UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di ingegneria

Corso di laurea in Ingegneria Informatica

TESI DI LAUREA

Algoritmo di Swarm Intelligence per il coordinamento di stormi implementato sul framework Mesa

RELATORI

Prof. Mario G.C.A. CIMINO

Prof. Gigliola VAGLINI

Ing. Federico A. GALATOLO

Candidato

Matteo GAMBELLA

ANNO ACCADEMICO 2018-2019

RIASSUNTO

Molti degli studi per l'implementazione di algoritmi di Swarm Intelligence si basano sul mondo animale, in particolare su quelli che vengono definiti *comportamenti emergenti*. Il flocking rientra tra questi comportamenti e segue le tre regole illustrate da Reynolds (separazione , allineamento e coesione) nel suo modello chiamato Boids. Per la realizzazione di un modello di flocking si è utilizzato Mesa , un framework ABM (Agent-Based Model) in Python. I risultati sono stati quelli sperati : gli agenti ,senza essere a conoscenza di cosa sia un gruppo, basandosi su scelte rigorosamente locali , si combinano e ricombinano andando a creare a livello globale gli stormi attesi. La formazione dei flock può essere facilmente osservata tramite la visualizzazione interattiva offerta da Mesa.

INDICE

1 INTRODUZIONE

La swarm intelligence (o intelligenza di sciame) nasce dall'osservazione delle incredibili potenzialità organizzative di alcune specie animali che mostrano comportamenti che combinano efficienza, flessibilità e robustezza e nello scenario moderno ha senz'altro assunto una notevole rilevanza. In futuro infatti ci si aspetta di poter osservare sciami (*swarm*) di droni multimodali, di cielo e di terra, che si coordinano per lo svolgimento di compiti come pick-up e consegna, sorveglianza e esplorazione. Da ciò ne deriva l'assoluta importanza degli algoritmi di Swarm intelligence e questo sarà lo scenario in cui ci andremo a muovere andandoci a focalizzare sui comportamenti emergenti e in particolare sul problema del *flocking.*

Per la stesura della tesi hanno contribuito in maniera significativa :

- La tesi di Giacomo Albi "Swarm intelligence : modelli matematici e stormi di uccelli" da cui ho preso degli spunti per i temi da trattare nella parte relativa alla Swarm Intelligence e ai comportamenti emergenti.
- La tesi di Alessia Ambu "Modelli dinamici per fenomeni collettivi in fisica dei sistemi complessi" per lo studio del flocking.
- L'articolo "Boids: Flocking made simple" di Harmen de Weerd per quanto riguarda lo studio delle tre regole del flocking di Reynolds.
- I tutorial e la documentazione ufficale di Mesa per la parte relativa al framework e all'implementazione poi del modello di flocking. (Sitografia: https://mesa.readthedocs.io/en/master/)
- Materiale sul sito e repository messi a disposizione da Federico Galatolo per la comprensione degli aspetti fondamentali della Swarm Intelligence (emergenza, stigmergia ..)

Per quanto riguarda la struttura la tesi si articola sostanzialmente in tre capitoli.

Nel primo andremo a trattare la swarm intelligence nei suoi aspetti chiave e poi osserveremo come in natura possiamo identificare comportamenti analoghi andando a distinguere tra quelli stigmergici (vedremo le formiche e termiti) e quelli invece in cui è presente un'interazione diretta basata sulla vicinanza degli agenti in cui non abbiamo come nella stigmergia l'uso dell'ambiente come deposito di segnale (pesci e uccelli).

Nel secondo capitolo ci concentreremo sui comportamenti emergenti , in particolare il flocking e il suo modello , Boids, introdotto da Reynolds.

Finita la parte teorica passiamo invece alla parte più pratica nel terzo capitolo in cui viene prima introdotto il framework Mesa e poi viene presentata un'implementazione in Mesa di modello di flocking che segue appunto le tre regole classiche introdotte nel precedente capitolo.

2 SWARM INTELLIGENCE

2.1 Caratteristiche, vantaggi e principali algoritmi

Il lavoro di squadra ha da sempre avuto una grande importanza strategica e funzionale. In qualsiasi ambito, a maggior ragione oggigiorno, si investe per migliorare il lavoro di gruppo.

Uno studio sui **sistemi auto-organizzati** ha ispirato, nel 1988, un progetto sui sistemi robotici ad opera di tre ricercatori: Gerardo Beni, Susan Hackood e Jing Wang. Il progetto, dal nome **I-Swarm** (Intelligent Small World Autonomous Robots for Micromanipulation), parte dal principio secondo il quale **un'azione complessa deriva da un'intelligenza collettiva**, così come succede per gli insetti, ovvero, tra numerosi elementi paritari all'interno di un sistema, può emergere una nuova intelligenza superiore ad ogni singolo elemento. L'Intelligenza dello Sciame prende spunto dunque dal mondo naturale: colonie di insetti, stormi di uccelli, banchi di pesci e simili.

Il termine fu coniato dai tre ricercatori che la definirono come : *"Proprietà di un sistema in cui il comportamento collettivo di agenti (non sofisticati) che interagiscono localmente con l'ambiente produce l'emergere di pattern funzionali globali nel sistema"*.

La Swarm Intelligence presenta le seguenti caratteristiche: ogni individuo presenta un repertorio di comportamento limitato; ogni individuo non conosce lo stato globale del sistema; assenza di un capo coordinatore (es. l'ape regina non coordina l'attività delle altre api) e infine il comportamento di un individuo influenza quello degli altri.

I vantaggi principali sono i seguenti: il gruppo si auto organizza, ha un comportamento flessibile, si adatta al contesto esterno e infine manifesta una complessiva robustezza.

Algoritmi di Swarm Intelligence più conosciuti:

ACO: Ant Colony Optimization, algoritmo che usa un meccanismo di feedback positivo come una sorta di ferormone virtuale per rafforzare quelle parti di soluzione che contribuiscono alla risoluzione del problema.

PSO e FSO (entrambi nell'ambito della metaeuristica): la PSO (Particle Swarm Optimization) ottimizza un problema utilizzando una popolazione di soluzioni candidate (dette "particelle", le *particle*) che si spostano nello spazio di ricerca sulla base di semplici formule, che tengono in considerazione la loro velocità di spostamento corrente, le loro conoscenze dello spazio di fitness (ovvero la migliore soluzione che hanno esplorato finora) e la conoscenza condivisa (cioè la miglior soluzione generale identificata). La Flock of Starlings Optimization, invece si ispira al volo degli uccelli in quanto l'algoritmo assume una dinamica simile grazie alla regola fondamentale su cui si basa che suggerisce che gli uccelli in volo mediano la propria velocità rispetto a 7 individui scelti a caso nello stormo.

Principali applicazioni di Swarm Intelligence:

Implementazioni ACO

- Problemi di combinatoria
- Robotica

- Telecomunicazioni

Implementazioni PSO e FSO

- Reti neurali artificiali
- Modelli di evoluzione grammaticale
- Distribuzione dell'energia elettrica

Altre applicazioni:

- Controllo di veicoli militari senza pilota
- Nanobot
- La programmazione di fabbrica;
- Controllo decentralizzato dei veicoli senza piloti;
- Implementazioni a scopo medico.

2.2 Intelligenza di sciame nel mondo animale

Come abbiamo visto in primis nel mondo animale c'è una forte tendenza a formare un collettivo, questo perché stare in un gruppo implica numerosi vantaggi: guadagno nella maggiore possibilità di riprodursi, protezione dai predatori, maggiore facilità nel procacciarsi cibo e infine aumento dell'efficacia locomotoria (gruppi di animali che si muovono in ambiente fluido risparmiano energia quando nuotano o volano insieme, proprio come i ciclisti possono mettersi nella scia l'uno dell'altro, in squadra).Non risulterà strano quindi che proprio nel mondo biologico abbiamo degli esempi lampanti dei meccanismi accennati prima parlando di swarm intelligence.

Le formiche operaie (**foragers**, cioè che vanno in cerca di cibo) comunicano le une con le altre lasciando una traccia durante il percorso che fanno. Queste tracce, sotto forma di feromoni, sono un chiaro segnale per le formiche che seguono, fino a che il segnale resta attivo. In base al tipo di feromone lasciato il comportamento delle formiche che seguono è influenzato.

Esiste una classificazione in 4 tipi di tracce o feromoni, per meglio dire:

- **feromoni traccianti (*trace*)** che rilasciati da un individuo vengono seguiti da appartenenti alla stessa specie come una traccia.
- **feromoni di allarme (*alarm*)** che vengono emessi in situazioni di pericolo, inducendo un maggiore stato di vigilanza in quanti li captano.
- **feromoni innescanti o scatenanti (*primer*)** che inducono nel ricevente modificazioni comportamentali e/o fisiologiche a lungo termine.
- **feromoni liberatori o di segnalazione (*releaser*)** che scatenano comportamenti eccitati da parte di chi li capta.

E’ evidente che il percorso in cui si riscontrano più quantità di feromoni traccianti diventa quello più seguito (in genere il più breve e che porta al rifugio e/o al cibo).
Il punto di forza di questo meccanismo consiste quindi nella creazione di un **“sistema intelligente distribuito”**.
Ciò significa che le scelte adottate non sono prese da un’unica entità che lavora al problema, bensì da una colonia (o un team), che agisce autonomamente e per mezzo dei feromoni riesce a condividere le soluzioni appena vengono trovate, adattandosi dinamicamente alla circostanza ed al contesto.

Il meccanismo delle tracce chimiche è alla base di numerose abilità delle formiche: raccolta di cibo e ottimizzazione dei percorsi (come già visto), strategie di combattimento, costruzione di formicai sotterranei.

Questo mi permette di introdurre un termine importante nell’ambito dei sistemi auto-organizzati che è la **stigmergia**, che letteralmente vuol dire “lavoro guidato da stimoli” in quanto deriva dalle parole greche “stigma” ed “ergon”, cioè “segno” e “lavoro”. Il termine fu coniato dallo zoologo Pierre-Paul Grassè ed essa viene definita come *“una forma di comunicazione che avviene alterando lo stato dell'ambiente in un modo che influenzerà il comportamento degli altri individui per i quali l'ambiente stesso è uno stimolo” (J. Kennedy, R. C. Eberhart, "Swarm Intelligence", Morgan Kaufmann Publishers, 2001)* .Ciò infatti è quello che avviene quando le formiche rilasciano questi ferormoni nell’ambiente, che viene quindi usato come “deposito del segnale”.

Un altro esempio di comportamento stigmergico si riscontra nelle termiti. Anche le termiti infatti usano i feromoni per costruire strutture molto complesse (come nell’immagine accanto) seguendo un semplice insieme di regole decentralizzato. Ogni insetto scava una pallina di fango dal suo ambiente, la copre di feromoni e la lascia sul terreno. Le termiti sono attratte dai feromoni degli individui dello stesso termitaio e quindi depositano le loro palline di fango vicino a quelle già depositate. Con il tempo questo comportamento porta a costruire pilastri, archi, gallerie e camere.

Non abbiamo invece stigmergia nei pesci e negli uccelli. Tra le innumerevoli specie di pesci molte presentano meccanismi di aggregazione, che originano da interazioni diverse da quelle viste per le formiche e le termiti. Non si tratta ora di utilizzare l’ambiente come “deposito del proprio segnale” ma **l’interazione è diretta**, riassumibile nella terna di regole repulsione-allineamento-attrazione, ognuna agente a scale differenti. La fase ordinata è individuata dal grado di allineamento dei pesci. Questa terna viene applicata anche negli uccelli i quali hanno un comportamento molto simile ma che riprenderemo più nel dettaglio successivamente. Ciò che è importante mettere subito a fuoco è che nei pesci come negli uccelli ha una fondamentale importanza il concetto di “vicinanza”, in quanto è essa in questo caso che permette un flusso di informazione globale.

3 COMPORTAMENTI EMERGENTI

3.1 Definizione e esempi di proprietà emergenti nel campo animale, fisico e biologico

Una delle caratteristiche di un sistema complesso è l'emergenza di un **comportamento collettivo** che ovviamente non è facilmente prevedibile a partire dal comportamento dei singoli componenti. Il comportamento collettivo descrive il movimento di un gruppo di animali della stessa specie che interagiscono tra loro coordinando le azioni del singolo con le azioni del gruppo spesso portando alla *formazione di strutture organizzate* che in sistemi biologici complessi viene detta **Emergenza**.

"L'emergenza è la generazione di nuovi e coerenti strutture, pattern e priorità data dalla auto-organizzazione in sistemi complessi. (Goldstein, '99)

Le proprietà emergenti nascono dalla ripetizione all'interno del gruppo di una stessa particolare interazione, regola e identificando questa regola possiamo riprodurre il fenomeno. Ciò che quindi fa scaturire un *comportamento emergente* o *proprietà emergente* è dunque la collettività. I comportamenti complessi non sono proprietà delle singole entità e non possono essere facilmente riconosciuti o dedotti dal comportamento di entità del livello più basso.

Abbiamo visto strutture emergenti già negli insetti sociali come gli sciami di api, ma sempre nel mondo animale anche i branchi di pesci e lupi, gli stormi di uccelli, greggi e mandrie di mammiferi in generale possono essere considerati tali.

Anche in campo fisico e biologico abbiamo degli esempi. La struttura spaziale e la forma delle galassie è una proprietà emergente, che caratterizza la distribuzione su larga scala dell'energia e della materia nell'universo. I fenomeni meteorologici come gli uragani sono comportamenti emergenti. Molti sono convinti che la coscienza e la vita stessa siano proprietà emergenti di una vasta rete di interazioni, rispettivamente di neuroni e di molecole complesse.

3.2 Introduzione al flocking e ai suoi filoni di studio.

Ora che abbiamo dato un quadro generale, nell'ambito dei comportamenti emergenti ci focalizzeremo ora in particolare sul *flocking*. Il flocking `e un tipico comportamento collettivo animale. Ad essere precisi il termine si riferisce all'aggregarsi degli uccelli in stormi, ma solitamente con esso si indica il moto collettivo che emerge anche in altre specie di animali, per esempio nei pesci, negli sciami di api o in mandrie di mammiferi. La particolarità del fenomeno consiste nel fatto che si manifesta in assenza di un coordinamento centrale. Semplicemente, ogni singolo componente segue autonomamente alcune regole, dando il risultato macroscopico di qualcosa di coordinato. Lo studio da un punto di vista teorico del flocking si divide essenzialmente in due filoni: quello delle simulazioni al calcolatore e quello dei modelli matematici. Di modelli matematici possiamo citare il modello di Vickseck , il modello di Cucker-Smale e il modello d'Orsogna-Bertozzi. Per quanto riguarda le simulazioni al calcolatore invece andremo a esaminare un programma chiamato Boids.

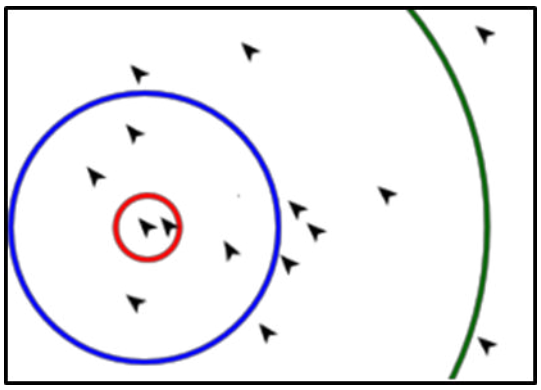
3.3 BOIDS

Nel 1986, Craig Reynolds sviluppò un modello di flocking chiamato Boids allo scopo di simulare il volo degli uccelli, in cui gli agenti si raggruppavano insieme tramite l'interazione di poche semplici regole. In questo modello gli agenti formavano gruppi pur non avendo alcuna concezione su cosa sia un gruppo. Il comportamento del gruppo era determinato da scelte prese localmente dai singoli agenti.

Il software si basa sull'interazione tra forme di vita artificiale, dette **boid**, che si muovono in un contesto tridimensionale. Nello scenario più semplice, le intelligenze artificiali decidono di modificare le proprie traiettorie sulla base di tre regole:

- **separazione (avoid)**: il boid sterza al fine di evitare il sovraffollamento locale (dunque si allontana dai boid vicini)
- **allineamento (align)**: il boid sterza al fine di allinearsi alle traiettorie di volo dei boid vicini
- **coesione (aproach)**: il boid sterza al fine di muoversi verso la posizione media (baricentro) dei boid vicini

In scenari più complessi sono state introdotte altre regole, finalizzate ad esempio ad evitare ostacoli ,raggiungere obiettivi o scappare da possibili predatori.



Il moto dei boid può essere caotico o ordinato.

Andiamo a vedere nel dettaglio le tre regole. I cerchi, in figura, servono a indicare il raggio di validità di tali regole perché esse per ogni boid interessano solo i boid nelle vicinanze (*neighbors*) che rientrano all'interno di tale area e l'area di interesse cambia a secondo di quale regola consideriamo.

La **regola di separazione** è quella con un raggio minore (cerchio rosso nel disegno) e previene che un boid si scontri con altri boid compagni (*flockmates*). Il trade-off di questa regola è che a un raggio piccolo fa corrispondere però la forza più significativa sul comportamento del boid. Un boid sottoposto ad essa ignora momentaneamente le altre due regole (align e aproach).

La **regola di allineamento** (cerchio blu) è intermedia per quanto riguarda il raggio di azione. Essa fa sì che i boid dello stesso flock assumino una direzione media comune e agisce in contemporanea insieme alla regola di coesione determinando comportamenti nel modello di flocking diversi a seconda di quanto si rendono intense tali forze (ma anche al variare del loro raggio di azione).

La **regola di coesione** (cerchio verde) è quella con il raggio maggiore (questo significa che i boid al di fuori di tale raggio sono ignorati). Essa fa sì che il boid tenda a muoversi verso il centro del gruppo di boid che esso riesce a vedere.

4 Mesa: Agent-based modeling in Python 3+

4.1 Introduzione ai modelli ABM e principali funzioni di Mesa

ABM (**Modello ad agenti , Agent-based Model)** 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.

Questi modelli si basano fondamentalmente su tre elementi che nel loro complesso vengono definiti "***agenthood***" (letteralmente la qualità di essere un agente) :

-agenti individuali (**individual agents**):essi sono programmi software intelligenti, capaci di percepire il loro ambiente e gli altri agenti in tale ambiente. Essi possiedono un set di regole e prendono decisioni sulla base di esse. Essi sono flessibili , autonomi e proattivi (ovvero prendono iniziativa, non agiscono solo in risposta all'ambiente).

-società di agenti (**agent societies**): per definizione questi modelli presentano più di un agente , per cui essi presenteranno ognuno un numero diverso di agenti, un diverso grado di cooperazione tra gli agenti, un tipo diverso di agenti (omogenei o eterogenei) ma anche di società (società aperta o chiusa a nuovi agenti rispetto a quelli definiti all'inizio).

-l'ambiente in cui sono situati (**situated environment**) : oltre agli agenti può presentare altri oggetti attivi o passivi. Gli oggetti si differenziano dagli agenti perché essi eseguono una subroutine senza prendere decisioni, la eseguono e basta, al contrario gli agenti hanno pieno controllo delle proprie azioni ("Objects do it for free, agents do it for money").

Questo framework permette velocemente di costruire modelli di questo tipo usando componenti predefinite (come griglie o agent schedulers) o implementazioni su misura; visualizzarli tramite un'interfaccia Browser-based e raccoglierne i dati tramite gli strumenti di analisi di Python.

4.2 Classe modello e classi agente. Moduli Time e Space. Oggetto griglia.

Per scrivere un qualunque codice in Mesa due tipi di classi hanno fondamentale importanza : la **classe modello** e le **classe agente**. La classe modello ha una visione globale del modello e gestisce gli agenti. Ogni istanza di tale classe corrisponde a una run. Ogni modello contiene più agenti i quali sono tutti istanze delle classi agente. Sia la classe modello che le classi agente sono classi figlie di **Agen**t e **Model** , classi generiche di Mesa (**mesa.Model** e **mesa.Agent**).

Un altro componente importante in Mesa è lo **scheduler**, il quale controlla l'ordine con cui gli agenti vanno attivati. Lo scheduler e i suoi metodi fanno parte del modulo Time (**mesa.time**). **mesa.time.RandomActivation** permette di attivare tutti gli agenti uno alla volta, in ordine randomico. Sia classe modello che classe agente possiedono un metodo **step:** nella classe agente il metodo rappresenta l'azione che l'agente compie ad ogni step della run, mentre nella classe modello esso fa avanzare il modello di uno step tramite lo scheduler.

Tutti questi elementi da soli non bastano per costruire un ABM infatti tali ambienti sono calati in un determinato ambiente per cui va introdotto l'elemento spaziale di Mesa,il modulo **mesa.space**. Mesa supporta due tipi di spazio : a griglia o continuo(**mesa.space.ContinuousSpace**).

Le griglie sono divise in celle e gli agenti possono essere visti come delle pedine che si muovono in tale scacchiera. Esistono due tipi di griglie: **mesa.space.SingleGrid** (al massimo un'unità per cella) e **mesa.space.MultiGrid**(supporta più unità per cella).Ogni agente presenta una tupla **pos** che indica le coordinate x,y di tale agente nella griglia.

Lo spazio continuo invece permette gli agenti di assumere posizioni arbitrarie.

In entrambi casi è frequente avere uno spazio di tipo toroidale, cioè con l'estremità sinistra collegata a quella destra e quella alta con quella in basso, in modo tale che tutte le celle abbiano uno stesso numero di vicini.

Dopo aver avuto un quadro generale di come è strutturato Mesa nei suoi moduli ,visto che poi l'implementazione del modello ispirato a quello di Boids sarà su griglia,ora concentriamoci su come realizzare essa e vediamo quali operazioni sono utili alla nostra rappresentazione.

L'oggetto predefinito griglia è il seguente:

***class* mesa.space.Grid(*width*, *height*, *torus*)**

Esso prende tre parametri : larghezza della griglia, altezza della griglia e un booleano per specificare se la griglia è toroidale o meno.

Metodi utili su griglia :

get_neighborhood(*pos*, *moore*, *include_center=False*, *radius=1*)
Ritorna la lista di celle (una lista di tuple) che sono nelle vicinanze di una determinata posizione. Se Moore è True ritorna le posizioni includendo le diagonali (Moore neighborhood), se invece è False esclude le diagonali (Von Neumann neighborhood).Radius definisce il raggio con cui prendere i vicini.

get_neighbors(*pos*, *moore*, *include_center=False*, *radius=1*)
Analogo al precedente , ma ritorna i vicini invece che le loro posizioni.

is_cell_empty(*pos*)
Ritorna True se la cella individuata dalla posizione pos è vuota , False altrimenti.

move_agent(*agent*, *pos*)
Muove l'agente nella posizione indicata.

place_agent(*agent*, *pos*)
Colloca l'agente sulla griglia alla posizione indicata. Setta l'attributo pos dell'agente.

torus_adj(*pos)*

Converte le coordinate in modo da permettere una gestione toroidale della griglia.

Dopo aver introdotto tutti questi elementi tiriamo un po' le somme e vediamo l'esempio di un file MoneyModel.py in cui degli agenti si muovono su una griglia e cedono soldi, sempre che ne possiedano, ai compagni di cella.

```
from mesa import Agent, Model
from mesa.time import RandomActivation
from mesa.space import MultiGrid

class MoneyModel(Model):
    """A model with some number of agents."""
    def __init__(self, N, width, height):
        self.num_agents = N
        self.grid = MultiGrid(width, height, True)
        self.schedule = RandomActivation(self)
        # Create agents
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
            self.schedule.add(a)
            # Add the agent to a random grid cell
            x = self.random.randrange(self.grid.width)
            y = self.random.randrange(self.grid.height)
            self.grid.place_agent(a, (x, y))

    def step(self):
        self.schedule.step()

class MoneyAgent(Agent):
    """ An agent with fixed initial wealth."""
    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.wealth = 1

    def move(self):
        possible_steps = self.model.grid.get_neighborhood(
            self.pos,
            moore=True,
            include_center=False)
        new_position = self.random.choice(possible_steps)
        self.model.grid.move_agent(self, new_position)

    def give_money(self):
        cellmates = self.model.grid.get_cell_list_contents([self.pos])
        if len(cellmates) > 1:
            other = self.random.choice(cellmates)
            other.wealth += 1
            self.wealth -= 1

    def step(self):
        self.move()
        if self.wealth > 0:
            self.give_money()
```

4.3 Visualizzazione interattiva del modello

Un ulteriore aspetto interessante di Mesa riguarda come visualizzare il modello creato. Mesa, infatti , come tanti altri framework ABM, permette di creare una visualizzazione interattiva del modello attraverso delle componenti predefinite messe a disposizione da Mesa stesso.

Vantaggi della visualizzazione del modello (step by step):

- individuazione di bug, comportamenti inaspettati.

- facilita la comprensione del modello , soprattutto per chi ha poca familiarità con il framework.

- permette di sviluppare nuove intuizioni o ipotesi.

La visualizzazione viene fatta in una finestra browser, usando Javascript per disegnare ciò che c'è da mostrare a video a ogni step del modello. Mesa lancia un piccolo server web che esegue il modello, trasforma ogni step in un oggetto JSON e invia gli step al browser. Ogni visualizzazione è composta di diversi modelli, non solo quelli predefiniti in quanto Mesa dà la possibilità di aggiungere anche i propri. Un esempio di modulo è quello per disegnare gli agenti. Ogni modulo ha una parte Python, che gira sul server e trasforma uno stato del modello in dati JSON; e un lato JavaScript, che prende quei dati JSON e li disegna nella finestra del browser.

Passando al lato pratico , oltre al file in Python definito come abbiamo visto primo (con le classi agente e la classe modello) si andrà a creare un nuovo file Python che deve importare la server class e la Canvas Grid Class (così chiamata perché usa HTML5 per disegnare la griglia).Ovviamente andrà fatto l'import anche dell'altro file.

Ecco un semplice esempio , in cui si considera le classi agente e modello nel file MoneyModel.py. Questo è come si presenterà un possibile file Viz_MoneyModel.py

```
from MoneyModel import *
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.ModularVisualization import ModularServer


def agent_portrayal(agent):
    portrayal = {"Shape": "circle",
                 "Filled": "true",
                 "Layer": 0,
                 "Color": "red",
                 "r": 0.5}
    return portrayal

grid = CanvasGrid(agent_portrayal, 10, 10, 500, 500)
server = ModularServer(MoneyModel,
                       [grid],
                       "Money Model",
                       {"N":100, "width":10, "height":10})
server.port = 8521 # The default
server.launch()
```

Canvas Grid lavora esaminando cella per cella della griglia e generando una rappresentazione (portrayal) per ogni agente che incontra. Per fare ciò l'unica cosa di cui si ha bisogno è di una funzione che prende un agente e ne restituisce un oggetto portrayal. Nell'esempio il metodo in questione è agent_portrayal(agent).

Oltre al metodo ovviamente va creata la Canvas Grid specificando larghezza e altezza sia in celle che in pixel 84 parametri).

Ora non resta che creare e lanciare il server. I parametri passati sono:

-la classe modello che vogliamo visualizzare : MoneyModel.

-una lista di moduli oggetto da includere nella visualizzazione; nell'esempio solo [grid].

-titolo del modello : "Money Model".

-tutti gli argomenti e input per il modello stesso.; nell'esempio numero degli agenti e larghezze e altezza dell'ambiente.

Una volta creato il server, settiamo poi la porta su cui stare in ascolto.

Quando siamo pronti usiamo il metodo launch() del server per avviare la visualizzazione.

Lanciando il file, partirà la visualizzazione interattiva e aprirà il web browser in automatico.

Nella visualizzazione sarà presente un pannello di controllo in cui è possibile impostare il rate con cui far muovere gli agenti durante la run e sono presenti i seguenti pulsanti:

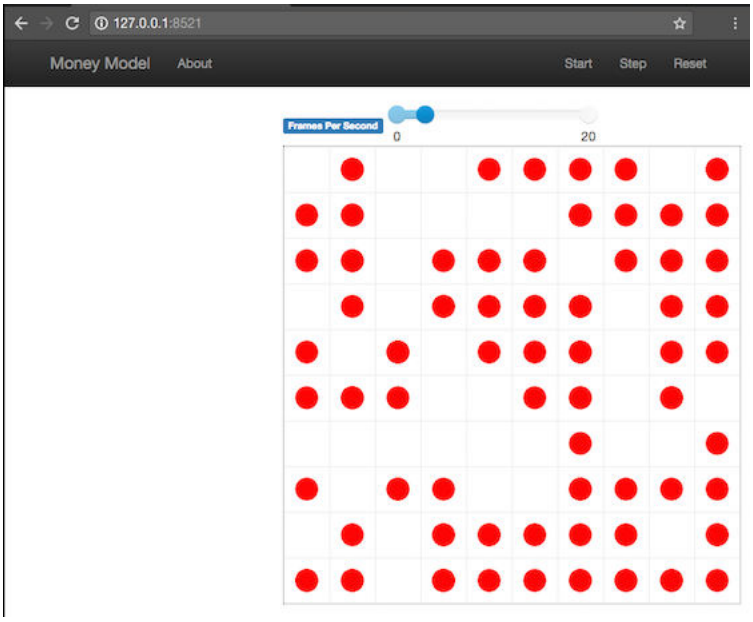
RESET : ripristina la griglia allo stato iniziale di una nuova run.

STEP: avanza il modello di uno step (si muovono gli agenti, sempre che sia previsto dal modello).

RUN: gli agenti continueranno a muoversi al rate (in fps) impostato .

PAUSE: mette in pausa il modello. Premendo RUN il modello riprende a muoversi. Rischiacciando RESET creeremo una nuova istanza del modello.

Per fermare la visualizzazione tornare dal terminale da cui è stata lanciata e premere Control + C.



4.4 FlockingModel : implementazione del modello ispirato a Boids

Possiamo ora passare a illustrare il modello di flocking realizzato con il framework Mesa precedentemente introdotto.

Il progetto consta 4 file : model.py, config.py ,utils.py e viz_model.py .

- **MODEL.PY**

È il file che contiene la classe modello FlockingModel e la classe agente FlockingAgent.

La classe modello è la più banale, infatti oltre al metodo step (di cui si è parlato prima) presenta un costruttore che crea la griglia , inizializza gli agenti dando loro una posizione e una direzione di movimento casuale e infine colloca gli agenti sulla griglia.

Molto più particolare è invece la classe FlockingAgent di cui andremo a spiegare step by step tutti i metodi.

COSTRUTTORE: il costruttore chiama il costruttore della super classe (passando l'id dell'agente e il modello) e inizializza la direzione dell'agente e setta a False l'avoidance malus, attributo che viene settato a True dopo che viene applicata la regola 1 sull'agente. Esso serve sostanzialmente ad evitare che l'agente da cui ci si vuole allontanare per mantenere le distanze possa inseguire quello che si allontana tramite un applicazione erronea delle altre regole e inoltre evita che la direzione dell'agente che fugge vada ad intaccare quella del flock, questo perché chi possiede l'avoidance malus per uno step risulta un fantasma per gli altri uccelli nelle vicinanze.

MOVE: il metodo move semplicemente sposta l'agente dalla posizione attuale alla posizione finale.

AVOID: il metodo simula la prima regola del flocking e il raggio dell'avoid è configurabile da file di configurazione config.py. Il parametro passato è la direzione in cui si riscontra un ostacolo, cioè un agente che si è avvicinato troppo. L'agente che invoca l'avoid cercherà prima di spostarsi nella direzione opposta a quella dell'ostacolo , se occupata tenterà nelle posizioni ortogonali e infine se anche queste fossero occupate proverà tutte e 4 le direzioni diagonali. L'avoid non viene invocata né se non ci sono ostacoli né se totalmente circondato per cui non ci sarebbe alcune direzione libera.

ALIGN: il metodo rappresenta la seconda regola del flocking. Esso utilizza una lista di valori collegati a ogni possibile direzione in cui l'agente potrebbe muoversi inizializzata a tutti 0. In pratica esamina ogni vicino nel raggio di align che non presenta l'avoidance malus e ne prende la direzione e la distanza . Per ogni direzione fa la somma dell'inverso della distanza per ogni uccello che va in quella direzione e infine restituisce l'align direction,che sarebbe la direzione che ha ottenuto la somma più alta. Da notare che andando a sommare l'inverso della distanza gli uccelli più lontani sono quelli meno influenti nel calcolo della direzione media. Il calcolo della distanza viene fatto tramite il metodo get_distance presente nel file utils.py e il raggio di align è un parametro configurabile da file di configurazione config.py .

APROACH: il metodo emula il comportamento della terza regola. Esso calcola la posizione media tra tutti gli altri uccelli nel raggio di aproach che non presentano l'avoidance malus e la utilizza per calcolare l'aproach direction. Il calcolo dell'aproach direction viene fatto tramite il metodo get_direction presente nel file utils.py e il raggio di aproach è un parametro configurabile da file di configurazione config.py .

STEP : Inizialmente setta a False l'avoidance malus. Se ci sono agenti troppo vicini intorno chiama l'avoid altrimenti calcola la direzione di align e di aproach e ne calcola la direzione risultante. Il calcolo della direzione risultante viene fatto tramite il metodo get_resulting_dir presente nel file utils.py .

```
from mesa import Agent,Model
from mesa.time import RandomActivation
from mesa.space import MultiGrid
import math
import config
import utils

possible_directions=[(-1,0),(-1,1),(-1,-1),(0,1),(0,-1),(1,1),(1,0),(1,-1)]

class FlockingAgent(Agent):

    def __init__(self, unique_id,x,y,dir,model):
        super().__init__(unique_id, model)
        self.dir=dir
        self.avoidance_malus=False

    def move(self):
        x,y=self.pos
        dx,dy=self.dir
        self.model.grid.move_agent(self,(x+dx,y+dy))

    def avoid(self,dir):  #regola 1
        dx,dy=dir
        x,y=self.pos
        avoid_direction=(0,0)

        if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x-dx,y-dy))): #opposite side
            avoid_direction=(-dx,-dy)
        else:
            if dx==0:
                if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x-1,y))):
                    avoid_direction=(-1,0)
                if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x+1,y))):
                    avoid_direction=(1,0)
                for i in [-1,1]:
                    for j in [-1,1]:
                          if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x+i,y+j))):
                              avoid_direction=(i,j)
            elif dy==0:
                if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x,y+1))):
                    avoid_direction=(0,1)
                if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x,y-1))):
                    avoid_direction=(0,-1)
                for i in [-1,1]:
                    for j in [-1,1]:
                          if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x+i,y+j))):
                              avoid_direction=(i,j)
                              break
            elif self.dir==(-1,1) or self.dir==(1,-1):
                if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x+1,y+1))):
                     avoid_direction=(1,1)
                if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x-1,y-1))):
                     avoid_direction=(-1,-1)
                for i in [-1,1]:
```

```
                    if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x+i,y))):
                        avoid_direction=(i,0)
                        break
            for j in [-1,1]:
                    if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x,y+j))):
                        avoid_direction=(0,j)
                        break
        else:
            if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x-1,y+1))):
                avoid_direction=(-1,1)
            if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x+1,y-1))):
                avoid_direction=(1,-1)
            for i in [-1,1]:
                    if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x+i,y))):
                        avoid_direction=(i,0)
                        break
            for j in [-1,1]:
                    if self.model.grid.is_cell_empty(self.model.grid.torus_adj((x,y+j))):
                        avoid_direction=(0,j)
                        break
    return avoid_direction

def aproach(self): #regola 3

    aproach_direction=(0,0)

    possible_flockmates = self.model.grid.get_neighbors(
        self.pos,
        moore=True,
        include_center=False,
        radius=config.aproach_radius
    )

    if len(possible_flockmates)!=0:

        sum_x=0
        sum_y=0
        count=0

        for birds in possible_flockmates:
                if birds.avoidance_malus:
                    continue
                pos_x,pos_y=birds.pos
                sum_x=sum_x+pos_x
                sum_y=sum_y+pos_y
                count=count+1

        if count==0:
            aproach_direction=(0,0)
        else:
            center=(math.floor(sum_x/count),math.floor(sum_y/count))
            aproach_direction=utils.get_direction(self.pos,center,self.model.grid.width,self
                                                                                .model.grid.height)

    return aproach_direction

def align(self):   #regola 2

    align_direction=(0,0)
```

```
    flockmates= self.model.grid.get_neighbors(
        self.pos,
        moore=True,
        include_center=False,
        radius=config.align_radius
    )

    if len(flockmates)!=0:

        directions=[0,0,0,0,0,0,0,0]
        no_sort_directions=[0,0,0,0,0,0,0,0]

        for bird in flockmates:
            if bird.avoidance_malus:
                continue
            index=possible_directions.index(bird.dir)
            directions[index]=directions[index]+1/utils.get_distance(self.pos,bird.pos,True,
                                                self.model.grid.width,self.model.grid.height)
            no_sort_directions[index]=no_sort_directions[index]+1/utils.get_distance(self
                          .pos,bird.pos,True,self.model.grid.width,self.model.grid.height)

        directions.sort(reverse=True)
        max_value=directions[0]
        ind=no_sort_directions.index(max_value)
        align_direction=possible_directions[ind]

    return align_direction


def step(self):

    if self.avoidance_malus:
        self.avoidance_malus=False

    neighbors= self.model.grid.get_neighbors(
        self.pos,
        moore=True,
        include_center=False,
        radius=config.avoid_radius
    )

    new_direction=self.dir

    if len(neighbors)!=0:
        if len(neighbors)!=8:
            conflict_dir=utils.get_direction(self.pos,neighbors[0].pos,self.model.grid.width,
                                                                        self.model.grid.height)
            new_direction=self.avoid(conflict_dir)
            self.avoidance_malus=True
    else:
        neighbors= self.model.grid.get_neighbors(
        self.pos,
        moore=True,
        include_center=False,
        radius=config.aproach_radius
        )
        if len(neighbors)!=0:
            align_direction=self.align()
            aproach_direction=self.aproach()
            if align_direction==(0,0):
```

```
                    new_direction=aproach_direction
                else:
                    new_direction=utils.calculate_resulting_dir(align_direction,aproach_direction)

        if new_direction!=(0,0):
            self.dir=new_direction

        self.move()

class FlockingModel(Model):

    def __init__(self,N,width,height):

        self.grid=MultiGrid(width,height,True)
        self.schedule=RandomActivation(self)
        self.running=True
        sum_x=0
        sum_y=0

        for i in range(N):

            x = self.random.randrange(self.grid.width)
            y = self.random.randrange(self.grid.height)
            dir=self.random.choice(possible_directions)
            agent = FlockingAgent(i,x,y,dir,self)
            self.schedule.add(agent)
            self.grid.place_agent(agent,(x, y))

    def step(self):
        self.schedule.step()
```

➢ UTILS.PY

File contenente le funzioni di utilità. Metodi:

GET_DISTANCE : Il metodo calcola la distanza a seconda del parametro torus. Se torus è True allora calcola la distanza toroidale, se False invece calcola la distanza non toroidale.

GET_DIRECTION: Il metodo fa uso del metodo precedente in quanto calcola la distanza toroidale e non toroidale e sceglie la direzione a seconda di quale è la distanza più piccola.

GET_RESULTING_DIR : Il metodo è concepito per far sì che la prima direzione possa al massimo essere deviata dalla seconda di 45° (es. la direzione (1,0) potrà al massimo deviare nelle direzioni (1,1) e (1,-1)).

```
import numpy as np

def get_distance(pos_1,pos_2,torus,width,height):

    x1, y1 = pos_1
    x2, y2 = pos_2

    dx = np.abs(x1 - x2)
    dy = np.abs(y1 - y2)

    if torus:
        dx = min(dx, width - dx)
```

```
        dy = min(dy, height - dy)

    return np.sqrt(dx * dx + dy * dy)

def get_direction(pos_1,pos_2,width,height):

    x1, y1 = pos_1
    x2, y2 = pos_2

    if x1!=x2:
      dx = (x2-x1)/np.abs(x2 - x1)
    else:
      dx=0

    if y1!=y2:
      dy = (y2-y1)/np.abs(y2 - y1)
    else:
      dy=0

    new_direction=(dx,dy)
    distance=get_distance(pos_1,pos_2,False,width,height)
    toroidal_distance=get_distance(pos_1,pos_2,True,width,height)

    if distance>toroidal_distance:
        new_direction=(-dx,-dy)

    dx,dy=new_direction
    new_direction=(int(dx),int(dy))  #converte gli indici da float a interi

    return new_direction

def calculate_resulting_dir(dir_1,dir_2):  #la dir_2 al massimo può solo deviare la dir_1
    resulting_dir=dir_1

    if dir_1==(0,1):
        if dir_2 in ((-1,-1),(0,-1),(-1,0)):
            resulting_dir=(-1,0)
        elif dir_2 in ((1,0),(1,-1)):
            resulting_dir=(1,1)

    if dir_1==(1,1):
        if dir_2 in ((-1,1),(-1,0),(-1,-1)):
            resulting_dir=(0,1)
        elif dir_2 in ((0,-1),(1,-1)):
            resulting_dir=(1,0)

    if dir_1==(1,0):
        if dir_2 in ((-1,1),(-1,0),(0,1)):
            resulting_dir=(1,1)
        elif dir_2 in ((-1,0),(1,-1)):
            resulting_dir=(1,-1)

    if dir_1==(1,-1):
        if dir_2 in ((-1,1),(-1,0),(-1,-1)):
            resulting_dir=(0,-1)
        elif dir_2 in ((0,1),(1,1)):
            resulting_dir=(1,0)

    if dir_1==(0,-1):
        if dir_2 in ((-1,1),(-1,0),(0,1)):
```

```
            resulting_dir=(-1,-1)
        elif dir_2 in ((1,0),(1,1)):
            resulting_dir=(1,-1)

    if dir_1==(-1,-1):
        if dir_2 in ((-1,1),(0,1),(1,1)):
            resulting_dir=(-1,0)
        elif dir_2 in ((1,0),(1,-1)):
            resulting_dir=(0,-1)

    if dir_1==(-1,0):
        if dir_2 in ((0,1),(1,1),(1,0)):
            resulting_dir=(-1,1)
        elif dir_2 in ((0,-1),(1,-1)):
            resulting_dir=(-1,-1)

    if dir_1==(-1,1):
        if dir_2 in ((1,1),(1,0),(1,-1)):
            resulting_dir=(0,1)
        elif dir_2 in ((0,-1),(-1,-1)):
            resulting_dir=(-1,0)

    return resulting_dir
```

➢ CONFIG.PY

File di configurazione con i 5 parametri configurabili.

```
birds = 150
map_size = 50
avoid_radius=1
align_radius=3
aproach_radius=5
```

➢ VIZ_MODEL.PY

File per la visualizzazione step by step del modello. Il numero di agenti e la grandezza della mappa sono parametri configurabili dal file di configurazione config.py .

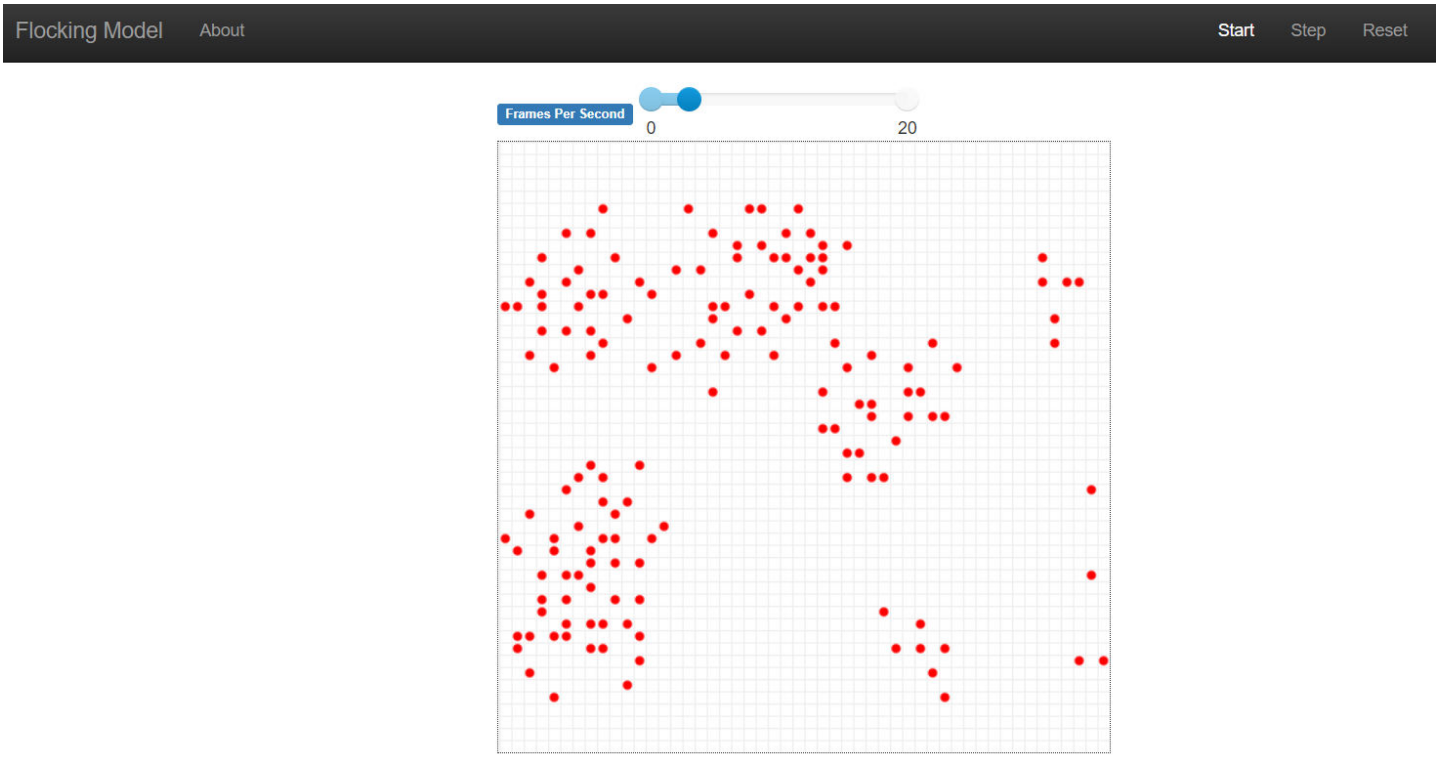
```
from model import *
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.ModularVisualization import ModularServer
import config


def agent_portrayal(agent):
    portrayal = {"Shape": "circle",
                 "Filled": "true",
                 "Layer": 0,
                 "Color": "red",
                 "r": 0.8}
    return portrayal

grid = CanvasGrid(agent_portrayal, config.map_size, config.map_size, 500, 500)
```

```
server = ModularServer(FlockingModel,
                       [grid],
                       "Flocking Model",
                       {"N":config.birds, "width":config.map_size, "height":config.map_size})
server.port = 8521 # The default
server.launch()
```

Esempio di visualizzazione



5 CONCLUSIONI

In conclusione è stata presentata la Swarm Intelligence ed evidenziati i principali comportamenti emergenti di fondamentale interesse. All'interno di quest'ultimi abbiamo esaminato il flocking, in particolare il modello enunciato da Reynolds. È stata esaminata infine un'implementazione su Mesa di modello di flocking che ha da un lato rispettato i principi cardine del flocking (le tre regole classiche ,l' assenza di un coordinamento centrale e la fondamentale importanza assunta dal concetto di vicinanza) e dall' altro ottenuto i risultati attesi : le scelte locali di ogni agente si riflettono a livello macroscopico in un coordinamento globale dando origine alla formazione dei *flock*.

6 RINGRAZIAMENTI

Un ringraziamento ai relatori che mi hanno seguito nel lavoro i quali hanno incentivato il mio interesse per l'argomento e che mi hanno garantito sempre grande disponibilità nel chiarire qualsiasi dubbio che mi si fosse presentato in corso d'opera.