6
Board Games AI

In this chapter, you will learn a family of algorithms for developing AI for board games:

- Working with the game-tree class
- Introducing Minimax
- Negamaxing
- AB Negamaxing
- Negascouting
- Implementing a tic-tac-toe rival
- Implementing a checkers rival

Introduction

In this chapter, you will learn about a family of algorithms for developing board game techniques to create artificial intelligence. They are based on the principle of a game tree (graph) that spans as we evaluate a state and decide to visit its neighbors. They also take into account board games for two rivals. But with a little bit of work, some of them can be extended to more players.

Working with the game-tree class

The game state can be represented in a lot of different ways, but you will learn how to create extendible classes in order to use the high-level board AI algorithms for different circumstances.

Getting ready...

It is important to be clear on object-oriented programming, specifically on inheritance and polymorphism. This is because we'll be creating generic functions that can be applied to a number of board game decisions and then writing specific subclasses that inherit and further specify these functions.

How to do it...

We will build two classes in order to represent game-tree with the help of the following steps:

1. Create the abstract class `Move`:

```
using UnityEngine;
using System.Collections;

public abstract class Move
{

}
```

2. Create the pseudo-abstract class `Board`:

```
using UnityEngine;
using System.Collections;

public class Board
{
    protected int player;
    //next steps here
}
```

3. Define the default constructor:

```
public Board()
{
    player = 1;
}
```

4. Implement the virtual function for retrieving the next possible moves:

```
public virtual Move[] GetMoves()
{
    return new Move[0];
}
```

5. Implement the virtual function for playing a move on the board:

```
public virtual Board MakeMove(Move m)
{
    return new Board();
}
```

6. Define the virtual function for testing whether the game is over:

```
public virtual bool IsGameOver()
{
    return true;
}
```

7. Implement the virtual function for retrieving the current player:

```
public virtual int GetCurrentPlayer()
{
    return player;
}
```

8. Implement the virtual function for testing the board's value for a given player:

```
public virtual float Evaluate(int player)
{
    return Mathf.NegativeInfinity;
}
```

9. Also, implement the virtual function for testing the board's value for the current player:

```
public virtual float Evaluate()
{
    return Mathf.NegativeInfinity;
}
```

How it works...

We have created the stepping stones for the next algorithms. The `Board` class works as a node in order to represent the current game state, and the `Move` class represents an edge. When the `GetMoves` function is called, we model the function for getting the edges in order to reach the neighbors of the current game state.

See also

For more theoretical insights about the techniques in this chapter, please refer to Russel and Norvig's *Artificial Intelligence: a Modern Approach* (adversarial search) and Ian Millington's *Artificial Intelligence for Games* (board games).

Introducing Minimax

Minimax is an algorithm based on the decision to minimize the possible loss for the worst case (maximum loss). Besides game development and game theory, Minimax is a decision rule and is also used in statistics, decision theory, and philosophy.

This technique was originally formulated for the two-player zero-sum game theory, meaning that one player's win is the opponent's loss. However, in this case, it is flexible enough to handle more than two players.

Getting ready...

It is important to know the difference between a dynamic member function and a static member function, as well as recursion. A dynamic member function is bound to the instance of the class, while the static member function is bound to the class itself. The static method allows us to call it without instantiating an object. This is great for general-purpose algorithms, such as the one developed in this recipe.

In the case of recursion, it's not always clear that (unlike iteration) this is an iterative process that requires a base case (also called the stop condition) and a recursive case (the one to keep iterating).

How to do it...

We will create the base class for handling all of our main algorithms and implement the `Minimax` function as follows:

1. Create the `BoardAI` class:

```
using UnityEngine;
using System.Collections;

public class BoardAI
{

}
```

2. Declare the `Minimax` function:

```
public static float Minimax(
        Board board,
        int player,
        int maxDepth,
        int currentDepth,
```

```
        ref Move bestMove)
{
    // next steps here
}
```

3. Consider the base case:

```
if (board.IsGameOver() || currentDepth == maxDepth)
    return board.Evaluate(player);
```

4. Set the initial values depending on the player:

```
bestMove = null;
float bestScore = Mathf.Infinity;
if (board.GetCurrentPlayer() == player)
    bestScore = Mathf.NegativeInfinity;
```

5. Loop through all the possible moves and return the best score:

```
foreach (Move m in board.GetMoves())
{
    // next steps here
}
return bestScore;
```

6. Create a new game state from the current move:

```
Board b = board.MakeMove(m);
float currentScore;
Move currentMove = null;
```

7. Start the recursion:

```
currentScore = Minimax(b, player, maxDepth, currentDepth + 1, ref
currentMove);
```

8. Validate the score for the current player:

```
if (board.GetCurrentPlayer() == player)
{
    if (currentScore > bestScore)
    {
        bestScore = currentScore;
        bestMove = currentMove;
    }
}
```

9. Validate the score for the adversary:

```
else
{
    if (currentScore < bestScore)
    {
        bestScore = currentScore;
        bestMove = currentMove;
    }
}
```

How it works...

The algorithm works as a bounded depth-first search. In each step, the move is chosen by selecting the option that maximizes the player's score and assuming the opponent will take the option for minimizing it, until a terminal (leaf) node is reached.

The move tracking is done using recursion, and the heuristic for selecting or assuming an option depends on the `Evaluate` function.

See also

- The *Working with the game-tree class* recipe in this chapter

Negamaxing

When we have a zero-sum game with only two players involved, we are able to improve Minimax, taking advantage of the principle that one player's loss is the other's gain. In this way, it is able to provide the same results as the Minimax algorithm. However, it does not track whose move it is.

Getting ready...

It is important to know the difference between a dynamic member function and a static member function, as well as recursion. A dynamic member function is bound to the instance of the class, while a static member function is bound to the class itself. The static method allows us to call it without instantiating an object. This is great for general-purpose algorithms, such as the one we are developing in this recipe.

In the case of recursion, it's not always clear that (unlike iteration) this is an iterative process that requires a base case (also called the stop condition) and a recursive case (the one to keep iterating).

How to do it...

We will add a new function to the `BoardAI` class as follows:

1. Create the `Negamax` function:

```
public static float Negamax(
        Board board,
        int maxDepth,
        int currentDepth,
        ref Move bestMove)
{
    // next steps here
}
```

2. Validate the base case:

```
if (board.IsGameOver() || currentDepth == maxDepth)
    return board.Evaluate();
```

3. Set the initial values:

```
bestMove = null;
float bestScore = Mathf.NegativeInfinity;
```

4. Loop through all the available moves and return the best score:

```
foreach (Move m in board.GetMoves())
{
    // next steps here
}
return bestScore;
```

5. Create a new game state from the current move:

```
Board b = board.MakeMove(m);
float recursedScore;
Move currentMove = null;
```

6. Start the recursion:

```
recursedScore = Negamax(b, maxDepth, currentDepth + 1, ref
currentMove);
```

7. Set the current score and update the best score and move, if necessary:

```
float currentScore = -recursedScore;
if (currentScore > bestScore)
{
    bestScore = currentScore;
    bestMove = m;
}
```

How it works...

The base algorithm works the same but, as we did before, there are some advantages. At each step in the recursion, the scores from the previous steps have their sign inverted. Instead of choosing the best option, the algorithm changes the sign of the score, eliminating the need to track whose move it is.

There's more...

As Negamax alternates the viewpoints between players at each step, the evaluate function used is the one with no parameters.

See also

- The *Working with the game-tree class* recipe
- The *Minimax* recipe

AB Negamaxing

There is still room for improving the Negamax algorithm. Despite its efficiency, the downside of the Negamax algorithm is that it examines more nodes than necessary (for example, board positions). To overcome this problem, we use Negamax with a search strategy called alpha-beta pruning.

Getting ready...

It is important to know the difference between a dynamic member function and a static member function, as well as recursion. A dynamic member function is bound to the instance of the class, while the static member function is bound to the class itself. The static method allows us to call it without instantiating an object. This is great for general-purpose algorithms, such as the one we are developing in this recipe.

In the case of recursion, it's not always clear that (unlike iteration) this is an iterative process that requires a base case (also called the stop condition) and a recursive case (the one to keep iterating).

How to do it...

We will add a new function to the `BoardAI` class as follows:

1. Create the `ABNegamax` function:

```
public static float ABNegamax(
        Board board,
        int player,
        int maxDepth,
        int currentDepth,
        ref Move bestMove,
        float alpha,
        float beta)
{
    // next steps here
}
```

2. Validate the base case:

```
if (board.IsGameOver() || currentDepth == maxDepth)
    return board.Evaluate(player);
```

3. Set the initial values:

```
bestMove = null;
float bestScore = Mathf.NegativeInfinity;
```

4. Loop through every available move and return the best score:

```
foreach (Move m in board.GetMoves())
{
    // next steps here
}
return bestScore;
```

5. Create a new game state from the current move:

```
Board b = board.MakeMove(m);
```

6. Set the values for calling the recursion:

```
float recursedScore;
Move currentMove = null;
int cd = currentDepth + 1;
float max = Mathf.Max(alpha, bestScore);
```

7. Start the recursion:

```
recursedScore = ABNegamax(b, player, maxDepth, cd, ref
currentMove, -beta, max);
```

8. Set the current score and update the best score and move if necessary. Also, stop the iteration if necessary:

```
float currentScore = -recursedScore;
if (currentScore > bestScore)
{
    bestScore = currentScore;
    bestMove = m;

    if (bestScore >= beta)
        return bestScore;
}
```

How it works...

Since we know the basic principle of the algorithm, let's concentrate on the search strategy. There are two values: alpha and beta. The alpha value is the lower score a player can achieve, thus avoiding considering any move where the opponent has the opportunity to lessen it. Similarly, the beta value is the upper limit; no matter how tempting the new option is, the algorithm assumes that the opponent won't give the opportunity to take it.

Given the alternation between each player (minimizing and maximizing), only one value needs to be checked at each step.

See also

- The *Working with the game-tree class* recipe
- The *Minimax* recipe
- The *Negamaxing* recipe

Negascouting

Including a search strategy also makes room for new challenges. Negascouting is the result of narrowing the search by improving the pruning heuristic. It is based on a concept called a **search window**, which is the interval between the alpha and beta values. So, reducing the search window increases the chance of a branch being pruned.

Getting ready...

It is important to know the difference between a dynamic member function and a static member function, as well as recursion. A dynamic member function is bound to the instance of the class, while the static member function is bound to the class itself. The static method allows us to call it without instantiating an object. This is great for general-purpose algorithms, such as the one we are developing in this recipe.

In the case of recursion, it's not always clear that (unlike iteration) this is an iterative process that requires a base case (also called the stop condition) and a recursive case (the one to keep iterating).

How to do it...

We will add a new function to the `BoardAI` class as follows:

1. Create the `ABNegascout` function:

```
public static float ABNegascout (
        Board board,
        int player,
        int maxDepth,
        int currentDepth,
        ref Move bestMove,
        float alpha,
        float beta)
{
    // next steps here
}
```

2. Validate the base case:

```
if (board.IsGameOver() || currentDepth == maxDepth)
    return board.Evaluate(player);
```

3. Set the initial values:

```
bestMove = null;
float bestScore = Mathf.NegativeInfinity;
float adaptiveBeta = beta;
```

4. Loop through every available move and return the best score:

```
foreach (Move m in board.GetMoves())
{
    // next steps here
}
return bestScore;
```

5. Create a new game state from the current move:

```
Board b = board.MakeMove(m);
```

6. Set the values for the recursion:

```
Move currentMove = null;
float recursedScore;
int depth = currentDepth + 1;
float max = Mathf.Max(alpha, bestScore);
```

7. Call the recursion:

```
recursedScore = ABNegamax(b, player, maxDepth, depth, ref
currentMove, -adaptiveBeta, max);
```

8. Set the current score and validate it:

```
float currentScore = -recursedScore;
if (currentScore > bestScore)
{
    // next steps here
}
```

9. Validate for pruning:

```
if (adaptiveBeta == beta || currentDepth >= maxDepth - 2)
{
    bestScore = currentScore;
    bestMove = currentMove;
}
```

10. Otherwise, take a look around:

```
else
{
    float negativeBest;
    negativeBest = ABNegascout(b, player, maxDepth, depth, ref
bestMove, -beta, -currentScore);
    bestScore = -negativeBest;
}
```

11. Stop the loop if necessary. Otherwise, update the adaptive value:

```
if (bestScore >= beta)
    return bestScore;

adaptiveBeta = Mathf.Max(alpha, bestScore) + 1f;
```

How it works...

This algorithm works by examining the first move of each node. The following moves are examined using a scout pass with a narrower window based on the first move. If the pass fails, it is repeated using a full-width window. As a result, a large number of branches are pruned and failures are avoided.

See also

- The *AB Negamaxing* recipe

Implementing a tic-tac-toe rival

In order to make use of the previous recipes, we will devise a way to implement a rival for a popular game: tic-tac-toe. Not only does it help us extend the base classes, but it also gives us a way to create rivals for our own board games.

Getting ready...

We will need to create a specific move class for the tic-tac-toe board derived from the parent class we created at the beginning of the chapter:

```
using UnityEngine;
using System.Collections;

public class MoveTicTac : Move
{
    public int x;
    public int y;
    public int player;

    public MoveTicTac(int x, int y, int player)
    {
        this.x = x;
        this.y = y;
        this.player = player;
    }
}
```

How to do it...

We will create a new class, deriving it from `Board`, override its parent's methods, and create new ones.

1. Create the `BoardTicTac` class, deriving it from `Board`, and add the corresponding member variables for storing the board's values:

```
using UnityEngine;
using System;
using System.Collections;
using System.Collections.Generic;

public class BoardTicTac : Board
{
    protected int[,] board;
    protected const int ROWS = 3;
    protected const int COLS = 3;
}
```

2. Implement the default constructor:

```
public BoardTicTac(int player = 1)
{
    this.player = player;
    board = new int[ROWS, COLS];
    board[1,1] = 1;
}
```

3. Define the function for retrieving the next player in turn:

```
private int GetNextPlayer(int p)
{
    if (p == 1)
        return 2;
    return 1;
}
```

4. Create a function for evaluating a given position regarding a given player:

```
private float EvaluatePosition(int x, int y, int p)
{
    if (board[y, x] == 0)
        return 1f;
    else if (board[y, x] == p)
        return 2f;
    return -1f;
}
```

5. Define a function for evaluating the neighbors of a given position regarding a given player:

```
private float EvaluateNeighbours(int x, int y, int p)
{
    float eval = 0f;
    int i, j;
    for (i = y - 1; i < y + 2; y++)
    {
        if (i < 0 || i >= ROWS)
            continue;
        for (j = x - 1; j < x + 2; j++)
        {
            if (j < 0 || j >= COLS)
                continue;
            if (i == j)
                continue;
            eval += EvaluatePosition(j, i, p);
        }
    }
    return eval;
}
```

6. Implement a constructor for building new states with values:

```
public BoardTicTac(int[,] board, int player)
{
    this.board = board;
    this.player = player;
}
```

7. Override the member function for getting the available moves from the current state:

```
public override Move[] GetMoves()
{
    List<Move> moves = new List<Move>();
    int i;
    int j;
    for (i = 0; i < ROWS; i++)
    {
        for (j = 0; j < COLS; j++)
        {
            if (board[i, j] != 0)
                continue;
            MoveTicTac m = new MoveTicTac(j, i, player);
            moves.Add(m);
        }
```

```
        }
        return moves.ToArray();
    }
```

8. Override the function for retrieving a new state from a given move:

```
    public override Board MakeMove(Move m)
    {
        MoveTicTac move = (MoveTicTac)m;
        int nextPlayer = GetNextPlayer(move.player);
        int[,] copy = new int[ROWS, COLS];
        Array.Copy(board, 0, copy, 0, board.Length);
        copy[move.y, move.x] = move.player;
        BoardTicTac b = new BoardTicTac(copy, nextPlayer);
        return b;
    }
```

9. Define the function for evaluating the current state, given a player:

```
    public override float Evaluate(int player)
    {
        float eval = 0f;
        int i, j;
        for (i = 0; i < ROWS; i++)
        {
            for (j = 0; j < COLS; j++)
            {
                eval += EvaluatePosition(j, i, player);
                eval += EvaluateNeighbours(j, i, player);
            }
        }
        return eval;
    }
```

10. Implement the function for evaluating the current state of the current player:

```
    public override float Evaluate()
    {
        float eval = 0f;
        int i, j;
        for (i = 0; i < ROWS; i++)
        {
            for (j = 0; j < COLS; j++)
            {
                eval += EvaluatePosition(j, i, player);
```

```
                eval += EvaluateNeighbours(j, i, player);
            }
        }
        return eval;
    }
```

How it works...

We define a new type of move for the board that works well with the base algorithms because they make use of it only at a high level as a data structure. The recipe's bread and butter come from overriding the virtual functions from the `Board` class in order to model the problem. We use a two-dimensional integer array for storing the players' moves on the board (0 represents an empty place), and we work out a heuristic for defining the value of a given state regarding its neighbors.

There is more...

The functions for evaluating a board's (state) score have an admissible heuristic, but it's probably not optimal. It is up to us to revisit this problem and refactor the body of the aforementioned functions in order to have a better tuned rival.

See also

- The *Working with the game-tree class* recipe

Implementing a checkers rival

You will learn how to extend the previous recipes with an advanced example. In this case, you will learn how to model a checkers (draughts) board and its pieces in order to comply with the necessary functions to be used with our board-AI framework.

This approach uses a chess board (8 x 8) and its respective number of pieces (12). However, it can be easily parameterized in order to change these values in case we want to have a differently sized board.

Getting ready...

First, we need to create a new type of movement for this particular case called `MoveDraughts`:

```
using UnityEngine;
using System.Collections;

public class MoveDraughts : Move
{
    public PieceDraughts piece;
    public int x;
    public int y;
    public bool success;
    public int removeX;
    public int removeY;
}
```

This data structure stores the piece to be moved, the new `x` and `y` coordinates if the movement is a successful capture, and the position of the piece to be removed.

How to do it...

We will implement two core classes for modeling the pieces and the board, respectively. This is a long process, so read each step carefully:

1. Create a new file called `PieceDraughts.cs` and add the following statements:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
```

2. Add the `PieceColor` data type:

```
public enum PieceColor
{
    WHITE,
    BLACK
};
```

3. Add the `PieceType` data enum:

```
public enum PieceType
{
    MAN,
    KING
};
```

4. Start building the `PieceDraughts` class:

```
public class PieceDraughts : MonoBehaviour
{
    public int x;
    public int y;
    public PieceColor color;
    public PieceType type;
    // next steps here
}
```

5. Define the function for setting up the piece logically:

```
public void Setup(int x, int y,
        PieceColor color,
        PieceType type = PieceType.MAN)
{
    this.x = x;
    this.y = y;
    this.color = color;
    this.type = type;
}
```

6. Define the function for moving the piece on the board:

```
public void Move (MoveDraughts move, ref PieceDraughts [,] board)
{
    board[move.y, move.x] = this;
    board[y, x] = null;
    x = move.x;
    y = move.y;
    // next steps here
}
```

7. If the move is a capture, remove the corresponding piece:

```
if (move.success)
{
    Destroy(board[move.removeY, move.removeX]);
    board[move.removeY, move.removeX] = null;
}
```

8. Stop the process if the piece is `King`:

```
if (type == PieceType.KING)
    return;
```

9. Change the type of piece if it is `Man` and it reaches the opposite border:

```
int rows = board.GetLength(0);
if (color == PieceColor.WHITE && y == rows)
    type = PieceType.KING;
if (color == PieceColor.BLACK && y == 0)
    type = PieceType.KING;
```

10. Define the function for checking if a move is inside the bounds of the board:

```
private bool IsMoveInBounds(int x, int y, ref PieceDraughts[,]
board)
{
    int rows = board.GetLength(0);
    int cols = board.GetLength(1);
    if (x < 0 || x >= cols || y < 0 || y >= rows)
        return false;
    return true;
}
```

11. Define the general function for retrieving the possible moves:

```
public Move[] GetMoves(ref PieceDraughts[,] board)
{
    List<Move> moves = new List<Move>();
    if (type == PieceType.KING)
        moves = GetMovesKing(ref board);
    else
        moves = GetMovesMan(ref board);
    return moves.ToArray();
}
```

12. Start implementing the function for retrieving the moves when the piece's type is `Man`:

```
private List<Move> GetMovesMan(ref PieceDraughts[,] board)
{
    // next steps here
}
```

13. Add the variable for storing the two possible moves:

```
List<Move> moves = new List<Move>(2);
```

14. Define the variable for holding the two possible horizontal options:

```
int[] moveX = new int[] { -1, 1 };
```

15. Define the variable for holding the vertical direction depending on the piece's color:

```
int moveY = 1;
if (color == PieceColor.BLACK)
    moveY = -1;
```

16. Implement the loop for iterating through the two possible options and return the available moves. We will implement the body of the loop in the next step:

```
foreach (int mX in moveX)
{
    // next steps
}
return moves;
```

17. Declare two new variable for computing the next position to be considered:

```
int nextX = x + mX;
int nextY = y + moveY;
```

18. Test the possible option if the move is out of bounds:

```
if (!IsMoveInBounds(nextX, y, ref board))
    continue;
```

19. Continue with the next option if the move is being blocked by a piece of the same color:

```
PieceDraughts p = board[moveY, nextX];
if (p != null && p.color == color)
    continue;
```

20. Create a new move to be added to the list because we're good-to-go:

```
MoveDraughts m = new MoveDraughts();
m.piece = this;
```

21. Create a simple move if the position is available:

```
if (p == null)
{
    m.x = nextX;
    m.y = nextY;
}
```

22. Otherwise, test whether the piece can be captured and modify the move accordingly:

```
else
{
    int hopX = nextX + mX;
    int hopY = nextY + moveY;
    if (!IsMoveInBounds(hopX, hopY, ref board))
```

```
            continue;
        if (board[hopY, hopX] != null)
            continue;
        m.y = hopX;
        m.x = hopY;
        m.success = true;
        m.removeX = nextX;
        m.removeY = nextY;
    }
```

23. Add the move to the list:

```
moves.Add(m);
```

24. Start to implement the function for retrieving the available moves when the piece's type is `King`:

```
private List<Move> GetMovesKing(ref PieceDraughts[,] board)
{
    // next steps here
}
```

25. Declare the variable for holding the possible moves:

```
List<Move> moves = new List<Move>();
```

26. Create the variables for searching in four directions:

```
int[] moveX = new int[] { -1, 1 };
int[] moveY = new int[] { -1, 1 };
```

27. Start implementing the loop for checking all the possible moves, and retrieve those moves. The next step will implement the body of the inner loop:

```
foreach (int mY in moveY)
{
    foreach (int mX in moveX)
    {
        // next steps here
    }
}
return moves;
```

28. Create the variables for testing the moves and advances:

```
int nowX = x + mX;
int nowY = y + mY;
```

29. Create a loop for going in that direction until the board's bounds are reached:

```
while (IsMoveInBounds(nowX, nowY, ref board))
{
    // next steps here
}
```

30. Get the position's piece reference:

```
PieceDraughts p = board[nowY, nowX];
```

31. If it is a piece of the same color, go no further:

```
if (p != null && p.color == color)
    break;
```

32. Define a variable for creating the new available move:

```
MoveDraughts m = new MoveDraughts();
m.piece = this;
```

33. Create a simple move if the position is available:

```
if (p == null)
{
    m.x = nowX;
    m.y = nowY;
}
```

34. Otherwise, test whether the piece can be captured and modify the move accordingly:

```
else
{
    int hopX = nowX + mX;
    int hopY = nowY + mY;
    if (!IsMoveInBounds(hopX, hopY, ref board))
        break;
    m.success = true;
    m.x = hopX;
    m.y = hopY;
    m.removeX = nowX;
    m.removeY = nowY;
}
```

35. Add the move and advance a step towards the current direction:

```
moves.Add(m);
nowX += mX;
nowY += mY;
```

36. Create a new class called `BoardDraughts` in a new file:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class BoardDraughts : Board
{
    public int size = 8;
    public int numPieces = 12;
    public GameObject prefab;
    protected PieceDraughts[,] board;
}
```

37. Implement the `Awake` function:

```
void Awake()
{
    board = new PieceDraughts[size, size];
}
```

38. Start implementing the `Start` function. It is important to note that this may vary depending on your game's spatial representation:

```
void Start()
{
    // TODO
    // initialization and board set up
    // your implementation may vary

    // next steps here
}
```

39. Throw an error message if the template object doesn't have an attached `PieceDraught` script:

```
PieceDraughts pd = prefab.GetComponent<PieceDraughts>();
if (pd == null)
{
    Debug.LogError("No PieceDraught component detected");
    return;
}
```

40. Add iterator variables:

```
int i;
int j;
```

41. Implement the loop for placing the white pieces:

```
int piecesLeft = numPieces;
for (i = 0; i < size; i++)
{
    if (piecesLeft == 0)
        break;
    int init = 0;
    if (i % 2 != 0)
        init = 1;
    for (j = init; j < size; j+=2)
    {
        if (piecesLeft == 0)
            break;
        PlacePiece(j, i);
        piecesLeft--;
    }
}
```

42. Implement the loop for placing the black pieces:

```
piecesLeft = numPieces;
for (i = size - 1; i >= 0; i--)
{
    if (piecesLeft == 0)
        break;
    int init = 0;
    if (i % 2 != 0)
        init = 1;
    for (j = init; j < size; j+=2)
    {
        if (piecesLeft == 0)
            break;
        PlacePiece(j, i);
        piecesLeft--;
    }
}
```

43. Implement the function for placing a specific piece. This could change in your game depending on its visualization:

```
private void PlacePiece(int x, int y)
{
    // TODO
    // your own transformations
    // according to space placements
    Vector3 pos = new Vector3();
```

```
        pos.x = (float)x;
        pos.y = -(float)y;
        GameObject go = GameObject.Instantiate(prefab);
        go.transform.position = pos;
        PieceDraughts p = go.GetComponent<PieceDraughts>();
        p.Setup(x, y, color);
        board[y, x] = p;
    }
```

44. Implement the `Evaluate` function with no parameters:

```
    public override float Evaluate()
    {
        PieceColor color = PieceColor.WHITE;
        if (player == 1)
            color = PieceColor.BLACK;
        return Evaluate(color);
    }
```

45. Implement the `Evaluate` function with a parameter:

```
    public override float Evaluate(int player)
    {
        PieceColor color = PieceColor.WHITE;
        if (player == 1)
            color = PieceColor.BLACK;
        return Evaluate(color);
    }
```

46. Start implementing the general function for evaluation:

```
    private float Evaluate(PieceColor color)
    {
        // next steps here
    }
```

47. Define the variables for holding the evaluation and assigning points:

```
    float eval = 1f;
    float pointSimple = 1f;
    float pointSuccess = 5f;
```

48. Create variables for holding the board's bounds:

```
    int rows = board.GetLength(0);
    int cols = board.GetLength(1);
```

49. Define variables for iteration:

```
    int i;
    int j;
```

50. Iterate throughout the board to look for moves and possible captures:

```
for (i = 0; i < rows; i++)
{
    for (j = 0; j < cols; j++)
    {
        PieceDraughts p = board[i, j];
        if (p == null)
            continue;
        if (p.color != color)
            continue;
        Move[] moves = p.GetMoves(ref board);
        foreach (Move mv in moves)
        {
            MoveDraughts m = (MoveDraughts)mv;
            if (m.success)
                eval += pointSuccess;
            else
                eval += pointSimple;
        }
    }
}
```

51. Retrieve the evaluation value:

```
return eval;
```

52. Start developing the function for retrieving the board's available moves:

```
public override Move[] GetMoves()
{
    // next steps here
}
```

53. Define the variables for holding the moves and the board's boundaries, and handling iteration:

```
List<Move> moves = new List<Move>();
int rows = board.GetLength(0);
int cols = board.GetLength(1);
int i;
int j;
```

54. Get the moves from all the available pieces on the board:

```
for (i = 0; i < rows; i++)
{
    for (j = 0; i < cols; j++)
    {
```

```
                PieceDraughts p = board[i, j];
                if (p == null)
                    continue;
                moves.AddRange(p.GetMoves(ref board));
            }
        }
```

55. Return the moves found:

```
return moves.ToArray();
```

How it works...

The board works in a similar fashion to the previous board, but it has a more complex process due to the rules of the game. The movements are tied to the pieces' moves, thus creating a cascading effect that must be handled carefully. Each piece has two types of movement, depending on its color and type.

As we can see, the high-level rules are the same. It just requires a little bit of patience and thinking in order to develop good evaluation functions and procedures for retrieving the board's available moves.

There is more...

The `Evaluate` function is far from being perfect. We implemented a heuristic based solely on the number of available moves and captured opponent pieces, giving room for improvement in order to avoid movements where a player's piece could be captured in the rival's next move.

Also, we should make our own changes to the `PlacePiece` function in the `BoardDraughts` class. We implemented a direct method that probably doesn't fit your game's spatial setup.