Analisi progetto

Introduzione

Il progetto presentato consiste in una implementazione del teorema Minimax applicata al “Gioco dei bastoncini”.

Prima andare ad analizzare le caratteristiche del gioco andrò a definire Minimax e quali sono le caratteristiche dell’implementazione, i suoi limiti e possibili migliorie.

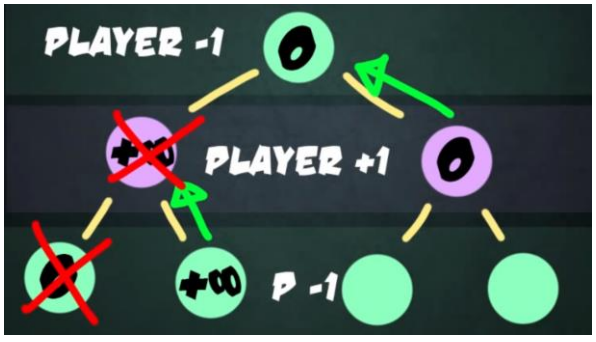
Minimax

Minimax è un teorema che si occupa di studiare come minimizzare la massima perdita possibile. L’applicazione del teorema come algoritmo permette, in un sistema di “gioco a somma zero” dove due giocatori possono avere turni simultanei o alternati, di stabilire quali mosse ti porteranno a vincere (o a minimizzare la sconfitta). Per realizzare questa previsione si affida ad un albero di cui ogni foglia ha N foglie per una profondità P, ove N sono i casi possibili per turno e P è la profondità dell’albero.

Ovviamente maggiore sarà la P più lungimiranti saranno le scelte fatte dall’algoritmo. Ogni foglia conterrà uno stato euristico del gioco e un riferimento al giocatore che può eseguire quella determinata azione.

L’algoritmo per prevedere quali mosse avranno un impatto meno negativo sul gioco scorre l’albero, cambiando il riferimento del giocatore ad ogni P, cercando il valore euristico che tenderà di più al proprio infinito.

Prendiamo come esempio l’albero in figura



Nell’immagine proposta immaginiamo di aver un gioco a turni e che i giocatori siano rappresentati matematicamente con i valori -1 e 1.

Tenendo conto dell’albero delle possibilità l’obbiettivo del giocatore -1 sarà sempre cercare il valore che tenda più velocemente a -infinito.

Nell’immagine il primo turno il player +1 sceglie il proprio infinito, mentre nel turno di -1 viene scelto 0 perché tra +infinito e 0 e quello che tende di più a -infinito.

Verso la fine del gioco è facile capire quali sono le mosse migliori; l'algoritmo minimax trova la mossa migliore in un dato momento cercandola a partire dalla fine del gioco e risalendo verso la situazione corrente. Ad ogni passo l'algoritmo assume che il giocatore 1 cerchi di massimizzare le sue probabilità di vincere, mentre -1 cerchi di minimizzare le probabilità di vittoria di 1, per esempio massimizzando le proprie chance di vittoria.

Limitazioni di minimax

Prevedere le mosse che possono portare a vincere non fa necessariamente vincere

Sebbene l’algoritmo cerchi di prevedere quali mosse fare per ridurre al minimo la perdita, ad ogni P possono esserci più di un N con un valore accettabile. Questo aspetto gli impedisce di avere un tasso di vittoria del 100%.

Dal debug del gioco:

I'm going to search for the nearest inf value from each children at any depth, then I'll return root's child in which is contained

[realVal nearest to inf 2147483647 (abs(MAXINT*playerNumber - bestValue)=0) has been retrived from depth 2 for player 1]

…

[realVal nearest to inf 2147483647 (abs(MAXINT*playerNumber - bestValue)=0) has been retrived from depth 6 for player 1]

[[possible bestChoice found: 1]]

[realVal nearest to inf -2147483647 (abs(MAXINT*playerNumber - bestValue)=0) has been retrived from depth 3 for player -1]

….

[realVal nearest to inf 2147483647 (abs(MAXINT*playerNumber - bestValue)=0) has been retrived from depth 6 for player 1]

[[possible bestChoice found: 2]]

Come è possibile riscontrare, ad un certo punto, è possibile selezionare sia 1 che due per vincere, tuttavia il pc selezionerà 2 e perderà poche mosse dopo.

Questa casistica compare per un numero di bastoncini pari a 11, selezionando la combinazione 2,1,1,1.

Tempo di processione dei nodi

Realizzare delle funzioni che scorrano/generino ricorsivamente per una certa P un certo numero di N, ha il suo impatto sia in termini di RAM che di CPU. Basti pensare che per una P=10 per N=6 viene generato un albero da 580MB. Quindi visto il peso in termini di RAM per prevedere solo le prossime 10 mosse su 6 alternative diventa impegnativo realizzare un albero per situazioni complesse senza allocare un quantitativo di memoria esagerato per il programma o senza far generare parzialmente l'albero.

Possibili soluzioni?

Si potrebbe prendere in considerazione di generare una generazione parziale dell'albero.

Tenendo conto che per la P=P(Max), ove P(Max) è la radice, venga generata solo una casistica e per i successivi 0<=P<P(Max) tutte le casistiche. In questo modo si eviterebbero eccezioni in lettura su memoria dovute al superamento della memoria allocabile. Trovato lo stato euristico più vantaggioso per il primo caso si potrebbe procedere con elaborare gli altri N per P=P(Max). Se si impiegassero due thread si potrebbe persino, tramite i semafori, impedire che vengano generati altri nodi finché uno di N thread sta lavorando e far spegnere il semaforo nel caso in cui il thread N sia alla fase di pulitura dei propri Nodi.

Non è in grado di evolversi

Trovato un caso per cui l'algoritmo fallisce, fallirà in eterno per quella casistica particolare.

Possibili soluzioni?

Le uniche soluzioni sarebbero o tener traccia delle sconfitte ricevute in passato e verificarle con l'albero corrente in modo da evitare la stessa sconfitta o comunque combinarlo con un algoritmo di apprendimento. Ci si potrebbe collegare ad un mongo o a un file binario che abbia come indice l'hash della situazione in analisi e la serie dei valori euristici che hanno portato alla sconfitta e confrontarli con le scelte possibili attuabili dal Minmax ad una certa P. Se è disponibile più di una scelta, anche se il valore non necessariamente è quello che tende di più ad inf può esser preso in considerazione come possibile situazione.

Caratteristiche del gioco e regole

Regole

Il gioco ha regole molto semplici di per sé e permette di regolare una serie di parametri.

L'obbiettivo è di ridurre esattamente a 0, prima dell'avversario, il numero di bastoncini rimanenti (NS). Per raggiungere l'obbiettivo ad ogni turno sarà possibile rimuovere un numero variabile di bastoncini (NR).

In ogni altro caso si perde.

Le condizioni sono:

- NS>=11 (11 è una costante per indicare il numero minimo di bastoncini totali)
- 2<=NR<NS (2 è una costante per indicare il numero minimo di bastoncini rimuovibili per turno)

Altri parametri configurabili sono: debug (se si vuole sapere cosa sta facendo il pc in background) e IA_LEVEL che indica quanto debba esser profondo l'albero.

Fasi di processione

1) Input dell'utente
2) Verifica lo stato del gioco (**checkGameEnd**)
3) Crea il nodo con le info correnti
4) Crea i Nodi per la root sopra creata (**createChildren**)
5) Viene cambiato il giocatore corrente e calcolata la mossa contromossa con perdita minore (**calcChoice**)
6) Vengono deallocati tutti i nodi (**freeAllNodes**)
7) Verifica lo stato del gioco (**checkGameEnd**) e cambia il turno del giocatore.

Quasi ogni funzionalità del gioco si basa sulla ricorsività e su puntatori. Il fatto di realizzare funzioni ricorsive permette di applicare la stessa logica su strutture dinamiche di cui non si conosce necessariamente l'entità o l'estensione in termini di lunghezza o profondità. Per quanto riguarda i puntatori permettono di passare i riferimenti in modo comodo attraverso funzioni ricorsive, per non parlare del fatto che permettono di realizzare alberi o vettori di puntatori di cui si conosce solo a runtime la dimensione.

Per quanto detto fino ad ora, per le fasi sopra citate, vale la pena analizzare solo quelle in cui sono presenti le funzioni:

- `createChildren`
- `calcChoice`
- `freeAllNodes`

CreateChildren

Come detto prima la funzione createChildren crea per una profondità P un numero di nodi N, pari al numero di casistiche. P ed N sono contenute dentro la struttura **MinMaxTreeConf** rispettivamente come **depth** e **numberOfSticksRemovable.** Per evitare di perdere il riferimento con il primo elemento all'interno della funzione ricorsiva e per comodità (dopo tutto dovendo allocare uno spazio di memoria bisogna comunque lavorare con un puntatore, per non parlare del fatto che la prop. children è un **) è stata implementata una soluzione che prevedere il passaggio per riferimento della radice dell'albero come parametro attuale. Questo permette di crearci sopra direttamente i figli e ricorsivamente di inserire nel **children**** i rifermenti dei figli.

Per necessità viene passata in ogni funzione ricorsiva anche la struttura MinMaxTreeConf. Il suo utilizzo diventa fondamentale per sapere il range per il quale esaminare verticalmente ed orizzontalmente l'albero poiché la sizeOf di un vettore di puntatori risulterà sempre uguale a quella di una singola istanza, rendendo quindi impossibile saperne l'estensione a monte.

Da notare inoltre come ogni singolo nodo tenga traccia dell'ipotetico stato che verrebbe a formarsi se fosse rimosso un determinato numero di bastoncini. Questo valore poi verrà moltiplicato per maxInt*+-remainingStiksAfterAtLeastOneIsPickedUp o verrà posto a 0 in base al valore di remainingStiksAfterAtLeastOneIsPickedUp. Questo permetterà di determinare lo stato euristico del nodo impiegato dall'algoritmo minimax per prendere decisioni.

```
void createChildren(Node *currentNode, MinMaxTreeConf minMaxTreeConf) {
        if (currentNode->depth >= 0) {
                currentNode->children = malloc(minMaxTreeConf.numberOfSticksRemovable * sizeof(Node));
                for (int i = 0;i < minMaxTreeConf.numberOfSticksRemovable;i++) {
                        int remainingStiksAfterAtLeastOneIsPickedUp = currentNode->remaningStriks - (i+1);
                        int playerNumber = -currentNode->playerNumber;
                        Node *child=malloc(sizeof(Node));
                        child->depth = currentNode->depth - 1;
                        child->gameState = getRealVal(remainingStiksAfterAtLeastOneIsPickedUp, playerNumber);
                        child->remaningStriks = remainingStiksAfterAtLeastOneIsPickedUp;
                        child->playerNumber = playerNumber;

                        createChildren(child, minMaxTreeConf);
                        (Node*)currentNode->children[i] = (Node*)child;
                }
        }
}
```

calcChoice

Questa funzione si occupa di calcolare qual è la scelta che tende di più a -inf e restituire il numero di bastoncini associati. Per far ciò vengono forniti la radice contenente lo stato precedente e il giocatore attuale. Lo step successivo consiste nel passare queste info e cercare per tutti i child contenuti in children i miniMax. Infatti il minimax viene calcolato ricorsivamente per ogni child grazie alla funzione calcMinMax.

```
int calcChoice(Node rootWithPreviusGameState,int currentPlayerNumber,MinMaxTreeConf minMaxTreeConf) {
        int playerNumber= currentPlayerNumber, depth=rootWithPreviusGameState.depth, nodeValue =
rootWithPreviusGameState.gameState,
                bestValue = MAXINT * -playerNumber,
                bestChoice = MAXINT * -playerNumber;
        printf("\n\nI'm going to search for the nearest inf value from each children at any depth, then I'll
return root's child in which is contained");
```

```
        for (int i = 0, x = minMaxTreeConf.numberOfSticksRemovable;i < x;i++) {
                Node child = getNodeValueFromChildren(rootWithPreviusGameState, i);
                nodeValue = calcMinMax(child, depth, -playerNumber, minMaxTreeConf);
                if (abs(MAXINT*playerNumber - nodeValue) <= abs(MAXINT*playerNumber - bestValue)) {
                        bestValue = nodeValue;
                        bestChoice = i+1;
                        if (minMaxTreeConf.debugEnabled == 1) {
                                printf("\n\n[[possible bestChoice found: %i]]\n\n", bestChoice);
                        }
                }
        }
        printf("\n\n[bestChoice %i with value realVal %i (abs(MAXINT*playerNumber - bestValue)=%i) has been
retrived from depth %i for player %i]\n\n", bestChoice, bestValue, abs(MAXINT*playerNumber - bestValue),
depth, currentPlayerNumber);
        return bestChoice;
}
```

```
int calcMinMax(Node node,int depth,int playerNumber, MinMaxTreeConf minMaxTreeConf) {
        int nodeValue=node.gameState,bestValue;
        if ((node.depth == 0) || (abs(nodeValue) == MAXINT)) {
                return nodeValue;
        }
        bestValue = MAXINT * -playerNumber;
        for (int i = 0, x = minMaxTreeConf.numberOfSticksRemovable;i<x;i++) {
                Node child = getNodeValueFromChildren(node, i);
                nodeValue = calcMinMax(child,depth-1,-playerNumber,minMaxTreeConf);
                if (abs(MAXINT*playerNumber - nodeValue) < abs(MAXINT*playerNumber - bestValue)) {
                        bestValue = nodeValue;
                }
        }
        if (minMaxTreeConf.debugEnabled == 1) {
                printf("\n[realVal nearest to inf %i (abs(MAXINT*playerNumber - bestValue)=%i) has been
retrived from depth %i for player %i]", bestValue, abs(MAXINT*playerNumber - bestValue), depth,
playerNumber);
        }
        return bestValue;
}
```

Da notare:

1) il primo if funge da condizione di uscita per la funzione ricorsiva e per evitare inutili ricorsioni;

2) ad ogni ricorsione viene cambiato l'user in modo da simulare i possibili scenari futuri.

FreeAllNodes

Tra tutte è la funzione più semplice; cicla ricorsivamente sopra i nodi per liberare ogni porzione di memoria assegnata ad ogni puntatore.

```
void freeAllNodes(Node *currentNode, MinMaxTreeConf minMaxTreeConf) {
        if (currentNode->depth >= 0) {
                int x = minMaxTreeConf.numberOfSticksRemovable;
                for (int i = 0;i < x;i++) {
                        freeAllNodes((Node*)currentNode->children[i], minMaxTreeConf);
                        free(currentNode->children[i]);
                }
                free(currentNode->children);
        }
}
```