Lab 01: Search Strategies

19125087 – Võ Khương Duy

# Check list

|  |  |
| --- | --- |
| **Implement UCS algorithm** | Done |
| **Implement tree-search DLS algorithm** | Done |
| **Implement IDS algorithm** | Done |
| **Implement graph-search GBFS** | Done |
| **Implement graph-search A\*** | Done |
| **Implement input reader** | Done |
| **Implement output printer** | Done |
| **Create additional sample mazes** | Done |

# Function Descriptions

* **model.py**

Define the models, or structures, that represents elements of the problem and solving algorithm.

In which:

* + *State, Cost, Heuristic, Priority: int*

Define, respectively, the identifier of a node in the maze, the path cost and heuristic value of a maze node, and the priority value when put a maze node into frontier.

* + *Node: class*

Define a search tree node, not to be confused with a maze node.

A Node includes:

* + - *state: State*

The identifier of a maze node of which the object represent.

* + - *parent: Node*

A reference to a class Node that defines the parent Node in the search tree.

* + - *cost: int*

Define the total cost to reach the search tree Node from the root.

* + *ProblemInput: List of str*

Define a list of strings that represent the input format.

* + *AdjMatrix: List of (List of State)*

Define a 2D list that represent the adjacency matrix.

* + *FrontierElem: (Priority, Node)*

Define an element within the frontier.

A tuple of 2 values, the first defines the priority of the element and the second refer to the search-tree Node of which the element represent.

* + *Frontier: List of FrontierElem*

Define the frontier.

* + *ExploredStates: List of State*

Define a list of explored states.

* + *Problem: class*

Define a representation of the problem

Includes:

* + - *size: int*

The size of the input maze.

* + - *adjacencyMatrix: AdjMatrix*

The adjacency matrix that defines the maze.

* + - *goalState: State*

Define the goal state of the problem

* + - *initState: State*

Define the initial state of the problem

Default value set to *0*

* + - *isGoalState(self, state): bool*

Take in a *State* object *state* and return a boolean value indicating whether it’s the goal state or not.

* + - *nextStatesFrom(self, state): List of State*

Take in a *State* object *state* and return a list of *State* object indicating it’s neighbours.

* **solver.py**

Defines the functions and classes used to solve the problem.

In which:

* + *readInputFromFile(fileDirectory): Problem*

Take in the directory to the input file, read the file as a list of string *lineList*, call *readInput(lineList)* and return a class Problem.

* + *readInput(input): Problem*

Take in the input string as list, the string should follow the format defined in the problem specification.

The function resolves this list and create a corresponding *Problem* object and return it.

* + *writeOutputToFile(fileDirectory, output): None*

Take in a directory and the output string.

Create the file if not existed and print the output string to that file.

* + *ManhattanHeuristic(problem, currentState): Heuristic*

Take in a *Problem* object and a *State* object, return a *Heuristic* object that define the heuristic value of the *currentState*.

* + *Solver: class*

A static class that defines the searching algorithms and their utility functions.

Includes:

* + - Searching algorithms:

Take in a *Problem* object and return a string, which can be a success or failed message.

* + - * *UCS(problem): string*

Define the Uniform Cost Search algorithm.

* + - * *IDS(problem): string*

Define the Iterative Deepening Depth-First Search algorithm.

* + - * *DLS(problem, depthLimit): string*

Define the Depth Limited Search algorithm.

* + - * *GBFS(problem): string*

Define the Greedy Best-First Search algorithm

* + - * *AStar(problem): string*

Define the A\* algorithm

* + - Utility functions:
      * *createNewPQElem(currentState, parentNode, priorityValue, addedCost): FrontierElem*

Take in a *State*, *Node*, *Priority*, and *Cost* object and return a corresponding *FrontierElem* object.

* + - * *successMessage(finalNode, exploredStates): string*

Take in a *Node* and an *ExploredStates* object and return a corresponding success message.

* + - * *failedMessage(exploredStates): string*

Take in an *ExploredStates* object and return a corresponding failed message.

* **main.py**

Used to read the input file and return the corresponding output to a new file.