Project 01: Ares's adventure

CSC14003 - Introduction to Artificial Intelligence

February 10, 2025

Contents

1 Overview

According to legend, hidden deep in a faraway kingdom is a gate that leads to a mysterious treasure. It's said that to open the gate, one must solve a maze by moving heavy stones onto secret switches. Though this may sound simple, many have tried and failed, as the maze is very tricky, and moving the large stones through tight spaces is tough. Even a small mistake could trap the challenger in the maze forever.

One day, a young adventurer named Ares decided to face this challenge. He knew that success would need careful planning, paying attention to every move, every detail, and never giving up. Your task is to help Ares on his adventure. Using the search algorithms you have learned in the Introduction to Artificial Intelligence course, you must guide him through the challenging maze, finding the path and positioning the stones on the switches to unlock the treasure gate.

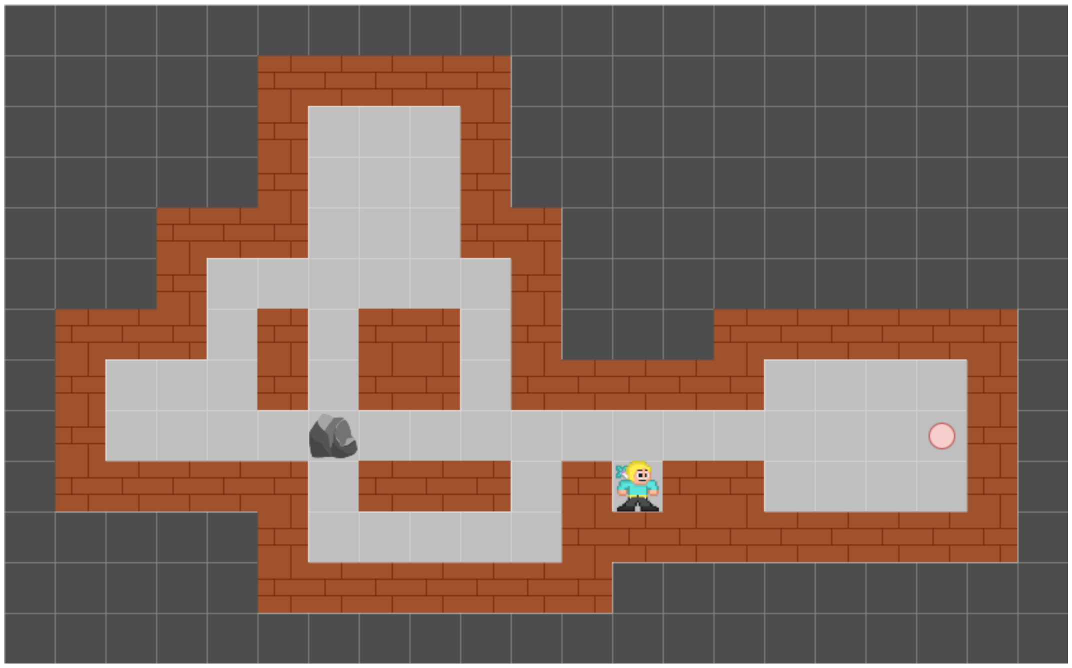


Figure 1: Example of a maze modeled in the 2D world.

Do you have the skills to help Ares conquer the maze and uncover the legendary treasure? Begin the journey and show your intelligence!

The maze is depicted as a grid of size $n \times m$, where each cell represents either a free space, a wall, a stone, or a switch. Ares (represented as an agent) can move one square at a time in 4 directions: **Up**, **Down**, **Left**, and **Right**. He cannot pass through walls or stones.

If there's a stone in the next space and the space beyond is free, he can push the stone there. He cannot pull stones, and stones cannot be pushed into walls or other stones.

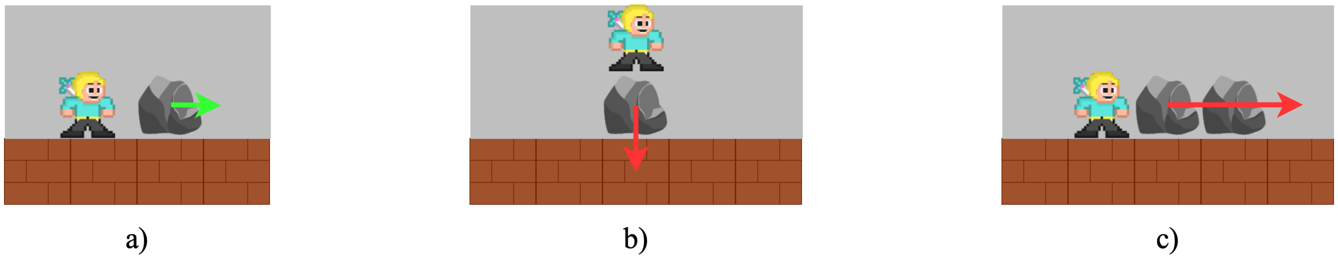


Figure 2: Ares pushing stones: (a) Valid move, (b) and (c) Invalid moves.

Each stone has a different weight, which means Ares must consider the most efficient way to push the stones onto switches. Pushing heavier stones takes more effort and limits his movement. This introduces the need for an optimal strategy when deciding which stones to push and where. The movement cost of Ares is calculated as follows:

- Each step Ares takes without pushing a stone incurs a base cost of 1.
- Pushing a stone increases the move's cost according to the stone's weight.

The objective is to push all stones onto the switches. Every switch can be activated by any stone of any weight, and the number of stones always equals the number of switches.

Students are required to implement the following search algorithms:

- **Breadth-First Search (BFS)**: a great algorithm; guarantees the shortest path.
- **Depth-First Search (DFS)**: a very bad algorithm for pathfinding; does not guarantee the shortest path.
- **Uniform Cost Search (UCS)**: based on the lowest path cost; guarantees the shortest path.
- **A* Search with heuristic**: arguably the best pathfinding algorithm; uses heuristics to guarantee the shortest path much faster than Dijkstra's Algorithm.
- **Greedy Best-first Search**: a faster, more heuristic-heavy version of A*; does not guarantee the shortest path.

Students are encouraged to study and implement the following search algorithms:

- **Dijkstra's Algorithm**: the father of pathfinding algorithms; guarantees the shortest path.
- **Swarm Algorithm**: a mixture of Dijkstra's Algorithm and A*; does not guarantee the shortest-path.

- **Convergent Swarm Algorithm**: the faster, more heuristic-heavy version of Swarm; does not guarantee the shortest path.
- **Bidirectional Swarm Algorithm**: Swarm from both sides; does not guarantee the shortest path.
- **Ant Colony Optimization**: is an algorithm inspired by the foraging behavior of ants described above. It's designed to solve combinatorial optimization problems, particularly those where we need to find the best possible solution among many.

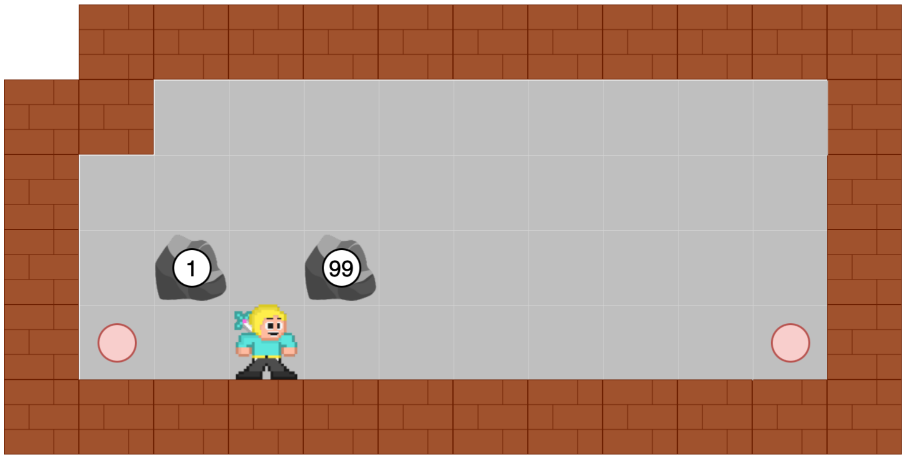
2 Requirements

2.1 Input

Students are required to design at least 10 different input files, named according to the structure `input-01.txt`, `input-02.txt`, ..., `input-10.txt`. Each input should have different parameters, such as the number of stones in the maze, the weight of each stone, or the structure of the maze. The format of the input file is as described below:

- The first line contains a list of integers representing the weights of each stone, in the order they appear in the grid, from left to right and top to bottom.
- The following lines describe the grid itself using the following characters:
  - "`#`" for walls.
  - "` `" (whitespace) for free spaces.
  - "`$`" for stones.
  - "`@`" for Ares.
  - "`.`" for switch places.
  - "`*`" for stones placed on switches.
  - "`+`" for Ares on a switch.

For example:



```
1 99
 ###########
##         #
#          #
# $ $      #
#. @      .#
############
```

Figure 3: Example of a maze layout (left) and its corresponding input file (right).

2.2 Output

The output must be both visualized using a Graphical User Interface (GUI) to display the process and saved as an output file. Below are the detailed output requirements:

- Output file:

  The output files should be named following the format `output-01.txt`, `output-02.txt`, etc. The content of each output file must include the following information:

  - The first line specifies the name of the algorithm used.
  - The second line provides detailed statistics, including:
    - The number of steps Ares must take,
    - The total weight he has to push,
    - The number of nodes generated by the algorithm,
    - The time taken to complete the search, and
    - The memory consumed during the process.
  - The third line presents the solution as a string representing Ares's sequence of actions. Lowercase letters `uldr` are used for movement, while uppercase letters `ULDR` represent pushing actions.

  For example:

```
BFS
```

```
Steps: 16, Weight: 695, Node: 4321, Time (ms): 58.12, Memory (MB): 12.56
uLulDrrRRRRRRurD
DFS
...
```

- Graphical User Interface (GUI):
  - The GUI must provide a visual representation all elements on the maze.
  - The GUI must animate **step by step** the process of Ares moving through the maze and pushing stones according to the solution sequence generated by the algorithm.
  - Display statistics, including the step count and the weight being pushed.
  - A button should be provided to start or pause the animation, as well as a reset option to restart the process. And a button to choose which algorithms are for visualization.

2.3 Programming language

The source code must be written in **Python**. You are allowed to use any supporting libraries; however, the main algorithms directly related to the search process must be implemented by you.

2.4 Report

- Member information (Student ID, full name, etc.)
- Work assignment table, which includes information on each task assigned to team members, along with the completion rate of each member compared to the assigned tasks.
- Self-evaluation of the project requirements.
- Detailed explanation of each algorithm (implementation process, heuristic function, etc.). Illustrative images and diagrams are encouraged.
- Description of the test cases and experiment results (memory usage, time complexity, etc.) Highlight challenges and compare the overall behavior of your algorithms.
- The report needs to be well-formatted and exported to PDF. If there are figures cut off by the page break, etc., points will be deducted.
- References (if any).

2.5 Demonstration videos

- Demo videos (recording the running process of your program for some test cases) should be uploaded to YouTube or Google Drive, and the public URLs are included in the report.
- In the video, students should start by compiling or running their code, then walk through the key steps of the program's execution to make it easy to follow.

2.6 Submission

- Your report, source code and test cases must be contributed in the form of a compressed file (.zip, .rar, .7z) and named according to the format **StudentID1_StudentID2_...**
- If the compressed file is larger than 25MB, prioritize compressing the report and source code. Test cases may be uploaded to Google Drive and shared via a link.

Details of the directory organization:

```
StudentID1_StudentID2_
├── Docs
│   ├── Report.pdf
│   ├── References_01.pdf
│   ├── References_02.pdf
│   └── ...
└── Source
    ├── Inputs
    │   ├── input-01.txt
    │   ├── input-02.txt
    │   └── ...
    ├── Outputs
    ├── helper_01.py
    ├── helper_02.py
    ├── main.py
    ├── requirements.txt (libraries to install)
    ├── README.txt (how to run source code)
    └── ...
```

3 Assessment

No	Criteria	Scores
1	Implement BFS correctly.	10%
2	Implement DFS correctly.	10%
3	Implement UCS correctly.	10%
4	Implement A* correctly.	10%
5	Generate at least 10 test cases for each level with different attributes.	10%
6	Result (output file and GUI).	15%
7	Videos to demonstrate all algorithms for some test cases.	10%
8	Report your algorithm and experiment with some reflection or comments.	25%
9	Implement, written report for Swarm Algorithm, Convergent Swarm Algorithm, or Bidirectional Swarm Algorithm.	20%
	Total	**120%**

4 Notices

Please pay attention to the following notices:

1. This is a GROUP assignment. Each group has 4 members.
2. Duration: about 3 weeks.
3. Any plagiarism, any tricks, or any lie will have a 0 point for the course grade.
4. AI-Generated Content must lower than 30% in the report.