Individual Course Project
Fall 2025

MineMind (CLI)

Objective: Create a complete, self-contained Minesweeper generator + solver in Python. No GUI, everything runs in a friendly command-line REPL (read–eval–print loop). You'll implement the game mechanics and a reasoning engine that can solve boards using logic and small exact searches.

Why This Project?

Learning outcomes (you will be assessed on these):

- Implement classic data structures: Union–Find, priority queues, LRU cache, bitset/bitmask sets, queues/stacks, and use BFS/DFS.
- Model logical constraints and perform deterministic inference (rule-based reasoning).
- Implement a bounded exact search on small components with early pruning, and compute cell-wise probabilities.
- Design a clean CLI/REPL with stable, deterministic behavior and save/load snapshots (JSON).
- Communicate correctness: tests, invariants, complexity analysis, and a concise design write-up.

What You Will Build

A Python package `minemind` that launches a REPL:

```
python -m minemind new --w 16 --h 16 --mines 40 --seed 1337
```

Then, you play or ask the solver for help using commands like `open 3 7`, `flag 4 9`, `hint`, `step`, `auto --guess`.

Board sizes: Beginner 9×9/10 mines, Intermediate 16×16/40, Expert 30×16/99, plus custom.

First-click safe: Mines are placed after your first `open X Y`, never on that cell or its neighbors.

Required Features

CLI commands (REPL)

```
help                                # list commands
new --w W --h H --mines M [--seed S]# start a new game
show [--reveal]                     # print board; --reveal shows mines
(debug/after loss)
open X Y                            # reveal cell at (X,Y)
flag X Y                            # toggle flag
chord X Y                           # on a revealed number: if flags match,
reveal remaining neighbors
hint                                # print one certain safe/mine move with
explanation
step                                # apply one deterministic solver step; or
exact small-component step
auto [--guess] [--limit N]          # run solver up to N steps; --guess
allows lowest-risk guesses
prob                                # print coarse ASCII probability heatmap
for unknown cells
frontier                            # summary: #components, sizes, unknowns
per component
save path.json                      # snapshot game state to JSON
load path.json                      # restore snapshot
quit | exit
```

Coordinates: zero-based; `open 3 7` = column 3, row 7.

Board printing (ASCII)

- Unknown: `·` (dot)
- Flag: `F`
- Revealed number: `1..8`
- Revealed zero: space `' '`
- Mine (on loss or `--reveal`): `*`

Include column header and row indices for orientation.

Generator & rules

- Defer mine placement until the first `open`.
- Do not place a mine on the first click or its 8 neighbors.
- Neighbor counts must be correct.
- `open` on a zero performs a flood fill (BFS/DFS) revealing the zero region and its perimeter numbers.
- `chord` on a revealed number reveals all adjacent unknowns only if the number of flagged neighbors equals the number shown.

Solver requirements

Frontier & constraints

- Build a frontier consisting of all revealed numbered cells that touch at least one unknown cell.
- For each such cell `v` with label `L` and unknown neighbors `N(v)`:
    - Constraint: `∑_{u ∈ N(v)} x_u = L - flags(v)` where `x_u ∈ {0,1}` (1 = mine).
- Represent sets as bitmasks over locally indexed unknowns to allow fast ops.

Component decomposition (Union–Find)

- Build an intersection graph over constraints: two constraints connect if they share any unknown cell.
- Use union–find to split the frontier into independent components; solve each component separately.

Deterministic rules (must implement)

1. **Singles**
    - If `remaining == 0` → all cells in scope are **SAFE**.
    - If `remaining == |scope|` → all cells in scope are **MINES**.
2. **Subset rule**
    - For constraints `(A, a)` and `(B, b)` with `A ⊆ B`:
        - If `a == b` → `B \ A` are **SAFE**.
        - If `b - a == |B \ A|` → `B \ A` are **MINES**.
3. **Scope equality/complements** (merge or simplify identical/complimentary scopes).

Your **`hint`** must print which rule fired and the involved coordinates.

Exact enumeration on small components

- For any component with `k ≤ k_max` unknowns (default **`k_max = 20`**):
    - Perform backtracking enumeration of satisfying assignments with early pruning.
    - Compute per-cell mine probability = (# solutions with cell=mine) / (total solutions).
    - If `total solutions == 0`, report an inconsistency (shouldn't occur with correct play).
- Use these probabilities to:
    - Confirm certain safe/mine moves (p=0 or p=1) when rules alone don't decide.
    - Support `prob` and `auto --guess`.

Guess selection (when required)

- Maintain a **min-heap** keyed by `(p_mine, tiebreakers)`; pick the cell with **lowest** estimated mine probability.
- Recommended tiebreakers: prefer central cells, or those that maximize expected information (simple heuristic is fine).

Caching

- Implement a small LRU cache keyed by a canonical signature of a component (sorted bitmask scopes + remaining vector).
- Invalidate entries when any cell in the component is opened/flagged/unflagged.

Data structures (Required)

- **Union–Find (Disjoint Set)** with path compression + union by rank/size.
- **Priority Queue** (wrap `heapq` cleanly; min-heap).
- **LRU Cache** (hand-rolled or `collections.OrderedDict` based; no third-party lib).
- **Bitmasks** for local sets (ints; use bit ops).
- **Queue/Stack** for flood-fill and undo (if you choose to support undo; optional).
- **Sorting + Binary Search** for deterministic frontier ordering and stable output.

You may use Python **stdlib** only. Do not use SAT/CSP libraries or external DS packages.

Project Structure

```
/minemind
  __main__.py           # entry: starts REPL
  cli.py                # REPL and command handlers
  render.py             # board ASCII printing
/core
  rng.py                # seeded RNG
  generator.py          # first-click-safe mine placement, neighbor counts
  board.py              # grid state, open/flag/chord, flood fill
  frontier.py           # build frontier, local indexing, component extraction
  dsu.py                # union-find
  rules.py              # singles, subset, scope merges
  solver.py             # exact enumeration, probabilities, auto/step/hint
  lru.py                # LRU cache
  signatures.py         # canonical component signature
  snapshot.py           # save/load JSON snapshot
/tests
  test_generator.py
  test_board.py
  test_rules.py
  test_solver_small.py
  test_frontier.py
```

Deliverables

1. Code organized as above (or equivalent clarity).
2. README.md
   - How to install/run.
   - Command cheatsheet with examples.
   - Known limitations.
3. DESIGN.md
   - Diagrams of frontier/components.
   - Invariants for DSU, board, and solver.
   - Data flow for `hint/step/auto`.
4. COMPLEXITY.md
   - Big-O for core operations (open, flood, build frontier, rule pass, enumeration).
   - Justify your `k_max` choice (e.g., 20).
5. TESTING.md
   - What each test covers; how to run `pytest`.
6. At least 15 unit tests hitting: generator properties, flood fill, rules, DSU components, small exact solve, probability sums.
7. Two sample snapshots (`.json`): one mid-game; one near endgame.
8. Short demo (text transcript or <3-min screen capture) showing: `new`, `open`, a rule-based `hint`, an `auto --guess`, `save/load`.

Grading rubric (100 pts + up to 7 bonus)

- **Functionality** (40 pts)
  - Generator (first-click safety, counts), play features (open/flag/chord/flood) … 15
  - Solver deterministic rules (`hint`, `step`) … 10
  - Exact enumeration & probabilities; `auto --guess` … 15
- **DS/Algorithm Correctness** (25 pts)
  - DSU partitioning & invariants … 6
  - Bitset subset logic & rule soundness … 8
  - Enumeration correctness, pruning, probability math … 11
- **Code Quality** (15 pts)
  - Modularity, naming, docstrings, invariants, typing … 15
- **Testing** (10 pts)
  - ≥15 tests; good edge cases; deterministic behavior with seeds … 10
- **Documentation** (10 pts)
  - README, DESIGN, COMPLEXITY, TESTING completeness & clarity … 10
- **Bonus** (up to +7 pts)
  - Monte Carlo fallback for `k > 20` components … +3
  - 3BV difficulty metric & histogram … +2
  - Verbose "proof viewer" in `hint` (shows coordinates/bitsets used) … +2

Acceptance criteria (Allows You to Self-Check Your Solution)

- **First click safety:** In 200 randomized trials, first click and its neighbors are never mines.
- **Counts correct:** Random boards pass property test `count == #adjacent_mines`.
- **Chord correctness:** Only reveals when flags match the number.
- **Rules fire:** Provide at least 3 reproducible scenarios where Singles and Subset both trigger.
- **Exact solve:** On a component with `k ≤ 10`, probabilities match brute force within floating-point tolerance.
- **`auto --guess`:** Works and terminates on complete game or step limit; prints final status.
- **Save/Load:** Round-trips without loss (same board state and clock).
- **Determinism:** With seed fixed, repeated sessions produce the same sequence of boards.

Implementation tips

- **Bitmasks:** map local unknowns 0..k-1 → `int` mask; use `&`, `|`, `^`, `~`, `bit_count()`.
- **Pruning:** maintain `remaining` for each constraint; if `remaining < 0` or `remaining > |scope_unassigned|`, backtrack.
- **Branching order:** choose the variable in most constraints first to reduce search.
- **LRU:** cache only after sorting scopes and normalizing remaining vector.
- **Testing harness:** seed PRNG; generate random boards; assert invariants before/after moves.

Policies

Allowed resources

- Python **3.10+** (prefer 3.11).
- Standard library only (e.g., `random`, `heapq`, `collections`, `dataclasses`, `argparse`, `json`).
- You may consult references/tutorials for Minesweeper rules and general DS ideas, but **core solver/DS code must be developed by you**.

AI/assistance policy

- You may use AI tools **for explanations or debugging ideas**, but **not** to generate the core DS/solver code.
- If you used any external sources for ideas, add a short note in `ACKNOWLEDGMENTS.md`.

Collaboration

- Discussion of concepts is OK; sharing code is not. Your tests and implementation must be your own.

Submission

- Submit a zip or repository link that includes: code, tests, snapshots, and all required docs.
- Ensure `python -m minemind new --w 9 --h 9 --mines 10 --seed 42` launches the REPL and `help` works.
- Include a short transcript in README showing a typical session.

Example Session

```
$ python -m minemind new --w 9 --h 9 --mines 10 --seed 42
New game: 9x9, 10 mines, seed=42
> show
    0 1 2 3 4 5 6 7 8
 0  · · · · · · · · ·
 ...
> open 3 3
(revealed 12 cells)
    0 1 2 3 4 5 6 7 8
 0        1 1 1 · · ·
 1        1   1 · · ·
 2        1 1 1 · · ·
 3        1   1 · · ·
 ...
> hint
SAFE: (5,4) — SUBSET: N(4,3) ⊆ N(4,4) and remaining equal → B\A safe
> step
Applied SINGLE at (2,5): remaining=0 → all neighbors safe
> auto --guess --limit 100
Auto: 27 steps; guessed (1,8) with p_mine=0.08; solved in 31 steps.
> save run1.json
Saved.
```

Glossary

- **Frontier:** set of numbered revealed cells adjacent to unknowns.
- **Component:** independent subset of constraints/unknowns (no overlap with others).
- **k_max:** max unknowns in a component for exact enumeration.
- **3BV:** board difficulty metric (bonus).