Tris Parametrico

Progetto di Esperienze di Programmazione

Ivan Sarno

Luglio 2016

Indice

Introduzione

Requisiti e Configurazione

Questo programma richiede JDK 1.8 o versione superiore.

Tutti i test sono stati eseguiti su un MacBook Air 2012, i5 1.6/2.7Ghz, 2core/4thread, 8GB ram a 1600Mhz, OS X 10.11.

Il gioco

Il Tris classico prevede una griglia 3x3 in cui i due giocatori a turno segnano una casella con un simbolo(X o O), vince il primo giocatore che riesce a segnare 3 caselle consecutive, su una riga una colonna o una diagonale col suo simbolo. Il gioco finisce in pareggio se nessun giocatore può fare una mossa.

Più formalmente è un gioco:competitivo, a turni, a due giocatori, a somma zero, deterministico, a informazione perfetta e statico; giocato in modo perfetto porta sempre a un pareggio.

In questa relazione e nel codice la lunghezza del lato della griglia è indicata come “size", il numero di caselle consecutive come “serie”.

La variante esaminata in questo progetto prevede che la dimensione della griglia e la serie siano parametriche. Il gioco mantiene le caratteristiche dell’originale, ma per serie >= size/2 (+1 per size dispari) esiste una strategia per cui il primo giocatore vince sempre.

Ho scelto questo gioco per la possibilità di essere scalato facilmente per aumentare progressivamente la difficoltà e sperimentare soluzioni algoritmiche e implementative differenti. L'obiettivo del progetto è giocare su una griglia 10x10 con una serie di 4.

Minimax

Minimax è un algoritmo studiato per i giochi competitivi a 2 giocatori che calcola la mossa ottimale a partire da una configurazione, dato uno stato iniziale l'algoritmo genera tutti i successori e restituisce il successore col valore più alto (o la mossa che lo ha generato).

Il valore di uno stato è dato dalla funzione valoreMax, che restituisce:

1. il punteggio che il gioco attribuisce allo stato, se lo stato è terminale
2. Il massimo valore della funzione valoreMin calcolata sui successori, altrimenti

La funzione valoreMin è speculare:

1. il punteggio che il gioco attribuisce allo stato, se lo stato è terminale
2. Il minimo valore della funzione valoreMax calcolata sui successori, altrimenti

Le chiamate ricorsive di queste 2 funzioni generano idealmente un albero in cui ogni nodo è uno stato, i cui figli sono i successori.

L'idea è scegliere la mossa migliore nel proprio turno e simulare un turno dell'avversario in cui sceglie la mossa peggiore per se.

Varianti

La complessità di Minimax è b^d dove d è la profondità dell'albero di esplorazione e b il numero di successori di ogni nodo. In una griglia 10x10 b = 100 alla prima mossa e diminuisce di 1 ad ogni livello dell'albero e di 2 ogni mossa, d è uguale al caso pessimo, ma spesso è minore perché esistono molti stati terminali raggiungibili con poche mosse, se un giocatore commette un errore. Questo rende impraticabile Minimax.

L'algoritmo AlphaBeta è un'ottimizzazione di Minimax che non esplora i successori che non possono influenzare la scelta. 2 variabili, alpha e beta (inizialmente -infinito e +infinito) rappresentano la migliore e la peggiore scelta corrente, in base a queste si decide se tagliare il ramo che si sta esplorando:

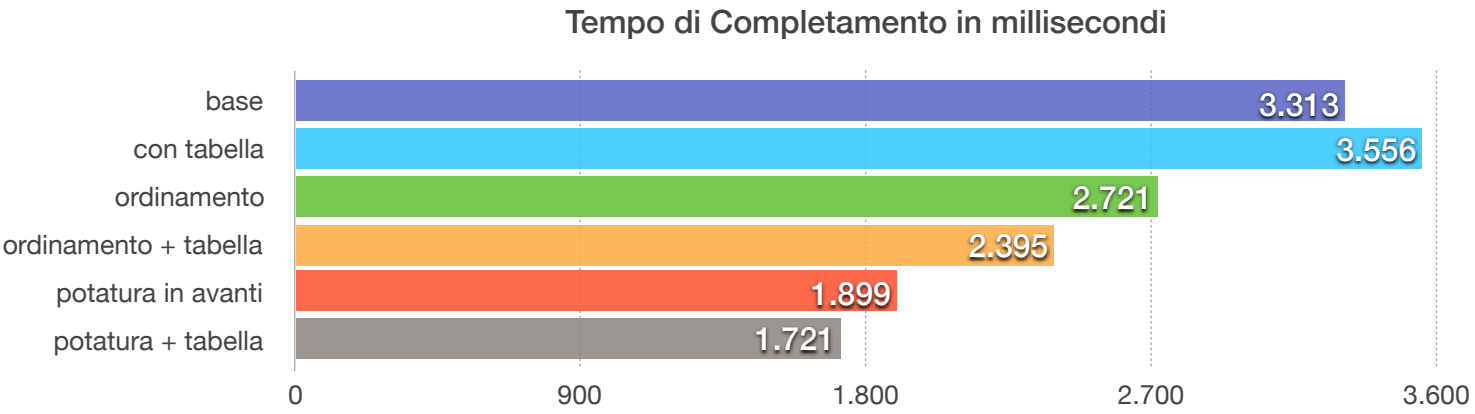
nel calcolo di valoreMax se si incontra un valore maggiore di beta si può interrompere l'esplorazione, perché comunque la funzione valoreMin che ha chiamato valoreMax scarterebbe quel ramo perché ne può scegliere uno con valore minore; altrimenti si aggiorna alpha. Il funzionamento di valoreMin è speculare.

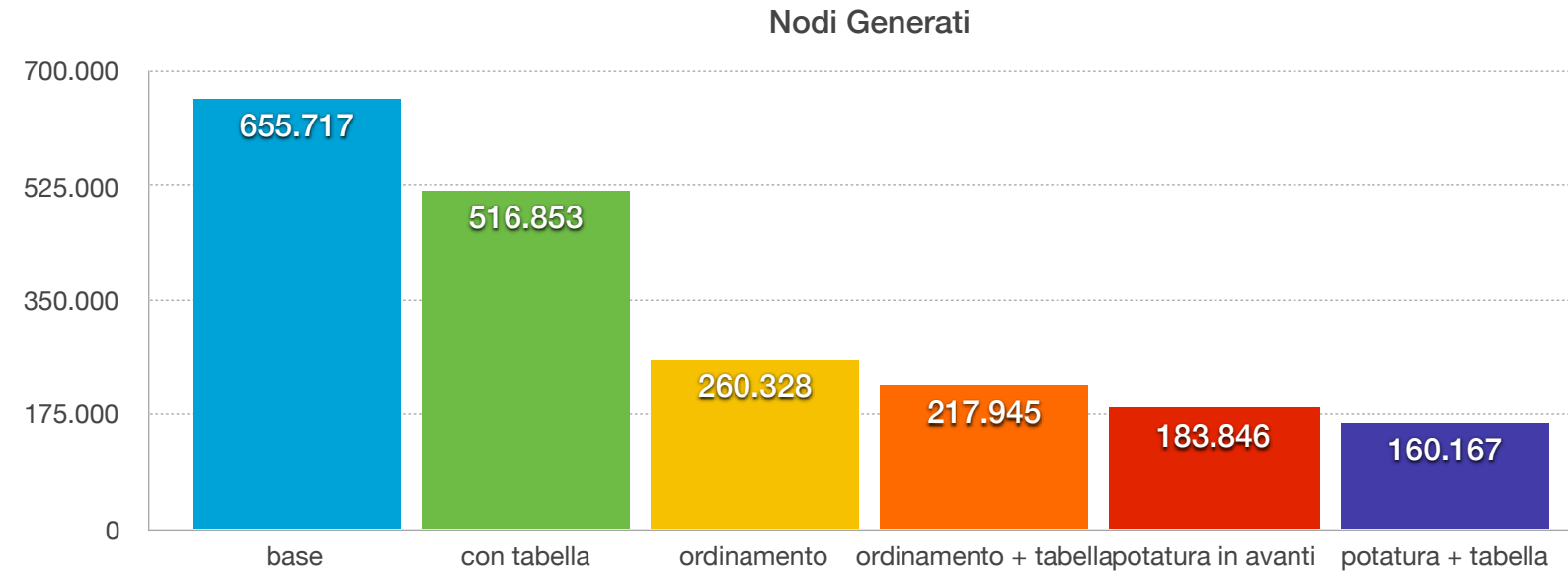
Possibili ottimizzazioni di AlphaBeta sono:

1. Terminazione anticipata: nei giochi a somma zero in cui tutti gli stati vincenti/perdenti hanno lo stesso valore, come il Tris, ad ogni livello è possibile interrompere la ricerca quando si incontra un valore minimo/massimo;
2. Ordinamento delle mosse: si ordinano le mosse in base a una valutazione euristica per aumentare la possibilità di taglio nei primi successori esplorati, riducendo b della metà;
3. Profondità limitata: si esplora fino ad una profondità fissata, se il nodo corrente non è terminale si restituisce una stima euristica del valore;

4. Tabella nodi esplorati: anche nei giochi che non presentano cicli, come il Tris, è possibile trovare la stessa configurazione in punti diversi dell'albero di minimax, per questo è utile conservare una tabella stato-valore in modo da non dover ricalcolare il valore degli stati già incontrati;
5. Potatura in Avanti: si esplorano solo i migliori k successori.

Le prime 2 tecniche non hanno effetto sulla qualità delle mosse o controindicazioni sulle performance, dunque vengono adottate all'inizio.





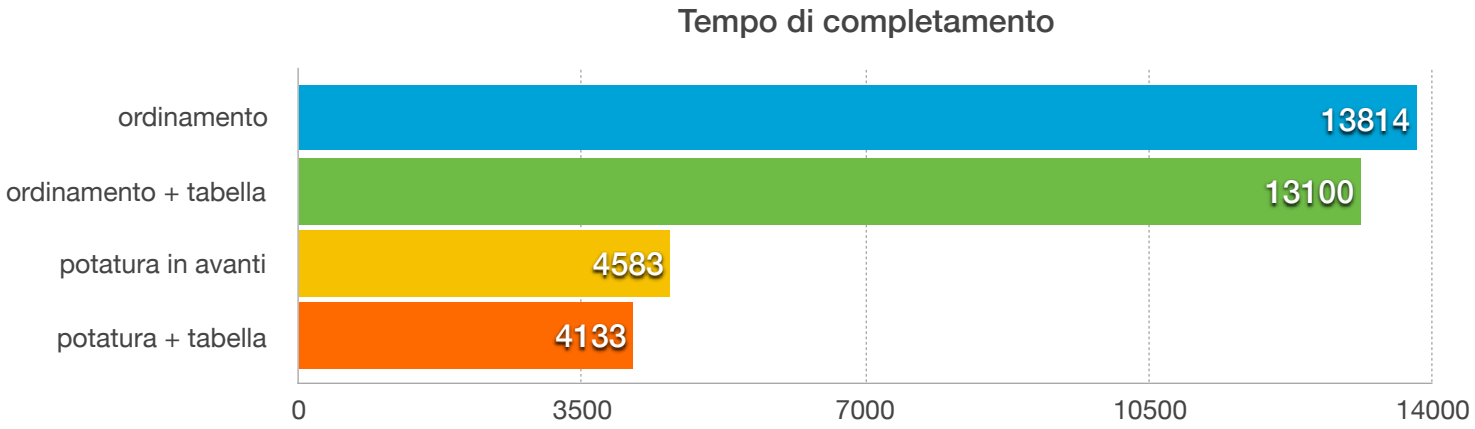
Il test consiste nel calcolare la prima mossa su una griglia vuota, scegliere la mossa migliore con l'avversario che inizia con uno dei 4 angoli o il centro.

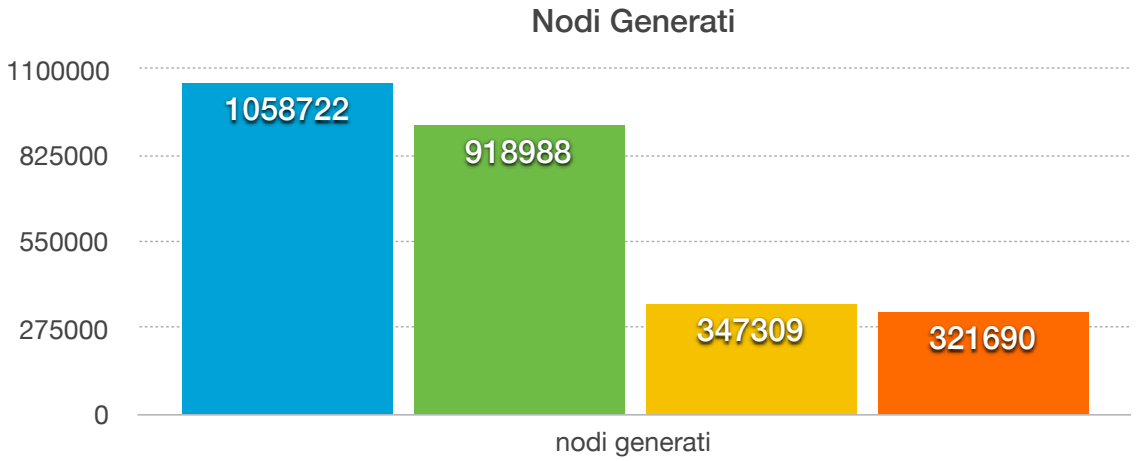
I parametri sono: size = 5, serie = 3, profondità = 4, numero di successori massimo per la potatura in avanti = 20. I risultati sono la media su 30 iterazioni.

Si nota subito che l'ordinamento permette una forte riduzione dei nodi esplorati, senza perdere soluzioni utili, e un buon vantaggio sul tempo di completamento.

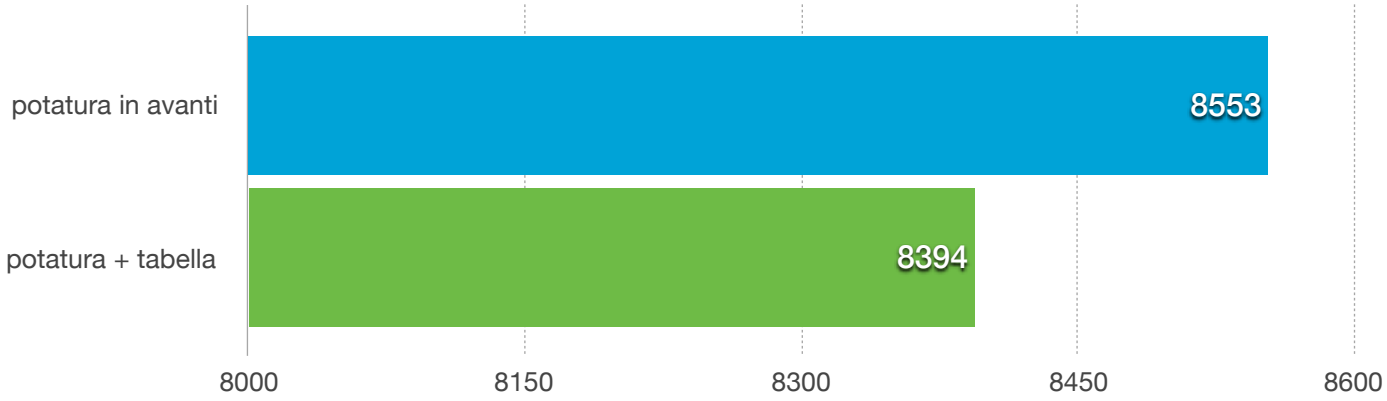
La potatura non sembra offrire grandi vantaggi, ma vengono esplorati l'80% dei successori.

Aumentando size a 6 e serie a 4, sempre con il limite di 20 successori, che ora corrisponde al 55% la situazione è:

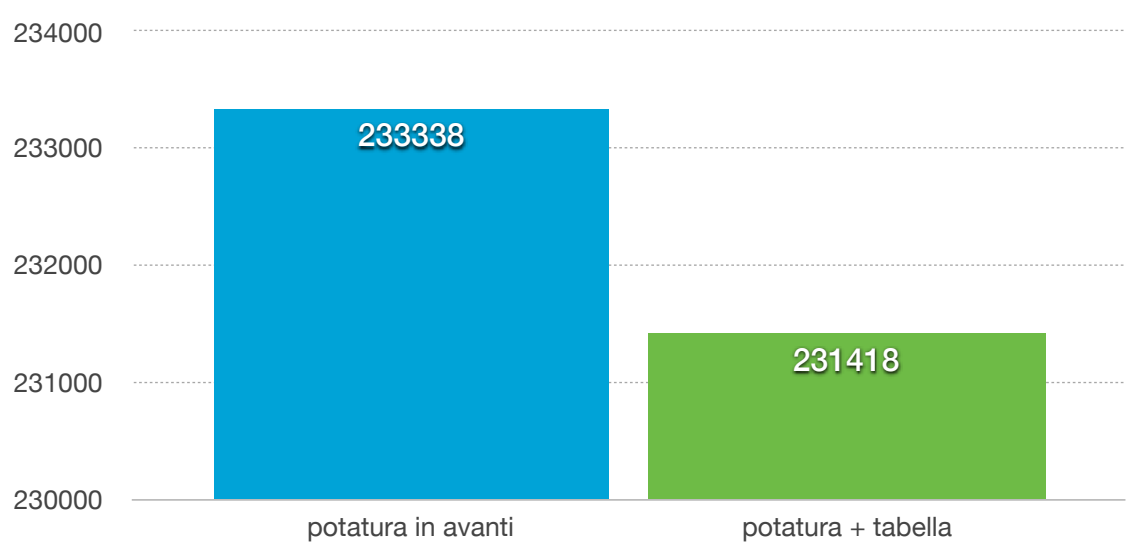




Inoltre esplorando almeno il 30% dei successori si è abbastanza sicuri di non perdere soluzioni utili. Dunque ho scelto di adottare la potatura in avanti.



L'efficacia della tabella è limitata e dipende molto dai cammini esplorati, ma nel caso peggiore non costituisce uno svantaggio.



Parametri test: serie 4, size 10, profondità 4, successori 10, iterationi 10.

Dunque scelgo l'algoritmo AlphaBeta con potatura in avanti e tabella dei nodi esplorati.

MiniMax, e le sue varianti, assume di giocare contro un avversario perfetto, dunque quando conclude che non ci sono strategie vincenti, cioè il valore di tutti i successori della radice hanno tutti il valore minimo, restituisce una mossa a caso(la prima); ma è comunque utile fare la migliore mossa possibile sperando che l'avversario sbagli. Per ottenere questo a parità di valore si restituisce la mossa con valutazione euristica migliore.

Euristica

Cercando in rete un'euristiche per il Tris, ho trovato un esempio didattico da usare come base per sviluppare un euristica per la variante parametrica. Questa euristica considera il numero di colonne, righe e diagonali su cui il giocatore può ancora vincere, e vi sottrae quelle dell'avversario.

Questa euristica oltre a non adattarsi al caso generale, perché assume che la serie e la dimensione della matrice siano uguali, ha un importante difetto.

Supponiamo di avere questa configurazione in cui X è l'avversario:

X X _

O _ _

_ _ _

Considerando solo il valore delle righe il programma stima questa configurazione come 2 righe su cui può vincere –2 righe su cui vince l'avversario, col valore di 0; alla mossa successiva 2 possibili successori sono:

X X O

O _ _

_ _ _

X X _

O _ _

_ O _

Entrambi hanno una valutazione pari a 1, ma la seconda porta a una sconfitta al prossimo turno.

È necessaria una valutazione più specifica e accurata. Definiamo una sequenza come un intervallo (su righe, colonne o diagonali) di caselle comprese tra due simboli dell'avversario o il bordo, la lunghezza della sequenza è il numero di caselle, il valore è il numero di simboli del giocatore. Una sequenza è significativa se la sua lunghezza è maggiore o uguale a quella della serie, altrimenti il suo valore è 0.

La funzione di valutazione è la somma dei valori di tutte le sequenze del giocatore meno quelle dell'avversario.

Questa funzione è specifica per il Tris parametrico, ma soffre dello stesso difetto di quella precedente. Per evitare che il programma privilegi tante sequenze piccole rispetto a poche lunghe, il valore di una sequenza è dato dal quadrato del numero di simboli.

Questa euristica offre una stima accurata del valore di una configurazione, ma ha ancora un difetto, esistono configurazioni in cui aumentare il valore delle proprie serie o aggiungerne nuove rispetto a spezzare quelle dell'avversario offre benefici maggiori nella valutazione, ma ovviamente peggiori nell'utilità di gioco.

Per evitare questo fenomeno le sequenze dell'avversario devono essere elevate ad un esponente maggiore di 2; elevato a 3 si ottiene un algoritmo che gioca bene in attacco, ma potrebbe trascurare qualche trappola dell'avversario, elevato a 4 si ottiene un algoritmo sulla difensiva che perde soltanto in casi di sconfitta certa, ma che potrebbe portare ad un numero eccessivo di pareggi; 3.6 è un buon compromesso anche considerando che in una partita perfetta o si pareggia o vince l'avversario.

Nella variante di MiniMax che tenta di giocare meglio possibile anche in caso di sconfitta sicura è necessaria un euristica che che porti l'algoritmo a diminuire le possibili mosse vincenti dell'avversario anche se non può neutralizzarle completamente. Ad esempio consideriamo una configurazione con serie = 3, size = 5

_ _ _ _ _

_ _ _ O _

_ G X X _

_ _ _ _ _

_ _ _ _ _

Qualsiasi sia la mossa scelta dall'algoritmo il valore delle sequenze dell'avversario non può essere alterato, ma comunque la mossa migliore è muovere nella casella segnata con "G", cioè se non si riesce a diminuire il valore di una serie si deve diminuire la sua lunghezza. Questo si ottiene moltiplicando il valore della sequenza per lunghezza.

Ricapitolando, il valore di una configurazione è dato dalla somma dei valori delle sequenze del giocatore meno la somma dei valori di quelle dell'avversario; una sequenza è l'intervallo (su righe, colonne o diagonali) di caselle comprese tra due simboli dell'avversario o il bordo;

il valore di una sequenza del programma è numero_di_simboli ^ 2 * lunghezza_della_sequenza; il valore di una sequenza dell'avversario è ((numero_di_simboli ^ 3.6) * lunghezza_della_sequenza).

Rappresentazione dello stato

Una configurazione di gioco è espressa da un oggetto TrisState, esso contiene una matrice di byte che codificano con 0 la casella vuota, con 1 una casella occupata da un simbolo del programma(Max) e con -1 una casella occupata da un simbolo dell'avversario.

Altri membri sono:

- heuristicValue: valore euristico della configurazione
- value: valore esatto della configurazione, è significativo solo se lo stato è terminale, 0 esprime uno stato di pareggio.
- isTerminal: true se il nodo è terminale, false altrimenti

Metodi:

- toString: produce la rappresentazione stampabile della griglia
- comparatorMin/comparatorMax: funzioni di comparazione per le funzioni evalMin e evalMax
- reset: riporta uno stato alla configurazione di default o la copia da un altro stato
- revalue: ricalcola isTerminal, heuristicValue e value, o li copia da un altro stato

Membri statici:

- size: dimensione della matrice.
- serie: lunghezza della serie necessaria per vincere
- maxValue/minValue: il massimo/minimo valore che uno stato può assumere, calcolato in funzione della dimensione della griglia in modo da essere sempre maggiore/minore dei possibili valori euristici. maxValue = size^6 * 10, cioè almeno 10 volte superiore al valore della griglia coperta da un solo simbolo; minValue = -maxValue.
- Init: metodo statico che inizializza i membri statici.

Altri metodi: sotto-funzioni per il calcolo dell'euristica o della condizione di terminazione, funzioni per copia, sovrascittura e azzeramento di matrici, equals, hashCode.

Algoritmo

La versione di AlphaBeta usata in questo progetto è implementata dalla classe Engine, essa possiede 3 campi, la profondità massima maxDepth, il numero massimo di successori esplorati per nodo maxElements, e una HashMap per la tabella dei nodi esplorati explored.

Inoltre ha 5 metodi:

1. evalMax: calcola il valore di utilità nella fase max.
    a. verifica le condizioni di terminazione, se il nodo è terminale, è presente nella tabella o la profondità massima è stata raggiunta restituisce il valore del nodo; altrimenti
    b. genera i successori con il metodo successorsMax e per ogni successore

    c. valuta la funzione evalMin sul successore corrente
    d. aggiorna alpha con il massimo tra alpha e il valore del successore corrente
    e. se alpha supera beta, aggiunge il nodo alla tabella con alpha come valore e restituisce alpha
    f. quando i successori sono terminati restituisce alpha, il massimo valore trovato e aggiunge il nodo alla tabella con alpha come valore.
2. evalMin: è speculare ad evalMax.
3. successorsMax: genera i successori del nodo corrente che rappresentano mosse di Max
    a. cerca il primo 0 nella matrice, una casella libera
    b. copia la matrice del nodo e sostituisce lo 0 con un 1
    c. crea un nuovo TrisState a partire dalla nuova matrice e lo aggiunge alla lista dei successori (ArrayList)
    d. ordina la lista dei successori con il comparatore comparatorMax.
4. SuccessorsMin: come successorsMax ma genera le mosse di Min, cioè sostituisce gli 0 con -1 e usa il comparatore comparatorMin.
5. nextState: restituisce la mossa migliore partendo da uno stato iniziale, è identico ad evalMax, ma invece che il valore massimo restituisce il nodo col valore massimo, o con valore euristico massimo a parità di valore, prima di terminare cancella inoltre la tabella dei nodi esplorati.

Ottimizzazioni

Questo paragrafo presenta delle ottimizzazioni implementative che aumentano sensibilmente le performance dell'algoritmo.

Parallelismo

La valutazione dei nodi della radice si presta ad essere parallelizzata, perché la valutazione di ogni nodo è indipendente, bisogna solo fermarsi se è stato trovato un cammino ottimo.

Per la gestione dei thread ho usato l'API Stream di Java 8, che gestisce automaticamente la distribuzione del carico e il numero di thread in base alla macchina.

Sono necessari pochi cambiamenti al programma; oltre a sostituire tutte le strutture globali con la loro variante concorrente, è necessario implementare la terminazione anticipata, che non è possibile avere in maniera nativa con gli Stream, che non hanno un meccanismo di cancellazione.

Per far questo la prima chiamata a evalMin viene incapsulata in una routine, viene aggiunto un booleano termination settato a false all'inizio di ogni calcolo di mossa; la prima routine che raggiunge un valore massimo setta termination a true, ogni altra routine all'inizio testa

termination se è true ritorna -infinito, evalMin ed evalMax testano termination, se true lanciano un eccezione, la quale viene catturata dalla routine(primo livello) che restituisce -infinito. Non è necessario garantire la mutua esclusione sulla variabile termination dato che se più thread restituiscono un risultato diverso da -infinito, comunque viene scelto il valore massimo.

L'efficacia del parallelismo dipende molto dal numero dei successori e dalla probabilità di interruzione, più sono pochi i successori e frequenti le interruzioni, più le performance sono simili a quelle sequenziali. Si hanno miglioramenti compresi tra il 20% e il 60%; in ogni caso non costituisce uno svantaggio.

Generazione efficiente dei successori

L'algoritmo genera tutti i nodi successori di cui la maggior parte vengono immediatamente scartati, è possibile risparmiare un significativo numero di nodi riutilizzandoli.

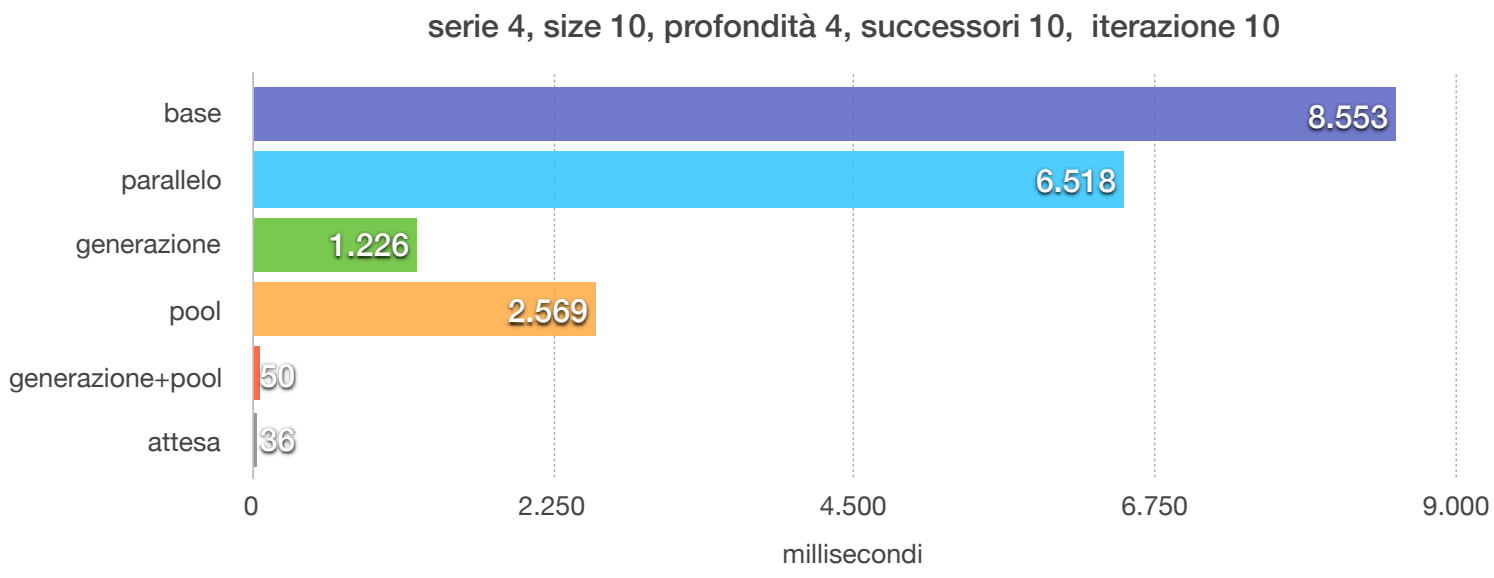
Dunque l'algoritmo genera il numero massimo di nodi richiesti, poi confronta con il nodo peggiore i successivi nodi generati, se il nodo verrebbe scartato basta rimettere a 0 la posizione corrente e continuare utilizzando la stessa matrice, altrimenti il nodo viene aggiunto ai successori; è anche possibile estrarre il nodo peggiore e resettarlo, mantenendo costante il numero dei successori. Questo si implementa facilmente con una coda a priorità.

Object pool

L'algoritmo genera migliaia di nodi che vengono deallocati ad ogni iterazione, con costi per una nuova inizializzazione e carico del Garbage Collector. Si possono gestire le allocazioni con un object pool, rappresentata dalla classe TrisPool; esistono 2 approcci, rilasciare gli stati appena non più utilizzati oppure permettere alla prima iterazione di allocare tutti i nodi di cui ha bisogno (di solito questa genera il numero maggiore di nodi) tenerne traccia nella pool e alla fine dell'iterazione dichiararli liberi e utilizzarli per la prossima. Il secondo approccio si rivela migliore, infatti è più semplice da gestire, con poco lavoro aggiuntivo per l'algoritmo.

Attesa Attiva

È possibile allocare all'avvio del programma una quantità di nodi, che dipende dalle dimensioni della matrice, e usare un metodo separato per il reset della pool. Queste operazioni possono essere effettuate nel frattempo che l'avversario pensa la sua mossa, sottraendo il loro costo dal tempo di completamento del calcolo di una mossa, in modo da dare all'utente una maggiore reattività.



Interfaccia

In questo progetto l'interfaccia non è stata presa in grande considerazione, dato che il suo scopo è solo quello di mostrare il funzionamento dell'algoritmo di gioco.

È dunque è presente un'interfaccia testuale spartana.

Essa si limita a chiedere all'utente la grandezza della griglia e la lunghezza della serie, il metodo getEngine restituisce un algoritmo di gioco configurato adeguatamente per questi due parametri, poi avvia un loop in cui il programma chiede all'utente le coordinate della casella da segnare, il programma fa la sua mossa, stampa la griglia, composta da X, O e _ per lo spazio vuoto, fino al termine del gioco, poi stampa l'esito.

Il programma stampa inoltre delle statistiche per ogni mossa: tempo di completamento, memoria allocare alla JVM (non corrisponde esattamente alla memoria usata dal programma), profondità di esplorazione, numero massimo di successori esplorati..

Il metodo getEngine restituisce un oggetto Engine inizializzato in modo da adattarsi alle dimensioni della matrice bilanciando accuratezza dell'esplorazione e prestazioni, i valori dei parametri sono stati ricavati da varie partite.

Conclusioni

La versione finale del programma gioca su una griglia 10x10 con serie da 4, con profondità di esplorazione 6 e un massimo di 10 successori esplorati per nodo, con in media un secondo per mossa.

Codice