UNIVERSITY OF SCIENCE
VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY

FACULTY OF INFORMATION TECHNOLOGY

COURSE: INTRODUCTION TO AI

LAB 01 - GEM MINER

Instructor:
Nguyen Hai Dang
Nguyen Tran Duy Minh
Nguyen Ngoc Thao
Nguyen Thanh Tinh

Student:
Nguyen Huu Anh Tri
(23127130 - 23CLC08)

Ho Chi Minh City, April 5th, 2024

Contents

1 Introduction

In this report, I explore the implementation and performance evaluation of various solving methods for a constraint satisfaction problem. The problem involves determining the placement of traps and gems on a game board based on given constraints. The constraints are provided in the form of digits in the cells, indicating the number of traps in the neighboring cells.

To address this problem, I have implemented three different solving methods: a library-based method using the PySAT library, a brute-force method, and a backtracking method. Each method is encapsulated in a respective class, inheriting from a common abstract class to ensure consistency and reusability of code.

The main objective of this report is to compare the performance of these solving methods in terms of the time taken to find a solution. I have designed a comprehensive test environment to run experiments on multiple test cases and collect relevant data. The results are analyzed to provide insights into the efficiency and effectiveness of each method.

By systematically evaluating the performance of these solving methods, I aim to identify the most efficient approach for solving the given constraint satisfaction problem. This report provides a comprehensive understanding of the implementation details and the comparative performance of the different methods.

2 Problem Statement

2.1 Rules

- The problem gives us a grid where some cells contain a number between 1 and 9.
- The number in a cell indicates the **total number of traps (T)** in adjacent cells (horizontally, vertically, and diagonally). In this solution, we will consider at most 8 cells surrounding every cell.
- The objective is to determine whether each empty cell contains a trap (T) or a gem (G) while ensuring that all numerical clues are satisfied.

2.2 Goals

- Every cell in the grid must be assigned either T (Trap) or G (Gem).
- The final grid configuration must satisfy all given numerical constraints, meaning the number in each numbered cell must correctly reflect the number of adjacent traps.

3 Solution formulation

3.1 Determine logic for a blank cell

- To formulate the problem in Conjunctive Normal Form (CNF), we need to define the logical representation of each blank cell. In this approach, the focus is on determining whether a given cell contains a Trap (T) or not.
- Each blank cell is initially assigned a positive integer. After solving the CNF constraints:

- If the resulting value remains positive, the cell contains a Trap (T).
- If the resulting value is negative, the cell contains a Gem (G).

3.2 Creating CNF [1]

Step 1: Getting the Neighbor Cells For each cell in the board that contains a digit, we identify its neighboring cells. These neighbors are the cells that could potentially be traps or gems. The digit in the cell indicates the number of traps among its neighbors. This information is crucial for generating the CNF clauses. The function `get_neighbors` from `class Board` at 5.1 is used to find the legal 8 neighbors around a given cell, ensuring that only empty cells (potential traps or gems) are considered.

Step 2: Using Combinations to Create CNF To encode the constraints into CNF, we use combinations of the neighboring cells. The CNF clauses are generated in two parts:

- **At Least k Traps**: We generate combinations of the neighboring cells such that at least k cells are traps. This is done by creating combinations of length $\text{len(neighbors)} - k + 1$. Each combination represents a scenario where at least one of the selected cells must be a trap. For example, if there are 3 neighbors and 2 traps, we generate combinations of 2 neighbors, ensuring that at least one of them is a trap.
- **At Most k Traps**: We generate combinations of the neighboring cells such that no more than k cells are traps. This is done by creating combinations of length $k + 1$. Each combination represents a scenario where at least one of the selected cells must be a gem. For example, if there are 3 neighbors and 2 traps, we generate combinations of 3 neighbors, ensuring that at least one of them is a gem.

By combining these two sets of combinations, we create a comprehensive set of CNF clauses that accurately represent the constraints for the number of traps in the neighboring cells. The function `generate_cnf` from `class Board` at 5.1 is responsible for generating these combinations and forming the CNF clauses.

Step 3: Remove Duplicate Clauses To ensure the CNF is efficient and free of redundancy, we remove duplicate clauses. This is achieved by converting each clause to a sorted tuple, adding it to a set to eliminate duplicates, and then converting it back to a list.

3.3 Solving by Library [2]

To solve the Gem Miner problem using a library, we utilize the PySAT library, which provides efficient SAT solvers. The process involves the following steps:

1. **Initialize the Solver**: After generating the CNF, we initialize a SAT solver from the PySAT library.
2. **Add Clauses to Solver**: The CNF clauses are added to the solver.
3. **Enumerate Models**: The `enum_models()` function is used to enumerate all possible models (solutions) provided by the solver. This is done in case the first model is not the correct answer.
4. **Validate Each Model**: Each model is validated to ensure it meets all the problem constraints.
5. **Transform and Record the Result**: The first valid model is transformed into a readable format, indicating which cells are traps and which are gems, and the result is recorded.

The method is implemented at the **class Tool_library** at 5.2.1

3.4 Solving by Brute-Force Method [4]

The brute-force method involves systematically exploring all possible configurations of traps and gems in the neighboring cells. The process involves the following steps:

1. **Initialize the First Model**: After generating the CNF, the first model is initialized with all positive intergers (which means all cells are traps).
2. **Generate Combinations of Gems**: Combinations of gems are generated as the next possible models.
3. **Check Each Configuration**: Each configuration is checked to see if it satisfies all the constraints encoded in the CNF.
4. **Validate the Solution**: If a configuration satisfies all the constraints, it is validated to ensure it meets all the problem constraints.
5. **Transform and Record the Result**: The valid configuration is transformed into a readable format, indicating which cells are traps and which are gems, and the result is recorded.

The method is implemented at the **class Tool_Brute_Force** at 5.2.2

3.5 Solving by Backtracking Method [3]

The backtracking method is a more efficient approach compared to brute-force, as it prunes the search space by abandoning configurations that cannot possibly lead to a solution. The process involves the following steps:

1. **Initialize Solution**: Start with an empty solution list.
2. **Recursive Backtracking**: Recursively assign each variable to either a trap or a gem, and check if the partial solution is valid.
3. **Prune Invalid Configurations**: If a partial solution cannot lead to a valid solution, it is abandoned, and the algorithm backtracks to explore other configurations.
4. **Fill All Blank Cells**: The method tries to find a valid model after filling all the blank cells.
5. **Validate the Solution**: If a complete valid solution is found, it is validated to ensure it meets all the problem constraints.
6. **Transform and Record the Result**: The valid solution is transformed into a readable format, indicating which cells are traps and which are gems, and the result is recorded.

The method is implemented at the **class Tool_Backtracking** at 5.2.3

4 Experimental Setup

4.1 Input and Output Format

The input for the experiments consists of test case files that describe the initial configuration of the game board. Each test case file contains a grid where each cell may contain a digit indicating the number of traps in the neighboring cells or be empty. The format of the input files is as follows:

- Each line in the file represents a row of the game board.
- Digits (0-8) indicate the number of traps in the neighboring cells.
- Empty cells are represented by a specific character (e.g., '_').

Example input:

```
_, _, _, 3, _
_, 5, _, _, 3
3, _, _, _, _
_, _, _, _, 3
_, _, 5, _, _
```

The output files contain the results of the solving methods applied to the test cases. The output files are named with the method name and stored in the `Result` folder. The format of the output files is as follows:

- Each line in the file represents a row of the game board.
- 'T' indicates a trap.
- 'G' indicates a gem.

Example output:

```
T, T, T, 3, T
T, 5, G, T, 3
3, T, G, T, G
G, T, T, T, 3
G, T, 5, T, G
```

4.2 Example experiments:

The implementation will be demonstrated using the `Survey` class and the main implementation. The experimental process includes the following steps with the help of the `Testcase` class at 5.3.1:

- **Initialize Testcase**: Create an instance of the `Testcase` class to manage the test case files and output files.
- **Set Folder Paths**: Define the folder paths for the test case files in the `Testcase` folder and the output files in the `Result` folder.
- **Run Methods**: For each solving method (library, brute force, backtracking), read the test case files, generate output file names respectively, and run the corresponding solver on each test case. Record the time taken for each test case and write the results to the output files.
- **Collect and Print Results**: Collect the time taken for each method and print the results in a tabular format, showing the performance of each method on each test case.

4.3 Adding Test Cases

To add a new test case, follow these steps:

- Go to the `Testcase` folder in folder `Source`.
- Create a new file named `Testcase<num>.txt`, where `<num>` is a number different from the current files.
- Write the test case in the file using the specified input format.

5 Implementation

5.1 Class Board

The `Board` class is responsible for handling the game board and generating the CNF clauses based on the constraints provided by the digits in the cells. The key functionalities of the `Board` class include:

- **File Handling**: The `handle_file` method reads the board configuration from a file and stores it in a 2D list. Each line of the file is processed by the `handle_line` method to extract the cell values.
- **Variable Assignment**: The `assign_variables` method assigns unique variables to each empty cell on the board. These variables are used in the CNF clauses to represent the state of the cells (trap or gem).
- **Neighbor Identification**: The `get_neighbors` method identifies the neighboring cells for a given cell. This is crucial for generating the CNF clauses based on the number of traps indicated by the digits in the cells.
- **CNF Generation**: The `generate_cnf` method generates CNF clauses based on the constraints provided by the digits in the cells and their neighboring cells. It ensures that the number of traps around a cell matches the digit in the cell.
- **Duplicate Removal**: To ensure efficiency, the `get_cnf` method removes duplicate clauses from the CNF. This is done by converting each clause to a sorted tuple, adding it to a set to eliminate duplicates, and then converting it back to a list.
- **Model Transformation**: The `transform_model` method transforms a model (solution) into a readable format, indicating which cells are traps and which are gems. This is done by mapping the variables in the model to their corresponding cells on the board.
- **Validity Check**: The `check_valid` method checks if a given model satisfies all the constraints. This involves verifying that the number of traps around each cell matches the digit in the cell.

5.2 Class Tool

The `Tool` class serves as an abstraction for different solving methods. It provides common functionalities and serves as a base class for specific solving strategies. The key functionalities of the `Tool` class include:

- **Initialization**: The constructor initializes the board, CNF, list of variables, and other necessary attributes. It also calls the `solve` method to find a solution.

- **Validity Check**: The `check_valid_clause` method checks if a given clause is satisfied by a solution. The `check_valid_solution` method checks if a given solution satisfies all the CNF constraints.
- **Solution Validation**: The `check_valid` method validates a model by transforming it and checking if it meets all the problem constraints.
- **Solving**: The `solve` method is a placeholder to be implemented by subclasses. It is called during initialization to find a solution.
- **Time Measurement**: The `get_time` method returns the time taken to find a solution.
- **Output Writing**: The `write_output` method writes the result to a file.

5.2.1 Class Tool_Library

The `Tool_Library` class extends the `Tool` class and uses the PySAT library to solve the problem. The key steps include:

- **Initialize the Solver**: Initialize a SAT solver from the PySAT library.
- **Add Clauses to Solver**: Add the CNF clauses to the solver.
- **Enumerate Models**: Use the `enum_models()` function to enumerate all possible models.
- **Validate Each Model**: Validate each model to ensure it meets all the problem constraints.
- **Record the Result**: Record the first valid model as the result.

5.2.2 Class Tool_Brute_Force

The `Tool_Brute_Force` class extends the `Tool` class and uses a brute-force method to solve the problem. The key steps include:

- **Initialize the First Model**: Initialize the first model with all cells as traps.
- **Generate Combinations of Gems**: Generate combinations of gems as the next possible models.
- **Check Each Configuration**: Check each configuration to see if it satisfies all the constraints.
- **Validate the Solution**: Validate the configuration to ensure it meets all the problem constraints.
- **Record the Result**: Record the valid configuration as the result.

5.2.3 Class Tool_Backtracking

The `Tool_Backtracking` class extends the `Tool` class and uses a backtracking method to solve the problem. The key steps include:

- **Initialize Solution**: Start with an empty solution list.
- **Recursive Backtracking**: Recursively assign each variable to either a trap or a gem, and check if the partial solution is valid.
- **Prune Invalid Configurations**: Abandon partial solutions that cannot lead to a valid solution.
- **Fill All Blank Cells**: Try to find a valid model after filling all the blank cells.
- **Validate the Solution**: Validate the complete solution to ensure it meets all the problem constraints.
- **Record the Result**: Record the valid solution as the result.

5.3 Class Survey

The `Survey` class is responsible for running experiments and collecting data on the performance of different solving methods. The key functionalities of the `Survey` class include:

- **Read Test Case Files**: The `read_files` method reads the test case files from a specified folder.
- **Generate Output File Names**: The `generate_output_files` method generates output file names based on the test case file names and the solving method used.
- **Clear Files**: The `clear_files` method clears the lists of test case files and output files
- **Run Experiments**: The `run_experiments` method runs experiments using different solving methods (library, brute-force, backtracking) on the test case files. It initializes the appropriate `Tool` subclass for each method and measures the time taken to find a solution.
- **Collect Data**: Collect data on the time taken, number of models checked, and other relevant metrics for each solving method.
- **Analyze Results**: Analyze the collected data to compare the performance of different solving methods.
- **Generate Reports**: Generate reports summarizing the findings of the experiments.

5.3.1 Class Testcase

The `Testcase` class is responsible for managing the test case files and generating corresponding output file names. The key functionalities of the `Testcase` class include:

- **Read Files**: The `read_files` method reads the test case files from a specified folder. It checks if the folder exists and is a directory, and then lists all the files in the folder.
- **Generate Output Files**: The `generate_output_files` method generates output file names based on the test case file names and the solving method used. It appends the method name to the base name of each test case file.
- **Clear Files**: The `clear_files` method clears the lists of test case files and output files.

5.4 Main Implementation

The main implementation involves setting up the test environment, running the experiments using different solving methods, and collecting the results. The process includes the following steps:

- **Initialize Testcase**: Create an instance of the `Testcase` class to manage the test case files and output files.
- **Set Folder Paths**: Define the folder paths for the test case files and the output files.
- **Run Methods**: For each solving method (library, brute force, backtracking), read the test case files, generate output file names respectively, and run the corresponding solver on each test case. Record the time taken for each test case and write the results to the output files.
- **Collect and Print Results**: Collect the time taken for each method and print the results in a tabular format, showing the performance of each method on each test case.

6 Results and Discussion

6.1 Correctness of Solutions

The correctness of the solutions generated by each solving method is evaluated by comparing the output files against the expected results. Each method (library, brute force, backtracking) is tested on multiple test cases to ensure that the solutions adhere to the given constraints. The following criteria are used to assess correctness:

- All traps ('T') and gems ('G') are placed correctly according to the constraints provided in the input.
- The number of traps in the neighboring cells matches the digits in the input grid.

The results given by different methods can be different, and they are acceptable solutions.

6.2 Efficiency Analysis

The efficiency of each solving method is analyzed based on the time taken to find a solution for each test case. The time taken is recorded and compared across the different methods. The following steps outline the efficiency analysis:

- **Data Collection**: Record the time taken for each method to solve each test case.
- **Comparison**: Compare the time taken by each method to identify the most efficient approach.
- **Tabular Representation**: Present the results in a tabular format to provide a clear comparison of the performance of each method. When running the code, you can see the table in the terminal screen.

The table below summarizes the performance of each method on the 9 given testcases (Testcase1-3: 5x5 grid; Testcase4-6: 11x11 grid; Testcase7-9: 20x20 grid) and the methods return the time counting after testing at most 1 billion models despite finding the solutions or not.

Order	Library (s)	Bruce Force (s)	Backtracking (s)
Testcase1.txt	0.000745	0.030759	0.044101
Testcase2.txt	0.000269	0.001482	0.044145
Testcase3.txt	0.000244	0.005878	0.000966
Testcase4.txt	0.000629	20.858312	0.025969
Testcase5.txt	0.000784	31.258310	1.490619
Testcase6.txt	0.000878	39.316023	3.234618
Testcase7.txt	0.005819	105.849028	6.558769
Testcase8.txt	0.010658	86.159180	2.599631
Testcase9.txt	0.009047	86.664150	1.753590

Figure 1: Comparision between different testcases

The results indicate that the library method is the most efficient, followed by the backtracking method, and then the brute force method. This analysis helps in identifying the most suitable method for solving the given constraint satisfaction problem based on efficiency.

7 Source code and Video demo

7.1 Link Github

The complete source code is available on GitHub at the following link

7.1.1 How to Run Source Code

To run the source code, follow these steps:

- **Clone the Repository**: Clone the GitHub repository to your local machine using the following command:

  ```
  git clone https://github.com/AnhTtis/GemMinner.git
  ```

- **Navigate to the Project Directory**: Change to the project directory:

  ```
  cd GemMinner
  ```

- **Run the Main Script**: Execute the main script to run the experiments:

  ```
  python Source/main.py
  ```

- **View Results**: The results will be saved in the `Result` folder, and the performance comparison will be printed in the console.

7.2 Video Demonstration

A video demonstration of the implementation and experimental process is available at the following link

The video covers the following aspects:

- Overview of the project and its objectives.
- Detailed explanation of the input and output formats.
- Step-by-step guide on how to run the source code.
- Demonstration of the solving methods and their performance.
- Analysis of the results and discussion on the efficiency of each method.

References

[1] Kristóf Bérczi et al. "Generating clause sequences of a CNF formula". In: *Theoretical computer science* 856 (2021), pp. 68–74.

[2] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. "PySAT: A Python toolkit for prototyping with SAT oracles". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2018, pp. 428–437.

[3] Inês Lynce and João P Marques-Silva. "An overview of backtrack search satisfiability algorithms". In: *Annals of Mathematics and Artificial Intelligence* 37 (2003), pp. 307–326.

[4] Nicolas Pietro Martignon. "Using SAT solvers and brute force approach to recover AES key from partial key schedule images". In: (2022).