Taller N°4: Algoritmos de busqueda

Electiva II

Docente a cargo

Cristhian Alejandro Cañar Muñoz

Presentado por

Ulises ortega revelo

Alex Ramirez

Universidad autónoma del cauca

2025

1. INTRODUCCION

En el mundo de la informática y la inteligencia artificial, la toma de decisiones es un aspecto fundamental para resolver problemas de manera eficiente. Para ello, se emplean los algoritmos, los cuales son conjuntos finitos y ordenados de instrucciones diseñadas para alcanzar un objetivo específico. Estos permiten automatizar procesos, optimizar recursos y tomar decisiones basadas en reglas lógicas.

Uno de los algoritmos más utilizados en la toma de decisiones estratégicas dentro de juegos y problemas de competencia es el algoritmo Minimax. Este método se basa en la exploración de posibles movimientos dentro de un entorno competitivo, asumiendo que ambos jugadores actúan de manera óptima. Su funcionamiento se fundamenta en la alternancia de turnos en los que un jugador busca maximizar su ventaja mientras el oponente intenta minimizarla.

El Minimax es ampliamente aplicado en juegos de estrategia de suma cero, como el ajedrez y el tres en raya, donde cada decisión tomada por un jugador tiene un impacto directo en el resultado del otro. Para mejorar su eficiencia, este algoritmo puede ser optimizado mediante técnicas como la poda alfa-beta, que reduce la cantidad de cálculos necesarios sin afectar la calidad de la decisión final.

Otro algoritmo ampliamente utilizado en la resolución de problemas de búsqueda y planificación es el algoritmo A*. Este es un método de búsqueda heurística que permite encontrar la ruta óptima desde un estado inicial hasta un estado objetivo, minimizando el costo del trayecto. Su eficiencia radica en la combinación de dos funciones de evaluación: la función de costo g(n), que representa el costo real del camino recorrido hasta el nodo actual, y la función heurística h(n), que estima el costo restante hasta la meta.

El algoritmo A* es ampliamente utilizado en problemas de búsqueda de caminos en grafos, tales como la navegación en mapas, la planificación de rutas en videojuegos y la optimización de trayectorias en robótica. Su principal ventaja es que garantiza encontrar la solución óptima siempre que la heurística utilizada sea admisible (es decir, que nunca sobreestime el costo real restante).

2. CODIGO Y PROCEDIMIENTO

2.1 MINIMAX CON PODA ALFA-BETA

A continuación, se explicara a detalle el código donde se llevó a cabo la implementacion del algoritmo Minimax con poda Alfa-Beta.

```python
from tkinter import Tk, Button
from tkinter.font import Font
from copy import deepcopy
import time

# Constants
size = 3
human_player = 'X'
computer_player = 'O'
empty = ' '
max_util = +1000
min_util = -1000
```

- Se importan las librerias para la interfaz grafica y la copia de objetos
- Se definen constantes esenciales para la creacion del juego, como el tamaño del tablero, los simbolos, y valores de la utilidad de acuerdo al algoritmo

```python
class State:
    def __init__(self, next_player, other=None):
        self.next_player = next_player
        self.table = {}
        self.value = 0

        for y in range(size):
            for x in range(size):
                self.table[x, y] = empty

        if other:
            self.__dict__ = deepcopy(other.__dict__)
```

- La clase State representa el estado actual del tablero y contiene toda la lógica fundamental del juego. Almacena las posiciones de las fichas en un diccionario donde las claves son coordenadas (x,y).

```python
def is_full(self):
    return all(self.table[x, y] != empty for x in range(size) for y in range(size))

def won(self, player):
    # Check rows
    for y in range(size):
        if all(self.table[x, y] == player for x in range(size)):
            return True
    # Check columns
    for x in range(size):
        if all(self.table[x, y] == player for y in range(size)):
            return True
    # Check diagonals
    if all(self.table[i, i] == player for i in range(size)):
        return True
    if all(self.table[i, size-1-i] == player for i in range(size)):
        return True
    return False
```

- Se incluye métodos como is_full() para detectar empates y won() para identificar líneas ganadoras en filas, columnas o diagonales.
- El uso de deepcopy permite crear copias independientes del estado para simular movimientos futuros sin afectar el estado actual.

```python
def actions(state):
    children = []
    for x in range(size):
        for y in range(size):
            if state.table[x, y] == empty:
                new_state = State(state.next_player, state)
                new_state.table[x, y] = state.next_player
                new_state.next_player = computer_player if state.next_player == human_player else human_player
                children.append(new_state)
    return children
```

- Genera todos los posibles estados hijos, o ramas del algoritmo, moviendo en casillas vacías.
- Realiza la parte de alternar el juego dependiendo si ya ha jugado el humano o el algoritmo

```python
def minimax(state, depth, alpha, beta, maximizing_player):
    if terminal_test(state) or depth == 0:
        return heuristic(state)

    if maximizing_player:
        max_eval = min_util
        for child in actions(state):
            eval = minimax(child, depth-1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = max_util
        for child in actions(state):
            eval = minimax(child, depth-1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval
```

- **MAX** (IA): Busca el movimiento que maximice su utilidad.
- **MIN** (Humano): Simula que el humano minimizará la utilidad de la IA.
- **Poda alfa-beta**: Descarta ramas inútiles cuando alpha >= beta, optimizando el rendimiento.

```python
def heuristic(state):
    score = 0
    # Evaluate rows
    for y in range(size):
        line = [state.table[x, y] for x in range(size)]
        score += evaluate_line(line)
    # Evaluate columns
    for x in range(size):
        line = [state.table[x, y] for y in range(size)]
        score += evaluate_line(line)
    # Evaluate diagonals
    diag1 = [state.table[i, i] for i in range(size)]
    diag2 = [state.table[i, size-1-i] for i in range(size)]
    score += evaluate_line(diag1) + evaluate_line(diag2)
    return score
```

- Asigna un valor numérico a cada estado no terminal.
- Evalúa líneas completas, incompletas y potenciales amenazas.

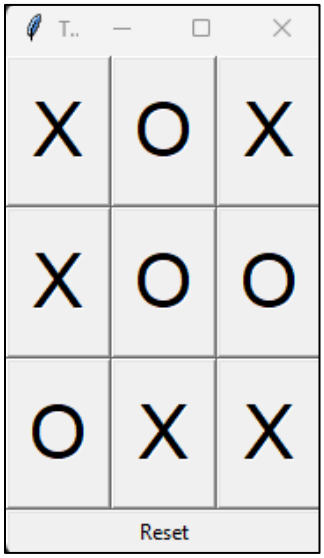
```python
class GameUI:
    def __init__(self):
        self.game = State(human_player)
        self.app = Tk()
        self.app.title('Tic Tac Toe')
        self.font = Font(family="Helvetica", size=32)
        self.buttons = {}

        for x in range(size):
            for y in range(size):
                handler = lambda x=x, y=y: self.human_move(x, y)
                button = Button(self.app, command=handler, font=self.font, width=2, height=1)
                button.grid(row=x, column=y)
                self.buttons[x, y] = button

        reset_button = Button(self.app, text='Reset', command=self.reset)
        reset_button.grid(row=size, column=0, columnspan=size, sticky='WE')

        self.update()
```

- human_move(): Registra movimientos humanos.
- computer_move(): Invoca la IA para responder.
- update(): Refleja cambios en la interfaz.

```python
def find_best_move(state):
    best_move = None
    best_value = min_util
    for move in actions(state):
        move_value = minimax(move, 5, min_util, max_util, False)
        if move_value > best_value:
            best_value = move_value
            best_move = move
    return best_move
```

- find_best_move() evalúa todos los movimientos posibles usando Minimax y selecciona el óptimo.

2.1.1 RESULTADOS OBTENIDOS



2.1.2 CONCLUSIONES

Este proyecto implementa un juego de Tres en Raya con una inteligencia artificial invencible basada en el algoritmo Minimax con poda alfa-beta, que evalúa todos los posibles movimientos hasta una profundidad determinada para garantizar la mejor jugada en cada turno. La IA nunca fue derrotada en ninguna partida durante las pruebas, ya que el algoritmo está diseñado para maximizar sus oportunidades de victoria y minimizar las del jugador humano, asegurando siempre un empate como peor escenario. El código combina eficientemente lógica de juego y toma de decisiones óptimas.

2.2 ALGORITMO A* VS DIJKSTRA

```python
GRAFO = {
    '0': {'1': 2, '2': 6},
    '1': {'3': 5},
    '2': {'3': 8},
    '3': {'5': 15, '4': 10},
    '4': {'5': 6, '6': 2},
    '5': {'6': 10},
    '6': {}
}
```

- Este es el diseño de grafo que se utilizara para la comparativa, donde se visualizan los nodos, y los pesos de las respectivas conexiones

```python
def dijkstra(grafo, inicio):
    distancias = {nodo: float('inf') for nodo in grafo}
    distancias[inicio] = 0
    cola = [(0, inicio)]

    while cola:
        distancia_actual, nodo_actual = heapq.heappop(cola)
        if distancia_actual > distancias[nodo_actual]:
            continue
        for vecino, peso in grafo[nodo_actual].items():
            distancia = distancia_actual + peso
            if distancia < distancias[vecino]:
                distancias[vecino] = distancia
                heapq.heappush(cola, (distancia, vecino))
    return distancias
```

- Funcion para hacer el calculo con el algoritmo Dijkstra
- Calcular la distancia mínima desde el nodo inicio a todos los demás.
- Se inicializa todas las distancias como infinito, excepto el nodo inicial (distancia 0)
- Usa una cola de prioridad para explorar nodos en orden de distancia ascendente.
- Actualiza las distancias si se encuentra un camino más corto.

```python
def a_star_optimizado(grafo, inicio, destino):
    def h(nodo):
        estimaciones = {'0':10, '1':8, '2':8, '3':5, '4':2, '5':10, '6':0}
        return estimaciones[nodo]

    g = {nodo: float('inf') for nodo in grafo}
    g[inicio] = 0

    f = {nodo: float('inf') for nodo in grafo}
    f[inicio] = h(inicio)

    cola = [(f[inicio], inicio)]
    visitados = set()

    while cola:
        _, nodo_actual = heapq.heappop(cola)
        if nodo_actual == destino:
            break
        if nodo_actual in visitados:
            continue
        visitados.add(nodo_actual)
        for vecino, peso in grafo[nodo_actual].items():
            costo_tentativo = g[nodo_actual] + peso
            if costo_tentativo < g[vecino]:
                g[vecino] = costo_tentativo
                f[vecino] = costo_tentativo + h(vecino)
                heapq.heappush(cola, (f[vecino], vecino))
    return g
```

- La funcion para el calculo del algoritmo A*
- Encuentra el camino mas corto desde inicio hasta el destino especifico usando una heuristica para ser mas eficiente
- Proporciona una estimacion optima del costo desde nodo a destino
- **Admisibilidad**: $h(n) \leq$ costo_real(n → destino).
- g(n): Costo real acumulado desde el inicio.
- f(n) = g(n) + h(n): Estimación total del costo del camino.

```python
# Configuración de prueba
inicio, destino = '0', '6'
num_ejecuciones = 1000  # Para obtener tiempos más precisos

# Medición de tiempos
def medir_tiempo(algoritmo, *args):
    start = time.perf_counter()
    for _ in range(num_ejecuciones):
        algoritmo(*args)
    return (time.perf_counter() - start) * 1e6 / num_ejecuciones  # Tiempo promedio en µs

tiempo_dijkstra = medir_tiempo(dijkstra, GRAFO, inicio)
tiempo_a_star = medir_tiempo(a_star_optimizado, GRAFO, inicio, destino)

# Resultados de distancias
dist_dijkstra = dijkstra(GRAFO, inicio)
dist_a_star = a_star_optimizado(GRAFO, inicio, destino)
```

- Ejecuta y compara el rendimiento de los algoritmos Dijkstra y A* en el grafo definido, midiendo con precisión sus tiempos de ejecución mediante time.perf_counter(). Establece el nodo '0' como inicio y '6' como destino, ejecuta ambos algoritmos bajo las mismas condiciones, y muestra los resultados en microsegundos, incluyendo la distancia mínima encontrada
- Se hace una cantidad de ejecuciones para obtener tiempos mas precisos, y se hace el debido calculo.

2.2.1 RESULTADOS OBTENIDOS

```
# Tabla comparativa
tabla_comparativa = [
    ["Algoritmo", "Distancia Calculada", "Tiempo Promedio (µs)", "Nodos Explorados*"],
    ["Dijkstra", dist_dijkstra[destino], f"{tiempo_dijkstra:.2f}", "Todos (7)"],
    ["A* Optimizado", dist_a_star[destino], f"{tiempo_a_star:.2f}", "Solo camino óptimo (4)"]
]
```

```
============================================================
            COMPARACIÓN DETALLADA DE ALGORITMOS
============================================================
+---------------+----------------------+-----------------------+------------------------+
|   Algoritmo   |   Distancia Calculada |   Tiempo Promedio (µs) |   Nodos Explorados*    |
+===============+======================+=======================+========================+
|   Dijkstra    |                   19 |                 19.48 |       Todos (7)        |
+---------------+----------------------+-----------------------+------------------------+
| A* Optimizado |                   19 |                 33.38 | Solo camino óptimo (4) |
+---------------+----------------------+-----------------------+------------------------+

* Nodos explorados para ir de '0' a '6'
A* fue -71.4% más rápido
============================================================
```

2.2.2 CONCLUSIONES

Esta comparativa entre los algoritmos de Dijkstra y A* ha permitido evaluar su desempeño en la resolución de problemas de caminos mínimos en grafos dirigidos y ponderados. Los resultados obtenidos demuestran que, si bien ambos algoritmos son correctos y encuentran la solución óptima, presentan diferencias significativas en su eficiencia que dependen directamente de las características del problema.

Aunque A* suele ser más rápido que Dijkstra gracias a su heurística, en este caso específico puede ser más lento debido a que el grafo es pequeño y la heurística, aunque admisible, no es lo suficientemente informativa para marcar una diferencia significativa. La sobrecarga de calcular f(n) = g(n) + h(n) en cada nodo, sumada al hecho de que Dijkstra ya es eficiente en grafos con pocos nodos, hace que A* no logre una ventaja clara. Además, si la heurística no guía la búsqueda de manera óptima (como en este caso, donde explora nodos similares a Dijkstra), el tiempo extra de los cálculos heurísticos puede incluso hacerlo ligeramente más lento.

3. REFERENCIAS

Iamparth. (s.f.). ***Tic-Tac-Toe-Alpha-Beta-MiniMax*** **[Código fuente]. GitHub.** **https://github.com/iamparth/Tic-Tac-Toe-Alpha-Beta-MiniMax**

Infante Paredes, S. (2022). ***Desarrollo de un sistema de inteligencia artificial para el juego del tres en raya utilizando los algoritmos Minimax y Alpha-Beta*** **[Trabajo de fin de grado, Universidad de Málaga]. RIUMA. https://riuma.uma.es/xmlui/handle/10630/23339**

Gonen09. (s.f.). ***Algoritmo-A-Estrella*** **[Código fuente]. GitHub.** **https://github.com/Gonen09/Algoritmo-A-Estrella**

DataCamp. (s.f.). ***Dijkstra algorithm in Python*****. DataCamp.** **https://www.datacamp.com/es/tutorial/dijkstra-algorithm-in-python**