Relazione progetto ia

Progetto: Agente Snake

Antico loco della scientia in Mediolanum

Intellighenzia delli spiriti metallici

Relazione Finale

# Introduzione

Il progetto oggetto di discussione verte sulla progettazione, programmazione e validazione di un sistema di intelligenza artificiale atto alla simulazione delle dinamiche tipiche di una operazione militare di tipo Search and Rescue (SAR) nella quale uno o più agenti della fazione A devono recuperare un obiettivo VIP all’interno di un’area sorvegliata dalla fazione E. Nella simulazione proposta ad affrontarsi vi sono entità virtuali dotate di specifici livelli di consapevolezza dell’ambiente circostante e capacità decisionali.

L’entità cardine, dotata del più alto grado di indipendenza, è rappresentata dalla figura dell’Agente Solid Snake: egli ha compito di infiltrarsi all’interno del territorio della fazione E per salvare un commilitone caduto prigioniero senza essere individuato.

Gli antagonisti dell’Agente Solid Snake sono concretizzati nella figura delle sentinelle: esse pattugliano l’ambiente della simulazione e rilevano qualsiasi presenza all’interno della personale area di influenza.

Il nucleo del progetto, come si intuisce, ruota attorno al problema della pianificazione e successiva attuazione di una sequenza di movimenti che permettano a Solid Snake di raggiungere l’ostaggio senza entrare nell’area di influenza delle sentinelle. Il modello utilizzato è quello relativo ai problemi di attraversamento di un grafo con nodi ed archi pesati e l’utilizzo di una euristica spaziale.

Il contesto “ludico” della simulazione è un omaggio al famoso Metal Gear Solid, videogioco d'azione stealth, il terzo in ordine di della serie Metal Gear ideata da Hideo Kojima. Metal Gear Solid è uscito nel 1998 su PlayStation.



## Tecnologie utilizzate

Per la realizzazione del presente progetto ci si è affidati all’interprete Prolog SWI-Prolog, utilizzando l’editor integrato nella versione Windows per scrivere il codice.

Il docente ha messo a disposizione un framework che ha permesso la costruzione del sistema di ragionamento dell’agente. In particolare, il framework comprende:

* Un modulo per l’implementazione di alberi di ricerca;
* Un meta-interprete per il ragionamento basato sulle assunzioni;
* Una base di dati dinamica per la conoscenza dell’agente;
* Un type checker;
* Un modulo che combini ricerca e assunzioni per permettere di definire un flusso di decisioni per l’agente.

# Rappresentazione del mondo esterno

Nel capitolo corrente verrà presentato il modello usato nella simulazione per la rappresentazione del mondo esterno con il quale l’entità protagonista, Solid Snake, dovrà interagire. Il mondo esterno è definito nelle sue componenti fondamentali tramite moduli distinti, rispettivamente per la gestione del tempo, delle sentinelle e della topografia del terreno simulato.

## Modulo tempo.pl

Tramite il modulo tempo.pl viene esposta la modellizzazione del concetto di tempo, e del suo scorrere, usata dal simulatore.

Il tempo si concretizza mediante l’utilizzo di un contatore, a numeri interi positivi, che viene incrementato costantemente durante tutto il processo della simulazione: ad ogni incremento del contatore corrisponde la fine un ciclo di elaborazione completo da parte dell’entità Solid Snake.

Il modulo espone tre predicati:

* clock(TEMPO) che funge da contatore
* aggiorna\_clock() che incrementa il contatore
* azzera\_clock() che azzera il contatore; tale operazione viene effettuata quando si riscontrano le condizione per la terminazione della simulazione

## Modulo sentinella.pl

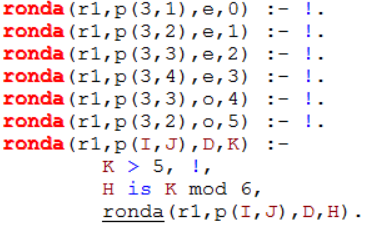
Il modulo sentinella.pl espone il modello usato per simulare l’azione di una sentinella dotata di un livello minimo di intelligenza. Ogni sentinella è caratterizzata da un percorso di ronda e da un’area di influenza all’interno della quale è in grado di individuare gli agenti nemici.

Sono state definite, all’interno dei file ronda\_X.pl, due distinte tipologie di ronda: con percorsi deterministici e con percorsi non deterministici. Entrambe le tipologie si basano sul predicato

ronda(ID\_RONDA, PUNTO\_RONDA, DIREZIONE, TEMPO\_VISITA)

### Percorsi di ronda deterministici

Un percorso di ronda deterministico rappresenta un classico circuito di ronda ed è definito da una successione di punti che la sentinella dovrà visitare in sequenza e senza possibilità di variazioni. Di seguito viene proposto il codice che descrive una ronda deterministica.



### Percorsi di ronda non deterministici

Per rendere la simulazione più interessante e verosimile, alcune sentinelle sono state dotate di un comportamento non deterministico: durante la ronda, capita che una stessa sentinella prenda una strada diversa da quella presa nell'iterazione precedente. Il comportamento tipico di una sentinella che segue una ronda non deterministica è schematizzato nel seguente diagramma.



* Nello stato *q0*, la sentinella si trova nella fase iniziale (deterministica) della ronda, composta di un certo numero di passi, al termine dei quali viene estratto un valore casuale compreso tra 0 e 1. A seconda del valore estratto, la sentinella passa allo stato *q1* o *q2*;
* Negli stati *q1* e *q2*, la sentinella prosegue la ronda in due modi diversi (a seconda dello stato). È importante che entrambe i percorsi alternativi abbiano la stessa lunghezza, altrimenti la sentinella non potrebbe risincronizzarsi con la fase deterministica del percorso;
* Nello stato *q3*, la sentinella ha raggiunto il punto finale delle ronde appartenenti a *q1* o a *q2*: tale punto è comune a entrambe le ronde. A questo punto, può fare altri passi (deterministici) e poi tornare allo stato *q0*.

In Prolog, lo stato attuale è indicato da un predicato dinamico

stato\_corrente(ID\_RONDA, STATO)

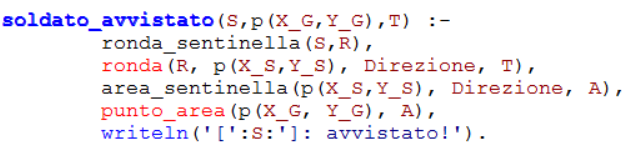
consultabile in ogni momento per stabilire il prossimo step della ronda.

### Area di influenza

L’area di influenza associata alla sentinella è un rettangolo 5 x 3 all’interno del quale la sentinella è in grado di identificare gli agenti nemici; tale capacità è espressa mediante il predicato

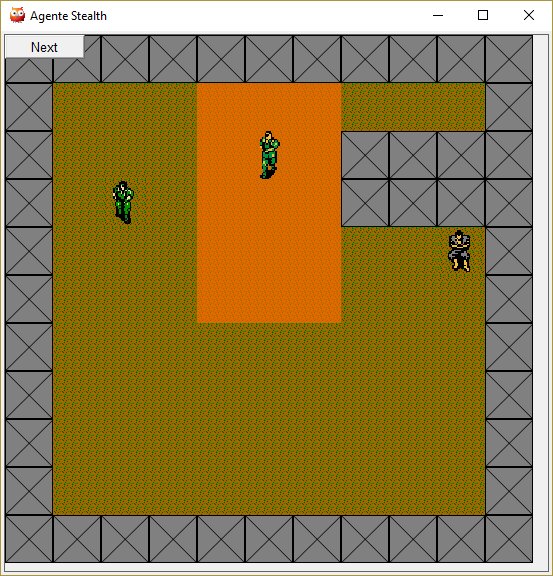
soldato\_avvistato(ID\_SENTINELLA, PUNTO, TEMPO)

di cui a seguire viene mostrato il codice



Il predicato punto\_area(PUNTO, AREA) è fornito dalle librerie di progetto ed è definito all’interno del file livello.pl

Nell’immagine seguente l’area di influenza è indicata in colore arancione



## Modulo livello.pl

Il modulo livello.pl definisce la geometria del mondo esterno usato nella simulazione.

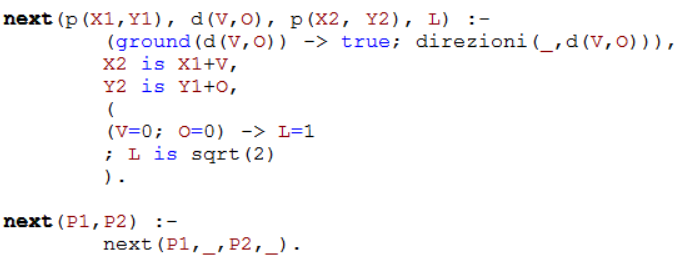
Fondamentale per il buon funzionamento del processo decisionale è la definizione del concetto di adiacenza tra punti dell’area di gioco. Tale concetto viene concretizzato dal predicato

adiacenti(PUNTO, VICINI)

che sfrutta il predicato di libreria

next(PUNTO\_1, PUNTO\_2)

di cui a seguire il codice

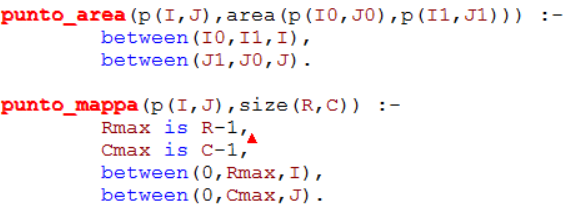


Un altro aspetto cardine per la simulazione è l’appartenenza di un punto ad un’area; concetto che viene utilizzato dal sistema per verificare l’avvistamento dell’agente Solid Snake da parte di una sentinella. I predicati che materializzano l’implementazione sono

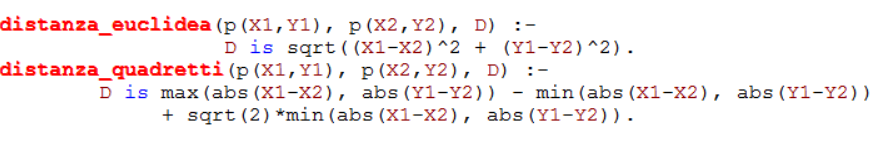
punto\_area(PUNTO, AREA)

e

punto\_mappa(PUNTO, DIMENSIONI\_MAPPA)



Molti degli algoritmi implementati nella simulazione, ad esempio l’euristica, richiedono il calcolo della distanza tra due punti dell’area considerata; è quindi importante predisporre un metodo di calcolo efficiente ed efficace. A tal proposito le librerie di progetto fornite dal docente implementano due diverse metodologie di calcolo della distanza tra punti che mostriamo di seguito



## Modulo GUI

Al fine di offrire una rappresentazione grafica più immediata rispetto a quella testuale, si è scelto di utilizzare il framework XPCE per rappresentare il mondo di gioco usando degli sprite, come se si trattasse di un vero videogioco.

L'implementazione della GUI si trova nel modulo gui.pl e si basa sull'utilizzo della versione *object oriented* di Prolog, su cui è costruito XPCE. In particolare, la finestra viene costruita mappando ogni elemento della matrice che rappresenta la mappa su una serie di sprite (contenuti nell'omonima directory *sprites*) e mandando all'oggetto che indica la finestra delle richieste di rendering per ogni casella da disegnare.

Nella GUI è presente anche un tasto *Next* che, se premuto, causa l'avanzamento della storia rappresentata dal programma.

La finestra della GUI viene aperta appena viene chiamato il predicato vai/1 e viene chiusa appena l'agente prende la decisione termino(evento). L'avanzamento della storia avviene tramite l'invio di un segnale asincrono al thread principale: il segnale viene inviato ogni volta che si preme il tasto *Next*.

# Rappresentazione interna dell’agente

Questo capitolo affronta le scelte di progettazione relative al comportamento dell'agente, esponendone e motivandone le caratteristiche e le relazioni con le librerie utilizzate.

## Il pattern decisionale dell’agente

L'interfaccia vai.pl fornita dal docente prevede l'implementazione del tipo *decisione*, utilizzato dall'agente per stabilire delle regole comportamentali durante l'esecuzione. Le decisioni possono determinare una pianificazione oppure rappresentare semplicemente gli step di esecuzione di un piano precedentemente elaborato. Nel progetto dell'agente stealth, le decisioni sono definite come segue.

* Decisioni che iniziano una pianificazione:
  + vado(punto, punto): l’agente decide di cercare un piano per andare da un punto a un altro della mappa;
  + attesa: l’agente decide di aspettare tre unità di tempo per osservare degli eventuali cambiamenti nella posizione delle sentinelle.
* Decisioni che eseguono gli step di un piano:
  + avanzo: l’agente fa un passo per avanzare nell’esecuzione del piano vado(\_,\_);
  + aspetto: l’agente sta fermo per un’unità di tempo;
  + termino(evento): l’agente termina l’esecuzione perché è stato avvistato, oppure ha raggiunto il prigioniero, oppure non è riuscito a trovare un percorso che gli permetta di raggiungere il prigioniero.

Verrà ora illustrato il processo di valutazione delle decisioni dell'agente, che trova un riscontro pratico nell'implementazione dei predicati decidi/3 di vai\_if.pl.

1. All'inizio della storia, l'agente decide di cercare un piano per raggiungere il prigioniero, facendo delle opportune assunzioni sui punti sorvegliati dalle sentinelle, Se lo trova, passa direttamente al punto 4;
2. Se non trova un piano, l'agente decide di aspettare per tre unità di tempo, per vedere se la configurazione delle sentinelle cambia e se viene rivelato un percorso sicuro;
3. L'agente cerca nuovamente un piano. Se lo trova, procede al punto successivo, altrimenti termina per impossibilità di raggiungere il prigioniero;
4. L'agente mette in pratica il piano. Se viene avvistato, termina per fallimento, altrimenti arriverà dal prigioniero e terminerà per raggiungimento del goal. Ad ogni passo, l'agente osserva le nuove posizioni assunte dalla sentinella e impara quali punti della mappa sono sorvegliati in quell'istante di tempo: tale informazione sarà utile nella prossima esecuzione sulla stessa mappa.

Per cercare un piano adatto al raggiungimento del prigioniero, l'agente utilizza la libreria di ricerca fornita dal docente, usando come euristica la distanza euclidea dal prigioniero, mentre la costruzione di un piano d'attesa consiste semplicemente nel restituire una lista contenente tre atomiche aspetto. Il pattern decisionale dell'agente è rappresentato qui di seguito mediante diagramma di flusso.

## Rappresentazione della conoscenza

La conoscenza dell'agente è relativa alla posizione delle sentinelle nel tempo. Affinché l'agente possa imparare e assumere delle informazioni in modo da operare significativamente nel mondo di gioco, ha bisogno di memorizzare (e ipotizzare) le posizioni delle singole sentinelle in ogni istante di tempo.

Per questo, l'agente utilizzerà un unico predicato assumibile, punto\_sorvegliato/3, la cui specifica è riportata di seguito:

**pred** punto\_sorvegliato(sentinella\_id, punto, tempo).

% punto\_sorvegliato(?S, ?P, ?T) SEMIDET

% Spec: vero sse il punto P e' sorvegliato (pericoloso) dalla sentinella S nel tempo T.

Come si può dedurre dalla specifica, il predicato punto\_sorvegliato/3 viene usato dall'agente per costruire un elenco dei punti della mappa in cui non dovrebbe trovarsi in certi istanti, poiché tali punti rientrano nell'area di visione di una sentinella.

L'implementazione degli assumibili nella libreria fornita dal docente richiede la specifica di determinate regole, mediante le quali l'agente decide se assumere un dato predicato o meno. In particolare, se necessario, è possibile implementare un predicato decide\_se\_assumere/2, specificato in vai\_if.pl, che viene usato ogni volta che l'agente si trova nelle condizioni di dover fare delle assunzioni (ovvero, ogni volta che viene chiamato il meta-interprete). Nel progetto dell'agente stealth si è scelto di regolare le assunzioni dell'agente, fornendogli due modalità distinte di assunzioni: una più “cauta” e una più “rischiosa”, che saranno affrontate nella

sezione “Regole di assunzione dell’agente”.

Le assunzioni (o la conoscenza effettiva) sui punti sorvegliati e la configurazione della mappa (nota all'agente) confluiscono nel predicato pensa\_sicuro/3, usato nel calcolo del percorso per il prigioniero. Quando l'agente, nell'esplorazione dell'albero delle decisioni, deve calcolare i vicini di un nodo, esclude tutti i nodi corrispondenti a punti in cui il predicato pensa\_sicuro/3 restituisce true.

Dato che, come spiegato nella sezione 2.2, le sentinelle possono avere un comportamento non deterministico, non esistono assunzioni contrarie. Potrebbe capitare, quindi, una situazione in cui l'agente ha imparato che (dunque *conosce*) la posizione della sentinella S nel tempo T sarà il punto P, ma che assuma anche che, sempre nel tempo T, la stessa sentinella S si trovi in un punto P1 diverso da P. Questo permette all'agente di pianificare un percorso che tenga conto anche del rischio di un cambio di ronda delle sentinelle.

## Implementazione del framework per la ricerca

Il framework per la ricerca fornito dal docente viene utilizzato nella ricerca di un percorso il più possibile breve per raggiungere il prigioniero. L’utilizzo del framework richiede l’implementazione dei predicati vicini/2, costo/3, h/2 e del tipo nc.

Il tipo nc è implementato rappresentando un nodo come un atomo passo(P,T), dove P è un punto della mappa e T è il tempo in cui l’agente prevede di trovarsi in quel punto, durante l’attuazione del piano. Questa implementazione rischia di rendere eccessivamente complesso il processo di ricerca, dando luogo a dei cicli dal punto di vista dei punti della mappa: ad esempio, l’albero di ricerca dell’agente potrebbe contenere più nodi (anche infiniti) corrispondenti allo stesso punto P della mappa, ma con un valore di T in passo(P,T) differente. Per questo motivo, si è deciso di adattare la strategia di taglio dei cicli per evitare che l’agente pianifichi di passare più volte per uno stesso punto. La nuova strategia è chiamata taglio\_cicli\_custom ed è possibile trovarla in lib\Ricerca, insieme alle altre strategie.

Il predicato vicini/2 riceve in ingresso un nodo e ne restituisce i gli adiacenti all’interno dell’albero di ricerca. I vicini sono calcolati considerando innanzitutto tutti i nodi corrispondenti a punti adiacenti al punto del nodo di cui si stanno calcolando i vicini. Dopodiché, da questi nodi vengono eliminati quelli a cui corrispondono ostacoli o sentinelle e quelli in cui l’agente prevede di essere avvistato. La ricerca procede poi esplorando i nodi rimasti.

Poiché si vuole simulare una missione di salvataggio da una situazione critica, il raggiungimento del prigioniero deve avvenire nel minor tempo possibile. Il costo di un singolo passo (così come di un’attesa) è fissato a 1, per fare in modo che non ci siano discrepanze temporali tra il movimento dell’agente e quello delle sentinelle. L’euristica di un nodo, invece, è calcolata sommando la distanza euclidea di quel nodo dal goal e il tempo passato fino a quel momento.

## Regole di assunzione per l’agente

Come anticipato nella sezione precedente, l'agente presenta due diverse modalità di assunzione, una più rischiosa e una più cauta. Prima di eseguire il programma, è possibile impostare la strategia di assunzione mediante il predicato strategia\_assunzioni/1. Segue una descrizione delle diverse strategie di assunzione.

### Modalità RISK

Si supponga che l'agente si trovi, al tempo T0, in una posizione P0 e che nella mappa ci sia una sentinella S in posizione Sp che guarda verso nord. L'agente inizia a pianificare il percorso e, quando chiama il predicato pensa\_sicuro/2, decide se assumere che un punto P1 sia sorvegliato o meno da una qualche sentinella. Tale decisione è affermativa nel caso in cui, detti Pne e Pso rispettivamente gli angoli nord-est e sud-ovest del campo visivo della sentinella, il punto P1 si trovi nel campo visivo della sentinella, oppure non oltre la riga che si trova a nord di Pne. Lo stesso ragionamento vale per tutte le altre direzioni in cui la sentinella può guardare.

In altre parole, l'agente assume che il campo visivo della sentinella si sposti (nella direzione in cui sta guardando) di non più di tre punti, ipotizzando un percorso di ronda rettilineo. Questo approccio, in generale, produce percorsi più brevi, ma avvistamenti più frequenti.

Si può osservare un esempio del ragionamento basato su assunzioni nella modalità risk in Fig. Risk.png. Le aree di colore arancione sono quelle relative al campo visivo delle sentinelle, mentre quelle rosse sono quelle in cui l'agente assume l'estensione del campo visivo.

### Modalità CAUTION

Come nel caso precedente, si assuma che l'agente, nel tempo T0, occupi la posizione P0, mentre la sentinella S occupi la posizione Ps e guardi verso nord. Quando l'agente pianifica il percorso, si ritroverà a chiamare più volte il predicato decide\_se\_assumere(punto\_sorvegliato(S, Ps1, Ti)). Per prima cosa, l'agente calcola la differenza temporale Tdiff = Ti - T0. Per ogni Tdiff > 0, l'agente assume che il punto Ps1 sia sorvegliato nel tempo Ti se e solo se tale punto rientra nel campo visivo attuale della sentinella, dilatato di Tdiff punti nella direzione in cui la sentinella guarda. In questo caso la dilatazione del campo visivo è molto più ampia e determina, come effetto nel mondo esterno, un percorso pianificato più lungo ma più sicuro.

Anche in questo caso, si può vedere un esempio in Fig. caution.png, con le stesse regole di visualizzazione dell'esempio precedente.

# Risultati sperimentali

Il comportamento dell'agente è stato studiato tramite l'esecuzione su 6 mappe differenti, ognuna pensata per studiare un particolare aspetto dell'implementazione. Questo capitolo espone i risultati dell'esecuzione sulle mappe da cui emergono i comportamenti più interessanti.

## Mappa 2

In questa mappa è presente una sola sentinella con ronda deterministica: tre passi avanti e tre passi indietro.

La mappa 2 è un esempio di come un approccio risk coincida con un approccio di tipo caution: in entrambi i casi il percorso calcolato è il medesimo, va però sottolineato come l’approccio risk porti ad esito positivo in maniera “fortuita” poiché risultano valide le assunzioni iniziali circa i movimenti della sentinella.

## Mappa 3

La mappa 3 contiene una sentinella con comportamento non deterministico:

1. Avanza di tre passi verso sud;
2. A seconda dello stato,
   1. Resta ferma;
   2. Avanza di altri due passi verso sud e poi torna indietro;
3. Torna al punto di partenza.

In questo caso, raggiungere il prigioniero in modalità risk potrebbe richiedere troppi fallimenti per costruire il percorso più sicuro: un percorso breve che funziona una volta, infatti, non necessariamente funziona anche all'iterazione successiva, perché la sentinella potrebbe inaspettatamente estendere il proprio campo visivo.

Un approccio in modalità caution, invece, causa al massimo un fallimento, prima di ottenere il percorso più sicuro.

## Mappa 4

Questa mappa, che contiene una sentinella dal comportamento deterministico (restando sempre nello stesso punto, alterna pattugliamento in direzione ovest e pattugliamento in direzione nord), serve a testare la capacità dell'agente di decidere di aspettare per qualche secondo, nel caso in cui non riesca a trovare un percorso per raggiungere il prigioniero. Aspettare per qualche secondo prima di prendere la decisione di terminare, infatti, può fare in modo che le assunzioni dell'agente si rivelino troppo “pessimistiche”, oppure che un percorso inizialmente sorvegliato dalle guardie venga momentaneamente “liberato”.

È questo il caso della mappa 4, in cui l'unica via per il prigioniero, inizialmente, sembra inaccessibile. Tuttavia, aspettando qualche secondo, la sentinella si gira verso nord e il percorso diventa accessibile in entrambe le modalità.

## Mappa 5

La mappa 5, contenente due sentinella (una con ronda deterministica e una con ronda non deterministica), è un esempio di mappa “difficile” per mettere alla prova l'agente in un ambiente con sorveglianza più alta.

In questa situazione l'agente dimostra un comportamento sufficientemente intelligente solo nella modalità caution, grazie alla quale genera un percorso particolarmente lungo ma di comprovata sicurezza.

Si nota invece come l’approccio risk costringa l’agente ad un eccessivo *trial and error* poiché, venendo forzato il calcolo del percorso più breve, si ottiene una situazione nella quale l’agente dovrà fallire innumerevoli volte nel tentativo di apprendere il maggior numero di posizioni spazio-temporali associate alle due sentinelle.

# Conclusioni

L’agente dimostra un comportamento soddisfacente nella totalità delle mappe se attivato in modalità caution, mentre dimostra delle discrete capacità di infiltrazione (dipendenti in gran parte dall’apprendimento basato su errori) se usato in modalità risk.

## Possibili miglioramenti

* Intelligenza nelle sentinelle: la Pursuit Evasion è un campo di ricerca nello studio della robotica e dell’intelligenza artificiale multi-agente, che mira a trovare algoritmi di esplorazione che consentano a dei pursuer (in questo caso le sentinelle) di esplorare un ambiente in modo da massimizzare la probabilità (se non assicurare) di trovare uno o più evader (in questo caso l’agente) che lo contaminano. In un possibile ampliamento del progetto, si potrebbe dotare le sentinelle di un’intelligenza più evoluta, implementando una versione opportunamente limitata di un algoritmo di Pursuit Evasion. Questa aggiunta, tuttavia, oltre a complicare di molto l’implementazione, implicherebbe la ricerca di una soluzione a problemi di elevata complessità (dell’ordine di NP-hard) per calcolare una strategia di pattugliamento sufficientemente intelligente.
* Nascondigli: si potrebbe decidere di complicare la conformazione del terreno inserendo dei “nascondigli”, ovvero dei punti della mappa in cui l’agente non allerta le sentinelle, anche se si trova nel loro campo visivo. Introducendo i nascondigli si dovrebbe modificare il pattern decisionale dell’agente per tenerne conto: ad esempio, nel caso in cui l’agente non riesca a trovare un percorso per il goal, potrebbe dirigersi innanzitutto verso il nascondiglio più vicino. A quel punto, l’agente potrebbe osservare l’ambiente per controllare se le posizioni delle sentinelle siano cambiate al punto di liberare un passaggio verso il goal o verso il nascondiglio successivo.
* Attesa nella ricerca del piano: Si potrebbe consentire all’agente di esplorare nodi corrispondenti a punti già considerati in tempi diversi, eliminando di fatto il taglio dei cicli. In realtà, per evitare che l’agente resti intrappolato in una ricerca infinita, si dovrebbe controllare opportunamente questa possibilità, ad esempio inserendo una limitazione che impedisca all’agente di tornare su un punto già visitato per più di tre volte. Dato che il tempo aumenta all’infinito senza limiti, però, la complessità aumenterebbe comunque di molto.