



CS420 - Artificial Intelligence

Project 1 - Search

Report

Student Information

Student 1

- Name: Vòng Vĩnh Toàn
- Student ID: 22125108

Student 2

- Name: Nguyễn Hoàng Phúc
- StudentID: 22125077

Student 3

- Name: Huỳnh Hà Phương Linh
- Student ID: 22125049

Student 4

- Name: Huỳnh Đăng Khoa
- Student ID: 22125038

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 diagram are encouraged.
- Description of the test cases and experiment results (memory usage, time complexity, etc.)
- Highlight challenges and compare 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).

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 the Google Drive and shared via a link.

Table of Content

Student Information.....	2
Introduction.....	6
Contribution table.....	6
Method.....	6
Project structure.....	6
Main pipeline.....	6
Problem class.....	9
Environment class.....	12
Search state class.....	13
Search node class.....	15
Search strategy pattern.....	16
Benchmarking.....	18
Command line argument.....	18
Visualization.....	20
Test cases.....	21
Test case analysis.....	21
Summary:.....	21
Search algorithms.....	25
Best First Search algorithm.....	25
Implementation.....	25
Complexity analysis.....	26
BFS (Breadth First Search).....	27
Implementation.....	27
Complexity analysis.....	28
DFS (Depth First Search).....	29
Implementation.....	29

Complexity analysis.....	30
UCS (Unified Cost Search).....	30
Implementation.....	30
Complexity analysis.....	31
A* algorithm.....	31
Heuristic function(s).....	31
Heuristic 1: Minimum Cost Matching with Manhattan Distance.....	31
Minimum Cost Matching Calculation:.....	32
Heuristic 2: Precomputed Shortest Path Distances.....	33
Implementation.....	34
Complexity analysis.....	35
1. Precomputation for Heuristic 2:.....	35
2. Per-Expansion Complexity (Minimum Cost Matching):.....	35
3. Search Phase (A* with Heuristic 2):.....	36
Results.....	36
Test cases results.....	36
Results analysis.....	36
Youtube link.....	40
GitHub link.....	40
Self-evaluation.....	40
Conclusion.....	42

Introduction

This report presents our work on Project Assignment 01 - Search, where our team was assigned to implement search algorithms to solve a variation of the classic puzzle game *Sokoban*. The project requirements included programming the solution in Python, developing a graphical user interface (GUI) for visual representation, and producing a demo video to showcase our final results.

To make our code more modular and manageable, we applied object-oriented programming (OOP) principles and utilized design patterns like the Strategy Method. This approach helped streamline our implementation and maintain cleaner code. Additionally, we incorporated a monitoring component within a terminal user interface (TUI) to track and display the progress of each search algorithm, giving us valuable insights into their performance. Lastly, we designed an intuitive and visually appealing GUI to enhance the overall user experience, allowing players to observe the algorithms' operations in real-time.

Contribution table

Method

Project structure

Main pipeline

For each test case, which is represented by a tuple consisting of:

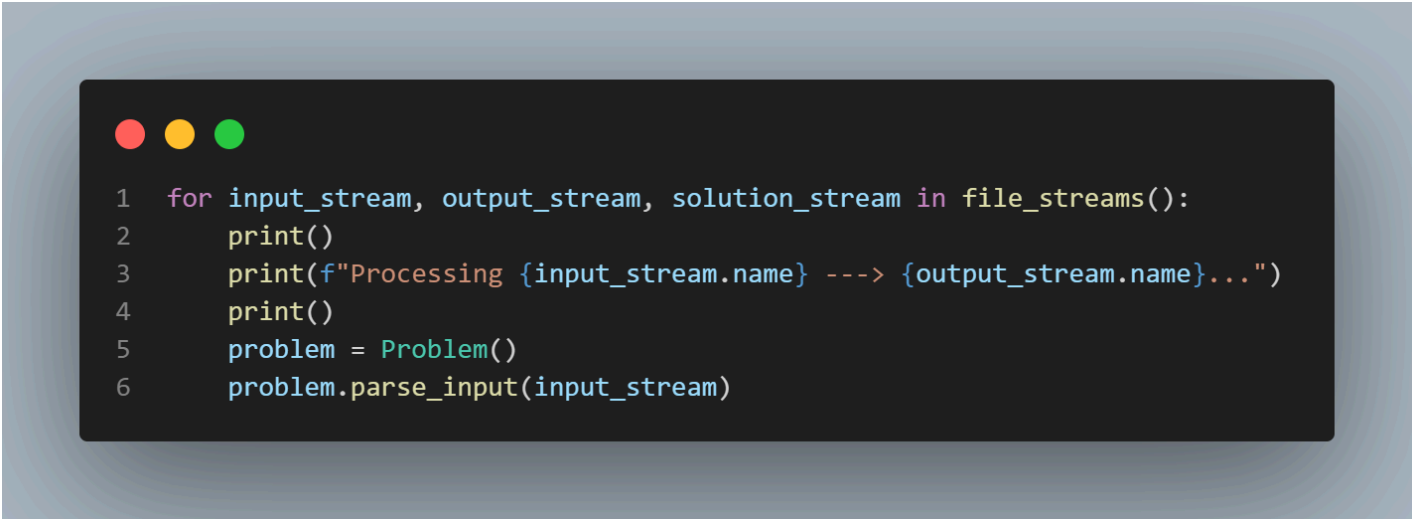
- `input_stream` (linked to the input file),

- `output_stream` (linked to the output file),
- `solution_stream` (used for creating a JSON representation of the result to simplify parsing for the visualizer),

we execute the following steps:

1. Problem Initialization

We iterate through each test case, creating a `problem` object. This object holds the map structure, the initial state, and a collection of helper functions to facilitate search operations.



```
1  for input_stream, output_stream, solution_stream in file_streams():
2      print()
3      print(f"Processing {input_stream.name} ---> {output_stream.name}...")
4      print()
5      problem = Problem()
6      problem.parse_input(input_stream)
```

2. Algorithm Execution

For each problem, we iterate over a list of `solver` objects. These solvers are selected from a predefined dictionary of search algorithms and represent the various search strategies applied to the problem.



```
1 solver_dict : dict[str, SearchStrategy] = {  
2     'DFS': DFS_solver,  
3     'BFS': BFS_solver,  
4     'UCS': UCS_solver,  
5     'A_star': A_star_solver  
6 }
```

3. Solving Process

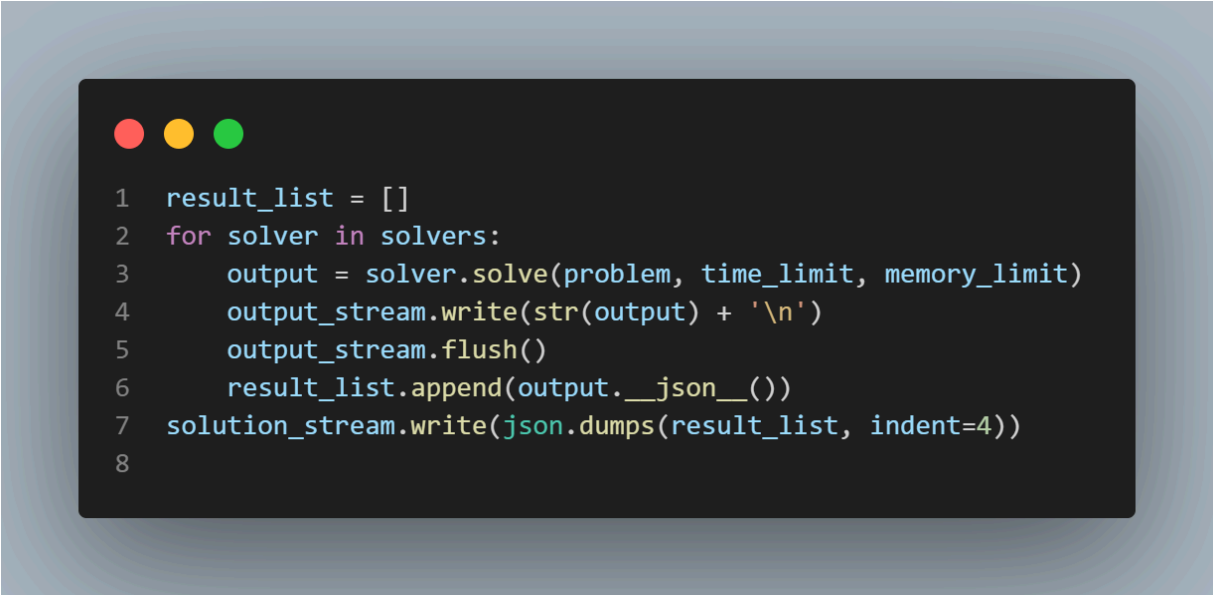
Each solver attempts to solve the problem within a specified time and memory limit. If any error occurs, such as *Time Limit Exceeded* or *Memory Limit Exceeded*, the solver handles it gracefully, outputting a clear error message to ensure comprehensibility.



```
1 for solver in solvers:  
2     output = solver.solve(problem, time_limit, memory_limit)
```


4. Output Handling

The search results, whether successful or error-reported, are printed to the output file via the `output_stream`, with flushing enabled to ensure real-time data output. This output includes both the solution details and any error specifications, providing a structured and readable format for each test case.



```
1 result_list = []
2 for solver in solvers:
3     output = solver.solve(problem, time_limit, memory_limit)
4     output_stream.write(str(output) + '\n')
5     output_stream.flush()
6     result_list.append(output.__json__())
7 solution_stream.write(json.dumps(result_list, indent=4))
8
```

This structured approach ensures efficient processing of each test case, minimizes errors, and enhances the readability of outputs, making it easier for the visualizer to parse and display results.

Problem class

The `Problem` class is designed to store the structure of the game map, initial state, and a set of helper functions. It provides a framework to interact with the environment and the search algorithms efficiently.



```
1 class Problem:
2     def __init__(self):
3         self.initial_state = SearchState()
4         self.environment = Environment()
5
6     def parse_input(self, input_stream: TextIO) -> None:
7         pass
8
9     def is_goal(self, state: SearchState) -> bool:
10        pass
11
12    def is_deadend(self, state: SearchState) -> bool:
13        pass
14
15    def actions(self, state: SearchState) -> Generator[Action, None, None]:
16        pass
17
18    def result(self, state: SearchState, action: Action) -> tuple[SearchState, int]:
19        pass
20
21    def expand(self, node: SearchNode) -> Generator[SearchNode, None, None]:
22        pass
```

One key observation is that some elements of the problem (like wall and switch locations) are immutable. Rather than including these in each search state, we store them as part of an `Environment` object within the `Problem` class, saving memory and computational resources. We'll discuss the `Environment` class in a later section.

The `initial_state` object represents the starting configuration of the problem, while helper functions define core interactions and are described as follows:

<code>parse_input(self, input_stream: TextIO)</code> -> <code>None</code>	This method reads from the input stream and initializes relevant variables, setting up the environment and the initial state.
<code>is_goal(self, state: SearchState)</code> -> <code>bool</code>	Checks if the given state meets the goal conditions of the problem
<code>is_deadend(self, state: SearchState)</code> -> <code>bool</code>	Determines if the given state is a dead-end. Here, a dead-end state is defined as one where there is a rock that is not on a switch and cannot be moved in the future.
<code>actions(self, state: SearchState)</code> -> <code>Generator[Action, None, None]</code>	This function yields valid actions for the provided state. If the state is a dead-end, it yields no actions, as there are none available.
<code>result(self, state: SearchState, action: Action)</code> -> <code>tuple[SearchState, int]</code>	Returns a tuple containing the resulting state after performing the specified action, along with the cost of that action. We opted for this format over pseudo-code conventions for simplicity and natural expression in Python.
<code>expand(self, node: SearchNode)</code> -> <code>Generator[SearchNode, None, None]</code>	Yields each successor node of the current node, effectively producing possible future states

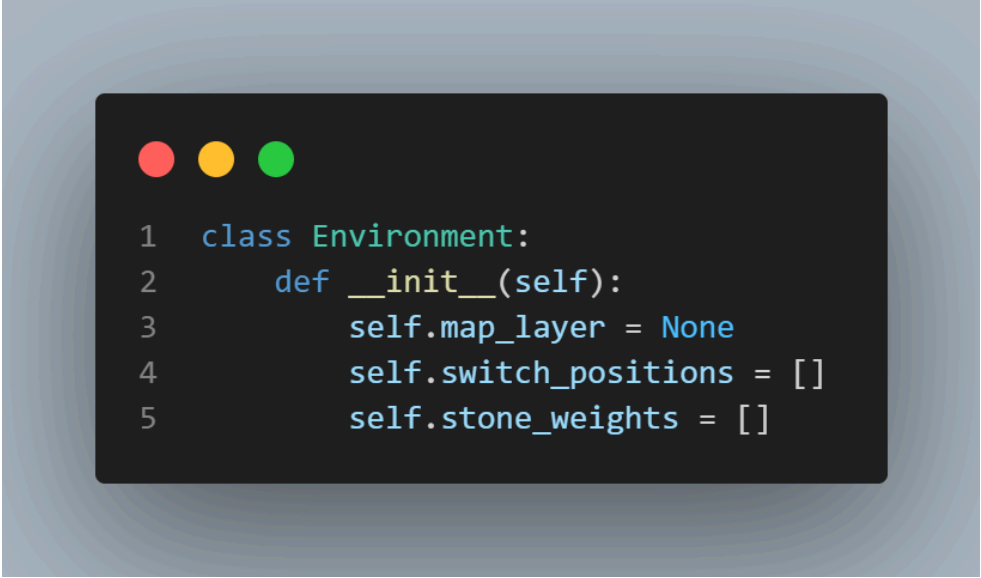
This setup enables the `Problem` class to provide the search algorithms with efficient access to map information, state transitions, and action results, all while ensuring that immutable components are handled separately. This approach enhances modularity, memory efficiency, and clarity.

Environment class

After careful analysis, we identified the immutable elements in the problem as:

- **Wall positions**
- **Switch positions**
- **Stone weights** (if the stones are arranged in a fixed-index array)

These elements remain constant throughout the game, so instead of repeatedly including them in each search state, we store them within an `Environment` object. Here's the class structure:



```
1 class Environment:
2     def __init__(self):
3         self.map_layer = None
4         self.switch_positions = []
5         self.stone_weights = []
```

Class Attributes:

- **map_layer**: This attribute acts as a lookup grid that helps determine the type of each cell (whether it's an empty cell, wall, or switch). The map layer allows for efficient checking of cell properties.

- **switch_positions**: Although this information could technically be derived by scanning the **map_layer**, storing the switch positions separately helps reduce computation. Accessing switch locations directly from **switch_positions** is much more efficient than iterating over the **map_layer** grid every time the information is needed, especially since the **Environment** object is shared across the entire program without duplication.
- **stone_weights**: By assigning stones to a fixed-indexed array, the weights of each stone remain consistent throughout the game, making them immutable. This attribute tracks those weights, ensuring they can be accessed efficiently without recalculating or re-fetching data.

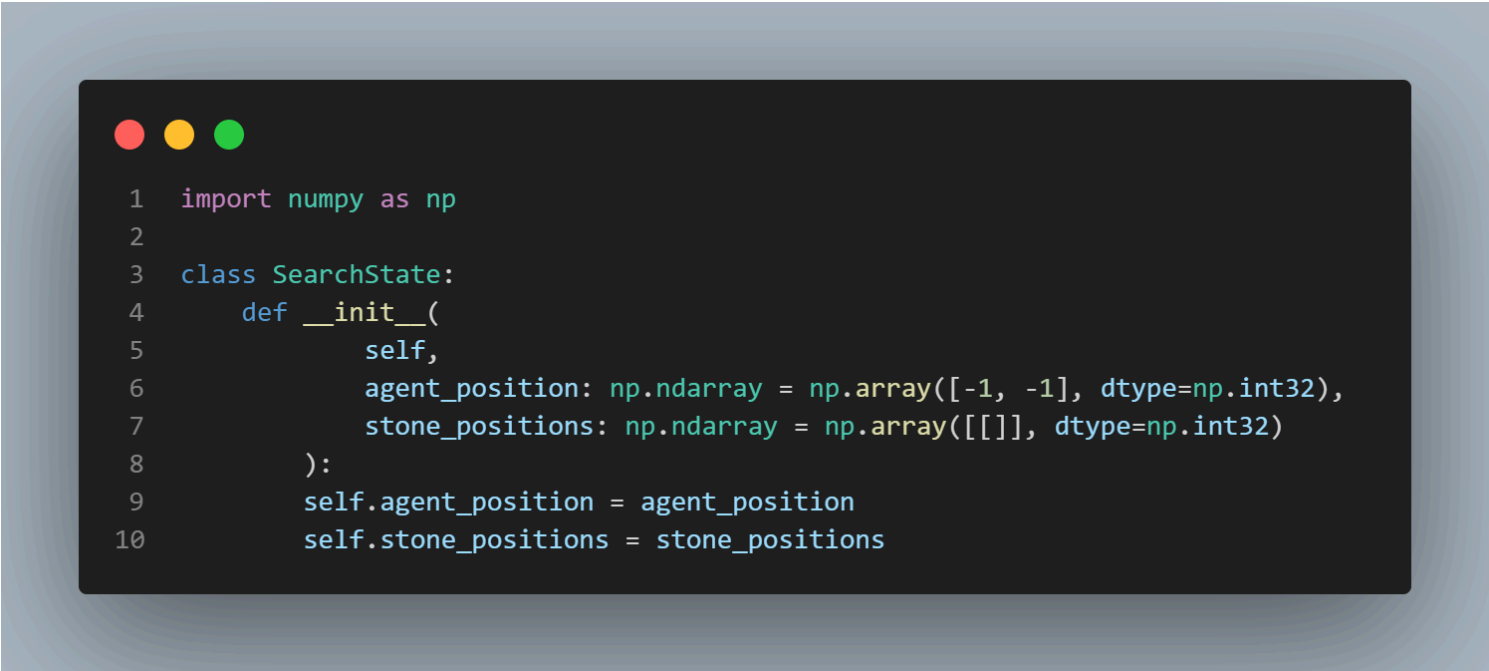
This design enhances efficiency by centralizing immutable data in the **Environment** class, thus avoiding redundant storage and unnecessary computation. The combination of **map_layer** and **switch_positions** provides quick access to map data while keeping the codebase optimized and readable.

Search state class

The **SearchState** class represents a single state in the search process. Each search state consists of two main components:

1. The **agent's position** (player's current location),
2. An **ordered list of stone positions** (each stone's coordinates, maintained in a specific order).

Here's the class definition:




```
1 import numpy as np
2
3 class SearchState:
4     def __init__(
5         self,
6         agent_position: np.ndarray = np.array([-1, -1], dtype=np.int32),
7         stone_positions: np.ndarray = np.array([], dtype=np.int32)
8     ):
9         self.agent_position = agent_position
10        self.stone_positions = stone_positions
```

Class Attributes:

- **agent_position**: This attribute holds the coordinates of the agent's current position on the grid as a NumPy 2-element array. Initializing it to `[-1, -1]` indicates a default value, which can be updated once the agent's actual start position is known.
- **stone_positions**: This ordered NumPy array stores the positions of each stone, with each entry representing a stone's coordinates on the map. Since the list is ordered and indexed, this structure enables quick and efficient reference to individual stones during search operations.

Search node class

The `SearchNode` class represents a node in the search tree. Each `SearchNode` encapsulates a specific search state, along with information needed to trace the path taken to reach that state. Here's the class definition:



```
1 class SearchNode:
2     def __init__(
3         self,
4         state: SearchState,
5         parent: 'SearchNode' = None,
6         action: Action = None,
7         path_cost: int = 0
8     ):
9         self.state = state
10        self.parent = parent
11        self.action = action
12        self.path_cost = path_cost
```

Class Attributes:

- **state**: This attribute holds the current `SearchState` object, representing the search state at this node.

- **parent**: A reference to the previous `SearchNode` in the path, enabling the algorithm to trace back the sequence of moves from the initial state to the current state. If this is the root node, `parent` is set to `None`.
- **action**: The action is taken to move from the parent node to the current node. This attribute is essential for reconstructing the solution path once the goal state is found.
- **path_cost**: The cumulative cost from the initial node to the current node. This attribute enables cost-based search strategies, like Uniform Cost Search, and helps keep track of the total "expense" of reaching a node.

Search strategy pattern

The `SearchStrategy` class employs the Strategy design pattern, allowing for flexible and interchangeable search algorithms. It defines two primary methods: `search` and `solve`. The `search` method represents the core search function, while `solve` wraps around `search`, managing additional functionalities like monitoring resource limits and collecting performance statistics.


```
1 class SearchStrategy:
2     def search(self, problem: Problem) -> SearchNode|None:
3         pass
4
5     def solve(
6         self,
7         problem: Problem,
8         time_limit: float = -1,
9         memory_limit: float = -1
10    ) -> ProblemResult:
11        result = ProblemResult()
12        search_result = self.search(problem)
13
14        result.set_result(search_result, problem, ...)
15
16        result.strategy_name = str(self)
17        result.time = ...
18        result.memory = ...
19        result.numNodeGenerated = ...
20
21        return result
```

Class Methods:

- **search(self, problem: Problem) -> SearchNode | None**

This is an abstract method meant to be implemented by subclasses. Each specific search algorithm (e.g., BFS,

DFS, A*) will provide its own version of `search`, which takes in a `Problem` and returns a `SearchNode` representing the goal node if a solution is found, or `None` if no solution exists.

- `solve(self, problem: Problem, time_limit: float = -1, memory_limit: float = -1) -> ProblemResult`

The `solve` method is a wrapper around `search`. It initializes a `ProblemResult` object and calls the `search` method. Once `search` completes, it collects statistics, including:

- `strategy_name`: The name of the strategy being used.
 - `time`: The elapsed time taken to complete the search, to be defined in the `Benchmarking` section.
 - `memory`: The memory used during the search process.
 - `numNodeGenerated`: The total number of nodes generated.
 - `verdict`: The runtime verdict such as Time Limit Exceeded, Memory Limit Exceeded, Interrupted By User, etc.
- The collected statistics are set in `result`, which will store all information about the search attempt.

Benchmarking

The benchmarking system leverages multithreading to simultaneously run the search algorithm and monitor system resources, allowing for real-time tracking of performance metrics such as time, memory usage, and node generation. By creating a node count variable within the `SearchNode` class, we also track the number of nodes generated, which is essential for analyzing the efficiency of different search algorithms.

Command line argument

The command-line interface for this application is implemented using Python's `argparse` module, enabling users to specify various settings for running the search algorithms. The `parse_args` function defines multiple options, allowing flexibility in configuring input files, output files, strategy choices, and resource limits.

1. **--strat or -s:**
Specifies a list of strategies to apply for solving the problem. Each strategy corresponds to a search algorithm, allowing users to select multiple strategies for comparison.
2. **--in or -i:**
Accepts a list of input files, each representing a test case for the search algorithms.
3. **--out or -o:**
Specifies a list of output files, where each file corresponds to the result of a particular test case.
4. **--sol or -so:**
Defines a list of solution files, which may store additional information such as detailed search paths or intermediate results to assist in visualizing and validating solutions.
5. **--inregex (-ir), --outregex (-or), --solregex (-sor):**
Allows users to provide regular expressions to match input, output, and solution files automatically. This option is especially useful for batch-processing multiple files.
6. **--time_limit (-TL) and --memory_limit (-ML):**
Sets time and memory limits for each strategy, enabling users to enforce constraints on resource usage.
7. **--file or -f:**
Specifies a configuration file from which arguments are read, providing an alternative to command-line input. This option is useful for storing and reusing configurations.

Example usage to run the code:

```
python main.py -f .\instructions.txt
```

or

```
python main.py -s .* -ir "input-(0[1-9]|10)\.txt" -or "output-(0[1-9]|10)\.txt" -sor  
"solution-(0[1-9]|10)\.json" -TL 100 -ML 100
```

Explanation of Each Argument

1. **-s .*:**

- The `-s` flag specifies the search strategies to use. Here, `.*` matches any strategy name, indicating that all available strategies should be applied.
- 2. `-ir input-(0[1-9]|10)\.txt:`
 - The `-ir` flag (input regex) uses a regular expression to match files with names like `input-01.txt` through `input-10.txt`.
 - `(0[1-9]|10)` specifies that files numbered from `01` to `10` should be included.
- 3. `-or output-(0[1-9]|10)\.txt:`
 - The `-or` flag (output regex) uses a similar pattern to specify the output files.
 - It will match files named `output-01.txt` through `output-10.txt`.
- 4. `-sor solution-(0[1-9]|10)\.json:`
 - The `-sor` flag (solution regex) identifies solution files with names like `solution-01.json` to `solution-10.json`.
 - This is useful if solutions need to be stored in JSON format for further parsing or visualization.
- 5. `-TL 100:`
 - Set the **time limit** for each strategy to 100 seconds. This means each search algorithm will be allowed a maximum of 100 seconds to find a solution for each input case.
 -
- 6. `-ML 100:`
 - Sets the **memory limit** for each strategy to 100 MB. This means each search algorithm will be allowed a maximum of 100 MB to find a solution for each input case.

Visualization

The visualization tool uses Pygame to simulate the solution of each algorithm.

Command line argument

1. `--map:`
 - Path of an input file, which represents a test case for the search algorithms.

2. `--sol`:

- Path of the solution file (JSON) corresponding to the input file, which stores additional information such as detailed search paths or intermediate results.

Example usage to run the code:

```
python main.py --map input-01.txt --sol solution-01.json
```

Experiment setup

Test cases

Test case analysis

Summary:

The test cases can be divided into three levels of difficulty:

- **Easy Cases:** input-01, input-02 (Clear paths, minimal obstacles).
- **Moderate Cases:** input-03, input-04, input-05 (Requires some planning and maneuvering).
- **Hard Cases:** input-06, input-07, input-08, input-09, input-10 (Complex pathfinding, multiple rocks, or inaccessible solutions due to design).

Detailed analysis of each test case:

The four search strategies to be analyzed are Depth-First Search (DFS), Breadth-First Search (BFS), Uniform Cost Search (UCS), and A* Search.

Each test case has **n** rows, **m** columns, and **k** rocks.

Test case	n	m	k	Description	Suitable strategies & test objective
input-01	4	11	1	Easy: <ul style="list-style-type: none"> • The maze features a spacious central area enclosed by walls. • The agent starts at the center of the maze. • No obstacles limit the agent's maneuvering. • The agent only needs to push the rock to the right to reach the switch. 	<ul style="list-style-type: none"> • All algorithms are suitable. • To test if the algorithms can handle a very basic maze with no complexities.
input-02	6	12	2	Easy: <ul style="list-style-type: none"> • The maze has moderate size with no obstacles blocking the agents. • The rock is positioned directly next to the agent. • The switch is located just a few steps away from the agent's starting position. • The path from the rock to the switch is clear and straightforward. 	<ul style="list-style-type: none"> • All algorithms are suitable. • Test how the algorithms perform in a simple maze without any obstacles.
input-03	8	12	2	Moderate: <ul style="list-style-type: none"> • Narrow pathways create a challenging environment for the agent. • Two switches are placed in different locations: <ul style="list-style-type: none"> ○ One near the top-right corner. ○ Another in the middle of the maze. 	<ul style="list-style-type: none"> • A* is recommended, BFS/UCS might work, but DFS fails. • Assess how algorithms navigate tight spaces and find optimal solutions when multiple goals need to be reached.

				<ul style="list-style-type: none"> The path is constrained by walls, allowing only a single way to push the rocks onto the switches. 	
input-04	8	18	4	<p>Moderate:</p> <ul style="list-style-type: none"> A vast rectangular map with scattered obstacles. Two switches are placed as a pair in the top-left region. The remaining switches are scattered in the middle region of the map. 	<ul style="list-style-type: none"> A* is recommended, BFS/UCS might work, but DFS fails. To evaluate how algorithms perform when dealing with larger mazes and a greater number of obstacles.
input-05	7	9	4	<p>Moderate:</p> <ul style="list-style-type: none"> The maze has a narrow base and gradually widens upward. The agent starts near the center of the maze, positioned on a switch and surrounded by rocks. One rock is pre-placed on an incorrect switch. Initial maneuvering is required for two rocks. The remaining steps after initial maneuvering are straightforward. 	<ul style="list-style-type: none"> A* is recommended, BFS/UCS might work, but DFS fails. assess the ability of algorithms to handle scenarios where objects need to be repositioned before reaching the final goal.
input-06	2	12	3	<p>Hard:</p> <ul style="list-style-type: none"> The maze is tight and complex, with switches positioned in various directions on the map. Rocks are placed near switches, with no obstacles blocking the paths to the goals. 	<ul style="list-style-type: none"> A* is recommended, BFS/UCS might work, but DFS fails. To test if algorithms can find solutions in highly constrained

				<ul style="list-style-type: none"> Two of the switches are surrounded by walls. 	and complex mazes with limited movement options.
input-07	7	20	5	Hard: <ul style="list-style-type: none"> The maze is large, featuring four switches in the corners and one in the center. Obstacles are mainly concentrated in the center, restricting the agent's movement. Switches and rocks are surrounded by these central obstacles. Two rocks positioned next to each other require the agent to choose an appropriate solution. 	<ul style="list-style-type: none"> A* is recommended, BFS/UCS might work, but DFS fails. To see how algorithms handle a bottleneck scenario where movement is restricted by a dense obstacle area.
input-08	7	12	3	Hard: <ul style="list-style-type: none"> The map's layout is complex and winding, with a fluctuating shape that is narrow at the top and bottom. Three switches are placed in a horizontal direction, with two adjacent to each other. Rocks are separated by walls, limiting their movement. 	<ul style="list-style-type: none"> A* is recommended, BFS/UCS might work, but DFS fails. To evaluate the ability of algorithms to find efficient paths through mazes with winding layouts and obstacles.
input-09	7	9	6	Hard: <ul style="list-style-type: none"> The maze is narrow and contains six rocks. Four rocks are incorrectly positioned on switches. Two rocks completely block the path. 	<ul style="list-style-type: none"> A* is recommended, BFS/UCS might work, but DFS fails. To examine how algorithms solve complex puzzles that require strategic object

				<ul style="list-style-type: none"> • Rocks cannot be pushed all the way, requiring careful maneuvering. • Strategic placement is required to create space and access other rocks. 	movement to create space and clear paths.
input-10	8	17	1	<p>Hard:</p> <ul style="list-style-type: none"> • No solution exists for this case. • The maze is large, with obstacles mainly positioned in the upper half of the map. • There is only one stone and one switch, separated by a wall. • The walls are designed to create a loop path, preventing the agent from reaching the solution. 	<ul style="list-style-type: none"> • None of the algorithms are suitable. • To prove algorithms can correctly identify cases where a solution is impossible.

Search algorithms

Best First Search algorithm

Implementation

The Best-First Search algorithm here is implemented using a priority queue to explore nodes based on a priority function, f , that estimates the cost or desirability of reaching the goal from the given node. We aligned the implementation to look as close as the pseudo code in the lecture slides.

```

1  def search(problem: Problem, f: Callable[[SearchNode], int]) -> SearchNode|None:
2      node = SearchNode(problem.initial_state)
3
4      frontier : PriorityQueue[SearchNode] = PriorityQueue(priority=f); frontier.put(node)
5      reached : dict[SearchState, SearchNode] = {node.state: node}
6
7      while not frontier.empty():
8          node = frontier.get()
9
10         if problem.is_goal(node.state):
11             return node
12
13         if f(reached[node.state]) < f(node):
14             continue
15
16         for child in problem.expand(node):
17             s = child.state
18             if s not in reached.keys() or f(child) < f(reached[s]):
19                 reached[s] = child
20                 frontier.put(child)
21
22     return None

```

Key Components

1. Priority Queue (**frontier**):

- **PriorityQueue** is used to manage the nodes based on the **f** function, which allows the algorithm to explore nodes with the lowest estimated cost or highest desirability first.

2. Reached Dictionary (**reached**):

- Tracks each **SearchState** and its associated **SearchNode** with the best-known cost. If a state is encountered with a lower cost than previously recorded, it replaces the prior entry to ensure that only optimal paths are explored.

3. Loop Execution:

- The algorithm runs in a loop until the priority queue (**frontier**) is empty, retrieving the lowest-priority node at each iteration.
- Each expanded node is checked to determine if it reaches the goal state.
- Nodes are added to the frontier only if they represent a more efficient path to a new or already reached state, which minimizes redundant processing of suboptimal paths.

Complexity analysis


The complexity of the Best-First Search algorithm depends on the effectiveness of the priority function **f**:

- **Time Complexity**: Varies based on **f**; in the worst case, it can approach $O(b^d)$, where b is the branching factor and d is the maximum depth.
- **Space Complexity**: Typically $O(b \times d)$ to store nodes in the priority queue and the **reached** dictionary.

Performance relies on how accurately **f** guides the search; a well-designed **f** can significantly reduce both time and space requirements.

BFS (Breadth-First Search)

Implementation



```
1  node = SearchNode(problem.initial_state)
2
3  frontier : Queue[SearchNode] = Queue()
4  frontier.put(node)
5  reached = {node.state: node}
6
7  while not frontier.empty():
8      node = frontier.get()
9
10     if problem.is_goal(node.state):
11         return node
12
13     for child in problem.expand(node):
14         s = child.state
15         if s not in reached:
16             reached[s] = child
17             frontier.put(child)
18
19  return None
```

We begin the BFS algorithm by initializing a Queue called frontier and adding the initial node. Additionally, we have a dictionary named reached which takes responsibility for storing visited states and mapping each state to its corresponding node to prevent revisiting the same state.

After that, the while loop is used until the **frontier** is empty. In each iteration, a node is dequeued from the **frontier**. Then we will check whether this node is a goal based on the if condition. If the goal is found, we will return it. Otherwise, we expand the current node, generating all possible child nodes and adding them into the **frontier** after checking if it is reached or not. When the **frontier** is empty while the goal is not reached, the algorithm will return **None**

Complexity analysis

- Time complexity: $O(b^d)$, where b is the average number of children per node and d is the depth of the shallowest goal node.
- Space complexity: The total number of nodes generated is $1 + b^2 + b^3 + \dots + b^d = O(b^d)$ where b is the average number of children per node and d is the depth of the shallowest goal node.

DFS (Depth First Search)

Implementation

```
1 def search(self, problem: Problem) -> SearchNode|None:
2     node = SearchNode(problem.initial_state)
3     dfsStack: list[tuple[SearchNode, Generator[SearchNode, None, None]]] = [(node, problem.expand(node))]
4     visited : set[SearchState] = {problem.initial_state}
5
6     while len(dfsStack) > 0:
7         node, expand = dfsStack[-1]
8         try:
9             child = next(expand)
10            if problem.is_goal(child.state):
11                return child
12            if child.state not in visited:
13                visited.add(child.state)
14                dfsStack.append((child, problem.expand(child)))
15        except StopIteration:
16            visited.remove(node.state)
17            dfsStack.pop()
18
19     return None
```

We begin the DFS algorithm by initializing a stack called **dfsStack** and adding a tuple containing the initial node and its expansion generator. Additionally, we also have a visited set to store expanded states to prevent revisiting the same state. After that, the while loop is used until the **dfsStack** is empty.

In each iteration, a node and its expansion generator are retrieved from the top of the stack. We then attempt to get the next child node from the generator using **next(expand)**.

If a child node is found and its state is the goal (checked using **problem.is_goal(child.state)**), it returns the child node. If the child node's state has not been visited, it is added to the visited set, and the child node along with its expansion generator is pushed onto the stack. If the generator is exhausted (raises StopIteration), the current node is removed from the visited set, and the node is popped from the stack. If no solution is found, the method returns None.

Complexity analysis

- Time Complexity: $O(b^m)$, where (b) is the branching factor (maximum number of successors of any node) and (m) is the maximum depth of the search tree. In the worst case, DFS explores all nodes up to depth (m).
- Space Complexity: $O(bm)$, where (b) is the branching factor and (m) is the maximum depth of the search tree. DFS only needs to store the nodes on the current path from the root to a leaf, which is at most (m) nodes deep, each with up to (b) children.

UCS (Unified Cost Search)

Implementation

This UCS implementation leverages the **Best First Search** method, prioritizing nodes by their path cost.

```
1 def search(self, problem: Problem) -> SearchNode|None:
2     return best_first_search.search(problem, lambda node: node.path_cost)
```

Complexity analysis

- **Time Complexity:** $O(b^{1+LC^*/\epsilon})$, where C^* is the cost of the optimal solution path, b is the branching factor, and ϵ is the minimum edge cost. When all step costs are equal, the complexity simplifies to $O(b^{1+d})$, where d is the depth of the solution.
- **Space Complexity:** Similar to time complexity, as UCS must store all frontier nodes to ensure the shortest path is found.

A* algorithm

Heuristic function(s)

The A* algorithm leverages heuristic functions to estimate the cost from a given state to the goal, helping prioritize paths that appear promising. Here, two heuristic functions are implemented to provide admissible and consistent estimates.

Heuristic 1: Minimum Cost Matching with Manhattan Distance

The first heuristic uses the **minimum cost perfect matching** of the Manhattan distance between each stone and its closest switch. Each distance is weighted by the stone's weight plus one, representing the effort required to move that stone to its target. This heuristic is **admissible** and **consistent** because it never overestimates the true cost and satisfies the triangle inequality.



```
1 def heuristic1(state: SearchState) -> int:
2     if problem.is_deadend(state):
3         return np.inf
4     cost_matrix = [[
5         manhattan(stone_position, goal_position) * (problem.environment.stone_weights[i] + 1)
6         for goal_position in problem.environment.switch_positions
7     ] for i, stone_position in enumerate(state.stone_positions)]
8
9     return minimum_cost_perfect_matching(cost_matrix)
```

Explanation of Admissibility:

1. Assume the actual cost to place each stone s_i on its respective switch $S(s_i)$ is at least the Manhattan distance from s_i to $S(s_i)$, multiplied by $W(s_i)+1$, where $W(s_i)$ is the weight of s_i .
2. Since the heuristic calculates the minimum matching cost, it provides a lower or equal estimate to any valid matching, making it admissible.

Minimum Cost Matching Calculation:

Using the **Munkres (Hungarian)** algorithm, the function calculates the minimum cost matching:



```
1 from munkres import Munkres
2
3 def minimum_cost_perfect_matching(cost_matrix):
4     return sum(cost_matrix[i][j] for i, j in Munkres().compute(cost_matrix))
```

Heuristic 2: Precomputed Shortest Path Distances

The second heuristic improves on the first by using **precomputed shortest path distances** from each switch to all non-wall cells, replacing the Manhattan distance with these true shortest path distances. Since the map size is small and these distances are always equal to or greater than the Manhattan distance, this heuristic **dominates** the first one, providing tighter estimates.

```

1 self.distances = [
2     self.distant_matrix(switch_position, problem)
3     for switch_position in problem.environment.switch_positions
4 ]
5
6 def heuristic2(state: SearchState) -> int:
7     if problem.is_deadend(state):
8         return float('inf')
9     cost_matrix = [[
10         distance[tuple(stone_position)] * (problem.environment.stone_weights[i] + 1)
11         for distance in self.distances
12     ] for i, stone_position in enumerate(state.stone_positions)]
13
14     return minimum_cost_perfect_matching(cost_matrix)


```

Here, `distant_matrix` precomputes the distances from each switch to other cells, allowing `heuristic2` to use more accurate path costs, further enhancing the effectiveness of the A* algorithm.

Implementation

Since `heuristic2` is a **dominant heuristic** compared to `heuristic1` (it provides a more accurate cost estimation without overestimating the true cost), it guarantees fewer node expansions in the A* algorithm. This makes it our optimal choice for the primary heuristic.

In implementing A* with this heuristic, we apply the **Best First Search** framework, using a priority function that combines the path cost to reach a node (`node.path_cost`) and the heuristic estimate from `heuristic2`:



```
1 def search(self, problem: Problem) -> SearchNode|None:
2     return best_first_search.search(problem, lambda node: node.path_cost + heuristic2(node.state))
```

Complexity analysis

The complexity analysis for the **pre-computation** and **search** phases can be broken down as follows:

1. Precomputation for Heuristic 2:

- **Precomputing Shortest Paths:** In `heuristic2`, the distance matrix is precomputed for each switch to every other reachable cell in the map. This involves performing a shortest path search (using BFS) from each switch position to all non-wall cells in the map.
 - **Time Complexity:**
 - For each of the k switches, we need to compute the shortest path to all $n \times m$ cells (where n is the number of rows and m is the number of columns).
 - This gives a time complexity of $O(k \times n \times m)$ for the precomputation.
 - **Space Complexity:**
 - We need to store the distance matrix for each switch, which requires $O(k \times n \times m)$ space.

2. Per-Expansion Complexity (Minimum Cost Matching):

- **Minimum Cost Matching:** For each state expansion, we need to calculate the **minimum cost perfect matching** of the stones to the switches. This involves solving an optimization problem, which in this case is typically done using the **Hungarian algorithm** (or Munkres algorithm).
 - **Time Complexity:**

- The Hungarian algorithm has a time complexity of $O(k^3)$, where k is the number of stones (and switches).
- Since each expansion involves computing the minimum cost matching for the k stones and k switches, this contributes $O(k^3)$ to the complexity per expansion.

3. Search Phase (A* with Heuristic 2):

- **Time Complexity:**
 - Each node expansion involves calculating the heuristic value (which costs $O(bk^3)$ per expansion) and updating the frontier.
 - The overall complexity of the A* search depends on the number of nodes expanded and is approximately $O(bdk^3)$, where b is the branching factor and d is the depth of the solution.
- **Space Complexity:**
 - The space complexity is dominated by the storage required for the frontier and the explored nodes, as well as the storage for the distance matrix and the state information.
 - The space complexity for the search phase is $O(bd)$ (for the frontier) plus $O(k \times n \times m)$ for the distance matrices used in heuristic calculations.


Results








Test cases results

Results analysis

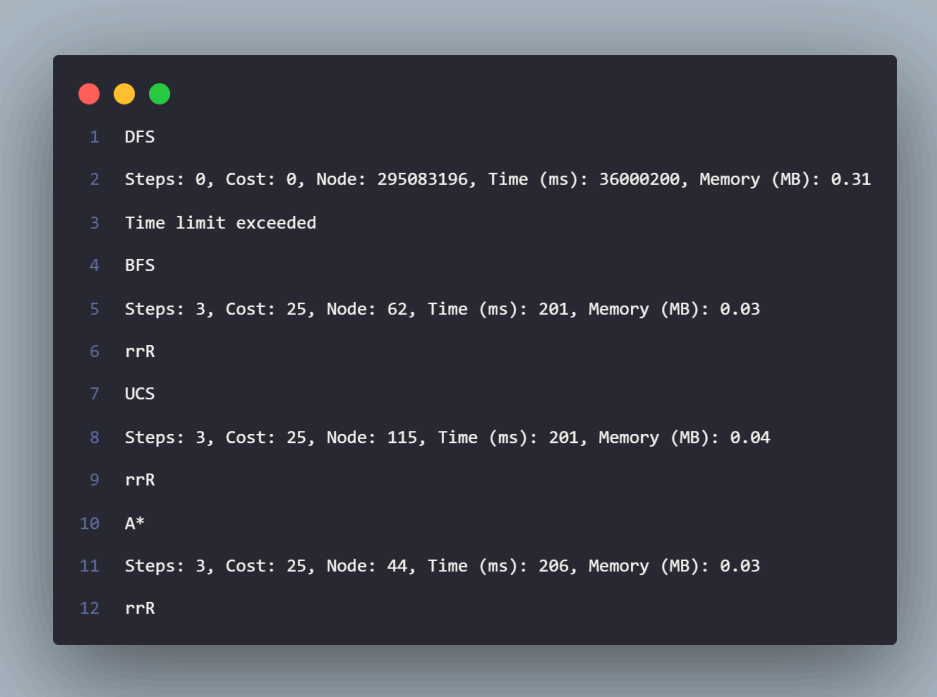
We analyze the results to identify the strengths and weaknesses of the four different search strategies used in the project:

1. Breadth-First Search (BFS):

- Completeness : BFS is guaranteed to find a solution if one exists within the search space.

- Optimality : When exists any positive weight stone, the action pushing this stone will cost more than 1 leading to un-uniform cost.
 - Memory Consuming: BFS exhibits high memory consumption as it stores all nodes at each depth level to avoid revisiting states. This can be a significant limitation in problems with large search spaces.
2. Uniform Cost Search (UCS):
- Completeness : UCS guarantees finding a solution if one exists, similar to BFS.
 - Optimality : UCS prioritizes finding the path with the lowest cumulative cost, regardless of the action cost, ensuring an optimal solution.
 - Time and Space Consuming: Similar to BFS, it stores all nodes it expands to avoid revisiting states. But since the optimal cost might be large leading to its larger time complexity in comparison to BFS.
3. Depth-First Search (DFS):
- Completeness : DFS guarantees finding a solution if one exists because since we store a set of the expanding paths to avoid a circle and the state space is finite.
 - Optimality : DFS does not guarantee optimality.
 - Memory Efficiency: Compared to every other searching algorithm, DFS requires significantly less memory, storing only the current path from the root to a leaf node along with unexplored siblings.
4. A* algorithm:
- Completeness : A* guarantees finding a solution if one exists, similar to BFS and UCS.
 - Optimality : Since our heuristic is admissible and consistent, the returned path is guaranteed to be optimal.
 - Efficiency: With our effective heuristic function, A* can outperform UCS by prioritizing promising paths and expanding fewer nodes.
 - Costly heuristic: Since the complexity of calculating the heuristic function is $O(k^3)$ with k is the number of stones. This can slow down the process about 125 times in our setup but since expanding fewer nodes, it can still be very fast.

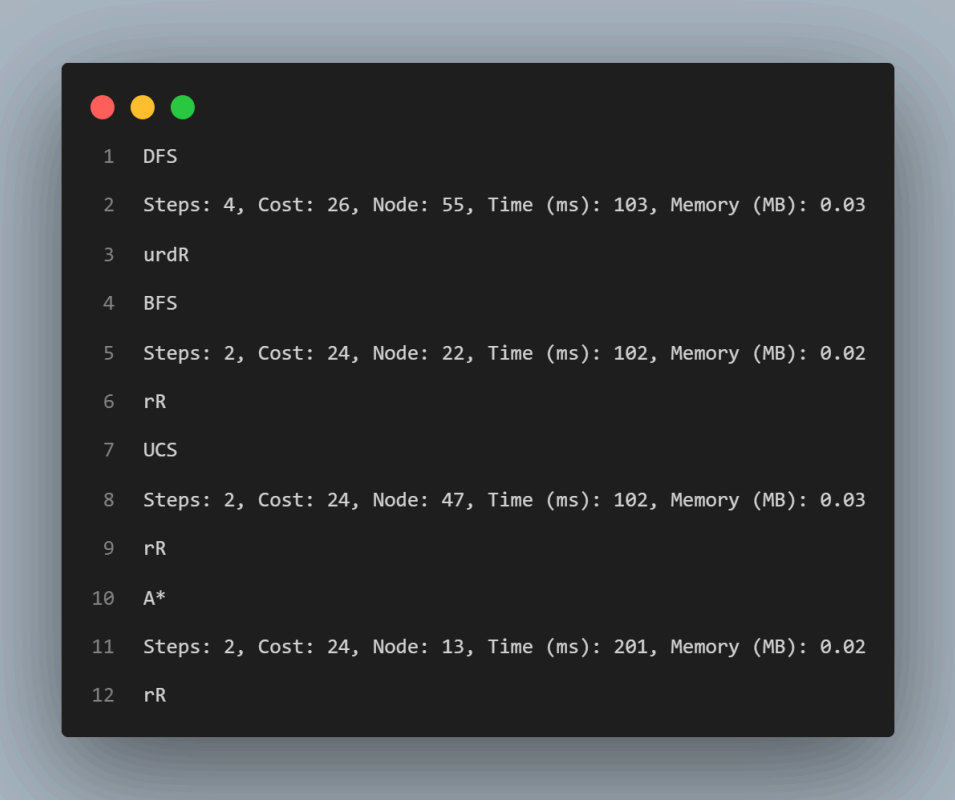
The following images can support our analysis. The first image contains the output of the easiest test case where it only has one rock.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays the results of three search algorithms: DFS, BFS, and A*. Each algorithm's output includes the number of steps, cost, number of nodes explored, time taken in milliseconds, and memory usage in MB. DFS is shown to have exceeded the time limit, while BFS and A* completed successfully with similar resource usage.

```
1 DFS
2 Steps: 0, Cost: 0, Node: 295083196, Time (ms): 36000200, Memory (MB): 0.31
3 Time limit exceeded
4 BFS
5 Steps: 3, Cost: 25, Node: 62, Time (ms): 201, Memory (MB): 0.03
6 rrR
7 UCS
8 Steps: 3, Cost: 25, Node: 115, Time (ms): 201, Memory (MB): 0.04
9 rrR
10 A*
11 Steps: 3, Cost: 25, Node: 44, Time (ms): 206, Memory (MB): 0.03
12 rrR
```

The first image of the outputs

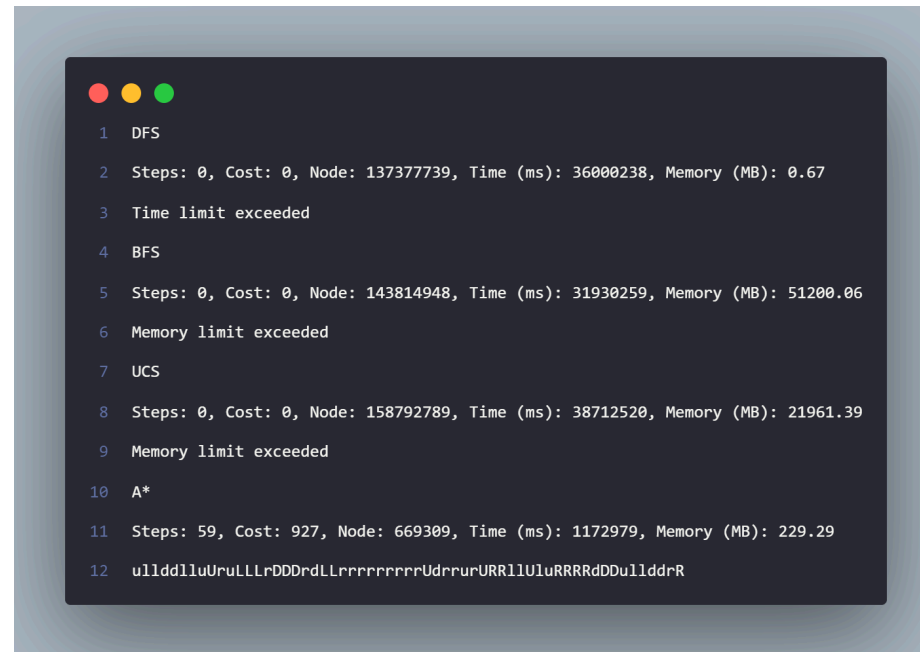
Breadth-First Search (BFS) is faster than A*, but BFS explores more paths. Both BFS and A* use the same resources. DFS is slow, exceeding the time limit in all test cases.

A terminal window with a dark background and light gray text. It displays the results of a search algorithm, listing 12 steps. Steps 1, 3, 5, 7, 9, 11, and 12 show the algorithm name (DFS, urdR, BFS, rR, UCS, A*, rR). Steps 2, 4, 6, 8, 10, and 12 show performance metrics: Steps, Cost, Node, Time (ms), and Memory (MB).

```
1 DFS
2 Steps: 4, Cost: 26, Node: 55, Time (ms): 103, Memory (MB): 0.03
3 urdR
4 BFS
5 Steps: 2, Cost: 24, Node: 22, Time (ms): 102, Memory (MB): 0.02
6 rR
7 UCS
8 Steps: 2, Cost: 24, Node: 47, Time (ms): 102, Memory (MB): 0.03
9 rR
10 A*
11 Steps: 2, Cost: 24, Node: 13, Time (ms): 201, Memory (MB): 0.02
12 rR
```

The second image of the outputs

The second image shows that even in a simple scenario, like finding a clear path from rocks to switches, Depth-First Search (DFS) can find a solution within a reasonable time. Additionally, A-star consistently proves to be the most efficient search strategy, using the least memory, and resources.

A terminal window with a dark background and light gray text. It displays the results of four different search algorithms: DFS, BFS, UCS, and A*. DFS and BFS both fail due to time and memory limits. UCS fails due to a memory limit. A* successfully finds a solution, showing the number of steps, cost, nodes visited, time, and memory used. The final line shows the solution path as a string of characters.

```
1 DFS
2 Steps: 0, Cost: 0, Node: 137377739, Time (ms): 36000238, Memory (MB): 0.67
3 Time limit exceeded
4 BFS
5 Steps: 0, Cost: 0, Node: 143814948, Time (ms): 31930259, Memory (MB): 51200.06
6 Memory limit exceeded
7 UCS
8 Steps: 0, Cost: 0, Node: 158792789, Time (ms): 38712520, Memory (MB): 21961.39
9 Memory limit exceeded
10 A*
11 Steps: 59, Cost: 927, Node: 669309, Time (ms): 1172979, Memory (MB): 229.29
12 ullddllluUruLLLLrDDDrDLLrrrrrrrrUdrrurURR1lUluRRRRdDDu1lldrR
```

The third image of the outputs

The third image shows that in a complex 7x20 maze with 5 rocks, A-star successfully finds a solution. However, Depth-First Search (DFS) times out, and both Breadth-First Search (BFS) and Uniform Cost Search (UCS) run out of memory. This clearly shows that A-star is the most efficient algorithm for this type of problem.

Youtube link

[GitHub link](#)

Self-evaluation

No.	Detail	Score	Self-evaluation
1.	Implement BFS correctly	10%	10%
2.	Implement DFS correctly	10%	10%
3.	Implement UCS correctly	10%	10%
4.	Implement A* correctly	10%	10%
5.	Generate at least 10 test cases for each level with different attributes	10%	10%
6.	Result (output file and UI)	15%	15%
7.	Video to demonstrate all algorithms for some test case	10%	10%
8.	Report your algorithm, experiment with some reflections or comments	25%	25%
Total		100%	100%

Work assignment

1. Project framework building division:

Student ID	Name	Percentage
22125108	Vòng Vĩnh Toàn	100%

2. Visualization division:

Student ID	Name	Percentage
22125077	Nguyễn Hoàng Phúc	100%

3. Validation division:

Student ID	Name	Percentage
22125038	Huỳnh Đăng Khoa	100%
22125049	Huỳnh Hà Phương Linh	100%

4. Algorithm implementations division:

Student ID	Name	Algorithm	Percentage
22125038	Huỳnh Đăng Khoa	Breadth-First Search	100%
22125049	Huỳnh Hà Phương Linh	Depth First Search	100%
22125077	Nguyễn Hoàng Phúc	Uniform Cost Search	100%
22125108	Vòng Vĩnh Toàn	A* Search	100%

5. Test generation division:

Student ID	Name	Test generation	Percentage
22125038	Huỳnh Đăng Khoa	input-01 to input-05	100%
22125049	Huỳnh Hà Phương Linh	input-06 to input-10	100%

Conclusion

In this project, we developed a search algorithm framework to solve a variation of the Sokoban puzzle, leveraging multiple search strategies, including Best-First Search and A*, to efficiently find solutions. We implemented two heuristics, with the second heuristic being dominant due to its better performance in guiding the search process. The use of a **consistent and admissible heuristic** ensures that the A* algorithm produces optimal solutions while maintaining efficiency.

Our approach integrates object-oriented design principles to structure the problem and search states, and we utilized advanced techniques like minimum cost perfect matching for the heuristic calculations. The **pre-computation** of shortest paths from switches to reachable cells ensures that the heuristic is both accurate and fast, but it also introduces an overhead that must be considered in terms of both time and space complexity.

In terms of **complexity**, the overall efficiency of the algorithm depends on the number of switches (k), the size of the map ($n \times m$), and the branching factor during the search phase.

Overall, this project demonstrates the effective application of search algorithms and heuristics in solving complex planning problems, providing a solid foundation for further optimization and experimentation.