Pathfinding su giochi grid-based

Lentisco Francesco
N86001092

Chapter 1

Introduzione

Un gioco grid-based (o tile-based) è un tipo di videogioco dove l'area di gioco consiste in una matrice di locazioni quadrate dette *tiles* che simulano una veduta dall'alto di una regione bidimensionale. L'insieme dei tile disponibili è chiamato *tileset*. In particolare, una mappa consiste in una griglia di larghezza e di altezza fissate ed ogni tile può essere traversabile o bloccato. L'insieme delle possibili locazioni traversabili è rappresentato mediante un grafo indiretto. I *tile* sono disposti in maniera adiacente l'uno dall'altro nella griglia.

Nel progetto proposto ci si è serviti di mappe *grid-based* generate in modo casuale a seconda della tipologia desiderata.

Ogni *tile* traversabile della mappa corrisponde ad un nodo nel grafo corrispondente. Un nodo, che corrisponde ad una locazione traversabile è connesso a tutti i nodi associati ai tile adiacenti e traversabili. I pesi degli archi che corrispondono a spostamenti ortogonali hanno costo unitario, gli archi tra due nodi collegati diagonalmente hanno un costo $\sqrt{2}$.

Per riferirsi a un nodo del grafo partendo dalle sue coordinate sulla grigla e viceversa si usano le usuali formule di conversioni: date le coordinate geometriche x e y di una locazione, il nodo corrispondente avrà come identificativo

$$nodo = y * \text{MAP\_WIDTH} + x \tag{1.1}$$

dove *MAP_WIDTH* si intende la larghezza della mappa. Per riottenere invece le coordinate geometriche di un nodo si utilizzano le seguenti formule:

$$x = nodo\%\text{MAP\_WIDTH} \tag{1.2}$$

$$y = nodo/\text{MAP\_WIDTH} \tag{1.3}$$

Considerando le tipiche mappe dei retro game sono state considerate tre diverse tipologie di generazione che andremo a discutere qui in seugito.

Chapter 2

Algoritmi di pathfinding

Il *path-planning* è un componente fondamentale della maggior parte dei videogiochi. Gli *agent* spesso si muovono nell'area di gioco. A volte questo movimento è predeterminato dagli sviluppatori del gioco, come ad esempio il percorso di pattugliamento di un area che una guardia deve seguire. I percorsi predeterminati sono semplici da implementare, ma per loro natura sono facilmente prevedibili. Agent più complessi non sanno in anticipo dove dovranno muoversi. Una unità in un gioco di strategia potrebbe ricevere in tempo reale l'ordine dal giocatore di muoversi in qualsiasi punto sulla mappa, oppure, in un gioco *tile-based* può succedere che un agent debba inseguire il giocatore nell'area di gioco. Per ognuna di queste situazioni, l'IA deve essere in grado di computare un percorso adeguato attraverso l'area di gioco per arrivare a destinazione, partendo dalla posizione attuale dell'agent. Inoltre vorremmo che il percorso trovato sia il più breve possibile.

La maggior parte dei giochi usa delle soluzioni di pathfinding basate sull'algoritmo chiamato A*. Nonostante sia molto semplice da implementare ed efficiente, A* non opera in genere sulla geometria della mappa di gioco. Esso richiede che tale mappa sia rappresentata in una particolare struttura dati: un grafo pesato e non-negativo. In questo capitolo verranno presentati gli algoritmi realizzati nel progetto, tra cui Dijkstra, A* e diverse sue varianti.

2.1 Dijkstra

Prima di entrare nel vivo degli algoritmi di pathfinding occorre fare alcune precisazioni su cosa si intende per *problema dei cammini minimi*. Nella teoria dei grafi, per shortest-path si intende il cammino minimo tra due

vertici, ossia quel percorso che collega due vertici dati e che minimizza la somma dei costi associati all'attraversamento di ciascun arco. Formalmente,

Sia $G = (V, E)$ un un grafo orientato con funzione di costo $\omega : E \to \mathbb{R}$ che associa ogni arco ad un valore nell'insieme dei reali. Sia $p = \langle v_0, v_1, ..., v_k \rangle$ un cammino che collega il vertice v_1 e al vertice v_k, allora il costo di p è la somma dei pesi degli archi che lo costituiscono:

$$\omega^*(p) = \sum_{i=1}^{k} \omega(v_{i-1}, v_i)$$

. Dato un cammino p da v_0 a v_k, p è *minimo* sse non esiste un altro cammino p' da v_0 a v_k tale che $w^*(p') < w^*(p)$.

Esistono dunque due varianti del problema: il problema del *cammino minimo* tra una coppia di vertici, e il *problema dei cammini minimi* da un vertice sorgente verso tutti gli altri vertici raggiungibili dalla sorgente. Chiaramente il primo è un sottocaso del secondo.

L'algoritmo di Dijkstra è un algoritmo che risolve il *problema dei cammini minimi* (o Shortest Paths, SP) in un grafo con o senza ordinamento e con pesi non negativi sugli archi.

Mentre nei videogiochi usualmente si computa il percorso minimo da un punto di partenza a un punto di arrivo, l'algoritmo di Dijkstra è realizzato in modo da trovare il percorso più breve da un punto di partenza verso tutti i restanti punti raggiungibili. La soluzione includerà ovviamente anche l'eventuale punto di arrivo, ma ciò comporterà in ogni caso uno spreco di risorse computazionali se siamo interessati a un solo punto di arrivo. Tuttavia può essere modificato in modo da generare solo il percorso nel quale si è interessati, ma sarà ancora inefficiente come algoritmo di pathfinding.

Dijkstra è un algoritmo iterativo. Ad ogni iterazione, considera un nodo nel grafo ed esamina i suoi archi uscenti (nella prima iterazione considera il nodo di partenza). Ci riferiremo per semplicità al nodo considerato ad ogni iterazione come "nodo corrente". Per ogni arco uscente dal nodo corrente, viene esaminato ogni suo nodo terminale v e si memorizza il costo totale del cammino totale percorso fino ad arrivare ad esso (*cost-so-far)* e l'arco dal quale si è arrivati ad esso in apposite strutture dati ($dist[v]$ e $prev[v]$ nello pseudocodice). Dopo la prima iterazione, il costo totale del cammino per il nodo terminale di ogni connessione del nodo corrente, è uguale alla somma del costo totale del nodo corrente e del peso della connessione verso il nodo terminale.

L'algoritmo tiene traccia dei nodi da visitare in una coda di priorità. La priorità viene stabilita sulla base del costo totale del cammino associato a

quel nodo. Nella prima iterazione la coda conterrà solo il nodo di partenza con costo totale uguale a 0. Nelle successive iterazioni l'algoritmo estrae dalla coda il nodo con il minor costo totale. Questo sarà processato come *nodo corrente*.

Ogni volta che viene esaminato un nodo terminale, l'algoritmo confronta il suo costo totale attuale (all'inizio tutti i nodi vengono inizializzati, assegnandoli un valore di costo di default pari a *infinito*) con quello appena calcolato. Se il costo totale appena calcolato è minore di quello precedentemente assegnato a quel nodo terminale, tale valore viene aggiornato. Ovviamente oltre al valore di costo totale viene aggiornato anche il suo predecessore, che diventerà l'arco che connette il nodo corrente a quello terminale in esame. Quando si verifica questa condizione significa che l'algoritmo ha trovato un percorso migliore verso quel nodo terminale.

L'implementazione canonica di Dijkstra termina quando la coda è vuota, ossia, quando tutti i nodi appartenenti al grafo sono stati visitati e processati. Tuttavia per risolvere un problema di pathfinding, l'algoritmo può terminare prima che tutti i nodi vengano processati, ossia, quando il nodo di arrivo è il nodo con la priorità più bassa nella in coda.

Una volta raggiunto il nodo di arrivo, viene estratto il cammino percorrendo a ritroso tutte le connessioni usate per arrivare a quel nodo fino a raggiungere il nodo di partenza.

Tenendo presente che la coda di priorità è implementata come uno *heap di Fibonacci* e che la complessità delle operazioni di **extractMin**() e **updatePriority**() sono rispettivamente $\mathcal{O}(log(n))$ e $\Theta(1)$, la complessità dell'algoritmo nel caso peggiore è di $\mathcal{O}(|E| + |V| \log |V|)$.

2.2 A*

La maggior parte dei sistemi di *pathfinding* odierni sono basati su questo algoritmo, data la sua efficienza, semplicità di implementazione e ampi margini di ottimizzazione. Diversamente dall'algoritmo di Dijkstra, A* è pensato per la ricerca del percorso minimo *point-to-point*.

Il funzionamento dell'algoritmo è molto simile a quello di Dijkstra. A differenza di quest'ultimo, che sceglie sempre prima il nodo con il minor *cost-so-far*, A* sceglierà il nodo candidato che *più probabilmente* porterà al percorso più breve. Per far ciò, A* utilizza delle funzioni *euristiche*. Maggiore sarà l'accuratezza della funzione euristica utilizzata, tanto più l'algoritmo sarà efficiente.

Algorithm 1: Dijkstra Shortest Path

```
1  Function DijkstraShortestPath(G, s)
       /* array per tenere traccia del costo totale del
          percorso fino a quel nodo                          */
2      dist[source] ⟵ 0;
       /* coda di priorità                                   */
3      Q ⟵ ∅;
4      foreach v ∈ Graph.vertexSet() do
5          if v ≠ source then
6              dist[v] ⟵ ∞;
7              prev[v] ⟵ ∅;
8          end
9          Q.add(v, dist[v]);
10     end
11     while !Q.isEmpty() do
12         u ⟵ Q.extractMin();
13         foreach neighbor v of u do
14             alt ⟵ dist[u] + lenght(u, v);
15             if alt <dist[v] then
16                 dist[v] ⟵ alt;
17                 prev[v] ⟵ u;
18                 Q.updatePriority(v, alt);
19             end
20         end
21     end
```

A* funziona iterativamente: in ogni iterazione considera un nodo del grafo ed esamina i suoi archi uscenti. Il nodo corrente viene scelto usando un criterio di selezione simile a quello di Dijkstra, ma con la significativa differenza dell'euristica.

Come Dijkstra, per ogni arco uscente dal nodo corrente, A* esamina il suo nodo terminale x e memorizza il costo totale del cammino percorso fino ad arrivare ad uno di essi (*cost-so-far)* e l'arco dal quale si è arrivati. Inoltre A* memorizza un valore in più, ovvero, una stima del costo totale $f(x)$ del percorso dal nodo di partenza al nodo di arrivo attraverso x. Questa stima è data dalla somma di due valori: il costo totale *reale* dal nodo sorgente fino al nodo x (*cost-so-far* al quale ci riferiremo come *g(x)*) e la distanza (euristica) dal nodo x fino al nodo di arrivo a cui ci riferiremo come *h(x)*.

Pertanto *f(x) = g(x) + h(x)*. Come vedremo, *f(x)* fornisce la chiave dell'ordinamento della coda di priorità dell'algoritmo.

A* tiene traccia dei nodi scoperti ma ancora da processare in una coda, a cui ci riferiremo come *Open*, e dei nodi già processati in una lista *Closed*. I nodi saranno inseriti nella coda *Open* appena saranno scoperti lungo gli archi uscenti dal nodo corrente. Essi verranno successivamente trasferiti nella lista *Closed* una volta che diventeranno essi stessi nodo corrente.

Diversamente da Dijkstra, viene estratto dalla coda *Open* il nodo con il minor valore *f(x)*. In tal modo si processeranno per primi i nodi più promettenti.

Potrebbe accadere che si arrivi ad esaminare un nodo appartenente alla coda *Open* o alla lista *Closed* e si debbano modificare i suoi valori di costo. In questo caso ricalcoleremo il valore di *g(x)* come al solito e se esso è minore di quello già memorizzato, allora verrà aggiornato.

Diversamente da Dijkstra, A* può trovare cammini migliori per nodi che sono già stati processati, e che quindi si trovano nella lista *Closed*. In particolare, nel caso in cui durante una ricerca si incontra un nodo x appartenente alla lista *Closed* si valuta se il valore *g(x)* è maggiore del costo del percorso appena scoperto vuol dire che abbiamo scoperto un percorso migliore, e dovremmo aggiornare il predecessore di x e il valore *g(x)*. Tuttavia quando un nodo viene messo nella lista *Closed*, vuol dire che tutti i suoi archi uscenti sono stati processati. Pertanto aggiornare i valori del nodo x non sarà sufficiente, giacchè la modifica deve essere propagata lungo tutte le sue connessioni uscenti.

Esiste un approccio molto semplice per ricalcolare e propagare i valori aggiornati di un nodo appartenente alla lista *Closed*, ossia estrarlo dalla lista *Closed* e inserirlo nuovamente nella coda *Open* con i suoi valori aggiornati. Esso stazionerà nella coda finchè non verrà estratto e processato nuovamente. In tal modo le sue connessioni potranno a loro volta aggiornate nuovamente.

In molte implementazioni dell'algoritmo, A* viene fatto terminare quando il nodo di arrivo è il nodo con la minima priorità nella coda *Open*. Tuttavia come abbiamo visto, un nodo con il minor costo potrebbe essere rivisitato in un secondo momento. Non possiamo più garantire pertanto che il nodo minimo nella coda *Open* sia quello che porti ad un cammino minimo. Di qui, la maggior parte delle implementazioni di A* possono produrre risultati non ottimali.

Altre implementazioni terminano non appena il nodo di arrivo viene visitato non aspettando che esso diventi il più piccolo nodo nella coda *Open*. In ogni caso si ammette che il risultato finale potrebbe essere non ottimale, pertanto sarà a discrezione dello sviluppatore scegliere quale approccio di

terminazione adottare.

Esattamente come per Dijkstra, si recupera il cammino compiuto percorrendo ricorsivamente tutte le connessioni accumulate dal nodo di arrivo fino a quello di partenza.

Generalmente la parte euristica dell'algoritmo viene implementata come una funzione. Di seguito riporteremo alcune delle più comuni funzioni euristiche. Siano u e v due nodi con coordinate rispettivamente di (x_1, y_1) e (x_2, y_2)

- Manhattan distance: $H(u,v) = |x_1 - x_2| + |y_1 - y_2|$
- Euclidean distance: $H(u,v) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
- Chebyshev distance: $H(u,v) = max\{|x_1 - x_2|, |y_1 - y_2|\}$
- Octile distance : $H(u,v) = \sqrt{2} * min\{|x_1 - x_2|, |y_1 - y_2|\} + ||x_1 - x_2| - |y_1 - y_2||$

Se la griglia permette movimenti in quattro direzioni (4-neighborhood), allora è opportuno scegliere la distanza di Manhattan, altrimenti se permette movimenti in otto direzioni (8-neighborhood), la *octile distance* è preferibile. Quest'ultima può essere considerata come una variante della distanza di Chebyshev, tenendo conto che il costo di uno spostamento diagonale è uguale a $\sqrt{2}$. Tutte le funzioni euristiche proposte sono consistenti. Da notare che una funzione euristica h() si definisce consistente (o monotona) sse per ogni nodo n del grafo e ogni successore n' di n, il costo stimato *h(n)* per raggiungere l'obiettivo è uguale, o inferiore, al peso dell'arco da n a n' sommato al costo stimato *h(n')*. Formalmente, dato un grafo $G = (V, E)$ e una funzione euristica *h()*, allora *h()* si dirà consistente solo se per ogni nodo $n \in V$ e ogni successore n' di n vale la seguente condizione:

$$h(n) \leq h(n') + c(n, n')$$

Quindi h() è consistente nel caso in cui soddisfa dal punto di vista geometrico la proprietà della *diseguaglianza triangolare*, la quale afferma che in un triangolo ogni singolo lato non può essere superiore alla somma degli altri due.

Una funzione euristica si dice ammissibile sse non sopravvaluta mai il costo minimo effettivo verso il nodo di arrivo. Una funzione euristica *consistente* è sempre anche *ammissibile* (non è sempre vero il contrario). Formalmente, $h()$ si definisce ammissibile sse $h(v) \leq h^*(v)$, dove $h^*()$ è una funzione euristica ideale che restituisce sempre il costo esatto del percorso ottimo per raggiungere il nodo di arrivo.

Algorithm 2: A* Shortest Path

```
1  Function AStarShortestPath(Graph, source, target)
2      g(source) ⟵ 0;
3      parent(source) ⟵ source;
4      Open ⟵ ∅;
5      Closed ⟵ ∅;
6      Open.Insert(source, g(source) + h(source));
7      while !Open.isEmpty() do
8          u ⟵ Open.extractMin();
9          if u = target then
10             return "path found"
11         end
12         Closed.add(u);
13         foreach neighbor v of u do
14             if v ∉ Closed then
15                 if v ∉ Open then
16                     g(v) ⟵ ∞;
17                     parent(v) ⟵ ∅;
18                 end
19                 UpdateVertex(u, v);
20             end
21         end
22     end
23     return "no path found"
24 Function UpdateVertex(u, v)
25     g_old ⟵ g(v);
26     ComputeCost(u, v);
27     if g(v)<g_old then
28         if v ∈ Open then
29             Open.Remove(v);
30         end
31         Open.Insert(v, g(v) + h(v));
32     end
33 Function ComputeCost(u, v)
34     if g(u) + c(u, v)<g(v) then
35         g(v) ⟵ g(u) + c(u, v);
36         parent(v) ⟵ u;
37     end
```

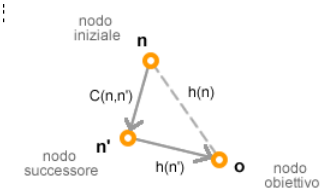


Figure 2.1: diseguaglianza triangolare

2.3 Theta*

Uno dei problemi centrali che concernono le Intelligenze Artificiali nei videogiochi è trovare percorsi minimi e allo stesso tempo che sembrino realistici. Il *path-planning* si divide generalmente i due parti: (*i*) discretizzazione, ossia, semplificare un ambiente continuo in forma di grafo e (*ii*) ricerca, attraverso la quale si vengono propagate le informazioni lungo il grafo per trovare un percorso da una locazione di partenza a un punto di arrivo. Per quanto riguarda la discretizzazione, è importante notare che esistono diversi approcci oltre alle regolari griglie bidimensionali, come ad esempio le mesh di navigazione (*nav-mesh*), grafi di visibilità e *waypoints*.

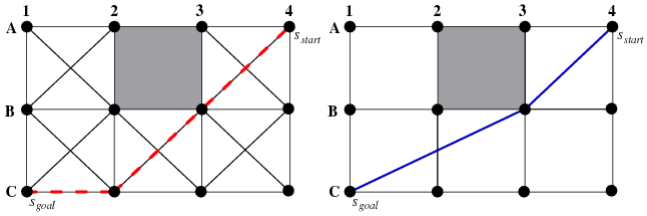


Figure 2.2: Square Grid: A* vs. Theta*

In ogni caso A* è quasi sempre il metodo di ricerca scelto a causa della sua semplicità e delle sue garanzie di trovare un percorso ottimo. Il problema con A* è che il percorso migliore sul grafo spesso non è equivalente al percorso migliore in un ambiente continuo. A* propaga le informazioni strettamente attraverso le connessioni del grafo e vincola i percorsi ad essere composti da tali connessioni. Nelle figure 2.2 e 2.3, un ambiente continuo è stato discretizzato rispettivamente in una griglia quadrata e in una navmesh. Il percorso minimo, sia nella griglia che nella mesh di navigazione (Figure 2.2 e 2.3, sinistra) è molto più lunga e non-realistica rispetto al percorso minimo nell'ambiente continuo (Figure 2.1 e 2.2, destra)

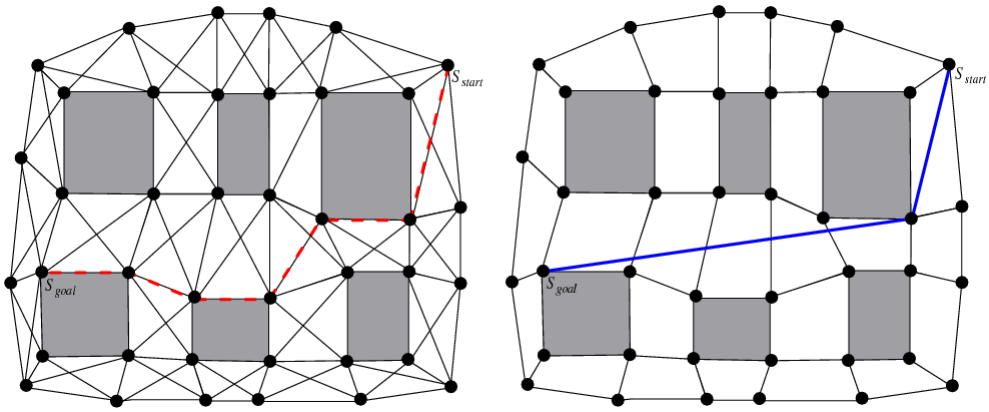


Figure 2.3: Navmesh: A* vs Theta*

La soluzione tipica a questo problema è di applicare una funzione di *path-smoothing* sui percorsi trovati dall'algoritmo. Tuttavia scegliere una buona tecnica di path-smoothing che ritorni un percorso che sembri realistico efficientemente può presentare alcune difficoltà. Uno dei principali motivi è che una ricerca di A* (con alcuni tipi di euristiche), garantisce di trovare solo una dei diversi cammini minimi, alcuni dei quali possono essere raffinati dalla funzione di *path-smothing* in modo più efficiente di altri. Ad esempio A* usato con un tipo di euristica *octile* è molto efficace sulle griglie che permettono movimenti diagonali, ma allo stesso tempo calcola percorsi non-realistici e molto difficili da raffinare perchè tutti i movimenti diagonali appaiono prima di tutti i movimenti lineari che compongono il percorso, come mostrato nella figura 2.4 dalla linea rossa discontinua.

L'algoritmo Theta* risolve queste problematiche. Essendo una variante di A*, similmente propaga le informazioni lungo gli archi del grafo ma senza vincolare i percorsi trovati ad essi (i percorsi con questa caratteristica soo detti "*any-angle*"). Similmente ad A*, Theta* è di facile implementazione ed efficiente (ha un runtime simile ad A*) ed inoltre calcola percorsi più "realistici", senza il bisogno di alcun processo postumo di *path-smoothing.*

Theta* combina due importanti proprietà che concernono il *path-planning*:

- **Grafi di Visibilità**: contengono il nodo di partenza, il nodo di arrivo, e gli spigoli di tutte le celle bloccate. Un nodo è connesso in linea retta ad un altro nodo se e solo se sono in linea di visibilità (*line-of-sight*) l'uno con l'altro, ossia, se non c'è alcuna cella bloccata lungo la linea retta che congiunge i due nodi. I cammini minimi sui grafi di visibilità

Algorithm 3: Line of Sight pt. 1

```
1  Function LineOfSight(u, v)
2      x_1 ← u.getX();
3      y_1 ← u.getY();
4      x_2 ← v.getX();
5      y_2 ← v.getY();
6      dx ← x_2 − x_1;
7      dy ← y_2 − y_1;
8      f ← 0;
9      signX ← 1;
10     signY ← 1;
11     x_offset ← 0;
12     y_offset ← 0;
13     if dy < 0 then
14         dy ← (−1) ∗ dy;
15         signY ← −1;
16         y_offset ← −1;
17     end
18     if dx < 0 then
19         dx ← (−1) ∗ dx;
20         signX ← −1;
21         x_offset ← −1;
22     end
23     if dx >= dy then
24         while x_1 ≠ x_2 do
25             f ← f + dy;
26             if f ≥ dx then
27                 if blocked(x_1 + x_offset, y_1 + y_offset) then
28                     return false
29                 end
30                 y_1 ← y_1 + signY;
31                 f ← f − dx;
32             end
33             if f ≠ 0 ∧ blocked(x_1 + x_offset, y_1 + y_offset) then
34                 return false
35             end
36             if dy = 0 ∧ blocked(x_1 + x_offset, y_1) ∧
                blocked(x_1 + x_offset, y_1 − 1) then
37                 return false
38             end
39             x_1 ← x_1 + signX;
40         end
41     end
```

Algorithm 4: Line of Sight pt.2

```
42
43      else
44          while y_1 ≠ y_2 do
45              f ← f + dx;
46              if f ≥ dy then
47                  if blocked(x_1 + x_offset, y_1 + y_offset) then
48                      return false
49                  end
50                  x_1 ← x_1 + signx;
51                  f ← f − dy;
52              end
53              if f ≠ 0 ∧ blocked(x_1 + x_offset, y_1 + y_offset) then
54                  return false
55              end
56              if dy = 0 ∧ blocked(x_1, y_1 + y_offset) ∧ blocked(x_1 − 1,
                 y_1 + y_offset) then
57                  return false
58              end
59              y_1 ← x_1 + signY;
60          end
61      end
62      return true
63  end
```

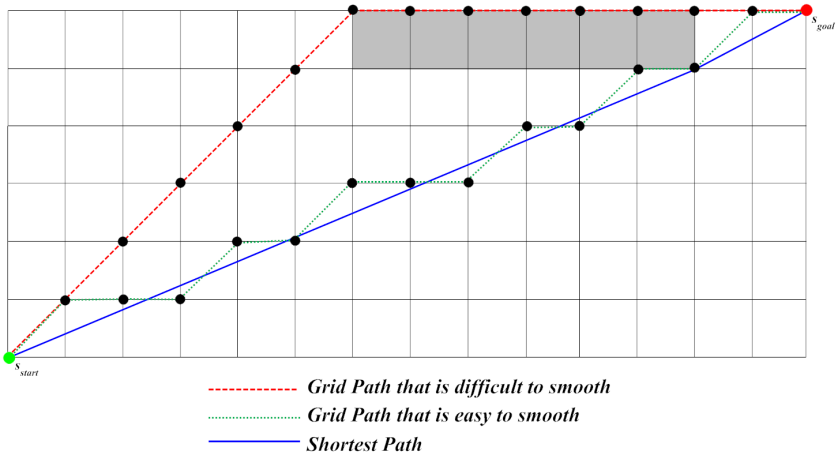


Figure 2.4: Percorso trovato da A* con differenti tecniche di path-smothing

sono tali anche in un ambiente continuo, ma purtroppo il *path-finding* su di essi è inefficiente perchè il numero di archi può essere quadratico rispetto il numero di celle.

- **Griglie**: il *path-finding* su di esse è più veloce rispetto i grafi di visibilità perchè il numero di archi è lineare sul numero di celle. Tuttavia i percorsi basate sugli archi delle griglie possono essere non-ottimali e dall'aspetto non-realistico.

La differenza principale tra Theta* e A* è che Theta* permette che il predecessore di un nodo sia un qualsiasi nodo, diversamente da A* che vincola il predecessore ad essere strettamente un nodo adiacente. La procedura principale e $UpdateVertex()$ sono del tutto simili ad A*, pertanto sono state omesse. Theta* è del tutto identico ad A* salvo per la procedure $ComputeCost()$, che aggiorna il valore $g()$ e il predecessore $parent()$ di un nodo non ancora espanso seguendo due possibili path:

- **Path 1**: per permettere percorsi *any-angle*, Theta* considera il percorso dal nodo di partenza al predecessore del nodo u ($parent(u)$ [= $g(parent(u))$] e dal predecessore del nodo u al nodo v in linea retta [= $c(parent(u), v)$] se e solo se i nodi v e $parent(u)$ sono in linea di visibilità (Linea 34). Il principio che c'è dietro questa considerazione è che il Path 1 non è più lungo del Path 2 per la **disuguaglianza triangolare** se il nodo v è in linea di visibilità con $parent(u)$.

Algorithm 5: Theta* Shortest Path

```
33 Function ComputeCost(u, v)
34     if LineOfSight(parent(u), v) then
35         if g(parent(u)) + c(parent(u), v)<g(v) then
36             parent(v) ⟵ parent(u);
37             g(v) ⟵ g(parent(u)) + c(parent(u), v) ;
38         end
39     end
40     else
41         if g(u) + c(u, v)<g(v) then
42             g(v) ⟵ g(u) + c(u, v);
43             parent(v) ⟵ u;
44         end
45     end
```

- **Path 2**: come visto in A*, Theta* considera il percorso dal nodo di partenza al nodo u [= *g(u)*] e dal nodo u in linea retta [= *c(u,v)*] ad un nodo adiacente v, risultante in un percorso di lunghezza g(u) + c(u,v) (Linea 41)

I controlli della linee di visibilità possono essere effettuati in modo efficiente con operazioni aritmetiche di soli interi su griglie quadrate. L'algoritmo usato esegue questi controlli con un metodo standard di *line-drawing* molto comune nelle applicazioni di *computer grafica*.

2.4 Bidirectional A*

L'algoritmo A*, applicato in modo unidirezionale, può effettuare una ricerca in due possibili direzioni opposte:

- **Forward**: A* effettua una ricerca dall'agent al target assegnando i loro nodi attuali rispettivamente al nodo di partenza e al nodo di arrivo. Ci si riferisce a questo approccio come **Forward A***.
- **Backward**: A* effettua una ricerca dal target all'agent assegnando i loro nodi attuali rispettivamente al nodo di partenza e al nodo di arrivo. Ci si riferisce a questo approccio come **Backward A*.**

L'idea del *bidirectional search* può dimezzare il tempo di ricerca di un algoritmo effettuando una ricerca in *forward* e una in *backward* simultane-

amente. Quando le due frontiere di ricerca si intersecano, l'algoritmo può ricostruire il percorso seguito che va dal nodo di partenza, passando per il "nodo frontiera", al nodo di arrivo. Tuttavia per garantire miglioramenti sostanziali della ricerca occorre che le due ricerche opposte si incontrino a metà strada.

Per dare l'idea generale dell'algoritmo proposto lo scomporremo nel seguente set di passi. Siano $Open_f$ e $Open_b$ rispettivamente le code di priorità delle due ricerche in *forward* e *backward* aventi rispettivamente $f_{f_{min}}$ e $f_{b_{min}}$ come valori f() minimi, $Closed_f$ e $Closed_b$ i loro set *Closed*, ed inoltre, sia α_{min} la lunghezza del percorso minimo dal nodo di partenza al nodo di arrivo (inizialmente inizializzato ad infinito):

1. Inizializza i nodi di *start* e di *goal*. Inserisci il nodo start e il nodo goal rispettivamente in $Open_f$ e in $Open_b$.

2. Decidi se effettuare una ricerca in forward (vai a step 3) o in backward (vai a step 4)

3. Espandi il fronte *forward* con *Forward-A** e vai allo step 5.

4. Espandi il fronte *backward* con *Backward-A** e vai allo step 5.

5. se si è scoperto un nodo n tale che $n \in Closed_f \cap Closed_b$, verrà eseguito il seguente assegnamento: $\alpha_{min} = min(\alpha_{min}, g_f(n) + g_b(n))$. Qui, se $\alpha_{min} \leq max(f_{f_{min}}, f_{b_{min}})$ allora l'algoritmo termina e il percorso con lunghezza α_{min} sarà restituito. Altrimenti torna allo step 2.

I punti critici di questo algoritmo si possono riassumere in (i) scelta tra le due ricerche e (ii) terminazione, di cui abbiamo già parlato nel passo 5. Per quanto riguarda invece il primo punto, si intende il criterio di scelta tra ricerca in *forward* o in *backward*. È importante notare che la strategia di scelta tra le due ricerche non influisce sulla correttezza dell'algoritmo quanto piuttosto sull'efficienza. Alcuni esempi di strategia di scelta possono essere:

- alternare una ricerca in *forward* e una in *backward* per ogni iterazione (procedura di Dantzig).

- siano $f_{f_{min}}$ e $f_{b_{min}}$ i valori f minimi nelle code $Open_f$ e $Open_b$, se $f_{f_{min}} < f_{b_{min}}$ allora espandi la frontiera in *forward*, altrimenti espandi la frontiera in *backward* (approccio di *Nicholson*).

- se $|Open_f| < |Open_b|$ allora espandi la frontiera in *forward*, altrimenti espandi la frontiera in *backward* (approccio *cardinality comparsion*).

L'ultima di queste strategie è stata riconosciuta in uno studio pubblicato nel 1969 come la più ragionevole[1] poichè la cardinalità dei set *Open* riflettono un indice di densità delle frontiere *forward* e *backward* e pertanto ci si è attenuti a questa nell'implementazione.

2.5 Adaptive A*

Nel contesto di un videgioco, gli *agent* devono spesso risolvere dei problemi di ricerca simili tra loro. *Adaptive A** è un recente algoritmo in grado di risolvere una serie di problemi di ricerca di natura simile tra loro più velocemente di A* perchè in grado di aggiornare i valori h utilizzando informazioni raccolte in ricerche precedenti. Questo algoritmo rende i valori $h()$ consistenti in valori $h()$ più accurati, conservando la loro consistenza. Ciò permette di calcolare percorsi minimi in un ambiente dove il costo di uno spostamento può essere incrementato visto che valori h consistenti rimangono tali dopo un incremento di costo. Tuttavia non è garantito che trovi il percorso minimo nel caso in cui ci si trovi in un ambiente dove il costo di uno spostamento (ossia, il peso di un arco) può diminuire, perchè i valori euristici consistenti non necessariamente rimangono tali dopo un decremento di costo. Pertanto i valori h devono essere aggiornati dopo un decremento di costo. Inoltre fino ad adesso si è considerato il problema della ricerca del percorso minimo dando per scontato che il target sia stazionario. Tuttavia ciò non è sempre vero in un contesto di videogiochi, dove spesso il target è a sua volta un *agent* e può dunque muoversi verso un altro target. Al fine di risolvere queste problematica, si è preferito implementare una versione di Adaptive A* che preveda che il target non sia stazionario: *Lazy Moving Target Adaptive A** ($LMTAA^*$[2]).

Adaptive A* è un algoritmo di ricerca *incrementale*, dove per incrementale si intende appunto la caratteristica di riusare informazioni raccolte durante le precedenti ricerche. Quando queste informazioni sono appunto i valori euristici calcolati dalle precedenti ricerche, allora si può definire l'algoritmo in questione come *heuristic learning incremental search.* Tuttavia AA* non può essere applicato in situazioni dove il target di ricerca

[1] Bi-directional and heuristic search in path problems, Ira Pohl, STANFORD LINEAR ACCELERATOR CENTER Stanford University Stanford, California, 94305

[2] Incremental Search-Based Path Planning for Moving Target Search, Xiaoxun Sun

può spostarsi. Questo perchè i valori euristici calcolati in ricerche precedenti allo spostamento del target risulterebbero incoerenti con la sua nuova posizione e si violerebbe la proprietà della *disuguaglianza triangolare*, che è una condizione necessaria affinchè una funzione euristica sia *consistente*.

Come già detto, *Lazy Moving Target Adaptive A** è una estensione di *AA** che risolve la problematica del target non stazionario, mantenendo consistenti i valori euristici durante le sue ricerche. Tuttavia, nè *AA** nè *MTAA** garantiscono di conservare la consistenza dei valori euristici h in caso di un decremento del costo di uno spostamento.

Si noti che, per semplcità nella descrizione dell'algoritmo, si è preferito modellare il problema in modo che l'agent e il target si muovano a turno di un passo alla volta invece che farli muoversi simultaneamente. La parte centrale dell'algoritmo è come al solito la procedura *ComputePath*(). Il cambiamento principale rispetto gli algoritmi visti fin'ora è che le ricerche possono essere effettuate in un ciclo iterativo per poter trovare ripetutamente nuovi cammini minimi dal nodo dell'agent al nodo del target. Dopo una ricerca, se viene trovato un percorso, l'agent si muove lungo di esso finchè il target non viene raggiunto, oppure nel caso in cui il target non si trovi più lungo il percorso, o ancora, finchè non avviene un cambiamento di costo per uno spostamento dell'agent. È stata inoltre introdotta una variabile *counter*, che indica quante ricerche sono state effettuate (inclusa la ricerca corrente). Si usa inoltre il *mapping search(s)* per indicare se i valori *g(s)* e *h(s)* di un nodo s sono stati inizializzati nella *counter*-esima ricerca. Inizialmente, per tutti i nodi s, si ha che *search(s) = 0*. Ciò significa che nessuno dei nodi è stato ancora inizializzato. In seguito l'algoritmo inizializza i nodi e inserisce il primo nella coda *Open*.

Viene dunque eseguita la procedura *InitializeState(s)* sui nodi s_{start} e s_{goal}, che vengono inizializzati prima di ogni ricerca s_{start} e s_{goal} (linee 51 e 52). La stessa procedura viene chiamata ogni volta che occorrono i valori *g(s)* e *h(s)* di un nodo s (linea 28). Durante la *counter*-esima ricerca (la ricerca corrente), nella procedura *InitializeState(s)*, se *h(s)* non è stata ancora inizializzato in nessuna ricerca precedente (*search(s) = 0*), la procedura inizializza h(s) con l'euristica fornita dall'utente (linea 6). Altrimenti, nel caso in cui $search(s) \neq counter$, ossia, se *h(s)* non è stata calcolata nella ricerca corrente, l'algoritmo controlla se il nodo s è stato espanso durante la *search(s)*-esima ricerca, dove è stata calcolato il valore *h(s)*: se $g(s) + h(s) < pathcost(search(s))$ (linea 9) allora il valore $f(s)$ $(= g(s) + h(s))$ è inferiore del valore $g(s_{goal})$ durante la *search(s)*-esima ricerca (linea 59).

Visto che il valore *g()* del nodo s_{goal} è sempre uguale al valore *f()* del nodo

Algorithm 6: Lazy Moving Target Adaptive A*

```
1  Function CalculateKey(s)
2      return g(s) + h(s);
3  Function InitializeState(s)
4      if search(s) = 0 then
5          g(s) ⟵ ∞;
6          h(s) ⟵ H(s, s_goal);
7      end
8      else if search(s) ≠ counter then
9          if g(s) + h(s) < pathcost(search(s)) then
10             h(s) ⟵ pathcost(search(s)) − g(s);
11         end
12         h(s) ⟵ h(s) − (deltah(counter) − deltah(search(s)));
13         h(s) ⟵ MAX(h(s), H(s, s_goal));
14         g(s) ⟵ ∞;
15     end
16     search(s) ⟵ counter;
17 Function UpdateState(s)
18     if s ∈ Open then
19         Open.DecreasePriority(s, CalculateKey(s));
20     end
21     else
22         Open.Insert(s, CalculateKey(s));
23     end
24 Function ComputePath()
25     while Open.Min() < CalculateKey(s_goal) do
26         u ⟵ Open.extractMin();
27         foreach neighbor v of u do
28             InitializeState(v);
29             if g(v) > g(u) + c(u, v) then
30                 g(v) ⟵ g(u) + c(u, v);
31                 parent(v) ⟵ u;
32                 UpdateState(v);
33             end
34         end
35     end
36     if Open = ∅ then
37         return false
38     end
39     return true
```

```
40
41  Function Main()
42      counter ⟵ 0;
43      s_start ⟵ current node of the agent;
44      s_goal ⟵ current node of the target;
45      deltah(1) ⟵ 0;
46      forall s ∈ G.vertexSet() do
47          search(s) ⟵ 0;
48      end
49  while s_start ≠ s_goal do
50      counter ⟵ counter + 1;
51      InitializeState(s_start);
52      InitializeState(s_goal);
53      g(s_start) ⟵ 0;
54      Open ⟵ ∅;
55      Open.Insert(s_start, CalculateKey(s_start));
56      if ComputePath() = false then
57          return false/* target out of reach                    */
58      end
59      pathcost(counter) ⟵ g(s_goal);
60      while target not caught AND action costs on path do not
          increase AND target on path from s_start to s_goal do
61          agent follows path from s_start to s_goal;
62      end
63      if agent caught target then
64          return true
65      end
66      s_start ⟵ current node of the agent;
67      s_newgoal ⟵ current node of the target;
68      if s_start ≠ s_newgoal then
69          InitializeState(s_newgoal);
70          if g(s_newgoal) + h(s_newgoal) < pathcost(counter) then
71              h(s_newgoal) ⟵ pathcost(counter) − g(s_newgoal);
72          end
73          deltah(counter + 1) ⟵ deltah(counter) + h(s_newgoal);
74          s_goal ⟵ s_newgoal;
75      end
76      else
77          deltah(counter + 1) ⟵ deltah(counter);
78      end
79      update the increased
80      action cost (if any)
81  end
82  return true
83 end
```

s_{goal} (solo con euristiche consistenti), il valore *f()* di un nodo s è inferiore del valore *f()* del nodo s_{goal} durante la ricerca *search(s)*. Pertanto, il nodo s deve essere stato espanso durante la ricerca *search(s)*. L'algoritmo pertanto aggiorna il valore *h(s)* a $pathcost(search(s)) - g(s)$ (linea 10). L'algoritmo corregge infine il valore *h(s)* secondo il nodo s_{goal} corrente nel caso il target si sia spostato dall'ultima volta che il nodo s è stato inizializzato dalla procedura *InitializeState(s)* (linee 12 e 13).

Spiegherò di seguito nel dettaglio come *Lazy MT-Adaptive A** corregge il valore h(s): dopo ogni ricerca, se il *target* si è mosso e quindi il nodo s_{goal} non è più lo stesso, la correzione per il nodo di arrivo della ricerca corrente diminuisce i valori h di tutti i nodi del valore *h()* del nodo di arrivo della ricerca corrente. Tale nuovo valore *h()* viene prima calcolato (linee 69-71) e in seguito aggiunto alla somma corrente di tutte le correzioni (linea 73). Nello specifico, il valore di *deltah(x)* durante la x-esima ricerca è uguale alla somma corrente di tutte le correzioni fino all'inizio della x-esima ricerca. Pertanto se *h(s)* è stato inizializzato in una ricerca precedente ma non in quella corrente ($search(s) \neq counter$), allora la procedura *InitializeState(s)* corregge il valore *h(s)* con la somma di tutte le correzioni delle ricerche tra le quali il nodo s è stato inizializzato e la ricerca corrente, che è uguale alla differenza tra i valori *deltah()* durante la ricerca corrente (*deltah(counter)*) e durante la ricerca *search(s)* (*deltah(search(s))*). In sintesi, il fattore di correzione è uguale a $deltah(counter) - deltah(search(s))$ (linea 12). In seguito viene scelto il massimo tra questo valore e il valore *h()* fornito dall'utente (linea 13), in linea con il nuovo nodo di arrivo rispetto la ricerca corrente.

In questo modo, Lazy MT-Adaptive A* aggiorna e corregge i valori *h()* di tutti i nodi solo nel caso in cui sono richiesti nelle future ricerche (approccio *lazy*), che è computazionalmente meno costoso rispetto ad aggiornare i valori *h()* di tutti i nodi espansi dopo ogni ricerca e di correggere i valori *h()* di tutti i nodi espansi dopo ogni ricerca dove il nodo di arrivo è cambiato (approccio *eager*).

Nei seguenti esempi si proverà a dare una idea del comportamento dell'algoritmo LMTAA*, evidenziando i vantaggi che esso offre, rispetto al tradizionale A*. Da tenere a mente che nei seguenti esempi è stata usata la distanza di Manhattan come funzione euristica e che sono stati ammessi solo spostamenti ortogonali nella griglia. In figura 2.5 vi è una legenda che indica come interpretare ogni cella dell'esempio. Inoltre le celle in grigio indicano che quel nodo è stato esplorato nella *counter*-esima ricerca. Inoltre va ricodato che LMTAA* è un algoritmo euristico incrementale. Ciò significa che dovremo simulare più iterazioni finchè le euristiche non diventino abbastanza informate e al fine di rendere visibili dei concreti vantaggi.

In figura 2.6 vi è una simulazione di una tipica situazione che potrebbe capitare su un terreno di gioco. La prima ricerca di LMTAA* non è diversa da una normale ricerca di A*.

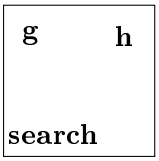


Figure 2.5: Legenda

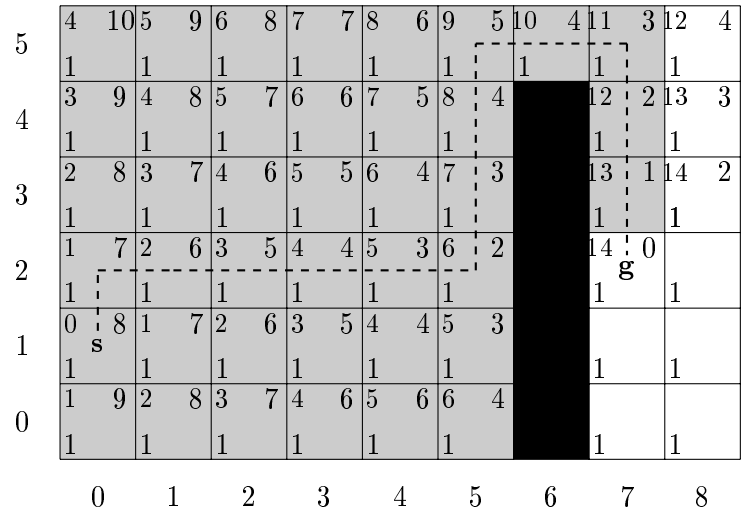


Figure 2.6: LMTAA*: termine iterazione 1
$counter = 1$
$pathcost(1) = 14$
$deltah(1) = 0$

A fine ricerca viene memorizzato il costo del percorso trovato nel mapping $pathcost(1)$. A questo punto l'agent sorgente si muove lungo il percorso da (0,1) a (1,2). L'agent target si muove da (7,2) a (8,3), uscendo fuori dal percorso che l'agent sorgente aveva calcolato. Quest'ultimo si vedrà dunque costretto a pianificare un nuovo percorso. La seconda iterazione è mostrata in figura 2.7.

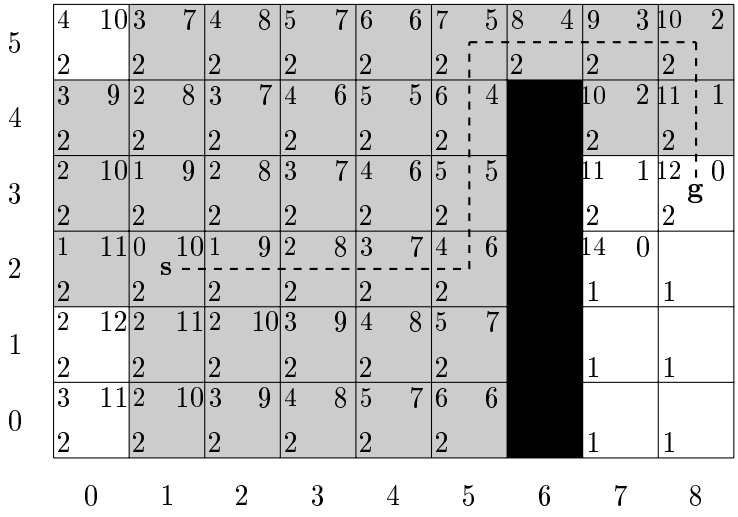


Figure 2.7: LMTAA*: termine iterazione 2
$counter = 2$
$pathcost(2) = 12$
$deltah(2) = 2$

Neanche qui sono visibili grandi vantaggi, a parte il fatto che le euristiche cominciano ad essere maggiormente accurate. Ciò vuol dire che ogni nodo sa con maggior precisione quanto dista dal nodo di goal. Più tale stima è accurata, minori saranno i nodi esplorati dalle ricerche future. A questo punto l'agent s si muove da (1,2) a (2,2). L'agent g si muove da (8,3) a (7,3). L'agent s dovrà pianificare un nuovo percorso per la terza volta. In figura 2.8 è mostrata la terza iterazione di LMTAA*.

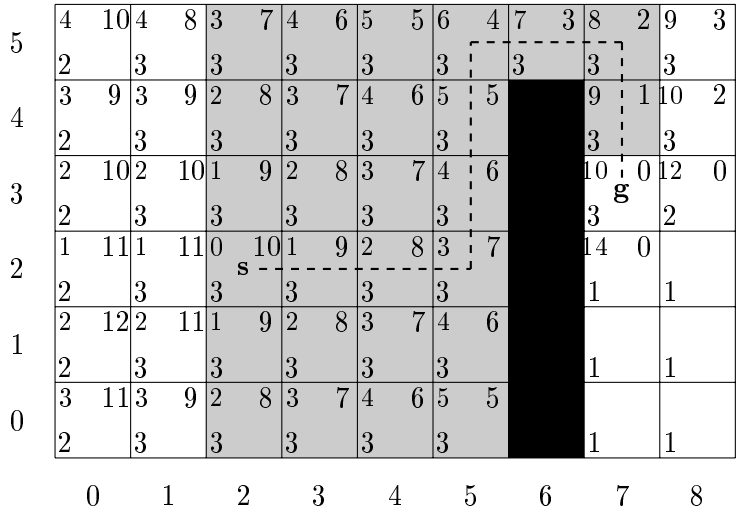


Figure 2.8: LMTAA*: termine iterazione 3
$counter = 3$
$pathcost(3) = 10$
$deltah(3) = 3$

A questo punto le euristiche sono abbastanza informate. Ogni nodo sa quasi esattamente quanto dista realmente dal nodo di arrivo. Solo in questa ultima iterazione, sono stati esplorati 27 nodi. Per rendere ben chiari i vantaggi di LMTAA* è stata mostrata, in figura 2.9, la stessa identica situazione di questa ultima iterazione con la differenza sostanziale che l'agent s si avvale di A* tradizionale e quindi di euristiche **non** informate.

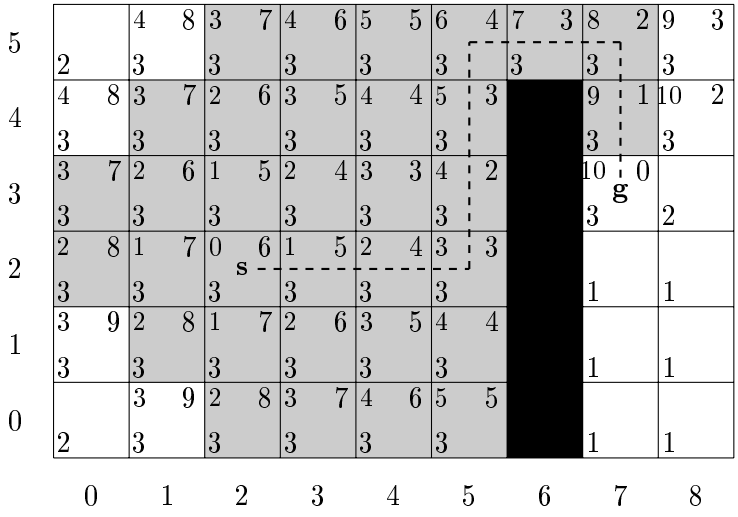


Figure 2.9: Plain A*: termine iterazione 3

A* esplorerebbe 33 nodi, contro i 27 di LMTAA*. A causa delle modeste dimensioni di tale esempio non si può apprezzare una grande differenza, ma si pensi a situazioni in cui un agent deve ripianificare continuamente un percorso verso un altro agent in una griglia con dimensioni 100 volte maggiori e in cui il runtime di una singola ricerca è un fattore critico per il funzionamento del sistema. In questo caso un algoritmo di ricerca incrementale è senza dubbio preferibile per via del suo vantaggio di migliorare le proprie prestazioni al crescere delle iterazioni.

2.6 Trailmax

Il problema del *moving target search* (noto anche come *pursuit-evasion* o in italiano *guardie e ladri*) è una famiglia di problemi già noti alla matematica e all'informatica, ed ha molte applicazioni in particolar modo ai videogiochi.

Spesso nei videogiochi il giocatore controlla un ladro che deve sfuggire ad un poliziotto controllato da una intelligenza artificiale. Tuttavia non è raro che lo stesso gioco venga capovolto, ad esempio in modo che il giocatore controlli il poliziotto e il ladro controllato da una intelligenza artificiale cerchi di sfuggirgli.

Visto che tutti gli algoritmi descritti fin'ora sono applicabili solo all'inseguimento, si è resa necessario lo sviluppo di un algoritmo che definisca una strategia di evasione per il target. In questo capitolo verrà dunque introdotto un

algoritmo che risponde a questa esigenza, chiamato TrailMax[3].

Per semplicità nella descrizione dell'algoritmo assumeremo che i due agent abbiano la stessa velocità. Tuttavia tutte le seguenti definizioni sono estendibili anche a differenti velcità. L'idea di fondo di questo algoritmo è che il target assume che l'inseguitore sappia in anticipo qual è il percorso migliore per raggiungerlo. Partendo da questa assunzione il target prova a massimizzare il tempo di cattura, scegliendo quindi un percorso che impiegherà all'inseguitore più tempo per raggiungere l'obiettivo.

Il percorso di evasione è calcolato espandendo simultaneamente i nodi intorno alla posizione dell'inseguitore e del target utilizzando una variante dell'algoritmo di Dijkstra. Vi saranno pertanto due code di priorità, una per l'inseguitore e una per il target. I nodi esplorati dal target sono confrontati con quelli esplorati dall'inseguitore al fine di controllare se l'inseguitore potrebbe già aver raggiunto quel nodo e catturato il target. Se è questo il caso, il nodo viene scartato. In caso contrario tale nodo appartiene alla copertura del target e pertanto espanso normalmente. Invece, i nodi presenti nella coda di priorità dell'inseguitore sono sempre esplorati normalmente. Nella figura 2.10 è mostrata una rappresentazione dell'espansione di TrailMax. L'area grigia contiene nodi che sono stati raggiunti per prima dal target, dichiarati come copertura del target, ma che non sono più stati espansi siccome sono stati catturati in seguito dall'inseguitore

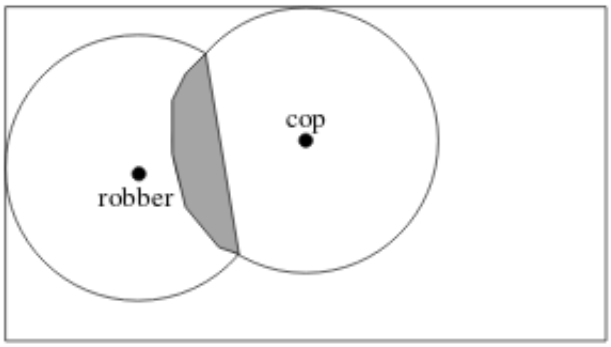
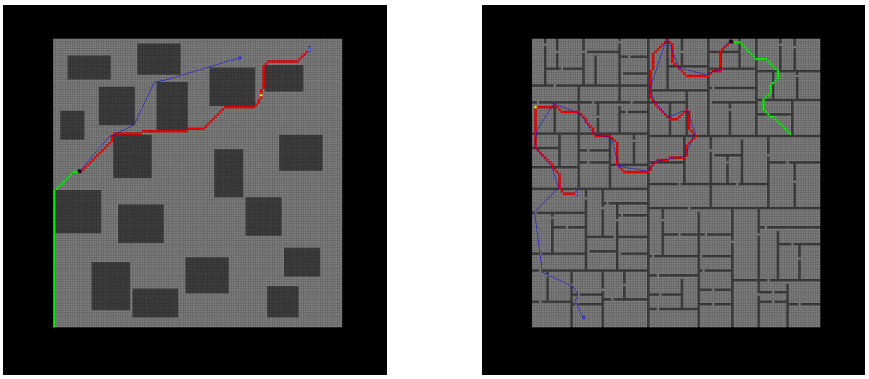


Figure 2.10: Visualizzazione della computazione di TrailMax.

L'algoritmo termina quando tutti i nodi appartenenti alla copertura

[3]Evaluating Strategies for Running from the Cops, Carsten Moldenhauer and Nathan R. Sturtevant Department of Computer Science University of Alberta Edmonton, AB, Canada T6G 2E8



(a) TrailMax su mappa outdoor (b) TrailMax su mappa indoor

Figure 2.11: Le immagini mostrano il percorso calcolato da TrailMax (verde) che massimizza la distanza dall'agent inseguitore (Rosso)

del target sono espansi anche dall'inseguitore. L'ultimo nodo esplorato dall'inseguitore sarà il nodo che il target dovrà raggiungere. Il percorso viene generato percorrendo a ritroso tutte le connessioni usate per arrivare a tale nodo, fino a raggiungere il nodo di partenza.

Chapter 3

Design ed Implementazione

Il class diagram dell'intera implementazione del progetto è mostrato nella figura 3.1. In questa sezione saranno oltre descritte le classi e i loro dettagli implementativi.

3.1 Game

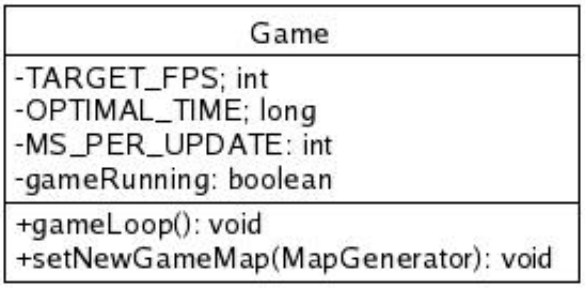


Figure 3.2: Classe Game

La classe *Game* mantiene le informazioni sullo stato del gioco, i riferimenti del giocatore e del relativo *controller*, delle entità instanziate e della mappa corrente. Inoltre implementa il *loop* principale all'interno del quale avviene la logica del gioco e il *rendering* degli elementi in gioco.

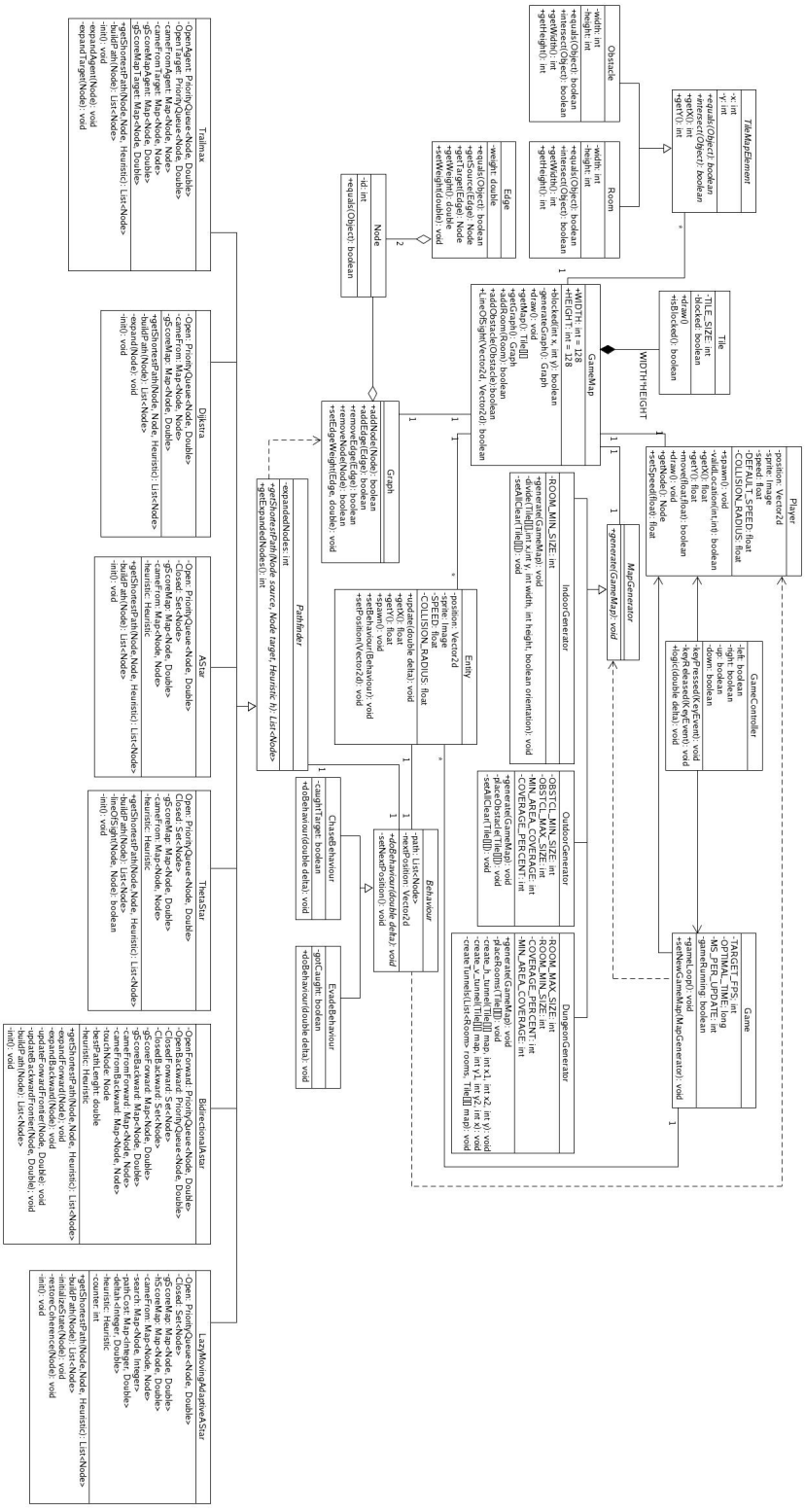


Figure 3.1: Class Diagram

3.2 Player

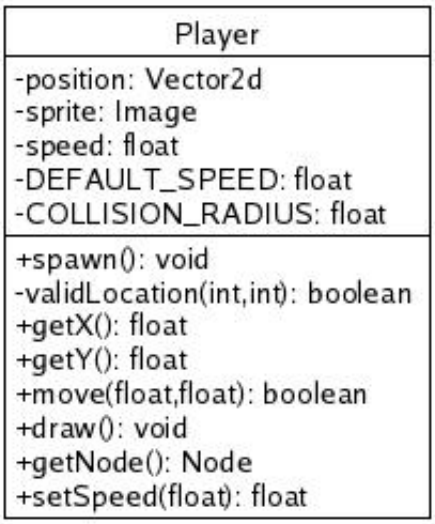


Figure 3.3: Classe Player

La classe *Player* mantiene le informazioni sullo stato del giocatore, la sua posizione sul terreno di gioco, velocità di spostamento e immagine da renderizzare.

3.3 Game Controller

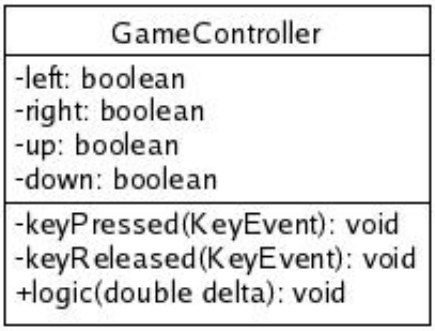


Figure 3.4: Classe GameController

La classe *GameController* implementa l'interfaccia *KeyListener* e gestisce gli input da tastiera, aggiornando in tal modo la posizione del giocatore, del quale mantiene un riferimento, similmente al *design pattern* MVC.

3.4 Tile

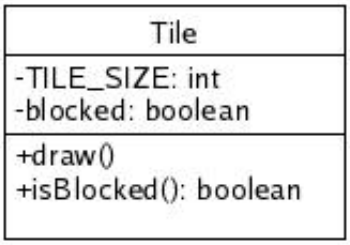


Figure 3.5: Classe Tile

La classe Tile rappresenta la più piccola unità che compone una mappa di gioco. Questa classe mantiene una variabile di stato che indica se il tile è bloccato e una costante statica che indica la grandezza di ciascun tile.

3.5 GameMap

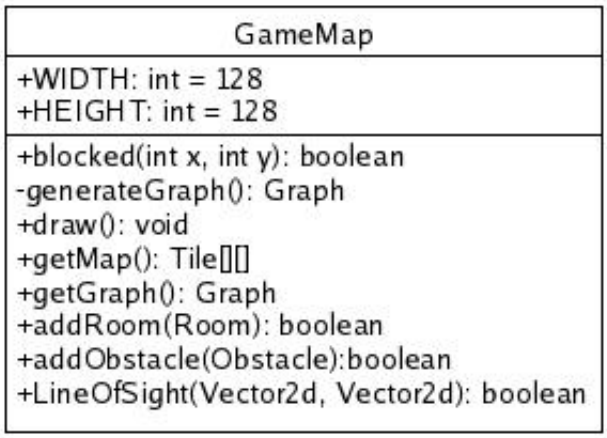


Figure 3.6: Classe GameMap

La classe *GameMap* si occupa di definire la geometria della mappa di gioco, mantenendo un riferimento di un array bidimensionale di oggetti Tile. Mantiene inoltre i riferimenti di eventuali stanze o ostacoli presenti sulla mappa. La geometria della mappa viene definita attraverso un oggetto che estende l'interfaccia *MapGenerator*, di cui viene mantenuto il riferimento all'interno della classe GameMap, e facendo uso del design pattern *Strategy* viene scelta la tipologia di mappa desiderata. Si occupa infine di generare il grafo associato a quella mappa basandosi sulla sua geometria.

3.6 TileMapElement

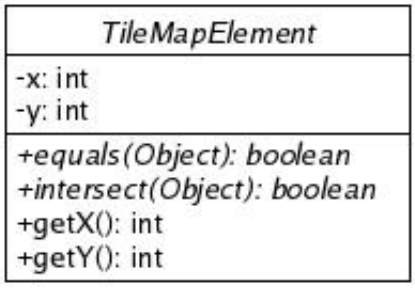


Figure 3.7: Classe TileMapElement

Classe astratta che definisce un elemento generico sulla mappa di gioco. Gli attributi x e y ne determinano il posizionamento sull'area di gioco.

3.7 Room

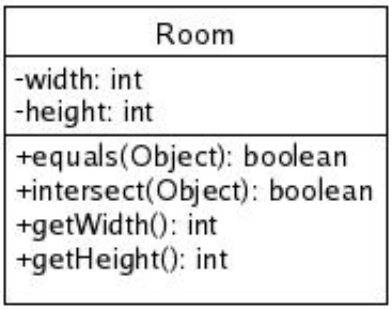


Figure 3.8: Classe Room

La classe *Room* estende la classe TileMapElement di cui sopra, ed è dotata di due attributi aggiuntivi che ne determinano l'altezza e la larghezza. In questo caso gli attributi x e y della superclasse TileMapElement si riconducono al Tile dell'angolo in basso a sinistra del rettangolo che la classe definisce. I Tile che appartengono al rettangolo definito dalla classe Room

sono tutti traversabili. La classe è inoltre dotata di un metodo *intersect*(), il quale restituisce true sse si verifica un *overlapping* con un altro oggetto Room in ingresso.

3.8 Obstacle

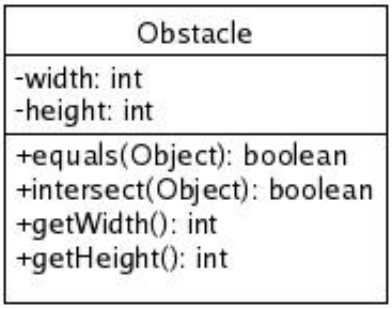


Figure 3.9: Classe Obstacle

Similmente alla classe Room, la classe *Obstacle* definisce un rettangolo di Tile ma in questo caso *non* traversabili.

3.9 MapGenerator

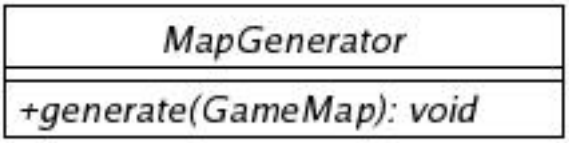


Figure 3.10: Classe MapGenerator

Classe astratta che definisce un generico generatore di mappe.

3.10 IndoorMapGenerator

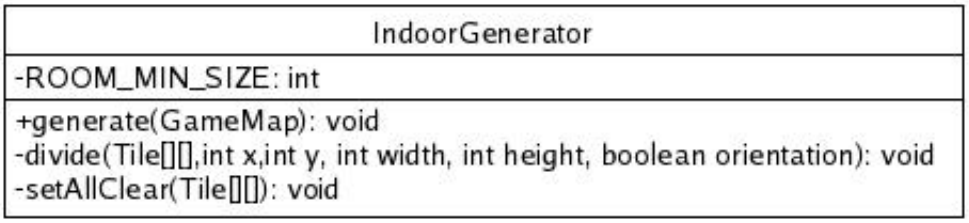


Figure 3.11: Classe IndoorMapGenerator

La classe *IndoorMapGenerator* estende la classe astratta *MapGenerator*. La costruzione della mappa *indoor* avviene nel metodo *generate()*, che fa a sua volta *overriding* sul metodo astratto della superclasse. L'algoritmo per la generazione di mappe indoor sarà descritto dettagliatamente in una sezione successiva.

3.11 OutdoorMapGenerator

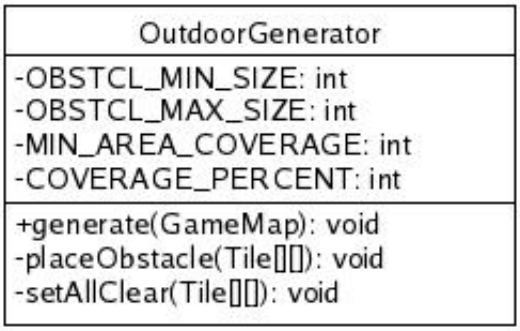


Figure 3.12: Classe OutdoorMapGenerator

La classe *OutdoorMapGenerator* estende la classe astratta *MapGenerator*. La costruzione della mappa *outdoor* avviene nel metodo *generate()*, che fa a sua volta *overriding* sul metodo astratto della superclasse. L'algoritmo

per la generazione di mappe outdoor sarà descritto dettagliatamente in una sezione successiva.

3.12 DungeonMapGenerator

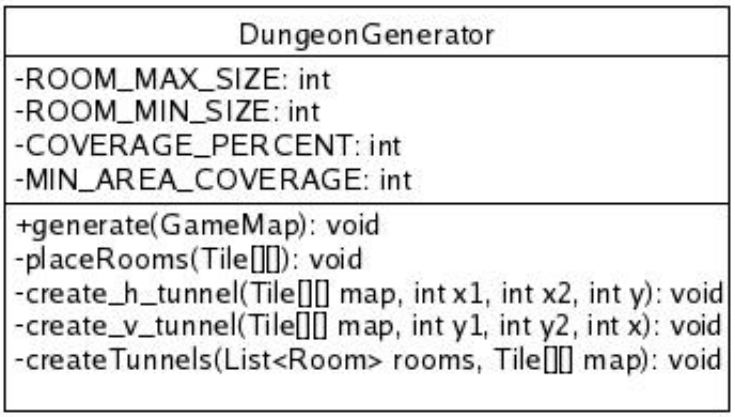


Figure 3.13: Classe DungeonMapGenerator

La classe *DungeonMapGenerator* estende la classe astratta *MapGenerator*. La costruzione della mappa *dungeon* avviene nel metodo *generate()*, che fa a sua volta *overriding* sul metodo astratto della superclasse. L'algoritmo per la generazione di mappe dungeon sarà descritto dettagliatamente in una sezione successiva.

3.13 Entity

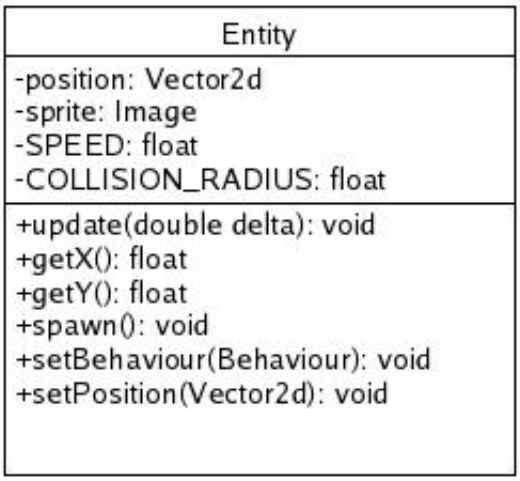


Figure 3.14: Classe Entity

La classe *Entity* definisce ogni agent all'interno dell'intero sistema di gioco, rendendola una delle classi più importanti dell'intero sistema. Essa tuttavia rappresenta solamente il *modello* di una entità. Infatti le funzionalità di cervello pensante, e quindi di *controller*, di ciascuna entità sono delegate ad un oggetto che estende la classe astratta *Behaviour* di cui si mantiene il riferimento all'interno della classe in esame. Attraverso il design pattern *Strategy* sulla classe astratta Behaviour è possibile definire molteplici comportamenti per un Entity che esamineremo più dettagliatamente in una sezione successiva. Attraverso il metodo *update*(), che riceve in ingresso la grandezza di un lasso di tempo, viene calcolato lo spostamento di un Entity.

3.14 Behaviour

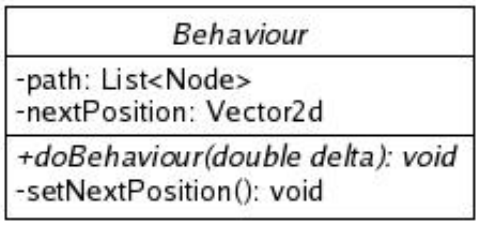


Figure 3.15: Classe Behaviour

Classe astratta che definisce un comportamento generico per un Entity. Le sue sottoclassi rappresentano il cervello di un Entity e si occupano di deciderne gli spostamenti, fungendo quindi da controller. Il comportamento di un Entity sarà dunque definito in una sua sottoclasse facendo *overriding* sul metodo *doBehaviour*(). La classe mantiene inoltre un riferimento ad un oggetto *Pathfinder*, che potrà variare a seconda del tipo effettivo della classe Behaviour.

3.15 ChaseBehaviour

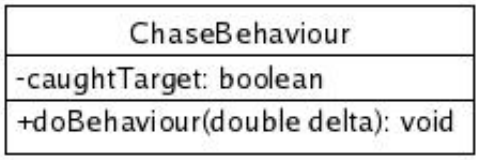


Figure 3.16: Classe ChaseBehaviour

La classe *ChaseBehaviour* estende la classe Behaviour e definisce un comportamento di inseguimento di un Entity rispetto un Player oppure un altro Entity. Utilizza di default come pathfinder una istanza di *LazyMovingAdaptiveAStar*. Mantiene inoltre una *flag* di stato che indica se l'Entity ha raggiunto l'obiettivo ed in tal caso smette di muovere l'Entity.

3.16 FleeBehaviour

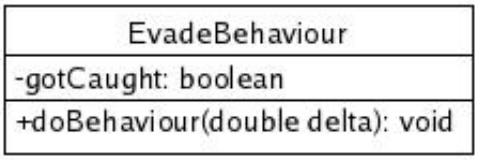


Figure 3.17: Classe EvadeBehaviour

La classe *FleeBehaviour* estende la classe Behaviour e definisce un comportamento di evasione di un Entity rispetto un Player oppure un altro Entity. Utilizza di default come pathfinder una istanza di *Trailmax*. Mantiene inoltre una flag di stato che indica se l'Entity è stato raggiunto dal giocatore o dall'Entity dal quale si sta scappando. In tal caso smette di muovere l'Entity.

3.17 Pathfinder

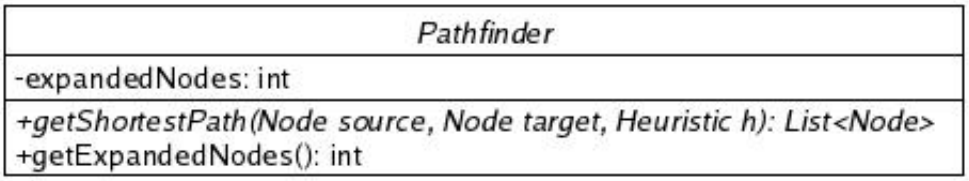


Figure 3.18: Classe Pathfinder

Classe astratta che definisce un pathfinder generico. Le sue sottoclassi che implementano un algoritmo di pathfinding dovranno fare overriding sul metodo astratto *getShortestPath*() il quale ritorna una lista di nodi che compongono il percorso trovato.

3.18 Dijkstra

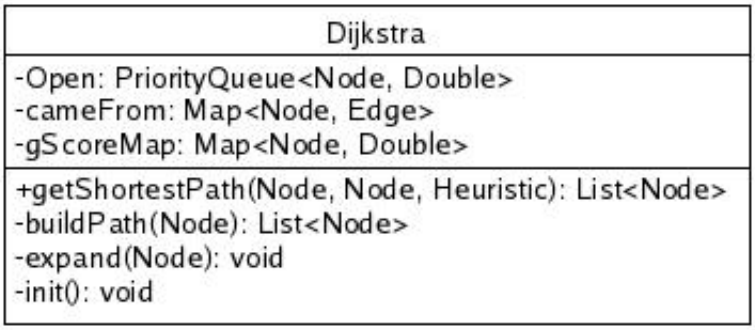


Figure 3.19: Classe Dijkstra

La classe *Dijkstra* estende la classe astratta *Pathfinder* ed implementa una versione dell'omonimo algoritmo di cui ne abbiamo già ampiamente discusso nella sezione 2.1.

3.19 AStar

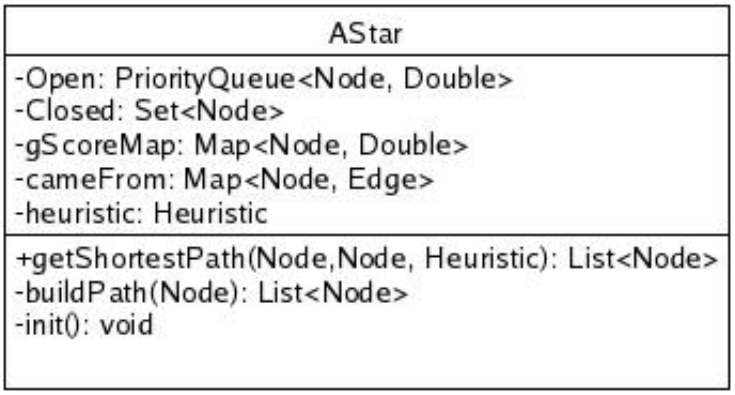


Figure 3.20: Classe AStar

La classe *AStar* estende la classe astratta Pathfinder ed implementa una versione dell'omonimo algoritmo di cui ne abbiamo già ampiamente discusso nella sezione 2.2

3.20 ThetaStar

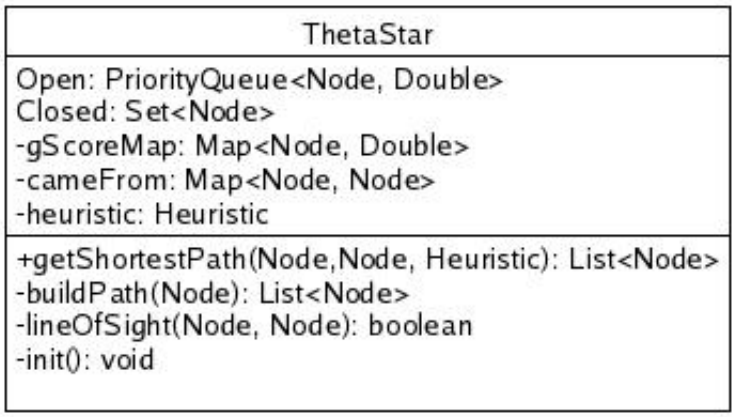


Figure 3.21: Classe ThetaStar

La classe *ThetaStar* estende la classe astratta Pathfinder ed implementa una versione dell'omonimo algoritmo di cui ne abbiamo già ampiamente discusso nella sezione 2.3

3.21 BidirectionalAStar

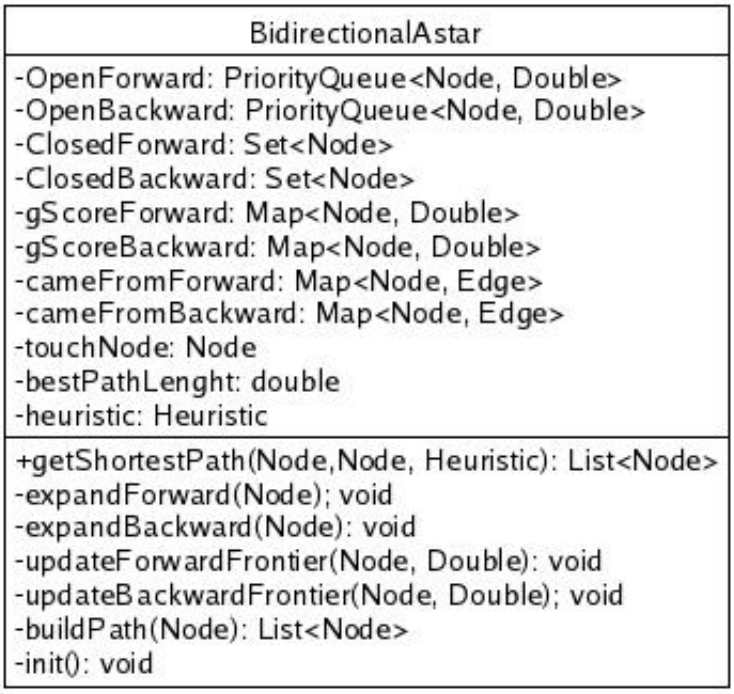


Figure 3.22: Classe BidirectionalAStar

La classe *BidirectionalAStar* estende la classe astratta Pathfinder ed implementa una versione dell'omonimo algoritmo di cui ne abbiamo già ampiamente discusso nella sezione 2.4

3.22 LazyMovingTargetAdaptiveAStar

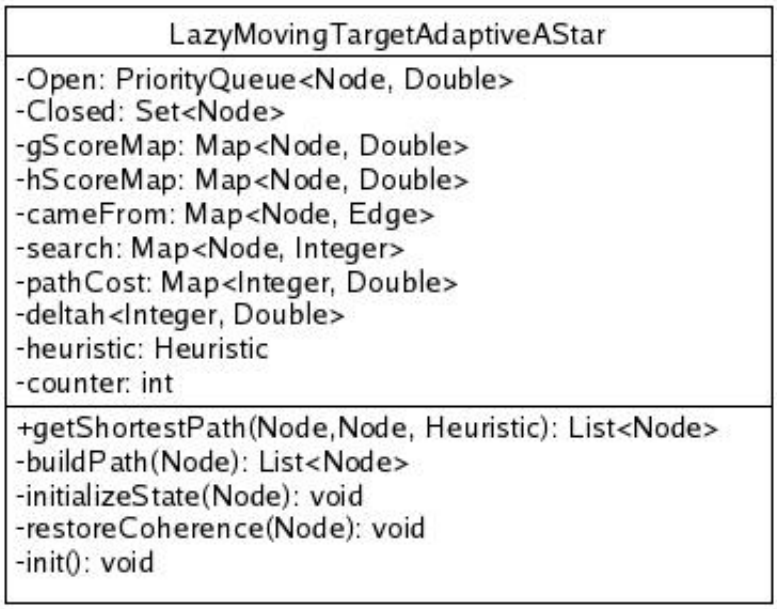


Figure 3.23: Classe LazyMovingTargetAdaptiveAStar

La classe *LazyMovingAdaptiveAStar* estende la classe astratta Pathfinder ed implementa una versione dell'omonimo algoritmo di cui ne abbiamo già ampiamente discusso nella sezione 2.5

3.23 Trailmax

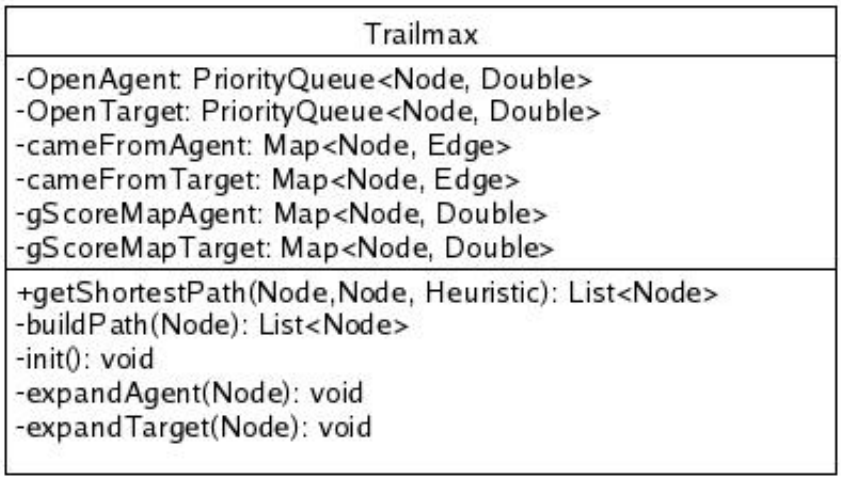


Figure 3.24: Classe Trailmax

La classe *Trailmax* estende la classe astratta Pathfinder ed implementa una versione dell'omonimo algoritmo di cui ne abbiamo già ampiamente discusso nella sezione 2.6

Chapter 4

Applicazioni e Risultati sperimentali

In questo capitolo verranno messi a confronto i diversi algoritmi di pathfinding implementati e verranno descritti gli esperimenti condotti per la valutazione di questi ultimi. Prima della parte sperimentale descriverò la logica di generazione degli ambienti dove saranno condotti gli esperimenti.

4.1 Le Mappe

Al fine dell'esperimento sono state sviluppate tre diverse metodologie di generazione pseudocasuale di mappe basate su griglie connesse in 8 direzioni. Le metodologie di generazione sono descritte nelle prossime sezioni.

4.1.1 Mappe *Dungeon*

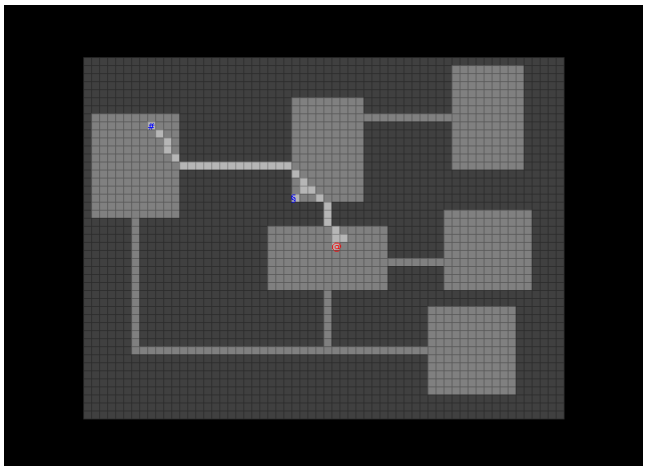


Figure 4.1: mappa *Dungeon*

Questo tipo di mappa si compone di grandi stanze che consistono in un rettangolo di tile traversabili collegate fra loro da lunghi corridoi. Per la realizzazione di questo tipo di mappa ci si è liberamente ispirati allo stile delle mappe sotterranee tipiche della saga di *Dungeons and Dragons* (vedi figura 4.2).

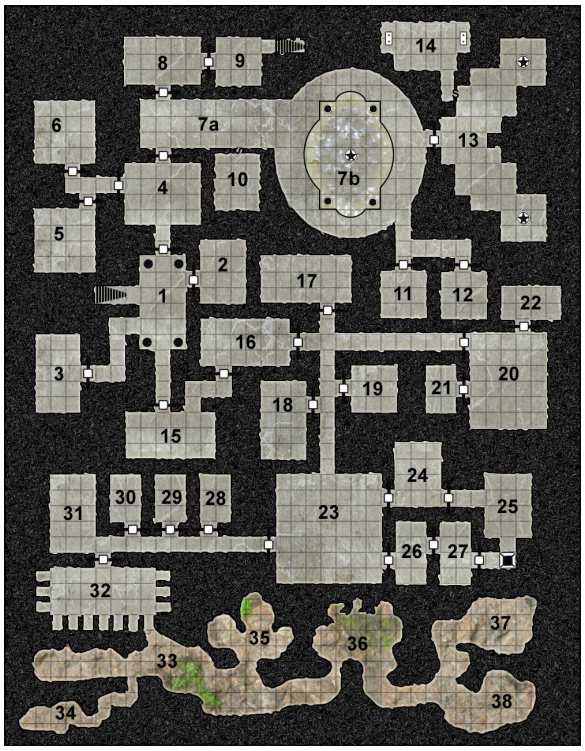


Figure 4.2: Esempio mappa Dungeons and Dragons

L'algoritmo utilizzato per la generazione di questo tipo di mappa prende in ingresso una griglia di interi interamente riempita di 1 con cui si rappresentano dei tile bloccati. In seguito sceglie in modo casuale (ma entro un certo range prestabilito) la posizione e le dimensioni di una stanza da posizionare nella mappa. Tali parametri vengono scelti in modo da non sovrapporsi con quelle già posizionate. Infine, quando la somma delle aree delle *stanze* posizionate sarà maggiore o uguale a una percentuale prestabilita, l'algoritmo provvedera' a connetterle creando dei tunnel ortogonali da una stanza a un'altra. In particolare, iterativamente ogni stanza verrà connessa con quella successivamente creata mediante un tunnel di locazioni traversabili i cui estremi sono i centri delle stanze di partenza e di arrivo. Nel caso in cui sia l'ascissa che l'ordinata dei due centri sono distinti, un fattore di casualità del 50% determinerà se il tunnel creato sarà prima orizzontale e poi verticale o viceversa. L'algoritmo utilizzato e' il seguente:

Algorithm 7: Dungeon map generator

Data: MAX_ROOM_SIZE: integer, MIN_ROOM_SIZE: integer, COVERAGE_PERCENTAGE: integer, rooms: list of *Room* object

Input: Array bidimensionale di interi con dimensioni *WIDTH* x *HEIGHT*

Result: L'array bidimensionale in ingresso viene elaborato in una mappa

```
1  Function main (map)
2      while covered_area < COVERAGE_PERCENTAGE do
           /* assegno le dimensioni della stanza e la sua
              posizione randomicamente                          */
3          w ← random(((ROOM_MAX_SIZE - ROOM_MIN_SIZE) +
            1) + ROOM_MIN_SIZE;
4          h ← random(((ROOM_MAX_SIZE - ROOM_MIN_SIZE) +
            1) + ROOM_MIN_SIZE;
5          x ← random(WIDTH - W - 1) + 1;
6          y ← random(HEIGHT - H - 1) + 1;
7          room ← new Room(w,h,x,y);
8          noGood ← false;
9          for r ∈ rooms /* controllo che non ci siano
              sovrapposizioni con le stanze gia' presenti       */
10         do
11             if room.intersect(r) then
12                 noGood ← true;
13                 break;
14             end
15         end
16         if !noGood /* riempio di zeri la griglia nelle
              coordinate corrispondenti alla stanza             */
17         then
18             for i = room.X to (room.X + room.W) do
19                 for j = room.Y to (room.Y + room.H) do
20                     map_ij ← 0;
21                 end
22             end
23             rooms.add(room);
24             covered_area ← covered_area + room.getArea();
25         end
26     end
27     createTunnels(map); /* connetti le stanze create      */
```

```
27 Function createTunnels(map)
28     prev ⟵ ∅;
29     for r ∈ rooms do
30         if r.hasPrev() then
31             prev ⟵ r.prev;
32             if random(range(0,100)) > 50 /* decido casualmente
                   se creare prima un tunnel orizzontale e poi
                   verticale o viceversa                      */
33              then
34                 createHorizontalTunnel( map, prev.getCenterX()
                                            r.getCenterX(), prev.getCenterY());
35                 createVerticalTunnel( map, prev.getCenterY()
                                          r.getCenterY(), r.getCenterX());
36             end
37             else
38                 createVerticalTunnel( map, prev.getCenterY()
                                          r.getCenterY(), prev.getCenterX());
39                 createHorizontalTunnel( map, prev.getCenterX()
                                            r.getCenterX(), r.getCenterY());
40             end
41         end
42     end
43 Function createHorizontalTunnel(map, x1, x2, y)
44     for i = min(x1, x2) to max(x1, x2) do
45         map_{i,y} ← 0;
46     end
47 Function createVerticalTunnel(map, y1, y2, x)
48     for j = min(y1, y2) to max(y1, y2) do
49         map_{x,j} ← 0;
50     end
```

4.1.2 Mappe *Outdoor*

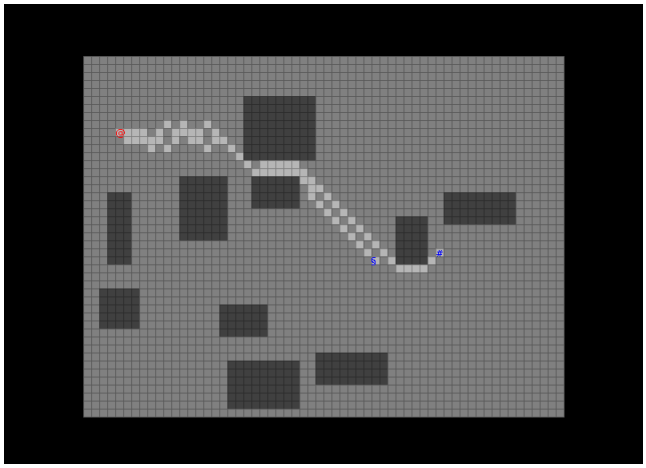


Figure 4.3: mappa *Outdoor*

Questo tipo di mappa, in maniera speculare al tipo di mappa della sezione precedente, si caratterizza da una griglia completamente riempita da tile traversabili dove vengono posizionati casualmente degli ostacoli rettangolari composti di tile bloccati di dimensioni a loro volta casuali. Come nella tipologia di mappe precedenti, gli ostacoli vengono posizionati in modo da non sovrapporsi, perchè dotati del medesimo metodo *intersect()*. L'algoritmo utilizzato e' grossomodo speculare a quello di generazione delle mappe di tipo *Hallways*, fatta eccezione per la creazione dei tunnel. L'algoritmo di generazione terminerà dunque quando una certa percentuale di area di gioco sarà coperta da ostacoli.

4.1.3 Mappe *Indoor*

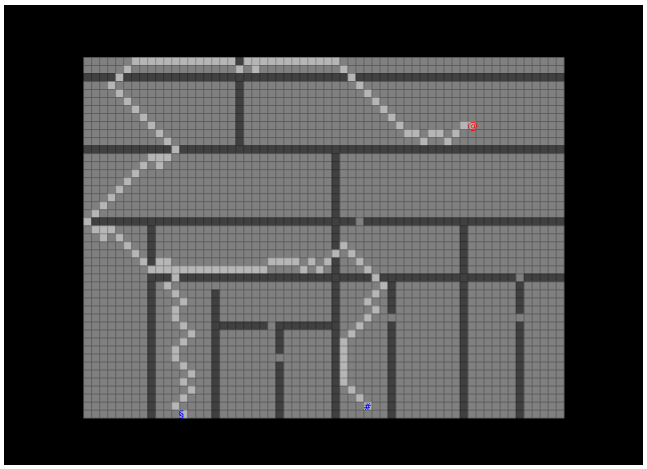


Figure 4.4: mappa Indoor

L'ultimo tipo di mappa realizzato consiste in uno spazio partizionato in diversi sottospazi (o stanze) servendosi di *tile* non traversabili come muri divisori. Per rendere raggiungibili le stanze fra loro ogni muro divisorio contiene un varco traversabile. L'algoritmo utilizzato divide ricorsivamente una griglia inizialmente totalmente traversabile. A seconda dell'orientamento, esso divide la griglia orizzontalmente o verticalmente disegnando un muro divisore e scegliendo un punto casuale su di esso dove posizionare il passaggio. In seguito dividerà ricorsivamente i due sottospazi partizionati con orientamento opposto, finchè non verra' raggiunto il caso base della ricorsione, ossia quando le stanze avranno dimensioni minori del minimo ammissibile.

Algorithm 8: indoor map generator

Data: ROOM_MIN_SIZE: integer
Input: ;
Array bidimensionale di interi con dimensioni *WIDTH* x *HEIGHT* di soli zeri ;
offset di x e y ;
larghezza ed altezza della stanza ;
orientamento della divisione da effettuare
Result: L'array bidimensionale in ingresso viene elaborato in una mappa di tipo indoor

```
1  Function divide (map, x_offset, y_offset, width, height, orientation)
2      if (width < ROOM_MIN_WIDTH) OR
         (heigth < ROOM_MIN_HEIGHT) then
3          return;
4      end
       /* divido orizontalmente o verticalmente                    */
5      horizontal ← orientation == true;
       /* scelgo da dove comincera' il muro                        */
6      if horizontal then
7          wx ← x_offset;
8          wy ← y_offset + random(height − 2);
9      end
10     else
11         wx ← x_offset + random(width − 2);
12         wy ← y_offset;
13     end
       /* scelgo un punto lungo il muro da usare come
          passaggio                                                */
14     if horizontal then
15         px ← wx + random(width);
16         py ← wy;
17     end
18     else
19         px ← wx;
20         py ← wy + random(height);
21     end
       /* scelgo la lunghezza del muro                             */
22     if horizontal then
23         lenght ← width;
24     end
25     else
26         lenght ← height;
27     end
```

```
28
    /* disegno il muro                                          */
29  if horizontal then
30      dx ← 1;
31      dy ← 0;
32  end
33  else
34      dx ← 0;
35      dy ← 1;
36  end
37  for i = 0 to lenght do
38      if wx ≠ px AND wy ≠ py then
39          map_{wx,wy} ← 1;
40      end
41      wx ← wx + dx;
42      wy ← wy + dy;
43  end
44  nx ← x_offset;
45  ny ← y_offset;
    /* se ho diviso orizzontalmente, dividi al di sopra del
       muro e poi al di sotto.  Altrimenti prima a sinistra
       e poi a destra                                           */
46  if horizontal then
47      new_width ← width;
48      new_height ← wy − y_offset + 1;
49  end
50  else
51      new_width ← wx − x_offset + 1;
52      new_height ← height;
53  end
54  divide(map, nx, ny, new_width, new_height, w < h);
55  if horizontal then
56      nx ← x_offset;
57      ny ← wy + 1;
58      new_width ← width;
59      new_height ← y_offset + height − wy − 1;
60  end
61  else
62      nx ← wx + 1;
63      ny ← y_offset;
64      new_width ← x_offset + width − wx − 1;
65      new_height ← height;
66  end
67  divide(map, nx, ny, new_width, new_height, w < h);
68 end
```

4.2 Setup sperimentale

Gli esperimenti sono stati condotti sui 3 tipi di mappe descritti in questo capitolo. Per i 3 tipi di mappe precedentemente descritte sono state generate 80 esemplari casuali di dimensioni 128×128. Per ammortizzare la varianza l'esperimento verrà ripetuto 30 volte per ogni mappa, scegliendo casualmente 30 coppie di nodi. Tutte le mappe generate ammettono spostamenti in 8 direzioni, ed è stata pertanto utilizzata la octile distance come funzione euristica. Compareremo Lazy Moving Target Adaptive-A* con Bidirectional-A*, A* e Dijkstra. Gli esperimenti sono stati condotti su un PC Lenovo ThinkPad T430 con processore Quad-Core Intel(R) Core(TM) i5-3360M CPU @ 2.80GHz e 4GB di memoria disponibili.

4.2.1 Moving Target Test

In questo tipo di test viene posizionato un agent *inseguitore* nella cella di partenza, il quale calcola un percorso verso l'agent posizionato nella cella di arrivo e si muove lungo quel percorso. A sua volta l'agent-target posizionato nella cella di arrivo calcola una via di fuga rispetto l'agent inseguitore con l'algoritmo Trailmax e si muove lungo tale percorso di evasione ma con una frequenza minore dell'agent inseguitore per garantire che l'agent-target venga raggiunto. In tal modo l'agent inseguitore è costretto a calcolare periodicamente un nuovo percorso ogni volta che l'agent-target si sposta lungo il suo percorso di fuga. Tutti gli algoritmi testati generano gli stessi percorsi e pertanto il numero di spostameti e di ricerche è approssimativamente lo stesso. Possono differire leggermente perchè l'agent target potrebbe calcolare diverse vie di fuga a parità di punti di partenza e di arrivo. Riporteremo tre parametri per la valutazione dell'efficienza degli algoritmi testati, ossia il numero di celle esplorate (finchè il target non è raggiunto) e il tempo totale di ricerca (finchè il target non è raggiunto) e il tempo medio di ricerca. Per calcolare il tempo medio si divide il tempo totale di ricerca per il numero di ricerche. Nelle parentesi quadre è riportata anche la deviazione standard della media del dato a cui si riferisce per dimostrare il significato statistico dei risultati. Il modo più ragionevole per comparare questi algoritmi potrebbe essere usando il tempo medio di ricerca. Tuttavia esistono diversi fattori in grado di influenzare questo dato, tra i quali il set di istruzioni del processore a basso livello, le ottimizzazioni effettuate dal compilatore e *coding decisions*. I risultati raccolti sono mostrati nella tabella 4.1.

Table 4.1: Risultati Moving Target Test

	(a)	(b)	(c)	(d)	(e)
Mappe Indoor [128 x 128]					
Dijkstra	59998	325955	196985912 [1374079.83]	457375 [3713.48]	7.62
A*	59527	325531	72426330 [738483.62]	232646 [2201.19]	3.90
BidirectionalA*	61674	333168	96354496 [897852.14]	318996 [3123.85]	5.17
LMTAA*	59850	325463	47225902 [467953.82]	183409 [1906.68]	3.06
Mappe Outdoor [128 x 128]					
Dijkstra	33034	184782	75214932 [369453.94]	137655 [1390.61]	4.16
A*	31283	182788	4713514 [22905.72]	16403 [419.91]	0.52
BidirectionalA*	31687	184195	5097733 [27219.57]	18066 [509.82]	0.57
LMTAA*	31507	182978	4656222 [22239.15]	23478 [460.13]	0.74
Mappe Dungeon [128 x 128]					
Dijkstra	94898	227818	139216186 [379096.35]	173209 [445.51]	1.82
A*	93844	226848	20696365 [92937.03]	31412 [127.43]	0.33
BidirectionalA*	94897	229313	24812703 [123594.4]	39316 [172.94]	0.41
LMTAA*	94059	227060	17834146 [73994.16]	41126 [155.69]	0.43

(a) = numero di ricerche finchè il target non è raggiunto;
(b) = numero di mosse finchè il target non è raggiunto;
(c) = totale delle celle espanse finchè il target non è raggiunto [deviazione standard della media];
(d) = tempo totale di ricerca finchè il target non è raggiunto (in millisecondi) [deviazione standard della media];
(e) = tempo medio di ricerca (in millisecondi)

Dai risultati raccolti si evince che per tutti e 3 i tipi di mappe LMTAA* è risultato il migliore in termini di nodi espansi rispetto tutti gli altri algoritmi di ricerca, ma non in termini di tempo di ricerca, per il quale A* è risultato per quasi tutti i casi sempre il migliore, fatta eccezione per le mappe indoor, dove LMTAA* è risultato migliore anche in tempo di ricerca.

Il motivo per il quale LMTAA*, pur espandendo meno nodi degli altri algoritmi di ricerca, non risulta il migliore anche in termini temporali di A* è perchè LMTAA* esegue un numero maggiore di operazioni basilari per una singola iterazione rispetto ad A*, quali ad esempio accessi ad *hashmap* di dimensioni molto grandi. Queste operazioni di accesso ad hashmap giustificano l'*overhead* generato da una singola iterazione di LMTAA* rispetto ad A*.

LMTAA* è risultato in particolar modo più efficace degli altri algoritmi di ricerca negli esperimenti condotti sulle mappe di tipologia indoor. L'algoritmo A*, ed in generale gli algoritmi di ricerca euristici, hanno difficoltà a calcolare una soluzione sulle mappe concave poichè, per loro natura, tendono ad esplorare i nodi più vicini al target, spesso allargando la loro frontiera di esplorazione ai nodi presenti nelle concavità della mappa che di fatto non sono utili a calcolare un percorso ottimo verso il target poichè conducono ad un vicolo cieco.

Questo è il caso delle mappe indoor, nelle quali l'algoritmo A*, ad ogni ricerca del nodo target, è forzato ad esplorare tutti i nodi che compongono una singola stanza, prima di arrivare al target, contrariamente ad LMTAA*, che all'aumentare delle ricerche diventa sempre più consapevole della distanza reale di ogni nodo verso il target come mostrato nell'esempio nella sezione 2.5, escludendo quindi quei nodi che di fatto conducono ad un vicolo cieco.