Relazione progetto
di
Algoritmi e Strutture Dati

Università di Bologna

Daniele D'Ugo: 0001027741

Progetto per l'anno accademico 2022/2023
Corso di laurea in Informatica
Dipartimento di Informatica - Scienza e Ingegneria
Università di Bologna

1 Introduzione

Il progetto svolto propone l'implementazione di un giocatore per Connect-X.

1.1 Connect X

Connect-X è un gioco in cui, a turno, due giocatori devono inserire un gettone in una delle colonne della tabella (nella prima riga libera per tale colonna), finché uno non allinei X gettoni, o non si arrivi a un pareggio per mancanza di mosse.
Nel progetto, si chiamano M, N, X, rispettivamente, il numero di righe, colonne e gettoni da allineare per vincere.
Il famoso Connect-4 è una versione specifica di questo gioco, dove M, N, X valgono, rispettivamente, 6, 7, 4.

1.2 Il progetto

Il progetto richiede l'implementazione di un giocatore per Connect-X, nel linguaggio java, che implementi l'interfaccia fornita. Attraverso di essa, il player può dunque essere utilizzato in una partita, contro un umano o un'altra intelligenza artificiale.
Si pone un limite di tempo parametrico per ogni mossa.

1.3 Note e convenzioni sulla relazione

Alcune note valide per questa relazione e per l'implementazione:

1. i possibili stati per una casella sono P1 (occupata dal player 1), P2 (player 2), FREE (libera);
2. i possibili esiti di una partita sono WINP1 (vittoria per P1), WINP2 (per P2), DRAW (pareggio), OPEN (non conclusa);
3. ove non specificato, P1 rappresenta il proprio giocatore, quello che l'implementazione cerca di far vincere;
4. avendo fatto uso di algoritmi non visti a lezione, ovvero *PN-search* e *DB-search*, le relative sezioni (2.5.1, 2.5.2) li spiegano brevemente, parlando al tempo stesso della loro implementazione;
5. in grassetto sono indicati classi, metodi, variabili che implementano ciò di cui si sta parlando (a volte tra parentesi quadre).

2 La soluzione implementata

2.1 La scelta

Essendo il progetto inserito nel contesto di un corso universitario di algoritmi e strutture dati, e dati gli argomenti affrontati e noti agli studenti all'esecuzione del progetto, si è scelto di implementare il giocatore attraverso un algoritmo su alberi di gioco, di tipo iterativo in modo che possa restituire un risultato allo scadere del tempo.

Per appassionamento e curiosità per la tesi *Searching for solutions in games and artificial intelligence* (fonte 1), si è scelto di adattare al progetto la combinazione di *Proof-number search* - come algoritmo di ricerca su alberi di gioco - e *Dependency-based search* - algoritmo di ricerca di sequenze vincenti di mosse, usato come evaluation statica.
Si è inoltre fatto uso di *transposition table* (che per comodità chiameremo anche *TT*) e *bitboard*, fondamentali per la natura di tali algoritmi e per la riuscita dell'implementazione.

2.2 Costanti e convenzioni preliminari

Come si vedrà in seguito, per ottimizzare lo spazio e facilitare l'accesso alle variabili, molte strutture sono state implementate personalmente. A tale scopo, si è sempre cercato di utilizzare la minima dimensione possibile per ogni variabile necessaria, per cui, nel progetto, si possono trovare:

- valori memorizzati come bit, con opportuni metodi e bitmask per l'accesso;
- array come bitstring (es.: *bitboard*);
- tipi di numeri più piccoli possibile: ad esempio, basandosi sulle specifiche del progetto, assumendo che *M,N* non superino 256, molti numeri sono di tipo *byte* (che basta a contenere a 256 valori), o di tipo *short* (che, con *16bit* e massimo 65.356 valori, può contenere, ad esempio, un'intera tabella *100*100*, da 10.000 valori);
- preferenza per tipi di base (es.: *int*) rispetto agli oggetti (es.: costanti piuttosto che *enum* di java).

2.3 La tabella (bitboard)

Prendendo spunto dalle fonti 2 e 3, il giocatore qui implementato utilizza, come tabelle, delle *bitboard* **[BoardBit]**, ovvero memorizza lo stato di una casella come un singolo bit.

2.3.1 Implementazione

Poiché un bit può contenere solo due valori, ma una casella si può trovare in tre stati diversi, l'implementazione della *board* contiene due matrici di bit:

- una "tabella principale" **[board]**, dove un bit vale 1 se la relativa casella è occupata da *P1*, 0 altrimenti;
- una "tabella-maschera" **[board_mask]**, dove un bit vale 1 se la relativa casella è occupata da un qualsiasi giocatore, 0 se è *FREE*.

Ogni sotto-tabella è implementata come array bidimensionale di *long*, per far entrare in una *bitstring* una colonna di più di *64* righe:

- la prima dimensione, che indica il numero della colonna, va da *0* a *N*;

- la seconda permette di rappresentare una colonna come sequenza di *long*: dunque, per rappresentare una colonna di *M* valori servono *ceil(M / 64)*, (arrotondamento per eccesso).

Una *board* può essere acceduta e modificata con operazioni sui bit.

Il controllo per la conclusione di una partita, invece, avviene esattamente come nel codice fornito nelle specifiche, ovvero controllando, a ogni mossa, massimo *X* valori intorno all'ultima pedina piazzata, in ogni direzione.

2.3.2 Vantaggi

Dal punto di vista della memoria, tale implementazione è molto compatta, e permette di tenere attivo un gran numero di *board* allo stesso tempo (requisito fondamentale per gli algoritmi usati).
Dal punto di vista computazionale, essendo la copia e l'inizializzazione lineari, risulta agevole crearne o duplicarne in abbondanza.

2.4 Proof-number search (PN-search)

La ricerca di tipo proof-number mira a dare una "dimostrazione" (*proof*) per un nodo dell'albero di gioco.

2.4.1 L'albero

L'albero usato (in realtà un *dag*, poiché un nodo può avere più padri) è un *AND/OR-tree*, cioè ogni suo nodo può avere i valori *true/false*, oppure *unknown*: un nodo *true* si dice *proved*, cioè dimostrato vero; uno *false disproved*.
Un nodo **[TTPnNode]** ha due numeri (*proof-number*) **[TTPnNode.n[]]** :

- *Proof* indica il minimo numero di nodi discendenti che devono essere dimostrati veri perché esso stesso lo sia;
- *Disproof* indica lo stesso, ma per renderlo *disproved*.

Nell'implementazione, *true* è associato a *P1* (e *false* a *P2*); inoltre, essendoci solo due valori, si indica il pareggio come vittoria per *P2*.

[TTPnNode.updateProofAndDisproofOrProve()]
I *proof-number* di un nodo possono essere calcolati applicando la seguente regola ricorsivamente:

- per un nodo di *P1*, *Proof* è uguale al minimo *Proof* dei figli, mentre *Disproof* è uguale alla somma dei *Disproof* dei figli;
- per un nodo di *P2*, *Disproof* è il minimo e *Proof* la somma.

2.4.2 L'algoritmo

Prima di tutto, definiamo *espanso* un nodo che è stato analizzato dall'algoritmo. La radice, inizialmente, non è *espansa*. I nodi non *espansi* sono inizializzati con *proof number* di default.

[PnSearch.visit()]
Dunque la *PN-search* consiste nel seguente ciclo, interrotto dalla dimostrazione della radice o dallo scadere del tempo:

1. si cerca un nodo non *espanso*, discendente della radice;

2. si *sviluppa* tale nodo, aggiornando i suoi *Proof-number* e generando i suoi figli (come non-*espansi*);
3. si aggiornano, ricorsivamente, i *Proof-number* dei suoi antenati, fino a tornare alla radice.

```
void visit() {
	TTPnNode most_proving_node;
	while(!root.isProved() && !isTimeEnded()) {
		most_proving_node = selectMostProving(root);
		developNode(most_proving_node);
		updateAncestors(most_proving_node);
	}
}
```

[PnSearch.selectMostProstProving()]
Esiste, di volta in volta, un nodo, detto *most-proving node*, capace di aggiornare i *proof number* della *root* prima degli altri. Si trova scendendo nell'albero, fino a una foglia, secondo questo criterio:

- da un nodo *OR*, si passa al figlio con lo stesso *Proof*;
- da un nodo *AND*, si passa al figlio con lo stesso *Disproof*.

[PnSearch.developNode()]
Lo sviluppo del *most-proving node* (non *espanso*) controlla se la partita sia finita **[PnSearch.evaluate()]**, o se si possa valutare subito la posizione con una DB-search **[PnSearch.evaluateDb()]**, altrimenti lo espande creandogli dei figli **[PnSearch.generateAllChildren()]**.

[PnSearch.updateAncestorsWhileChanged()]
Infine, **updateAncestors** aggiorna i numeri del nodo corrente (il *most-proving*) e poi quelli di tutti i suoi antenati, fino alla radice (vedere i miglioramenti per il cambio di nome).

2.5.3 Ulteriori dettagli

Il tempo
Per il tempo, si controlla solo, all'inizio di ogni ciclo, che rimangano meno di un certo limite di millisecondi al termine del turno **[PnSearch.isTimeEnded()]**. Le prove mostrano come l'esecuzione di un ciclo abbia una durata insignificante, per cui non sembra esserci rischio di superare il limite in casi straordinari.

La board
La *board* **[BoardBitPn]** su cui si provano le mosse è un singolo oggetto **[PnSearch.board]**, statico e costante, essendo condiviso, come variabile globale, da classi e metodi.
Inoltre rappresenta l'unico modo per "muoversi" nell'albero di gioco: dopo aver marcato/cancellato una casella, è possibile ottenere il nodo nella *tabella di trasposizione* grazie all'*hash* della *board*.
Questo approccio è sufficiente per il proprio obiettivo, in quanto i movimenti nell'albero sono solo tra nodi adiacenti. In più, comporta i seguenti vantaggi:

1. usando puntatori a padri/figli, il loro numero cresce esponenzialmente con l'aumentare delle dimensioni della scacchiera, e così l'uso della memoria e il tempo di gestione;
2. i test non mostrano differenza nelle prestazioni;
3. era difficile mantenere coerente la struttura del dag, con puntatori a padri/figli.

2.5 Dependency-based search (DB-search)

La *Dependency-based search* (*DB-search*) **[DbSearch]** è un algoritmo di ricerca di sequenze vincenti di mosse su alberi di gioco. A partire da un determinato nodo, essa cerca una *sequenze forzata* di *mosse*, ovvero in cui a una propria mossa debba per forza corrisponderne una specifica dell'avversario, poiché quest'ultimo perderebbe eseguendone un'altra e quindi ignorando la *minaccia*. Nell'implementazione, il suo **selectColumn()** ritorna la prima mossa di una sequenza (se trovata).

Prerequisiti
Si dicono *attaccante* (*attacker*) il giocatore per cui si cerca una sequenza vincente di mosse, *difensore* (*defender*) l'avversario.

Per motivi di efficienza, l'algoritmo fa assunzioni che avvantaggiano il *defender*: dunque, si potrebbero ignorare alcune sequenze per l'*attacker*, ma quelle trovate sono sicuramente valide.
Una conseguenza è anche la definizione di una mossa, che, per essere *forzata*, è composta da una pedina dell'*attacker* e zero o più del *defender*, rappresentanti tutte le possibili risposte alla sua minaccia. Di conseguenza, unendo più mosse in una, l'albero di gioco per la *DB* è molto ristretto.

Per l'applicazione dell'algoritmo è necessario che valgano le assunzioni:
1. una mossa non può essere annullata;
2. un gettone in più è sempre un vantaggio.

La seconda, in particolare, non vale sempre in *Connect-X*. In parte, si risolve il problema con il miglioramento dell'operatore *stacked* - che si vedrà in seguito.

2.5.1 Mosse, minacce e allineamenti

Un *allineamento* **[threats.Alignment / Operators.AlignmentPattern - a volte è chiamato `line`]** è un insieme di caselle adiacenti, libere o occupate dall'*attacker*, che segue uno degli schemi di riconoscimento prestabiliti. Per rendere il riconoscimento di *alignment* semplice, si considerano solo quelli lungo la stessa direzione (orizzontali, verticali o diagonali): le combinazioni della *DB-search* le sfrutteranno al meglio.
Indichiamo i seguenti attributi per un *alignment*:
- *start*, *end*: prima e ultima celle occupate delle sue;
- *marked*: numero di sue celle occupate;
- *lined*: distanza end-start (le direzioni convenzionali fanno in modo che tale differenza sia sempre positiva);
- *before*, *after*: numero di celle libere presenti, rispettivamente, prima di *start* e dopo *end*;
- *type*: codice identificativo per il suo schema;
- *tier*: uguale a *X-marked*; un *tier* più basso è più forte.

Una *minaccia* (*threat*) è composta da un *allineamento* e dalle possibili caselle su cui possono essere posizionate una pedina dell'*attacker* - per migliorare il tier - e delle pedine del defender - per interrompere l'*alignment* in ogni modo possibile. Il *tier* di una minaccia è lo stesso del suo *alignment*. **[threats/Threat]**

Un *operator* (anche chiamato *applier*) **[Operators.Applier.getThraetCells()]** è una funzione che crea una *threat* () da un *alignment*.

Ricerca di *alignment*

La *board* **[BoardBitDb]** implementa un algoritmo per individuare tutti gli *allineamenti*, decisi una direzione, i punti di inizio (*first*) e fine (*second*) e il rango massimo, come una ricerca lineare tra questi ultimi, al variare degli attributi dell'*alignment*. In particolare, si usa una lunga funzione ricorsiva **[BoardBitDb._findAlignmentsInDirection()]** con le 3 variabili ausiliarie *c1*, *c2*, *c3*, indicanti delle coordinate: prima si fa scorrere *c1*, da *first*, fino a posizionarlo su una cella dell'*attacker*, accumulando eventuali celle libere in *before*; poi si fa avanzare *c2*, da *c1*, per cercare un *alignment* (quindi solo su celle dell'*attacker* e *free*); per ultimo, si fa avanzare *c3*, da *c2*, solo su celle vuote, per estendere *after*; nei casi opportuni si controlla che si sia trovato un *alignment* valido.
La *board* salva gli *allineamenti* trovati in un'apposita struttura **[BoardBitDb.alignments_by_dir]**, suddivisi in liste concatenate per direzione **[structures.AlignmentsRows]** e poi per riga **[structures.BiList_Alignments]** (per esempio, quelli verticali sono divisi tra N liste concatenate): si è cercato di darle una conformazione più semplice possibile, visto che gli *alignment* vengono acceduti tutti insieme senza un ordine. Sono implementati anche vari metodi per accesso e iterazione.

Applicazione di *alignment*
Una volta noto un *alignment*, anche l'*operatore* può essere applicato linearmente o con un numero finito di operazioni (in termini computazionali, in modo lineare o costante) - a seconda del caso -, controllando, tra tutte le celle nell'intervallo *start-end*, quali siano appropriate per creare una *minaccia*.

Per finire, una *minaccia* può essere applicata marcando, tra le celle che indica, una per l'*attacker* e, per il *defender*, tutte quelle rimaste **[BoardBitDb.markThreat() marca, addThreat() crea una threats.ThreatApplied]** .

[DbNode]
Per quanto riguarda i nodi per la *DB-search*, ognuno ha una sua *board* associata. Per collegarsi tra loro, ogni nodo ha un puntatore al primo figlio e a un fratello, visto che l'unica loro esigenza è di accedere a tutti i figli senza un ordine preciso.

2.5.3 L'algoritmo

[DbSearch.visit()]
La *DB-search* alterna due fasi (*dependency stage* **[DbSearch.addDependencyStage()]** e *combination stage* **[DbSearch.addCombinationStage()]**), interrompendosi solo quando si sia vinta la partita, marcato una delle *goal square* (che si vedranno dopo), o quando non ci siano più *alignment* da sfruttare per creare *minacce*.
Tuttavia potrebbero verificarsi le seguenti situazioni:

1. il *defender*, piazzando i suoi gettoni per difendersi, crea involontariamente un *allineamento* di *X*;
2. l'*attacker* può applicare delle *minacce*, ma il *defender*, invece di rispondere direttamente, potrebbe applicarne di più "forti", cioè raggiungerebbe la propria vittoria più velocemente.

Per il punto 1 basta controllare, a ogni pedina piazzata, lo stato di gioco, ed evitare di usare una configurazione persa.
Per il punto 2, una volta trovata una sequenza per l'*attacker*, si cerca una *global defense*.

```
boolean visit(DbNode root, boolean attacking, int max_tier) {
            LinkedList<DbNode> lastDependency = new LinkedList<DbNode>(), lastCombination = new
LinkedList<DbNode>();

            boolean found_goal_state = false;
            while(isTreeChanged() && !found_goal_state) {
```

```
                lastDependency.clear();
                int max_tier_t = attacking? max_tier : root.getMaxTier();
                        if(addDependencyStage(lastDependency, lastCombination, root, attacking,
max_tier_t))
                    found_goal_state = true;

                if((attacking && !foundWin()) || (!attacking && !found_goal_state))
                {
                    lastCombination.clear();
                    addCombinationStage(lastDependency, lastCombination, root, attacking);
                }
            }
            return found_goal_state;
}
```

Si usano due contenitori: il *dependency stage* riempie *lastDependency* (dopo averlo svuotato) con i nodi prodotti e usa i nodi in *lastCombination*; il *combination stage* fa l'opposto. *lastCombination* è inizializzato con la radice e tutti i suoi figli (per l'*attacker*, nessuno).
Il *dependency stage*, per ogni nodo a sua disposizione, applica tutte le possibili *minacce*, creando per ognuna un nuovo nodo - figlio del precedente -; ripete lo stesso anche per ogni figlio così generato. E' sufficiente controllare le partite terminate solo in *addDependentChildren*.

Il *combination stage* prova a combinare tutte le possibili coppie nodi (a sua disposizione) *A* e *B*, ovvero a creare, per ogni tentativo, una tabella (e quindi un nodo) che abbia marcate tutte le caselle marcate in almeno uno dei due nodi padri. Per ottimizzare, prima di provare a creare una combinazione si verifica **[DbNode.validCombinationWith()]** che sia *USEFUL* (e non *USELESS* o *CONFLICT*), cioè sia A che B contribuiscano con almeno una *minaccia.*
Inoltre, sono previste le seguenti interazioni tra combinazioni:

- più nodi possono essere combinati insieme nello stesso stage, due alla volta;
- per non ripetere una combinazione (il che porterebbe a un ciclo infinito), la si aggiunge solo a un padre e si usa una *transposition table* **[DbSearch.TT]** .

La *tabella di trasposizione* viene svuotata a ogni *stage*.

[DbSearch.visitGlobalDefense()]
Se si trova una possibile sequenza vincente, si cerca una *global defense*. Ora il *defender* attacca per vincere per primo (con una sequenza di *tier* più forte), pareggiare o occupare una *goal square* **[DbSearch.GOAL_SQUARES]** , ovvero una delle caselle coinvolte nella sequenza: se ne occupasse una in anticipo, impedirebbe una *minaccia* della sequenza. Questa seconda visita, dunque, cerca solo *minacce* di *tier* più basso, e non applica euristiche, per non tralasciare alcuna difesa.
Prima di applicare la nuova *DB-search*, però, per ogni *threat* della sequenza si crea un nodo, figlio del precedente **[DbSearch.createDefensiveRoot()]** ; tuttavia in un nodo si marcano solo la mossa dell'*attacker* e le mosse del *defender* della *minaccia* precedente: in questo modo il *defender*, invece di applicare subito una risposta a ogni *minaccia*, può cercare una *global defense*.

Esempio
Mostriamo l'efficienza dell'algoritmo con un esempio.
Qui *M,N,X* valgono *9,9,5*, 'x' e 'o' rappresentano *attacker* e *defender*, le ultime mosse fatte sono marcate con simboli maiuscoli, e righe e colonne sono numerate da 0.
A differenza di un algoritmo come PN-search, a *DB-search* basta visitare pochi nodi: dal *dependency stage* della figura 1 emergerebbero due nodi, rispettivamente dalle minacce originate dagli *alignment* in

riga 3 (figura 2) e in colonna 3 (figura 3) (entrambi di *tier 2*); successivamente, dalla loro combinazione l'*attacker* vincerebbe (figura 4). Pertanto la mossa vincente sarebbe "colonna 3".
Si noti anche che:

1. non esistono *global defense*: anche se il *defender* giocasse, dopo *3,3 (riga,colonna)* dell'*attacker*, *3,4*, l'*attacker* avrebbe comunque una vittoria con *4,3*;
2. un *allineamento di tier 1* non ha risposte, poiché porta a una vittoria.

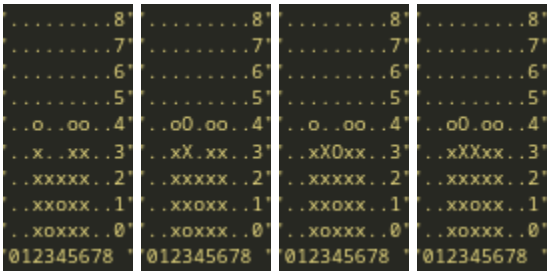


figure 1,2,3,4

2.6 Le tabelle di trasposizione

2.6.1 Implementazione

Le (3) *TT* **[tt.TranspositionTable]** utilizzate nel progetto sono delle *hash table*, con funzione *Zobrist hashing* e concatenamento per le collisioni.
Dunque, sono implementate come array di dimensione potenza di 2, indicizzati con la chiave dell'elemento in modulo della dimensione stessa.

Zobrist hashing
Lo *Zobrist hashing* è applicato così:

1. una chiave ha *64 bit*;
2. viene generato un numero casuale per ogni possibile combinazione di riga *i*, colonna *j*, giocatore *p* (quindi *M*N*2* numeri) **[TranspositionTable.moves, TranspositionTable.initMovesHashes()]** ;
3. la *chiave*, o *hash*, di base di una tabella è 0;
4. ad ogni mossa effettuata, il nuovo *hash* della tabella si ottiene applicando a quest'ultimo uno *xor* con il numero casuale per la mossa fatta.

2.6.2 Uso

Nella *DB-search*, si usa una *TT* solo per non ripetere le combinazioni, esattamente come già detto. Data la necessità minimale, i suoi elementi **[tt.TTElementBool]** non contengono informazioni, poiché basta solo conoscerne l'esistenza.

La *PN-search*, invece, usa un sistema di due *TT*, e ogni nodo può trovarsi solo in una delle due.
TTdag rappresenta l'albero della PN-search.
TTproved memorizza, per ogni nodo dimostrato **[tt.TTElementProved]** :

1. la profondità **[depth_cur]**, usata anche come controllo insieme alla chiave, per ridurre il rischio di collisioni;
2. il valore (*true/false*) **[won()]**;

3. la profondità della migliore foglia raggiungibile **[depth_reachable]** ;
4. in entrambi i casi, la prima mossa da eseguire, da tale nodo, per raggiungere tale foglia **[col()]** .

Il parametro 3, quindi, permette di trovare la vittoria/sconfitta migliore: una vittoria in meno mosse è meno prona a successivi errori; una sconfitta più lenta fa sperare che l'avversario, nel frattempo, compia un errore. In più, un pareggio si memorizza con profondità *M*N+1* (cioè oltre la massima profondità); in tal modo:

- in un nodo perso per *P1*, per cui il pareggio è una sconfitta, esso sarà preferito a una vera sconfitta, essendo il più profondo per forza;
- in un nodo vinto per *P2*, per cui il pareggio è una vittoria, esso sarà messo in secondo piano da vere vittorie, per forza meno profonde.

In realtà **updateAncestors** propaga, oltre ai *proof-number*, anche i valori di nodi dimostrati.

3 Miglioramenti

Questa sezione illustra i miglioramenti apportati a ognuno degli algoritmi, strutture o idee utilizzati, senza un preciso ordine.

3.1 Memoria

Il problema
Un problema centrale nello sviluppo saturazione della memoria principale (di default, la java virtual machine usa 3GB), dovuto all'elevato numero di nodi creati: di conseguenza, la garbage collection, venendo avviata in automatico, faceva spesso sforare il limite di tempo (con i 3GB pieni può richiedere diversi secondi). La difficoltà nell'individuazione, dunque, è stata anche nel fatto che questi ritardi di tempo si verificassero soprattutto con grandi configurazioni (dove si creano anche più nodi), per cui venivano scambiati per problemi di performance.

La soluzione
La soluzione prevede i seguenti provvedimenti:

1. implementazione del dag con *TT* e *board* come già visto, invece di avere *N* puntatori da ogni nodo;
2. si esegue una *garbage collection* all'inizio di ogni turno, per evitare che questa venga invocata proprio prima della fine, facendo sforare il limite di tempo;
3. quando un nodo viene dimostrato, tutti i suoi figli vengono rimossi **[PnSearch.pruneTrees()]** ;
4. quando inizia un turno, e quindi la nuova radice (se era già stata visitata in un turno precedente) si trova due livelli oltre la precedente, si eliminano tutti i nodi (discendenti) che siano raggiungibili dalla vecchia radice ma non da quella attuale.

I punti 3,4 , purtroppo, implicano la perdita di calcoli già svolti che potrebbero servire in seguito.

removeUnmarkedTree
Il punto 4 è implementato con il tagging del sotto-albero della nuova radice **[TTPnNode.tagTree()]** e con la successiva rimozione di tutti i nodi discendenti della vecchia ma non della nuova, quindi con un *tag* diverso da quello attuale **[TTPnNode.removeUnmarkedTree()]** . Il tagging di un nodo sfrutta un solo bit di *tag/mark*, che ha valore *(turno / 2) % 2*, e sia **tagTree()** che **removeUnmarkedTree()** interrompono la ricorsione se trovano un nodo con già il *mark* giusto, poiché tutti i discendenti di un nodo hanno sempre il suo stesso *mark*.
Una dimostrazione di correttezza si ha analizzando i seguenti casi:

- dal turno precedente non sono sopravvissuti nodi con il *mark* attuale (che erano stati rimossi con *removeUnnmarkedTree*);
- quindi, un nodo con lo stesso *mark* deve essere stato appena segnato con l'attuale esecuzione di **tagTree** e, di conseguenza, già è stata eseguita la ricorsione su di esso.

removeUnmarkedTree ha anche un vantaggio computazionale: senza, **updateAncestors** aggiornerebbe anche antenati non raggiungibili dalla radice.

Altri dettagli
Inoltre, quasi tutte le variabili sono implementate o con tipi primitivi (a cui non corrispondono oggetti), o con variabili statiche che vengono riutilizzate (es.: *c1,c2,c3* in BoardBitDb). Anche i metodi di MovePair (classe ausiliaria che implementa una coordinata) sono pensati per non creare mai nuovi oggetti, ma modificarne soltanto (stesso esempio: uso di *c1,c2,c3*).

3.2 PN-search

I miglioramenti applicati per la *PN-search* sono:

1. **TTPnNode.most_proving_col** : *TTPnNode* memorizza la mossa per raggiungere il proprio *most-proving node*, così che non si debba sempre ricalcolare;
2. **updateAncestorsWhileChanged()** : si interrompe la propagazione di un valore verso gli antenati quando i *proof-number* non vengono aggiornati;
3. inizializzazione euristica dei *proof-number* **[PnSearch.initProofNumbers()]** ;
4. conta degli *alignment* come priorità per i figli **[DbSearch.getThreatCounts()]** ;
5. i figli "di pari importanza" sono riordinati in modo casuale, perché ci siano pari probabilità per ognuno di essere analizzato prima (altrimenti, per esempio, ordinandoli in base alla colonna della mossa, si preferirebbero sempre le prime colonne);
6. *null-move* e *implicit threat heuristic*: *DB-search* aggiuntiva per dare le priorità ai figli;

3, 4, 5, 6 generateAllChildren() e initProofNumbers()

I *proof-number* iniziali possono essere sfruttati per dare priorità ai figli: un nodo non sarà mai *most-proving* prima di un altro con numeri inferiori.
Per non favorire branch dell'albero troppo profondi, i *proof-number* vengono inzializzati a `*depth / 2 + 1*` (con l'1 per evitare 0), a cui si somma un *offset* specificato in **generateAllChildren**.

Per l'altra euristica, definiamo una *implicit threat* come una sequenza vincente per l'avversario in un proprio nodo: cioè, per trovarne una in un nodo di *P1*, basta eseguire una *DB-search* per *P2* (come se *P1* avesse fatto una mossa nulla). L'euristica propone dunque di ignorare completamente i nodi che non appaiano nell'*implicit threat* (se trovata), poiché, essendo una sequenza sicuramente vincente, *P1* perderebbe non occupando una di quelle caselle. Tuttavia, in mancanza di una dimostrazione per l'applicazione a *Connect-X*, l'implementazione corrente non ignora gli altri figli, ma li inizializza solo con *proof-number* molto alti.

Dopo i miglioramenti 3,4,5,6 , *generateAllChildren* esegue i seguenti passaggi (prima di creare i figli):

1. cerca una *implicit threat* e ritorna le colonne usate in essa;
2. cerca gli *alignment* attuali per il *player* attuale;
3. assegna offset crescenti, dando priorità prima a figli per colonne in *implicit threat*, poi a figli per colonne con più *alignment*, e disponendo casualmente quelli con stessa priorità.

3.3 DB-search

I miglioramenti applicati per la *DB-search* sono:

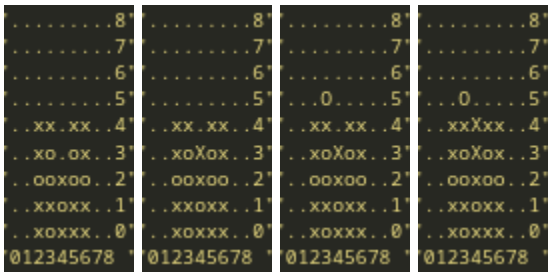
1. **BoardBitDb.alignments_n** : la *board* salva il numero di *alignment*, per evitare di eseguire del codice quando non ce ne siano;
2. **AlignmentsRows** inizializza le **BiList_Alignments** a *null* e le crea solo con l'inserimento di un elemento, per ridurre significativamente l'impatto sulla memoria;
3. **DbSearch.MAX_THREAT_SEQUENCES** : per una radice, la visita si interrompe dopo aver provato a verificare (contro le *global defense*) un certo numero di sequenze; così si previene il caso in cui ci siano troppe sequenze da analizzare inutilmente, perché verrebbero tutte stroncate dalla stessa *global defense*;
4. operatore *stacked*.

Applicare DB-search al gioco sbagliato

Non si è parlato nello specifico degli operatori, ma questi sono degli esempi con *X=5* ('x'=*P1*, 'o'=*P2*, '_'=FREE, le lettere maiuscole evidenziano l'ultima mossa, '->' indica l'applicazione della minaccia):

- _x_xx_ -> _xXxx_ (*tier* 2): l'applicazione di questa mossa non ha difese perché è praticamente una vittoria, al turno successivo;
- _x__x_ -> OxXOxO oppure OxOXxO : l'*attacker*, mettendo una pedina tra le due presenti, minaccia di creare l'*alignment* precedente (che sarebbe vincente), e il *defender* risponde con tutte le 3 possibili difese contemporaneamente.

Senza l'operatore *stacked*, la *DB-search* non sfrutterebbe la verticalità del gioco e non vedrebbe minacce come la seguente (vale la stessa notazione e *X=5*):



Dopo la mossa dell'immagine 2 è stato creato un nuovo *alignment* (orizzontale), e *P2* perderebbe ignorandolo (immagini 3,4).

In generale, dunque, questa applicazione della *DB-search* non è completamente corretta (cioè una sequenza non è garantita): non valendo la già citata assunzione "un gettone in più è sempre un vantaggio" (2.5), in rari casi si potrebbe tralasciare una *global defense*; per lo meno, *stacked* ne dovrebbe coprire alcune (o magari tutte, ma servirebbe una dimostrazione).

Quindi, il "meta-operatore" *stacked* controlla anche se possa crearsi un *allineamento* di tipo *stacked*, considerato come uno verticale, in seguito all'impilazione di pedine. Ciò avviene con una funzione esterna per **_findAlignmentsInDirection** : **findAlignmentsInDirection**, per la direzione verticale, prova a impilare pedine, e con ogni numero di pedine impilate chiama **_findAlignmentsInDirection**, ma indicandogli *max_tier* diminuito del numero di pedine impilate (così che vale la definizione di *tier=X-marked*, cioè se ne possono impilare fino a *2*) e, come cella da controllare, quella appena sopra.

4 Implementazione

4.1 Struttura dei file

- doc: documentazione
- out: file .class
- src: source code
    - connectx
        - pndb: propria implementazione
            - constants: costanti / funzioni generiche
                - Auxiliary.java: funzioni helper generiche
                - CellState.java: cell state
                - Constants.java: costanti generiche
                - GameState.java: game state
                - MovePair.java: classe MovePair
            - structs: strutture
                - DbSearchResult.java: tipo di ritorno di una DB-search
            - structures: strutture dati
                - AlignmentsRows.java: struttura per memorizzare threat in BoardBitDb
                - BiList_Alignments.java: coppia di BiList per Alignment, una per ogni giocatore
                - BiList.java: lista concatenata bi-direzionale, generica
            - threats: classi per definizioni di minacce
                - Alignment
                - Threat
                - ThreatApplied
            - tt: transposition table
                - TranspositionTable.java: Classe transposition table
                    - TranspositionTable.Element: classe astratta per l'elemento
                - TTElementBool.java: element per la TT di DB-search
                - TTElementProved.java: element per TTproved
            - BoardBit.java: classe astratta per la bitboard
            - BoardBitDb.java: bitboard per DB-search
            - BoardBitPn.java: bitboard per PN-search
            - DbNode.java: nodo per DB-search
            - DbSearch.java: DB-search
            - Operators.java: definizione dei singoli alignment, threat, operator, e come riconoscerli (applier)
            - PnSearch.java: PN-search
            - TesterDb.java: testa DB-search su una sola configurazione, per una mossa
            - TesterPn.java: testa PN-search su una sola configurazione, per una mossa
            - TTPnNode.java: nodo per PN-search (implementante anche TranspositionTable.Element)

4.2 Analisi computazionale

Seguono i costi dei metodi implementati, divisi per categoria - alla fine di ogni categoria si analizzano i risultati principali. Le piccole funzioni qui tralasciate sono costanti o non rilevanti nell'implementazione.
Per semplificare, si usa solo *N* per indicare sia *N* che *M*. Inoltre: in ogni momento, *created_n* è il numero di nodi PN creati in totale; *loops_n* è il numero di cicli eseguiti in una visita PN; *root* è la radice a ogni **selectColumn()**.

Tabelle di trasposizione

creazione	O(size)
operazioni (insert, remove, get)	O(1) -- alcuni O(1+alpha)*

*Il fattore di carico *alpha* è, di solito, molto basso, quindi si può considerare costante (vedere i dati in testing).
Dunque le *TT* non sono costose da usare, e l'inizializzazione, di ben 2**23 celle tra *TTdag* e *TTproved* (302MB), avviene solo nell'*initPlayer*.

BoardBit, BoardBitDb, BoardBitPn

scopo - funzione()	**caso peggiore/tutti**	**caso migliore**
creazione, copia - costruttori, copy(), createStructures(), getCopy()	O(5N)	O(3N) se M <= 64
creazione+copia - costruttori con copia	O(10N)	O(6N) se M <= 64
mark, unmark, cellState, cellFree, funzioni su TT	O(1)	
check, markCheck, isWinningMove	O(4X)	O(1)
[BoardBitDb]		
markAny(), mark(MovePair), mark(i, j)	O(4X + removeAlignments)	O(1 + removeAlignments)
markMore(cells[]), markThreat(cells[])	O(cells.length * 4X)	O(cells.length)
_findAlignmentsInDirection(), findAlignmentsInDirection() (senza considerare stacked)	O(2 * distance + 18*possible_alignments)**	
alignment per una cella in 4 direzioni - findAlignments(cell)	O(40X + 18*possible_alignments)**, cioè 4 direzioni (4*4X) e fino a due *stacked*, quindi 2 volte per 3 direzioni, esclusa la verticale (2*3*4X)	
tutti gli *alignment* per una *board* - findAllAlignments()	O(12N**2)**, cioè 6N iterazioni di findAlignmentsInDirection()	
_addAllValidAlignments()	O(18) essendo _addValidAlignments() per max 3 volte	O(1) se si cerca un tier più basso di quello trovato
check+findAlignments - checkAlignments(cell), checkAlignments(cells)	O(cells.length * (40X + 18*possible_alignments)), come findAlignments, senza contare miglioramenti	O(4X) se la partita è conclusa
removeAlignments(cell)	O(numero di alignment in direzione con cell, per ogni direzione)	
getDependant()	O(6N)**	
getCombined()	O(6N + added_threats_n * 3 * 40X)**	O(6N)** se nessuna threat aggiunta
iterazione su *alignments_by_dir* - nextStartOfRow_inDir()	O(N) per righe e colonne, O(2N) per diagonali, O(6N) per tutti	
validCombinationWith()	O(2N)	O(N) se M <= 64
getApplicableOperators()	O(6N + alignments_n * X), cioè copia e uso di *applier*	

	per ogni *alignment*	
getThreatCounts()	O(12N**2), per findAllAlignments()	
[Operators]		
applier - applied()	O(X) per i più complessi	O(1)

visto che ognuno dei 4 array che compongono **alignments_by_dir è inizializzato al primo utilizzo, per una board può accadere, una sola volta, che ci sia da aggiungere il costo di inizializzazione O(2N) per righe e colonne e O(3N) per diagonali, fino a un massimo di O(10N).

Le operazioni di base sono costanti (tranne il controllo per la vittoria, che richiede O(4X)), mentre O(N) è richiesto dalla copia, e quindi anche da varie funzioni che la usano: i costruttori-copia, getDependant, getCombined (influenzato possibilmente anche dal numero di minacce aggiunte nella combinazione).

La ricerca degli *alignment* può essere eseguita in vari modi, ma vale la costante che bisogna scorrere le celle analizzate 2 volte per ogni direzione (con **_findAlignmentsInDirection()**)***:*** quindi, per una singola cella si usano 2 volte solo le 2X celle intorno, per un'intera fila se ne usano 2N, mentre, per un intervallo tra due punti allineati *first* e *second*, `*first - second*` celle. In più, per verificare e, in caso, salvare dei possibili *allineamenti*, si chiama ogni volta ***_addAllValidAlignments()***.

Inoltre, se si sta usando la direzione verticale, l'operatore *stacked* cerca *allineamenti* per fino a 2 celle impilate, ma senza ripetere la direzione verticale. Di conseguenza, gli *alignment* intorno a una cella, in 4 direzioni, richiedono O(4X) per le 4 direzioni, seguiti da O(4X) per le 2 celle impilate, lungo solo 3 direzioni (totale: `*4*4X+2*3*4X=40X*`).

Nodi

[TTPnNode]	
tagTree(), removeUnmarkedTree()	O(tree.size) in media, O(created_n - created_n_last), cioè nodi creati nell'ultimo selectColumn()
hasParents()	O(N)
getMoveToBestChild()	O(2N)
updateProofAndDisproofOrProve()	O(N)
[DbNode]	
copy(), validCombinationWith()	come BoardBitDb

DbSearch

scopo - funzione()	**caso peggiore/tutti**	**caso migliore**
init()	O(TT.size), cioè 2**16	
selectColumn()	O(11 (12N**2 + N*threats_n**2))***, dato dal numero di *threat* presenti nella radice	O(12N**2 + N*threats_n**2)
addDependentChild()	O(6N)**, come board.getDependant()	
addCombinationChild()	O(6N + added_threats_n*3*40X)**, come board.getCombined()	
addDependentChildren(node) (solo un'iterazione senza ricorsione)	O(6N * (1 + applied_threats_n))**	O(1), se non ci sono *alignment*
findAllCombinationNodes(partner, node) (solo un'iterazione senza ricorsione)	O(7N + combined_threats_n * 3 * 40X)**, altrimenti	O(N) se non si creano combinazioni
addDependencyStage(lastCombination)	O(6N*(1 + applied_threats_n) per ogni nodo in (lastCombination + nodes_created_n))**, ovvero dipende dal numero totale di *threat* applicate in questo stage	
addCombinationStage(lastDependency)	O(N*lastDependency.length**2)	
visit(), visitGlobalDefense() (una visita)	O(N*lastDependency.length**2)	
per euristica di generateAllChildren - getThreatCounts()	O(12N**2), come BoardBitDb	

[ausiliarie]		
initLastCombination(node)	O(node.descendants_n), linear in the size of the sub-tree	
removeCombinationTTEntries(lastCombination)	O(lastCombination.size)	
createRoot()	O(10N), come board.copy	O(6N)
createDefensiveRoot()	O(12N**2), come *findAllAlignments()*	
markGoalSquares(applied_threats)	O(4*applied_threats.size), essendo una *threat* di max 4 celle	
valore di ritorno di selectColumn - getReturnValue()	O(N)	

**stesso asterisco dei costi per le *board*.

Il costo per *selectColumn()* va ricostruito dai suoi tasselli più piccoli:

- la creazione di un nodo (**addDependentChild**, **addCombinationChild**) riflette i relativi metodi in **BoardBitDb**;
- le singole iterazioni di **addDependentChildren** e **findAllCombinationNodes** lo stesso (a parte i casi ottimi), ma si noti, per le combinazioni, che O(N) deriva da **validCombinationWith**, che viene applicato sempre, mentre gli altri costi si hanno solo alla creazione di un nodo combinato;
- dunque, **addDependencyStage** crea un nodo (con costo 6N) per ogni *minaccia* presente in qualsiasi nodo tra la scorsa *lastCombination* e i nodi così ottenuti (visto che un nodo può continuare a produrre discendenti nello stesso stage), mentre in **addCombinationStage** il costo maggiore è quello dell'*O(N)* applicato a tutte le *lastDependency.length**2* combinazioni provate (perché le combinazioni effettivamente create non sono più di quelle provate, e il costo di quelle trovate è asintoticamente minore o uguale di questo);
- una visita in **visit** è influenzata dal solo *combination stage* di ogni loop (**addDependencyStage** è asintoticamente minore, del tipo *O(N*lastDependency.length)*);
- in **selectColumn**, oltre ai **findAllAlignments** (*O(12N**2)*) per la radice e per ogni ricerca di una *global defense*, si eseguono fino a 11 **visit**, di cui una per l'*attacker* e fino a *10* (per l'euristica) per il *defender* (anche se le *global defense* dovrebbero essere molto più veloci, lavorando solo con *tier* migliori).

*** Il risultato finale dipendente dal numero di *minacce* applicabili nel nodo radice (`O(11(12N**2 + N**threats_n**2) )`) deriva dal fatto che, tipicamente, si eseguono pochi *stage* (1-3), e che il numero di *threat* non cresce tra questi (approssimazione grossolana, ovviamente andrebbero considerate le *threat* trovate a ogni *stage*).

PnSearch

scopo - funzione()	**caso peggiore/tutti**	**caso migliore**
initPlayer()	O(TT.size), cioè 2**23	
selectColumn()	O(2(created_n - created_n_last) + loops_n*276N**2 + (22N*threats_n**2 per ogni most_proving) + 3N*(created_n - created_n_last)*loops_n), per visit() e memory management	O(2(created_n - created_n_last) + loops_n*276N**2 + (22N*threats_n**2 per ogni most_proving) + 3N*(created_n - created_n_last)), per visit() e memory management
visit()	O(loops_n*276N**2 + (22N*threats_n**2 per ogni most_proving) + 3N*(created_n - created_n_last)*loops_n), per developNode() e updateAncestors()	O(loops_n*276N**2 + (22N*threats_n**2 per ogni most_proving) + 3N*(created_n - created_n_last))
selectMostProvingNode()	O(4X (most_proving.depth - root.depth))	
developNode()	O(276N**2 + 22N*threats_n**2), per evaluate() e generateAllChildren()	O(1), se partita finita

evaluate()	O(132N**2 + 11N*threats_n**2), per evaluateDb	O(1) se partita finita
evaluateDb()	O(132N**2 + 11N*threats_n**2), per Db	O(12N**2 + N*threats_n**2)
generateAllChildren()	O(144N**2 + 11N*threats_n**2), per DB e getThreatCounts()	O(12N**2 + N*threats_n**2)
updateAncestorsWhileChanged()	O(3Nn), con n numero di nodi aggiornati	O(1), per un solo nodo
setProofAndDisproofNumbers()	O(2N), se nodo espanso	O(1)
scollega nodo/i in alberi - pruneTree(), pruneTrees()	O((intersezione dei tree).size), cioè non si ripetono visite a un nodo in più alberi	
[ausiliarie]		
initProofAndDisproofNumbers(), isBetterChild()	O(1)	
annulla mosse di un loop di visit - resetBoard()	O(most_proving.depth - root.depth)	
updateProved(), getColFromEntryProved()	O(N)	

Nella *PN-search*, il costo maggiore è dato dalle *DB-search* e da *updateAncestors()* per la visita e dalla gestione della memoria.

In media, si può dire che la gestione della memoria richieda al massimo *2(created_n - created_n_last)*, poiché sia la *garbage collection* che la combinazione di *tagTree()* più *removeUnmarkedTree()* visitano ogni nodo creato nell'ultimo turno.

Per l'esecuzione di una *visit()*, sempre in media e cercando di usare dati concreti, un'approssimazione per eccesso prevede che si abbia:

- *loops_n* volte il costo di *O(276N**2)*, risultante dalle *DB-search* e da *getThreatCounts()*;
- tra `*created_n - created_n_last*` e `*(created_n - created_n_last) * loops_n*` volte *O(3N)*, cioè il minimo e massimo numero di nodi visitati in totale da *updateAncestors()* (due estremi molto improbabili);
- `*loops_n*` volte *O(22N*threats_n**2)* per le *DB-search*, con il numero di *threat* presenti nel *most proving* di ogni loop.

5 Esecuzione e testing

5.1 Esecuzione

Per eseguire, si possono usare i seguenti comandi (da linux):

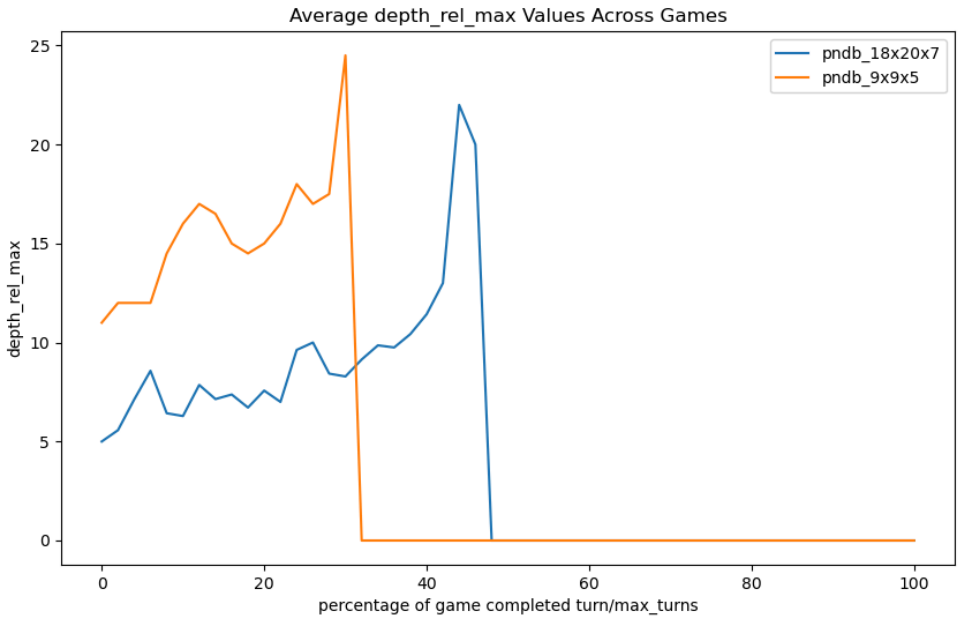
1. `./lcompile` : creazione file .class in ./out/ : `javac -d out -sourcepath src src/connectx/pndb/*.java src/connectx/pndb/*/*.java`
2. `./lplay connectx.pndb.PnSearch [PARAMS]` : giocare umano vs. player : `java -cp out connectx.CXGame pndb.PnSearch [PARAMS]`
    - specificando eventuali PARAMS per CXGame
3. `./ltest connectx.pndb.PnSearch PLAYER2 [PARAMS]` : giocare player vs. player : `java -cp out connectx.CXPlayerTester pndbPnSearch PLAYER2 [PARAMS]`
    - specificando PLAYER2 ed eventuali PARAMS per CXPlayerTester
4. `./lclean` : rimuovere i file .class : `rm -r out/*`

5.2 Testing

Le migliorie sono state applicate gradualmente, creando di volta in volta una nuova classe, in modo da poterne constatare i risultati con i test. Ovviamente sono stati usati anche i due player forniti (L1, L2), ma solo per verificare che non ci fossero errori banali nell'implementazione, essendo molto scarsi.

5.2.1 Modalità e strumenti

Essendo il testing lungo, più configurazioni e combinazioni di avversari erano raccolti in un unico script, che veniva eseguito in background su macchine proprie o di laboratorio (da remoto), reindirizzando l'output su file per poterlo leggere al termine.
In fase di sviluppo, ogni player creato aveva un'opzione per stampare i passaggi eseguiti in file aggiuntivi, che veniva attivata per risolvere problemi di implementazione. Solo dopo veniva disattivata per testarne le effettive performance (essendo questa opzione molto dispendiosa in tempo e memoria).

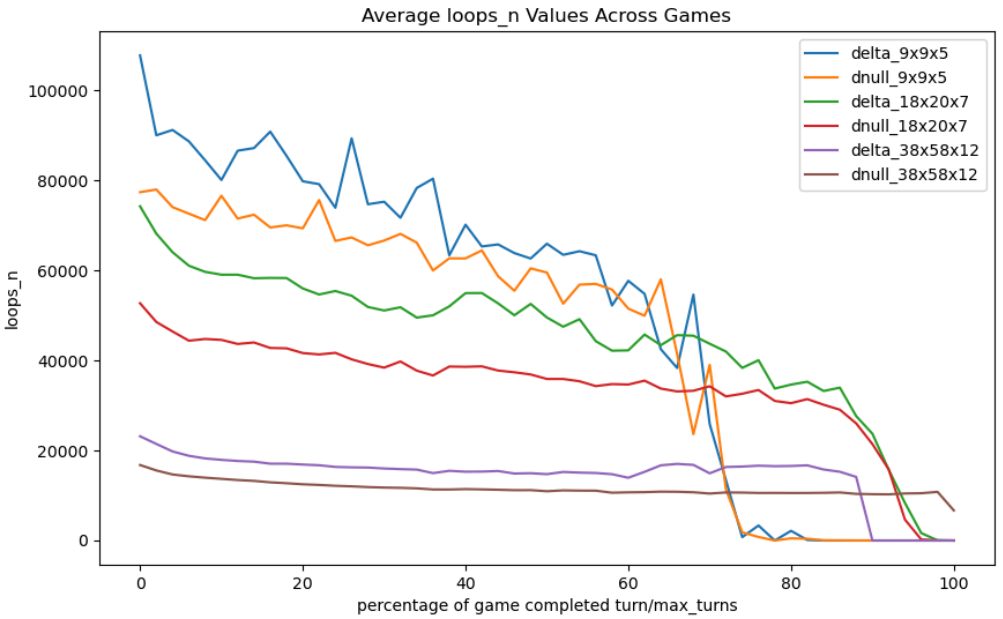


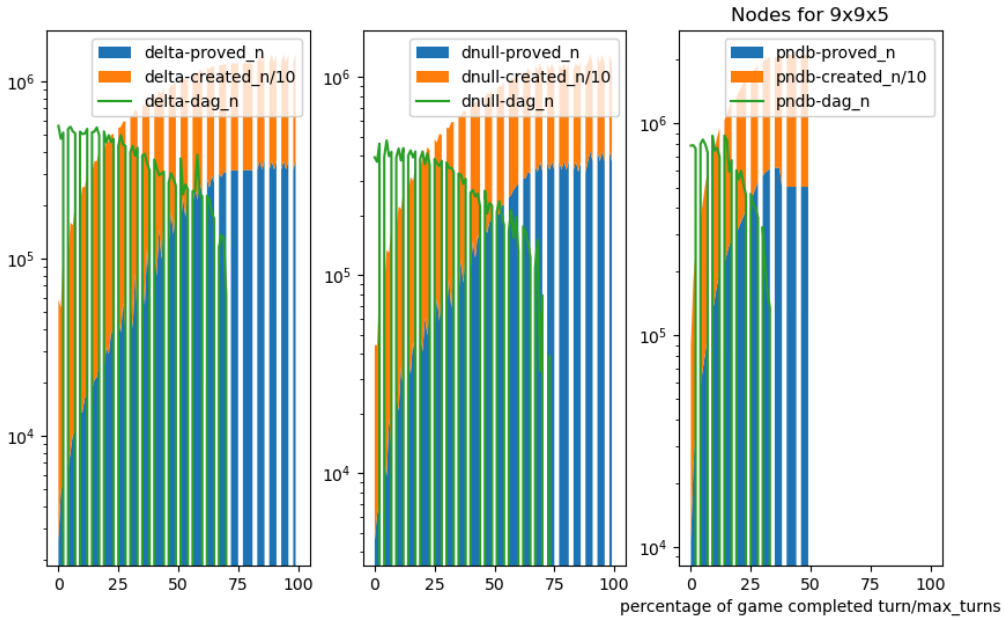
Per avere un'idea del numero di oggetti creati e della memoria utilizzata per ogni classe, si è fatto uso di:

- aggiornamento e stampa (alla fine di ogni mossa) delle variabili **PnSearch.created_n** (nodi creati in totale, quindi anche rimossi in seguito) e **TranspositionTable.count** (aggiornato con nodi aggiunti e rimossi; usato per **TTdag** e **TTproved**, aggiornato in **BoardBitDb** e **BoardBitPn**) - al momento le relative righe sono commentate;
- stampa della quantità usata di tempo (`*System.currentTimeMillis() - timer_start*`) e memoria **[Auxiliary.freeMemory(runtime)]**, alla fine di ogni mossa o anche in momenti intermedi;
- programma `visualvm`, con un'interfaccia grafica per l'uso della memoria e delle classi.

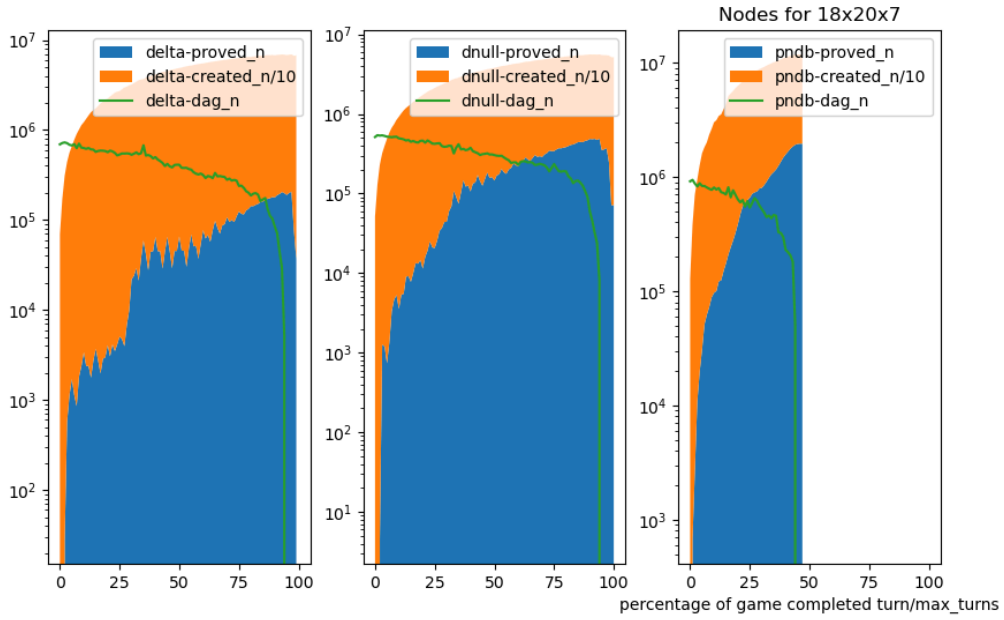
5.2.2 Risultati

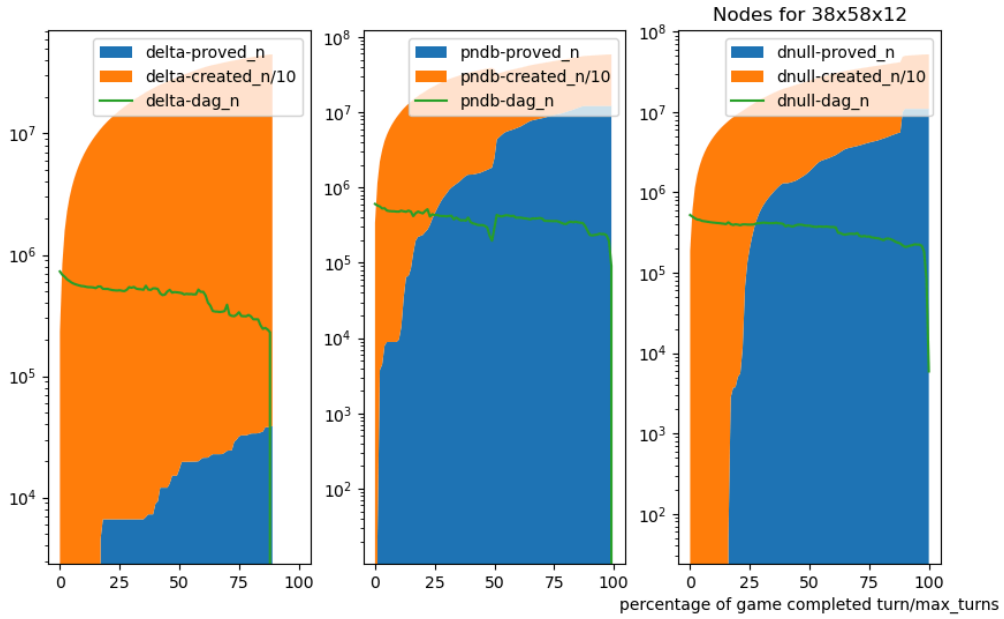
In linea generale, una nuova versione vinceva più partite contro la precedente; tuttavia, non sempre un miglioramento incrementava le performance: ciò ha evidenziato che è più importante ridurre il numero di nodi analizzati che rendere più efficiente una singola operazione.

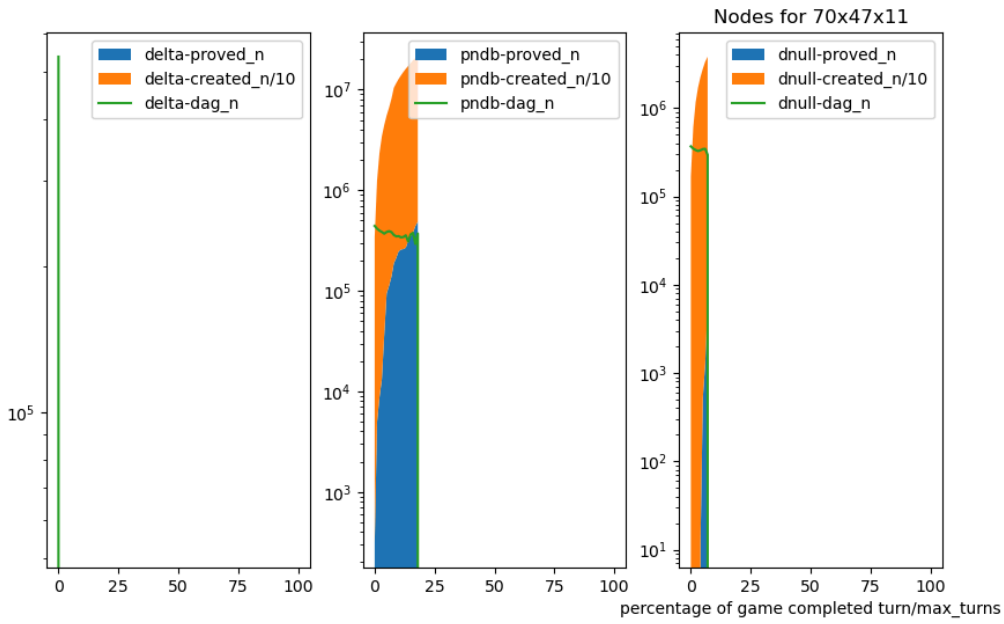




Sfortunatamente, ci sono stati molti fix di pari passo con le migliorie, per cui alcuni confronti non hanno senso per via di bug rimasti solo nelle vecchie versioni. Per esempio, non ci sono dati sull'efficacia del riordinamento dei figli, mentre un grande contributo è stato dato dai provvedimenti per l'uso della memoria. Si riportano qui solo i dati per le ultime 3 versioni, con tempo limite a 9s invece di 9.9s: *delta* (che non usa *implicit threat*), *dnull* (che le usa), *pndb* (*dnull* con gli ultimi







fix e stampe di debugging rimosse). I grafici mostrano le metriche già viste (con `*created_n/10*` essendo molto grande) al variare della percentuale `*turno/(M*N)*`. I valori possono essere imprecisi, essendo medie su pochi campioni, che comunque hanno richiesto lunghi test (soprattutto per le tabelle più grandi), a volte anche soggetti a bug; tuttavia, si possono constatare queste stime, che riflettono alcune aspettative:

loops_n: si eseguono più cicli di *PN-search* con tabelle più piccole (essendo l'albero più stretto e l'*evaluation* più semplice), e il numero descresce con l'avanzare della partita (introducendosi più possibilità per *minacce*) - se ne eseguono circa 100.000 per 9,9,5, 50.000 per 18,20,7, 20.000 per 38,58,12 (che tuttavia descrescono più lentamente);

depth_rel_max: al contrario, si scende di più con un albero più stretto, anche grazie alla memorizzazione di nodi tra turni, e forse all'inizializzazione euristica dei *proof number* - si va da 5 a 25 livelli di profondità massima raggiunta;

implicit threat: permettono di dimostrare molti più nodi già da subito (+100% **proved_n** che senza nei primi turni, per scendere subito verso i +50%, e linearmente fino a +10-30% alla fine); più *DB-search* implicano meno nodi visitati e meno loop (**loops_n** e **dag_n** sono intorno al -25%);

memoria: (per 9,9,5 e 18,20,7) la garbage collection richiede circa 50-150ms, **tagTree()** al massimo 50ms, **removeUnmarkedTree()** 200ms, con rari casi di 50ms o 500ms.

5.2.3 Confronto con costi computazionali

Per quanto approssimati, si può vedere come tutte le variabili prese in considerazione nei costi calcolati contribuiscano effettivamente alle prestazioni della *PN-search*:

- le 3 fasi di gestione della memoria richiedono un tempo più o meno fisso ma non ignorabile, e soprattutto dipendente dal *created_n* di ogni turno; la *gc* è più efficiente, probabilmente per una migliore implementazione;
- l'euristica delle *related squares*, pur creando meno nodi, migliora i risultati: da un lato può essere che funzioni veramente (essendoci più nodi dimostrati), dall'altro il numero minore di nodi creati comporta un costo minore sia per la gestione della memoria che per *updateAncestors*;
- nei turni più avanzati, e soprattutto in quelli intermedi, si creano meno nodi ma se ne provano di più: dunque, essendoci più possibilità di creare *minacce*, c'è più lavoro per la *DB-search*.

6 Idee future

Molti altri miglioramenti sono stati presi in considerazione, ma non implementati o per mancanza di tempo o perché richiedono tempo per testarli, e non sempre dimostrano una maggiore efficienza. Eccone alcuni:

1. *bitboard*: se ne è sfruttata solo la compattezza, ma le operazioni potrebbero essere gestite in modo più efficiente grazie ai calcoli con i bit; inoltre, la tabella è di tipo *long*, ma potrebbe usare tipi di numeri diversi in base a *M* e *N*;
2. provare diverse inizializzazioni dei *proof-number*;
3. l'efficacia di un algoritmo e di un miglioramento introdotto dipendono anche da *M*,*N*, quindi se ne potrebbero implementare di più ed eseguire quello più opportuno di volta in volta;
4. controllare *trasposizioni* con *tabelle di trasposizione*, per esempio con *board* simmetriche;
5. continuare con le visite finché c'è tempo, invece di fermarsi quando si è dimostrata la radice, per trovare una vittoria/sconfitta migliore;
6. gestire in modo più efficiente i nodi che, al momento, vengono eliminati (con *removeUnmarkedTree*, o quando si dimostra un nodo, o all'applicazione di una nuova *DB-search*); per esempio, si potrebbe usare una struttura per non perderli del tutto, e poi eliminarli quando diventino troppi o troppo vecchi;
7. se una *DB-search* trova una *global defense* di *tier t*, nelle successive ricerche per sequenze vincenti non ha senso usare minacce di *tier* > *t*, perché verrebbero annullate dalla stessa *global defense*;
8. usare *transposition table* in *DB-search*;
9. nella combinazione di due *board A* e *B* in *DB-search*, ci possono essere differenze tra l'aggiungere le minacce di *A* a *B* e le minacce di *B* ad *A*: servirebbe un modo per decidere quale approccio sia più conveniente;
10. *iterated related squares* (da fonte 1): se *DB-search* trova una sequenza per *P2* in un nodo di *P2*, il suo nodo padre (di *P1*) dovrebbe considerare solo le mosse che coinvolgano tale sequenza, per lo stesso ragionamento fatto con le *implicit threat* (quindi eliminando i nodi non coinvolti o dandogli meno precedenza).

7 Fonti

Le seguenti fonti hanno ispirato significativamente lo sviluppo: i loro concetti, che spiegano applicati a vari ambiti, sono stati liberamente riadattati a questo progetto.

- **Fonte 1** http://fragrieu.free.fr/SearchingForSolutions.pdf: tesi che spiega PN-search e DB-search, per poi applicarli a vari giochi, e in particolare per risolvere il gioco del gomoku, ovvero un (M,N,K)-Game con valori 15,15,5 (e una restrizione sulle mosse).
- **Fonte 2** https://towardsdatascience.com/creating-the-perfect-connect-four-ai-bot-c165115557b0 e le guide presenti a http://blog.gamesolver.org/ (**Fonte 3**): utili per avere un'introduzione e delle prime idee sul gioco, ma soprattutto per l'idea della bitboard.