Artificial Intelligence Project

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

The goal of this project is to build an artificial intelligent that can solve *n-puzzle problem* using informed and uninformed search methods.

1.2 Scope

This project is a software that can be used to create number-puzzle games and a solver for the problem using *Artificial Intelligence* by implementing multiple search method. In addition, the software is designed to help the user to identify the different benefits of each search method and their downfalls.

1.3 The Problem

Given a n×n board with n^2 tiles (every tile has one number from 1 to n^2-1) and one empty space. The objective is to place the numbers on tiles to match final configuration using the empty space. You can slide four adjacent tiles (left, right, down and up) into the empty space. Figure below shows the target when n=4. Furthuermore, each state will be represented as 2D array (array[n][n]), and the transtion between states will be through moving the empty square, which will have the value of 0.

1.4 Definitions and Acronyms

Heuristic function: a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow (estimation cost from the current state to the goal state), and we use *Manhattan distance* as a *Heuristic function* for this problem.
Manhattan distance: the distance between two points measured along axes at right angles.
GValue: the actual cost from the initial state to reach the current state.
FValue: *GValue + Heuristic function .*

2. SYSTEM OVERVIEW

2.1 Problem Formulation

1- States: The Location of tiles in the matrix.
2- Actions: moving the empty tile left, right, up, and down.
3- Goal test: the displayed numbers are in ascending order.
4- Path cost: every move cost 1.
5- Heuristic function: manhattan distance.

2.2 Dissecting the Problem

1- The program should generate number puzzles with different (cases) dimensions.
2- The Puzzle will be represented as 2D array.
3- The program must be able to represent the puzzle, and the outputs using GUI.
4- Every tile can only move to the blank space adjacent to its position by swapping positions with the blank tile.
5- Every puzzle has only one blank tile.
6- The puzzle should be checked if it's solvable or not.
7- The program must be able to allow the user to enter the initial state of the puzzle, in addition to being able to randomize the initial state.
8- The output of the program has the following displayed in the GUI:
   i- Total number of steps to reach solution (path cost); g(n) of the goal state.
   ii- Total number of processed nodes until the strategy finds solution.
   iii- Maximum number of nodes that have been stored concurrently;
   using a maximum variable to count the number of nodes in the frontier.

3. Software Design

3.1 Code Specification

- **Class *Node*:**
  Node class Is used as nodes for the Goal tree class, in addition to having the expansion function and any other cost functions.

  Attributes:
  state: the state of the n-puzzle in the current node.
  GValue(g): the depth of the current node (root depth is 0).
  parent: pointer to the parent of the node (root node doesn't have a parent).
  Action: the action that lead into creating the state within the node (Left, Right, Up, Down) (root node doesn't have an action).
  children: list of the node's children (expansions).

  Functions:
  __init__ (state, g, parent, action, children): it's the constructor of the *Node* class that initiate the attributes' values of the node by the passing parameters.
  equal(state): return true if the parameter's state is equals to current, false otherwise.
  mannhaten_distance (): calculate the distance between every square and its goal position measured along axes at right angles.
  get_h(): return the heuristic value from the current state to the goal state, using the Manhattan distance.
  get_f(): return the F value of the node, which is $f = GValue + HValue$.
  expand(): return a list of all possible states generated directly from the state within the current node.
  *move(*state, x1, y1, x2, y2*):* Move the blank space in the given direction (x2,y2) and return a new state after moving the blank. And it returns None if the positions value are out of limits. (Used by *expand* function)
  copy(state): return a similar created matrix of the given node. (Used by *move* function)
  find(state,x): return the position of (Specifically) the blank space. (Used by *expand* function).

- **Class *GoalTree*:**
  Goal tree class is the main class of the software, having the search methods and the closed list, in addition to other things.

  Attributes:
  root: the root node of the tree.

  Functions:
  __init__(initial_state): it's the constructor of the *GoalTree* class that initiate the attributes' values of the *GoalTree* by the passing parameters.
  Static - isGoal(state): goal test function, return true if the parameter is a matrix of increasing order (is the goal), false otherwise.
  Solve(strategy): the interactive method of the class takes a strategy (search method), then return a tuple of results that will solve the puzzle according to the strategy.

  results*:* a tuple that has (sol, g, processed_nodes, max_stored_nodes, flag, root.state):
  - sol.state: the solutions' state of the problem.
  - sol: a list of moves (e.g. right, left, ...) that if fallowed will solve the puzzle.
  - g: the cost of the solution (i.e. the number of moves to reach the solution).
  - processed_nodes: the number of nodes that has been tested using the goal function.
  - max_stored_nodes: the maximum number of nodes stored concurrently in the frontier.
  - flag: True if it has found a solution, False otherwise.

  bredthFirst(): return a tuple of results that will solve the puzzle according to the breadth first search strategy.
  UniformedCost(): return a tuple of results that will solve the puzzle according to the uniformed cost search strategy.
  depthFirst(): return a tuple of results that will solve the puzzle according to the depth first search strategy.
  depthLimited(l): return a tuple of results that will solve the puzzle according to the depth limited search strategy, with respect to *l*.

iterativeDeepening(): return a tuple of results that will solve the puzzle according to the iterative deepening search strategy.
greedy(): return a tuple of results that will solve the puzzle according to the best-first search strategy.
aStar(): return a tuple of results that will solve the puzzle according to the A* search strategy.
Static - solution(node): return a list of actions that lead to the solution.

- **Global Functions:**
  Solvable(state): return true if the given *state* is solvable, false otherwise.
  random_state(n): generate random states with *n* dimension through expansion to ensure solvability.
  number_of_inversion(state): return the number of inversions of a given *state*. (Used by *Solvable* function)
  row_of_blank_from_bottom(state): return the row of the blank from the bottom of a given *state*. (used by *Solvable* function)

- **Min-Heap Functions (Used as the frontier of the uniform, greedy and A* strategy searches):**
  *HeapPush(heap, item)***:** Push *item* onto the *heap*, maintaining the heap invariant.
  HeapPop(heap): Pop the smallest item off the *heap* and return it, maintaining the heap invariant.
  _siftdown(heap, startpos, pos): "*heap*" is a heap at all indices >= *startpos*, except possibly for *pos*. *pos* is the index of a leaf with a possibly out-of-order value. Restore the heap invariant. (Used by *HeapPush* function)
  _siftup(heap, pos): sifting up the *heap* to restore the heap invariant.

- **Usage of built-in libraries:**
  - *From collections* class we *import deque* function to help us in the breadth-first search.
  (Used in *bredthFirst* function)
  - *From random* class we *import ** that help us to make a random initial state.
  (Used in *random_state* function)

3.2 Code

The project has been implemented by *Python* language. The implementation code has been attached with the document.