Bicol University
College of Science
CSIT Department
AY 2023-2024

PROGRAMMING PROJECT 1
ARTIFICIAL INTELLIGENCE
1ST SEMESTER

Dionisio, Ryan Kim
Garcia, Jhon Carl Angelo
Muncal, Aragorn
Tolosa, Noli Joseph
Vergara, Joshua

Write a program in C/C++ implementing a blind search strategy, i.e., Iterative Deepening Search (IDS) and a heuristic search strategy, i.e. A* Search with graph search to solve the 15-puzzle problem using Manhattan distance as the heuristic. Your program should use the board configuration below as the goal state and lets the user input the initial/start board configuration.

CODE DOCUMENTATION

Flow of code

1. Uses #define to define the value of variable N to 4.
2. Initialize the necessary variables to measure the running time.
3. Define a structure for IDS and A* Algorithms.
4. Declare global variables such as the variables “goal”, “dy”, “dx”, “move”, “nMoves”, and “node”.
5. Declare all of the functions that are needed for the program. This includes for both IDS and A* functions.
6. Get user-input for the initial state of the puzzle, and then check the inputs and it should only consist of numbers from 0 - 15 (0 represents the empty space). After that, it will proceed to an if statement to check if there are duplicating numbers.
7. Then calls the function print_INPUT. It prints the initial state and the goal state side by side making it easier for the user to understand the puzzle.
8. The program proceeds to accept user input and employs the required functions to solve the 15-puzzle problem.
9. Every time the program computes the A* and IDS the clock starts to take note of the running time individually for the two.
10. The program then prints all the needed variables and the computed output of the program.

Functions

- print_INPUT

This function is designed to print the initial state and goal state of a puzzle side by side in a visually structured format. It uses loops to go through the rows and columns of the puzzle and prints each number in a table-like layout, separating the initial and goal states with a horizontal line. The function provides a clear and organized way to display the puzzle's starting and target configurations, making it easier for the user to understand the puzzle setup.

ITERATIVE DEEPENING SEARCH FUNCTIONS

- ids

This function solves the given states implementing the IDS algorithm. This runs by repeatedly calling a depth-limited search (dls) function with increasing depth limits until it finds a solution or exhausts the search space. The max_depth variable controls the depth limit, and the

search starts with a depth limit of 0 and increases by 1 in each iteration. When a solution is found, the loop breaks, and the search terminates.

- dls

This function explores the state space of the puzzle up to a certain depth limit, considering possible moves in different directions and building a path string as it goes. As this function explores every state, the “node” variable increments 1 value for every node added until it reaches the goal state thus giving it the number of nodes expanded during solving the 15-puzzle or it’s solution path, and increments the variable nMoves by 1 in every valid move direction made to solve the 15-puzzle. If it reaches the maximum depth and finds a goal state, it reports success; otherwise, it returns failure.

- do_move

This function is responsible for the change in state. as long as the move is valid and doesn't take the empty space out of bounds. It performs the move by swapping the empty space with the adjacent tile in the specified direction and updates the state accordingly.

- is_goal

Function checks whether a given puzzle state matches the predefined goal state by comparing each element of the state's board with the corresponding element in the goal board. If all elements match, the function returns 1, indicating that the provided state is the goal state. If any element doesn't match, it returns 0, indicating that it's not the goal state.

A* SEARCH FUNCTIONS

- astar_search

This function is implementing the A* search algorithm. It starts by creating open and closed lists and initializes a start node with the initial puzzle configuration. The algorithm then enters a loop where it selects the node with the lowest estimated cost from the open list, moves it to the closed list, and checks if it's the goal state. If the goal state is reached, it reconstructs and prints the path from the start to the goal. If no path to the goal is found, it eventually terminates, indicating that there is no valid solution.

- calculate_solution_cost

This function calculates the number of moves required to solve a puzzle from a given goal node to the initial state. It achieves this by tracing the path from the goal node back to the start node, incrementing a "cost" variable for each step in the path. Once the function reaches the start node, it returns the total "cost," indicating the minimum number of moves needed to solve the puzzle.

- reconstruct_path

This function creates a list of puzzle states from the goal state to the initial state to show the steps needed to solve the puzzle. It figures out how the empty space moves in each step

and prints the sequence of moves you'd need to make to solve the puzzle, like a set of directions from start to finish. This helps you understand the exact moves required to solve the puzzle.

- calculate_manhattan_distance

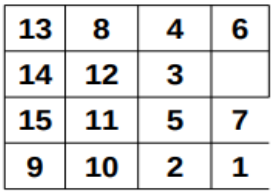
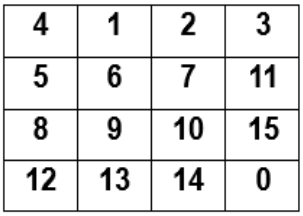
This function calculates the Manhattan distance heuristic for a puzzle, measuring how far each number is from its correct position by summing the horizontal and vertical differences between the current and desired locations. It adds up these distances for all numbers in the puzzle and returns the total Manhattan distance. This heuristic helps guide the search algorithm in making efficient moves when solving the puzzle by estimating the cost to reach the goal state.

- generate_successors

This function finds the empty space in a puzzle and generates potential next states by moving a nearest tile into that empty space, creating a move in up, down, left, or right directions. It checks whether these new states have been previously visited by comparing them to states in the closed and open lists. If a new state is unique, it calculates its cost to reach from the initial state, its estimated cost to the goal, and the total cost, aiding in determining the most promising moves. If the new state is not already explored, it is added to the open list for further examination, helping the search algorithm find a solution by exploring different puzzle configurations.

Analysis and comparison of the performance of the IDS and A* search algorithms

Initial State		IDS	A*
Easy 4 2 3 _ 5 1 6 7 8 9 10 11 12 13 14 15	Solution Path	left -> left -> down -> left -> up	left -> left -> down -> left -> up
	Solution Cost	5	5
	Number of nodes expanded	61	15
	Running Time	0.0020 seconds	0.0030 seconds
Medium 5 6 2 3 1 _ 10 7 4 13 9 15 8 12 11 14	Solution Path	up -> left -> down -> down -> down -> right -> up -> right -> down -> right -> up -> left -> up -> left -> up -> left	up -> left -> down -> down -> down -> right -> up -> right -> down -> right -> up -> left -> up -> left -> up -> left
	Solution Cost	16	16
	Number of nodes expanded	257701889	542
	Running Time	17.5110 seconds	0.0200 seconds
Hard 1 5 2 3 6 _ 7 11 4 12 14 10 9 8 13 15	Solution Path	left -> down -> right -> down -> left -> up -> right -> down -> right -> up -> right -> up -> left -> left -> up -> left	left -> down -> right -> down -> left -> up -> right -> down -> right -> up -> right -> up -> left -> left -> up -> left
	Solution Cost	16	16
	Number of nodes expanded	120891098	261
	Running Time	11.1090 seconds	0.0280 seconds

Worst  13 8 4 6 14 12 3 15 11 5 7 9 10 2 1	Solution Path	UNSOLVABLE	UNSOLVABLE
	Solution Cost	UNSOLVABLE	UNSOLVABLE
	Number of nodes expanded	UNSOLVABLE	UNSOLVABLE
	Running Time	UNSOLVABLE	UNSOLVABLE
Your preferred initial configuration  4 1 2 3 5 6 7 11 8 9 10 15 12 13 14 0	Solution Path	up -> up -> left -> left ->left -> up	up -> up -> left -> left ->left -> up
	Solution Cost	6	6
	Number of nodes expanded	762	19
	Running Time	0.0030 seconds	0.0020 seconds

PARTICIPATION/CONTRIBUTION OF EACH MEMBER

TASKS	MEMBER
Programmed the IDS Algorithm	Noli Joseph Tolosa Jhon Carl Angelo Garcia
Programmed the A* Search Algorithm	Ryan Kim Dionisio Aragorn O. Muncal
Debugger, Added Restrictions & Features, Compiled all the Codes	Joshua Vergara
Tests and Analyze the Performance of IDS and A* Search Algorithms	All Members
Documentation	All Members