Laboratorium 2 - Gry dwuosobowe (connect4)

Marcin Pietrzykowski

mpietrzykowski@wi.zut.edu.pl

wersja 1.03

Algorytm Min-Max, Przycinanie alfa-beta i program connect4

1 Cel zadania

Celem zadania jest zapoznanie się za algorytmami **Min-Max**, **przycinanie alfa-beta** oraz napisanie w języku C# programu **Connect4 (czwórki)**. W celu wykonania zadania należy pobrać rozwiązanie Visual Studio zwierające klasy bazowe z implementację algorytmu **przycinanie alfa-beta**. Klasy bazowe należy pobrać stąd. W pobranym archiwum znajduje się katalog rozwiązania (*ang. Solution*) Visual Studio. Struktura plików oraz sposób implementacji jest analogiczny do zadania poprzedniego przedstawionego na Laboratorium 1.

1.1 IState.cs

Interfejs zawierający w sobie deklaracje metod i właściwości używanych w klasie `AlphaBetaSearch.cs`. **Nie należy modyfikować tego pliku.**

1.2 State.cs

Klasa abstrakcyjna dziedzicząca po interfejsie `IState.cs` zawierający w sobie częściową implementację metod i właściwości używanych przez `AlphaBetaSearch.cs`. **Nie należy modyfikować tego pliku.**

1.3 AlphaBetaSearch.cs

Klasa abstrakcyjna implementująca algorytm **Przycinanie alfa-beta**. W trakcie działania operuje na stanach implementujących interfejs `IState.cs`. **Nie należy modyfikować tego pliku.**

1.4 Program.cs

Plik główny programu zawierający w sobie metodę `Main`.

Uwaga! Korzystanie z nowych wersji plików opisanych powyżej jest obligatoryjne do wykonania zadania. Korzystanie ze starych plików źródłowych będzie traktowane jako plagiat bez możliwości poprawy zadania.

2 Implementacja

Należy utworzyć dwie klasy potomne `Connect4State.cs` i `Connect4Search.cs` dziedziczące odpowiednio po klasach bazowych `State.cs` i `AlphaBetaSearch.cs`. Dobra praktyka programowania mówi, że pojedynczy plik powinien zawierać w sobie implementację pojedynczej klasy. Najpierw należy wczytać plik rozwiązania `Laboratory2.sln` w Visual Studio. Aby dodać klasę do projektu należy kliknąć PPM na Projekcie *Laboratory2* w *Solution Explorer* w *Visual Studio*. Projekt będzie pogrubiony i będzie znajdował się poniżej rozwiązania *Solution 'Laboratory2'*. Następnie wybrać Add → New Item. W otworzonym oknie dialogowym zaznaczyć *Class*, następnie wpisać odpowiednią nazwę w polu *Name:* i klikną *Add*.

2.1 Connect4State.cs

Nowo utworzoną pustą klasę należy zdefiniować jako publiczną i dziedziczącą po klasie bazowej.

```
1  public class Connect4State : State
```

Następnie należy kliknąć PPM na nazwie klasy bazowej (`State`) i wybrać opcję *Implement Abstract Class*. W przypadku nowszej wersji Visual Studio należy skorzystać z opcji Refactor, w której znajduje się równoważne polecenie. W wyniku tego działania zostały utworzone dwa szablony wymagające samodzielnego wypełnienia. Jest to metoda abstrakcyjna i właściwość abstrakcyjna, które muszą zostać zaimplementowana w klasie potomnej. Oczywiście metody, które chcemy nadpisać można dodać ręcznie. Szablony zawierają automatycznie dodane wyjątki `throw new NotImplementedException()`, które należy usunąć w procesie implementacji.

```
1  public override string ID {
2          get { throw new NotImplementedException(); }
3  }
4
5  public override double ComputeHeuristicGrade() {
6          throw new NotImplementedException();
7  }
```

Właściwość ID Właściwość ma na celu zwrócenie string'a jednoznacznie identyfikującego konkretny stan planszy identycznie jak w przypadku zadania poprzedniego. Nie powinno dochodzić do konfliktów czyli dwa odmienne stany powinny posiadać dwie różne wartości **ID**. Natomiast dwa stany reprezentujące ten sam układ na planszy ale będące dwoma różnymi instancjami klasy powinny zwracać tą samą wartość **ID**. Proponuje się zaimplementowanie właściwości w następujący sposób.

```
1  private string id;
2
3  public override string ID {
4          get { return this.id; }
5  }
```

Gdzie `private string id` jest polem klasy inicjalizowanym w konstruktorze.

Konstruktory W klasie nie powinno zabraknąć konstruktorów. Do poprawnej implementacji potrzebne będą dwa konstruktory. Pierwszy tworzący pustą planszę Connect4. Drugi konstruktor jest odpowiedzialny za utworzenie potomka stanu podanego w parametrze o wartościach podanych w parametrze. Poniżej przedstawiono fragmenty kodu wymagane w drugim konstruktorze, które są niezbędne do poprawnego działania programu.

```
1  public Connect4State(Connect4State parent, ... /*pozostale niezbedne
       parametry*/) : base(parent) {
2          //reszta implementacji
3
4          //ustawienie stringa identyfikujacego stan.
5          this.id = ...
6          //ustawienie na ktorym poziomie w drzwie znajduje sie stan.
7          this.depth = parent.depth + 0.5;
8
9          //Bardzo wazne nie ustawiany na czubek drzewa z ktorego budujemy
               stany. Tylko na pierwsze pokolenie stanow potomnych
10         if (parent.rootMove == null) {
11                 this.rootMove = this.id;
12         }
13         else {
14                 this.rootMove = parent.rootMove;
15         }
16         //Ustawienie stanu jako potomka rodzica
17         parent.Children.Add(this);
18 }
```

ComputeHeuristicGrade i przykładowa heurystyka Metoda ma na celu zwrócenie wartości heurystyki dla konkretnego stanu planszy Connect4. Przykładowa heurystyka może mieć następującą postać. Pojedynczy stan jest punktowany za 1 punkt dwa stany pod rząd (w pionie, w poziomie i po skosie) jako 4 punkty, 3 stany pod rząd jako 16 punktów.

liczba stanów pod rząd	gracz maksymalizujący	gracz minimalizujący
1 stan	1	-1
2 stany	4	-4
3 stany	16	-16
4 stany	∞	$-\infty$

Rozważając przykładową planszę connect4

		o	x				
		x	o				
	x	x	o				

zakładając że 'x' jest graczem maksymalizującym a 'o' minimalizującym planszę możemy ocenić w następujący sposób: gracz max zbierze 24 punkty (2 dwójki i 1 trójka) a gracz min zbierze -8 punktów (2 dwójki). Dlatego ocena heurystyczna tego konkretnego stanu planszy wynosi 24 - 8 = 16. Czyli przewagę posiada gracz maksymalizujący. **Jest to jedynie propozycja heurystyki**, student może zaproponować własną. W finalnej heurystyce można uwzględniać następujące rzeczy: preferowanie centrum niż boków lub odwrotnie, bliskość do sufitu, itd. **Uwaga!** w pierwszej kolejności upewnić się, że metoda zwraca odpowiednie 'nieskończoności' (`double.PositiveInfinity`, `double.NegativeInfinity`) dla stanów zwycięzkich.

2.2 Connect4Search.cs

Nowo utworzoną pustą klasę należy zdefiniować jako publiczną i dziedziczącą po klasie bazowej. Oraz zaimplementować metody abstrakcyjne.

```
1 public class SudokuSearch : AlphaBetaSearch
```

Pola klasy Klasa nie będzie wymagała żadnych dodatkowych pól.

Konstruktor W klasie wystarczy zdefiniować pusty konstruktor

```
1 public Connect4Search(IState startState, bool isMaximizingPlayerFirst, int
    maximumDepth) : base(startState, isMaximizingPlayerFirst, maximumDepth)
    { }
```

Parametry są następujące:

- **startState** Wybrany stan planszy connect4, dla której chcemy wykonać algorytm alfa-beta w celu znalezienia kolejnego ruchu.
- **isMaximizingPlayerFirst** Definiuje który z graczy zaczyna rozgrywkę.
- **maximumDepth** Definiuje głębokość przeszukiwania w drzewie.

builChildren Wymagane jest zaimplementowanie metody abstrakcyjnej `buildChildren`. Metoda ma za zadanie zbudowanie potomków wybranego stanu. Rozważając następujący stan:

		o	x		
		x	o		
	x	x	o		

i zakładając, że kolejny jest ruch gracza 'o' stany potomne będą miały postać:

		o	x		
		x	o		
o	x	x	o		

		o	x		
	o	x	o		
	x	x	o		

...

		o	x		
		x	o		
	x	x	o		o

MovesMiniMaxes Jest to właściwość, która na koniec każdego przeszukiwania zawiera w sobie stany zwrócone przez algorytm wraz z wartością przypisanej im heurystyki. Ze zbioru należy wybrać stan o największej wartości heurystyki w przypadku gracza Max lub najmniejszej wartości heurystyki w przypadku gracza Min. W celu przeszukania kolekcji można posłużyć się pętlę `foreach`.

```
1 foreach (KeyValuePair<string, double> kvp in this.MovesMiniMaxes) {
2         //...
3 }
```

2.3 Program.cs

Klasa `Program.cs` zawiera metodę `Main`. W metodzie należy zaimplementować rozgrywkę pomiędzy człowiekiem a sztuczną inteligencją. Rozgrywka powinna odbywać się według następującego schematu. Po wykonaniu ruchu przez człowieka, stan powinien zostać podany w konstruktorze klasy `Connect4Search`. W wyniku działania algorytmu **alfa-beta** (metoda `DoSearch`) klasa zwraca stan wybrany przez sztuczną inteligencję (właściwość `MovesMiniMaxes`). Taki stan powinien zostać wyświetlony użytkownikowi. Następnie użytkownik wykonuje swój ruch, itd. Oczywiście po każdym ruchu należy sprawdzać czy dany gracz nie wygrał.

W przypadku wykorzystania stanu w kolejnym ruchu jako stanu początkowego należy pamiętać o tym aby wyczyścić następujące właściwości `this.parent`, `this.rootMove` oraz `this.depth`.

3 Cel zadania

Napisać program pozwalającą na rozgrywkę w konsoli pomiędzy człowiekiem a sztuczną inteligencją. Zapewnić przełącznik pozwalający na rozpoczęcie dowolnemu graczowi, oraz możliwość ustawienia głębokości przeszukiwania drzewa przed rozpoczęciem rozgrywki. W kodzie powinna istnieć łatwa możliwość zmiany rozmiaru planszy. W czasie gry wyświetlać na ekran heurystyczne oceny ruchów. Interakcja gracza z programem może polegać na wyborze cyfr 1-9 identyfikujących numer kolumny, w której gracz będzie wstawiał swojego pionka.

Możliwe jest dodanie funkcjonalności pozwalającej na rozgrywkę na różnych poziomach. Zadanie to można zrealizować na dwa sposoby: zmieniając głębokość przeszukiwania drzewa oraz poprzez wybieranie stanów, które **nie są najlepsze** z właściwości `MoveMiniMaxes`.