Complex Game Systems

Technical Design Document

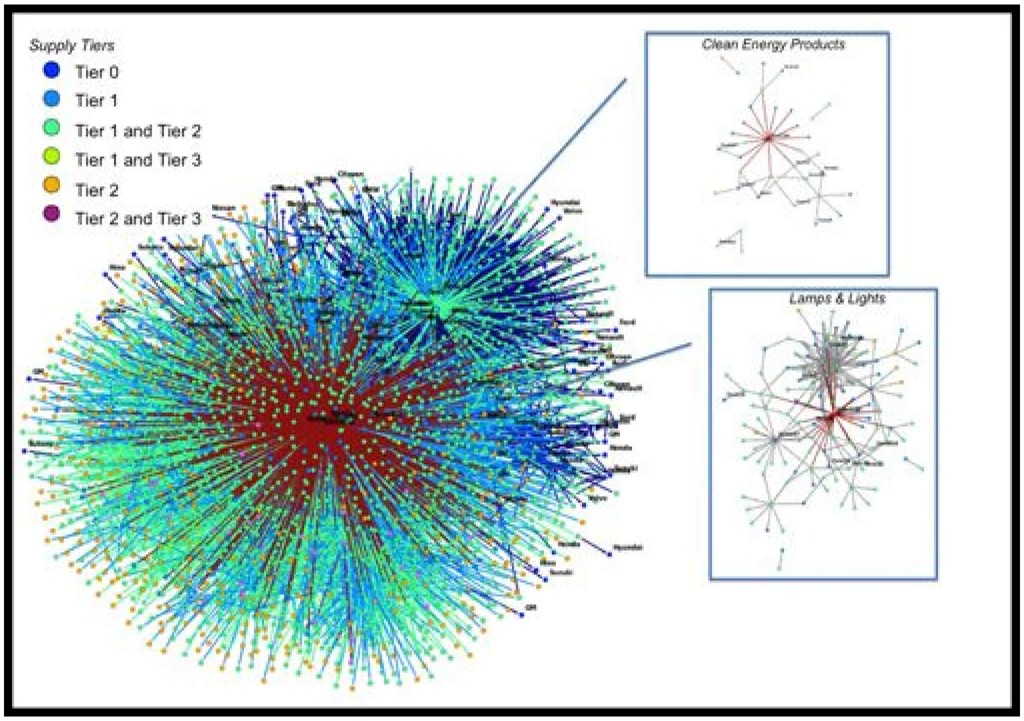
Jackson Luff

Part One: Artificial Intelligence

- What AI method did you use and why?
- What are the main steps of the AI method?
- Why and what games use this AI method?

Part Two: Implementation

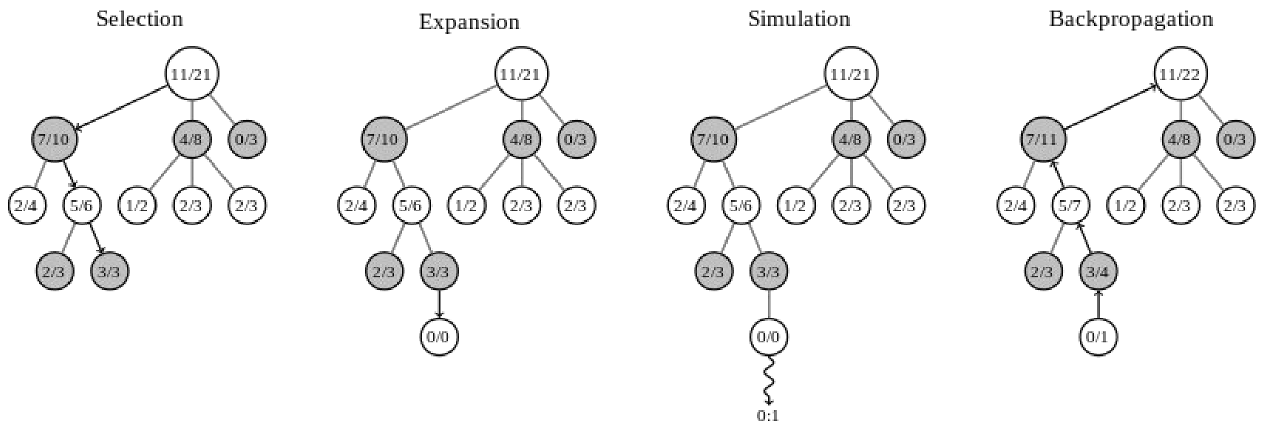
- How did you implement your AI method?
- What issues were faced when doing so?



Artificial Intelligence

Granted we are to create a 8x8 game of Checkers, I chose Monte Carlo Tree Search (MCTS) algorithm for the AI components functionality. As research has shown, MCTS has proven to be notably employed in game playing. So the question is, what is it? MCTS is a heuristic based search algorithm that randomly samples a search tree in order to return the optimum play to go forth with. For instance, in a regular MCTS the superior path is derived from the calculations of the random playthroughs that entail the most amount of success for the AI. This method uses a leaf node (child of the root) system.

The MCTS algorithm has 4 main steps:
Selection - Child nodes are selected to expand based on their heuristic values.
Expansion - Once selected, expand child nodes down by one or more.
Simulation - Goes forth with a random playthrough from expanded child node.
Backpropagation - Use resulted playthrough to update heuristic values on root nodes.



Being that MCTS is a heuristic based search algorithm, it becomes both simple and effective when it comes to making a decisions. (which explains the excessive use of it in the game industry)
Many AAA games have used this technique throughout their games, for example:
StarCraft - Tactical Planning
TOTAL WAR: ROME II - Campaign AI
Atari Cabinets - Offline Planning

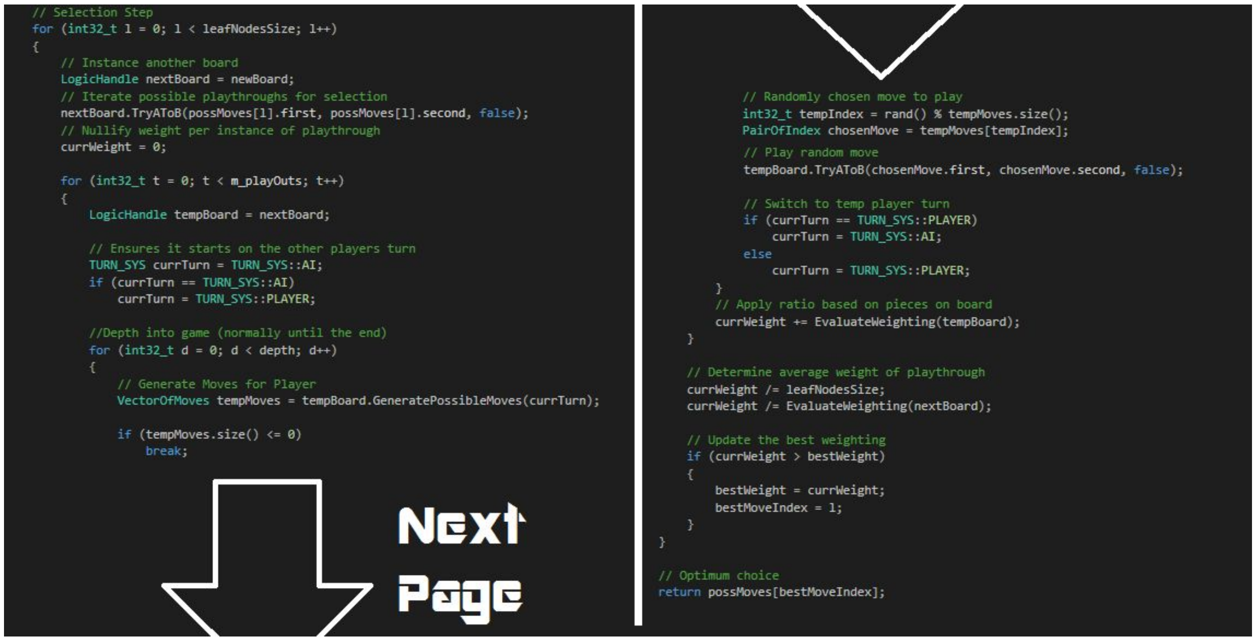
<u>Implementation:</u>

In this game of checkers, I focused on the regular version of MCTS. In saying that, due to the time constraints (Both application time and needed to calculate an optimum choice within a second) I captured the essence of MCTS instead. Thus, essentially cropping out the *Simulation* and *Backpropagation* stages. Although, probably not the best implementation to be had, it's still effective nonetheless.

Something i found worth noting (as it too is odd) is that even due to previously stated efforts, MCTS was still not as efficient as one could hope. Thus, I resulted in limiting the playthrough amount and only evaluating once at the end of a playthrough. My guess would be that cycling through such quantities of data would indeed be hefty upon use. What ended up as a pre-set value was a 'depth' variable. Instead of continuing until Game Over, the AI only proceeds x amount of depth in per game.

In order to keep things simply less kludgy, for the calculation of the node weightings, I simply determine the board ratio (0 - all to enemy, 1 - all to AI) and if it tips in the AI's favour. E.g. if I just received a resulting 0.75 board ratio, that means that the performed move tips 75% in my favour. Once iterated through all playthroughs, we return the board ratio with the highest average weight.

As shown in the code snippet below, I save off a move into a pair of indexes. The first of the pair is the current index, and the second being the target of which I want to move to.



```
// Selection Step
for (int32_t l = 0; l < leafNodesSize; l++)
{
    // Instance another board
    LogicHandle nextBoard = newBoard;
    // Iterate possible playthroughs for selection
    nextBoard.TryAToB(possMoves[l].first, possMoves[l].second, false);
    // Nullify weight per instance of playthrough
    currWeight = 0;

    for (int32_t t = 0; t < m_playOuts; t++)
    {
        LogicHandle tempBoard = nextBoard;

        // Ensures it starts on the other players turn
        TURN_SYS currTurn = TURN_SYS::AI;
        if (currTurn == TURN_SYS::AI)
            currTurn = TURN_SYS::PLAYER;

        //Depth into game (normally until the end)
        for (int32_t d = 0; d < depth; d++)
        {
            // Generate Moves for Player
            VectorOfMoves tempMoves = tempBoard.GeneratePossibleMoves(currTurn);

            if (tempMoves.size() <= 0)
                break;
```

```
            // Randomly chosen move to play
            int32_t tempIndex = rand() % tempMoves.size();
            PairOfIndex chosenMove = tempMoves[tempIndex];
            // Play random move
            tempBoard.TryAToB(chosenMove.first, chosenMove.second, false);

            // Switch to temp player turn
            if (currTurn == TURN_SYS::PLAYER)
                currTurn = TURN_SYS::AI;
            else
                currTurn = TURN_SYS::PLAYER;
        }
        // Apply ratio based on pieces on board
        currWeight += EvaluateWeighting(tempBoard);
    }

    // Determine average weight of playthrough
    currWeight /= leafNodesSize;
    currWeight /= EvaluateWeighting(nextBoard);

    // Update the best weighting
    if (currWeight > bestWeight)
    {
        bestWeight = currWeight;
        bestMoveIndex = l;
    }
}

// Optimum choice
return possMoves[bestMoveIndex];
```

References:

- AIE SlideShow - Game Tree Search
- https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
- http://aigamedev.com/open/coverage/mcts-rome-ii/
- https://project.dke.maastrichtuniversity.nl/games/files/bsc/Soemers_BSc-paper.pdf
- http://papers.nips.cc/paper/5421-deep-learning-for-real-time-atari-game-play-using-offline-monte-carlo-tree-search-planning.pdf