

```

1  // Implementation of the expectimax algorithm.
2  // Description: expectimax is an adversarial search algorithm that assumes near-optimal
   play(stochastic 'mistakes' are permitted.)
3  //           it performs this by utilizing the idea of minimax (shown here with
   alpha-beta pruning heuristic -
   http://inst.eecs.berkeley.edu/~cs61b/fal4/ta-materials/apps/ab_tree_practice/)
4  //           but instead of treating the values as absolutes it will average the
   benefit of actions.
5  //           The importance of this algorithm can be demonstrated by visualizing the
   following...
6  //           The World is an infinite 1-Dimensional Grid
7  //           [ | | | | | | | | | | ]
8  //           The Protagonist Needs to move 3 spaces right to reach it's goal
   giving it a score of +9999
9  //           [ | | |P| | |G| | | | ]
10 //           However an antagonistic agent is between...
11 //           [ | | |P| |A|G| | | | ]
12 //           Each move the current score ticks down by 1, touching the
   antagonist ends the game with the current score.
13 //
14 //           minimax would assume that A will play perfectly and P will decide
   the most optimal action is to suicide into A.
15 //           expectimax however doesn't assume that A is perfect and will leave
   things up to chance that A will move right twice
16 //           giving P the chance to get a score of +9999
17 //
18 // The Problem: Unlike minimax expectimax doesn't have heuristics that are garunteed to
   return the absolute best action (Alpha-Beta Pruning)
19 //           certain heuristics can provide psuedo-optimal play such as a "Better
   Than Last Time" heuristic where it is short-circuited when
20 //           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
21 //           strength of expectimax. For this reason parallelization is key to
   improving the performance of expectimax.
22
23 action* expectimax(state<action> s, int maxDepth);
24 float expectiminValue(action* currAction, state<action> s, int depth);
25 float expectimaxValue(action* currAction, state<action> s, int depth);
26 float expectimaxValueParallel(action* currAction, state<action> s, int depth, int
   remainingThreads);
27 float expectiminValueParallel(action* currAction, state<action> s, int depth, int
   remainingThreads);
28 action* expectimaxParallel(state<action> s, int maxDepth, int numThreads);
29
30 // Serial Implementation
31 action* expectimax(state<action> s, int maxDepth){
32     // Begin with score of negative infinity
33     float bestScore = -9999;
34     action* bestAction = NULL;
35     for(int i = 0; i < NUM_ACTIONS; i++){
36         // Get Each Action
37         action* currAction = (s.Agents[s.CurrentAgent]).Actions[i];
38         //if(!(currAction->Possible(s))){ continue; }
39         // Get the utility of each action.
40         float expmaxVal = expectiminValue(currAction,s, maxDepth);
41         // Check to see if the current action is better than previous.
42         if(expmaxVal > bestScore){
43             bestScore = expmaxVal;
44             bestAction = currAction;
45         }
46     }
47     // return the best action to take.
48     return bestAction;
49 }

```

```

50 float expectiminValue(action* currAction, state<action> s, int depth){
51     // If end has been reached then evaluate the action and terminate.
52     if(currAction->Evaluate(s) == 9999 || depth <= 0) return currAction->Evaluate(s);
53     // Otherwise initialize the utility value as 0.
54     float expVal = 0;
55     // Get the next state.
56     state<action> sS = currAction->Perform(s);
57     for(int i = 0; i < NUM_ACTIONS; i++){
58         // Get each of the successor actions..
59         action* nextAction = (sS.Agents[sS.CurrentAgent]).Actions[i];
60         // if(!(nextAction->Possible(sS))) continue;
61         // get the utility of the resulting states.
62         expVal += expectimaxValue(nextAction, sS, depth-1);
63     }
64     // Average the utility among possible actions.
65     return expVal / NUM_ACTIONS;
66 }
67 // Essentially repeats the main function call, the catch is that the score is what is
68 // cared about, not the action.
69 float expectimaxValue(action* currAction, state<action> s, int depth){
70     // If end has been reached then evaluate the action and terminate.
71     if(currAction->Evaluate(s) == 9999 || depth <= 0) return currAction->Evaluate(s);
72     // Otherwise initialize the best score as negative infinity.
73     float bestScore = -9999;
74     // Get the next state.
75     state<action> sS = currAction->Perform(s);
76     for(int i = 0; i < NUM_ACTIONS; i++){
77         // Get each of the successor actions
78         action* nextAction = (sS.Agents[sS.CurrentAgent]).Actions[i];
79         //if(!(nextAction->Possible(sS))) continue; }
80         // Get the utility of each of the successor states.
81         float expminVal = expectiminValue(nextAction, sS, depth-1);
82         // Whenever a better utility is found use it.
83         if(expminVal > bestScore) bestScore = expminVal;
84     }
85     // return the found utility.
86     return bestScore;
87 }
88
89 // Parallel Implementation
90 int NeededThreads(int numThreads, int partitions){
91     int output = (numThreads < NUM_ACTIONS) ? numThreads : NUM_ACTIONS;
92     if(output <= 0){ output = 1; }
93     return output;
94 }
95 action* expectimaxParallel(state<action> s, int maxDepth, int numThreads){
96     action* bestAction = NULL;
97     float expmaxVal[NUM_ACTIONS];
98
99     /// First parallelizable section. This section provides the most benefit.
100    /// A linear speedup should be possible until numThreads > NUM_ACTIONS
101    int forThreads = NeededThreads(numThreads, NUM_ACTIONS);
102    #pragma omp parallel for num_threads(forThreads)
103    for(int i = 0; i < NUM_ACTIONS; i++){
104        // Get Each Action
105        action* currAction = (s.Agents[s.CurrentAgent]).Actions[i];
106        // Get the utility of each action.
107        expmaxVal[i] = expectiminValueParallel(currAction, s, maxDepth, numThreads -
108        forThreads);
109    }
110    // Begin with score of negative infinity
111    float bestScore = -9999;
112    /// Unparallelizable section is moved out. Computation of bestScore depends on

```

```

previous values.
112 for(int i = 0; i < NUM_ACTIONS; i++){
113     // Check to see if the current action is better than previous.
114     if(expmaxVal[i] > bestScore){
115         bestScore = expmaxVal[i];
116         bestAction = (s.Agents[s.CurrentAgent]).Actions[i];
117     }
118 }
119 // return the best action to take.
120 return bestAction;
121 }
122 float expectiminValueParallel(action* currAction, state<action> s, int depth, int
remainingThreads){
123     // If end has been reached then evaluate the action and terminate.
124     if(currAction->Evaluate(s) == 9999 || depth <= 0) return currAction->Evaluate(s);
125     // Otherwise initialize the utility value as 0.
126     float expVal = 0;
127     // Get the next state.
128     state<action> sS = currAction->Perform(s);
129     /// Second parallelizable section. Diminshing returns should be observed based on
tree depth.
130     /// The explicit split is because of how many instances may be created, the
overhead of parallel_for on 1 thread was noticable.
131     int forThreads = NeededThreads(remainingThreads, NUM_ACTIONS);
132     if(forThreads > 1){
133         #pragma omp parallel for num_threads(forThreads) reduction(+ : expVal)
134         for(int i = 0; i < NUM_ACTIONS; i++){
135             // Get each of the successor actions..
136             action* nextAction = (sS.Agents[sS.CurrentAgent]).Actions[i];
137             // get the utility of the resulting states.
138             expVal += expectimaxValueParallel(nextAction, sS, depth-1, remainingThreads
- forThreads);
139         }
140     } else {
141         for(int i = 0; i < NUM_ACTIONS; i++){
142             // Get each of the successor actions..
143             action* nextAction = (sS.Agents[sS.CurrentAgent]).Actions[i];
144             // get the utility of the resulting states.
145             expVal += expectimaxValue(nextAction, sS, depth-1);
146         }
147     }
148     // Average the utility among possible actions.
149     return expVal / NUM_ACTIONS;
150 }
151 // Essentially repeats the main function call, the catch is that the score is what is
cared about, not the action.
152 float expectimaxValueParallel(action* currAction, state<action> s, int depth, int
remainingThreads){
153     float expminVal[NUM_ACTIONS];
154     // If end has been reached then evaluate the action and terminate.
155     if(currAction->Evaluate(s) == 9999 || depth <= 0) return currAction->Evaluate(s);
156     // Otherwise initialize the best score as negative infinity.
157     float bestScore = -9999;
158     // Get the next state.
159     state<action> sS = currAction->Perform(s);
160     /// Third parallelizable section. Diminishing returns should be observed based on
tree depth.
161     /// The explicit split is because of how many instances may be created, the
overhead of parallel_for on 1 thread was noticable.
162     int forThreads = NeededThreads(remainingThreads, NUM_ACTIONS);
163     if(forThreads > 1){
164         #pragma omp parallel for num_threads(forThreads)
165         for(int i = 0; i < NUM_ACTIONS; i++){
166             // Get each of the successor actions

```

```

167         action* nextAction = (sS.Agents[sS.CurrentAgent]).Actions[i];
168         //if(!(nextAction->Possible(sS)){ continue; }
169         // Get the utility of each of the successor states.
170         expminVal[i] = expectiminValueParallel(nextAction, sS, depth-1,
            remainingThreads);
171     }
172 } else {
173     for(int i = 0; i < NUM_ACTIONS; i++){
174         // Get each of the successor actions
175         action* nextAction = (sS.Agents[sS.CurrentAgent]).Actions[i];
176         //if(!(nextAction->Possible(sS)){ continue; }
177         // Get the utility of each of the successor states.
178         expminVal[i] = expectiminValue(nextAction, sS, depth-1);
179     }
180 }
181
182 /// Second section which couldn't be parallelized. It is moved out.
183 // Whenever a better utility is found use it.
184 for(int i = 0; i < NUM_ACTIONS; i++)
185     if(expminVal[i] > bestScore) bestScore = expminVal[i];
186 // return the found utility.
187 return bestScore;
188 }

```