Python Project Report

Minesweeper with SAT Solver

01286121 Computer Programming

Software Engineering Program

By

68011354 Jiramet Harnjan

Python Project

Minesweeper with SAT Solver

Project Description

This is a program that recreates the game ***Minesweeper***. "***Minesweeper*** is a logic puzzle video game genre. The game features a grid of clickable tiles, with hidden "mines" dispersed throughout the board. The objective is to clear the board without detonating any mines, with help from clues about the number of neighboring mines in each field.

Cells have three states: unopened, opened and flagged. An unopened cell is blank and clickable, while an opened cell is exposed. Flagged cells are those marked by the player to indicate a potential mine location.

A player selects a cell to open it. If a player opens a mined cell, the game ends. Otherwise, the opened cell displays either a number, indicating the number of mines vertically, horizontally or diagonally adjacent to it, or a blank tile (or "0"), and all adjacent non-mined cells will automatically be opened. Players can also flag a cell, visualised by a flag being put on the location, to denote that they believe a mine to be in that place. Flagged cells are still considered unopened, and a player can click on them to open them"[1]

This program also features a SAT solver that automatically open cells and flag mines.

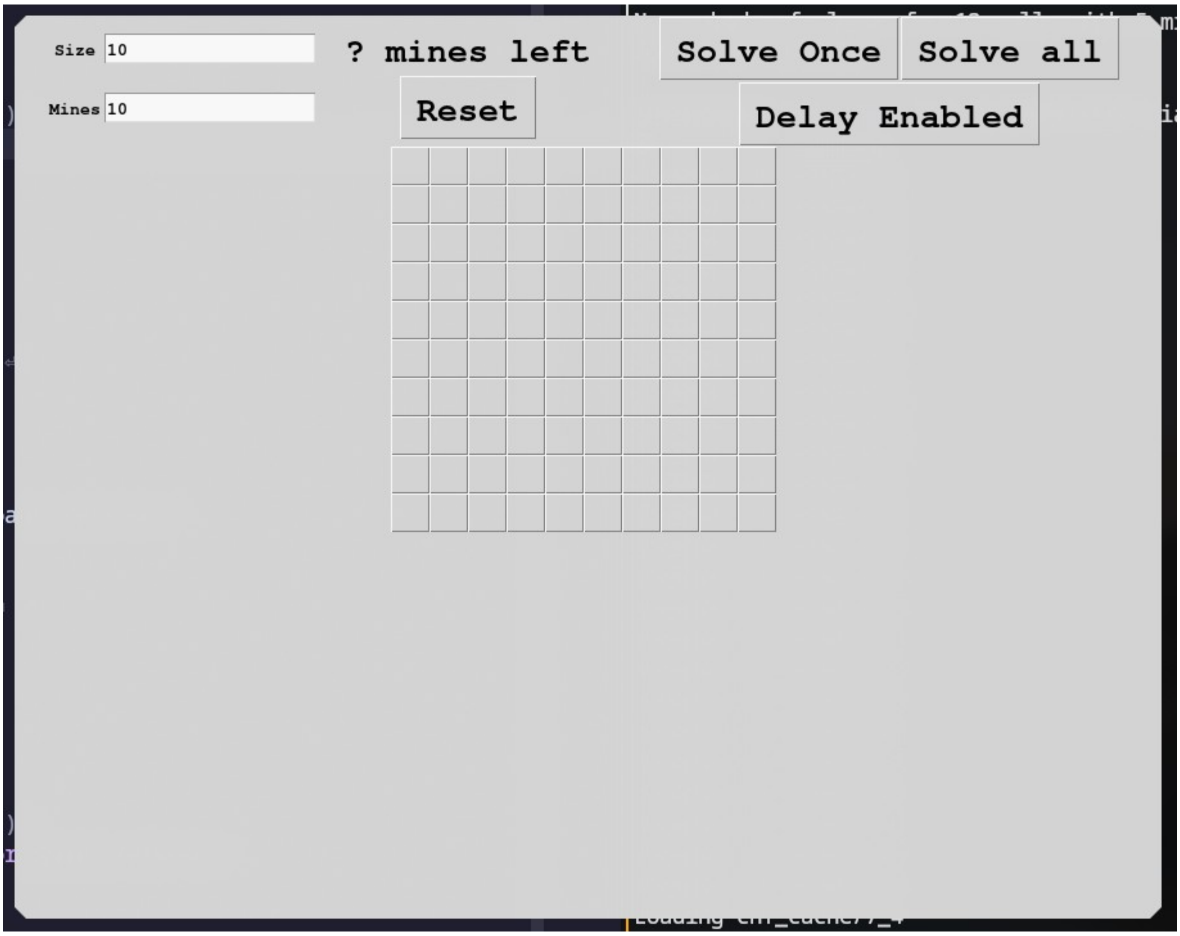
The full code can be found here[2].

1 https://en.wikipedia.org/wiki/Minesweeper_(video_game)

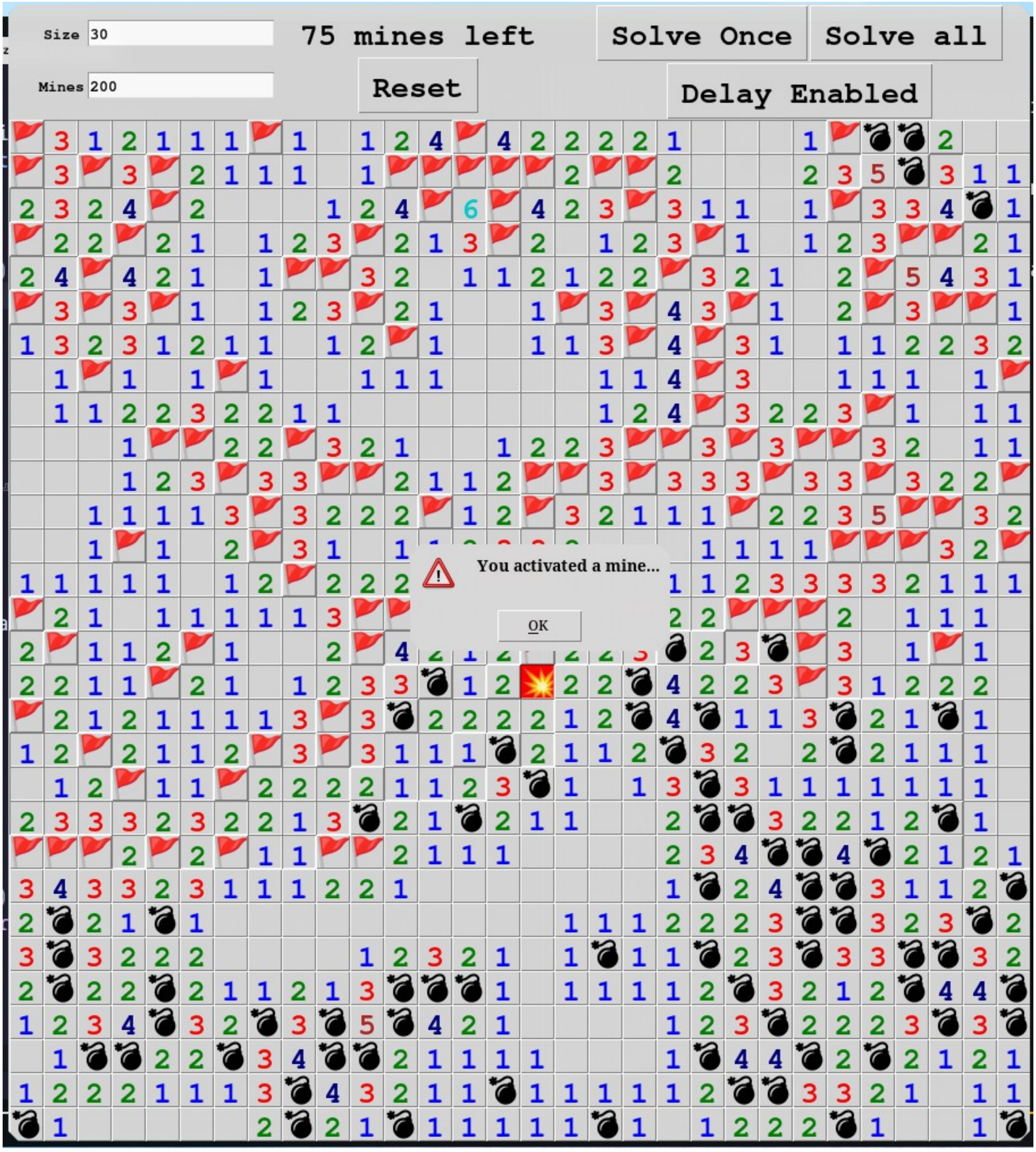
2 https://github.com/peach-on-the-way/tkinter-minesweeper

Screen Capture of your Program

Screen1



Screen2


Size 30
Mines 200
75 mines left
Solve Once
Solve all
Reset
Delay Enabled
You activated a mine...
OK

Screen 3

Size 30
Mines 200
0 mines left
Solve Once
Solve all
Reset
Delay Disabled
All mines found!
OK

Project Source Codes

game.py:

```
import tkinter as tk
from tkinter import messagebox
import random
import solver
import time

font = ("Courier", 20, "bold")
small_font = ("Courier", 10, "bold")

dpos = [(1, 0), (-1, 0), (0, 1), (0, -1), (1, 1), (-1, -1), (1, -1), (-1, 1)]
cell_number_colors = [
    "blue",
    "green",
    "red",
    "navy",
    "firebrick",
    "darkturquoise",
    "black",
    "darkgray",
]

cell_content_bomb = "💣"
cell_content_bomb_exploded = "💥"
cell_content_flag = "🚩"
cell_content_flag_wrong = "❌"

class Board(tk.Frame):
    def __init__(self, master, board_size, mines, cell_size):
        super().__init__(master)
        self.board_size = board_size
        self.mines = mines
        self.cell_size = cell_size
        self.board_generated = False
        self.initialize_cell_buttons()
        self.interaction_enabled = True

    def reset(self):
        for column in self.cell_buttons:
            for button in column:
                button.destroy()
        self.board_generated = False
        self.initialize_cell_buttons()

    def initialize_board_data(self):
        self.exploded = False
        self.cells_revealed = set()
        self.cells_flagged_locations = set()
        self.cells_grid_flagged = [[False for i in range(self.board_size)] for i in range(self.board_size)]
        self.cells_grid_info = [[0 for i in range(self.board_size)] for i in range(self.board_size)]
```

```
    self.cells_unrevealed = { (x, y) for x in range(self.board_size) for y in range(self.board_size)}
    self.mine_locations = set()

def setup_grid_info(self):
    for x, y in self.mine_locations:
        for dx, dy in dpos:
            test_x = x + dx
            test_y = y + dy
            if not 0 <= test_x < self.board_size or not 0 <= test_y < self.board_size:
                continue
            if not type(self.cells_grid_info[test_x][test_y]) is int:
                continue

            self.cells_grid_info[test_x][test_y] += 1

def generate_random_board(self):
    self.initialize_board_data()
    for _ in range(self.mines):
        mine_x = random.randint(0, self.board_size - 1)
        mine_y = random.randint(0, self.board_size - 1)
        self.cells_grid_info[mine_x][mine_y] = '*'
        self.mine_locations.add((mine_x, mine_y))
    self.setup_grid_info()

def setup_custom_board(self, mine_locations=None, cells_to_reveal=None, cells_to_flag=None):
    self.initialize_board_data()
    if mine_locations:
        self.mine_locations = set(mine_locations)
        for x, y in self.mine_locations:
            self.cells_grid_info[x][y] = '*'
    self.setup_grid_info()
    if cells_to_reveal:
        for pos in cells_to_reveal:
            board.reveal_cell(*pos)
    if cells_to_flag:
        for pos in cells_to_flag:
            board.cell_flag(*pos)

def initialize_cell_buttons(self):
    self.cell_buttons = []
    for x in range(self.board_size):
        column = []
        for y in range(self.board_size):
            # Make button square
            cell_button_frame = tk.Frame(
                self,
                width=self.cell_size,
                height=self.cell_size,
            )
            cell_button_frame.grid_propagate(False) # Prevent the frame from resizing
            cell_button_frame.grid_columnconfigure(0, weight=1) # Allows the button to fill the frame
            cell_button_frame.grid_rowconfigure(0, weight=1)
            cell_button_frame.grid(column=x, row=y, sticky=tk.NSEW)

            cell_button = tk.Button(
                cell_button_frame,
```

```
                font=font,
                highlightthickness=0,
            )
            cell_button.configure(command=self.on_cell_left_clicked(x, y))
            cell_button.bind("<Button-3>", self.on_cell_right_clicked(x, y))
            cell_button.grid(column=0, row=0, sticky=tk.NSEW)
            column.append(cell_button_frame)
        self.cell_buttons.append(column)

def button_at(self, x, y):
    if not (0 <= x < self.board_size \
        and 0 <= y < self.board_size):
        return None

    return self.cell_buttons[x][y].winfo_children()[0]

def pos_inside_board(self, x, y):
    return 0 <= x < self.board_size and 0 <= y < self.board_size

def cell_is_revealed(self, x, y):
    return (x, y) in self.cells_revealed

def cell_is_empty(self, x, y):
    return self.cells_grid_info[x][y] == 0

def cell_is_number(self, x, y):
    val = self.cells_grid_info[x][y]
    if not isinstance(val, int):
        return False
    return 1 >= val >= 8

def cell_is_bomb(self, x, y):
    return self.cells_grid_info[x][y] == "*"

def cell_is_flagged(self, x, y):
    return self.cells_grid_flagged[x][y]

def show_cell_button(self, x, y):
    if self.cell_is_empty(x, y):
        if self.cell_is_flagged(x, y):
            self.button_at(x, y).config(
                text=cell_content_flag,
            )
    elif self.cell_is_bomb(x, y):
        if self.cell_is_flagged(x, y):
            self.button_at(x, y).config(
                text=cell_content_flag,
            )
        else:
            color = "black"
            self.button_at(x, y).config(
                text=cell_content_bomb,
                disabledforeground=color,
                foreground=color
            )
    else:
```

```
            if self.cell_is_flagged(x, y):
                self.button_at(x, y).config(
                    text=cell_content_flag_wrong,
                    foreground="red",
                    disabledforeground="red",
                )
            else:
                color = cell_number_colors[self.cells_grid_info[x][y] - 1]
                content = str(self.cells_grid_info[x][y])
                self.button_at(x, y).config(
                    text=content,
                    disabledforeground=color,
                    foreground=color
                )

    def reveal_cell(self, x, y):
        if (x, y) in self.cells_revealed:
            return

        self.cells_unrevealed.remove((x, y))
        self.cells_revealed.add((x, y))
        if self.cell_is_empty(x, y) and not self.cell_is_flagged(x, y):
            for dx, dy in dpos:
                testx = x + dx
                testy = y + dy
                if 0 <= testx < self.board_size \
                    and 0 <= testy < self.board_size \
                    and type(self.cells_grid_info[testx][testy]) is int:
                    self.reveal_cell(testx, testy)

        self.button_at(x, y).config(
            relief=tk.SUNKEN,
            state=tk.DISABLED,
        )

        if self.cell_is_empty(x, y):
            pass
        elif self.cell_is_bomb(x, y):
            self.button_at(x, y).config(
                text=cell_content_bomb_exploded,
                disabledforeground="black",
                background="red"
            )
        else:
            self.button_at(x, y).config(
                text=str(self.cells_grid_info[x][y]),
                disabledforeground=cell_number_colors[self.cells_grid_info[x][y] - 1]
            )
        self.cells_revealed.add((x, y))

    def reveal_all(self):
        for x in range(self.board_size):
            for y in range(self.board_size):
                if (x, y) in self.cells_revealed:
                    continue
```

```
            self.show_cell_button(x, y)
            self.button_at(x, y).config(
                state=tk.DISABLED
            )

    def find_number_cells_adjacent_to_unrevealed_cell(self):
        cells = set()
        for mine_x, mine_y in self.mine_locations:
            for dx, dy in dpos:
                testx = mine_x + dx
                testy = mine_y + dy
                if not self.pos_inside_board(testx, testy) \
                    or not self.cell_is_revealed(testx, testy) \
                    or (testx, testy) in self.mine_locations:
                    continue

                number_cell_x, number_cell_y = testx, testy
                for dx, dy in dpos:
                    testx = number_cell_x + dx
                    testy = number_cell_y + dy
                    if not self.cell_is_revealed(testx, testy):
                        continue
                    cells.add((number_cell_x, number_cell_y))
                    break
        return cells


    def cell_flag_without_event(self, x, y):
        if (x, y) in self.cells_revealed or self.exploded:
            return
        self.button_at(x, y).config(text=cell_content_flag, state=tk.DISABLED)
        self.cells_grid_flagged[x][y] = True
        self.cells_flagged_locations.add((x, y))


    def cell_flag(self, x, y):
        self.cell_flag_without_event(x, y)
        self.event_generate("<<CellFlagged>>")

    def cell_unflag(self, x, y):
        if (x, y) in self.cells_revealed or self.exploded:
            return
        self.button_at(x, y).config(text="", state=tk.NORMAL)
        self.cells_grid_flagged[x][y] = False
        self.cells_flagged_locations.remove((x, y))
        self.event_generate("<<CellUnflagged>>")


    def on_cell_left_clicked(self, x, y):
        def on_cell_left_clicked_inner():
            if not self.interaction_enabled:
                return

            # Attempts til giving up generating good start
            attempts = 1000
            while not self.board_generated:
```

```
            self.generate_random_board()
            attempts -= 1
            if not self.cell_is_empty(x, y) and attempts > 0:
                continue
            self.board_generated = True
            self.event_generate("<<BoardGenerated>>")
            break

        if self.cells_grid_info[x][y] == "*":
            self.reveal_cell(x, y)
            self.reveal_all()
            self.event_generate("<<CellExploded>>")
            self.exploded = True
        else:
            self.reveal_cell(x, y)
            self.event_generate("<<CellRevealed>>")

    return on_cell_left_clicked_inner

  def on_cell_right_clicked(self, x, y):
    def on_cell_right_clicked_inner(event):
      if not self.interaction_enabled:
        return
      if not self.board_generated \
        or (x, y) in self.cells_revealed \
        or self.exploded:
        return
      if self.cells_grid_flagged[x][y]:
        self.cell_unflag(x, y)
      else:
        self.cell_flag(x, y)
    return on_cell_right_clicked_inner


def update_mines_left_label(args):
  mines_left = len(board.mine_locations) - len(board.cells_flagged_locations)
  mines_left_str.set(f"{mines_left} mines left")

def check_board_completed(args):
  if len(board.mine_locations) == len(board.cells_flagged_locations) \
    and len(board.cells_revealed) == board.board_size * board.board_size - len(board.mine_locations):
    messagebox.showinfo("Game completed", "All mines found!")
    board.interaction_enabled = False

def exploded(args):
  messagebox.showwarning("Game over", "You activated a mine...")

def reset():
  try:
    board.board_size = int(board_size_var.get())
    board.mines = int(mines_var.get())
  except:
    pass
  board.reset()
  mines_left_str.set(f"? mines left")
  board.interaction_enabled = True
```

```python
def tksleep(t):
    'emulating time.sleep(seconds)'
    ms = int(t*1000)
    root = tk._get_default_root()
    var = tk.IntVar(root)
    root.after(ms, lambda: var.set(1))
    root.wait_variable(var)


def solve_once():
    if not board.board_generated or board.exploded:
        return

    result = solver.solve_once(board)
    if result == None:
        print("Cannot solve")
        return

    # Nice animation
    delay = 0.005
    for cell, is_mine in result.items():
        if not board.board_generated or board.exploded:
            break

        if enable_delay.get():
            tksleep(delay)
        x, y = cell
        if is_mine:
            board.cell_flag_without_event(x, y)
        else:
            if not board.exploded:
                board.reveal_cell(x, y)
            if board.cells_grid_info[x][y] == "*":
                board.reveal_all()
                board.exploded = True
                break

    if board.exploded:
        board.event_generate("<<CellExploded>>")
    board.event_generate("<<CellFlagged>>")

def solve_all():
    while True:
        result = solver.solve_once(board)
        if result == None:
            print("Solve all finished")
            return
        if not board.board_generated or board.exploded:
            return

        delay = 0.005
        for cell, is_mine in result.items():
            if enable_delay.get():
                tksleep(delay)
            x, y = cell
```

```
            if is_mine:
                board.cell_flag_without_event(x, y)
            else:
                if not board.exploded:
                    board.reveal_cell(x, y)
                if board.cells_grid_info[x][y] == "*":
                    board.reveal_all()
                    board.exploded = True

        if board.exploded:
            board.event_generate("<<CellExploded>>")
        board.event_generate("<<CellFlagged>>")

def toggle_solver_delay():
    if enable_delay.get():
        enable_delay.set(False)
        toggle_delay_text.set("Delay Disabled")
    else:
        enable_delay.set(True)
        toggle_delay_text.set("Delay Enabled")

# Preemptively load caches for common cases
for neighbors_count in range(1, 8):
    for mines_count in range(1, neighbors_count + 1):
        solver.load_cache(neighbors_count, mines_count)

root = tk.Tk()

top_bar = tk.Frame()
top_bar.grid(column=0, row=0, ipadx=50)
top_bar.grid_columnconfigure(0, weight=1)
top_bar.grid_columnconfigure(1, weight=1)

gameplay_ui = tk.Frame(top_bar)
gameplay_ui.grid(column=0, row=0)
gameplay_ui.grid_rowconfigure(0, weight=1)
gameplay_ui.grid_rowconfigure(1, weight=1)

mines_left_str = tk.StringVar()
mines_left_str.set("? mines left")
mines_left_label = tk.Label(gameplay_ui, textvariable=mines_left_str, font=font)
mines_left_label.grid(column=2, row=0)

reset_button = tk.Button(gameplay_ui, text="Reset", font=font, command=reset)
reset_button.grid(column=2, row=1)

def board_size_entry_on_invalid():
    board_size_entry.delete(0, "end")
    board_size_entry.insert(0, board.board_size)

board_size_var = tk.StringVar()
board_size_var.set(10)
board_size_entry = tk.Entry(
    gameplay_ui,
    textvariable=board_size_var,
    font=small_font,
```

```
    validate="focusout",
    validatecommand=(
        root.register(lambda v: v.isdigit()),
        "%P"
    ),
)
    invalidcommand=board_size_entry_on_invalid
board_size_entry.grid(column=1, row=0, padx=(0, 20))
size_label = tk.Label(gameplay_ui, text="Size", font=small_font)
size_label.grid(column=0, row=0)

def mines_entry_on_invalid():
    mines_entry.delete(0, "end")
    mines_entry.insert(0, str(board.mines))

mines_var = tk.StringVar()
mines_entry = tk.Entry(
    gameplay_ui,
    textvariable=mines_var,
    font=small_font,
    validate="focusout",
    validatecommand=(
        root.register(lambda v: v.isdigit() and int(v) < int(board_size_var.get()) * int(board_size_var.get())),
        "%P"
    ),
    invalidcommand=mines_entry_on_invalid,
)
mines_entry.insert(0, "10")
mines_entry.grid(column=1, row=1, padx=(0, 20))
mines_label = tk.Label(gameplay_ui, text="Mines", font=small_font)
mines_label.grid(column=0, row=1)

solver_ui = tk.Frame(top_bar)
solver_ui.grid(column=1, row=0)
solver_ui.grid_rowconfigure(0, weight=1)
solver_ui.grid_rowconfigure(1, weight=1)

solve_once_button = tk.Button(solver_ui, text="Solve Once", font=font, command=solve_once)
solve_once_button.grid(column=0, row=0)

solve_all_button = tk.Button(solver_ui, text="Solve all", font=font, command=solve_all)
solve_all_button.grid(column=1, row=0)

toggle_delay_text = tk.StringVar()
toggle_delay_text.set("Delay Enabled")
enable_delay = tk.BooleanVar()
enable_delay.set(True)
delay_button = tk.Button(solver_ui, textvariable=toggle_delay_text, font=font,
command=toggle_solver_delay)
delay_button.grid(column=0, row=1, columnspan=2)

board = Board(root, 10, 10, 30)
board.grid(column=0, row=1)

# test_board(board)
```

```
board.bind("<<BoardGenerated>>", update_mines_left_label)
board.bind("<<CellFlagged>>", update_mines_left_label)
board.bind("<<CellUnflagged>>", update_mines_left_label)

board.bind("<<BoardGenerated>>", check_board_completed, True)
board.bind("<<CellFlagged>>", check_board_completed, True)
board.bind("<<CellUnflagged>>", check_board_completed, True)
board.bind("<<CellRevealed>>", check_board_completed, True)

board.bind("<<CellExploded>>", exploded)

root.mainloop()
```

solver.py:

```python
from pysat.formula import CNF
from pysat.solvers import Solver

dpos = [(1, 0), (-1, 0), (0, 1), (0, -1), (1, 1), (-1, -1), (1, -1), (-1, 1)]

cache_directory = "cnf_cache"

cache = {}

# Return True if cache exists, False otherwise.
def load_cache(cells_count, mines_count):
    cnf = []
    path = f"{cache_directory}/{cells_count}_{mines_count}"
    try:
        f = open(path)
    except OSError as e:
        return False

    print(f"Loading {path}")
    for line in f:
        if line.startswith("="):
            break
        clause = filter(lambda x: x != '', line.strip().split(" "))
        clause = map(int, clause)
        cnf.append(list(clause))
    cache[(cells_count, mines_count)] = cnf
    return True


def get_cached_cnf(cells_count, mines_count):
    if not (cells_count, mines_count) in cache:
        has_cache = load_cache(cells_count, mines_count)
        if not has_cache:
            return None

    return cache[(cells_count, mines_count)]

# The translate the atoms within local CNF from the cache
# to atoms that uniquely represent the cells
def translate_cached_cnf(cnf, cell_ids, mines_count):
    new_cnf = []
    for clause in cnf:
        new_clause = []
        for atom in clause:
```

```
            if atom < 0:
                new_atom = -cell_ids[(-atom) - 1]
            else:
                new_atom = cell_ids[atom - 1]
            new_clause.append(new_atom)
        new_cnf.append(new_clause)
    return new_cnf

def get_cnf(cell_names, mines_count):
    cnf = get_cached_cnf(len(cell_names), mines_count)
    if cnf == None:
        return None
    return translate_cached_cnf(cnf, cell_names, mines_count)


def solve_once(board):
    # Converts (x, y) positions to the integer format accepted by pysat
    def pos_to_atom(x, y, board_width):
        index = y * board_width + x
        return index + 1

    def atom_to_pos(index, board_width):
        index = index - 1
        return (index % board_width, index // board_width)

    adjacent_cnf = CNF()

    number_cells_to_check =
board.find_number_cells_adjacent_to_unrevealed_cell()
    for x, y in number_cells_to_check:
        flagged_neighbor = set()
        unrevealed_neighbor = set()
        for dx, dy in dpos:
            testx = x + dx
            testy = y + dy
            if not board.pos_inside_board(testx, testy):
                continue
            if board.cell_is_flagged(testx, testy):
                flagged_neighbor.add((testx, testy))
            if not board.cell_is_revealed(testx, testy):
                unrevealed_neighbor.add((testx, testy))

        # Or also called the number for the number cell
        mine_nearby_count = board.cells_grid_info[x][y]

        unrevealed_neighbor = [pos_to_atom(x, y, board.board_size) for x, y in
unrevealed_neighbor]
        adjacent_cnf_clauses = get_cnf(unrevealed_neighbor, mine_nearby_count)
        if adjacent_cnf_clauses == None:
            print("A CNF Clause is unavailable")
```

```
            return None
        adjacent_cnf.extend(adjacent_cnf_clauses)

    solver = Solver(bootstrap_with=adjacent_cnf)

    # All the cells that both the agent and user has flagged is assumed
    # to be mines.
    cells_flagged_assumed_mine = { pos_to_atom(x, y, board.board_size) for x,
y in board.cells_flagged_locations }

    solver.solve(assumptions=list(cells_flagged_assumed_mine))
    if not solver.solve():
        return None

    # Exhaustive checks
    # If an assignment of a cell is negated and the solution is unsatisfiable,
    # then that cell must not be changed, e.g.
    #     1. The first model for a specific cell is false
    #     2. Try solve again but assumed to be true
    #     3. If the result is unsatisfiable -> The cell must be false, e.g. the
cell must be empty
    #     3. If the result is satisfiable -> The cell could either be empty or a
mine
    # or
    #     1. The first model for a specific cell is true
    #     2. Try solve again but assumed to be false
    #     3. If the result is unsatisfiable -> The cell must be true, e.g. the
cell must be a mine
    #     3. If the result is satisfiable -> The cell could either be empty or a
mine

    safe_model = set()

    initial_model = solver.get_model()
    for initial_atom in initial_model:
        if initial_atom in cells_flagged_assumed_mine:
            continue
        satisfiable = solver.solve(assumptions=set([-initial_atom]) |
cells_flagged_assumed_mine)
        if not satisfiable:
            safe_model.add(initial_atom)

    # Attempting to solve from known leftover mines.
    if len(safe_model) == 0:
        print("Solving leftover")
        all_unknown_cells = board.cells_unrevealed -
board.cells_flagged_locations
        all_unknown_cells = { pos_to_atom(x, y, board.board_size) for x, y in
all_unknown_cells }
        mines_left = len(board.mine_locations) -
len(board.cells_flagged_locations)
```

```
        mines_left_cnf = get_cnf(list(all_unknown_cells), mines_left)
        if mines_left_cnf == None:
            print(f"No cached cnf clause for {len(all_unknown_cells)} cells
with {mines_left} mines exists")
            return None


        mines_left_cnf = CNF(from_clauses=mines_left_cnf)
        # Combines with previous adjacent CNF
        mines_left_cnf.extend(adjacent_cnf)

        solver = Solver(bootstrap_with=mines_left_cnf)
        cells_flagged_assumed_mine = { pos_to_atom(x, y, board.board_size) for
x, y in board.cells_flagged_locations }
        solver.solve(assumptions=list(cells_flagged_assumed_mine))
        if not solver.solve():
            # No valid solution possible without guessing
            return None

        # Exhaustive checks, same as previously
        initial_model = solver.get_model()
        for initial_atom in initial_model:
            if initial_atom in cells_flagged_assumed_mine:
                continue
            satisfiable = solver.solve(assumptions=set([-initial_atom]) |
cells_flagged_assumed_mine)
            if not satisfiable:
                safe_model.add(initial_atom)

        # Rarely a board can be solved from leftover so an annoucement is
exciting
        if len(safe_model) > 0:
            print("Solution found from leftover")

    if len(safe_model) == 0:
        # No valid solution possible without guessing
        return None

    # Safe solution found
    result = {}
    for atom in safe_model:
        if atom > 0:
            is_mine = True
            pos = atom_to_pos(atom, board.board_size)
        else:
            is_mine = False
            pos = atom_to_pos(-atom, board.board_size)
        result[pos] = is_mine
    return result
```

compute_cnf.py:

```
# Converts DNF to CNF
#
# Some preemptive elimination is necessary to be able to compute
# the CNF clauses within reasonble amount of memory and time.
#
# It's the bottleneck.
def convert_normal_form(form):
    def inner(first_clause, tail_clauses):
        # Use set to eliminate a or a within a clause.
        if len(tail_clauses) == 0:
            return set(frozenset(set([x])) for x in
first_clause)

        form = set()
        for tail_atoms in inner(tail_clauses[0],
tail_clauses[1:]):
            for first_atom in first_clause:
                # Eliminate clause that contains ~a or a
                if -first_atom in tail_atoms:
                    continue
                form.add(frozenset(set([first_atom]) |
tail_atoms))
        return form

    return inner(form[0], form[1:])

# Remove duplicate clause to reduce memory usage.
def dedup(clauses):
    new_clauses = set()
    for clause in clauses:
        clause = set(clause)
        eliminate_clause = False
        for atom in clause:
            # Eliminate clauses that contains ~a or a again just
in case.
            if atom < 0 and -atom in clause:
                eliminate_clause = True
```

```
                break
        if eliminate_clause:
            continue
        new_clauses.add(frozenset(clause))
    return [list(clause) for clause in new_clauses]

# Sort a bit so it's easier to inspect some patterns in case we
were able
# to generate CNF clause directly without going through the DNF-
>CNF conversion
# in the future.
def sort(clauses):
    for clause in clauses:
        clause.sort(key=lambda a: -a if a < 0 else a)

    def key(clause):
        l = len(clause) * 10000
        # if len(clause[0]) == 1:
        #     l += ord('1') - ord(clause[0]) + 1
        for i, atom in enumerate(clause):
            if atom < 0:
                l -= 2**i
        return l

    clauses.sort(reverse=True, key=key)

# Attempts to generate combinations of possible mine locations in
DNF clause.
def generate_mine_dnf_clauses(cells, mines_count):
    def inner(current_clause, cells, mines_count):
        if len(cells) == 0:
            clauses = set([frozenset(current_clause)])
            return clauses

        if mines_count > 0:
            # There are more mine positions we can combine.
            clauses = set()
            for x in cells:
                tail_clauses = inner(
                    current_clause | set([x]),
                    cells - set([x]),
                    mines_count - 1,
```

```
                )
                clauses = clauses | tail_clauses
            return clauses
        else:
            # Mines positions has been taken over by other cells
            # Return the rest negated, e.g. not a mine.
            clause = current_clause | set([-x for x in cells])
            return set([frozenset(clause)])

    clauses = inner(set(), cells, mines_count)
    return [list(clause) for clause in clauses]

def generate(cell_count, mines_count):
    cells = [i for i in range(1, 1 + cell_count)]

    dnf = generate_mine_dnf_clauses(set(cells), mines_count)

    cnf = convert_normal_form(dnf)
    cnf = dedup(cnf)
    sort(cnf)

    with open(f"cache/{cell_count}_{mines_count}", "w",
encoding="utf-8") as f:
        for clause in cnf:
            for atom in clause:
                f.write(f"{atom:>2} ")
            f.write(f"\n")
         # Specifiy an ending marker in case the programs stops
midway for whatever reasons
         # to prevent from incomplete CNF being in used
        f.write(f"=")

    print(f"{cell_count} {mines_count} finished")

from concurrent.futures import ProcessPoolExecutor

# with ProcessPoolExecutor(max_workers=16) as e:
#     for cell_count in range(1, 13):
#         for mines_count in range(0, cell_count + 1):
#             e.submit(generate, cell_count, mines_count)

for cell_count in range(1, 13):
```

```
    for mines_count in range(0, cell_count + 1):
        generate(cell_count, mines_count)
```