22nd April, 2020

The General Mutilated Chessboard

Nishant Mahesh
Parul Joshi
Siddhartha Prabakar Nadathur
Soham De

Ashoka University

Contents

1. Problem Description

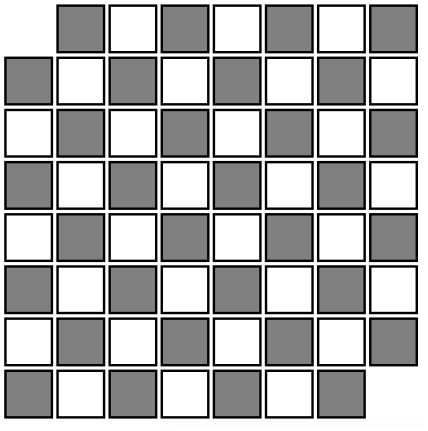
1.1 Abstract

Everyone has played, or at least heard of, chess in their life - a game based on an 8 x 8 grid and opponents with pieces that have varying powers. Almost 75 years ago, British-American philosopher Max Black, a leading figure in analytic philosophy during the years post World War 2, proposed a now-famous tiling puzzle in his book *Critical Thinking* called the Mutilated Chessboard problem [3]. While various mathematicians have tackled this problem in different perspectives, this report takes a look at some modified versions of the original question. Although done as the Semester Project for CS 1104 at Ashoka University, we go beyond the scope of the original question, addressing more advanced variants and also building a WebApp and a GUI to visually represent the solution.

1.2 Original Mutilated Chessboard Problem

The original Mutilated Chessboard Problem, proposed by Max Black takes an 8 x 8 chessboard where two squares at diagonally opposite corners of the board are cut-off. The problem asks us to check if it is possible to place 31 given dominoes (where each domino covers two adjacent squares) on the resulting grid in such a manner, that they don't overlap but also cover all of the remaining 62 squares on the board [1].

Figure 1.1: Two diagonally opposite squares removed



1.3 General Mutilated Chessboard Problem

We now pose a modern version of the previous question - A General Mutilated Chessboard Problem. Like before, we are given an 8 x 8 chessboard in which two positions are cut-off and are not available for placing any piece. While these may be diagonally opposite corners, the positions are not restricted by any constraint. Now, with the available 31 dominoes, where a single domino can cover exactly two consecutive squares on the board i.e. it can cover exactly one black tile and one of its neighbouring white tiles, we are required to check if the remaining Chessboard can be tiled or not.

The following figures show the Original Chessboard without any squares removed, and a Sample Output, where after removing 2 squares, it **is possible** to cover the remaining with dominoes.

Figure 1.2: Original Chessboard

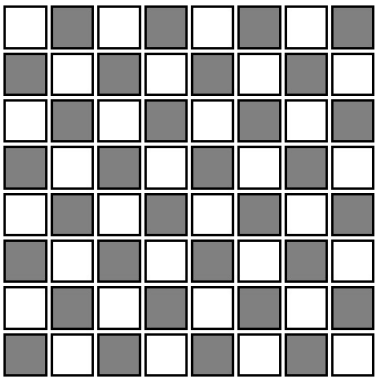
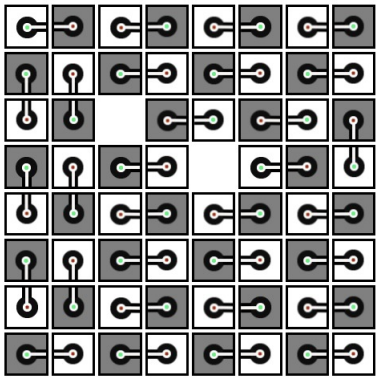


Figure 1.3: Sample Output



1.4 Advanced Mutilated Chessboard Problem

Lastly, we propose another variant of the given problem - The Advanced Mutilated Chessboard Problem. While the 'General' variant of the original problem looks at the possibility of tiling after removing any 2 squares, the proposed 'Advanced' variant looks at the following - ***"If one removes 'n' squares from an 'm $\times$ m' chessboard (where $n < m^2$), will it still be possible to completely tile the remaining chessboard with a given number of dominoes?"***

Assumptions :

The Chessboard remains connected after removing the desired squares, i.e: for every $C_{i,j}$, at least one of $C_{i+1,j}, C_{i,j+1}, C_{i-1,j}, C_{i,j-1}$, exists in the remaining Chessboard

1.5 Definitions and Nomenclature used

- ***Chessboard :*** A Chessboard is an m x n matrix, where each cell is denoted by $C_{i,j}$, where i $\in$ [0, m - 1] and j $\in$ [0, n - 1]. It may also be treated as a collection of indexed 'squares' or 'nodes'. The indexing of cells starts from (0,0) in the top-left corner, to (m,n) in the bottom-right corner. If $(i + j)$ is even, the cell is colored **white**. If the sum is odd, it is colored **black**. Unless stated otherwise, $m = n = 8$ for a standard Chessboard.

- ***Nodes / Squares :*** Every cell in a Chessboard, $C_{i,j}$ has been alternatively called a Node/ Vertex (while treating the Chessboard as a Graph) or a Square. Squares that are omitted/ removed are denoted by the set **E**, which is further divided into E_b (black) and E_w (white). In some places, **B** and **W** have been used to describe the set of Black Nodes and White Nodes, respectively.

- ***Dominoes :*** Dominoes are 2 x 1 or 1 x 2 matrices. They may also be treated as 2 x 1 or 1 x 2 subsets of a Chessboard, i.e sets of adjacent Black and White Squares.

- ***Tiling / Matching :*** A Tiling or a Matching is an arrangement of Dominoes on a Chessboard, such that no domino overlaps each other and the number of Dominoes placed is exactly half the total number of cells in the Chessboard. It is an example of an *exact cover*[9]. Alternatively, it may also be considered as a partition of the Squares in the Chessboard Set into finite number of dominoes.

- ***Neighbours :*** A node $C_{m,n}$ is said to be the neighbour of another node $C_{i,j}$ if both of them share a common edge, when represented as a Chessboard. Mathematically, it implies **exactly one** of the following possibilities:

  $$m = i + 1$$
  $$m = i - 1$$
  $$n = j + 1$$
  $$n = j - 1$$

  We also, often denote the set of Neighbours of a Node 'A' as N(A). Therefore, in the above illustration, $C_{m,n} \in N(C_{i,j})$. This is not to be confused with the number of elements in a set, which we have denoted by $|N(A)|$

- ***Collective Neighbours :*** The Collective Neighbours of all nodes in a set of Nodes $S = \bigcup_{N(k)} \forall k \in S$. We use $CN(S)$ to denote this set.

- ***Connected (Chessboard) :*** After removing desired number of squares from a Chessboard, the remaining set of squares (i.e remaining Chessboard) are (is) said to be *connected* iff for every $C_{i,j}$, at least one of $C_{i+1,j}, C_{i,j+1}, C_{i-1,j}, C_{i,j-1}$, exists in the remaining Chessboard.

2. Design of Solution

Even though the final variant of the solution we have proposed uses Hamiltonian paths, we are aware that there are numerous other ways to go about solving this. In searching for simpler yet more effective methods, we designed an alternative algorithm in python that factors the distance of the cut-outs from two reference sides. Since the indices of these can be represented as (odd/even, odd/even) we are able to reduce the question to one of sixteen possible outcomes. To tile this, we first look at the quadrilateral formed between the two cut-outs and then later at the remaining parts of the grid, placing dominoes horizontally or vertically based on the distances to the sides. However, we decided against going further with this as our current solution was more elegantly implemented. Furthermore, we use an entirely different Recursive Backtracking algorithm to generate the tiling for the advanced case, as discussed in 2.2.3.

2.1 Solution to General Mutilated Chessboard Problem

Over the course of this paper, we take a look at one possible way of solving that uses **Hamiltonian paths**. Looking at the question, one can find there exists a myriad number of unique tiling patterns that can satisfy the conditions to given to us. For example, in reference to this, prior to removing the two squares the chessboard can be tiled in a whopping $3604^2 = 1298816$ number of distinct ways [2]. This raises two very important questions, (i) How is it that such a large number can be easily reduced to 0 when such a trivial alteration is caused and (ii) Would it be easier to solve if a specific rule is applied to the way in which one goes about placing the dominoes? The answer to these questions already exist in the form of a well-known theorem, as stated below.

Gomory's Theorem (1973): If two squares of opposite colors are removed, then it is always possible to tile the remaining board with dominoes [8]. In the following sections, we explain why this theorem holds, and further discuss ways to generate tiling patterns, if the remaining Chessboard can be tiled with Dominoes.

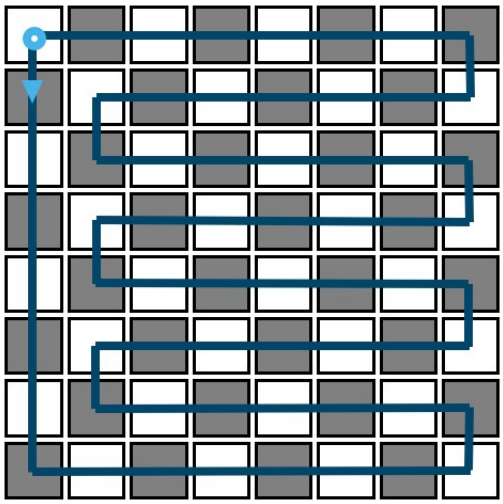
2.1.1 Coloring Proof

We use the standard coloring of a chessboard to demonstrate the solution for the General Problem. A domino, as defined previously, is a set of 2 adjacent squares of different colors, while a Chessboard is a matrix of equal number of black and white squares. Right off the bat, a necessary condition for tiling is obvious: The number of white and black squares remaining in the chessboard after removing the desired squares has to be equal. If not, they cannot be tiled by dominoes (as they themselves have an equal number of white and black squares)

[6]. It turns out, when exactly 2 squares of alternate colors are removed, this condition is sufficient for tiling, the reason for which is explained in the next sub-section. However, if more than 2 squares are removed, this condition is no longer sufficient, and is addressed later in 2.2. The same conclusion can be arrived at mathematically, when one considers the co-ordinate indexes of the cut-outs. [7]

2.1.2 Tiling using Hamiltonian Cycles

Figure 2.1: The Hamiltonian Path used in the WebApp/ Python Implementation



Hamiltonian Paths are paths that visit every vertex in a graph exactly once and do not intersect themselves. If the path ends where it starts, it may also be called a **Hamiltonian Cycle**. For the General variant of the problem, we define a Rook's journey around a Chessboard as a Hamiltonian Cycle, as shown in 2.1 (implemented as in 3.3). Obviously, the number of squares in this path is 64. When we remove exactly 2 squares from this path, we break it into 2 sub-paths. As long as each of these two sub-paths are of **even lengths**, we can fill them each with dominoes of 'length' 2. It is easy to see, that when 2 squares of opposite colors are removed, the resulting 2 sub-paths are indeed each of even length [8]. Further, we also adopt this approach to tile the remaining chessboard with dominoes as demonstrated in 3.3.

2.2 Solution to Advanced Mutilated Chessboard Problem

This section uses representations and definitions listed previous in 1.5

We observe, that a necessary condition for a set of nodes S to be exactly cover-able by dominoes is that the number of black nodes must be equal to the number of white nodes in S. This has been illustrated previously in Section 2.2.1. However, this isn't a sufficient condition for the Advanced case. In order to find a sufficient

condition, we represent the chessboard as a graph and apply Hall's theorem, as demonstrated in the following subsections.

2.2.1 Representing Chessboard as a Bipartite Graph

The Chessboard is represented as a Bipartite Graph by defining 2 disjoint and independent sets - the set of Black Nodes and the set of White Nodes. A vertex exists between an element from Black Nodes and an element from White Nodes iff they are neighbours. More precisely, it can be described as $G(B \cup W, E)$, where B = Set of Black Nodes, W = Set of White Nodes, as defined previously in 1.5.

2.2.2 Matching using Hall's Theorem

Theorem 1 (Phillip Hall's Theorem) : [4] In a finite bipartite graph with the bipartitions X and Y , there exists a matching that covers X if and only if every subset A of X is connected to at least $|A|$ vertices in Y.

Theorem 2 (Extended Hall's Theorem) : [4] In a bipartite graph with the bipartitions X and Y such that the degree of every vertex is finite, there exists a matching that covers X if and only if every finite subset A of X is connected to at least $|A|$ vertices in Y.

Hall's theorem provides a necessary and sufficient condition for the existence of a matching in the graph. In our problem, the chessboard can be seen as the bipartitions of black squares and white squares. A pair of black and white squares are connected when they are neighbours. According to the Hall's Theorem, domino tiling is possible if for all $b \subset B$ (suppose $b \cap E_b = \phi$), $|CN(b) - E_w| \geq |b|$. This essentially means that every subset of Black squares must be collectively connected to at least the equal number of White Squares. Alternatively, Every subset of Black Squares must have at least as many collective neighbours as the number of elements in the subset [5]. Thus the *sufficient condition* is summarized as follows:

1. Number of Black Squares = Number of White Squares
2. Each subset of Black Squares is collectively connected to at least as many White Squares as the number of Black Squares in the subset.

which is equivalent to the statement: **"Every subset of Squares must be collectively connected to at least as many neighbours as the elements in the subset"**.

In the following diagrams, we are presented with 2 different scenarios, both of them having the same number of white and black squares (the *necessary condition* for tiling). It is evident that 2.2 can be tiled by dominoes, but 2.3 cannot. To explain why 2.3 is not 'coverable', consider b = Set of Black Squares marked by blue

dots. These 2 Black Squares are collectively connected to only 1 White Square (marked by the green dot), i.e $|CN(b)| < |b|$. Thus, from Hall's Theorem, we conclude that it cannot be tiled, although it fulfils the *necessary condition*.

Figure 2.2: This can be tiled by Dominoes

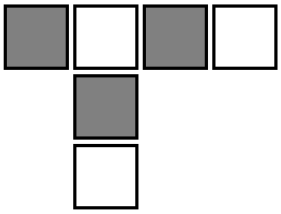
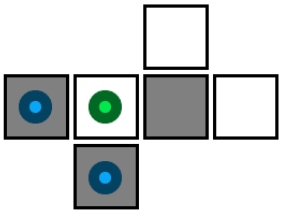


Figure 2.3: This can't be tiled by Dominoes



2.2.3 Recursive Backtracking

Once we have applied Hall's Theorem and solved the existential question, we attempt to tile a general **connected** chessboard with any number of removed squares. While this can also be solved using the previous approach involving a Hamiltonian Cycle, to find a suitable Hamiltonian Cycle for a general case would be a harder challenge. Thus we adopt a recursive approach. We start with the first Black Node and iterate through it's neighbours. For each valid neighbour (i.e: a neighbour which hasn't already been used in the current solution), we recurse to the next black node and continue the process. If we reach a stage where the current black node has no unused neighbours left, we backtrack to the previous node and select an alternate neighbour (if it exists). This algorithm is better described below. (for the implementation details in Python, see 3.3)

Algorithm 1 Recursive Backtracker for Tiling

```
1: function MATCH(index)
2:     if base 1 then return false          ▷ Current black node has no unused neighbours left
3:     if base 2 then return false          ▷ Last black node is matched
4:     for i ∈ neighbours[black[index]] do
5:         if VALID(i, solution) then       ▷ Ensures i is not used in current solution
6:             solution ← [black[index], i]  ▷ Recursive Step
7:             if MATCH(index + 1) then
8:                 return true
9:             remove [black[index], i] from solution   ▷ Back Tracking Step
```

3. Software Documentation

This section lists certain important snippets of code, some of which have been used in the WebApp and the tKinter application. While the source code is well documented in general, we provide additional short explanations for the following modules. Furthermore, a detailed software documentation for the Python and JS modules is available *here*

3.1 GUI using tKinter

tKinter is one of Python's standard GUI (Graphic User Interface) packages. The tKinter application that has been designed for this project consists of 3 primary components:

1. The Chessboard
2. Instructions
3. Button to execute tiling of dominoes (if possible)

The source code for the GUI imports a custom-made python module that we call *MutilatedChessboardSolver*, which has functionality to take in the data of the coordinates of squares that have been omitted (in the form of a list of 2-tuples such as [[1, 2], [3, 3]]) and return a list of **pairs of 2-tuples** representing the positions of dominoes that are to be placed ((i.e), if the board can be tiled, will return that it is impossible to tile the chessboard otherwise).

For example, a pair of 2-tuples of the form: [[0, 0], [1, 0]] represents a **vertical** domino placed in the **top-left corner** of the board. The way the module arrives at this tiling is based on finding a Hamiltonian Cycle in the chessboard (as in 2.1, implemented as in 3.3). How a Hamiltonian cycles results in the correct tiling is also explained in 2.1. The following snippet is the interaction between the tKinter GUI and the *MutilatedChessboardSolver* module:

```
def ok_click():
  if(clickedSquareCount != 2):
    return
  solution = mutilatedChessboardSolver.solve()
  solution_size = len(solution)
  domino = [None] * solution_size

  for i in range(solution_size):
```

```
    domino[i] = Label(bg = "grey")

    if(isHorizontal(solution[i])):
      row_val = solution[i][0][0]
      col_val = min(solution[i][0][1], solution[i][1][1])
      domino[i].configure(padx = 44, pady = 15, borderwidth = 2, bg = "grey", relief = "groove")
      domino[i].grid(row = row_val, column = col_val, columnspan = 2)

    else:
      row_val = min(solution[i][0][0], solution[i][1][0])
      col_val = solution[i][0][1]
      domino[i].configure(padx = 20, pady = 42, borderwidth = 2, bg = "grey", relief = "groove")
      domino[i].grid(row = row_val, column = col_val, rowspan = 2)


for w in root.winfo_children():
  w.configure(state="disabled")
titleLabel.configure(state="active")
if(solution_size == 0):
  fail_label = Label(root, text = "The Chessboard can't be tiled!", font = "Arial 10")
  fail_label.grid(row = 4, column = 10)

else:
  success_label = Label(root, text = "The Chessboard can be tiled and here's how!", font = "Arial
  10")
  success_label.grid(row = 4, column = 10)
button_reselect = Button(root, padx =40, text = "Reselect", command = reselect_click)
button_reselect.grid(row = 5, column = 10)
print_instructions()
button_exit = Button(root, padx =40, text = "Exit", command = exit_click)
button_exit.grid(row = 6, column = 10)
```

Listing 3.1: tKinter Interface

The *.solve()* method in the *mutilatedChessboardSolver* module returns the domino placements in the format described above.We then create an array of tKinter Labels, where each label will serve as a domino.
For each pair of 2-tuples in the solution returned by our solver module, we determine whether it is a horizontal tile or a vertical tile by comparing the row coordinates of the each of the coordinates.We then place all of these labels in their corresponding positions using the *.grid()* method in tKinter.

Of course, if the size of the list returned by the *.solve()* method was **0**, no valid tiling exists. In that case, we print the non-tileability error message on the screen, subsequently giving the user the option to try a new pair of squares (or) exit the application.Here is an example of the GUI application at work:

Figure 3.1: Board initially empty

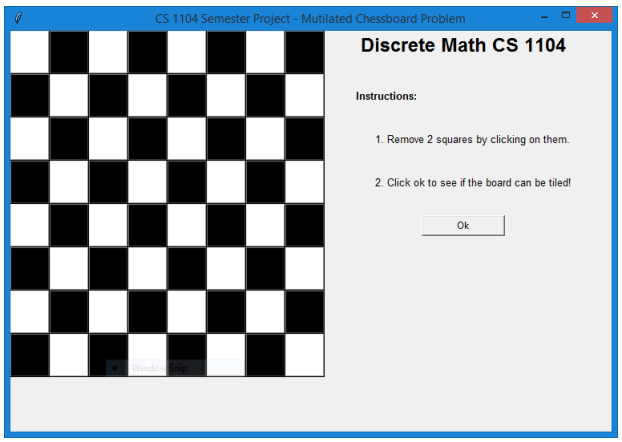


Figure 3.2: 2 holes selected

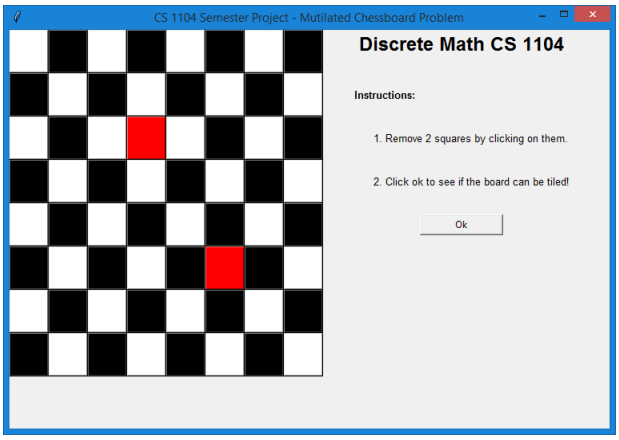
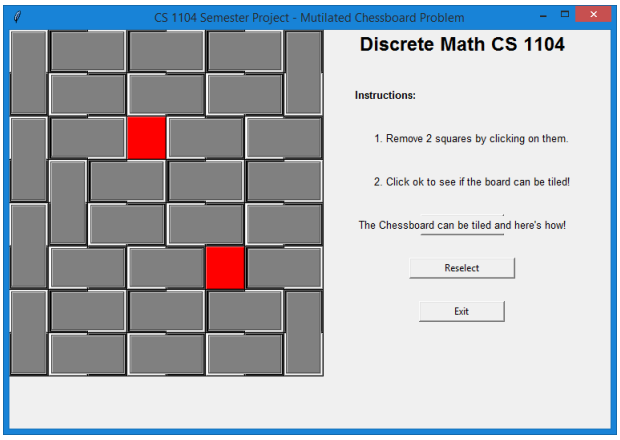


Figure 3.3: The board with dominoes placed



3.2 Vanilla JS WebApp

The following snippet ensures that the user can select exactly 2 squares on the chessboard. Once a square is selected, it is recolored and re-bordered 'white', to give the illusion of disappearance, and it's index is appended to the *selected_squares_index* array. The 3 functions that are bound to the Event Listeners are *Restore()*: restores all removes squares, *Check()*: Implements the existential check and *Tile()*: Implements the actual tiling. The final WebApp is currently hosted on this online server (url: https://general-mutilated-chessboard.000webhostapp.com/).

```
document.querySelectorAll('.black').forEach(element => {
  element.addEventListener('click', event => {
  if (clickcount<2){
```

```
    clickcount = clickcount + 1;
        let str = element.getAttribute('row').concat(" ", element.getAttribute('column'));
        selected_squares_index.push(str);
        element.style.backgroundColor = "white";
        element.style.borderColor = "white";
       } }) })
document.querySelectorAll('.white').forEach(element => {
  element.addEventListener('click', event => {
      if (clickcount<2){
        clickcount = clickcount + 1;
        let str = element.getAttribute('row').concat(" ", element.getAttribute('column'));
         selected_squares_index.push(str);
         element.style.backgroundColor = "white";
         element.style.borderColor = "white";
       } }) })
document.getElementById("reselect_button").addEventListener("click", Restore);
document.getElementById("check_button").addEventListener("click", Check);
document.getElementById("tile_button").addEventListener("click", Tile);
```

Listing 3.2: JS Code used in the WebApp

3.3 Python Modules

Python modules were independently built prior to making of the WebApp to test various functionalities. Some of the more important modules are discussed in this section. The code snippet listed below stores the Hamiltonian Path (Cycle) described in 2.1 as a 1D array of squares.

```
hamiltonian_cycle = []
    for i in range(1,n+1):
        hamiltonian_cycle.append([i,1]) # first column
    c = 1
    for i in range(n , 0, -1):
        if c % 2 == 1:
            for j in range(2, n+1):
                hamiltonian_cycle.append([i,j])
        else:
            for k in range(n, 1, -1):
                hamiltonian_cycle.append([i,k])
        c = c + 1
```

Listing 3.3: Hamiltonian Path

The following code snippet implements 2.2.1 in Python. The graph is stored as an Adjacency List, implemented through the dictionary data structure in Python. The black nodes form the 'keys' and all the

neighbours of each 'key' are stored in a list as the 'value'.

```
def chessToGraph():
    bipartite_graph = {}  # this is a dictionary
    # '00' denotes an omitted square
    for i in range(row):
        for j in range(col):
            if chessboard[i][j]!= '00':
                neighbours = set()  # this is a set
                if j > 0 and chessboard[i][j - 1]!='00':
                    neighbours.add(chessboard[i][j - 1])
                if j < col-1 and chessboard[i][j + 1]!='00':
                    neighbours.add(chessboard[i][j + 1])
                if i > 0 and chessboard[i - 1][j]!='00':
                    neighbours.add(chessboard[i - 1][j])
                if i < row-1 and chessboard[i + 1][j]!='00':
                    neighbours.add(chessboard[i + 1][j])
                bipartite_graph[chessboard[i][j]] = list(neighbours) # saved as a list in graph
                del neighbours

    return bipartite_graph
```

Listing 3.4: Representing the Chessboard as a Bipartite Graph using Adjacency Lists

The following code snippet implements the algorithm 1 described previously in 2.2.3 in Python. A set called *selected_white_nodes* is used to keep a track of all the white nodes used in current solution set, in order to implement the *validate()* function.

```
def match(b_index):
    count = 0
    for j in range(0, len(graph[black_nodes[b_index]]) ):
        if validate(graph[black_nodes[b_index]][j]):
            count = count + 1
    if count == 0:return False # base case 1: no white neighbours left

    if(b_index == len(black_nodes) - 1):
        for k in range(0, len(graph[black_nodes[b_index]]) - 1):
            if validate(graph[black_nodes[b_index]][k]):
                solution.append([black_nodes[b_index], graph[black_nodes[b_index]][k]])
                return True # base case 2: on completing all black nodes

        return False

    for i in range(0, len(graph[black_nodes[b_index]])):
```

```
        current_white_neighbour = graph[black_nodes[b_index]][i]
        if validate(graph[black_nodes[b_index]][i]): # check if white neighbour has been selected
            solution.append([black_nodes[b_index], current_white_neighbour])
            selected_white_nodes.add(graph[black_nodes[b_index]][i])

            if match(b_index + 1): # recurse to next black_node
                return True

            solution.remove([black_nodes[b_index], current_white_neighbour]) # back-tracking
            selected_white_nodes.remove(graph[black_nodes[b_index]][i])  # restore white neighbour
```

Listing 3.5: Recursive Backtracking Algorithm Implementation

3.4 Results and Future Improvements

The WebApp for the General Mutilated Chessboard Problem, implemented in VanillaJS is hosted here.

For an updated version of this WebApp, we are currently in the process of building a Flask WebApp, with additional capabilities of visualizing the Advanced Mutilated Chessboard Problem as well.

One issue we ran into for the Advanced variant of the problem, was the existential check using Hall's Theorem. While we have successfully implemented the algorithm, it runs at a time complexity of $O(2^{n^2})$, where n is the length of the chessboard. This leads to a bottle-neck, which can't be resolved, as the existential check involves an iteration through the entire Power Set of Black Nodes.

This can, however, be bypassed, by deploying an alternative check using the Recursive Backtracker algorithm we have designed for this project (1). If it returns an empty list OR a list does not cover the entire remaining chessboard, we can conclude that the subset of the chessboard in question cannot be tiled, as by design, the Backtracker will traverse through all possible tilings before such an output.

Bibliography

[1] Jeremy Avigad. *A formalization of the mutilated chessboard problem*. URL: http://www.andrew.cmu.edu/user/avigad/Papers/mutilated.pdf. (accessed: 22.04.2020).

[2] Norman Do. *Lectures on topics in geometry*. URL: http://users.monash.edu/~normd/documents/MATH-348-lecture-41.pdf.

[3] Martin Gardner. *My Best Mathematical and Logic Puzzles*. Dover, 1994. ISBN: 0-486-2815-23.

[4] Phillip Hall. "On Representatives of Subsets". In: *Journal of London Mathematical Society* 10.1 (1935), pp. 26–30. DOI: doi:10.1112/jlms/s1-10.37.26.

[5] Supanat Kamutue. *Domino-tiling Problem*. URL: https://math.mit.edu/research/undergraduate/spur/documents/2013Kamtue.pdf.

[6] Benjamin Kiesl Marijn J.H. Heule and Armin Biere. *Clausal Proofs of Mutilated Chessboards*. URL: https://www.cs.utexas.edu/~marijn/publications/mchess.pdf. (accessed: 22.04.2020).

[7] Lawrence C. Paulson. *A simple formalization and proof for the mutilated chessboard*. URL: http://www.andrew.cmu.edu/user/avigad/Papers/mutilated.pdf.

[8] John J. Watkins. *Across the board: the mathematics of chessboard problems*. 2004. (accessed: 22.04.2020).

[9] Wikipedia. *Exact Cover*. URL: https://en.wikipedia.org/wiki/Exact_cover. (accessed: 22.04.2020).

Individual Contributions

Nishant Mahesh - tKinter GUI Design and Implementation - Windows and Mac, GUI + general tiling algorithm integration, Report.

Parul Joshi - Hamiltonian Path Implementation, Tiling Algorithm Implementation in Python, Design of alternate solutions, Report.

Sidhartha Prabakar Nadathur - JS WebApp Front End Design, Proof of Correctness of General Solution, Alternate solution algorithm design/implementation, Report.

Soham De - Advanced Variant Implementation in Python, Solution/Algorithm Design for all variants (General, Advanced) and Implementation of associated sub problems (Tilings, Checkings etc). WebApp logic implementation using VanillaJS, Overall Debugging, Software Documentation, Report.