ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

Báo cáo đồ án 1

Chủ đề: Các thuật toán tìm kiếm

Môn học: CƠ SỞ TRÍ TUỆ NHÂN TẠO

Sinh viên thực hiện:

22120060 - Trương Tiến Đạt

22120220 - Phạm Văn Hoàng Nam

22120403 - Đỗ Văn Tư

22120408 - Đỗ Thanh Tùng

Giảng viên:

Thầy Nguyễn Thanh Tình

Ngày 2 tháng 11 năm 2024

Mục lục

1 Phân Chia Công Việc và Video Demo

1.1 Phân chia công việc

Em xin được phép giới thiệu về nhóm 3, bao gồm các thành viên như sau:

- Đỗ Thanh Tùng - 22120408
- Trương Tiến Đạt - 22120060
- Đỗ Văn Tư - 22120403
- Phạm Văn Hoàng Nam - 22120220

Nhóm em xin gửi lời cảm ơn sâu sắc đến các giảng viên đã mang đến một đồ án thú vị và bổ ích, giúp em có cơ hội ôn lại các kiến thức đã học cũng như làm quen hơn với việc áp dụng các thuật toán vào thực tế, đặc biệt là qua việc cài đặt chúng bằng Python.
Dưới đây là bảng phân công công việc cùng mức độ hoàn thành của các thành viên trong nhóm đối với từng nhiệm vụ đã được giao.

Công việc	**Người thực hiện**	**Mức độ hoàn thành**
Giải thuật UCS + Video demo	Trương Tiến Đạt	100%
Giải thuật BFS, DFS + Tạo test case	Phạm Văn Hoàng Nam	100%
Giải thuật A* + Báo cáo	Đỗ Văn Tư	100%
Giao diện	Đỗ Thanh Tùng	100%

Bảng 1: Bảng phân công công việc

1.2 Video Demo

URL Video : https://youtu.be/w0riSjCeQ4I

2 Giải thích thuật toán

2.1 Breadth first search (BFS)

2.1.1 Mô tả thuật toán

BFS là thuật toán tìm kiếm theo tầng, mở rộng các nút theo từng mức. Bắt đầu từ một nút gốc, BFS sẽ duyệt qua tất cả các nút lân cận trước khi di chuyển sang các nút ở tầng tiếp theo. Thuật toán sử dụng hàng đợi để lưu trữ các nút cần duyệt.

- Khởi tạo hàng đợi với nút gốc và đánh dấu nó là đã thăm.
- Lặp lại cho đến khi hàng đợi rỗng:
  - Lấy nút đầu hàng đợi, duyệt qua các nút lân cận chưa thăm và thêm chúng vào hàng đợi.
- Kết thúc khi tất cả các nút có thể duyệt được từ gốc đã được thăm.

2.1.2 Giải thích các hàm phụ

Các phương thức cần thiết, và thuật toán BFS cùng DFS được chứa trong cùng một class SokobanSolver, dưới đây là phần giải thích cụ thể công dụng của từng phương thức:

- `__init__(self, weights, maze)`
  - **Mô tả:** Đây là hàm khởi tạo cho class `SokobanSolver`.
  - **Tham số:**
    - `weights`: Danh sách trọng lượng của các tảng đá (stones).
    - `maze`: Ma trận thể hiện mê cung cần giải.
  - **Chức năng:** Khởi tạo các biến thành viên:
    - `weights`: Lưu trữ trọng lượng của các tảng đá.
    - `maze`: Lưu trữ cấu trúc của mê cung.
    - `walls`: Cấu trúc dữ liệu `set` rỗng để lưu trữ các tọa độ của tường trong mê cung.
    - `switches`: `Set` rỗng để lưu trữ các tọa độ của các điểm chuyển (switches) trong mê cung.

    * `stones`: `Dict` rỗng để lưu trữ tọa độ của các tảng đá và trọng lượng của chúng.
    * `ares_pos`: Tọa độ của vị trí của nhân vật (Ares), được khởi tạo là `None`.
    * `maze_height`: Chiều cao của mê cung, được tính bằng số hàng trong ma trận.
    * `maze_width`: Chiều rộng của mê cung, được tính bằng chiều dài tối đa của các hàng trong ma trận.
    * `weight_index`: Biến chỉ số để theo dõi vị trí của trọng lượng trong danh sách `weights`, được khởi tạo bằng 0.
    * `initial_state`: Biến lưu trữ trạng thái ban đầu của trò chơi, được thiết lập thông qua phương thức `initialize_state()`.

- `initialize_state(self)`
  - **Mô tả:** Hàm phân tích mê cung và thiết lập trạng thái ban đầu.
  - **Chức năng:**
    * Duyệt qua từng ký tự trong mê cung để xác định và lưu trữ các thành phần như:
      · Tường (`#`): Lưu vào `walls`.
      · Vị trí nhân vật (`@` hoặc `+`): Lưu vào `ares_pos`.
      · Đá (`$` hoặc `*`): Lưu vào `stones` và cập nhật trọng lượng.
      · Điểm mục tiêu (`.` hoặc `+`): Lưu vào `switches`.
    * Trả về dictionary chứa trạng thái ban đầu với vị trí nhân vật, các tảng đá

- `check_finish(self, state)`
  - **Mô tả:** Kiểm tra xem mê cung đã được giải xong chưa (tất cả tảng đá được đưa về điểm mục tiêu).
  - **Tham số:**
    * `state`: Trạng thái cần kiểm tra.
  - **Chức năng:**
    * Kiểm tra xem tất cả các tảng đá đã nằm trên điểm mục tiêu hay chưa. Nếu có, trả về `True`, ngược lại trả về `False`.

- `get_memory(self)`

  - **Mô tả:** Phương thức này dùng để kiểm tra lượng bộ nhớ (RAM) hiện tại mà chương trình đang sử dụng.
  - **Chức năng:**
    * Sử dụng thư viện `psutil` để lấy thông tin về tiến trình hiện tại của chương trình, dựa trên ID tiến trình (`os.getpid()`).
    * Lấy dung lượng bộ nhớ RAM mà tiến trình đang sử dụng, dưới dạng thông tin `rss` (Resident Set Size), tức là bộ nhớ vật lý thực sự chiếm dụng bởi chương trình.
    * Trả về giá trị bộ nhớ sử dụng (tính bằng MB).
    * Giúp đánh giá và kiểm tra việc sử dụng tài nguyên khi thực hiện các thuật toán.

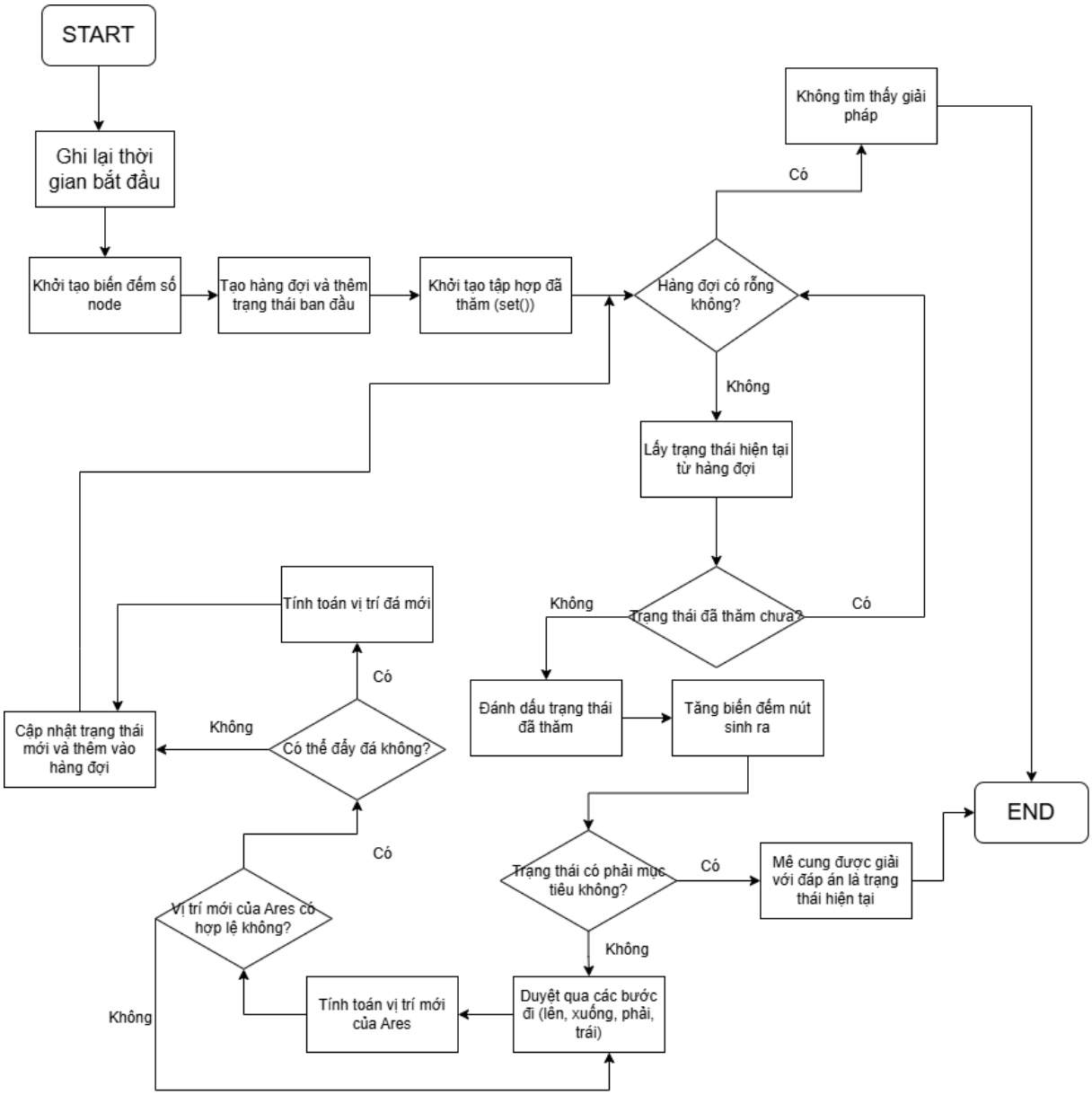
- `solve(self, algo_name)`
  - **Mô tả:** Giải quyết bài toán với thuật toán được chỉ định.
  - **Tham số:**
    * `algo_name`: Tên của thuật toán tìm kiếm muốn dùng.
  - **Chức năng:** Gọi hàm tương ứng với thuật toán được chỉ định và trả về kết quả nếu tìm thấy cách giải mê cung, nếu không trả về `None`.

- `write_output(self, filename, algo_name, steps, total_weight, nodes_generated, time_tak memory_used, path)`
  - **Mô tả:** Ghi kết quả tìm kiếm vào file `txt`
  - **Tham số:**
    * `filename`: Tên tệp để ghi kết quả.
    * `algo_name`: Tên thuật toán sử sử dụng.
    * `steps`: Số bước `Ares` thực hiện.
    * `total_weight`: Tổng trọng lượng đã đẩy.
    * `nodes_generated`: Số node được tạo ra.
    * `time_taken`: Thời gian thực hiện thuật toán.
    * `memory_used`: Lượng bộ nhớ thuật toán đã sử dụng.
    * `path`: Đường đi để giải mê cung (Danh sách các hành động).

2.1.3 Minh họa thuật toán BFS

Đây là flowchart cụ thể của thuật toán `BFS`

- **Lưu ý:**
  - Trạng thái bao gồm:
    - `ares_pos`: Vị trí của Ares ((x, y))
    - `stones`: Các tảng đá có trong mê cung được lưu theo dạng {vị trí : trọng lượng}
    - `actions`: Mảng lưu các hành động đã thực hiện
    - `total_weights`: Tổng trọng lượng đã đẩy trên đường đi
  - Hành động:
    - Chữ thường (u, d, l, r): Di chuyển không đẩy đá.
    - Chữ hoa (U, D, L, R): Di chuyển có đẩy đá.
  - Vị trí hợp lệ khi:
    - Nằm trong kích thước mê cung.
    - Không phải là tường (wall).
    - Không trùng vị trí đá khác (khi đẩy đá).



Hình 1: Flowchart minh họa thuật toán BFS

2.2 Depth first search (DFS)

2.2.1 Mô tả thuật toán

Để tối ưu khả năng tìm kiếm của thuật toán, em quyết định dùng biến thể `Iterative Deepening Depth-First Search (IDDFS)` thay vì dùng `DFS` thông thường. Ở mỗi lần lặp, `IDDFS` nhận vào một giới hạn độ sâu `max_depth`, độ sâu tối đa này sẽ được tăng dần qua mỗi vòng lặp.

IDDFS kết hợp ưu điểm của cả DFS và BFS bằng cách thực hiện tìm kiếm theo chiều sâu với các giới hạn độ sâu tăng dần. Tức là, IDDFS dùng DFS để duyệt sâu vào các nhánh, nhưng dừng lại khi đạt đến giới hạn độ sâu hiện tại. Khi giới hạn này tăng dần, IDDFS sẽ đảm bảo duyệt theo từng tầng (như BFS) trong khi vẫn tiết kiệm bộ nhớ (như DFS).
Cụ thể:

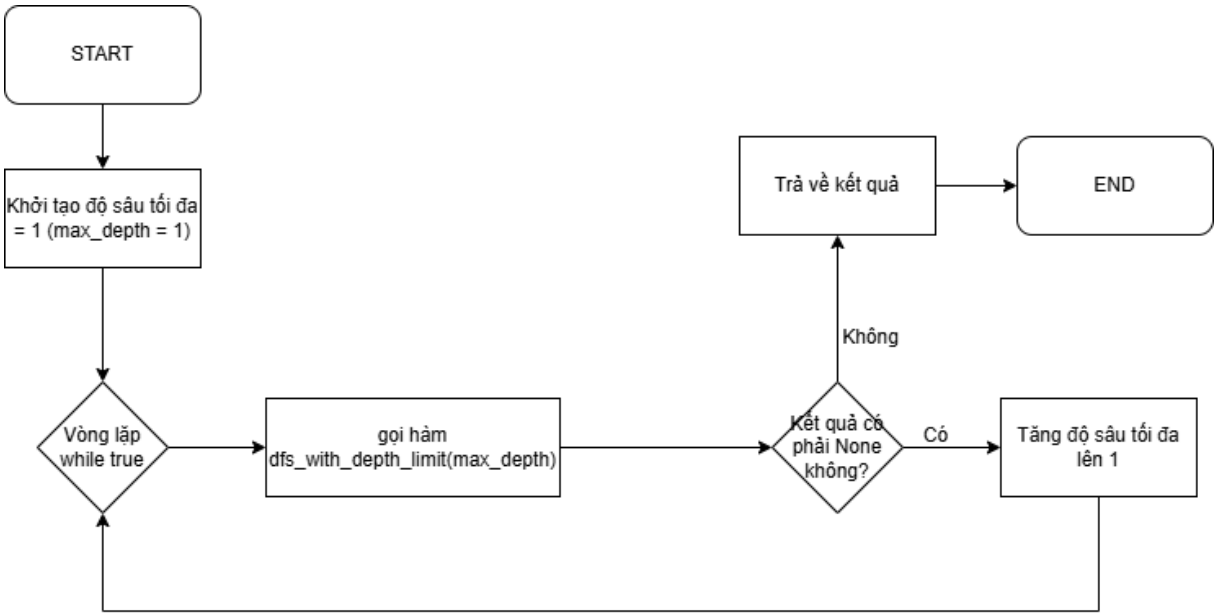
- Giới hạn độ sâu: Bắt đầu với một giới hạn độ sâu nhỏ (thường là 0), sau đó tăng giới hạn này lên sau mỗi lần duyệt.
- Duyệt DFS đến giới hạn:
  * Với mỗi giới hạn độ sâu, thực hiện DFS từ nút gốc.
  * Trong DFS, mỗi lần đến một nút mới, nếu độ sâu hiện tại chưa đạt giới hạn, tiếp tục đi sâu vào nhánh đó.
  * Nếu đạt giới hạn độ sâu, dừng lại và quay lui để duyệt các nhánh khác trong cùng tầng.
- Tăng giới hạn độ sâu:
  * Sau khi duyệt hết ở giới hạn hiện tại mà chưa tìm thấy kết quả, tăng giới hạn thêm 1 và lặp lại DFS.
  * Quá trình này tiếp tục cho đến khi tìm thấy nút mục tiêu hoặc đạt đến độ sâu mong muốn.

2.2.2 Giải thích các hàm phụ

Các hàm phụ được sử dụng tương tự như với thuật toán BFS phía trên.

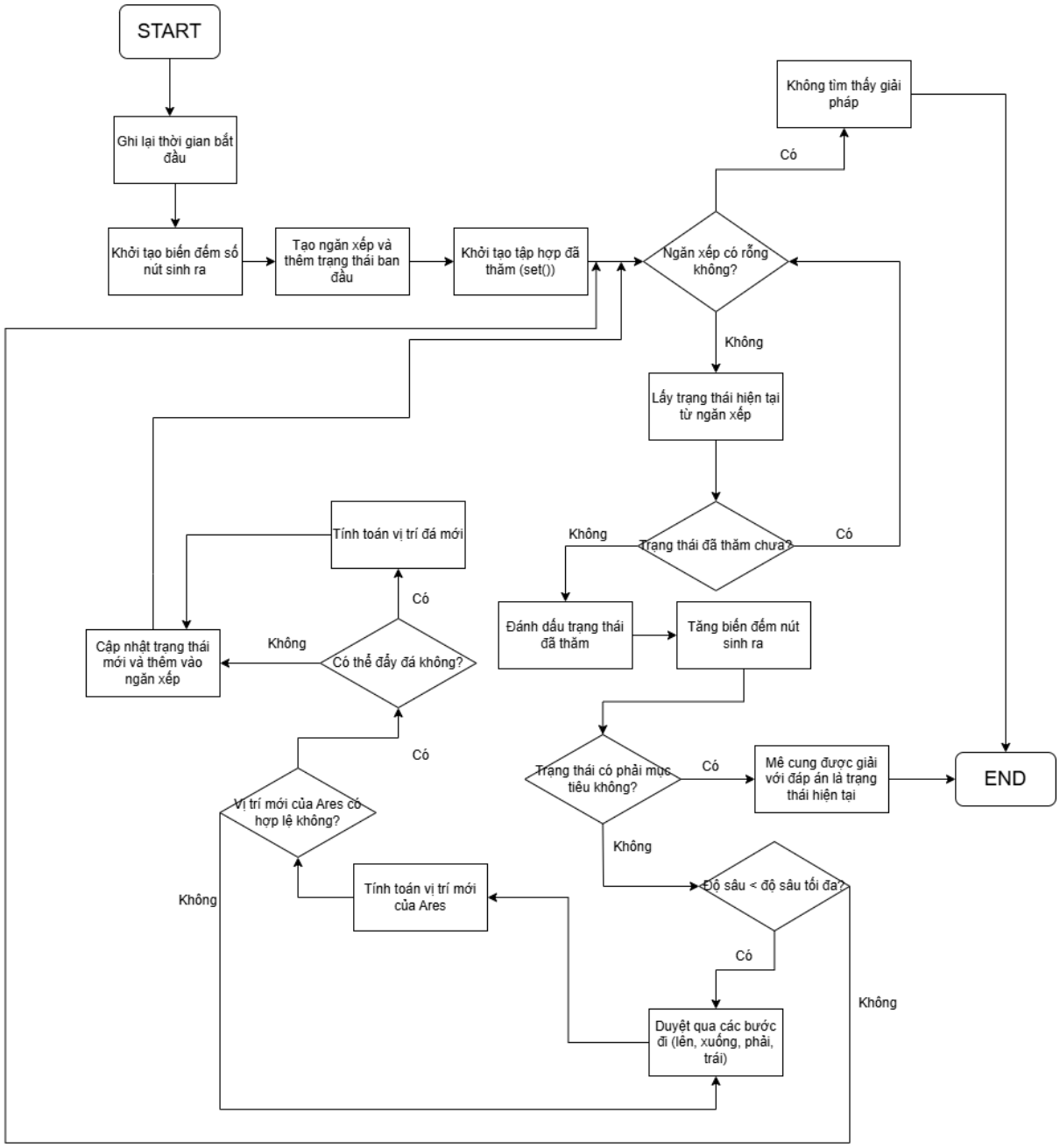
2.2.3 Minh họa thuật toán Deepening Depth-First Search(IDDFS)

Đây là flowchart cụ thể của thuật toán IDDFS



Hình 2: Flowchart minh họa thuật toán IDDFS

Còn bên dưới là những gì sẽ diễn ra khi gọi hàm `dfs_with_depth_limit` và truyền vào tham số `max_depth`



Hình 3: Flowchart minh họa thuật toán DFS

2.3 Uniform cost search (UCS)

2.3.1 Mô tả thuật toán

Thuật toán được sử dụng để tìm đường đi tối ưu từ vị trí ban đầu của nhân vật đến trạng thái mục tiêu. Giải thuật này bao gồm các bước:

- Quy ước chi phí là công cần thiết mà Ares cần thực hiện để đẩy cục đá đến đích: Ví dụ ở mỗi bước di chuyển nếu không đẩy đá thì chi phí là 1, ngược lại chi phí là khối lượng của cục đá vừa đẩy + 1
- Sử dụng min-heap để lưu các trạng thái mở. Trạng thái tại mỗi thời điểm bao gồm:

```
{
    "ares_pos": Vị trí Ares (x,y),
    "stones": {"(2,3) : 23 } Với key là vị trí đá (x,y) value là cân nặng đá,
    "actions": [] Là các bước đã di chuyển  ,
    "total_weight": 0 Tổng khối lượng đá đã đẩy được,
    "cost_so_far": 0  Chi phí (công) đã thực hiện
}
```

- Lựa chọn di chuyển tối ưu tại mỗi bước dựa trên chi phí đã đi và ước lượng chi phí còn lại.
- Theo dõi trạng thái đã duyệt để tránh lặp lại.

2.3.2 Giải thích các hàm phụ

1. `__init__(self, weights, grid)`

   Mô tả: Hàm khởi tạo (constructor) của lớp, thiết lập các thuộc tính ban đầu cho solver.

   Tham số:

   - `weights`: Một danh sách các trọng số của các khối đá trong bài toán Sokoban.
   - `grid`: Lưới chứa thông tin về vị trí tường, khối đá, công tắc và vị trí của người chơi.

   Cách hoạt động: Hàm khởi tạo lấy lưới và trọng số làm đầu vào, phân tích lưới để xác định vị trí của các vật thể và lưu vào các thuộc tính của đối tượng, như tường (`walls`), công tắc (`switches`), vị trí của các khối đá (`stones`), và vị trí của người chơi (`ares_pos`).

   Chức năng: Chuẩn bị trạng thái ban đầu cho solver, giúp khởi tạo dữ liệu cần thiết cho việc giải quyết bài toán.

2. `initialize_state(self)`

   Mô tả: Hàm tạo trạng thái khởi đầu của trò chơi dựa trên lưới đầu vào.

Tham số: Không có tham số đầu vào.

Cách hoạt động: Duyệt qua từng ô trong `grid` để xác định các vị trí đặc biệt (tường, công tắc, khối đá, vị trí người chơi), và lưu thông tin đó vào các thuộc tính của đối tượng. Các khối đá được liên kết với trọng số của chúng theo thứ tự.

Chức năng: Xác định và lưu trữ thông tin về trạng thái ban đầu, bao gồm vị trí của người chơi, khối đá, tổng trọng số và các hành động đã thực hiện.

3. `print_json(self, obj, indent=4)`

   Mô tả: In thông tin của một đối tượng dưới dạng JSON.

   Tham số:

   - `obj`: Đối tượng cần in ra dạng JSON.
   - `indent`: Số lượng khoảng trắng để thụt đầu dòng trong JSON (mặc định là 4).

   Cách hoạt động: Chuyển đổi đối tượng đầu vào thành dạng có thể tuần tự hóa (JSON serializable) bằng cách chuyển đổi các cặp tọa độ và các cấu trúc dữ liệu phức tạp sang dạng chuỗi.

   Chức năng: Hỗ trợ hiển thị trạng thái hoặc thông tin của solver dưới dạng JSON để dễ theo dõi và gỡ lỗi.

4. `write_output(self, filename, algorithm_name, steps, total_weight, nodes_generated, time_taken, memory_used, solution)`

   Mô tả: Ghi kết quả của quá trình giải Sokoban vào tệp.

   Tham số:

   - `filename`: Tên tệp xuất kết quả.
   - `algorithm_name`: Tên của thuật toán sử dụng.
   - `steps`: Số bước đi trong lời giải.
   - `total_weight`: Tổng trọng số của các khối đá đã di chuyển.
   - `nodes_generated`: Số lượng nút đã được tạo ra.
   - `time_taken`: Thời gian giải (tính bằng mili giây).
   - `memory_used`: Bộ nhớ sử dụng (tính bằng MB).
   - `solution`: Chuỗi biểu diễn các hành động trong lời giải.

   Cách hoạt động: Mở tệp với tên được chỉ định và ghi các thông tin về kết quả của lời

giải vào đó.

Chức năng: Tạo một bản ghi của kết quả giải để người dùng có thể kiểm tra lại sau khi thuật toán chạy xong.

5. `is_goal_state(self, state)`

   Mô tả: Kiểm tra xem trạng thái hiện tại có phải là trạng thái đích không.

   Tham số:

   – `state`: Trạng thái hiện tại, chứa thông tin về vị trí các khối đá.

   Cách hoạt động: Duyệt qua tất cả các khối đá và kiểm tra xem mỗi khối đá có nằm trên một ô công tắc không.

   Chức năng: Xác định khi nào thuật toán tìm thấy trạng thái đích để kết thúc quá trình tìm kiếm.

6. `get_memory_usage(self)`

   Mô tả: Lấy lượng bộ nhớ đang sử dụng của quá trình hiện tại.

   Tham số: Không có tham số đầu vào.

   Cách hoạt động: Sử dụng thư viện `psutil` để lấy thông tin bộ nhớ của tiến trình hiện tại và chuyển đổi đơn vị sang MB.

   Chức năng: Hỗ trợ đo lường hiệu suất bộ nhớ để báo cáo trong kết quả cuối cùng của lời giải.

7. `UCS_search(self)`

   Mô tả: Thuật toán tìm kiếm UCS để tìm đường đi từ trạng thái khởi đầu đến trạng thái đích với việc mở các nút liền kề với các nút có chi phí tập nhất, cập nhật chi phí và đóng nút vừa mở.

   Tham số: Không có tham số đầu vào.

   Cách hoạt động:

   – Sử dụng một hàng đợi ưu tiên (`open_list`) để lưu các trạng thái chưa được khám phá theo thứ tự $f = \text{cost} + \text{heuristic}$.

   – Tính f cho mỗi trạng thái con và kiểm tra xem trạng thái đã khám phá hết chưa bằng tập hợp `closed_set`.

   – Khi đạt đến trạng thái đích, hàm trả về thông tin chi tiết về lời giải.

   Chức năng: Thực hiện thuật toán UCS để tìm lời giải cho bài toán Sokoban.

8. `solve(self)`

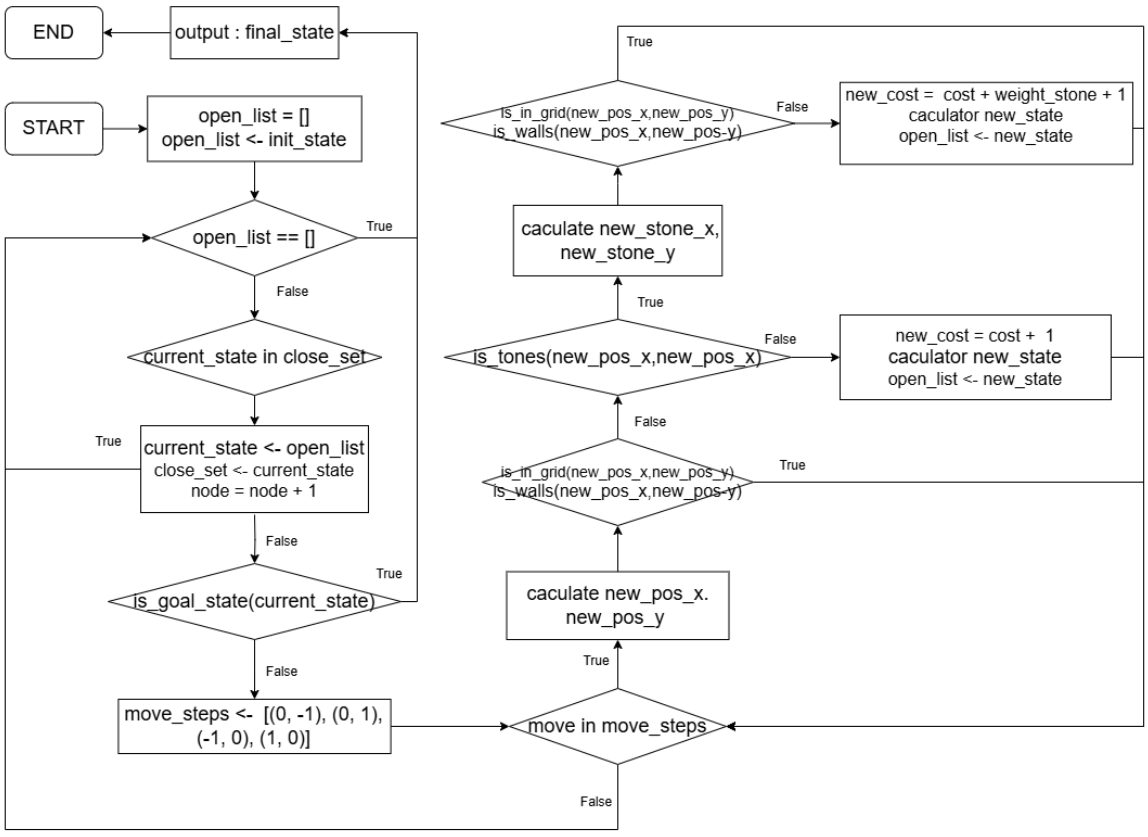
Mô tả: Hàm chính để giải quyết bài toán Sokoban.

Tham số: Không có tham số đầu vào.

Cách hoạt động: Khởi tạo trạng thái ban đầu và chạy thuật toán UCS, sau đó trả về kết quả.

Chức năng: Kết hợp các chức năng khởi tạo và tìm kiếm để tìm ra giải pháp từ trạng thái khởi đầu đến trạng thái đích và in ra kết quả JSON.

2.3.3 Flow Chart của thuật toán UCS



Hình 4: Hình minh thuật toán UCS

2.4 A star (A*)

2.4.1 Mô tả thuật toán

Thuật toán A* được sử dụng để tìm đường đi tối ưu từ vị trí ban đầu của nhân vật đến trạng thái mục tiêu. Giải thuật này bao gồm các bước:

- Quy ước chi phí là công cần thiết mà Ares cần thực hiện để đẩy cục đá đến đích: Ví dụ ở mỗi bước di chuyển nếu không đẩy đá thì chi phí là 1, ngược lại chi phí là khối lượng của cục đá vừa đẩy
- Hàm heuristic h được tính bằng:

$$h = \sum_{i=1}^{n} \min_{s \in \text{switches}} (d(\text{stone}_i, s) \times \text{weight}_i)$$

  Trong đó:

  - $\sum_{i=1}^{n}$: Ký hiệu tổng từ $i = 1$ đến n, với n là tổng số khối đá (stones).

  - $\min_{s \in \text{switches}}$: Tìm giá trị nhỏ nhất khi s thuộc tập hợp các công tắc switches.

  - $d(\text{stone}_i, s)$: Hàm khoảng cách giữa vị trí khối đá stone_i và công tắc s. Trong bài toán này, khoảng cách này có thể là khoảng cách Manhattan.

  - weight_i: Trọng số của khối đá thứ i, tương ứng với trọng lượng cần dùng để di chuyển khối đá.

- Sử dụng min-heap để lưu các trạng thái mở. Trạng thái tại mỗi thời điểm bao gồm:

```
{
    "ares_pos": Vị trí Ares (x,y),
    "stones": {"(2,3) : 23 } Với key là vị trí đá (x,y) value là cân nặng đá,
    "actions": [] Là các bước đã di chuyển  ,
    "total_weight": 0 Tổng khối lượng đá đã đẩy được,
    "cost_so_far": 0  Chi phí (công) đã thực hiện
}
```

– Lựa chọn di chuyển tối ưu tại mỗi bước dựa trên chi phí đã đi và ước lượng chi phí còn lại.

– Theo dõi trạng thái đã duyệt để tránh lặp lại.

2.4.2 Giải thích các hàm phụ

1. `__init__(self, weights, grid)`

   Mô tả: Hàm khởi tạo (constructor) của lớp, thiết lập các thuộc tính ban đầu cho solver.

   Tham số:

   – `weights`: Một danh sách các trọng số của các khối đá trong bài toán Sokoban.

   – `grid`: Lưới chứa thông tin về vị trí tường, khối đá, công tắc và vị trí của người chơi.

   Cách hoạt động: Hàm khởi tạo lấy lưới và trọng số làm đầu vào, phân tích lưới để xác định vị trí của các vật thể và lưu vào các thuộc tính của đối tượng, như tường (`walls`), công tắc (`switches`), vị trí của các khối đá (`stones`), và vị trí của người chơi (`ares_pos`).

   Chức năng: Chuẩn bị trạng thái ban đầu cho solver, giúp khởi tạo dữ liệu cần thiết cho việc giải quyết bài toán.

2. `initialize_state(self)`

   Mô tả: Hàm tạo trạng thái khởi đầu của trò chơi dựa trên lưới đầu vào.

   Tham số: Không có tham số đầu vào.

   Cách hoạt động: Duyệt qua từng ô trong `grid` để xác định các vị trí đặc biệt (tường, công tắc, khối đá, vị trí người chơi), và lưu thông tin đó vào các thuộc tính của đối tượng. Các khối đá được liên kết với trọng số của chúng theo thứ tự.

   Chức năng: Xác định và lưu trữ thông tin về trạng thái ban đầu, bao gồm vị trí của người chơi, khối đá, tổng trọng số và các hành động đã thực hiện.

3. `print_json(self, obj, indent=4)`

   Mô tả: In thông tin của một đối tượng dưới dạng JSON.

   Tham số:

   – `obj`: Đối tượng cần in ra dạng JSON.

   – `indent`: Số lượng khoảng trắng để thụt đầu dòng trong JSON (mặc định là 4).

Cách hoạt động: Chuyển đổi đối tượng đầu vào thành dạng có thể tuần tự hóa (JSON serializable) bằng cách chuyển đổi các cặp tọa độ và các cấu trúc dữ liệu phức tạp sang dạng chuỗi.

Chức năng: Hỗ trợ hiển thị trạng thái hoặc thông tin của solver dưới dạng JSON để dễ theo dõi và gỡ lỗi.

4. `write_output(self, filename, algorithm_name, steps, total_weight, nodes_generated, time_taken, memory_used, solution)`

   Mô tả: Ghi kết quả của quá trình giải Sokoban vào tệp.

   Tham số:

   - `filename`: Tên tệp xuất kết quả.
   - `algorithm_name`: Tên của thuật toán sử dụng.
   - `steps`: Số bước đi trong lời giải.
   - `total_weight`: Tổng trọng số của các khối đá đã di chuyển.
   - `nodes_generated`: Số lượng nút đã được tạo ra.
   - `time_taken`: Thời gian giải (tính bằng mili giây).
   - `memory_used`: Bộ nhớ sử dụng (tính bằng MB).
   - `solution`: Chuỗi biểu diễn các hành động trong lời giải.

   Cách hoạt động: Mở tệp với tên được chỉ định và ghi các thông tin về kết quả của lời giải vào đó.

   Chức năng: Tạo một bản ghi của kết quả giải để người dùng có thể kiểm tra lại sau khi thuật toán chạy xong.

5. `heuristic(self, state)`

   Mô tả: Hàm ước lượng chi phí từ trạng thái hiện tại đến trạng thái đích.

   Tham số:

   - `state`: Trạng thái hiện tại của trò chơi, bao gồm vị trí các khối đá và người chơi.

   Cách hoạt động: Tính tổng khoảng cách Manhattan (có trọng số) * Khối lượng từ mỗi khối đá đến công tắc gần nhất, giúp xác định mức công tối thiểu để đi đến trạng thái đích.

Chức năng: Cung cấp một hàm ước lượng cho thuật toán A* để lựa chọn trạng thái tốt nhất để mở tiếp theo, giúp giảm số bước và tăng hiệu suất của thuật toán.

6. `is_goal_state(self, state)`

   Mô tả: Kiểm tra xem trạng thái hiện tại có phải là trạng thái đích không.

   Tham số:

   - `state`: Trạng thái hiện tại, chứa thông tin về vị trí các khối đá.

   Cách hoạt động: Duyệt qua tất cả các khối đá và kiểm tra xem mỗi khối đá có nằm trên một ô công tắc không.

   Chức năng: Xác định khi nào thuật toán tìm thấy trạng thái đích để kết thúc quá trình tìm kiếm.

7. `get_memory_usage(self)`

   Mô tả: Lấy lượng bộ nhớ đang sử dụng của quá trình hiện tại.

   Tham số: Không có tham số đầu vào.

   Cách hoạt động: Sử dụng thư viện `psutil` để lấy thông tin bộ nhớ của tiến trình hiện tại và chuyển đổi đơn vị sang MB.

   Chức năng: Hỗ trợ đo lường hiệu suất bộ nhớ để báo cáo trong kết quả cuối cùng của lời giải.

8. `a_star_search(self)`

   Mô tả: Thuật toán tìm kiếm A* để tìm đường đi từ trạng thái khởi đầu đến trạng thái đích với chi phí thấp nhất.

   Tham số: Không có tham số đầu vào.

   Cách hoạt động:

   - Sử dụng một hàng đợi ưu tiên (`open_list`) để lưu các trạng thái chưa được khám phá theo thứ tự $f = \text{cost} + \text{heuristic}$.
   - Tính f cho mỗi trạng thái con và kiểm tra xem trạng thái đã khám phá hết chưa bằng tập hợp `closed_set`.
   - Khi đạt đến trạng thái đích, hàm trả về thông tin chi tiết về lời giải.

   Chức năng: Thực hiện thuật toán A* để tìm lời giải cho bài toán Sokoban.

9. `solve(self)`

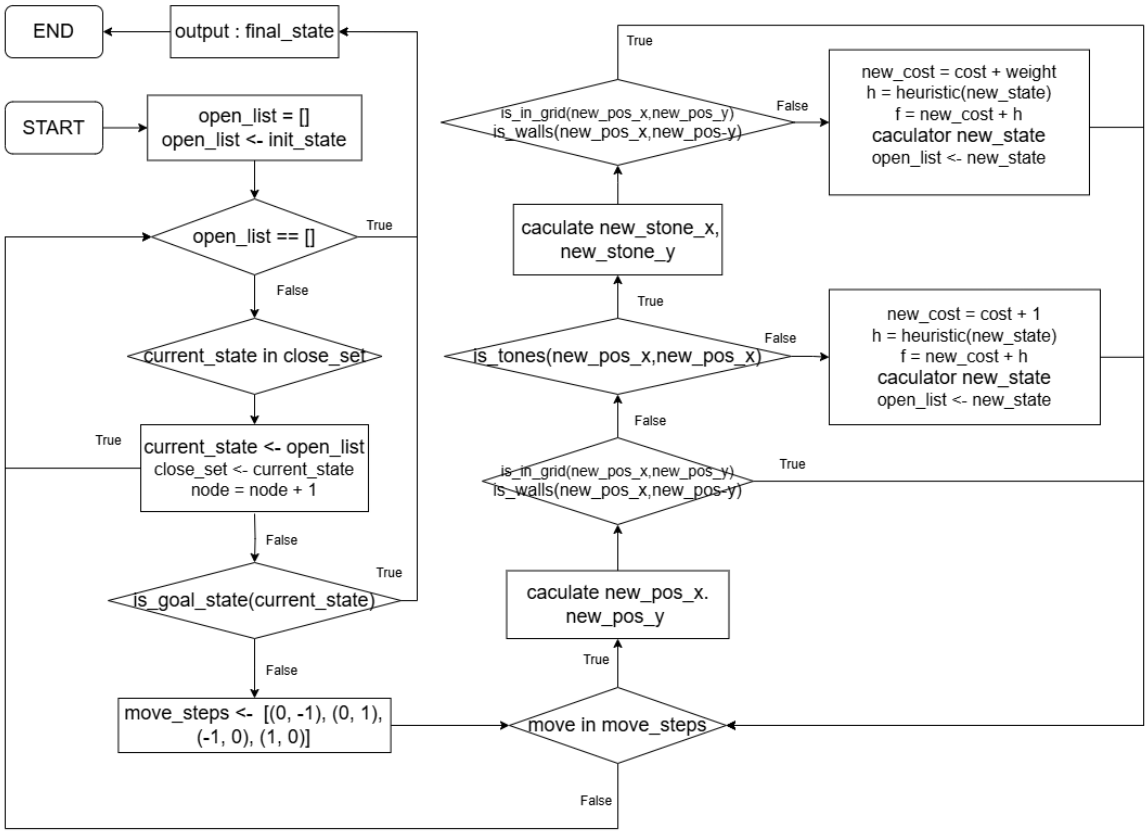
Mô tả: Hàm chính để giải quyết bài toán Sokoban.

Tham số: Không có tham số đầu vào.

Cách hoạt động: Khởi tạo trạng thái ban đầu và chạy thuật toán A*, sau đó trả về kết quả.

Chức năng: Kết hợp các chức năng khởi tạo và tìm kiếm để tìm ra giải pháp từ trạng thái khởi đầu đến trạng thái đích và in ra kết quả JSON.

2.4.3 Minh họa flow hoạt động của hàm `a_star_search(self)`

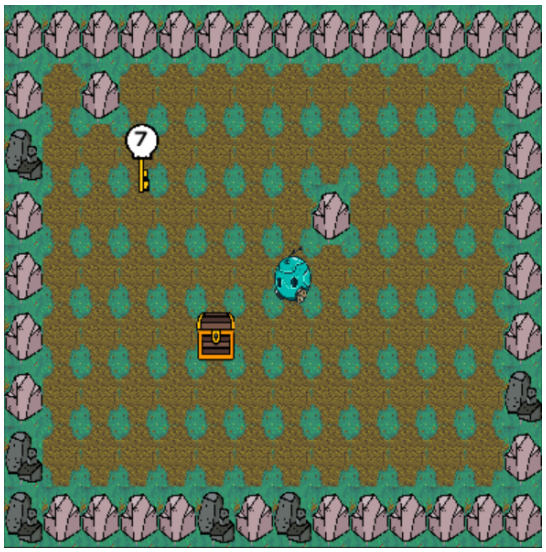


Hình 5: Hình minh thuật toán A*

3 Kiểm tra và đánh giá thuật toán

3.1 Test case 1

Map



Hình 6: Map test case 1

Output

Algorithm	Steps	Weight	Node	Time (ms)	Memory (MB)
A*	14	35	93	1.46	32.51
BFS	14	35	532	4.10	16.27
DFS	18	35	119	1.02	16.04
UCS	14	35	2001	29.28	33.86

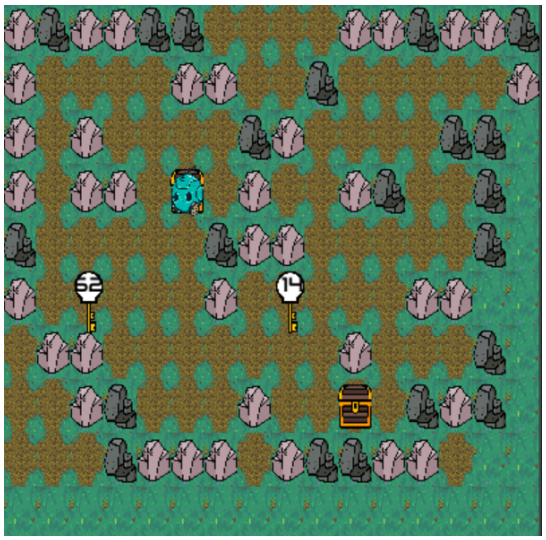
Bảng 2: Kết quả các thuật toán với test case 1

Path

- **DFS:** lllllluruRRlurrDDD
- **BFS:** uuullllDDDldRR
- **UCS:** ullllluRRurDDD
- **A*:** ullllluRRurDDD

3.2 Test case 2

Map



Hình 7: Map test case 2

Output

Algorithm	Steps	Weight	Node	Time (ms)	Memory (MB)
A*	44	366	974	19.88	33.41
BFS	32	394	15595	189.05	27.21
DFS	71	394	3580	33.23	20.35
UCS	34	366	19393	277.24	43.32

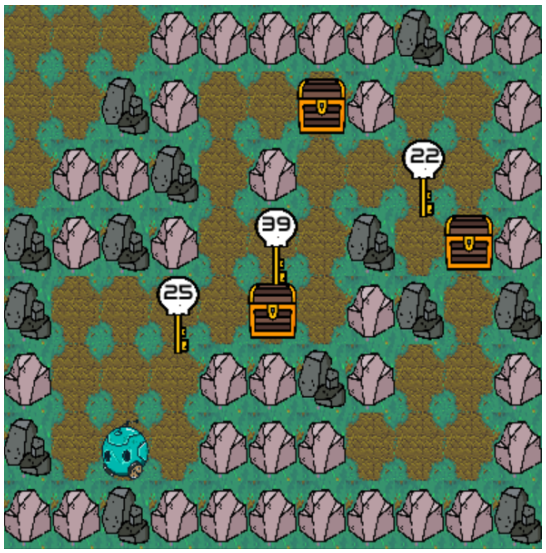
Bảng 3: Kết quả các thuật toán với test case 2

Path

- **DFS:** lulullddrrrrdlldrrrruRldrrUrrurrulullldDDDldRlullllllurrullllldRRRldrrUU
- **BFS:** dllllldRRRdrUUddrruRRurrdLulDDldR
- **UCS:** dllllldRRRdrUUddrruRddrrruuullDDldR
- **A*:** dddrruRdllluullllldRRRdrUUddrrrdrrruuullDDldR

3.3 Test case 3

Map



Hình 8: Map test case 3

Output

Algorithm	Steps	Weight	Node	Time (ms)	Memory (MB)
A*	39	361	4154	114.51	35.11
BFS	39	361	7671	110.10	27.25
DFS	53	361	7074	64.09	20.85
UCS	39	361	8937	111.77	43.35

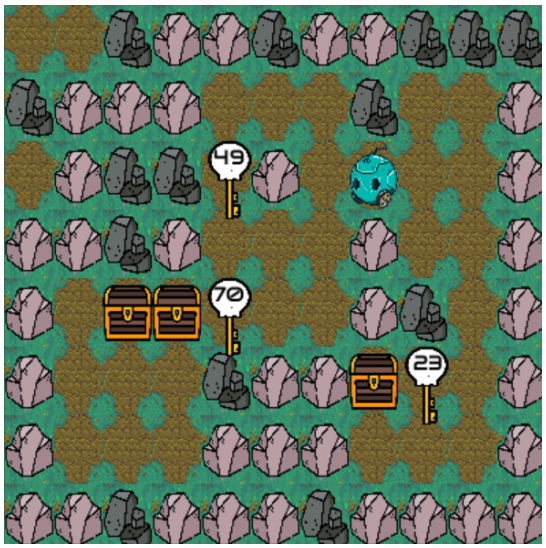
Bảng 4: Kết quả các thuật toán với test case 3

Path

- **DFS:** luruRRuuurrdddLLLrrruuullddRldrrUlldldlluRRRurrUrRurD
- **BFS:** uuRRuuurrdddLLuRdrUdllLdlluRRRurrUrRurD
- **UCS:** uuRRuuurrdddLLuRdrUdllLdlluRRRurrUrRurD
- **A*:** uuRRuuurrdddLLuRdrUdllLdlluRRRurrUrRurD

3.4 Test case 4

Map



Hình 9: Map test case 4

Output

Algorithm	Steps	Weight	Node	Time (ms)	Memory (MB)
A*	30	310	384	9.11	35.11
BFS	30	310	1762	15.34	26.75
DFS	41	310	140	1.00	20.35
UCS	30	310	2135	23.09	42.73

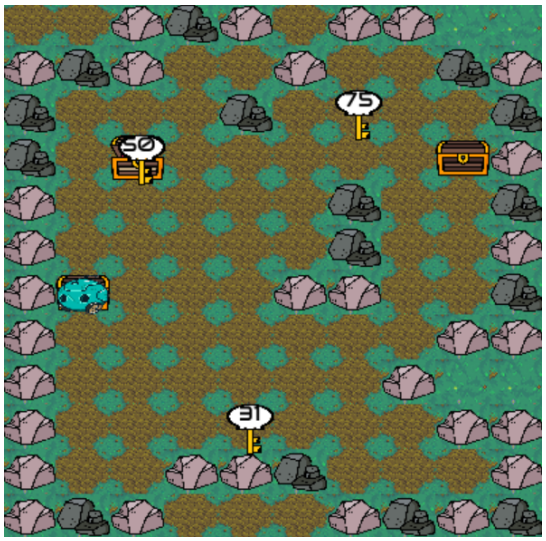
Bảng 5: Kết quả các thuật toán với test case 4

Path

- **DFS:** lullDurrdrrrdddLruululldldLLrrruuulldDrdL
- **BFS:** lddlLLrurruullDDrdLurrurrdrddL
- **UCS:** lddlLLrurruullDDrdLurrurrdrddL
- **A*:** lddlLLrurruullDDrdLurrurrdrddL

3.5 Test case 5

Map



Hình 10: Map test case 5

Output

Algorithm	Steps	Weight	Node	Time (ms)	Memory (MB)
A*	41	411	719	25.18	35.11
BFS	31	411	165670	4665.04	188.04
DFS	55	561	23432	154.82	32.58
UCS	31	411	312209	13083.91	235.74

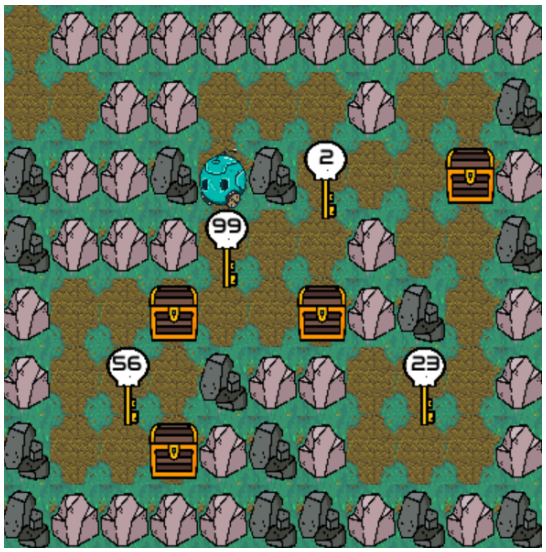
Bảng 6: Kết quả các thuật toán với test case 5

Path

- **DFS:** rrrdllldrrrrrdlLLLdlUUUrrrrrrrululurullluRldrrruLulDldRR
- **BFS:** dddrrrrdLLLdlUUUruuururruRurDldR
- **UCS:** dddrrrrdLLLdlUUUruuururruRurDldR
- **A*:** dddrrrrdLLuuuuuuurruRurDldRddddldddlllLdlUUU

3.6 Test case 6

Map



Hình 11: Map test case 6

Output

Algorithm	Steps	Weight	Node	Time (ms)	Memory (MB)
A*	50	406	1790	70.26	35.13
BFS	42	604	18082	270.53	45.59
DFS	58	604	6446	36.52	30.45
UCS	50	406	10335	99.79	212.88

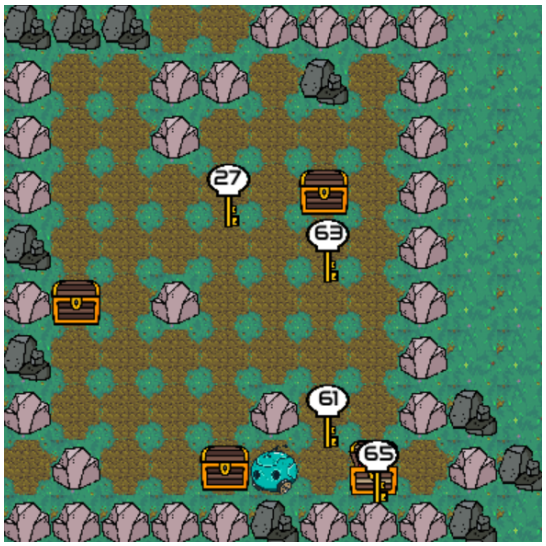
Bảng 7: Kết quả các thuật toán với test case 6

Path

- **DFS:** urrDrrrddddlluRldrrUUUlullDldllllldRlurrrruruulldDrdLLDlluR
- **BFS:** DuurrDrrdrddddlluRdrUUUlullDldLLddlluuRDldR
- **UCS:** urrDDldllllldRurDurrururrdrddddlluRdrUUUlullulldDrdL
- **A*:** urrDrrdrddddlluRdrUUUlullDldllllDldRuurrrruruulldDrdL

3.7 Test case 7

Map



Hình 12: Map test case 7

Output

Algorithm	Steps	Weight	Node	Time (ms)	Memory (MB)
A*	36	503	61112	3507.69	81.50
BFS	24	625	547901	15266.97	667.32
DFS	30	625	34067	200.68	46.80
UCS	24	503	1306356	68163.95	995.89

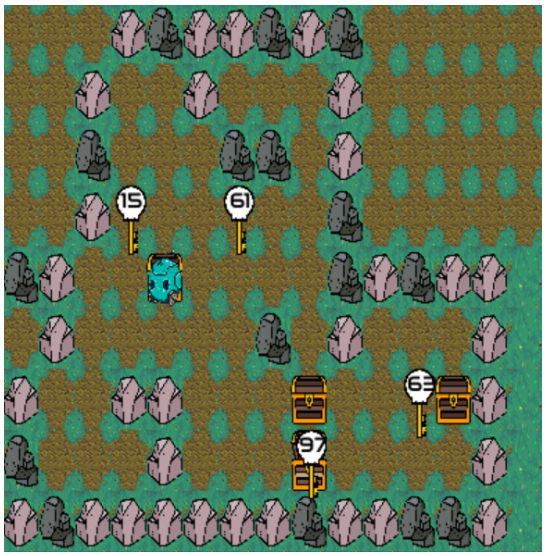
Bảng 8: Kết quả các thuật toán với test case 7

Path

- **DFS:** rUruLLLLLrrrrrulUluulDDDDDlllU
- **BFS:** rUUruLUdLuuLLLulDDrrrDDD
- **UCS:** rUruLLurUluLLLulDDrrrdDD
- **A*:** luuuuruLLLulDDrddddrrrrUruLLurUdllDD

3.8 Test case 8

Map



Hình 13: Map test case 8

Output

Algorithm	Steps	Weight	Node	Time (ms)	Memory (MB)
A*	49	366	7620	221.88	79.52
BFS	34	754	751422	20871.74	840.53
DFS	57	366	44618	334.88	53.80
UCS	37	366	413955	17360.03	685.08

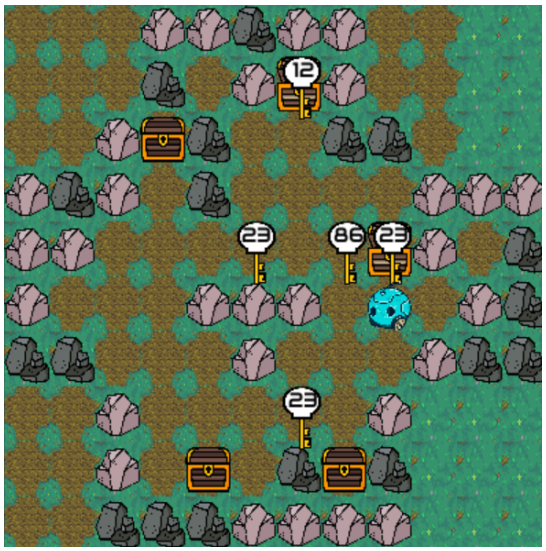
Bảng 9: Kết quả các thuật toán với test case 8

Path

- **DFS:** lldrrrrurrddrrRlllllllullurrrrrrrulLLullDurDlDldRRRurDldRR
- **BFS:** rrruLLulDulDDldRRRurDldRRdRRurRdLL
- **UCS:** drdrrrrrRllluuulLLulDulDDldRRRurDldRR
- **A*:** rrruLdlluulDrddlluRRRRddrrrrRllllluuRurDDuullLulD

3.9 Test case 9

Map



Hình 14: Map test case 9

Output

Algorithm	Steps	Weight	Node	Time (ms)	Memory (MB)
A*	39	551	28011	1203.56	82.36
BFS	39	551	89558	1677.79	148.84
DFS	63	643	74502	694.98	125.85
UCS	39	551	394297	16931.67	561.62

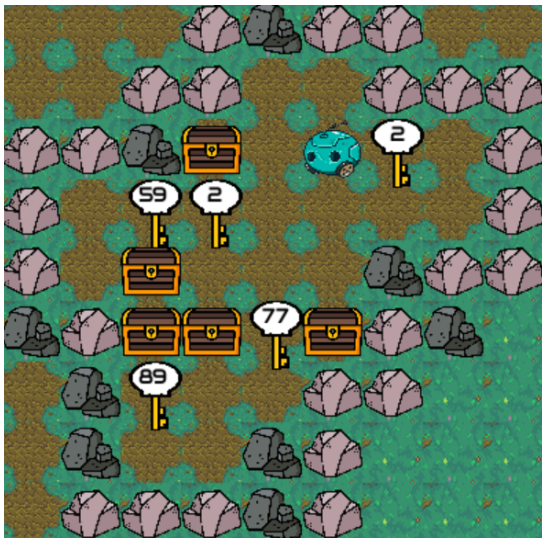
Bảng 10: Kết quả các thuật toán với test case 9

Path

- **DFS:** lddLLdlluRlulururRurrrDulDLLLrrrDDldLdlluulurUUdldrdrDrrurrUldD
- **BFS:** dldLLdlluurDuluurRurrDLLLrrrDDDllluluUU
- **UCS:** dldLLdlluurDuluurRurrDLLLrrrDDDllluluUU
- **A*:** dldLLdlluurDuluurRurrDLLLrrrDDDllluluUU

3.10 Test case 10

Map



Hình 15: Map test case 10

Output

Algorithm	Steps	Weight	Node	Time (ms)	Memory (MB)
A*	40	300	34005	1725.32	84.64
BFS	37	475	2416873	85040.30	3953.88
DFS	53	568	366509	4936.96	456.92
UCS	40	300	780297	46575.22	1135.41

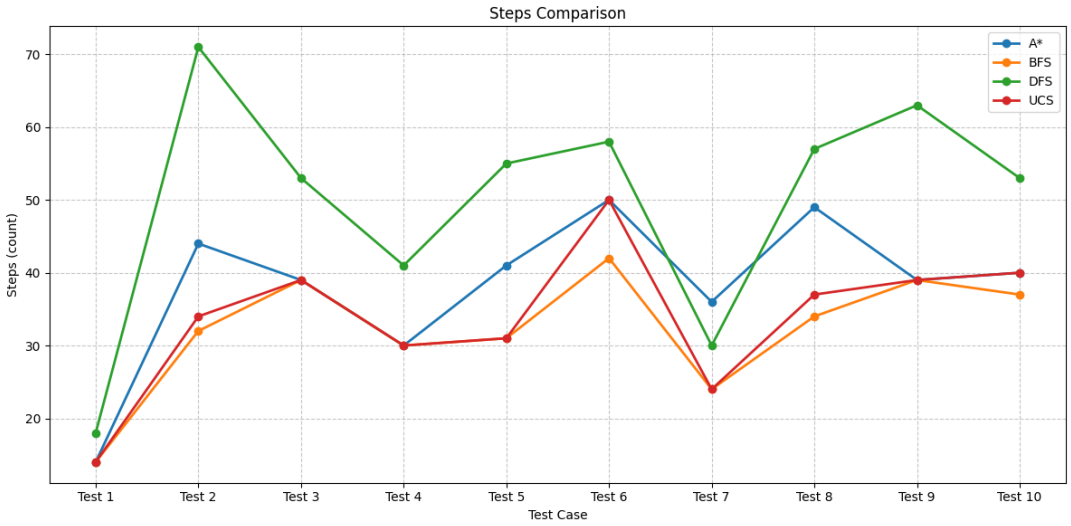
Bảng 11: Kết quả các thuật toán với test case 10

Path

- **DFS:** ldrrruLrdllldlldrdrUlddlUrruruululDDlluRRururDDLrDLLU
- **BFS:** dddLulUdlluRRRurDrruLLulDDrDLLrddldlU
- **UCS:** dddLulUdlluRRRurDrruLLulDDrDLLrddldlU
- **A*:** ddddlddlUruRuuulDDlluRdrUrrrruLLulDDrdLL

3.11 So sánh các thuật toán

- **Steps**



Hình 16: So sánh Steps của 4 thuật toán qua 10 test case

Nhận xét

- Thuật toán A* có nhiều step hơn so với BFS, UCS vì A* cần tối ưu sao cho chi phí đẩy đá ít nhất nên sẽ tốn thêm một số step
- DFS có số bước lớn nhất vì nó duyệt sâu vào một nhánh trước khi quay lại, dẫn đến nhiều bước không cần thiết nếu đích nằm ở các nhánh khác hoặc gần gốc hơn.

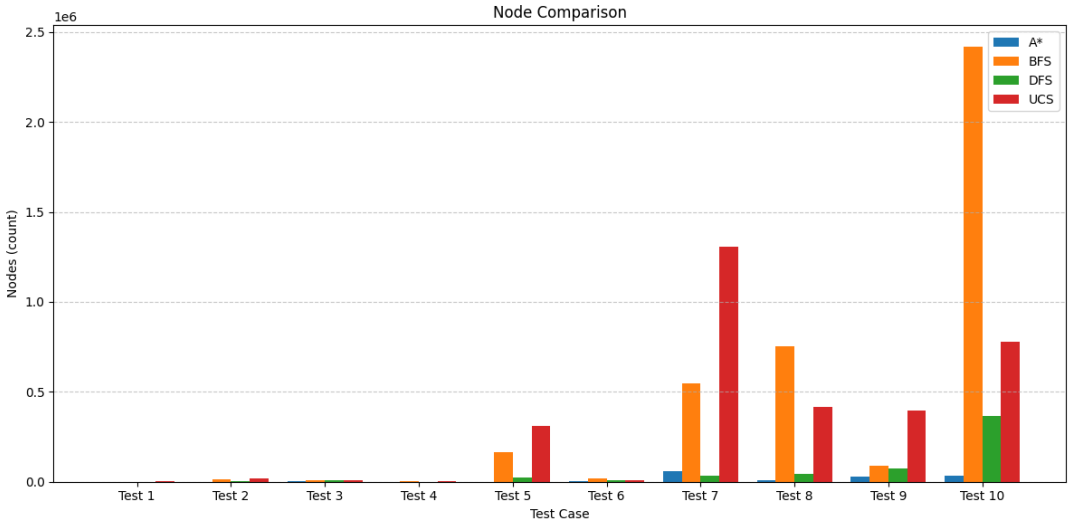
- **Weight**

Hình 17: So sánh Weight của 4 thuật toán qua 10 test case

Nhận xét

- Nhìn chung thuật toán A* và UCS có weight thấp vì chiến A* và UCS dựa trên đánh giá chi phí bé để tối ưu.
- DFS và BFS không tối ưu về chi phí

- **Node**

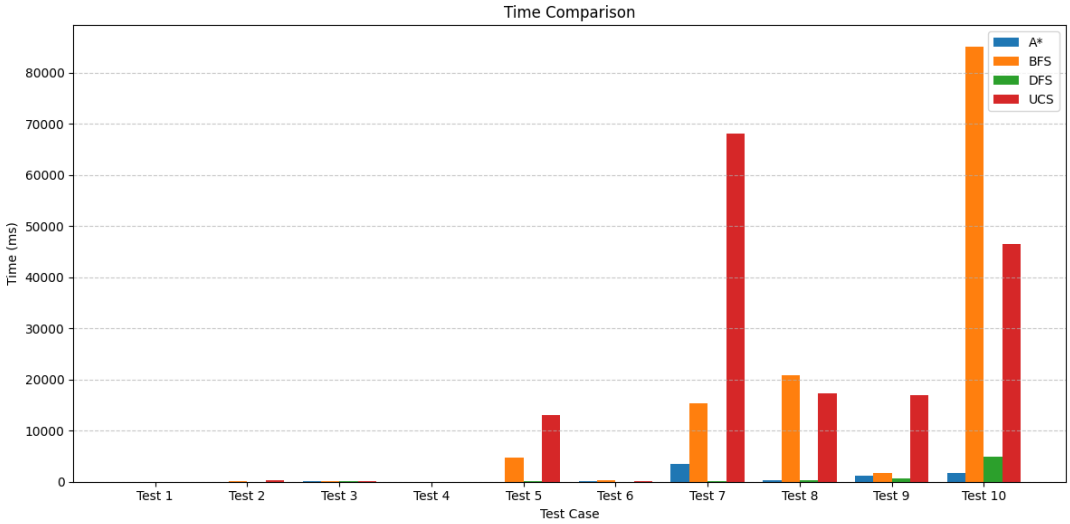


Hình 18: So sánh Node của 4 thuật toán qua 10 test case

Nhận xét

- Trong những test case đầu tiên có số lượng đá ít (1-2) cục, số node duyệt qua của 4 thuật toán tương đối tương đồng nhau.
- Nhưng khi số lượng cục đá tăng lên, UCS và BFS có số node lớn đáng kể so với A* và DFS

- **Time**

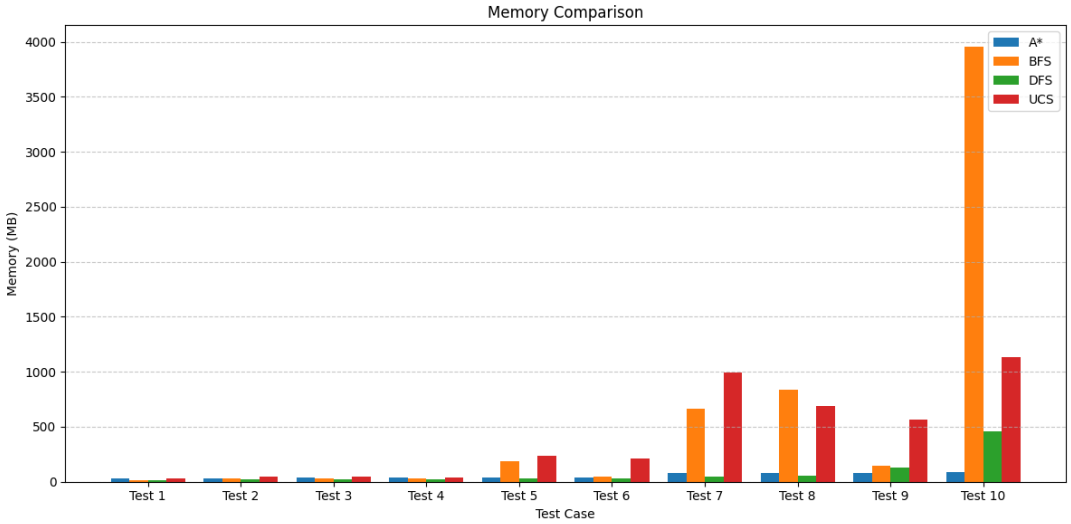


Hình 19: So sánh Time của 4 thuật toán qua 10 test case

Nhận xét

- Thời gian của thuật toán tỉ lệ thuận với số node đã duyệt. Vì vậy sẽ có kết quả tương đồng với biểu đồ Node ở trên
- Thuật toán A* với độ ổn định cao trong cả 10 test case.

- **Memory**



Hình 20: So sánh Memory của 4 thuật toán qua 10 test case

Nhận xét

- Mỗi lần duyệt qua 1 node, chương trình sẽ lưu state node đó vào bộ nhớ => Tỉ lệ thuận với số node.
- Ở test case 10 với số đá là 5, BFS mất nhiều thời gian, bộ nhớ để tìm được trạng thái đích.

3.11.1 Nhận xét chung

Dựa vào 5 biểu đồ so sánh hiệu năng của 4 thuật toán (A*, BFS, DFS, UCS) qua 10 test case với số lượng cục đá tăng dần, chúng ta rút ra các nhận xét sau:

1. **Hiệu quả tìm kiếm và số bước:**
   - Thuật toán DFS có xu hướng thực hiện ít bước hơn so với các thuật toán khác, do chỉ duyệt sâu vào một nhánh mà không quay lại cho đến khi gặp ngõ cụt. Tuy nhiên, vì DFS không tối ưu trọng số nên không phải lúc nào cũng cho ra kết quả tốt nhất về quãng đường.
   - A* và UCS thường yêu cầu số bước nhiều hơn DFS nhưng ít hơn BFS. Cả hai đều tìm kiếm theo hướng tối ưu trọng số, với A* sử dụng heuristic để giảm số bước, giúp tiết kiệm tài nguyên.
2. **Tối ưu trọng số:**
   - UCS và A* nổi bật về khả năng tìm kiếm đường đi có trọng số tối ưu. A* có lợi thế khi sử dụng heuristic, giúp giảm số lượng node và các bước cần mở rộng so với UCS.
   - DFS và BFS không có cơ chế tối ưu trọng số, nên kết quả có thể không ngắn nhất về khoảng cách trong nhiều test case.
3. **Khả năng mở rộng node:**
   - BFS và UCS thường mở rộng nhiều node nhất, đặc biệt là ở các test case có số lượng cục đá cao. Điều này dẫn đến việc tiêu tốn nhiều tài nguyên, đặc biệt là bộ nhớ.
   - DFS có khả năng mở rộng node ít hơn, nhờ vào việc chỉ duyệt một nhánh đến khi gặp đích hoặc ngõ cụt.
   - A* cân bằng giữa việc mở rộng node và trọng số, chỉ mở rộng những node có khả năng dẫn đến đích nhanh hơn.
4. **Thời gian xử lý:**
   - DFS tỏ ra vượt trội về thời gian xử lý trong các test case phức tạp, nhờ vào cách tiếp cận đơn giản và không cần lưu trữ nhiều trạng thái.
   - A* có thời gian xử lý thấp hơn UCS và BFS nhờ vào heuristic. Tuy nhiên, thời gian của A* có thể tăng lên trong các bài toán phức tạp, nhưng vẫn hiệu quả hơn so với

UCS.

- UCS và BFS tiêu tốn thời gian nhiều nhất, do phải duyệt qua nhiều trạng thái và node trước khi đạt đến đích.

5. **Sử dụng bộ nhớ:**
   - DFS sử dụng bộ nhớ ít nhất trong các thuật toán, do chỉ lưu trữ các node trong nhánh hiện tại.
   - BFS và UCS có yêu cầu bộ nhớ lớn nhất, do phải lưu trữ toàn bộ các node trong hàng đợi mở rộng. Điều này gây ra vấn đề khi không gian tìm kiếm lớn.
   - A* nằm giữa DFS và BFS/UCS về sử dụng bộ nhớ, với khả năng tiết kiệm bộ nhớ tốt hơn BFS/UCS nhờ vào heuristic.

Tổng kết chung:

- DFS là thuật toán hiệu quả khi yêu cầu tốc độ nhanh và tài nguyên hạn chế, nhưng không phù hợp khi cần tối ưu trọng số.
- BFS phù hợp khi cần duyệt toàn bộ không gian tìm kiếm, nhưng gặp hạn chế về bộ nhớ và thời gian khi không gian quá lớn.
- UCS và A* tối ưu hơn về trọng số, với A* tỏ ra ưu thế hơn trong việc tiết kiệm bộ nhớ và thời gian nhờ heuristic. Trong các trường hợp yêu cầu độ chính xác cao và tối ưu hóa tài nguyên, A* là lựa chọn tốt nhất.

Qua các biểu đồ, có thể thấy rằng A* thường là giải pháp tối ưu về hiệu năng khi cần cân bằng giữa thời gian, số bước, bộ nhớ, và khả năng tối ưu trọng số, trong khi DFS là giải pháp tốt nhất về tài nguyên khi không cần đến trọng số.

4 Tài liệu tham khảo

1. Implementation of A* Algorithm for Solving Sokoban Logic Games
https://pdfs.semanticscholar.org/4eb4/9b35e6071a2274301ad5ea95b458446ffdd0.pdf

2. Graph search algorithm to solve Sokoban
https://sokoban.dk/wp-content/uploads/2016/02/Timo-Virkkala-Solving-Sokoban-Master
pdf
3. UCS algorithm
https://courses.grainger.illinois.edu/cs440/fa2021/lectures/search4.html
4. Depth-first iterative-deepening: An optimal admissible tree search
https://www.sciencedirect.com/science/article/abs/pii/0004370285900840
5. Pygame documentation
https://www.pygame.org/docs/