Sudoku

This assignment aims to analyze the implementation of three SAT (Boolean satisfiability problem) solving algorithms—GSAT, walkSAT, and DPLL—to solve Sudoku puzzles. We study their efficiency, feasibility, and particular merits and drawbacks in the context of solving Sudoku puzzles. I also demonstrate their performances on different problem sets.

GSAT

- **Overview**

  GSAT (Greedy SAT) is a local search algorithm for solving SAT problems. It uses a greedy strategy and works by flipping variable assignments to maximize the number of satisfied clauses.

- **Implementation**

  The implementation includes a random initialization of variable assignments. Then, in each iteration, the algorithm either flips a random variable or chooses the variable that maximizes the number of satisfied clauses when flipped.

  - The GSAT algorithm aims to solve Boolean satisfiability problems by iteratively flipping variable assignments to `maximize` the number of `satisfied clauses`.
  - The code initializes a `random assignment`` for all variables and enters a loop where it either flips a random variable or selects the variable that, when flipped, maximizes the number of satisfied clauses. The algorithm stops and returns the assignment when all clauses are satisfied or when it hits the maximum iteration limit.
  - To avoid getting stuck in a `local minimum`, the code includes a mechanism for flipping 20 random variables if the best score repeats more than `10 times` and the algorithm is at least `99% complete` in satisfying clauses. This is governed by the `best_score_repeat_count` and `threshold` variables. If a solution isn't found within the given number of iterations (`max_iter`), the function returns `None`.

- **Discussion**

  While GSAT is functional, its performance is hindered by the need to iterate through all clauses to calculate the score for each randomly flipped variable. This makes the algorithm susceptible to getting stuck in local minima, especially because initial variable assignments are random. I attempted to mitigate this by resetting some variables when the algorithm is close to finding a solution, but this tweak hasn't substantially improved the algorithm's efficiency.

walkSAT

- **Overview**

  walkSAT is another local search algorithm for solving SAT problems but incorporates randomness to escape local minima.

- **Implementation**

  The algorithm begins with a random assignment of variables and iteratively refines the assignment to satisfy more clauses. It incorporates randomness to escape local minima.

  - The WalkSAT algorithm aims to solve the SAT problem using a heuristic, `local search approach`. The code starts by randomly initializing a variable assignment for all clauses in the SAT problem. The algorithm iterates up to a predefined maximum number of iterations max_iter. During each iteration, it first checks if the current assignment satisfies all clauses. If it does, the function returns the assignment as the solution. Otherwise, it picks an `unsatisfied` clause at random `random_clause` for further examination.

  - The algorithm then decides whether to flip a random variable in the selected unsatisfied clause or to choose the variable that maximizes the number of satisfied clauses when flipped. This decision is made based on a threshold parameter `threshold`. If a random float is less than the threshold, a random variable from the clause is chosen. Otherwise, the algorithm calculates scores for each variable in the clause by flipping it and evaluating how many clauses are satisfied with that flip. It selects the variable that maximizes this score for the next iteration. After the variable is chosen, its value is flipped in the assignment, and the algorithm proceeds to the next iteration.

- **Discussion**

  In summary, WalkSAT is generally more efficient than GSAT because it employs a localized search strategy that targets unsatisfied variables, thereby reducing the search space. This focused approach speeds up the search for a satisfying assignment. Although WalkSAT is still susceptible to getting stuck in local minima, the inclusion of random variable flips in the assignment helps it navigate around these obstacles to some extent.

DPLL (Bonus)

- **Overview**

  DPLL (Davis-Putnam-Logemann-Loveland) is a backtracking algorithm for solving SAT problems, offering complete solutions.

- **Implementation**

  The algorithm employs unit propagation and pure literal elimination to simplify the problem. It then recursively tries to find a satisfying assignment for variables.

- The `DPLL` function serves as the primary recursive search algorithm, making use of helper functions like `unit_propagate` and `pure_literal_assign` to simplify the clause set. `unit_propagate` handles unit clauses by removing or updating other clauses that are affected by a unit clause's literal. It assigns the value `True` to that literal in the assignment. On the other hand, `pure_literal_assign` takes care of pure literals, which are literals that appear in only one polarity across all clauses. It assigns them a value that makes all the containing clauses true, thereby eliminating those clauses from further consideration.

- The main `DPLL` function orchestrates the algorithm's flow, starting with unit propagation followed by pure literal assignment. If it finds that all clauses are satisfied or that an empty clause is generated, it returns accordingly. Otherwise, it performs a recursive search by picking a random literal and trying both possible boolean assignments. Each assignment leads to a new recursive call, effectively exploring the space of all possible assignments. The function keeps track of its recursion depth with a `count` variable, which can be useful for debugging or analysis. At the end, `DPLL_SAT` simply initiates this entire process and prints out whether the set of clauses is satisfiable, along with the satisfying assignment if it exists.

- **Discussion**

  To be candid, I'm uncertain about the correctness of my implementation. Despite running for a considerable amount of time, the solver didn't arrive at a solution within the time constraints. The algorithm is designed to speed up by eliminating pure literals, thereby reducing the search space. Yet, my code has been struggling to find a solution, leading me to suspect that there may be an issue with the implementation.

Problem Tests

puzzle1 - walksat 26146 iterations taken to get the solution...

```
5 7 8 | 4 2 6 | 1 3 9
6 1 4 | 7 9 3 | 5 2 8
9 2 3 | 1 8 5 | 7 6 4
---------------------
8 3 2 | 5 6 1 | 4 9 7
1 6 9 | 8 7 4 | 3 5 2
7 4 5 | 2 3 9 | 8 1 6
---------------------
4 9 1 | 6 5 8 | 2 7 3
2 8 6 | 3 1 7 | 9 4 5
3 5 7 | 9 4 2 | 6 8 1
```

puzzle2 - walksat
14438 iterations taken to get the solution...

```
5 3 6 | 9 8 2 | 4 1 7
2 7 8 | 1 4 5 | 6 3 9
```

```
4 9 1 | 6 7 3 | 5 2 8
---------------------
8 4 9 | 2 6 1 | 7 5 3
6 1 3 | 8 5 7 | 9 4 2
7 5 2 | 3 9 4 | 1 8 6
---------------------
9 6 5 | 4 2 8 | 3 7 1
1 2 4 | 7 3 9 | 8 6 5
3 8 7 | 5 1 6 | 2 9 4
```