SOMMARIO

1. Introduzione

Il progetto FIAsarum nasce da una particolare melma/muffa (diverse fonti riportano nomi diversi, per comodità ci riferiremo ad essa come melma) chiamata, per l'appunto, melma policefala. Questa melma però, a differenza delle muffe, ha un particolare meccanismo di ricerca del cibo. La melma policefala è dotata di un particolare meccanismo di memoria "esterna": per estendersi secerne una secrezione che usa per ricordare dove sia già passata.

Il motivo per cui un organismo unicellulare di questo tipo ci interessa è perché scienziati da diverse parti del mondo hanno sfruttato questa sua caratteristica per diversi scopi. Il più semplice e ovvio è quello di risolvere labirinti, ma i ricercatori dell'università dell'Hokkaido l'hanno usata per ricreare la rete metropolitana di Tokyo. La melma è riuscita a ricreare alla perfezione dimostrandone l'efficienza e "l'intelligenza".

1.1. Obiettivo

Lo scopo del nostro progetto è quello di ricreare un agente in grado di simulare il comportamento della melma policefala. Useremo una meta, che rappresenterebbero la fonte di cibo, in una griglia.

1.2 Specifica PEAS

Performance: le misure di prestazione usate per valutare l'operato dell'agente, in questo caso valutiamo se il percorso preso dall'agente sia il più efficiente

Environment: Descrizione dell'ambiente in cui opera l'agente.

Actuators: Gli attuatori a disposizione dell'agente per compiere azioni. Nel nostro caso sono le direzioni che potrà intraprendere

Sensors: i sensori che avrà l'attore per ricevere input percettivi.

1.2.1 caratteristiche dell'ambiente

- L'ambiente è con **singolo agente:** la nostra "melma" che cercherà di raggiungere il cibo.
- L'ambiente è **Parzialmente osservabile**: sappiamo dove si trova la meta, ma dobbiamo esplorare il labirinto per capire quale sia il percorso migliore.
- L'ambiente è **statico:** l'ambiente resta invariato dalle azioni eseguite dall'agente.
- L'ambiente è **episodico**: l'azione successiva dipende dall'azione appena intrapresa.

1.3 Analisi del problema

Il problema è descrivibile così:

- **Stato iniziale**: Lo stato iniziale è definito tramite una scena in Unity dove l'agente potrà agire. La scena consiste in una griglia, di dimensione selezionabile, e in nella possibilità di poter vedere il punto di partenza e il punto di arrivo (Rosso)
- **Azioni possibili**: le azioni possibili per il nostro agente sono 4 direzioni: nord, sud, est, ovest.
- **Modello di transazione** **parte modificabile, questa è solo un'idea**: Ad ogni azione l'agente controllerà in quali direzioni può andare, principalmente saranno direzioni che vanno VERSO l'obiettivo. (se non ricordo male avevamo deciso che) A ogni bivio viene messo una sorta di "checkpoint", dove si ritorna nel caso una strada porti a un vicolo cieco, in maniera tale da scegliere un altro percorso ed esplorare quello.
- **Test obiettivo**: l'obiettivo della melma è quello di raggiungere il cibo più vicino con il miglior percorso possibile.
- **Costo cammino**: il costo del cammino è lo stesso (a meno che non decidiamo di voler creare un terreno difficile che costa di più).

Questo problema può essere affrontato in molti modi diversi. Possiamo subito pensare a metodi come la ricerca in ampiezza, per la ricerca non informata, oppure con l'A* per la ricerca informata.

2 Algoritmi di ricerca

2.1.1 Ricerca informata

Gli algoritmi di ricerca informata utilizzano informazioni aggiuntive per una maggiore efficienza nel trovare la soluzione; queste informazioni aggiuntive sono dette **euristiche**. Questi algoritmi sfruttano delle funzioni di valutazione che stimano il costo rimanente per raggiungere lo stato obiettivo utilizzando il minor numero di nodi esplorati.

Nel nostro caso si tratta quindi della distanza tra la "radice" della melma e la posizione del cibo nella griglia. Proprio perché il problema è modellato su una griglia, abbiamo calcolato l'euristica usando la distanza tra due punti.

2.1.2 Ricerca informata A*

L'algoritmo di ricerca informata A* viene utilizzata per trovare il percorso più breve.
L'algoritmo utilizza le funzioni:
- **g()** -> il costo *reale* dal nodo iniziale fino al nodo obiettivo;
- **h()** -> la nostra euristica; è il costo *stimato* dal nodo corrente al nodo obiettivo.

La somma delle due funzioni forma **f()**, ovvero il costo *stimato* del percorso più adatto dal nodo iniziale al nodo obiettivo.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AStar : MonoBehaviour
{
    public static List<Node> FindPathToGoal(Node start, Node goal)
    {
        PriorityQueue<Node> openList = new PriorityQueue<Node>();        // frontiera
        HashSet<Node> closedList = new HashSet<Node>();     // nodi esplorati

        // inizializziamo i valori del nodo di partenza
        start.g = 0f;
        start.h = EuclideanDistance(start, goal);
        start.f = start.h;
        start.parent = null;

        openList.Enqueue(start, start.f);  // inizializziamo la frontiera col nodo di partenza

        while (openList.Count > 0)
        {
            Node current = openList.Dequeue();
            if (current == goal)
                return ReconstructPath(current);

            closedList.Add(current);

            // controlliamo i vicini e calcoliamo le funzioni f, g, h di ognuno di loro
            foreach (Node n in current.neighbors)
            {
                if (!n.isObstacle && !closedList.Contains(n))
                {
                    float tentativeG = current.g + 1f;

                    if (tentativeG < n.g)
                    {
                        n.g = tentativeG;
                        n.h = EuclideanDistance(n, goal);
                        n.f = n.g + n.h;
                        n.parent = current;
```

```
                    openList.Enqueue(n, n.f);
                }
            }
        }
    }

    return null;    // nessun percorso trovato
}

/* ricostruiamo tutto il percorso dal nodo iniziale al nodo goal */
public static List<Node> ReconstructPath(Node current)
{
    List<Node> newPath = new List<Node>();

    while (current != null)
    {
        newPath.Add(current);
        current = current.parent;
    }
    newPath.Reverse();

    return newPath;
}

/* la nostra euristica; è stata scelta la distanza Euclidea perché è la scelta migliore in quanto la muffa si sposta in 8 direzioni */
public static float EuclideanDistance(Node a, Node b)
{
    return Mathf.Sqrt(Mathf.Pow(a.transform.position.x - b.transform.position.x, 2) + Mathf.Pow(a.transform.position.y - b.transform.position.y, 2));
}

}
```

L'algoritmo A* è **ottimale** ed è **completo**, secondo la completezza temporale.

- L'ottimalità dipende da due fattori dell'euristica: deve essere ammissibile e consistente.
- L'euristica è ***ammissibile*** se non sbaglia mai per eccesso la stima del costo per raggiungere l'obiettivo.
- L'euristica è ***consistente*** se per ogni nodo n, ed ogni nodo n' successore di n che viene generato al compiersi di un'azione a, il costo per raggiungere l'obiettivo a partire da n non è superiore alla somma del costo per raggiungere n' a partire da n e raggiungere l'obiettivo a partire da n'.
- La completezza dell'algoritmo A* è la completezza ***temporale***. Essa rende l'algoritmo ottimamente efficiente. Questo significa che nessun altro algoritmo, a parità di euristica, espande meno nodi.

2.2.1 Ricerca non informata

Passiamo poi alla ricerca non informata, trattasi di algoritmi di ricerca in cui non si dispone delle informazioni aggiuntive sugli stati.

2.2.2 Ricerca in Ampiezza

Per implementare la ricerca in profondità ci siamo serviti di una coda in cui inseriamo al suo interno tutti i nodi esplorati. Ad ognuno di questi, quando viene inserito nella coda, viene controllato se corrispondano o meno al "traguardo". Questo processo si ripete ogni volta per la lunghezza della coda, che varia asseconda della grandezza della griglia.

```
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class BFS : MonoBehaviour
{
    public static List<Node> exploredNodes;
    public static List<Node> FindPathToGoal(Node root)
    {
        Queue<Node> openList = new Queue<Node>();     // frontiera
        HashSet<Node> closedList = new HashSet<Node>();    // nodi esplorati

        // inizializziamo la frontiera col nodo di partenza
        root.parent = null;
        openList.Enqueue(root);

        while(openList.Count > 0)
        {
            Node current = openList.Dequeue();
            if (TestGoal(current))
            {
                exploredNodes = closedList.ToList<Node>();
                return ReconstructPath(current);
            }

            closedList.Add(current);

            foreach (Node n in current.neighbors)
            {
                if (!n.isObstacle && !closedList.Contains(n) && n != null)
                {
                    if (!openList.Contains(n))
                    {
                        openList.Enqueue(n);
                    }

                    n.parent = current;
                    n.previouslyExploredByBFS = true;
                }
```

```
            }
        }

        return null;   // nessun percorso trovato
    }

    /* ricostruiamo tutto il percorso dal nodo iniziale al nodo goal */
    public static List<Node> ReconstructPath(Node current)
    {
        List<Node> newPath = new List<Node>();

        while (current != null)
        {
        newPath.Add(current);
        current = current.parent;
        }
        newPath.Reverse();

        return newPath;
    }

    public static bool TestGoal(Node n)
    {
        if(GameObject.Find("Food(Clone)").transform.position == n.transform.position)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

L'algoritmo di ricerca in ampiezza è completo, con complessità temporale esponenziale e complessità spaziale in funzione al numero massimo di successori.

- **Completezza**: L'algoritmo troverà il nodo corretto dopo aver espanso tutti gli altri nodi che lo precedono.
- La **complessità temporale** è **esponenziale**: per dimostrarlo, considerando di dover ricercare una soluzione in un albero in cui ogni nodo ha il massimo numero di successori stabilito, che nomineremo b. La radice genererà una quantità b di nodi e ogni nodo genererà una quantità b di nodi a loro volta. Questo significa che ci troviamo una quantità di nodi uguale a b^2 al secondo livello. Più si scende e più la potenza aumenta (b^3 al terzo livello, b^4 al quarto e così via). Nel caso pessimo, scenderemo una quantità di b^n nodi, dove n è la profondità massima.
- La **complessità spaziale** è in funzione al numero massimo di successori, semplicemente perché l'algoritmo memorizza ogni nodo espanso nell'insieme esplorato. Nel caso specifico della ricerca in ampiezza, verranno memorizzati tutti i nodi generati in memoria. Quindi avremo una quantità di nodi nell'insieme esplorato uguale a O(numero massimo di nodi successori elevato alla profondità minima-1) e

avremmo una frontiera uguale a O(numero massimo di nodi successori elevato alla profondità minima). Quindi la complessità dipende dalla dimensione della frontiera.