Complex Game Systems

Technical Design Document – AI

Andrew Ross Giannopoulos

Monte Carlo Tree Search

The AI technique I chose to implement in this project was the Monte Carlo Tree Search algorithm. This heuristic search algorithm uses random sampling of the search tree to determine the optimum path to take. The way the heuristic is calculated in the regular version of the algorithm is by conducting a specified number of random playouts of each leaf node (in the Checkers example, leaf nodes are possible moves to be considered) and making calculations based off how many playouts result in the AI's victory.

The main algorithm has 4 main steps:
Selection – child nodes are selected to expand down based off calculated heuristic values
Expansion – create a one or more child nodes off the selected node
Simulation – play a random game from the created node
Backpropagation – use the resulting playout to update heuristic values on ancestor nodes

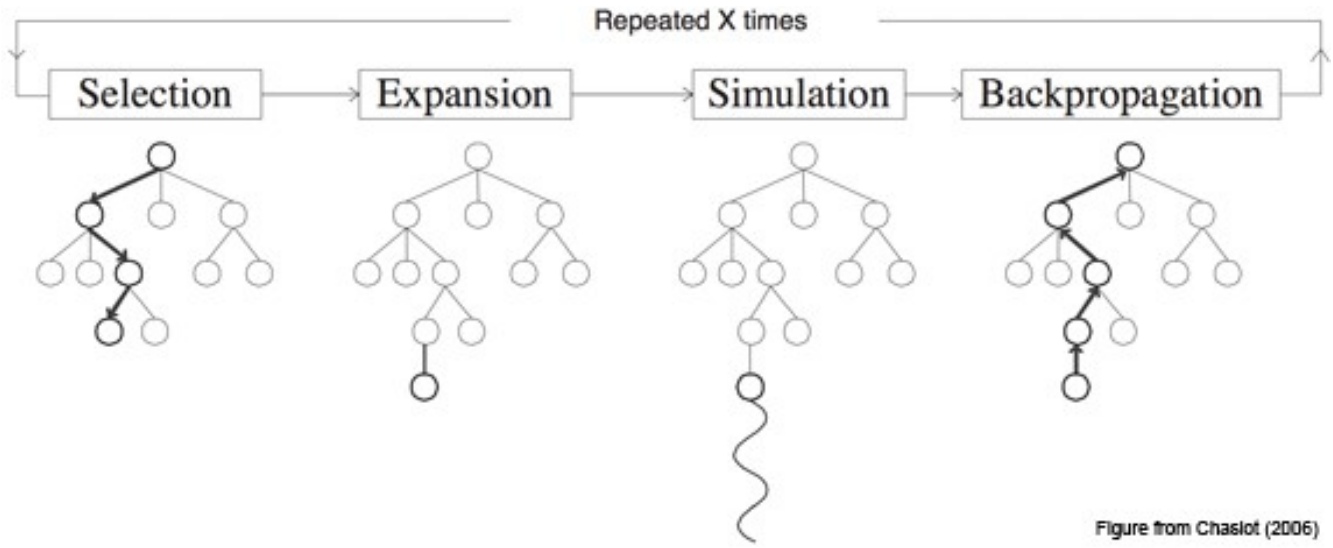

Figure from Chaslot (2006)

Due to the time constraints (AI choices must be calculated within 1 second) I opted to implement a simplified version of this algorithm that more or less eliminated the Selection and Backpropagation aspects of the algorithm. Of course in hindsight this may not have been such a good idea, as pre-calculating sections of the search tree on a different thread while the player makes their turn and caching results would likely have increased both time efficiency and effectiveness of the AI.

One notable optimization that was made was limiting the length of playouts and evaluating the fitness of the board at the end of the playout. It was found that by simply using random moves, simulated games could take a significant amount of time to resolve, time that could be better spend exploring

different playout scenarios. Limiting the possible depth of the simulated playout games to approximately 10-15 moves gave enough time for the effectiveness of a particular move to become apparent, and allowed me to increase the total number of playouts threefold.

The way I decided to evaluate the fitness was by simply taking taking a ratio of the AI's colour pieces of all pieces on the board, resulting in 0 if all pieces belong to the enemy, 1 if all pieces belong to the AI, and 0.5 if it's 50% each way. The average board fitness for each move was calculated and (instead of utilizing roulette selection) the move with the highest average fitness was simply chosen.

Below is a code excerpt from the final algorithm that was used.

```
for (int i = 0; i < possibleMovements.size(); i++)
{
	Board nextBoard = newBoard;
	Checkers::RunMove(nextBoard, possibleMovements[i]);

	// play a number of simulated games
	for (int j = 0; j < m_playouts; j++)
	{
		// create a board copy for each simulation
		Board tempBoard = nextBoard;

		// play the game to a certain depth
		for (int i = 0; i < DEPTH; i++)
		{
			if (moves.size() == 0)
			{
				break;
			}

			int index = rand() % moves.size();

			Movement chosenMove = moves[index];

			Checkers::RunMove(tempBoard, chosenMove);

			// swap turns
			if (turn == WHITE)
			{
				turn = BLACK;
			}
			else
			{
				turn = WHITE;
			}
		}

		fitnessValue += EvaluateFitness(tempBoard);
	}

	fitnessValue /= playouts;

}

return possibleMovements[bestMoveIndex];
```