CS300 Artificial Intelligent
Lab Midterm Project

> **Instructors:** *Dr. Nguyen Ngoc Thao, Msc. Do Trong Le - Nguyen Quang Thuc*
>
> **Students:** *Dao Minh Duc - 2159003, Nguyen Bao Ngoc - 2159009*

Project Description

This project is a part of the course CS300 - Artificial Intelligent at ITEC, HCMUS. The project is about implementing a simple AI to play the game "Sokoban", programmed in Python, using the Pygame library for the GUI. The game is a classic puzzle game where the player has to push boxes to the target locations. The game is considered solved when all boxes are on the target locations.

The rules of the game are as follows:

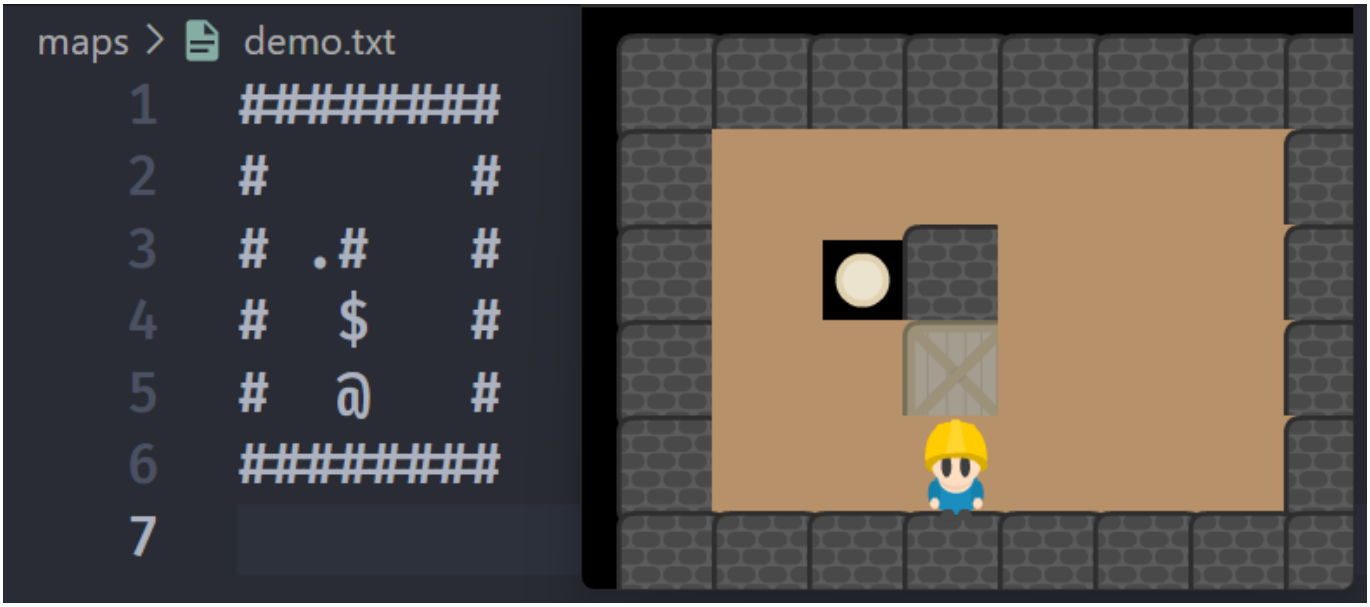
- The player can move in four directions: up, down, left, right.
- The player can push a box if there is no obstacle behind the box.
- The player can only push one box at a time.
- The player cannot move through walls or boxes.
- The game is considered solved when all boxes are on the target locations.
- The cost of every move is equal.

Game State Representation

Each step of the solution will be represented as a state of the game. Each state is represented as a 2D array, where each cell contains the information of the object in that cell:

- Empty: ` ` (space)
- Wall: `#`
- Box: `$`
- Target: `.`
- Player: `@`
- Box on target: `*`
- Player on target: `+`

Example:



The neighbor states of a state are the states that can be reached from the current state by moving the player in one of the four directions, or by pushing a box in one of the four directions. States will be deduplicated and stored in a queue, stack, or priority queue, depending on the algorithm used.

Algorithms

The program will read the input from a file, which contains the initial state of the game. It will then use a specific search algorithm specified by user to solve the game. The solution will be displayed on the GUI, and the additional information will be printed to the console. The following algorithms will be implemented:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- A* Search
- Uniform Cost Search (UCS)
- Greedy Best-First Search
- and a custom algorithm

Heuristics

For algorithms that use heuristics (A*, UCS, GBFS), the sum of Manhattan distances of each box on the board to its nearest target will be used.

Hamming distance will not be suitable for this problem, since until a state with at least one box on a target is reached, the Hamming distance will be equal to $N \times 2$, with N being the number of boxes on the board.

Benchmark

System Specification

Property	Value
OS	Windows 11 Pro 23H2 (22631.3007)
Processor	Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
Installed RAM	32.0 GB (31.8 GB usable, 30.2 GB free)
System type	64-bit operating system, x64-based processor

Results

Each algorithm will be ran 3 times. The result will be the average time (rounded to 3 decimal places), number of expanded nodes, and path length of the 3 tests.

`maps/demo_1.txt` (8x6):

Algorithm	Time (s)	Expanded Nodes	Path Length
BFS	0.240	344	6
DFS	0.238	344	6
UCS	0.296	404	6
A*	0.078	140	6
GBFS	0.015	36	6

`maps/demo_2.txt` (10x8):

Algorithm	Time (s)	Expanded Nodes	Path Length
BFS	31.618	9652	26
DFS	31.241	9652	26
UCS	29.712	9288	26
A*	9.975	3964	26
GBFS	2.271	904	32

Usage

Arguments:

- `--map` : The path to the file containing the initial state of the game. The default value is `maps/demo_1.txt` .
- `--strategy` : The search algorithm used to solve the game. The default value is `bfs` . The possible values are `bfs` , `dfs` , `ucs` , `astar` , `greedy` , and `custom` .

```
python main.py --map maps/custom_map.txt --strategy astar
```

Notes

- The program requires Python 3, and the Pygame library to be installed (`pip install pygame`).
- The program will not check if the initial state is solvable, and it will not prevent the player from creating unsolvable states during the game. Therefore, the program will only stop when the game is solved, or when the algorithm has reached the maximum number of iterations.

Project Structure

- `assets` : The folder containing the assets used in the GUI.
- `main.py` : The main file of the program.
- `maps` : The folder containing the initial state files of the game.
- `modules` : The folder containing the main modules of the program.
  - `game_state.py` : The module containing the `GameState` class, which represents the state of the game.
  - `game_visualization.py` : The module containing the `GameVisualization` class, which is responsible for the GUI.
  - `solver.py` : The module containing the `Solver` class, which is responsible for solving the game using different algorithms.