Optimal Brainmaster on a Phone: Bayesian Adaptive Design with Lookahead

Brainmaster solver project
Technical report

21 November 2026

Abstract

We describe a small, framework-free web app that plays the codebreaker in Mastermind (also sold as Brainmaster) against a human codemaker. It treats each guess as an experiment in a Bayesian adaptive design. The posterior over the M^P codes is kept as an explicit table, and guesses are ranked by their expected information gain. Pure information-greedy play is myopic. The app therefore re-ranks the most informative guesses by one-step lookahead (a rollout of the greedy policy), and it switches to an exact branch-and-bound search for the minimum expected number of moves once few codes remain. For the classic game with 6 colours and 4 positions, the app needs 4.3403 guesses on average (5625/1296). This equals the proven optimum, so the app plays the classic game optimally. Pure information-greedy play needs 4.4151. The app never needs more than 6 guesses, and every decision takes well under a second in a phone browser. We report the method, the implementation, and an empirical analysis of its behaviour and expected number of moves across 23 game sizes.

1 Introduction

In Mastermind, the codemaker hides a code of P pegs (classically $P = 4$), each one of M colours ($M = 6$); colours may repeat. The codebreaker guesses codes and receives, for each guess, a number of **black** pins (right colour in the right position) and **white** pins (right colour in the wrong position). The board has ten rows.

The app is meant for the table. Someone else picks a secret, the app proposes every guess, and the user taps in the pins the codemaker places. The aim is to show that a principled strategy reliably cracks the code in four or five moves. The app also shows, at every move, the quantities that make the strategy legible: the number of codes still possible, the remaining uncertainty in bits, the expected information gain of the proposed guess, its probability of being the secret, and the exact expected number of remaining moves.

Three questions shaped the design. Can the app use Bayesian adaptive experimental design (Lindley 1956; Chaloner and Verdinelli 1995) literally, with an explicit hypothesis table? How far is information-greedy play from optimal play? And how much of optimal play can a phone afford? Section 2 derives the information-gain criterion and its cost. Section 3 develops the decision policy, Section 4 the app, and Section 5 the analysis.

2 The codebreaker as a Bayesian experimenter

Let $\mathcal{C} = \{0, \ldots, M-1\}^P$ be the set of codes, $N = |\mathcal{C}| = M^P$. A guess $g \in \mathcal{C}$ against a secret s yields the feedback $\varphi(g, s) = (b, w)$. The feedback alphabet $\mathcal{O}$ consists of all (b, w) with $b + w \le P$ except $(P-1, 1)$, so $|\mathcal{O}| = (P+1)(P+2)/2 - 1$, which is 14 for $P = 4$.

Posterior. Under a uniform prior and noise-free feedback, the posterior after a history $(g_1, f_1), \ldots, (g_t, f_t)$ is uniform on the consistent set

$$S_t = \{s \in \mathcal{C} : \varphi(g_i, s) = f_i \ \text{ for all } \ i \le t\}.$$

The app stores S_t as a list of code indices; the remaining uncertainty is $H(S_t) = \log_2 |S_t|$ bits (10.34 bits at the start of the classic game).

Expected information gain. A guess g partitions S into classes $S_f = \{s \in S : \varphi(g, s) = f\}$ with sizes n_f, $\sum_f n_f = n = |S|$. Since the posterior after observing f is uniform on S_f, the expected information gain (Lindley 1956) is

$$\mathrm{EIG}(g) = \log_2 n - \sum_f \frac{n_f}{n} \log_2 n_f = -\sum_f \frac{n_f}{n} \log_2 \frac{n_f}{n},$$

the entropy of the partition induced by g. A guess that cannot be the secret still has positive EIG if it splits S. Such **probe** guesses are often the most informative.

Cost. One might expect a quadratic sum over pairs of feasible codes. Because the feedback is a deterministic function of (g, s), a single pass over S that counts class sizes suffices. Scoring every possible guess therefore costs $O(N \cdot n)$ table look-ups: 1.7 million for the first move of the classic game. Up to $N = 4096$ the app precomputes the full $N \times N$ feedback table (one byte per entry); above that it scores on the fly.

Symmetry. If a colour has not appeared in any guess so far, all such colours are interchangeable: permuting them maps S_t onto itself. It therefore suffices to consider **canonical** guesses, in which unused colours appear in increasing order of first occurrence. This reduces the 1296 first guesses to 15, one per set partition of the positions, and it is exact, not a heuristic.

Displayed statistics. For the current guess g: $\Pr[\text{win}] = 1/n$ if $g \in S$ and 0 otherwise; the EIG as above; and the expected number of remaining moves $T_\pi(S)/n$, where $T_\pi(S)$ is the total number of guesses the app's own policy π needs, summed over all $s \in S$ (the winning guess included). The app computes T_π exactly by simulating the policy against every consistent secret (Section 4).

3 From information to expected moves

The objective of the game is the number of guesses, not information. The optimal expected-moves policy satisfies the Bellman recursion

$$T^*(S) = \min_{g \in \mathcal{C}} \left[n + \sum_{f \neq (P,0)} T^*(S_f) \right], \quad T^*(\{s\}) = 1,$$

which charges one guess to every remaining secret and stops for the secret that was hit. Information-greedy play maximises a one-step proxy for this objective. The proxy is good, but it ignores, for instance, that guessing a feasible code may win immediately.

Exact search. The app solves the recursion exactly by depth-first branch and bound when S is small. Any guess yields at most $|\mathcal{O}| - 1$ non-winning classes, so a set of size n needs at least

$$L(n) = n + \sum_{j=1}^{|\mathcal{O}|-1} L(m_j), \quad L(1) = 1,$$

total guesses, where (m_j) is a balanced split of $n - 1$ into $|\mathcal{O}| - 1$ parts. This bound is valid because L is convex (verified numerically for all $n \le 10^4$ in the test suite), so a balanced split minimises the sum. At each node:

- all canonical candidates are scored in one pass, which yields their bound $n + \sum_f L(n_f)$, their EIG, and a hash of the induced partition, so that guesses inducing identical partitions are evaluated only once;
- candidates are explored in order of increasing bound, ties broken by EIG;
- children are solved largest first under the remaining budget best $-$ partial $-$ $\sum L$;
- solved sets are memoised, as are proven lower bounds.

On small games the search agrees with an unpruned exhaustive search over all guesses.

Rollout. For medium-sized sets, exact search is too slow for a phone. The app then applies one step of policy improvement (Bertsekas et al. 1997) to a base policy π_0, which is the EIG-greedy rule plus exact search on small sets. Among the k guesses with the highest EIG, it picks the one minimising $n + \sum_f T_{\pi_0}(S_f)$, where each child's total is obtained by simulating π_0 to the end. Section 3.1 states the operator precisely and shows that the resulting policy is never worse than π_0, at any point of the game.

Tiers. The policy is deterministic, and its tier depends on n only: exact search if $cnN \le 60 \cdot 1296$ (i.e. $n \le 60$ for the classic game), rollout if $cnN \le 1296^2$, and plain max-EIG otherwise (only the first move or two of the largest games). The constant c is 1 when the feedback table is available and 3 otherwise, which reflects the cost of scoring on the fly. In hard mode the candidates are restricted to S, and n^2 replaces nN. Rollout uses $k = 20$ candidates for games with $N \le 1296$, decreasing as $k = \max(10, \lfloor 20 \cdot 1296/N \rfloor)$ for larger games. Exact search has one more shortcut: if a feasible guess splits the remaining codes into singletons, it meets the bound $L(n) = 2n - 1$ and is accepted immediately. This avoids scanning all N candidates on the many tiny sets that rollout simulations visit. Because the policy is deterministic, its first two moves for every configuration can be precomputed into a 28 kB **opening book** (5 kB compressed). A test checks that the book is a cache, not an approximation: recomputing its entries reproduces them exactly.

3.1 Rollout as policy improvement

Background. Rollout is policy iteration cut down to one improvement step, carried out on line at the states actually reached (Bertsekas et al. 1997; Bertsekas 2020). At the current state it tries each admissible decision, follows a base policy from the resulting state, and takes the decision with the lowest total cost. It was introduced to improve heuristics for combinatorial optimisation, with conditions under

which the result is never worse than the base (Bertsekas et al. 1997). It needs no training, only a base policy that can be simulated; the price is many simulations per decision.

The operator. The solver's state is (S, U), where U is the set of colours used so far; U fixes the canonical candidates $\mathcal{G}$ ($\mathcal{G} = S$ in hard mode). Guess g leads to the states $\left(S_f^g, U \cup U_g\right)$, with S_f^g its classes and U_g its colours; we keep writing $T_\pi(S)$. For a deterministic policy π, let $Q_\pi(S, g) = n + \sum_{f \neq (P,0)} T_\pi\left(S_f^g\right)$ be the total for guessing g once and following π afterwards, so $T_\pi(S) = Q_\pi(S, \pi(S))$. On rollout-tier states the app plays

$$\pi_1(S) = \arg\min_{g \in C_k(S)} Q_{\pi_0}(S, g),$$

where $C_k(S)$ holds the k candidates of highest EIG and ties go to the higher-ranked guess; elsewhere $\pi_1 = \pi_0$. The base π_0 is the app with rollout switched off ("Max EIG + exact endgame" in Table 1): the branch-and-bound optimum on exact-tier states, where $T_{\pi_0} = T^*$, and the max-EIG guess on all other states, rollout-tier ones included, with ties broken in favour of feasible codes, then of the smaller code index. `baseTotal` evaluates T_{π_0} over all children without sampling, so the compared values of Q_{π_0} are exact.

Restricting the candidates. Improvement rests on the one-step inequality

$$Q_{\pi_0}(S, \pi_1(S)) \leq Q_{\pi_0}(S, \pi_0(S)) = T_{\pi_0}(S), \tag{1}$$

which needs only $\pi_0(S) \in C_k(S)$, not that the best guess is a candidate: the filter limits how much rollout gains, not whether ($k = 1$ gives back π_0). At a rollout-tier state, $\pi_0(S)$ is the max-EIG guess, and `rankByEig` uses the same candidates, score and tie rule as `greedy`. So the base guess is ranked first and lies in $C_k(S)$ for every $k \geq 1$, up to one floating-point caveat: both treat EIG values within 10^{-12} as equal. In the classic game, its hard variant, and the 5×4, 7×4 and 4×5 games, the base guess was ranked first at all 50 rollout-tier states the policy visits. The guarantee would fail if filter and base disagreed, e.g. if π_0 used exact search where candidates are still filtered by EIG.

Over the whole game. Equation 1 covers one deviation followed by π_0, but π_1 deviates again at every later rollout-tier state. In the terminology of rollout algorithms (Bertsekas et al. 1997; Bertsekas 2020), π_0 is *sequentially consistent*: after one step it continues exactly as it would have anyway, because its guess is a deterministic function of (S, U), and because tiers depend on n alone and grow with n, so exact-tier states have exact-tier children and T^* is the cost of actually following π_0 there. With $\pi_0(S) \in C_k(S)$, the restricted rollout is then *sequentially improving*, meaning that Equation 1 holds at every rollout-tier state, and $T_{\pi_1} \leq T_{\pi_0}$ follows on every state by induction on n. Children are strictly smaller, since a guess that neither splits S nor can win has $Q_{\pi_0} = n + T_{\pi_0}(S)$, which Equation 1 excludes for π_1, and the max-EIG guess has positive EIG for $n \geq 2$. On exact-tier states π_1 and π_0 agree on the whole subtree; on greedy-tier states they make the same guess, and the induction hypothesis applies child by child; on rollout-tier states, with $g = \pi_1(S)$,

$$T_{\pi_1}(S) = n + \sum_f T_{\pi_1}\left(S_f^g\right) \leq Q_{\pi_0}(S, g) \leq T_{\pi_0}(S).$$

The feedback is random from the codebreaker's point of view, but T sums over all children exactly, so the deterministic argument carries over. Two qualifications remain. An exact search that exceeds its

work cap ($4 \cdot 10^8$ units) falls back to the greedy guess, and whether it does depends on the memo, so π_0 would no longer be a function of the state; the cap was never reached in the runs above. And the guarantee is a weak inequality relative to π_0 only: it does not make π_1 optimal, nor comparable with rollout on another base, as the non-monotone exact-threshold ablation in Figure 2 shows.

Cost. A rollout decision scores all candidates ($|\mathcal{G}|\ n$ look-ups) and then, for each of the k candidates, simulates π_0 to the end on every child: a max-EIG step on every non-exact set below, at most N look-ups per code and level, and memoised exact searches on the small sets at the bottom. That is about $kdNn$ look-ups plus the exact searches, where d is the number of non-exact levels below S, against $|\mathcal{G}|\ n$ for a greedy decision. A candidate is dropped once its partial total reaches the best so far, so cost grows more slowly than k: the full-tree times in Table 2 rise from 1.19 s at $k = 1$ to 5.94 s at $k = 20$. Computing T_{π_1} for the displayed expected moves repeats this at every rollout-tier state of the tree.

4 Implementation

The app consists of static files with no dependencies and no build step: `index.html`, `styles.css` and `app.js` for the interface, `solver.js` (about 600 lines) for the solver, which runs unchanged in Web Workers and in Node, `worker.js`, the generated `book.js`, and a service worker and manifest that make it installable and usable offline.

Two workers. A **suggest** worker returns the hypothesis-set size, the proposed guess, its EIG and its answer distribution. An **expected** worker simulates the policy against every consistent secret to obtain T_π. A new request can supersede an expected-moves simulation that is still running; the app then simply terminates and respawns that worker, so the interface never waits for stale work. If the user overrides the guess, the simulation starts from the user's guess.

Interaction. The answer pad is a triangular grid of the valid (b, w) outcomes, drawn as pins and annotated with the app's predicted probability n_f/n, so the prediction can be checked against what happens. Answers with probability zero are disabled: they would contradict an earlier answer, so the codemaker must have mis-scored a row, and *Undo* is one tap away. Any peg of the proposed guess can be changed through a palette, and the statistics update for the user's guess. This shows directly how a plausible-looking alternative costs expected moves. State is kept in local storage.

Testing. The Node test suite covers:

- feedback against an independent reference implementation for all pairs in several configurations, including the table-free row scorer;
- filtering against brute force;
- that symmetry reduction never loses the best EIG;
- convexity of L;
- exact search against unpruned exhaustive search, including hard mode;
- agreement of `policyStats` with step-by-step play of every secret;
- that hard mode only plays feasible codes;
- the opening book against recomputation;
- the published totals of the entropy strategy (5722) and Knuth's minimax strategy (5801);
- that the app's policy attains the optimal total of 5625 for the classic game.

Strategy	$\mathbb{E}$	Total	Max	1	2	3	4	5	6	7
Max EIG (entropy)	4.4151	5722	6	1	4	71	612	596	12	
Most parts	4.3735	5668	6	1	12	72	635	569	7	
Min. expected size	4.3951	5696	6	1	10	54	645	583	3	
Minimax (Knuth)	4.4761	5801	5	1	6	62	533	694		
Max EIG + exact endgame	4.3989	5701	6	1	5	76	620	582	12	
App: EIG + rollout + exact	4.3403	5625	6	1	8	82	665	538	2	
Max EIG, hard	4.4645	5786	6	1	13	103	502	620	57	
Minimax, hard	4.4969	5828	6	1	12	99	468	662	54	
App, hard	4.4429	5758	6	1	13	100	514	633	35	
Optimum (Koyama and Lai 1993)	4.3403	5625	6							

Table 1: Classic game ($M = 6$, $P = 4$): expected number of guesses $\mathbb{E}$, total over the 1296 secrets, worst case, and the number of secrets solved in exactly d guesses. One-step heuristics use the same symmetry reduction and break ties in favour of feasible guesses.

A headless-Chrome test plays complete games through the interface. It covers undo, persistence, overrides, settings, offline reload, and the absence of horizontal scrolling at 320–390 px widths.

5 Results

All numbers below were produced by `tools/analyze.mjs` on a 32-thread AMD Ryzen 9 7950X3D 16-Core Processor with Node v23.4.0. Each policy is evaluated exactly, by following it against every one of the N secrets, with no sampling.

5.1 The classic game

Table 1 places the app within the classic heuristics. Three entries serve as checks. The entropy strategy reproduces the published total of 5722 (4.415). The minimax strategy reproduces Knuth's five-guess worst case and his total of 5801 (Knuth 1976). The app's total of 5625 equals the minimum over all strategies established by exhaustive search (Koyama and Lai 1993). No strategy can do better, so the app's policy is an optimal strategy for the expected number of guesses, although it never searches the full game tree. It is 0.075 guesses per game better than pure information-greedy play.

The exact endgame alone recovers only about a fifth of the gap between greedy and optimal play. Most of the gap comes from the first two guesses, and that is where rollout matters. Its first decision illustrates this. The most informative opening is `1234` (3.0567 bits), but rollout prefers `1123` (3.0437 bits). The pair of identical pegs costs 0.013 bits of information, but `1123` leads to better-balanced follow-ups. Because the policy that opens with `1123` attains the optimum, `1123` is an optimal opening for the expected-moves objective.

The app optimises the mean, not the worst case. Only 2 of the 1296 secrets need six guesses, whereas Knuth's minimax strategy needs at most five, at the price of 0.136 extra guesses per game. Hard mode, which only ever plays codes that could be the secret, costs 0.103 guesses per game.

5.2 Ablations

Figure 2 and Table 2 show that a single rollout candidate ($k = 1$, which is plain greedy with an exact endgame) leaves most of the gain on the table. Two or three candidates already capture most of

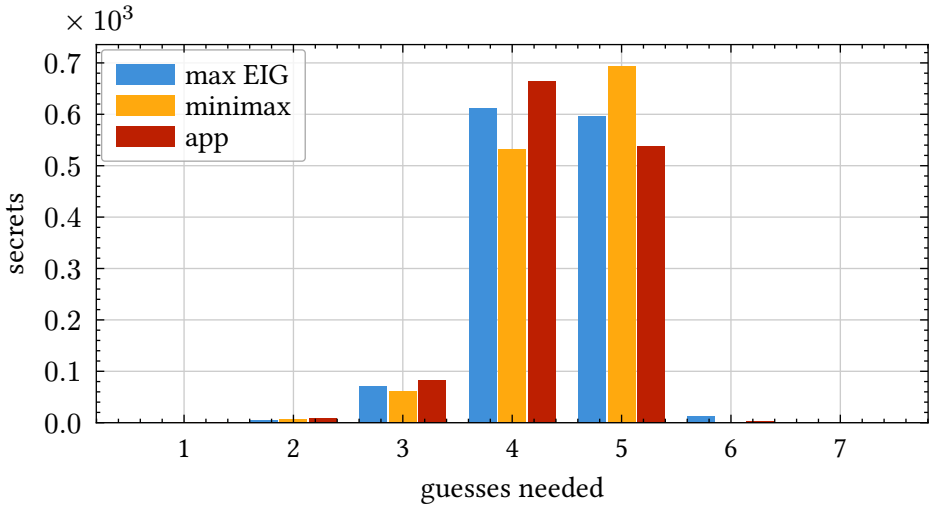


Figure 1: Distribution of game lengths for the classic game. The app shifts mass from five to four guesses; minimax trades a worse mean for the absence of six-guess games.

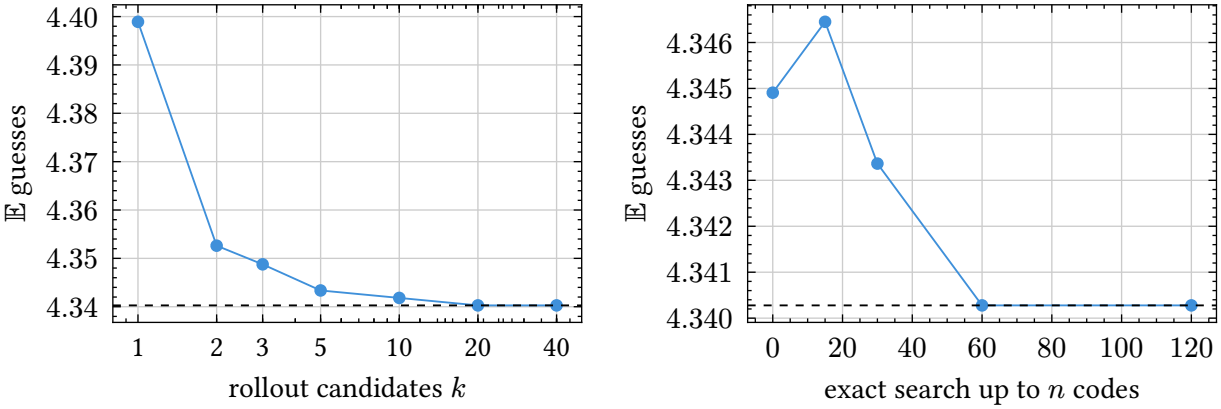


Figure 2: Sensitivity of the classic game to the number of rollout candidates (left, exact threshold 60) and to the exact-search threshold (right, $k = 20$). Dashed: optimum.

it. Reaching the optimum takes $k = 20$; more candidates change nothing. The optimum is therefore always among the 20 most informative guesses in the positions where rollout operates. Information ranking is an excellent filter, even where it is a poor final criterion. The exact threshold matters less once rollout is in place, and its effect is not monotone. Raising it changes the base policy whose cost rollout uses to rank candidates, so a slightly better base can lead to a slightly worse choice higher up. With $k = 20$, thresholds of 60 or more reach the optimum.

5.3 How the information is gathered

Figure 3 shows the information budget of a typical game. The first two guesses each remove about three bits, roughly 2^3, that is, an eight-fold reduction of the hypothesis set. By move three the set is small (on average 17.6 codes, geometric mean) and the exact search takes over; from then on the guesses are chosen for winning rather than for information. Averaged over uniformly drawn secrets, the information actually obtained equals the expected information gain exactly, because the posterior is correct, so it is not plotted separately. The decision tiers used per move were: move 1: rollout 1296; move 2: rollout 1180, exact 115; move 3: exact 1287; move 4: exact 1205; move 5: exact 540; move 6: exact 2.

Setting	$\mathbb{E}$	Total	Full-tree time (s)
$k = 1$	4.3989	5701	1.19
$k = 2$	4.3526	5641	1.69
$k = 3$	4.3488	5636	2
$k = 5$	4.3434	5629	3.19
$k = 10$	4.3418	5627	4.57
$k = 20$	4.3403	5625	5.94
$k = 40$	4.3403	5625	7.8
exact $n \leq 0$	4.3449	5631	1.51
exact $n \leq 15$	4.3465	5633	1.17
exact $n \leq 30$	4.3434	5629	2.22
exact $n \leq 60$	4.3403	5625	5.79
exact $n \leq 120$	4.3403	5625	14.26

Table 2: Ablation data. "Full-tree time" is the cost of evaluating the policy against all 1296 secrets from a cold start (the same computation that yields the starting expected-moves figure).

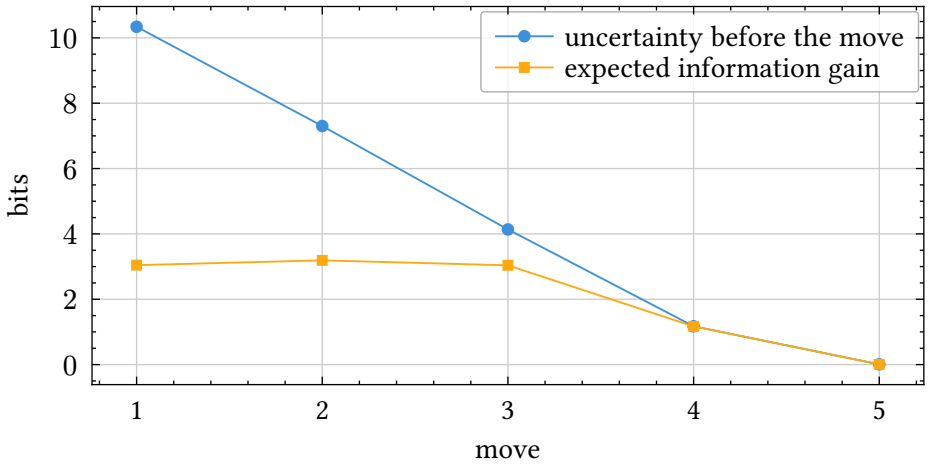


Figure 3: Classic game, app policy, averaged over the secrets still unsolved at each move: remaining uncertainty $\log_2|S|$ and expected information gain of the guess.

5.4 Other game sizes

Table 3 extends the analysis to all 23 supported configurations. The app's advantage over pure max-EIG is largest for mid-sized games, where rollout operates on the first moves. For the largest games the first moves are still chosen by max EIG, and the gap narrows. The counting bound is loose (it assumes that every guess splits the set into equal classes), but it shows how the difficulty grows: roughly $\log_{|\mathcal{O}|} N$ plus a constant.

5.5 Speed

Table 4 and Table 5 show that the hybrid policy fits a phone budget. No decision in any configuration takes longer than 236 ms on the desktop, and the classic game never exceeds 21 ms. We could not measure phone latency directly. Chrome's CPU throttling applies to the page thread only, not to the Web Workers that run the solver. Assuming the usual factor of 3–5 between desktop and phone single-

M	P	N	**Bound**	**Max EIG**	**App**	**Max**	**App hard**	**Max**	**Open-ing**
2	3	8	1.875	2.250	2.250	3	2.250	3	112
3	3	27	2.630	2.704	2.704	4	2.704	4	112
4	3	64	2.844	3.250	3.219	4	3.219	4	112
5	3	125	3.336	3.624	3.608	5	3.640	5	123
6	3	216	3.616	3.981	3.954	5	4.000	6	123
7	3	343	3.758	4.338	4.297	6	4.382	7	123
8	3	512	3.838	4.678	4.607	6	4.764	8	123
9	3	729	4.084	4.996	4.933	6	5.143	9	123
10	3	1000	4.332	5.310	5.247	7	5.529	10	123
2	4	16	2.063	2.750	2.750	4	2.750	4	1112
3	4	81	2.815	3.086	3.037	4	3.049	4	1123
4	4	256	3.227	3.563	3.535	4	3.547	5	1123
5	4	625	3.683	3.962	3.941	5	3.962	5	1123
6	4	1296	3.847	4.415	4.340	6	4.443	6	1123
7	4	2401	3.926	4.739	4.712	6	4.759	7	1234
8	4	4096	4.371	5.051	5.021	7	5.095	8	1234
9	4	6561	4.607	5.323	5.303	7	5.408	9	1234
10	4	10000	4.742	5.621	5.602	8	5.740	10	1234
2	5	32	2.344	3.313	3.031	4	3.031	4	11122
3	5	243	2.914	3.358	3.358	4	3.391	5	11123
4	5	1024	3.607	3.884	3.866	5	3.889	6	11223
5	5	3125	3.871	4.350	4.328	6	4.349	6	11223
6	5	7776	4.017	4.769	4.760	7	4.769	7	11223

Table 3: Expected number of guesses for every supported configuration ($M^P \leq 10^4$): the counting bound $L(N)/N$, pure max-EIG, the app in normal and hard mode with worst cases, and the app's opening (colours numbered from 1).

thread JavaScript performance, the slowest decision stays around one second, and the classic game stays instantaneous. Thanks to the opening book, the expensive first two decisions are free, and so is the expensive initial expected-moves figure (several seconds of computation for the classic game). Later decisions involve sets that the tiers keep small. The slowest positions are mid-sized sets in the largest games, where rollout simulations score many candidates without a feedback table.

6 Discussion

What the app demonstrates. The explicit hypothesis table makes the Bayesian reading concrete: the board shows how many codes are left after each answer, and the answer pad shows the predicted distribution of the next answer. Information gain explains most of good play. The remaining few

Game	States	Sampled	Suggest p50	p95	max	Expected p95	max
7×3	35	35	0.1	0.5	4.2	0.7	0.8
7×3 hard	34	34	0	0.1	0.3	0.1	0.3
8×3	37	37	0.1	5.9	9.3	1	1.1
8×3 hard	35	35	0	0.7	0.9	0.2	0.3
9×3	36	36	2.1	24	100	1.3	1.5
9×3 hard	35	35	0.1	1.3	3.5	0.2	0.2
10×3	56	56	1	88	151	5.6	14
10×3 hard	35	35	0.4	17	26	0.4	0.6
4×4	58	58	0	0.1	0.1	0	0.2
4×4 hard	59	59	0	0	0	0	0.2
5×4	91	91	0	0.4	1.5	0	0.3
5×4 hard	90	90	0	0.1	0.1	0	0.3
6×4	102	102	0.3	2.4	21	0.3	1.3
6×4 hard	94	94	0	0.2	2.9	0	0.4
7×4	336	114	0.9	63	135	6	9
7×4 hard	105	105	0	2	14	0.1	11
8×4	856	103	1.2	110	179	26	104
8×4 hard	108	108	0.1	18	130	0.2	0.5
9×4	1931	101	1.8	78	92	247	480
9×4 hard	247	109	0.9	49	207	2	19
10×4	3091	102	2.6	28	68	469	1336
10×4 hard	352	103	0.8	107	236	8.8	63
4×5	175	108	0	0.5	2.7	0	0.4
4×5 hard	165	103	0	0.1	0.2	0	2.1
5×5	640	100	1	37	74	2	8.2
5×5 hard	213	127	0	0.2	4.3	0	0.4
6×5	2062	102	2.4	130	146	59	110
6×5 hard	318	110	0.1	23	30	0.3	0.8

Table 4: Desktop latency in ms (Node, cold solver per state) for positions beyond the opening book that the policy actually reaches. “Suggest” is the time to choose the next guess; “Expected” is the time to compute the exact expected number of remaining moves. The largest 40 positions are always included in the sample.

hundredths of a guess come from planning: preferring guesses that leave well-structured follow-ups and, late in the game, guesses that can win.

Limitations. Optimality is established for the classic game only, by comparison with the published optimum. For other sizes, the policy is guaranteed only to be no worse than its greedy base (Section 3.1), and the cost-aware tiers deliberately give up up to 0.01 guesses per game in the largest games to keep

Game	Moves	First	Suggest	Expected
6×4	4.5	61	32	33
6×4 hard	4.5	37	33	34
7×4	4.75	72	33	96
8×4	5	168	132	330
10×4	5.75	20	34	629
5×5	4.5	141	33	146
6×5	4.25	22	82	113

Table 5: End-to-end latency in headless desktop Chrome (ms, maximum over four games per row, measured by polling the DOM, so about 30 ms is the floor). “First”: page load to first usable suggestion, including building the feedback table in the worker. “Suggest” and “expected”: time from tapping an answer to the next suggestion and to the expected-moves figure, for moves 2 onwards.

every decision under a second on a phone. Full exact optimisation from the first move remains out of reach in a browser for general M, and the app does not attempt it. The symmetry reduction exploits colour permutations only. Adding position permutations would shrink the candidate sets further and would be the first step towards computing optimal opening books offline. The app assumes the codemaker scores correctly. It detects impossible answers but does not reason about noisy feedback, which would require a likelihood model instead of hard elimination.

References

Dimitri P. Bertsekas (2020). *Rollout, Policy Iteration, and Distributed Reinforcement Learning*. Belmont, MA: Athena Scientific.

Dimitri P. Bertsekas, John N. Tsitsiklis, and Cynara Wu (1997). Rollout Algorithms for Combinatorial Optimization. In *Journal of Heuristics* 3.3, pages 245–262.

Kathryn Chaloner and Isabella Verdinelli (1995). Bayesian Experimental Design: A Review. In *Statistical Science* 10.3, pages 273–304.

Donald E. Knuth (1976). The Computer as Master Mind. In *Journal of Recreational Mathematics* 9.1, pages 1–6.

Kenji Koyama and Tony W. Lai (1993). An Optimal Mastermind Strategy. In *Journal of Recreational Mathematics* 25.4, pages 251–256.

Dennis V. Lindley (1956). On a Measure of the Information Provided by an Experiment. In *The Annals of Mathematical Statistics* 27.4, pages 986–1005.