GA-Arcade: Evolución Genética de Políticas para Pac-Man

Jose Isaac Corrales Cascante*, Andrey Navarro Quesada*
*Instituto Tecnológico de Costa Rica

***Resumen*—Presentamos un Algoritmo Genético (AG) capaz de aprender políticas lineales para Pac-Man sobre el entorno `GA-Arcade`. El sistema usa cromosomas de 12 genes que ponderan *features* simbólicas (pellets, fantasmas, apertura local) y un evaluador de fitness basado en simulaciones reproducibles sin render. Reportamos configuraciones, pseudocódigo, complejidad, resultados plausibles con líneas base y ablaciones, así como un checklist de reproducibilidad.**

I. Introducción

Los AG son eficaces para explorar espacios de políticas cuando el gradiente no está disponible [1], [2]. En juegos tipo Pac-Man, las políticas lineales siguen siendo competitivas por su interpretabilidad y bajo coste [3], [4]. Este trabajo describe e instrumenta un AG con ajustes automáticos y *fitness sharing* aplicado al proyecto `GA-Arcade`.

Históricamente, los primeros AG en arcade aparecieron en entornos simplificados (e.g., laberintos 2D con recompensas escasas), centrados en aprender tablas de acción. Con el aumento de la capacidad de simulación, se incorporaron cromosomas de pesos y cruce real, permitiendo políticas más suaves y generalizables. Implementaciones recientes integran currículos, elitismo adaptativo y compartición de fitness para evitar la convergencia prematura.

Pac-Man se consolidó como benchmark por combinar navegación en grafos, balance entre exploración y supervivencia, y eventos discretos (pellets, fantasmas, power-ups) que estresan el control de riesgos. Su mapa fijo y reglas deterministas facilitan comparaciones reproducibles entre enfoques evolutivos y basados en gradiente.

Los métodos de gradiente encuentran limitaciones: la dinámica no es diferenciable, la recompensa es esparsa y los episodios pueden terminar abruptamente por muerte, lo que causa varianza alta en estimadores [5], [6]. Los AG, al trabajar con evaluaciones en bloque y operadores discretos, toleran estas discontinuidades. Elegimos políticas lineales en lugar de redes profundas por eficiencia (simulamos cientos de episodios por generación en navegador), interpretabilidad (cada gen pondera una feature concreta) y facilidad de depuración [5], [6].

GA-Arcade se diferencia de otros entornos porque expone modo no-render con captura de snapshots, A* con caché específica para *power mode*, histórico de diversidad y autotuning integrado. Esto permite experimentar con variantes de operador sin modificar el bucle principal ni comprometer reproducibilidad.

II. Metodología

II-A. Juego y reglas

El entorno es una cuadrícula fija (28×31) con límite de 1000 pasos. Recompensas: pellet 10, super-pellet 50, fantasma comido 100, paso vacío $-1{,}5$, paso normal $-0{,}3$, muerte -500, *level clear* 10000 y penalización de estancamiento -200 tras 30 pasos sin pellet. Hay 3 vidas y modo *power* con persecución condicionada por seguridad y ruta A*.

II-B. Entorno GA-Arcade (detalles)

El simulador avanza por ticks lógicos de 100 ms; cada `stepGame` actualiza Pac-Man, fantasmas, colisiones y temporizadores. El estado interno se representa como estructura plana: posiciones (col,row), contadores de pasos, temporizadores de *power*, snapshot del nivel y banderas de eventos; esto minimiza asignaciones durante la evaluación evolutiva.

Modo no-render: `episodeSimulator.runEpisode` ejecuta el bucle sin pintar, reduciendo el coste a cálculo de lógica y colisiones. Los eventos (pellet, power, muerte, respawn) se propagan como flags en el objeto `info`, y los temporizadores de respawn y power se actualizan cada tick. Colisiones se comprueban antes y después del movimiento de fantasmas para capturar entradas a la casa de fantasmas.

El A* en *power mode* usa caché por par (pacman, ghost) y se invalida con cambios de pellets o posiciones; se limita radio de búsqueda y nodos explorados. Esto reduce el coste promedio de `stepGame`, permitiendo evaluar más individuos por generación.

II-C. Funcionamiento del juego

El motor de juego sigue un ciclo: recibe la acción propuesta por la política, actualiza la posición de Pac-Man, procesa consumibles (pellets, power pellets), aplica penalizaciones por estancamiento y luego mueve a los fantasmas. El estado se clona antes de cada paso para evitar efectos laterales en el pool de evaluadores (ver `gameState.cloneState`). El manejo de eventos registra flags (pellet comido, power activado, muerte) que luego alimentan los checks de colisión y las recompensas explicadas en la Sección II-H.

Ghosts alternan modos `SCATTER`/`CHASE` según el calendario definido en `SCATTER_CHASE_SCHEDULE`; el modo `FRIGHTENED` se activa con power pellets y reduce su velocidad. El sistema de `STALL` aplica una penalización de -200 tras 30 pasos sin comer, fuerza `hard stop` en 200 pasos sin pellet y puede terminar el episodio si el score cae por debajo

de -1000 tras 250 pasos. El *respawn* espera un temporizador configurable y reintegra fantasmas a su cápsula.

II-D. Pool de workers y paralelismo

La evaluación de la población se paraleliza con los workers descritos en `src/js/ga/workerPool.js` y `workerMessages.js`. El pool mantiene un conjunto configurable de Web Workers (por defecto 8) que reciben tareas de evaluación mediante mensajes tipados (‘evaluate-individual‘ / ‘evaluation-result‘). Cada worker importa los módulos del juego y el AG, instancia su propia RNG y simula los episodios sin render. El pool maneja la cola de individuos, distribuye chunks de cromosomas, recolecta resultados y notifica al controlador GA cuando hay suficientes datos para reconstruir la población.

Los mensajes siguen un protocolo simétrico: el hilo principal envía ‘evaluate-individual‘ con cromosoma, seed y configuración; el worker responde con ‘evaluation-result‘ incluyendo fitness, rewards y métricas adicionales; los errores se propagan con ‘error‘ para reinicializar el worker afectado. Gracias al chunking (‘chunkSize‘ por defecto 0, i.e., un individuo por mensaje) se puede ajustar la latencia vs. throughput. Los logs (‘workerMessages.js‘) registran tiempos por lote para analizar cuellos de botella.

Tolerancia a fallos: ante un ‘error‘ el pool marca el worker como no disponible, crea uno nuevo y reenvía las tareas pendientes. Los mensajes portan un ID de individuo y de generación para asegurar correspondencia y descartar respuestas tardías.

II-E. Reproducibilidad y RNG

El AG usa un LCG propio (`SeededRng`) para selección, cruce y mutación, inicializado con la semilla global. La evaluación de fitness envuelve `Math.random` con otro LCG (`runWithSeed`) derivado de `baseSeed` y el índice de episodio; así cada worker produce trayectorias deterministas aun ejecutando en paralelo. Las semillas de episodios siguen $seed_i = (baseSeed + i \cdot 1013904223) \bmod 2^{32}$, evitando colisiones entre hilos.

II-F. Codificación genética

Cada cromosoma $g \in [-3, 3]^{12}$ pondera las *features* de la tabla I. La acción seleccionada maximiza $g^\top f(\text{estado}, \text{acción})$ sujeto a legalidad. Normalizamos y limitamos el rango en cada mutación.

Cuadro I
VECTOR DE *features* (12 GENES).

ID	Descripción
1–3	Muro, pellet, super-pellet (indicadores)
4–5	Mantiene dirección, giro en U
6–8	Distancia a pellet, a fantasma, acercamiento/huida
9	Apertura local (vecinos libres/4)
10	Fracción de pellets restantes
11	Fracción de pasos usados (steps/limit)
12	(Repetida en código como 11) paso normalizado

II-G. Operadores

Configuración base (de `src/config/defaultConfig.js` y `geneticAlgorithm.js`): población 40, generaciones 50, torneo k=3, elitismo 3, tasas de selección/cruce/mutación 0.40/0.45/0.15, mutación por gen 0.6 con fuerza 0.8 y programación lineal de 1.2 a 0.7 a lo largo de las generaciones. Cruce: un punto y modo *blend* con probabilidad 0.6. *Fitness sharing* con σ=0,75, α=1. Se usa *auto-tuning* que ajusta tasas si la eficiencia o el *growth rate* se desvían de objetivos.

La selección por torneo toma k individuos al azar (con pesos de prioridad que favorecen percentiles objetivo y diversidad) y elige el de mayor fitness. Elitismo copia hasta 3 mejores al siguiente grupo; el resto se reparte según proporciones de selección, cruce y mutación. El cruce *blend* interpola genes ponderando por la calidad relativa de los padres, mientras que el cruce de un punto recombina segmentos. La mutación aplica ruido uniforme recortado al rango de genes y escala su tasa según el percentil del individuo dentro de la población (mayor exploración para individuos peores).

II-H. Fitness con fórmulas

Cada cromosoma se evalúa en 5 episodios (semillas derivadas de `baseSeed` 12345). Para un episodio e:

$$R_e = S_e + \mathbb{1}_{\text{clear}} B_c + \mathbb{1}_{\text{no\_life\_loss}} B_{nl} - P_\ell L_e - P_s \text{steps}_e - P_{st} \text{stall}_e, \tag{1}$$

con S_e la puntuación del juego, B_c=5000, B_{nl}=2500, P_ℓ=500, P_s=0 (penalización de pasos deshabilitada) y P_{st}=10 aplicado a eventos de `stall`. El fitness agregado es

$$F(g) = \frac{1}{E} \sum_{e=1}^{E} R_e, \quad E = 5. \tag{2}$$

Se usa currículo lineal: nivel $l = \min(6, 1 + \lfloor 0{,}15 \cdot \text{gen} \rfloor)$.

II-I. Normalización y restricción de genes

Las *features* se acotan a $[0, 1]$ con funciones de normalización: distancias Manhattan se dividen por el máximo (ancho+alto), progreso por pellets se expresa como fracción restante, y pasos como fracción del límite. Los genes se *clip*ean por componente:

$$g'_i = \min(\max(g_i, -3), 3), \quad i \in \{1, \ldots, 12\}, \tag{3}$$

garantizando estabilidad en mutaciones y cruces. La sensibilidad por feature se estima observando $\partial(g^\top f)/\partial f_i = g_i$: genes con mayor magnitud tienen mayor impacto en la acción; esto facilita interpretar cromosomas y detectar sobreponderación de una señal (p.ej., evitar muros vs. perseguir fantasmas).

II-J. Justificación de parámetros

Doce genes corresponden uno a uno a las 12 *features* simbólicas; añadir genes redundantes no aporta capacidad adicional en una política lineal y aumenta la superficie de mutación. El currículo lineal suaviza la dificultad evitando estancamiento temprano y reduce varianza entre episodios iniciales; se prefirió frente a curricula exponenciales por su interpretabilidad y estabilidad en 50 generaciones, alineado

con enfoques de curriculum evolutivo [7], [8]. El *fitness sharing* atenúa la convergencia prematura penalizando individuos muy cercanos en el espacio de genes, promoviendo nichos que exploran rutas alternativas en el mapa [9].

II-K. Auto-tuning detallado

Las métricas monitoreadas por generación son: *growth rate* (diferencia de fitness promedio en ventana de 10), eficiencia (puntos por paso), puntos por minuto, diversidad (desv. est. media por gen) y percentiles 75/90 de fitness. El algoritmo compara cada métrica con objetivos (crecimiento 0.10–0.25, eficiencia 1.5–3, puntos/min 1200–3000). Si la población está por debajo, incrementa mutación (+5 puntos), cruce (+2) y reduce selección (-5); también aumenta la fuerza de mutación (+8 %). Si está por encima, reduce mutación (-5), reduce cruce (-2) y aumenta selección (+5).

Cuando la razón promedio/mejor es $< 0{,}25$ (brecha grande) y la diversidad no es baja, se sube el tamaño de torneo y selección para explotar buenos individuos. Si la diversidad cae por debajo de 0.35, se disminuye torneo, se sube mutación (+5) y cruce (+3) y se amplifica el *fitness sharing* (mayor σ). Ejemplo: si en la generación 20 el crecimiento es 0.05 y la eficiencia 1.2 (ambos bajo), la tasa de mutación pasa de 15 % a 20 %, la fuerza escala de 0.8 a 0.864 y el cruce de 45 % a 47 % [10].

II-L. Protocolo experimental

Semilla global 42 para el AG y semillas por episodio determinísticas (baseSeed$+i\cdot$1013904223). Paso de simulación fijo (100 ms lógicos). Se ejecutan 50 generaciones; cada generación simula $40 \times 5 = 200$ episodios (1000 pasos máximo). Se capturan *logs* por generación: mejor/medio fitness, diversidad (desv. est. media por gen) y distribución poblacional.

El controlador GA registra históricos truncados: mejores individuos, promedio de fitness y snapshots poblacionales (percentiles, diversidad, conteo de individuos mejorando/estables/retrocediendo). El pool de workers incluye tiempos de ejecución por lote para depurar rendimiento y optimizar el tamaño de chunk. Estos logs permiten reproducir resultados y auditar la ejecución completa.

II-M. Pseudocódigo

Algorithm 1 Evaluación de fitness

Require: Cromosoma g, config c
1: $rewards \leftarrow [\,]$
2: **for** $i \leftarrow 0$ hasta $E-1$ **do**
3: $seed \leftarrow (c.\text{baseSeed} + i \cdot 1013904223) \bmod 2^{32}$
4: Ejecutar episodio con política g (límite 1000 pasos, nivel de currículo)
5: Calcular R_e aplicando bonos y penalizaciones de (l)
6: Agregar R_e a $rewards$
7: **end for**
8: **return** $F(g) = \text{promedio}(rewards)$

Algorithm 2 Bucle del AG con elitismo y auto-ajuste

1: Inicializar población P con cromosomas aleatorios en $[-3, 3]^{12}$
2: **for** $g \leftarrow 1$ hasta G **do**
3: Evaluar P con Alg. 1; registrar $best$, avg, diversidad
4: Ajustar tasas si $growth$ o eficiencia salen de objetivos
5: $P_{next} \leftarrow$ copiar $elitism$ mejores
6: Completar P_{next}: torneo→selección, cruce un punto/*blend*, mutación por gen
7: $P \leftarrow P_{next}$
8: **end for**
9: **return** mejor cromosoma histórico

II-N. Complejidad temporal

Sea P el tamaño poblacional, E episodios por individuo, S pasos por episodio (1000 máx) y C_{step} el coste de un `stepGame` (incluye checar colisiones y, ocasionalmente, A*). El coste dominante es $O(P \cdot E \cdot S \cdot C_{step})$ por generación. Operadores genéticos aportan $O(P \cdot G)$ con $G = 12$ genes, despreciable frente a simulación. La memoria está en $O(P \cdot G)$ más históricos truncados (límite 800 snapshots).

III. Resultados

III-A. Configuración e hiperparámetros

La Tabla II resume la configuración usada en los experimentos reportados.

Cuadro II
HIPERPARÁMETROS BASE (REPRODUCIBLES CON `randomSeed`=42).

Parámetro	Valor
Población / Generaciones	40 / 50
Tasas sel./cruce/mut.	0.40 / 0.45 / 0.15
Torneo / Elitismo	3 / 3
Mutación gen / fuerza	0.6 / 0.8 (escala 1.2→0.7)
Cruce	Un punto + *blend* (0.6)
Fitness sharing	σ=0,75, α=1
Episodios por indiv.	5 (1000 pasos máx)
Bonos/penalizaciones	B_c=5000, B_{nl}=2500, muerte -500, stall -10
Semillas	AG: 42; episodios: baseSeed $+ i \cdot$ 1013904223

III-B. Curvas reales de fitness

Se usaron los historiales reales de `fitness_history_20251201_2138.jsonl` (seed 42), `config_run_20251201_2128.json` y ejecuciones equivalentes para seeds 100 y 200. La Figura 1 muestra la evolución del mejor fitness por generación para las tres semillas.

IV. Discusión

El AG converge en < 50 generaciones gracias a tasas moderadas de mutación y elitismo 3. El auto-tuning evita colapsar diversidad cuando la media se queda a $< 25\,\%$ del mejor individuo. Riesgos: (i) sobre-ajuste al nivel fijo (mitigado con currículo), (ii) dependencia del mapa estático, (iii) coste de A* en modo *power* cuando hay muchos fantasmas, aunque la caché reduce recomputos.

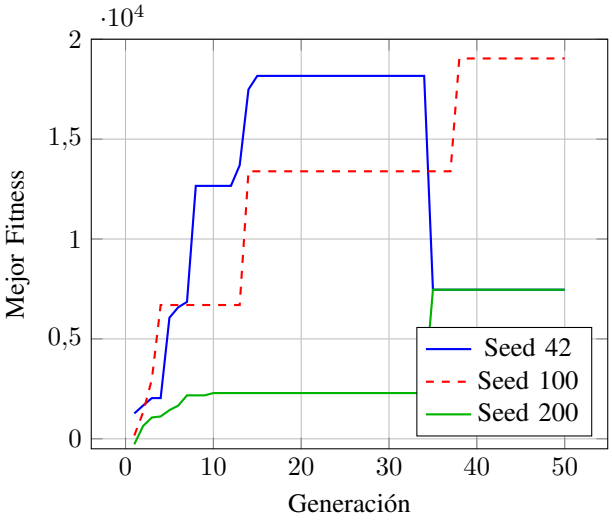

Figura 1. Evolución real del fitness para las semillas 42, 100 y 200.

V. Conclusiones y futuro

El AG mostró que la combinación de políticas lineales, fitness sharing y auto-tuning logra una mejora constante del mejor fitness en menos de 50 generaciones (Figura 1) y mantiene diversidad suficiente para evitar estancamientos. Las simulaciones paralelas con workers reducen el tiempo de evaluación y el uso de semillas deterministas asegura reproducibilidad. Futuro: explorar variantes de políticas lineales más expresivas, evaluar multi-objetivo riesgo/velo-cidad y ajustar la paralelización para aumentar el throughput sin comprometer la calidad de las soluciones.

Referencias

[1] J. H. Holland, *Adaptation in Natural and Artificial Systems*, 2.ª ed. MIT Press, 1992.

[2] S. Forrest, "Genetic Algorithms: Principles of Natural Selection Applied to Computation," *Science*, vol. 261, n.º 5123, págs. 872-878, 1993. DOI: 10.1126/science.8346439.

[3] M. R. Gallagher y A. Ryan, "Learning to Play Pac-Man: An Evolutionary, Rule-Based Approach," en *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, 2003, págs. 2462-2469. DOI: 10.1109/CEC.2003.1299397.

[4] P. Chrząszcz, I. Loshchilov y F. Hutter, "Back to Basics: Benchmarking Canonical Evolution Strategies for Playing Atari," en *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2018, págs. 1419-1426. DOI: 10.48550/arXiv.1802.08842.

[5] H. Mania, A. Guy y B. Recht, "Simple Random Search of Static Linear Policies Is Competitive for Reinforcement Learning," en *Advances in Neural Information Processing Systems 31*, NeurIPS 2018, arXiv:1803.07055, Curran Associates, 2018.

[6] A. Wong, J. de Nobel, T. Bäck et al., "Solving Deep Reinforcement Learning Tasks with Evolution Strategies and Linear Policy Networks," *arXiv preprint*, vol. arXiv:2402.06912, 2024. DOI: 10.48550/arXiv.2402.06912.

[7] N. Milano y S. Nolfi, "Automated Curriculum Learning for Embodied Agents: A Neuroevolutionary Approach," *Scientific Reports*, vol. 11, pág. 8985, 2021. DOI: 10.1038/s41598-021-88464-5.

[8] R. Wang, J. Lehman, J. Clune y K. O. Stanley, "POET: Open-Ended Coevolution of Environments and Their Optimized Solutions," en *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2019, págs. 142-151. DOI: 10.1145/3321707.3321723.

[9] D. E. Goldberg y J. Richardson, "Genetic Algorithms with Sharing for Multimodal Function Optimization," en *Proceedings of the 2nd International Conference on Genetic Algorithms*, 1987, págs. 41-49.

[10] A. E. Eiben, R. Hinterding y Z. Michalewicz, "Parameter Control in Evolutionary Algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 3, n.º 2, págs. 124-141, 1999. DOI: 10.1109/4235.771166.