BURSA ULUDAĞ ÜNİVERSİTESİ

BİLGİSAYAR MÜHENDİSLİĞİ

2024-2025 EĞİTİM ÖĞRETİM YILI GÜZ DÖNEMİ

YAPAY ZEKA RAPORU

A* ALGORİTMASI İLE SUDOKU ÇÖZEN AJAN

032290004 BARIŞ IŞIK

032290004@ogr.uludag.edu.tr

032290008 MURAT BERK YETİŞTİRİR

032290008@ogr.uludag.edu.tr

032290037 BUĞRA ÖZGEN

032290037@ogr.uludag.edu.tr

Sudoku Oynayan Ajan İçin PEAS Tanımı

- **Performance Measure:** Bulmacayı çözmek, kurallara uymak
- **Environment:** Bulmaca tahtası
- **Actuators:** Hamleler
- **Sensors:** Bulmaca kareleri

Ortam Özelliklerinin Sınıflandırılmasını

- **Observability:** Tamemen gözlenebilir
- **Determinism:** Deterministik
- **Episodic vs. Sequential:** Sıralı
- **Static vs Dynamic:** Dinamik
- **Discrete vs Continuous:** Ayrık
- **Benign vs. Adversarial:** İyi huylu
- **Single-agent vs Multi-agent:** Tek ajan

Ajan Tipi: Hedef Odaklı Ajan

Durum, Hedef, Ardıl Fonksiyonu, Bedel (Değer) Fonksiyonu Veya Sezgi fonksiyonu

- **Durum Fonksiyonu:** Sudoku probleminin her durumu, 9x9'luk bir sudoku tablosunda hücrelere doldurulmuş olan sayılar dizisidir. Başlangıç durumu, bize verilen kısmi dolu sudoku tablosudur. Çözüm yolunda her durum, önceki durumun devamı olan ve belirli kurallara göre yeni bir sayının tabloya yerleştirildiği bir tablo halidir.
- **Hedef Fonksiyonu:** Ajanın hedefi, tüm hücrelerin 1'den 9'a kadar rakamlarla doldurulmuş olduğu, sudoku kurallarına göre geçerli bir tabloya ulaşmaktır. Yani:
    - Her satırda 1'den 9'a kadar olan tüm rakamlar yalnızca bir kez yer alır.
    - Her sütunda 1'den 9'a kadar olan tüm rakamlar yalnızca bir kez yer alır.
    - 3x3'lük alt bloklarda 1'den 9'a kadar olan tüm rakamlar yalnızca bir kez yer alır.
- **Ardıl Fonksiyonu:** Ajanın mevcut durumdan bir sonraki duruma geçişini belirleyen fonksiyondur. Bu durumda, ajan tabloya kurallara uygun olarak bir rakam yerleştirebilir. Her ardıl durumda ajan, boş bir hücreye geçerli olan bir rakamı yerleştirir ve bu işlem, tüm hücreler dolana kadar devam eder.
- **Bedel (Değer) Fonksiyonu:** Sudoku çözme sürecinde her hamle aynı bedelde olduğundan her adımın maliyeti 0.5 olarak alındı. A* algoritmasında, başlangıçtan mevcut duruma kadar olan toplam adım sayısı veya "g" fonksiyonu olarak ifade edilir.
- **Sezgi Fonksiyonu:** A* algoritmasında sezgi fonksiyonu (h), bir durumun hedef duruma olan tahmini uzaklığını hesaplar. Bu ajan için sezgi fonksiyonu henüz doldurulmamış hücre sayısı olarak ele alındı.

A* ile Hedef Arasındaki Bağlantı

- Ajan A* algoritması ile hedefe ulaşabiliyor.
- **Zaman Karmaşıklığı**
    - **En Kötü Durum**: A* algoritması, sezgi fonksiyonu iyi değilse, gereksiz düğümleri genişleterek üstel zaman karmaşıklığına ulaşabilir, yani O(b^d),Burada b ortalama dallanma faktörü (boş hücrelerde denenen aday sayısı) ve d, çözüm derinliğidir.
- **Uzay Karmaşıklığı**
    - A*, genişletilen tüm düğümleri bellekte saklar, bu nedenle uzay karmaşıklığı da O(b^d) olabilir. Bellek ihtiyacı, algoritmanın uygulanabilirliğini sınırlandırabilir.

Çözüm Analizi

- **Bütünlük:** A* algoritması, sudoku kurallarına uygun geçerli bir çözüm bulur, yani çözüm her zaman bütünlük sağlar.
- **Uygunluk:** Eğer A* algoritması doğru sezgi fonksiyonu kullanıyorsa, çözüm en uygun (en kısa adımlarla) olabilir. Ancak sezgi fonksiyonunun kalitesine göre, çözüm optimum olmayabilir. Bu yüzden en iyi çözümü bulup bulmadığı, kullanılan sezgi fonksiyonunun etkinliğine bağlıdır.

Kod

```
using System;
using System.Collections.Generic;
using System;
using System.Diagnostics;
using System.Threading;

namespace SudokuSolver
{
    public class SudokuSolver
    {
        private int[,] puzzle;
        private PriorityQueue<int[,], int> openSet;

        public SudokuSolver(int[,] initialPuzzle)
        {
            puzzle = initialPuzzle;
            openSet = new PriorityQueue<int[,], int>();
        }
        public void Solve()
        {
            long l = Stopwatch.GetTimestamp();
            int[,] initialState = puzzle;
            openSet.Enqueue(initialState, 0);
            while (openSet.Count > 0)
            {
                int[,] current = openSet.Dequeue();
```

```
                if (IsSolved(current))
                {
                    PrintSolution(current);
                    System.Console.WriteLine("Solved in {0}",
Stopwatch.GetElapsedTime(l).TotalMilliseconds);
                    return;
                }
                GetNeighbors(current, ref openSet);
                /*
                foreach (var neighbor in GetNeighbors(current))
                {
                    openSet.Enqueue(neighbor, 0);
                }
                */
            }

            Console.WriteLine("No solution found.");
        }

        private bool IsSolved(int[,] board)
        {
            // Check if there are any zeros left (meaning incomplete
board)
            foreach (var cell in board)
            {
                if (cell == 0) return false;
            }
            return true;
        }

        private IEnumerable<int[,]> GetNeighbors(int[,] state, ref
PriorityQueue<int[,], int> pq)
```

```
        {
            var neighbors = new List<int[,]>();

            for (int row = 0; row < 9; row++)
            {
                for (int col = 0; col < 9; col++)
                {
                    if (state[row, col] == 0)
                    {
                        for (int num = 1; num <= 9; num++)
                        {
                            if (IsValid(state, row, col, num))
                            {
                                int[,] newBoard = CopyBoard(state);
                                newBoard[row, col] = num;

                                pq.Enqueue(newBoard,
CalculateCost(newBoard));

                                //neighbors.Add(newBoard);
                            }
                        }
                        return neighbors;
                    }
                }
            }
            return neighbors;
        }
        private bool IsValid(int[,] board, int row, int col, int num)
        {
            for (int i = 0; i < 9; i++)
                if (board[row, i] == num || board[i, col] == num ||
```

```
                    board[row - row % 3 + i / 3, col - col % 3 + i %
3] == num)
                    return false;
            return true;
        }
        private int CalculateCost(int[,] board)
        {
            int emptyCells = 0;
            foreach (var cell in board)
            {
                if (cell == 0) emptyCells++;
            }
            return emptyCells; // Fewer empty cells means closer to
solution
        }
        private int[,] CopyBoard(int[,] board)
        {
            int[,] newBoard = new int[9, 9];
            Array.Copy(board, newBoard, board.Length);
            return newBoard;
        }
        private void PrintSolution(int[,] board)
        {
            for (int row = 0; row < 9; row++)
            {
                for (int col = 0; col < 9; col++)
                {
                    Console.Write(board[row, col] + " ");
                }
                Console.WriteLine();
            }
        }
    }
```

```
    // Usage
    class Program
    {
        static void Main()
        {
            int[,] puzzle = {
            {0, 0, 9, 0, 0, 0, 0, 0, 2},
            {8, 7, 5, 0, 0, 0, 0, 0, 0},
            {0, 0, 1, 0, 0, 0, 3, 0, 9},
            {0, 0, 0, 0, 0, 0, 7, 0, 0},
            {0, 0, 0, 5, 0, 7, 0, 9, 0},
            {1, 0, 0, 8, 0, 0, 5, 0, 0},
            {4, 0, 0, 0, 0, 9, 0, 0, 0},
            {0, 0, 0, 0, 3, 0, 0, 4, 6},
            {0, 8, 0, 0, 1, 0, 0, 0, 0}
        };
            SudokuSolver solver = new SudokuSolver(puzzle);
            solver.Solve();
        }
    }
}
```

Kodun Ekran Çıktısı