Testrist
Testing Documentation and Known Issues

Robert Rice

December 4, 2025

1. Testing Documentation

1.1 Testing Strategy

The goal of testing for **Testrist** is to ensure that the game meets its functional requirements and that the leaderboard backend behaves correctly and safely.

We used a mix of:

- **Black-box testing**
  These tests focus on the game from the player's point of view. They cover piece spawns, player controls, collision detection, line clearing, scoring, level progression, game over, and the start/pause/reset controls. They are implemented as end-to-end tests using **Playwright**, which automates a real browser and interacts with the UI like a user would.
- **White-box testing**
  These tests focus on the internal behavior of the Python FastAPI leaderboard backend. They call the API endpoints directly, check HTTP responses, and verify the database state in the SQLite database.

For each test, we record:

- Inputs (key presses, button clicks, API payloads, etc.),
- Expected outputs (visual changes, API responses, database effects),
- Requirement IDs (for black-box tests),
- Source code references (for white-box tests),
- Pass/Fail status when the test is actually run.

This structure makes it easy to re-run the same tests after any code change (regression testing) and to show how our tests are connected to both the requirements and the source code.

1.2 Requirements and Traceability

To keep traceability clear, we assigned simple requirement IDs.

1.2.1 Game Requirements (Frontend)

- **G1.0 Tetromino Spawns**
  - G1.1: A tetromino spawns when a new game starts.
  - G1.2: A new tetromino spawns after each piece locks.
  - G1.3: The next tetromino is chosen pseudo-randomly.
- **G2.0 Player Controls**
  - G2.1: Player can move the piece left and right within the board.
  - G2.2: Player can soft-drop and hard-drop the piece.
  - G2.3: Player can rotate the piece, and rotation respects walls and the existing stack.
- **G3.0 Collision Detection**
  - G3.1: Falling pieces stop when they hit the stack or floor.
  - G3.2: Pieces cannot move out of bounds.

  - G3.3: Rotations cannot overlap existing blocks.

- **G4.0 Line-Clearing Logic**
  - G4.1: A fully filled horizontal line is cleared.
  - G4.2: Multiple lines can be cleared at once (double, triple, "Tetris").
  - G4.3: Blocks above cleared lines fall down correctly.
- **G5.0 Scoring System / Leaderboards**
  - G5.1: Score increases according to the number of lines cleared.
  - G5.2: Score is displayed while the game is running.
  - G5.3: Final score is shown on the game over screen.
  - G5.4: Player can submit a score to the leaderboard.
  - G5.5: Leaderboard lists top scores in the correct order.
- **G6.0 Level Progression**
  - G6.1: Level increases after a defined number of cleared lines.
  - G6.2: Piece fall speed increases with each level.
- **G7.0 Game Over**
  - G7.1: Game ends when a new piece cannot spawn due to filled cells.
  - G7.2: Game over screen is shown and normal movement input is disabled.
- **G8.0 Start / Pause / Reset**
  - G8.1: Start button begins a new game.
  - G8.2: Pause button freezes piece movement and timers.
  - G8.3: Resume continues the game from a paused state.
  - G8.4: Reset clears board, score, and level and starts fresh.

1.2.2 Line Clearing and Gravity

ID	Req.	Rationale	Inputs	Expected Output	P/F
BB–006	G4.1, G3.1	Confirm single line clear works.	Fill exactly one full row with pieces and let the last piece lock.	Full row disappears; rows above move down by one; cleared row is empty.	
BB–007	G4.2, G4.3	Confirm multi-line clear (for example, a "Tetris").	Set up board so that one piece clears 2–4 full rows at once.	All completed lines are removed together; score and line count reflect a multi-line clear; blocks above fall correctly.	
BB–008	G4.3	Ensure gravity behaves after partial clears.	Create a pattern where clearing a lower line should make a group of blocks drop straight down.	After the clear, blocks fall straight down with no diagonal shifts or floating gaps.	

Each black-box test case lists the Gx.y requirement(s) it covers. Each white-box test case lists the Lx.y requirement(s) and the related source code element.

1.3 Test Environment and Tools

- **Frontend:** A React-based Testrist game running in a web browser.
- **Backend:** A Python FastAPI application with an SQLite database.
- **Automation tools:**
  - Playwright for end-to-end UI tests (black-box).
  - A Python test client (for example, pytest + FastAPI test client) for API and database verification (white-box).
- **Browsers:** Chromium (primary test target); optional tests in Firefox or WebKit as needed.

1.4 Black-Box Test Cases (Playwright)

The following test cases are implemented as Playwright scripts that open the game, click buttons, and send key presses just like a real player.

1.4.1 Core Gameplay and Controls

ID	Req.	Rationale	Inputs (Actions)	Expected Output	P/F
BB–001	G1.1, G8.1	Verify that the game starts with a piece on the board.	Open game URL; click *Start*.	A tetromino appears at the top spawn row; game state is "running".	
BB–002	G1.2, G3.1	Make sure a new piece spawns only after the old one locks.	Start new game; soft-drop or wait until the current piece reaches the bottom and locks.	Current piece locks into the stack; a new tetromino spawns at the top.	
BB–003	G2.1, G3.2	Test left/right movement and wall boundaries.	Start; repeatedly press *Left* until at left wall, then press *Left* again; repeat on right side with *Right*.	Piece moves until it touches board edges; extra key presses do not move it out of bounds.	
BB–004	G2.2, G3.1	Verify soft-drop and hard-drop behavior.	Start; hold *Down* to soft-drop; in a new game press hard-drop key once.	Soft-drop moves piece faster but stepwise; hard-drop moves piece to lowest valid position and locks immediately.	
BB–005	G2.3, G3.1, G3.3	Check rotation behavior near walls and stack.	Start; move piece next to left wall and press *Rotate*; drop on stack and attempt rotations on top of blocks.	Rotations that would overlap walls or stack are blocked or adjusted; no overlapping blocks appear.	

1.4.2 Line Clearing and Gravity

ID	Req.	Rationale	Inputs	Expected Output	P/F
BB–006	G4.1, G3.1	Confirm single line clear works.	Fill exactly one full row with pieces and let the last piece lock.	Full row disappears; rows above move down by one; cleared row is empty.	
BB–007	G4.2, G4.3	Confirm multi-line clear (for example, a “Tetris”).	Set up board so that one piece clears 2–4 full rows at once.	All completed lines are removed together; score and line count reflect a multi-line clear; blocks above fall correctly.	
BB–008	G4.3	Ensure gravity behaves after partial clears.	Create a pattern where clearing a lower line should make a group of blocks drop straight down.	After the clear, blocks fall straight down with no diagonal shifts or floating gaps.	

1.4.3 Scoring and Level Progression

ID	Req.	Rationale	Inputs	Expected Output	P/F
BB–009	G5.1, G5.2	Verify scoring for single and multi-line clears.	In one game, clear a single line; in a fresh game, clear two lines at once; compare scores.	Score increases after each clear; multi-line clear yields more points than a single line; display updates immediately.	
BB–010	G5.3, G7.1, G7.2	Show final score on game over screen.	Play until the grid is filled and game over is triggered.	Game stops; game over screen shows final score matching last in-game score; movement controls are disabled.	
BB–011	G6.1, G6.2	Confirm level progression and increased fall speed.	Start; clear enough lines to trigger a level-up (for example, ten lines).	Level indicator increases; piece fall interval is shorter after the level-up.	

1.4.4 Game State Controls: Start, Pause, Reset

ID	Req.	Rationale	Inputs	Expected Output	P/F
BB–012	G8.1, G8.2, G8.3	Test pause and resume behavior.	Start; let piece fall; click *Pause*; wait a few seconds; click *Resume*.	While paused, piece does not move and controls do nothing; after resume, falling continues from the same position and controls work again.	
BB–013	G8.4, G5.2, G6.1	Test full reset behavior.	Start; play until score, lines, and level are all non-zero; click *Reset*.	Board is cleared; score, lines, and level are set back to starting values; game is ready for a new run.	

1.4.5 Leaderboard via UI (End-to-End)

ID	Req.	Rationale	Inputs	Expected Output	P/F
BB–014	G5.4, G5.5, L1.1, L2.1	Ensure that players can submit scores and see them sorted correctly.	Finish a game; enter a name; submit score; open leaderboard screen.	Leaderboard shows the new entry with correct name, level, lines, and score, in correct sorted order.	

1.5 White-Box Test Cases (Backend API)

To avoid crowding, the white-box tests are summarized in a four-column table: ID, requirements, a short description (including source, inputs, and rationale), and expected output.

1.6 Regression Testing

Any time we change core game logic (spawning, collisions, scoring, levels, game over) or leaderboard behavior, we re-run the tests:

- All UI tests BB–001 through BB–014.
- All API tests WB–001 through WB–006.

For each test run, we keep a simple log that includes:

- Date of the run,
- Version tag of the build (for example, `v1.3.0`),
- List of test cases executed,
- Any failures and a pointer to the related bug or issue ID.

This helps us to repeat the same test sets later, after bug fixes or refactoring.

ID	Req.	Test Description	Expected Output
WB–001	L1.1, L1.2	**Source:** `getLeaderboard`. Pre-seed DB with scores 100, 200, 150. Call `GET /leaderboard` with no query parameters.	Response contains entries ordered 200, 150, 100; each entry has correct `rank`, `name`, `level`, `lines`, and `score`.
WB–002	L1.1, L1.2	**Source:** `getLeaderboard`. With same dataset, call `GET /leaderboard?limit=1&offset=1`.	Response contains exactly one entry (second-best score) with `rank = 2`.
WB–003	L2.1, L2.2	**Source:** `basicTamperCheck`, `addToLeaderboard`. Compute valid HMAC `sec` for a payload; send `POST /leaderboard` with matching data and `sec`.	Request returns success; new row is inserted into database with the given fields and the same `sec`.
WB–004	L2.1, L2.2	**Source:** `basicTamperCheck`. Send `POST /leaderboard` with a random or incorrect `sec`.	Request returns HTTP 403; no new row is inserted into the database.
WB–005	L2.3	**Source:** `addToLeaderboard`. Send a valid `POST /leaderboard` with some `sec`; repeat the request with the same `sec`.	First request succeeds; second request returns a conflict-style error and the database still has only one row with that `sec`.
WB–006	L2.4	**Source:** startup / lifespan logic. Attempt to start the app without the required environment secret.	Application startup fails with a clear error message; server does not run in an insecure state.

2. Testing and Results

Because this problem is tricky and depends on removing exactly the right edges, I spent extra time designing test cases to cover as many situations as possible. Table summarizes the core test cases I used. All of these test cases passed with my final implementation.

Test #	Purpose	Expected Output	Result
1	Unique path only; removing shortest path leaves no path	-1	Passed
2	Two shortest paths removed; find heavier alternate path	7	Passed
3	Multiple shortest paths with shared prefix edges	6	Passed
4	Shortest-path subgraph with cycles; BFS must not loop	Valid almost shortest or -1 (by construction)	Passed
5	Many edges but every S–D path uses a shortest-path edge	-1	Passed
6	D unreachable from S even before removals	-1	Passed
7	Smallest graph ($N = 2$) with a single edge $S \to D$	-1	Passed
8	Very cheap shortest path and very expensive alternate path	Large finite distance	Passed
9	Multiple outgoing edges from S; only some removed	Correct almost shortest distance	Passed
10	Random medium-sized graphs for stress and performance	Finite distance or -1	Passed

2.1 Basic Sanity Tests

Test 1: Single simple path

- Graph: $0 \to 1 \to 2 \to 3$ with weights $(1, 1, 1)$.
- $S = 0$, $D = 3$.
- There is only one path from 0 to 3, and it is also the shortest path.
- After removing all shortest-path edges, no path remains.
- **Expected output:** -1.

This checks that the code correctly removes all edges on the unique shortest path and handles the "no almost shortest path" case.

Test 2: Simple alternative path

- Edges:
  - $0 \to 1$ (1), $1 \to 3$ (3),
  - $0 \to 2$ (1), $2 \to 3$ (3),
  - $0 \to 4$ (2), $4 \to 3$ (5).
- $S = 0$, $D = 3$.
- Shortest paths: 0-1-3 and 0-2-3 with a total cost of 4.
- These edges are removed; the remaining path is 0-4-3 with a cost of 7.
- **Expected output:** 7.

This verifies that the algorithm removes *all* shortest path edges (both paths of cost 4) and still finds the correct almost shortest-path.

2.2 Multiple Shortest Paths and Shared Prefixes

Test 3: Shared start segment

- Edges:
  - $0 \to 1$ (1),
  - $1 \to 2$ (1), $1 \to 3$ (1),
  - $2 \to 4$ (1), $3 \to 4$ (1),
  - $0 \to 5$ (5), $5 \to 4$ (1).
- $S = 0$, $D = 4$.
- Shortest paths: 0-1-2-4 and 0-1-3-4, both with a cost of 3.
- All edges on these shortest paths, including the shared prefix edge $0 \to 1$, must be removed.
- Remaining path: 0-5-4 with a total cost of 6.
- **Expected output:** 6.

This test checks that shared prefix edges are marked as critical and removed from all shortest paths that use them.

Test 4: Shortest-path cycles

- A small graph where some shortest paths contain a cycle of zero net extra cost (for example, going around a 2-edge loop with the same total weight as a single edge).
- The algorithm should still mark all edges in any shortest path but should not get stuck in the backward BFS.
- The visited array in the BFS prevents infinite loops.

2.3 No Almost Shortest Path Cases

Test 5: Many edges but no alternative path

- Build a graph where there are several shortest paths from S to D, but *every* path from S to D uses at least one shortest path edge.
- After removing all critical edges, D becomes unreachable.
- **Expected output:** -1.

This checks that the answer is correctly reported as -1 even when the original graph is dense.

Test 6: Target unreachable from the start

- S and D are in different components; there is no path even before any removal.
- **Expected output:** -1 (the first Dijkstra already gives the distance ∞).

2.4 Edge-Case and Boundary Tests

Test 7: Smallest possible graph

- $N = 2$, $M = 1$, edge $0 \to 1$ with some positive weight.
- $S = 0$, $D = 1$.
- The only path is the shortest one; removing it leaves no alternative.
- **Expected output:** -1.

This checks the behavior at the lower bound of the input size.

Test 8: Heavy alternate path

- There is a very cheap shortest path and also a very expensive alternate path.
- After removing the cheap shortest-path edges, the expensive path should still be found and its large cost should be reported correctly.

This helps catch any mistakes with the `INF` constant or integer overflow.

Test 9: Multiple edges out of the source

- Several edges start directly from S with different weights.
- Some are used on the shortest paths and must be removed; others should remain valid for the second Dijkstra.

This confirms that removal is done per edge, not per node, and that the second Dijkstra still explores the remaining edges from S.

2.5 Integration and Performance Tests

Test 10: Random medium-sized graphs

- Generate random graphs with N of up to a few hundred nodes and M within the problem limits.
- Run the program to ensure that it finishes quickly and does not run out of memory or memory.

These tests focus on stress and integration, but combined with the earlier targeted cases, they provide broad coverage of:

- unique vs. multiple shortest paths,
- existence vs. non-existence of an almost shortest path,
- edge-removal correctness,
- performance under realistic constraints.

3. Known Bugs and Issues

At the time of writing, we track any remaining bugs or limitations in a small table. If there are no known bugs, we can state that explicitly. Otherwise, each row provides an ID, area, description, impact, and any workaround.

ID	Area	Description	Impact	Workaround / Notes
KB–01	Gameplay	Occasionally, rotation near the stack can feel "stiff" and may not wall-kick as expected for certain shapes.	Low	Player can move one cell away from the wall before rotating. Future work could refine the rotation system.
KB–02	Performance	On very low-end machines, very high levels (fast gravity) can cause minor frame drops.	Medium	Closing other heavy browser tabs usually fixes the issue. Performance tuning is possible in later versions.
KB–03	Leaderboard	Leaderboard is local to this deployment and does not sync across multiple servers or devices.	Low	This behavior is acceptable for the course project; global sync is out of scope.

If by submission time all known issues are fixed, we can instead state:

> At this time, there are no known open bugs. All released functionality passes the tests listed in Section 1.