Relazione del progetto di **Ingegneria della Conoscenza** - Estensione di *Dimensions Break*

Gruppo di lavoro:

- *Corvaglia Carlo,* 718859, c.corvaglia7@studenti.uniba.it
- *Gallo Angela,* 715989, a.gallo51@studenti.uniba.it

Repository:
https://github.com/h-angel22/ICON_Project/tree/main

Video dimostrativo delle funzionalità:
https://youtu.be/dxzGpVI0eQo

INDICE

INTRODUZIONE

Nella seguente documentazione verrà descritta la relazione tecnica del progetto
di Ingegneria della Conoscenza Anno Accademico 2022-2023.
Il progetto svolto consiste nella espansione di un videogioco realizzato da noi in precedenza.
L'espansione consiste in 3 fondamentali elementi:

- Generatore di livelli (inteso come risolutore di CSP)
- Un oggetto presente in ogni livello generato che indica la strada al giocatore (implementando un algoritmo di pathfinding)
- Un comportamento più intelligente del boss di fine livello (mediante albero di decisione appreso da un dataset fornito da noi)

L'idea è nata da un interessamento generale da parte del gruppo nel migliorare il videogioco.
Per questo progetto si è utilizzato il linguaggio Python (versione 3.10), Godot e VisualStudio Code come IDE.

Tutti i file sorgente realizzati per il progetto si trovano nella cartella *project/ICON_src/*. Al suo interno si trova anche la cartella python in cui si trovano gli script che implementano le funzioni descritte sopra.

PASSI PROPEDEUTICI

Siamo partiti definendo formalmente la struttura di un livello come una tripla che comprende il grafo delle stanze, la stanza iniziale e la stanza del boss (queste ultime appartenenti all'insieme delle stanze).

$G = (R, D)$
$L = (G, s \in R, b \in R)$

Questo ci torna utile per poter implementare un pathfinding e per definire i vincoli per il CSP.
Abbiamo poi integrato nella classe che gestisce il comportamento del boss una serie di parametri che è in grado di leggere per percepire lo stato della stanza e quindi del combattimento, parametri anche usati negli esempi dell'apprendimento automatico.

CONSTRAINT SATISFACTION PROBLEM

Sommario

In intelligenza artificiale, un CSP (Constraint Satisfaction Problem) è un problema in cui si deve trovare un'assegnazione di valori a un insieme di variabili in modo che un insieme di vincoli sia soddisfatto. I vincoli sono relazioni tra due o più variabili che definiscono le combinazioni di valori consentite per le variabili.

Lo scopo del CSP nel nostro caso è la generazione di un livello.

Come detto in precedenza, il livello è strutturato come un grafo, per cui il nostro CSP utilizza come variabili gli archi che collegano il grafo. Il dominio di una variabile è composto da tutti i possibili collegamenti che si possono avere tra le stanze selezionate (scelte casualmente da una lista, a cui si aggiungono tre stanze speciali, quali, quella iniziale, quella del boss, e quella del tesoro).

I vincoli sono stati definiti in modo da ottenere un livello funzionante in tutti i suoi aspetti.

Strumenti utilizzati

Abbiamo utilizzato per la risoluzione del CSP i moduli messi a disposizione da "AIPython: Python Code for AIFCA". Il risolutore di CSP viene richiamato dal gioco in Godot che fa partire il processo python (chiamando il file mainCSP.py) e ne prende i risultati dallo standard output.

```
func _generate_level():
    var level_str = [""]
    while level_str[0] == "":
        var exit_code = OS.execute("python3", [py_path], true, level_str)
        print(level_str)
    var l = Level.new()
    l.initialize(level_str[0])
    ResourceSaver.save("user://lvl.tres", l)
    levels.append("user://lvl.tres")
    emit_signal("loaded")
    call_deferred("_load_level",loadlvl)
```

Decisioni di Progetto

L'output è fornito come stringa che associa alla path del file della stanza i collegamenti alle stanze adiacenti, da cui è possibile formarne un grafo.

La base di conoscenza è fornita dal file rooms.txt con l'elenco delle possibili stanze da scegliere associate a 4 numeri. Questi 4 numeri rappresentano le possibili direzioni da cui ci si può muovere in una stanza (sinistra, destra, su e giù). In questo file i numeri assumono solo i valori di -1 e -2. Una direzione segnata da -2 indica una direzione non ammessa da quella stanza. Il -1 indica una direzione consentita ma non assegnata. L'output viene emesso con la stessa sintassi.

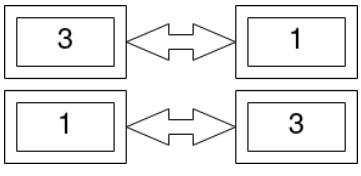
```
CheckpointRoom.tscn -1 -1 -1 -1
CompassRoom.tscn -1 -1 -1 -1
NewBossRoom.tscn -2 -2 -1 -1
NewRoom1.tscn -1 -2 -2 -1
NewRoom10.tscn -1 -1 -2 -1
NewRoom2.tscn -1 -1 -1 -1
NewRoom3.tscn -2 -1 -1 -1
```

Abbiamo definito l'arco come una quadrupla di valori di cui 2 devono rimanere sempre nulli. Gli altri 2 valori assegnati indicano le due stanze che vengono collegate dall'arco. Quindi, se l'arco assume valore del tipo (x, y, null, null) abbiamo un collegamento che dalla porta di destra della stanza x si arriva alla porta di sinistra della stanza y. Nel caso (null, null, x, y) la porta in basso della stanza x conduce alla porta in alto della stanza y. IL risolutore di CSP lavora sull'assegnamento di queste quadruple agli archi.

```
CheckpointRoom.tscn -1 -1 1 -1
CompassRoom.tscn 5 6 4 0
NewBossRoom.tscn -2 -2 -1 5
NewRoom7.tscn -1 4 -1 -1
NewRoom2.tscn 3 -1 5 1
NewRoom3.tscn -2 1 2 4
NewRoom4.tscn 1 -1 -2 -1
```

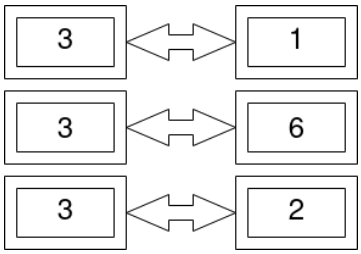
I vincoli che abbiamo definito sono i seguenti:

- Non possono esserci 2 archi uguali, perché noi andiamo a definire un numero finito di archi assegnabili e la ridondanza risulterebbe uno spreco.
- Non possono esserci archi con valori specchiati, questo causerebbe l'esistenza di porte che conducono alla stessa stanza da cui si sta uscendo. Esempio:
  Arco:

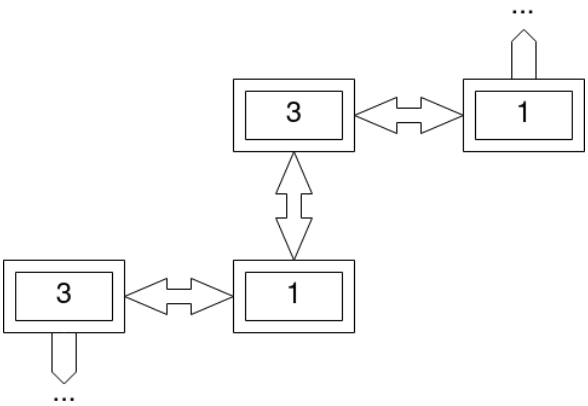


(1, 3, null, null)
(3, 1, null, null)
Qui muoversi verso destra o verso sinistra porta sempre alla stessa stanza, ciclicamente.

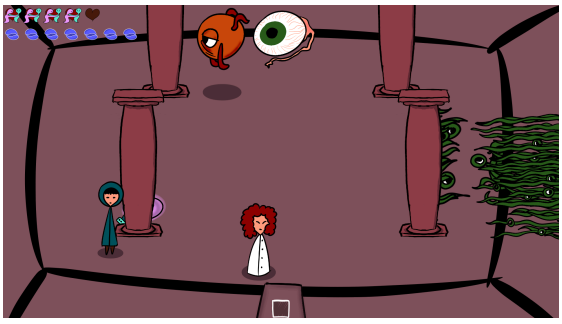
- Non si può ripetere un numero in una stessa posizione dell'arco, questo collegherebbe più di una stanza a una stessa porta e quella porta avrebbe un comportamento imprevedibile. Esempio:
  Archi:
  (3, 1, null, null)
  (3, 6, null, null)
  (3, 2, null, null)
  Nel caso riportato la porta sinistra delle stanze 1, 6 e 2 conduce alla porta destra di 3. Non è possibile prevedere a quale delle tre stanze la porta destra di 3 si collegherà.



- Non possono esserci archi orizzontali e verticali con gli stessi valori, il che significherebbe che la stessa stanza sarebbe adiacente sia in direzione verticale (verso l'alto o il basso) che orizzontale (verso destra o sinistra). Esempio:
  Archi:
  (3, 1, null, null)
  (null, null, 3, 1)
  In questo caso la stanza 1 si trova sia a destra della stanza 3 che sotto, cosa che porterebbe logicamente a una mappa infinita.



- Non ci può essere il collegamento se una stanza ha un ostacolo. Nell'esempio in questa immagine la stanza ha due ostacoli: l'occhio in alto e la pianta a destra. Non è accettata una assegnazione che utilizzi una porta in quelle posizioni. Anche se in questo caso non è presente è consentita una porta verso sinistra.



- Verifica che usi tutte le stanze tra quelle che sono state selezionate all'inizio.

- Deve risultare un grafo collegato. Altrimenti parte della mappa sarebbe irraggiungibile e quindi sarebbe anche probabile avere un livello irrisolvibile.

L'algoritmo risolutivo scelto da noi per la risoluzione del CSP è lo Stochastic Local Search (SLS). Il motivo principale risiede nelle sue performance nel nostro caso specifico.

Se il risolutore non dovesse trovare una soluzione entro 200000 passaggi. Se non dovesse trovare una soluzione in questi passaggi è molto probabile che non esistano soluzioni o che queste siano molto poche e quindi difficili da trovare. Quindi ricomincia dall'inizio ma con un insieme di stanze da collegare differente. Questo finché non trova una configurazione accettabile.

```
solver = RoomProblem(sub_dim, rooms_to_use)
solution = solver.solve()
if solution == None:
    print('Fallito tentativo', file=sys.stderr)
```

```
class RoomProblem():
    def __init__(self, sub_dim, rooms) -> None:
        self.dim = sub_dim
        self.num_archs = self.dim + 2
        self.rooms_to_use = rooms
```

Abbiamo definito il problema nella classe RoomProblem il cui costruttore prende la lista di stanze da riunire in un livello e la dimensione del livello (in numero di stanze). Questo ci ha permesso di implementare la soluzione attraverso una strategia di Divide Et Impera. Avevamo infatti notato che la soluzione veniva trovata molto in fretta se il numero di stanze non era superiore a 5. Oltre le 5 stanze si ha una crescita esponenziale del tempo. Quindi abbiamo implementato la classe Block che eredita da Room. Nella classe Block andiamo a salvare la soluzione di 4 stanze ottenute velocemente.

Dividendo quindi il numero di stanze da inserire nel livello per 4 si possono risolvere i gruppi di 4 molto rapidamente per passare la lista dei Block come fosse una lista di stanze al RoomProblem.

```
solver = RoomProblem(len(blocks), blocks)
archs_to_blocks(blocks, solver.solve())

print_rooms(final_rooms)
```

Per la generazione del livello abbiamo scelto arbitrariamente 11 stanze, quindi verranno creati 3 blocchi.

Valutazione

Le performance dell'algoritmo SLS sono state valutate secondo le metriche del tempo di esecuzione e della diversificazione dell'output. Entrambe queste metriche vengono beneficiate dalla tecnica di backtracking utilizzata nell'algoritmo. Il vantaggio sta infatti nel partire da un'assegnazione totale per poi modificare con passaggi successivi della ricerca locale, cosa che permette di trovare una soluzione con un numero di controlli ridotto rispetto ad altri algoritmi testati. Altro beneficio dato dalla ricerca locale è che ogni livello ha una disposizione di stanze completamente diverse tra loro. Con altri algoritmi, come per esempio il DFS la tendenza era quella di trovare una soluzione che sviluppasse il livello da sinistra verso destra e dall'alto verso il basso.

Abbiamo testato un approccio diverso utilizzando un algoritmo iterativo per confrontarlo con le prestazioni del CSP. Come per quest'ultimo, l'algoritmo prende 11 stanze random, incluse la stanza iniziale, la stanza della bussola, la stanza del boss. Utilizza 3 liste diverse, una per le stanze non ancora modificate (rooms_to_use), una per quelle che sono state modificate almeno una volta (used_rooms), una per le stanze che sono state modificate e non si possono più modificare in quanto tutte le porte disponibili sono occupate (completed_rooms).

```
rooms_to_use = load_file(full_path, dim)
used_rooms = []
completed_rooms = []
```

Inizialmente viene presa la prima stanza utile e inserita nella lista used_rooms per poter iniziare così la generazione del livello. A questo punto continuiamo a prendere stanze da rooms_to_use per inserirle come adiacenti a stanze della lista used_rooms casuali.

```
while len(rooms_to_use) != 0:
    random.shuffle(used_rooms)
    r = used_rooms[0]

    dir = r.get_direction()

    match dir:···

    for r_index in rooms_to_use:
        if r_index.can(dir_to_find):
            rooms_to_use.remove(r_index)
            attach_rooms(r, r_index, dir)

            if r.is_completed():
                used_rooms.remove(r)
                completed_rooms.append(r)

            if r_index.is_completed():
                completed_rooms.append()
            else:
                used_rooms.append(r_index)
            break
```

Usando questo metodo sistematico il tempo di esecuzione con lo stesso numero di stanze è ridotto al punto da essere un'esecuzione istantanea. Il vantaggio in questo caso è dato dal fatto che non controlla lo stato delle altre stanze per andare ad assegnare la nuova posizione. Il problema principale di come è stato

implementato il CSP è che i Constraint dovevano essere definiti per tenere conto dello stato di tutti gli archi contemporaneamente, questi controlli ripetuti così tante volte aumentano il tempo di esecuzione. Questo fenomeno si è mitigato notevolmente nel momento in cui è stato implementato il Divide Et Impera.

Per misurare le prestazioni abbiamo misurato il tempo impiegato dalle nostre tre implementazioni a generare livelli composti da 7, 9, 11 e 23 stanze (nel caso dell'implementazione senza Divide et Impera del CSP non abbiamo considerato il caso delle 23 stanze perché impiega troppo tempo). In entrambi i casi in cui usiamo il CSP per risolvere il problema notiamo che la deviazione standard è anche molto alta. Ciò è dovuto alla natura del algoritmo SLS che potrebbe trovare la soluzione immediatamente come anche poter impiegare minuti.

CSP SLS Base			CSP SLS Divide et Impera				No CSP			
7 stanze	9 stanze	11 stanze	7 stanze	9 stanze	11 stanze	23 stanze	7 stanze	9 stanze	11 stanze	23 stanze
10.864	9.743	158.536	0.133	0.106	0.874	2.217	0.012	0.018	0.018	0.017
5.905	11.628	181.316	0.117	0.895	0.132	2.840	0.014	0.018	0.018	0.017
0.832	4.432	631.394	0.032	0.283	0.300	7.576	0.012	0.018	0.018	0.018
10.275	9.207	119.048	0.194	0.723	0.219	20.945	0.011	0.017	0.017	0.018
0.267	1.009	529.085	0.070	0.173	0.121	1.152	0.012	0.017	0.018	0.018
1.009	3.509	58.925	0.067	0.808	1.084	0.373	0.012	0.017	0.018	0.017
1.732	1.409	241.011	0.176	0.033	0.408	0.879	0.012	0.018	0.018	0.018
7.993	32.038	101.199	0.141	0.574	0.290	14.235	0.013	0.018	0.018	0.017
1.196	0.243	573.433	0.093	0.304	0.488	4.052	0.013	0.017	0.018	0.018
5.524	31.257	176.065	0.056	0.172	0.185	12.420	0.012	0.018	0.018	0.017
Medie			Medie				Medie			
3.628	6.820	178.691	0.105	0.294	0.295	3.446	0.012	0.018	0.018	0.018
Varianze			Varianze				Varianze			
16.797	140.20	46122.02	0.003	0.099	0.106	48.875	0.000	0.000	0.000	0.000
Deviazioni standard			Deviazioni standard				Deviazioni standard			
409.9%	1184%	21476%	5.34%	31.5%	32.5%	699.1%	0.08%	0.05%	0.03%	0.05%

Da notare che alcune delle funzionalità della risoluzione tramite CSP non sono state implementate nella risoluzione tradizionale, queste però sarebbero state comunque realizzate in modo da essere eseguite in tempo costante (per esempio la possibilità nel livello di avere porte speciali che conducono in luoghi del livello più lontani). Semplicemente non era necessario per verificare le prestazioni in termini di tempo e ci ha risparmiato il dover scrivere diverse soluzioni ad hoc. Questo ci porta al vantaggio della soluzione mediante CSP, cioè l'ordine e la semplicità di modifica. Aggiungere i dettagli mancanti nel algoritmo iterativo ci avrebbe portati a scrivere diverso codice per specifiche soluzioni disseminate in diversi punti del codice e in più file. Invece per la risoluzione tramite CSP basta definire nuovi constraint all'interno della classe RoomProblem per modificarne il comportamento.

```
constraints = []

constraints.append( Constraint(vars, not_equals, "non possono esserci 2 archi uguali") )
constraints.append( Constraint(vars, not_mirrors, "non possono esserci archi con valori specchi
constraints.append( Constraint(vars, not_samecolumn, "non si può ripetere un unumero in una ste
constraints.append( Constraint(vars, not_opposite, "non possono esserci archi orizzontali e ver
constraints.append( Constraint(vars, can_connect(self.rooms_to_use), "non ci può essere il coll

constraints.append( Constraint(vars, all_used(self.rooms_to_use), "verifica che usi tutte le st
constraints.append( Constraint(vars, is_valid(self.rooms_to_use), "deve risultare un grafo coll

csp = CSP("stanze", vars, constraints)
solution = any_conflict_solver(csp)
```

PATHFINDING

Sommario

Il pathfinding è il processo di trovare un percorso tra due nodi in un grafo. Il percorso può essere definito come una sequenza di nodi che collega il nodo di partenza al nodo di destinazione.

Nel nostro progetto, abbiamo assegnato alle varie stanze un costo, che dipende dallo stato di completamento: se una stanza è già stata attraversata e non ha nemici al suo interno il costo sarà minore rispetto a quello di una stanza ancora da affrontare. Il livello era già rappresentato come un grafo collegato per valori.

Strumenti utilizzati

Abbiamo utilizzato per la ricerca del percorso i moduli messi a disposizione da "AIPython: Python Code for AIFCA". Sempre attraverso la chiamata di sistema di Godot abbiamo avviato il file mainSearch.py a cui viene passato anche l'array degli archi e dei rispettivi costi. Lo script python risolve il pathfinding attraverso l'algoritmo A* e passa attraverso lo standard output il risultato.

```
func _path_finder() -> Array:
	var path: Array
	var path_str = []
	OS.execute("python3", [py_path, _archs_str(), current_room], true, path_str)
	path = path_str[0].split(" --> ")
	print(path)
	return path
```

Decisioni di Progetto

Il percorso, quindi, viene aggiornato nel caso in cui dovesse cambiare lo stato delle stanze.

Lo scopo del pathfinding è il raggiungimento della stanza del boss, da qualunque stanza, a condizione che il giocatore possieda l'oggetto "bussola", che si trova nella stanza del tesoro. La

chiamata per calcolare il pathfinding viene effettuata nuovamente se il giocatore dovesse scegliere di ignorare le indicazioni.

L'input al pathfinder è dato sotto forma di archi rappresentati da 3 numeri (le due stanze collegate ed il loro peso). Questo ritorna semplicemente l'array con l'ordine delle stanze da seguire.

```
Input path finder:0 4 2 - 1 4 2 - 1 5 2 - 2 4 2 - 3 6 2 - 3 5 2 - 3 4 2 - 4 0 2 - 4 2 2 - 4 3 2 - 4 1 2 - 5 3 2 - 5 1 2 - 6 3 2
Output path finder:[1, 4, 2
]
Input path finder:0 4 2 - 1 4 2 - 1 5 2 - 2 4 2 - 3 6 2 - 3 5 2 - 3 4 2 - 4 0 2 - 4 2 2 - 4 3 2 - 4 1 2 - 5 3 2 - 5 1 2 - 6 3 2
Output path finder:[4, 2
]
Input path finder:0 4 2 - 1 4 2 - 1 5 2 - 2 4 2 - 3 6 2 - 3 5 2 - 3 4 2 - 4 0 2 - 4 2 2 - 4 3 2 - 4 1 2 - 5 3 2 - 5 1 2 - 6 3 2
Output path finder:[2
]
```

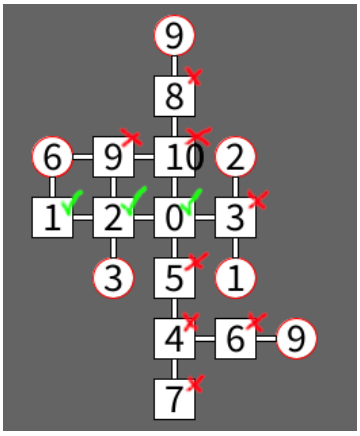
Il costo dell'arco è impostato in base a quale stanza sia la stanza di destinazione. Se la stanza non è ancora stata esplorata, vuol dire che avrà ancora dei nemici all'interno, quindi sarebbe una stanza che farebbe perdere più tempo al giocatore. Di conseguenza il costo dell'arco può assumere i valori:

- 2 se la stanza di destinazione non è ancora stata esplorata
- 1 se la stanza di destinazione è già stata esplorata

L'output è espresso come l'array di stanze da percorrere in ordine in modo da raggiungere la Boss Fight. Dopo la prima volta l'algoritmo di pathfinding viene richiamato solo se il giocatore non segue la path indicata per non dover fare più volte lo stesso calcolo.

Valutazione

Per dimostrare il funzionamento esaminiamo il seguente esempio.

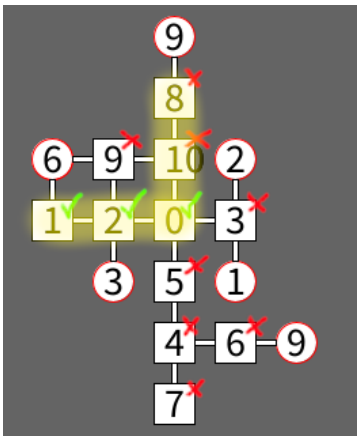


Nel livello rappresentato qui abbiamo segnato con una spunta verde le stanze già esplorate dal giocatore e con una x rossa le stanze non ancora esplorate.

La stanza 1 è quella che contiene l'oggetto che attiva il pathfinding mentre la stanza 8 è quella della Boss Fight che si vuole raggiungere.

Invocando lo script python gli passiamo gli argomenti dello stato attuale del livello. Gli archi sono unidirezionali e sono espressi tutti

due volte per indicare il costo di ogni spostamento nelle due direzioni.



Otteniamo quindi l'array [1, 2, 0, 10, 8], cioè la sequenza di stanze da seguire in ordine.

Visualizzando questo percorso sulla mappa si nota che è stato individuato il modo più veloce per muoversi dalla stanza 1 alla stanza 8.

```
Situazione livello: 0 10 2 - 0 5 2 - 0 2 1 - 0 3 2 - 1 3 2 - 1 2
1 - 2 3 2 - 2 1 1 - 2 0 1 - 10 8 2 - 10 0 2 - 10 9 2 - 4 5 2 - 4
7 2 - 4 6 2 - 5 0 2 - 5 4 2 - 6 4 2 - 6 9 2 - 10 8 2 - 10 0 2 - 1
0 9 2 - 8 9 2 - 8 10 2 - 9 8 2 - 9 6 2 - 9 10 2 - 10 8 2 - 10 0 2
 - 10 9 2 -
Stanza attuale: 1
Destinazione: 8
[1, 2, 0, 10, 8
]
```

ALBERO DECISIONALE APPRESO

Sommario

Un albero decisionale appreso è un modello di apprendimento automatico che rappresenta una serie di regole decisionali che possono essere utilizzate per classificare o predire un valore target.

Il processo di apprendimento di un albero decisionale inizia con un insieme di dati di training, che consiste di esempi di input e output. Nel nostro caso questi esempi sono stati generati mediante il log su console della lettura dei dati con il nostro output desiderato aggiunto manualmente,

```
left_wall_dist right_wall_dist top_wall_dist bottom_wall_dist x y target_dist at_risk action
1905.375854, 173.191956, 330.178955, 720.363708, 0.757249, -0.653127, 581.379395, 1, dash_left
1905.375854, 173.191956, 330.178955, 720.363708, 0.757249, -0.653127, 581.379395, 1, dash_left
1905.375854, 173.191956, 330.178955, 720.363708, -0.127768, -0.991804, 847.926697, 1, dash_left
1905.375854, 173.191956, 330.178955, 720.363708, -0.68257, -0.73082, 1014.550659, 1, dash_left
1905.375854, 173.191956, 330.178955, 720.363708, -0.85639, -0.516329, 1164.522705, 1, dash_left
1905.375854, 173.191956, 330.178955, 720.363708, -0.911922, -0.410365, 1281.172363, 1, dash_left
1905.375854, 173.191956, 330.178955, 720.363708, -0.939616, -0.34223, 1389.492188, 1, dash_left
1905.375854, 173.191956, 330.178955, 720.363708, -0.87638, -0.481621, 1197.85083, 1, dash_left
1905.375854, 173.191956, 330.178955, 720.363708, -0.630922, -0.775846, 989.556152, 1, dash_left
1905.375854, 173.191956, 330.178955, 720.363708, 0.072274, -0.997385, 806.273438, 1, dash_left
197.777222, 1877.073486, 316.053101, 689.757812, -0.921893, -0.387445, 368.059235, 1, dash_right
197.777222, 1877.073486, 316.053101, 689.757812, -0.882891, -0.469577, 476.303375, 1, dash_right
197.777222, 1877.073486, 316.053101, 689.757812, -0.801978, -0.597354, 592.910522, 1, dash_right
197.777222, 1877.073486, 316.053101, 689.757812, -0.588615, -0.808414, 726.199036, 1, dash_right
197.777222, 1877.073486, 316.053101, 689.757812, -0.150345, -0.988634, 851.169861, 1, dash_right
197.777222, 1877.073486, 316.053101, 689.757812, 0.260984, -0.965343, 942.820068, 1, dash_right
197.777222, 1877.073486, 316.053101, 689.757812, 0.662648, -0.748931, 1076.134277, 1, dash_right
197.777222, 1877.073486, 316.053101, 689.757812, 0.755623, -0.655007, 1134.460815, 1, dash_right
197.777222, 1877.073486, 316.053101, 689.757812, 0.755623, -0.655007, 1134.460815, 1, dash_right
197.777222, 1877.073486, 316.053101, 689.757812, 0.900867, -0.434096, 1334.441895, 1, dash_right
1442.730713, 596.496216, 858.729919, 202.099976, -0.228244, 0.973604, 425.493835, 0, shoot_right
1442.730713, 596.496216, 858.729919, 202.099976, -0.100612, 0.994926, 455.089264, 0, shoot_right
1442.730713, 596.496216, 858.729919, 202.099976, 0.286507, 0.958078, 562.179688, 0, shoot_right
1442.730713, 596.496216, 858.729919, 202.099976, 0.92795, 0.372706, 668.543152, 0, shoot_right
1442.730713, 596.496216, 858.729919, 202.099976, 0.852267, 0.523107, 832.080566, 0, shoot_right
1442.730713, 596.496216, 858.729919, 202.099976, 0.587014, 0.809577, 837.446045, 0, shoot_right
1442.730713, 596.496216, 858.729919, 202.099976, 0.988469, -0.151426, 1072.617188, 0, shoot_right
1442.730713, 596.496216, 858.729919, 202.099976, 0.946537, 0.322595, 1215.950562, 0, shoot_right
596.496216, 1442.730713, 858.729919, 202.099976, -0.940071, 0.34098, 337.925568, 0, shoot_left
596.496216, 1442.730713, 858.729919, 202.099976, -0.978441, 0.206529, 367.176636, 0, shoot_left
596.496216, 1442.730713, 858.729919, 202.099976, -0.962502, 0.271275, 348.288391, 0, shoot_left
596.496216, 1442.730713, 858.729919, 202.099976, -0.93387, 0.357612, 337.34964, 0, shoot_left
596.496216, 1442.730713, 858.729919, 202.099976, -0.900104, 0.435675, 346.689636, 0, shoot_left
596.496216, 1442.730713, 858.729919, 202.099976, -0.863107, 0.505021, 374.795319, 0, shoot_left
596.496216, 1442.730713, 858.729919, 202.099976, -0.829367, 0.558703, 281.280823, 0, shoot_left
596.496216, 1442.730713, 858.729919, 202.099976, -0.898901, 0.438152, 230.956833, 0, shoot_left
596.496216, 1442.730713, 858.729919, 202.099976, -0.95267, 0.304007, 286.269104, 0, shoot_left
596.496216, 1442.730713, 858.729919, 202.099976, -0.986111, 0.166086, 395.257721, 0, shoot_left
1442.730713, 596.496216, 858.729919, 202.099976, -0.780657, 0.624959, 272.643402, 0, shoot_right
```

Ogni nodo interno rappresenta una domanda sulla variabile di input. I rami che si diramano da un nodo interno rappresentano le possibili risposte alla domanda. Il nodo foglia rappresenta la decisione finale che viene presa per un dato input.

Strumenti utilizzati

Anche in questo caso abbiamo utilizzato per la ricerca del percorso i moduli messi a disposizione da “AIPython: Python Code for AIFCA”. E anche in questo caso abbiamo avviato il

file learnDT.py attraverso lo script in Godot che gli passa l'array della lettura dello stato della stanza. Ritorna quindi la stringa che rappresenta il dizionario degli output.

Decisioni di Progetto

Lo stato della stanza è rappresentato da:

- La distanza del boss dai 4 muri (left_wall_dist, right_wall_dist, top_wall_dist, bottom_wall_dist)
- La distanza dal giocatore (target_dist)
- La direzione verso cui si trova il giocatore (rappresentata mediante x e y di un vettore direzionale normalizzato)
- La presenza di un proiettile in prossimità del boss (at_risk: valore booleano rappresentato da 0 e 1)

L'albero decisionale agisce come classificatore che identifica la situazione della stanza classificandola in una delle 5 possibili azioni che il boss può compiere (action):

- jump
- shoot_left
- shoot_right
- dash_left
- dash_right

Ogni volta che il boss avrà terminato di eseguire un'azione farà una nuova lettura della stanza, passando l'array dello stato come argomento allo script che ritornerà le coppie che associano le classi (azioni da svolgere) alle loro probabilità. Queste coppie vengono inserite in un dizionario e quella con il valore più alto viene svolta dal boss.

Valutazione

La metrica adatta a misurare le prestazioni dell'albero, nel nostro caso è l'accuracy. Questa è una metrica appropriata per la classificazione perché misura direttamente la capacità dell'albero decisionale di classificare i dati correttamente.

Abbiamo effettuato le misurazioni dell'accuratezza attraverso una cross-validation k-fold con k=10, riportando qui i nostri risultati. Per ottenere i dati abbiamo sfruttato la classe

K_fold_dataset messa a disposizione dallo strumento *AIPython*. Usando poi la funzione check_error() per ottenere l'array dei risultati. Gli esempi del nostro dataset erano 320.

Questa classe e questa funzione si trovano nel file learnCrossValidation.py

```
absolute_path = os.path.dirname(__file__)
relative_path = "./data/bossfight.csv"
full_path = os.path.join(absolute_path, relative_path)
data = Data_from_file(full_path, separator=',', target_index=-1)
accuracy_array = check_error(data, criterion=Evaluate.accuracy, maxx=11)

print(accuracy_array)
```

Accuracy k-fold (k=10)	
77.23214%	
80.80357%	
76.78571%	
77.23214%	
75.44643%	
70.98214%	
68.75000%	
70.98214%	
67.41071%	
66.07143%	
Media	73.16964%
Varianza	0.24689%
Deviazione Standard	4.96877%

Per il nostro caso d'uso la media di 73.17% è soddisfacente in quanto una accuratezza maggiore renderebbe il combattimento impossibile. La varianza relativamente bassa suggerisce che il modello è abbastanza stabile nelle sue prestazioni su diverse partizioni dei dati durante la cross-validation. Tuttavia, la deviazione standard indica che c'è una certa variabilità nei risultati, il che potrebbe indicare che il modello potrebbe beneficiare di un set di dati più ampio per una valutazione più accurata delle sue prestazioni.

Il funzionamento percepito in gioco è abbastanza soddisfacente il che si riscontra nei dati della misurazione.

Uno dei principali problemi di un albero decisionale è il rischio di overfitting, cioè il troppo adattamento ai dati di training e la perdita della capacità di generalizzazione. Un fenomeno di questo tipo probabilmente accade nei casi delle classi *dash_left* e *dash_right,* che tendono ad avere comportamenti a volte scorretti e vengo attivati soprattutto quando il boss si trova vicino alla posizione in cui molti dei casi di esempio di dash sono stati registrati.