THE EDGE-DISTINGUISHING GAME: ALGORITHM SUPPLEMENT

NATHANIEL BENJAMIN, ELISA BENTHEM, COOPER BURKEL, MARISSA CHESSER, AND MIKE JANSSEN

Abstract. In this supplementary work, we describe the Python code developed to explore the Edge-Distinguishing Game (EDGe).

1. Code for constructing the Digraph

Herein we provide an in-depth exploration of the algorithm and code used to construct the digraph introduced in Section 2 of the paper. Our algorithm was implemented in Python and uses the following packages: `itertools`, `networkx`, `string`, and `matplotlib.pyplot`.

For every board, we must enter the EDCN as a constant into the program. If the EDCN of a graph is unknown, one can use the code available at [edg24], which will determine the EDCN of the graph. To represent a game board in the program, we enter the graph as an *adjacency list*, where each entry in the list represents a vertex, and is itself a list of the vertices adjacent to that vertex. For ease of use, we have designed the program to iterate through the alphabet, so each vertex *must* be entered as a letter, first using the lowercase letters in ascending order then the uppercase letters in the same manner. Variables that will contain different versions of the boards are also initialized here. The example in the code is P_5 where we know $\lambda(P_5) = 3$, with the leftmost vertex label with an "a".

```
EDCN = 3

ADJACENCY_LIST = [["b"],["a","c"],["b","d"],["c","e"],["d"]]
possible_boards_list = []

legal_boards = []
legal_boards_list = []

playable_boards = []
unplayable_boards = []

playable_boards_list = []
unplayable_boards_list = []
```

We define each of the available colors as a number and store them in the list `COLORS`, including a -1 to represent an uncolored vertex. A version of the list without -1 is made for future use and is stored as `MOVES`.

```
COLORS = []
MOVES = []
COLORS.append(-1)

#  Adding all the colors based off of the EDCN starting at 1
for i in range(1,EDCN+1):
    COLORS.append(i)
    MOVES.append(i)
```

Next, we define a series of functions that, when used in conjunction, will generate all of the possible game states and determine whether a legal move can be played on any given board. The

Date: July 4, 2025.

first function, `possible_boards`, takes the adjacency list and uses it to construct and return a dictionary, which will be used to represent a board. Each key in a dictionary is the name of a vertex while the value is a list where the first entry is the color of the vertex and the other is a list of the adjacent vertices. The color of each vertex is initialized to 0 which indicates nothing has happened to the board.

```
#  Makes the blank boards
def possible_boards(board):
    inter_board = {}
    vertices = []

    for i, j in zip(board, st.ascii_letters):
        inter_board[j] = [0, i]
        vertices.append(j)

    return inter_board
```

Now we generate all of the possible board colorings by taking the Cartesian product of `COLORS` with itself a number of times equal to the number vertices in the board, which allows for any coloring which has color repetition. This function also calls `possible_boards` to create a new board for each coloring of the graph. All of the possible boards are added to a global list for further use.

```
#  Generate the possible colorings for every board
def make_boards(adj_list, colors):
    global possible_boards_list

    #  Create all of the permutations of the board colorings
    for possible_coloring in it.product(colors, repeat=len(adj_list)):
        board = possible_boards(ADJACENCY_LIST)
        for index, vertex in enumerate(board):
            board[vertex][0] = possible_coloring[index]

        possible_boards_list.append(board)

    return
```

Because we are allowing for every possible color permutation, many of the generated board colorings will be illegal. `no_repeat_edges` takes each individual board and checks whether there are any repeated colorings by using an edge list. Each edge list contains a series of two entry lists where the entries are the colors of each vertex creating the edge. The edge list contains twice the number of vertices to account for the reverse coloring of the edges. For example, suppose we have two adjacent vertices of colors 1 and 3 respectively. The forward edge coloring formed by these vertices is `[1,3]` while the reverse edge coloring would be `[3,1]`. In this way, we construct the edge list.

```
#  Remove all boards with illegal edge colorings
def no_repeat_edges(edges):
    no_repeats = True

    for side in edges:
        #  Allow for the empty board
        if side == [-1,-1] and (edges.count(side) == len(edges)):
            return True

        #  Edges only appearing once are legal
        elif edges.count(side) == 1:
            continue

        elif edges.count(side) > 1:
            #  Allow for repeated colorings if one of the vertices is uncolored
            if -1 in side:
                continue
```

```
18
19              #  Allow for a double edge if it is a color next to itself
20              elif side[0] == side[-1]:
21                  if edges.count(side) <= 2:
22                      continue
23                  else:
24                      return False
25
26              #  Return False for any other case
27              else:
28                  return False
29
30      # Returns True for all boards without a repeated edge
31      if no_repeats == True:
32          return True
```

In `is_safe`, we actually create the edge list used in `no_repeat_edges`. The edge list is then passed to `no_repeat_edges` when we call the function. If it returns `True`, we know that the board has a legal configuration.

```
1  #  Determine if a board is legal with the repeat edges function
2  def is_safe(board):
3      edges = []
4      vertices = []
5
6      #  Create the edge list by iterating through the vertices and pairing the vertex
       color with the color of the adjacent vertices
7      for vertex in board:
8          vertices.append(board[vertex][0])
9          for i in range(len(board[vertex][1])):
10             edges.append([board[vertex][0], board[board[vertex][1][i]][0]])
11
12     #  Will return true if the board is legal
13     if no_repeat_edges(edges):
14         return True
15
16     return False
```

To construct the digraph and properly label the boards, we will need to know which boards are markable and unmarkable, so here we determine whether a board is in a terminal state.

```
1  #  Return True for all boards which are PLAYABLE
2  def terminating_board(board, color_list):
3      value = board.values()
4      board_color = []
5
6      for entry in value:
7          board_color.append(entry[0])
8
9      #  All fully colored boards are terminating
10     if -1 not in board_color:
11         return False
12
13     #  If a board is not fully colored, check the uncolored vertices to see if they
       are markable
14     for vertex in board:
15         if board[vertex][0] != -1:
16             continue
17         else:
18             for color in color_list:
19                 board[vertex][0] = color
```

```
                if is_safe(board):
                    board[vertex][0] = -1
                    return True
                else:
                    board[vertex][0] = -1
                    continue

    #  Return False for all terminating boards
    return False
```

This is where all of the game states are generated and sorted. The function takes in the adjacency list and the color list and then creates each possible board and sorts them based on whether they are markable or unmarkable. Once each of the boards has been classified, the colorings of each board are recorded and added to a new list for simplicity. For the generation of the digraph we do not need to know which vertices in the graph are adjacent so we represent the graph in a simpler form.

```
#  Will "play" the whole game and make the game states
def play_game(adj_list, colors):
    global possible_boards_list
    global vertex_list
    global legal_boards
    global unplayable_boards
    global playable_boards
    global legal_boards_list
    global playable_boards_list
    global unplayable_boards_list
    global MOVES

    make_boards(adj_list, colors)

    for board in possible_boards_list:
        if is_safe(board):
            legal_boards.append(board)

    for board in legal_boards:
        if terminating_board(board, MOVES):
            playable_boards.append(board)
        else:
            unplayable_boards.append(board)

    #  Changing all the dictionaries into lists of vertex colors
    for board in playable_boards:
        temp_board = []
        for vertex in board:
            temp_board.append(board[vertex][0])

        playable_boards_list.append(temp_board)

    for board in unplayable_boards:
        temp_board = []
        for vertex in board:
            temp_board.append(board[vertex][0])

        unplayable_boards_list.append(temp_board)

    legal_boards_list.extend(playable_boards_list)
    legal_boards_list.extend(unplayable_boards_list)

    return

```

```
#  Call the function
play_game(ADJACENCY_LIST, COLORS)
```

Now that all of the game states have been generated and sorted, we can begin the process of constructing the digraph. To ensure that the digraph starts with an empty board and progresses through the game in the correct order, we will create a new list of the possible game boards. The list is sorted from the board with the least number of moves, the empty board, to the fully colored boards. Once all the boards are sorted, we label all terminating boards as being in a P-position.

```
empty_board = []
for i in range(len(legal_boards_list[0])):
    empty_board.append(-1)

#  Initialize the DiGraph
digraph = nx.DiGraph()
board_moves = [[]]

for i in range(len(ADJACENCY_LIST)):
    board_moves.append([])

#  Sort the board by how many vertices have been colored i.e. moves
for board in legal_boards_list:
    vertex_colors = 0
    for vertex in board:
        if vertex != -1:
            vertex_colors += 1

    board_moves[vertex_colors].append(board)

    #  Add boards to digraph and indicate whether they are in an end position "p"
    if board in playable_boards_list:
        digraph.add_node(tuple(board), position = "", layer = vertex_colors)
    else:
        digraph.add_node(tuple(board), position = "p", layer = vertex_colors)
```

We now walk through all of the game states. If a board is playable, we play the moves which can be made from that board. After that, we create a directed edge to connect the current node to a node in the next layer of the digraph that matches the coloring the move produces. Once this is completed for every node, we will have a fully constructed digraph.

```
for i in range(len(legal_boards_list[0])):
    for board in board_moves[i]:

        #  Make edges in the digraph based on the next move
        if digraph.nodes[tuple(board)]["position"] == "":

            #  Check possible moves from a board to another in the next layer
            for i in range(len(board)):
                if board[i] == -1:
                    game_state = list(board)

                    for move in MOVES:
                        game_state[i] = move

                        #  Make all of the edges in the DiGraph
                        if tuple(game_state) in digraph.nodes:
                            digraph.add_edge(tuple(board), tuple(game_state))
```

Starting at the end of the digraph, we walk through the game and label all terminating boards as being in a P-position. Any board which leads to a P-position is labeled as an N-position. We also label all boards which only lead to an N-position as boards in a P-position.

```
#  Label nodes in the digraph as n and p positions starting at the end of the game
for i in range(len(legal_boards_list[0])+1):
    for board in board_moves[len(board_moves)-i-1]:
        if (digraph.in_degree(tuple(board)) == 0) and (i != len(board_moves)-1):
            turns = 0
            for entry in board:
                if entry != 0:
                    turns += 1
            digraph.remove_node(tuple(board))

        #  A P-position if in an end state
        elif len(digraph.edges(tuple(board))) == 0:
            digraph.nodes[tuple(board)]["position"] = "p"

        else:
            #  N-position if directed to a P-position
            for edge in digraph.edges(tuple(board)):
                if digraph.nodes[edge[1]]["position"] == "p":
                    digraph.nodes[tuple(board)]["position"] = "n"

            #  P-position if only directed to N-positions
            if digraph.nodes[tuple(board)]["position"] == "":
                digraph.nodes[tuple(board)]["position"] = "p"
```

Finally, we check to see if the empty board is in a P or N-position. The result will tell us which player has the winning strategy.

```
#  Determine which player has a winning strategy
if digraph.nodes[tuple(empty_board)]["position"] == "n":
    print("Player 1 has a winning strategy")
else:
    print("Player 2 has a winning strategy")
```

The players can use this final section to play the game. All possible moves from a given node are printed and the player with the winning strategy will want to play the specified move on any vertex labeled with a P. If Player 2 has the winning strategy, any move by Player 1 will lead to at least one winning board for Player 2.

```
#  Print possible moves based on board
for edge in digraph.edges(tuple(empty_board)):
    print(edge, "\t", digraph.nodes[edge[1]]["position"])
```

References

[edg24] *EDCN determination program*, https://github.com/Cjburkel/EDCN-Determination-Program, July 2024.