

```

1  #include<cmath>
2  #define NUM_AGENTS 2
3  #define NUM_ACTIONS 4
4  #define NUM_OBSTACLES 2
5
6  // The world is an infinite two-dimensional grid. As such a physical array is not
   needed, only positions.
7
8  // Data component of agent.
9  template<typename ACTION_TYPE>
10 struct agent{
11     // Current Position of Agent
12     int x, y;
13     // The list of actions the agent can perform.
14     ACTION_TYPE* Actions[NUM_ACTIONS];
15 };
16
17 // State Information
18 template<typename ACTION_TYPE>
19 struct state{
20     // The goal of the protagonist agent. The goal of the antagonnist agent is to touch
   the protagonist.
21     agent<ACTION_TYPE> Goal;
22     // Current Agent
23     int CurrentAgent;
24     // Which agent is the protagonist.
25     int ProtagonistID;
26     // The list of impassible spaces.
27     agent<ACTION_TYPE> Obstacles[NUM_OBSTACLES];
28     // The list of agents in the space.
29     agent<ACTION_TYPE> Agents[NUM_AGENTS];
30 };
31
32 // Action Interface
33 struct action{
34     // How much utility is there in performing the action right now?
35     virtual int Evaluate(state<action>) = 0;
36     // Can the action actually be performed?
37     virtual bool Possible(state<action>) = 0;
38     // What is the state returned by the action being performed?
39     virtual state<action> Perform(state<action>) = 0;
40 };
41
42
43 int PrevAgent(state<action> s){
44     return (s.CurrentAgent-1) % NUM_AGENTS;
45 }
46
47 // Computes the manhattan distance between two points.
48 int ManhattanDistance(agent<action> a, agent<action> b){
49     return abs(b.x - a.x) + abs(b.y - a.y);
50 }
51
52 bool Overlap(agent<action> a, agent<action> b){
53     return a.x == b.x && a.y == b.y;
54 }
55
56 // Defines common parts of the interface for moving.
57 struct moveAction : action {
58     int Evaluate(state<action> s){
59         int dist; s = Perform(s);
60         agent<action> dest, winDest;
61
62         if(PrevAgent(s) == s.ProtagonistID){

```

```

63         // Protagonist Lose Condition
64         for(int i = 0; i < NUM_AGENTS; i++){
65             if(i == s.ProtagonistID) continue;
66             if(Overlap(s.Agents[PrevAgent(s)],(s.Agents[i]))) return -9999;
67         }
68         dest = s.Goal;
69     } else {
70         dest = s.Agents[s.ProtagonistID];
71     }
72     if(Overlap(s.Agents[PrevAgent(s)],dest)) return 9999;
73     dist = ManhattanDistance(s.Agents[PrevAgent(s)],dest);
74     return (dist == 0) ? 2000 : 1000/dist;
75 }
76
77 bool Possible(state<action> s){
78     s = Perform(s);
79     for(int i = 0; i < NUM_OBSTACLES; i++)
80         if(Overlap(s.Agents[PrevAgent(s)],s.Obstacles[i])) return false;
81     return true;
82 }
83 };
84
85 // The following four classes represent moving in the four cardinal directions.
86 struct moveLeft : moveAction{
87     state<action> Perform(state<action> s){
88         int i = s.CurrentAgent;
89         s.Agents[i].x -= 1;
90         s.CurrentAgent = (i+1) % NUM_AGENTS;
91         return s;
92     }
93 };
94 struct moveRight : moveAction{
95     state<action> Perform(state<action> s){
96         int i = s.CurrentAgent;
97         s.Agents[i].x += 1;
98         s.CurrentAgent = (i+1) % NUM_AGENTS;
99         return s;
100     }
101 };
102 struct moveUp : moveAction{
103     state<action> Perform(state<action> s){
104         int i = s.CurrentAgent;
105         s.Agents[i].y -= 1;
106         s.CurrentAgent = (i+1) % NUM_AGENTS;
107         return s;
108     }
109 };
110 struct moveDown : moveAction{
111     state<action> Perform(state<action> s){
112         int i = s.CurrentAgent;
113         s.Agents[i].y += 1;
114         s.CurrentAgent = (i+1) % NUM_AGENTS;
115         return s;
116     }
117 };

```