Tablut - Team “Franco”

Overview - scelte architetturali

Linguaggio di programmazione utilizzato: Kotlin.

Motivazioni:

- Conciso e pratico
- Interoperabile
- Multipiattaforma
- Sviluppo futuro di Tablut per Android?

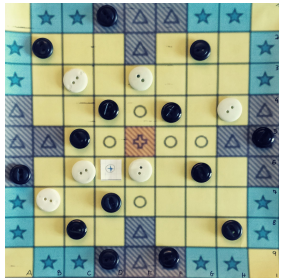
Progettazione

Utilizzo di *design patterns* e *SOLID principles* per rendere il software riutilizzabile, flessibile, manutenibile e modulare.

Supporto di diverse versioni del gioco grazie a:

- Implementazione di interfacce predefinite e utilizzo di Factory method pattern
- Separazione della rappresentazione del dominio dall’algoritmo di ricerca
- Possibilità di leggere la board tramite file di configurazione

Euristica



Ricerca di euristiche tramite il “learn-by-doing”.

Trade-off tra complessità euristica e nodi dell’albero di ricerca esplorati.

Euristica adottata:

- Bianco: distanza di Manhattan (distanza del re dall’uscita libera più vicina)
- Nero: numero di cavalieri neri rimanenti per la cattura del re (in base alla sua posizione)

Dettagli implementativi

Rappresentazione stato: NxN (9x9) celle, giocatore attuale.

Algoritmo di ricerca nello spazio degli stati: IterativeDeepeningAlphaBetaSearch.

Librerie utilizzate:

- AIMA
- GSON

Sviluppi futuri

Performance:

- Kotlin coroutines, multithreading
- Sfruttamento delle simmetrie del gioco

Euristica:

- Sviluppo di euristiche più complesse
- Machine/deep learning

UX:

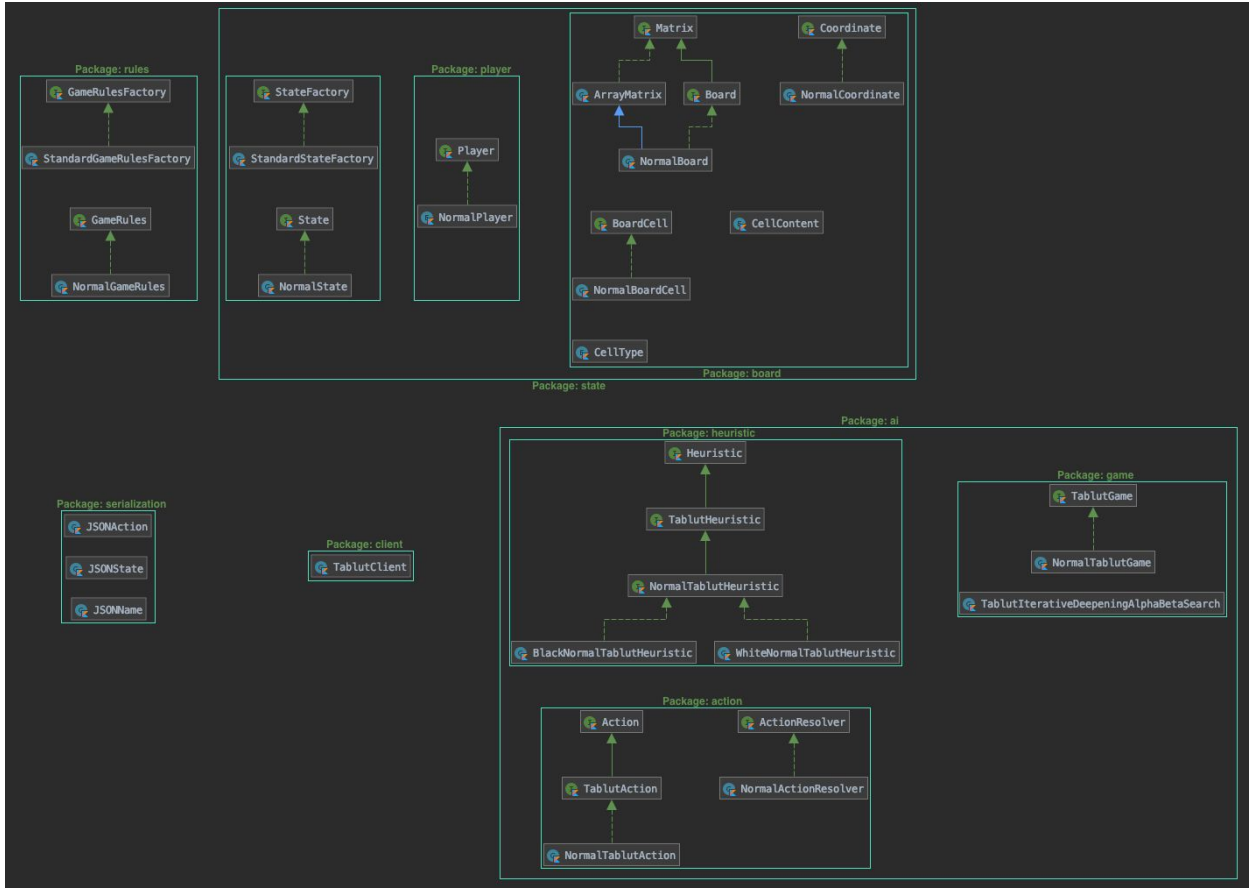
- Tablut per Android

Grazie dell’attenzione

Q&A

UML

UML snello: no associazioni, campi, metodi.



Euristica: bianco

```
class WhiteNormalTablutHeuristic : NormalTablutHeuristic {
    override fun getValue(state: State, player: Player): Double {
        return distanceFromNearestExit(state as NormalState, player as NormalPlayer)
    }

    fun distanceFromNearestExit(state: NormalState, player: NormalPlayer): Double{
        var min = 20.0
        var res = 0.0
        var dist: Double

        val exits = state.board.getExitBoardCells()
        for(exit in exits) {
            if (exit.content == CellContent.NOTHING) {
                dist = calculateDist(state.board.getKingBoardCell(), exit)
                if (dist < min) {
                    min = dist
                }
            }
        }

        val m = floor(min).toInt()
        when(m){
            4 -> res = -0.9
            3 -> res = -0.5
            2 -> res = 0.0
            1 -> res = 0.5
            0 -> res = 0.9
        }

        return res
    }

    fun calculateDist(king: NormalBoardCell, exit: NormalBoardCell): Double{
        return sqrt( x: (king.coordinate.x-exit.coordinate.x).toDouble().pow( n: 2)+(king.coordinate.y-exit.coordinate.y).toDouble().pow( n: 2))
    }
}
```

Euristica: nero

```
class BlackNormalTablutHeuristic : NormalTablutHeuristic {
    override fun getValue(state: State, player: Player): Double{
        return numberOfPawnsToCaptureKing(state as NormalState, player as NormalPlayer)
    }

    fun numberOfPawnsToCaptureKing(state: NormalState, player: NormalPlayer): Double{
        val surrounding = state.board.getBlackBoardCellAdjKing().size
        var res = 0.0


        if(state.board.getKingBoardCell().coordinate.equals(NormalCoordinate.getMiddleCoordinate())){
            when(surrounding){
                4 -> res = 0.9
                3 -> res = 0.5
                2 -> res = 0.0
                1 -> res = -0.5
                0 -> res = -0.9
            }
        }else if(NormalCoordinate(NormalCoordinate.getMiddleCoordinate()).adjCoordinates().contains(state.board.getKingBoardCell().coordinate)){
            when(surrounding){
                3 -> res = 0.9
                2 -> res = (1.toDouble()/3.toDouble())
                1 -> res = -(1.toDouble()/3.toDouble())
                0 -> res = -0.9
            }
        }else{
            when(surrounding){
                2 -> res = 0.9
                1 -> res = 0.0
                0 -> res = -0.9
            }
        }

        return res
    }
}
```

Stato

```
data class NormalState(override var board: NormalBoard<NormalBoardCell>, override var player: NormalPlayer) : State{
    override fun toString(): String {
        return "NormalState(board=\n$board, player=$player)"
    }
}
```

Factory method pattern - Stato

```
class StandardStateFactory : StateFactory {
    override fun createFromGameVersion(version: String, boardTypePath: String, boardContentPath: String): State {
        return when(version){
            "Normal" -> NormalState(NormalBoard<NormalBoardCell>( rows: 9,  cols: 9, boardTypePath, boardContentPath), NormalPlayer.WHITE)
            //"Brandubh" -> BrandubhState()
            //"Classic" -> ClassicState()
            //"Modern" -> ModernState()
            else -> throw Exception("State not found")
        }
    }
}
```