Matematicko-fyzikální fakulta Univerzity Karlovi

Programátorská dokumentace k programu Šachy

Letní semestr 2023/24

Adam Řeřicha, II. ročník bakalářského studia

Obsah

1 Anotace 2
2 Program 3
2.1 Hlavní třídy a rozhraní programu 3
2.1.1 IPiece 3
2.1.2 Board 3
2.1.3 ChessRules 4
2.1.4 ChessAI 4
2.1.5 Program 4
2.2 Minimax 5
2.3 Hra přes síť 5
2.3.1 Komunikační protokol 5
3 Průběh prací 5
4 Závěr 6

1 Anotace

Tento program je vytvořen pro hraní šachů v konzoli ve dvou hráčích na jednom počítači, na dvou počítačích připojených ke stejné síti a v jednom hráči proti umělé inteligenci využívající paralelní verzi algoritmu minimax. Program podporuje všechna základní pravidla hry, a i braní mimochodem, rošádu a proměnu pěšáka. Díky tomu, že se hrají v konzoli, tak je to paměťově nenáročný s minimalistickým vzhledem a je určen pro všechny, kteří se zajímají o šachy.

2 Program

V této části popisuji hlavní třídy a rozhraní programu, jak spolu pracují a komunikují a celkově jak vypadá struktura celého programu.

2.1 Hlavní třídy a rozhraní programu

Zde popisuji hlavní třídy a rozhraní programu.

2.1.1 IPiece

Toto rozhraní implementuje každá třída figurky (`Pawn`, `Knight`, `Bishop`, `Rook`, `Queen`, `King`) a také „nefigurky“, které reprezentují volná políčka (`NoPiece`, `EnPassant`). `EnPassant` dědí od `NoPiece`. Má tyto členy:

- `bool IsWhite { get; init; }`

  Informace, jestli je figurka bílá (jinak je považována za černou)

- `char Symbol { get; }`

  Symbol figurky, který se bude vypisovat na šachovnici

- `bool Moved { get; set; }`

  Informace, jestli se daná figurka už tuto hru pohnula (pro určování toho, jestli se může zahrát rošáda, nebo jestli pěšec může jít o dvě políčka dopředu)

- `int Value { get; }`

  Hodnota figurky

- `IEnumerable<(int, int)> MoveVectors { get; }`

  Vektory tahů pro pohyb figurky (! Z implementačních důvodů musí být seřazeny od nejmenších po největší vektory. Platí i pro `AttackVectors`)

- `IEnumerable<(int, int)> AttackVectors { get; }`

  Vektory tahů pro útok figurky (hlavně kvůli pěšci, který má jiné tahy pro pohyb a útok)

2.1.2 Board

Tato třída reprezentuje šachovnici. Umí vypsat šachovnici do konzole (`Print()`) a komunikuje s `ChessRules` o platnosti tahu a provedení tahu (`TryMakeMove()`). Má tyto veřejné členy:

- `ImmutableDictionary<Square, IPiece> Squares`

  Reprezentuje políčka šachovnice

- `ImmutableDictionary<char, List<Square>> WhitePieces`
- `ImmutableDictionary<char, List<Square>> BlackPieces`

  Slovníky pro zapamatování pozic všech figurek na šachovnici pro rychlejší hledání krále nebo hledání figurek, které útočí na krále. Každá figurka ve slovníku je reprezentována svým symbolem.

- `Square? EnPassantSquare`

  Políčko, na které může zaútočit pěšec a provést braní mimochodem.

2.1.2.1 bool TryMakeMove(string text, bool whitePlaying)

Vrací `true`, pokud tah zadaný textem (`text`) je platný a byl proveden. Jinak vrací `false`. Tato metoda se ptá na validní tahy od `ChessRules` a mění `text` na `Move`. `Move` je struktura, která je dvojice políček `From` a `To`. Pokud validní tahy obsahují `Move` z `text`, tak je tah validní, provede se tah a metoda vrací `true`.

2.1.3 ChessRules

Tato třída je stavební kámen celého programu. Zná pravidla šachů, vytváří novou pozici šachovnice podle tahu (`MakeMove()`), vyhodnocuje šachovnici (`EvaluateBoard()`, určuje, zda nastal šachmat nebo remíza), `ChessAI` a hlavní program se dotazují na možné tahy (`GetAvailableMoves()`). Všechny metody jsou statické.

2.1.3.1 List<Move> GetAvailableMoves(Board board, bool white)

Vrací všechny tahy, které jsou s šachovnicí `board` s hráčem barvy `white` (bílý, pokud `white` je `true`) možné. Hledá figurky, které se nesmí pohnout z důvodu, že brání krále od šachu. Dále zjišťuje, jestli může zahrát rošádu. Pokud je král v šachu, tak se vrátí jen ty tahy, které zabraňují šachu.

2.1.3.2 Board MakeMove(Move move, Board board)

Vrací novou instanci šachovnice, ve které se provedl tah `move` na předané šachovnici `board`. Pozná, jaký to je tah (proměna pěšce, rošáda, braní mimochodem, nebo běžný tah) a provede ho. `Board` má políčka jako `ImmutableDictionary` (kvůli paralelizaci minimaxu), takže se vytváří nová políčka a také se vytváří nové pozice figurek (`WhitePieces`, `BlackPieces`).

2.1.4 ChessAI

Třída pro šachovou umělou inteligenci implementující paralelní minimax pro určení nejlepšího tahu. Prohledává do hloubky dva kvůli náročnosti prohledávání všech možných šachových tahů. Metoda, která vyhodnotí nejlepší tah paralelně je `ChooseBestMove()`. První vrstvu minimaxu si paralelně rozdělí, aby se větvě vypočítávaly naráz. Pro získání možných tahů a provedení tahů komunikuje `ChessAI` s `ChessRules`.

2.1.5 Program

`Program` je hlavní třída programu. Je zde funkce `Main()` a rozhoduje se, jaký herní mód se spustí podle výběru uživatele (`OfflineMultiplayer()`, `SoloAI()`, `NetworkMultiplayer()`). Po vybrání se vytvoří šachovnice, spustí se hlavní smyčka hry a dotazuje se na tahy a zobrazuje šachovnici. Komunikuje s `ChessRules` o platných tazích a podle toho reaguje (vypíše chybovou hlášku, nebo napíše, že je na řadě druhý hráč).

2.2 Minimax

Minimax algoritmus je klíčová metoda používaná při rozhodování v tahových hrách, jako jsou šachy nebo piškvorky, kde se dva hráči střídají. Cílem tohoto algoritmu je najít optimální tah pro hráče, který je na tahu, za předpokladu, že protihráč bude také hrát optimálně.

Minimax pracuje na principu rekurze a procházení herního stromu, kde každý uzel představuje herní situaci (stav hry) a každá hrana tah, který přechází do jiného stavu hry. Pokud je na tahu Max, algoritmus si vybere potomka s maximální hodnotou. Pokud je na tahu Min, algoritmus si vybere potomka s minimální hodnotou.

2.3 Hra přes síť

Hlavní program při vybrání herního režimu LAN multiplayer funguje jako server. Druhý program ChessClient se připojuje k serveru jako klient pomocí zadané IP adresy. Pro připojení se používá TcpListener a TcpClient.

2.3.1 Komunikační protokol

Klient neví nic o tom, jaká jsou pravidla šachu. Posílá svůj tah zadaný uživatelem na server a ten mu odpoví šachovnicí, anebo mu přijde zpráva ‚invalid' a uživatel na klientovi musí znovu napsat tah.

Na začátku komunikace musí určit barvu figurek, kterou bude hrát. Klientova barva se pak pošle jako ‚W' anebo ‚B'. Poté už začíná hra.

Když odehraje server, tak pošle klientovi stav šachovnice ve tvaru FEN (Forsyth-Edwards Notation), políčko, na které se pohnula figurka (pro zvýraznění posledního tahu), a pokud je konec hry, tak se ještě pošle buď ‚WIN', ‚LOSS', nebo ‚DRAW'. Toto všechno je jeden řetězec oddělený mezerami.

[šachovnice_ve_tvaru_fen] [políčko_na_které_se_pohnula_figurka] [vyhodnocení_šachovnice]

Toto klient přijme, vypíše šachovnici do konzole a kdyžtak ukončí hru. Když uživatel na klientovi zadá tah, tak se pošle na server jen tento tah a čeká se na validaci od serveru.

3 Průběh prací

Na začátku jsem si neuvědomoval, kolik práce to dá jen naprogramovat samotné šachy. Strávil jsem tím více času, než jsem plánoval. Také jsem to podcenil při návrhu programu. Měl jsem navrhováním strávit o něco více času, abych si ušetřil čas přepisováním kódu.

Ze začátku jsem také myslel, že budu filtrovat políčka pomocí LINQ, ale to jsem rychle zavrhl kvůli vysoké neefektivitě. Kdybych chtěl najít třeba jen jednu konkrétní figurku nebo políčko, tak bych musel projít celou šachovnici. Proto jsem přešel na doporučení pana cvičícího k `ImmutableDictionary`, který se hodí pro paralelizaci a políčko najdu asymptoticky konstantně.

Dále byl velký problém rychlost minimaxu. Možných tahů je mnohem více, než jsem si představoval a tím, že strom minimaxu roste exponenciálně, tak jde projít méně vrstev v rozumném čase, než jsem plánoval. Takže jsem musel optimalizovat. Nestačilo posílat jen políčka, ale musím posílat i pozice figurek, aby se nemuseli procházet všechna políčka, když se například hledá král pro vyhodnocení šachovnice.

Pokud bych někdy programoval šachy znovu, tak bych zkusil reprezentovat šachovnici jenom pozicemi figurek, protože prázdná místa si pamatovat nemusím.

V tomto programu, když se hledají možné tahy, tak se zkontrolují, jestli jsou platné. Například, když hledám možné tahy pro krále, tak se na to políčko podívám a hledám, jestli na něj nekouká nepřítel nebo jestli je na tomto políčku nepřítel, tak jestli je chráněný anebo jestli ho král může vzít. Dělám to tímto způsobem, protože jsem si myslel, že zkoušení tahů a kontrolování, jestli král je v šachu, by bylo moc neefektivní kvůli tomu, že bych pro každý tah musel vytvořit novou šachovnici, protože mám políčka jako `ImmutableDictionary`. Ale možná to všechno počítání navíc je náročnější než průchod 64 políček dvakrát.

Kdybych také nepoužíval reflection při vybírání figurky pro tah, tak by zlepšilo efektivitu, protože reflection je výkonnostně náročná.

4 Závěr

Na závěr jsem na sebe pyšný, že jsem to dotáhl až do konce. Vytvořit takto velký projekt je pro mě úspěch. Naučil jsem se z toho hodně v oblasti návrhu programu, síťování, používání reflection a naprogramování si protihráče je pro mě něco úžasného, i když nehraje nejlépe. Šachy hraju rád, a i když to bylo pro mě časově i psychicky náročné, tak jsem rád, že jsem pracoval na tomto programu.