Pathfinding su giochi grid-based

Lentisco Francesco
N86001092

1 Le Mappe

Un gioco grid-based (o tile-based) è un tipo di videogioco dove l'area di gioco consiste in piccole immagini grafiche rettangolari, quadrate o esagonali, note come *tiles*. L'insieme completo dei *tile* disponibili per l'area di gioco è chiamato *tileset*. I *tile* sono disposti in maniera adiacente l'uno dall'altro nella griglia. I giochi *tile-based* di solito simulano una veduta dall'alto (*top-down*) dell'area di gioco e sono generalmente bidimensionali.

Nel progetto proposto ci si è serviti di mappe *grid-based* generate in modo randomico a seconda del *pattern* desiderato. La mappa consiste in una griglia con larghezza e altezza fissate ed ogni cella (*tile*) puo' essere bloccata o traversabile. L'implementazione dei grafi usati per costruire il sistema di navigazione di ogni mappa è fornita dalla libreria *JGrapht*[1].

Ogni *tile* della mappa corrisponde ad un nodo nel grafo corrispondente. Ogni nodo è connesso con tutti i nodi associati ai *tile* adiacenti e non bloccati (in 8 direzioni). I *tile* non attraversabili sono associati a un nodo con zero archi entranti. I pesi degli archi ortogonali hanno un costo unitario, pertanto gli archi tra due nodi collegati diagonalmente hanno un costo di $\sqrt{2}$.

Per riferirsi a un nodo del grafo partendo dalle sue coordinate sulla grigla e viceversa si usano delle semplici formule di conversioni: siano x e y le cordinate geometriche del nodo del grafo che si vuole referenziare

$$nodo = y * MAP\_WIDTH + x \tag{1}$$

dove MAP_WIDTH si intende la larghezza della mappa. Per convertire invece un nodo del grafo nelle sue coordinate geometriche si utilizzano le seguenti:

$$x = nodo\%MAP\_WIDTH \tag{2}$$

$$y = nodo/MAP\_WIDTH \tag{3}$$

[1] http://jgrapht.org/

Le mappe possono essere generate secondo 3 diversi *pattern* di generazione al fine di rappresentare lo stile delle mappe tipiche dei retro game.

1.1 Mappe *Hallways*

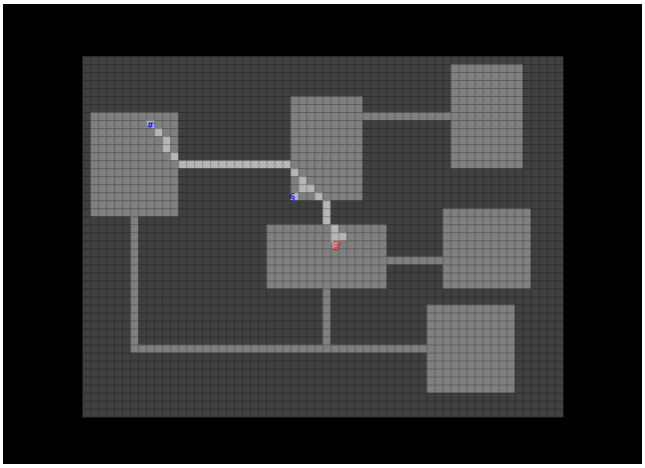

Figure 1: mappa *Hallways*

Questo tipo di mappa si compone di grandi stanze collegate da lunghi corridoi. Una classe *stanza* e' cosi' definita:

Listing 1: Classe java "*Room*"

```
public class Room implements TileMapElement{
        private int x, y, width, height, area;

        public Room(int x,int y, int width, int height){
                this.x=x;
                this.y=y;
                this.width=width;
                this.height=height;
                this.area=width*height;
        }
}
```

Una stanza consiste in un rettangolo di *Tile* traversabili. Gli attributi x e y sono le coordinate dell'angolo in basso a sinistra della stanza mentre *width* e *height* rispettivamente larghezza e altezza della stanza (espressi in numero di *Tile*). Tale classe e' inoltre dotata di un metodo *intersect()* così definito:

Listing 2: Funzione *Intersect*

```
public boolean intersect(Object other){
    if(other.getClass()!=Room.class)
        return false;
    Room r = (Room) other;
    return !(this.x+ this.width < r.x
        || r.x + r.width < this.x
        || this.y+this.height < r.y
        || r.y + r.height < this.y);
}
```

L'algoritmo utilizzato per la generazione di questo tipo di mappa prende in ingresso una griglia di interi interamente riempita di 1, ossia totalmente riempita di *Tile* non traversabili. In seguito sceglie in modo randomico (ma di un certo range prestabilito) la posizione e la grandezza delle stanze da posizionare nella mappa. Le stanze vengono scelte in modo da non sovrapporsi. Infine, dopo aver posizionato tutte le stanze, l'algoritmo provvedera' a connettere tutte le stanze creando dei tunnel ortogonali da una stanza a un'altra riempendo di zeri la griglia nelle coordinate dei punti che compongono il tunnel. L'algoritmo utilizzato e' il seguente:

Algorithm 1: Hallways map generator

Data: MAX_ROOM_SIZE: integer, MIN_ROOM_SIZE: integer, MAX_ROOMS: 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      for i = 0 to MAX_ROOMS 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         end
25     end
26     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
```

1.2 Mappe *Outdoor*

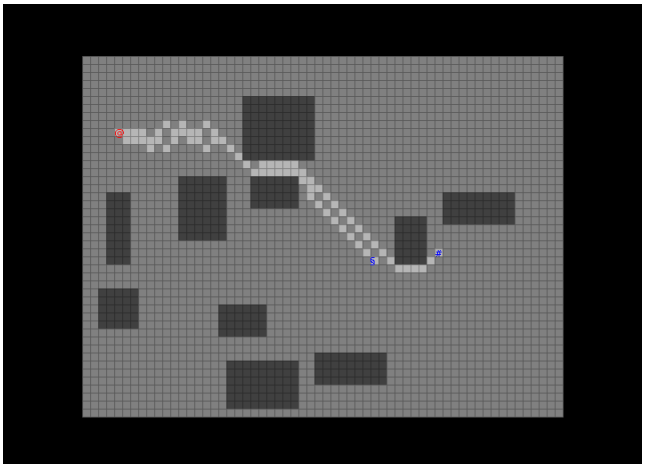

Figure 2: mappa *Outdoor*

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 da *Tile* non attraversabili di dimensioni a loro volta casuali. Così come le stanze del tipo di mappa precedente, anche gli ostacoli di questo tipo di mappa non si sovrappongono. L'algoritmo utilizzato e' del tutto simile e speculare a quello di generazione delle mappe di tipo *dungeon* eccetto per la creazione dei tunnel.

1.3 Mappe *Indoor*

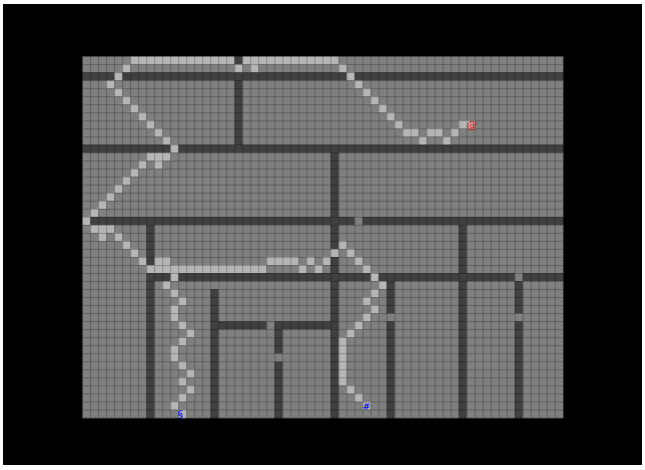

Figure 3: mappa Indoor

L'ultimo tipo di mappa realizzato consiste in uno spazio partizionato in diversi sottospazi (o stanze) servendosi di *tile* non traversabili come *muri* divisori. Le stanze sono raggiungibili da altre stanze mediante un *tile* che viene lasciato attraversabile in ogni muro divisore. L'algoritmo utilizzato altro non è che una variante dell'algoritmo della divisione ricorsiva dello spazio. 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, finchè non verra' raggiunto il caso base della ricorsione, ossia quando le stanze avranno dimensioni minori del minimo ammissibile.

Algorithm 2: 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
```

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 fissato dagli sviluppatori del gioco, come ad esempio il percorso di pattugliamento di un area che una guardia deve seguire. I percorsi fissi sono semplici da implementare, ma possono essere facilmente ingannati, ad esempio se un oggetto si frappone nel percorso. *Agent* più complessi non sanno in anticipo dove dovranno muoversi. Una unità in un gioco di strategia in tempo reale potrebbe ricevere 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 può lavorare direttamente sulla geometria del livello di gioco. Esso richiede che l'area di gioco sia rappresentata in una particolare struttura dati: un grafo diretto pesato e non-negativo. In questo capitolo introdurremo la struttura dati grafo e in seguito verranno presentati tutti gli algoritmi implementati nel progetto, tra cui Dijkstra, A* e diverse sue varianti.

2.1 Dijkstra

L'algoritmo di Dijkstra è un algoritmo utilizzato per cercare i cammini minimi (o Shortest Paths, SP) in un grafo con o senza ordinamento, ciclico e con pesi non negativi sugli archi. Tale algoritmo non è stato progettato per il pathfinding nel senso inteso nei videogochi (*point-to-point)*, ma piuttosto per risolvere il problema dei *cammini minimi*.

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. 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 siamo 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 in ogni iterazione come "nodo corrente". Per ogni arco esamina il suo nodo terminale e memorizza il costo totale del cammino percorso fino ad arrivare ad esso (*cost-so-far)* insieme all'arco dal quale si è arrivati in apposite strutture dati. Per quanto riguarda la prima iterazione, il nodo corrente è il nodo di partenza e il costo totale per ogni sua connessione è semplicemente il costo di quella connessione. 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 confronterà 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 **decreasePriority**() sono rispettivamente $\mathcal{O}(log(n))$ e $\Theta(1)$, la complessità dell'algoritmo nel caso peggiore è di $\mathcal{O}(|E| + |V| \log |V|)$.

Algorithm 3: Dijkstra Shortest Path

```
1  Function DijkstraShortestPath(Graph, source)
       /* 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.decreasePriority(v, alt);
19             end
20         end
21     end
```

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 più breve percorso. Questa nozione di probabilità è data da funzioni *euristiche*. Più la funzione euristica utilizzata sarà accurata, più l'algoritmo sarà efficiente

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

Per ogni arco uscente dal nodo corrente, A* esamina il suo nodo terminale e memorizza il costo totale del cammino percorso fino ad arrivare ad esso (*cost-so-far)* insieme all'arco dal quale si è arrivati in apposite strutture dati, esattamente come Dijkstra. Inoltre A* memorizza un valore in più: una stima euristica del costo totale del percorso(*f(x)*), passando per quel nodo, dal nodo di partenza al nodo di arrivo. Questo costo totale stimato è la somma di due valori: il costo totale (*cost-so-far*, da questo punto in poi ci riferiremo a questo valore come *g(x)*) di quel nodo e la distanza (euristica) dal nodo in esame fino al nodo di arrivo. Questa stima è tipicamente generata da una funzione euristica che di solito non fa parte dell'algoritmo stesso.

Queste stime sono chiamate *valori euristici* e ci riferiremo per semplicità a essi come *h(x)*. Pertanto $f(x) = g(x) + h(x)$.

A* tiene traccia dei nodi visitati 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* alla fine della loro stessa iterazione.

Diversamente da Dijkstra, viene estratto dalla coda *Open* il nodo con il minor valore *f(x)*. Questa variazione permette di valutare per primi i nodi più promettenti. Se la stima è accurata allora i nodi che sono più vicini al nodo di arrivo vengono considerati per prima, restringendo il campo di ricerca nelle aree più proficue.

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 calcoleremo il valore *g(x)* come al solito e se esso è minore di quello già memorizzato per quel nodo, allora verrà aggiornato con il suo nuovo valore. Da notare che effettueremo questo controllo solamente sul valore *g(x)*, giacchè è l'unico valore non influenzato dalla stima euristica.

Diversamente da Dijkstra, A* può trovare cammini migliori per nodi che sono già stati processati, e che quindi si trovano nella lista *Closed*. Può infatti accadere che una stima effettuata precendentemente sia stata molto *ottimista*, e che un nodo sia processato pensando che sia la scelta migliore, quando di fatto non lo è.

Se un nodo "dubbio" viene messo nella lista *Closed*, vuol dire che tutti i suoi archi uscenti sono stati processati. Può quindi accadere che un intero insieme di nodi può essere stato influenzato da quella valutazione erronea. In questo caso aggiornare i valori di quel nodo non sarà sufficiente, giacchè la modifica deve essere propagata lungo le sue connessioni uscenti. Nel caso in cui il nodo "dubbio" si trova invece nella coda *Open*, ciò non è necessario, visto che le sue connessioni uscenti non sono ancora state visitate.

Esiste un approccio molto semplice per ricalcolare e propagare il valore aggiornato di un nodo "dubbio", ossia estrarlo dalla lista *Closed* e inserirlo nuovamente nella coda *Open* con i suoi valori aggiornati. Esso stazionerà nella coda finchè arriva il suo turno di essere processato e in tal modo le sue connessioni saranno riconsiderate. Ogni nodo i quali valori si basano su valori erronei verrà quindi riconsiderato e processato 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 valore *f(x)* 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. In altre parole, la maggior parte delle implementazioni di A* si basano sul fatto che può 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 non essere 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

Algorithm 4: 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
```

una funzione. Di seguito riporteremo alcune delle sue implementazioni molto comuni in linguaggio Java utilizzate nel progetto.

Listing 3: Alcune funzioni euristiche

```
private class EuclideanDistance implements
    AStarAdmissibleHeuristic<Integer>{

    @Override
    public double getCostEstimate(Integer sourceVertex,
        Integer targetVertex) {
        int sourceX, sourceY, targetX, targetY;
        sourceX=sourceVertex%GameMap.WIDTH;
        sourceY=sourceVertex/GameMap.WIDTH;
        targetX=targetVertex%GameMap.WIDTH;
        targetY=targetVertex/GameMap.WIDTH;
        return Math.sqrt(Math.pow(sourceX - targetX,2)
                        +Math.pow(sourceY - targetY,2));
    }
}
```

```
private class ManhattanDistance implements
    AStarAdmissibleHeuristic<Integer> {

    @Override
    public double getCostEstimate(Integer sourceVertex,
        Integer targetVertex) {
        int sourceX, sourceY, targetX, targetY;
        sourceX=sourceVertex%GameMap.WIDTH;
        sourceY=sourceVertex/GameMap.WIDTH;
        targetX=targetVertex%GameMap.WIDTH;
        targetY=targetVertex/GameMap.WIDTH;
        return Math.abs(sourceX - targetX)
              +Math.abs(sourceY - targetY);
    }
}
```

```
private class OctileDistance implements
    AStarAdmissibleHeuristic<Integer> {

    @Override
    public double getCostEstimate(Integer sourceVertex,
        Integer targetVertex) {
        int sourceX, sourceY, targetX, targetY;
        sourceX=sourceVertex%GameMap.WIDTH;
        sourceY=sourceVertex/GameMap.WIDTH;
        targetX=targetVertex%GameMap.WIDTH;
        targetY=targetVertex/GameMap.WIDTH;
        return Math.abs(Math.abs(sourceX - targetX) - Math.
            abs(sourceY - targetY))
             + Math.sqrt(2)*Math.min(Math.abs(sourceX -
                 targetX), Math.abs(sourceY - targetY));
    }
}
```

In particolare le ultime due corrispondono al costo minimo di uno spostamento tra due nodi quando tutte le celle adiacenti non sono bloccate. 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*. Tutte le funzioni euristiche proposte, oltre ad essere consistenti rispetto il target, soddisfano anche la disuguaglianza 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*.

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 4 e 5, 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 4 e 5, sinistra) è molto più lunga e non-realistica rispetto al percorso minimo nell'ambiente continuo (Figure 4 e 5, destra)

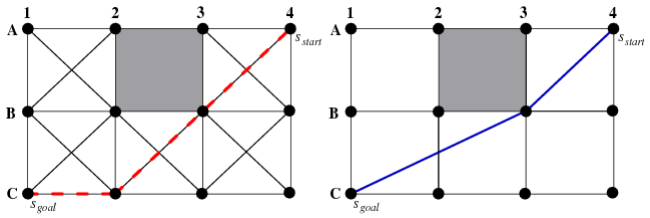

Figure 4: Square Grid: A* vs. Theta*

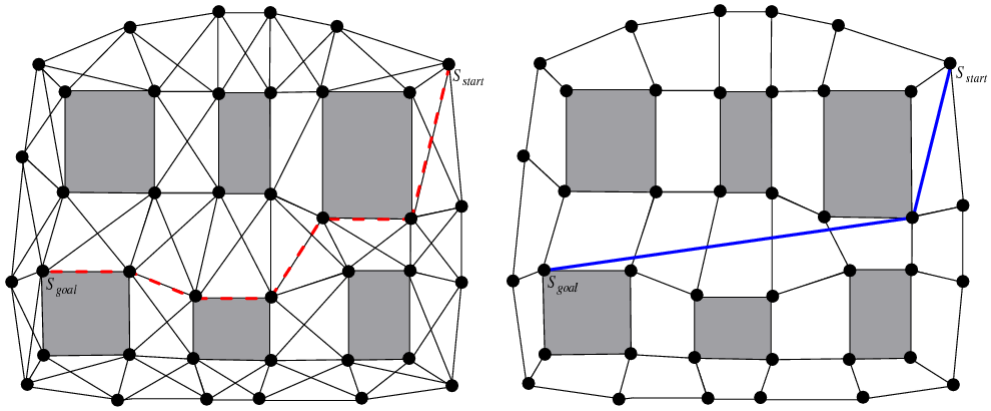

Figure 5: Navmesh: A* vs Theta*

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-smoothing* 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 6 dalla linea rossa discontinua.

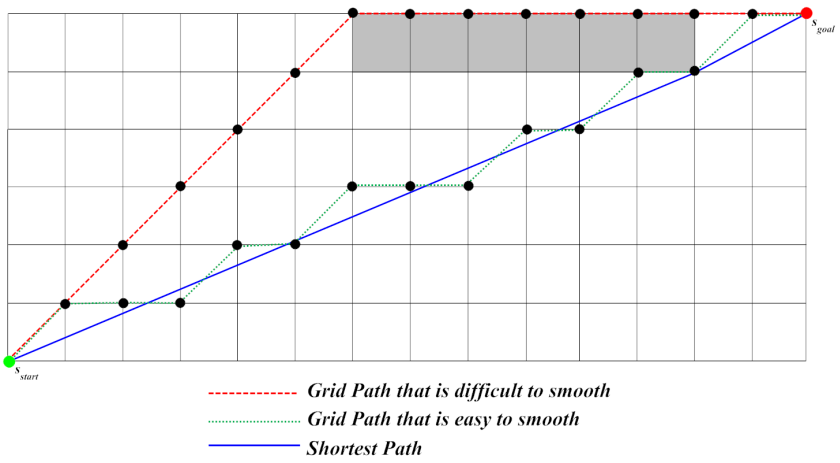

Figure 6: Percorso trovato da A* con differenti tecniche di path-smothing

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 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 vin-

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
```

cola 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 *Compute-Cost*(), 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)*.

- **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* utilizzando solamente operazioni logiche e operazioni su interi, come mostrato nel seguente blocco di codice.

Listing 4: Funzione Line Of Sight

```
public static boolean lineOfSight(int x1, int y1, int x2,
    int y2) {
     int dy = y2 - y1;
     int dx = x2 - x1;
     int f = 0;

     int signY = 1;
     int signX = 1;
     int offsetX = 0;
     int offsetY = 0;

     if (dy < 0) {
          dy *= -1;
          signY = -1;
          offsetY = -1;
     }
     if (dx < 0) {
          dx *= -1;
          signX = -1;
          offsetX = -1;
     }

     if (dx >= dy) {
          while (x1 != x2) {
               f += dy;
               if (f >= dx) {
                    if (blocked(x1 + offsetX, y1 + offsetY))
                         return false;
                    y1 += signY;
                    f -= dx;
               }
               if (f!=0 && blocked(x1 + offsetX, y1 + offsetY))
                    return false;


               if (dy == 0 && blocked(x1 + offsetX, y1)
                           && blocked(x1 + offsetX, y1 - 1))
                    return false;

               x1 += signX;
          }
     }else{
          while (y1 != y2) {
               f += dx;
               if (f >= dy) {
                    if (blocked(x1 + offsetX, y1 + offsetY))
```

```
                    return false;
                x1 += signX;
                f -= dy;
            }
            if (f!=0 && blocked(x1 + offsetX, y1 + offsetY))
                return false;
            if (dx == 0 && blocked(x1, y1 + offsetY)
                        && blocked(x1 - 1, y1 + offsetY))
                return false;

            y1 += signY;
        }
    }
    return true;
}
```

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* simultaneamente. 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* e *forwardClosed* e *backwardClosed* i loro set *Closed*:

1. Inizializza i nodi di *start* e di *goal*. Inserisci il nodo start e il nodo goal rispettivamente in *forwardOpen* e in *backwardOpen*.

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 $\exists n$: $n \in Open_f \cap Open_b$ allora, sia x = min(x, $g_f(n) + g_b(n)$) allora, se x $\leq$ max($Open_f.min(), Open_b.min()$) allora termina l'algoritmo restituendo il percorso passante per il nodo n. Altrimenti torna allo step 2.

I punti critici di questo algoritmo si possono riassumere in: (i) scelta tra le due ricerche e (ii) terminazione. Per quanto riguarda 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[2] 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, in modo più performante di A* perchè in grado di aggiornare i valori h utilizzando informazioni raccolte in ricerche precedenti. In parole povere esso trasforma

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

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 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**[3]).

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 può cambiare. Questo perchè i valori euristici calcolati in ricerche precedenti allo spostamento del target risulterebbero incoerenti con la sua nuova posizione. Si ricorda che i valori euristici per essere coerenti ed ammissibili non devono mai violare la proprietà della *disuguaglianza triangolare*. 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. Il nocciolo 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 *while*, per poter trovare ripetutamente nuovi cammini minimi dal nodo corrente dell'agent al nodo corrente del target. Dopo una ricerca, se viene trovato un percorso, l'agent si muove lungo di esso finchè il

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

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      end
48      search(s) ⟵ 0;
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        26
80      action cost (if any)
81  end
82  return true
83 end
```

target non viene raggiunto, oppure 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 la variabile *search(s)* per indicare se i valori *g(s)* e *h(s)* di un nodo s sono stati inizializzati nella *counter*-esima ricerca. I valori *h(s)* e *g(s)* sono stati inizializzati nella *counter*-esima ricerca se *search(s) = counter*. 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 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 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*).