**Brute – Force:**

1. **Overview:** The brute-force approach tries all possible combinations of trap and gem placements to solve the puzzle, validating each arrangement against the numbers indicated on the grid.
2. **Test case results:**
   1. **Map 5x5:**

* Execution Time: 0.064 seconds.
* Observation: The smaller map size facilitated quicker computation, allowing the brute-force method to solve the puzzle effectively within a short time.
  1. **Larger Maps (9x9, 11x11, 15x15, 20x20):**
* Execution Time: Exceeded reasonable limits for brute-force computation.
* Observation: As the grid size increased, the number of possible states expanded exponentially, making the brute-force approach impractical due to excessive computation times.
  1. **Special Maps:**

**2.3.1 Sporadic Map:**

* Grid Layout:

A black background with white numbers

Description automatically generated

* Execution Time: 0.184 seconds.
* Expected Result: Spaces not adjacent to numbered tiles remained unchanged as "\_", indicating that there was no direct information about traps in those locations.

A black background with white letters

Description automatically generated

* Explanation: Without adjacent numbers, it is impossible to determine the presence of traps logically, thus they remain undetermined.
* Evaluate: The Sporadic map, which contains cells with numbers as well as many empty cells, represents a relatively simple scenario for the brute-force algorithm. The algorithm likely identified the cells that could be definitively determined based on their adjacent numbered cells and did not have to explore further combinations for

**2.3.2 Empty Map:**

* Grid Layout:

A black and white background with white lines

Description automatically generated

* Execution Time: 8.516 seconds.
* Expected Result: No changes made to the grid.

A black and white background with white lines

Description automatically generated

* Explanation: The lack of any numbers implies there are no clues available to place traps or gems, leading to an unchanged grid state.
* Evaluate: Surprisingly, the Empty map, devoid of any numbers to guide the placement of traps or gems, took significantly longer—8.516 seconds—to reach a conclusion. This may seem counterintuitive since there are no constraints to consider. However, the brute-force algorithm would still enumerate all possible combinations of traps and gems for every cell, which grows exponentially with the size of the map. Despite the lack of information, the algorithm expends considerable time exploring every possible configuration, which makes it inefficient for such scenarios.

**2.3.3 Wrong Map:**

* Grid Layout:

A black background with white numbers

Description automatically generated

* Execution Time: 0.195 seconds.
* Expected Result: “No solution”.
* Explanation: If the given numbers on the grid are inconsistent with any possible arrangement of traps and gems, the brute-force method will terminate without finding a solution, as the conditions stipulated by the numbers cannot be met.
* Evaluate: The Wrong map likely presented a configuration of numbers that were inconsistent or infeasible based on the game's logic. The brute-force algorithm would attempt to place traps and gems according to the numbers but would quickly find that no arrangement could satisfy the conditions given. Consequently, it would terminate and return "No solution" in a short time (0.195 seconds), as no valid configurations exist.

1. **Conclusion:**

* The results indicate that the brute-force method is not necessarily faster when there are fewer constraints. Rather, the presence of guiding information (even if it leads to no solution) can reduce computation time.
* The brute-force approach, while straightforward and theoretically sound, proves inefficient for larger grids due to the exponential growth in possible states. It performs well on smaller grids or less complex problems but becomes computationally prohibitive as complexity increases. Special cases like the "Sporadic" and "Empty" maps highlight scenarios where brute-force either cannot decide or no computation is necessary. For larger maps or more intricate puzzles, more sophisticated algorithms or methods like constraint programming or SAT solvers might be required to achieve feasible results within practical time limits.

1. **References:**
2. <https://www.freecodecamp.org/news/brute-force-algorithms-explained/>
3. <https://www.codecademy.com/learn/learn-data-structures-and-algorithms-with-python/modules/brute-force-algorithms/cheatsheet>
4. <https://stackoverflow.com/questions/11747254/python-brute-force-algorithm>

**PYSAT:**

1. **Methodology:**

* Create test case: Browse through the elements in the grid array that have been read and filtered from the \*.txt file. If any element is a number, we will create a logical proposition based on that number. Consider which neighbor element around that cell is empty ('\_'). Then the neighbors added to the clause can be trap(T) / goal(G). The clause cnf is a list, and includes sublists [a, b, ...] symbolizing a v b v ... The idea is that the sublists in cnf will have a disjunctive relationship with each other ( l1 ^ l2 ^ . .. ).
* Evaluation based on criteria:

+ Accuracy: Proportion of traps (T) and goals (G) correctly identified.

+ Time complexity: With this PYSAT algorithm. We coded it with O(2^v) complexity. Generating cnf takes O(n\*m) for a grid size of nxm and solving with solver() is O(2^v) where v is the number of variables in the problem.

1. **Experimental Results:**

* With a 5x5 map, the average execution time is 4.12 ms (average of 3 tests).
* 9x9: 7.22 ms
* 11x11: 5.42 ms
* 15x15: 13.92 ms
* 20x20: 12.35 ms

A screenshot of a computer

Description automatically generated

* Comment:
* Figures show that execution time increases as map size increases. But there is a slight decrease when comparing 15x15 vs 20x20 maps. This may be due to:

+ Complexity of the problem: Although the number of variables is greater, it is possible that simpler statements lead to easier CNF processing -> faster time.

+ Modern algorithms like Glucose used in PySAT have effective optimization techniques -> minimize solution time even for problems with more variables and statements.

* Variation in execution time: Theoretical resolution time will increase proportionally to map size. However, with actual testing, the 20x20 map runs faster than 15x15.

1. **Special Maps:**
   1. **Sporadic Map:**

A number with white and red numbers

Description automatically generated with medium confidence

* Execute time: 4.4 ms

A screen shot of a computer

Description automatically generated

* 1. **Empty Map:**

**A screenshot of a computer

Description automatically generated**

* Execute time: 0.00 ms

**A screenshot of a computer

Description automatically generated**

The solve function using the pysat library is optimally effective in the case of an empty map with no numbers. Therefore, the execution time is very small to the point of being almost zero.

* 1. **Wrong Map:**
* We have wrong map like:

**A screenshot of a computer

Description automatically generated**

* In cell 3 as circled below, we can deduce that the 3 empty cells surrounding cell 3 must all be T. This means that cell 1 next to the left of cell 3 must have a number of at least 2. But that cell has the number 1 -> input is wrong.

**A screenshot of a computer

Description automatically generated**

* Execute time: 8.75 ms

**A number of numbers on a dark background

Description automatically generated**

[**https://pysathq.github.io/docs/html/api/solvers.html**](https://pysathq.github.io/docs/html/api/solvers.html)

**DPLL:**

1. **Methology:**

* The **Davis–Putnam–Logemann–Loveland (DPLL**) algorithm is a deep search algorithm used to solve the SAT decision problem. This algorithm is designed to improve the efficiency of the Davis–Putnam algorithm by adding techniques such as:

+ **Unit propagation**: If a clause leaves only one undefined variable, that variable must be assigned a value such that the clause is satisfied.

+ **Pure literal elimination:** If a variable appears in only one form (not its negation) in all clauses, it can be assigned a satisfactory value without affecting the other clauses.

* The [DPLL](http://en.wikipedia.org/wiki/DPLL_algorithm) is essentially a **backtracking algorithm**, and that's the main idea behind the recursive calls.

1. **Experimental result:**

* Map 5x5: 4.03 ms ( average of 3 times running)

A screenshot of a computer program

Description automatically generated

* Map 9x9: 8.72 ms

A screenshot of a computer

Description automatically generated

* Map 11x11: 22.11 ms

A screenshot of a computer program

Description automatically generated

* Map 15x15: 70.67 ms

A screenshot of a computer screen

Description automatically generated

* Map 20x20: 326.57 ms

A screenshot of a computer screen

Description automatically generated

* **Comment:**
* Large growth time: It can be seen that the execution time increases exponentially compared to the size of the map. Consistent with the characteristics of the DPLL algorithm and the nature of the SAT problem. Complexity increases sharply as the number of variables and clauses increases.
* The algorithm has the ability to effectively handle small and medium maps. With larger inputs, the efficiency decreases.

1. **Special Maps:**
   1. **Sporadic Map:**

A screenshot of a computer program

Description automatically generated

* 1. **Empty Map:**

A screenshot of a computer

Description automatically generated

* 1. **Wrong Map:**

A computer screen shot of a number

Description automatically generated

<https://stackoverflow.com/questions/12547160/how-does-the-dpll-algorithm-work>