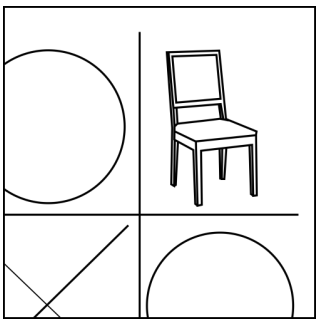

CadregaBot

Di Francesco Goretti (0000971634) e Matilde Mariano (0000970476)

Progetto di Algoritmi e Strutture Dati
AA 2021/2022

Alma Mater Studiorum - Università di Bologna

INDICE:

Introduzione:

Per costruire un algoritmo in grado di giocare ad (M, N, K) game si possono seguire diversi approcci, ognuno con i suoi punti di forza e con le sue peculiarità.
Per esempio si può puntare su tecniche per la valutazione statica della tabella di gioco, cercando di basare la scelta su una visione della tabella solo all'effettivo momento della valutazione; in genere questo tipo di algoritmi è molto veloce, ma avendo una visione così limitata può restituire risultati che non portano a qualcosa di buono a lungo termine.
Altrimenti si può optare per valutazioni più dinamiche, che costruiscono un albero contenente tutte le (o buona parte delle) mosse future e poi prendono una decisione valutandole tutte insieme, in questo modo si ha maggiore probabilità (se non in certi casi la certezza) di ottenere un risultato ottimo, tuttavia in questo modo l'algoritmo ha un costo computazionale parecchio alto.
Il metodo utilizzato in CadregaBot è un'ibridazione di questi due approcci progettato con lo scopo di sfruttare al meglio il tempo a disposizione, minimizzando nel mentre le possibilità che l'algoritmo non ritorni un risultato in tempo.
Per fare ciò viene applicata una prima euristica statica che è in grado di fornire un risultato in O(f*K) (con f numero di celle ancora libere e K numero di celle contigue per la vittoria), dopodiché viene controllato dinamicamente il tempo e, finché ne rimane, si procede alla costruzione ed alla visita dell'albero di gioco, verificando via via se la mossa scelta è la migliore o se ne esistono altre più promettenti (ed in questo caso si andrà poi a restituire la mossa migliore trovata).
Calcolare il costo computazionale di questo algoritmo non risulta quindi banale, in quanto esso decide quante istruzioni eseguire proprio in funzione del tempo che gli è dato per eseguirsi.
Per questioni di esposizione in questa relazione si tratteranno le varie funzioni come se non fossero soggette a limitazioni di tempo (si veda in appendice "Tabella costi computazionali") e si indicheranno esplicitamente quei passaggi in cui sarà invece necessario considerare ciò.

Notazioni e convenzioni:

Nella relazione vengono adottate alcune convenzioni:

- con f si indica il numero di celle libere nella tabella in un dato momento;
- con M, N e K si indicano rispettivamente righe, colonne e celle consecutive necessarie per vincere del gioco;
- con $-\infty$ e $+\infty$ si indicano rispettivamente un numero estremamente piccolo ed uno estremamente grande, ma non il concetto matematico di infinito, vengono infatti utilizzate notazioni del tipo $-\infty + 1$ o $+\infty - 1$ per indicare numeri diversi da $-\infty$ e $+\infty$ (cosa che non avrebbe senso utilizzando la concezione canonica);
- con “Alphabeta” si intende l’implementazione di CadregaBot dell’algoritmo Minimax con Alphabeta pruning;
- i nomi delle funzioni dello pseudocodice coincidono con quelli del codice effettivo per facilitarne l’identificazione, anche se ciò può portare a nomi poco intuitivi;
- nello pseudocodice tutte le variabili che non sono parametri formali o introdotte da un comando `let` sono da considerarsi globali.

Struttura dell'algoritmo:

CadregaBot utilizza un'euristica che permette di trovare una mossa buona (o in certi casi anche ottima) in un tempo O(f*K) (si veda il paragrafo "Euristiche").
Tuttavia, ci sono situazioni in cui essa fallisce: in caso l'avversario ricorra a "trucchetti" particolari, l'euristica non è in grado di prevederli e quindi la mossa fatta non evita la vittoria avversaria. Un esempio è il seguente (deve giocare la O):

Configurazione della tabella

X		
	O	
		X

Valori calcolati dall'euristica

X	4	6
4	O	4
6	4	X

La mossa scelta dall'euristica sarà quindi una delle due con valore 6, che tuttavia porterà inevitabilmente alla vittoria della X.
Si rende quindi necessario, dopo aver valutato la configurazione corrente, l'uso di un algoritmo aggiuntivo per evitare situazioni del genere. È stato quindi deciso di utilizzare un algoritmo Minimax con potatura Alphabeta.
Infine, è stato necessario considerare il limite temporale. L'algoritmo deve restituire una mossa entro un tempo prefissato, per cui sono state prese le seguenti misure per evitare che esso non ritorni:

- come anticipato nell'Introduzione, viene eseguita la valutazione euristica della configurazione della tabella prima di eseguire l'Alphabeta. Nel caso esso non trovi una mossa migliore entro il tempo massimo prefissato, viene ritornata la mossa migliore trovata dall'euristica. Anche se questa non fosse la mossa ottima, ritornarla permette di evitare l'immediata sconfitta a tavolino;
- l'implementazione dell'Alphabeta pruning tiene traccia della mossa migliore trovata mano a mano che viene eseguito. In caso l'algoritmo non dovesse terminare entro il tempo prefissato, viene comunque ritornata una mossa (potenzialmente) migliore di quella individuata dall'euristica;
- ogni volta che un nodo viene visitato, viene controllato il tempo rimanente e, in caso non ce ne sia più a disposizione, l'Alphabeta viene interrotto;
- ogni nodo visitato dall'Alphabeta viene salvato per il turno successivo. Questo permette, a scapito della memoria occupata, di evitare alcuni calcoli al turno successivo se l'avversario ha compiuto una mossa che era già stata valutata in precedenza. Questo trade-off premia la velocità d'esecuzione sulla memoria occupata, in quanto su di quest'ultima non sono presenti grandi limitazioni.
- infine, la profondità della visita viene calcolata ad ogni inizio turno basandosi su una media dei nodi visitati nei turni precedenti. Questo viene compiuto dalla funzione `optimizedDepth`. In questo modo si riesce a completare l'Alphabeta nella maggior parte dei casi. È comunque presente una profondità minima di visita impostata al valore di 3, in quanto scendere sotto tale profondità rende l'Alphabeta incapace di prevedere alcunché, non dando quindi alcun valore aggiunto all'euristica.

Quindi, per concludere, lo pseudocodice di `selectCell` è il seguente:

```
algorithm selectCell(FC, MC) -> Cell {
    nodesAverage = calculateNodesAverage(nodeCounter , nodesAverage);
    nodeCounter = 0; // Reset the node counter
    board.addMove(MC[MC.length - 1]); // Save opponent move

    // Create alphabeta tree root
    let root;
    if (oldRoot != null && oldRoot.contains(MC[MC.length - 1])) {
        // This line calls complexEvaluateTmpBoard if oldRoot doesn't contains
        // a child for the MC[MC.length - 1] move
        root = oldRoot.getOrCreateChild(MC[MC.length - 1]);
    } else {
        root = new Node(complexEvaluateTmpBoard(FC));
    }

    let depth = optimizedDepth(/*minimum depth*/ 3, FC.length, nodesAverage);
    bestMove = alphabetaStart(root, −∞, +∞, depth);

    board.addMove(bestMove); // Save CadregaBot move
    return bestMove;
}
```

Il costo computazionale verrà calcolato nelle conclusioni una volta ottenuto il costo delle altre funzioni.

Euristiche:
Per la valutazione statica della tabella di gioco sono stati utilizzati come base gli algoritmi `isWinning` e `simpleEvaluate`.
`isWinning` controlla se il giocatore può vincere in una mossa.
Invece `simpleEvaluate` a dispetto del nome è molto più complicato: il suo scopo è contare tutte le possibili configurazioni vincenti in cui si trova la cella di interesse (`cell`), sommando poi a tale quantità il numero di caselle già occupate per ognuna di tali configurazioni.
Per poter calcolare queste due funzioni è necessario considerare la riga, la colonna, la diagonale principale e la diagonale secondaria su cui si trova la cella; per questo motivo sono entrambe divise in quattro sotto-funzioni di cui si riporta lo pseudocodice soltanto delle verticali (per le altre il procedimento è analogo):

```
algorithm isWinningVertical(cell, player) -> boolean {
    let first = cell.i, last = cell.i;
    let counter = 1;
    let minFirst = max { first - K + 1, 0 };
    let maxLast = min { last + K - 1, M - 1 };
    while (first > minFirst) {
        if (board[first - 1][cell.j] == player) { counter++; }
        else { break; }
        first--;
    }

    while (last < maxLast) {
        if (board[last + 1][cell.j] == player) { counter++; }
        else { break; }
        last++;
    }
    return (counter >= K);
}
```

```
algorithm simpleEvaluateVertical(cell, player) -> natural {
    // cell.i is the cell's row, cell.j is the cell's column
    let first = cell.i, last = cell.i;
    let minFirst = max { first - K + 1, 0 };
    let maxLast = min { last + K - 1, M - 1 };

    while (first > minFirst && (board[first - 1][cell.j] == player ||
           board[first - 1][cell.j] == FREE)) {
        first--;
    }
    while (last < maxLast && (board[last + 1][cell.j] == player ||
           board[last + 1][cell.j] == FREE)) {
        last++;
    }

    // Assigning points
    let n = last - first + 1;
    if (n < K) { return 0; }
```

```
    let maxN = n - K + 1, max = 0, eval = maxN;
    while (first < cell.i) {
        if (max < maxN) { max++; }
        if (board[first][cell.j] == player) { eval += max; }
        first++;
    }
    max = 0;
    while (last > cell.i) {
        if (max < maxN) { max++; }
        if (board[last][cell.j] == player) { eval += max; }
        last--;
    }
    return eval;
}
```

Si può notare che i passaggi di `simpleEvaluateVertical` sono poco intuitivi, ma è stato necessario porli in questa forma per minimizzare il numero di letture della colonna e conseguentemente il costo computazionale.
Questa funzione in una prima fase ricerca gli estremi entro cui può essere utile leggere le celle (per esempio, se k = 5 non ha senso cercare a 10 caselle di distanza, né ha senso continuare a cercare se è già stata incontrata una casella avversaria), mentre l'assegnazione del punteggio avviene successivamente.
Nella seconda fase invece vengono fatti entrambi i conteggi in un'unica passata: nella variabile `maxN` viene calcolato il numero di configurazioni vincenti che passano per `cell`, mentre nel ciclo successivo si controlla se nell'area definita nella prima fase sono già presenti celle del giocatore e nel caso si somma al punteggio (inizialmente `maxN`) il numero di volte che quella casella amica compare nelle configurazioni favorevoli (contenuto nella variabile `max`).

Queste due funzioni non sono utilizzate direttamente, ma vengono richiamate da altre il cui scopo è fornire una valutazione della tabella di gioco in vari contesti:
`simpleEvaluate` viene utilizzata in `simpleEvaluateTmpBoard`, la quale è usata durante l'esecuzione di Alphabeta per valutare un nodo nel caso sia stata raggiunta la profondità massima (si veda il paragrafo "Alphabeta").
Invece come euristica per l'ordinamento delle celle (si veda il paragrafo "Alphabeta") e come euristica che rende l'algoritmo in grado di ritornare in ogni situazione viene utilizzata la funzione `complexEvaluateTmpBoard` che a sua volta sfrutta contemporaneamente `isWinning` e `simpleEvaluate`.

Nota bene: sempre per ottimizzazione, al posto di invocare le due funzioni una dopo l'altra ne è stata creata una terza che calcola entrambe le cose contemporaneamente, tale funzione è chiamata `evaluate`.

Le funzioni `simpleEvaluateTmpBoard` e `complexEvaluateTmpBoard` hanno la seguente struttura:

```
algorithm simpleEvaluateTmpBoard(cells, player, opponent) -> integer {
    let value = 0;
```

```
    for cell in cells {
        value += simpleEvaluate(cell, player);
        value -= simpleEvaluate(cell, opponent);
    }
    return value;
}
```

```
algorithm complexEvaluateTmpBoard(cells, player) -> EvaluatedCell[] {
    let evaluatedCells = new EvaluatedCell[cells.length];

    for cell in cells {
        let evalPlayer = evaluate(cell, player);
        let evalOpponent = evaluate(cell, player.opponent);
        if (evalPlayer.isWin()) {
            // If the player wins then the move has +∞ value
            return new EvaluatedCell[1].add(cell, +∞);
        } else if (evalOpponent.isWin()) {
            // The opponent is winning, check if it's possible to beat it
            for cell remaining in cells {
                if (isWinning(cell, player)) {
                    return new EvaluatedCell[1].add(cell, +∞);
                }
            }
            // The opponent will win, block it
            //
            // If the opponent wins then the move has +∞ - 1 value, this way
            // this move has a lower value than the ones which make the player
            // wins
            return new EvaluatedCell[1].add(cell, +∞ - 1);
        } else {
            evaluatedCells.add(cell, evalPlayer  + evalOpponent);
        }
    }

    radixSort(evaluatedCells);
    return evaluatedCells;
}
```

Ovvero `simpleEvaluateTmpBoard` per ogni cella della tabella somma al punteggio complessivo `simpleEvaluate` del giocatore e sottrae quello dell'avversario, restituendo poi un punteggio che valuta l'intera tabella (più il valore della configurazione è alto più essa è favorevole per il giocatore, più è basso e più la configurazione è buona per l'avversario); `complexEvaluateTmpBoard` al contrario assegna un valore ad ogni singola cella passata sommando il punteggio che ottiene giocando come giocatore a quello che ottiene giocando come avversario (una mossa è utile se porta ad un buon risultato per se stessi e/o se impedisce all'avversario di fare una buona mossa).

Per quanto riguarda il costo computazionale `isWinning` esegue 4 volte funzioni analoghe a `isWinningVertical`, la quale nel caso pessimo visita 2K celle (con operazioni O(1)); quindi il suo costo computazionale è O(8K) = O(K).
`simpleEvaluateVertical` nel caso pessimo visita 2K celle (con operazioni O(1)) nella prima fase e 2K nella seconda, quindi siccome anche `simpleEvaluate` esegue 4 volte funzioni analoghe ad essa, il costo computazionale totale risulta O(16K) = O(K).
Anche `evaluate` ha costo computazionale O(K), in quanto è una versione leggermente ottimizzata dell'invocazione di entrambe le funzioni e quindi è O(K).
`simpleEvaluateTmpBoard` invoca due volte `simpleEvaluate`, quindi il suo costo computazionale è O(K).
Infine `complexEvaluateTmpBoard` richiama `evaluate` 2 volte per ogni cella libera e di conseguenza il suo costo diventa O(f*K).

Alphabeta:
L'algoritmo utilizzato è una variante del Minimax con Alphabeta pruning:

- per evitare calcoli ripetuti, viene salvato l'albero generato durante la visita (sono ovviamente generati e salvati solo i nodi effettivamente visitati dall'algoritmo) e, all'inizio del turno successivo, viene ricercata la mossa compiuta dall'avversario nell'albero salvato: se presente, si evita di ricalcolare l'euristica sulle configurazioni già calcolate. I rami dell'albero che vengono scartati da questa ricerca vengono eliminati per liberare memoria. Infine, durante la visita, vengono generati i nodi mancanti, ovvero quelli non visitati nei turni precedenti;
- viene verificato il tempo rimanente tramite la funzione `checkTime`, la quale termina l'esecuzione dell'algoritmo in caso il tempo a disposizione sia terminato;
- per rendere possibile il calcolo di `optimizedDepth`, viene mantenuto il conteggio dei nodi visitati durante l'esecuzione dell'Alphabeta.

Si riporta di seguito lo pseudocodice:

```
algorithm alphabeta(node, alpha, beta, depth, player) -> integer {
    checkTime(); // Halt Alphabeta if the time is up
    nodeCounter = nodeCounter + 1; // Keep track of the count of analyzed nodes

    if (node.isLeaf()) {
        if (node.isDraw()) { return 0; }
        else if (player == CadregaBot) { return (+∞ - node.depth); }
        else { return (-∞ + node.depth); }
    }
    if (depth == 0) return simpleEvaluateTmpBoard(node.getSortedCells());

    if (player == CadregaBot) {
        let value = -∞;
        for cell in node.getSortedCells() {
            let child = node.getOrCreateChild(cell, player);
            let alphabetaValue = alphabeta(child, alpha, beta, depth - 1,
                                           player.opponent);
            value = max { value, alphabetaValue };
            alpha = max { value, alpha };
            if (beta <= alpha) { break; }
        }
        return value;
    } else {
        let value = +∞;
        for cell in node.getSortedCells() {
            let child = node.getOrCreateChild(cell, player);
            let alphabetaValue = alphabeta(child , alpha, beta, depth - 1,
                                           player.opponent);
            value = min { value, alphabetaValue };
            beta = min { value, beta };
            if (beta <= alpha) { break; }
        }
        return value;
    }
}
```

```
algorithm simpleEvaluateTmpBoard(cells) -> integer {
    let value = 0;
    for cell in cells {
        value += simpleEvaluate(cell, CadregaBot);
        value -= simpleEvaluate(cell, CadregaBot.opponent);
    }
    return value;
}
```

Nota bene: nel codice vero e proprio non esiste la funzione `getOrCreateChild` (utilizzata nel codice di Alphabeta): al suo posto vengono utilizzate le funzioni `Node#addChild`, `Node#getChildren` e `CadregaBot#complexEvaluateTmpBoard`.
L'uso di tale funzione è stato fatto per chiarezza e compattezza dello pseudocodice.

La chiamata ricorsiva si interrompe quando:

- si ha un pareggio, in tal caso il valore ritornato è 0;
- vince CadregaBot, in tal caso viene ritornato `+∞ - profonditàNodo`. Viene sottratta la profondità del nodo all'interno dell'albero di gioco per dare un punteggio migliore alle mosse che portano ad una vittoria più velocemente;
  Nota bene: all'interno del codice vero e proprio invece che utilizzare la profondità del nodo all'interno dell'albero di gioco si utilizza `startingDepth - depth`, cioè la profondità relativa al primo nodo analizzato durante il turno. Questo permette di non dover tener traccia della profondità assoluta dei nodi.
- vince l'avversario, in tal caso viene ritornato `-∞ + profonditàNodo`. Viene sommata la profondità del nodo all'interno dell'albero di gioco sempre per il motivo spiegato pocanzi;
- `depth` raggiunge 0, in tal caso la visita viene interrotta e si utilizza l'euristica per valutare la configurazione della tabella raggiunta. Viene utilizzata a questo scopo la funzione `simpleEvaluateTmpBoard` la quale, invece che ritornare un array di celle ordinate come `complexEvaluateTmpBoard`, ritorna direttamente la valutazione euristica della configurazione. Infatti, per ogni cella libera, viene sommata la valutazione euristica della mossa per il giocatore massimizzante (CadregaBot) e sottratta quella del giocatore minimizzante (l'avvversario).

Il costo di `getOrCreateChild` è lo stesso di `CadregaBot#complexEvaluateTmpBoard`, in quanto la ricerca (che viene eseguita passando l'indice della cella all'interno dell'array di celle ordinate) e l'aggiunta di un figlio sono eseguite in tempo costante. Perciò, il costo di tale operazione è O(f*K), dove f è il numero di celle libere all'inizio dell'esecuzione dell'Alphabeta.

Siccome l'Alphabeta analizza nel caso pessimo $O(\frac{f!}{(f - depth)!})$ nodi e per ogni nodo sono fatte O(f*K) operazioni, allora nel caso pessimo si ha costo $O(f * K * \frac{f!}{(f - depth)!})$.

Nel caso ottimo, invece, si ottiene $O(f * K * \sqrt{\frac{f!}{(f - depth)!}})$.

Optimized depth:

Rimane però ancora il delicato problema di decidere come calcolare la profondità da raggiungere durante la visita dell'albero: una profondità troppo bassa porterebbe a scarsa capacità di previsione, mentre una troppo alta porterebbe a dover troncare la visita e darebbe una visione solo parziale delle possibili soluzioni.
Risulta quindi necessario avere un algoritmo (`optimizedDepth`) in grado di decidere (velocemente, in questo caso in O(f)) quale profondità sia ottimale per una specifica visita.
Sperimentalmente è stato provato che, grazie all'ordinamento con `complexEvaluateTmpBoard` delle celle dei figli di ogni nodo, le prestazioni di Alphabeta sono molto più vicine al caso ottimo che al caso medio; per questo motivo nel corpo di `optimizedDepth` si suppone sempre che i nodi visitati siano $\sqrt{\text{freeCells!}}$.
La funzione quindi risulta:

```
algorithm optimizedDepth(startingDepth, freeCells, nodesToVisit) -> natural {
    let depth = startingDepth;
    let visitedNodes = sqrt(freeCells);
    for (i = 1; i < startingDepth && freeCells > 0; i++) {
        freeCells--;
        visitedNodes = visitedNodes * sqrt(freeCells);
    }

    while (visitedNodes < nodesToVisit) {
        freeCells--;
        if (freeCells <= 0) {
            return depth;
        }
        visitedNodes = visitedNodes * sqrt(freeCells);
        depth++;
    }

    return max { startingDepth, depth - 2 };
}
```

Dove `startingDepth` è la profondità minima della visita, `freeCells` sono le celle libere al momento della chiamata della funzione e `nodesToVisit` sono i nodi che l'Alphabeta riesce a visitare nel tempo prefissato; quest'ultimo valore viene calcolato facendo una media dei nodi visitati nei round precedenti.

Un punto del codice sul quale può valere la pena di spendere delle parole è l'ultima riga: `return max { startingDepth, depth - 2 };`.
Il massimo tra `startingDepth` e `depth - 2` viene calcolato poiché la profondità di visita non può scendere sotto `startingDepth`.
È invece più interessante il motivo per cui è necessario fare `depth - 2`: è necessario sottrarre 1 alla profondità in quanto alla fine del ciclo while ci si ritrova in una situazione in cui `visitedNodes > nodesToVisit`, quindi è necessario "tornare indietro" di un'iterazione; l'altra sottrazione di un'unità è invece giustificata dal fatto che in realtà i nodi visitati non sono gli stessi del caso ottimo, ma sono leggermente superiori, risulta quindi utile fermarsi ad una profondità inferiore per avere del tempo margine per calcolare i nodi in eccesso.

Conclusioni:
Si può a questo punto calcolare il costo computazionale di `selectCell`.
Senza considerare limiti di tempo si ottiene che l'algoritmo costa O(f*K*f!). All'inizio del gioco si ha quindi un costo pari a O(M*N*K*(M*N)!). Ciò si riduce drasticamente quando si considera che nell'Alphabeta c'è una profondità massima.
In questo caso il costo diventa $O(^{f * K * \frac{f!}{(f - depth)!}})$ nel caso pessimo e $O(^{f * K * \sqrt{\frac{f!}{(f - depth)!}}})$ nel caso ottimo.

Si ricorda comunque che nella realtà il costo è limitato dal tempo a disposizione in quanto depth è calcolato in base ad esso e l'algoritmo attraverso delle eccezioni è in grado di ritornare in un qualunque momento della sua esecuzione.

APPENDICE:

Informazioni utili alla comprensione del codice:

- le celle libere sono state mantenute in una tabella hash (`HashSet`) in modo da poterle aggiungere/rimuovere in O(1). Questo è particolarmente utile durante l'Alphabeta, in quanto durante la visita le celle vengono occupate e liberate molteplici volte. Inoltre, tenere le celle libere in una struttura dati iterabile permette di evitare la scansione di tutta la tabella, concentrandosi solo sulle celle effettivamente libere;
- `checkTime` lancia una `RuntimeException` quando il tempo rimanente scende sotto una certa soglia (viene tenuto un margine di mezzo secondo) che viene poi catturata all'interno di `selectCell`;
- tutti i calcoli vengono fatti su una tabella temporanea (chiamata `tmpBoard`) in modo da non dover modificare direttamente la tabella vera e propria (chiamata `board`). Questo rende possibile l'utilizzo delle eccezioni per terminare l'esecuzione dell'Alphabeta quando finisce il tempo a disposizione senza dover effettuare ulteriori calcoli.
  Ad ogni inizio turno viene copiata la `board` nella `tmpBoard`.

initPlayer:

Durante l'inizializzazione vengono impostate al valore iniziale tutte le variabili utilizzare dall'algoritmo e viene invocato `selectCell` su una configurazione fittizia in modo da avere già, durante il primo round, il conteggio dei nodi visitati al round precedente inizializzato. In questo modo è possibile sfruttare anche al primo round il calcolo di `optimizedDepth`.

Costi computazionali:

Funzione:	**Costo computazionale (caso pessimo):**
`CadregaBot#alphabeta`	O(f*K*f!) dove f è il numero di celle libere (non si sta considerando la profondità massima di visita)
`CadregaBot#alphabetaStart`	O(f*K*f!) dove f è il numero di celle libere (non si sta considerando la profondità massima di visita)
`CadregaBot#checkTime`	Θ(1)
`CadregaBot#complexEvaluateTmpBoard`	O(f*16K + f) = O(f*K + f) = O(f*(K+1)) = O(f*K) (siccome K cost. > 0) dove f è il numero di celle libere
`CadregaBot#copyTmpBoard`	Θ(N*M)
`CadregaBot#saveMove`	Θ(1)
`CadregaBot#selectCell`	O(f*K*f!) dove f è il numero di celle libere (non si sta considerando la profondità massima di visita durante l'Alphabeta). All'inizio del gioco si ha quindi O(M*N*K*(M*N)!)
`CadregaBot#simpleEvaluateTmpBoard`	O(f*32K) = O(f*K) dove f è il numero di celle libere
`EvaluatedCell#getCell`	Θ(1)
`EvaluatedCell#getValue`	Θ(1)
`EvaluateUtil#evaluate`	O(16K) = O(K)
`EvaluateUtil#evaluateHorizontal`	O(4K) = O(K)
`EvaluateUtil#evaluateInvertedDiagonal`	O(4K) = O(K)
`EvaluateUtil#evaluateMainDiagonal`	O(4K) = O(K)
`EvaluateUtil#evaluateVertical`	O(4K) = O(K)
`EvaluateUtil#isWinningCell`	O(8K) = O(K)
`EvaluateUtil#isWinningHorizontal`	O(2K) = O(K)

`EvaluateUtil#isWinningInvertedDiagonal`	O(2K) = O(K)
`EvaluateUtil#isWinningMainDiagonal`	O(2K) = O(K)
`EvaluateUtil#isWinningVertical`	O(2K) = O(K)
`EvaluateUtil#simpleEvaluate`	O(16K) = O(K)
`EvaluateUtil#simpleEvaluateHorizontal`	O(4K) = O(K)
`EvaluateUtil#simpleEvaluateInvertedDiagonal`	O(4K) = O(K)
`EvaluateUtil#simpleEvaluateMainDiagonal`	O(4K) = O(K)
`EvaluateUtil#simpleEvaluateVertical`	O(4K) = O(K)
`Node#addChild`	Θ(1)
`Node#getCell`	Θ(1)
`Node#getChildren`	Θ(1)
`Node#getParent`	Θ(1)
`Node#setParent`	Θ(1)
`Node#getSortedCells`	Θ(1)
`Node#selectChildByMove`	O(n) dove n è il numero di figli del nodo
`OptimizedDepth#optimizedDepth`	O(f) dove f è il numero di celle libere
`SortUtil#bucketSort`	O(n) dove n è il numero di elementi da ordinare
`SortUtil#digitCount`	Θ(1)
`SortUtil#getDigit`	Θ(1)
`SortUtil#radixSort`	O(n) dove n è il numero di elementi da ordinare

Sitografia:

L’euristica utilizzata è stata tratta dal seguente documento:
http://www.micsymposium.org/mics2016/Papers/MICS_2016_paper_28.pdf