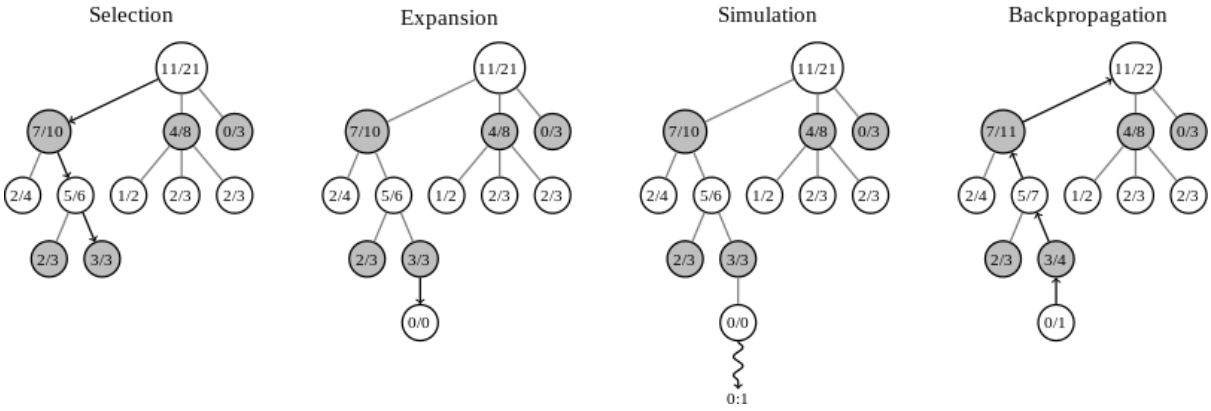
COMPLEX GAME SYSTEMS
CHECKERS AI

By Iain Dowling

MONTE-CARLO TREE SEARCH

The method of decision making that I have to chosen to implement for my Checkers AI is a simplified Monte-Carlo tree search.

The original Monte-Carlo tree search uses 4 steps to choose the best action: Selection, Expansion, Simulation and Backpropagation.



The selection stage randomly selects a node in the current tree and recursively selects random child nodes until it hits a leaf.
The expansion stage looks at that leaf node, checking if it ends that game. If not, it creates a potential child of that leaf.
The simulation stage runs a simulated playout of the game until the end state is reached.
The backpropagation stage updates the current path of the tree with the simulated result.

My method looks at the current state of the game board, looking for all valid moves. For each valid move it finds, it simulates the rest of the game using purely random moves for each player as if it had made that move. The amount of simulations it runs for each move determines how smart the AI will appear. For each simulated win the AI adds 1 to the value of the initial move, for each loss, -1. After every possible move has been simulated the AI chooses the move that had the best value; the move that had the most wins in simulations accredited to it, and the AI plays this move against the player.

The pseudocode would likely appear as such:

```
Find all potential actions
For each action:
    Give action initial value of 0
    For each playout:
        Make a clone of the current game state
        Perform the action in the clone
        While the game has not ended:
            Choose random action for next player to take
            Perform action
        If the AI wins, add 1 to value
        Else minus 1 from value
    Push back value
Choose and perform action with best value
```

IMPLEMENTATION

In my implementation of the Monte-Carlo tree search, the first step is to get the current state of the game board:

```
PieceType** board = m_board->GetBoardState();
```

Following that the AI checks if there are any valid moves. If not, it concedes the game to the player. If one or more valid moves are found the AI continues.

As valid moves and the pieces that make the move are stored separately, the AI must look through its list of pieces to find the piece that matches the move to be made:

```
//find piece associated with move
for (int x = 0; x < 12; ++x)
                              {
       if (m_pieces[x].m_type == false)
       {
              if (((m_pieces[x].m_x == i - 1 &&
                    m_pieces[x].m_y == j - 1) ||
                   (m_pieces[x].m_x == i + 1 &&
                    m_pieces[x].m_y == j - 1)) &&
                    m_pieces[x].m_alive == true)
              {
                     currentPiecePos = x;
              }
//etc...
```

In retrospect I would have chosen a different way to implement this, perhaps by passing both valid moves and the piece that made it into the AI, removing the need for this search.

The AI then simulates that move and the rest of the game for a certain amount of playouts. In my implementation the number of playouts is 100:

```
for (int playout = 0; playout < 100; ++playout)

{
       //clone the game
       CheckerBoard clone = *m_board;
//etc...
```

In each playout the AI simulates moves for each player by choosing randomly from the valid moves:

```
while (!boardValid) //choose random valid move
{
       n = (int)(rand() % 8);
       m = (int)(rand() % 8);

       if (cloneBoard[n][m] == VALID)
              boardValid = true;
}
```

At the end of each playout the AI updates the value of the intital move:

```
//if win +1 to value
if (clone.GetWinState() == AI)
{
	value++;
}
else//-1 from value
{
	value--;
}
```

Finally the AI chooses the move with the best value and performs that move.