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

```
from typing import Iterable

# DS1: Graph
class Vertex(object):
    pass


class Edge(object):
        def __init__(self, u: 'Graph.Vertex', v: 'Graph.Vertex'):
            self._source = u
            self._target = v

        @property
        def source(self):
            """ Returns the source vertex of edge. """
            return self._source
```

```python

        @property
        def target(self):
            """ Returns the target vertex of edge. """
            return self._target


class Graph(object):
    def __init__(self):
        # Dict: {vertex: (set(<in-edge>), set(<out-edge>))}
        self._vertex_edge_map = dict()
        self._edges = set()               # Set of all edges of graph

    def __repr__(self):
        return f"{self.__class__.__name__}\
        (|V|={len(self._vertex_edge_map)}, |E|={len(self._edges)})"

    def add_edge(self, e):
        u = e.source
        v = e.target

        self._vertex_edge_map[u][1].add(e)
        self._vertex_edge_map[v][0].add(e)
        self._edges.add(e)

    def add_vertex(self, v):
        self._vertex_edge_map[v] = (set(), set())

    def in_edges(self, v):
        if isinstance(v, Vertex):
            return list(self._vertex_edge_map[v][0])

        elif isinstance(v, Iterable):
            in_edges = (self._vertex_edge_map[u][0] for u in v)
            return list((reduce(set.union, in_edges)))

        raise AssertionError(f"Vertex {v} must \
        be a single or an iterable of {Vertex} objects.")

    def out_edges(self, v):
        if isinstance(v, Vertex):
            return list(self._vertex_edge_map[v][1])

        elif isinstance(v, Iterable):
            out_edges = (self._vertex_edge_map[u][1] for u in v)
            return list(reduce(set.union, out_edges))

        raise AssertionError(f"Vertex {v} must \
        be a single or an iterable of {Vertex} objects.")

    def out_neighbors(self, v):
        if isinstance(v, Vertex):
```

```
            return list(e.target for e in self._vertex_edge_map[v][1])

        elif isinstance(v, Iterable):
            return list(e.target for u in v for e in \
                        self._vertex_edge_map[u][1])

        raise AssertionError(f"Vertex {v} must \
        be a single or an iterable of {Vertex} objects.")

    @property
    def edges(self):
        return list(self._edges)

    @property
    def vertices(self):
        return list(self._vertex_edge_map.keys())

```

```
# DS2: Game

class GameVertex(Vertex):
    def __init__(self, name, turn):
        self._name = name
        self._turn = turn

    def __hash__(self):
        return self.name.__hash__()

    def __eq__(self, other):
        return self.name == other.name and self.turn == other.turn

    @property
    def name(self):
        """ Returns the name of game vertex. """
        return self._name

    @property
    def turn(self):
        """ Returns the id of player who will make move in current state. """
        return self._turn


class GameEdge(Edge):
    def __init__(self, u, v, act):
        super().__init__(u, v)
        self._act = act

    @property
    def act(self):
        """ Returns action associated with game edge. """
        return self._act
```

```python


class Game(Graph):
    def __init__(self):
        super().__init__()
        self._final = set()

    @property
    def final(self):
        """ Returns the set of final states of the game. """
        return self._final

    def mark_final(self, v):
        """
        Adds the given state to the set of final states in the game.

        :param v: (:class:`Game.Vertex`) Vertex to be marked as final.
        """
        if v in self._vertex_edge_map:
            self._final.add(v)

```

```python
# DS3: Hypergame

class HState:
    """Represents Hypergame State"""
    # state = (s1, s2, turn, q, i)
    # s1 = ((pos_v, pos_k, pos_a), (health_v, health_k, health_a), res)
    # s2 = (health_tower1, health_tower2)
    # st = (((pos_v, pos_k, pos_a), (health_v, health_k, health_a), res), \
    #                             (health_tower1, health_tower2), turn, q, i)

    def __init__(self, state):
        self._state = state
        self.type = None

    def __hash__(self):
        return hash(self._state)

    def __eq__(self, other):
        return self._state == other._state

    def __str__(self):
        return f"HState(s: {self.arena_state}, q: {self.aut_state}, \
                                               i: {self.igraph_state})"

    __repr__ = __str__

    @property
    def p1_loc(self):
        return self._state[0][0]
```

```python

    @property
    def p1_health(self):
        return self._state[0][1]

    @property
    def p1_res(self):
        return self._state[0][2]

    @property
    def p2_health(self):
        return self._state[1]

    @property
    def turn(self):
        return self._state[2]

    @property
    def arena_state(self):
        return self._state[0:3]

    @property
    def aut_state(self):
        return self._state[3]

    @property
    def igraph_state(self):
        return self._state[4]


class HypergameVertex(Vertex):

    # ------------------------------------------------------------------------
    # INTERNAL CLASSES
    # ------------------------------------------------------------------------
    def __init__(self, hstate):
        # self._name = f"(game_v={game_v.name}, igraph_v={igraph_v.name})" \
                                            # str((game_v.name, igraph_v.name))
        self._hstate = hstate
        self._name = str(hstate)
        self._game_v = hstate.arena_state + (hstate.aut_state, )
        self._igraph_v = hstate.igraph_state
        self._turn = hstate.turn

    def __repr__(self):
        string = f"Vertex(name={self._name}"
        return string

    def __hash__(self):
        return hash(self._hstate)

    def __eq__(self, other):
```

```python
    def __eq__(self, other):
        if isinstance(other, HypergameVertex):
            return self._hstate == other._hstate
        elif isinstance(other, HState):
            return self._hstate == other

    # ---------------------------------------------------------------------------
    # PUBLIC PROPERTIES
    # ---------------------------------------------------------------------------
    @property
    def name(self):
        """ Returns the name of game vertex. """
        return self._name

    @property
    def turn(self):
        """ Returns the id of player who will make move in current state. """
        return self._turn

    @property
    def game_vertex(self):
        return self._game_v  # pragma: no cover

    @property
    def igraph_vertex(self):
        return self._igraph_v  # pragma: no cover

    @property
    def hstate(self):
        return self._hstate


class HypergameEdge(Edge):
    def __init__(self, u, v, act):
        super().__init__(u, v)
        self._act = act

    @property
    def act(self):
        """ Returns action associated with game edge. """
        return self._act


class Hypergame(Graph):
    def __init__(self):
        super().__init__()
        self._final = set()

    @property
    def final(self):
        """ Returns the set of final states of the game. """
        return self._final
```

```python

    def mark_final(self, v):
        """
        Adds the given state to the set of final states in the game.
        :param v: (:class:`Game.Vertex`) Vertex to be marked as final.
        """
        if v in self._vertex_edge_map:
            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

```python
# Algorithm 1: Sure winning region computation

class SureWinning(object):
    def __init__(self, game):
        self._game = game
        self._p1_win = set()
        self._p2_win = set()

    @property
    def p1_win(self):
        """ Returns the P1's winning region. Returns None, \
                    if the solver has not solved the game. """
        return self._p1_win

    def _pre1(self):

        # Initialize an empty set
        pre1 = set()

        # Iterate over all states in winning region
        for v in self._p1_win:

        # Iterate over all incoming edges to check all \
        # potential states to add to Pre1
            for e in self._game.in_edges(v):

                # Get the candidate vertex to add to Pre1
                u = e.source

                # If u is P1's vertex and not already added, then add it to Pre1
                if u.turn == 1 and u not in self._p1_win:
                    pre1.add(u)

        return pre1

```

```
    def _pre2(self):
        # Initialize an empty set
        pre2 = set()

        # Iterate over all states in winning region
        for v in self._p1_win:

        # Iterate over all incoming edges to check all \
        # potential states to add to Pre1
            for e in self._game.in_edges(v):

                # Get the candidate vertex to add to Pre1
                u = e.source

                # If u is P2's vertex AND not already added AND all outgoing \
                # edges are winning, then add it to Pre1
                if u.turn == 2 and u not in self._p1_win and \
                set(self._game.out_neighbors(u)).issubset(self._p1_win):
                    pre2.add(u)

        return pre2

    def solve(self):
        """ Runs the solver. """
        # Initialize/Reset Solution data structures
        final = set()
        for v in self._game.vertices:
            if v in self._game.final:
                final.add(v)

        # Zielonka's algorithm
        self._p1_win = final
        while True:
            pre1 = self._pre1()
            pre2 = self._pre2()
            p1_win = set.union(self._p1_win, pre1, pre2)

            if p1_win == self._p1_win:
                break

            self._p1_win = p1_win
```

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

class DeceptiveAlmostSureWinning(object):
    def __init__(self, hypergame, is_permissive):
        self._hypergame = hypergame
        # Function which checks whether given edge is permissive or not.
        self.is_permissive = is_permissive
        self._p1_win = set()
        self._p2_win = set()
```

```python

    @property
    def p1_win(self):
        """ Returns the P1's winning region. Returns None, \
            if the solver has not solved the game. """
        return self._p1_win

    def _dapre11(self, Uk):
        """DAPre_1^1"""

        dapre11 = set()

        for u in Uk:
          # If turn of vertex is not P1, inspect next vertex
            if u.turn != 1:
                continue

          # If final, keep it (equivalent of making final states as sink states)
            if u in self.final:
                dapre11.add(u)
                continue

          # Check if any of the neighbors are included in Uk
            out_neighbors = self._hypergame.out_neighbors(u)
            if len(set(out_neighbors).intersection(Uk)) > 0:
                dapre11.add(u)

        return dapre11

    def _dapre12(self, Uk):
        """DAPre_1^2"""
        dapre12 = set()

        for u in Uk:
          # If turn of vertex is not P2, inspect next vertex
            if u.turn != 2:
                continue

          # If final, keep it (equivalent of making final states as sink states)
            if u in self.final:
                dapre12.add(u)
                continue

          # Check if any of the neighbors are included in Uk
            out_edges = self._hypergame.out_edges(u)
            out_neighbors = list()

            for e in out_edges:
                if self.is_permissive(e):
                    out_neighbors.append(e.target)

            if set(out_neighbors).issubset(Uk):
```

```
                dapre12.add(u)

        return dapre12

    def _dapre21(self, Uk):
        """DAPre_2^1"""
        dapre21 = set()

        for u in Uk:
            # If turn of vertex is not P1, inspect next vertex
            if u.turn != 1:
                continue

            # Check if any of the neighbors are included in Uk
            out_neighbors = self._hypergame.out_neighbors(u)
            if set(out_neighbors).issubset(Uk):
                dapre21.add(u)

        return dapre21

    def _dapre22(self, Uk):
        """DAPre_2^2"""
        dapre22 = set()

        for u in Uk:
            # If turn of vertex is not P1, inspect next vertex
            if u.turn != 2:
                continue

            # Check if any of the neighbors are included in Uk
            out_edges = self._hypergame.out_edges(u)
            perm_neighbors = list()
            for e in out_edges:
                if self.is_permissive(e):
                    perm_neighbors.append(e.target)

            if set(perm_neighbors).issubset(Uk):
                dapre22.add(u)


        return dapre22

    def _safe_1(self, Uk):

        # Initialize an empty set
        safe1 = Uk
        # print("\t================")
        # print("\t", f"Safe-1({Uk})")

        while True:
            dapre1 = self._dapre11(safe1)
```

```
            dapre2 = self._dapre12(safe1)
            res = set.intersection(safe1, set.union(dapre1, dapre2))

            # print("\t\t=================")
            # print("\t\t", f"U={safe1}")
            # print("\t\t", f"DAPRE_1^1 = {dapre1}")
            # print("\t\t", f"DAPRE_1^2 = {dapre2}")
            if res == safe1:
                break

            safe1 = res

        # print()
        # print("\t", f"Safe-1 = {safe1}")
        return safe1

    def _safe_2(self, Uk):
        # Initialize an empty set
        safe2 = Uk
        # print("\t=================")
        # print("\t", f"Safe-2({Uk})")

        while True:
            dapre1 = self._dapre21(safe2)
            dapre2 = self._dapre22(safe2)
            res = set.intersection(safe2, set.union(dapre1, dapre2))

            # print("\t\t=================")
            # print("\t\t", f"U={safe2}")
            # print("\t\t", f"DAPRE_2^1 = {dapre1}")
            # print("\t\t", f"DAPRE_2^2 = {dapre2}")

            if res == safe2:
                break

            safe2 = res

        # print()
        # print("\t", f"Safe-2 = {safe2}")
        return safe2

    def solve(self, p1_sw_win=None):
        """ Runs the solver. """
        # Initialize/Reset Solution data structures
        self.final = set(self._hypergame.final)

        # Deceptive almost-sure winning algorithm
        self._p1_win = self.final
        Ck = []
        while True:
            Ck.append(self._safe_2(set.difference(set(self._hypergame.vertices),\
                                                  self._p1_win)))
```

```python
            p1_win = self._safe_1(set.difference(set(self._hypergame.vertices),\
                                                  Ck[-1]))
            if p1_win == self._p1_win:
                break

            self._p1_win = p1_win

```

Game Parameters

```python
# Global Variables
REGIONTYPE_FREE = "free"
REGIONTYPE_TOWER = "tower"
REGIONTYPE_OBST = "obst"
REGIONTYPE_STONE = "stone"
REGIONTYPE_RELIC = "relic"

# Configuration settings for AoE example
REGIONS = (1, 2, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16)
TOWERS = (6, 9)
P1_STONE = (1,)
P2_STONE = (16,)
SAFE = (1, 2)
UNSAFE = tuple(set(REGIONS) - set(SAFE))
RELIC = (14,)
TOWER_INNER = (7, 10)
TOWER_OUTER = (8, 11, 13, 14)
TOWER1 = (7, 8, 13, 14)
TOWER2 = (10, 11, 13, 14)
FREE_REGIONS = tuple(set(REGIONS) - set(TOWERS))
TOWER_REGIONS = tuple(TOWER_INNER + TOWER_OUTER)
STONE_REGIONS = tuple(P1_STONE + P2_STONE)
COST_CASTLE = 500
FIXED_RESOURCE_STEP = 10
FIXED_HEALTH_STEP = 10
LOW_HIT_STEP = 5
HIGH_HIT_STEP = 10

# Initial State in the AoE example
v0_tuple = (((2, 2, 2), ("high", "high", "high"), 1), ("high", "high"), 1, 0, 0)

# print(f"Safe regions = {SAFE}.")
# print(f"Unsafe regions = {UNSAFE}.")
# 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

```
import itertools

# Labelling function for hypergame states with type labels
def hstate_type(hstate):
    p1_position = hstate.p1_loc
    p1_res = hstate.p1_res
    pos_v = p1_position[0]
    pos_k = p1_position[1]
    pos_a = p1_position[2]

    if pos_v in TOWERS or pos_k in TOWERS or pos_a in TOWERS:
        return REGIONTYPE_OBST

    elif pos_v in STONE_REGIONS:
        return REGIONTYPE_STONE

    elif pos_v in RELIC:
        return REGIONTYPE_RELIC

    elif pos_v in TOWER_REGIONS or pos_k in TOWER_REGIONS or pos_a in\
                                                             TOWER_REGIONS:
        return REGIONTYPE_TOWER

    else:
        return REGIONTYPE_FREE


# Construct Hypergame States
P1_REGIONS = [FREE_REGIONS, FREE_REGIONS, FREE_REGIONS]
p1_pos = list(itertools.product(*P1_REGIONS))
p1_health = tuple(itertools.product(("low", "med", "high"), repeat = 3))
# p1_health = tuple(itertools.product(range(0, 101, 34), repeat = 3))
p1_res = (0, 1, 2, 3)

# Approach 1
p1_list = [p1_pos, p1_health, p1_res]
p1_states = list(itertools.product(*p1_list))
p2_states = tuple(itertools.product(("low", "med", "high"), repeat = 2))
# p2_states = tuple(itertools.product(range(0, 101, 34), repeat = 2))
```

```
# Approach 2
# p1_states = tuple(itertools.product(p1_pos, p1_health, p1_res))
# p2_states = tuple(itertools.product(range(0, 101, 34), repeat = 2))

turn = (1, 2)
aut_states = (0, 1, 2)
igraph_states = (0, 1, 2, 3, 4, 5, 6, 7)
hstates = tuple(itertools.product(p1_states, p2_states, turn, aut_states,\
                                                              igraph_states))
hmap = dict(zip(hstates, map(HState, hstates)))
print(f"We have {len(hmap)} number of hypergame vertices.")
# print(f"Length of hstates = {len(hstates)}.")
# print(f"Length of p1_pos = {len(p1_pos)}, p1_health = {len(p1_health)},\
#                                                     p1_res = {len(p1_res)}.")
# print(f"Length of p1_states = {len(p1_states)}, p2_states = {len(p2_states)}.")
# print(f"Length of hstates = {len(p1_states)*len(p2_states)*2*3*8}\
#                                                      (manually calculated)")

# Label states with types
for val in hmap.values():
    val.type = hstate_type(val)
```

```
We have 46656000 number of hypergame vertices.
```

```
# Define Automaton Transition Function

def aut_trans(q, p1_loc):
    """ Specification: Eventually(RELIC & Eventually(SAFE)) """

    # DFA considers location of P1 villager unit
    v_loc = u.p1_loc[0]
    relic_collected = False
    if q == 0:
        if v_loc in RELIC:
            relic_collected = True
            return 1
        # direct transition (unreachable)
        elif v_loc in SAFE and relic_collected == True:
            return 2
        else:
            return 0

    elif q == 1:
        if v_loc in SAFE:
            return 2
        else:
            return 1

    elif q == 2:
        return 2

```

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

```
# Define P2 Information Graph Transition Function

def igraph_trans(i, action_name, res):
"""
Inference Graph (IGraph) has 8 states.
0: P2 does not know "suicide" and does not know if P1, P2 resources are collected.
1: P2 knows "suicide" and does not know if P1, P2 resources are collected.
2: P2 does not know "suicide" and knows only P2 resource is collected.
3: P2 knows "suicide" and knows only P2 resource is collected.
4: P2 does not know "suicide" and knows only P1 resource is collected.
5: P2 knows "suicide" and knows only P1 resource is collected.
6: P2 does not know "suicide" and knows both P1, P2 resources is collected.
7: P2 knows "suicide" and knows both P1, P2 resources is collected.
"""

    # considering one action based transitions, at a time
    # check v_pos to determine the resources collected
    if i == 0:
        if "suicide" == action_name:
            return 1
        elif res == 1:
            return 2
        elif res == 2:
            return 4
        # elif res == 3:
        #     return 6
        else:
            return 0

    elif i == 1:
        if res == 1:
            return 3
        elif res == 2:
            return 5
        # elif res == 3:
        #     return 7
        else:
            return 1

    elif i == 2:
        if "suicide" == action_name:
            return 3
        elif res == 3:
            return 6
        else:
            return 2

    elif i == 3:
```

```
        if res == 3:
            return 7
        else:
            return 3

    elif i == 4:
        if "suicide" == action_name:
            return 5
        elif res == 3:
            return 6
        else:
            return 4

    elif i == 5:
        if res == 3:
            return 7
        else:
            return 5

    elif i == 6:
        if "suicide" == action_name:
            return 7
        else:
            return 6

    elif i == 7:
        return 7

    else:
        raise ValueError("Unknown igraph state.")
```

Classes and helper functions for tracking game level information

```
# Class for the units and players
# Track game level information such as player resources and unit health
class Unit:
    def __init__(self, name, health):
        self._name = name
        self._health = max(health, 0)

    def __str__(self):
        return f"Name of unit: {self._name}"

    def get_health(self):
        if self._health >= 0:
            return self._health
        # else:
        #     # print(f"{self._name} is dead.")
        #     return self._health

```

```
    def set_health(self, hval):
        self._health = max(hval, 0)
        # if self._health == 0:
            # print(f"{self._name} is dead.")
        # print(f"Health of {self._name} updated to {self._health}")

# Class for Player 1
class Player1(Unit):
    units = ("villager", "knight", "archer")

    def __init__(self, resource):
        self._units = []
        self.def_health = 100
        self._resource = max(resource, 0)
        for name in self.units:
            self._units.append(Unit(name, self.def_health))

    def get_resource(self):
        if self._resource >= 0:
            return self._resource
        # else:
        #     print(f"{self.__class__.__name__} is out of resources.")

    def set_resource(self, rval):
        self._resource = max(rval, 0)
        # if self._resource == 0:
        #     print(f"{self.__class__.__name__} is out of resources.")

    def get_health(self, name):
        if name == "villager":
            return self._units[0].get_health()
        elif name == "knight":
            return self._units[1].get_health()
        else:
            return self._units[2].get_health()

    def set_health(self, name, hval):
        if name == "villager":
            return self._units[0].set_health(hval)
        elif name == "knight":
            return self._units[1].set_health(hval)
        else:
            return self._units[2].set_health(hval)

# Class for Player 2
class Player2(Unit):
    units = ("tower1", "tower2")

    def __init__(self):
        self._units = []
        self.def_health = 100
```

```
        for name in self.units:
            self._units.append(Unit(name, self.def_health))

    def get_health(self, name):
        if name == "tower1":
            return self._units[0].get_health()
        else: # tower2
            return self._units[1].get_health()

    def set_health(self, name, hval):
        if name == "tower1":
            return self._units[0].set_health(hval)
        else: # tower2
            return self._units[1].set_health(hval)

# Create the players
# P1 includes 1 villager, 1 archer, 1 knight with 100 resources to begin with.
# P2 includes 2 watch towers.
# Each unit begins with 100 health.
p1 = Player1(100)
p2 = Player2()
```

```
# Auxillary functions for use in Actions

# Get the neighbor regions of a given region
def get_neighbors(region):
    switch = {
                1: (2,),
                2: (1, 15,),
                7: (8,),
                8: (7, 13, 14, 15,),
                13: (8, 11, 14, 15,),
                14: (8, 11, 13, 15),
                10: (11,),
                11: (10, 13, 14, 15, 16,),
                15: (8, 11, 13, 14, 16,),
                16: (11, 15,),
              }
    return switch.get(region, 2)

# Convert health of player units from absolute values to discrete steps
# Continuous: 0 t0 100; allowable in steps of 5, 10
# Discrete: 3 step intervals; [0 to 34) = low; [34, 68) = med; [68, 100] = high

def get_discrete_health(h_val):
    if h_val not in range(0, 101):
        if h_val > 100:
            return "high"
        else: # h_val < 0
            return "low"

```

```python
    else:
        if 0 <= h_val < 34:
            return "low"
        elif 34 <= h_val < 68:
            return "med"
        else: # 68 <= h_val <= 100
            return "high"

# Update health of the unit during a battle scenario
# Step 1: Check if the unit launching the attack is in tower inner/outer region
# Step 2: Select the hitpoint according to the region and the source unit
# Step 3: Decrement the health for target unit as per the hitpoint
# Step 4: Return the discretized health to update state for the target unit
def update_health(**kwargs):

    region = kwargs.get("region")
    if region == "tower1_inner" or region == "tower2_inner":
        hit_step = HIGH_HIT_STEP
    elif region == "tower1_outer" or region == "tower2_outer":
        hit_step = LOW_HIT_STEP
    else: # any other arbitrary region
        hit_step = LOW_HIT_STEP

    source = kwargs.get("source")
    if source == "knight": # knight kills faster than other units
        hitpoint = hit_step * 2
    else:
        hitpoint = hit_step

    target = kwargs.get("target")
    if target == "archer":
        p1.set_health("archer", p1.get_health("archer") - hitpoint)
        health_bin = get_discrete_health(p1.get_health("archer"))
    elif target == "knight":
        p1.set_health("knight", p1.get_health("knight") - hitpoint)
        health_bin = get_discrete_health(p1.get_health("knight"))
    elif target == "villager":
        p1.set_health("villager", p1.get_health("villager") - hitpoint)
        health_bin = get_discrete_health(p1.get_health("villager"))
    elif target == "tower1":
        p1.set_health("tower1", p1.get_health("tower1") - hitpoint)
        health_bin = get_discrete_health(p1.get_health("tower1"))
    elif target == "tower2":
        p1.set_health("tower2", p1.get_health("tower2") - hitpoint)
        health_bin = get_discrete_health(p1.get_health("tower2"))
    else:
        raise ValueError("Unknown target unit.")

    return health_bin


# Convert resources collected by P1 from absolute values to discrete steps
```

```
# Continous: 0 to 600 in steps of 10
# Discrete: (resource in P1 region, resource in P2 region) --> 0 1 2 3

def update_resource(res_val, v_pos):
    visited = set()

    if res_val <= 100 and v_pos not in visited:
        res_state = 0 # only default resources, none collected
    elif 100 < res_val <= 250 and v_pos in P1_STONE:
        res_state = 2 # only collected from P1 region
        visited.add(P1_STONE)
    elif 500 <= res_val <= 600 and v_pos in P2_STONE:
        res_state = 1 # only collected from P2 region
        visited.add(P2_STONE)
    elif 600 < res_val <= 750:
        res_state = 3 # collected from both P1 and P2 region
    else:
        res_state = 0
    return res_state
```

```
# ACTIONS
# -------

# Define the Actions for the players
# act(state) -> new-state

def archer_attack(u, **kwargs):

    # Preconditions
    # attack is accessible to P1 only in the tower/relic region
    if u.turn == 2 or u.type in (REGIONTYPE_OBST or REGIONTYPE_STONE \
                                                 or REGIONTYPE_FREE):
        return None

    # Apply action to arena state
    archer_loc = u.p1_loc[2]
    archer_health = u.p1_health[2]
    knight_health = u.p1_health[1]
    vill_health = u.p1_health[0]
    p1res = u.p1_res
    t1health = u.p2_health[0]
    t2health = u.p2_health[1]

    if u.turn == 1:
        # Scenario: archer attacking towers
        # handle health update for tower 1 due to archer
        if archer_loc in TOWER1:
            if archer_loc in TOWER_INNER:
                t1health_10 = update_health(source = "archer",\
                                    target = "tower1", region = "tower1_inner")
                v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
```

```
                                                    (t1health_10, t2health), 2)
            else: # archer_loc in TOWER_OUTER:
                t1health_5 = update_health(source = "archer",\
                                           target = "tower1",\
                                           region = "tower1_outer")
                v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
                                                     (t1health_5, t2health), 2)

        # handle health update for tower 2 due to archer
        elif archer_loc in TOWER2:
            if archer_loc in TOWER_INNER:
                t2health_10 = update_health(source = "archer",\
                                            target = "tower2",\
                                            region = "tower2_inner")
                v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
                                                    (t1health, t2health_10), 2)
            else: # archer_loc in TOWER_OUTER:
                t2health_5 = update_health(source = "archer",\
                                           target = "tower2",\
                                           region = "tower2_outer")
                v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
                                                     (t1health, t2health_5), 2)

        else:
            v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
                                                              (u.p2_health), 2)


    else:   # u.turn == 2:
        # Scenario: towers attacking archer
        # handle health update for archer due to tower 1
        if archer_loc in TOWER1:
            if archer_loc in TOWER_INNER:
                a_health_10 = update_health(source = "tower1",\
                                            target = "archer",\
                                            region = "tower1_inner")
                v_arena_state = ((u.p1_loc, (a_health_10, knight_health,\
                                                vill_health), u.p1_res),\
                                                      (u.p2_health), 1)
            else: # archer_loc in TOWER_OUTER:
                a_health_5 = update_health(source = "tower1",\
                                           target = "archer",\
                                           region = "tower1_outer")
                v_arena_state = ((u.p1_loc, (a_health_5, knight_health,\
                                               vill_health), u.p1_res),\
                                                              (u.p2_health), 1)

        # handle health update for archer due to tower 2
        elif archer_loc in TOWER2:
            if archer_loc in TOWER_INNER:
                a_health_10 = update_health(source = "tower2",\
                                            target = "archer",\
```

```
                                            target = "archer",\
                                            region = "tower2_inner")
                v_arena_state = ((u.p1_loc, (a_health_10, knight_health,\
                                             vill_health), u.p1_res),\
                                                              (u.p2_health), 1)
            else: # archer_loc in TOWER_OUTER:
                a_health_5 = update_health(source = "tower2",\
                                           target = "archer",\
                                           region = "tower2_outer")
                v_arena_state = ((u.p1_loc, (a_health_5, knight_health,\
                                             vill_health), u.p1_res),\
                                                              (u.p2_health), 1)

        else:
            v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
                                                              (u.p2_health), 1)

    # Apply automaton transition
    v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])

    # Apply igraph transition
    v_igraph_state = igraph_trans(u.igraph_state, "archer_attack", u.p1_res)

    # Postconditions
    try:
        v = hmap[v_arena_state + (v_aut_state, ) + (v_igraph_state, )]
    except KeyError:
        return None

    return v if v.type != REGIONTYPE_OBST else None


def knight_attack(u, **kwargs):

    # Preconditions
    # attack is accessible to P1 only in the tower/relic region
    if u.turn == 2 or u.type in (REGIONTYPE_OBST or REGIONTYPE_STONE\
                                                 or REGIONTYPE_FREE):
        return None

    # Apply action to arena state
    knight_loc = u.p1_loc[1]
    archer_health = u.p1_health[2]
    knight_health = u.p1_health[1]
    vill_health = u.p1_health[0]
    p1res = u.p1_res
    t1health = u.p2_health[0]
    t2health = u.p2_health[1]

    if u.turn == 1:
        # Scenario: knight attacking towers
        # handle health update for tower 1 due to knight
```

```
        if knight_loc in TOWER1:
            if knight_loc in TOWER_INNER:
                t1health_10 = update_health(source = "knight",\
                                            target = "tower1",\
                                            region = "tower1_inner")
                v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
                                                  (t1health_10, t2health), 2)
            else: # knight_loc in TOWER_OUTER:
                t1health_5 = update_health(source = "knight",\
                                           target = "tower1",\
                                           region = "tower1_outer")
                v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
                                                   (t1health_5, t2health), 2)

        # handle health update for tower 2 due to knight
        elif knight_loc in TOWER2:
            if knight_loc in TOWER_INNER:
                t2health_10 = update_health(source = "knight",\
                                            target = "tower2",\
                                            region = "tower2_inner")
                v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
                                                  (t1health, t2health_10), 2)
            else: # knight_loc in TOWER_OUTER:
                t2health_5 = update_health(source = "knight",\
                                           target = "tower2",\
                                           region = "tower2_outer")
                v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
                                                   (t1health, t2health_5), 2)

        else:
            v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
                                                            (u.p2_health), 2)

    else:   # u.turn == 2:
        # Scenario: towers attacking knight
        # handle health update for knight due to tower 1
        if knight_loc in TOWER1:
            if knight_loc in TOWER_INNER:
                k_health_10 = update_health(source = "tower1",\
                                            target = "knight",\
                                            region = "tower1_inner")
                v_arena_state = ((u.p1_loc, (archer_health, k_health_10,\
                                             vill_health), u.p1_res),\
                                                          (u.p2_health), 1)
            else: # knight_loc in TOWER_OUTER:
                k_health_5 = update_health(source = "tower1",\
                                           target = "knight",\
                                           region = "tower1_outer")
                v_arena_state = ((u.p1_loc, (archer_health, k_health_5,\
                                             vill_health), u.p1_res),\
                                                          (u.p2_health), 1)

```

```
        # handle health update for knight due to tower 2
        elif knight_loc in TOWER2:
            if knight_loc in TOWER_INNER:
                k_health_10 = update_health(source = "tower2",\
                                            target = "knight",\
                                            region = "tower2_inner")
                v_arena_state = ((u.p1_loc, (archer_health, k_health_10,\
                                             vill_health), u.p1_res),\
                                                         (u.p2_health), 1)
            else: # knight_loc in TOWER_OUTER:
                k_health_5 = update_health(source = "tower2",\
                                           target = "knight",\
                                           region = "tower2_outer")
                v_arena_state = ((u.p1_loc, (archer_health, k_health_5,\
                                             vill_health), u.p1_res),\
                                                         (u.p2_health), 1)

        else:
            v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
                                                         (u.p2_health), 1)

    # Apply automaton transition
    v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])

    # Apply igraph transition
    v_igraph_state = igraph_trans(u.igraph_state, "knight_attack", u.p1_res)

    # Postconditions
    try:
        v = hmap[v_arena_state + (v_aut_state, ) + (v_igraph_state, )]
    except KeyError:
        return None

    return v if v.type != REGIONTYPE_OBST else None


def vill_attack(u, **kwargs):

    # Preconditions
    # attack is accessible to P1 only in the tower/relic region
    if u.turn == 2 or u.type in (REGIONTYPE_OBST or REGIONTYPE_STONE\
                                                 or REGIONTYPE_FREE):
        return None

#     print("\t\tN: Precondition ok.")

    # Apply action to arena state
    vill_loc = u.p1_loc[0]
    archer_health = u.p1_health[2]
    knight_health = u.p1_health[1]
    vill_health = u.p1_health[0]
```

```
    p1res = u.p1_res
    t1health = u.p2_health[0]
    t2health = u.p2_health[1]

    # u.turn == 2:
    if u.turn == 2:
        # Scenario: towers attacking villager
        # handle health update for villager due to tower 1
        if vill_loc in TOWER1:
            if vill_loc in TOWER_INNER:
                v_health_10 = update_health(source = "tower1",\
                                            target = "villager",\
                                            region = "tower1_inner")
                v_arena_state = ((u.p1_loc, (archer_health, knight_health,\
                                                  v_health_10), u.p1_res),\
                                                          (u.p2_health), 1)
            else: # vill_loc in TOWER_OUTER:
                v_health_5 = update_health(source = "tower1",\
                                           target = "villager",\
                                           region = "tower1_outer")
                v_arena_state = ((u.p1_loc, (archer_health, knight_health,\
                                                    v_health_5), u.p1_res),\
                                                           (u.p2_health), 1)

        # handle health update for villager due to tower 2
        elif vill_loc in TOWER2:
            if vill_loc in TOWER_INNER:
                v_health_10 = update_health(source = "tower2",\
                                            target = "villager",\
                                            region = "tower2_inner")
                v_arena_state = ((u.p1_loc, (archer_health, knight_health,\
                                                  v_health_10), u.p1_res),\
                                                          (u.p2_health), 1)
            else: # vill_loc in TOWER_OUTER:
                v_health_5 = update_health(source = "tower2",\
                                           target = "villager",\
                                           region = "tower2_outer")
                v_arena_state = ((u.p1_loc, (archer_health, knight_health,\
                                                    v_health_5), u.p1_res),\
                                                           (u.p2_health), 1)

        else:
            v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res),\
                                                           (u.p2_health), 1)

    else: # u.turn == 1
        v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res), (u.p2_health), 2)


    # Apply automaton transition
    v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])

```

```
    # Apply igraph transition
    v_igraph_state = igraph_trans(u.igraph_state, "vill_attack", u.p1_res)

    # Postconditions
    try:
        v = hmap[v_arena_state + (v_aut_state, ) + (v_igraph_state, )]
    except KeyError:
        return None

    return v if v.type != REGIONTYPE_OBST else None


def archer_move(u, **kwargs):

    # Precondition 1: move is accessible to P1 archer in FREE/TOWER/RELIC region
    if u.turn == 2 or u.type in REGIONTYPE_OBST:
        return None

    # Precondition 2: to perform move, check if the target belongs to neighbors
    # of the current region i.e. target is reachable from current region
    target = kwargs.get("target", -1)
    archer_loc = u.p1_loc[2]
    neighbors = get_neighbors(archer_loc)
    if target not in neighbors:
        return None

    # Apply action
    # Step 1: move the unit to the target
    u_p1_loc = (target, u.p1_loc[1], u.p1_loc[0])
    v_arena_state = ((u_p1_loc, u.p1_health, u.p1_res), (u.p2_health), 2)

    # Apply automaton transition
    v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])

    # Apply igraph transition
    v_igraph_state = igraph_trans(u.igraph_state, "archer_move", u.p1_res)

    # Postconditions
    try:
        v = hmap[v_arena_state + (v_aut_state, ) + (v_igraph_state, )]
    except KeyError:
        return None

    # Postconditions
    return v if v.type != REGIONTYPE_OBST else None


def knight_move(u, **kwargs):

    # Precondition 1: move is accessible to P1 knight in FREE/TOWER/RELIC region
    if u.turn == 2 or u.type in REGIONTYPE_OBST:
        return None
```

```
        return None

    # Precondition 2: to perform move, check if the target belongs to neighbors
    # of the current region i.e. target is reachable from current region
    target = kwargs.get("target", -1)
    knight_loc = u.p1_loc[1]
    neighbors = get_neighbors(knight_loc)
    if target not in neighbors:
        return None

    # Apply action
    # Step 1: move the unit to the target
    u_p1_loc = (u.p1_loc[2], target, u.p1_loc[0])
    v_arena_state = ((u_p1_loc, u.p1_health, u.p1_res), (u.p2_health), 2)

    # Apply automaton transition
    v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])

    # Apply igraph transition
    v_igraph_state = igraph_trans(u.igraph_state, "knight_move", u.p1_res)

    # Postconditions
    try:
        v = hmap[v_arena_state + (v_aut_state, ) + (v_igraph_state, )]
    except KeyError:
        return None

    # Postconditions
    return v if v.type != REGIONTYPE_OBST else None


def vill_move(u, **kwargs):

    # Precondition 1: move is accessible to P1 knight in FREE/TOWER/RELIC region
    if u.turn == 2 or u.type in REGIONTYPE_OBST:
        return None

    # Precondition 2: to perform move, check if the target belongs to neighbors
    # of the current region i.e. target is reachable from current region
    target = kwargs.get("target", -1)
    vill_loc = u.p1_loc[0]
    neighbors = get_neighbors(vill_loc)
    if target not in neighbors:
        return None

    # Apply action
    # Step 1: move the unit to the target
    u_p1_loc = (u.p1_loc[2], u.p1_loc[1], target)
    v_arena_state = ((u_p1_loc, u.p1_health, u.p1_res), (u.p2_health), 2)

    # Apply automaton transition
    v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])
```

```
    # Apply igraph transition
    v_igraph_state = igraph_trans(u.igraph_state, "vill_move", u.p1_res)

    # Postconditions
    try:
        v = hmap[v_arena_state + (v_aut_state, ) + (v_igraph_state, )]
    except KeyError:
        return None

    # Postconditions
    return v if v.type != REGIONTYPE_OBST else None


def archer_suicide(u, **kwargs):

    # Precondition 1: suicide is accessible to P1 archer only in
    # FREE/TOWER/RELIC/STONE region
    if u.turn == 2 or u.type in REGIONTYPE_OBST :
        return None

    # Apply action

    # return the absolute health value of archer bw 0-100
    a_health = p1.get_health("archer")
    # set health of archer to 0
    p1.set_health("archer", 0)
    archer_health = get_discrete_health(a_health)

    # discretize the health in steps
    # Boost health of remaining friendly units
    # Step 1: Get the absolute health value (bw 0-100) of the archer unit
    # Step 2: Distribute archers' health equally to remaining friendly units
    # Step 3: Scale absolute health of friendly units back to discrete steps
    k_health = p1.get_health("knight")
    k_health = max(k_health + a_health*0.5, 100)
    knight_health = get_discrete_health(k_health)

    v_health = p1.get_health("villager")
    v_health = max(v_health + a_health*0.5, 100)
    vill_health = get_discrete_health(v_health)

    # Construct the state
    v_arena_state = ((u.p1_loc, (archer_health, knight_health, vill_health),\
                                                      u.p1_res), (u.p2_health), 2)

    # Apply automaton transition
    v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])

    # Apply igraph transition
    v_igraph_state = igraph_trans(u.igraph_state, "archer_suicide", u.p1_res)
```

```
    # Postconditions
    try:
        v = hmap[v_arena_state + (v_aut_state, ) + (v_igraph_state, )]
    except KeyError:
        return None

    # Postconditions
    if v.type not in REGIONTYPE_OBST:
        return v


def vill_build_castle(u, **kwargs):

    # Precondition 1: build is accessible to P1 villager only in
    # FREE/TOWER/RELIC region
    if u.turn == 2 or u.type in (REGIONTYPE_OBST, REGIONTYPE_STONE):
        return None

    # Precondition 2: build castle requires 500 stones or resources
    if u.p1_res != 3:
        # raise ValueError("Insufficient resources.")
        return None

    # Apply action
    # Step 1: update the resources for castle build action
    res = p1.get_resource()
    if res < COST_CASTLE:
        return None
    else: # res >= COST_CASTLE
        res -= COST_CASTLE
        p1.set_resource(res)

    # Construct the state
    v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res), (u.p2_health), 2)

    # Apply automaton transition
    v_aut_state = aut_trans(u.aut_state, v_arena_state[0][0])

    # Apply igraph transition
    v_igraph_state = igraph_trans(u.igraph_state, "vill_build_castle", u.p1_res)

    # Postconditions
    try:
        v = hmap[v_arena_state + (v_aut_state, ) + (v_igraph_state, )]
    except KeyError:
        return None

    # Postconditions
    if v.type not in REGIONTYPE_OBST:
        return v

```

```python

def vill_collect(u, **kwargs):

    # Precondition 1: collect stone is accessible to P1 villager only in
    # STONE region
    if u.turn == 2 or u.type not in REGIONTYPE_STONE:
        return None

    # Apply action
    # Step 1: update the resources due to collect action
    res = p1.get_resource()
    res += FIXED_RESOURCE_STEP
    p1.set_resource(res)

    # Step 2: update resources collected to player state from villager position
    pos_v = u.p1_loc[0]
    p1_res = update_resource(p1.get_resource(), pos_v)
    v_arena_state = ((u.p1_loc, u.p1_health, p1_res), (u.p2_health), 2)

    # Apply automaton transition
    v_aut_state = aut_trans(u.aut_state, u.p1_loc)

    # Apply igraph transition
    v_igraph_state = igraph_trans(u.igraph_state, "vill_collect", p1_res)

    # Postcondition
    try:
        v = hmap[v_arena_state + (v_aut_state, ) + (v_igraph_state, )]
    except KeyError:
        return None

    return v if v.type != REGIONTYPE_OBST else None


def nop(u, **kwargs):

    # Apply action
    if u.turn == 1:
        v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res), (u.p2_health), 2)
    else: # u.turn == 2
        v_arena_state = ((u.p1_loc, u.p1_health, u.p1_res), (u.p2_health), 1)

    # Apply automaton transition
    v_aut_state = aut_trans(u.aut_state, u.p1_loc)

    # Apply igraph transition
    v_igraph_state = igraph_trans(u.igraph_state, "nop", u.p1_res)

    # Postcondition
    try:
        v = hmap[v_arena_state + (v_aut_state, ) + (v_igraph_state, )]
    except KeyError:
```

```
        return None

    return v if v.type != REGIONTYPE_OBST else None

# P2's perceived action set of P1's at different states of IGraph
ACT_P1_ISTATE_0 = (archer_move, knight_move, vill_move, nop)
ACT_P1_ISTATE_1 = (archer_move, knight_move, vill_move, nop, archer_suicide,\
                                  archer_attack, knight_attack, vill_attack)
ACT_P1_ISTATE_2 = (archer_move, knight_move, vill_move, nop, vill_collect,\
                                  archer_attack, knight_attack, vill_attack)
ACT_P1_ISTATE_3 = (archer_move, knight_move, vill_move, nop, archer_suicide,\
                    vill_collect, archer_attack, knight_attack, vill_attack)
ACT_P1_ISTATE_4 = (archer_move, knight_move, vill_move, nop, vill_collect,\
                                  archer_attack, knight_attack, vill_attack)
ACT_P1_ISTATE_5 = (archer_move, knight_move, vill_move, nop, archer_suicide,\
                    vill_collect, archer_attack, knight_attack, vill_attack)
ACT_P1_ISTATE_6 = (archer_move, knight_move, vill_move, nop, vill_collect,\
                   vill_build_castle, archer_attack, knight_attack, vill_attack)
ACT_P1_ISTATE_7 = (archer_move, knight_move, vill_move, nop, archer_suicide,\
                              vill_collect, vill_build_castle, archer_attack,\
                                                  knight_attack, vill_attack)

# P2's action set (known accurately by P1 and P2)
ACT_P2 = (archer_attack, knight_attack, vill_attack, nop)
```

```
# Construct Hypergame Edge
# ------------------------

# HELPER FUNCTIONS: NEIGHBORHOOD

# Tuple of move actions
MOVE_ACTIONS = (archer_move, knight_move, vill_move)

def apply_actions_p1(u, hmap):
    V_u = []
    for act in ACT_P1_ISTATE_7:
        for region in REGIONS:
    # Iterate through each available region and set target equal to this region
    # only for the move actions
            if act in MOVE_ACTIONS:
                v = act(u, target = region)
            else:
                v = act(u)
            if v is not None:
                V_u.append((v, act))

    return tuple(V_u)


def apply_actions_p2(u, hmap):
    V_u = []
```

```
    for act in ACT_P2:
        v = act(u)
        if v is not None:
            V_u.append((v, act))

    return tuple(V_u)


def neighbors(u, hmap):

    if u.turn == 1:
        return apply_actions_p1(u, hmap)
    else:
        return apply_actions_p2(u, hmap)


# ADD EDGES
# Set initial states
v0 = hmap[v0_tuple]
print(v0, v0.type)
print("----------------------------")

# Construct the edges of hypergame

hedges = set()
stack = [v0]
visited = set()

while len(stack) > 0:
    u = stack.pop()
    visited.add(u)

    succ = neighbors(u, hmap)
    for v, act in succ:
        hedges.add((u, v, act))
        if v not in visited:
            stack.append(v)

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.
```

```
# ELIMINATE UNREACHABLE STATES
# ----------------------------
# As we have computed edges based on initial state,
# we use edges to eliminate unreachable states.

# Prune the hstates based on reachability
reachable = set()
```

```
for u, v, _ in hedges:
    reachable.add(u)
    reachable.add(v)

prune = set()
for v_tuple in hmap.keys():
    if hmap[v_tuple] not in reachable:
        prune.add(v_tuple)

for v_tuple in prune:
    hmap.pop(v_tuple)

print(f"len(hmap)={len(hmap)}, len(hedges)={len(hedges)}")
```

```
len(hmap)=98529, len(hedges)=721477
```

```
# PROJECTION OF HYPERGAME ONTO GAME
# ---------------------------------

# Construct a simple game with perfect information
# Compute projection of hmap onto gmap

gstates = set()
for hstate in hmap.values():
    gstates.add(hstate.arena_state + (hstate.aut_state, ))

gedges = set()
for hedge in hedges:
    u_h, v_h, a = hedge
    u_g = u_h.arena_state + (u_h.aut_state, )
    v_g = v_h.arena_state + (v_h.aut_state, )
    gedges.add((u_g, v_g, a))

print(f"len(gstates)={len(gstates)}, len(gedges)={len(gedges)}")

# Formulate a game object
game = Game()

gmap = dict()
for st in gstates:
    gv = GameVertex(name=st, turn=st[2])
    gmap[st] = gv
    game.add_vertex(gv)

for u, v, a in gedges:
    game.add_edge(GameEdge(u=gmap[u], v=gmap[v], act=a))

print(game)
```

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

Perfect Information Game Construction and Solution

```
# SURE-WINNING SOLUTION
# ---------------------

# Mark final states
for v in game.vertices:
    if v.name[-1] == 2:   # add final aut_state here
        game.mark_final(v)


# Invoke solver
sw_solver = SureWinning(game=game)
sw_solver.solve()
print(f"sw_solver.p1_win={len(sw_solver.p1_win)} of which {len(game.final)}\
                                                          are final states.")
# 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

```
# HYPERGAME OBJECT CONSTRUCTION
# ----------------------------


hypergame = Hypergame()

# Add vertices
for v in hmap.values():
    hypergame.add_vertex(HypergameVertex(hstate=v))

# Add edges
for u, v, a in hedges:
    hypergame.add_edge(HypergameEdge(u=HypergameVertex(hstate=u),\
                                     v=HypergameVertex(hstate=v), act=a))

# Mark final states
final = set()
p1_win = {u.name for u in sw_solver.p1_win}
for v in hypergame.vertices:
#     if v.game_vertex in p1_win:
    if v.hstate.aut_state == 2:
        final.add(v)
        hypergame.mark_final(v)

print(f"len(final)={len(final)}")
print(f"-------------------------------")
```

```
# Statistics of hypergame construction
print(hypergame)
print(f"Num(P1-states)={len({v for v in hypergame.vertices\
                                  if v.hstate.turn == 1})}")
print(f"Num(P2-states)={len({v for v in hypergame.vertices\
                                  if v.hstate.turn == 2})}")
print(f"Num(aut_state=0)={len({v for v in hypergame.vertices\
                                  if v.hstate.aut_state == 0})}")
print(f"Num(aut_state=1)={len({v for v in hypergame.vertices\
                                  if v.hstate.aut_state == 1})}")
print(f"Num(aut_state=2)={len({v for v in hypergame.vertices\
                                  if v.hstate.aut_state == 2})}")
print(f"Num(igraph_state=0)={len({v for v in hypergame.vertices\
                                  if v.hstate.igraph_state == 0})}")
print(f"Num(igraph_state=1)={len({v for v in hypergame.vertices\
                                  if v.hstate.igraph_state == 1})}")
print(f"Num(igraph_state=2)={len({v for v in hypergame.vertices\
                                  if v.hstate.igraph_state == 2})}")
print(f"Num(igraph_state=3)={len({v for v in hypergame.vertices\
                                  if v.hstate.igraph_state == 3})}")
print(f"Num(igraph_state=4)={len({v for v in hypergame.vertices\
                                  if v.hstate.igraph_state == 4})}")
print(f"Num(igraph_state=5)={len({v for v in hypergame.vertices\
                                  if v.hstate.igraph_state == 5})}")
print(f"Num(igraph_state=6)={len({v for v in hypergame.vertices\
                                  if v.hstate.igraph_state == 6})}")
print(f"Num(igraph_state=7)={len({v for v in hypergame.vertices\
                                  if v.hstate.igraph_state == 7})}")
print(f"Num(archer_attack, knight_attack, vill_attack-edges)=\
{len({e for e in hypergame.edges if e.act in (archer_attack, knight_attack,\
                                                            vill_attack)})}")
print(f"Num(archer_move, knight_move, vill_move-edges)=\
{len({e for e in hypergame.edges if e.act in (archer_move, knight_move,\
                                                              vill_move)})}")
print(f"Num(archer_suicide-edges)=\
          {len({e for e in hypergame.edges if e.act in (archer_suicide, )})}")
print(f"Num(vill_build_castle-edges)=\
        {len({e for e in hypergame.edges if e.act in (vill_build_castle, )})}")
print(f"Num(vill_collect-edges)=\
              {len({e for e in hypergame.edges if e.act in (vill_collect, )})}")
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
```

```python
# DECEPTIVE ALMOST-SURE WINNING REGION
# -----------------------------------

# Define a function to mark whether a hypergame edge is permissive or not.

def is_permissive(e):
    turn = e.source.hstate.turn
    if turn == 2:
        return True

    igraph_state = e.source.hstate.igraph_state
    act = e.act

    if igraph_state == 0:
        res = act in ACT_P1_ISTATE_0

    elif igraph_state == 1:
        res = act in ACT_P1_ISTATE_1

    elif igraph_state == 2:
        res = act in ACT_P1_ISTATE_2

    elif igraph_state == 3:
        res = act in ACT_P1_ISTATE_3

    elif igraph_state == 4:
        res = act in ACT_P1_ISTATE_4

    elif igraph_state == 5:
        res = act in ACT_P1_ISTATE_5

    elif igraph_state == 6:
        res = act in ACT_P1_ISTATE_6

    elif igraph_state == 7:
        res = act in ACT_P1_ISTATE_7


# Invoke DASW Solver
```

```
dasw = DeceptiveAlmostSureWinning(hypergame=hypergame,\
                                  is_permissive=is_permissive)
dasw.solve()
print(f"dasw.p1_win={len(dasw.p1_win)}")
```

```
dasw.p1_win=43969
```

Results Summary

```
# Problem Setup Parameters
print("Problem Setup Parameters")

print("   ", f"REGIONS        = {REGIONS}")
print("   ", f"FREE_REGIONS   = {FREE_REGIONS}")
print("   ", f"TOWERS         = {TOWERS}")
print("   ", f"TOWER1         = {TOWER1}")
print("   ", f"TOWER2         = {TOWER2}")
print("   ", f"TOWER_INNER    = {TOWER_INNER}")
print("   ", f"TOWER_OUTER    = {TOWER_OUTER}")
print("   ", f"STONE_REGIONS  = {STONE_REGIONS}")
print("   ", f"RELIC          = {RELIC}")
print("   ", f"P1_STONE       = {P1_STONE}")
print("   ", f"P2_STONE       = {P2_STONE}")
print("   ", f"SAFE           = {SAFE}")
print("   ", f"UNSAFE         = {UNSAFE}")
print("   ", f"v0             = {v0_tuple}")
print()
print("   ", f"ACT_P1_ISTATE_0 = {[a.__name__ for a in ACT_P1_ISTATE_0]}")
print("   ", f"ACT_P1_ISTATE_1 = {[a.__name__ for a in ACT_P1_ISTATE_1]}")
print("   ", f"ACT_P1_ISTATE_2 = {[a.__name__ for a in ACT_P1_ISTATE_2]}")
print("   ", f"ACT_P1_ISTATE_3 = {[a.__name__ for a in ACT_P1_ISTATE_3]}")
print("   ", f"ACT_P1_ISTATE_4 = {[a.__name__ for a in ACT_P1_ISTATE_4]}")
print("   ", f"ACT_P1_ISTATE_5 = {[a.__name__ for a in ACT_P1_ISTATE_5]}")
print("   ", f"ACT_P1_ISTATE_6 = {[a.__name__ for a in ACT_P1_ISTATE_6]}")
print("   ", f"ACT_P1_ISTATE_7 = {[a.__name__ for a in ACT_P1_ISTATE_7]}")
print("   ", f"ACT_P2          = {[a.__name__ for a in ACT_P2]}")

print()
print("Game with Perfect Information")
print("   ", f"game: |V|={len(game.vertices)} and |E|={len(game.edges)}")
print("   ", f"p1_win={len(sw_solver.p1_win)} of which {len(game.final)} are\
                                                          final states.")

print()
print("Hypergame")
print("   ", f"hgame: |V|={len(hypergame.vertices)} and\
                                          |E|={len(hypergame.edges)}")
print("   ", f"DASW={len(dasw.p1_win)} of which {len(hypergame.final)}\
                                                 are final states.")
print("   ", f"proj(DASW onto S)=\
```

```
print("   ", f"proj(DASW onto S)={
{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
```