```
// Implementation of the expectimax algorithm.
// Description: expectimax is an adversarial search algorithm that assumes near-optimal play(stochastic 'mistakes' are permitted.)
//              it performs this by utilizing the idea of minimax (shown here with alpha-beta pruning heuristic - http://inst.eecs.berkeley.edu/~cs61b/fa14/ta-materials/apps/ab_tree_practice/)
//              but instead of treating the values as absolutes it will average the benefit of actions.
//              The importance of this algorithm can be demonstrated by visualizing the following...
//                  The World is an infinite 1-Dimensional Grid
//                  [ | | | | | | | | | | |]
//                  The Protagonist Needs to move 3 spaces right to reach it's goal giving it a score of +9999
//                  [ | | |P| | |G| | | | |]
//                  However an antagonistic agent is between...
//                  [ | | |P| |A|G| | | | |]
//                  Each move the current score ticks down by 1, touching the antagonist ends the game with the current score.
//
//                  minimax would assume that A will play perfectly and P will decide the most optimal action is to suicide into A.
//                  expectimax however doesn't assume that A is perfect and will leave things up to chance that A will move right twice
//                  giving P the chance to get a score of +9999
//
// The Problem: Unlike minimax expectimax doesn't have heuristics that are garunteed to return the absolute best action (Alpha-Beta Pruning)
//              certain heuristics can provide psuedo-optimal play such as a "Better Than Last Time" heuristic where it is short-circuited when
//              an action would put it in a better state than the previous action did. But this ignores the chance for 'big wins' which is a
//              strength of expectimax. For this reason parallelization is key to improving the performance of expectimax.

action* expectimax(state<action> s, int maxDepth);
float expectiminValue(action* currAction,state<action> s, int depth);
float expectimaxValue(action* currAction,state<action> s, int depth);
float expectimaxValueParallel(action* currAction,state<action> s, int depth, int remainingThreads);
float expectiminValueParallel(action* currAction,state<action> s, int depth, int remainingThreads);
action* expectimaxParallel(state<action> s, int maxDepth, int numThreads);

// Serial Implementation
action* expectimax(state<action> s, int maxDepth){
    // Begin with score of negative infinity
    float bestScore = -9999;
    action* bestAction = NULL;
    for(int i = 0; i < NUM_ACTIONS; i++){
        // Get Each Action
        action* currAction = (s.Agents[s.CurrentAgent]).Actions[i];
        //if(!(currAction->Possible(s))){ continue; }
        // Get the utility of each action.
        float expmaxVal = expectiminValue(currAction,s, maxDepth);
        // Check to see if the current action is better than previous.
        if(expmaxVal > bestScore){
            bestScore  = expmaxVal;
            bestAction = currAction;
        }
    }
    // return the best action to take.
    return bestAction;
}
```

```
float expectiminValue(action* currAction,state<action> s, int depth){
    // If end has been reached then evaluate the action and terminate.
    if(currAction->Evaluate(s) == 9999 || depth <= 0) return currAction->Evaluate(s);
    // Otherwise initialize the utility value as 0.
    float expVal = 0;
    // Get the next state.
    state<action> sS = currAction->Perform(s);
    for(int i = 0; i < NUM_ACTIONS; i++){
        // Get each of the successor actions..
        action* nextAction = (sS.Agents[sS.CurrentAgent]).Actions[i];
        // if(!(nextAction->Possible(sS))) continue;
        // get the utility of the resulting states.
        expVal += expectimaxValue(nextAction,sS, depth-1);
    }
    // Average the utility among possible actions.
    return expVal / NUM_ACTIONS;
}
// Essentially repeats the main function call, the catch is that the score is what is cared about, not the action.
float expectimaxValue(action* currAction,state<action> s, int depth){
    // If end has been reached then evaluate the action and terminate.
    if(currAction->Evaluate(s) == 9999 || depth <= 0) return currAction->Evaluate(s);
    // Otherwise initialize the best score as negative infinity.
    float bestScore = -9999;
    // Get the next state.
    state<action> sS = currAction->Perform(s);
    for(int i = 0; i < NUM_ACTIONS; i++){
        // Get each of the successor actions
        action* nextAction = (sS.Agents[sS.CurrentAgent]).Actions[i];
        //if(!(nextAction->Possible(sS))){ continue; }
        // Get the utility of each of the successor states.
        float expminVal = expectiminValue(nextAction, sS, depth-1);
        // Whenever a better utility is found use it.
        if(expminVal > bestScore) bestScore = expminVal;
    }
    // return the found utility.
    return bestScore;
}


// Parallel Implementation
int NeededThreads(int numThreads,int partitions){
    int output = (numThreads < NUM_ACTIONS) ? numThreads : NUM_ACTIONS;
    if(output <= 0){ output = 1; }
    return output;
}
action* expectimaxParallel(state<action> s, int maxDepth, int numThreads){
    action* bestAction = NULL;
    float expmaxVal[NUM_ACTIONS];

    /// First parallelizable section. This section provides the most benefit.
    /// A linear speedup should be possible until numThreads > NUM_ACTIONS
    int forThreads = NeededThreads(numThreads,NUM_ACTIONS);
    #pragma omp parallel for num_threads(forThreads)
    for(int i = 0; i < NUM_ACTIONS; i++){
        // Get Each Action
        action* currAction = (s.Agents[s.CurrentAgent]).Actions[i];
        // Get the utility of each action.
        expmaxVal[i] = expectiminValueParallel(currAction,s, maxDepth, numThreads - forThreads);
    }
    // Begin with score of negative infinity
    float bestScore = -9999;
    /// Unparallelizable section is moved out. Computation of bestScore depends on
```

```
    previous values.
    for(int i = 0; i < NUM_ACTIONS; i++){
        // Check to see if the current action is better than previous.
        if(expmaxVal[i] > bestScore){
            bestScore  = expmaxVal[i];
            bestAction = (s.Agents[s.CurrentAgent]).Actions[i];
        }
    }
    // return the best action to take.
    return bestAction;
}
float expectiminValueParallel(action* currAction,state<action> s, int depth, int remainingThreads){
    // If end has been reached then evaluate the action and terminate.
    if(currAction->Evaluate(s) == 9999 || depth <= 0) return currAction->Evaluate(s);
    // Otherwise initialize the utility value as 0.
    float expVal = 0;
    // Get the next state.
    state<action> sS = currAction->Perform(s);
    /// Second parallelizable section. Diminshing returns should be observed based on tree depth.
    /// The explicit split is because of how many instances may be created, the overhead of parallel_for on 1 thread was noticable.
    int forThreads = NeededThreads(remainingThreads,NUM_ACTIONS);
    if(forThreads > 1){
        #pragma omp parallel for num_threads(forThreads) reduction(+ : expVal)
        for(int i = 0; i < NUM_ACTIONS; i++){
            // Get each of the successor actions..
            action* nextAction = (sS.Agents[sS.CurrentAgent]).Actions[i];
            // get the utility of the resulting states.
            expVal += expectimaxValueParallel(nextAction,sS, depth-1, remainingThreads - forThreads);
        }
    } else {
        for(int i = 0; i < NUM_ACTIONS; i++){
            // Get each of the successor actions..
            action* nextAction = (sS.Agents[sS.CurrentAgent]).Actions[i];
            // get the utility of the resulting states.
            expVal += expectimaxValue(nextAction,sS, depth-1);
        }
    }
    // Average the utility among possible actions.
    return expVal / NUM_ACTIONS;
}
// Essentially repeats the main function call, the catch is that the score is what is cared about, not the action.
float expectimaxValueParallel(action* currAction,state<action> s, int depth, int remainingThreads){
    float expminVal[NUM_ACTIONS];
    // If end has been reached then evaluate the action and terminate.
    if(currAction->Evaluate(s) == 9999 || depth <= 0) return currAction->Evaluate(s);
    // Otherwise initialize the best score as negative infinity.
    float bestScore = -9999;
    // Get the next state.
    state<action> sS = currAction->Perform(s);
    /// Third parallelizable section. Diminshing returns should be observed based on tree depth.
    /// The explicit split is because of how many instances may be created, the overhead of parallel_for on 1 thread was noticable.
    int forThreads = NeededThreads(remainingThreads,NUM_ACTIONS);
    if(forThreads > 1){
        #pragma omp parallel for num_threads(forThreads)
        for(int i = 0; i < NUM_ACTIONS; i++){
            // Get each of the successor actions
```

```
                action* nextAction = (sS.Agents[sS.CurrentAgent]).Actions[i];
                //if(!(nextAction->Possible(sS))){ continue; }
                // Get the utility of each of the successor states.
                expminVal[i] = expectiminValueParallel(nextAction, sS, depth-1,
                remainingThreads);
            }
        } else {
            for(int i = 0; i < NUM_ACTIONS; i++){
                // Get each of the successor actions
                action* nextAction = (sS.Agents[sS.CurrentAgent]).Actions[i];
                //if(!(nextAction->Possible(sS))){ continue; }
                // Get the utility of each of the successor states.
                expminVal[i] = expectiminValue(nextAction, sS, depth-1);
            }
        }

        /// Second section which couldn't be parallelized. It is moved out.
        // Whenever a better utility is found use it.
        for(int i = 0; i < NUM_ACTIONS; i++)
            if(expminVal[i] > bestScore) bestScore = expminVal[i];
        // return the found utility.
        return bestScore;
    }
```