

Action Deception: Synthesizing AoE example

Contributors: Mohammed Tousif Zaman, Abhishek N. Kulkarni, Jie Fu

This code implements the work titled - Synthesizing Action Deceptive Strategy in Two-player Strategy
In this code, we show the construction of transition system and proceed with building the game graph discussed in the case study.

The structure of code is as follows:

1. Data structures
2. Solution Algorithms
3. Parameters for the AoE game scenario
4. Construction of Hypergame, Game States and Game Edges
5. Game Construction with Perfect Information and Solution
6. Hypergame Construction and Solution
7. Results Summary for AoE game scenario

References: The code structure is taken from the implementation of the example highlighted in IJCAI and parts 3-7 are adapted to suit the scenario in the above Case Study.

▼ 1. Data structures

1. Graph: Base class
2. Game: Represents perfect information game
3. Hypergame: Represents game with action misperception

```
1 from typing import Iterable
2
3 # DS1: Graph
4 class Vertex(object):
5     pass
6
7
8 class Edge(object):
9     def __init__(self, u: 'Graph.Vertex', v: 'Graph.Vertex'):
10         self._source = u
11         self._target = v
12
13     @property
14     def source(self):
15         """ Returns the source vertex of edge. """
16         return self._source
```

```

17
18     @property
19     def target(self):
20         """ Returns the target vertex of edge. """
21         return self._target
22
23
24 class Graph(object):
25     def __init__(self):
26         # Dict: {vertex: (set(<in-edge>), set(<out-edge>))}
27         self._vertex_edge_map = dict()
28         self._edges = set()          # Set of all edges of graph
29
30     def __repr__(self):
31         return f"{self.__class__.__name__}\
32         (|V|={len(self._vertex_edge_map)}, |E|={len(self._edges)})"
33
34     def add_edge(self, e):
35         u = e.source
36         v = e.target
37
38         self._vertex_edge_map[u][1].add(e)
39         self._vertex_edge_map[v][0].add(e)
40         self._edges.add(e)
41
42     def add_vertex(self, v):
43         self._vertex_edge_map[v] = (set(), set())
44
45     def in_edges(self, v):
46         if isinstance(v, Vertex):
47             return list(self._vertex_edge_map[v][0])
48
49         elif isinstance(v, Iterable):
50             in_edges = (self._vertex_edge_map[u][0] for u in v)
51             return list(reduce(set.union, in_edges))
52
53         raise AssertionError(f"Vertex {v} must \
54         be a single or an iterable of {Vertex} objects.")
55
56     def out_edges(self, v):
57         if isinstance(v, Vertex):
58             return list(self._vertex_edge_map[v][1])
59
60         elif isinstance(v, Iterable):
61             out_edges = (self._vertex_edge_map[u][1] for u in v)
62             return list(reduce(set.union, out_edges))
63
64         raise AssertionError(f"Vertex {v} must \
65         be a single or an iterable of {Vertex} objects.")
66
67     def out_neighbors(self, v):
68         if isinstance(v, Vertex):

```

```

69         return list(e.target for e in self._vertex_edge_map[v][1])
70
71     elif isinstance(v, Iterable):
72         return list(e.target for u in v for e in \
73                     self._vertex_edge_map[u][1])
74
75     raise AssertionError(f"Vertex {v} must \
76     be a single or an iterable of {Vertex} objects.")
77
78     @property
79     def edges(self):
80         return list(self._edges)
81
82     @property
83     def vertices(self):
84         return list(self._vertex_edge_map.keys())
85

```

```

1 # DS2: Game
2
3 class GameVertex(Vertex):
4     def __init__(self, name, turn):
5         self._name = name
6         self._turn = turn
7
8     def __hash__(self):
9         return self.name.__hash__()
10
11     def __eq__(self, other):
12         return self.name == other.name and self.turn == other.turn
13
14     @property
15     def name(self):
16         """ Returns the name of game vertex. """
17         return self._name
18
19     @property
20     def turn(self):
21         """ Returns the id of player who will make move in current state. """
22         return self._turn
23
24
25 class GameEdge(Edge):
26     def __init__(self, u, v, act):
27         super().__init__(u, v)
28         self._act = act
29
30     @property
31     def act(self):
32         """ Returns action associated with game edge. """
33         return self._act

```

```

34
35
36 class Game(Graph):
37     def __init__(self):
38         super().__init__()
39         self._final = set()
40
41     @property
42     def final(self):
43         """ Returns the set of final states of the game. """
44         return self._final
45
46     def mark_final(self, v):
47         """
48         Adds the given state to the set of final states in the game.
49
50         :param v: (:class:`Game.Vertex`) Vertex to be marked as final.
51         """
52         if v in self._vertex_edge_map:
53             self._final.add(v)
54

```

```

1 # DS3: Hypergame
2
3 class HState:
4     """Represents Hypergame State"""
5     # state = (s1, s2, turn, q, i)
6     # s1 = ((pos_v, pos_k, pos_a), (health_v, health_k, health_a), res)
7     # s2 = (health_tower1, health_tower2)
8     # st = (((pos_v, pos_k, pos_a), (health_v, health_k, health_a), res), \
9     #       (health_tower1, health_tower2), turn, q, i)
10
11     def __init__(self, state):
12         self._state = state
13         self.type = None
14
15     def __hash__(self):
16         return hash(self._state)
17
18     def __eq__(self, other):
19         return self._state == other._state
20
21     def __str__(self):
22         return f"HState(s: {self.arena_state}, q: {self.aut_state}, \
23                    i: {self.igraph_state})"
24
25     __repr__ = __str__
26
27     @property
28     def p1_loc(self):
29         return self._state[0][0]

```

```

30
31     @property
32     def p1_health(self):
33         return self._state[0][1]
34
35     @property
36     def p1_res(self):
37         return self._state[0][2]
38
39     @property
40     def p2_health(self):
41         return self._state[1]
42
43     @property
44     def turn(self):
45         return self._state[2]
46
47     @property
48     def arena_state(self):
49         return self._state[0:3]
50
51     @property
52     def aut_state(self):
53         return self._state[3]
54
55     @property
56     def igrph_state(self):
57         return self._state[4]
58
59
60 class HypergameVertex(Vertex):
61
62     # -----
63     # INTERNAL CLASSES
64     # -----
65     def __init__(self, hstate):
66         # self._name = f"(game_v={game_v.name}, igrph_v={igrph_v.name})" \
67             # str((game_v.name, igrph_v.name))
68         self._hstate = hstate
69         self._name = str(hstate)
70         self._game_v = hstate.arena_state + (hstate.aut_state, )
71         self._igrph_v = hstate.igrph_state
72         self._turn = hstate.turn
73
74     def __repr__(self):
75         string = f"Vertex(name={self._name}"
76         return string
77
78     def __hash__(self):
79         return hash(self._hstate)
80
81     def eq (self, other):

```

```

81     def __eq__(self, other):
82         if isinstance(other, HypergameVertex):
83             return self._hstate == other._hstate
84         elif isinstance(other, HState):
85             return self._hstate == other
86
87     # -----
88     # PUBLIC PROPERTIES
89     # -----
90     @property
91     def name(self):
92         """ Returns the name of game vertex. """
93         return self._name
94
95     @property
96     def turn(self):
97         """ Returns the id of player who will make move in current state. """
98         return self._turn
99
100    @property
101    def game_vertex(self):
102        return self._game_v # pragma: no cover
103
104    @property
105    def igraph_vertex(self):
106        return self._igraph_v # pragma: no cover
107
108    @property
109    def hstate(self):
110        return self._hstate
111
112
113    class HypergameEdge(Edge):
114        def __init__(self, u, v, act):
115            super().__init__(u, v)
116            self._act = act
117
118        @property
119        def act(self):
120            """ Returns action associated with game edge. """
121            return self._act
122
123
124    class Hypergame(Graph):
125        def __init__(self):
126            super().__init__()
127            self._final = set()
128
129        @property
130        def final(self):
131            """ Returns the set of final states of the game. """
132            return self._final

```

```

133
134     def mark_final(self, v):
135         """
136         Adds the given state to the set of final states in the game.
137         :param v: (:class:`Game.Vertex`) Vertex to be marked as final.
138         """
139         if v in self._vertex_edge_map:
140             self._final.add(v)

```

▼ 2. Solution Algorithms

1. SureWinning: Computes sure winning region in perfect information game
2. DeceptiveAlmostSureWinning: Computes almost-sure winning region in a game with action mis

```

1 # Algorithm 1: Sure winning region computation
2
3 class SureWinning(object):
4     def __init__(self, game):
5         self._game = game
6         self._p1_win = set()
7         self._p2_win = set()
8
9     @property
10    def p1_win(self):
11        """ Returns the P1's winning region. Returns None, \
12            if the solver has not solved the game. """
13        return self._p1_win
14
15    def _pre1(self):
16
17        # Initialize an empty set
18        pre1 = set()
19
20        # Iterate over all states in winning region
21        for v in self._p1_win:
22
23            # Iterate over all incoming edges to check all \
24            # potential states to add to Pre1
25            for e in self._game.in_edges(v):
26
27                # Get the candidate vertex to add to Pre1
28                u = e.source
29
30                # If u is P1's vertex and not already added, then add it to Pre1
31                if u.turn == 1 and u not in self._p1_win:
32                    pre1.add(u)
33
34        return pre1
35

```

```

36     def _pre2(self):
37         # Initialize an empty set
38         pre2 = set()
39
40         # Iterate over all states in winning region
41         for v in self._p1_win:
42
43             # Iterate over all incoming edges to check all \
44             # potential states to add to Pre1
45             for e in self._game.in_edges(v):
46
47                 # Get the candidate vertex to add to Pre1
48                 u = e.source
49
50                 # If u is P2's vertex AND not already added AND all outgoing \
51                 # edges are winning, then add it to Pre1
52                 if u.turn == 2 and u not in self._p1_win and \
53                     set(self._game.out_neighbors(u)).issubset(self._p1_win):
54                     pre2.add(u)
55
56         return pre2
57
58     def solve(self):
59         """ Runs the solver. """
60         # Initialize/Reset Solution data structures
61         final = set()
62         for v in self._game.vertices:
63             if v in self._game.final:
64                 final.add(v)
65
66         # Zielonka's algorithm
67         self._p1_win = final
68         while True:
69             pre1 = self._pre1()
70             pre2 = self._pre2()
71             p1_win = set.union(self._p1_win, pre1, pre2)
72
73             if p1_win == self._p1_win:
74                 break
75
76         self._p1_win = p1_win

```

1 # Algorithm 2: Deceptive Almost-Sure Winning Region Computation

2

```

3 class DeceptiveAlmostSureWinning(object):
4     def __init__(self, hypergame, is_permissive):
5         self._hypergame = hypergame
6         # Function which checks whether given edge is permissive or not.
7         self.is_permissive = is_permissive
8         self._p1_win = set()
9         self._p2_win = set()

```



```

10
11 @property
12 def p1_win(self):
13     """ Returns the P1's winning region. Returns None, \
14         if the solver has not solved the game. """
15     return self._p1_win
16
17 def _dapre11(self, Uk):
18     """DAPre_1^1"""
19
20     dapre11 = set()
21
22     for u in Uk:
23         # If turn of vertex is not P1, inspect next vertex
24         if u.turn != 1:
25             continue
26
27         # If final, keep it (equivalent of making final states as sink states)
28         if u in self.final:
29             dapre11.add(u)
30             continue
31
32         # Check if any of the neighbors are included in Uk
33         out_neighbors = self._hypergame.out_neighbors(u)
34         if len(set(out_neighbors).intersection(Uk)) > 0:
35             dapre11.add(u)
36
37     return dapre11
38
39 def _dapre12(self, Uk):
40     """DAPre_1^2"""
41     dapre12 = set()
42
43     for u in Uk:
44         # If turn of vertex is not P2, inspect next vertex
45         if u.turn != 2:
46             continue
47
48         # If final, keep it (equivalent of making final states as sink states)
49         if u in self.final:
50             dapre12.add(u)
51             continue
52
53         # Check if any of the neighbors are included in Uk
54         out_edges = self._hypergame.out_edges(u)
55         out_neighbors = list()
56
57         for e in out_edges:
58             if self.is_permissive(e):
59                 out_neighbors.append(e.target)
60
61         if set(out_neighbors).issubset(Uk):

```

```

62         dapre12.add(u)
63
64     return dapre12
65
66     def _dapre21(self, Uk):
67         """DAPre_2^1"""
68         dapre21 = set()
69
70         for u in Uk:
71             # If turn of vertex is not P1, inspect next vertex
72             if u.turn != 1:
73                 continue
74
75             # Check if any of the neighbors are included in Uk
76             out_neighbors = self._hypergame.out_neighbors(u)
77             if set(out_neighbors).issubset(Uk):
78                 dapre21.add(u)
79
80         return dapre21
81
82     def _dapre22(self, Uk):
83         """DAPre_2^2"""
84         dapre22 = set()
85
86         for u in Uk:
87             # If turn of vertex is not P1, inspect next vertex
88             if u.turn != 2:
89                 continue
90
91             # Check if any of the neighbors are included in Uk
92             out_edges = self._hypergame.out_edges(u)
93             perm_neighbors = list()
94             for e in out_edges:
95                 if self.is_permissive(e):
96                     perm_neighbors.append(e.target)
97
98             if set(perm_neighbors).issubset(Uk):
99                 dapre22.add(u)
100
101
102     return dapre22
103
104     def _safe_1(self, Uk):
105
106         # Initialize an empty set
107         safe1 = Uk
108         # print("\t=====")
109         # print("\t", f"Safe-1({Uk})")
110
111         while True:
112             dapre1 = self._dapre11(safe1)
113             # print("\t", f"Safe-1({Uk})")

```



```

165         p1_win = self._safe_1(set.difference(set(self._hypergame.vertices),\
166                                             Ck[-1]))
167         if p1_win == self._p1_win:
168             break
169
170         self._p1_win = p1_win
171

```

▼ Game Parameters

```

1 # Global Variables
2 REGIONTYPE_FREE = "free"
3 REGIONTYPE_TOWER = "tower"
4 REGIONTYPE_OBST = "obst"
5 REGIONTYPE_STONE = "stone"
6 REGIONTYPE_RELIC = "relic"
7
8 # Configuration settings for AoE example
9 REGIONS = (1, 2, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16)
10 TOWERS = (6, 9)
11 P1_STONE = (1,)
12 P2_STONE = (16,)
13 SAFE = (1, 2)
14 UNSAFE = tuple(set(REGIONS) - set(SAFE))
15 RELIC = (14,)
16 TOWER_INNER = (7, 10)
17 TOWER_OUTER = (8, 11, 13, 14)
18 TOWER1 = (7, 8, 13, 14)
19 TOWER2 = (10, 11, 13, 14)
20 FREE_REGIONS = tuple(set(REGIONS) - set(TOWERS))
21 TOWER_REGIONS = tuple(TOWER_INNER + TOWER_OUTER)
22 STONE_REGIONS = tuple(P1_STONE + P2_STONE)
23 COST_CASTLE = 500
24 FIXED_RESOURCE_STEP = 10
25 FIXED_HEALTH_STEP = 10
26 LOW_HIT_STEP = 5
27 HIGH_HIT_STEP = 10
28
29 # Initial State in the AoE example
30 v0_tuple = (((2, 2, 2), ("high", "high", "high"), 1), ("high", "high"), 1, 0, 0)
31
32 # print(f"Safe regions = {SAFE}.")
33 # print(f"Unsafe regions = {UNSAFE}.")
34 # print(f"Reachable regions = {FREE_REGIONS}.")

```

▼ Construction of States and Edges for Game and Hypergame

1. Construct hypergame states

2. Transition function for automaton (DFA)
3. Inference function for player 2 (P2)
4. Action functions for P1 and P2
5. Classes and helper functions for tracking game level information
6. Construct edges of the hypergame
7. Eliminate unreachable states in hypergame
8. Projection of hypergame states and edges onto the game

```

1 import itertools
2
3 # Labelling function for hypergame states with type labels
4 def hstate_type(hstate):
5     p1_position = hstate.p1_loc
6     p1_res = hstate.p1_res
7     pos_v = p1_position[0]
8     pos_k = p1_position[1]
9     pos_a = p1_position[2]
10
11     if pos_v in TOWERS or pos_k in TOWERS or pos_a in TOWERS:
12         return REGIONTYPE_OBST
13
14     elif pos_v in STONE_REGIONS:
15         return REGIONTYPE_STONE
16
17     elif pos_v in RELIC:
18         return REGIONTYPE_RELIC
19
20     elif pos_v in TOWER_REGIONS or pos_k in TOWER_REGIONS or pos_a in\
21         TOWER_REGIONS:
22         return REGIONTYPE_TOWER
23
24     else:
25         return REGIONTYPE_FREE
26
27
28 # Construct Hypergame States
29 P1_REGIONS = [FREE_REGIONS, FREE_REGIONS, FREE_REGIONS]
30 p1_pos = list(itertools.product(*P1_REGIONS))
31 p1_health = tuple(itertools.product(("low", "med", "high"), repeat = 3))
32 # p1_health = tuple(itertools.product(range(0, 101, 34), repeat = 3))
33 p1_res = (0, 1, 2, 3)
34
35 # Approach 1
36 p1_list = [p1_pos, p1_health, p1_res]
37 p1_states = list(itertools.product(*p1_list))
38 p2_states = tuple(itertools.product(("low", "med", "high"), repeat = 2))
39 # p2_states = tuple(itertools.product(range(0, 101, 34), repeat = 2))
40

```

```

40
41 # Approach 2
42 # p1_states = tuple(itertools.product(p1_pos, p1_health, p1_res))
43 # p2_states = tuple(itertools.product(range(0, 101, 34), repeat = 2))
44
45 turn = (1, 2)
46 aut_states = (0, 1, 2)
47 igrph_states = (0, 1, 2, 3, 4, 5, 6, 7)
48 hstates = tuple(itertools.product(p1_states, p2_states, turn, aut_states,\
49                                     igrph_states))
50 hmap = dict(zip(hstates, map(HState, hstates)))
51 print(f"We have {len(hmap)} number of hypergame vertices.")
52 # print(f"Length of hstates = {len(hstates)}.")
53 # print(f"Length of p1_pos = {len(p1_pos)}, p1_health = {len(p1_health)},\
54 #                                             p1_res = {len(p1_res)}.")
55 # print(f"Length of p1_states = {len(p1_states)}, p2_states = {len(p2_states)}.")
56 # print(f"Length of hstates = {len(p1_states)*len(p2_states)*2*3*8}\
57 #                                             (manually calculated)")
58
59 # Label states with types
60 for val in hmap.values():
61     val.type = hstate_type(val)

```

☞ We have 46656000 number of hypergame vertices.

```

1 # Define Automaton Transition Function
2
3 def aut_trans(q, p1_loc):
4     """ Specification: Eventually(RELIC & Eventually(SAFE)) """
5
6     # DFA considers location of P1 villager unit
7     v_loc = u.p1_loc[0]
8     relic_collected = False
9     if q == 0:
10         if v_loc in RELIC:
11             relic_collected = True
12             return 1
13         # direct transition (unreachable)
14         elif v_loc in SAFE and relic_collected == True:
15             return 2
16         else:
17             return 0
18
19     elif q == 1:
20         if v_loc in SAFE:
21             return 2
22         else:
23             return 1
24
25     elif q == 2:
26         return 2
27

```

```
28     else:
29         raise ValueError("Unknown automaton state.")
```

```
1 # Define P2 Information Graph Transition Function
2
3 def igrph_trans(i, action_name, res):
4     """
5     Inference Graph (IGraph) has 8 states.
6     0: P2 does not know "suicide" and does not know if P1, P2 resources are collected.
7     1: P2 knows "suicide" and does not know if P1, P2 resources are collected.
8     2: P2 does not know "suicide" and knows only P2 resource is collected.
9     3: P2 knows "suicide" and knows only P2 resource is collected.
10    4: P2 does not know "suicide" and knows only P1 resource is collected.
11    5: P2 knows "suicide" and knows only P1 resource is collected.
12    6: P2 does not know "suicide" and knows both P1, P2 resources is collected.
13    7: P2 knows "suicide" and knows both P1, P2 resources is collected.
14    """
15
16    # considering one action based transitions, at a time
17    # check v_pos to determine the resources collected
18    if i == 0:
19        if "suicide" == action_name:
20            return 1
21        elif res == 1:
22            return 2
23        elif res == 2:
24            return 4
25        # elif res == 3:
26        #     return 6
27    else:
28        return 0
29
30    elif i == 1:
31        if res == 1:
32            return 3
33        elif res == 2:
34            return 5
35        # elif res == 3:
36        #     return 7
37    else:
38        return 1
39
40    elif i == 2:
41        if "suicide" == action_name:
42            return 3
43        elif res == 3:
44            return 6
45        else:
46            return 2
47
48    elif i == 3:
```

```

49         if res == 3:
50             return 7
51         else:
52             return 3
53
54     elif i == 4:
55         if "suicide" == action_name:
56             return 5
57         elif res == 3:
58             return 6
59         else:
60             return 4
61
62     elif i == 5:
63         if res == 3:
64             return 7
65         else:
66             return 5
67
68     elif i == 6:
69         if "suicide" == action_name:
70             return 7
71         else:
72             return 6
73
74     elif i == 7:
75         return 7
76
77     else:
78         raise ValueError("Unknown igrph state.")

```

▼ Classes and helper functions for tracking game level information

```

1 # Class for the units and players
2 # Track game level information such as player resources and unit health
3 class Unit:
4     def __init__(self, name, health):
5         self._name = name
6         self._health = max(health, 0)
7
8     def __str__(self):
9         return f"Name of unit: {self._name}"
10
11     def get_health(self):
12         if self._health >= 0:
13             return self._health
14         # else:
15         #     # print(f"{self._name} is dead.")
16         #     return self._health
17

```



```

17
18     def set_health(self, hval):
19         self._health = max(hval, 0)
20         # if self._health == 0:
21             # print(f"{self._name} is dead.")
22         # print(f"Health of {self._name} updated to {self._health}")
23
24 # Class for Player 1
25 class Player1(Unit):
26     units = ("villager", "knight", "archer")
27
28     def __init__(self, resource):
29         self._units = []
30         self.def_health = 100
31         self._resource = max(resource, 0)
32         for name in self.units:
33             self._units.append(Unit(name, self.def_health))
34
35     def get_resource(self):
36         if self._resource >= 0:
37             return self._resource
38         # else:
39         #     print(f"{self.__class__.__name__} is out of resources.")
40
41     def set_resource(self, rval):
42         self._resource = max(rval, 0)
43         # if self._resource == 0:
44         #     print(f"{self.__class__.__name__} is out of resources.")
45
46     def get_health(self, name):
47         if name == "villager":
48             return self._units[0].get_health()
49         elif name == "knight":
50             return self._units[1].get_health()
51         else:
52             return self._units[2].get_health()
53
54     def set_health(self, name, hval):
55         if name == "villager":
56             return self._units[0].set_health(hval)
57         elif name == "knight":
58             return self._units[1].set_health(hval)
59         else:
60             return self._units[2].set_health(hval)
61
62 # Class for Player 2
63 class Player2(Unit):
64     units = ("tower1", "tower2")
65
66     def __init__(self):
67         self._units = []
68         self.def_health = 100

```

```

69         for name in self.units:
70             self._units.append(Unit(name, self.def_health))
71
72     def get_health(self, name):
73         if name == "tower1":
74             return self._units[0].get_health()
75         else: # tower2
76             return self._units[1].get_health()
77
78     def set_health(self, name, hval):
79         if name == "tower1":
80             return self._units[0].set_health(hval)
81         else: # tower2
82             return self._units[1].set_health(hval)
83
84 # Create the players
85 # P1 includes 1 villager, 1 archer, 1 knight with 100 resources to begin with.
86 # P2 includes 2 watch towers.
87 # Each unit begins with 100 health.
88 p1 = Player1(100)
89 p2 = Player2()

```

```

1 # Auxillary functions for use in Actions
2
3 # Get the neighbor regions of a given region
4 def get_neighbors(region):
5     switch = {
6         1: (2,),
7         2: (1, 15,),
8         7: (8,),
9         8: (7, 13, 14, 15,),
10        13: (8, 11, 14, 15,),
11        14: (8, 11, 13, 15,),
12        10: (11,),
13        11: (10, 13, 14, 15, 16,),
14        15: (8, 11, 13, 14, 16,),
15        16: (11, 15,),
16    }
17    return switch.get(region, 2)
18
19 # Convert health of player units from absolute values to discrete steps
20 # Continuous: 0 to 100; allowable in steps of 5, 10
21 # Discrete: 3 step intervals; [0 to 34) = low; [34, 68) = med; [68, 100] = high
22
23 def get_discrete_health(h_val):
24     if h_val not in range(0, 101):
25         if h_val > 100:
26             return "high"
27         else: # h_val < 0
28             return "low"
29

```

```

30     else:
31         if 0 <= h_val < 34:
32             return "low"
33         elif 34 <= h_val < 68:
34             return "med"
35         else: # 68 <= h_val <= 100
36             return "high"
37
38 # Update health of the unit during a battle scenario
39 # Step 1: Check if the unit launching the attack is in tower inner/outer region
40 # Step 2: Select the hitpoint according to the region and the source unit
41 # Step 3: Decrement the health for target unit as per the hitpoint
42 # Step 4: Return the discretized health to update state for the target unit
43 def update_health(**kwargs):
44
45     region = kwargs.get("region")
46     if region == "tower1_inner" or region == "tower2_inner":
47         hit_step = HIGH_HIT_STEP
48     elif region == "tower1_outer" or region == "tower2_outer":
49         hit_step = LOW_HIT_STEP
50     else: # any other arbitrary region
51         hit_step = LOW_HIT_STEP
52
53     source = kwargs.get("source")
54     if source == "knight": # knight kills faster than other units
55         hitpoint = hit_step * 2
56     else:
57         hitpoint = hit_step
58
59     target = kwargs.get("target")
60     if target == "archer":
61         p1.set_health("archer", p1.get_health("archer") - hitpoint)
62         health_bin = get_discrete_health(p1.get_health("archer"))
63     elif target == "knight":
64         p1.set_health("knight", p1.get_health("knight") - hitpoint)
65         health_bin = get_discrete_health(p1.get_health("knight"))
66     elif target == "villager":
67         p1.set_health("villager", p1.get_health("villager") - hitpoint)
68         health_bin = get_discrete_health(p1.get_health("villager"))
69     elif target == "tower1":
70         p1.set_health("tower1", p1.get_health("tower1") - hitpoint)
71         health_bin = get_discrete_health(p1.get_health("tower1"))
72     elif target == "tower2":
73         p1.set_health("tower2", p1.get_health("tower2") - hitpoint)
74         health_bin = get_discrete_health(p1.get_health("tower2"))
75     else:
76         raise ValueError("Unknown target unit.")
77
78     return health_bin
79
80
81 # Convert resources collected by P1 from absolute values to discrete steps

```

```

82 # Continous: 0 to 600 in steps of 10
83 # Discrete: (resource in P1 region, resource in P2 region) --> 0 1 2 3
84
85 def update_resource(res_val, v_pos):
86     visited = set()
87
88     if res_val <= 100 and v_pos not in visited:
89         res_state = 0 # only default resources, none collected
90     elif 100 < res_val <= 250 and v_pos in P1_STONE:
91         res_state = 2 # only collected from P1 region
92         visited.add(P1_STONE)
93     elif 500 <= res_val <= 600 and v_pos in P2_STONE:
94         res_state = 1 # only collected from P2 region
95         visited.add(P2_STONE)
96     elif 600 < res_val <= 750:
97         res_state = 3 # collected from both P1 and P2 region
98     else:
99         res_state = 0
100     return res_state

```

```

1 # ACTIONS
2 # -----
3
4 # Define the Actions for the players
5 # act(state) -> new-state
6
7 def archer_attack(u, **kwargs):
8
9     # Preconditions
10    # attack is accessible to P1 only in the tower/relic region
11    if u.turn == 2 or u.type in (REGIONTYPE_OBST or REGIONTYPE_STONE \
12                                or REGIONTYPE_FREE):
13        return None
14
15    # Apply action to arena state
16    archer_loc = u.p1_loc[2]
17    archer_health = u.p1_health[2]
18    knight_health = u.p1_health[1]
19    vill_health = u.p1_health[0]
20    p1res = u.p1_res
21    t1health = u.p2_health[0]
22    t2health = u.p2_health[1]
23
24    if u.turn == 1:
25        # Scenario: archer attacking towers
26        # handle health update for tower 1 due to archer
27        if archer_loc in TOWER1:
28            if archer_loc in TOWER_INNER:
29                t1health_10 = update_health(source = "archer",\
30                                             target = "tower1", region = "tower1_inner")
31                v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\

```

```

32                                     (t1health_10, t2health), 2)
33     else: # archer_loc in TOWER_OUTER:
34         t1health_5 = update_health(source = "archer",\
35                                     target = "tower1",\
36                                     region = "tower1_outer")
37         v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
38                           (t1health_5, t2health), 2)
39
40 # handle health update for tower 2 due to archer
41 elif archer_loc in TOWER2:
42     if archer_loc in TOWER_INNER:
43         t2health_10 = update_health(source = "archer",\
44                                     target = "tower2",\
45                                     region = "tower2_inner")
46         v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
47                           (t1health, t2health_10), 2)
48     else: # archer_loc in TOWER_OUTER:
49         t2health_5 = update_health(source = "archer",\
50                                     target = "tower2",\
51                                     region = "tower2_outer")
52         v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
53                           (t1health, t2health_5), 2)
54
55 else:
56     v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
57                       (u.p2_health), 2)
58
59
60 else: # u.turn == 2:
61     # Scenario: towers attacking archer
62     # handle health update for archer due to tower 1
63     if archer_loc in TOWER1:
64         if archer_loc in TOWER_INNER:
65             a_health_10 = update_health(source = "tower1",\
66                                           target = "archer",\
67                                           region = "tower1_inner")
68             v_arena_state = ((u.p1_loc, (a_health_10, knight_health,\
69                                           vill_health), u.p1_res),\
70                               (u.p2_health), 1)
71         else: # archer_loc in TOWER_OUTER:
72             a_health_5 = update_health(source = "tower1",\
73                                         target = "archer",\
74                                         region = "tower1_outer")
75             v_arena_state = ((u.p1_loc, (a_health_5, knight_health,\
76                                           vill_health), u.p1_res),\
77                               (u.p2_health), 1)
78
79     # handle health update for archer due to tower 2
80     elif archer_loc in TOWER2:
81         if archer_loc in TOWER_INNER:
82             a_health_10 = update_health(source = "tower2",\
83                                           target = "archer",\
84                                           region = "tower2_inner")
85             v_arena_state = ((u.p1_loc, (a_health_10, knight_health,\
86                                           vill_health), u.p1_res),\
87                               (u.p2_health), 1)
88         else: # archer_loc in TOWER_OUTER:
89             a_health_5 = update_health(source = "tower2",\
90                                         target = "archer",\
91                                         region = "tower2_outer")
92             v_arena_state = ((u.p1_loc, (a_health_5, knight_health,\
93                                           vill_health), u.p1_res),\
94                               (u.p2_health), 1)
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
8
```

```

83         target = "archer", \
84         region = "tower2_inner")
85     v_arena_state = ((u.p1_loc, (a_health_10, knight_health, \
86         vill_health), u.p1_res), \
87         (u.p2_health), 1)
88     else: # archer_loc in TOWER_OUTER:
89         a_health_5 = update_health(source = "tower2", \
90         target = "archer", \
91         region = "tower2_outer")
92     v_arena_state = ((u.p1_loc, (a_health_5, knight_health, \
93         vill_health), u.p1_res), \
94         (u.p2_health), 1)
95
96     else:
97         v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res), \
98         (u.p2_health), 1)
99
100     # Apply automaton transition
101     v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])
102
103     # Apply igrph transition
104     v_igrph_state = igrph_trans(u.igrph_state, "archer_attack", u.p1_res)
105
106     # Postconditions
107     try:
108         v = hmap[v_arena_state + (v_aut_state, ) + (v_igrph_state, )]
109     except KeyError:
110         return None
111
112     return v if v.type != REGIONTYPE_OBST else None
113
114
115 def knight_attack(u, **kwargs):
116
117     # Preconditions
118     # attack is accessible to P1 only in the tower/relic region
119     if u.turn == 2 or u.type in (REGIONTYPE_OBST or REGIONTYPE_STONE \
120         or REGIONTYPE_FREE):
121         return None
122
123     # Apply action to arena state
124     knight_loc = u.p1_loc[1]
125     archer_health = u.p1_health[2]
126     knight_health = u.p1_health[1]
127     vill_health = u.p1_health[0]
128     p1res = u.p1_res
129     t1health = u.p2_health[0]
130     t2health = u.p2_health[1]
131
132     if u.turn == 1:
133         # Scenario: knight attacking towers
134         # handle health update for tower 1 due to knight

```

```

135     if knight_loc in TOWER1:
136         if knight_loc in TOWER_INNER:
137             t1health_10 = update_health(source = "knight",\
138                                         target = "tower1",\
139                                         region = "tower1_inner")
140             v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
141                             (t1health_10, t2health), 2)
142         else: # knight_loc in TOWER_OUTER:
143             t1health_5 = update_health(source = "knight",\
144                                         target = "tower1",\
145                                         region = "tower1_outer")
146             v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
147                             (t1health_5, t2health), 2)
148
149     # handle health update for tower 2 due to knight
150     elif knight_loc in TOWER2:
151         if knight_loc in TOWER_INNER:
152             t2health_10 = update_health(source = "knight",\
153                                         target = "tower2",\
154                                         region = "tower2_inner")
155             v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
156                             (t1health, t2health_10), 2)
157         else: # knight_loc in TOWER_OUTER:
158             t2health_5 = update_health(source = "knight",\
159                                         target = "tower2",\
160                                         region = "tower2_outer")
161             v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
162                             (t1health, t2health_5), 2)
163
164     else:
165         v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
166                         (u.p2_health), 2)
167
168 else: # u.turn == 2:
169     # Scenario: towers attacking knight
170     # handle health update for knight due to tower 1
171     if knight_loc in TOWER1:
172         if knight_loc in TOWER_INNER:
173             k_health_10 = update_health(source = "tower1",\
174                                         target = "knight",\
175                                         region = "tower1_inner")
176             v_arena_state = ((u.p1_loc, (archer_health, k_health_10,\
177                                         vill_health), u.p1_res),\
178                             (u.p2_health), 1)
179         else: # knight_loc in TOWER_OUTER:
180             k_health_5 = update_health(source = "tower1",\
181                                         target = "knight",\
182                                         region = "tower1_outer")
183             v_arena_state = ((u.p1_loc, (archer_health, k_health_5,\
184                                         vill_health), u.p1_res),\
185                             (u.p2_health), 1)
186

```

```

187         # handle health update for knight due to tower 2
188         elif knight_loc in TOWER2:
189             if knight_loc in TOWER_INNER:
190                 k_health_10 = update_health(source = "tower2",\
191                                             target = "knight",\
192                                             region = "tower2_inner")
193                 v_arena_state = ((u.p1_loc, (archer_health, k_health_10,\
194                                             vill_health), u.p1_res),\
195                                 (u.p2_health), 1)
196             else: # knight_loc in TOWER_OUTER:
197                 k_health_5 = update_health(source = "tower2",\
198                                             target = "knight",\
199                                             region = "tower2_outer")
200                 v_arena_state = ((u.p1_loc, (archer_health, k_health_5,\
201                                             vill_health), u.p1_res),\
202                                 (u.p2_health), 1)
203
204         else:
205             v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
206                             (u.p2_health), 1)
207
208         # Apply automaton transition
209         v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])
210
211         # Apply igrph transition
212         v_igrph_state = igrph_trans(u.igrph_state, "knight_attack", u.p1_res)
213
214         # Postconditions
215         try:
216             v = hmap[v_arena_state + (v_aut_state, ) + (v_igrph_state, )]
217         except KeyError:
218             return None
219
220         return v if v.type != REGIONTYPE_OBST else None
221
222
223 def vill_attack(u, **kwargs):
224
225     # Preconditions
226     # attack is accessible to P1 only in the tower/relic region
227     if u.turn == 2 or u.type in (REGIONTYPE_OBST or REGIONTYPE_STONE\
228                                 or REGIONTYPE_FREE):
229         return None
230
231     # print("\t\tN: Precondition ok.")
232
233     # Apply action to arena state
234     vill_loc = u.p1_loc[0]
235     archer_health = u.p1_health[2]
236     knight_health = u.p1_health[1]
237     vill_health = u.p1_health[0]

```



```

238 p1res = u.p1_res
239 t1health = u.p2_health[0]
240 t2health = u.p2_health[1]
241
242 # u.turn == 2:
243 if u.turn == 2:
244     # Scenario: towers attacking villager
245     # handle health update for villager due to tower 1
246     if vill_loc in TOWER1:
247         if vill_loc in TOWER_INNER:
248             v_health_10 = update_health(source = "tower1",\
249                                         target = "villager",\
250                                         region = "tower1_inner")
251             v_arena_state = ((u.p1_loc, (archer_health, knight_health,\
252                                         v_health_10), u.p1_res),\
253                             (u.p2_health), 1)
254         else: # vill_loc in TOWER_OUTER:
255             v_health_5 = update_health(source = "tower1",\
256                                       target = "villager",\
257                                       region = "tower1_outer")
258             v_arena_state = ((u.p1_loc, (archer_health, knight_health,\
259                                       v_health_5), u.p1_res),\
260                             (u.p2_health), 1)
261
262     # handle health update for villager due to tower 2
263     elif vill_loc in TOWER2:
264         if vill_loc in TOWER_INNER:
265             v_health_10 = update_health(source = "tower2",\
266                                         target = "villager",\
267                                         region = "tower2_inner")
268             v_arena_state = ((u.p1_loc, (archer_health, knight_health,\
269                                         v_health_10), u.p1_res),\
270                             (u.p2_health), 1)
271         else: # vill_loc in TOWER_OUTER:
272             v_health_5 = update_health(source = "tower2",\
273                                       target = "villager",\
274                                       region = "tower2_outer")
275             v_arena_state = ((u.p1_loc, (archer_health, knight_health,\
276                                       v_health_5), u.p1_res),\
277                             (u.p2_health), 1)
278
279     else:
280         v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
281                         (u.p2_health), 1)
282
283 else: # u.turn == 1
284     v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res), (u.p2_health), 2)
285
286
287 # Apply automaton transition
288 v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])
289

```

```

290 # Apply igrph transition
291 v_igrph_state = igrph_trans(u.igrph_state, "vill_attack", u.p1_res)
292
293 # Postconditions
294 try:
295     v = hmap[v_arena_state + (v_aut_state, ) + (v_igrph_state, )]
296 except KeyError:
297     return None
298
299 return v if v.type != REGIONTYPE_OBST else None
300
301
302 def archer_move(u, **kwargs):
303
304     # Precondition 1: move is accessible to P1 archer in FREE/TOWER/RELIC region
305     if u.turn == 2 or u.type in REGIONTYPE_OBST:
306         return None
307
308     # Precondition 2: to perform move, check if the target belongs to neighbors
309     # of the current region i.e. target is reachable from current region
310     target = kwargs.get("target", -1)
311     archer_loc = u.p1_loc[2]
312     neighbors = get_neighbors(archer_loc)
313     if target not in neighbors:
314         return None
315
316     # Apply action
317     # Step 1: move the unit to the target
318     u.p1_loc = (target, u.p1_loc[1], u.p1_loc[0])
319     v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res), (u.p2_health), 2)
320
321     # Apply automaton transition
322     v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])
323
324     # Apply igrph transition
325     v_igrph_state = igrph_trans(u.igrph_state, "archer_move", u.p1_res)
326
327     # Postconditions
328     try:
329         v = hmap[v_arena_state + (v_aut_state, ) + (v_igrph_state, )]
330     except KeyError:
331         return None
332
333     # Postconditions
334     return v if v.type != REGIONTYPE_OBST else None
335
336
337 def knight_move(u, **kwargs):
338
339     # Precondition 1: move is accessible to P1 knight in FREE/TOWER/RELIC region
340     if u.turn == 2 or u.type in REGIONTYPE_OBST:
341         return None

```

```

341         return None
342
343     # Precondition 2: to perform move, check if the target belongs to neighbors
344     # of the current region i.e. target is reachable from current region
345     target = kwargs.get("target", -1)
346     knight_loc = u.p1_loc[1]
347     neighbors = get_neighbors(knight_loc)
348     if target not in neighbors:
349         return None
350
351     # Apply action
352     # Step 1: move the unit to the target
353     u_p1_loc = (u.p1_loc[2], target, u.p1_loc[0])
354     v_arena_state = ((u_p1_loc, u.p1_health, u.p1_res), (u.p2_health), 2)
355
356     # Apply automaton transition
357     v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])
358
359     # Apply igrph transition
360     v_igrph_state = igrph_trans(u.igrph_state, "knight_move", u.p1_res)
361
362     # Postconditions
363     try:
364         v = hmap[v_arena_state + (v_aut_state, ) + (v_igrph_state, )]
365     except KeyError:
366         return None
367
368     # Postconditions
369     return v if v.type != REGIONTYPE_OBST else None
370
371
372 def vill_move(u, **kwargs):
373
374     # Precondition 1: move is accessible to P1 knight in FREE/TOWER/RELIC region
375     if u.turn == 2 or u.type in REGIONTYPE_OBST:
376         return None
377
378     # Precondition 2: to perform move, check if the target belongs to neighbors
379     # of the current region i.e. target is reachable from current region
380     target = kwargs.get("target", -1)
381     vill_loc = u.p1_loc[0]
382     neighbors = get_neighbors(vill_loc)
383     if target not in neighbors:
384         return None
385
386     # Apply action
387     # Step 1: move the unit to the target
388     u_p1_loc = (u.p1_loc[2], u.p1_loc[1], target)
389     v_arena_state = ((u_p1_loc, u.p1_health, u.p1_res), (u.p2_health), 2)
390
391     # Apply automaton transition
392     v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])

```

```

393
394 # Apply igrph transition
395 v_igrph_state = igrph_trans(u.igrph_state, "vill_move", u.p1_res)
396
397 # Postconditions
398 try:
399     v = hmap[v_arena_state + (v_aut_state, ) + (v_igrph_state, )]
400 except KeyError:
401     return None
402
403 # Postconditions
404 return v if v.type != REGIONTYPE_OBST else None
405
406
407 def archer_suicide(u, **kwargs):
408
409     # Precondition 1: suicide is accessible to P1 archer only in
410     # FREE/TOWER/RELIC/STONE region
411     if u.turn == 2 or u.type in REGIONTYPE_OBST :
412         return None
413
414     # Apply action
415
416     # return the absolute health value of archer bw 0-100
417     a_health = p1.get_health("archer")
418     # set health of archer to 0
419     p1.set_health("archer", 0)
420     archer_health = get_discrete_health(a_health)
421
422     # discretize the health in steps
423     # Boost health of remaining friendly units
424     # Step 1: Get the absolute health value (bw 0-100) of the archer unit
425     # Step 2: Distribute archers' health equally to remaining friendly units
426     # Step 3: Scale absolute health of friendly units back to discrete steps
427     k_health = p1.get_health("knight")
428     k_health = max(k_health + a_health*0.5, 100)
429     knight_health = get_discrete_health(k_health)
430
431     v_health = p1.get_health("villager")
432     v_health = max(v_health + a_health*0.5, 100)
433     vill_health = get_discrete_health(v_health)
434
435     # Construct the state
436     v_arena_state = ((u.p1_loc, (archer_health, knight_health, vill_health),\
437                      u.p1_res), (u.p2_health), 2)
438
439     # Apply automaton transition
440     v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])
441
442     # Apply igrph transition
443     v_igrph_state = igrph_trans(u.igrph_state, "archer_suicide", u.p1_res)
444

```

```

445     # Postconditions
446     try:
447         v = hmap[v_arena_state + (v_aut_state, ) + (v_igraph_state, )]
448     except KeyError:
449         return None
450
451     # Postconditions
452     if v.type not in REGIONTYPE_OBST:
453         return v
454
455
456 def vill_build_castle(u, **kwargs):
457
458     # Precondition 1: build is accessible to P1 villager only in
459     # FREE/TOWER/RELIC region
460     if u.turn == 2 or u.type in (REGIONTYPE_OBST, REGIONTYPE_STONE):
461         return None
462
463     # Precondition 2: build castle requires 500 stones or resources
464     if u.p1_res != 3:
465         # raise ValueError("Insufficient resources.")
466         return None
467
468     # Apply action
469     # Step 1: update the resources for castle build action
470     res = p1.get_resource()
471     if res < COST_CASTLE:
472         return None
473     else: # res >= COST_CASTLE
474         res -= COST_CASTLE
475         p1.set_resource(res)
476
477     # Construct the state
478     v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res), (u.p2_health), 2)
479
480     # Apply automaton transition
481     v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])
482
483     # Apply igraph transition
484     v_igraph_state = igraph_trans(u.igraph_state, "vill_build_castle", u.p1_res)
485
486     # Postconditions
487     try:
488         v = hmap[v_arena_state + (v_aut_state, ) + (v_igraph_state, )]
489     except KeyError:
490         return None
491
492     # Postconditions
493     if v.type not in REGIONTYPE_OBST:
494         return v
495

```

```

496
497 def vill_collect(u, **kwargs):
498
499     # Precondition 1: collect stone is accessible to P1 villager only in
500     # STONE region
501     if u.turn == 2 or u.type not in REGIONTYPE_STONE:
502         return None
503
504     # Apply action
505     # Step 1: update the resources due to collect action
506     res = p1.get_resource()
507     res += FIXED_RESOURCE_STEP
508     p1.set_resource(res)
509
510     # Step 2: update resources collected to player state from villager position
511     pos_v = u.p1_loc[0]
512     p1_res = update_resource(p1.get_resource(), pos_v)
513     v_arena_state = ((u.p1_loc, u.p1_health, p1_res), (u.p2_health), 2)
514
515     # Apply automaton transition
516     v_aut_state = aut_trans(u.aut_state, u.p1_loc)
517
518     # Apply igrph transition
519     v_igrph_state = igrph_trans(u.igrph_state, "vill_collect", p1_res)
520
521     # Postcondition
522     try:
523         v = hmap[v_arena_state + (v_aut_state, ) + (v_igrph_state, )]
524     except KeyError:
525         return None
526
527     return v if v.type != REGIONTYPE_OBST else None
528
529
530 def nop(u, **kwargs):
531
532     # Apply action
533     if u.turn == 1:
534         v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res), (u.p2_health), 2)
535     else: # u.turn == 2
536         v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res), (u.p2_health), 1)
537
538     # Apply automaton transition
539     v_aut_state = aut_trans(u.aut_state, u.p1_loc)
540
541     # Apply igrph transition
542     v_igrph_state = igrph_trans(u.igrph_state, "nop", u.p1_res)
543
544     # Postcondition
545     try:
546         v = hmap[v_arena_state + (v_aut_state, ) + (v_igrph_state, )]
547     except KevError:

```

```

548         return None
549
550     return v if v.type != REGIONTYPE_OBST else None
551
552 # P2's perceived action set of P1's at different states of IGraph
553 ACT_P1_ISTATE_0 = (archer_move, knight_move, vill_move, nop)
554 ACT_P1_ISTATE_1 = (archer_move, knight_move, vill_move, nop, archer_suicide,\
555                    archer_attack, knight_attack, vill_attack)
556 ACT_P1_ISTATE_2 = (archer_move, knight_move, vill_move, nop, vill_collect,\
557                    archer_attack, knight_attack, vill_attack)
558 ACT_P1_ISTATE_3 = (archer_move, knight_move, vill_move, nop, archer_suicide,\
559                    vill_collect, archer_attack, knight_attack, vill_attack)
560 ACT_P1_ISTATE_4 = (archer_move, knight_move, vill_move, nop, vill_collect,\
561                    archer_attack, knight_attack, vill_attack)
562 ACT_P1_ISTATE_5 = (archer_move, knight_move, vill_move, nop, archer_suicide,\
563                    vill_collect, archer_attack, knight_attack, vill_attack)
564 ACT_P1_ISTATE_6 = (archer_move, knight_move, vill_move, nop, vill_collect,\
565                    vill_build_castle, archer_attack, knight_attack, vill_attack)
566 ACT_P1_ISTATE_7 = (archer_move, knight_move, vill_move, nop, archer_suicide,\
567                    vill_collect, vill_build_castle, archer_attack,\
568                    knight_attack, vill_attack)
569
570 # P2's action set (known accurately by P1 and P2)
571 ACT_P2 = (archer_attack, knight_attack, vill_attack, nop)

```

```

1 # Construct Hypergame Edge
2 # -----
3
4 # HELPER FUNCTIONS: NEIGHBORHOOD
5
6 # Tuple of move actions
7 MOVE_ACTIONS = (archer_move, knight_move, vill_move)
8
9 def apply_actions_p1(u, hmap):
10     V_u = []
11     for act in ACT_P1_ISTATE_7:
12         for region in REGIONS:
13             # Iterate through each available region and set target equal to this region
14             # only for the move actions
15             if act in MOVE_ACTIONS:
16                 v = act(u, target = region)
17             else:
18                 v = act(u)
19             if v is not None:
20                 V_u.append((v, act))
21
22     return tuple(V_u)
23
24
25 def apply_actions_p2(u, hmap):
26     V_u = []

```

```

27     for act in ACT_P2:
28         v = act(u)
29         if v is not None:
30             V_u.append((v, act))
31
32     return tuple(V_u)
33
34
35 def neighbors(u, hmap):
36
37     if u.turn == 1:
38         return apply_actions_p1(u, hmap)
39     else:
40         return apply_actions_p2(u, hmap)
41
42
43 # ADD EDGES
44 # Set initial states
45 v0 = hmap[v0_tuple]
46 print(v0, v0.type)
47 print("-----")
48
49 # Construct the edges of hypergame
50
51 hedges = set()
52 stack = [v0]
53 visited = set()
54
55 while len(stack) > 0:
56     u = stack.pop()
57     visited.add(u)
58
59     succ = neighbors(u, hmap)
60     for v, act in succ:
61         hedges.add((u, v, act))
62         if v not in visited:
63             stack.append(v)
64
65 print(f"Found {len(hedges)} edges.")

```

```

↳ HState(s: (((2, 2, 2), ('high', 'high', 'high'), 1), ('high', 'high'), 1), q: 0, i: 0) f
-----
Found 721477 edges.

```

```

1 # ELIMINATE UNREACHABLE STATES
2 # -----
3 # As we have computed edges based on initial state,
4 # we use edges to eliminate unreachable states.
5
6 # Prune the hstates based on reachability
7 reachable = set()

```



```

8 for u, v, _ in hedges:
9     reachable.add(u)
10    reachable.add(v)
11
12 prune = set()
13 for v_tuple in hmap.keys():
14     if hmap[v_tuple] not in reachable:
15         prune.add(v_tuple)
16
17 for v_tuple in prune:
18     hmap.pop(v_tuple)
19
20 print(f"len(hmap)={len(hmap)}, len(hedges)={len(hedges)}")

```

↳ len(hmap)=98529, len(hedges)=721477

```

1 # PROJECTION OF HYPERGAME ONTO GAME
2 # -----
3
4 # Construct a simple game with perfect information
5 # Compute projection of hmap onto gmap
6
7 gstates = set()
8 for hstate in hmap.values():
9     gstates.add(hstate.arena_state + (hstate.aut_state, ))
10
11 gedges = set()
12 for hedge in hedges:
13     u_h, v_h, a = hedge
14     u_g = u_h.arena_state + (u_h.aut_state, )
15     v_g = v_h.arena_state + (v_h.aut_state, )
16     gedges.add((u_g, v_g, a))
17
18 print(f"len(gstates)={len(gstates)}, len(gedges)={len(gedges)}")
19
20 # Formulate a game object
21 game = Game()
22
23 gmap = dict()
24 for st in gstates:
25     gv = GameVertex(name=st, turn=st[2])
26     gmap[st] = gv
27     game.add_vertex(gv)
28
29 for u, v, a in gedges:
30     game.add_edge(GameEdge(u=gmap[u], v=gmap[v], act=a))
31
32 print(game)

```

↳ len(gstates)=88608, len(gedges)=646835
Game(|V|=88608, |E|=646835)

▼ Perfect Information Game Construction and Solution

```
1 # SURE-WINNING SOLUTION
2 # -----
3
4 # Mark final states
5 for v in game.vertices:
6     if v.name[-1] == 2: # add final aut_state here
7         game.mark_final(v)
8
9
10 # Invoke solver
11 sw_solver = SureWinning(game=game)
12 sw_solver.solve()
13 print(f"sw_solver.p1_win={len(sw_solver.p1_win)} of which {len(game.final)}\
14                                     are final states.")
15 # print(f"sw_solver.p2_win={len(sw_solver.p2_win)}")
```

➞ sw_solver.p1_win=43968 of which 30720 are final states.

▼ Hypergame Construction and Solution

```
1 # HYPERGAME OBJECT CONSTRUCTION
2 # -----
3
4
5 hypergame = Hypergame()
6
7 # Add vertices
8 for v in hmap.values():
9     hypergame.add_vertex(HypergameVertex(hstate=v))
10
11 # Add edges
12 for u, v, a in hedges:
13     hypergame.add_edge(HypergameEdge(u=HypergameVertex(hstate=u),\
14                                     v=HypergameVertex(hstate=v), act=a))
15
16 # Mark final states
17 final = set()
18 p1_win = {u.name for u in sw_solver.p1_win}
19 for v in hypergame.vertices:
20     if v.game_vertex in p1_win:
21         if v.hstate.aut_state == 2:
22             final.add(v)
23             hypergame.mark_final(v)
24
25 print(f"len(final)={len(final)}")
26 print(f"-----")
```

```

26 print(
27
28 # Statistics of hypergame construction
29 print(hypergame)
30 print(f"Num(P1-states)={len({v for v in hypergame.vertices\
31                               if v.hstate.turn == 1})}")
32 print(f"Num(P2-states)={len({v for v in hypergame.vertices\
33                               if v.hstate.turn == 2})}")
34 print(f"Num(aut_state=0)={len({v for v in hypergame.vertices\
35                               if v.hstate.aut_state == 0})}")
36 print(f"Num(aut_state=1)={len({v for v in hypergame.vertices\
37                               if v.hstate.aut_state == 1})}")
38 print(f"Num(aut_state=2)={len({v for v in hypergame.vertices\
39                               if v.hstate.aut_state == 2})}")
40 print(f"Num(igraph_state=0)={len({v for v in hypergame.vertices\
41                                   if v.hstate.igraph_state == 0})}")
42 print(f"Num(igraph_state=1)={len({v for v in hypergame.vertices\
43                                   if v.hstate.igraph_state == 1})}")
44 print(f"Num(igraph_state=2)={len({v for v in hypergame.vertices\
45                                   if v.hstate.igraph_state == 2})}")
46 print(f"Num(igraph_state=3)={len({v for v in hypergame.vertices\
47                                   if v.hstate.igraph_state == 3})}")
48 print(f"Num(igraph_state=4)={len({v for v in hypergame.vertices\
49                                   if v.hstate.igraph_state == 4})}")
50 print(f"Num(igraph_state=5)={len({v for v in hypergame.vertices\
51                                   if v.hstate.igraph_state == 5})}")
52 print(f"Num(igraph_state=6)={len({v for v in hypergame.vertices\
53                                   if v.hstate.igraph_state == 6})}")
54 print(f"Num(igraph_state=7)={len({v for v in hypergame.vertices\
55                                   if v.hstate.igraph_state == 7})}")
56 print(f"Num(archer_attack, knight_attack, vill_attack-edges)=\
57 {len({e for e in hypergame.edges if e.act in (archer_attack, knight_attack,\
58                                                vill_attack})})}")
59 print(f"Num(archer_move, knight_move, vill_move-edges)=\
60 {len({e for e in hypergame.edges if e.act in (archer_move, knight_move,\
61                                                vill_move})})}")
62 print(f"Num(archer_suicide-edges)=\
63       {len({e for e in hypergame.edges if e.act in (archer_suicide, )})}")
64 print(f"Num(vill_build_castle-edges)=\
65       {len({e for e in hypergame.edges if e.act in (vill_build_castle, )})}")
66 print(f"Num(vill_collect-edges)=\
67       {len({e for e in hypergame.edges if e.act in (vill_collect, )})}")
68 print(f"Num(nop-edges)={len({e for e in hypergame.edges if e.act in (nop, )})}")

```



```

len(final)=30720
-----
Hypergame(|V|=98529, |E|=721477)
Num(P1-states)=47425
Num(P2-states)=51104
Num(aut_state=0)=37409
Num(aut_state=1)=30400
Num(aut_state=2)=30720
Num(igraph_state=0)=1
Num(igraph_state=1)=0
Num(igraph_state=2)=83648
Num(igraph_state=3)=0
Num(igraph_state=4)=0
Num(igraph_state=5)=0

```

```

1 # DECEPTIVE ALMOST-SURE WINNING REGION
2 # -----
3
4 # Define a function to mark whether a hypergame edge is permissive or not.
5
6 def is_permissive(e):
7     turn = e.source.hstate.turn
8     if turn == 2:
9         return True
10
11     igrph_state = e.source.hstate.igraph_state
12     act = e.act
13
14     if igrph_state == 0:
15         res = act in ACT_P1_ISTATE_0
16
17     elif igrph_state == 1:
18         res = act in ACT_P1_ISTATE_1
19
20     elif igrph_state == 2:
21         res = act in ACT_P1_ISTATE_2
22
23     elif igrph_state == 3:
24         res = act in ACT_P1_ISTATE_3
25
26     elif igrph_state == 4:
27         res = act in ACT_P1_ISTATE_4
28
29     elif igrph_state == 5:
30         res = act in ACT_P1_ISTATE_5
31
32     elif igrph_state == 6:
33         res = act in ACT_P1_ISTATE_6
34
35     elif igrph_state == 7:
36         res = act in ACT_P1_ISTATE_7
37
38
39 # Invoke DASW Solver

```

```

40 dasw = DeceptiveAlmostSureWinning(hypergame=hypergame,\
41                                     is_permissive=is_permissive)
42 dasw.solve()
43 print(f"dasw.p1_win={len(dasw.p1_win)}")

```

```

↳ dasw.p1_win=43969

```

▼ Results Summary

```

1 # Problem Setup Parameters
2 print("Problem Setup Parameters")
3
4 print(" ", f"REGIONS          = {REGIONS}")
5 print(" ", f"FREE_REGIONS     = {FREE_REGIONS}")
6 print(" ", f"TOWERS           = {TOWERS}")
7 print(" ", f"TOWER1            = {TOWER1}")
8 print(" ", f"TOWER2            = {TOWER2}")
9 print(" ", f"TOWER_INNER       = {TOWER_INNER}")
10 print(" ", f"TOWER_OUTER      = {TOWER_OUTER}")
11 print(" ", f"STONE_REGIONS    = {STONE_REGIONS}")
12 print(" ", f"RELIC            = {RELIC}")
13 print(" ", f"P1_STONE         = {P1_STONE}")
14 print(" ", f"P2_STONE         = {P2_STONE}")
15 print(" ", f"SAFE            = {SAFE}")
16 print(" ", f"UNSAFE          = {UNSAFE}")
17 print(" ", f"v0              = {v0_tuple}")
18 print()
19 print(" ", f"ACT_P1_ISTATE_0 = {[a.__name__ for a in ACT_P1_ISTATE_0]}")
20 print(" ", f"ACT_P1_ISTATE_1 = {[a.__name__ for a in ACT_P1_ISTATE_1]}")
21 print(" ", f"ACT_P1_ISTATE_2 = {[a.__name__ for a in ACT_P1_ISTATE_2]}")
22 print(" ", f"ACT_P1_ISTATE_3 = {[a.__name__ for a in ACT_P1_ISTATE_3]}")
23 print(" ", f"ACT_P1_ISTATE_4 = {[a.__name__ for a in ACT_P1_ISTATE_4]}")
24 print(" ", f"ACT_P1_ISTATE_5 = {[a.__name__ for a in ACT_P1_ISTATE_5]}")
25 print(" ", f"ACT_P1_ISTATE_6 = {[a.__name__ for a in ACT_P1_ISTATE_6]}")
26 print(" ", f"ACT_P1_ISTATE_7 = {[a.__name__ for a in ACT_P1_ISTATE_7]}")
27 print(" ", f"ACT_P2           = {[a.__name__ for a in ACT_P2]}")
28
29 print()
30 print("Game with Perfect Information")
31 print(" ", f"game: |V|={len(game.vertices)} and |E|={len(game.edges)}")
32 print(" ", f"p1_win={len(sw_solver.p1_win)} of which {len(game.final)} are\
33                                     final states.")
34
35 print()
36 print("Hypergame")
37 print(" ", f"hgame: |V|={len(hypergame.vertices)} and\
38                                     |E|={len(hypergame.edges)}")
39 print(" ", f"DASW={len(dasw.p1_win)} of which {len(hypergame.final)}\
40                                     are final states.")
41 print(" ", f"proj(DASW onto S)=\

```

```

42  {len({v.hstate.arena_state + (v.hstate.aut_state, ) for v in dasw.p1_win}})})

```

Problem Setup Parameters

```

REGIONS          = (1, 2, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16)
FREE_REGIONS     = (1, 2, 7, 8, 10, 11, 13, 14, 15, 16)
TOWERS           = (6, 9)
TOWER1           = (7, 8, 13, 14)
TOWER2           = (10, 11, 13, 14)
TOWER_INNER      = (7, 10)
TOWER_OUTER      = (8, 11, 13, 14)
STONE_REGIONS    = (1, 16)
RELIC             = (14,)
P1_STONE         = (1,)
P2_STONE         = (16,)
SAFE             = (1, 2)
UNSAFE           = (6, 7, 8, 9, 10, 11, 13, 14, 15, 16)
v0               = (((2, 2, 2), ('high', 'high', 'high'), 1), ('high', 'high'), 1, 0, 0

ACT_P1_ISTATE_0 = ['archer_move', 'knight_move', 'vill_move', 'nop']
ACT_P1_ISTATE_1 = ['archer_move', 'knight_move', 'vill_move', 'nop', 'archer_suicide']
ACT_P1_ISTATE_2 = ['archer_move', 'knight_move', 'vill_move', 'nop', 'vill_collect',
ACT_P1_ISTATE_3 = ['archer_move', 'knight_move', 'vill_move', 'nop', 'archer_suicide']
ACT_P1_ISTATE_4 = ['archer_move', 'knight_move', 'vill_move', 'nop', 'vill_collect',
ACT_P1_ISTATE_5 = ['archer_move', 'knight_move', 'vill_move', 'nop', 'archer_suicide']
ACT_P1_ISTATE_6 = ['archer_move', 'knight_move', 'vill_move', 'nop', 'vill_collect',
ACT_P1_ISTATE_7 = ['archer_move', 'knight_move', 'vill_move', 'nop', 'archer_suicide']
ACT_P2          = ['archer_attack', 'knight_attack', 'vill_attack', 'nop']

```

Game with Perfect Information

game: $|V|=88608$ and $|E|=646835$
 p1_win=43968 of which 30720 are final states.

Hypergame

hgame: $|V|=98529$ and $|E|=721477$
 DASW=43969 of which 30720 are final states.
 proj(DASW onto S)=43968

