

# VICKY

---

Autore: Marco Ceriani

*Allegati*

– *Vicky: IA per la ricerca dei percorsi con algoritmo A\* in ampiezza in C#*

## Sommario

Introduzione .....	2
Ricerca in ampiezza .....	2
Applicazione nella robotica .....	3
Adattamento per mappe 3d.....	4
Analisi metodi del modulo.....	4
map_loader() .....	4
pathfind() .....	6
esamina_nodo().....	6
calcola_H() .....	7
calcola_G() .....	7
calcola_F().....	8
Breve descrizione del flusso del modulo.....	8

## Introduzione

Vicky è un'intelligenza artificiale di tipo "PathFinding".

Si intenda IA di tipo PathFinding l'implementazione di un algoritmo in grado di trovare, data una mappa schematizzata in celle e i punti di partenza e arrivo, il percorso più breve tra quest'ultimi esaminando il minor numero di nodi possibili, imitando quindi i meccanismi di ragionamento umano.

## Ricerca in ampiezza

Vicky utilizza un algoritmo di ricerca di tipo A\* in ampiezza, vediamo di analizzarlo in dettaglio.

Una volta ricevute le coordinate del punto di partenza e di quelle del punto di arrivo, ha inizio il processo di ricerca del percorso.

Per prima cosa vengono "scoperte" tutte le celle che circondano quella di partenza, e, per ognuna, viene calcolato un "costo totale".

Il costo totale di ogni cella è ricavato nel seguente modo:

(costo asse + costo intrinseco) = costo di movimento

Costo di movimento + costo euristico = costo totale

Il "costo asse" è un valore che dà un peso al movimento verso una cella a seconda dell'asse di provenienza, e viene utilizzato per rendere il percorso più "umano".

Infatti, giocando con questo parametro, è possibile rendere più convenienti i movimenti orizzontali piuttosto che quelli diagonali, evitando che l'IA effettui un percorso a "zig zag".

Il costo intrinseco, invece, è un valore con cui è possibile pesare una caratteristica della cella, si può per esempio aumentarne tale costo se la cella rappresenta un percorso in salita o discesa, o addirittura settarlo ad "infinito" nel caso in cui si volesse indicare una strada a senso unico presa contromano, come nelle IA utilizzate dai navigatori satellitari.

Il calcolo euristico è il calcolo della distanza tra la cella in esame e quella di arrivo ed è un elemento molto importante dell'algoritmo. Questo elemento non deve mai fornire una distanza esatta (per averla bisognerebbe esaminare tutte le celle, non avendo più un'intelligenza e quindi tutti i vantaggi derivati) e non deve mai sovradimensionarla.

La scelta, in Vicky, è caduta sulla tecnica "Manhattan".

La tecnica "Manhattan" effettua semplicemente il calcolo della distanza delle coordinate, ignorando tutti i dati della mappa, terreni, ostacoli e qualsiasi altra cosa.

A questo punto viene scelta la cella con il costo totale minore.

Nel caso si abbiano più celle con lo stesso costo totale, l'esame proseguirà su tutte queste, procedendo appunto in "ampiezza".

Una volta aperta la nuova cella si prosegue nel medesimo modo, ma con un'eccezione:

al costo di movimento verrà sommato anche il costo totale della cella padre (la cella da cui la si sta scoprendo, che avrà a sua volta sommati i valori delle precedenti "celle padre").

Se la cella era già stata scoperta nell'analisi di un altro percorso il costo totale ottenuto verrà confrontato con quest'ultimo e verrà "assimilata" dalla cella padre solo se questo risulterà minore.

Così facendo, le celle, avranno nei loro "costi totali" memoria del percorso migliore da cui sono state scoperte.

Questa operazione continuerà ad essere effettuata in parallelo su tutti i percorsi in esame.

Se uno di questi percorsi porta a un vicolo cieco verrà terminato, mentre se risulterà essere peggiore di un altro percorso in esame verrà sospeso.

Se tutti i rami in esame vengono terminati perché ciechi, verrà ripreso l'esame dell'ultimo percorso "migliore" sospeso.

Infine, se tutti i rami vengono terminati, vuol dire che è impossibile raggiungere l'obiettivo.

## Applicazione nella robotica

L'algoritmo appena illustrato è ottimo per un software come videogiochi e navigatori satellitari, ma non è utilizzabile in campi come quello della robotica.

L'IA, infatti, prevede l'esame di più percorsi in parallelo, cosa impossibile nel caso di un robot che non ha una visione globale della mappa, ma solo i dati ottenuti dal suo sistema sensoriale.

Per ovviare a questo problema vanno modificati alcuni punti nell'algoritmo.

Nel caso in cui più celle in esame ottengano lo stesso costo totale, l'IA deve procedere con l'esame di solo UNA di queste. Non avrà, così facendo, la possibilità di "sospendere" i rami peggiori e dovrà continuare con un esame in profondità del ramo scelto.

Nel caso in cui arrivi in un vicolo cieco dovrà poi tornare alla cella precedente scegliendo la seconda cella con costo più basso, e così via, ripercorrendo a ritroso il ramo ogni volta che avrà esaurito un nodo.

L'algoritmo così modificato non permette di avere la certezza di aver scelto il percorso più breve (anche se, nei test che ho effettuato, in almeno il 95% dei casi vi riesce) ma ottiene comunque sempre un percorso corretto verso l'obiettivo e senza esaminare tutte le possibilità, ovviando, inoltre, al problema di non poter esaminare più percorsi contemporaneamente.

## Adattamento per mappe 3d

Gli algoritmi fin qui visti prendono tutte in considerazione un “mondo bidimensionale”. Non vengono infatti tenuti in considerazione molti fattori di un mondo 3d.

Riporto qua di seguito l’analisi grezza che feci per un possibile adattamento di Vicky a mondi tridimensionali, capace quindi di potersi adattare a qualsiasi possibile scenario. E’ da notare che molti riferimenti in analisi sono ad un possibile scenario di un videogioco. Tutte le modifiche che risultano da questa analisi all’algoritmo originale, non sono state implementate causa mancanza di tempo, vaglierò in futuro un possibile sviluppo di un modulo simile. Lo riporto comunque nel caso qualche considerazione potesse tornare utile a qualcuno che stesse affrontando uno sviluppo simile.

---

### Analisi metodi del modulo

#### map\_loader()

**Nota:** Viene richiamato in fase di inizializzazione dell’oggetto, ma credo sia il caso di renderlo pubblico, in quando dovrebbe essere richiamato a ogni modifica della mappa (porte che diventano apribili, muri che cadono diventando ostacoli o aprendo nuove strade, etc etc).

#### Descrizione funzionale:

Il metodo map\_loader() è assolutamente uno degli elementi più critici del modulo.

Il suo compito è quello di scansionare la mappa (grezza o pre-elaborata da moduli esterni) e di generarne una rappresentazione a celle conforme alle esigenze dell’algoritmo di analisi.

Il metodo caricherà la mappa in un array ( o altro a seconda delle possibilità del linguaggio in uso ) sotto forma di celle logiche.

I dati che dovrà immagazzinare per ogni cella, per poter essere in grado di gestire qualsiasi situazione, devono essere le seguenti:

**ID\_univoco:** indispensabile per identificare in modo univoco la cella, sarà una stringa composta dalle coordinate X,Y,Z in modo da poter ricostruire le posizioni delle celle in un mondo 3D. esempio: X12Y2Z1

**Costo intrinseco della cella:** i costi sono la valutazione della “fatica” nell’affrontare una cella che avrà l’IA quando la esaminerà. Essi si possono dividere in due tipi di costi, di movimento e intrinseco. Il costo intrinseco definisce la difficoltà che si ha a percorrere una determinata cella a causa delle sue proprietà, esso quindi potrà variare a seconda del tipo di terreno (pianura, sconnesso, palude, acqua ....etc etc etc!). Il costo di movimento verrà analizzato più avanti in quanto non è una problematica della mappa.

**Array connessioni:** deve essere un array di dati contenente due informazioni, ID\_univoci delle celle connesse e un int che indica se la connessione è attiva o meno con un livello di categoria.

La scelta di aggiungere questo array di dati è stata sofferta (si aumenta la mole di dati caricati) ma indispensabile per risolvere alcune problematiche specifiche.

Il progetto iniziale prevedeva che ogni cella si identificasse come ostacolo o meno, ma questa soluzione non avviava a un problema particolare che mi è venuto in mente (e immagino a molti altri con le stesse caratteristiche che non mi sono venuti in mente).

Immaginiamo un primo piano di celle, come un pavimento che occuperà la dimensione Z 0. In questo piano è presente una scala che conduce a un secondo piano di un ipotetica casa. La scala si troverà quindi in una dimensione Z 1, dove sarà presente solo lei come cella percorribile. Il secondo piano della casa sarà invece un altro piano di celle posizionato alla dimensione Z 2.

Il problema è che una normale scala sarà accessibile solo da un lato, avrà passamani ai lati e non sarà accessibile da dietro (anche se connessa alla cella dietro!). Quindi la cella su cui ha la base la scala non potrà essere flaggata come ostacolo (è accessibile da davanti) ma nemmeno semplicemente come cella libera (è bloccata su tutti i lati tranne uno).

La scelta di sostituire un booleano “ostacolo” con un array di “connessioni” risolve questo problema inserendo una maggiore elasticità (se viene rotto un passamano basterà flaggare l’int della connessione corrispondente ad “aperto” e ricaricare la mappa) dando la possibilità all’intelligenza di adattarsi ad ostacoli “perimetrici” sulla cella. Un muro avrebbe invece tutte le connessioni “chiuso”, venendo quindi valutato un ostacolo da ogni posizione, o potrà essere rappresentato come una connessione su un solo lato di una cella libera (scelte a questo punto non più influenti per l’IA).

Perché quindi usare un int invece di un booleano?

La risposta è semplice, per permettere una categorizzazione dei personaggi, potendo quindi distinguere “quale personaggio può passare dove”.

Infatti non tutti i tipi di personaggio avranno lo stesso tipo di limitazioni. Un uccello può volare mentre un fantasma può attraversare i muri. La categorizzazione indicherà che la connessione è aperta a tutte le categorie superiori al numero fissato sulla connessione, e chiusa a tutte quelle inferiori. Questo parametro viene passato dalla Fuzzy Logic alla funzione pathfind ().

## **pathfind()**

**Nota:** potrebbe essere Pubblico. Dovrebbe essere l'unico metodo richiamabile del modulo (oltre al metodo `map_loader()`), anche se, teoricamente, sarebbe utilizzato solamente dalla Fuzzy Logic (si potrebbe quindi inibirne l'utilizzo al di fuori dell'IA del personaggio). Da valutare se è utile anche al resto dell'applicazione (Fuzzy escluso).

### **Descrizione funzionale:**

Il metodo verrà richiamato da tutte le Fuzzy Logic dei personaggi in scena ogni volta che uno di questi avrà un cambio di obiettivo (sia questo dovuto a un cambio comportamentale o per aver raggiunto l'obiettivo precedente) ed avrà bisogno di un ricalcolo del percorso.

Il metodo avrà bisogno di tre informazioni in ingresso, ovvero l'ID univoco di partenza (composto, come visto prima, dalle 3 coordinate XYZ) , l'ID univoco di arrivo e la categoria del personaggio.

`PathFind()` sarà inoltre il metodo che accederà a tutti gli altri della classe per scatenare l'algoritmo di ricerca.

Tornerà un array di ID univoci (che rappresenteranno il percorso).

Sarà poi compito della Fuzzy Logic del personaggio interagire col mondo fisico.

## **esamina\_nodo()**

**Nota:** metodo privato , vuole in ingresso ID in esame e ID di arrivo del percorso e categoria del personaggio

### **Descrizione funzionale:**

Il metodo `esamina_nodo()` verrà richiamato dall'algoritmo principale ogni volta che verrà aperta una nuova cella in esame.

Il suo compito sarà quello di esaminare tutte le connessioni della cella aperta (circoscrivendo un cubo intorno alla cella) e di ritornare all'algoritmo tutte le celle raggiungibili con i rispettivi costi intrinseci+movimento ( richiamando il metodo `calcola_G` ) e le rispettive euristiche. Per calcolare l'euristica delle celle connesse avrà bisogno dell'ID di arrivo come dato in ingresso ed utilizzerà il metodo `calcola_H()`. La categoria del personaggio verrà utilizzata nella rilevazione delle celle raggiungibili. Un valore superiore alla categoria passata indicherà una cella inibita.

## calcola\_H()

**Nota:** metodo privato, vuole in ingresso ID in esame e ID di arrivo del percorso

### Descrizione funzionale:

Questo metodo viene chiamato dal metodo `esamina_nodo()` passando la cella della connessione in esame e quella di arrivo e torna il valore euristico della cella.

Il calcolo euristico è il calcolo della distanza tra la cella in esame e quella di arrivo ed è un elemento molto importante dell'algoritmo che potrebbe essere modificato in casi particolari (ed è per questo che è stato estrapolato dal metodo `esamina_nodo()` che gli ripassa l'ID di arrivo del percorso).

Questo elemento non deve mai fornire una distanza esatta (per averla bisognerebbe esaminare tutte le celle, non avendo più un'intelligenza e quindi tutti i vantaggi derivati) e non deve mai sovradimensionarla. La scelta è quindi caduta sulla tecnica "Manatthan", modificata però per lavorare in un ambiente 3D.

La tecnica "Manatthan" effettua il calcolo della distanza delle coordinate, ignorando quindi tutti i dati della mappa, terreni, ostacoli e qualsiasi altra cosa.

## calcola\_G()

**Nota:** metodo privato, vuole in ingresso la cella attuale e quella in esame

### Descrizione funzionale:

Il compito del metodo `calcola_G()` è quello di calcolare il costo di movimento (accennato in precedenza) per raggiungere una cella e di sommarvi il costo intrinseco, tornando quindi il costo totale. Il costo di movimento diventa quindi un'altra variabile manipolabile che dona ulteriore elasticità all'algoritmo. Questo costo serve per differenziare la fatica che si può avere compiendo i movimenti sui vari assi (un movimento in salita sarà più faticoso di uno in discesa o uno in piano).

Questo costo non è legato alla mappa ma alla "gravità" del mondo o alle preferenze che si vogliono avere (si usa pesare leggermente di più i movimenti obliqui in quanto, per logica umana (e non di costo!) non è "usanza" zizzare camminando laddove se ne potrebbe fare a meno ( ma in giochi come Geometry Wars, alcuni personaggi potrebbero invece preferire zizzare! ).

Bisogna quindi decidere come parametrizzare questi costi, se differenziarli per personaggi, o renderli fissi all'avvio dell'applicazione.

E' da notare che al peso di movimento dovrà essere sommato anche il peso di tutti i pesi (movimenti+ intrinseci) fino a quel momento per il percorso preso, questo per avere una "memoria" del percorso sui costi

## **calcola\_F()**

**Nota:** metodo privato, vuole in ingresso un nodo

### **Descrizione funzionale:**

Questo metodo viene semplicemente utilizzato per sommare il costo totale all'euristica per tornare la valutazione finale di una cella. Questa operazione è stata isolata, come per le altre operazioni "elementari" dell'algoritmo, per permetterne eventuali modifiche in futuro senza intaccare il resto del codice.

## **Breve descrizione del flusso del modulo.**

In inizializzazione viene richiamato il metodo `map_loader()` che caricherà tutti i dati relativi alla mappa in memoria.

Questa funzione dovrà essere richiamata tutte le volte che la mappa subirà delle modifiche.

Ogni volta che un personaggio vorrà effettuare un movimento, richiamerà il metodo `pathfind()` passandogli le coordinate di partenza e arrivo sotto forma di ID univoci.

Il metodo `pathfind()` inizierà così la ricerca del percorso.

Aprirà la cella di partenza col metodo `esamina_nodo()`.

Questo metodo esaminerà tutte le connessioni della cella in cui mi trovo, circoscrivendo un cubo intorno al punto di partenza.

Tornerà quindi tutte le celle accessibili dalla mia attuale posizione (sfruttando gli int delle connessioni di cui si è parlato inizialmente) e i loro costi ( movimento + intrinseco dalla funzione `calcola_G`) e euristiche (`calcola_H`).

Una precisazione va fatta sulla funzione `calcola_G` che avrà due comportamenti differenti a seconda del fatto che la cella sia stata o meno già esaminata in precedenza.

Se la cella non è mai stata esaminata, gli verrà settato come "ID padre" l'ID univoco della cella da cui è stata scoperta, verrà sommato il suo costo intrinseco con il costo di movimento e con il costo totale della cella padre.

Il risultato sarà il costo totale di questa cella (che avrà quindi "memoria" del percorso).

Se, invece, la cella era già stata scoperta, verrà sommato il suo costo intrinseco con il costo di movimento e con il costo totale della cella da cui si proviene, il risultato verrà quindi confrontato con l'attuale costo totale della cella (che avrà memoria del percorso precedente da cui era stata scoperta).

Se l'attuale costo totale è minore, questo verrà sostituito a quello memorizzato nella cella e verrà sovrascritto l'ID padre con l'attuale ID di provenienza (scoperto un percorso migliore per raggiungere la stessa cella), sennò costo totale e padre verranno lasciati immutati.

Una volta ottenute tutte le celle accessibili con costi e euristiche corrispondenti, queste vengono impilate in memoria e ordinate per il risultato di `calcola_F()` (eseguito su queste celle).

A questo punto `pathfind()` seglierà la cella con `calcola_F()` minore eliminandola dalla memoria e rieseguirà `esamina_nodo()` dalla nuova posizione, impilando e riordinando le nuove celle scoperte nella pila principale.

Nel caso ci fossero più celle con lo stesso valore F, l'operazione di apertura nodo dovrà essere eseguita su tutte quante prima di proseguire alla scelta (per questo viene detta A\* in ampiezza, in quanto esamina più percorsi contemporaneamente lasciando morire gli alberi peggiori. "A\*" sta invece per il metodo di apertura "a stella" delle celle in esame).

Una corretta implementazione dell'algoritmo porterà al raggiungimento della cella di arrivo ( se raggiungibile ).

A questo punto verrà rieseguito il percorso a ritroso tracciando gli ID padri delle celle, partendo da quella obbiettivo. Si avrà così il percorso migliore tra la cella di partenza e quella di arrivo, tenuto conto di uno spazio tridimensionale e tenute conto variabili quali, ostacoli (con un esame perimetrico di questi), tipi di terreno (costo intrinseco della cella), comportamenti del personaggio (costo del movimento ortogonale/diagonale) e gravità del mondo (costo del movimento con influenza dell'asse Z).

L'array di ID ottenuto verrà ritornato alla Fuzzy Logic chiamante.