Beeson

Simulatore di alveare in Jason

Giulio Crestani

giulio.crestani@studio.unibo.it

Gianluca Spadazzi

gianluca.spadazzi@studio.unibo.it

Fabio Ugolini

fabio.ugolini@studio.unibo.it

26 gennaio 2016

Abstract

Lo scopo del progetto è quello di simulare ciò che accade all'interno di un alveare. Più in particolare vengono svolte alcune delle azioni che è possibile incontrare dentro un vero alveare come la raccolta di cibo da parte di un certo tipo di api, la costruzione/riparazione delle api operaie e la nascita/morte di queste. Oltre a questo vengono aggiunti degli ostacoli che rendono più o meno difficile (regolabile dall'utente) lo sviluppo delle azioni come, per esempio, la presenza di ostacoli nella mappa in modo che le api non possano passarci sopra o la carenza di cibo all'interno dell'alveare. Per implementare ciò, ci si avvale del paradigma di programmazione ad agenti fornito da Jason.

Buona lettura.

Indice

1 Vision

Si vuole valutare se, con l'ambiente di sviluppo Jason è possibile costruire un sistema simulato che normalmente viene implementato da altri tool come per esempio NetLogo. Per fare ciò ci si avvale di vecchi esempi visti a lezione.

2 Goals

- Descrivere in modo più completo possibile il comportamento di un alveare fisico.
- Implementare in maniera più realistica possibile quelli che sono i comportamenti delle api.
- Permettere una comunicazione tra queste che sia chiara e concisa.

3 Requisiti

Si vuole che il sistema rispecchi quello che accade all'interno di un vero alveare fisico. Per fare ciò si crea una mappa di dimensioni regolabili con all'interno due elementi caratteristici:

- **Alveare**: contenente il cibo;
- **Ostacoli**: oggetti che non permettono alle api di passarci attraverso.

Più in particolare si vogliono descrivere 3 figure all'interno di questo ambiente:

- **FoodBee**: ovvero tutte quelle api che hanno il compito di raccogliere il cibo dalla mappa e depositarlo all'interno dell'alveare;
- **BuilderBee**: l'insieme di tutte le api addette alla manutenzione dell'alveare in caso di rottura;
- **ParentBee**: le api che fanno nascere le altre dopo un determinato lasso di tempo. Esse rimangono sempre all'interno dell'alveare.

Tutte queste api dovranno consumare una certa quantità di cibo presente nell'alveare.

Inoltre si prevede la gestione delle morti casuali di un'ape scelta a caso dopo che è passato un determinato lasso di tempo.

Infine si prevedono che accadano anche i seguenti eventi:

- creazione di cibo in un punto a caso della mappa in modo che le FoodBee possano accorgersene e andarlo a raccogliere non appena lo percepiscono;
- distruzione di caselle di alveare in modo che le BuilderBee possano poi ripararle;
- gestione di emergenze riguardanti la carenza eccessiva di cibo dall'alveare;
- gestione di emergenze riguardanti l'eccessiva distruzione dell'alveare;

Si vuole che tutti i parametri possano essere regolabili da parte dell'utente che, manovrandoli, può sperimentare vari scenari possibili.

4 Analisi dei requisiti e progettazione

Una volta analizzati i requisiti del sistema, bisogna scegliere come affrontarlo. Per prima cosa si considerano quali, tra tutte le cose che si devono implementare, devono essere considerate delle entità attive e quindi degli agenti e quali, invece, hanno solo bisogno di essere istanziate e quindi svolgono un'azione più passiva. Gli agenti sono sistemi computazionali in grado di compiere azioni autonome e di percepire eventi dall'ambiente in cui sono immersi. Per questo si adattano perfettamente al nostro problema. Le api infatti possono essere modellate come agenti, in quanto possiedono le loro caratteristiche, tra le quali:

- **Reactiveness**: dato che le api hanno bisogno di reagire agli stimoli dell'ambiente;
- **Situatedness**: le azioni compiute dalle api dipendono anche dal contesto in cui si trovano; è diverso essere fuori o dentro dall'alveare, devono tenere due comportamenti diversi;
- **Proactiveness**: non devono attendere solamente che qualcosa accada, ma prendono iniziativa per raggiungere i propri obiettivi (come per esempio la raccolta del cibo o la riparazione dell'alveare);
- **Social ability**: non devono agire da sole ma collaborare per raggiungere i propri obiettivi. Questo per esempio accade se, durante la raccolta del cibo, due api sono vicine, allora si scambiano informazioni con una specie di danza in modo da aiutarsi nella ricerca di altre fonti;
- **Stato**: hanno bisogno di uno stato, ovvero conoscenze che vengono aggiornate grazie allo scambio di informazioni con le altre api e con la percezione dall'ambiente.

Grazie a questo, esse possono avere diverse azioni da mettere in campo qualora si presentassero certi avvenimenti o certe emergenze. Inoltre, esse possono comunicare tra loro in modo da potersi meglio gestire a fronte dell'arrivo di nuove informazioni.

Dopo aver evidenziato quali sono gli agenti si è passati alla struttura dell'ambiente circostante. Viene creato un ambiente contenente un numero di caselle deciso da input e all'interno delle quali si è inserito un alveare e un certo numero di ostacoli. Il primo è possibile inserirlo in vari punti della mappa; più precisamente si è fatto in modo di metterlo in 5 punti considerati i più comuni: in alto a sinistra, in alto a destra, in basso a sinistra, in basso a destra ed infine al centro. Invece per quanto riguarda gli ostacoli si è deciso di generarli a caso. Un'accortezza che è stata fatta è quella di *non* posizionare gli ostacoli all'interno dell'alveare e di non posizionare un ostacolo sopra l'altro (è anche possibile scegliere a che distanza posizionare gli ostacoli tra di loro). Tutti questi parametri e molti altri che verranno citati sono manipolabili nel file *java.utilities.SysKb.java*.

Si è pensato di gestire il fatto del cibo che ogni agente ape consuma con una diminuzione di una certa quantità presa dal 'magazzino' dell'alveare ogni volta che l'agente esegue un'azione. Più precisamente si è pensato di far consumare un determinato valore di cibo in base al genere di ape. Questi valori sono tutti configurabili da input.

Inizialmente tutte le api hanno come base di conoscenza solo il luogo in cui si trova l'alveare dove nascono e nient altro.

Si passa ora alla descrizione delle varie tipologie di api implementate e ad un altro agente che si è pensato di introdurre: **God**.

4.1 Bee

Nella Figura 1 vengono rappresentate le interfacce usate e come esse si estendono.

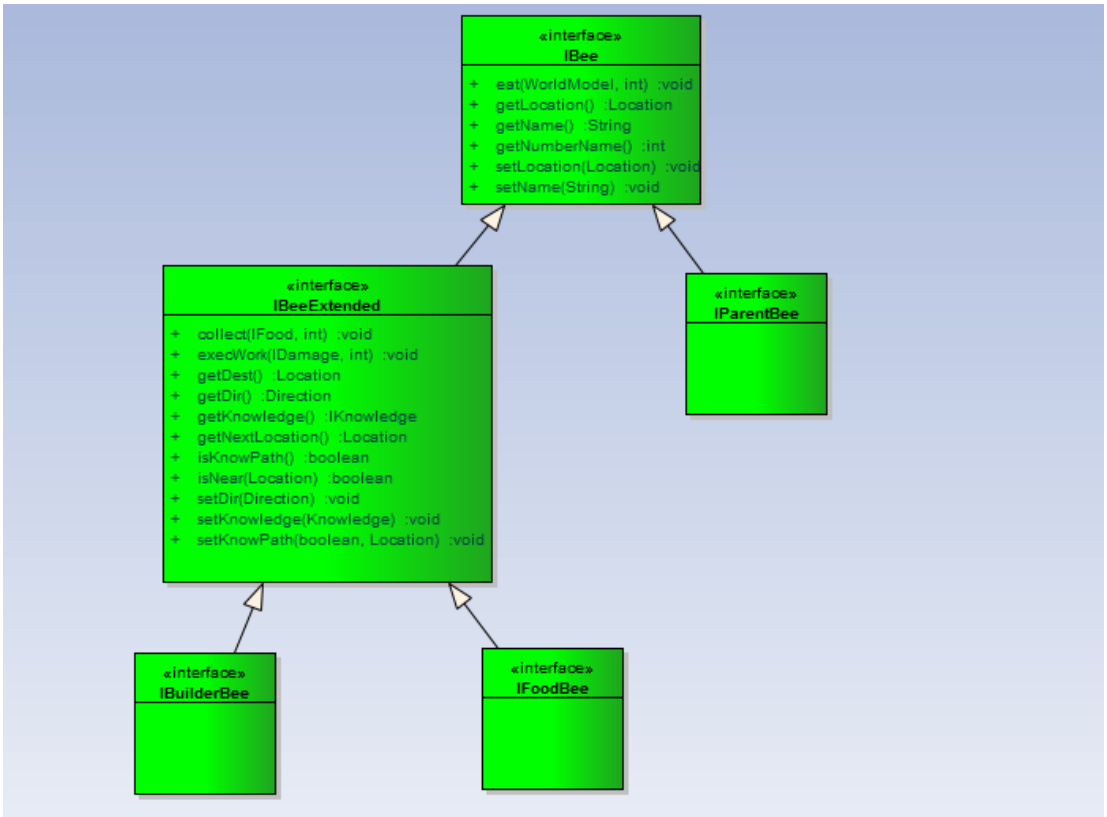

Figura 1: Interfacce usate per descrivere le api.

Come si nota, tutte le interfacce delle api estendono da una generica *IBee* che contiene i metodi condivisi tra tutte le api. Questa interfaccia viene estesa da altre due: IParentBee e IBeeExtended. La prima si riferisce già all'ape che fa nascere le altre non avendo ulteriori metodi da aggiungere alla sua struttura. La seconda invece raccoglie tutti quelli che sono i metodi comuni alle BuilderBee ed alle FoodBee e quindi tutti quelli che gestiscono i movimenti ed i compiti che svolgono. Può sembrare che alcuni metodi siano usati solo da alcuni tipi di ape (come per esempio il metodo *collect* per la BuilderBee) ma questo non è vero perché in caso di emergenza (come viene descritto nella Sottosezione 4.7) un certo tipo di ape si "trasforma" momentaneamente in un altro tipo. Le due

interfacce IBuilderBee e IFoodBee si è deciso di lasciarle vuote ma di mantenerle lo stesso nel caso in cui si voglia estendere il sistema con delle funzionalità specifiche di quel tipo di ape.

In questa parte viene descritto un comportamento che hanno in comune tutte le api che estendono IBeeExtended ovvero come muoversi nella mappa secondo una certa direzione e verso un certo punto oppure girare a caso non seguendo un determinato schema. Tralasciando la parte in cui si muovono a caso (della quale se ne parlerà in seguito) per quanto riguarda il movimento verso un certo punto, si è deciso di utilizzare il concetto di **direzione**. Le direzioni prese in considerazione sono le seguenti: NN (nord), NE (nord-est), NO (nord-ovest), SS (sud), SE (sud-est), SO (sud-ovest), OO (ovest), EE (est). Se per esempio il punto da raggiungere è in una posizione diversa rispetto all'ape, allora il comportamento che si è deciso è il seguente:

- Se la coordinata $x_{ape} = x_{target}$ allora l'ape si sposterà solo in verticale cambiando solo la coordinata y e quindi verso la direzione SS o NN;
- Se la coordinata $y_{ape} = y_{target}$ allora l'ape si sposterà solo in orizzontale cambiando solo la coordinata x e quindi verso la direzione OO o EE;
- Infine, se sia $x_{ape} \neq x_{target}$ che $y_{ape} \neq y_{target}$ allora l'ape si sposterà in diagonale cambiando entrambe le coordinate x e y. Più precisamente può spostarsi verso quattro direzioni: NE (se il punto è più in alto e più a destra), NO (se il punto è più in alto e più a sinistra), SE (se il punto è più in basso e più a destra), SO (se il punto è più in basso e a sinistra).

Inoltre può capitare che un'ape durante il suo tragitto possa imbattersi in un **ostacolo** (contrassegnato da un quadrato nero). In questo caso quello che si è deciso di fare è farle superare questo problema facendole "circumnavigare" in senso orario l'ostacolo e proseguendo verso la destinazione una volta che si trovano la strada libera.

4.2 FoodBee

Questo tipo di agente rispecchia il ruolo delle api addette alla raccolta di cibo per tutto l'alveare. Una volta che trovano una fonte di cibo ne raccolgono una certa quantità e la trasportano verso l'alveare. Una volta giunte a quest'ultimo rincominciano a ricercare cibo. Questa è un'introduzione al comportamento; una descrizione più completa la si trova nella parte di Comportamento.

4.2.1 Interazione

Questo tipo di ape interagisce sia con le altre FoodBee che con le BuilderBee. Con queste ultime interagisce nel caso in cui avvengano delle emergenze, ma questo viene descritto meglio nella Sottosezione 4.7. Con le FoodBee, invece, interagisce scambiando le informazioni che ha nella sua base di conoscenza (ovvero i luoghi della mappa dove sono presenti le varie fonti di cibo e i luoghi dove essa finisce una fonte) in modo che tutte queste sappiano dove è possibile recuperare il cibo. Più in particolare, una volta che un'ape scorge una fonte di cibo, inserisce nella sua conoscenza il luogo di questo ritrovamento e, tutte le volte che incontra una BuilderBee o una FoodBee condivide questa posizione in modo che anch'esse ne siano a conoscenza. Se poi questa fonte di cibo viene

esaurita, allora l'ape che l'ha esaurita si salva al suo interno il luogo della terminazione. Questo viene fatto in modo che, se l'ape che ha finito il cibo in un certo luogo incontra delle altre FoodBee, condivida loro l'avvenuto esaurimento della fonte così che queste ultime, se hanno al loro interno la posizione terminata, la eliminino dalla conoscenza.

Se però le FoodBee non vengono a conoscenza dell'avvenuta terminazione di una determinata fonte, allora le FoodBee per accorgersene devono necessariamente avvicinarsi a questo luogo in modo che lo capiscano ed anch'esse possono eliminare queste coordinate dalla loro lista.

4.2.2 Comportamento

Alla nascita, questo tipo di ape non ha, al suo interno, nessuna informazione dell'ambiente circostante (fatta eccezione dell'alveare, come detto poco sopra) e, più precisamente per quello che è il suo compito, non ha conoscenza di nessuna fonte di cibo nelle vicinanze. Quindi il suo comportamento iniziale è quello di girare in modo casuale fino a quando non accade uno di di questi due avvenimenti:

1. Durante il suo tragitto si accorge che, nelle vicinanze (parametro configurabile da input), è presente una fonte di cibo.
   In questo caso l'ape inserisce nella sua base di conoscenza la consapevolezza che in quel determinato punto è presente una fonte di cibo e quindi inizia a raccoglierlo e a portarlo all'alveare.

2. Incontra un'altra ape che sia FoodBee o BuilderBee.
   In questo caso interagisce con esse scambiandosi quelle che sono le basi di conoscenza. Se nella base di conoscenza che si vede arrivare sono presenti uno o più luoghi dove sono state ritrovate delle fonti di cibo, allora finisce di girare a caso ed inizia ad andare a raccogliere dalla fonte di cibo più vicina in linea d'aria.

Una precisazione per quanto riguarda il modo casuale con cui vaga. Se per caso incontra un ostacolo nel suo cammino, allora cerca di evitarlo cambiando direzione e anche quando arriva ai limiti della mappa, l'unica cosa che può fare è quella di trovare un'altra strada.

Il comportamento della FoodBee subisce un cambiamento a fronte di un'emergenza abitativa. Ogni volta che questo tipo di agente torna nell'alveare controlla se è in corso o meno questo tipo di emergenza che verrà descritta nella Sottosezione 4.7.2.

Infine in Figura 2 è possibile vedere il riassunto del comportamento appena descritto.

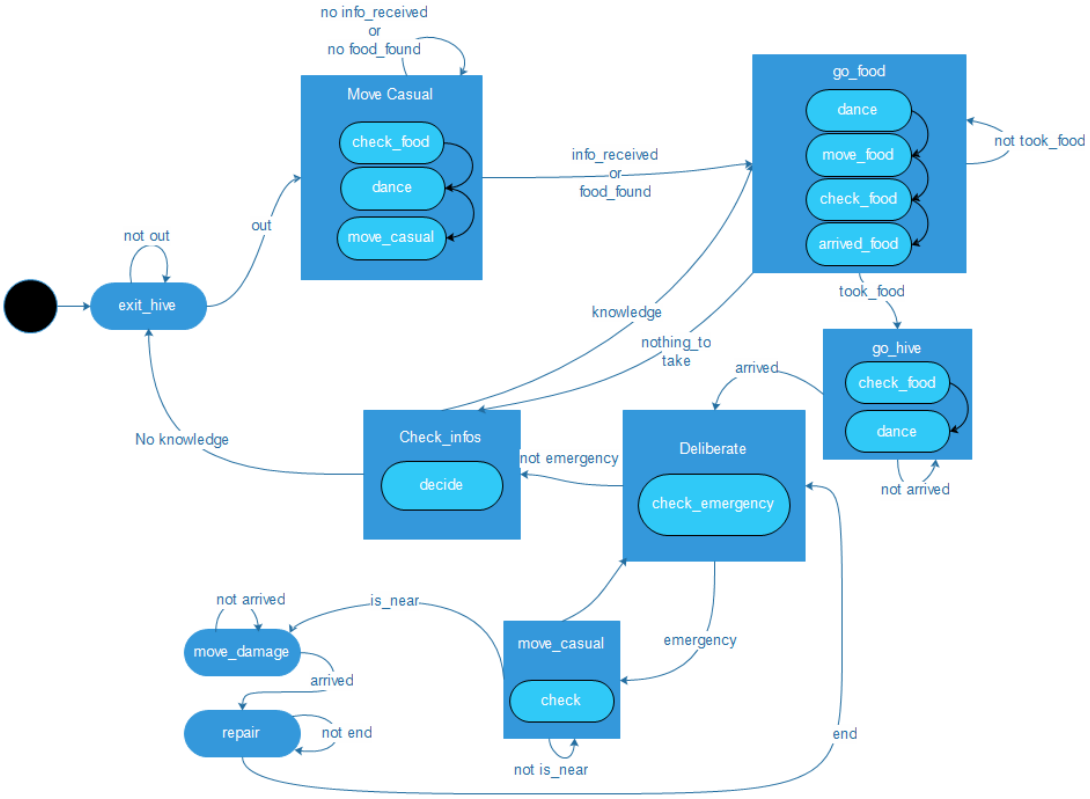

Figura 2: Riassunto comportamento FoodBee.

4.3 BuilderBee

Questo agente svolge il compito delle api operaie all'interno dell'alveare. Più precisamente se incontra delle "crepe" o delle "rotture" le sistema ricostruendo l'alveare. Per riparare non deve fare altro che posizionarsi sulla rottura e attendere che venga riparato. Graficamente una rottura viene mostrata come una casella bianca mentre l'alveare viene rappresentato come un rettangolo di celle giallo ocra.

4.3.1 Interazione

Questo tipo di agente interagisce sia con altri agenti del suo stesso tipo sia con le FoodBee ma queste interazioni avvengono solo quando è in corso un'emergenza per carenza di cibo. In questo caso, come descritto nella Sottosezione 4.7.1, le BuilderBee diventano a tutti gli effetti delle FoodBee andando a raccogliere il cibo e quindi si scambiano informazioni circa i luoghi dove sono presenti queste fonti.

4.3.2 Comportamento

Appena nate, queste api controllano che non sia in corso un'emergenza di cibo e, se è presente allora assumono il comportamento descritto nella Sottosezione 4.7.1. Se, invece, non è presente nessun tipo di emergenza allora il comportamento che tiene è semplicemente quello di rimanere all'interno dell'alveare e ricercare delle rotture (chiamati **Damage**) di questo. Per fare ciò non fa altro che muoversi a caso e, se si accorge di avere nelle vicinanze una rottura, si reca verso quest'ultima e la ripara. Si è deciso di assegnare ad ogni Damage un solo agente BuilderBee in modo che gli altri possano continuare a ricercare.

In Figura 3 è possibile vedere il riassunto del comportamento appena descritto.

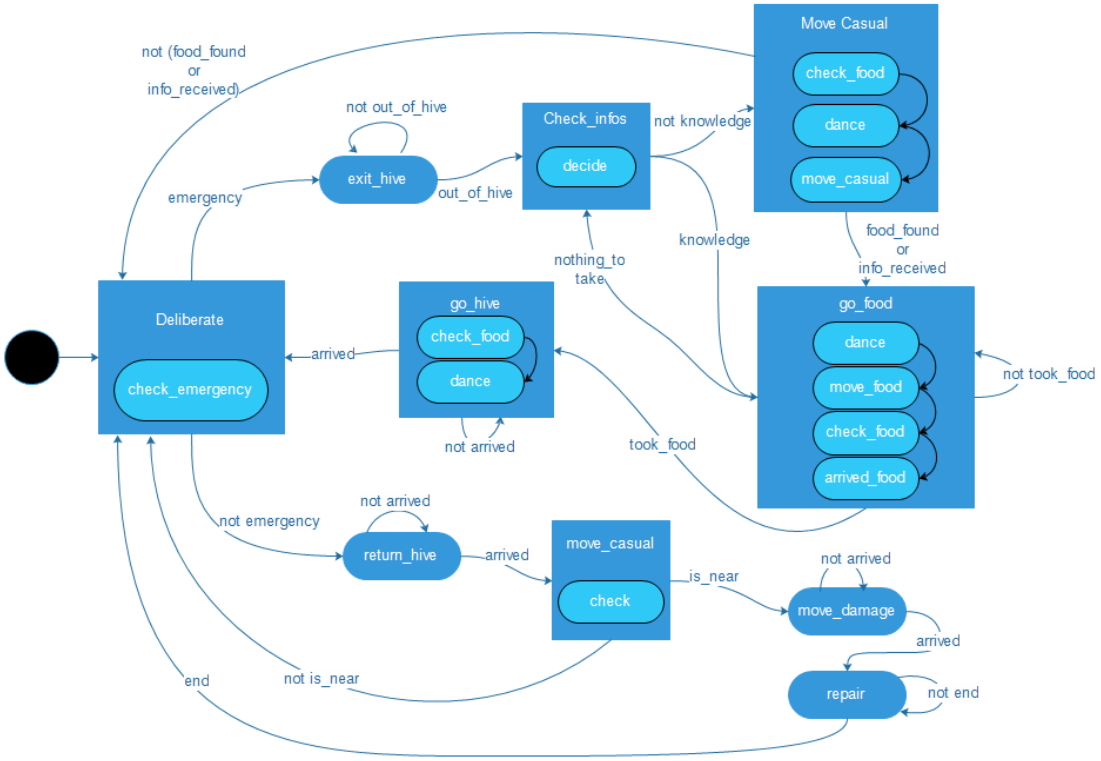

Figura 3: Riassunto comportamento BuilderBee.

4.4 ParentBee

È l'agente che rispecchia quella particolare ape che svolge il ruolo di addetta a far nascere le altre api. Essa quando nasce occupa una casella all'interno dell'alveare e da questo punto non si muoverà più.

4.4.1 Interazione

Le ParentBee non interagiscono con le altre api e non cambiano mai la loro posizione nell'ambiente.

4.4.2 Comportamento

La ParentBee per prima cosa inizia a covare e cioè, tradotto in termini più informatici, fa passare una certa quantità di tempo (che tradotto in termini biologici possiamo vederlo come il tempo di gestazione). Una volta trascorso questo tempo valuta che tipo di ape è opportuno far nascere in base a certi parametri qui sotto descritti.

1. Se il numero di ParentBee è calato (morte di una di esse) allora verrà fatta nascere una di queste;
2. Se la quantità di cibo è sotto una certa soglia allora è necessario che la raccolta di cibo sia prioritaria e perciò verrà fatta nascere una FoodBee;
3. Se invece la quantità di cibo è nella norma verrà fatta nascere una BuilderBee.

Questo elenco rispecchia anche le priorità che si sono date alle nascite.

In Figura 4 è possibile vedere il riassunto del comportamento appena descritto.

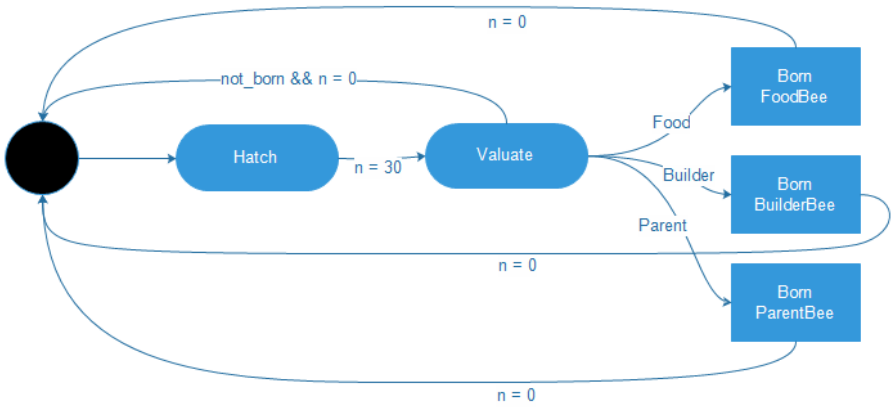

Figura 4: Riassunto comportamento ParentBee.

4.5 God

Questo agente è stato introdotto per gestire tre importanti avvenimenti che possono accadere all'interno del sistema:

- **Creazione cibo**: in una locazione casuale della mappa (dove non sia già presente una fonte di cibo o dove non ci sia l'alveare o non all'interno di un ostacolo) si va a creare un'ulteriore fonte di cibo (chiamata **IFood**).

Questo agente la crea con una certa probabilità specificata nel parametro *probCreate*. Quindi la probabilità sarà di:

$$p_{creazione} = \frac{1}{probCreate}$$

- **Distruzione alveare**: stessa cosa del punto precedente ma in questo caso invece di creare si distrugge. Si sceglie una posizione casuale all'interno dell'alveare sopra la quale non sia presente nessuna ape e dove non ci sia già una qualche rottura e, con una probabilità specificata nel parametro *probDestroy*, distrugge quella cella (quindi si viene a creare un nuovo oggetto **IDamage**). Graficamente questa distruzione non fa altro che cambiare il colore della casella dell'alveare da giallo ocra a bianca. Quindi la probabilità che venga distrutta una cella è di:

$$p_{distruzione} = \frac{1}{probDestroy}$$

- **Morte improvvisa ape**: questa azione, infine, non fa altro che "uccidere" un agente ape a caso. Il funzionamento non è con le probabilità ma, in questo caso, dopo che è passata una certa quantità di tempo, viene scelto un agente ed eliminato.

4.6 Elementi dell'ambiente

Si descrivono ora quelli che sono gli elementi passivi (e quindi non agenti) che sono presenti all'interno della mappa e che servono per aumentare o diminuire la difficoltà per le api di sopravvivere.

4.6.1 IFood

Descrivono tutte le fonti di cibo presenti nella mappa e create dall'agente God. Sono caratterizzate da una posizione nella mappa (*Location*) e da un ammontare di cibo contenuto (*amount*).

Code 1: Food.java

```
1  public class Food implements IFood {
2    private Location l;
3    private int amount;
4
5    public Food(Location l, int amount) {...}
6    /**
7     * Ritorna la posizione (Location) della fonte di cibo.
8     */
9    public Location getLocation();
10   /*
11    * Ritorna l'ammontare di cibo contenuta nella fonte.
12    */
13   public int getAmount();
14   /**
15    * Setter della quantita' di cibo.
16    */
```

```
17   public void setAmount(int amount);
18   /**
19    * Controlla Location l e' vicina al cibo.
20    */
21   public boolean isNear(Location loc);
22 }
```

4.6.2 IDamage

Queste entità descrivono invece quelle che sono le varie rotture dell'alveare prodotte dall'agente God e riparate dalle BuilderBee. Sono caratterizzate anch'esse da una locazione all'interno dell'alveare, da un ammontare del danno e da un parametro che indica se è già stato preso come obiettivo da una BuilderBee.

Code 2: Damage.java

```
1  public class Damage implements IDamage {
2    private Location l;
3    private int amount;
4    private boolean isTarget;
5
6    public Damage(Location l, int amount) {...}
7    /**
8     * Ritorna la Location del Damage all'interno dell'alveare.
9     */
10   public Location getLocation();
11   /**
12    * Getter dell'ammontare di danno.
13    */
14   public int getAmount();
15   /**
16    * Setter dell'ammontare di danno.
17    */
18   public void setAmount(int amount);
19   /**
20    * Controlla se Location l e' vicina al Damage.
21    */
22   public boolean isNear(Location l);
23   /**
24    * Ritorna un booleano che informa se il Damage e' gia' stato
          'targettato' da una Builder o no
25    */
26   public boolean isTarget();
27   /**
28    * Imposta il valore del booleano suddetto.
29    */
30   public void setTarget(boolean isTarget);
31 }
```

4.6.3 IObstacle

Questi elementi di forma quadrata sono aggiunti alla mappa per non consentire il pieno movimento delle api che raccolgono il cibo. Un'ape che incontra un

ostacolo non può passarci sopra ma deve per forza evitarlo (passandoci a fianco). Il numero di ostacoli e la loro grandezza sono configurabili (rispettivamente i parametri *numObstacles* e *obstacleDim*).

Essi sono caratterizzati da due *Location*: quella iniziale (in alto a sinistra) e quella finale (in basso a destra).

Code 3: Obstacle.java

```
public class Obstacle implements IObstacle {
  private Location init, end;

  public Obstacle(Location init, Location end) {...}
  /**
   * Controlla se la Location l e' all'interno dell'ostacolo.
   */
  public boolean isInside(Location l);
  /**
   * Ritorna la posizione iniziale.
   */
  public Location getInit();
  /**
   * Setter della posizione iniziale.
   */
  public void setInit(Location init);
  /**
   * Ritorna la posizione finale.
   */
  public Location getEnd();
  /**
   * Setter della posizione finale.
   */
  public void setEnd(Location end);
  // Altri metodi per controllare se api o ostaboli sono all-interno
      dell-ostacolo
  public boolean isInside2(Location l1, Location l2);
  public boolean isInsideForBees(Location l);
  public boolean isInside2Obs(Location l1, Location l2);
  public boolean isInsideForObs(Location l);
```

4.6.4 IHive

Infine, ma non per questo meno importante, viene descritto l'alveare. Esso è completamente configurabile da input partendo dalla posizione all'interno della mappa (parametro *hivePos*), altezza e larghezza (*hiveHeight* e *hiveWidth*) fino ad arrivare all'ammontare di cibo iniziale al suo interno (*amountFoodOnHive*).

È caratterizzato da alcuni valori:

Code 4: Hive.java

```
public class Hive implements IHive {
  private Location init, end;
  private double amountFood;
  private static Hive hive;

```

```
6  public Hive(Location init, Location end) {...}
7  /**
8   * Ritorna la posizione iniziale dell'alveare.
9   */
10 public Location getInit();
11 /**
12  * Permette di aggiungere una quantita' di cibo
13  */
14 public void addFood(int food);
15 /**
16  * Per controllare se Location l e' all'interno dell'alveare
17  */
18 public boolean isInside(Location l);
19 /**
20  * Per controllare se un ostacolo sta dentro l'alveare
21  */
22 public boolean isInside2(Location l1, Location l2);
23 /**
24  * Ritorna la coordinata x di sinistra
25  */
26 public int getMinorX();
27 /**
28  * Ritorna la coordinata x di destra
29  */
30 public int getMaxX();
31 /**
32  * Ritorna la coordinata y di sinistra
33  */
34 public int getMinorY();
35 /**
36  * Ritorna la coordinata y di destra
37  */
38 public int getMaxY();
39 /**
40  * Ritorna l'ammontare di cibo contenuto all'interno
41  */
42 public double getAmountFood();
43 /**
44  * Setter dell'ammontare di cibo
45  */
46 public void setAmountFood(double amountFood);
47 /**
48  * Controlla se l'ape contrassegnata dalla Location l e' all'interno
49  * dell'alveare.
50  */
51 public boolean isInside4Bees(Location l);
```

4.7 Gestione emergenze

In questa parte si descrivono quelli che sono i due tipi di emergenze che si possono incontrare durante il tempo di vita del sistema: carenza di cibo ed emergenza abitativa. In particolare *se si presentano entrambe* queste due emergenze si da priorità all'*emergenza del cibo*.

4.7.1 Carenza di cibo

Questo avvenimento si presenta nel caso in cui, per un qualche motivo, il livello di cibo presente nell'alveare si abbassa fino al di sotto di una certa soglia (parametro *foodEmergency* in *SysKb.java*). In questo caso la gestione dell'emergenza è la seguente: tutte le BuilderBee che stanno svolgendo il compito di riparare l'alveare cambiano ruolo momentaneamente (fino a quando l'emergenza non è finita) svolgendo anch'esse il ruolo delle api FoodBee. Iniziano quindi a spostarsi casualmente all'interno della mappa alla ricerca di cibo o, se all'interno della loro conoscenza hanno delle fonti di cibo, si dirigono verso quella posizione. Successivamente, una volta che l'emergenza è stata risolta, esse tornano a svolgere il compito per le quali sono nate.

Se la carenza di cibo si fa più pesante ed il cibo continua a scendere anche con il cambiamento di ruolo della BuilderBee in FoodBee (quindi se il cibo cala al di sotto di un'altra soglia chiamata *liveEmergency*) allora si è deciso di simulare una sorta di *carestia*. Sotto questa soglia, ogni volta che un'ape consuma del cibo c'è una probabilità che muoia (indicata in *SysKb.java* con *probToDeath*).

4.7.2 Emergenza abitativa

Questa emergenza si presenta quando il numero di *damage* all'interno dell'alveare supera una certa soglia (denominata *builderEmergency* in *SysKb.java*). In questo caso, contrariamente a quello appena descritto, sono le FoodBee che cambiano ruolo ed iniziano a svolgere il compito delle BuilderBee cominciando a girare casualmente all'interno dell'alveare e, non appena si accorgono di avere un Damage nelle vicinanze, si spostano su di esso per ripararlo.

4.8 Altri elementi utilizzati

Infine in questa sottosezione vengono presentati quei componenti che si sono messi in campo per poter meglio gestire le problematiche della interazione tra le api. Più in particolare si descrive l'oggetto messo in campo per rendere più semplice ed intuitivo lo scambio di informazioni.

4.8.1 IKnowledge

Oggetto utilizzato per simulare la conoscenza di ogni Bee. Più in particolare, ogni BuilderBee e FoodBee contengono al loro interno un oggetto di questo tipo. Questo oggetto è caratterizzato da tre campi privati e si basa su un parametri in *SysKb.java*:

1. ***listFoods***: Lista delle fonti di cibo che si credono ancora attive nella mappa;
2. ***finishedFoods***: Lista delle fonti di cibo terminate dall'ape;
3. ***howManyInfosAdded***: Semplice parametro che indica quante locazioni sono state aggiunte dall'ultima condivisione con un'altra ape;
4. ***howManyDance*** Parametro configurabile da *SysKb.java* che rappresenta il numero di volte che l'ape deve condividere ad un'altra il fatto che ha esaurito una certa fonte di cibo. Questo viene fatto perché se l'ape

continuasse a condividere il fatto che in quella posizione non è presente più cibo ma l'agente God ne ricrea uno proprio in quel punto, nessuno se ne accorgerebbe.

Code 5: Knowledge.java

```
public class Knowledge implements IKnowledge {
  private List<IFood> listFoods;
  private List<IFood> finishedFoods;
  private int howManyInfosAdded;

  public Knowledge() {...}
  /**
   * Ritorna la lista delle fonti di cibo che l'ape conosce.
   */
  public List<IFood> getListFoods();
  /**
   * Setter della suddetta lista.
   */
  public void setListFoods(List<IFood> listFoods);
  /**
   * Ritorna la grandezza della lista suddetta.
   */
  public int sizeFoodsList();
  /**
   * Per aggiungere una fonte di cibo alla lista.
   */
  public void addFoodToList(IFood f);
  /**
   * Per rimuovere una fonte di cibo dalla lista.
   */
  public void removeFoodFromList(IFood f);
  /**
   * Controlla quale tra i vari IFood all'interno della lista e' il piu'
       vicino
   * in linea d'aria.
   */
  public IFood nearestFood(Location bee);
  /**
   * Per concatenare due liste di IFood.
   */
  public void addFoodListToList(List<IFood> list);
  /**
   * Controlla se la Location e' tra gli IFood della lista.
   */
  public boolean isPresentFoodLoc(Location l);
  /**
   * Rimuove un IFood dalla lista passando la Location
   */
  public void removeFoodFromListByLoc(Location l);
  /**
   * Controlla se sono state aggiunte delle fonti di cibo nell'ultima
   * interazione.
   */
```

```
48   public boolean checkIfInfosAdded();
49   // Metodi simili ai precedenti solo per la lista di cibi terminati
50   public List<IFood> getFinishedFoods();
51   public void setFinishedFoods(List<IFood> finishedFoods);
52   public void addFinishedFoodToList(IFood f);
53   public void removeFoodFinished(List<IFood> list);
54 }
```

5 Progetto

In questa parte si descrive il progetto con le interfacce grafiche. In Figura 5 è possibile vedere come si presenta la grafica del sistema.

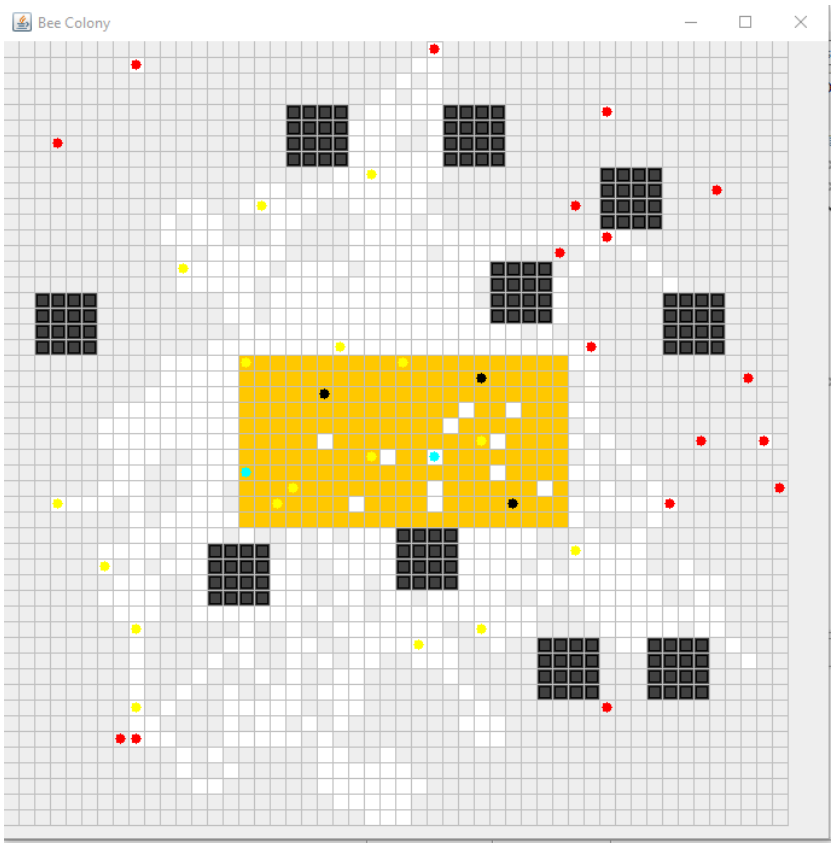

Figura 5: Screenshot del sistema in funzione.

In questa immagine si notano tutti i componenti descritti fino a questo punto:

- **BuilderBee** rappresentate con un pallino di colore azzurro;
- **FoodBee** contrassegnate da un pallino giallo;
- **ParentBee** con un pallino nero;

- **IFood** ovvero i fiori o fonti di cibo, contrassegnati da un pallino rosso;
- **IObstacle** rappresentati come quadrati di celle nere;
- **IHive** al centro della mappa di forma rettangolare e di colore giallo ocra;
- **IDamage** segnalati come celle bianche all'interno dell'alveare.

Questo screenshot è stato catturato quando non erano presenti emergenze. Come si può notare, tutte le BuilderBee sono all'interno dell'alveare che stanno cercando delle rotture per ripararle. All'interno dell'alveare sono presenti anche tutte le ParentBee che rimangono sempre immobili e non fanno altro che far nascere nuovi agenti. Le FoodBee, invece, sono sia dentro che fuori dall'alveare che ricercano e raccolgono cibo.

Queste ultime, per comunicare, utilizzano un metodo chiamato *dance* che cerca di simulare una danza proprio come accade con le api in natura. Questo metodo non fa altro che condividere le basi di conoscenza di un'ape all'altra e viene invocato ogni volta che due api sono sopra la stessa cella. Siccome il comportamento degli agenti è *proattivo*, cioè esso non rimane in attesa che le cose accadano ma deve agire a fronte di un cambiamento, se durante questa condivisione di fonti di cibo è presente un luogo più vicino all'agente, la cosa che deve fare è riconoscere che c'è l'opportunità di arrivare ad un subgoal in un tempo inferiore e quindi cambia la sua destinazione con l'IFood più vicino.

In Figura 6 si illustra invece una seconda interfaccia creata per visualizzare alcuni campi, per informare l'utente se sono in corso delle emergenze, per settare altri campi e per uccidere delle api se lo si desidera.

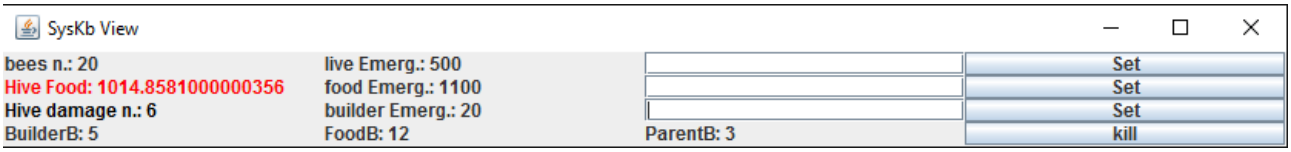

Figura 6: Screenshot dell'interfaccia grafica di supporto.

Più in particolare i campi dei quali si mostra il valore sono 9:

1. *bees*: numero di api presenti nella mappa;
2. *Hive Food*: ammontare di cibo all'interno dell'alveare;
3. *Hive damage*: numero di rotture dell'alveare;
4. *BuilderB*: numero di BuilderBee presenti nella mappa;
5. *FoodB*: numero di FoodBee presenti;
6. *ParentB*: numero di ParentBee;
7. *live Emerg.*: soglia critica di carestia;
8. *food Emerg.*: soglia di cibo sotto la quale parte l'emergenza cibo;

9. *builder Emerg.*: soglia di crepe dell'alveare sopra la quale c'è emergenza abitativa.

dei quali gli ultimi tre configurabili direttamente dall'interfaccia. In questo modo, pilotando su questi valori, è possibile osservare il mutarsi del comportamento in presenza di emergenze.

Nel caso in cui siano presenti queste ultime, il relativo campo verrà colorato di rosso per contrassegnare meglio questo fatto (come in Figura 6 nella quale è in corso l'emergenza del cibo).

Infine, in basso a destra dell'interfaccia di supporto, è presente il tasto *kill* il quale permette, se premuto, di agire sull'insieme di agenti uccidendone uno.

6 Sviluppi futuri

Vengono ora illustrati quelli che, secondo i progettisti, potrebbero essere alcuni ulteriori sviluppi al progetto presentato.

6.1 Ciclo di vita del fiore

La prima cosa che si è subito pensata è stata quella di introdurre un ciclo di vita per i vari fiori (fonti di cibo). Per rendere il sistema più realistico, si può creare un ciclo di vita per ogni fiore, proprio come in natura. Più precisamente la pianta del fiore nasce che ha al suo interno poca quantità di cibo perché deve ancora crescere e formarsi. Passata una certa quantità di tempo la pianta cresce e quindi al suo interno viene a formarsi più quantità di cibo fino a raggiungere un certo limite. Infine, come in natura che i fiori appassiscono, anche qui si vedrà calare la quantità di cibo contenuta ed infine scomparire dalla mappa.

6.2 Spore e odori del fiore

Un'evoluzione a quello che è il punto precedente potrebbe essere l'abilità di un fiore di creare delle spore che, fatte depositare nelle celle vicine, possano creare altri fiori come "figli" del primo fiore preso in considerazione. Questo può rispecchiare ancora più fedelmente quello che accade in natura.

Se si volessero seguire tutte le regole presenti in natura, un'altra abilità del fiore potrebbe essere quella di emanare un odore fino ad una certa distanza. In questo modo se un'ape percepisce questo odore allora sa che nelle vicinanze è presente un fiore.

6.3 Ostacoli mobili

Un'altra operazione che è possibile fare potrebbe essere quella di rendere mobili tutti gli ostacoli presenti nella mappa. Questo aumenta la difficoltà del sistema per quanto riguarda la progettazione ma rende più realistico lo scenario. Si immagini solo quanti ostacoli mobili è capace di incontrare un'ape quando va a raccogliere del cibo fuori dall'alveare come per esempio delle persone che si spostano o degli altri animali.

Altra cosa possibile sarebbe quella di simulare una sorta di albero (visto come ostacolo) dove, con l'andare del tempo, cresce nel tronco e quindi tradotto, l'ostacolo cresce di ampiezza mentre tutto il sistema è in funzione.

6.4 Aggiunta predatori

Un'aggiunta possibile potrebbe essere quella di inserire oltre a delle api anche degli altri animali che le cacciano e che quindi devono evitare perché altrimenti rischiano di morire. Magari i predatori sono anch'essi degli agenti che cercano del cibo (e che quindi ruberanno alle api) e che, oltre a cibarsi delle stesse cose delle api si cibano anche delle api stesse.

6.5 Simulare le varie stagioni

Come ultima cosa si è pensato ad uno stravolgimento dell'ambiente all'interno del quale vivono le api. Un possibile sviluppo potrebbe essere quello di simulare le quattro stagioni in modo che, nelle stagioni più calde (primavera e autunno) i fiori sono presenti in gran quantità mentre in inverno ne sono presenti un numero molto scarso. In questo modo le api possono sopravvivere solo se si impegnano a raccogliere il cibo al momento giusto.

7 Conclusioni

In conclusione a questa prova di simulazione in un ambiente di lavoro non nato per tale scopo, si possono fare alcune considerazioni. Prima fra tutte si nota subito che, se si vuole andare a cercare qualche cosa per documentarsi su quali comandi usare o su come funziona Jason, ci si imbatte nel nulla più totale. Perciò, per quanto riguarda questa parte, si è dovuto provare fino a quando non si aveva capito bene come funzionavano i singoli comandi.

Altra nota dolente è la gestione dell'interfaccia grafica che ci mette a disposizione Jason. Ogni tanto è capitato che in alcune celle succedessero delle cose inaspettate e che non si è capito il motivo. Un'altra cosa: se si va a ridimensionare la finestra succede il finimondo, mostrando tutte le caselle e le api distorte.

Su altri software di simulazione (come ad esempio NetLogo), si abbattono i tempi per la creazione e la gestione dei problemi dell'interfaccia grafica. Però si è scelto Jason per poter testare la potenza della programmazione ad agenti e che permette, sorvolando quelli che sono i problemi di interfaccia grafica, un'estensione molto più ampia per quanto riguarda degli sviluppi futuri, come per esempio la creazione di un sistema ad agenti distribuito su più macchine (grazie anche al supporto di Java che permette di gestire tutto al meglio). Invece, nei software basati sulla simulazione, si è vincolati solo a quest'ultima non garantendo nulla di più se per caso si volesse ampliare il sistema.

D'altra parte però, Jason ci permette di implementare tutti i concetti degli agenti BDI consentendo la completa comunicazione tra questi e la percezione dell'ambiente. Altro punto a favore è il fatto che è in grado di gestire un numero abbastanza elevato di agenti e quindi di api.