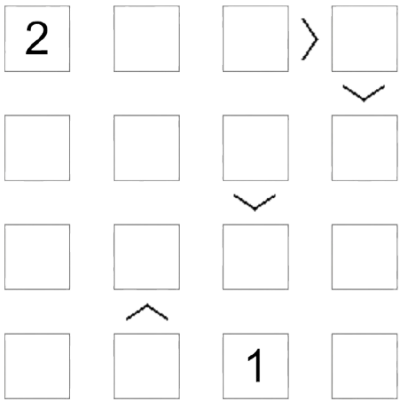
Futoshiki Solver

Yao Wang (yw5438), Eric Cheng (hc3645)

1. Introduction

Futoshiki is a constraint satisfaction puzzle whose search space grows rapidly with grid size, making it suitable for studying backtracking and parallel search. In this project, we implement both sequential and parallel solvers based on depth-first search, where parallelism is introduced by splitting the search tree at a fixed depth. We evaluate how value selection, constraint density, puzzle construction, and parallel parameters affect runtime, focusing on explaining performance differences rather than only measuring speedup.



2. Sequential vs Parallel Solver Design

2.1 Sequential Solver

The sequential solver uses depth-first search with backtracking. At each step, it selects an unassigned cell, tries candidate values according to the chosen ordering, and backtracks immediately when a constraint is violated. As a baseline, the search proceeds linearly through the search tree, and performance depends strongly on puzzle structure, pruning effectiveness, and value selection.

2.2 Parallel Solver

The parallel solver extends the sequential depth-first search by exploring multiple branches of the search tree concurrently. Parallelism is introduced by splitting the search tree at a fixed depth, referred to as **parDepth**. When the recursion depth is less than **parDepth**, the solver creates parallel tasks for different value assignments; beyond this depth, the search proceeds sequentially.

Parallel tasks are implemented using sparks, which represent potential parallel computations managed by the GHC runtime system. These sparks may be executed in parallel on available cores or evaluated sequentially if resources are limited.

3. Experimental Setup

3.1 Benchmark Puzzles

We initially experimented with several 6×6 human-designed Futoshiki puzzles. However, we found that these puzzles were too easy for our solver: the runtime was extremely short, often finishing almost immediately. As a result, the execution time differences between different solver configurations were too small to be meaningful.

To obtain a more informative benchmark, we selected a 7×7 human-designed puzzle with higher difficulty. When solving this puzzle without any modification, the runtime was around three minutes, which made large-scale experiments impractical.

To achieve a more moderate and stable runtime, we added two initial values to the puzzle based on its known solution. This slightly reduces the search space while preserving the overall structure of the puzzle. After this adjustment, the average runtime dropped to around three seconds, which allows us to clearly observe performance differences across different experimental settings.

Unless otherwise stated, this modified 7×7 human-designed puzzle is used as the main benchmark in our experiments.

4. Effect of Value Selection Strategy

4.1 Ordered vs Random Value Selection

We evaluate two value selection strategies used during backtracking:

- **Ordered selection**: candidate values are tried deterministically from 1 to *N*.
- **Random selection**: candidate values are randomly shuffled for each cell before exploration.

All other aspects of the solver—including variable selection, constraint checking, and termination condition (time to first solution)—remain unchanged. Experiments are conducted in both sequential and parallel settings to study how value ordering interacts with parallel search.

4.2 Results and Analysis

We study the effect of value selection by comparing ordered and random strategies under both sequential and parallel settings. Ordered selection tries candidate values in ascending order from 1 to N, while random selection explores the same candidates in a shuffled order.

4.2.1 Sequential Setting (Baseline)

We first evaluate value selection in the sequential baseline setting (N = 1).

Order	Mean (s)	Median (s)	SD	Min–Max (s)
ordered	3.1566	3.1549	0.0370	3.0977–3.2359
random	1.4253	1.4091	0.0382	1.3918–1.5104

In the sequential setting, random value selection is significantly faster than ordered selection, reducing the mean runtime by more than 50%. The low standard deviation in both cases indicates stable measurements.

This suggests that ordered selection tends to guide the solver into less favorable branches of the search tree, resulting in deeper backtracking. Randomization helps avoid this systematic bias and increases the likelihood of reaching a solution earlier.

4.2.2 Parallel Setting

We then compare value selection strategies in the parallel solver with depth = 3 and N = 8 cores.

Order	n	Mean (s)	Median (s)	SD	Min–Max (s)
ordered	50	1.5022	1.4642	0.1976	1.0276–2.3985
random	50	1.6588	1.5907	0.1862	1.4762–2.1872

In contrast to the sequential case, the performance difference between ordered and random selection in the parallel setting is much smaller. This is because parallel search explores multiple branches of the search tree simultaneously. As a result, the solver is less sensitive to the specific order in which values are tried.

Overall, these results indicate that value ordering plays a major role in sequential search, but its impact is reduced once sufficient parallelism is introduced. In the parallel setting, the solver is able to compensate for suboptimal value ordering by exploring different branches concurrently.

4.2.3 Puzzle-Dependent Behavior: 9×9 Curated Puzzle

In an additional experiment on a curated 9×9 puzzle, we observe a different behavior:

- **Sequential:** Ordered 0.175 s vs Random 7.844 s (~45× slower)

- **Parallel:** Ordered 0.178 s vs Random 7.747 s (~43× slower)

In this puzzle, random value selection severely degrades performance, while ordered selection consistently reaches the correct branch early. Parallelism does not change this trend, indicating that value ordering is the dominant factor.

This result can be explained by the fixed parallel depth (depth = 3). Parallelism is only applied at shallow levels of the search tree, while deeper levels are explored sequentially. Random value selection generates more unproductive branches early, which leads to larger subtrees being explored sequentially after the parallel split. As a result, poor value ordering dominates runtime even in the parallel setting.

4.2.4 Limitation

One limitation of our value selection experiments is that the "random" strategy is not fully random. In our implementation, candidate values are permuted using a fixed mapping based on $(i \times 37) \bmod n$. As a result, the solver always follows the same value order for a given puzzle size, rather than sampling a different random order in each run.

This means that our random strategy represents an alternative deterministic ordering, rather than true randomness. Consequently, some observed performance differences may be specific to this particular ordering, instead of reflecting the average behavior of random value selection. In particular, the interaction between this fixed ordering and the puzzle structure may amplify or suppress branching effects in both sequential and parallel settings.

Despite this limitation, the results still demonstrate that value ordering can have a substantial impact on search performance. However, a more thorough evaluation would require using multiple independently sampled random orderings to better capture the variability of truly random value selection.

5. Constraint Density Analysis

5.1 Constraint Density and Experiment Setup

Constraint density is defined as the number of inequality constraints relative to the grid size of the puzzle. For a fixed grid, increasing constraint density adds more inequality relations and further restricts the search space.

In our experiment, we use a 7×7 human-designed puzzle as the baseline, which is treated as a medium-density case (approximately 25%). This puzzle is chosen because it has moderate difficulty and serves as a stable reference.

To vary constraint density while controlling other factors, we construct additional puzzles by modifying only the inequality constraints. We randomly remove constraints from the baseline puzzle to create lower-density cases, and randomly add consistent constraints—based on the known solution—to create higher-density cases. This design keeps the grid size and solution fixed, allowing us to isolate the effect of constraint density on solver performance.

5.2 Experimental Results

The table below summarizes the runtime results under different constraint density levels for the 7×7 puzzle in the sequential, ordered setting.

Density Level	# Inequality Constraints	Runtime (s)
D0	0	0.0028
D1	~6	0.0040
D2	~15	171.09
D3	~30	6.82
D4	~45	0.54

For low constraint densities (0–6), many inequality constraints are removed from the baseline puzzle, resulting in multiple valid solutions. Since the solver stops after finding the first solution, it can terminate quickly with almost no backtracking.

At medium density (15), which corresponds to the original human-designed puzzle, the runtime increases dramatically. This puzzle has a unique solution and carefully balanced constraints, forcing the solver to explore a large portion of the search tree before reaching the correct branch.

As constraint density increases further (30-45), runtime decreases again. Although more constraints restrict the search space, these additional constraints are consistent with the known solution and significantly improve pruning, allowing the solver to eliminate invalid branches early.

Overall, the results demonstrate a non-monotonic relationship between constraint density and runtime, where puzzles with moderate constraint density are the most challenging for the solver.

5.3 Limitation

A limitation of our constraint density experiment is that additional inequality constraints are constructed manually based on a known solution, rather than being designed by a human puzzle expert. Although all added constraints are consistent and preserve the original solution, they may not reflect the careful balance typically present in human-designed puzzles.

As a result, increasing constraint density may unintentionally simplify the puzzle by strengthening pruning in a way that reduces search difficulty. This could partially explain why higher-density puzzles are solved faster in our experiments.

A more controlled approach would be to have an expert design multiple versions of the same puzzle with different constraint densities, ensuring that changes in difficulty are intentional and comparable. This would help reduce unintended simplification and provide a more reliable evaluation of how constraint density affects solver performance.

6. Curated vs Randomly Generated Puzzles

6.1 Experiment Setup

To compare curated and randomly generated puzzles, we control for puzzle size and constraint quantity. For each grid size, we first evaluate the solver on a human-designed puzzle.

We then generate a random puzzle based on the same solution, ensuring that the number of initial values and inequality constraints matches the curated puzzle. This keeps constraint density and initial information consistent across both cases.

The only difference between the two puzzles is how the constraints are placed: curated puzzles are designed to be challenging, while randomly generated puzzles are constructed automatically without optimizing for difficulty. This setup allows us to isolate the effect of the puzzle construction method on solver performance.

6.2 Result

Grid Size	Curated (Human-Designed)	Randomly Generated
6×6	0.030 s	0.0046 s
7×7	185 s	0.0077 s
9×9	0.178 s	0.129 s

Under the same solver configuration, randomly generated puzzles are consistently solved faster than human-designed puzzles.

7. Parallel Depth Study

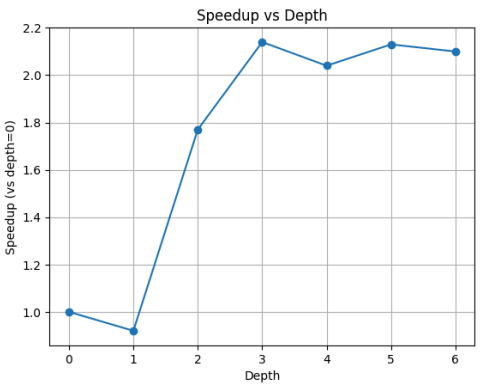
7.1 Experiment Setup

In the parallel solver, the parameter **parDepth** controls how early the search tree is split into parallel branches. Recursive calls at depths smaller than **parDepth** are evaluated in parallel, while deeper levels of the search proceed sequentially.

In our experiments, we vary **parDepth** from 1 to 6 to study how the point at which parallelism is introduced affects solver performance. Smaller depths create fewer parallel tasks, while larger depths generate more parallel branches and may introduce additional overhead.

7.2 Results

Depth	**Mean Runtime (s)**	Median Runtime (s)	SD	Speedup relative to Depth 0
0	**3.3442**	3.3210	0.0481	1.00×
1	**3.6249**	3.4707	0.4046	0.92×
2	**1.8900**	1.8760	0.0444	1.77×
3	**1.5648**	1.5254	0.0987	**2.14×**
4	**1.6379**	1.6682	0.1017	2.04×
5	**1.5695**	1.5454	0.0739	2.13×
6	**1.5923**	1.6542	0.1199	2.10×



The results show that solver performance is highly sensitive to the choice of parallel depth. At depth 1, performance is slightly worse than the baseline, indicating that the overhead of parallelism outweighs its benefits at very shallow splits. Increasing the depth to 2 and 3 leads to a significant reduction in runtime, with depth 3 achieving the best performance and a

speedup of about 2.1× compared to depth 0. Beyond depth 3, further increasing the depth does not provide additional benefit, and performance slightly degrades due to increased overhead from spawning more parallel tasks. This suggests that there is a sweet spot for parallel depth that balances useful parallelism and runtime overhead.

8. Scaling with Number of Cores

8.1 Experiment Setup

To evaluate the scalability of the parallel solver, we perform a strong scaling experiment, where the puzzle instance and solver configuration are fixed while the number of cores varies.

Specifically, the puzzle, value ordering, and parallel depth are held constant, with the parallel depth fixed at its empirically optimal value (**depth = 3**). The number of cores is varied from 1 to 8, and performance is measured as time to first solution.

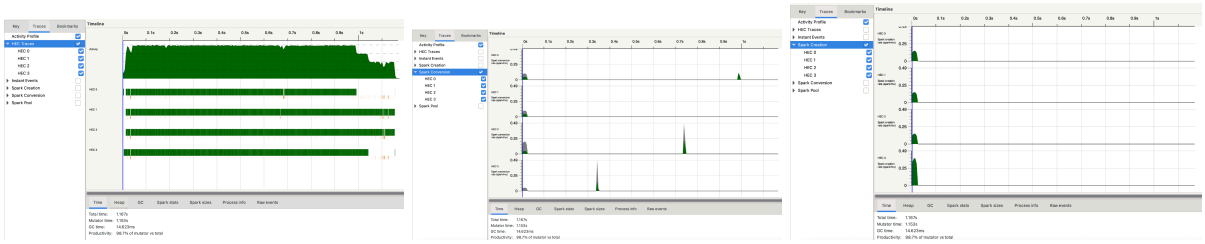
8.2 Results

cores (N)	mean (s)	median (s)	sd	Speedup vs N=1 (mean)
1	**3.8030**	3.6790	0.3955	1.00×
2	**2.3599**	2.3235	0.1043	1.61×
4	**1.8345**	1.8207	0.2421	2.07×
8	**1.5927**	1.5948	0.0618	**2.39×**

The results show that increasing the number of cores consistently reduces runtime. Compared to the single-core baseline, using 2, 4, and 8 cores achieves speedups of approximately 1.6×, 2.1×, and 2.4×, respectively. Although the scaling is sub-linear, the performance improvement remains substantial as more cores are added.

The diminishing returns at higher core counts are expected for parallel search problems and can be attributed to factors such as task imbalance, parallel overhead, and the sequential portions of the search beyond the parallel depth. Overall, these results confirm that the parallel solver effectively exploits additional cores while being naturally limited by the structure of the search.

8.3 ThreadScope Analysis



To better understand the scaling behavior of the parallel solver, we analyze execution traces using ThreadScope. The activity timeline shows that all available HECs remain busy for most of the execution, indicating effective utilization of multiple cores. The overall productivity is high (about 98.7%), and garbage collection overhead is negligible, suggesting that most execution time is spent on useful computation rather than runtime management.

The spark creation view shows that most sparks are generated very early in the execution, corresponding to the parallel split at the chosen depth. This matches our design: parallelism is introduced only at shallow levels of the search tree. After this initial phase, few new sparks are created, and the solver proceeds largely sequentially within each branch.

Similarly, the spark conversion timeline shows only occasional spikes, indicating that only a subset of generated sparks are actually converted into parallel work. This behavior is expected for a search problem, where workload imbalance and early termination can limit how many parallel tasks are useful.

Overall, the ThreadScope traces confirm that the solver successfully exploits parallelism at shallow depths, while deeper parts of the search remain sequential. This explains the observed sub-linear scaling: although cores are well utilized initially, the limited parallel region and uneven branch sizes constrain further speedup.

9. Limitations

First, there is no unified metric for puzzle difficulty. Even with controlled grid size and constraint density, human-designed puzzles can differ significantly in structure and search difficulty, limiting direct comparability.

Second, in the constraint density experiment, higher-density puzzles are created by manually adding constraints based on a known solution. Although consistent, these constraints are not expert-designed and may unintentionally simplify the puzzle through stronger pruning.

Third, the “random” value selection strategy uses a fixed pseudo-random permutation, resulting in a deterministic ordering rather than true randomness. This may exaggerate interactions between value ordering and puzzle structure.

Fourth, parallelism is limited to a fixed depth, after which the search becomes sequential. This restricts achievable speedup and makes performance sensitive to early branching decisions.

Finally, the benchmark set is relatively small. A larger and more diverse collection of puzzles would be needed to support more general conclusions about solver performance and scalability.

11. Conclusion

In this project, we implemented both sequential and parallel solvers for the Futoshiki puzzle and systematically evaluated how different design choices affect performance. Our experiments show that Futoshiki is a challenging constraint satisfaction problem whose runtime is highly sensitive to puzzle structure, heuristics, and parallel parameters.

We find that human-designed puzzles are consistently harder for the solver than randomly generated puzzles, even when grid size and constraint counts are controlled. This suggests that carefully placed constraints can significantly increase search difficulty. Experiments on constraint density further reveal a non-monotonic relationship between density and runtime: puzzles with very low or very high constraint density are relatively easy to solve, while moderately constrained, human-designed puzzles are the most challenging.

Value selection also plays an important role. In the sequential setting, alternative value orderings can greatly reduce runtime, while in the parallel setting, the impact of value ordering is often reduced but not eliminated. For certain puzzles, early branching decisions dominate performance even under parallel execution, highlighting the interaction between heuristics and parallel depth.

Our parallel solver achieves clear speedups as the number of cores increases, with the best performance obtained at a moderate parallel depth. Scaling is sub-linear, as expected for search-based problems, and ThreadScope analysis confirms that parallelism is effectively utilized at shallow depths while deeper search remains sequential.

Overall, this work demonstrates that parallelism can significantly improve solver performance, but its effectiveness depends on careful parameter tuning and puzzle characteristics. These results highlight the importance of combining parallel search with well-designed heuristics and controlled experimental evaluation when studying constraint satisfaction problems.