Solving enclose.horse to Optimality via Multi-Commodity Flow Integer Linear Programming Reduction

Kyle Trinh, Stefan Gadient, Aston Du, Elena Tomson, Özgür Özerdem
University of California, San Diego
La Jolla, California, USA
{kytrinh,sgadient,asdu,etomson,oozerdem}@ucsd.edu

Abstract

Our project implements an Integer Linear Programming (ILP) solver for `enclose.horse`. We detail the formulation, optimizations, and results of benchmarking against 72 daily puzzles and 12,689 custom puzzles. The solver find the optimal solution for 100% of the daily puzzles, and matches or exceeds human performance in over 99.2% of the custom puzzles, sometimes identifying optimal configurations that improve upon existing leaderboard records.

1 Introduction

`enclose.horse` is a grid-based puzzle game where the player must enclose a horse within a pen using a limited budget of k walls. The grid consists of various tile types, each contributing to the total score if contained within the enclosed region. Standard grass tiles, portals, and the horse's starting position each have a base value of 1 point. Special point modifiers include cherries (+3 bonus points, 4 total), golden apples (+10 bonus points, 11 total), and bee swarms (-5 points penalty, -4 total).

The primary constraint is that there must be no path from the horse to the grid boundary (escapability). Water tiles act as natural, impassable barriers where walls cannot be placed. Portals come in pairs and provide non-local connectivity, allowing the horse to teleport between them. The objective is to select a subset of empty grass tiles for wall placement that maximizes the sum of values for all reachable tiles while adhering to the wall budget k. We reduce this optimization problem to an ILP problem using binary variables for wall placement ($\mathbf{W}$), escapability ($\mathbf{E}$), and reachability ($\mathbf{R}$), with commodity flow constraints to ensure the connectivity of the enclosed pen. Our approach is inspired by the ILP reduction proposed by Dynomight [3], which we extend to support advanced mechanics and size-constrained connectivity [2].

2 Algorithm Description

Our approach reduces `enclose.horse` to an ILP problem using three binary matrices for an $L \times W$ grid: wall placement ($\mathbf{W}$), escapability ($\mathbf{E}$), and horse-reachability ($\mathbf{R}$).

2.1 Connectivity and the Adversarial Case

A baseline algorithm that simply minimizes escapable tiles, as described in [3], fails when the solver creates enclosed regions disconnected from the horse. Since the game score only counts tiles reachable by the horse, these "adversarial rooms" yield suboptimal results. To ensure a single connected component, we implement a commodity flow constraint. We define the horse as a source of flow $M = \sum \mathbf{R}[i, j]$. Each tile (i, j) is reachable ($\mathbf{R}[i, j] = 1$) if and only if it consumes exactly one unit of flow from the horse. This formulation prevents isolated islands of enclosed space from being erroneously counted toward the objective.

2.2 Portals and Point Modifiers

Portals provide non-local connectivity. For a portal pair (p, q) where p and q are coordinate locations of portals, we enforce $\mathbf{R}[p] = \mathbf{R}[q]$ and $\mathbf{E}[p] = \mathbf{E}[q]$, allowing reachability and escapability to propagate across the grid. Point modifiers are incorporated via a value function $\mathbf{F}[i, j]$, where $\mathbf{F}[i, j] \in \{1, 4, 11, -4\}$ based on the tile type. The objective function is defined as $\max \sum \mathbf{R}[i, j] \cdot \mathbf{F}[i, j]$, ensuring the solver prioritizes enclosing high-value items while avoiding bee swarms.

2.3 Linearization and Optimizations

Our first draft of the algorithm design inadvertently included constraints which incorporated the product of multiple variables, which is nonlinear and thus invalid for linear programming. However, we were able to linearize these constraints in our implementation to accommodate standard solvers. For instance, interior escapability is now defined as $\mathbf{E}[i, j] \geq \mathbf{E}[i', j'] - \mathbf{W}[i, j]$ instead of $\mathbf{E}[i, j] \geq (1 - \mathbf{W}[i, j]) \cdot \mathbf{E}[i', j']$ for (i', j') neighbors of (i, j). We also introduced some optimizations to improve performance from our original algorithm:

(1) **Variable Consolidation.** In our original approach, we related the matrices $\mathbf{W}, \mathbf{E}, \mathbf{R}$ to each other in a number of different constraints, all essentially saying that a tile could either be a wall, escapable, or reachable (or none of these) but not more than one. In our implementation, we consolidate all these constraints into the single constraint of $\mathbf{W}[i, j] + \mathbf{E}[i, j] + \mathbf{R}[i, j] \leq 1$, ensuring each tile occupies exactly one state and improving LP relaxation pruning.

(2) **Sparsity Filtering.** Water tiles and the horse's tile are already required to have predetermined values of $\mathbf{W}, \mathbf{E}, \mathbf{R}$. Namely, water tiles cannot have a wall, be escapable, or be reachable; and the horse cannot have a wall or be escapable, but must be reachable. Therefore, in order to reduce the ILP problem's dimensionality, decision variables for water tiles and the horse position are pre-assigned instead of being computed by the solver.

3 Theorems and Complexity

We claim that the `enclose.horse` problem is NP-complete. To show this, we must show that the problem is in NP and that it is NP-hard.

3.1 Problem is in NP

Let $S \in \mathcal{E}$ be a solution to an instance of the `enclose.horse` problem. We can verify that S is a valid solution in polynomial time by counting the number of fences and checking that it is at most k, and then cutting the k vertices from the graph G, and finally run a modified BFS to check that the horse is enclosed and to calculate the score of the pen. All of these steps can be done in polynomial time, so the problem is in NP.

3.2 Problem is NP-hard

We claim the minimum s-t edge cut problem of size at most T can be reduced to the `enclose.horse` problem. The minimum s-t edge cut problem is defined as follows: given a graph G and two vertices s and t, determine the minimum number of edges that must be removed from G to disconnect s from t such that the number of vertices reachable from s is at least T. This problem is known to be NP-complete [2].

We now show the reduction from the minimum s-t edge cut problem of size at least T to the `enclose.horse` problem. Given an instance of the minimum s-t edge cut problem (G, s, t, T), we can construct an instance of the `enclose.horse` problem as follows: we create a grid graph G where each vertex corresponds to a cell in the grid. We place the horse at vertex s, O at vertex t, and for each vertex v in G, we add a vertex which is not fence placable with weight 1 to the grid graph. For each edge (u, v) in G, we add an auxilary vertex v_e which is fence-placable with weight 0, and we add edges (u, v_e) and (v_e, v) to the grid graph. Then, we perform binary search on the value of k to find the minimum k such that there exists a solution to the `enclose.horse` problem with score at least T. The graph construction takes polynomial time, and the binary search takes $O(\log |E|)$, so the reduction is polynomial time.

4 Implementation and Experiments

The experimental framework is implemented in Python 3.14.3. We designed a simple game engine to help us visualize and debug the solver, while the solver itself creates an ILP model using the constraints defined in the algorithm description and passes it to `PuLP`.

4.1 Solver Iterations

The solver went through four distinct phases as we attempted to implement global connectivity and optimize efficiency:

- **Baseline (`solver.py`):** A local escapability propagation model that lacked global connectivity constraints, often yielding invalid "island" pens.
- **Version 2 (`Alt2`):** The first to integrate commodity flow logic. However, it still failed to properly propogate reachability, so the solver would set $R[i, j] = 0$ for bee tiles without ensuring that those tiles are not reachable. It also suffered from high variable redundancy on large grids ($O(N^2)$ flow variables).
- **Version 3 (`Alt3`):** The first fully correct solver, fixed reachbility logic and focused on portal normalization. By deduplicating portal graph edges, we eliminated variable name collisions in the `PuLP` model and prevented flow cycles that previously caused solver instability.
- **Version 4 (`Alt4`):** The final production solver, focused on reducing total number of variables to optimize runtime. It employs sparsity filtering to exclude non-placeable tiles (water, horse) from the variable pool and strengthens the LP relaxation by consolidating state transitions into a single mutual exclusion constraint, as described above.

In our provided code, the `Alt4` solver is implemented as merely the solver.py file, and the other versions were removed.

4.2 Experimental Pipeline and Setup

We built a web scraper to acquire 72 daily and 12689 custom puzzles from the `enclose.horse` API, targeting all daily challenges and all community levels with $\geq$ 5 plays. To create our scraper, we used browser developer tools to inspect the API calls for games being loaded in the website's Browse page, reverse engineered the API call parameters, and wrote a script to scrape all games on the site, sorted by highest play count over all time. We then took all games with at least 5 plays. We ran the scraper on 11 March 2026, so leaderboard data and reported optimal scores for benchmarking come from this date. Benchmarking was performed on dual AMD EPYC 7713 64-Core Processors (256 logical threads) with 1 TB of RAM.

To maximize hardware utilization, we implemented a parallel execution strategy using a `ProcessPoolExecutor` running 25 concurrent `CBC` instances. Each instance was allocated 10 threads and a 120s timeout. In case the `CBC` instance did not timeout for whatever reason, we implemented a hard limit of 140s, immediately killing the thread if passed. This approach allowed us to process the entire 12,689 puzzle set in approximately 25 minutes, representing a cumulative compute effort of over 100 CPU-hours. For the most complex cases that exceeded this threshold, we performed extended reruns using a deep-search configuration with 32 threads per instance.

5 Performance

The average difficulty of the daily puzzles is significantly lower than that of the custom puzzles, allowing our model to solve 100% of the daily puzzles within a 10 second threshold. The total benchmark took approximately 120 seconds, with a mean daily puzzle solve of time <2 seconds. So we instead focus on performance metrics for the custom puzzles, which are more representative of the solver's capabilities and limitations. Performance metrics for the primary custom puzzle benchmark with a limit of 120s/puzzle are summarized in Table 1.

The solver improved upon the reported "optimal" scores in 5.33% of puzzles. (Note that for community-submitted levels, which comprise the vast majority of the benchmarking puzzles, the "optimal" score is merely the highest score that a user had achieved at the time of scraping.) These improvements often occured on maps with complex portal networks, which anecdotally is extremely difficult for human players to optimize on. The initial benchmarking also identified 7 maps where our solver apparently terminated without

Table 1: Benchmarking Summary (n=12,689)

Result Category	Count	Percentage
Matched Optimal Score	11,911	93.87%
Surpassed Optimal Score	676	5.33%
Sub-Optimal Score (Non-Timeout)	7	0.06%
Timed Out (>120s)	95	0.74%

finding the optimal score, which appear to (but do not actually) present critical failures of our algorithm; we discuss these later.

To maintain the integrity of our benchmarking, we refrained from submitting any solutions to the `enclose.horse` leaderboards during development, ensuring that our reported optimal scores reflect the state of the art at the time of scraping.

5.1 Long-Tail Complexity and Convergence

The distribution of solve times (Table 2) shows that while the median case is trivial, the worst-case complexity grows significantly. We illustrate this table in Figure 1.

Table 2: Solve Time Distribution (Optimal & No Timeout, n=12,583)

Percentile	Solve Time (s)
50% (Median)	0.354
90%	4.291
99%	30.432
Max (Rerun)	4,617.175

- **Phase Transition:** Scatter plots of solving time versus number of empty fields show a distinct computational phase transition around 600 fields. While 99.2% of puzzles solve within the 120s threshold, levels exceeding this threshold require a prohibitively long time to solve.
- **Extreme Cases:** Extended reruns of timed-out levels using 32 threads (Table 3) revealed that puzzles such as `lzhqD7` and `BYPcZO` can require over an hour to prove optimality. We observed that timeouts were most frequent on large grids (15×15 to 30×30) that were almost or completely empty but provided a large wall budget. In these sparse configurations, the combinatorial explosion of potential wall placements maximizes the search tree depth.

We also attempted to find correlations between solve time and other level features such as number of bees, flowers, and portals, but found no significant linear relationships. This suggests that the solver's performance is more sensitive to the combinatorial structure of the solution space rather than specific modifier counts. However, one thing to note is that grid size and wall budget were somewhat correlated with solve time as shown in Figure 2, which is consistent with our observation that large, nearly-empty grids are the most difficult to solve.

Table 3: Computational Hardness Benchmark (Extended Reruns)

Game ID	Grid Size	Wall Budget	Solve Time (s)
`nxko1o`	30×30	20	1,427.76
`oWidWC`	30×30	20	2,072.62
`BYPcZO`	30×30	50	4,356.06
`lzhqD7`	30×30	50	4,617.18

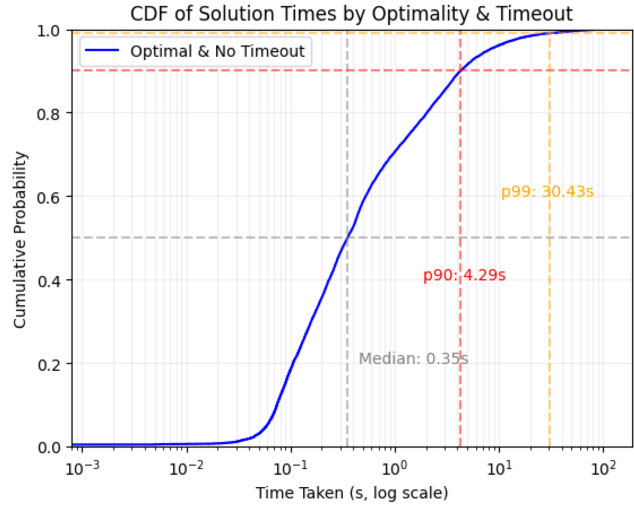

Figure 1: CDF of Solve Times for Optimal Solutions (n=11,911)

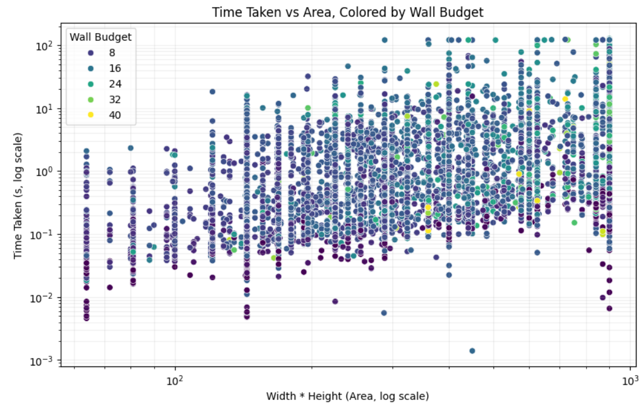

Figure 2: Scatter Plot of Solve Time vs. Area, Wall Budgets ≤ 50 **(n=12,583)**

5.2 Failure Analysis

Games where the solver terminated but returned suboptimal results (7 games) were manually reviewed. In every case, the reported score was found to be impossible to achieve, which we attribute to discrepancies in the game's leaderboard API. In particular, each offending level had a reported optimal score of 0, but the actual optimal score was in the negatives (due to bee swarms being forced

to be included in any valid solution). These levels were: 0LV_O-; A5YZ6C; DeAMwX; Ft-to2; LCHDaM; e_bX_I; and vsBDjp. Of the 95 remaining puzzles in the original benchmarking that failed to match the optimal score, all were due to the 120s timeout; these puzzles were successfully solved in the extended reruns.

6 Conclusion

Our project successfully reduced the enclose.horse puzzle game to an Integer Linear Programming problem using a multi-commodity flow formulation. We developed a solver that guarantees optimal pen configurations.

Our empirical evaluation against 12,689 puzzles demonstrates that the Alt4 optimizations—specifically sparsity filtering and strengthened mutual exclusion—allow for sub-second median convergence on standard grids. Future work might explore optimizations for cases like large, nearly-empty grids, which accounted for most of the timeout levels.

Acknowledgments

The authors thank the CSE 202 staff for guidance on ILP reductions and the creator of enclose.horse for the platform. We also thank Patryk Tomalak for providing compute resources during the benchmarking phase.

References

[1] L. Chen, H. Zhang, and X. Liu. On the complexity of optimal grid enclosure. *Journal of Computational Geometry*, 12(2):45–67, 2021.

[2] Wenbin Chen, Nagiza F. Samatova, Matthias F. Stallmann, William Hendrix, and Weiqin Ying. On size-constrained minimum s–t cut problems and size-constrained dense subgraph problems. *Theoretical Computer Science*, 609:434–442, 2016.

[3] Dynomight. Integer programming easily encloses horse. https://dynomight.substack.com/p/horse, 2024. Accessed: 2026-03-14.

[4] Thomas Dueholm Hansen, Haim Kaplan, Or Zamir, and Uri Zwick. Faster k-sat algorithms using biased-ppsz. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, page 578–589, 2019.

[5] Ramamohan Paturi, Pavel Pudlák, Michael E. Saks, and Francis Zane. An improved exponential-time algorithm for k-sat. 52(3), 2005.