DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

ARTIFICIAL INTELLIGENCE

Program for Search using UCS and A* - Game Construction.

Team

Gkovaris Christos-Grigorios
Kotopoulos Vasileios
Spanou Maria

Search Program using UCS and A*

**This exercise was implemented in Java, in 3 classes.*

Our first laboratory exercise is a variation of the well-known 8-puzzle problem, where in addition to the usual moves (horizontal and vertical) to a neighboring empty space, diagonal movement is also allowed.

Our goal is to find the minimum sequence of actions from an initial state (IS) to a final state (FS).

To achieve this, we will implement two search methods, the Uniform Cost Search (UCS) method and the A Search using the best possible heuristic function h(n).

Timetable of the Assignment

Version	Date	Progress
1.0	27/3/2024	Announcement of the 1st laboratory exercise
2.0	29/3 - 15/4	Implementation of the exercise
3.0	16/5/2024	Comments on the code
4.0	17/5/2024	Implementation of the laboratory exercise report

Feature Analysis

The assignment was implemented on a machine (Lenovo Ideapad 3), with the following specifications:

- AMD Ryzen 7 (7730U)
- 8 CPU cores
- 16 Threads
- Boost Clock up to 4.5GHz
- Base Clock 2.0GHz
- CPU Socket FP6
- Intergraded Graphics (Radeon Graphics)

Furthermore, the assignment was implemented on **Windows 11 Home**, using the **Visual Studio**.

PuzzleSolver.java:

In this class, we have defined the field **private int[][] goalState**, which sets the final state as stated in the problem statement.

Then, we have implemented the following methods:

public void solve(): This method takes the initial state as an argument and uses either the UCS algorithm or the A* algorithm depending on the value of the aStar parameter. The solve method checks if the initial state is valid and creates a PriorityQueue based on the cost of the states. The choice of cost is made based on the aStar parameter, which determines whether the heuristic A* algorithm will be used. It creates a set (explored) to store the states that have already been explored and a set (inFrontier) to map the elements in the PriorityQueue for quick checking of the existing states. There is a while loop that runs until the queue is empty. In each iteration, it removes the first state from the queue and checks if it is the final state. If yes, it prints the solution and terminates; otherwise, it adds the state to the explored set and generates all possible states resulting from the current move, adding them to the queue only if they have not been explored or are not already in the queue. Then, if the queue is empty and no solution has been found, it prints an error message.

private boolean isValidState(): This method takes an array representing the state as an argument. It is used to confirm the validity of a state in the 8-puzzle, i.e., if it contains all the numbers from 0 to 8 exactly once. We create a HashSet called tiles to store the tile numbers contained in the puzzle state. Then it checks each element of the state array and adds it to the tiles set. Finally, it checks if the tiles contain all the numbers from 0 to 8 corresponding to the puzzle tiles. If any of these numbers are missing, the method returns false (indicating that the state is not valid); otherwise, the method returns true (indicating that the state can be used to solve the puzzle).

private void printSolution(): This method prints the solution path from the initial state to the final state, along with the cost and depth of the final state. Initially, it takes the path (from IS to FS) using the getPathFromStart() method of the state we have given as an argument. It then prints the path followed by listing the states at each step of the path. For each state, it uses the getState() method to get the state array and prints each row of the array. At the end of each state, it prints a blank line to separate the different states that follow. It also displays the cost of the final state and its depth, using the getCost() and getDepth() methods, respectively, of the state.

private int findZeroRow(): This method searches to find the 0 in the rows of the state.

private int findZeroCol(): This method searches to find the 0 in the columns of the state.

PuzzleState.java:

In this class, we have defined the fields:

private int[][] state: This field represents the current state of the puzzle.

private PuzzleState parent: This field represents the parent state of the current state.

private int zeroRow, zeroCol: This field represents the position of the empty element in the state.

private int cost: This field represents the cost.

private int depth: This field represents the number of moves to reach a specific state.

Then, we have implemented the following methods:

public List<PuzzleState> generateSuccessors(): This method creates and returns the successive states by moving the empty cell in different directions. Specifically, it creates an empty list to contain the successive states. Then it defines an array (directions) that contains the 8 possible movement directions for the empty cell (up, down, right, left, and the 4 diagonal moves). It then iterates through the directions array, calculating the new row and column of the empty cell based on the current position of the empty cell (zeroRow and zeroCol) and the direction. It checks if the new row and column are within the bounds of the 3x3 grid. If yes, it creates a new state array (newState), which is a deep copy of the current state array. It swaps the values of the empty cell with those of the new position, simulating the move. Finally, it creates a PuzzleState object with the new state array, the new coordinates of the empty cell, the current state as the parent, the cost of the game (increased by 1), and the depth (increased by 1). This new object is added to the list of successive states to be returned. This method is crucial for finding the solution to the puzzle as it generates the successive states that are checked and evaluated by the search algorithm.

private int[][] deepCopy: This method takes a two-dimensional array (int[][] original) and creates a deep copy of that array. First, a new array called copy is created with the same number of rows as the original array. Then, for each row of the original, a new array with the same number of elements as the original row is created using the Arrays.copyOf() method. Finally, all the values of the original are copied row by row into the copy array. The result is a new array that contains exactly the same elements as the original but is independent of it, ensuring a deep copy of the data.

public boolean isGoal(): This method is used to check if the current state of the problem, represented by an int[][] this.state array, is the goal defined by the goalState array. The method uses the static Arrays.deepEquals method, which compares two multidimensional arrays by depth, checking if their contents are the same. Thus, the isGoal() method returns true (if the current state of the problem is the same as the goal), otherwise, it returns false. This allows the program to check if the desired goal has been achieved or not (based on the two state arrays).

public List<PuzzleState> getPathFromStart(): This method creates a list (path) that contains the path from the initial state of the puzzle to the current state. Initially, an empty path list is created, and the variable current is initialized with the current state of the puzzle. In a while loop, the method checks if the current state is not null. If it is not, it adds the current state to the beginning of the path list using the add(0, current) method, which inserts the element at the beginning of the list. Then, the current variable is updated with the parent of the current state stored in the parent field. This process continues until current becomes null (until it reaches the initial state). Finally, the path list is returned with the full path from the initial state to the current state.

public int getCost(), getDepth(), getState(): These methods are accessor methods.

public int misplacedTilesHeuristic(): This method implements one of the simplest heuristic methods for the puzzle problem. Specifically, it uses the number of locations where the tiles are not in the correct position relative to the goal state. Initially, a misplacedTiles variable is defined to count the misplaced tiles. Then the method checks each position of the state array and compares the value of each position with the corresponding position in the goalState array. If the value is not zero (i.e., there is a tile) and is not the same as the corresponding value in the goal, the misplacedTiles counter is incremented. Finally, the method returns the number of misplaced tiles as the heuristic cost used to evaluate each state of the puzzle.

public boolean equals: The equals method is used to check whether a PuzzleState object is equal to another object. Initially, the method checks whether the this object is the same as the obj object using the == operator. If it is, it immediately returns true as this means that the two objects are exactly the same. Then it checks whether the obj object is null or does not belong to the same class as the this object using the getClass() method. If so, it returns false because two objects that do not belong to the same class cannot be equal. Finally, if the conditions of the previous checks are not met, a new PuzzleState object is created using obj, and the equality of the state arrays of this and the new object is checked using the static Arrays.deepEquals method. If the state arrays are equal, the method returns true; otherwise, it returns false. The method is important for confirming the equality between PuzzleState objects, mainly based on the comparison of the state arrays.

public int hashCode(): The hashCode method is used to create a unique numerical code (hash code) corresponding to each object of the PuzzleState class. In this implementation, the numerical code is calculated using the static Arrays.deepHashCode method, which creates a hash code based on the contents of the state array by depth. The created hash code is necessary for efficient storage and retrieval of objects in data structures based on hash, such as HashMap collections.

Test.java:

The main method is the entry point to our program and initializes the solution to a puzzle. Initially, it creates a Scanner object to receive input from the user via the keyboard. Then it asks the user to enter the initial state of the puzzle, row by row, using a space as a delimiter. The numbers entered are stored in a 3x3 array representing the initial state of the puzzle. Then, a PuzzleSolver class object is created, which will be used to solve the puzzle. It runs two different solution search algorithms, the Uniform Cost Search (UCS) and the A* Search. Each solution algorithm is called with the initial state of the puzzle entered by the user. The parameter false is passed to the solve method for the execution of UCS, while the value true is passed for A* Search. Finally, the solutions of each algorithm are displayed on the screen.

Game Construction

**This exercise was implemented in Java, in 2 classes.*

The game consists of two players (MIN and MAX), where each player alternates and places one of the three letters (C, S, E) in any empty position of a 3x3 grid.

The game starts from an initial state where the letter S is mandatory either in the left or right position of the middle row. The game ends if a player forms one of the two final triplets ('CSE' or 'ESC') horizontally, vertically, or diagonally in consecutive positions of the grid, or if the grid fills without forming any of the desired triplets, resulting in a draw.

For the construction of the program, the MINIMAX algorithm will be used to choose the move made by the MAX player each time, taking into account the current state of the game.

Timeline for 2st Laboratory Exercise

Version	Date	Progress
1.0	27/3/2024	Announcement of the 2nd laboratory exercise
2.0	22/4 - 1/5	Implementation of the exercise
3.0	16/5/2024	Comments on the code
4.0	17/5/2024	Implementation of the laboratory exercise report

Feature Analysis

The assignment was implemented on a machine (Lenovo Ideapad 3), with the following specifications:

- AMD Ryzen 7 (7730U)
- 8 CPU cores
- 16 Threads
- Boost Clock up to 4.5GHz
- Base Clock 2.0GHz
- CPU Socket FP6
- Intergraded Graphics (Radeon Graphics)

Furthermore, the assignment was implemented on **Windows 11 Home**, using the **Visual Studio**.

GridGame.java:

In this class, we have defined the fields:

private static final char EMPTY: This field represents an empty cell on the game board.

private static final int MAX: This field represents the constant value for the MAX player.

private static final int MIN: This field represents the constant value for the MIN player.

private char[][] board: This field represents the game board.

private int currentPlayer: This field keeps track of which player's turn it is. It is initialized to the MAX player.

private Map<String, Integer> memo: This field is used for memorization, storing board states as strings and their corresponding scores. It helps to avoid repetitive steps in the MINIMAX algorithm.

Then, we have implemented the following methods:

public static void main(): The main method is the starting point of the program. Here, a new instance of the GridGame class is created with the name game, and then the runGame() method is called to start the game. The runGame() method is responsible for managing the game loop and interacting with the user.

private void bestMove(): This method implements the functionality of finding the best possible move for the MAX player using the Minimax algorithm. Initially, it initializes variables for the best value and the best move. Then, it explores all possible moves that the MAX player can make, i.e., the empty positions on the board. For each empty position, it tries all possible letters that the MAX player can place (C, E, S). In each iteration, it places a letter in that position and calculates the value of the move using the minimax method. Then, it removes the letter from the position to reuse the position for other moves. During this step, the method updates the best value and the best move based on the value of the move returned by the Minimax algorithm. Finally, it applies the best move to the board and prints it on the screen to inform the user of the move made by the MAX player.

private int minimax(): This method implements the Minimax algorithm to determine the optimal move in a game. The method takes the search depth and a boolean variable indicating whether the current player is the Maximizing Player as arguments. Initially, it checks if the board state has already been stored in the cache. If so, it returns the corresponding result from the cache. Then, it checks if there is a win or if the board is full. If either condition is met, it returns the score based on the player's turn and stores it in the cache. Then, it chooses the initial bestValue based on the current player. It then checks every possible move that can be made by the player in an empty cell. For each move, it recursively calls itself, increasing the search depth and alternating the player's turn. After getting the result of the recursive call, it returns the best value for the current player, either the maximum if it is the Maximizing Player or the minimum if it is the Minimizing Player. Finally, it stores the best value for the current board state in the cache and returns it. This way, the method ensures that the Minimax algorithm explores the move tree in depth, evaluating and storing the results for future use to find the optimal move.

private void runGame(): This method implements the main game loop. Initially, it initializes the game board and prints its initial state. It then enters an infinite while loop, which runs until there is a winner or a draw. During each iteration of the loop, it checks whose turn it is to play. If the current player is MAX, the bestMove() method is called to choose the best possible move. Otherwise, a message is printed indicating MIN player's turn, and the user is asked to enter the row, column, and letter they want to place on the board. After checking the validity of the move, if the tile can be placed in the position chosen by the MIN player, it is placed on the board. Otherwise, an error message is printed, and the iteration continues. The board is then printed after the move. Next, it checks if there is a winner or if the board is full. If either condition is met, the corresponding win or draw messages are printed, and the loop terminates. Finally, the player's turn changes, preparing for the next round. When the loop terminates, the method closes the scanner used for user input.

private String boardToString(): This method creates a representation of the current game board as a string. It uses a StringBuilder object to build the string incrementally. The method iterates through each row of the board, and for each cell in the row, it adds its value to the StringBuilder. Finally, it returns the resulting string. This way, the method converts the game board into a string that can be used for various purposes, such as saving the game state or using it as a key for the cache in the Minimax algorithm.

private void initializeBoard(): This method initializes the game board with a random initial state. It uses a Random class object to generate a random number between 0 and 1. Based on the generated random number, it chooses whether to place the letter 'S' in one of the two predefined positions on the board. This creates different initial conditions for the game, adding variety and interest each time the game starts. This method is executed once at the beginning of the game to set the initial state of the board.

private boolean canPlace(): This method checks if a letter can be placed in a specific position on the game board. Specifically, it checks if the coordinates (row, col) given are within the board's limits (i.e., between 0 and 2 for a 3x3 board) and if the specific position is empty, meaning no letter has already been placed in that position. The method returns true if the move is valid and can be made, and false otherwise. This function is important for checking the validity of moves made by the player in the game, ensuring that moves are made only in valid board positions and do not replace existing letters.

private boolean checkWin(): This method checks if there is a winner in the game. Specifically, it checks if there is a row, column, or diagonal on the game board consisting of the letters "C," "S," and "E" in any order. To achieve this, the method calls the checkLines method with predefined patterns representing the various ways a winning line can be formed, such as "CSE" and "ESC." If any of these conditions are met, the method returns true, indicating that there is a winner in the game. If none of the above conditions are met, the method returns false, indicating that there is no winner yet. This function is crucial for checking the game status and determining the winner.

private boolean checkLines(): This method checks if there is a line or diagonal on the game board that contains one of the predefined letter sequences specified by the patterns passed to the method. The method takes an array of patterns as input, containing the patterns to be checked for a win. During execution, the method creates two strings (rowString and colString) for each row and column of the board, respectively, gathering the letters in each row and column. It then checks if any of these strings contain one of the patterns specified in the patterns array. If so, it returns true, indicating that there is a winner. Then, the method checks the two diagonals of the board for the same patterns. If either diagonal contains one of the patterns, it returns true. If none of the above conditions are met, the method returns false, indicating that there is no winner on the game board. This function is critical for checking the game status and recognizing the winner.

private boolean isBoardFull(): This method checks if the game board is full, meaning all cells on the board have a value other than the EMPTY string. During execution, the method iterates through all cells of the board using two nested variables, i and j, which run from 0 to 2 (the board size is 3x3). Each time a cell is checked, the method checks if its value is equal to the EMPTY string. If it finds even one empty cell, the method returns false, indicating that the board is not full. If all cells have a value other than EMPTY, the method returns true, indicating that the board is full and no more letters can be placed. This function is crucial for checking the game status and recognizing if the board is full to determine if a draw has occurred.

private void printBoard(): This method is responsible for printing the game board on the screen. During execution, the method displays a "board" identifier indicating the start of the board. Then, it uses two nested variables, i and j, to access each cell of the board. For each cell, the method checks if it is empty (i.e., has the value EMPTY) and prints the symbol " instead of the cell value. If the cell is not empty, it prints the cell value. The method continues this way until all rows of the board are printed. This way, the method ensures that the game board is clearly displayed on the screen, with each empty cell represented by the symbol.

Move.java:

This class contains the constructor Move. It represents a move in a game and includes three member variables: row, col, and letter (which represent the row, column, and letter placed in a cell of the game board, respectively). The constructor accepts the values of these variables as arguments and assigns them to the corresponding fields of the class. Using this class, we can store the moves made by each player during the game. This allows for the recording of players' actions and the management of the game based on the moves that have been made. Additionally, using this class makes it easier to transfer and process the data related to the moves at any point in the program.