AI Project Phase 2 Report: Heuristic Search Agent for ShoverWorld

January 3, 2026

Abstract

This report documents the design and implementation of a Phase-2 AI player for the ShoverWorld puzzle environment. The agent solves levels by selecting a box and applying a push direction, with the objective of removing all boxes while managing limited energy (stamina). Two classical state-space search algorithms are supported (A* and RBFS), with emphasis on a robust A* configuration driven by strong domain heuristics. Key contributions include: (i) a compact state representation tailored to selection-based pushes, (ii) a distance-to-removal heuristic computed via grid BFS, (iii) search decomposition into sub-goals to keep planning feasible, (iv) action tie-breakers that prefer continuing to push the same currently-moving box when beneficial, and (v) a rule-based "nearly complete square" completion mechanism that triggers *Hellify* or *Barrier Maker* specials at high value moments.

Contents

1 Problem Overview **3**
- 1.1 Cell Encoding and Board Size . . . 3
- 1.2 Actions . . . 3

2 Environment Dynamics and Scoring **3**
- 2.1 Push Costs and "Stationary" Cost . . . 3
- 2.2 Rewards and Stamina Updates on Removal . . . 4
- 2.3 Special Actions . . . 4

3 State Representation **4**
- 3.1 Sub-goal Decomposition . . . 5

4 Search Algorithms **5**
- 4.1 A* Search . . . 5
- 4.2 RBFS . . . 5

5 Heuristic Design for A* **5**
5.1 Distance-to-Removal Heuristic (BFS on the Grid) . . . 5
5.2 Blockage and Congestion Penalties . . . 6
5.3 Square Opportunity Term . . . 6
5.4 Final Heuristic . . . 6

6 Action Ordering and Behavioral Rules **6**
6.1 Continue-Push Preference (Moving Box Priority) . . . 6
6.2 Nearly Complete Square Completion . . . 7
6.3 Fail-safe Fallback (Never "Stuck" at a Corner) . . . 7

7 Implementation Details **7**
7.1 Planner Loop . . . 7
7.2 Engineering for Runtime Constraints . . . 7

8 Experimental Methodology **8**
8.1 Test Maps . . . 8
8.2 Evaluation Metrics . . . 8
8.3 Reproducibility . . . 9

9 Discussion **9**
9.1 Why Hellify Matters . . . 9
9.2 Why Barrier Maker is Used Conservatively . . . 9
9.3 Common Failure Modes and Mitigations . . . 9

10 Conclusion **10**

A Appendix A: Action Encoding Table **10**

B Appendix B: Example Visualization **10**

1 Problem Overview

ShoverWorld is a grid-based puzzle in which each cell can be empty, contain a box, be a barrier, or be lava. The player does **not** navigate an avatar around the board; instead, each move **selects a box cell** and issues a direction. The environment then applies a push to the selected box; if multiple boxes are aligned, a chain of boxes may be pushed. The goal is to remove **all** boxes (win). If stamina reaches zero while boxes remain, the agent loses.

1.1 Cell Encoding and Board Size

The implementation supports both the ASCII-map format used by the GUI and the numeric format used in the provided evaluation maps:

- 0: empty
- 1: box
- 100: barrier
- -100: lava

The default board size required for Phase 2 is 13×9 (13 columns, 9 rows).

1.2 Actions

Each step returns an action in the format:

$$((r, c), z)$$

where (r, c) is the selected cell and z encodes the operation:

- $z \in \{0, 1, 2, 3\}$: push direction (Up, Right, Down, Left)
- $z = 4$: *Barrier Maker* (convert a full $n \times n$ square of boxes into barriers)
- $z = 5$: *Hellify* (convert an $n \times n$ square into lava)

2 Environment Dynamics and Scoring

2.1 Push Costs and "Stationary" Cost

A push has a stamina (energy) cost composed of:

- **Base force**: $10 \times L$, where L is the number of pushed boxes in the chain.
- **Initial force**: 40 if the selected box is considered *stationary* at the moment of selection.

Intuitively, the initial force discourages constantly starting new pushes. If the agent continues to push the same box while it is still "moving" (domain-dependent), the environment should not re-charge the initial force.

2.2 Rewards and Stamina Updates on Removal

Removing a box (e.g., pushing into lava or out of the board when edges behave as lava) provides:

$$\Delta\text{reward} = +40 \quad \text{and} \quad \Delta\text{stamina} = +40.$$

In addition, the implementation ensures that the **reward function reflects the push cost as well**:

$$\Delta\text{reward} = 40 \cdot (\#\text{removed}) - \big(10 \cdot L + 40 \cdot \mathbb{1}[\text{stationary}]\big).$$

This makes the immediate reward consistent with the stamina bookkeeping and avoids misleading reward signals.

2.3 Special Actions

When an $n \times n$ perfect square of boxes is detected, special actions become available:

- **Barrier Maker:** converts the square to barriers and grants $\Delta\text{stamina} = 10n^2$.
- **Hellify:** converts the square to lava (strategically useful for removing nearby boxes).

Because these actions may fundamentally change the board topology (and therefore future removals), the agent reasons about them explicitly rather than treating them as cosmetic.

3 State Representation

A naive state representation includes the full grid, stamina, step count, and "moving" metadata. However, exact modeling is intractable for difficult boards. We therefore use an **approximate but practical** representation for planning:

- Grid occupancy (boxes, barriers, lava).
- A lightweight stamina estimate (used as a pruning and tie-breaking signal, not as an exact value function).
- Last action summary: last selected cell and direction (to support the "continue pushing" preference).

3.1 Sub-goal Decomposition

A major scalability improvement is to decompose planning into repeated **sub-goals**:

> *Plan until at least one box is removed, execute the plan prefix, update the board, then replan.*

This dramatically reduces search depth and keeps A* expansions bounded, while still producing consistent progress toward clearing the board.

4 Search Algorithms

4.1 A* Search

A* maintains an open set ordered by:

$$f(s) = g(s) + w \cdot h(s)$$

where $g(s)$ is the accumulated estimated cost and $h(s)$ is a heuristic estimate of remaining work. A modest weight $w > 1$ can significantly improve speed (Weighted A*), at the cost of possible suboptimality. In ShoverWorld, *optimality is not required* and fast completion is preferred; thus a small weight is acceptable.

4.2 RBFS

Recursive Best-First Search is also supported, primarily as a baseline. RBFS is memory efficient but can revisit subtrees and can be slow on wide branching problems. In this project, RBFS is guarded by explicit limits (depth, expansions, time), and A* is treated as the primary solver for challenging boards.

5 Heuristic Design for A*

The effectiveness of A* in this domain is almost entirely determined by $h(s)$ and action ordering. The final agent uses a **multi-component heuristic** with strong domain guidance.

5.1 Distance-to-Removal Heuristic (BFS on the Grid)

For each box, we estimate the minimum number of pushes needed to remove it by computing distance to a **kill position**. A kill position is a cell adjacent to lava, or (if the environment treats edges as lava) any border-adjacent removal path. We compute a BFS distance field on the grid treating barriers as obstacles and empties as traversable. Then:

$$h_{\text{kill}}(s) = \sum_{b \in \text{boxes}} \min_{p \in \text{killPos}} d(b, p).$$

This is not an admissible heuristic in a strict theoretical sense (pushing is not the same as walking), but it is a strong *progress metric* that correlates well with actual removals.

5.2 Blockage and Congestion Penalties

A common failure mode is wasting actions on heavily blocked boxes. We add a penalty for boxes surrounded by barriers/other boxes in all four directions:

$$h_{\text{jam}}(s) = \#\{b : \text{all adjacent cells are blocked}\}.$$

This encourages the agent to first create space, open corridors, or form lava zones via Hellify.

5.3 Square Opportunity Term

Perfect squares are valuable because they enable Hellify/Barrier Maker. We incorporate a square term:

$$h_{\text{sq}}(s) = -\alpha \cdot \max_{q \in \text{availableSquares}} (n_q^2),$$

so larger available squares reduce heuristic value (i.e., become more attractive). This helps the solver quickly exploit large squares instead of ignoring them.

5.4 Final Heuristic

The final A* heuristic is:

$$h(s) = h_{\text{kill}}(s) + \beta \cdot h_{\text{jam}}(s) + h_{\text{misc}}(s) + h_{\text{sq}}(s),$$

where h_{misc} includes small tie-breaking preferences (e.g., prefer actions that remove a box now).

6 Action Ordering and Behavioral Rules

Search alone is not enough; good action ordering reduces branching and prevents pathological loops.

6.1 Continue-Push Preference (Moving Box Priority)

To reduce repeated stationary costs, after a push the agent checks whether it can **continue pushing the same selected box** in the same direction. Let the last selected box be at (r, c) and direction be $\Delta = (dr, dc)$. The next position of that box is $(r+dr, c+dc)$. The agent attempts:

$$((r + dr, c + dc), z_\Delta)$$

only if the action is valid and either:

- it removes at least one box immediately, or
- it does not worsen the heuristic compared with the planned alternative.

This implements the user-requested behavior: *prefer the currently moving box* when it is possible and better, without blindly switching to unrelated moved boxes.

6.2 Nearly Complete Square Completion

Before running search at each decision point, the agent scans the board for **nearly complete** squares: an $n \times n$ region with exactly one missing box cell (and no lava/barriers inside). If the missing cell can be filled in **one push** by sliding a box from just outside the square into the missing cell, the agent performs that push. On the following step, the square becomes perfect and a special action can be applied:

- Prefer Hellify for $n \geq 3$
- Prefer Barrier Maker for $n = 2$ (optional, conservative)

This rule converts multi-move strategic planning into a simple high-value tactical pattern.

6.3 Fail-safe Fallback (Never "Stuck" at a Corner)

A known bug in earlier versions caused repeated selection of the top-left corner and invalid actions after the plan was exhausted. The final agent includes a fail-safe:

> If no valid planned action exists, select any existing box and choose a direction with a valid push; if none exist, return a no-op.

This guarantees the agent continues to the end of the episode even if it is losing.

7 Implementation Details

7.1 Planner Loop

Algorithm 1 summarizes the overall decision-making logic.

7.2 Engineering for Runtime Constraints

To keep runtime within a few minutes on typical PCs:

- A* is run with a strict expansion budget per sub-goal.
- The branching factor is reduced by preferring actions that interact with boxes near kill positions.
- Replanning occurs after each removal rather than attempting a full-horizon plan.

Input: Current observation *obs*, info dict
Output: Action $a = ((r, c), z)$
Decode grid from *obs*
Update internal memory (last action, moved boxes)
if *A special action is immediately attractive (perfect square available)* **then**
  return Hellify/Barrier Maker
else
  if *A nearly complete square can be completed in one push* **then**
    return the completing push
  else
    if *Continue-push is valid and better* **then**
      return continue-push action
    else
      Run bounded A* to remove at least one box; store plan
      if *plan non-empty* **then**
        return plan.pop(0)
      else
        return greedy-valid action
      end
    end
  end
end

Algorithm 1: High-level Control Loop (A*)

8 Experimental Methodology

8.1 Test Maps

Five numeric maps were provided as benchmarks (map1–map5), using the encoding $0, 1, 100, -100$. The maps include a mixture of:

- interior barriers that create corridors and bottlenecks,
- lava patches that act as local removal sinks,
- large box blocks that can form perfect squares.

8.2 Evaluation Metrics

We evaluate:

- **Solved?** (all boxes removed)
- **Number of steps**
- **Final stamina**
- **Time to first removal** (proxy for search effectiveness)

8.3 Reproducibility

The repository includes:

- `run_player_ai.py` to run headless and print step logs.
- `watch_ai_steps.py` to visualize selection-based play without an avatar walking.

For consistent results, delete `__pycache__` when swapping code versions and run with the same algorithm and limits.

9 Discussion

9.1 Why Hellify Matters

When the board contains large solid blocks of boxes, pushing them one-by-one into distant lava is inefficient. Hellify converts a perfect square into a lava region, which:

- creates new kill positions close to remaining boxes,
- reduces path length in h_{kill},
- increases the number of immediate-removal actions available.

Therefore, the agent is designed to *actively create* and *exploit* square patterns instead of treating them as incidental.

9.2 Why Barrier Maker is Used Conservatively

Barrier Maker can increase stamina, but it also creates permanent obstacles. On some boards, converting a 2×2 block too early may trap remaining boxes behind barriers. For this reason, the agent prioritizes:

- Hellify for $n \geq 3$,
- Barrier Maker only when it clearly improves mobility or is required by a specific level design.

9.3 Common Failure Modes and Mitigations

- **Search blow-up:** mitigated by sub-goal decomposition and budgets.
- **Looping near corners:** mitigated by the fail-safe valid-action selector.
- **Wasting initial force:** mitigated by continue-push preference.
- **Ignoring large squares:** mitigated by h_{sq} and the nearly-complete square completion rule.

10 Conclusion

This project implements a strong, practical heuristic-search agent for ShoverWorld Phase 2. The core design principles are: exploit domain structure (lava adjacency, squares), reduce planning horizon via sub-goals, and add carefully constrained behavioral rules that respect the environment's selection-based mechanics. A* with the proposed heuristic and action ordering provides reliable performance on challenging maps while staying within practical runtime limits.

A Appendix A: Action Encoding Table

z	Meaning
0	Push Up
1	Push Right
2	Push Down
3	Push Left
4	Barrier Maker
5	Hellify

Table 1: Action encoding used by the agent and runner scripts.

B Appendix B: Example Visualization

Figure 1 shows an example board state in the selection-based viewer.

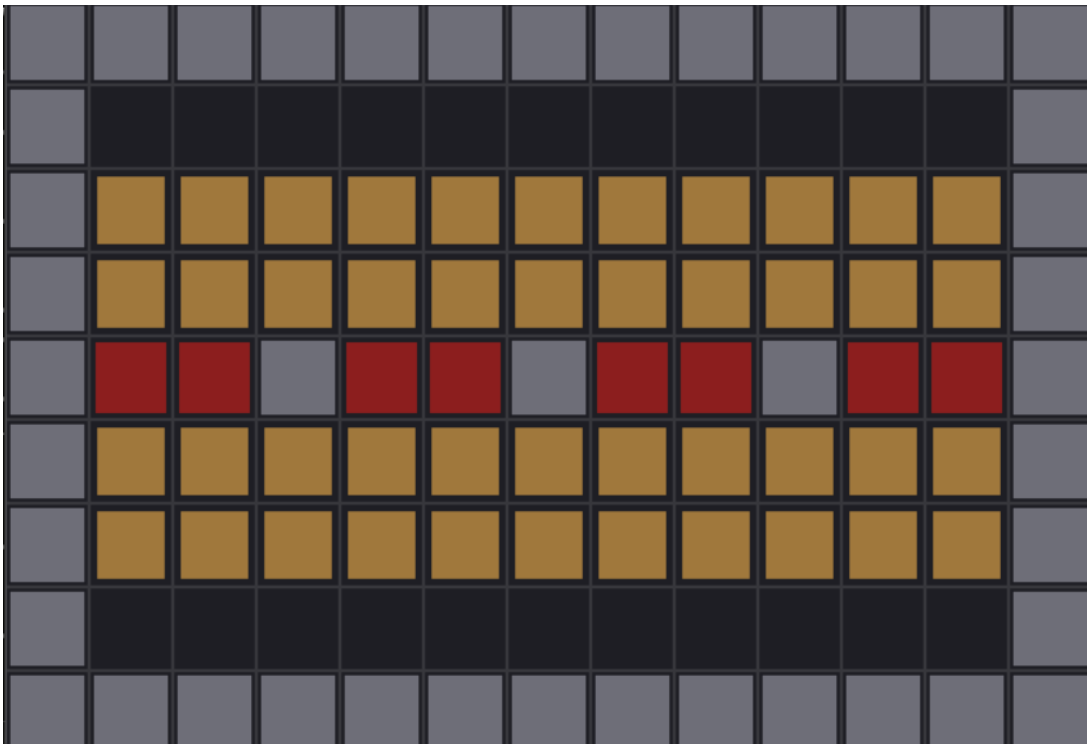

Figure 1: Example state snapshot from the step viewer.