

ANTS: AI CHALLENGE 2011

Progetto di Ingegneria della conoscenza

Studenti:

- Debellis Rocco (Mat.: 656530) – rocdebellis@gmail.com
- Lorusso Claudia (Mat.: 654441) – lorusso.claudia@outlook.it

Professore: Nicola Fanizzi



SOMMARIO:

NOTA INIZIALE.....	4
ANTS: AI CHALLENGE 2011	5
1. Introduzione	5
Informazioni sulla Challenge	6
Vincitore della challenge.....	6
Descrizione del problema	7
Condizioni per la terminazione del gioco	8
Turni.....	8
Fasi	9
Accumulazione di punti.....	10
Regole di interruzione	11
Impostazioni di gioco	12
2. Informazioni preliminari	13
Azioni autonome di una formica	13
Risoluzione della battaglia	14
• Focus nella risoluzione della battaglia.....	14
Esempio: focus nel combattimento.....	16
• Descrizione tecnica della battaglia	17
• Possibili scenari della battaglia.....	18
1) One-on-One	18
2) Two-on-One.....	18
3) One-on-One-on-One.....	18
4) Ant Sandwich	19
5) One-on-Two-on-One.....	19
6) Wall Punch.....	19
Input.....	20
Informazioni sui parametri	20
Informazioni sul turno	21
Informazioni in input e Fog of War.....	23
Output e concetto di <i>ordine</i>	24
La mappa ed i possibili spostamenti.....	25
Il nostro concetto di <i>tile</i>	27
Esempio: rappresentazione e contenuto di una Tile	29
3. Il nostro bot: MarksHive.....	30
Fun Fact: la natura del nome del nostro bot.....	32
Moduli e Task	33
Metodi di ricerca	34
• Metodi di ricerca: BFS modificata	37
Esempio: frontiera BFS modificata e ordini	38
• Metodi di ricerca: A* modificata	44
• Metodo di ricerca statica e gli Offset	46
Esempio: gli Offset e la ricerca statica	47
A. Modulo di combattimento.....	48
Riconoscimento di situazioni	49
Generatore di mosse	50
Funzione Euristica di Valutazione	51
Ricerca spazio-stato min-MAX	52
B. Modulo di difesa.....	53

Progetto Ingegneria della Conoscenza – Debellis Rocco, Lorusso Claudia

Esempio: modulo di difesa.....	54
C. Modulo di riproduzione (e di raccolta del cibo)	56
D. Modulo di attacco.....	57
E. Modulo di esplorazione.....	58
Modulo di esplorazione: zone inesplorate	58
Modulo di esplorazione: zone invisibili	58
Modulo di esplorazione: zone prioritarie.....	59
Modulo di esplorazione: distanziamento	59
F. Modulo di visione	60
G. Modulo del tempo	61
a. Tempo modulo di combattimento	61
b. Tempo negli altri moduli.....	61
4. Conclusioni	63

Nota iniziale

Il documento è volto all'esplicazione dettagliata del funzionamento del software e delle tecniche adottate per lo sviluppo del progetto di Ingegneria della Conoscenza, a.a. 2019 – 2020.

I componenti del gruppo, quali fautori del progetto, sono i seguenti:

- Debellis Rocco (mat. 656530) –
e-mail UNIBA: r.debellis10@studenti.uniba.it
e-mail iscrizione piattaforma ADA: rocdebellis@gmail.com
- Lorusso Claudia (mat. 654441) –
e-mail UNIBA: c.lorusso36@studenti.uniba.it
e-mail iscrizione piattaforma ADA: lorusso.claudia@outlook.it

È opportuno specificare che:

1. Si è deciso di includere illustrazioni, tabelle ed esempi – la maggior parte dei quali da noi ideati – in modo da alleggerire la lettura e da rendere più facilmente comprensibili i concetti da noi espressi;
2. Per la scrittura del documento si è fatto riferimento alle informazioni rilasciate sul sito <http://ants.aichallenge.org/> ed al manuale “Artificial Intelligence, Foundations of Computational Agents” di David L. Poole e Alan K. Mackworth, se non alle dispense rilasciate sul sito ADA dal professore Nicola Fanizzi, nella sezione dedicata al corso di Ingegneria della Conoscenza.

Per l'avvio del software si consulti il file intitolato “README.me”.

Ants: AI Challenge 2011

1. Introduzione

Il progetto si basa su di una challenge istituita nel 2011, raggiungibile tramite l'url <http://ants.aichallenge.org/>, e sponsorizzata da Google denominata “Ants: AI Challenge” dove *Ants* significa *formiche*.

La challenge riguarda infatti la costituzione di una colonia di formiche in grado di effettuare diversi compiti, anche detti *task*, al fine di prevalere sulle colonie nemiche.

Abbiamo scelto una sfida che riguarda questo *topic* perché chi più di una colonia di formiche è la mera rappresentazione in natura di un agente che possiede intelligenza ed è in grado di adattarsi al cambiamento, fronteggiando varie sfide per la difesa del loro nido e la raccolta del cibo.

Abbiamo dunque deciso di combinare i due mondi, *naturale* e *virtuale*.

La vera sfida è quella di progettare ed analizzare un sistema intelligente in grado di adattarsi ai cambiamenti ambientali, di imparare dall'esperienza e di effettuare scelte appropriate in vista di un obiettivo.

Il sito sopracitato mette a disposizione uno *starter-kit* in modo da poter ottenere un programma, scritto nel linguaggio prediletto, che permetta alla colonia di formiche di interagire con l'ambiente esterno senza però effettuare scelte effettivamente ponderate.

Il linguaggio da noi prescelto è Java, anche se per poter testare il programma è necessario disporre di un interprete Python (consultare il file README per maggiori informazioni).

Lo starter kit rappresenta soltanto un punto di partenza per la realizzazione di un agente che però non è in grado di prendere decisioni utili al raggiungimento del traguardo.

Il nostro compito è stato quello di far assumere all'agente, per ogni *task*, atteggiamenti intelligenti e strategici volti al raggiungimento dell'obiettivo prefissato ossia l'annientamento delle colonie nemiche.

Informazioni sulla Challenge¹

La **AI Challenge** è una competizione basata sulla creazione di un'intelligenza artificiale indetta dal **Club di informatica dell'Università di Waterloo**.

Inizialmente il contest era rivolto ai soli studenti dell'università di Waterloo.

Nel 2010, la sfida è stata sponsorizzata da Google che gli ha permesso di estenderla al pubblico in generale.

Ogni partecipante doveva scrivere un programma autonomo per sfidare nel gioco un avversario, per poi caricare il codice su di un server.

Il server del contest ha utilizzato l'algoritmo di classificazione **Trueskill** per i **matchmaking** e per generare la classifica dei vincitori.

Dopo il match, gli spettatori potevano visionare lo scontro usando un browser. Il contest era open source.

Abbiamo eseguito numerosi tentativi per rendere operativo a distanza di quasi 10 anni il server del contest:

al seguente indirizzo <https://github.com/roccodebellis/aichallenge.git> è difatti disponibile la repository *fork* di quella originaria contenente il server del contest comprensiva di modifiche ed aggiornamenti.

Purtroppo, questi ultimi, non ci hanno comunque permesso di renderlo nuovamente operativo, ma c'è tutto il necessario per riprovare in futuro.

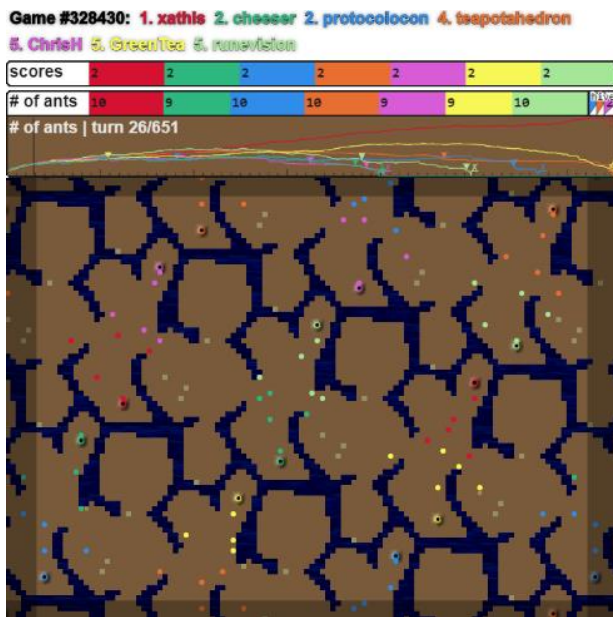
Vincitore della challenge

Il vincitore della challenge è stato Mathis Lichtenberger il quale ha partecipato con il nickname **Xathis**.

A seguito di una breve ricerca sul web, siamo riusciti a recuperare il suo bot per poter far allenare e potenziare il nostro.

¹ https://en.wikipedia.org/wiki/AI_Challenge

Descrizione del problema²



Ants è un gioco di strategia multiplayer, per un minimo di 2 ad un massimo di 10 giocatori, che ha luogo su di un terreno rettangolare i cui ostacoli sono individuati da blocchi di acqua ed in cui il cibo viene generato casualmente.

Ogni giocatore possiede una o più *hill* ossia i nidi da cui hanno origine le formiche.

L'*obiettivo di ogni giocatore* è di andare alla ricerca e sterminare le *hill* nemiche

mentre difendono le proprie.

I giocatori devono anche raccogliere cibo in modo da riprodurre nuove formiche; nonostante ciò, se le *hill* del giocatore sono state distrutte, sebbene si continui a raccogliere cibo, le sue formiche **non** saranno più in grado di generarne delle nuove.

L'*obiettivo* è quello di creare un programma, ossia un *bot*, che giochi al gioco di Ants il più intelligentemente possibile.

Un **bot** è un programma che legge gli input contenenti le informazioni sulle porzioni di mappa, anche dette *tile*, che può vedere in quell'istante e restituisce come output gli *ordini* che includono le mosse delle sue formiche nella mappa.

Ogni formica può vedere soltanto l'area che la circonda entro un certo raggio, detto *view radius* ossia il raggio di visione.

Di conseguenza i bot inizieranno ogni turno con una visione **parziale** della mappa.

Un *bot* può assegnare al più un ordine per ogni formica durante ogni singolo turno.

Ogni *ordine* specifica una formica, identificata dalle sue coordinate nella mappa, e la direzione verso cui spostarla: Nord, Est, Sud oppure Ovest. Una volta che l'ordine viene eseguito, le formiche si spostano di una *tile* nella data direzione.

² http://ants.aichallenge.org/problem_description.php

Condizioni per la terminazione del gioco

Talvolta un bot crasha o va in time-out determinando la fine del gioco di *quel* giocatore.

Le sue formiche resteranno semplicemente immobili nella stessa posizione e potranno comunque essere attaccate ed uccise.

La partita **termina** quando un giocatore riesce a distruggere tutte le *hill* nemiche oppure tutte le formiche avversarie (disponendo ancora di almeno una *hill*).

Il gioco potrebbe terminare anche nel caso in cui si vada oltre un certo numero di turni (il gioco potrebbe anche terminare prematuramente se nessun bot fa progressi).

Turni³

Il gioco è basato su *turni*.

Al *turno 0* vengono fornite ad ogni bot delle informazioni di partenza sul gioco; queste includono le dimensioni della mappa, il numero massimo di turni ed il tempo di inizio di ogni turno.

Dopo che il bot ha processato tali dati, deve restituire “go”.

Dopo che tutti i bot sono pronti, inizierà il gioco.

Quando tutti i bot avranno terminato di effettuare le loro computazioni ed avranno comunicato di essere pronti ad iniziare, il motore del gioco effettuerà i seguenti passi ripetutamente:

- Manda lo stato del gioco ai giocatori;
- Riceve gli ordini dei giocatori;
- Esegue le *fasi* (vd. sezione successiva) ed aggiorna lo stato del gioco;
- Controlla se si siano verificate le [condizioni per la terminazione del gioco](#).

Un *turno* è definito dai passi riportati qui in alto; questi sono eseguiti sino ad un numero massimo di turni dopodiché il gioco termina.

I bot inizieranno ogni turno con una visione **parziale** della mappa: ad ogni turno, infatti, si forniscono soltanto le informazioni correnti per le *tile* che le formiche ancora in vita del proprio agente sono in grado di osservare.

³ <http://ants.aichallenge.org/specification.php#Turns-and-Phases>

Fasi

Dopo aver ricevuto tutti gli *ordini* dai giocatori, il motore del gioco aggiorna lo stato del gioco portandolo nel turno successivo. Ciò avviene in 6 *fasi*:

- **far muovere** tutte le formiche (le formiche che collidono sulla stessa *tile* muoiono), ossia eseguire gli ordini;
- **attaccare** le formiche nemiche se si trovano nel raggio di visione;
- **radere al suolo** le *hill* nemiche;
- **riprodurre nuove formiche**;
- **raccogliere** cibo (il cibo scompare se due nemici sono entrambi nelle sue vicinanze);
- **riprodurre nuovo cibo**.

A seguito di queste fasi, il bot riceve lo stato del gioco successivo ed ordina nuove mosse.

Accumulazione di punti

L'**obiettivo** del gioco è quello di ottenere lo score più alto.

Si possono ottenere punti attaccando e difendendo le *hill*:

- Ogni bot inizia con un 1 punto per ognuna delle proprie *hill*;
- Radere al suolo una *hill* nemica vale 2 punti aggiuntivi;
- Perdere una *hill* comporterà la perdita di un punto.

Ciò significa che se non si attaccano *hill* nemiche e se si perdono tutte le proprie *hill* si terminerà con un punteggio pari a 0.

Se il gioco termina con un solo bot in gioco, tale bot verrà premiato assegnandogli 2 punti per ogni *hill* nemica che non è stata rasa al suolo.

Ciò è fatto in modo che l'unico bot in gioco non venga penalizzato nel caso in cui un altro bot crashi o vada in time out.

Questi sono chiamati *punti bonus*:

- 2 punti bonus per ogni *hill* all'unico bot rimasto in gioco;
- 1 punto per ogni *hill* da sottrarre al proprietario delle *hill* rimanenti.

Regole di interruzione

Per garantire che sul server vengano eseguiti match significativi, esistono diverse regole che permettono l'abbreviazione delle partite per i seguenti motivi.

- **Il cibo non viene raccolto:** Se un match è costituito da bot che non sono in grado di raccogliere cibo, il match viene interrotto - presupponendo che si tratti di bot principianti o bot poco sofisticati. La terminazione avviene nel caso in cui la quantità totale di cibo disponibile sia pari al 90% del totale distribuito per 150 turni consecutivi;
- **Unico sopravvissuto:** Se nel gioco è rimasto in vita solo 1 bot, il gioco viene interrotto: ciò avviene se tutti gli altri bot sono stati completamente eliminati (nessuna formica sulla mappa) o si sono bloccati (crash, nessun nuovo ordine impartito). Si assegna un bonus per ogni *hill* nemica rimanente all'ultimo bot e si sottrae un *malus* ai proprietari delle *hill*;
- **Stabilizzazione del rango:** Se non ci sono bot, con *hill* rimanenti, che possono guadagnare abbastanza punti per dominare, il match viene interrotto. I bot senza *hill* rimanenti non continuano ad ottenere punti, risultando ormai fuori dal gioco; quindi i punti vengono assegnati solo nel caso in cui il giocatore, bot, abbia almeno una *hill*. Per ognuno di questi ultimi, il punteggio minimo (calcolato assumendo che possa catturare tutte le *hill* nemiche rimanenti) viene confrontato con il punteggio minimo di ciascun avversario (calcolato assumendo che perderebbe tutte le *hill* rimanenti). Se una qualsiasi differenza di punteggio può portare a superare o risolvere un pareggio, il match continua. Se nessuno dei bot soddisfa questi criteri il match viene interrotto.

Ad esempio, in un match con 4 player, se il bot A rade al suolo le colline dei bot B e C, i punteggi sono A=5, B=0, C=0, D=1. Anche se il bot D rade al suolo la collina del bot A il punteggio sarebbe A=4 e D=3, quindi D non può fare meglio del 2° posto ed il match termina.

Un ulteriore esempio, per lo stesso match a 4 player, se il bot A rade al suolo la collina del bot B e il bot B ha ancora formiche, è libero di tentare di guadagnare punti. Ma dopo che il bot A ha raso al suolo la collina del bot C, non vi è più nessuna possibilità di ottenere altri punti anche se potrebbe terminare con un punteggio di A=4, B=2, C=0 e D=0.

- **Limite turni raggiunto:** C'è un limite massimo di turni per ogni mappa. A ogni bot viene assegnato questo limite. Raggiunto questo limite il match termina.

Impostazioni di gioco⁴

Di seguito si elencano le impostazioni date al motore del gioco determinanti le opzioni che caratterizzano un match.

Alcune di queste impostazioni possono essere modificate dinamicamente prima di eseguire il software (si veda il file README).

In genere, il limite di turni è regolato in base alla mappa prescelta.

Di seguito sono elencate le opzioni di gioco:

turns	150	<i>Numero massimo di turni</i>
loadtime	3000	<i>Durata (in millisecondi, ms) per l'avvio</i>
turntime	50000	<i>Durata (in ms) di ogni turno</i>
viewradius2	55	<i>Raggio di visione elevato al quadrato</i>
attackradius2	5	<i>Raggio di attacco elevato al quadrato</i>
spawnradius2	1	<i>Raggio di di raccolta cibo elevato al quadrato</i>
food_rate	5, 11	<i>Tasso di rilascio cibo</i>
food_turn	19, 37	<i>Turni di rilascio cibo</i>
food_start	75, 175	<i>Percentuale di cibo sparsa sul campo</i>
food_visible	3, 5	<i>Quantità assicurata di cibo iniziale</i>
food	symmetric	<i>Metodo di spargimento cibo (random, simmetrico, etc.)</i>
attack	focus	<i>Metodo di attacco (usato dalla macchina)</i>
kill_point	2	<i>Punteggio assegnato per l'annientamento di una hill</i>

⁴ http://ants.aichallenge.org/game_settings.php

2. Informazioni preliminari

Affinché si possa capire appieno la nostra implementazione è bene rivedere e specificare nel dettaglio determinati concetti.

Azioni autonome di una formica

Una formica è una creatura davvero semplice.

Essa è di fatto in grado di eseguire soltanto **due azioni autonome** senza che le vengano assegnati comandi esterni.

La **prima** è **riprodursi** (*spawning*) ossia creare un'altra formica dopo aver raccolto del cibo che le è abbastanza vicino.

Questa *azione* è di certo alquanto *complessa* per una formica ed è per questo che il cibo deve essere molto vicino per poter essere raccolto (*spawning radius*).

I *risultati* però sono alquanto *semplici*: se il cibo è abbastanza vicino, questo si trasforma in un'altra formica della stessa fazione/id di quella che l'ha raccolto.

L'unica eccezione è quando formiche di fazioni diverse riescono ad avvicinarsi entrambe abbastanza al cibo nello stesso momento; in tal caso il cibo viene distrutto quando entrambe provano a raccoglierlo.

La **seconda** azione che una formica può eseguire autonomamente è quella di **combattere** contro le loro avversarie.

Questa è un'*azione* ancor più *semplice* rispetto alla "riproduzione" perciò può essere eseguita ad una maggiore distanza, *attack radius*.

Nonostante ciò, i *risultati* possono essere un po' più *complessi* da capire rispetto all'azione del "riprodursi".

Per comprenderne al meglio le dinamiche si veda la sezione successiva dedicata alla [risoluzione della battaglia](#).

Risoluzione della battaglia⁵

Per poter affrontare il [modulo del combattimento](#), è prima necessario comprendere le dinamiche di una battaglia.

Nello specifico, è bene analizzare:

- [Il concetto di focus nella risoluzione della battaglia](#);
 - [Descrizione tecnica della battaglia](#);
 - [I possibili scenari della battaglia](#).
- Focus nella risoluzione della battaglia

Per introdurre la risoluzione della battaglia e, nello specifico, per poter trattare del concetto di *focus* è bene dare uno sguardo al seguente esempio.

A è una formica avente **ID = 3** mentre **B** è una formica avente **ID = 2** (ciò implica che le due formiche sono nemiche).

1. **A** sarà uccisa da **B** se **B**, nel suo raggio di attacco, è circondata da un *numero minore o uguale* di formiche ad essa nemiche (aventi $ID \neq 3$) rispetto al numero di nemici da cui è circondata **A** (\Rightarrow che il numero di nemici da cui è circondata **A** è maggiore rispetto al numero di nemici da cui è circondata **B**);
2. **B** ucciderà **A** se **A** è circondata (nel suo raggio di attacco) da un *numero maggiore o uguale* di nemici rispetto al suo target.

Come già anticipato nella sezione riguardante le [azioni autonome](#) delle formiche, ogni formica è in grado di **combattere** autonomamente contro le loro avversarie.

Questa è un'azione ancor più semplice rispetto alla "riproduzione" perciò può essere eseguita ad una maggiore distanza, *attack radius*.

Nonostante ciò, i risultati possono essere un po' più complessi da capire rispetto all'azione del "riprodursi".

Il *caso più semplice* è quello in cui due sole formiche avversarie sono abbastanza vicine l'una all'altra. Si *focalizzeranno/concentreranno* entrambe soltanto l'una sull'altra.

Poiché il loro **focus** è equo, finiranno entrambe per essere ammazzate.

⁵ http://ants.aichallenge.org/specification_battle.php

Quando una sola formica incontra due formiche nemiche allo stesso tempo, *dividerà* il suo *focus* su entrambi i nemici.

Gli avversari, ovviamente, potranno entrambi concentrarsi completamente sulla formica mentre quest'ultima, essendo più distratta, morirà lasciando le altre ancora in vita.

Di conseguenza è possibile osservare che una formica può *sopravvivere* ai suoi nemici se ha più *focus* rispetto a loro.

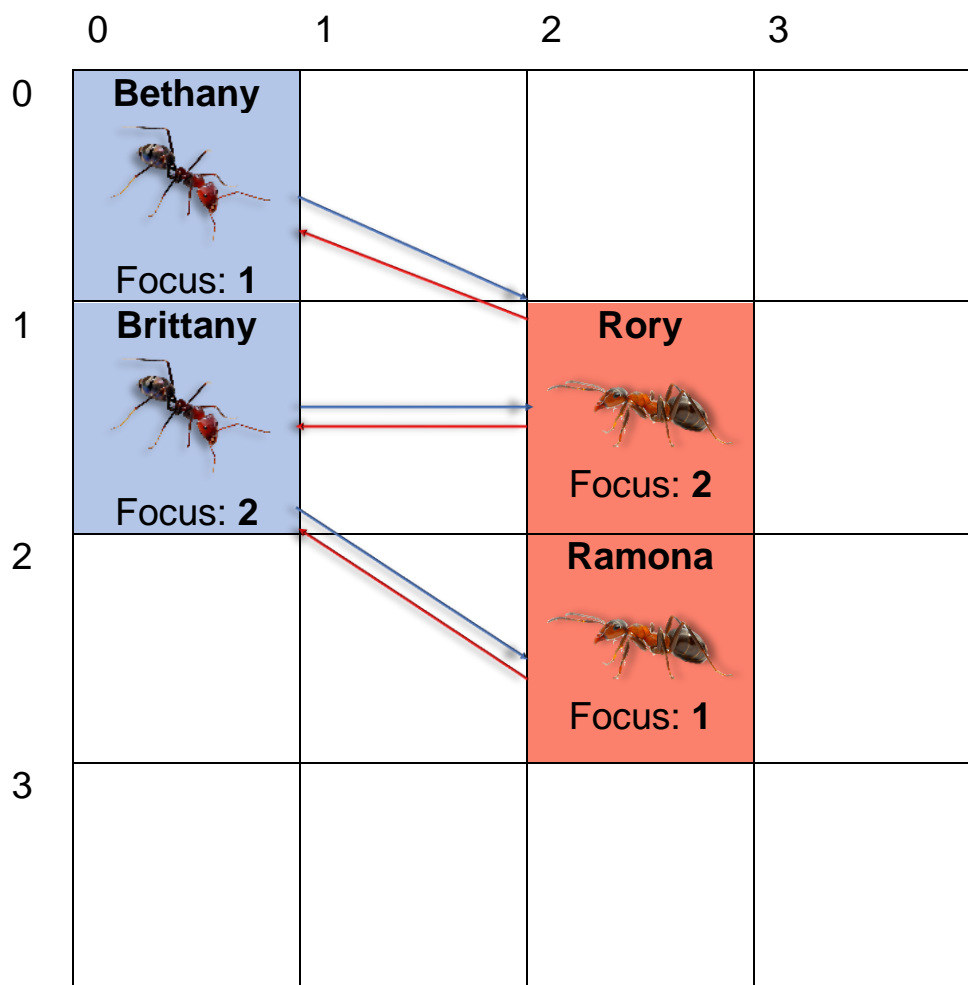
Se il *focus* è minore o uguale rispetto agli avversari, allora la formica morirà.

Le formiche non distinguono i colori dei nemici. Se le formiche di colori differenti, ad esempio verde, rosso e blu, si trovano ognuna nel raggio di attacco dell'altra, allora ognuna di loro avrà il *focus* diviso in 2 e ne conseguirà la morte di tutte e tre le formiche.

Ma, se una formica verde ha una formica rossa da un lato ed una formica blu dall'altro lato, supponendo che la formica rossa e quella blu non si trovino l'una nel raggio di attacco dell'altra, allora la formica verde morirebbe poiché avrebbe il *focus* diviso sia su quella rossa che su quella blu; queste ultime invece avranno si focalizzeranno totalmente su quella verde e rimarranno dunque in vita.

Per una maggiore comprensione del concetto di *focus* si dia uno sguardo all'esempio che segue.

ESEMPIO: FOCUS NEL COMBATTIMENTO



Si considerino due formiche **blu**, **Bethany** e **Brittany**, e due formiche **rosse**, **Rory** e **Ramona**, che si spostano nella mappa.

Dopo essersi mossa, Brittany è abbastanza vicina ad entrambe le formiche rosse per un combattimento; Bethany, invece, può combattere solo con Rory.

La situazione qui descritta è che Brittany e Rory hanno il **focus** diviso tra due avversari.

Bethany e Ramona, invece, sono in grado di concentrarsi solo su di un avversario, il cui focus è diviso su più nemici.

In questo modo moriranno sia Brittany che Rory mentre Bethany e Ramona sopravvivono alla battaglia.

NB. L'immagine è puramente a scopo illustrativo: non capiterà mai che vi siano meno di due *tile* di distanza tra due formiche nemiche, in accordo all'*attack radius*, senza che siano già entrate in modalità combattimento.

Ogni combattimento funziona allo stesso modo, se una formica ha il focus diviso su di un numero di nemici minore rispetto a quello su cui è diviso il focus dei suoi avversari, allora sopravvive.

Di seguito si analizzano i casi specifici.

- **Descrizione tecnica della battaglia**

Ogni formica può avere nel suo raggio diverse formiche nemiche. In tal caso la si può considerare “occupata” con il compito di combattere queste formiche.

Ogni formica può avere un numero differente di nemici con cui è “occupata”.

Le formiche che sono occupate da un numero minore di nemici sono considerate *meno focalizzate* sul loro task di uccidere quei nemici rispetto ad una formica che è occupata da più nemici.

Le formiche che sono equamente occupate sono in grado di uccidersi a vicenda.

- La formica A è nel raggio di attacco di B;
- A è occupata/sta combattendo con 2 nemici;
- B sta combattendo contro 4 nemici;
- B ha più pensieri per la testa, è più distratta e, dunque, meno forte rispetto ad A;
- A può quindi uccidere B;
- Nel frattempo, potrebbero accadere altre cose con C (sin ora ci si è focalizzati su A e B)
 - C ucciderebbe entrambe A e B anche se A e B non fossero l'una vicino all'altra.

	0	1	2	3
0				C
1	C	A	B	C
2				C
3				

Ogni formica potrebbe avere più di una formica in grado di ucciderla che potrebbe avere più di una formica da cui può essere uccisa. *Una formica può contemporaneamente sia essere uccisa che uccidere nello stesso turno.*

	0	1	2	3
0				1
1	1	2	4	1
2				1
3				

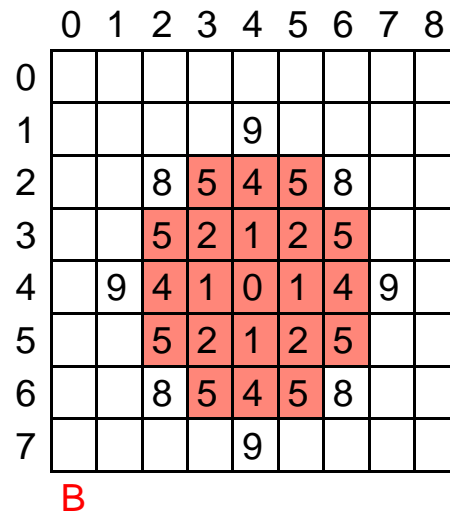
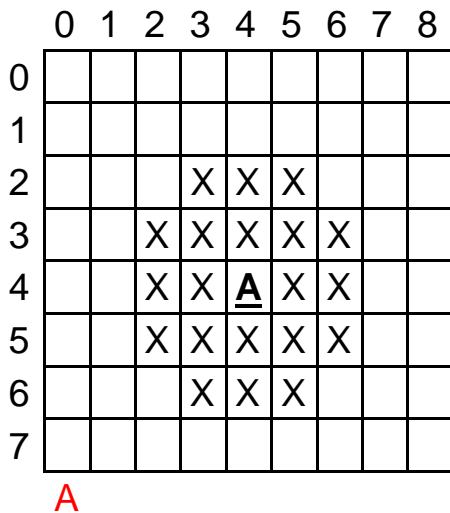
Si osservi la mappa a sinistra in cui è riportato il numero di nemici da cui ogni formica è circondata.

Di conseguenza, se ogni formica è vicina ad un nemico con numero minore o uguale, morirà (come è possibile osservare nella mappa a destra).

	0	1	2	3
0				C
1	C	X	X	C
2				C
3				

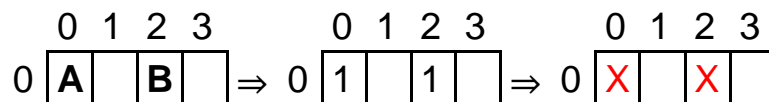
- Possibili scenari della battaglia

Tutti i possibili scenari assumono che $attackradius2 = 5$, che ha la forma raffigurata nella figura **A** (la figura **B** illustra ulteriori raggi oltre ad $attackradius2$; in rosso sono riportate le caselle incluse in un raggio al quadrato pari a 5):



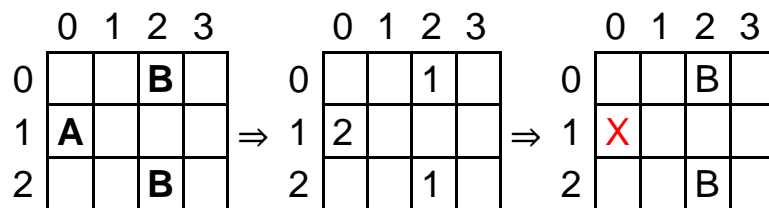
1) One-on-One

Muiono entrambe le formiche:



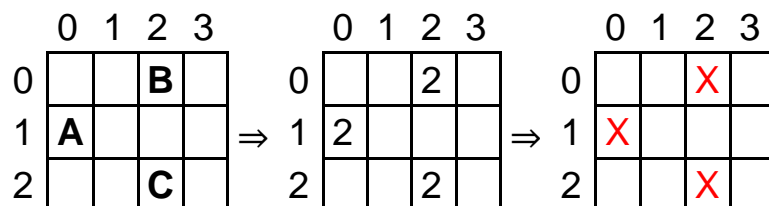
2) Two-on-One

A muore:



3) One-on-One-on-One

Muiono tutte le formiche:



4) Ant Sandwich

La formica **B**, al centro, muore mentre **A** e **C** sopravvivono:

	0	1	2	3	4		0	1	2	3	4		0	1	2	3	4
0						⇒						⇒					
1	A		B		C		1	1		2		1		1	A		C
2							2						2				

5) One-on-Two-on-One

B e **C** muoiono:

	0	1	2	3		0	1	2	3		0	1	2	3
0			B		⇒			3		⇒			X	
1	A		A			1	2		2		1	A		A
2			C			2			3		2		X	

6) Wall Punch

Ne muoiono più di una:

	0	1	2	3	4	5	6	7	8
0	A	A	A	A	A	A	A	A	A
1				B	B	B			
2				B	B	B			

⇒

	0	1	2	3	4	5	6	7	8
0	0	1	3	5	6	5	3	1	0
1				5	5	5			
2				3	3	3			

	0	1	2	3	4	5	6	7	8
0	A	A	X	X	X	X	X	A	A
⇒ 1				X	X	X			
2				X	B	X			

Input

Le informazioni che concernono gli 'input' sono riportate esclusivamente per informare il lettore sulle modalità di comunicazione tra il bot ed il motore che coordina un intero match.

Informazioni sui parametri

All'inizio di un match, ad ogni bot vengono trasmessi i parametri generali del gioco il quale inizia con l'inserimento di "turno 0" sulla propria riga.

I parametri verranno quindi passati su righe separate nel seguente formato

type value

Il *type* di *value* è determinato dal tipo di parametro.

Attualmente tutti i valori, ad esclusione di `player_seed`, sono interi con segno a 32 bit mentre `player_seed` è un intero con segno a 64 bit.

Se il bot dovesse ricevere un parametro sconosciuto, dovrebbe trattarlo come una stringa *opaca* e non cercare di analizzarlo in nessun modo.

Di seguito l'elenco dei parametri che verranno passati ai bot:

"loadtime"	#in millisecondi, tempo dato per l'avvio del bot dopo che è stato dato "ready"
"turntime"	#in millisecondi, tempo dato al bot ogni turno
"righe"	#numero di righe nella mappa
"cols"	#numero di colonne nella mappa
"turns"	#numero massimo di turni nel gioco
"viewradius2"	#raggio di visualizzazione al quadrato
"attackradius2"	#raggio di battaglia al quadrato
"spawnradius2"	#raggio di raccolta del cibo al quadrato
"player_seed"	#seme per il generatore di numeri casuali, utile per riprodurre i giochi

Una volta che tutti i parametri sono stati passati, si riceverà "*ready*" su di una riga separata; a quel punto si è liberi di eseguire le varie computazioni per tutto il tempo specificato dal *loadtime*.

Informazioni sul turno

Ogni turno successivo inizia con una delle seguenti righe:

```
turn turnNo  
end
```

“*end*” indica che il gioco è terminato; il vincitore del gioco riceverà, in seguito, le informazioni per lo stato finale del gioco in modo da poterli utilizzare per i test locali.

Se il gioco è finito, i bot riceveranno due righe che indicano il numero di giocatori e punteggi nel seguente formato:

```
players noPlayers  
score p1Score ... pnScore
```

Vengono dunque trasmesse le informazioni sulle *tile*, che può attualmente vedere il *bot*, nel seguente formato:

```
w row col          #water  
f row col          #food  
h row col owner    #ant hill  
a row col owner    #live ant  
d row col owner    #dead ant
```

La fine dell’input per un turno è indicata dalla ricezione di “*go*” sulla propria riga.

Le informazioni vengono sempre trasmesse come se il giocatore fosse il giocatore zero; il primo nemico che appare è sempre identificato come giocatore uno e così via. Questo aiuta a garantire che si conosca il numero dei giocatori che partecipano al match.

Le informazioni su un tile d’acqua vengono inviate solo durante il primo turno in cui è visibile da una delle tue formiche vive (per ridurre la quantità di dati trasferiti). Le informazioni sulle tile d’acqua vengono inviate solo durante il primo turno in cui è visibile da una delle proprie formiche in vita (per ridurre la quantità di dati trasferiti).

Verranno trasmesse informazioni sulle formiche vive (nemiche), cibo e *hill* ogni volta che sono in vista da una delle formiche del proprio bot.

Cibo e *hill* non si muovono e verranno trasmesse le loro informazioni ad ogni turno.

Se viene raccolto un cibo o se viene rasa al suolo una *hill* aldi fuori del raggio di visione delle formiche del bot, quest'ultimo non verrà informato.

Quando la formica del bot torna nel raggio d'azione, non riceverà alcuna informazione sul cibo mancante o sulle *hill* rase al suolo.

Le informazioni vengono fornite per le formiche morte durante la collisione o la risoluzione della battaglia del turno precedente se si trovano in una delle *tile* attualmente visibili dalle tue formiche vive.

Si forniscono sempre le informazioni sul proprio bot ossia sulle formiche morte e sulle proprie *hill* distrutte anche se è possibile visionare la *tile*.

Queste informazioni possono essere utilizzate dal partecipante, se lo desidera, altrimenti possono essere trattate come spazzatura.

Informazioni in input e Fog of War

All'inizio del match si forniscono al bot vari parametri tra cui `viewradius2`; quest'ultimo indica la distanza entro la quale ogni formica è in grado di vedere attorno ad essa. Al momento è impostato a 55, ottenendo un raggio di visione approssimativamente pari a 7,4 *tile*.

Come già anticipato, ogni formica è in grado di vedere soltanto l'area che la circonda entro un certo raggio, detto *view radius* ossia il raggio di visione della singola formica.

Di conseguenza i bot inizieranno ogni turno con una visione **parziale** della mappa ottenendo determinate informazioni da processare per determinare il miglior comportamento da assegnare alle formiche di fronte ai cambiamenti del mondo esterno.

La loro visione sarà infatti offuscata dalla cosiddetta **fog of war** che occulterà varie informazioni su ciò che avviene nel mondo.

Ad ogni turno vengono fornite in *input* al bot le seguenti informazioni per tutte le *tile* che sono visibili dalle proprie formiche:

- una lista delle *tile* che contengono **acqua**, che non erano state viste in precedenza;
- una lista delle *tile* contenenti le **formiche**, incluso l'id del proprietario;
- una lista delle *tile* contenenti il **cibo**;
- una lista delle *tile* che ospitano le **hill**, incluso l'id del proprietario;
- una lista delle formiche **morte** (durante l'ultima fase di attacco), incluso l'id del proprietario.

Si è deciso di azzerare, alla fine di ogni turno, tutte le informazioni elencate in precedenza, ad eccezione delle *tile* contenenti acqua, per poi continuare ad acquisirne altre all'inizio del turno successivo; ciò viene fatto perché, a differenza dell'acqua che è immobile, le formiche possono spostarsi o morire e non trovarsi dunque più dove si trovavano nel turno precedente, le *hill* possono essere distrutte da un turno all'altro ed il cibo può essere raccolto da una formica o scomparire poiché circondato da formiche di fazioni differenti.

Output e concetto di *ordine*

Nel corso del turno si assegnano dei task alle formiche sotto forma di *ordini* che dovranno essere, se possibile, immediatamente restituiti in output al server.

Un bot può assegnare al più un ordine per ogni formica durante ogni singolo turno.

Ogni *ordine* specifica:

- una formica, identificata dalle sue coordinate nella mappa;
- la direzione verso cui spostare la formica: Nord, Est, Sud oppure Ovest.

Una volta che l'ordine viene eseguito, le formiche si spostano di una *tile* nella data direzione.

Si osservi la mappa in figura.

	0	1	2
0	A		
1			
2			
3		F	

Si considerino la formica, **A**, che stanza nella *tile* in posizione (0,0), ed il cibo, **F**, in posizione (3,1).

Si supponga che nel corso del turno sia stato assegnato come obiettivo ad **A** il cibo **F** e che l'[algoritmo di ricerca](#) (di cui parleremo in seguito) abbia determinato che la *tile* su cui si debba spostare **A** per raggiungere **F** sia quella in posizione (3,0), verso NORD.

In questo caso l'*ordine* da restituire come *output* sarà: **o 0 0 N**.

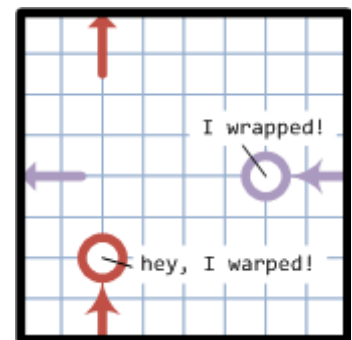
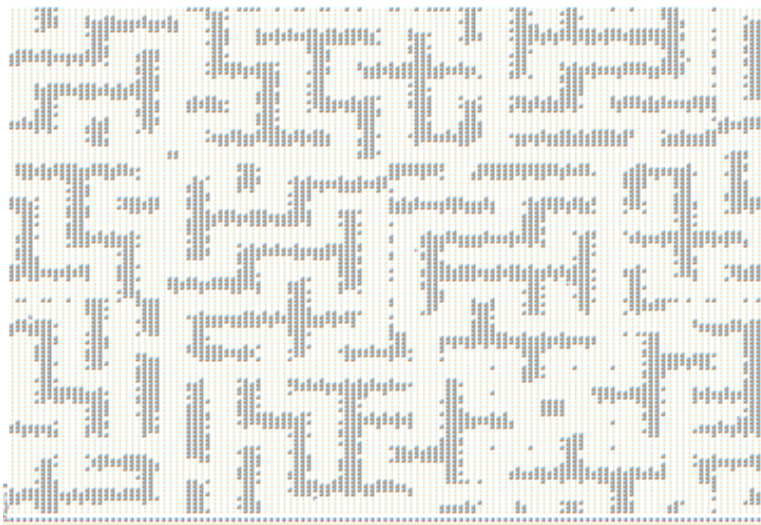
Se il bot non ha raggiunto le [condizioni di terminazione del gioco](#), la formica effettuerà detto spostamento nel turno corrente.

La mappa ed i possibili spostamenti

Il *turno 0* è riservato all'acquisizione delle [impostazioni di gioco](#) ed all'inizializzazione della mappa del gioco.

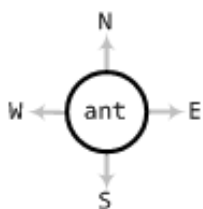
Lo starter-kit mette a disposizione diverse mappe su cui poter far allenare il proprio bot (vd. figura sottostante).

Tutte le mappe sono simmetriche, sebbene in alcuni casi i dettagli risultino essere ruotati o a specchio, in modo che ciò che circonda il punto di partenza di ogni giocatore sia lo stesso per tutti i giocatori. Ciò viene fatto al fine di dare le medesime opportunità ad ognuno dei concorrenti.




La **mappa** è una griglia rettangolare suddivisa in celle di forma quadrata, dette *tile*, che si avvolge ai bordi (esattamente come nel famoso gioco per cellulari *Snake*, in cui viene adottata una mappa di tipo *Torus*⁶).

Ciò significa che se una formica supera il confine NORD della mappa apparirà da quello SUD o se attraversa il confine EST apparirà ad OVEST.



Le formiche possono effettuare un solo **spostamento** per ogni turno e tale spostamento può avvenire verso una delle quattro direzioni cardinali: Nord, Est, Sud ed Ovest.

Si escludono dunque movimenti diagonali.

Solo una formica per volta può occupare una *tile* e se più formiche cercano di camminare sulla stessa *tile* nello stesso turno, moriranno. 

⁶ <https://en.wikipedia.org/wiki/Torus>

Se una formica resta ferma su di una *hill*, non sarà possibile far nascere ulteriori formiche.

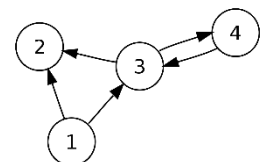
È dunque bene evitare di far stanziare le formiche dell'agente sull'ingresso del formicaio.

Inizialmente della mappa si conosce soltanto la dimensione, deducibile dal suo numero di righe e di colonne; di conseguenza non si conosceranno le posizioni delle *tile* contenenti acqua, cibo, formiche e *hill*.

Le mappe sono generate casualmente utilizzando un programma progettato per generare mappe “interessanti” che non consentono ai giocatori di codificare una particolare strategia. Ogni mappa rispetta i seguenti criteri:

- sono limitate da un minimo di 2 ad un massimo di 10 giocatori;
- sono (semi) simmetriche;
- i nidi sono distanti da 20 a 150 passi dai nidi dei nemici (quelli appartenenti alla stessa fazione possono essere più vicini);
- ogni nido ha un percorso attraversabile da blocchi di 3x3;
- non sono presenti isole;
- è possibile percorrere al massimo 200 passi in ogni direzione;
- per ogni giocatore ci sono da 900 a 5000 *tile*, con una superficie totale massima di 25000.

Si è deciso di memorizzare le *tile* della mappa, ossia le caselle su cui le formiche possono, o meno, spostarsi, utilizzando la struttura dati del **grafo**.



Ogni *tile* corrisponde ad un nodo, il quale ha come vicini i nodi corrispondenti alle *tile* posizionate nei quattro punti cardinali.

Il **fattore di ramificazione uscente** (*forward branching factor/forward*), come anche quello **entrante**, da e per un nodo è minore o uguale a 4.

Il “minore”, nella precedente affermazione, deriva dalla scelta di minimizzare il limite superiore dei fattori di ramificazione per semplificare le computazioni da parte degli [algoritmi di ricerca](#), riducendone la complessità; infatti, nei turni successivi al primo, non appena le formiche cominceranno ad ottenere informazioni sul mondo, si è deciso di rimuovere i nodi e, quindi, anche i relativi archi, delle *tile* contenenti acqua.

Il nostro concetto di *tile*

Una **tile** è un tassello della mappa che fornisce le 8 seguenti informazioni a seconda dello stato del gioco:

1. **Numero di riga e di colonna** che identificano le coordinate della posizione della *tile* nella mappa;
2. **Tipologia della *tile***. Ad ogni *tile* viene associata una delle seguenti tipologie:
 - a. UNEXPLORED, ovvero il valore che si associa ad una *tile* non ancora vista da nessuna delle formiche del bot;
 - b. LAND, ovvero porzione di mappa che può essere vuota, contenente del cibo o occupata da una formica; nell'ultimo caso, si estrapola e si conserva l'*id* corrispondente al giocatore a cui appartiene;
 - c. WATER, ovvero le *tile* della mappa che non possono essere calpestate dalle formiche. Una volta che una *tile* assume questo valore, dopo essere entrata per la prima volta nel raggio di visione di una formica, non lo cambierà per il resto della partita; ciò invece non accade per le altre tipologie in quanto suscettibili di cambiamento nel corso del gioco;
 - d. HILL, caratterizza una *tile* contenente un formicaio; tale valore è destinato a trasformarsi in *LAND* qualora il formicaio venisse distrutto. Come avviene anche per la tipologia *LAND*, si estrapola e conserva l'*id* corrispondente al giocatore a cui appartiene il formicaio.
3. **Il riferimento alle *tile* (calpestabili e) vicine** a quella corrente ossia quelle posizionate rispettivamente a NORD, EST, SUD ed OVEST di esse. Se ad una *tile* vicina è stato assegnato il valore *WATER*, il riferimento a questa viene cancellato. *Tile* di tipo *WATER* non hanno vicini. Questa scelta è stata fatta per semplificare le computazioni da parte degli [algoritmi di ricerca](#);
4. **L'indice di visibilità** ossia un numero intero che può assumere diversi valori:
 - a. è pari allo zero se la *tile* è attualmente visibile da una delle formiche dell'agente, a patto che non sia di tipo *WATER*;
 - b. è pari ad un intero positivo che incrementa il suo valore di una unità per ogni turno in cui non è visibile dalle formiche, a patto che non sia di tipo *WATER* o *HILL*;
 - c. è pari ad un intero negativo che decresce il suo valore di una unità per ogni turno in cui non è visibile dalle formiche, se e solo se è di tipo *WATER*;

d. è pari ad un intero positivo che corrisponde al numero massimo di turni possibili per la partita, se è di tipo *HILL* e non è attualmente visibile da nessuna delle formiche.

Tra questi, il valore maggiore risulterà prioritario in fase di esplorazione delle *tile* non visibili nel turno di riferimento, al fine di aumentare la percentuale di visione sulla mappa.

Inoltre, ciò permette di avere un maggiore controllo delle *hill* dell'agente che, se sotto attacco, saranno più facilmente raggiungibili dalle sue formiche in quanto ce ne sarà almeno una nelle sue prossimità. Questi concetti riguardano il modulo di esplorazione e quello di difesa di cui si tratterà nei paragrafi successivi.

5. **Lo stato di occupazione della *tile***, espresso da un valore booleano che sarà pari a *true* se la stessa è occupata da una formica ovvero a *false* nel caso contrario;
6. **L'identificativo del giocatore** a cui appartiene la *hill* o la formica che occupano la *tile*. Se la formica appartiene all'agente avrà sempre valore pari allo *zero*; nel caso contrario assumerà valore incrementale in base all'ordine di incontro di nuovi avversari (1, 2, 3, 4 ...);
7. **La disponibilità (calpestabilità) della *tile***, ovvero la possibilità di calpestare o meno quella determinata porzione di mappa a seconda del suo stato: una *tile* di tipo *WATER* non è mai calpestabile mentre una di tipo *HILL* può essere o meno calpestabile secondo criteri di opportunità stabiliti a priori.

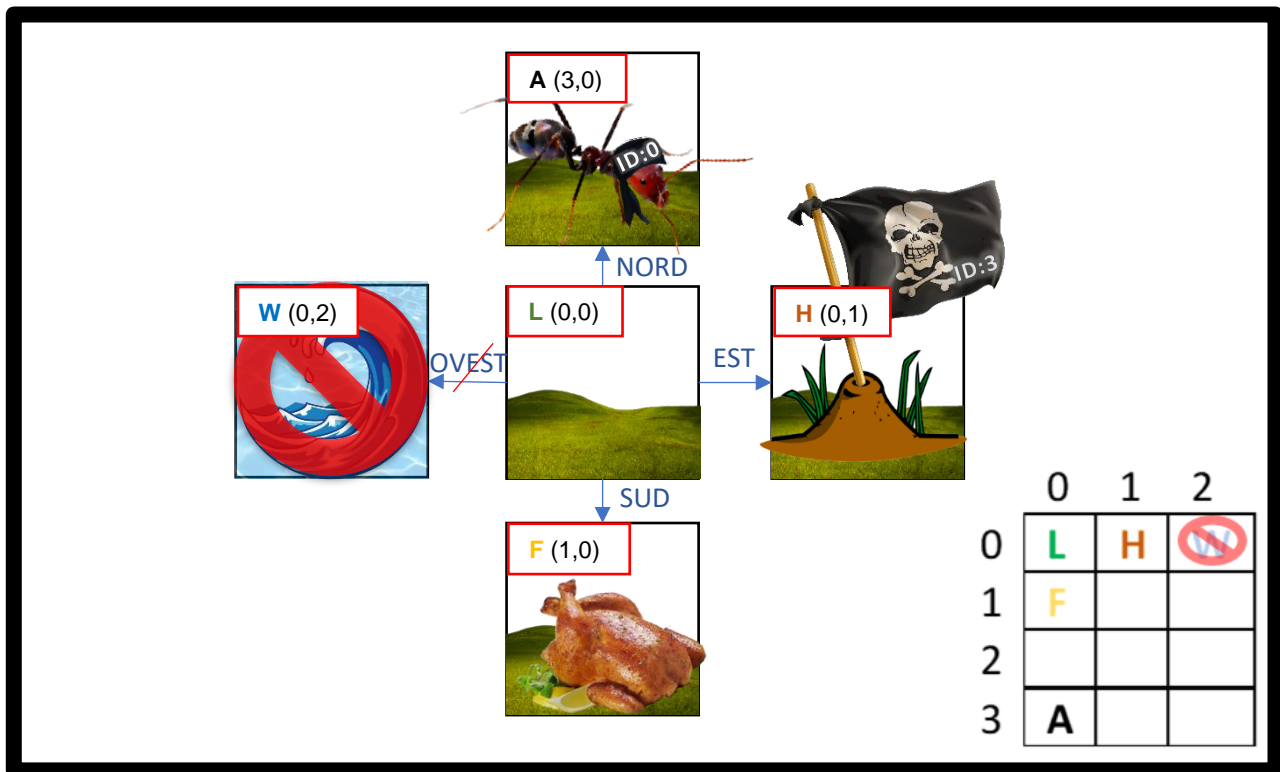
In linea generale, si è cercato di rendere le *hill* dell'agente preferibilmente *non* calpestabili in quanto questo comportamento ostruirebbe la generazione di nuove formiche; tuttavia, in particolari casi, come ad esempio nel caso in cui ci si trovi costretti ad indietreggiare per difendere la propria *hill* è stata data la possibilità alle formiche di calpestare momentaneamente quest'ultima.

Anche in questo caso vengono impiegati i valori *true* e *false*;

8. **La presenza di cibo** nella *tile*, sempre per mezzo di una variabile *booleana*.

ESEMPIO: RAPPRESENTAZIONE E CONTENUTO DI UNA TILE

Al fine di rendere più chiari i concetti sopra espressi si prenda come riferimento la figura sottostante:



Ipotizzando che **L** sia la *tile* corrente:

1. La sua riga e la sua colonna hanno valore '0';
2. È una *tile* di tipo *LAND*;
3. Ha come vicini le *tile*:
 - a. a NORD: **A** in posizione (3,0), in cui è situata una formica avente id pari a '0' che appartiene dunque all'agente;
 - b. ad EST: **H** in posizione (0,1), in cui è situata una *hill* nemica. Quest'ultima ha id pari a '3' poiché si tratta della tana appartenente al terzo avversario che l'agente ha avvistato nel corso della partita;
 - c. a SUD: **F** in posizione (1,0), in cui è contenuto del cibo;

Sebbene **L** sia altresì confinante ad OVEST con **W** nella mappa, il riferimento a quest'ultima *tile* sarà assente poiché la stessa contiene acqua;

4. L'indice di visibilità è pari a '0' poiché è visibile dalla formica **A**;
5. Lo stato di occupazione della *tile* è pari a *false* poiché non è occupata da alcuna formica;
6. Alla *tile* non è associato alcun identificativo del proprietario poiché non contiene né formiche né *hill*;
7. La *tile* risulta calpestabile. Supponendo che ad **A** sia stato assegnato come obiettivo il raggiungimento della *hill* nemica **H** al fine di distruggerla, questa potrà tranquillamente spostarsi su **L** per raggiungerla;
8. Non contiene del cibo.

3. Il nostro bot: MarksHive

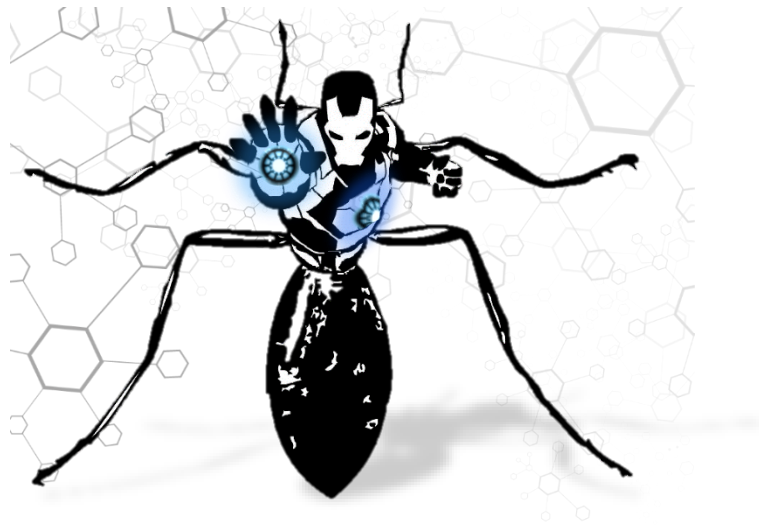
Per introdurre il nostro bot si faccia riferimento alle **dimensioni della complessità**, significative per la progettazione dell'agente:

1. **Modularità**: riguarda il grado di decomposizione di un sistema in **moduli** che interagiscono tra di loro e che possono essere studiati separatamente. Si è optato per una **modularità gerarchica** che ci ha permesso di decomporre la struttura organizzativa dell'agente in [moduli](#) e sotto-moduli ottenendo dunque più livelli di astrazione. I vari moduli sono stati individuati in base ai *task* che l'agente deve svolgere in vista dell'obiettivo finale ossia la "vittoria". Ogni modulo è trattato singolarmente nelle sezioni successive;
2. **Orizzonte**: misura quanto lontano nel tempo sia prevista la pianificazione del lavoro dell'agente. Il nostro bot è un **pianificatore ad orizzonte finito** poiché va alla ricerca di un numero prefissato e finito di fasi. Ogni formica dell'agente prende, infatti, la decisione che gli porterà più vantaggi nel turno successivo a quello in cui si effettuano le computazioni: nel [modulo del combattimento](#), ad esempio, l'agente valuta se sia più conveniente indietreggiare, avanzare o far restare immobili le sue formiche in base ad una serie di vantaggi e svantaggi che conseguono ai vari spostamenti (numero di formiche perse, *hill* distrutte, nemici sconfitti, etc.). Essendo un pianificatore ad una sola fase, l'agente è detto *greedy* (miope);
3. **Limiti sulle Risorse Computazionali**: è la dimensione che determina se l'agente tenga conto, o meno, dei suoi *limiti computazionali* ossia limiti di tempo, memoria, precisione, etc. Il nostro bot è a **razionalità limitata** in quanto tiene conto dei suoi limiti computazionali. L'agente non ha, infatti, tempo illimitato per eseguire i suoi calcoli ed impartire i propri ordini descrivendo quali formiche muovere e in quali direzioni spostarle. I giocatori che non rimangono entro il tempo limite vengono squalificati e non possono fare più mosse; le loro formiche continuano a partecipare al gioco, ma non eseguono più mosse. Infatti, il [modulo del tempo](#) assegna a tutti i [moduli](#), in cui si è partizionata la struttura organizzativa dell'agente, un frangente di tempo entro cui poter eseguire i propri calcoli.
4. **Apprendimento**: è la dimensione che determina se la conoscenza sul mondo è data o se necessita di essere appresa. Il bot dispone di una **base di conoscenza**, fornitagli dal server ad ogni turno, che identifica il numero di formiche dell'agente, il numero di turni a disposizione, *hill* ancora in piedi ma, essendo il mondo *non pienamente osservabile*, alcuni dati necessitano di essere appresi nel corso dei vari turni (acqua,

formiche nemiche, hill nemiche etc) tramite computazioni offline grazie alle quali si ottiene una **conoscenza di fondo**;

5. **Incerteza sull'osservazione**: riguarda la possibilità di determinare lo stato del mondo attraverso le osservazioni. Nel nostro caso, lo stato del mondo risulta essere **parzialmente osservabile**: si è già introdotta la *fog of war* che non permette all'agente di avere una visione completa di tutta la mappa, del numero effettivo di nemici sul campo come anche della posizione di tutte le *tile* di acqua, etc.
6. **Incerteza sull'effetto**: riguarda la capacità dell'agente di conoscere gli effetti e le conseguenze delle proprie azioni ovvero se sia in grado di predire *precisamente* lo stato risultante dall'esecuzione dell'azione sullo stato. Si può dire che le dinamiche degli effetti della dimensione dell'incerteza sono **aleatorie** in quanto vi sono delle **probabilità** sulla distribuzione degli stati risultanti: dispone di una *distribuzione di probabilità* sugli effetti possibili ma non si ha la certezza che ad un'azione ne consegua, necessariamente, una specifica reazione. Ad esempio, nel [modulo del combattimento](#) si possono soltanto supporre le conseguenze di un avanzamento, di una ritirata o del restare fermi nella stessa posizione: ciò dipende dalla reazione dei nemici.
7. **Preferenza**: riguarda la scelta dell'azione che porterà a risultati più desiderabili. Il nostro agente è in grado di giocare in modo dignitosamente intelligente al gioco Ants. Essendo costruito utilizzando una struttura modulare, con singoli moduli che incorporano diversi elementi della logica di gioco. Mantiene un sistema di stati interno per modellare lo stato del gioco durante il match e impiega una varietà di algoritmi per un processo decisionale intelligente e rapido. L'algoritmo chiave è *adaptiveSearch()* descritto successivamente.
8. **Numero di Agenti**: essendo il gioco multiplayer, il nostro bot si ritrova a ragionare in uno stesso ambiente con più avversari. Per questo adotta un **ragionamento multi-agente** che prende in considerazione il ragionamento degli altri agenti in vista di obiettivi comuni il cui raggiungimento dipende dal comportamento di tutti i partecipanti;
9. **Interazioni**: considera se l'agente effettua un *ragionamento offline*, ossia se determina cosa fare prima di interagire con l'ambiente, oppure se effettua un *ragionamento online*, ovvero se determina l'azione da effettuare mentre interagisce con l'ambiente. Come anticipato, il bot ha a disposizione un frangente di tempo, *offline*, entro cui effettuare le varie computazioni. L'agente *non* prende, dunque, decisioni, in fase di interazione con l'ambiente.

Fun Fact: la natura del nome del nostro bot⁷



Abbiamo deciso di assegnare al nostro bot il nome **MarksHive** in onore delle varie armature del supereroe della Marvel Iron Man.

Iron Man non utilizza un unico esoscheletro, anzi: quello che può essere definito come l'**armadio tecnologico** di Tony Stark si *aggiorna* di continuo con l'aggiunta di nuove e sempre più sofisticate armature.

Tutti i modelli hanno una cosa in comune: il loro nome è composto **dalla parola “Mark” seguita da un numero**.

Il termine **“Mark” non è un vero e proprio nome**, ma è una parola che può essere tradotta come “modello”.

In pratica l'armatura Mark_01 non è altro che il **“modello 1”**.

Nel corso della progettazione, proprio come Tony Stark, abbiamo realizzato più esoscheletri del nostro bot, il cui codice seriale incrementava in vista delle varie migliorie fino a giungere al bot finale: **MarksHive**.

⁷ <https://www.lettera43.it/howto/perche-le-armature-di-iron-man-si-chiamano-mark/>

Moduli e Task

Per prima cosa, abbiamo individuato i *task* fondamentali del gioco, dividendoli in quelli che caratterizzano i *moduli* principali del programma:

- A. **Modulo di combattimento**: nel momento in cui una formica nemica entra nel raggio di visione dell'agente, il modulo di combattimento si occupa di simulare le possibili azioni dell'agente in modo da assegnargli la *mossa migliore* a seconda del contesto (ad esempio tenendo conto della quantità di cibo raccolta a seguito della mossa, del numero di formiche o nidi nemici annientati, del numero di formiche perse nel combattimento, etc.);
- B. **Modulo di difesa**: si sostanzia nella scelta del posizionamento delle formiche dell'agente a difesa dei propri nidi all'arrivo di una minaccia nemica;
- C. **Modulo di riproduzione**: gestisce le mosse volte al raccoglimento del cibo da parte dell'agente per la conseguente "riproduzione" di nuove formiche;
- D. **Modulo di attacco**: organizza i movimenti dell'agente al fine di distruggere i nidi nemici;
- E. **Modulo di esplorazione**: assegna all'agente ordini volti alla visualizzazione di una porzione sempre più ampia della mappa in cui agisce;
- F. **Modulo di visione**: permette di ottenere le informazioni riguardanti le formiche ed i nidi dell'agente e dei nemici ed in generale sui cambiamenti del mondo circostante;
- G. **Modulo del tempo**: gestisce ed assegna il tempo necessario alla computazione ed all'esecuzione dei moduli precedenti.

Ogni modulo verrà trattato singolarmente nelle sezioni successive. Tutti i moduli possono essere considerati come atomi. Questi sono responsabili di singoli aspetti del gioco ed hanno minime dipendenze l'uno dall'altro. Questa struttura consente di costruire e mantenere separatamente diversi elementi dell'agente. I moduli possono essere attivati o disattivati indipendentemente al momento della configurazione o al momento del gioco. Questo permette una configurazione flessibile dell'agente a seconda delle funzioni richieste, nonché una manutenzione e un debug più semplice dei singoli moduli. A

Tutti i moduli interagiscono utilizzando un semplice sistema di eventi. Al momento dell'inizializzazione, tutti i moduli attivi vengono istanziati e registrati per monitorarne i tempi di esecuzione.

Metodi di ricerca

Prima di effettuare l'azione nel mondo, l'agente deve andare alla ricerca della sequenza di azioni che gli permetteranno di raggiungere un determinato scopo.

Questo problema lo si può ricondurre a quello *matematico* della ricerca di un **percorso** in un *grafo orientato*.

I moduli che vanno dalla *A.* alla *E.* necessitano dell'utilizzo di **algoritmi di ricerca** in modo da determinare il percorso ottimale partendo da uno stato iniziale detto *sorgente* sino ad uno stato obiettivo detto *target*; quest'ultimo può o meno essere conosciuto a priori a seconda del modulo da cui è partita la ricerca.

Come già anticipato nel paragrafo relativo alle specifiche della [mappa](#), il fattore di ramificazione (entrante ed uscente) è sempre minore o uguale a 4.

Abbiamo scelto di implementare due versioni *modificate* delle seguenti *strategie di ricerca*:

- **Breadth-first search, [BFS](#)** (modificata): la *ricerca in ampiezza* è una strategia di ricerca “non informata” che non tiene, dunque, conto della posizione del target. Nello specifico, questo algoritmo simula tutti i possibili spostamenti finché non trova il percorso ottimale per il raggiungimento di un target che risulterà essere quello più vicino alla sorgente. Entra in gioco quando si ignora *dove* ci si sta dirigendo finché non si trova un obiettivo dichiarando il successo;
- **[Ricerca A*](#)** (modificata): è utilizzata per le ricerche a lunghe distanze in quanto più efficiente rispetto alla BFS. Si utilizza, inoltre, quando si conosce e si assegna a priori l'obiettivo della ricerca. Assegnare preventivamente l'obiettivo alla sorgente permette di sfruttare l'*informazione euristica* che si calcola (tra l'altro) per mezzo della distanza tra la sorgente ed il target stessi; tale informazione è fondamentale per il corretto funzionamento di A^* .

Ai due metodi di ricerca sopraelencati se ne somma un terzo ossia la [ricerca statica](#) che non restituisce un insieme di *ordini* bensì un insieme di target.

I suddetti algoritmi, ad eccezione della ricerca statica, sono stati da noi modificati al fine di:

- permettere la ricerca a partire da una o più sorgenti indifferentemente che si tratti della modalità *classica* o di quella *backward*;
- consentire di effettuare una ricerca *backward* scegliendo tra due modalità:
 - **one target per source**: permette di effettuare una ricerca *inversa* a partire dai target verso sorgenti necessariamente diverse l'una dall'altra. Di conseguenza la ricerca restituirà un ordine per ogni sorgente verso *target* differenti;
 - **reverse**: permette di effettuare una ricerca *backward* a partire dai target verso sorgenti che possono essere comuni. Di conseguenza la ricerca restituirà un *ordine* per ogni sorgente verso *target* comuni (o meno).

L'algoritmo di ricerca adattiva viene utilizzato per trovare gli oggetti richiesti sulla mappa di gioco, dato un insieme di *tile* origine e un elenco di obiettivi.

L'algoritmo è utilizzato in molti elementi della logica di gioco dell'agente.

Esempi di applicazioni sono:

- assegnare la raccolta di cibo a tutte le formiche;
- individuare gruppi di formiche;
- trovare le formiche disponibili per rinforzi in battaglie;
- trovare le *tile* non visibili più vicine per l'esplorazione.

La ricerca viene inizializzata creando un'istanza della classe *Search*.

Essa può essere configurata utilizzando i setter d'istanza.

Infine può essere eseguita una o più volte chiamando la funzione *adaptiveSearch()*.

Poiché è stato utilizzato in molte occasioni in diversi moduli dell'agente, l'algoritmo di ricerca generale richiede una grande varietà di dati di input e di output.

Per tenere conto di ciò e per ottimizzare i tempi, l'algoritmo si adatta a seconda delle opzioni settate durante la fase di inizializzazione dei dati di input.

Si ottiene dunque un'esecuzione diversa in base alle varie configurazioni:

- Se la ricerca è limitata a un certo raggio attorno a ciascuna sorgente, verrà eseguita una *ricerca statica* (ciò accade per esempio nel modulo di visione o nel riconoscimento della situazione di combattimento);
- se è disponibile un'euristica ammissibile per tutti i potenziali *tile target*, verrà eseguita una *ricerca extended A star* (ciò accade in determinate situazioni del modulo di esplorazione, quando le formiche devono essere inviate su lunghe distanze);
- in caso contrario, verrà eseguita una *ricerca extended BFS*.

Si è cercato di massimizzare determinismo intrinseco del linguaggio di programmazione da noi utilizzato (JAVA) il quale definendo tramite la classe enumerativa le quattro direction pone un ordinamento totale sugli stessi. In questo modo vengono selezionati (ad esempio nella lista dei neighbour) sempre nello stesso ordine, mostrandoun comportamento "monotono", degli spostamenti ad "L", prediligendo le prime direzioni selezionate escludendo dunque la possibilità di percorsi più complessi e preferibili. Si è fatto in modo di restituire pseudo casualmente i neighbour/le direzioni (i vicini) di una tile (nodo) in modo da ovviare a tutti i precedenti problemi e quindi come già affermato diminuire il non determinismo nelle scelte del software.

ATTENZIONE: È opportuno precisare che nel caso in cui la ricerca sia del tipo [one-target-per-source](#) oppure [reverse](#) la sorgente corrisponderà all'obiettivo mentre il *target* sarà la sorgente.

La motivazione è che, in detti casi, la ricerca è *backward* ossia parte dal target andando alla ricerca della sorgente.

Tuttavia, nel momento in cui si andrà a creare l'[ordine](#) questo dovrà necessariamente partire da una sorgente (formica dell'agente) in direzione dell'obiettivo (cibo, *hills*, formiche nemiche, etc); l'ordine andrà dunque adeguato ai requisiti.

- Metodi di ricerca: BFS modificata

Basata su una classica ricerca in ampiezza, la nostra variante consente l'uso di più sorgenti. Tutte le sorgenti vengono aggiunte inizialmente alla frontiera e vengono trattati come nodi non espansi. La ricerca restituisce tutti i tile di destinazione trovati, le rispettive origini e due direzioni (direzione di inizio del percorso dall'origine e direzione di inizio percorso inverso dalla destinazione).

L'idea di base è quella di avere una frontiera che contenga percorsi con nodi di partenza gli uni diversi dagli altri.

I nodi di partenza sono detti *sorgenti* mentre i nodi-obiettivo sono detti *target*.

Il funzionamento dell'algoritmo è analogo alla BFS classica con la differenza che i vari percorsi nella frontiera, *frontier*, potrebbero avere come radice sorgenti differenti.

A partire da una sorgente, se un percorso ha come nodo terminale un *target* significa che uno tra i vari obiettivi è stato raggiunto; in tal caso si memorizza l'*ordine* contenente il primo spostamento verso quel target a partire da quella sorgente: l'ordine è caratterizzato dalla sorgente e dalla direzione verso cui ci si deve spostare per raggiungere il target.

Al termine della ricerca, si restituisce l'insieme di tutti gli ordini.

L'algoritmo da noi implementato si discosta ulteriormente dalla BFS classica per via delle 3 modalità anticipate nella sezione introduttiva ai [metodi di ricerca](#) (*classica*, *one target per source* e *reverse*).

È una scelta quella di controllare se un nodo è goal prima di aggiungerlo alla frontiera in quanto in BFS il percorso restituito è quello più breve. Nel nostro caso gli archi hanno tutti costo pari ad 1 quindi il percorso più breve coincide con il percorso con costo minore ottimale. In questo modo non si cercano, e quindi non vengono computati, altri percorsi (eventualmente di stessa lunghezza) risparmiando spazio e tempo. Altro motivo è che non troppo costoso effettuare il controllo sul goal e quindi non risulta necessario posporre quest'azione quando l'elemento viene selezionato e rimosso dalla frontiera.

Nell'esempio che segue si illustra il funzionamento della BFS in modalità *classica* con cenni alle altre due modalità; successivamente si analizzerà nel dettaglio il funzionamento dell'algoritmo contemplando le tre casistiche sopracitate.

ESEMPIO: FRONTIERA BFS MODIFICATA E ORDINI

Frontiera al primo step (modalità *classica*)

A1(0,0)	A2(2,1)	A3(1,3)
---------	---------	---------

Es. di frontiera al secondo step (modalità *classica*)

A2(2,1)	A3(1,3)	A1(1,0)	A1(0,4)	A1(3,0)
---------	---------	---------	---------	---------

Es. di frontiera al terzo step (modalità *classica*)

A3(1,3)	A1(1,0)	A1(0,4)	A1(3,0)	A2(2,1)	A3(1,3)	A1(1,0)	A1(0,4)	A1(3,0)	A2(1,1)	A2(2,2)
---------	---------	---------	---------	---------	---------	---------	---------	---------	---------	---------

etc.

	0	1	2	3	4
0	A1		F2		
1				A3	
2		A2			
3				F1	

Si consideri la mappa in figura ed un ipotetico percorso con nodo di partenza in **A1**(0, 0):

{ [(0, 0), WEST], [(0, 4), NORTH], [(3, 4) WEST] }

Poiché ad ovest della *tile* in posizione (3, 4) vi è un nodo target, ossia **F1**(3, 3), l'ordine da memorizzare è:

o 0 0 W

Si consideri il percorso con nodo di partenza in **A3**(1, 3):

{ [(1, 3), WEST], [(1, 2), NORTH] }

Poiché a nord della *tile* in posizione (1, 2) vi è un nodo target, ossia **F2**(0, 2), l'ordine da memorizzare è:

o 1 3 W

Si consideri ora il percorso con nodo di partenza in **A2**(2, 1):

{ [(2, 1), EAST], [(2, 2), NORTH], [(1, 2), NORTH] }

A nord della *tile* in posizione (1, 2) vi è un nodo target, ossia **F2**(0, 2); nel caso in cui la modalità di ricerca fosse *classica*, l'ordine da memorizzare sarebbe:

o 2 1 E

Nel caso in cui la modalità fosse *one target per source* la ricerca sarebbe terminata prima di trovare quest'ultimo percorso poiché tutti i *target* sarebbero già stati raggiunti da altre sorgenti (*one target per source* prevede che ogni target venga raggiunto da una sola sorgente).

L'insieme di ordini da restituire (nel caso di modalità *classica*) potrebbe essere la seguente:

[(**o 0 0 W**), (**o 1 3 W**), (**o 2 1 E**)]

Se la ricerca fosse stata avviata in modalità *reverse* o *one target per source* avrebbe avuto origine dall'insieme di target; le sorgenti sarebbero state dunque le seguenti:

$$sorgenti' \cong target := \{ \mathbf{F2}(0, 2), \mathbf{F1}(3, 3) \}$$

Frontiera al primo step

F2(0,2)	F1(3,3)
---------	---------

Esempio di frontiera alla seconda iterata

F1(3,3)	F2(1,2)
---------	---------

Esempio di frontiera alla terza iterata

F2(1,2)	F1(2,3)	F1(3,4)
---------	---------	---------

etc.

I target sarebbero invece stati quelli che in realtà corrispondono alle sorgenti:

$$target' \cong sorgenti := \{ \mathbf{A1}(0, 0), \mathbf{A2}(2, 1), \mathbf{A3}(1, 3) \}$$

Essendo, le due sopracitate, delle ricerche *backward*, i percorsi hanno radice nei *target* ed hanno come obiettivo le *sorgenti*.

Un esempio di percorso identificato da queste ricerche, con nodo di partenza in $\mathbf{F3}(3, 3)$, potrebbe essere:

$$\{ [(3, 3), \text{NORTH}], [(2, 3), \text{NORTH}] \}$$

Poiché a nord della *tile* in posizione (2, 3) vi è un nodo target, ossia $\mathbf{A3}(1, 3)$, l'ordine da memorizzare è:

o 1 3 S

Si noti che, nonostante la ricerca abbia avuto origine in $\mathbf{F3}(3, 3)$, essendo quest'ultimo in realtà un target, l'ordine da memorizzare non potrà avere origine in detta posizione; per definizione di *ordine*, infatti, un ordine ha origine in una *tile* contenente una formica a cui viene assegnata una direzione verso cui spostarsi.

Le sorgenti di un ordine *devono* essere necessariamente le formiche dell'agente (es. il cibo non si può spostare verso una formica).

In questo caso l'ordine avrà dunque origine nella *tile* in posizione (1, 3).

Sarà dunque necessario invertire l'ultima direzione trovata, in questo caso "NORTH" si tramuta in "SOUTH", ed assegnare come sorgente quella che per $\mathbf{F3}$ corrispondeva ad un *target* ossia (1, 3).

Un esempio di insieme di ordini che restituirebbe la ricerca *one target per source* potrebbe essere:

$$[(\mathbf{o 0 0 W}), (\mathbf{o 1 3 S})]$$

Un esempio di insieme di ordini che restituirebbe la ricerca *reverse* potrebbe essere:

$$[(\mathbf{o 0 0 W}), (\mathbf{o 1 3 S}), (\mathbf{o 2 1 E})]$$

Si considerino le seguenti variabili:

- *frontier*: coda, avente struttura *FIFO* (*first-in, first-out*), che conterrà le varie *tile* che si andranno a visitare nel corso della valutazione dei percorsi. Una volta che una *tile* viene rimossa dalla frontiera non verrà più reinserita in accordo al principio di *path pruning* (potatura dei cicli);
- *pathSources*: struttura le cui chiavi corrispondono alle *tile*, ossia i nodi della frontiera, ed i cui valori sono le *sorgenti* da cui hanno avuto origine le chiavi. Utile al fine di tenere traccia della sorgente da cui ha avuto origine una *tile*;
- *visited*: struttura la cui chiave è una *tile* ed il cui valore è l'insieme di *tile sorgenti* da cui ha avuto origine;
- *completedSources*: controlla che una sorgente non abbia già restituito un percorso; difatti, controlla se partendo da una determinata sorgente è stato trovato un percorso che porta ad un *target*;
- *directionFromSource*: struttura contenente un insieme di chiavi-valore; la chiave corrisponde alla *tile* mentre il valore è la direzione che ha permesso di arrivare sulla *tile* partendo da quella in cui si trovava precedentemente;
directionFromSource è utile per la ricerca *classica* in modo da recuperare tutte le sorgenti e tutte le direzioni verso cui far spostare le formiche;
- *directionFromTarget*: struttura contenente un insieme di chiavi-valore: la chiave corrisponde alla *tile* mentre il valore è la direzione che ha permesso di arrivare sulla *tile* a partire dalla sua vicina.
directionFromTarget è utile per la ricerca *reverse* ed anche per quella *one_target_per_source* in modo da recuperare tutte le sorgenti (identificate dalle *tile* che sono *target*) e tutte le direzioni verso cui far spostare le formiche. Delle direzioni bisognerà considerare il loro opposto (es. NORTH -> SOUTH, EAST -> WEST, etc.).

Pseudocodice:

```

sources := set di tile                                //se la ricerca è classica, è l'insieme delle sorgenti altrimenti è
                                                         l'insieme degli obiettivi;
targets := set di tile                                //se la ricerca è classica, è l'insieme degli obiettivi altrimenti è
                                                         l'insieme delle sorgenti
pathSources := map vuoto                               //ad ogni tile è associata quella da cui ha avuto origine
directionFromSource := map vuoto                       //ad ogni tile è associata una direzione per raggiungerla
directionFromTarget := map vuoto                       //ad ogni tile è associata una direzione per raggiungerla
function BFS()
    frontier := coda vuota                             //lista di tile che compongono la frontiera della BFS
    visited := map vuoto                               //ad ogni tile è associato l'insieme delle sue sorgenti
    completedSources := set vuoto                       //insieme di sources che hanno già raggiunto un target
    orders := set vuoto                                //insieme che conterrà gli ordini
    for all sources
        source = sources[next]
        add source to frontier
        pathSources[source] = source
        visited[source] = add source to a new set
    end for
    while frontier != {}
        curTile = select AND remove frontier[first]
        curTileSource = pathSources[curTile]
        if completedSources contains curTileSource
            continue while
        end if
        while curTile has neighbours                    //neighbours = vicini
            select neighbourTile
            select neighbourDirection                  //direzione da curTile verso neighbourTile
            if isSuitable(neighbourTile) AND           //è calpestabile
                ((BFS is reverse or one_target_per_source
                AND pathSources does NOT contain neighbourTile)
                OR (BFS is classic AND visited does NOT contain neighbourTile))
                pathSources[neighbourTile] = curTileSource
                if(BFS is classic)
                    if directionFromSource[neighbourTile] is set
                        directionFromSource[neighbourTile] =
                            directionFromSource[curTile]
                    else directionFromSource[neighbourTile] = neighbourDirection
                    end if
                else directionFromSource[curTile] = neighbourDirection
                end if
                directionFromTarget[neighbourTile] = get opponent of neighbourDirection
                if targets contains neighbourTile        //se è target
                    if BFS is one_target_per_source
                        add new Order(neighbourTile,
                                    directionFromTarget[neighbourTile]) to orders
                        remove neighbourTile from targets
                    else if BFS is reverse
                        add new Order(neighbourTile,
                                    directionFromTarget[neighbourTile]) to orders
                    else add new Order(curTileSource,
                                    directionFromSource[neighbourTile]) to orders
                    end if
                if BFS is one_target_per_source OR classic
                    add curTileSource to completedSources
                end if
            end while
        end while
    end while
end function

```

```

                                end if
                                add neighbourTile to results
                            else                                     //se non è target
                                if (visited does NOT contain neighbourTile OR visited
                                    contains neighbourTile BUT hasn't curTileSource as a value)
                                    add curTileSource to visited[neighbourTile]
                                end if
                                add neighbourTile to frontier
                            end while
                        end while
                    end while
                return orders

```

Come per la BFS classica, anche in questo caso si può notare come ogni percorso nella frontiera abbia lo stesso numero di archi (al più uno di differenza).

Tutte le *tile sorgente* vengono inserite in *frontier* ed allo stesso tempo si inizializza *pathSources* inserendo come coppia chiave-valore ognuna delle *tile* e loro stessa per simboleggiare che le sorgenti delle *tile* sono esse stesse.

Finché *frontier* non è vuota, si seleziona e rimuove il suo primo elemento, *curTile*.

Si salva in *curTileSource* la sorgente di *curTile* contenuta in *pathSources* (NB. alla prima iterata *curTile* = *curTileSource*).

Si controlla se *curTileSource* non sia contenuta in *completedSources*.

Se *completedSources* contiene *curTileSource* si deve selezionare una nuova *tile* da *frontier* in quanto bisogna impedire l'espansione di percorsi da detta sorgente verso un target dato che ne esiste già uno (nel caso contrario ci si aspetterebbe che una formica faccia un passo in due direzioni differenti, verso target differenti nello stesso turno – il che è assurdo!).

Nel caso contrario, si seleziona il primo vicino di *curTile* ossia *neighbourTile*.

Se *neighbourTile* è [calpestable](#) e non appartiene a *visited* (ossia se non è già stata visitata nei percorsi espansi da altre *tile* - o da *curTile* - che hanno origine da *curTileSource* - **path pruning**) allora si aggiorna *pathSources* inserendo *neighbourTile*, che indica la *tile* corrente, come chiave e *curTileSource* che simboleggia la sorgente da cui ha avuto origine, come valore.

In seguito:

- Se la ricerca è [reverse](#) oppure [one-target-per-source](#) allora si inserisce in *directionFromSource* la chiave *neighbourTile* e come valore la direzione memorizzata in *directionFromSource* che ha come chiave *curTile*, se esiste, altrimenti la direzione che ha permesso di giungere a *neighbourTile*, ossia *neighbourDirection*;
- Altrimenti si memorizza in *directionFromSource* la chiave *curTile* e come direzione *neighbourDirection*.

In ambo i casi si memorizza in *directionFromTarget* la chiave *neighbourTile* il cui valore è la direzione opposta a *neighbourDirection*.

Se *neighbourTile* è un target allora:

- Se la ricerca è [one-target-per-source](#) si crea un ordine che parte da *neighbourTile* (che corrisponde alla sorgente) in direzione di *curTileSource* (ossia il target). Poiché i target possono essere assegnati ad un'unica sorgente, si rimuove *neighbourTile* da *targets* (che in questo caso è l'insieme delle sorgenti);
- Se la ricerca è [reverse](#) allora viene creato un ordine che parte da *neighbourTile* (che corrisponde alla sorgente) in direzione di *curTileSource* (ossia il target);
- Altrimenti viene creato un ordine che parte da *curTileSource* in direzione di *neighbourTile*.

Gli ordini risulteranno validi e potranno in seguito essere restituiti al server se la *tile* su cui si vuole far spostare la formica non è già stata precedentemente ordinata ad un'altra.

Se *neighbourTile* non è target, si aggiunge *neighbourTile* alla frontiera e si aggiorna la lista di *tile* visitate.

Se *curTile* ha altri vicini, si seleziona il *neighbourTile* successivo; altrimenti se la frontiera contiene altre *tile* si ripetono nuovamente gli step precedenti.

- Metodi di ricerca: A* modificata

Basato sulla classica ricerca A*, la nostra variante contiene l'uso di più sorgenti (è presente un'ulteriore variante ad una sola sorgente).

Per mantenere l'ammissibilità dell'euristica, la funzione euristica è impostata come distanza di Manhattan modificata, che tiene conto dei bordi avvolti, dall'obiettivo sino alla formica più vicina.

Analogamente ad *extended BFS*, tutte le sorgenti vengono inizialmente aggiunte alla frontiera e trattati come nodi non espansi.

Si tratta del classico algoritmo per la ricerca in un grafo orientato con informazione Euristica.

Si dispone di una coda prioritaria *frontier* in cui vengono inserite le *tile* ossia i nodi terminali del grafo.

Ogni nodo visitato è inserito all'interno di un insieme detto *closed Set* che permette di effettuare potatura di cicli: non si può, infatti, visitare nuovamente un nodo che è già stato esplorato nel corso della ricerca; se il nodo è già stato visitato, infatti, significa che esisterà un percorso nella frontiera che ha già valutato quella *tile* e che possiede un'euristica avente costo minore rispetto al percorso appena trovato.

Si inserisce nella frontiera la tile-sorgente sottoforma di *Nodo*.

Ogni *Nodo* è caratterizzato da:

- La *tile* corrente;
- Il costo del percorso;
- Il valore dell'euristica;
- Il target che gli è stato assegnato.

Si avvia a questo punto la ricerca che terminerà non appena la frontiera risulterà vuota:

1. Si seleziona e rimuove il primo nodo, *cur*, dalla frontiera: poiché *frontier* è una coda prioritaria la cui priorità è dettata dall'euristica, il primo nodo sarà sempre quello con euristica avente costo minore;
2. Si aggiunge *cur* a *closed Set*;
3. Si verifica se *cur* sia un target o meno: nel primo caso, si crea un nuovo ordine e lo si restituisce in output; altrimenti si prosegue con le fasi successive;
4. Si selezionano i vicini, uno per volta, di *cur*.

- a. Se la *tile* vicina, *neighbour*, è calpestabile e non è stato ancora visitato, si crea il *Nodo* corrispondente inserendo tutte le informazioni necessarie per la sua identificazione (*tile* corrente, costo del percorso, etc.). Si salvano inoltre la direzione di precorrimiento da e verso la *tile* in due set, *directionFromSource* e *directionFromTarget*, che permetteranno, in seguito, la ricostruzione del percorso (*backward* oppure a partire dalla sorgente). Ciò è utile per la creazione dell'ordine da restituire al server.
- b. Altrimenti, si ricomincia dal passo 1.

- Metodo di ricerca statica e gli Offset

Questo metodo di ricerca, a differenza degli algoritmi visti in precedenza, non restituisce [ordini](#).

L'output sarà infatti un insieme di *target* contenuti entro un certo raggio a partire da tutte le sorgenti in input.

Ciò viene fatto per mezzo degli *offset*.

Un *offset* è una zona che circonda un punto di partenza, ossia la sorgente.

Il raggio di questa “circonferenza” è fornito in input in modo da determinare le *delta righe* e le *delta colonne* delle *tile* che comporranno la zona desiderata.

Una volta definito un *offset* per un certo raggio questo verrà memorizzato in modo tale che, se si dovesse necessitare successivamente di un *offset* avente la stessa dimensione, si eviterà di effettuare nuovamente le varie computazioni.

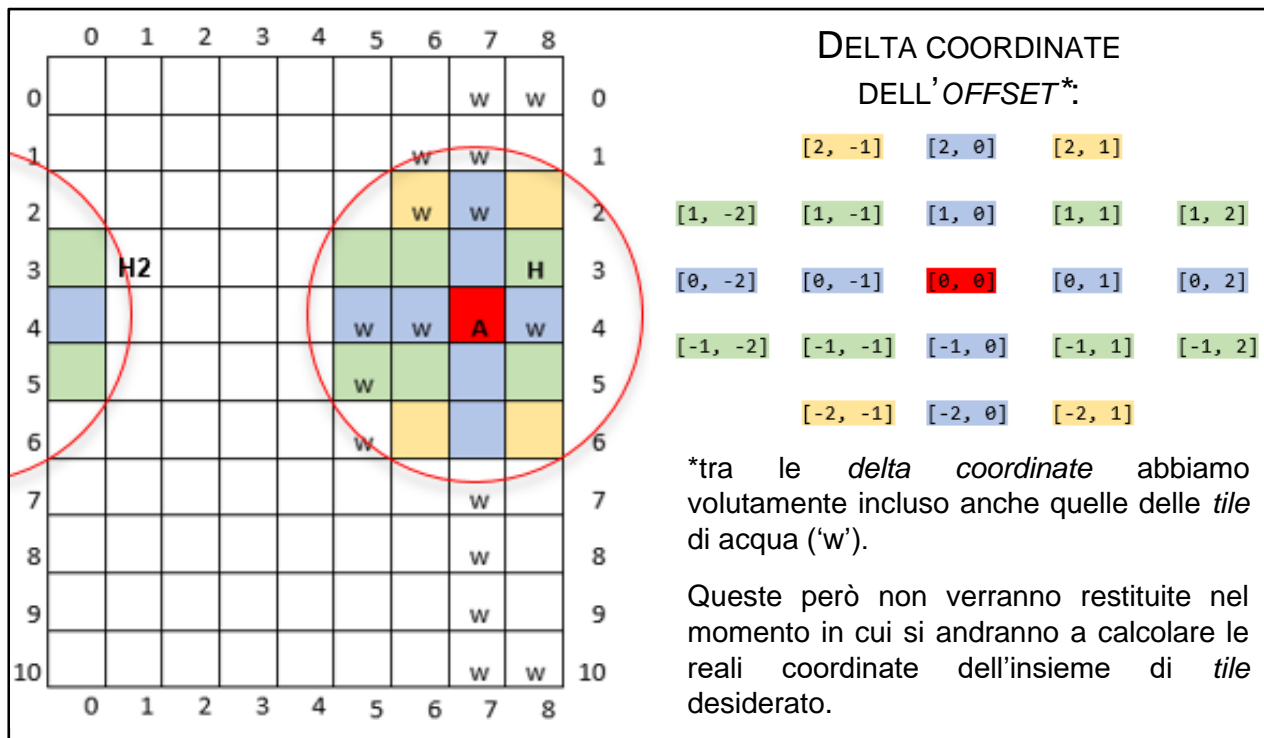
Si è deciso di escludere dall'insieme di *tile*, determinate per mezzo degli *offset*, quelle contenenti acqua.

Per avere più chiaro il concetto di *offset* si veda l'[esempio](#) nella pagina successiva.

ESEMPIO: GLI OFFSET E LA RICERCA STATICA

Si consideri la figura sottostante.

Si supponga che il raggio, detto *radius*, sia pari a 5 e che il punto di partenza sia **A**(4,7).



Intuitivamente si pensi di voler inscrivere un cerchio in un quadrato.

Il centro del cerchio, (0,0), in questo caso è **A**(4,7) con raggio pari alla radice quadrata di *radius* ossia $\sqrt{5} \approx 2,23$.

L'*offset* comprenderà tutte le *delta coordinate* che in figura risultano essere colorate.

Dall'*offset* si determineranno poi le *tile* che effettivamente circondano la sorgente. Tra di esse non saranno presenti quelle contenenti acqua ('w').

Si supponga che uno tra i moduli principali del programma abbia chiamato la *ricerca statica* con lo scopo di scoprire se in una determinata zona della mappa vi siano *hill* nemiche.

La ricerca ha come *sorgente* la formica **A** in posizione (4,7).

Supponendo di scegliere come *radius al quadrato* il valore 25, l'algoritmo di ricerca restituirà in output un insieme formato da un unico elemento ossia la *tile* **H** in posizione (3,8). La *hill* **H2** in posizione (3,1) è esclusa dalla soluzione poiché non appartiene all'insieme ricavato per mezzo dell'*offset*.

A. Modulo di combattimento

ATTENZIONE: Prima di affrontare questa sezione è necessario visionare la sezione dedicata alla [Risoluzione della battaglia](#).

Il modulo del combattimento è il primo tra i moduli ad essere mandato in esecuzione.

Questo modulo processa tutte le situazioni in cui le formiche degli agenti incontrano formiche nemiche ad una certa distanza, ossia *attackRadius*.

Si proverà a determinare le migliori mosse per le formiche dell'agente in una data situazione usando una ricerca **spazio-stati**, *simulando* le possibili mosse che potrebbero compiere l'agente ed il nemico e valutando le situazioni risultanti.

Dopo aver constatato che l'approccio naïve di costruire un albero degli stati del gioco con tutti i possibili movimenti delle formiche non fosse possibile: nella fattispecie, la posizione iniziale delle formiche per ogni mappa (e quindi l'intero albero successivo) è diverso, e può essere generato un numero praticamente infinito di mappe diverse che sono tutte considerabili valide (in confronto negli scacchi tutte le pedine hanno una posizione fissa all'inizio di ogni partita e tutt'oggi viene considerato un gioco complesso). L'approccio considerato è quello di riconoscere le situazioni di combattimento, ossia formiche vengono raggruppate in *cluster*, e simulando mosse di gruppo che avvengono in modo sincronizzato; ciò consente sofisticate tecniche di combattimento e allo stesso tempo riduce la complessità computazionale.

Il modulo entra in azione nel momento in cui una formica del bot riconosce una situazione di pericolo, fanno partire il primo dei quattro elementi che costituiscono questo modulo. Dopo aver trovato diversi gruppi di formiche avversarie, un generatore di mosse (2° componente), che genera possibili mosse che devono essere eseguite dai gruppi di formiche. In seguito una funzione di valutazione (3°), assegna un valore a una data situazione sulla base delle perdite di combattimento da parte di entrambe le parti, e l'algoritmo di ricerca (4°) principale combina questi elementi per identificare la mossa migliore per le formiche dell'agente in una situazione di combattimento.

Riconoscimento di situazioni

Dopo aver incontrato una formica nel raggio di visione di una delle formiche dell'agente, il modulo provvederà prima a trovare tutte le formiche che potrebbero partecipare alla potenziale battaglia. L'algoritmo utilizza una ricerca adattiva limitata al raggio in ogni fase di espansione. Funziona come segue

1. Inizializza il set di formiche con la formica originale
2. Inizializza il set di formiche nemiche con il nemico originale avvistato
3. Ripeti quanto segue, finchè non vengono aggiunte nuove formiche
 - a. Esegui un adaptive search con il set delle formiche dell'agente alla ricerca di formiche nemiche, limita il raggio a $3 \cdot \text{attackradius}$.
Aggiungi tutte le formiche trovate al set di formiche del nemico
 - b. Esegui un adaptive search con il set delle formiche del nemico alla ricerca delle formiche dell'agente, limita il raggio a $3 \cdot \text{attackradius}$.
Aggiungi tutte le formiche trovate al set di formiche del agente.

I due set di formiche risultanti saranno considerati la situazione corrente. La situazione è considerata valida fin quanto il set non è vuoto.

Generatore di mosse

Il generatore di mosse è la parte del modulo di simulazione di combattimento che genera le possibili mosse di uno dei gruppi avversari in una data situazione. Le mosse generate vengono simulate sulla situazione data, generando un insieme di stati figli. Questi vengono quindi valutati dalla funzione di valutazione e viene scelta la mossa migliore. Il generatore di mosse è in grado di generare i seguenti tipi di mosse:

- **attack:** nella mossa di attacco, ogni formica di un lato inizia una ricerca alla volta della formica o della collina nemica più vicina e cerca di eseguire un passo nella direzione di quel nemico.
- **Hold:** con questa mossa, ogni formica proverà a mantenere una distanza esatta dalle formiche nemiche, in modo da rimanere appena fuori dalla distanza di attacco (ingaggio). La distanza target sarà $2 + \text{attackradius}$ per le formiche dell'agente (supponendo che la parte nemica possa fare una mossa prima della risoluzione del combattimento) e $\text{attackradius} + 1$ per le formiche nemiche (poiché la risoluzione del combattimento prende parte immediatamente dopo che le formiche nemiche si muovono). Questa mossa implementa una mossa sicura di avanzamento e ritirata in una sola, poiché a seconda della distanza attuale dal nemico, le formiche saranno avanzate o si ritireranno per raggiungere la distanza richiesta.
- **Idle:** con questa mossa nessuna formica del gruppo sarà spostata.
- **Directional:** le quattro mosse direzionali cercheranno semplicemente di muovere tutte le formiche dell'agente (o del nemico) in una direzione: nord, sud, est o ovest.

Durante lo sviluppo sono state implementati e testati più tipi di mosse, ma alla fine non sono stati apportati miglioramenti visibili alle prestazioni rispetto questo set di mosse. Poiché ogni tipo di mossa aggiunta aumenta il fattore di ramificazione nella ricerca, si è deciso di lasciare tutte le mosse non essenziali e di concentrarsi sull'ottimizzazione dei tipi di base.

Funzione Euristica di Valutazione

La valutazione dello stato si basa principalmente sulla quantità di perdite su entrambi le parti nella risoluzione del combattimento nel nodo corrente e lungo tutti i nodi lungo il percorso migliore del bambino più in basso nell'albero. Le perdite dei nemici contribuiscono positivamente sul il risultato della valutazione, mentre le perdite delle formiche dell'agente lo riducono. Viene dato un bonus per l'eliminazione completa di tutte le formiche nemiche e un malus viene sottratto se tutte le formiche dell'agente vengono perse. Il valore delle perdite nemiche aumenta se il numero di formiche dell'agente è sufficientemente superiore al numero di formiche nemiche (forza preponderante), o quando il gioco si avvicina al turno finale.

Tutti i valori numerici sono stati regolati manualmente attraverso più esecuzioni di test e hanno determinato un comportamento appropriato. Si è tentato senza successo di mettere a punto la funzione di valutazione utilizzando algoritmi genetici.

Ricerca spazio-stato min-MAX

Dopo aver identificato la situazione, viene creato un albero di ricerca per determinare la mossa migliore per l'agente.

In Ants, tutti i giocatori muovono le loro formiche contemporaneamente. Il motore di ricerca quindi tratta ogni turno di gioco di Ants come due strati di ricerca: uno è la mossa dell'agente e un altro è il movimento del nemico e la risoluzione del turno (combattimento e cibo). Per questo motivo, la profondità di ricerca è sempre uniforme.

La ricerca viene inizializzata con un nodo neutro (lo 0) contenente lo stato di gioco corrente e la situazione riconosciuta (gruppi di formiche). Successivamente, il generatore di spostamento viene utilizzato per generare i nodi per il successivo strato di ricerca. Dopo che la mossa del nemico è stata eseguita, hanno luogo la risoluzione del combattimento e la raccolta di cibo. Ogni nodo viene quindi valutato utilizzando la funzione euristica.

Gli insiemi di nodi figlio di ogni nodo dell'albero del gioco sono insiemi di alberi ordinati. L'aggiunta di un nodo figlio appena generato e valutato lo colloca nell'albero secondo l'ordine naturale, che si basa sulla valutazione dei nodi. Pertanto, il primo elemento della struttura ad albero contiene sempre il miglior nodo di spostamento, eliminando la necessità di un ordinamento esplicito.

La profondità di ricerca è dinamica e limitata da una soglia superiore così come dal tempo assegnato alla simulazione di combattimento.

La simulazione viene interrotta alla massima profondità o quando non è disponibile tempo sufficiente sino alla scadenza del turno.

I nodi figli vengono generati dal generatore di mosse.

A seguito di ogni movimento nemico, viene invocata la risoluzione del combattimento e la raccolta di cibo.

I nodi vengono valutati quando vengono aggiunti alla struttura ad albero dei nodi figlio di ogni nodo.

B. Modulo di difesa



La strategia adottata per la difesa delle proprie *hill* è la seguente:

1. Controllare la presenza di formiche nemiche attorno alle *hill* dell'agente;
2. Controllare il numero di formiche che si hanno a disposizione per la difesa delle proprie *hill*;
3. Posizionare una o più formiche, in base al numero di nemici che accerchiano le *hill*, in 4 zone strategiche⁸ circostanti le *hill*: NORD-OVEST (1,-1), NORD-EST (1,1), SUD-EST (-1,1) e SUD-OVEST (-1,-1).

	-1	0	1
1	○		○
0		H	
-1	○		○

Non appena una formica nemica entra nel *viewradius2* della *hill*, una formica dell'agente viene mandata in difesa del proprio nido.

Le informazioni su ciò che circonda la *hill* non vengono, però, fornite esplicitamente.

Una *hill* è, infatti, un'entità immobile ed inanimata che non è in grado di osservare l'ambiente circostante; non fornisce, infatti, informazioni su formiche nemiche, *tile* di acqua, etc. che la circondano.

Risulta dunque necessaria la presenza di almeno una formica nella zona ad essa circostante per poter individuare possibili minacce.

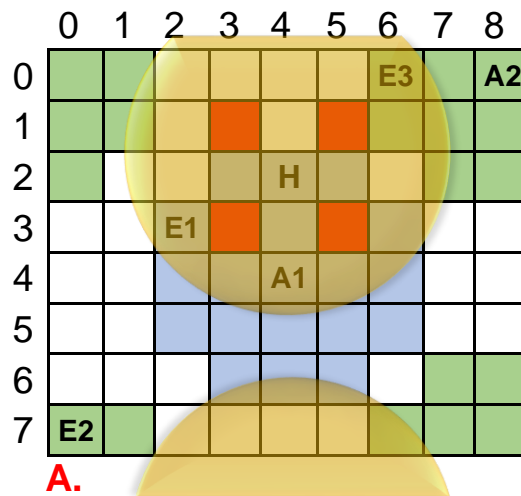
Come già anticipato nel momento in cui si è parlato dell'[indice di visibilità di una tile](#), abbiamo fatto in modo che almeno una formica del bot restasse nei pressi della *hill* in modo da poter parzialmente (o totalmente, nel caso in cui ne nascesse una in quel momento) osservare ciò che la circonda.

Ciò viene fatto in modo da rendere le *hill* più facilmente raggiungibili dalle sue formiche (in quanto ce ne sarà almeno una nelle loro prossimità) ed anche per poter osservare i cambiamenti del mondo attorno ad esse.

⁸ Non sempre sono disponibili tutte e quattro le postazioni. Talvolta la mappa è ideata in modo che in uno di quei punti vi sia dell'acqua.

ESEMPIO: MODULO DI DIFESA

Si supponga la seguente situazione:



Si consideri la mappa **A.** sovrastante, avente 8 righe e 9 colonne:

1. Sono presenti due formiche con ID = 0: **A1**, in posizione (4,4), e **A2**, in posizione (0,8);
2. Il raggio di visione della formica **A1** è quello contrassegnato in blu (sono incluse le tile rosse in posizione (3,3) e (3,5));
3. Il raggio di visione della formica **A2** è quello contrassegnato in verde;
4. La *hill*, **H**, in posizione (2,4), ha ID pari a 0;
5. Sono presenti tre formiche nemiche (con ID \neq 0): **E1**, in posizione (3,2), **E2**, in posizione (7,0), ed **E3**, in posizione (0,6);
6. Il raggio di visione di **H** è la circonferenza gialla che include **A1**, **E1** ed **E3**.

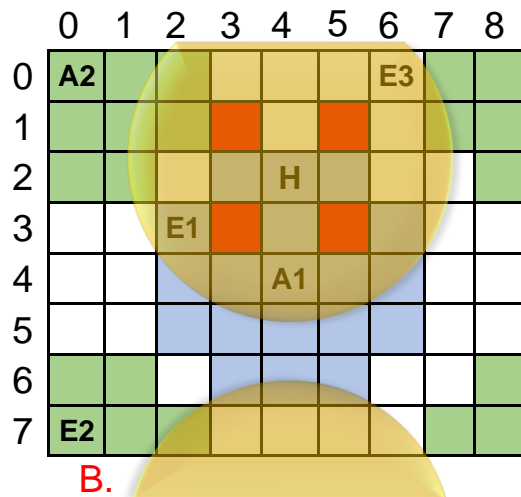
A1, accorgendosi della presenza di **E1** nel raggio di visione della *hill*, si andrà a posizionare in una delle 4 postazioni di difesa: (1,3), (1,5), (3,3) o (3,5).

Stessa cosa vale per **A2** che, vedendo **E3** nei pressi della *hill*, si muove in difesa.

E2, nella situazione attuale, non sembra essere una potenziale minaccia per la *hill*; supponendo infatti l'assenza di **E3**, **A2** non sarebbe accorsa in difesa.

Fortunatamente **A1** e **A2** si trovano nei pressi di **H** e sono in grado di individuare i due nemici nelle sue vicinanze.

Il *modulo di difesa*, notando la presenza di due nemici nei pressi di **H** ed avendo a disposizione esattamente due formiche, le manda *entrambe* in difesa in due tra le quattro *tile* contrassegnate in rosso.



B.

Si supponga lo scenario illustrato in figura B..

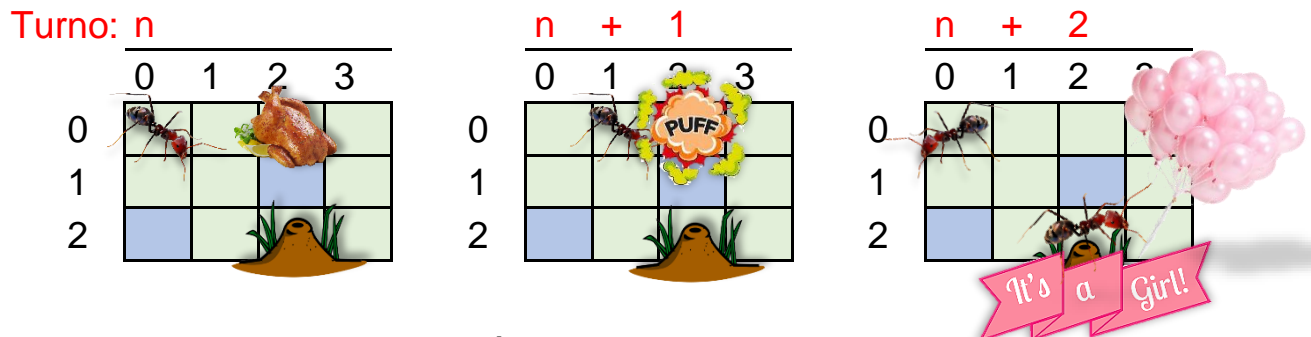
A2 è in posizione (0,0) (in figura **A.** si trovava in posizione (0,8)), il resto invece rimane invariato rispetto all'esempio precedente.

Si noti che la formica **E3** non può essere vista né da **A1**, né da **A2** e che, allo stesso tempo, minaccia la *hill* dell'agente.

In questo scenario, solo **A1** si sposterà in difesa poiché **A2** ignora la presenza di un ulteriore nemico nei pressi della propria *hill*.

C.Modulo di riproduzione (e di raccolta del cibo)

Come anticipato, la raccolta del cibo ha come conseguenza la nascita di una nuova formica.



La raccolta del cibo è un task che viene svolto in [autonomia](#) dalla formica ma è anche molto faticoso, tant'è che dev'essere svolto ad una certa distanza (*spwanradius2*).

Raccogliendo cibo, il formicaio dell'agente aumenta il suo numero ed un gran numero di formiche nel campo di gioco fornisce una serie di vantaggi:

1. Permette di avere una visione più ampia della mappa;
2. Offre chance più alte nella vincita di una battaglia;
3. Permette di dividere in modo equo i task, dettati dai vari moduli, a tutte le formiche del formicaio.

Il **modulo di riproduzione e di raccolta del cibo**, è creato affinché le formiche raccolgano quanto più cibo possibile. È eseguito durante il calcolo del turno, dopo il modulo di simulazione e combattimento, per evitare di mettere in pericolo le formiche quando sono inviate a raccogliere cibo. La raccolta di cibo in situazione di pericolo è gestita dal modulo di simulazione e combattimento, quindi questo modulo si occupa solo di una raccolta efficiente del cibo in situazioni sicure. Quando viene eseguito avvia una ricerca avendo in input le **formiche disponibili**⁹ e le *tile* di cibo che sono attualmente in vista dalle formiche dell'agente in quel turno.

La ricerca restituirà degli ordini che potranno essere (o meno) restituiti in output al server.

⁹ Le **formiche disponibili** sono quelle a cui non è stato ancora assegnato alcun ordine.

D. Modulo di attacco



Lo scopo del gioco è quello di radere al suolo *hill* nemiche e di uccidere i propri avversari.

Il **modulo di attacco** si occupa di mandare le formiche dell'agente verso *hill* e formiche nemiche che possono essere viste nel turno corrente.

Esso si struttura in modo analogo a quello del modulo di [raccolta del cibo](#).

Effettua, infatti, una ricerca avendo in input le formiche disponibili le formiche nemiche e le *hill* avversarie.

La ricerca restituirà degli ordini che potranno essere (o meno) restituiti in output al server.

Il modulo è responsabile esclusivamente di dirigere le formiche sulle *hill* nemiche in modo pacifico, in una situazione di combattimento, il codice di combattimento dirigerà le formiche sulla collina nemica quando possibile.

Il sistema di difesa conta le formiche nel raggio visivo della/delle *hill* dell'agente. Se questo numero supera determinate soglie, effettua una ricerca la quale dirigerà le formiche disponibili su prefissati target, con il risultato che alcune formiche faranno da "sentinelle" o "guardie" del nido e saranno pronte a difenderlo. I target sono stati scelti in modo da permettere la normale produzione (spawing) di formiche da parte del nido. Alla vista delle formiche nemiche, le stesse formiche dell'agente saranno pronte a difendere il nido.

NB. Questo modulo si occupa semplicemente di spostare le formiche verso gli avversari ma non effettua nessuna computazione che riguardi il combattimento; di ciò se ne occupa soltanto il [modulo di combattimento](#).

E. Modulo di esplorazione



È bene avere il controllo della maggior parte della mappa in modo che il bot sia in grado di raccogliere più cibo per creare più formiche dandogli una maggiore chance per radere al suolo *hill* nemiche mentre difende le proprie. La diffusione di formiche attraverso l'area avviene al fine di ridurre al minimo la distanza media e continuare a mantenere un'ampia area visiva.

Il **modulo di esplorazione** si occupa appunto di spargere le *formiche disponibili* nella mappa seguendo determinati criteri di priorità:

1. Mandarle verso [zone inesplorate](#);
2. Mandarle verso [zone \(attualmente\) invisibili](#);
3. Mandarle verso [zone prioritarie](#);
4. Farle allontanare l'una dall'altra ([distanziamento](#)).

Di seguito ogni task è trattato singolarmente.

Modulo di esplorazione: zone inesplorate

Si è deciso di tenere traccia delle zone che non sono mai state viste e dunque esplorate dalle formiche dell'agente.

Questo sotto-modulo si occupa di avviare una ricerca che ha in input le *tile* inesplorate e le formiche disponibili durante quel turno.

La ricerca restituirà degli ordini che potranno essere (o meno) restituiti in output al server.

Modulo di esplorazione: zone invisibili

Come già anticipato nelle [impostazioni del gioco](#), il gioco sfrutta il concetto di *fog of war* secondo cui le formiche sono in grado di vedere entro un certo raggio, *viewradius*, superato il quale tutto il resto rimane ignoto.

Le zone invisibili sono quelle in cui domina questa nebbia ossia quelle che nel turno corrente non sono nel raggio di visione di nessuna delle formiche dell'agente.

Questo sotto-modulo si occupa di avviare una ricerca che ha in input le *tile* attualmente invisibili e le formiche disponibili durante quel turno. La ricerca restituirà degli ordini che potranno essere (o meno) restituiti in output al server.

Modulo di esplorazione: zone prioritarie

Abbiamo individuato tre zone prioritarie verso cui far spostare le formiche:

1. Zone inesplorate;
2. Hill nemiche;
3. Formiche nemiche.

Questo sotto-modulo si occupa di avviare 3, seguendo l'ordine in cui sono state elencate qui in alto.

Ogni ricerca ha in input le *tile* prioritarie e le formiche disponibili durante quel turno; le ricerche saranno avviate ciclicamente, una dopo l'altra; l'iterazione terminerà nel momento in cui si giungerà ad una delle tre condizioni sottostanti:

1. Non ci sono più formiche disponibili;
2. Non ci sono più target disponibili;
3. Tutte e tre le ricerche non hanno restituito percorsi (non sono stati restituiti ordini) partendo dalle formiche verso *tile* prioritarie.

Il modulo restituirà degli ordini che potranno essere (o meno) restituiti in output al server.

Modulo di esplorazione: distanziamento

Può accadere che i sotto-moduli precedenti terminino senza aver assegnato ordini a tutte le formiche dell'agente.

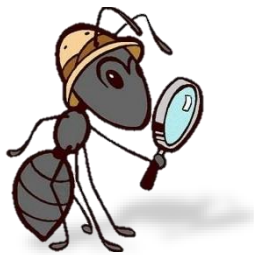
Nel caso in cui ci dovessero essere ancora delle *formiche disponibili*, il sotto-modulo del distanziamento si occupa di distanziarle di un passo dalle loro alleate.

Questo viene fatto per due motivi:

1. Massimizzare la percentuale di copertura della mappa;
2. Fare in modo che le formiche non restino ferme (non è sempre bene che le formiche non si muovano, a meno che non si tratti di una situazione di combattimento).

Questo sotto-modulo si occupa di avviare una ricerca che ha in input le *formiche disponibili* e, come target, tutte le formiche dell'agente. La ricerca restituirà degli ordini che potranno essere (o meno) restituiti in output al server.

F. Modulo di visione



Il **modulo di visione** è il primo modulo ad entrare in esecuzione all'inizio del turno dell'agente.

Il suo compito è quello di fornire agli altri moduli informazioni riguardo le *tile* della mappa distinguendo quelle attualmente visibili dalle formiche da quelle coperte dalla *fog-of-war* (vd. [impostazioni del gioco](#)).

Fornisce i metodi per poter impostare tutte le *tile* ad "invisibili", in modo da aggiornare quelle in vista nel turno corrente.

Aggiorna le seguenti informazioni in base a ciò che sono in grado di vedere le formiche nel turno di riferimento:

4. Le *tile* in vista ossia quelle di cui l'agente ha ottenuto informazioni esplicite;
5. Le *tile* che le formiche non riescono a vedere (utili per il modulo di esplorazione nel momento in cui si ricercano le [zone invisibili](#));
6. I nemici che ogni formica riesce a vedere e la distanza che intercorre tra di loro;
7. Le *hill* dell'agente che non sono state rase al suolo (e che necessitano, dunque, di essere difese).

Per ottenere le *tile* visibili si fa uso degli [offset](#).

Ogni *tile* contenuta nell'insieme risultante dall'offset con raggio *viewradius2* viene contrassegnata come visibile.

Questo modulo si occupa, inoltre, dell'immagazzinamento e del calcolo degli *offset*.

G. Modulo del tempo

Uno dei limiti computazionali dell'agente è il tempo a disposizione per le varie computazioni. Il modulo del tempo è responsabile di tenere traccia del tempo a disposizione durante il turno dell'agente e di informare il modulo di combattimento di quanto tempo avrà presumibilmente a disposizione per completare la sua esecuzione. I tempi vengono misurati per tutti i moduli effettuando una media pesata mobile dei tempi di esecuzione durante gli ultimi turni. Questi tempi misurati vengono quindi utilizzati per prevedere i tempi di esecuzione di questi moduli nei turni successivi e per allocare il tempo del modulo di combattimento (quello il cui dispendio temporale è più elevato).

a. Tempo modulo di combattimento

Il modulo di combattimento ha di gran lunga i tempi di esecuzione più lunghi di tutti i moduli, e quindi viene gestito separatamente, come parte della logica di temporizzazione implementata nel modulo di combattimento stesso. In base ai tempi di esecuzione previsti degli altri moduli, al modulo di combattimento viene assegnato quanto più tempo possibile per l'esecuzione, consentendogli di massimizzare la profondità di ricerca per quante più battaglie possibili.

Quando il modulo di combattimento viene avviato, riceve un timestamp che indica il tempo massimo che ha a disposizione. Il riconoscimento della situazione viene eseguito per primo, identificando tutte le battaglie che devono essere simulate e tutti i gruppi di formiche che partecipano a quelle battaglie. Il tempo disponibile per la simulazione viene quindi suddiviso tra queste battaglie in base alla quantità di formiche in ciascuna battaglia-

Le simulazione di combattimento vengono quindi eseguite, con le battaglie più piccole che vengono simulate per prime. Il motivo è semplice: se le battaglie più piccole finiscono la loro simulazione in meno tempo di quanto le è stato assegnato, il tempo in eccesso viene riassegnato alle battaglie più grandi, che verranno eseguire in seguito.

b. Tempo negli altri moduli

Per ogni modulo eccetto il modulo di combattimento, il tempo di esecuzione viene registrato dal modulo di temporizzazione. Ad esempio per il modulo di difesa la previsione del tempo di esecuzione è data dalla seguente formula di exponential smoothing:

$$\widehat{T_{difesa}(t+1)} = T_{difesa}(t) * 0,8 + \widehat{T_{difesa}(t)} * 0,2$$

Per determinare il tempo disponibile per il modulo di combattimento, è utilizzata la seguente formula

$$T_{combat} = T_{remaining} - \sum_m E_m$$

Dove $T_{remaining}$ è il tempo totale rimanente finché il turno non termini e E_m è la stima del tempo di esecuzione per ogni modulo m eseguito dopo il combattimento.

4. Conclusioni

Senza ombra di dubbio, la parte più difficile della creazione dell'agente, oltre alle ovvie problematicità iniziali, è stata l'implementazione di un sistema di combattimento.

Questo elemento è risultato particolarmente impegnativo a causa dell'elevato numero di mosse possibili di un gruppo di formiche, combinato con il rigoroso limite di tempo.

Tutti i concorrenti che hanno raggiunto le prime posizioni nella sfida originaria, hanno implementato sofisticati sistemi di combattimento.

Alla fine dello sviluppo dell'agente, la performance osservabile in relazione agli obiettivi inizialmente fissati (ossia, non *dummy*) è stata ritenuta *soddisfacente*. L'agente sarebbe stato sicuramente in grado di gareggiare nel contest e giocare in modo intelligente, pur mantenendo parecchi limiti producendo risultati dignitosi.

All'inizio del gioco, l'agente inizia ad esplorare l'area intorno alle hill di partenza. Man mano che più formiche vengono generate dal cibo raccolto, l'orizzonte di esplorazione si allarga e le formiche si dividono per esplorare diverse aree della mappa.

La raccolta del cibo è fondamentale per la sopravvivenza dell'agente.

Tutti gli alimenti visibili vengono raccolti dalle formiche vicine.

In situazioni di combattimento, il cibo viene raccolto e viene impedito che gli avversari lo raccolgano (quando è possibile).

L'agente mostra prestazioni non molto eccezionali in combattimento. Impegna le formiche nemiche in vista, si ritira quando trova contro una forza nemica superiore, utilizza il terreno circostante e insegue aggressivamente il nemico quando è in vantaggio ma non riesce ad ottenere rinforzi dalle formiche circostanti nell'area.

L'agente è in grado di prendere decisioni in combattimento sulla base di previsioni di più turni, evitando di rimanere intrappolato nel terreno e conducendo formiche nemiche in trappole.

Tutto il territorio esplorato viene mantenuto visibile fino alla fine del gioco, numero di formiche permettendo.

Per questo viene utilizzato solo un numero minimo di formiche al fine di fornire il maggior numero possibile di formiche all'esplorazione e al combattimento. Questo si traduce nella raccolta completa del cibo per la deposizione delle uova.

Il sistema di cronometraggio (timing) funziona bene.

I test hanno rilevato che aumentando la profondità di ricerca minima dell'algoritmo MIN-MAX l'agente avrebbe superato il tempo limite del turno.

La prestazione finale non è buona come ci si aspettava durante lo sviluppo dell'agente.

Il sito della challenge metteva a disposizione un forum su cui i partecipanti potevano confrontarsi sulle varie strategie per affrontare al meglio la sfida.

Dopo aver analizzato i risultati, gli approcci e le implementazioni di altri concorrenti, nonché l'osservazione delle discussioni sui forum di AI Challenge, è stata compilata una raccolta di ragione che spiega questo risultato.

PRESTAZIONI DELUDENTI:

- L'agente non è in grado di fronteggiare gruppi di formiche che lo stanno circondando da direzioni diverse, non reagendo in modo ottimale.
- Il comportamento difensivo in gruppi molto piccoli porta a perdere terreno, in quanto non vengono richiamati rinforzi.
- In molte occasioni l'agente ingaggia formiche nemiche in combattimento e continua a perseguirle attraverso la mappa, anziché interrompere l'inseguimento e dirigersi verso altri obiettivi che reputa più importanti.

MIGLIORAMENTI: Sulla base delle prestazioni del nostro agente e del confronto con le implementazioni degli altri concorrenti, sono stati identificati i possibili miglioramenti, i quali avrebbero contribuito ad una migliore prestazione complessiva.

- Implementare test in tutti i moduli, consentendo una migliore scoperta di errori di programmazione e portando ad un numero minore di questi nella presentazione corrente.
- Implementare l'approccio di ottimizzazione genetica in una forma modificata. Il quale permetterebbe un'ottimizzazione per la funzione di valutazione utilizzata nella logica di combattimento, riducendo drasticamente il tempo di simulazione richiesto.