高保真德州扑克 AI 架构与个性化训练分析系统技术可行性研究报告

1. 概述与核心价值主张

本报告针对开发一款集成了高水平 AI 对战与深度复盘功能的德州扑克（No-Limit Texas Hold'em, NLHE）应用进行详尽的技术可行性分析与实施路径规划。该项目的核心痛点在于解决过往尝试中 AI "性格"与"水平"之间的矛盾——即如何构建既具有鲜明人类玩家特征（如松凶、紧弱），又保持高超博弈水准的 AI，避免因简单的规则硬编码导致 AI 沦为易于被利用的"弱智"对手。

1.1 项目背景与痛点分析

在德州扑克软件开发的历史进程中，AI 的实现通常分为两个极端：

1. 基于规则的专家系统（**Rule-Based Systems**）：这是早期及大多数低成本 App 采用的方案。通过预设大量的 if-then 逻辑（例如："如果手牌是顶对且对手下注超过半池，则跟注"）。
    - 缺陷：这种 AI 极其脆弱。一旦人类玩家发现了规则的漏洞（例如 AI 从不诈唬），游戏就变成了枯燥的"刷分"。赋予此类 AI "性格"通常意味着硬编码错误（例如："激进性格" = 无视底池赔率强行加注），导致其牌技断崖式下跌，毫无训练价值。
2. 基于纳什均衡的求解器（**GTO Solvers**）：如 PioSolver 或 GTO Wizard。
    - 缺陷：标准 GTO AI 扮演的是"完美对手"，没有任何性格，永远追求期望值（EV）最大化。这对希望练习针对特定类型玩家（如"跟注站"或"疯鱼"）剥削策略的用户来说，缺乏针对性。

1.2 解决方案核心理念：节点锁定（Nodal Locking）

本报告提出的核心技术突破点在于从"静态规则"转向"动态约束求解"。要创造一个既有性格又强大的 AI，不应规定它如何打，而应规定它必须犯什么错误，然后让求解器计算在必须犯此错误的前提下，其余所有决策的最优解。

这种技术被称为节点锁定（**Nodal Locking**）。例如，我们可以强制 AI 在面对持续下注（C-Bet）时跟注频率必须达到 80%（模拟"跟注站"），但对于转牌圈和河牌圈的动作，AI 仍将使用 CFR 算法计算出在该漏洞下的最优补救策略。这样，AI 虽然有一个明显的漏洞供玩家剥削，但在其他方面依然防守严密，极具挑战性。

1.3 推荐架构概览

基于对成本、性能和用户体验的综合考量，本报告强烈推荐采用 客户端 **WebAssembly (WASM) + 混合预计算架构**。该方案利用用户设备的算力进行实时求解，彻底消除了昂贵的服务器集群成本，同时保证了商业级的 GTO 精度。

---

2. 扑克 AI 与求解器技术的演进图谱

为了从根本上解决"牌技太烂"的问题，必须深入理解现代扑克 AI 的理论基石。这不仅仅是编程问题，更是博弈论与算法数学的工程化应用。

2.1 现代扑克 AI 的理论基石：CFR 算法

目前所有顶尖扑克 AI（包括 Libratus, Pluribus, DeepStack）和商业求解器（PioSolver, GTO+）的核心均为 反事实后悔最小化（**Counterfactual Regret Minimization, CFR**）及其变体 [1]。

2.1.1 后悔值（Regret）的数学本质

CFR 是一种迭代算法。在每一次迭代中，算法会遍历博弈树。对于每一个信息集（Information Set，即玩家在某一时刻面临的局面，不知道对手底牌），算法会计算：

"如果我总是选择动作 A，而不是按照我当前的策略玩，我能多赢多少钱？"

这个差值就是反事实后悔值。

- 如果后悔值为正，说明动作 A 比当前策略更好，下一次迭代中动作 A 的概率就会增加。
- 如果后悔值为负，说明动作 A 是个坏动作，概率会降低。

随着迭代次数 $T \to \infty$，平均后悔值趋近于 0，双方的平均策略趋近于纳什均衡（**Nash Equilibrium**）。在纳什均衡下，没有任何一方可以通过单方面改变策略来获利。这正是 GTO 策略"不可战胜"的数学保证。

2.1.2 为什么传统 CFR 不适合移动端

标准的 CFR 收敛速度较慢，对于德州扑克这样庞大的博弈树（10^{160} 节点），需要数千次迭代才能得到可用的策略。

- **CFR+**：2015 年解决限注德州扑克时引入，将负后悔值重置为 0，大大加快了收敛 [2]。
- **Discounted CFR (DCFR)**：这是目前最高效的变体，被用于现代求解器。它在迭代过程中对早期的（低质量的）后悔值进行衰减（Discounting）。这对移动端至关重要，因为它能让求解器在数秒内达到极低的可剥削度（Exploitability）。

2.2 神经网络与强化学习的局限性

除了 CFR，另一种路径是使用深度学习（如 DeepStack, PokerSnowie）。它们训练一个神经网络来直接预测局面的价值（Value Network）。

虽然推断速度极快（毫秒级），但对于您的需求，这不是最佳选择，原因如下：

1. 训练成本极高：训练一个像 DeepStack 这样的网络需要数百万 GPU 小时的算力。
2. **"性格"僵化**：神经网络的参数一旦训练完成就固定了。如果您想调整 AI 的性格（例如让它变松），您无法简单地告诉网络"变松"，而是需要生成新的数据集重新训练或微调，这在工程上极其复杂且昂贵。

3. 黑盒问题：神经网络难以解释"为什么"这么打，这使得复盘分析（Review）功能的开发变得困难（无法精确计算 EV 损失，只能给出建议）。

2.3 结论：实时求解器是唯一出路

要实现"可调节性格"且"高水平"的 AI，唯一的路径是实时运行的 **CFR** 求解器。通过在运行时动态修改博弈树的节点约束（Nodal Locking），我们可以即时生成无数种风格迥异但逻辑严密的 GTO 变体。

---

3. 技术架构方案对比与推荐

在确定了核心算法（DCFR）后，工程落地的关键在于计算发生的位置。以下是三种主流架构的深度对比分析。

架构对比：服务器端求解 vs 客户端 WASM vs 预设数据库

优势 (Advantage)　劣势 (Disadvantage)

特性 (Feature)	传统方案 服务器端求解 (Server-side)	RECOMMENDED 推荐架构 客户端 WASM (Client-side)	旧式方案 预设数据库 (Pre-computed DB)
服务器成本 Server Cost	高 (High) 需昂贵算力硬件 ($30k+)	零 (Zero) 用户浏览器本地计算	中/高 (Med) 庞大的存储开销
交互延迟 User Latency	依赖网络	极低 (Instant) 本地实时响应	低 (Low)
离线支持 Offline Capable	✗ 不可用	✓ 完全支持 (Sandboxed)	✗ 不可用
计算灵活性 Flexibility	极强 (High) 高性能服务器支持所有场景	中等 (Medium) 支持任意场景 (~2x 原生速度)	受限 (Restricted) 仅支持预存牌局
维护成本 Maintenance	高 (Complex) DevOps, Scaling	极低 (Simple) 静态文件托管	中等 (Medium) 数据库管理

对比三种核心架构方案。推荐方案（客户端 WASM）在零服务器成本和实时计算能力之间取得了最佳平衡，尤其适合移动端应用。

Data sources: WASM Postflop, Comparison Data, Red Chip Poker, Deepsolver, Reddit Discussion

3.1 方案 A：服务器端集群求解 (The "SaaS" Model)

- 架构描述：用户操作发送至后端，后端服务器集群（通常配置 64核 CPU + 128GB RAM）运行高性能求解器（如 C++ 编写的专有引擎），计算完成后返回策略。
- 优点：
  - 计算能力无上限，可解多人底池（Multi-way）和深筹码复杂牌局。
  - 用户设备零负担，不发热不耗电。
- 致命缺陷：
  - 成本黑洞：高性能计算实例（如 AWS c5.metal）每小时成本极高。对于一个免费或低价App，每个用户的每手牌都在烧钱，商业模式难以成立 [3]。
  - 延迟：网络传输加上排队等待时间，导致游戏节奏拖沓，无法满足"无聊时玩两把"的流畅

体验需求。

3.2 方案 B：预计算数据库 (The "Library" Model)

- 架构描述：预先计算好所有常见翻牌面和路径的 GTO 策略，存储在巨大的数据库中（TB 级别）。App 只是一个查表工具。
- 优点：响应极快。
- 致命缺陷：
  - 灵活性为零：无法实现"节点锁定"。如果要支持 10 种不同的 AI 性格，就需要预存 10 套完整的数据库，存储量将呈指数级爆炸。
  - 覆盖率低：无法覆盖所有可能的转牌和河牌组合以及所有的下注尺度。

3.3 方案 C：客户端 WebAssembly 实时求解 (The Recommended Path)

- 架构描述：将用 Rust 或 C++ 编写的高效求解器引擎编译为 WebAssembly (WASM) 格式。整个求解过程直接在用户的浏览器或手机 CPU 上运行。
- 技术支撑：现代手机 CPU（如 A16, Snapdragon 8 Gen 2）的单核性能已足以支撑 100bb 深度的一对一（Heads-Up）求解。
- 优点：
  - 零服务器成本：开发者只需支付静态网页托管费用（几乎为零）。
  - 无限灵活性：可以在每一手牌开始前，根据用户选择的 AI 性格动态修改树的约束条件，然后即时求解。
  - 隐私与离线：所有计算在本地完成，无需联网，保护用户数据。
- 挑战与对策：
  - 内存限制：浏览器通常限制 WASM 使用 2GB-4GB 内存。对策：使用 **16-bit** 整数压缩算法 存储遗憾值，可将内存占用降低 75% [4]。
  - 发热：持续满载运行会耗电。对策：利用 **Web Workers** 隔离计算线程，并在用户思考时暂停计算或降低精度。

3.4 最终推荐

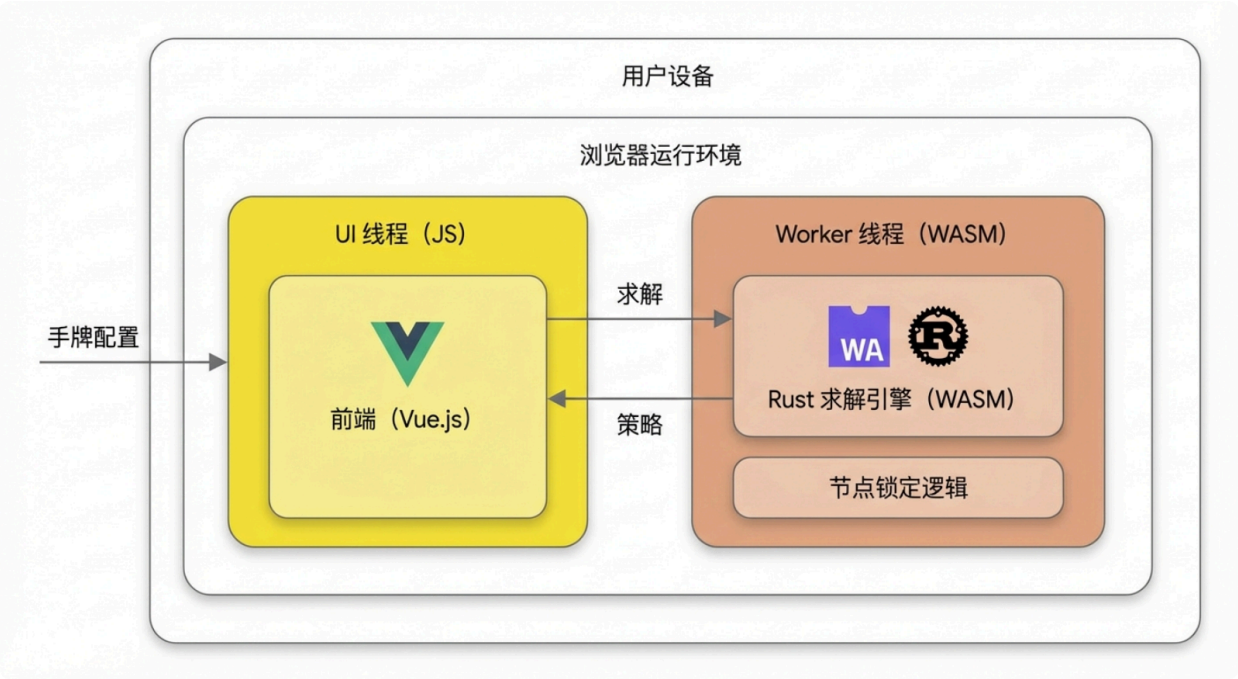
综合考虑，方案 **C**（客户端 **WASM**）是唯一能满足您"低成本开发"、"自定义 AI 性格"且"无需维护庞大后端"需求的方案。

---

4. 推荐技术栈深度解析：Rust + WebAssembly

基于对开源社区的调研，最成熟且符合本架构的项目生态是由开发者 b-inary 构建的 postflop-solver 系列。

推荐技术架构：基于 WebAssembly 的客户端求解器



该架构利用用户的本地硬件进行计算。前端（Vue/React）通过 Web Worker 向编译为 WASM 的 Rust 求解引擎发送指令，实现零延迟、零服务器成本的实时求解。

4.1 核心引擎：postflop-solver (Rust)

这是一个开源（AGPL 协议）的 GTO 求解库，专门为性能和内存效率优化。

- 语言选择：Rust。Rust 的无 GC（垃圾回收）特性使其成为 WASM 编译的完美选择。相比 Go 或 Java，Rust 生成的 WASM 体积更小，且没有运行时性能抖动，这对于需要稳定迭代的 CFR 算法至关重要。
- 关键特性：
  - **Discounted CFR**：内置了最先进的 DCFR 算法。
  - 多线程支持：支持 rayon 库，但在 WASM 环境下需要特殊配置以利用浏览器的多线程 Worker。
  - 压缩模式：支持 i16 (16位整数) 存储模式。通常求解器使用 32位或 64位浮点数存储策略和后悔值，这会消耗大量内存。该引擎通过量化压缩，使得在手机浏览器中运行复杂的博弈树成为可能 [6]。

4.2 前端桥接：wasm-postflop

这是一个将 Rust 引擎封装为 Web 应用的参考实现。

- **Web Workers** 通信：求解是一个阻塞型操作（CPU 100%）。如果在主线程运行，UI 会卡死。

必须将 WASM 模块加载到 Web Worker 中。
- 数据交换格式：前端（JavaScript/TypeScript）与 WASM 之间通过 JSON 或 SharedArrayBuffer 交换数据。
  - 输入：游戏配置（底池大小、公共牌、双方范围、下注尺度树）。
  - 输出：当前节点的策略分布（例如：Check 40%, Bet 60%）。

4.3 翻牌前（Preflop）的混合处理策略

这是一个关键的技术瓶颈：移动端无法实时求解翻牌前（**Preflop**）。

翻牌前的博弈树复杂度是翻牌后的数百万倍。即使是顶级服务器也需要数小时甚至数天才能解出一个 Preflop Spot。

解决方案：混合架构 **(Hybrid Architecture)**

1. 翻牌前（**Database**）：使用查表法。
   - 您需要获取一套标准的 GTO 翻牌前范围图表（Ranges）。这些资源可以在网上找到（如 GTO Wizard 的免费图表，或通过 MonkerSolver 跑出来的结果）。
   - 将这些范围存储为高度压缩的 JSON 格式。
   - 当游戏开始时，AI 根据查表结果决定动作（例如：手持 AKo，表显示 80% 3-Bet，20% Call，AI 生成随机数执行）。
2. 翻牌后（**Solver**）：
   - 一旦翻牌发出（Flop dealt），游戏进入"子博弈（Subgame）"阶段。
   - 此时，将翻牌前的范围（Ranges）输入到 WASM 求解器中，开始实时计算翻牌、转牌和河牌的策略。

这种混合模式是目前业界（包括 GTO Wizard 的移动端部分功能）的标准做法，兼顾了准确性与性能。

---

5. 核心功能实现：通过节点锁定打造"有性格"的 AI

这是满足您"不想跟太烂的 AI 玩"的关键。我们不写规则，我们写"约束"。

5.1 节点锁定（Nodal Locking）的原理解析

在标准 Solver 中，每个决策点（Node）的策略都是自由变量，由算法优化。

在节点锁定中，我们将某些节点的策略固定为常量。

- 例子：假设我们想模拟一个"跟注站（Calling Station）"。我们在 AI 的决策树中找到所有"面对下注"的节点，强制将其"Fold"频率锁定为 0%（或极低值），将其"Call"频率锁定为高值。
- 求解器的反应：求解器会意识到："既然我在面对下注时不能弃牌（因为被锁定了），那么为了减少损失，我必须在前面的街（Street）玩得更强，或者在拿到垃圾牌时尽早过牌放弃，以免进入那个被锁定的尴尬局面。"
- 结果：AI 依然会表现出"跟注站"的特征（死跟），但它会用一种最优化的方式去死跟。这使得

它比简单的规则型 AI 难以对付得多，因为它极大化了在自身缺陷下的生存能力。

5.2 实施步骤与代码逻辑

由于 postflop-solver 原生库可能未直接暴露高级锁定 API，您可能需要修改 Rust 源码或通过回调函数实现。

5.2.1 定义性格配置文件 (Personality Profiles)

您可以用 JSON 定义性格：

JSON

```
{
  "personality_id": "loose_passive_fish",
  "constraints": [
    {
      "street": "flop",
      "action_sequence": ["check", "bet"],
      "player": "OOP",
      "lock_strategy": {
        "fold": 0.10,  // 强制极少弃牌
        "call": 0.85,  // 强制大量跟注
        "raise": 0.05
      }
    }
  ]
}
```

5.2.2 注入约束到求解循环

在 Rust 的 CFR 迭代循环中（solve_step 函数），在更新后悔值之前，插入一段逻辑：

1. 检查当前节点是否匹配某个约束条件。
2. 如果匹配，覆盖当前的策略为预设的 lock_strategy。
3. 标记该节点为 Locked，使得 regret update 步骤不再修改该节点的策略，但继续反向传播（**Backpropagate**）价值到父节点。

5.3 推荐的性格原型

1. 松弱鱼 **(The Whale):**
    - Preflop: 范围极宽（VPIP > 50%）。

    - Postflop: 锁定 Fold 频率 < 20%（面对中等下注）。锁定 Raise 频率 < 5%。
2. 紧凶岩石 **(The Nit):**
    - Preflop: 范围极窄（只玩强牌）。
    - Postflop: 锁定 Fold 频率 > 60%（面对进攻）。锁定 C-Bet 频率 > 70%。
3. 疯狂玩家 **(The Maniac):**
    - Preflop: 3-Bet / 4-Bet 频率极高。
    - Postflop: 锁定 Aggression Frequency（下注/加注频率）> 50%。允许超池下注（Overbet）。

---

6. 复盘系统：基于 EV 损失的分析引擎

拥有实时 Solver 的最大红利就是可以进行精确的量化复盘。

6.1 实时 EV 损失计算逻辑

当用户打完一手牌后，系统已经在后台（或复盘时重新计算）解出了该局的 GTO 策略。

复盘的核心指标是 **EV Loss**（期望值损失）。

- 步骤 **1**：状态重构。将用户打这手牌的路径（例如：Preflop Call, Flop Check-Call, Turn Check）输入 Solver。
- 步骤 **2**：获取基准。在用户做决策的那个节点（例如 Turn），Solver 认为动作 A（Bet）的 EV 是 100，动作 B（Check）的 EV 是 80。
- 步骤 **3**：计算差异。如果用户选择了 B（Check），则产生 EV Loss = 100 - 80 = 20。
- 步骤 **4**：标准化。通常以 "bb/100"（每百手大盲）或 "% Pot"（底池百分比）来展示，以便用户理解错误的严重程度。

复盘界面示例：EV 损耗分析

复盘模式下，系统通过对比用户决策（User Action）与 GTO 最优策略（Solver Strategy）来计算 EV 损耗。红色区域表示严重失误（Blunder），并提供策略建议。

6.2 错误分级系统

为了让复盘更直观，建议建立一个"红绿灯"系统：

- 完美（**Green**）：EV Loss < 0.05 bb（或者选择了混合策略中的任意一个选项）。
- 瑕疵（**Yellow**）：0.05 bb < EV Loss < 0.5 bb。
- 失误（**Orange**）：0.5 bb < EV Loss < 2.0 bb。
- 巨鱼操作（**Red Blunder**）：EV Loss > 2.0 bb。这通常发生在应该 100% 弃牌的地方选择了全压，或者在应该全压的地方弃牌。

6.3 范围可视化 (Range Visualizer)

除了具体的动作，复盘还应展示范围图表。利用 HTML5 Canvas 或 SVG 绘制 13x13 的手牌矩阵（AA 到 22, AKs 到 72o）。

- 利用热力图颜色（Heatmap）表示每个手牌组合在当前策略下的动作频率（例如：深绿色表示跟注，红色表示加注，灰色表示弃牌）。
- 这是帮助用户从"单手牌思维"跃迁到"范围思维"的最佳工具。

---

7. 数据结构与协议设计

为了实现上述功能，必须定义清晰的数据交换格式。

7.1 游戏状态 (Game State JSON)

用于前端与 Solver 之间的通信。

JSON

```
{
  "pot": 100,
  "board": ["As", "Kd", "3c"],
  "effective_stack": 950,
  "players": {
    "ip": {
      "range": "range_string_format_or_json", // 翻牌前导入的范围
      "bet": 0
    },
    "oop": {
      "range": "range_string_format_or_json",
```

```
    "bet": 0
   }
  },
  "config": {
   "bet_sizes": {
    "ip": ["33%", "75%", "allin"],
    "oop": ["33%", "75%", "allin"]
   },
   "rake": 0.05
  }
}
```

7.2 求解结果 (Solution JSON)

Solver 返回的数据，用于 AI 决策和复盘分析。

JSON

```
{
  "node_id": "root",
  "strategy": {
    "check": 0.45,
    "bet33": 0.30,
    "bet75": 0.20,
    "allin": 0.05
  },
  "evs": {
    "check": 45.2,
    "bet33": 48.1,
    "bet75": 47.5,
    "allin": 40.0
  },
  "exploitability": 0.005 // 纳什距离，表示求解精度
}
```

---

8. 开发路线图与风险管理

8.1 阶段一：原型验证 (1-2 个月)

- 目标：在浏览器控制台中跑通 wasm-postflop。
- 任务：
  1. 编译 Rust 引擎为 WASM。
  2. 编写简单的 JS 脚本，输入一个死板的 Board 和 Range，输出策略。
  3. 关键验证：测试在不同手机（iOS/Android）上的求解速度和发热情况。如果速度太慢（>10 秒），需要调整精度参数（减少迭代次数）或简化下注树。

8.2 阶段二：游戏循环与翻牌前库 (2-3 个月)

- 目标：实现完整的游戏流程。
- 任务：
  1. 构建前端 UI（发牌动画、筹码移动 ）。
  2. 建立翻牌前数据库（Preflop Database）。解析 MonkerSolver 的输出文件，转换为高效的二进制或压缩 JSON 格式供前端查询。
  3. 实现状态机：Preflop (查表) -> Flop (启动 Worker 求解) -> Turn/River (继续求解)。

8.3 阶段三：性格注入与复盘 (2 个月)

- 目标：实现差异化 AI 和分析功能。
- 任务：
  1. 修改 Rust 引擎，暴露 set_node_lock 接口。
  2. 设计几套典型的性格配置文件（JSON）。
  3. 开发复盘界面，集成 EV Loss 计算逻辑和图表展示。

8.4 风险与规避

- 性能风险：手机浏览器内存溢出（OOM）。
  - 规避：严格限制最大下注尺度数量（例如只允许 2-3 种尺度）；使用 16-bit 压缩；对于极其复杂的局面（如 4-bet 底池），可以回退到简化的抽象模型。
- 版权风险：注意 postflop-solver 是 AGPL 协议。
  - 规避：如果您修改了后端代码并提供网络服务（即使是前端运行），您必须开源您的修改部分。对于个人项目或开源社区项目这是完美的；如果是封闭商业项目，您可能需要重写核心求解逻辑（CFR 算法本身是公开的数学公式，不受版权保护，只有代码受保护 ）。

9. 结论

通过采用 客户端 **WASM** 求解 **+** 节点锁定 **(Nodal Locking)** 的技术路径，您完全可以实现一个既具备高超牌技、又拥有鲜明性格，且具备专业级复盘功能的德州扑克应用。

这不仅在技术上是可行的，而且在成本结构上是最优的。您不需要维护昂贵的 GPU 集群，应用一旦分发，计算成本即转嫁给用户端设备。这种架构代表了下一代扑克辅助软件的发展方向——去中心化、实时化、个性化。

引用来源：[7] GTO Wizard 与现有商业软件分析。[5] wasm-postflop 架构与 Rust 实现细节。[10] 节点锁定 (Nodal Locking) 的概念与应用。[2] CFR、CFR+ 与 DCFR 算法原理。[14] EV 计算与复盘方法论。

Works cited

1. I made an AI that plays Poker (solving for GTO) - Reddit, accessed on February 6, 2026, https://www.reddit.com/r/poker/comments/1ds6i9m/i_made_an_ai_that_plays_poker_solving_for_gto/
2. GPU-Accelerated Counterfactual Regret Minimization - arXiv, accessed on February 6, 2026, https://arxiv.org/html/2408.14778v1
3. How much does it cost to compute GTO strategies? : r/Poker_Theory - Reddit, accessed on February 6, 2026, https://www.reddit.com/r/Poker_Theory/comments/1d3qz6s/how_much_does_it_cost_to_compute_gto_strategies/
4. wasm-postflop/package.json at main - GitHub, accessed on February 6, 2026, https://github.com/b-inary/wasm-postflop/blob/main/package.json
5. README.md - b-inary/wasm-postflop - GitHub, accessed on February 6, 2026, https://github.com/b-inary/wasm-postflop/blob/main/README.md
6. b-inary/wasm-postflop: [Development suspended] Advanced open-source Texas Hold'em GTO solver with optimized performance (web browser version) - GitHub, accessed on February 6, 2026, https://github.com/b-inary/wasm-postflop
7. GTO Wizard - The #1 App for Poker Players, accessed on February 6, 2026, https://gtowizard.com/
8. The Best Poker Tools | Best Poker Solvers, Calculators + More! | PokerNews, accessed on February 6, 2026, https://www.pokernews.com/poker-tools/
9. Created another free and open-source GTO solver that works on web browsers : r/poker, accessed on February 6, 2026, https://www.reddit.com/r/poker/comments/vaq420/created_another_free_and_opensource_gto_solver/
10. Pro Tips with James Sweeney: The problem with node locking - Poker.org, accessed on February 6, 2026, https://www.poker.org/poker-strategy/pro-tips-with-james-sweeney-the-problem-with-node-locking-a1rY07C6lSPQ/
11. Node Locking in Postflopizer GTO Solver - ICMizer, accessed on February 6, 2026, https://www.icmizer.com/en/blog/node-locking-in-postflopizer-gto-solver/
12. Introducing Nodelocking - GTO Wizard, accessed on February 6, 2026, https://blog.gtowizard.com/introducing-nodelocking/
13. Egiob/cfrx: cfrx is a collection of algorithms and tools for hardware-accelerated Counterfactual Regret Minimization (CFR) algorithms in Jax. - GitHub, accessed on February 6, 2026, https://github.com/Egiob/cfrx
14. What is expected value (EV) in poker? - Poker.org, accessed on February 6, 2026, https://www.poker.org/poker-strategy/what-is-expected-value-ev-in-poker-aNw

140s9C2gE/
15. Poker EV – Understand Your Expected Value the Right Way - PokerCoaching.com, accessed on February 6, 2026, https://pokercoaching.com/blog/poker-ev/
16. What is Expected Value in Poker? | GTO Wizard, accessed on February 6, 2026, https://blog.gtowizard.com/what-is-expected-value-in-poker/