Path-Finding-AI

Antonio Ambrosio

Ranieri Cristian

February 2024

Contents

1 Introduzione

Il seguente progetto nasce con l'idea di creare un intelligenza artificiale per dare vita a un nemico all'interno di un tile game. Nello specifico abbiamo deciso tramite AI di rendere capace il nemico di indviduare la tua posizione e raggiungerti attraverso il percorso ottimale.
All'interno della sezione riferimenti è presente il link alla repository del progetto [1].

2 Definizione del problema

2.1 Obbiettivi

Lo scopo del progetto è quello di realizzare un AI capace di scoprire la posizione del giocatore e raggiungerlo.
Molte delle risorse pre-esistenti che abbiamo individuato online, si concentravano solo ed esclusivamente sulla questione di raggiungere il player attraverso il percorso ottimale. Noi abbiamo deciso di non concentrarci solo sul seguente aspetto.
L'AI viene utilizzata all'interno dei videogiochi per produrre divertimento e far confrontare il player con un entità che reagisca con una logica alle sue azioni utilizzando solamente le informazioni di cui dovrebbe essere in possesso. In molti di questi progetti l'intelligenza artificiale non possiede un range massimo di visibilità rispetto al player, una direzione nella quale sta guardando e addirittura rileva il nemico attraverso i muri. Questo comportamento delle AI, più che creare divertimento può risultare poco intuitivo e addirittura frustrante. In altri casi invece, il nemico persa la posizione del player, non è capace di fare altro che tornare alla sua posizione iniziale. All'interno del seguente progetto abbiamo deciso di affrontare tutti quanti questi problemi.

2.2 Specifica PEAS

- **Performance**: La misura di prestazione adottata prevede di scegliere la mossa valida più appropriata rispetto alla situazione nel minore tempo possibile.
- **Enviroment**: L'ambiente in cui opera l'agente è costituito dalla mappa del livello nel quale si trova l'enemy e dalla posizione del player.
- **Actuators**: L'agente potrà ruotare la direzione nella quale guarda e muoversi in una delle 4 direzioni cardinali.
- **Sensors**: L'agente percepisce la parte a lui visibile della mappa tramite una rappresentazione virtuale di quest'ultima.

2.2.1 Caratteristiche dell'ambiente

- **Parzialmente osservabile**, in quanto il nemico ha una visione parziale della mappa.
- **Deterministico**, in quanto lo stato successivo dell'ambiente è completamente determinato dallo stato corrente e dall'azione dell'agente.
- **Sequenziale**, in quanto l'azione scelta non dipende solamente dalle percezioni correnti ma anche da quelle passate. In questo caso infatti, l'enemy decide cosa fare anche sulla base dell'ultima posizione registrata nella quale ha visto il player.

- **Statico**, in quanto l'ambiente rimane invariato mentre l'agente sta deliberando.
- **Discreto**, l'ambiente fornisce un numero limitato di percezioni e azioni possibili. L'agente infatti percepirà solamente lo stato di un numero finito di tile presenti nel suo campo visivo.
- **Multi agente**, all'interno dell'ambiente è presente quanto meno un enemy e un player. Il seguente ambiente è ritenuto competitivo in quanto l'obbiettivo dell'enemy è quello di raggiungere il player. Quello del player invece, l'esatto contrario.

2.3 Analisi del problema

Nella seguente sezione in particolare, ci riferiamo alla sola questione del pathfinding(individuare il percorso più rapido per raggiungere il player).

Formalizzazione del problema:

- **Stato iniziale** : Lo stato iniziale è definito attraverso un file che descrive la composizione in tile del livello. Ciascun tile possiede due carrateristiche. La prima:
  - percorribile
  - non percorribile

  e la seconda:
  - non occupato
  - occupato da un enemy
  - occupato da un player
- **Descrizione delle azioni possibili** : L'agente è in grado di:
  - ruotare la visuale nelle 4 direzioni
  - muoversi in una delle 4 direzioni
- **Modello di transizione** : Restituisce la mappa con le posizioni o le direzioni degli agenti modificate.
- **Test obbiettivo** : Il nemico raggiunge un tile adiacente a quello in cui si trova il player(adiacente nelle quattro direzioni cardinali).
- **Costo del cammino** : Il Costo di rotazione della visuale e di spostamento tra tile sono diversi. Tuttavia rimangono identici rispetto alle 4 direzioni cardinali.

3 Soluzione

3.1 Tecnologie utilizzate

Dato che parte del nostro focus, era anche quello di riuscire a realizzare un AI che potesse essere impiegata realmente in un videogioco, abbiamo affrontato seriamente il problema dell'efficienza.
L'efficienza non dipende però solamente dall'algoritmo e dalle euristiche scelte. Tutto può essere vano se non è presente una implementazione corretta.
Per riuscire a fare questo è stato necessario (oltre che selezionare l'algoritmo giusto e l'euristica migliore)ragionare sull'ordine degli accessi alla memoria, sul layout di quest'ultima e poter realizzare delle particolari rappresentazioni dei dati.
Per tutti quanti i motivi elencati, la scelta è ricaduta su C++.

3.2 Algoritmo scelto per il PathFinding

Nella seguente sezione ci concentreremo sullo sviscerare le scelte in merito al sistema capace di individuare il percorso più breve per raggiungere il player.

3.2.1 Razionale

La scelta è ricaduta su un algoritmo di ricerca informata. Un algoritmo di ricerca non informata infatti non avrebbe utilizzato le informazioni per navigare strategicamente lo spazio delle soluzioni, e avrebbe quindi dovuto valutare molte più possibilità.
Il costo dietro un organizzazione dell'esplorazione delle soluzioni è comunque molto minore rispetto al costo necessario per valutare tutte quante le altre soluzioni.
Tra gli algoritmi di ricerca informata, abbiamo poi scelto l'algoritmo A*, per il quale abbiamo valutato una serie di euristiche differenti.
La scelta è ricaduta sull'A*, poichè quest'ultimo nella nostra situazione è chiaramente completo e risulta essere ottimale sotto certi vincoli. L'ottimalità del percorso individuato rappresentava uno dei nostri design goals. In particolare la sua ottimalità dipende dal fatto che l'euristica sia ammissibile e consistente.
Abbiamo scartato tra gli algoritmi di ricerca informata Best-First-Greedy in quanto non ci garantiva nè l'ottimalità nè tanto meno la completezza. IDA* è stato scartato poichè non era tra le nostre necessità quella di limitare la memoria utilizzata e in molte situazioni sarebbe risultato inefficiente, causa le riesplorazioni necessarie. La Best-First ricorsiva è stata scartata, in quanto non saremmo stati capaci di fornire un implementazione che avrebbe manipolato poco lo stack delle chiamate. Modificare continuamente lo stack delle chiamate potrebbe rappresentare un collo di bottiglia per le prestazioni.

3.2.2 Rappresentazione del livello

Per effettuare una corretta rappresentazione del livello, abbiamo inanzitutto deciso di porre in due strutture differenti i due aspetti che caratterizzano un tile. All'interno di una struttura **tiles** è infatti presente una matrice che ci consente di mantenere per ciascun tile se quest'ultimo è percorribile o risulta essere un ostacolo. All'interno dell'altra struttura **occupied** è presente una matrice che ci consente di verificare se ciascun tile è libero, occupato da un player oppure occupato da un enemy.
Le due seguenti strutture non vengono modificate o ricreate per generare le possibili soluzioni per il pathfinding. Viene infatti utilizzata una struttura addizionale che rappresenta un grafo per riuscire a rappresentare le soluzioni e per effettuare l'esplorazione. In particolare quest'ultima è costiuita da una serie di nodi, che mantengono le seguenti informazioni:

- parent: il nodo da cui si proviene.
- cost: il costo per raggiungere il nodo.
- pos: la posizione del tile che il nodo rappresenta.

Questo insieme di nodi rappresentano i nodi presi in considerazione e i percorsi individuati fino a quel momento, la individueremo con il nome di **graph**. L'altra struttura che abbiamo considerato è una matrice che ci consente di andare a verificare se abbiamo già esplorato quel determinato nodo o meno. Gli assegneremo il nome di **visited**.

3.2.3 Funzione di valutazione

La seguente funzione di valutazione viene utilizzata per andare a valutare le possibili soluzioni per permettere ad un enemy di raggiungere il player. La funzione di valutazione è la seguente:

$$f(n) = g(n) + h(n)$$

Dove $g(n)$ rappresenta il costo del cammino fino al nodo n, in particolare si calcola come $g(n) = numero\_tile\_attraversati + 0.1 * numero\_rotazioni\_visuale$. $h(n)$ invece è una euristica a scelta tra:

- manhattan
- euclidean
- chebyshev

Altre euristiche rispettavano le condizioni per essere utilizzate all'interno di A*, come Octile e Hamming. Tuttavia abbiamo deciso di non utilizzarle.
Hamming si adattava veramente poco al problema che stavamo cercando di risolvere.

Octile poteva risultare interessante nel caso in cui il movimento non fosse limitato sulle direzioni cardinali. Nel caso in cui il movimento potesse avvenire anche in diagonale, octile ci avrebbe permesso di considerare due pesi diversi per il movimento diagonale e orizzontale/verticale.

3.2.4 Implementazione

All'interno della seguente sezione chiariremo come abbiamo implementato l'algoritmo ponendo l'accento su alcune scelte fatte per motivi di efficienza. Dato che il codice diviso in frammenti potrebbe essere difficile da seguire, può essere utile avere visione dell'intera implementazione [2].
Chiariamo inanzitutto alcune scelte riguardanti la rappresentazione dei dati. Le matrici **tiled** e **occupied**, nonostante siano delle strutture bidimensionali vengono implementate come una struttura monodimensionale. In questo modo la struttura dati risulta essere più cache friendly [3]. Lo stesso identico discorso vale per la matrice utilizzata per memorizzare i nodi visitati, **visited**.
Per quanto concerne **graph** quest'ultimo dovrebbe essere generato a ogni esecuzione dell'algoritmo. Tuttavia i nodi di cui è formato, sono istanziati una sola volta e vengono riciclati a ogni esecuzione seguendo il pattern Object Pool [4].
Facciamo inoltre chiarezza riguardo cosa viene restituito dal seguente sistema. Computando il percorso dal punto di start al punto di end per capire quale è il primo tile da visitare per seguire il percorso, sarebbe necessario invertire la lista puntata che rappresenta il percorso. Questa operazione di inversione, è una inutile inefficienza che può essere aggirata invertendo l'inizio e la fine. Abbiamo quindi deciso di sacrificare un pò di chiarezza nel codice per risparmiare cicli di computazione.
Vediamo ora l'effettiva implementazione dell'algoritmo. Dopo aver aggiunto il primo nodo da esplorare, ovvero quello di fine, cicliamo fino a che il nostro heap(la nostra frontiera) non sarà vuota. Una volta recuperato dal nostro heap il nodo con un valore di $f(n)$ minore di tutti, rimuoviamo tale nodo dalla heap.
Controlliamo se le coordinate del nodo estratto corrispondono alle coordinate di inizio e se è così, restituiamo il nodo, a partire dal quale riusciremo a recuperare il percorso.
Se così non è, poniamo il nodo a esplorato all'interno della struttura **visited** e procediamo ad esplorarlo.

```
1  //Until the heap is empty loop
2  while (!minHeap.empty())
3  {
4    //Retrieve node on top of heap and pop from heap that node
5    PathNode& currentNode = *minHeap.top();
6    minHeap.pop();
7    //If reached the start position, return path
8    if (currentNode.pos.x == start.x && currentNode.pos.y == start.y)
```

```
9       { return &currentNode; }
10     //Set to explored the current node
11     graph.visited[currentNode.pos.x + currentNode.pos.y * world->
    currentLevel.dim.x] = true;
```

Esplorare un nodo significa valutare tutti quanti i nodi a lui adiacenti e aggiungerli alla frontiera. A partire quindi dalla posizione del nodo estratto, andiamo a generare i 4 possibili tile adiacenti.
Non tutte e quattro le posizioni generate rappresentano tile validi oppure da esplorare. Ad esempio alcune posizioni potrebbero essere non valide come coordinate. Il tile potrebbe essere un tile non attraversabile.
Dopo aver appurato che il tile di cui abbiamo computato la posizione è valido, possiamo andare a ricavare il costo del passaggio a quest'ultimo. Difatti se il passaggio a questo tile avviene nella stessa direzione in cui è avvenuto lo spostamento precedente, il costo sarà quello della sola azione di movimento verso il tile(1). Nel caso in cui invece bisogna effettuare una rotazione della visuale prima di passare al prossimo tile, allora è necessario considerare anche questo costo(totale 1.1).

```
1  for (int i = 0; i < dx.size(); i++)
2  {
3    //Compute the position of the nextNode to evalue
4    nextNodePos.x = currentNode.pos.x + dx[i];
5    nextNodePos.y = currentNode.pos.y + dy[i];
6    //Skip if the nextNodePos is not a valid position for the level
7    if (! Level::isInLevel(currentLevel, nextNodePos.x, nextNodePos.y))
8      continue;
9    //Skip if the tile is a not walkable
10   if (currentLevel.tileMap.tiles[currentLevel.dim.x * nextNodePos.y +
      nextNodePos.x].logicType == LogicType::Wall)
11     continue;
12     //Assign correct cost on the base of direction
13     cost = 1;
14     if (currentNode.parent != nullptr
15 && Vector2i((nextNodePos - currentNode.pos) - (currentNode.pos -
16 currentNode.parent->pos)) != Vector2i({0, 0}))
17         cost = 1.1;
```

Dopo aver verificato che il nodo è valido, e aver assegnato un costo corretto all'azione considerata, dobbiamo verificare che il nodo in questione non è stato già inserito nella frontiera. Nel caso in cui il nodo è stato già inserito nella frontiera, non ci resta che andare a updatare il percorso per raggiungere tale nodo, qualora il costo in questo caso risulti migliore.

```
1  //Search a node with nextNodePos in heap
2  int found = minHeap.find(nextNodePos);
3
```

```
//If found a node with nextNodePos in the heap, update cost, estimation
//and father if found a shorter path
if (found != -1)
{
    PathNode nextNode;
    int index = found;
    nextNode.parent = &currentNode;
    nextNode.cost = currentNode.cost + cost;
    nextNode.estimation = (*estimation)(nextNodePos, start);
    nextNode.pos = nextNodePos;
    minHeap.updateKey(index, &nextNode);
}
```

Nel caso in cui il nodo non sia già presente nella frontiera, se non risulta già esplorato, il nodo viene aggiunto alla frontiera.

```
//If the node is not explored and is not in the heap, add to the heap
if (graph.visited[nextNodePos.x + nextNodePos.y * world->currentLevel.
   dim.x] == false && found == -1)
{
    PathNode* newNode = &graph.pathNodes[graph.nextNode];
    graph.nextNode++;
    newNode->pos = nextNodePos;
    newNode->cost = currentNode.cost + cost;
    newNode->estimation = (*estimation)(newNode->pos, start);
    newNode->parent = &currentNode;
    minHeap.insert(newNode);
}
```

Nel caso in cui, vengano terminati i nodi presenti all'interno dello heap, questo significa che non è possibile in nessun modo andare a raggiungere il tile in posizione end a partire dal tile in posizione start. Di conseguenza restituiamo il valore null.

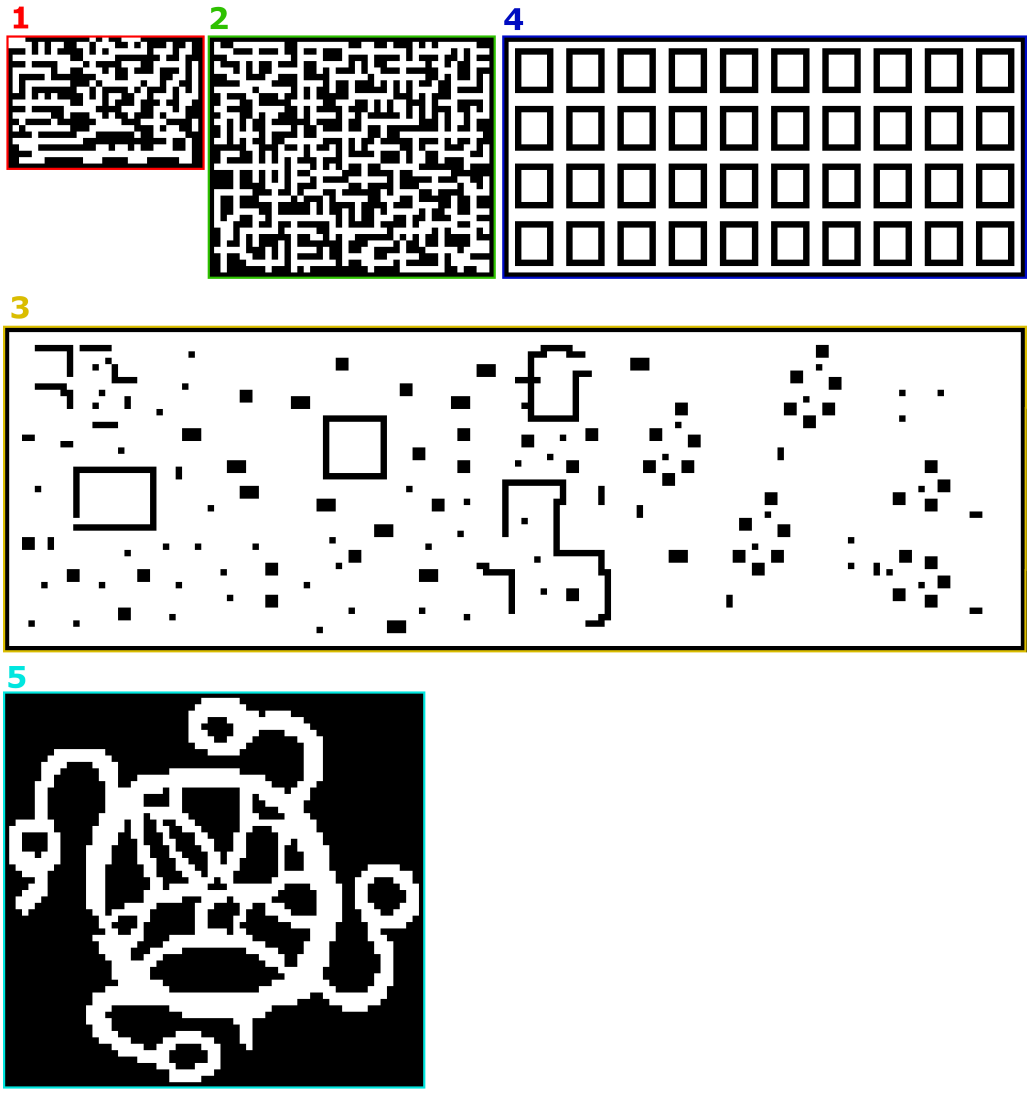
3.2.5 Valutazione dell'algoritmo e delle euristiche

Per riuscire a garantire che la nostra implementazione rispettasse i requisiti di efficienza, abbiamo deciso di accompagnare l'implementazione a quelli che sono dei test delle performance.
In particolare, all'interno di una singola mappa di gioco abbiamo realizzato molteplici terreni su cui mettere alla prova il nostro algoritmo.
Ci siamo basati su due metriche all'interno dei nostri test, elapsed time(tempo impiegato per eseguire il path finding) e nodi esplorati. E' da sottolineare che per confrontare le varie euristiche tra di loro, viene utilizzato il numero di nodi esplorati. L'elapsed time è stato calcolato solamente per avere un'idea di quanto questa implementazione possa effettivamente funzionare all'interno di un gioco. Di seguito i risultati per ciascuno dei test.

ID Test	Euristica	Elapsed Time	Explored Nodes
1	Manhattan	60μs	162
1	Euclidean	51μs	168
1	Chebyshev	56μs	171
2	Manhattan	76μs	335
2	Euclidean	89μs	420
2	Chebyshev	90μs	455
3	Manhattan	1169μs	1813
3	Euclidean	4545μs	5343
3	Chebyshev	5669μs	6177
4	Manhattan	49μs	139
4	Euclidean	237μs	1117
4	Chebyshev	264μs	1124
5	Manhattan	286μs	1058
5	Euclidean	254μs	1188
5	Chebyshev	908μs	1188
6	Manhattan	276μs	941
6	Euclidean	248μs	1040
6	Chebyshev	288μs	1122
//	Euristica	Average Execution Time	Average Explored Nodes
//	Manhattan	317μs	741
//	Euclidean	1138μs	1546
//	Chebyshev	1088μs	1706

ID Test	ID Mappa	Start Point	End Point
1	1	(7, 13)	(18, 0)
2	2	(33, 1)	(75, 36)
3	3	(156, 52)	(20, 71)
4	4	(79, 36)	(158, 1)
5	5	(3, 130)	(61, 125)
6	5	(29, 103)	(33, 155)

Come detto ciascun test è avvenuto su un terreno di gioco differente, proprio per valutare empiricamente il comportamento di ciascuna delle euristiche in ciascuna delle situazioni. Di seguito vi riportiamo una immagine riassuntiva di tutti quanti i terreni di gioco(numerati) che sono stati utilizzati per i test.



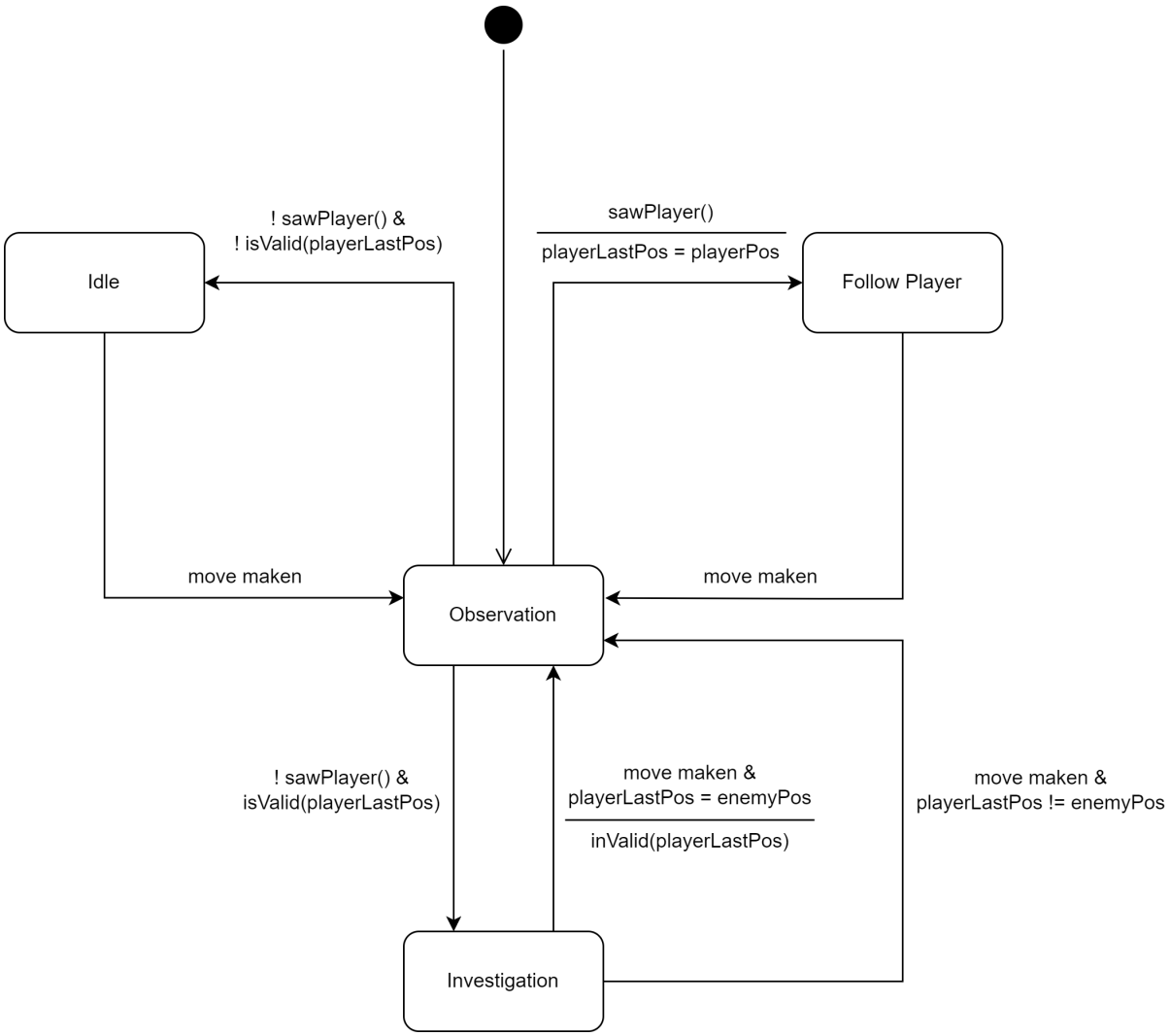
Nello specifico abbiamo utilizzato mappa 1 e mappa 2 per verificare come si comporta il nostro algoritmo all'interno di situazioni simili a labirinti. La mappa 3 è stata utilizzata per verificare come funziona l'algoritmo in una zona vasta e con una disposizione degli ostacoli casuale. La mappa 4 invece dovrebbe rappresentare una situazione simile a una città in un tilegame, gli ostacoli sono disposti regolarmente e sono di dimensioni molto simili. La mappa 5 invece, è stata utilizzata per testare come l'algoritmo riesce a muoversi all'interno di questi cunicoli non molto lineari.
Le conclusioni più banali che abbiamo tratto, è che in generale le euristiche tendono a restituire risultati molto simili. L'unico test in cui un euristica è stata sensibilmente più efficiente delle altre, è il test 3 riguardante la mappa 3. Questo è probabilmente dovuto al fatto che è la mappa più grande su cui abbiamo effettuato il test e c'era la maggiore distanza da coprire tra il punto di inizio e fine.
Quella di Manhattan è risultata particolarmente conveniente rispetto alle altre due euristiche, e dato che negli altri casi Manhattan ha ottenuto risultati simili alle altre, possiamo concludere che è quella più efficiente da utilizzare.
Ragionando su questo risultato abbiamo trovato una spiegazione . All'interno del nostro gioco i movimenti avvengono solamente in orizzontale e in verticale, e sommare quindi quella che è la distanza orizzontale e verticale tra due tile, risulta essere il modo migliore per rappresentare la distanza. A sostegno di questa nostra teoria c'è la cossidetta Geometria del Taxi [5] [6]. La Geometria del Taxi fornisce un ottima spiegazione del perchè Manhattan sia la scelta migliore quando i movimenti avvengono in verticale e orizzontale. Viene infatti utilizzata anche per misurare la distanza dal punto in cui una torre si trova al punto in cui una torre deve arrivare all'interno del gioco degli scacchi.
Anche se manhattan sembrerebbe la scelta migliore in assoluto, osservando quello che è il percorso risultante abbiamo notato una interessante differenza tra quelli computati attraverso la distanza euclidea e quella di manhattan. A parità di ottimo, i percorsi computati attraverso l'algoritmo di euclide risultano essere più naturali mentre invece quelli computati attraverso manhattan risultano essere più innaturali. Quelli computati tramite la distanza euclidea, si sviluppano in diagonale, mentre invece quelli realizzati tramite manhattan si sviluppano maggiormente sui due assi.

3.3 AI completa

Nella seguente sezione della documentazione analizzeremo il resto dell'AI e come abbiamo migliorato i sensori dell'agente per rendere l'enemy più interessante da affrontare.
Abbiamo deciso di realizzare in tre moduli differenti la struttura dell'AI, il sistema per il pathfinding e il sensore utilizzato per garantire il disaccoppiamento.

3.3.1 Vari stati dell'agente

Non sempre l'agente è capace di rilevare il player e quindi di computare il percorso più rapido per raggiungerlo. In tutti quanti gli altri casi l'obbiettivo è che l'enemy prenda comunque quella che è la scelta migliore per riuscire a raggiungere il player.
Il seguente diagramma degli stati ben descrive il funzionamento di tutta quanta l'AI da noi realizzata.



Lo stato centrale è quello di **Observation**. Quando si trova nello stato **observation** l'agente sta osservando l'ambiente circostante e sta deliberando, in base alle condizioni quale è l'azione migliore da eseguire. Da **observation** si passa allo stato **Idle**, nel caso

in cui l'agente non possa vedere il player e l'ultima posizione in cui ha visto il player non è più valida. Da **observation** si passa allo stato **Follow Player** nel caso in cui l'agente riesce a vedere il player. In questo caso viene aggiornata anche la lastPosPlayer. Da **observation** si passa a **Investigation** nel caso in cui l'agente non può vedere il player ma possiede una lastPosPlayer valida.
Nello stato di **Idle** l'enemy, esplora la zona intorno a sè in cerca del player seguendo un pattern per lui definito. Dallo stato di **Idle** si passa a quello di **observation** quando viene effettuata la prossima mossa definita nel pattern.
Nello stato di **Follow Player** l'enemy dopo aver computato lo shortest path segue quest'ultimo per tentare di raggiungere il player. Dallo stato di **Follow Player** si passa a quello di **observation** dopo aver eseguito un azione seguendo lo shortest path computato.
Nello stato di **Investigation** l'enemy segue lo shortest path alla lastPlayerPos. Dallo stato di **Investigation** si passa a quello di **Observation** in due possibili casi differenti. Nel primo caso quando segue una mossa ma non raggiunge la lastPlayerPos, in questo caso non ci sono post-condizioni. Il secondo caso in cui esegue una mossa per raggiungere la lastPlayerPos e arriva nella lastPlayerPos. In questo caso la post-condizione è di invalidare la lastPlayerPos.
Notare che all'interno del diagramma degli stati è presente solamente lo stato iniziale ma non quello finale. Questa imprecisione è data dal fatto che l'unico evento che può causare il passaggio allo stato finale, è la "morte" del enemy, da noi non considerata in quanto meccanica non presente allo stato attuale.

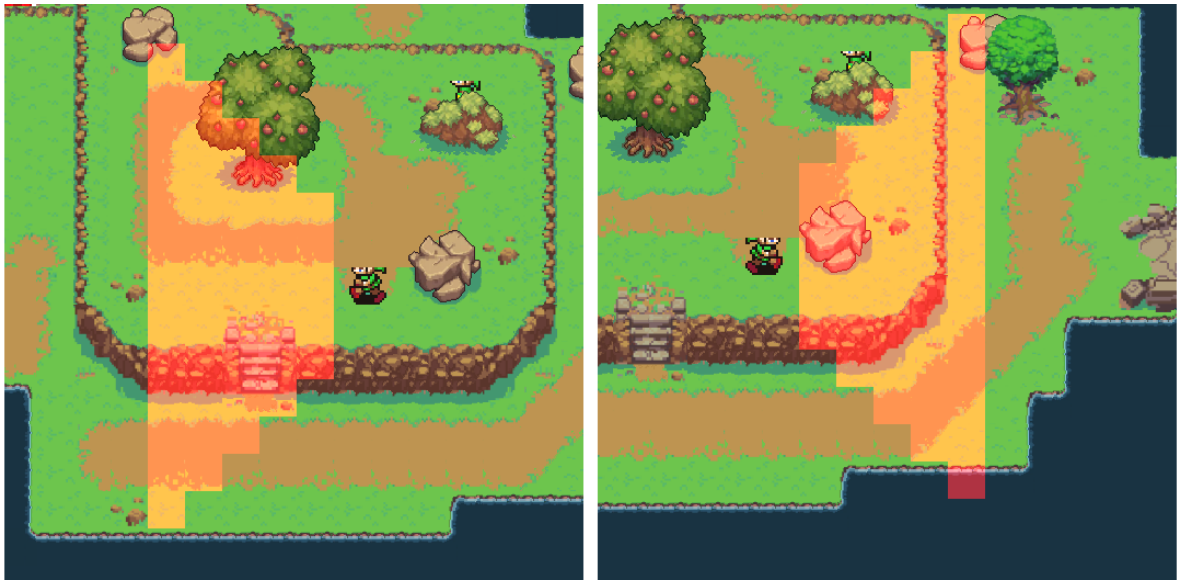
3.3.2 Razionale dietro gli stati

Spieghiamo ora perchè abbiamo deciso di includere determinati stati(e quindi determinate scelte) per il nostro enemy.
Idle serve ad aggiungere un pò di sfida nel caso in cui il player voglia evitare di farsi notare dall'enemy. Seguendo il pattern l'enemy può rendere la vita più difficile al player.
Follow Player è la più ovvia, e non necessita di ulteriori giustificazioni.
Infine abbiamo **Investigation**, quest'ultima risulta essere molto utile per rendere l'enemy più tenace nella ricerca. Interrotto infatti il contatto visivo con il player, senza questo stato l'enemy sarebbe fin troppo facile da aggirare. Per evitare che questo avvenga, abbiamo deciso di dare la possibilità all'enemy di fare la cosa più sensata, recarsi nell'ultima locazione dove ha visto il player.

3.3.3 Miglioramento dei sensori

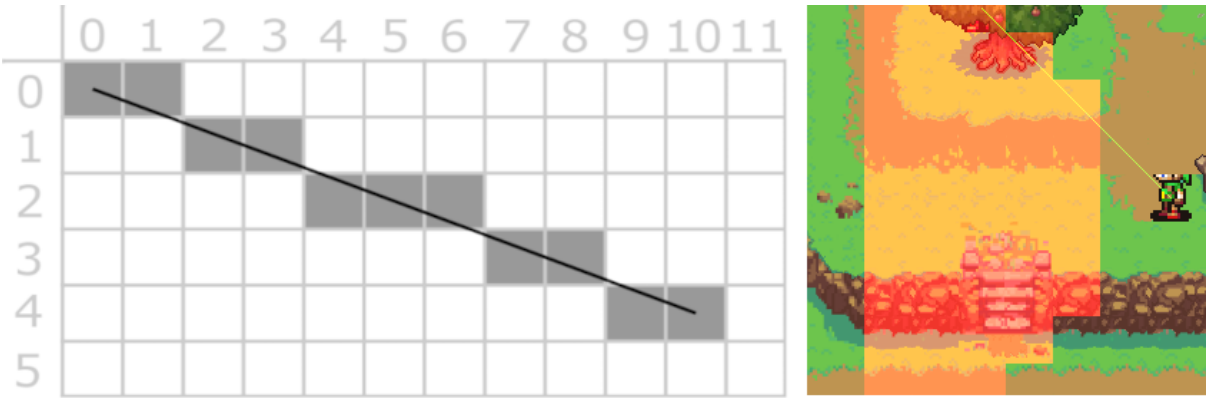
Avere degli enemies intelligenti e capaci di reagire in modo sensato, non dovrebbe significare avere dei nemici che utilizzano informazioni di cui non dovrebbero essere in possesso.

Ricordiamoci infatti che abbiamo detto precedentemente che l'ambiente è parzialmente osservabile. L'agente non dovrebbe quindi avere visione di tutta quanta la mappa.
Molte delle risorse che abbiamo individuato e progetti simili mancavano infatti di corretti sensori per questi agenti. Se a dividere me e il nemico ci sono dei muri, per quale motivo lui dovrebbe percepirmi e conoscere precisamente la mia posizione?
Aumentare la difficoltà in un gioco non deve renderlo un gioco impari(se non altro la bilancia non deve pendere dal lato del nemico).
Per garantire questo abbiamo deciso di equipaggiare l'agente intelligente con dei sensori più appropriati. L'enemy possederà un certo campo visivo, nel nostro specifico caso abbiamo approssimato l'area di un cono. I tile che rientrano nel cono, vengono percepiti, gli altri vengono ritenuti fuori dal campo visivo. Solo se il player si trova in uno dei tile percepibili allora tale informazione potrà essere utilizzata dall'agente.
Vorremmo inoltre fare in modo che i blocchi non navigabili(muri per capirci) blocchino il campo visivo.
Per rispettare i nostri obbiettivi abbiamo quindi dovuto realizzare una tecnica, che si chiama **Line of Sight** [7]. Questa tecnica normalmente è però riferita ad ambienti in cui il movimento non è basato a griglie. Abbiamo quindi dovuto adattare la seguente tecnica a questo contesto considerando ancora una volta i limiti imposti da un tilegame.
Nello specifico, il cono visivo(la zona visibile dall'agente) è stato approssimato a un triangolo/trapezio realizzato con i tile. Il triangolo/trapezio in questione risulta ruotato rispetto alla direzione nella quale sta guardando l'enemy.



Il Line of Sight, non prende in considerazione solamente la direzione nella quale sta guardando il nemico. Il **Line Of Sight** permette anche di nascondere quelli che sono i tile che si trovano dietro agli ostacoli(la cui visibilità dovrebbe essere interrotta).
Cerchiamo ora di adattare questa tecnica al contesto di un tilegame. Abbiamo ragionato sul fatto che per poter percepire il player in una posizione nel campo visivo, una linea

retta che passa per la posizione del tile in cui trova l'enemy e quello in cui si trova il player, dovrebbe intersecare solo tile che non sono degli ostacoli.
Per capire quali tile vengono attraversati da una retta, abbiamo paragonato il seguente problema a:*quali tile sarebbero utilizzati per disegnare questa retta*, paragonando i tile a dei pixel.



Per riuscire a calcolare quali sono questi tiles, abbiamo trovato due tecniche differenti, che si chiamano **Bresenham's line algorithm** [8] e **Digital Differential Analyzer** [9]. Ai fini del nostro progetto, non vi era una grande differenza tra le due. Abbiamo infine deciso di utilizzare la DDA.

4 Riferimenti

[1] Repository del progetto: PathFindingAI
[2] Implementazione PathFinding del seguente progetto: PathFindingSystem.cpp
[3] Data locality per la questione del cache friendly: Data Locality
[4]: Object Pool
[5]: Geometria del taxi
[6]: Geometria del taxi più approfondita
[7]: Line of sight
[8]: Breheman's Algorithm
[9]: Digital Differential Analyzer

4.1 Risorse utilizzate

Di seguito una lista delle risorse che abbiamo utilizzato ma che non è stato necessario citare.

- Algoritmo A*
- Interessante articolo su A*