在这一页

Minmax 策略师

许多早在电子游戏时代就很受欢迎的游戏——特别是棋盘游戏，如双陆棋、国际象棋、三字棋和围棋——本质上都是逻辑游戏。玩家通过利用一切机会计划最佳动作，同时考虑到对方玩家（或多名玩家）的可能动作，从而成功玩此类游戏。不管是对方是另一个人还是电脑玩家，这些类型的游戏都可以玩到很刺激。

在GameplayKit中，*策略师*是一种简单的人工智能形式，您可以使用它为这些类型的游戏创建计算机对手。`GKMinmaxStrategist`类和支持的API实现*minmax*策略，用于玩具有以下许多（如果不是全部的话）特征的游戏：

- **顺序。**每个玩家都必须按顺序行事——也就是说，玩家轮流行动。
- **对抗。**一个玩家要想赢，另一个玩家（或多个玩家）必须输。游戏设计假设在每个回合中，玩家都会做出一个动作，使该玩家更接近胜利，或者其他玩家更接近失败。
- **确定性。**玩家可以自信地计划他们的动作，假设动作会按计划进行——也就是说，玩家的行动不受机会的影响。
- **完美的信息。**所有对游戏结果至关重要的信息在任何时候都对所有玩家可见。例如，一盘棋是根据棋子的位置来赢或输的，双方玩家都可以随时看到所有棋子的位置。如果存在一种国际象棋变体，即每个玩家秘密握着一张牌，每张牌对应一个可能的动作，那么该变体将不是一个完美的信息游戏，因为一个玩家的可能动作不会让另一个玩家知道。

您可以使用minmax策略师处理不符合上述所有特性的游戏（或更复杂游戏中的元素），前提是您可以将游戏设计转换为与GameplayKit API兼容的一致描述。例如，在单人游戏中（缺乏对手），您可以使用策略师来建议最佳动作。或者，在一些元素在通过游戏性揭示之前的游戏中，策略师可以在可用信息的约束下计划行动。然而，一般来说，如果你的游戏模式更适合这些特征，策略师会产生更优化的游戏玩法（即更接近赢得或捆绑每场比赛）。

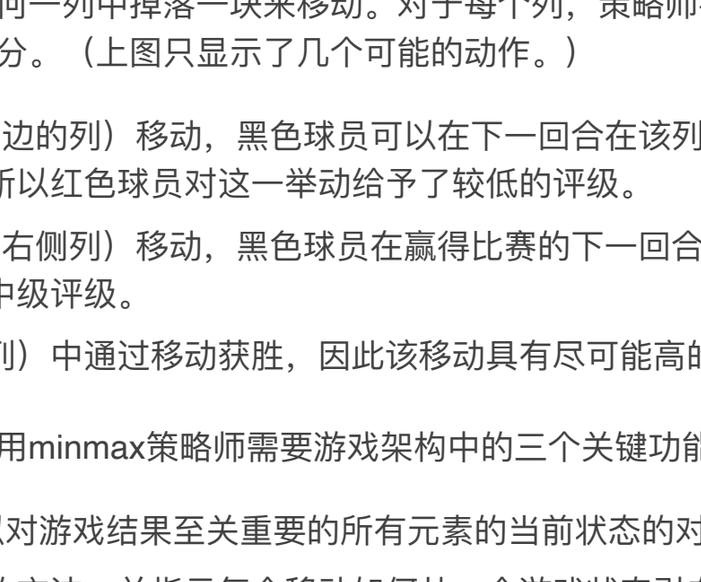
为Minmax策略师设计你的游戏

Minmax策略师通过预测和评级决策树中可能的未来游戏状态来工作，然后通过找到评分最高的树路径来选择下一步。对于自己的每回合，如果可能移动导致胜利，策略师会高评价这些动作；对于策略师对手的每个回合，策略师都会对可能导致对手获胜的可能动作给予低评级。（“minmax”这个名字来自最大化自己动作评分和最小化对手动作评分的策略。）

考虑一个游戏，每个玩家（红色或黑色）通过将彩色芯片放入网格的一列来移动，玩家通过水平、垂直或对角线连续放置四个芯片即可获胜。图5-1显示了此类游戏的部分决策树。FourInARow示例代码项目实现了这款游戏。

> 注
>
> 本章讨论示例代码项目*FourInARow*的功能*：将GameplayKit Minmax策略师用于对手人工智能*。下载它，在Xcode中跟随。

图5-1 可能在四合一的比赛中转弯



红色玩家可以在七根柱子中的任何一列中掉落一块来移动。对于每个列，策略师在该列中移动后考虑董事会的状态，并根据结果对移动进行评分。（上图只显示了几个可能的动作。）

- 如果红色球员在1列（最左边的列）移动，黑色球员可以在下一回合在该列移动并赢得比赛。这对红色球员来说不是一个好结果，所以红色球员对这一举动给予了较低的评级。
- 如果红色球员在第7列（最右侧列）移动，黑色球员在赢得比赛的下一回合没有可能的移动，因此红色球员为这一移动分配了一个中级评级。
- 红色球员在第四列（中间列）中通过移动获胜，因此该移动具有尽可能高的评级。

从这个例子中，应该很清楚，使用minmax策略师需要游戏架构中的三个关键功能：

- 游戏“棋盘”的表示；即模拟对游戏结果至关重要的所有元素的当前状态的对象。
- 一种为玩家指定可能移动的方法，并指示每个移动如何从一个游戏状态引向另一个游戏状态。
- 一种对游戏模型的每个可能状态进行评分或评级的方法，以便将更有利的移动所产生的状态与其他状态区分开来。

一个架构良好的游戏往往已经具备了这些功能。例如，FourInARow示例代码项目使用Model-View-Controller架构将其游戏模型与驱动用户交互和显示的代码分开。清单5-1显示了实现游戏模型的`AAPLBoard`类的主要功能：

四合一游戏模型的5-1 界面清单

```
@interface AAPLBoard : NSObject
{
    AAPLChip _cells[AAPLBoardWidth * AAPLBoardHeight];
}
@property AAPLPlayer *currentPlayer;
- (AAPLChip)chipInColumn:(NSInteger)column row:(NSInteger)row;
- (BOOL)canMoveInColumn:(NSInteger)column;
- (void)addChip:(AAPLChip)chip inColumn:(NSInteger)column;
- (BOOL)isFull;
- (NSArray<NSNumber *> *)runCountsForPlayer:(AAPLPlayer *)player;
@end
```

这些属性和方法暴露了策略师玩游戏所需的所有信息：

- AAPLBoard对象本身表示游戏状态（或游戏的潜在未来状态）。它包含一个矩形的`AAPLChip`值数组（一个整数枚举，带有红色播放器、黑色播放器和空格的标记大小写）。它还引用了`AAPLPlayer`对象，它们表示当前轮到其使用的播放器。
- `canMoveInColumn`和`addChip`方法提供了一种方法来指定任何给定时间的可能移动并模拟这些移动的结果。为了推测游戏未来可能的状态，策略师复制了代表当前状态的“官方”`AAPLBoard`对象，然后在副本上调用`addChip`方法。
- `isFull`方法确定游戏是否因平局而结束，`runCountsForPlayer`方法提供信息，帮助确定任何一名玩家离赢得比赛有多近。如果您添加生成其他`AAPLBoard`对象以猜测游戏未来状态的能力，这些方法可以作为对这些未来状态进行评分或评级的基础。

在游戏中使用Minmax策略师

您使用`GKMinmaxStrategist`类的实例来实现游戏中的minmax策略师。本课程使用您定义的游戏模型来推理游戏状态，为玩家随时采取的“最佳”下一步行动提供建议。然而，在`GKMinmaxStrategist`类能够推理您的游戏模型之前，您必须以GameplayKit能够理解的标准化术语描述该模型。您通过采用`GKGameModel`、`GKGameModelUpdate`和`GKGameModelPlayer`协议来做到这一点。

> 注
>
> 本章讨论示例代码项目*FourInARow*的功能*：将GameplayKit Minmax策略师用于对手人工智能*。下载它，在Xcode中跟随。

在游戏模型中采用GameplayKit协议

三个关键协议定义了GameplayKit用来推理游戏状态的界面。

- `GKGameModel`协议适用于表示游戏模型独特状态的对象。在FourInARow示例中，`AAPLBoard`类是采用此协议的良好候选者。
- `GKGameModelPlayer`协议是游戏中代表玩家的对象的抽象。在本例中，`AAPLPlayer`类适用于此情况。
- `GKGameModelUpdate`协议适用于描述两个游戏状态之间过渡的对象。在示例中，没有这样的对象——`addChip`方法*执行*转换。在这种情况下，采用`GKGameModelUpdate`协议需要创建一个新类。

下面的代码列表展示了如何扩展FourInARow示例中的模型类以采用GameplayKit协议。首先，清单5-2增加了一个采用`GKGameModelUpdate`协议的`AAPLMove`类。

清单5-2 移动类采用GKGameModelUpdate协议

```
@interface AAPLMove : NSObject <GKGameModelUpdate>
@property (nonatomic) NSInteger value;
@property (nonatomic) NSInteger column;
+ (AAPLMove *)moveInColumn:(NSInteger)column;
@end
```

The `GKGameModelUpdate` protocol requires one property: `value`. The strategist assigns a score to this property when weighing the desirability of each possible move. For the rest of your implementation of this protocol, create properties or methods that describe a move in terms particular to your game. In the FourInARow game, the only information needed to specify a move is which column to drop a chip into.

接下来，清单5-3增加了`GKGameModelPlayer`对`AAPLPlayer`类的一致性。

清单5-3 玩家类采用GKGameModelPlayer协议

```
@interface AAPLPlayer (AAPLMinmaxStrategy) <GKGameModelPlayer>
@end

@implementation AAPLPlayer (AAPLMinmaxStrategy)
- (NSInteger)playerId { return self.chip; }
@end
```

`AAPLPlayer`类简单地将其`AAPLChip`枚举的整数值公开为其`playerId`属性。

为`AAPLBoard`类添加`GKGameModel`兼容性需要三个功能领域：跟踪玩家、处理和评分动作以及复制状态信息。清单5-4显示了其中的第一个区域。

列举5-4 GKGameModel实现：管理玩家

```
- (NSArray<AAPLPlayer *> *)players { return [AAPLPlayer allPlayers]; }
- (AAPLPlayer *)activePlayer { return self.currentPlayer; }
```

The `players` getter always returns the two players in the game. For the FourInARow game, the number and identity of players is constant, so this array is stored in the `AAPLPlayer` class. The `activePlayer` getter simply maps to the `currentPlayer` property already implemented in the `AAPLBoard` class.

`GKGameModel`实现的一个关键部分是模拟每个玩家的预期动作。清单5-5显示了`AAPLBoard`类如何处理这些任务。

清单5-5 GKGameModel实现：查找和处理移动

```
- (NSArray<AAPLMove *> *)gameModelUpdatesForPlayer:(AAPLPlayer *)player {
    NSMutableArray<AAPLMove *> *moves = [NSMutableArray arrayWithCapacity:AAPLBoardWidth];
    for (NSInteger column = 0; column < AAPLBoardWidth; column++) {
        if ([self canMoveInColumn:column]) {
            [moves addObject:[AAPLMove moveInColumn:column]];
        }
    }
    return moves; // will be empty if isFull
}

- (void)applyGameModelUpdate:(AAPLMove *)gameModelUpdate {
    [self addChip:self.currentPlayer.chip inColumn:gameModelUpdate.column];
    self.currentPlayer = self.currentPlayer.opponent;
}
```

The `gameModelUpdatesForPlayer:` method returns the possible moves for the specified player. Each possible move is an instance of your game's implementation of the `GKGameModelUpdate` protocol—in this example, the `Move` class. The FourInARow game implements this method by returning an array of `AAPLMove` objects corresponding to each non-full column.

> 注
>
> If your game does not implement the `isWinForPlayer:` and `isLossForPlayer:` methods (see Listing 5-6 and discussion below), your `gameModelUpdatesForPlayer:` method should make sure to return `nil` if the game model represents the end state of a game.

`applyGameModelUpdate:`方法接受一个`GKGameModelUpdate`对象，并执行该对象指定的任何游戏操作。在本示例中，游戏模型更新是一个`AAPLMove`对象，因此该板通过调用其`addChip:`方法来在`AAPLMove`对象指定的列中删除当前玩家颜色的芯片来应用一个动作。这种方法还负责让GameplayKit知道游戏中的转弯顺序，这样当策略师提前计划几个转弯时，它就知道哪个玩家在哪个转弯。此示例是双人游戏，因此`applyGameModelUpdate:`方法在每次移动时都会将`activePlayer`属性切换到当前玩家的对手。

为了让minmax策略师确定哪种可能的动作是最好的选择，您的游戏模型必须指示它何时或多或少代表游戏的理想状态，如清单5-6所示。

上市5-6 GKGameModel实现：评估董事会状态

```
- (BOOL)isWinForPlayer:(AAPLPlayer *)player {
    NSArray<NSNumber *> *runCounts = [self runCountsForPlayer:player];
    // The player wins if there are any runs of 4 (or more, but that shouldn't happen in a regular game).
    NSNumber *longestRun = [runCounts valueForKeyPath:@"@max.self"];
    return longestRun.integerValue >= AAPLCountToWin;
}

- (BOOL)isLossForPlayer:(AAPLPlayer *)player {
    return [self isWinForPlayer:player.opponent];
}

- (NSInteger)scoreForPlayer:(AAPLPlayer *)player {
    NSArray<NSNumber *> *playerRunCounts = [self runCountsForPlayer:player];
    NSNumber *playerTotal = [playerRunCounts valueForKeyPath:@"@sum.self"];
    NSArray<NSNumber *> *opponentRunCounts = [self runCountsForPlayer:player.opponent];
    NSNumber *opponentTotal = [opponentRunCounts valueForKeyPath:@"@sum.self"];
    // Return the sum of player runs minus the sum of opponent runs.
    return playerTotal.integerValue - opponentTotal.integerValue;
}
```

The first step in enabling a strategist to select optimal moves is identifying when a game model represents a win or a loss. You provide this information in the `isWinForPlayer:` and `isLossForPlayer:` methods. In the FourInARow example, the `runCountsForPlayer` method provides a convenient way to test for wins—as the name of the game suggests, the player wins with a run of four chips in a row.

Because this game is for two players only, a loss for the player is by definition a win for the player's opponent, and vice versa, so you can implement the `isLossForPlayer:` method in terms of the `isWinForPlayer:` method. (In a game with more than two players, this assumption might not be true. For example, a winner might be determined only after all other players have lost.)

Armed with only the knowledge of winning and losing game states, the minmax strategist can play the game. The `GKMinmaxStrategist` class automatically ignores possible moves after a win or loss, and it weighs its decisions based on how many moves in the future an expected win or loss is. However, at this point the strategist does not play the game very well, because there are a number of possible states of the game with equal numbers of possible future wins or losses.

To improve the strategist's gameplay, use the `scoreForPlayer:` method to implement a *heuristic*—an estimate of how desirable the current state of the board is to the player. Alternately, a score represents an estimate of how likely the player is to win, or how soon, the player can win. The FourInARow example uses a heuristic based on its `runCountsForPlayer` method, assuming that, for example, a player with two runs of two chips each is more likely to win soon than a player with no runs. Because the player's success or failure is related to that of the opponent, the `scoreForPlayer:` method totals the player's number and length of runs and subtracts from it the opponent's number and length of runs.

> 实验
>
> 这个例子的启发式足以产生适度强大的游戏性，但不是这款游戏的最佳启发式。对于一个更强大的计算机控制对手，请尝试改进启发式：
>
> - 体重更长更强劲。比如三连三连三连三连三应该比三连三要好。
> - 描述球员可以通过填补空白赢得胜利的安排。例如，如果玩家连续有两个芯片，然后是一个开放空间，然后是另一个芯片，则该安排应该与连续三个芯片的一块芯片得分等价，然后是缺口。
> - 已阻止跑步的折扣安排。例如，如果玩家连续有三个筹码，但对手在该行的两侧有筹码，则该跑动不能导致胜利。

最后，游戏模型类必须支持复制自身及其内部状态，以便策略师可以复制多个板块，以检查未来可能移动的结果。清单5-7显示了`AAPLBoard`类如何复制自己。

清单5-7 GKGameModel实现：复制状态信息

```
- (void)setGameModel:(AAPLBoard *)gameModel {
    memcpy(_cells, gameModel->_cells, sizeof(_cells));
    self.currentPlayer = gameModel.currentPlayer;
}

- (id)copyWithZone:(NSZone *)zone {
    AAPLBoard *copy = [[[self class] allocWithZone:zone] init];
    [copy setGameModel:self];
    return copy;
}
```

The `setGameModel:` method copies the internal state of one `GKGameModel` object to another. Here, the `AAPLBoard` class simply copies its `_cells` array and `currentPlayer` property. The `GKGameModel` protocol requires conformance to the `NSCopying` protocol, so the `copyWithZone:` method uses the `setGameModel:` method to turn a new `AAPLBoard` instance into a copy.

To simulate possible moves, the strategist uses several copies of your `GKGameModel` class and calls the `setGameModel:` method many times. For example, looking ahead six turns in the FourInARow game (which has up to seven possible moves at any given time) requires examining hundreds of thousands of board states. (Rather than continually creating new instances of your game model class, GameplayKit makes only a few copies, and reuses those instances with the `setGameModel:` method. This optimization improves memory efficiency.)

> 重要
>
> 由于GameplayKit每次计划移动时都可以评估数千个游戏状态，因此其性能受到`GKGameModel`类和`setGameModel:`方法的大小和复杂性的限制。确保您的类仅包含最小描述进行中游戏状态的数据，并且您的`setGameModel:`方法可以快速复制该数据（例如，无需创建新对象或分配内存）。

使用策略对象来计划动作

在游戏模型类中采用`GKGameModel`和相关协议后，使用minmax策略师只需要几个简单步骤。要了解FourInARow游戏中如何实现这些步骤，请下载示例代码项目。

1. 在管理游戏模型的任何控制器代码中保留`GKMinmaxStrategist`实例——例如，视图控制器或SpriteKit场景。
2. 使用`GKMinmaxStrategist`对象的`gameModel`属性指向游戏的当前状态——即实现`GKGameModel`协议的模型对象。

   通常，在初始化新游戏时，您只能这样做一次，因为即使该`GKGameModel`对象的内部状态发生变化，该属性也会引用该对象。但是，如果您开始使用模型类的不同实例（例如，在为新游戏重置板时），请务必再次设置`gameModel`属性。
3. 将`maxLookAheadDepth`属性设置为您要计划的连续未来移动次数。

   一般来说，更深入的前瞻性会导致更强大的计算机控制播放器。然而，更深入的前瞻性也指数级地增加了战略家为规划行动而必须做的工作量。选择一个前瞻性的深度，在不牺牲性能的情况下为您的游戏提供所需的难度水平。

   For example, the FourInARow game requires four chips in a row to win. Therefore, if a computer-controlled player always makes the optimal move, it should theoretically always be able to either win or force a draw when the lookahead depth is seven—four of the strategist's own turns plus the intervening three opponent turns. (Whether that theory holds true depends on the robustness of your `scoreForPlayer:` method.) For this game, a lookahead depth of seven results in a search space of nearly a million possible board states.
4. To plan the computer-controlled player's next move, use the `bestMoveForPlayer:` method. This method returns a `GKGameModelUpdate` object (in the FourInARow example, an `AAPLMove` object) corresponding to the chosen move. To perform the move, use your gameplay model's `applyGameModelUpdate:` method.

   或者，使用*GKMinmaxStrategist类参考*中列出的其他方法之一，该方法为处理多个“最佳”移动或随机移动的案例提供了选项。

> 注
>
> 找到最好的动作可能需要很长时间，特别是在向前看几个弯时。为了保持游戏的用户界面性能流畅，请使用后台队列进行策略工作。具体内容请参见*并发编程指南*。

◀ 状态机　　　　寻路 ▶