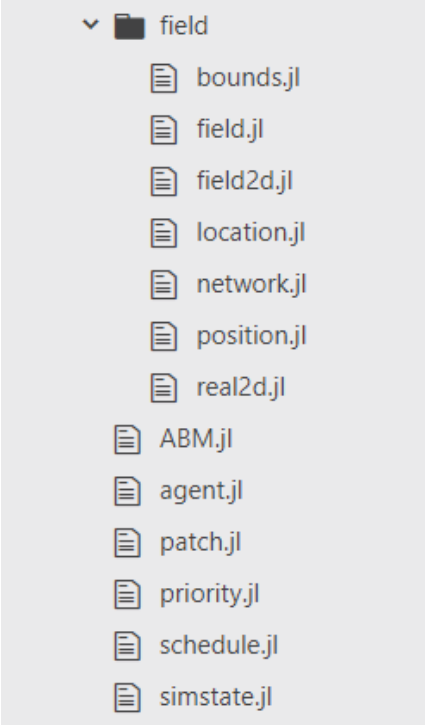

Figura 2.1: Vista ad albero della struttura del progetto.

Segue la descrizione della struttura:

- **ABM.jl**
  E' il manifesto del progetto, in *Julia* viene chiamato "Module". Presenta una lista di moduli esterni da importare (Codice 2.1), una di codice da includere al momento dell'attivazione della directory (Codice 2.2) e una di strutture e funzioni da esportare, così da renderle disponibile anche al di fuori del loro ambiente (Codice 2.3).

Listing 2.1: Moduli importati in ABM.jl.

```
11 using Distributed
```

```
12 using BenchmarkTools
13 using Compat
14 using Random
15 using Distributions
16 using UUIDs
17 using DataStructures
18 using LightGraphs
19 using DistributedArrays
```

Listing 2.2: Codice incluso in ABM.jl al momento dell'attivazione.

```
11 include("priority.jl")
12 include("agent.jl")
13 include("patch.jl")
14 include("field/position.jl")
15 include("field/location.jl")
16
17 include("field/real2d.jl")
18 include("field/field.jl")
19 include("field/field2d.jl")
20 include("field/bounds.jl")
21 include("simstate.jl")
22 include("schedule.jl")
```

Listing 2.3: Strutture e funzioni da esportare.

```
11 export Agent
12 export Patch
13
14 export Position
15 export Real2D,
16 toroidal, distance
17
18 export Location
19 export Field
```

```
20 export Field2D,
21 setObjectLocation!, tTransform,
22 getObjectLocation, getObjectsAtLocation,
      getNeighborsWithinDistance, getAllObjects,
23 numObjectsAtLocation,
24 remove!, clean!
25 export Network
26 export Bounds,
27 checkBoundCircle
28
29 export Priority
30 export Schedule,
31 scheduleOnce!, scheduleRepeating!,
32 stop!,step!
33
34 export SimState,
35 addfield!, update!, random
36
37 export simulate!
```

Inoltre è presente la funzione `simulate!()` (Codice 2.4) che ha lo scopo di avviare la simulazione.

Questa prende in input 2 parametri: un oggetto di tipo "Schedule", il quale rappresenta una coda a priorità contenente gli eventi da eseguire durante la simulazione e un intero che indica il numero di step da compiere prima che la simulazioni si arresti.

Listing 2.4: Funzione `simulate!`

```
69 function simulate!(schedule::Schedule, nsteps::Int64)
70  while schedule.steps < nsteps
71   step!(schedule)
72  end
73 end
```

- **agent.jl**
  Presenta la struttura *Agent* che permette di istanziare agenti usando una logica simile a quella dei linguaggi object-oriented (Codice 2.5).

Listing 2.5: struttura *Agent* e relativo costruttore

```
5  struct Agent{T}
6   step::Function
7   state::T
8   id::UUID
9  end
10 Agent(step::Function, data::T, rng::AbstractRNG=
      Random.GLOBAL_RNG) where T = Agent{T}(step, data,
      UUIDs.uuid4(rng))
```

  Le proprietà che lo caratterizzano sono le seguenti:

  - *step:* la funzione che l'agente svolge a ogni passo della simulazione.
  - *state:* la struttura di tipo T (generico, di qualsiasi tipo) che racchiude le peculiarità dell'agente.
  - *id:* è semplicemente un *UUID (Universally unique identifier)*, cioè un identificativo usato nelle infrastrutture software, standardizzato dalla *Open Software Foundation* (*OSF*) [9].

- **patch.jl**
  In questo file è presente la struttura *Patch* che rappresenta un oggetto nell'ambiente (field) della simulazione. Simile ad *Agent* si differenzia per non avere nessun tipo di comportamento: presenta solo le proprietà caratteristiche del singolo oggetto.

Listing 2.6: struttura *Patch* e relativo costruttore

```
5  mutable struct Patch{T}
6   state::T
7   id::UUID
8   Patch(data::T, rng::AbstractRNG=Random.GLOBAL_RNG)
       where T = new{T}(data,UUIDs.uuid4(rng))
```

```
9 end
```

Così come *Agent*, presenta le proprietà *state* e *id* che sono del tutto identiche a quelle della struttura precedentemente illustrata.

- **priority.jl**
  All'interno di questo file è definito il tipo *Priority* (Codice 2.7).

Listing 2.7: struttura *Priority* e relativo costruttore

```
3 struct Priority<:Real
4  time::Float64
5  priority::UInt
6 end
7 Priority(time::Float64) = Priority(time, rand(UInt))
```

Priority rappresenta un evento a priorità, che viene gerarchizzata seguendo due caratteristiche: il tempo di durata dell'evento e un unsigned integer che indica il grado d'importanza (più piccolo è il numero associato e maggiore è sua la priorità)

Come mostrato nel listato 2.7, il costruttore prende come paramentro solo la durata dell'evento (*time*), mentre il grado di priorità viene generato casualmente tramite l'uso della funzione `rand()`.

- **schedule.jl**
  Rappresenta una coda di scheduling in cui è possibile pianificare eventi in modo che questi siano disponibi in futuro per essere eseguiti.

Listing 2.8: struttura *Schedule* e relativo costruttore

```
23 mutable struct Schedule
24  simstate::SimState
25  events::PriorityQueue{Agent,Priority}
26  endevents::Dict{Agent,Agent}
27  steps::Int64
28  time::Float64
```

```
29  Schedule(simstate::SimState) = new(simstate,
        PriorityQueue{Agent,Priority}(), Dict{Agent,Agent
        }(), 0, 0.0)
30  end
```

La struttura *Schedule* racchiude tutte le informazioni essenziali per eseguire una corretta simulazione.

Tiene traccia dello stato della simulazione per mezzo del campo *simstate*; degli eventi da schedulare tramite una coda a priorità (*events*), dove a ogni agente viene accoppiata una priorità di tipo *Priority*; degli eventi terminati grazie all'ausilio di una struttura dati *endevents* di tipo $Dict\{K,V\}$, ovvero un dizionario, in questo caso con i parametri K e V entrambi di tipo *Agent*.

Inoltre vengono memorizzati i passi effettuati durante la simulazione (nel campo *steps*) e il tempo trascorso dall'avvio di questa (nel campo *time*).

Questa viene istanziata richiamando il rispettivo costruttore mostrato alla riga 29 del listato 2.8. Viene passata come parametro esclusivamente la simulazione di tipo *Simstate*; il dizionario e la coda a priorità vengono inizializzate con i rispettivi costruttori vuoti e ovviamente *steps* e *time* sono settati di default entrambi a zero.

In questo modulo sono poi presenti le funzioni che regolano la logica di scheduling.

Tra queste c'è la funzione `step!()` (Codice 2.9), che si occupa di estrarre dalla coda degli eventi quello con priorità maggiore e di eseguirlo.

Listing 2.9: funzione `step!`

```
67  function step!(schedule::Schedule)
68   schedule.steps+=1
69   if length(schedule.events) == 0 return end
70   schedule.time = peek(schedule.events).second.time
71   ctime = schedule.time
72   cevents = Vector{Pair{Agent,Priority}}()
```

```
73  while true
74   if(isempty(schedule.events)) break end
75   @inbounds event = peek(schedule.events)
76   if(event.second.time > schedule.time) break end
77   ctime = event.second.time
78   @inbounds dequeue!(schedule.events)
79   push!(cevents,event)
80  end
81
82  for e in cevents
83   ce = e.first
84   ce.step(schedule.simstate,e.first)
85   if(!haskey(schedule.endevents,e.first))
86    enqueue!(schedule.events, ce, Priority(ctime+1.0,
        e.second.priority))
87   end
88  end
89  schedule.endevents = Dict{Agent,Agent}()
90 end
```

Gli eventi vengono clonati prima di eseguire ogni step della simulazione. Quando un agente viene schedulato non ci sono cambiamenti nella sua memoria, fino al prossimo step di simulazione.

Ciò garantisce che la memoria dell'agente rimanga invariata durante la fase di simulazione, in modo che gli agenti possano vedere lo stato degli altri coerentemente con il tempo di simulazione corrente.

Solo dopo che sia terminata l'esecuzione della funzione *step* (definita nella struttura dell'agente, da non confondere confondere con questa appena presa in esame), lo stato della struttura *Schedule* viene correttamente aggiornato.

- **simstate.jl**
  Presenta la struttura *Simstate* (Codice 2.10) e la funzione `addField!()` (Codice 2.11).

Listing 2.10: struttura *Simstate* e relativo costruttore

```
4 mutable struct SimState
5  fields::Vector{Field}
6 end
7 SimState() = SimState(Vector{Field}())
```

E' la struttura che tiente traccia dello stato corrente di simulazione.

Presenta il solo campo *fields*, il quale è un vettore di *Field*, ovvero l'oggetto che rappresenta l'ambiente del modello.

Questo è inizializzato come vettore vuoto dal costruttore, ma viene in seguito aggiornato per mezzo della funzione apposita illustrata nel listato seguente.

Listing 2.11: funzione `addField!`

```
10 function addfield!(simstate::SimState,field::Field)
11  push!(simstate.fields,field)
12 end
```

- **field**

  E' la directory che contiene la logica dell'ambiente nel quale la simulazione ha luogo.

  Le componenti principali di questo "package" sono le seguenti:

  - **bounds.jl**

    Il file *Bounds* presenta una struttura utile per la memorizzazione delle misure di un qualsiasi oggetto presente nell'ambiente di simulazione.

    Oltre alla struttura, mostrata nel listato seguente (Codice 2.12), il modulo contiene una funzione per il confronto di due variabili di questo tipo e una per controllare se le misure dell'oggetto preso in esame rientrano in un cerchio di raggio passato come parametro (*checkBoundCircle*)

Listing 2.12: struttura *Bounds* e relativo costruttore

```
67 struct Bounds{T<:Real, P<:Position, F<:Field}
68  nw::P
69  ne::P
70  sw::P
71  se::P
72  f::F
73  function Bounds(f::Field2D{T}, bagID::Real2D{Int
       }) where {T<:Real}
74   nw = Real2D{T}(bagID.x * f.discretization,
        bagID.y * f.discretization)
75   ne = Real2D{T}(nw.x, min(nw.y + f.
        discretization, f.height))
76   sw = Real2D{T}(min(nw.x+f.discretization, f.
        width), nw.y)
77   se = Real2D{T}(sw.x, ne.y)
78   bounds = new{T,Real2D{T},Field2D{T}}(nw, ne, sw
        , se, f)
79  return bounds
80  end
81 end
```

– **field.jl**

Field (Codice 2.13) è un tipo astratto che rappresenta il concetto di campo, l'ambiente dove vengono eseguiti gli eventi della simulazione.

Listing 2.13: definizione del tipo astratto *Field*

```
4 abstract type Field end
```

I tipi astratti non possono essere istanziati e servono solo come nodi nel grafico dei tipi, descrivendo in tal modo insiemi di tipi concreti correlati: quei tipi concreti che sono i loro discendenti [10].

In questo caso *Field* è il nodo padre del tipo concreto *Field2D*, esposto nel punto seguente.

– **field2D.jl**

Come annunciato poco sopra, è in questo file che viene rappresentata la vera e propria logica del campo di simulazione. Presenta la struttura che lo identifica e le funzioni per aggiugere, rimuovere, posizionare agenti e/o oggetti al suo interno e, in generale, per gestire i cambiamenti di stato di questo durante l'arco dell'intera simulazione.

Listing 2.14: struttura *Field2D* e relativo costruttore

```
5  mutable struct Field2D{T<:Real,D<:Real} <: Field
6   width::T
7   height::T
8   discretization::D
9   toroidal::Bool
10  f::Dict{Real2D{Int}, Dict{Union{Agent,Patch},
       Location}}
11  f0::Dict{Union{Agent,Patch},Real2D{Int}}
12
13  Field2D(width::T, height::T, discretization::D,
       toroidal::Bool) where {T<:Real,D<:Real} =
14          new{T,D}(width, height, discretization,
               toroidal, Dict{Real2D{Int},Dict{Union{
               Agent,Patch},Location}}(), Dict{Union{
               Agent,Patch},Real2D{Int}}())
15  end
```

I campi che la struttura presenta sono i seguenti:

* *width:* la larghezza del campo.
* *height:* l'altezza del campo
* *discretization:* la costante di discretizzazione per convertire i paramentri continui in discreti.

- *toridal:* un boolean che che tiene presente se il campo sia toroidale o no, ovvero se sia implementato seguendo la logica della figura del toro o se sia delemitato agli estremi da dei margini non oltrepassabili.
- *f:* E' il dizionario che permette di trovare un certo agente o oggetto nel campo utilizzando come chiave la posizione di questo.
- *f0:* E' il dizionario speculare a *f*, presenta gli stessi elementi all'interno, ma la chiave per la ricerca è l'oggetto e il valore è la posizione.

Il costruttore sopra definito permette di settare l'altezza, la larghezza, la discretizzazione e il boolean *toroidal* al momento dell'inizializzazione.

– **location.jl**

In questo file è possibile trovare la struttura *Location* (Codice 2.15), la locazione di un oggetto o un agente.

Listing 2.15: struttura *Location*

```
6 mutable struct Location{P<:Position}
7  pos::P
8  object::Union{Agent,Patch}
9 end
```

Semplicemente associa una posizione a un oggetto, che può essere di tipo *Agent* o *Patch*.

– **position.jl**

Position, così come è stato spiegato per *Field* in precedenza, è un tipo astratto e quindi non presenta nessun tipo di logica e non è instanziabile. E' solo un nodo nella gerarchia dei tipi, il quale ha il compito di rappresentera l'idea che identifica, lasciando la parte concreta ai nodi figli. In questo caso *Position* identifica il concetto di posizione nello spazio.

Listing 2.16: definizione del tipo astratto *Position*

```
2 abstract type Position end
```

– **real2d.jl**

La struttura *Real2D*, la quale è un sottotipo di *Position*, presenta la logica per identificare un elemento nello spazio.

Listing 2.17: struttura *Real2D*

```
3 struct Real2D{T<:Real} <: Position
4  x::T
5  y::T
6 end
```

Banalmente memorizza le cordinate spaziali delle ascisse e delle ordinate rispettivamente nei campi x e y, entrambi sottotipi di *Real*. Questo significa che è possibile definire un punto nello spazio con qualsiasi numero appartentente all'insieme dei numeri reali.

Il file possiede anche funzioni per regolare l'aritmetica dei punti e per calcolare le distanze tra questi (sia in un campo toroidale che non).

2.2 GUI

Per ottenere un interfaccia grafica, in grado di mostrare a schermo una corretta visualizzazione dello stato di simulazione, si è deciso di utilizzare la libreria *Matplotlib*, più nello specifico il modulo *PyPlot*, che fornisce un wrapper alla libreria di *Python*.

Matplotlib è una libreria, disponibile nel linguaggio sopracitato, di grafici 2D che produce dati sulla qualità della pubblicazione in una varietà di formati cartacei e ambienti interattivi su piattaforme diverse [11].

E' possibile utilizzare il linguaggio di programmazione *Julia* tramite il modulo *PyCall.jl*. Questo viene chiamato direttamente all'interno di *PyPlot* ed

offre la possibilità di rivolgersi direttamente e interagire completamente con *Python* e tutte le sue componenti [12].

Più nel dettaglio, questa libreria permette di scrivere codice con la sintassi di *Julia* per poi effettuare un wrapping di questo e eseguirlo in *PyPlot*, ovvero il backend grafico scelto.

La rappresentazione grafica può avvenire utilizzando due costrutti differenti: la funzione `gui()` o la macro `@animate`.

La funzione `gui()` permette di mostrare il plot in una finestra a tempo di esecuzione [13].

E' possibile creare un plot dinamico ponendo questo comando in un `for`, in coppia con la funzione `scatter(x, y)`, che visualizza cerchi colorati nelle posizioni specificate dai vettori x e y (che devono avere le stesse dimensioni) [14].

Praticamente verrà stampato ad ogni iterazione del ciclo, andando a sovrascrive il grafico della precedente iterazione con quello aggiornato, ad ogni step della simulazione, in modo che la rapida successioni delle immagini dia l'impressione di movimento.

Listing 2.18: snippet per mostrare in tempo reale il plot

```
for i = 1:nsteps
...
 scatter(x, y)
 gui()
 step!(schedule)
end
```

Le animazioni vengono create in 3 passaggi:

- Inizializza un oggetto di tipo *Animation*.
- Salva ogni fotogramma dell'animazione con un `frame(anim)`.
- Converte i fotogrammi in GIF animate con `gif(anim, nome file, numero di fps)` o in video .mp4 con `mp4(anim, nome file, numero di fps)`.

Le macro `@gif` e `@animate` semplificano questo codice, nascondendo allo sviluppatore i passaggi sopraelencati [15].

Listing 2.19: macro `@animate`

```
anim = (@animate for i = 1:nsteps
...
 scatter(x, y)
 step!(schedule)
end)

mp4(anim, "/tmp/anim_fps15.mp4", fps = 15)
```

In questo caso la nostra visualizzazione verrà rappresentata sottoforma di video in formato mp4. Per fare questo c'è la necessità di creare un'animazione dal ciclo che si occupa di aggiornare lo stato e la grafica, la quale verrà passata alla funzione `mp4()` (o in modo del tutto analogo alla funzione `gif()`) insieme al nome del file da creare e il numero di fps che dovranno essere catturati durante l'esecuzione.

A differenza del primo metodo per la visualizzazione dei dati, non avremo un output immediato, ma dovremo aspettare sia terminato il render del video (o dell'immagine nel caso sia una GIF).

Capitolo 3

Simulazione Boids

In questo capitolo viene illustrata la simulazione basata su agenti *boids.jl*, descrivendo nel dettaglio la logica, l'architettura e l'implementazione. Inoltre vengono mostrati i risultati prestazionali effettuati in diverse sessioni di benchmark e dei confronti di queste con simulazioni implementate in differenti linguaggi e tool.

3.1 La logica

Boids è un programma di vita artificiale, sviluppato da Craig Reynolds nel 1986, che simula il comportamento di uno stormo di uccelli, i quali vengono chiamati *boids*, abbreviazione di "bird-oid object", che si riferisce a un oggetto dai comportamenti simili a quelli dei volatili. [7].

Il fenomeno che scaturisce dal comportamento dei volatili (evitare ostacoli o aggregarsi ai propri simili per creare stormi) emerge dalle interazioni effettuate dai singoli agenti che aderiscono a un set di semplici regole.

Le regole applicate in questa simulazione sono:

- **Avoidance:** il volatile sterza al fine di evitare il sovraffollamento locale (dunque si allontana dai boid vicini)
- **Consistency:** il volatile sterza al fine di allinearsi alle traiettorie di volo dei boid vicini.

- **Cohesion:** il volatile sterza al fine di muoversi verso la posizione media (baricentro) dei boid vicini.
- **Randomness:** il volatile esegue un movimento dettato dal caso.

3.2 L'architettura

In questa sezione viene illustrata la struttura della simulazione *Boids.jl*.

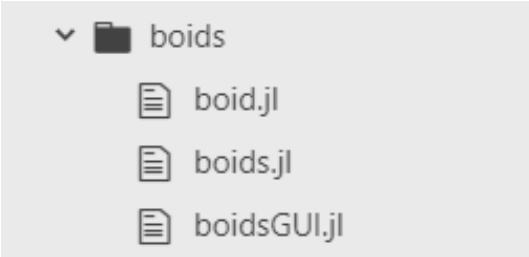

Figura 3.1: Vista ad albero della struttura di *Boids.jl*.

Segue la descrizione della struttura:

- **boid.jl**
  Contiene la struttura e le funzioni che stabiliscono le proprietà e i comportamenti dell'agente di tipo *boid*.
- **boids.jl**
  Presenta lo script che si occupa di inizializzare correttamente l'ambiente di simulazione, inserire gli agenti nella coda a priorità e avviare l'esecuzione.
- **boidsGUI.jl**
  E' lo script che implementa la visualizzazione grafica. Come la sua controparte sprovvista d'interfaccia grafica, prima dell'esecuzione, inizializza sia l'ambiente (*Field*) sia la coda degli eventi (*Schedule*)

3.3 L'implementazione

3.3.1 L'agente

L'agente di tipo *boid* viene descritto nella simulazione tramite la struttura che ne definisce le proprietà e dalla funzione `fstep()` che regola il comportamento di questo a ogni passo dell'esecuzione.

Listing 3.1: struttura *BoidData* e relativo costruttore

```
7  mutable struct BoidData
8   pos::Real2D
9   isDead::Bool
10  lastPos::Real2D
11
12  BoidData(location::Real2D) = new(location, false, Real2D
       (0, 0))
13 end
```

BoidData memorizza al suo interno la posizione corrente e quella appena precedente del volatile grazie all'utilizzo dei campi *pos* e *lastPos*, entrambi di tipo *Real2D*. Inoltre è presente un boolean(*isDead*) per definire se l'agente *boid* preso in esame sia in vita o no, così da poter sfruttare quelli defunti come ostacoli nella simulazione.

Listing 3.2: funzione `fstep`

```
145 function fstep(state::SimState, agent::Agent)
146
147  boids = BoidsData()
148  thisBoid = agent.state
149  field = state.fields[length(state.fields)]
150
151  if thisBoid.isDead
152   return
153  end
154
155  neighbors = getNeighbors(field, thisBoid)
```

```
156
157  avoid = avoidance(neighbors, field, thisBoid)
158  cohe = cohesion(neighbors, field, thisBoid)
159  rand = randomness()
160  cons = consistency(neighbors, field)
161  mome = thisBoid.lastPos
162
163  dx = boids.cohesion * cohe.x + boids.avoidance * avoid.x
         + boids.consistency * cons.x + boids.randomness *
        rand.x + boids.momentum * mome.x
164  dy = boids.cohesion * cohe.y + boids.avoidance * avoid.y
         + boids.consistency * cons.y + boids.randomness *
        rand.y + boids.momentum * mome.y
165
166  dis = sqrt(dx*dx + dy*dy)
167  if dis > 0
168   dx = dx / dis * boids.jump
169   dy = dy / dis * boids.jump
170  end
171
172  thisBoid.lastPos = Real2D(dx, dy)
173  thisBoid.pos = Real2D(tTransform(thisBoid.pos.x + dx,
        field.width), tTransform(thisBoid.pos.y + dy, field.
        height))
174
175  setObjectLocation!(field, agent, thisBoid.pos)
176 end
```

Nella funzione `fstep()` (Codice 3.2) è presente il codice che regola le relazioni tra il boid che l'esegue e i sui vicini.

Tra questo e i suoi simili vicini vengono calcolate tutte le possibili posizioni che potrebbe avere in base alle regole definite dalla simulazione. Tra questi viene fatto un calcolo per restituire le coordinate di ascisse e ordinate dell'effettiva nuova posizione. In seguito vengono aggiornati i dati relativi

alla locazione di questo sia nei campi della sua struttura che nel campo di simulazione.

3.3.2 Script per il run

Nel file *boids.jl* è contenuta la logica per l'esecuzione della simulazione, definendo le costanti nella struttura *boidsData* e inizializzando la coda degli eventi e il campo di simulazione.

Listing 3.3: struttura *BoidsData* e relativo costruttore

```
8  mutable struct BoidsData
9   cohesion::Float16
10  avoidance::Float16
11  randomness::Float16
12  consistency::Float16
13  momentum::Float16
14  deadFlockerProbability::Float16
15  neighborhood_distance:: Real
16  jump::Float64
17
18  BoidsData() = new(1.0, 1.0, 1.0, 1.0, 1.0,0.1, 10, 0.7)
19 end
```

La struttura (Codice 3.3) contiene i parametri (Costanti, percentuali di probabilità, etc...) da settare per personalizzare l'esecuzione in base alle esigenze.

Listing 3.4: script per l'avvio della simulazione *Boids*

```
21 simstate = SimState()
22 myschedule = Schedule(simstate)
23 width = 200.0
24 height = 200.0
25
26 global boids = BoidsData()
27 global field = Field2D(width,height,boids.
     neighborhood_distance/1.5,true)
```

```
28
29 addfield!(simstate,field)
30
31 numBoids = 100
32 step = 1000
33
34 for i in 1:numBoids
35  pos = Real2D(rand(Uniform(0, width)), rand(Uniform(0,
       height)))
36  d = BoidData(pos)
37  if rand() <= boids.deadFlockerProbability
38  d.isDead = true
39  end
40  boid = Agent(fstep,d)
41  setObjectLocation!(field, boid, pos)
42  scheduleRepeating!(myschedule,boid)
43 end
44
45 output1 = @timed  simulate!(myschedule,step);
46 time1 = output1[2];
47
48 println("time1: $time1, step/s: $(step/time1)");
```

Dopo aver creato e popolato il campo di simulazione e la coda a priorità per gli eventi, l'esecuzione viene fatta partire con l'invocazione di `simulate!()` e di questa viene memorizzato il tempo con l'ausilio della macro `@timed` posta precedentemente alla funzione.

A fine esecuzione verrà mostrato a schermo il tempo impiegato e la velocità media, misurata in step al secondo.

In questo caso specifico viene creata una simulazione composta da 100 agenti in un campo di dimensioni 200x200. I passi da compiere prima che questa si arresti sono settati a 1000.

3.3.3 Esecuzione grafica

Per rappresentare visivamente la simulazione si è deciso di salvare ad ogni iterazione del ciclo di esecuzione le cordinate degli agenti in due array, uno per le coordinate sull'asse delle x e l'altro per quelle sull'asse delle y.

Listing 3.5: funzione `simulateGraphics!`

```
78 function simulateGraphics!(schedule::Schedule, nsteps::
      Int64)
79  for i = 1:nsteps
80   field = schedule.simstate.fields[length(schedule.
        simstate.fields)]
81   points = collect(getAllObjects(field))
82   x = []
83   y = []
84
85   for j = 1:length(points)
86    push!(x, points[j].x)
87    push!(y, points[j].y)
88   end
89
90   scatter(x, y, shape = :rtriangle, color = :red,
        bacground_color = :blue, markersize = 6,
        foreground_color_subplot = :red, xlims = (0, width),
         ylim = (0, height), size = (800, 800))
91   gui()
92   step!(schedule)
93  end
94 end
```

Con l'utilizzo di queste vengono stampati a schermo dei triangoli rossi nelle posizioni indicate negli array, questo per mezzo della funzione `scatter()`. Oltre alla semplice visualizzazione base che questa ci permette, è stato possibile settare dei parametri aggiuntivi per rendere l'interfaccia grafica più dettagliata, gradevole e personalizzabile: il colore di background, la forma degli agenti, la loro grandezza, etc...

Nel listato 3.5 è mostrata la funzione grafica per poter visualizzare la simulazione a run-time, con l'utilizzo del comando `gui()`. E' comunque possibile raffigurare e salvare in un file l'esecuzione tramite la funzione `animate()` e le macro `@mp4` e `@gif`

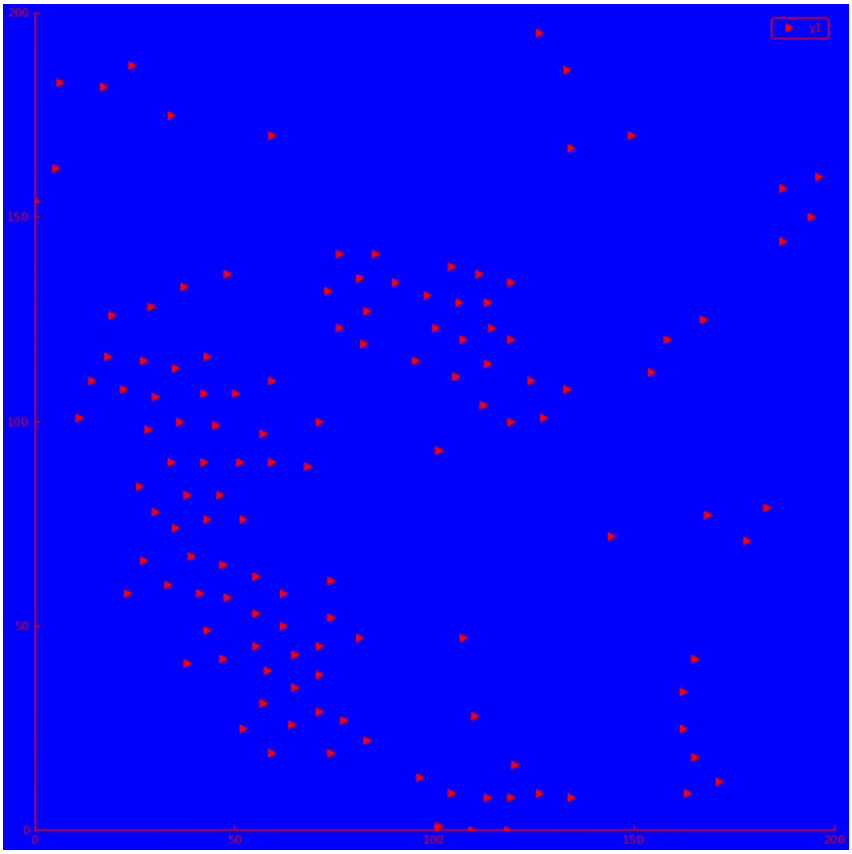

Figura 3.2: Fotogramma dell'esecuzione grafica di *Boids*.

3.4 Prestazioni

Per valutare le reali prestazioni della simulazione, sono stati svolti dei test di benchmark.

La simulazione è stata eseguita su una macchina avente le seguenti caratteristiche:

- Hardware
  - CPU: 2 x Intel(R) I7-8700T CPU @ 2.40GHz(#core 6, #thread 12)
  - RAM: 16GB
- Software
  - Ubuntu 19.04
  - Julia SDK 1.0.4

3.4.1 Benchmark

I test sono stati svolti in tre diverse sessioni. In ognuna di queste è stata messa alla prova la simulazione su aspetti differenti.

- **Densità costante**
  E' stata eseguita la simulazione mantenendo la densità costante, posta a 400, per tutta la simulazione, mostrando il decadimento delle prestazioni all'aumentare delle misure del campo e del numero di agenti. Si è partiti da un campo quadrato 200x200 con 100 agenti, per arrivare con step esponenziali a un campo 25600x25600 di ben 1.638400 agenti.
- **Densità variabile**
  Il campo è stato mantenuto del valore di 200x200 per tutta la sessione di benchmark e si sono analizzati i valori restituiti dalle simulazioni che ad ogni nuova esecuzione presentavano il doppio degli agenti di quella precedente.

- **Agenti costante**
  In questo caso il numero di agenti non è stato aumentato ad ogni nuova iterazione, ma sono stati mantenuti costanti al valore di 102.400 ed è stato il campo a variare di dimensione tra un'esecuzione e l'altra.

3.4.2 Risultati

In seguito sono riportati i grafici dei risultati di benchmark ottenuti per la simulazione *boids* in *ABM.jl*, confrontati con quelli delle implementazioni di engine concorrenti: *MASON*, *RustAB*, *NetLogo*.

Per rendere maggiormente comprensibili i grafici dei benchmark, ecco una panoramica sui competitor presi in esame per questi confronti:

- **MASON**
  Nato dallo sforzo congiunto del *George Mason University's Computer Science* e del *George Mason University Center for Social Complexity*, *MASON* è un sistema open-source e free [16].

  Questo è un toolkit interamente scritto in *Java*, progettato per essere flessibile abbastanza da essere usato sia per semplici simulazioni, sia per simulazioni massive di milioni di agenti.

- **NetLogo**
  NetLogo è un linguaggio di programmazione agent-based oltre che un *IDE* (*Integrated Development Environment*), il quale fornisce un ambiente completo per lo sviluppo *NetLogo*, dall'implementazione del modello simulativo alla creazione dell'interfaccia grafica di supporto alla simulazione [17].

- **RustAB**
  RustAB è una librearia ready-to-use sviluppata per effettuare simulazione agent based nel linguaggio di programmazione *RUST* [18].

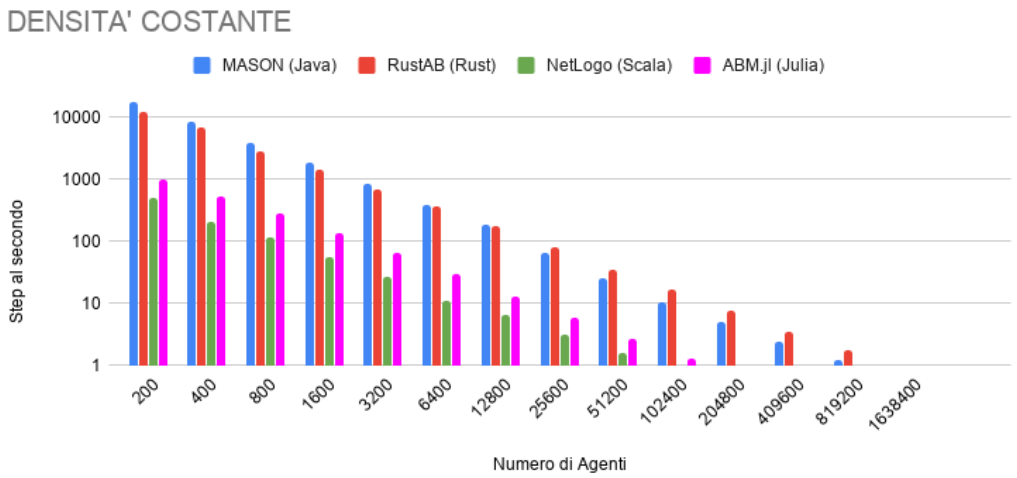

Figura 3.3: Confronto delle prestazioni con densità costante.

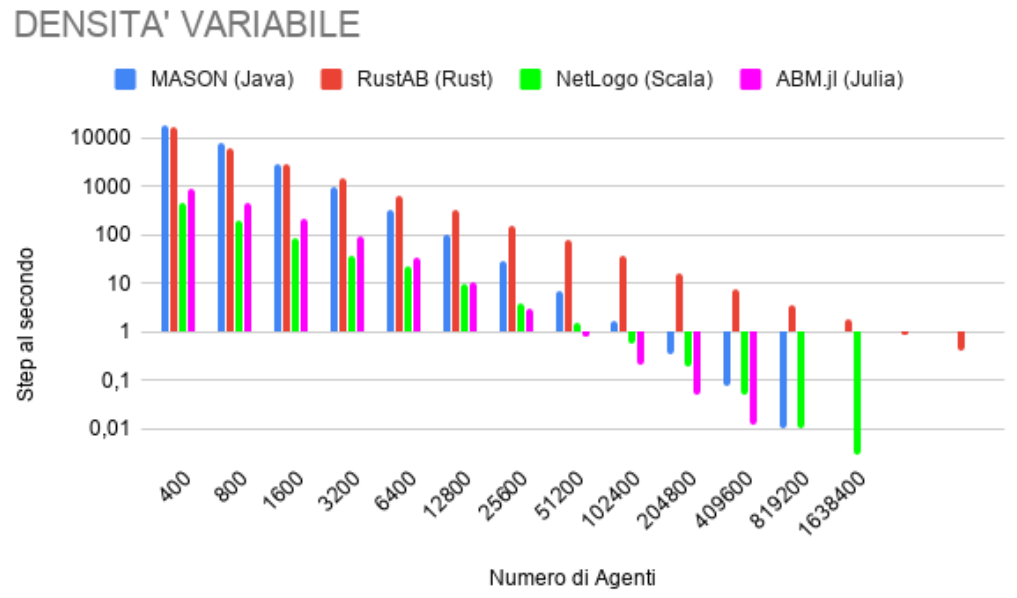

Figura 3.4: Confronto delle prestazioni con densità variabile.

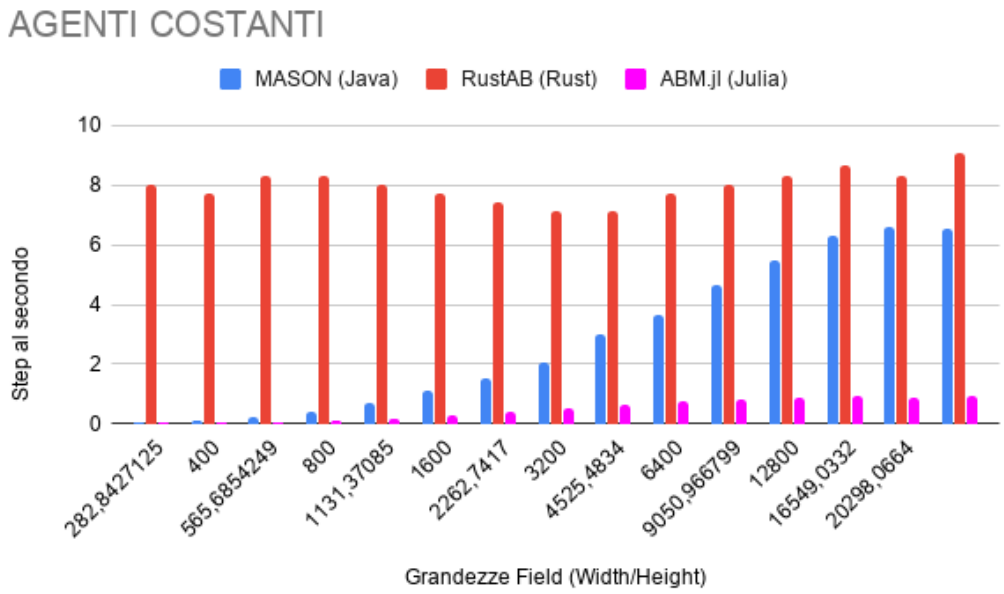

Figura 3.5: Confronto delle prestazioni con agenti costanti.

Capitolo 4

Simulazione Ant Foraging

In questo capitolo viene illustrata la simulazione basata su agenti *antForaging.jl*, descrivendo nel dettaglio la logica, l'architettura e l'implementazione. Inoltre vengono mostrati i risultati prestazionali fatte in diverse sessioni di benchmark e dei confronti di queste con simulazioni implementate in differenti linguaggi e tool.

4.1 La logica

L'idea nasce dall'osservazione dello sfruttamento delle risorse alimentari da parte delle formiche. Infatti, quest'ultime, anche se limitate alle capacità cognitive di una singola formica, sono in grado, collettivamente, di trovare il percorso più breve tra una fonte di cibo ed il formicaio [8].

E' stato dimostrato da un gruppo di biologi, dopo una serie di esperimenti, che una colonia di formiche, con una scelta tra due percorsi di lunghezza diversa che conducono ad una fonte di cibo, tende ad utilizzare il percorso più breve.

Un modello che spiega il comportamento è il seguente:

1. una formica si muove più o meno a caso attorno all'ambiente della colonia, rilasciando una scia di feromone;

2. se scopre una fonte di cibo, ritorna più o meno direttamente alla colonia, lasciando nel suo percorso una scia di feromone differente da quella diffusa in fase di ricerca;

3. il feromone diventa attraente, e le formiche che passano lì vicino tenderanno a seguire, più o meno direttamente, il percorso per arrivare al cibo;

4. tornando alla colonia queste formiche rafforzeranno il percorso;

5. se è possibile ottenere la stessa fonte alimentare da un percorso più breve, le formiche tenderanno a seguirlo;

6. il percorso più breve sarà sempre più rafforzato, e quindi più attraente;

7. i percorsi che non sono quelli ottimali finiranno per scomparire, infatti il feromone si volatilizza;

8. alla fine tutte le formiche determinano e "scelgono" il percorso più breve.

Le formiche usano l'ambiente come mezzo di comunicazione: si scambiano informazioni indirettamente depositando feromone, in modo da descrivere lo stato del loro "lavoro". Le informazioni vengono scambiate in ambito locale, infatti una formica ha accesso al feromone solo se si trova nel luogo dove questo è stato depositato.

4.2 L'architettura

In questa sezione viene illustrata la struttura della simulazione *AntForaging.jl*.

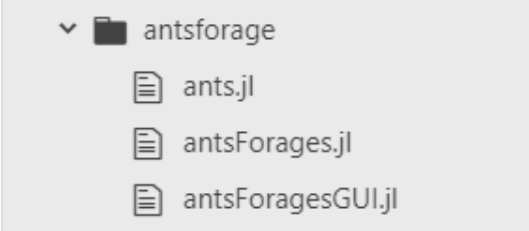

Figura 4.1: Vista ad albero della struttura di *AntForaging.jl*.

Segue la descrizione della struttura:

- **ants.jl**
  Contiene la struttura e le funzioni che stabiliscono le proprietà e i comportamenti dell'agente di tipo *ant*.

- **antsForages.jl**
  Presenta lo script che si occupa di inizializzare correttamente l'ambiente di simulazione, inserire gli agenti nella coda a priorità e avviare l'esecuzione.

- **antForagesGUI.jl**
  E' lo script che implementa la visualizzazione grafica. Come la sua controparte sprovvista d'interfaccia grafica, prima dell'esecuzione, inizializza sia l'ambiente (*Field*) sia la coda degli eventi (*Schedule*)

4.3 L'implementazione

4.3.1 L'agente

L'agente di tipo *ants* viene descritto nella simulazione tramite la struttura che ne definisce le proprietà e dalla funzione `fstep()` che regola il comportamento di questo a ogni passo dell'esecuzione.

Listing 4.1: struttura *AntData*

```
211 mutable struct AntData
212  name::String
213  pos::Real2D
214  reward::Float16
215  hasFoodItem::Bool
216  lastPos::Real2D
217 end
```

AntData memorizza al suo interno la posizione corrente e quella appena precedente della formica grazie all'utilizzo dei campi *pos* e *lastPos*, entrambi di tipo *Real2D*. Inoltre è presente un boolean (*hasFoodItem*) per definire se l'agente *ant* preso in esame porti con se del cibo oppure no, così da poter stabilire il comportamento che questo dovrebbe seguire.

Sono presenti i campi *name* e *reward*, che rispettivamente rappresentano il nome dell'agente e il valore della "ricompensa" che viene data alla formica ogni volta che riesce a raggiungere un punto d'interesse presente nel campo di simulazione.

Listing 4.2: funzione `fstep`

```
201 function fstep(state::SimState, agent::Agent)
202
203  afd.toFoodGrid *= afd.evaporationConstant
204  afd.toHomeGrid *= afd.evaporationConstant
205
206  depositPheromone(state::SimState, agent::Agent)
207  act(state::SimState, agent::Agent)
208 end
```

Nella funzione `fstep()` (Codice 4.2) è presente il codice che regola il rilascio di feromone dell'agente preso in esame e il suo movimento, sia quello per la fase di ricerca, sia quello per ritornare alla tana una volta recuperato il cibo.

Questi comportamenti vengono simulati tramite l'invocazione delle due funzioni `depositPheromone` e `act`, entrambe che prendono come paramentri lo stato della simulazione e l'agente corrente.

Ad ogni step della simulazione sono poi aggiornate le griglie contenenti il feromone per ritornare a casa e per arrivare al cibo, in modo da diminuire costantemente il valore di questo e così da rappresentare la sua reale evaporazione con il passare del tempo.

4.3.2 Script per il run

Nel file *antsForages.jl* è contenuta la logica per l'esecuzione della simulazione, definendo le costanti nella struttura *antsData* e inizializzando la coda degli eventi e il campo di simulazione.

Listing 4.3: struttura *AntsForageData* e relativo costruttore

```
8  mutable struct AntsForageData
9  HOME::Int8
10 FOOD::Int8
11 updateCutDown::Float64
12 diagonalCutDown::Float64
13 evaporationConstant::Float64
14 momentumProbability::Float64
15 randomActionProbability::Float64
16 afReward::Float64
17
18 IMPOSSIBLY_BAD_PHEROMONE::Float16
19 LIKELY_MAX_PHEROMONE::Float16
20
21 toFoodGrid
22 toHomeGrid
23
24 locationGrid
25 obstacleGrid
26
```

```
27 AntsForageData() = new(1, 2, 0.9, 0.9^?2, 0.999, 0.80,
       0.10, 1.0, -1, 3, zeros(Int64, convert(Int, height),
       convert(Int, width)), zeros(Int64, convert(Int, height
       ), convert(Int, width)), zeros(Int64, convert(Int,
       height), convert(Int, width)), zeros(Int64, convert(
       Int, height), convert(Int, width)))
28
29 end
```

La struttura (Codice 4.3) contiene i parametri (Costanti, percentuali di probabilità, etc...) da settare per personalizzare l'esecuzione in base alle esigenze.

Listing 4.4: script per l'avvio della simulazione *AntsForaging*

```
21 simstate = SimState()
22 myschedule = Schedule(simstate)
23 width = 150
24 height = 150
25 neighborhood_distance = 10
26
27 global field = Field2D(width,height,neighborhood_distance
       /10,false)
28
29 global afd = AntsForageData()
30
31 #coordinates HOME
32 global const HOME_X = 135
33 global const HOME_Y = 15
34
35 #coordinates FOOD
36 global const FOOD_X = 25
37 global const FOOD_Y = 115
38
39 #POINT HOME AND FOOD
40 posHome = Real2D(HOME_X, HOME_Y)
```

```
41 posFood = Real2D(FOOD_X, FOOD_Y)
42
43 afd.locationGrid[posHome.x, posHome.y] = afd.HOME
44 afd.locationGrid[posFood.x, posFood.y] = afd.FOOD
45
46 for x = 40:80
47  for y = 70:80
48   afd.obstacleGrid[x, y] = 1
49  end
50 end
51
52 numAnts = 300
53
54 addfield!(simstate,field)
55
56 pos = Real2D(HOME_X, HOME_Y)
57
58 for i in 1:numAnts
59  a = AntData("Ant", pos, afd.afReward, false, posHome)
60  ant = Agent(fstep,a)
61  setObjectLocation!(field, ant, pos)
62  scheduleRepeating!(myschedule,ant)
63 end
64
65 output1 = @timed  simulate!(myschedule,step);
66 time1 = output1[2];
67
68 println("time1: $time1, step/s: $(step/time1)");
```

Dopo aver creato e popolato il campo di simulazione e la coda a priorità per gli eventi, l'esecuzione viene fatta partire con l'invocazione di `simulate!()` e di questa viene memorizzato il tempo con l'ausilio della macro `@timed` posta precedentemente alla funzione.

A fine esecuzione verrà mostrato a schermo il tempo impiegato e la velocità media, misurata in step al secondo.

Più precisamente, in questa simulazione viene istanziato un campo caratterizzato da un ostacolo che si frappone tra il punto di partenza (*HOME*) e il punto posto come obiettivo (*FOOD*), i quali si trovano rispettivamente a sud-est e nord-ovest del *field*.

Questo, di misure 150x150, è popolato da 300 esemplari di agente *ant*.

4.3.3 Esecuzione grafica

Per rappresentare visivamente la simulazione si è deciso di salvare ad ogni iterazione del ciclo di esecuzione le cordinate degli agenti in due array, uno per le coordinate sull'asse delle x e l'altro per quelle sull'asse delle y. Lo stesso viene fatto anche per le cordinate delle diverse regioni del campo (l'area del cibo, della tana e dove sono presenti gli ostacoli) e per i due differenti tipi di feromone: quello per ritornare a casa e quello per arrivare al cibo.

Listing 4.5: funzione `simulateGraphics!`

```
78 function simulateGraphics!(schedule::Schedule, nsteps::
       Int64)
79  mp4(@animate(for i = 1:nsteps
80   field = schedule.simstate.fields[length(schedule.
         simstate.fields)]
81   points = collect(getAllObjects(field))
82   x = []
83   y = []
84
85   foodx = []
86   foody = []
87
88   obstaclex = []
89   obstacley = []
90
91   homex = []
92   homey = []
93
94   locationx = []
```

```
95   locationy = []
96
97   for j = 1:length(points)
98    push!(x, points[j].x)
99    push!(y, points[j].y)
100  end
101
102  for v = 1:width
103   for w = 1:height
104    if afd.toFoodGrid[v, w] != 0
105     push!(foodx, v)
106     push!(foody, w)
107    end
108
109    if afd.toHomeGrid[v, w] != 0
110     push!(homex, v)
111     push!(homey, w)
112    end
113
114    if afd.locationGrid[v, w] != 0
115     push!(locationx, v)
116     push!(locationy, w)
117    end
118
119    if afd.obstacleGrid[v, w] != 0
120     push!(obstaclex, v)
121     push!(obstacley, w)
122    end
123   end
124  end
125
126  scatter(homex, homey, shape = :square, color = :blue,
        markersize = 3,
127   xlims = (0, width), ylim = (0, height), size = (800,
        800), legend =false)
```

```
128
129   scatter!(foodx, foody, shape = :square, color = :yellow
          , markersize = 3,
130    xlims = (0, width), ylim = (0, height), size = (800,
          800), legend =false)
131
132   scatter!(obstaclex, obstacley, shape = :square, color =
          :orange, markersize = 3,
133    xlims = (0, width), ylim = (0, height), size = (800,
          800), legend =false)
134
135   scatter!(locationx, locationy, shape = :square, color =
          :pink, markersize = 10,
136    xlims = (0, width), ylim = (0, height), size = (800,
          800), legend =false)
137
138   scatter!(x, y, shape = :circle, color = :red,
          markersize = 5,
139    xlims = (0, width), ylim = (0, height), size = (800,
          800), legend =false)
140
141   step!(schedule)
142
143  end), "ants.mp4", fps = 30)
144 end
```

Con l'utilizzo di queste vengono stampati a schermo dei puntini rossi, rappresentanti le formiche; scie di colore blu e giallo per rappresentare rispettivamente il feromone rilasciato in fase di ricerca e in fase di ritorno alla tana; i punti d'interesse presenti nella simulazione (*HOME* e *FOOD*); gli ostacoli che obbligano gli agenti a trovare percorsi differenti per raggiungere il proprio obiettivo.

Tutto questo grazie all'utilizzo della funzione `scatter()`.

Oltre alla semplice visualizzazione base che questa ci permette, è stato possibile settare dei parametri aggiuntivi per rendere l'interfaccia grafica più

dettagliata, gradevole e personalizzabile: il colore di background, la forma degli agenti, la loro grandezza, etc...

Nel listato 4.5 è mostrata la macro `@mp4`, la quale ha permesso, in questo caso, di poter salvare in memoria un video riproducibile a 30 frame per secondo intitolato "ants.mp4", nel quale è possibile visualizzare l'esecuzione. E' comunque possibile raffigurare la simulazione in tempo reale tramite l'utilizzo della funzione `gui()`

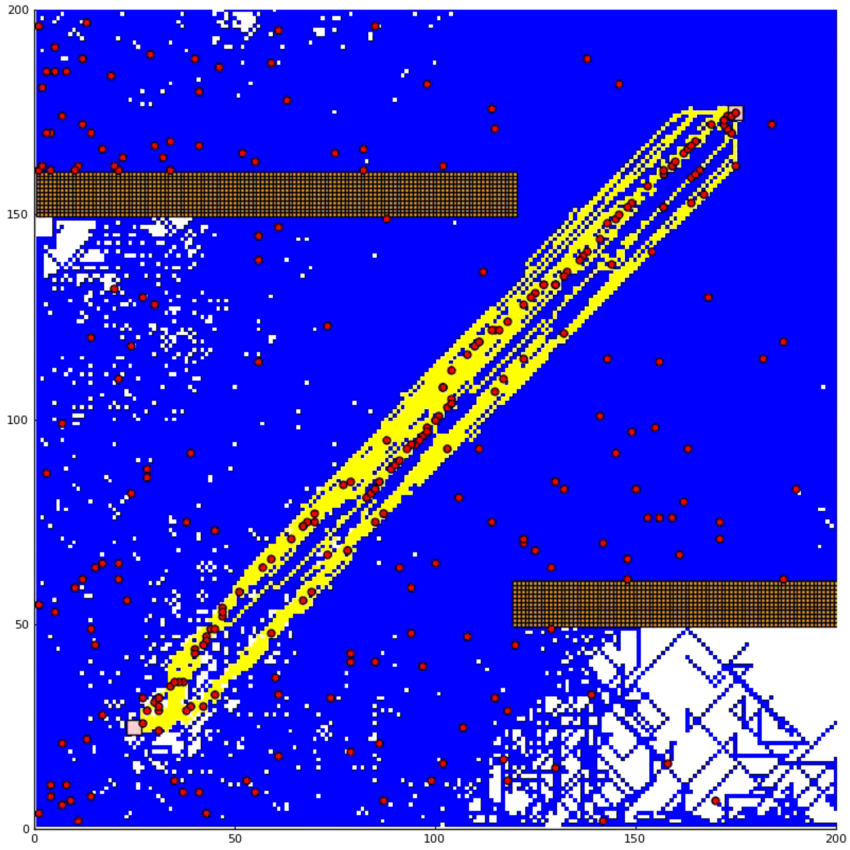

Figura 4.2: Fotogramma dell'esecuzione grafica di *AntsForaging*.

4.4 Prestazioni

Per valutare le reali prestazioni della simulazione, sono stati svolti dei test di benchmark.

La simulazione è stata eseguita su una macchina avente le seguenti caratteristiche:

- Hardware
  - CPU: 2 x Intel(R) I7-8700T CPU @ 2.40GHz(#core 6, #thread 12)
  - RAM: 16GB
- Software
  - Ubuntu 19.04
  - Julia SDK 1.0.4

4.4.1 Benchmark

I test sono stati svolti in tre diverse sessioni. In ognuna di queste è stata messa alla prova la simulazione su aspetti differenti.

- **Densità costante**
  E' stata eseguita la simulazione mantenendo la densità costante, posta a 400, per tutta la simulazione, mostrando il decadimento delle prestazioni all'aumentare delle misure del campo e del numero di agenti. Si è partiti da un campo quadrato 50x50 con 6 agenti, per arrivare con step costanti a un campo 500x500 di ben 625 agenti.
- **Densità variabile**
  Il campo è stato mantenuto del valore di 200x200 per tutta la sessione di benchmark e si sono analizzati i valori restituiti dalle simulazioni che ad ogni nuova esecuzione presentavano cento agenti in più di quella precedente.

- **Agenti costante**
  In questo caso il numero di agenti non è stato aumentato ad ogni nuova iterazione, ma sono stati mantenuti costanti al valore di 300 ed è stato il campo a variare di dimensione tra un'esecuzione e l'altra.

4.4.2 Risultati

In seguito sono riportati i grafici dei risultati di benchmark ottenuti per la simulazione *boids* in *ABM.jl*, confrontati con quelli delle implementazioni di engine concorrenti: *MASON*, *NetLogo*.

Per rendere maggiormente comprensibili i grafici dei benchmark, ecco una panoramica sui competitor presi in esame per questi confronti:

- **MASON**
  Nato dallo sforzo congiunto del *George Mason University's Computer Science* e del *George Mason University Center for Social Complexity*, *MASON* è un sistema open-source e free [16].

  Questo è un toolkit interamente scritto in *Java*, progettato per essere flessibile abbastanza da essere usato sia per semplici simulazioni, sia per simulazioni massive di milioni di agenti.

- **NetLogo**
  NetLogo è un linguaggio di programmazione agent-based oltre che un *IDE* (*Integrated Development Environment*), il quale fornisce un ambiente completo per lo sviluppo *NetLogo*, dall'implementazione del modello simulativo alla creazione dell'interfaccia grafica di supporto alla simulazione [17].

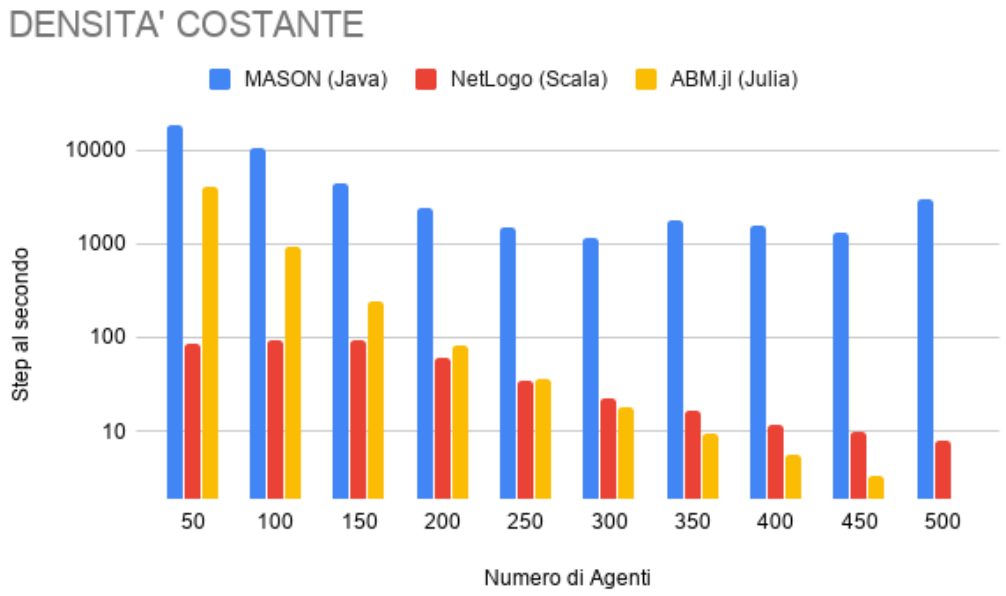

Figura 4.3: Confronto delle prestazioni con densità costante.

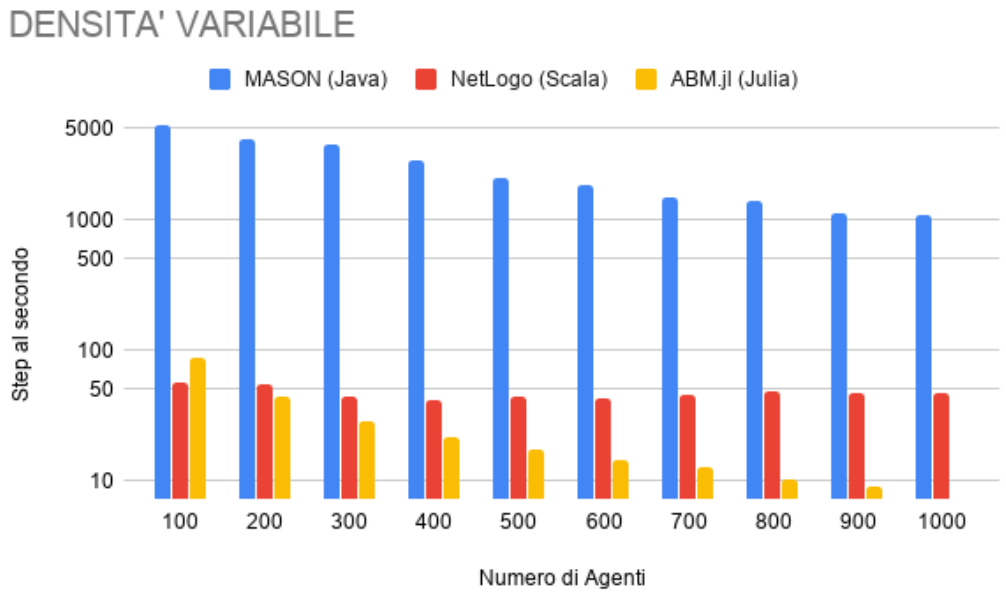

Figura 4.4: Confronto delle prestazioni con densità variabile.

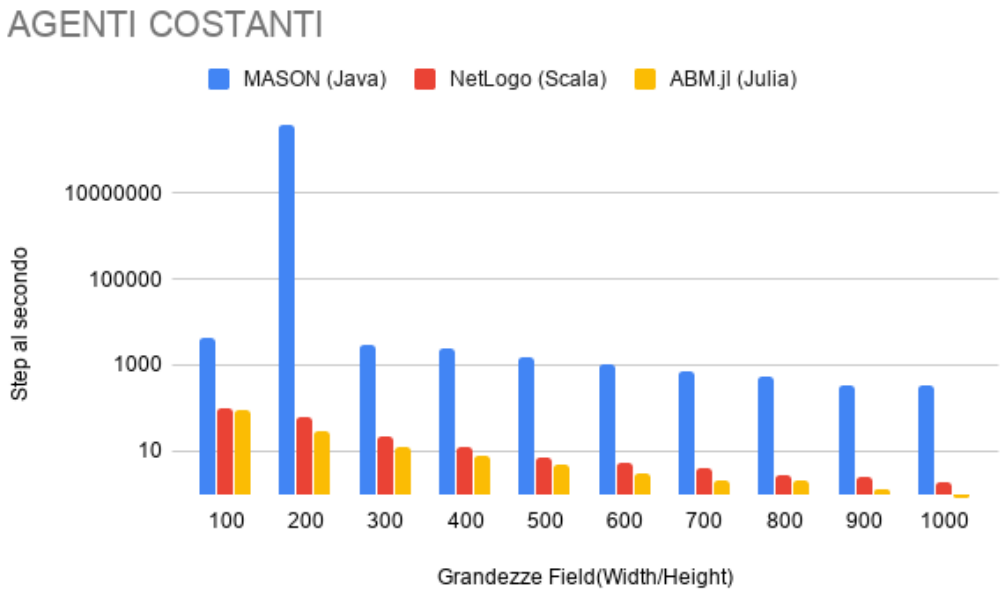

Figura 4.5: Confronto delle prestazioni con agenti costanti.

Capitolo 5

Conclusioni

In questo lavoro è stata affrontata la realizzazione di due distinte simulazioni basate su agenti: *Boids.jl* e *AntForaging.jl*
L'obiettivo è stato quello di riscriverle in un nuovo linguaggio di programmazione, il quale non era ancora stato adoperato in questo campo. Il linguaggio in questione è *Julia*, scelta fatta con l'intento di apportare nuovi benefici rispetto a quelli degli sviluppi passati, quali maggiori performance e una fase di implementazione più semplice e veloce. In aggiunta questo è un vantaggio per tutti i programmatori con una conoscenza di questo linguaggio e che hanno intenzione di affrontare il tema delle simulazioni basate su agenti.

I risultati presentati in questo lavoro mostrano un corretto funzionamento delle simulazioni, anche in casi che mettono a dura prova la macchina su cui sono eseguiti (elevato numero di agenti o dimensioni del campo considerevoli), e una gradevole interfaccia di visualizzazione grafica che permette di tener traccia dei mutamenti di stato degli agenti e dell'ambiente in modo intuitivo e durante il corso di esecuzione.

Nonostante i test di benchmark abbiano evidenziato come *ABM.jl* sia uno strumento che in termini di prestazioni ha un ampio margine di miglioramento, questa libreria rappresenta un'importante opportunità per i ricercatori che usano abitualmente Julia come linguaggio di programmazione, in quanto questi potranno avere la possibilità di lavorare su simulazioni basate ad agenti senza dover necessariamente impare un diverso linguaggio.

Per il futuro si prevede un lavoro di identificazione di eventuali colli di bottiglia per poter incrementare le prestazioni e cercare di raggiungere risultati che siano almeno alla pari delle simulazioni analoghe implementate con engine differenti.

Inoltre si proverà a migliorare il sistema di visualizzazione, rimuovendo i bug ora presenti e aggiungendo nuovi elementi grafici per rendere il tutto più dettagliato, quindi maggiormente comprensibile e gradevole alla vista.

Bibliografia

[1] Modello basato sull'agente. https://it.wikipedia.org/wiki/Modello_basato_sull'agente 1.1.1

[2] Tesi di Enrico Sisti, *Un approccio Agent Based Model per la decisione nelle scelte strategiche*, 2014. 1.1.1

[3] Julia (linguaggio di programmazione). https://it.wikipedia.org/wiki/Julia_(linguaggio_di_programmazione) 1.1.2

[4] Compilatore just-in-time. https://it.wikipedia.org/wiki/Compilatore_just-in-time 1.1.2

[5] Julialang. https://julialang.org/ 1.1.2

[6] Repository GitHub. *spagnuolocarmine/ABM.jl* https://github.com/spagnuolocarmine/ABM.jl 2

[7] Boids. https://en.wikipedia.org/wiki/Boids 3.1

[8] Algoritmo delle colonie di formiche. https://it.wikipedia.org/wiki/Algoritmo_delle_colonie_di_formiche#Origine 4.1

[9] Universally unique identifier. https://it.wikipedia.org/wiki/Universally_unique_identifier 2.1

[10] Abstract Types in Julia. https://docs.julialang.org/en/v1/manual/types/index.html#Abstract-Types-1 2.1

[11] Matplotlib. https://matplotlib.org 2.2

[12] PyCall.jl. https://github.com/JuliaPy/PyCall.jl 2.2

[13] Funzione gui(). https://docs.juliaplots.org/latest/basics/ 2.2

[14] Funzione scatter(). http://www.ece.northwestern.edu/local-apps/matlabhelp/techdoc/ref/scatter.html#targetText=scatter%20(MATLAB%20Functions)&targetText=scatter(X%2CY%2CS,and%20Y%20or%20a%20scalar. 2.2

[15] Animazioni in Julia. https://docs.juliaplots.org/latest/animations/ 2.2

[16] MASON. https://en.wikipedia.org/wiki/MASON_(Java) 3.4.2, 4.4.2

[17] Netlogo. https://it.wikipedia.org/wiki/NetLogo 3.4.2, 4.4.2

[18] Tesi di Daniele De Vinco, *Modello di simulazione basato su agenti con Rust*, 2019. 3.4.2