Trường Đại học Khoa học tự nhiên – Khoa Công nghệ thông tin.

Đồ án thực hành 01

Cơ sở trí tuệ nhân tạo – Fundamentals of Artificial Intelligence.

22120369_22120400_22120410_22120412
Tháng 10, 2024.

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN THỰC HÀNH SỐ 01

Bộ môn: Cơ sở trí tuệ nhân tạo.

Tên đề tài: *"Ares's Adventure".*

Tên nhóm: 22120369_22120400_22120410_22120412.

Thành viên:

1. 22120369 – Quan Phan Tiến.
2. 22120400 – Trần Anh Tú.
3. 22120410 – Dương Hữu Tường.
4. 22120412 – Nguyễn Anh Tường.

Thông tin chung:

1. **Bộ môn:** Cơ sở trí tuệ nhân tạo.

2. **Giảng viên lý thuyết:** Thầy Bùi Duy Đăng.

3. **Giảng viên thực hành:** Thầy Nguyễn Thanh Tình.

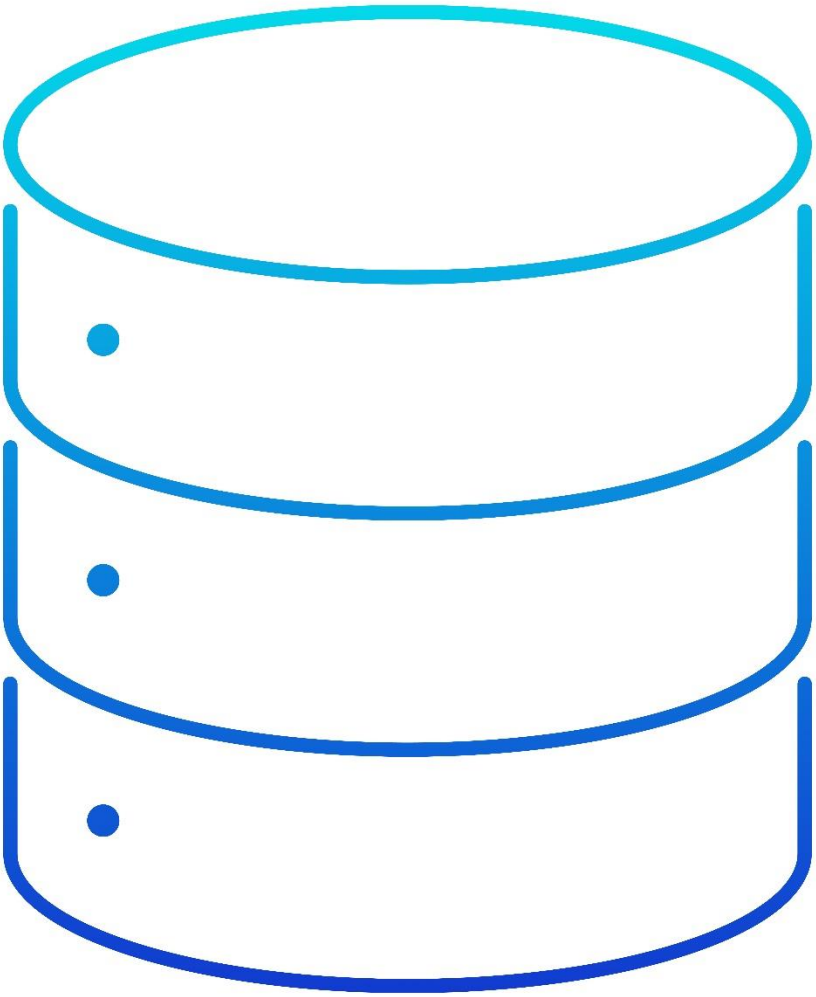
4. **Mã lớp:** 22_4.

5. **Tên nhóm:** 22120369_22120400_22120410_22120412.

6. **Danh sách thành viên:**
    a. 22120369 – Quan Phan Tiến.
    b. 22120400 – Trần Anh Tú.
    c. 22120410 – Dương Hữu Tường.
    d. 22120412 – Nguyễn Anh Tường.

MỤC LỤC

Section 1: *Work Assignments*



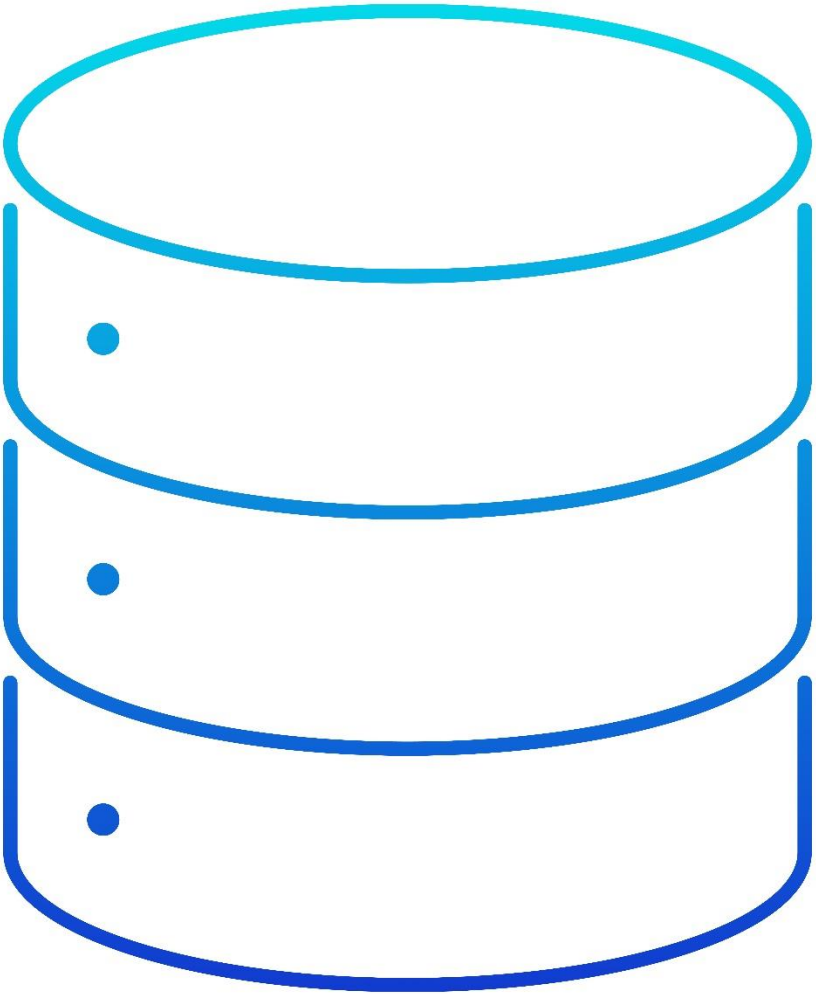
Bảng phân công công việc

Mã số	Họ tên	Công việc	Tiến độ
22120412	Nguyễn Anh Tường	BFS	100%
22120400	Trần Anh Tú	DFS	100%
22120410	Dương Hữu Tường	UCS	100%
22120369	Quan Phan Tiến	A*	100%
22120400	Trần Anh Tú	Quay video demo.	100%
22120412	Nguyễn Anh Tường	Edit video demo.	100%
---	Cả 4 người	Làm report (tự viết phần của mình).	100%
---	Cả 4 người	Làm testcase (10 testcase / 4 người).	100%
22120369	Quan Phan Tiến	Thống kê thuật toán	100%
22120410	Dương Hữu Tường	GUI trò chơi	100%
22120412	Nguyễn Anh Tường	GUI menu	100%
22120400	Trần Anh Tú	Optimize source code	100%

Đánh giá mức độ hoàn thành đồ án

100%.

Section 2: *BFS Algorithms.*



I. Thành viên thực hiện:

- 22120412 – Nguyễn Anh Tường.

II. Trình bày, mô tả giải thuật:

Bước 1: Khởi tạo trạng thái ban đầu, frontier, visited và node counter.

Bước 2: Thêm vào frontier trạng thái ban đầu, khởi tạo đường đi, khối lượng và ma trận hiện tại.

Tức là frontier sẽ chứa: (Vị trí của Ares hiện tại | Vị trí của các tảng đá, khối lượng tương ứng | Khối lượng mà Ares đã đẩy | Trạng thái của ma trận).

Bước 3: Sử dụng vòng lặp while, điều kiện dừng là khi frontier trống.

Bước 3.1: Lấy ra các element từ frontier. Gồm:

- Vị trí của Ares hiện tại.
- Vị trí của các hòn đá cùng với khối lượng tương ứng.
- Đường đi hiện tại đã đi được.
- Khối lượng đá Ares đã đẩy.
- Trạng thái của ma trận hiện tại.

Bước 3.2: Kiểm tra xem tất cả tảng đá đã nằm trên các switch hay chưa?

Bước 3.2.1: Nếu đã đã vào đúng vị trí thì sẽ dừng chương trình và trả về các giá trị sau:

- Đường đi.
- Số node.
- Khối lượng đá đã đẩy.
- Trạng thái của ma trận hiện tại.

Bước 3.2.2: Nếu đá vẫn chưa vào đúng hết vị trí thì chạy tiếp chương trình.

Bước 3.3: Ares bước đi theo nhiều hướng. Với mỗi hướng di chuyển thì ta sẽ có một trạng thái mới được sinh ra. Do vậy với mỗi bước di chuyển ta thực hiện các hành động sau:

Bước 3.3.1: Kiểm tra xem Ares có thể di chuyển ở bước đi này không. Điều kiện thì vị trí Ares đi tiếp theo không phải tường và không phải một trong các vị trí của tảng đá.

- Nếu có thể đi => Tạo một state mới gồm:
  - Trạng thái ma trận mới, cập nhật lại kí tự vị trí của Ares trên ma trận đó.
  - Kiểm tra xem state đó đã tồn tại trong **visited** hay chưa. Nếu chưa thì ta sẽ thêm state đó vào trong frontier cùng với việc **cập nhật**

đường đi, khối lượng và ma trận mới. Cập nhật vào visited và tăng node counter lên 1 đơn vị.

Bước 3.3.2: Nếu vị trí tiếp theo của Ares là tảng đá thì tức nghĩa là Ares đang cố gắng đẩy tảng đá đó.

⇨ Nếu tảng đá có thể được di chuyển (không có vật cản, không bị kẹt, …) thì ta sẽ thực hiện tiếp các bước sau:

Bước 3.3.2.1: Tạo một trạng thái mới. (clone trạng thái hiện tại).

Bước 3.3.2.2: Xóa vị trí của tảng đá cũ trong trạng thái mới, cập nhật lại vị trí mới của tảng đá trong trạng thái mới.

Bước 3.3.2.3: Nếu tảng đá đang ở trong một switch nào đó thì tức nghĩa là Ares đang cố gắng đẩy tảng đá ra khỏi switch.

Lúc này ta cập nhật lại trạng thái của cả tảng đá và cả switch (đổi kí tự '*' thành '.' trong ma trận được copy).

Bước 3.3.2.4: Nếu tảng đá được đẩy vào một switch thì cập nhật lại vị trí của tảng đá và cả switch (đổi kí tự '.' thành kí tự '*' trong ma trận được copy).

Bước 3.3.2.5: Cập nhật lại vị trí của Ares trong trạng thái mới này.

Bước 3.3.2.6: Cập nhật lại tổng khối lượng đá đã đẩy của Ares.

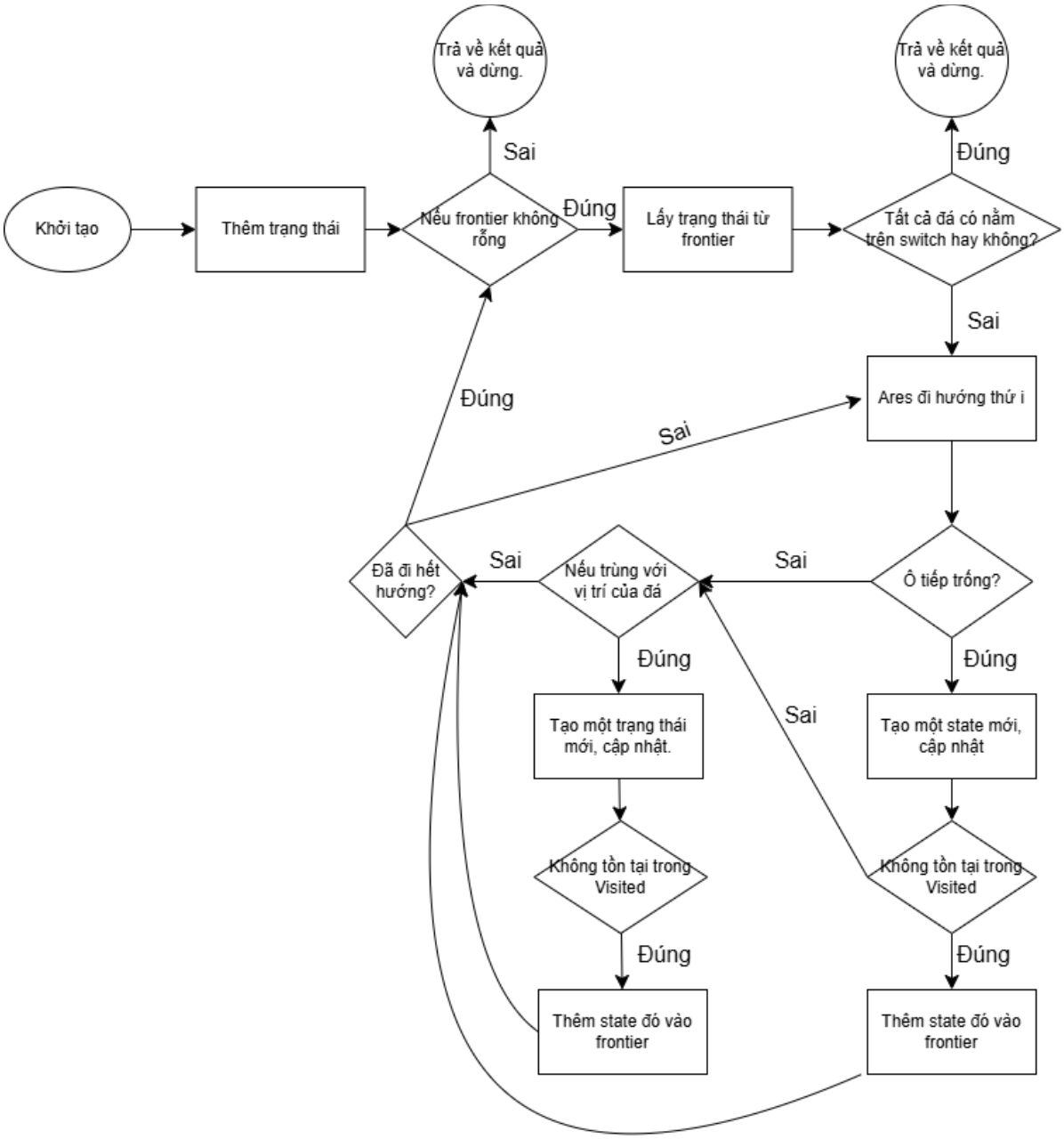
Bước 3.3.2.7: Kiểm tra xem trạng thái được clone và chỉnh sửa từ nãy đến giờ đã tồn tại trong visited hay chưa?

Nếu chưa thì ta sẽ thêm trạng thái này vào visited, thêm trạng thái này cùng với việc cập nhật đường đi của Ares vào trong frontier và tăng node counter lên 1 đơn vị.

Bước 4: Lặp lại quá trình ở bước 3 cho đến khi frontier rỗng.

Bước 5: Nếu như ở bước 3 không trả về kết quả thì tức là không có đường đi, tại đây ta trả về kết quả **None** cho đường đi, **0** cho số node được tạo ra, **0** cho tổng khối lượng đá và **None** cho ma trận kết quả.

III. Mô tả thuật toán bằng flowchart:



IV. Mô tả testcase và kết quả kì vọng:

Trong đó chấm xanh là Ares và chấm đỏ là tản đá, khi đá nằm trên switch thì sẽ chuyển thành màu vàng, ô xanh lá là switch.

Lưu ý: Đây chỉ là hình ảnh minh họa vị trí của các element và hình dạng của ma trận. Khi chạy thực tế hình ảnh sẽ đẹp hơn.

1. Test case 01, 02, 03, 04 – mức độ dễ:

Đây là những test case chỉ có 1 tản đá, vị trí của switch cũng không quá lắc léo.

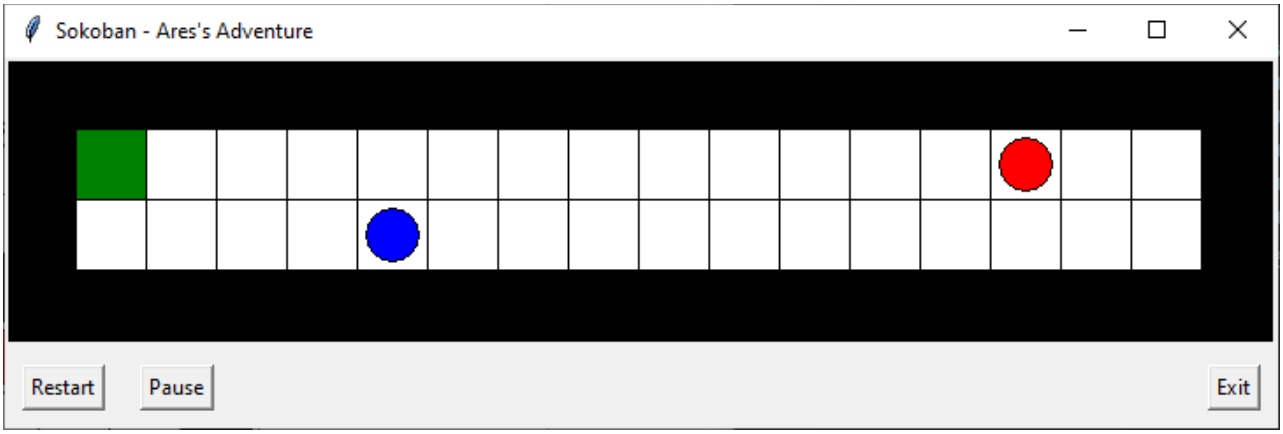


Figure 1. Testcase 01.

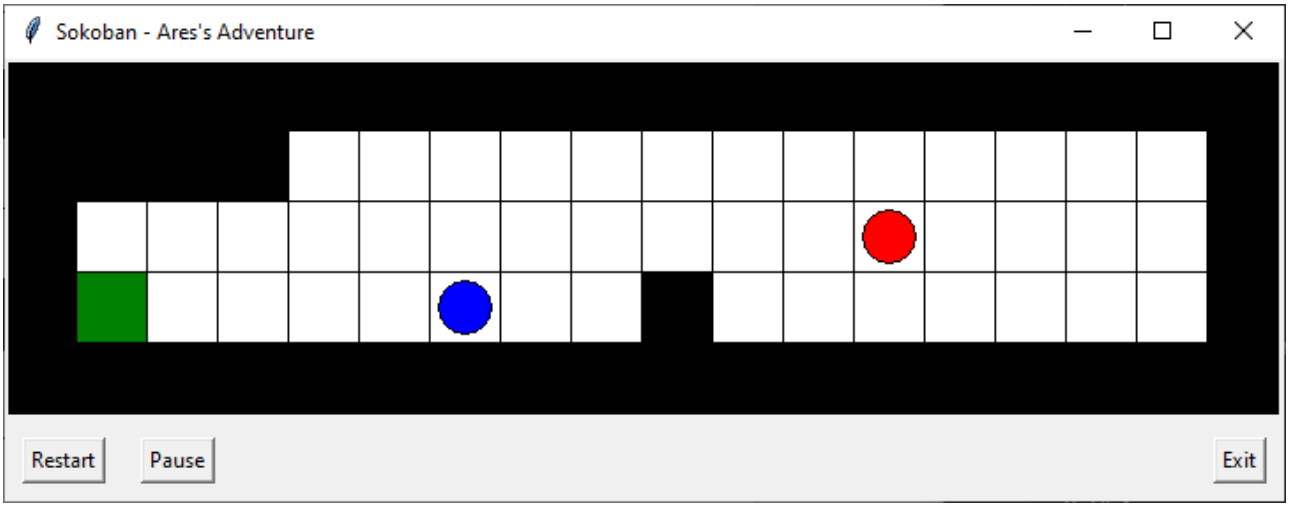


Figure 2. Testcase 02.

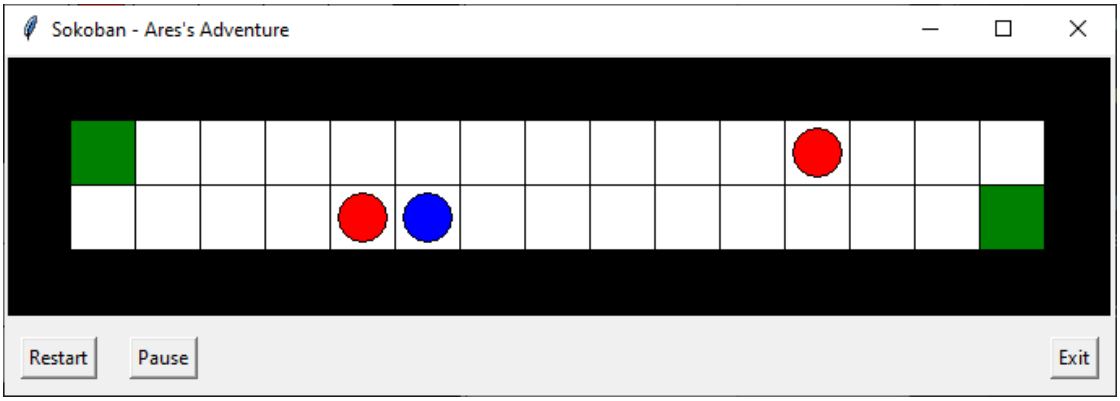


Figure 3. Testcase 03.

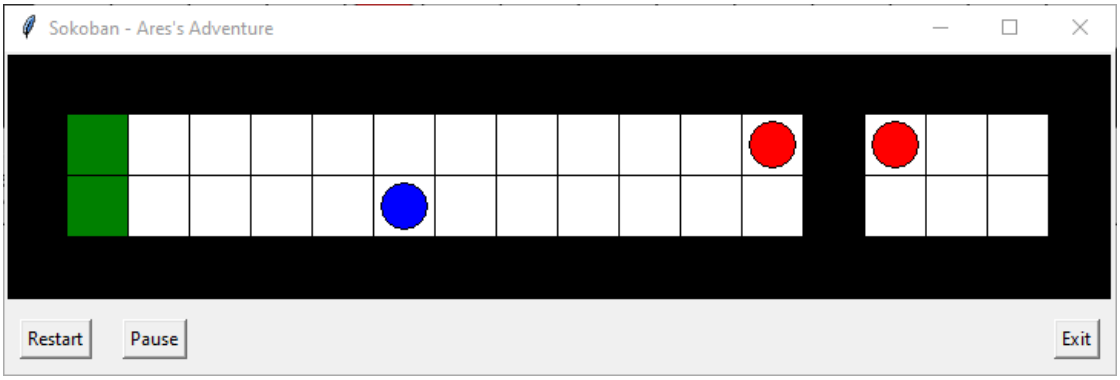


Figure 4. Testcase 04.

Case 4 là trường hợp không thể giải (deadlock).

2. Test case 05, 06, 07 – mức độ trung bình:

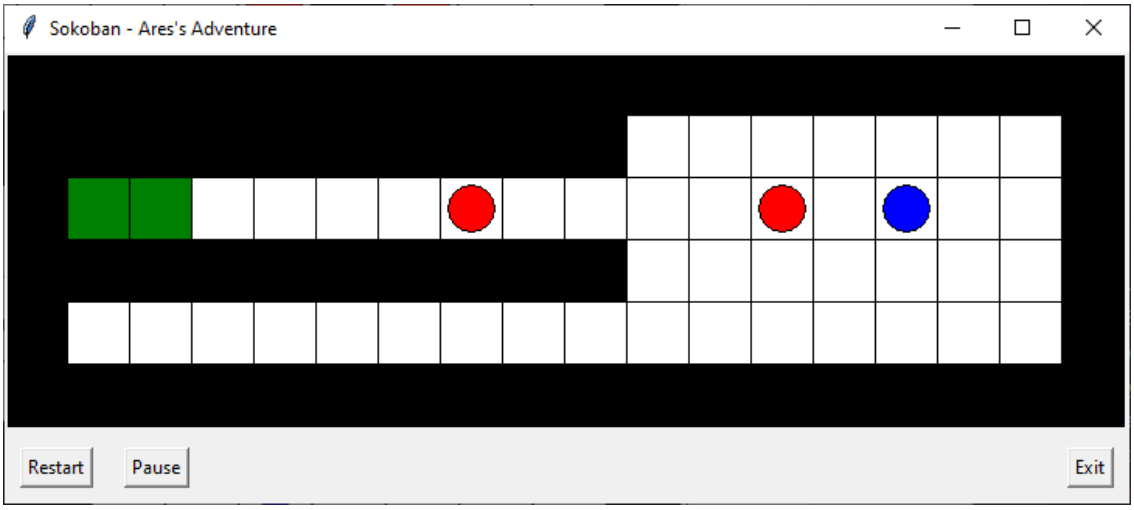


Figure 5. Testcase 05.

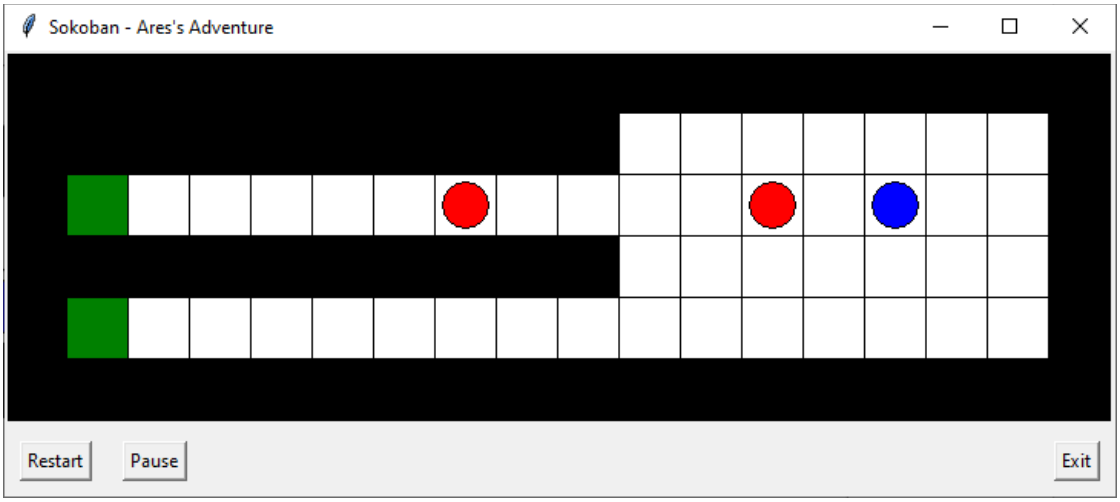


Figure 6. Testcase 06.

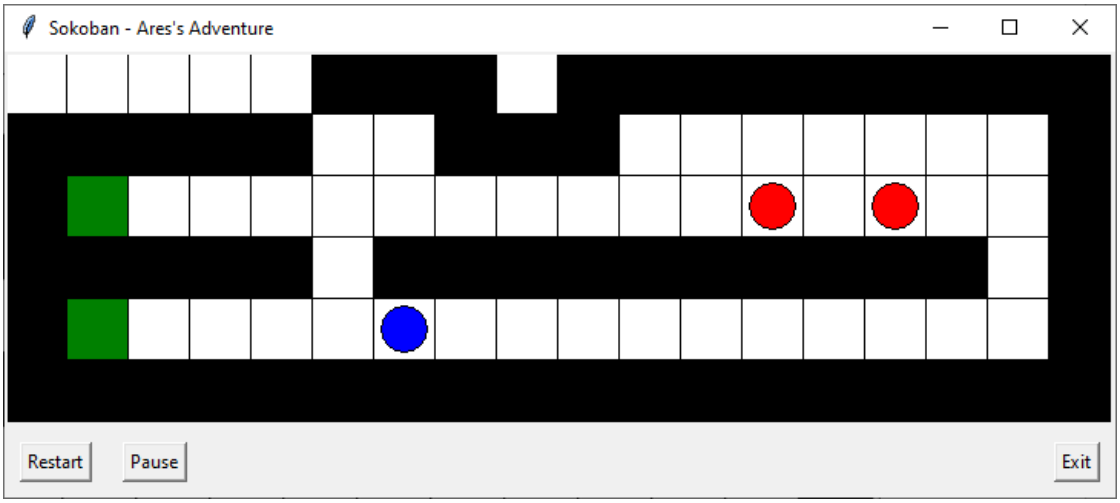


Figure 7. Testcase 07.

3. Test case 08, 09, 10 – mức độ khó:

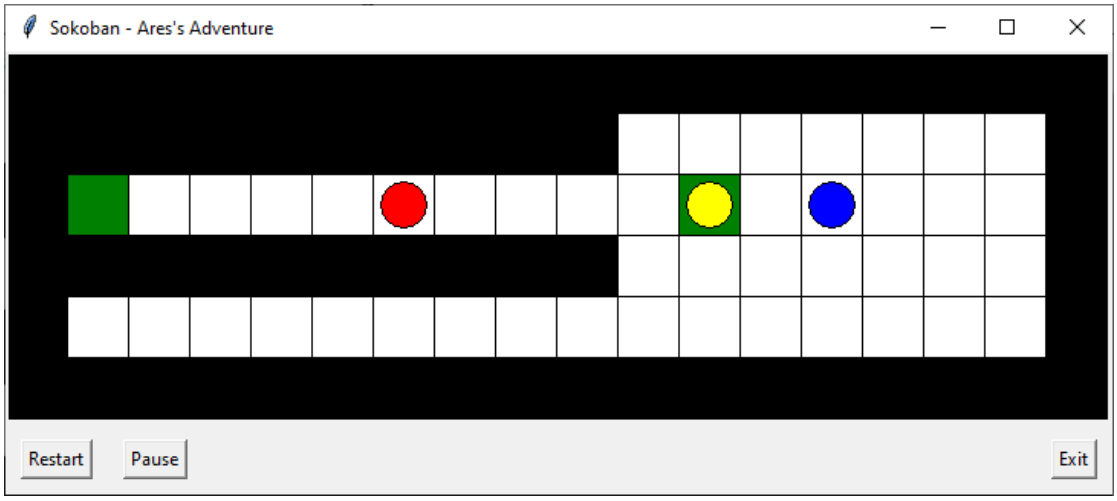


Figure 8. Testcase 08.

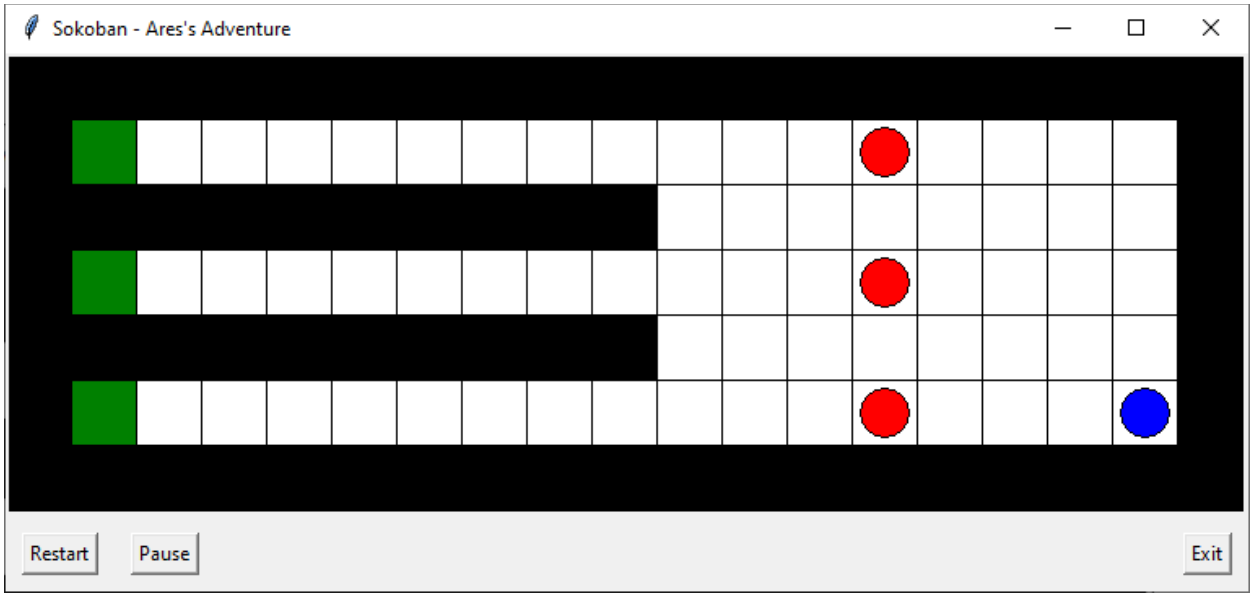


Figure 9. Testcase 09.

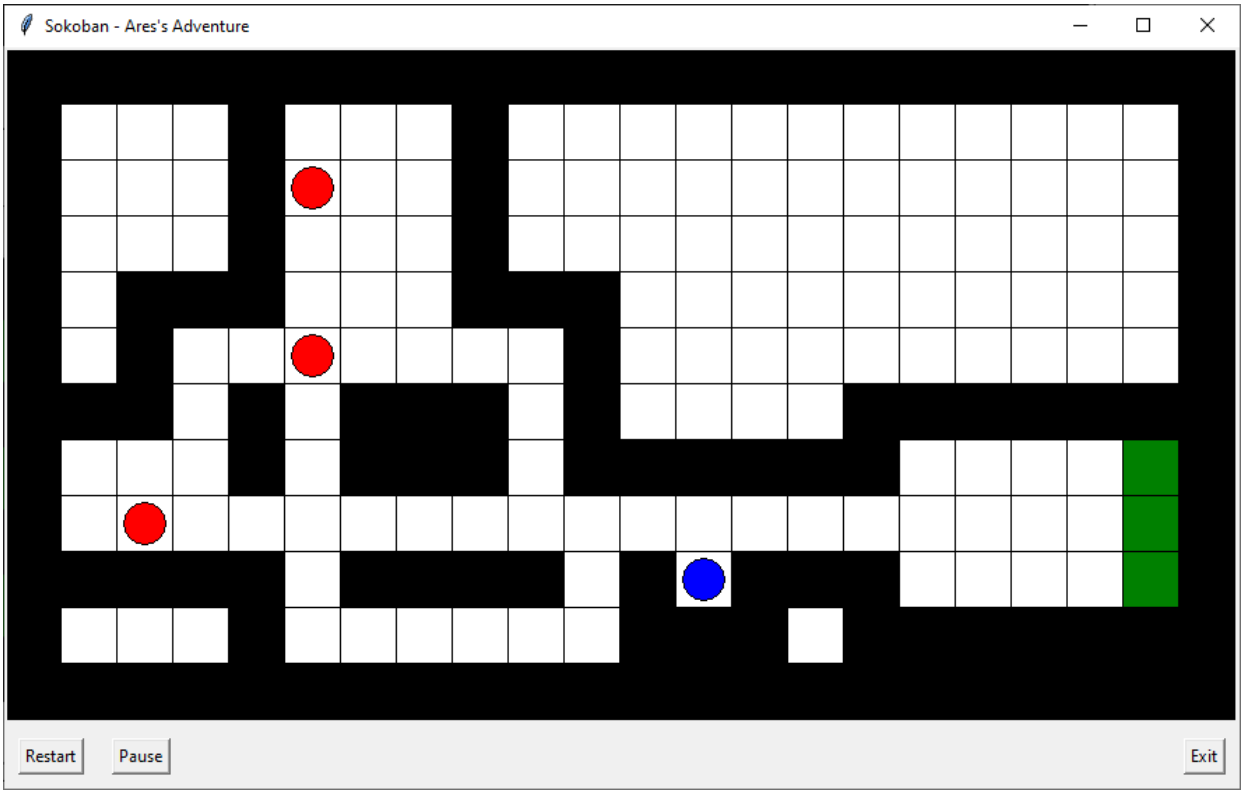
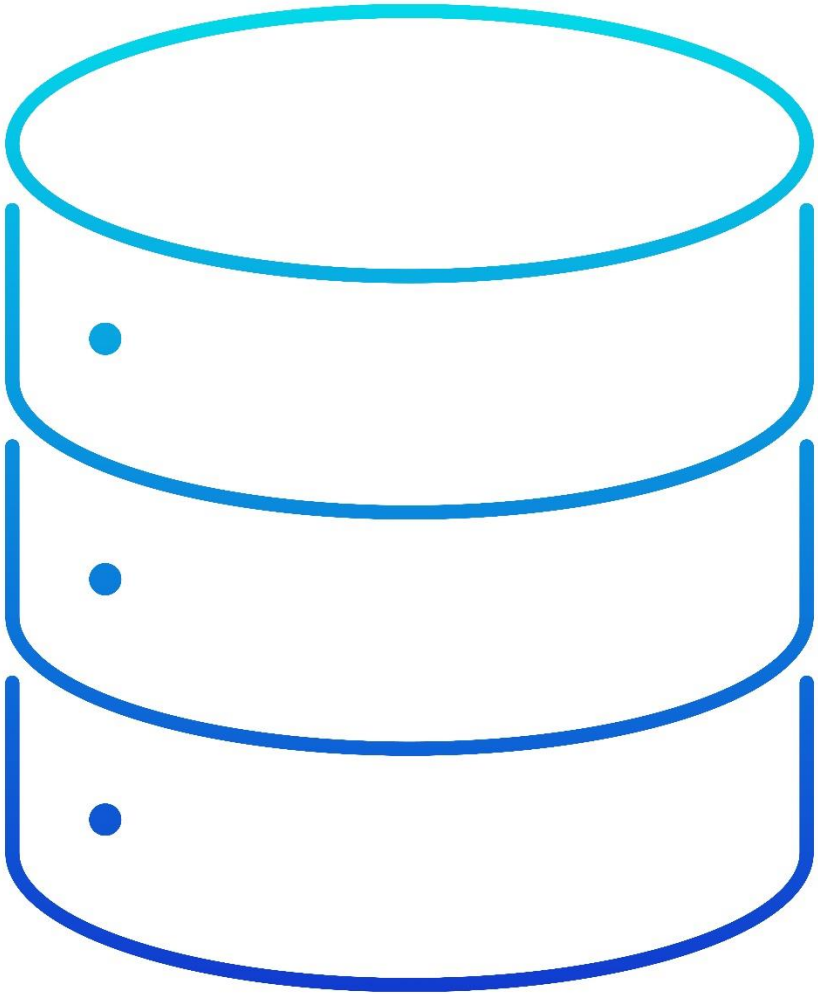


Figure 10. Testcase 10.

Section 3: *DFS Algorithms.*



I. Thành viên thực hiện.

- 22120400 - Trần Anh Tú.

II. Trình bày, mô tả giải thuật.

Bước 1: Xây dựng bài toán.

- State của bài toán được xác định là mảng 2 chiều chứa tất cả các thông tin về vị trí của stones, ares, switches.
- Initial state: chính là mảng 2 chiều ban đầu của file input.
- Goal state: chính là trạng thái mà ở đó các viên đá đã nằm ở vị trị switch.
- Actions: di chuyển trái, phải, trên, dưới nếu có thể.
- Transition model: trả về hành động và state sau khi thực hiện hành động.
- Goal test: kiểm tra xem các viên đá đã nằm ở đúng vị trí các switch chưa.
- Path cost: mỗi bước di chuyển là 1 path cost cộng thêm số cân nặng của đá (nếu có).

Bước 2: Cài đặt class state, node, frontier cho bài toán.

- State là state của bài toán bên cạnh đó còn chứa thêm thông tin về vị trị ares và stones nhằm mục đích tính toán.
- Node là class chứa state, action (hành động để đến được state) và parent (node trước của state).
- Frontier là một stack chứa các node.

Bước 3: Cài đặt class solution chứa các thông tin cần thiết và thuật toán.

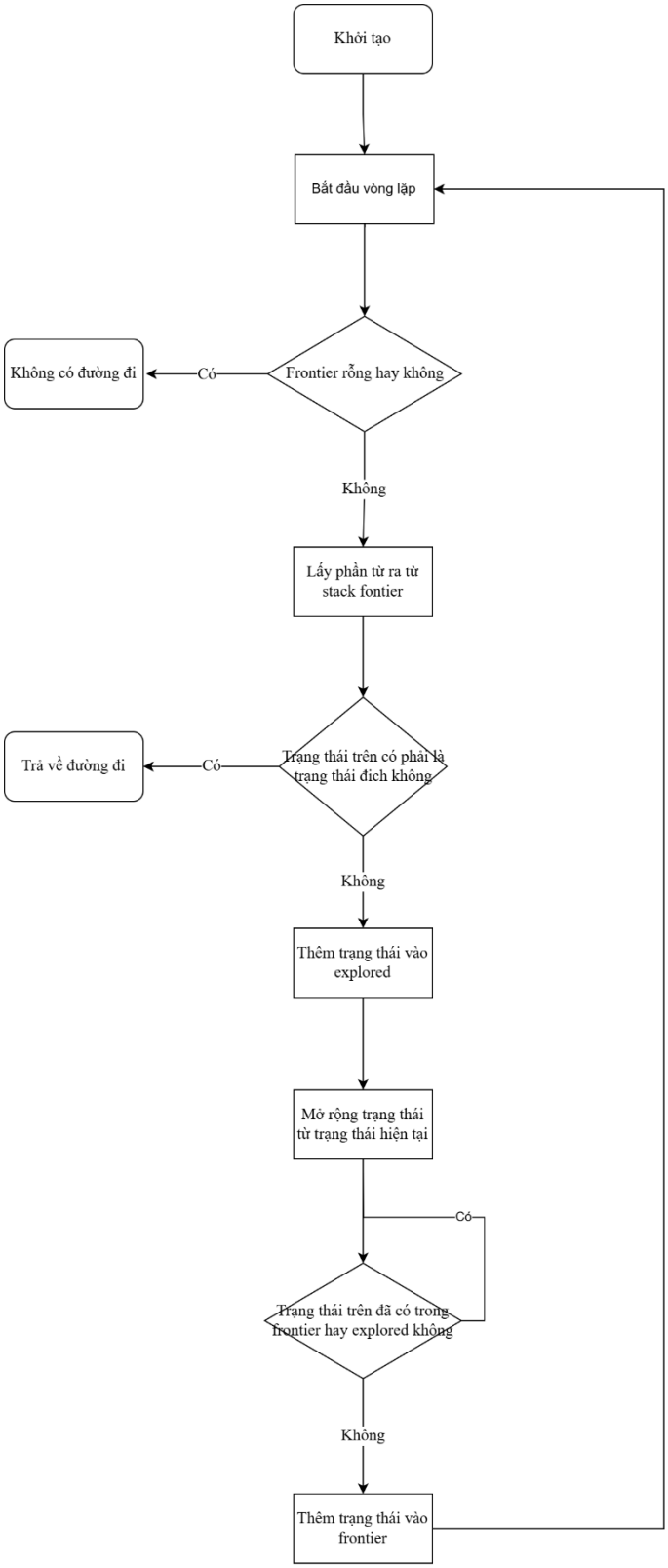
- Cài đặt hàm khởi tạo cho phép đọc dữ liệu từ file input sau đó lưu một mảng chứa stone weight và một mảng 2 chiều chứa thông tin của màn chơi.
- Tiếp đó tiến hành khởi tạo initital state (start state) và goal state.
- Cài đặt hàm actions: hàm có tác dụng nhận vào 1 state và trả về các action có thể thực hiện được và state sau khi thực hiện action dó đó.

Bước 4: Cài đặt thuật toán DFS.

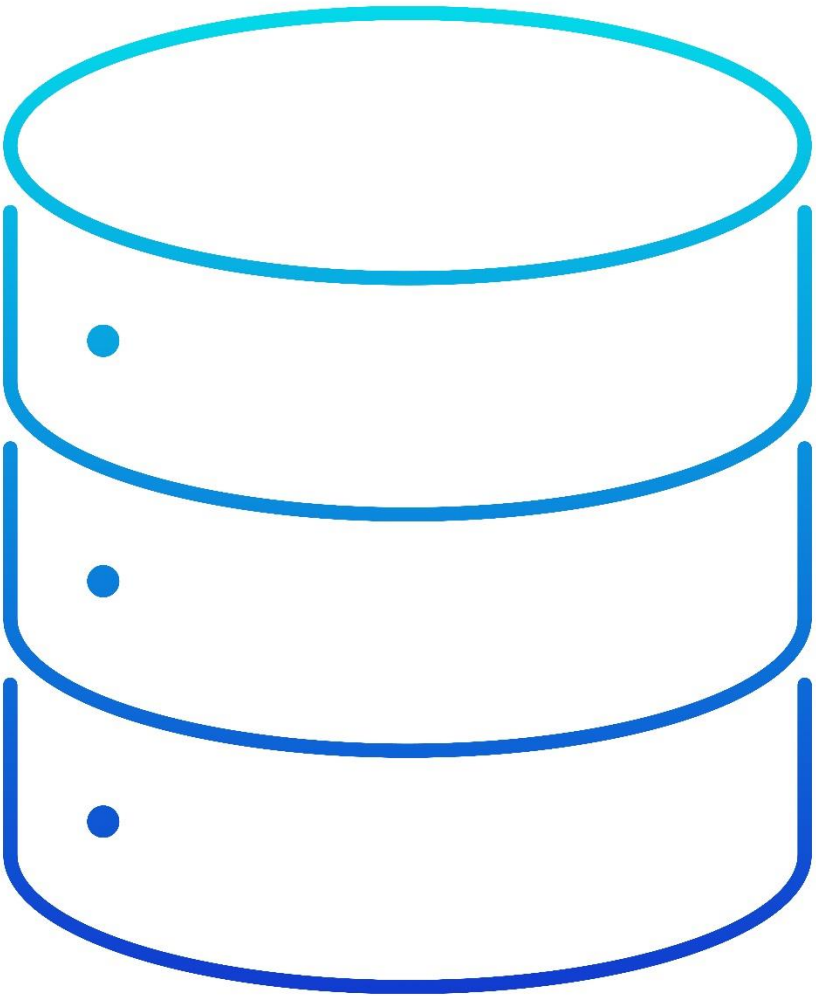
- Khởi tạo một frontier là một stack chứa các node. Khởi tạo explored là một set lưu các node đã được explored. Khởi tạo start node thông qua initial state đã tạo từ trước.
- Bắt đầu giải thuật: ý tưởng chính của thuật toán DFS trong bài toán này chính là lấy một node từ stack frontiter tiến hành kiểm tra xem node đó có phải là node chứa goal state không, sau đó tiến hành mở rộng ra bằng cách sử dụng hàm actions (trả về các bước di chuyển hợp lệ), sau đó thêm các state nhận được từ hàm actions vào stack frontier lặp lại quá trình trên. Bởi vì frontier là một stack nên các node sẽ được mở rộng theo chiều sâu.
- Cài đặt thuật toán:
    - 1. Kiểm tra xem frontier có trống hay không nếu trống thì không tìm thấy solution.
    - 2. Lấy node ra khỏi stack frontier thông qua phương thức pop().

- 3. Kiểm tra xem node trên có phải goal không, nếu phải thì kết thúc thuật toán và trả về đường đi còn nếu không thì tiếp tục thuật toán.
- 4. Thêm state của node hiện tại vào explored.
- 5. Mở rộng state bằng cách sử dụng hàm actions sau đó thêm các state vừa mở rộng được vào frontier.
- 6. Tiếp tục lặp lại quá trình trên.

III. Mô tả thuật toán bằng flowchart.



Section 4: *UCS Algorithms.*



I. Thành viên thực hiện.

- 22120410 - Dương Hữu Tường.

II. Trình bày, mô tả giải thuật.

1. Khởi tạo.

Bước 1: Trong bước khởi tạo của thuật toán, ta sẽ tạo một trạng thái ban đầu (***initial_state***) của trò chơi với các giá trị sau:

- *Tổng Chi Phí Ban Đầu*: Đặt tổng chi phí ban đầu bằng 0, vì chưa có nước đi nào được thực hiện.
- *Vị Trí Ban Đầu Của Ares*: Lưu trữ vị trí ban đầu của Ares từ biến ***self.initial_ares***.
- *Vị Trí Ban Đầu Của Các Tảng Đá*: Lưu trữ vị trí ban đầu của các tảng đá từ biến ***self.initial_stones*** dưới dạng tuple để đảm bảo tính bất biến, không xảy ra những thay đổi không mong muốn trong quá trình thực hiện thuật toán.
- *Chuỗi Di Chuyển:* Khởi tạo chuỗi này là một chuỗi rỗng, dùng để lưu lại lịch sử các bước di chuyển của Ares từ trạng thái ban đầu.
- *Khối Lượng Đá Đẩy Được*: Bắt đầu với khối lượng đá đã đẩy bằng 0, vì Ares chưa thực hiện bất kỳ hành động nào.

Bước 2: Khởi tạo ***frontier*** dùng để lưu trữ các nút cần được khám phá tiếp theo trong thuật toán dưới dạng một hàng đợi ưu tiên, sau đó thêm trạng thái ban đầu (***initial_state***) vào hàng đợi.

Bước 3: Khởi tạo ***frontier_nodes*** dưới dạng từ điển (dictionary) giúp theo dõi các trạng thái (nút) đã thêm vào hàng đợi ***frontier*** cùng với chi phí tương ứng. Mỗi trạng thái được lưu trữ dưới dạng cặp khóa-giá trị, trong đó khóa là vị trí của Ares và các tảng đá, còn giá trị là chi phí để đạt được trạng thái đó. Giúp nhanh chóng kiểm tra xem trạng thái đã tồn tại trong hàng đợi chưa và so sánh chi phí của trạng thái đó với chi phí hiện tại để quyết định xem có nên cập nhật hay không. Ban đầu frontier_nodes chứa khóa là vị trí ban đầu của Ares và vị trí ban đầu của các tảng đá, giá trị để đạt được trạng thái ban đầu là 0.

Bước 4: Khởi tạo một ***explored set*** để đánh dấu các nút đã được duyệt.

2. Vòng lặp để tìm kiếm.

Bước 1: Trong mỗi vòng lặp, thuật toán UCS sẽ lấy ra trạng thái có chi phí thấp nhất trong ***frontier***. Sau đó:

- Kiểm tra xem trạng thái này có trong ***frontier_nodes*** không, nếu có thì xóa nó để đồng bộ với việc pop trạng thái khỏi ***frontier***. Khi thuật toán khám phá một trạng thái mới với chi phí thấp hơn, cần phải cập nhật chi phí của trạng thái

đó. Việc xóa trạng thái cũ khỏi ***frontier_nodes*** là cần thiết để đảm bảo rằng trạng thái mới với chi phí cập nhật có thể được thêm vào hàng đợi ***frontier***.

- Kiểm tra xem trạng thái này có phải là trạng thái đích không (tức là tất cả các tảng đá đã được đặt đúng vị trí trên các công tắc). Nếu đúng, trả về các giá trị bao gồm: đường đi, thời gian chạy, số lượng nút đã mở rộng, bộ nhớ tiêu thụ và khối lượng đá đã đẩy.
- Nếu nút chưa phải trạng thái đích, thêm nó vào ***explore set.***

Bước 2: Phát sinh các trạng thái con của trạng thái hiện tại:

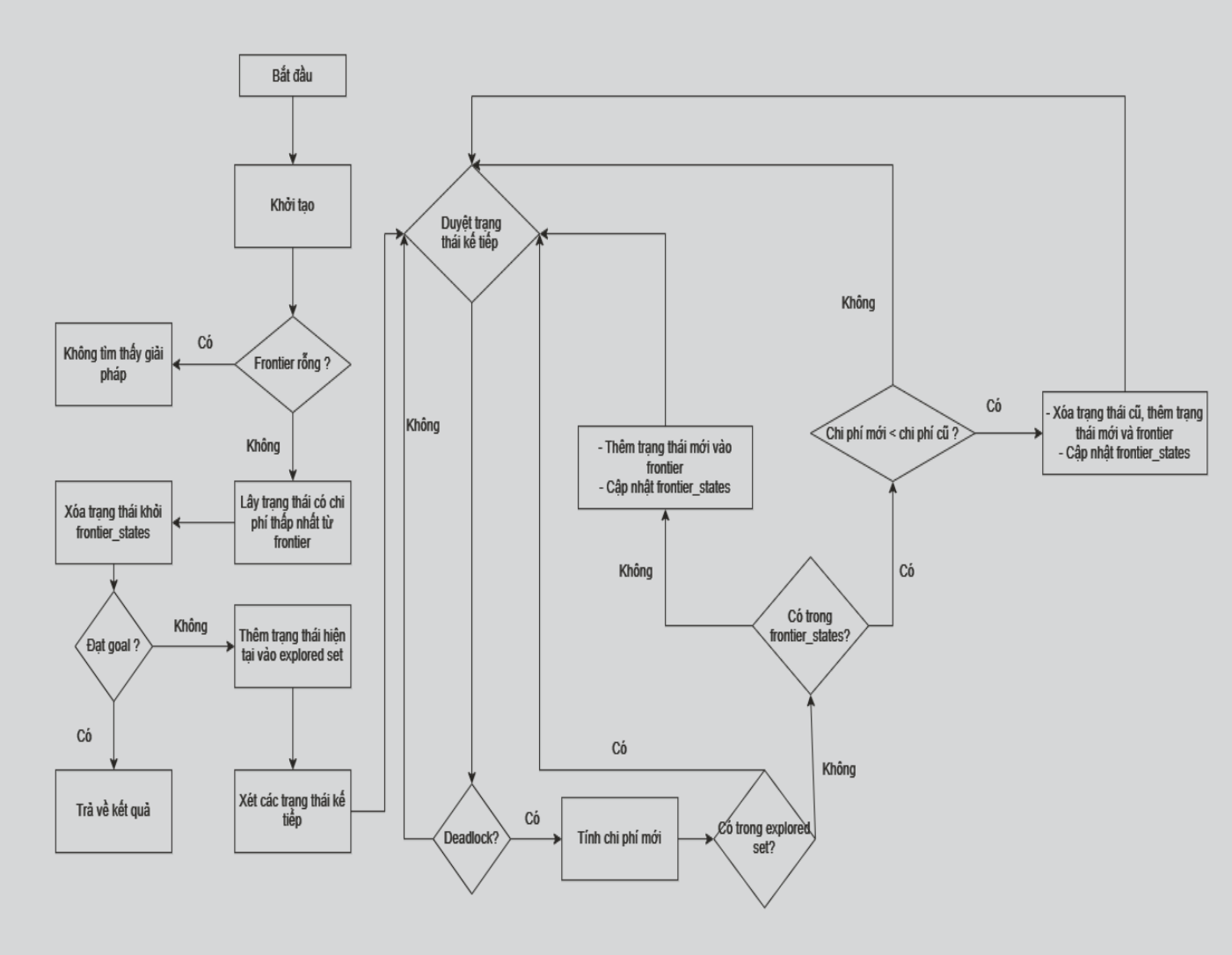
- *Duyệt từng "hàng xóm":* Với mỗi trạng thái, giải thuật sinh ra các trạng thái hàng xóm từ các nước đi hợp lệ của Ares và các viên đá. Thuật toán sẽ duyệt qua các trạng thái hàm xóm này.
- *Kiểm Tra Deadlock:* Nếu một trạng thái mới rơi vào tình huống "deadlock" (không thể tiếp tục đẩy đá đến đích, ở đây ta xét trường hợp các viên đá sẽ bị kẹt ở góc tường), giải thuật sẽ bỏ qua trạng thái này để tránh lãng phí tài nguyên.
- *Cập nhật tổng chi phí và khối lượng đá đã đẩy:* Tính toán chi phí mới (***new_cost***) bằng cách cộng chi phí hiện tại (***cost***) với chi phí bước đi (***move_cost***) từ trạng thái cũ sang trạng thái mới.
- Tạo một nút con (***child_state***) đại diện cho trạng thái mới của Ares và các tảng đá, trong đó các tảng đá được sắp xếp thành một ***tuple*** để đảm bảo tính duy nhất và bất biến.
- Kiểm tra trạng thái con ***(child_state)*** đã có trong ***explored_set*** hay chưa. Nếu có, tức là trạng thái này đã được duyệt trước đó, bỏ qua vòng lặp này và tiếp tục với trạng thái hàng xóm tiếp theo.
- Kiểm tra trạng thái trong ***frontier_states***:
  - Nếu trạng thái con (***child_state***) đã tồn tại trong ***frontier_states:***
    + Kiểm tra xem chi phí mới (***new_cost)*** có thấp hơn chi phí đã lưu cho trạng thái đó không. Nếu có, tiến hành cập nhật.
    + Cập nhật hàng đợi frontier*:* Xóa trạng thái cũ tương ứng với ***child_state*** khỏi hàng đợi ***frontier***. Sau đó, sử dụng ***heapq.heapify(frontier)*** để duy trì tính chất của hàng đợi ưu tiên.
    + Thêm trạng thái mới vào hàng đợi: Thêm trạng thái mới vào hàng đợi ***frontier*** với các thông tin chi phí mới (***new_cost***), vị trí mới của Ares (***ares_pos***), trạng thái mới của các tảng đá (***stones***) và đường di chuyển đã thực hiện (***path***). Cập nhật ***frontier_states*** với chi phí mới cho ***child_state***.

- Nếu trạng thái con (***child_state***) chưa tồn tại trong ***frontier_states:***
  + Tăng số lượng nút đã mở rộng (***expanded_nodes***) lên 1. Thêm trạng thái mới vào frontier và cập nhật chi phí trong ***frontier_states***.
  + Thêm trạng thái mới vào hàng đợi: Thêm trạng thái mới vào hàng đợi ***frontier*** với các thông tin chi phí mới (***new_cost***), vị trí mới của Ares (***ares_pos***), trạng thái mới của các tảng đá (***stones***) và đường di chuyển đã thực hiện (***path***). Cập nhật ***frontier_states*** với chi phí mới cho ***child_state***.

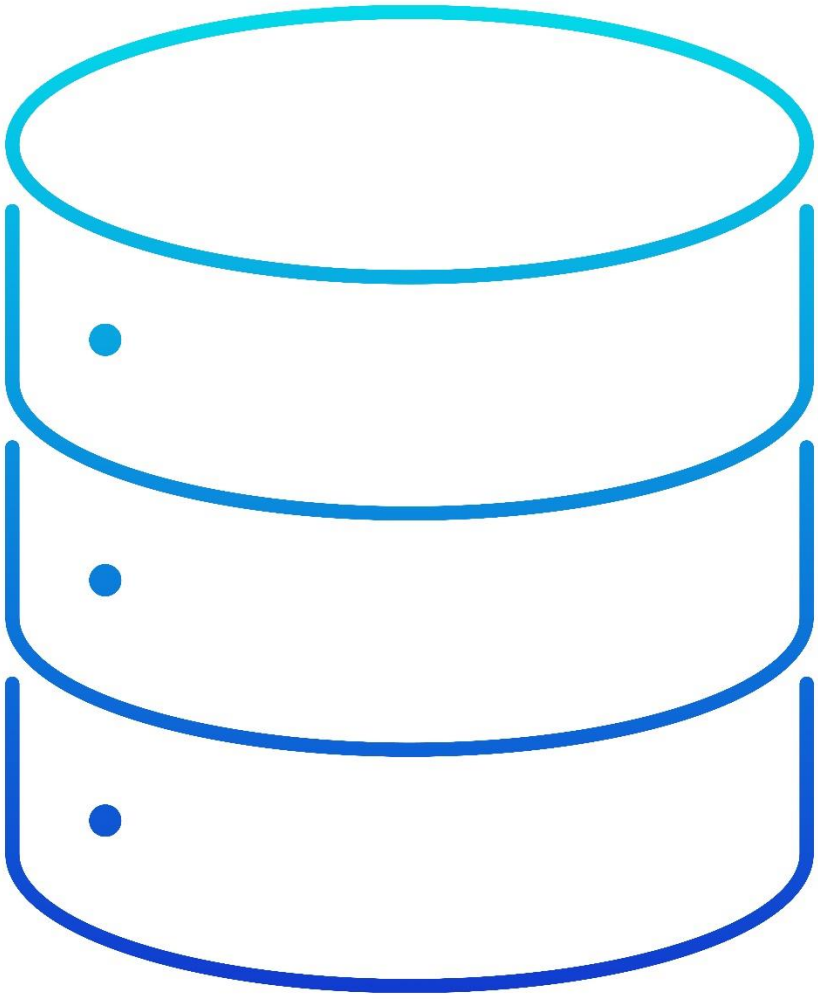
3. Kết thúc

Nếu giải thuật duyệt qua hết tất cả các trạng thái có thể mà không tìm được giải pháp, nó sẽ trả về kết quả cho biết không có giải pháp (chi phí vô cùng, thời gian và bộ nhớ vô cùng).

III. Mô tả thuật toán bằng flowchart



Section 5: *A* Algorithms.*



I. Thành viên thực hiện:

- 22120369 - Quan Phan Tiến.

II. Trình bày, mô tả giải thuật:

- **Thuật toán:**

```
problem – An instance of the graph
node – A node with State =problem.InitialState, PathCost =0
node.TotalCost := node.PathCost + Heuristic(node)
frontier – A priority queue ordered by TotalCost, with node as the only element
explored – An empty set
while not frontier.IsEmpty?() do
    node := frontier.Pop()/* Returns the lowest-cost node */
    if problem.IsGoalState?(node.State) then
        return Solution(node)
    explored.Add(node)
    foreach action in problem.Actions(node.State) do
        child := ChildNode(problem, node, action)
        child.TotalCost = child.PathCost + Heuristic(node)
        if not (child.State in explored or child.State in frontier) then
            frontier.Insert(child)
        else if child.State in frontier with higher TotalCost then
            frontier.Replace(child)/* Replace the higher-cost state */
return failure
```

- **Khởi tạo:** Bắt đầu từ vị trí ban đầu của Ares, thêm vào danh sách frontier (danh sách các nút sẽ được kiểm tra).
- **Vòng lặp chính:**
    - Lấy nút có chi phí thấp nhất (dựa trên hàm f(n)=g(n)+h(n)), với g(n) là chi phí đi từ lúc bắt đầu đến vị trí hiện tại, tính bằng tổng số bước đi và tổng khối lượng đá đã đẩy, h(n) là hàm heuristic.

    - Nếu nút hiện tại là nút đích, trả về đường đi và thông tin về chi phí, thời gian chạy và bộ nhớ.
    - Nếu không, đánh dấu nút này là đã được kiểm tra và tiếp tục tìm kiếm các nút kề.
        - Lần lượt xét 04 hướng lên, xuống, trái phải của nút hiện tại, tìm các nút hợp lệ thông qua các hàm kiểm tra.
    - Với mỗi nút kề, nếu node chưa nằm trong explored_nodes, thực hiện tính toán chi phí f và cập nhật danh sách frontier nếu tìm thấy đường đi có giá trị f nhỏ hơn. Nếu node không nằm trong frontier, thực hiện push node vào frontier.
    - Lặp lại vòng lặp đến khi frontier rỗng.
- **Kết thúc:** Nếu không tìm thấy đường đi đến nút đích, trả về không tìm thấy giải pháp.

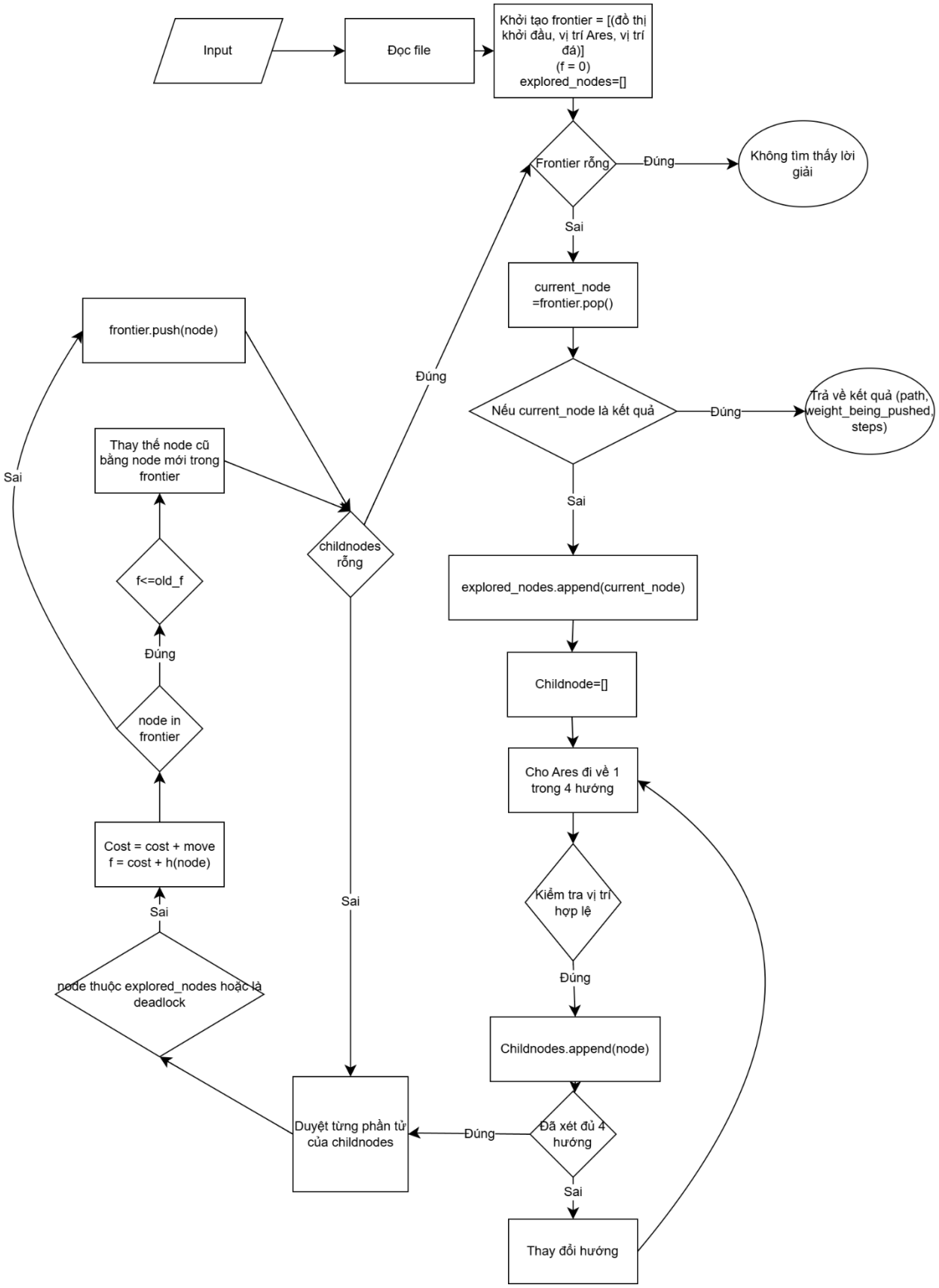
Hàm heuristic:

- Tổng khoảng cách Manhattan từ một tảng đá đến một trong các công tắc gần nó nhất.
- Giá trị của hàm heuristic luôn nhỏ hơn giá trị thật của đường đi ngắn nhất đến lời giải. => admissible.
- Giá trị hàm heuristic của một node luôn nhỏ hơn chi phí giữa node đó và một node khác cộng với giá trị hàm heuristic của node đó. (f(n) <= f(n') + g(n, n')) => Tính nhất quán (Consistency).

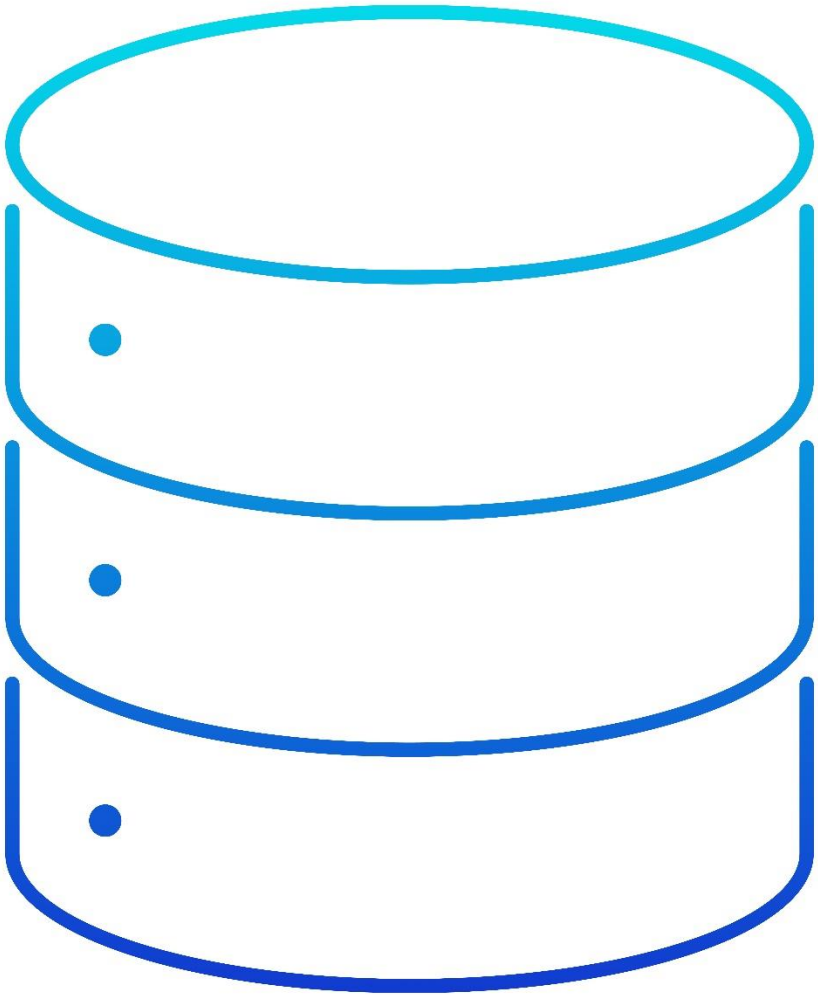
Hàm kiểm tra deadlock:

- Duyệt qua từng viên đá.
- Loại bỏ những viên đá nằm trên công tắc.
- Kiểm tra nếu viên đá bị mắc kẹt ở góc (tức là bị bế tắc).
- Trả về kết quả True nếu có viên đá bị bế tắc hoặc False nếu tất cả các viên đá đều có thể di chuyển.

III. Mô tả thuật toán bằng flowchart:

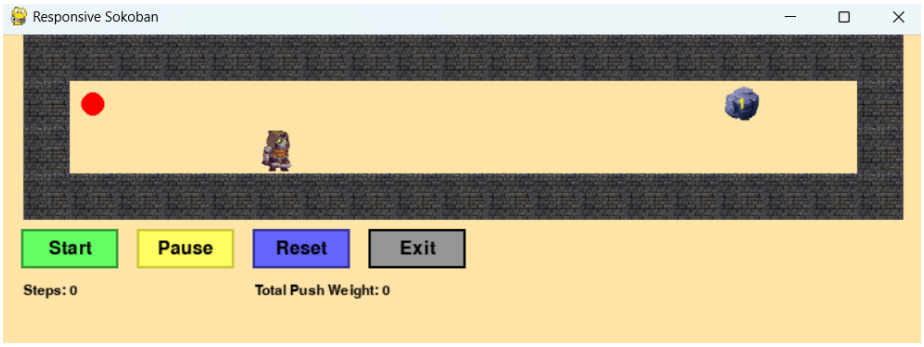


Section 6: *Mô tả testcase và kết quả kì vọng.*

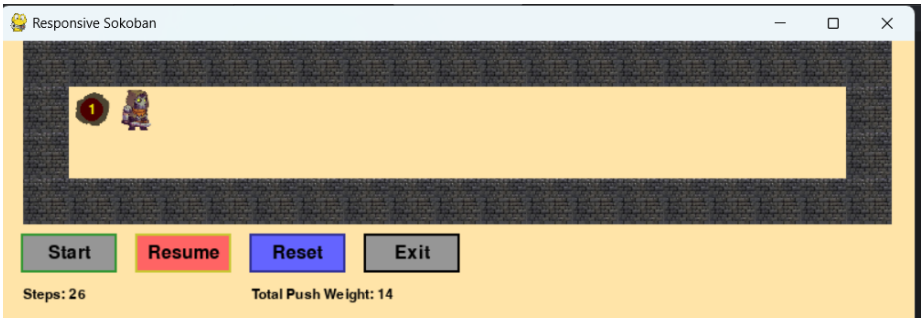


Testcase 01:

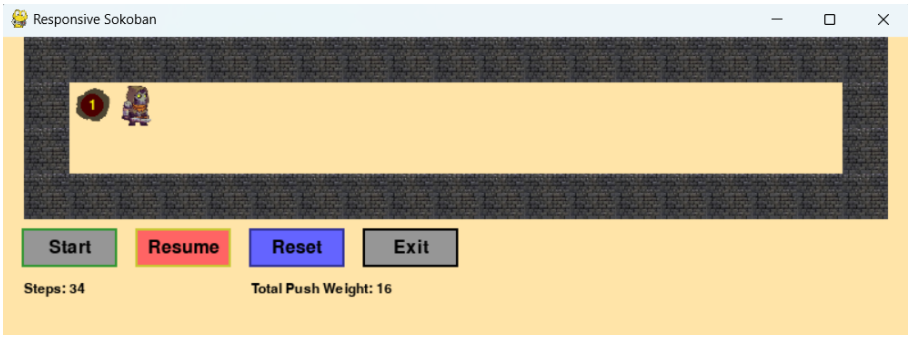
- Input:



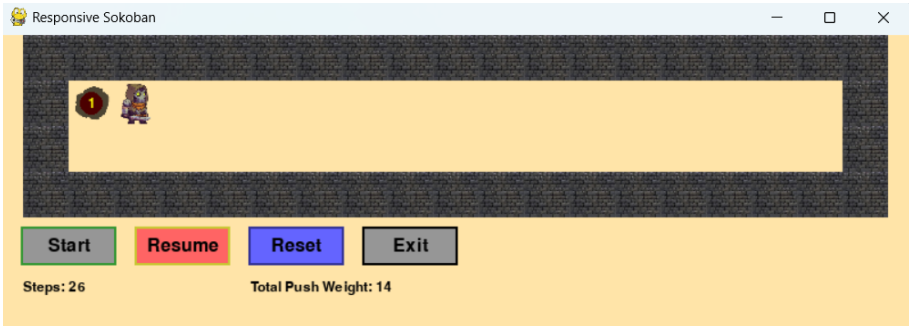
- Output:
  - BFS:



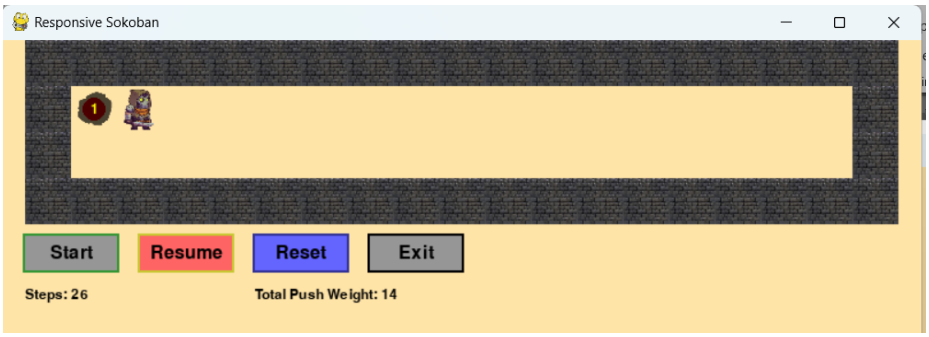
  - DFS:



- UCS:

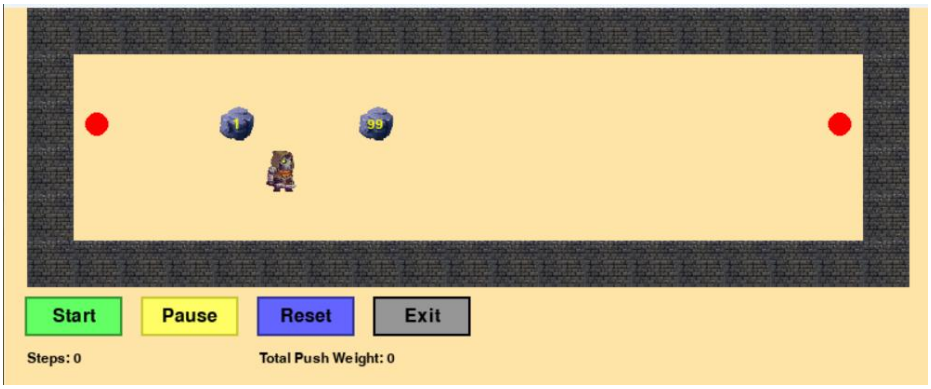

Responsive Sokoban
Start
Resume
Reset
Exit
Steps: 26
Total Push Weight: 14

- A*:

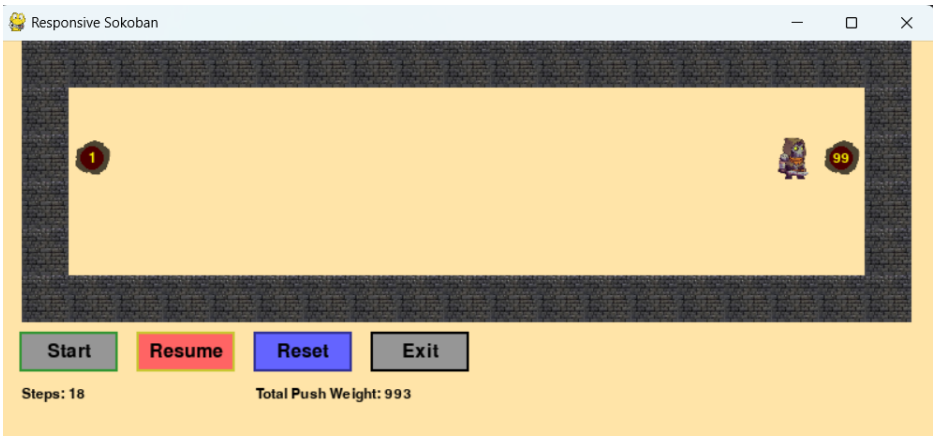

Responsive Sokoban
Start
Resume
Reset
Exit
Steps: 26
Total Push Weight: 14

Testcase 02:

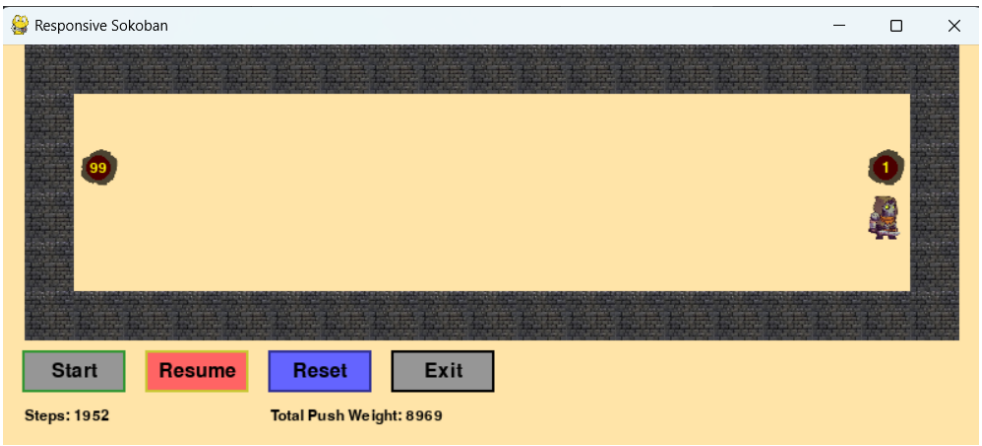
- Input:



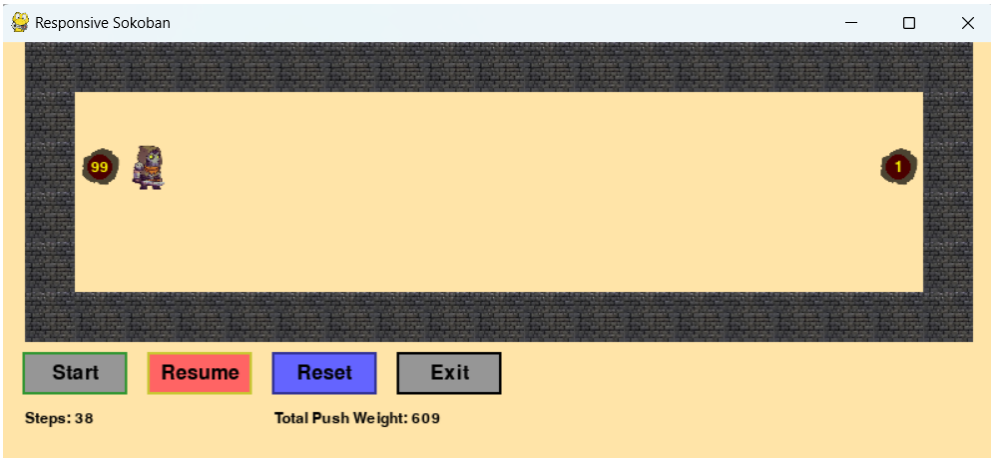
- Output:
  - BFS:



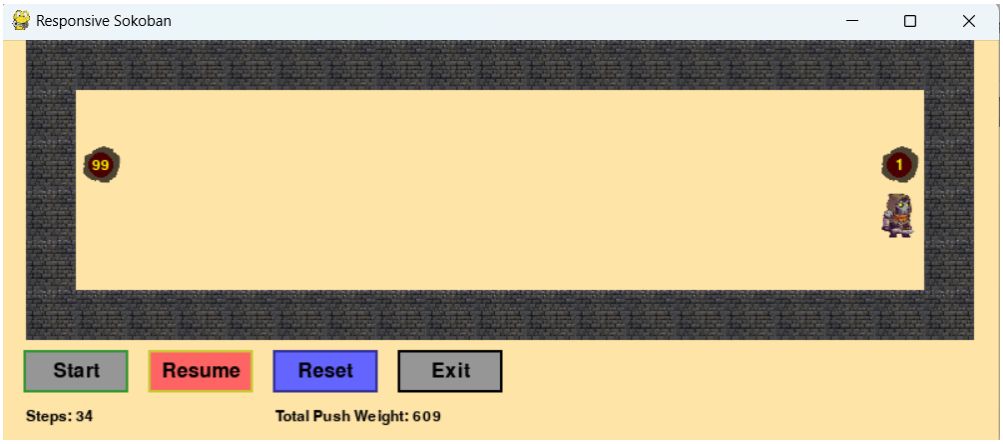
  - DFS:



- UCS:


Responsive Sokoban
99
1
Start
Resume
Reset
Exit
Steps: 38
Total Push Weight: 609

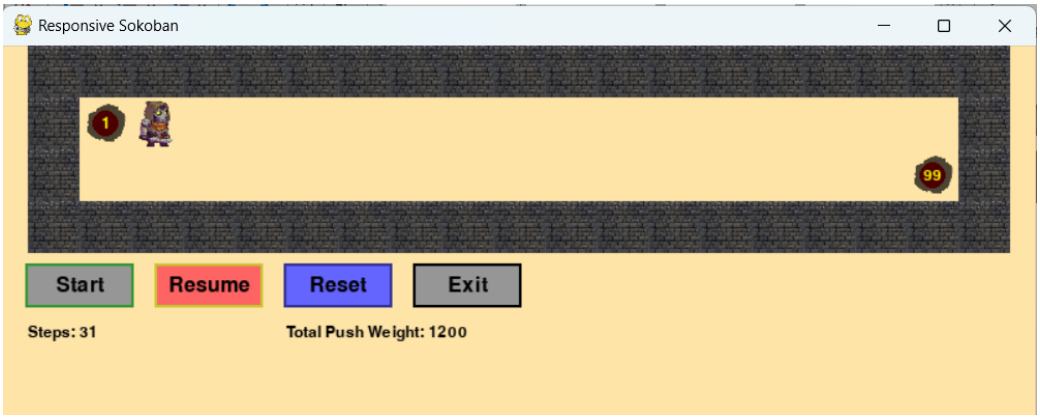
- A*:


Responsive Sokoban
99
1
Start
Resume
Reset
Exit
Steps: 34
Total Push Weight: 609

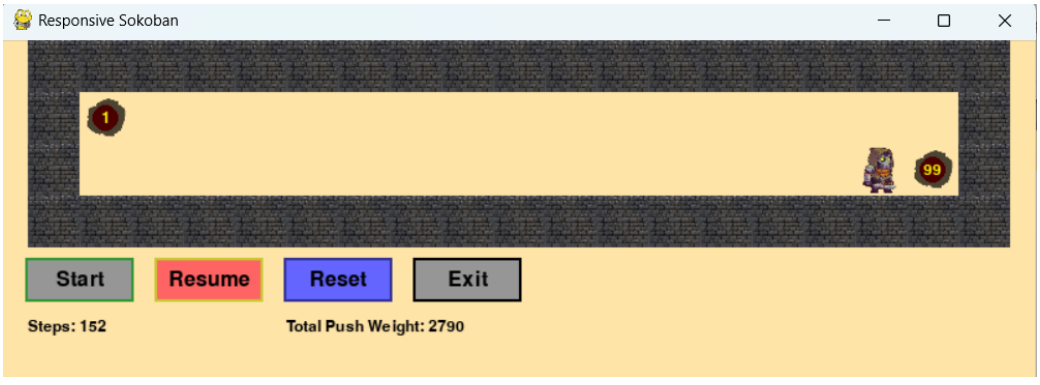
Testcase 03:

- Input:

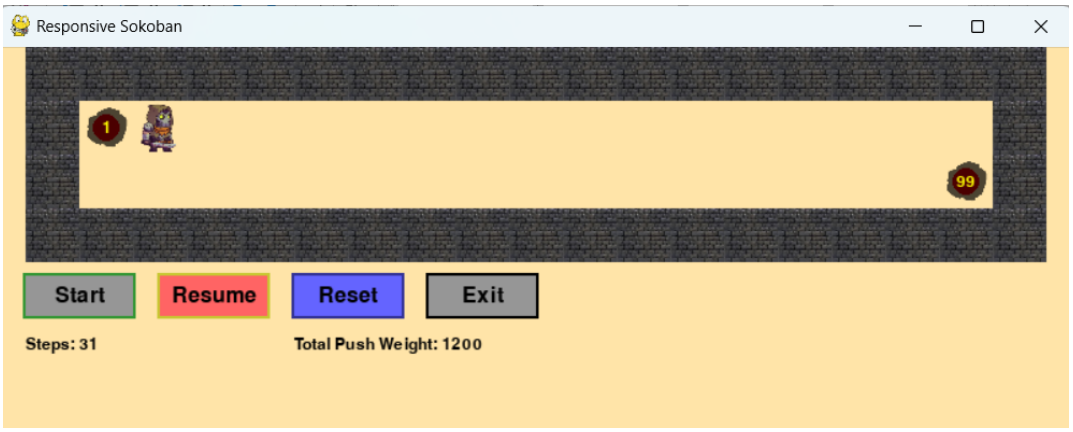
- Output:
  - BFS:



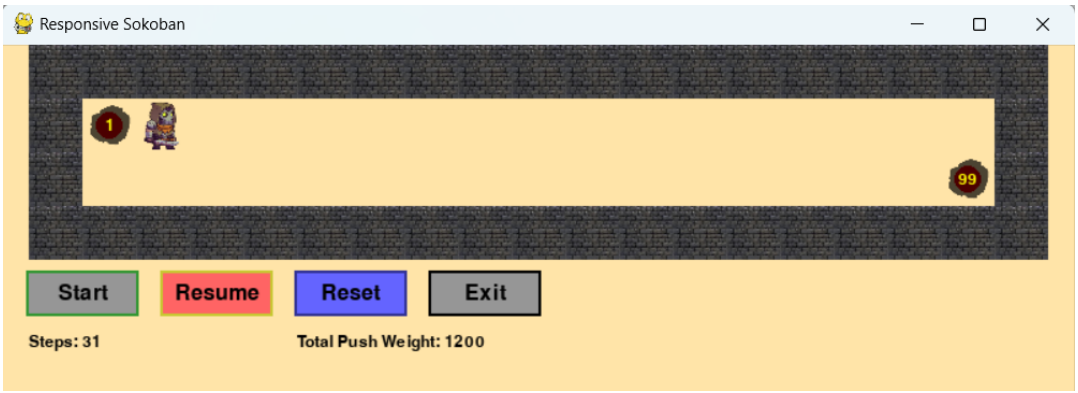
  - DFS:



- UCS:


Responsive Sokoban
1
99
Start
Resume
Reset
Exit
Steps: 31
Total Push Weight: 1200

- A*:


Responsive Sokoban
1
99
Start
Resume
Reset
Exit
Steps: 31
Total Push Weight: 1200

Testcase 04:

Testcase 04 là bài toán sai không thể giải, dùng để kiểm tra khả năng phát hiện các trường hợp không thể giải.

- Input:

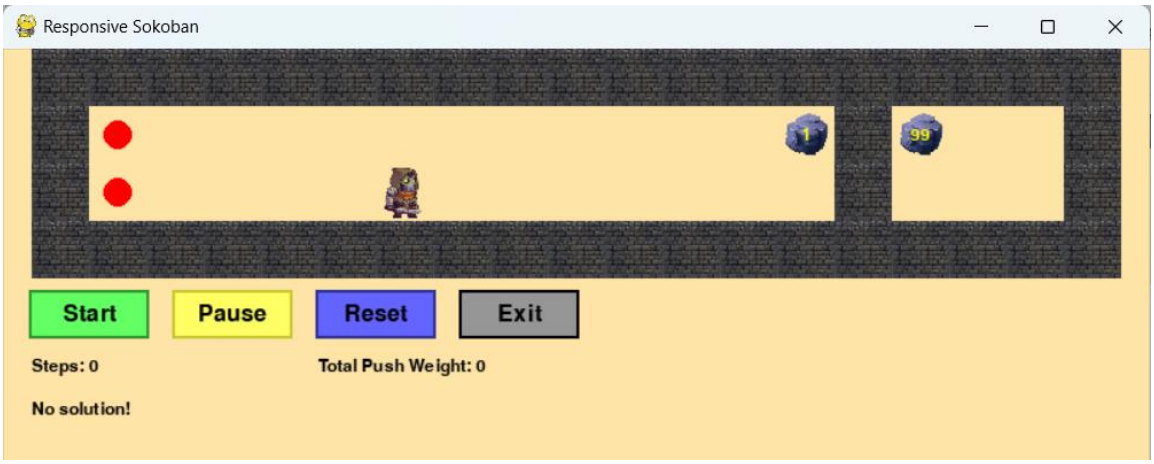
- Output:
  - BFS:

  - DFS:

- UCS:

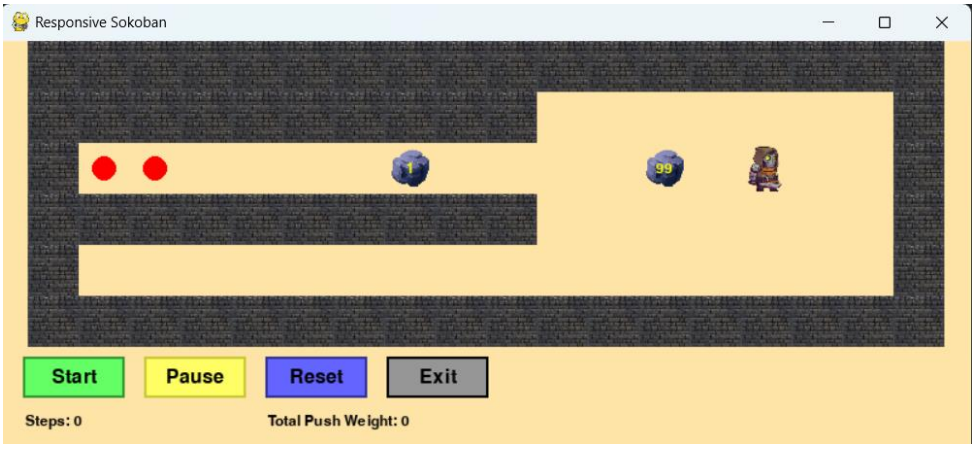

Responsive Sokoban
Start
Pause
Reset
Exit
Steps: 0
Total Push Weight: 0
No solution!

- A*:

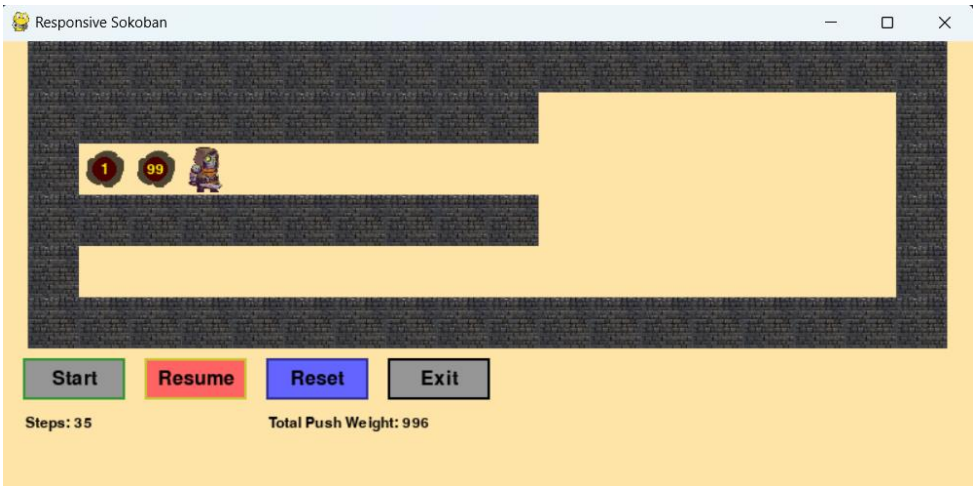

Responsive Sokoban
Start
Pause
Reset
Exit
Steps: 0
Total Push Weight: 0
No solution!

Testcase 05:

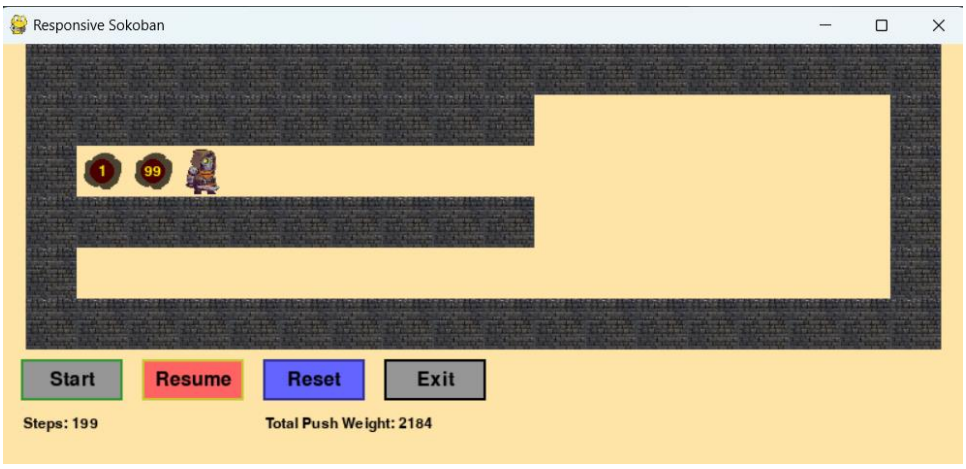
- Input:



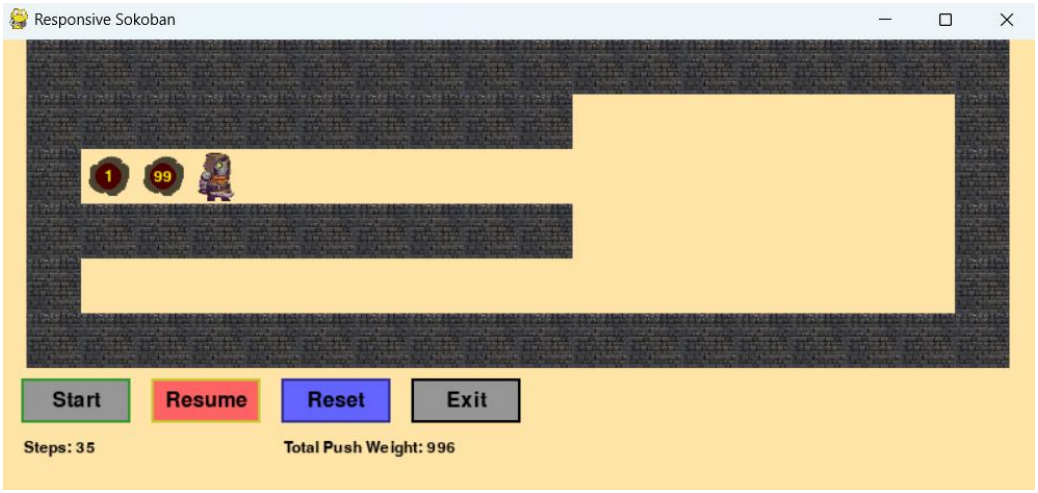
- Output:
  - BFS:



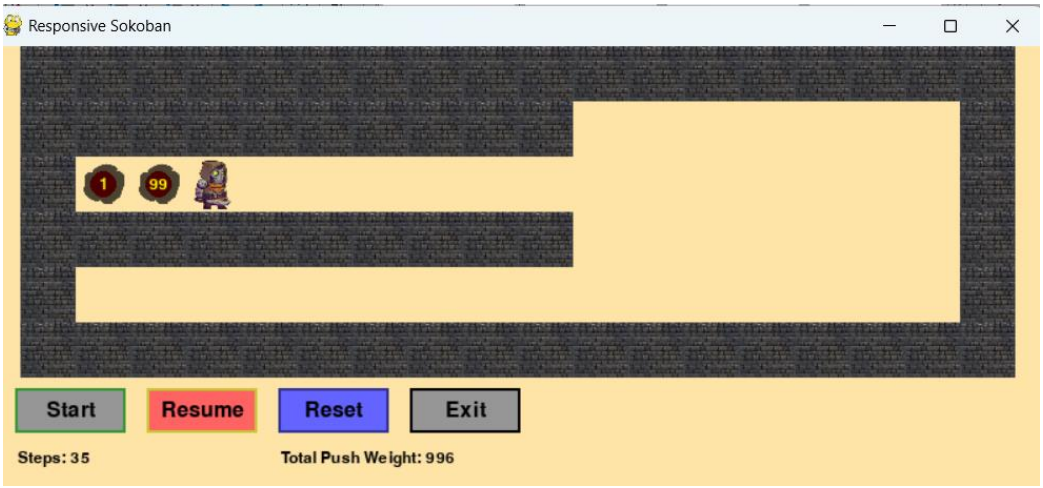
  - DFS:



- UCS:

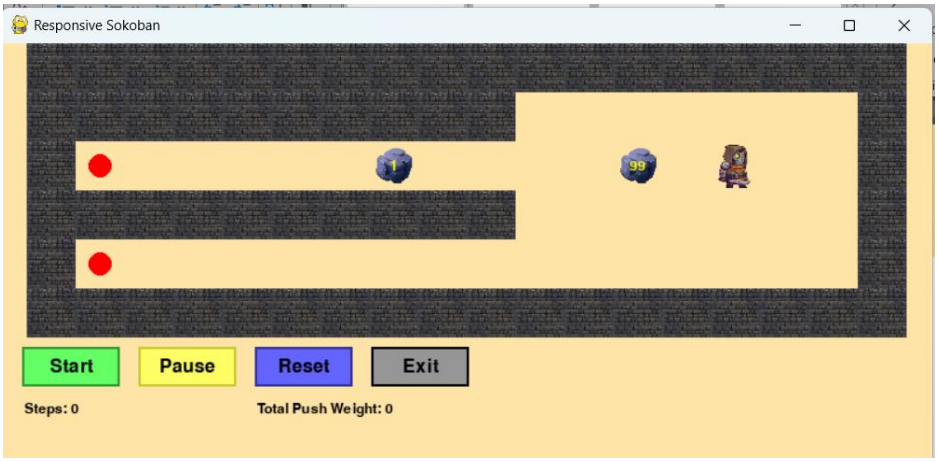

Responsive Sokoban
1
99
Start
Resume
Reset
Exit
Steps: 35
Total Push Weight: 996

- A*:


Responsive Sokoban
1
99
Start
Resume
Reset
Exit
Steps: 35
Total Push Weight: 996

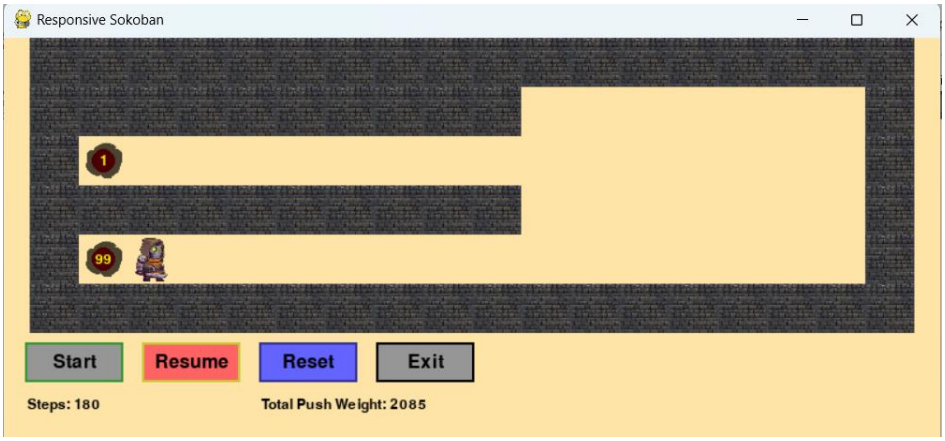
Testcase 06:

- Input:



- Output:
  - BFS:

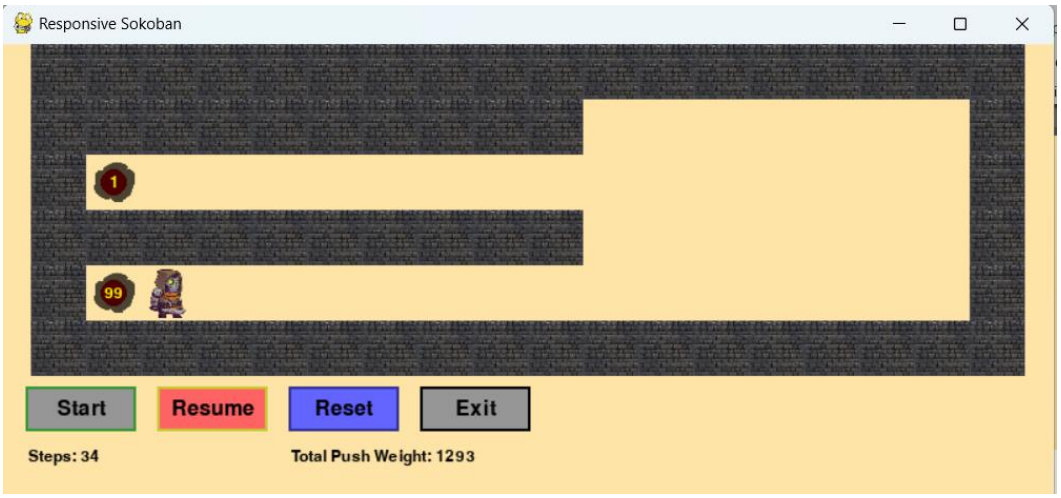
  - DFS:



- UCS:


Responsive Sokoban
1
99
Start
Resume
Reset
Exit
Steps: 34
Total Push Weight: 1293

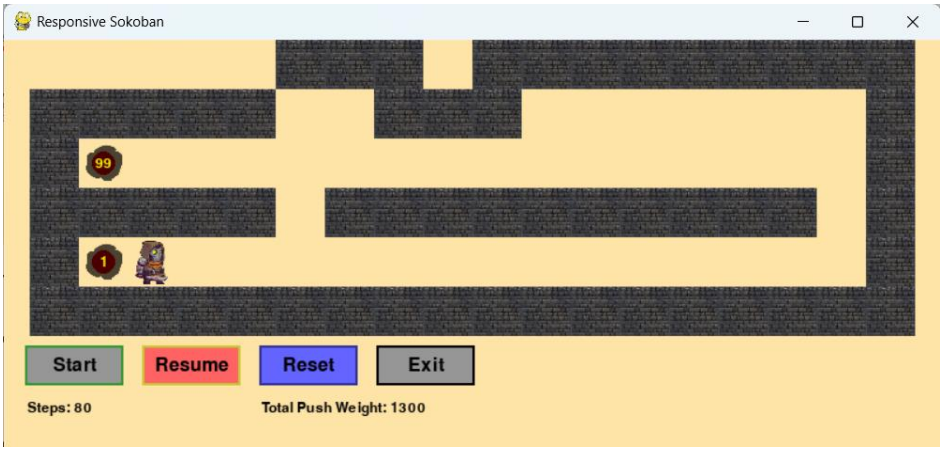
- A*:


Responsive Sokoban
1
99
Start
Resume
Reset
Exit
Steps: 34
Total Push Weight: 1293

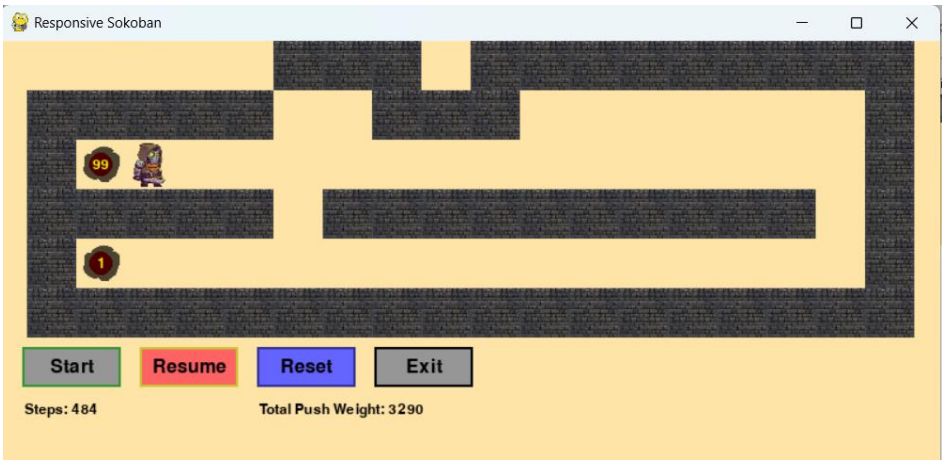
Testcase 07:

- Input:

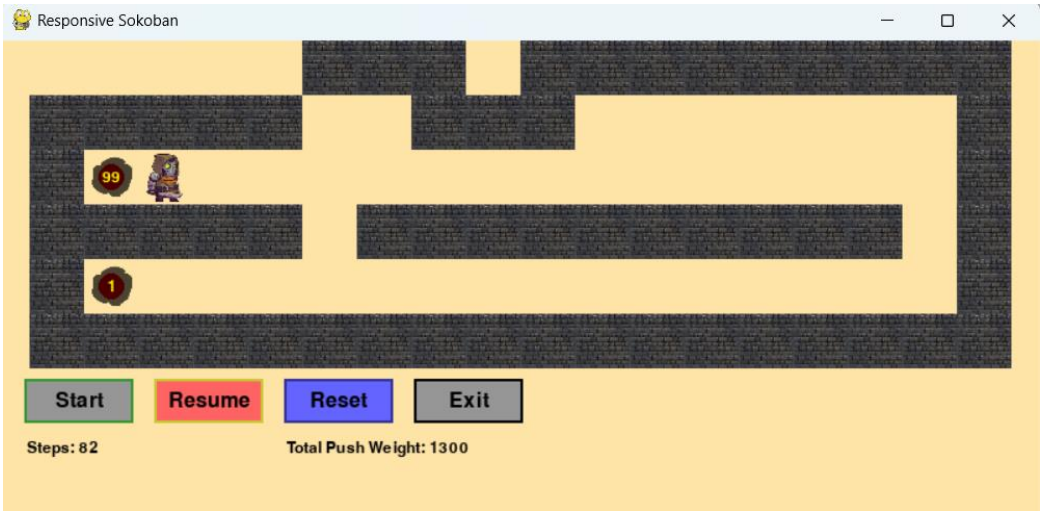
- Output:
  - BFS:



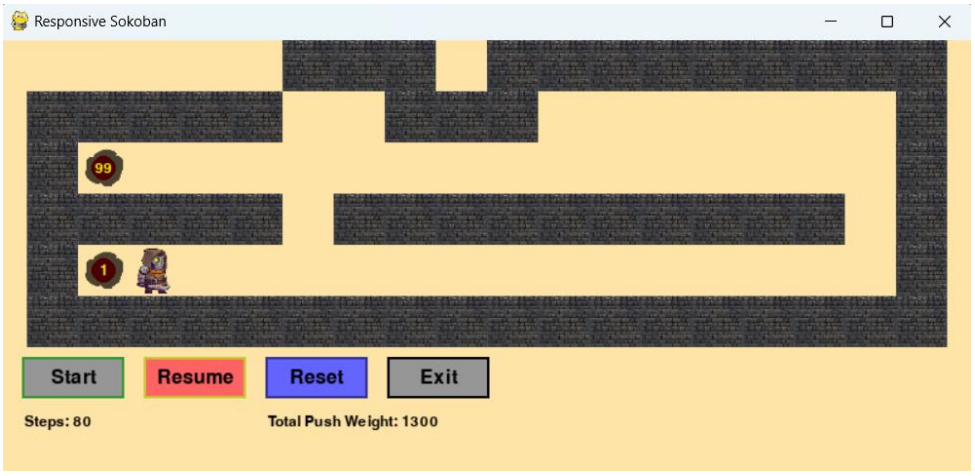
  - DFS:



- UCS:


Responsive Sokoban
99
1
Start
Resume
Reset
Exit
Steps: 82
Total Push Weight: 1300

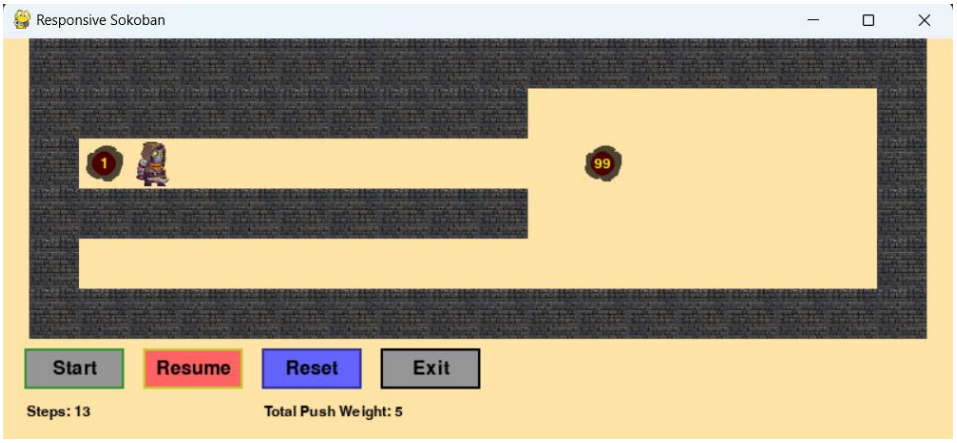
- A*:


Responsive Sokoban
99
1
Start
Resume
Reset
Exit
Steps: 80
Total Push Weight: 1300

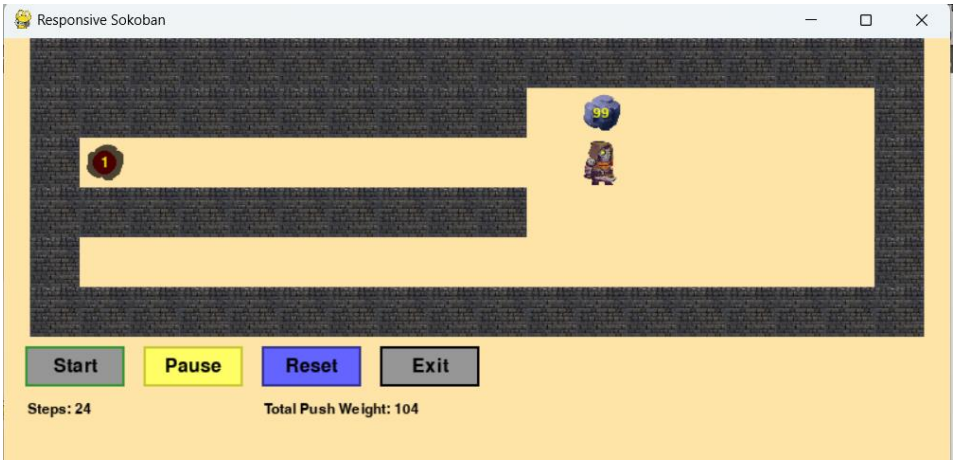
Testcase 08:

- Input:

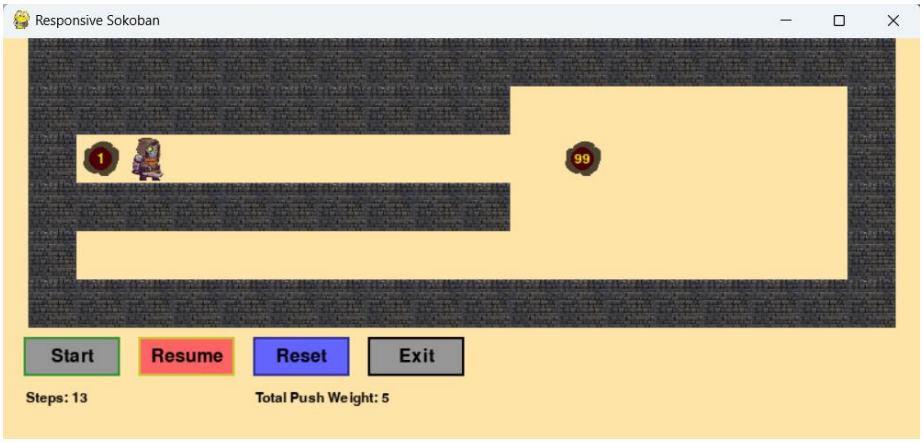
- Output:
  - BFS:



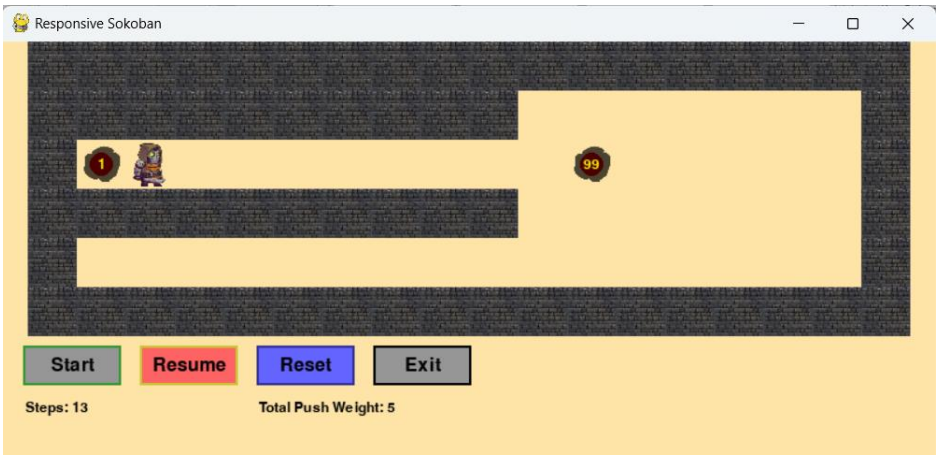
  - DFS:



- UCS:

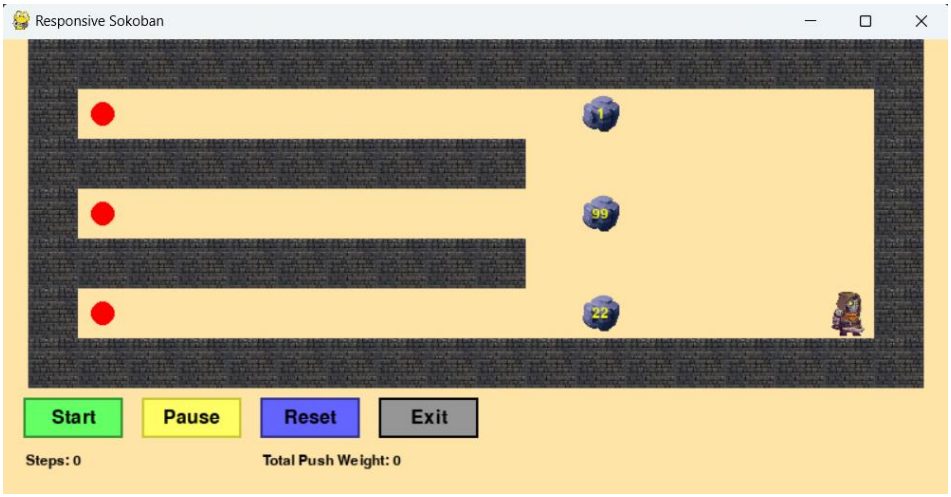

Responsive Sokoban
1
99
Start
Resume
Reset
Exit
Steps: 13
Total Push Weight: 5

- A*:

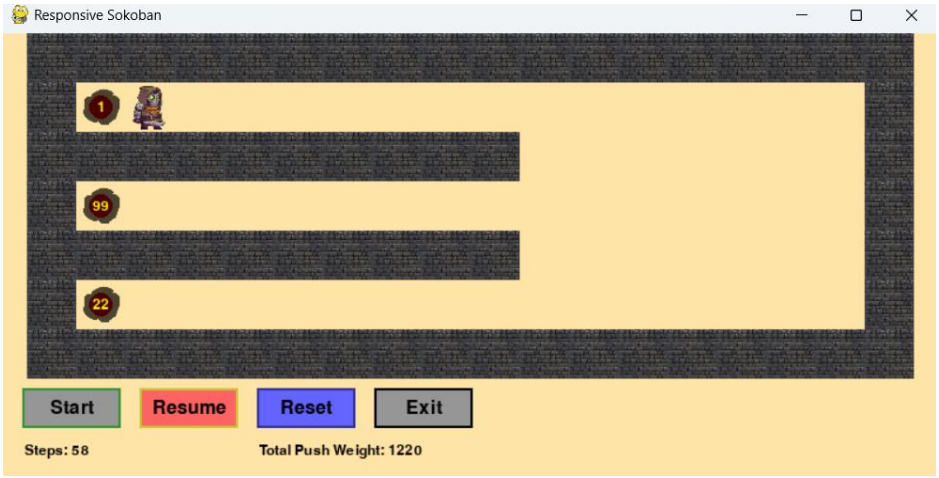

Responsive Sokoban
1
99
Start
Resume
Reset
Exit
Steps: 13
Total Push Weight: 5

Testcase 09:

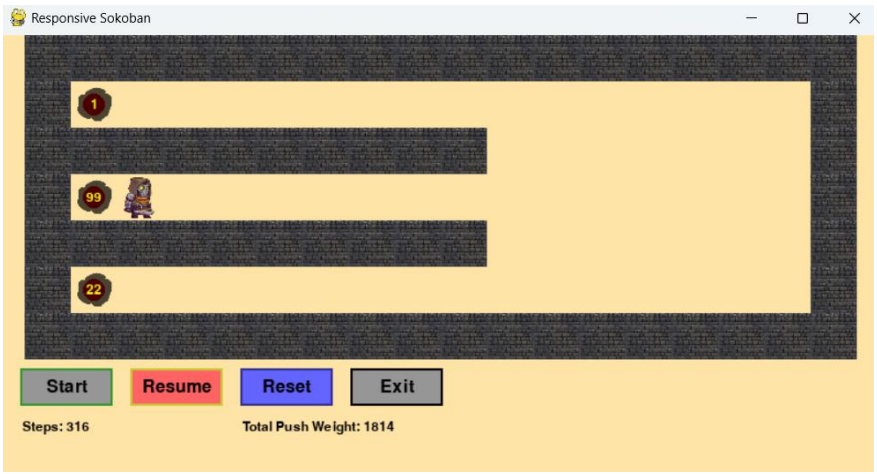
- Input:



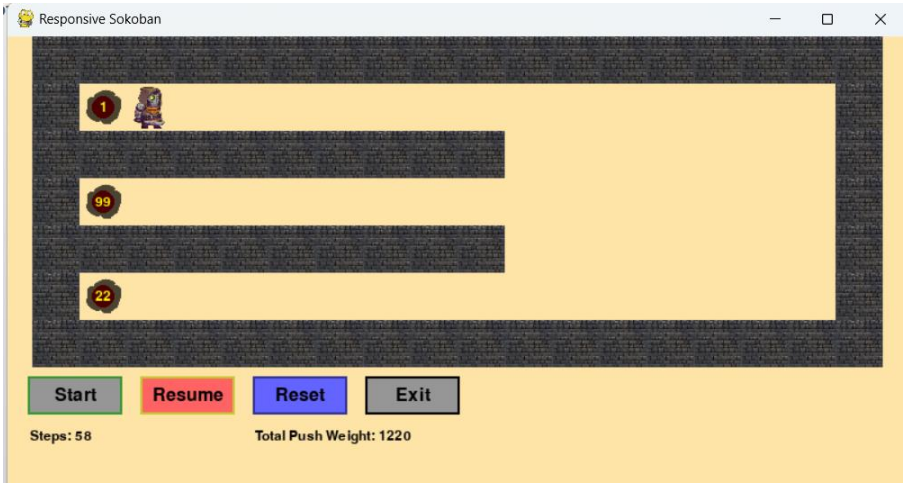
- Output:
  - BFS:



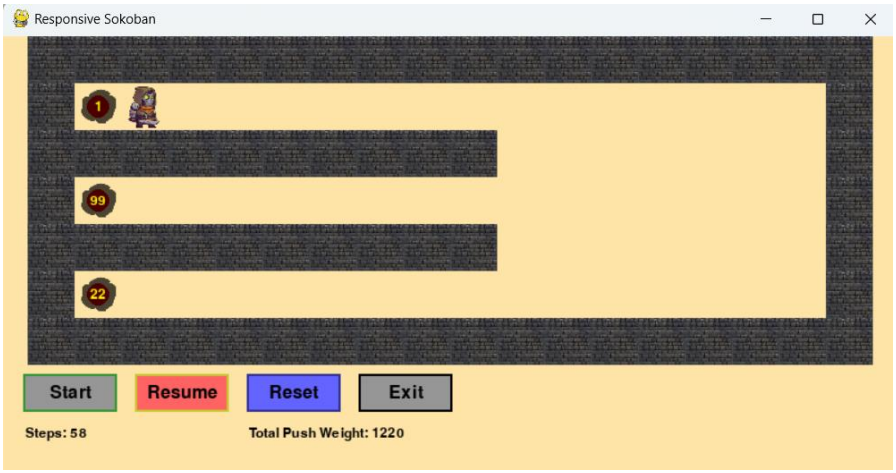
  - DFS:



- UCS:

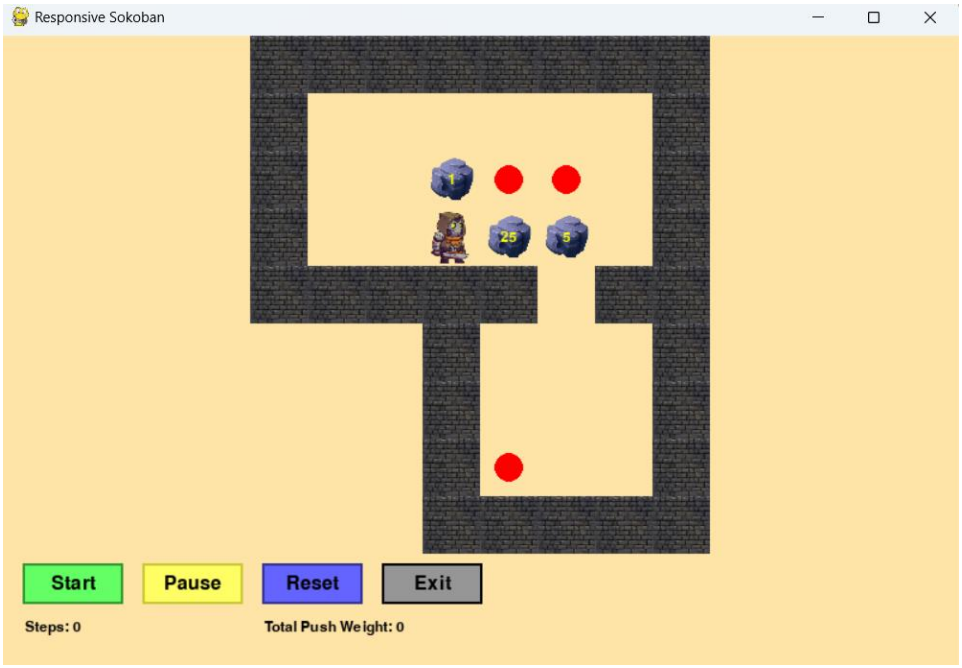

Responsive Sokoban
1
99
22
Start
Resume
Reset
Exit
Steps: 58
Total Push Weight: 1220

- A*:

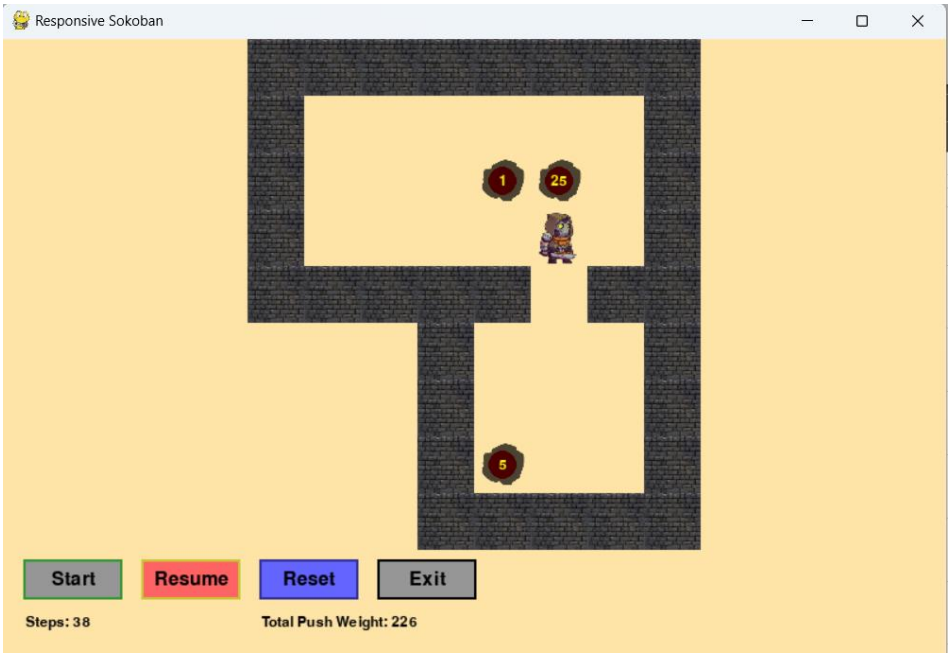

Responsive Sokoban
1
99
22
Start
Resume
Reset
Exit
Steps: 58
Total Push Weight: 1220

Testcase 10:

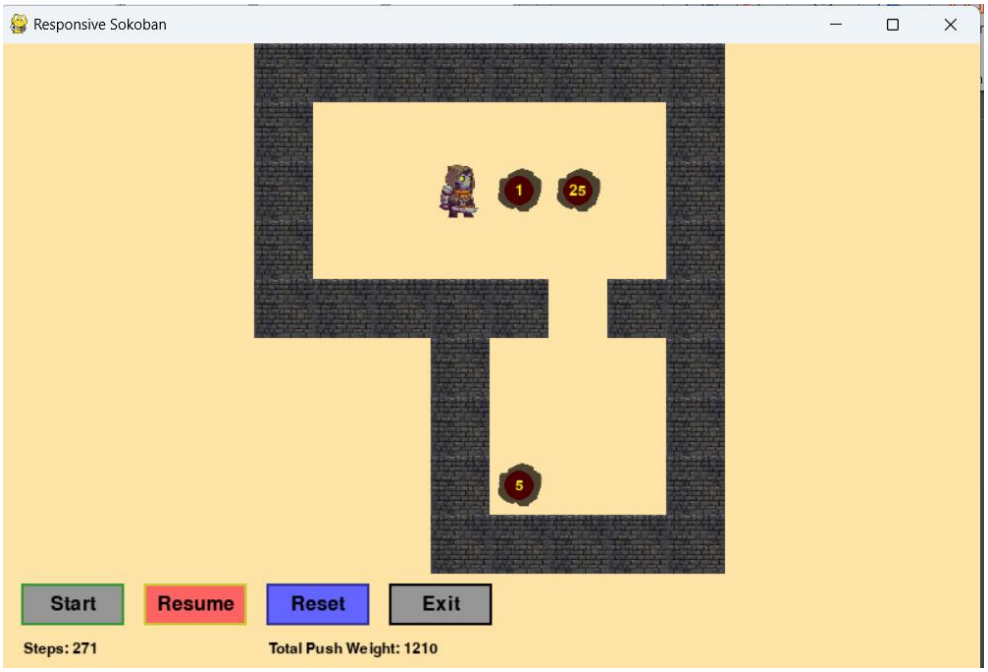
- Input:



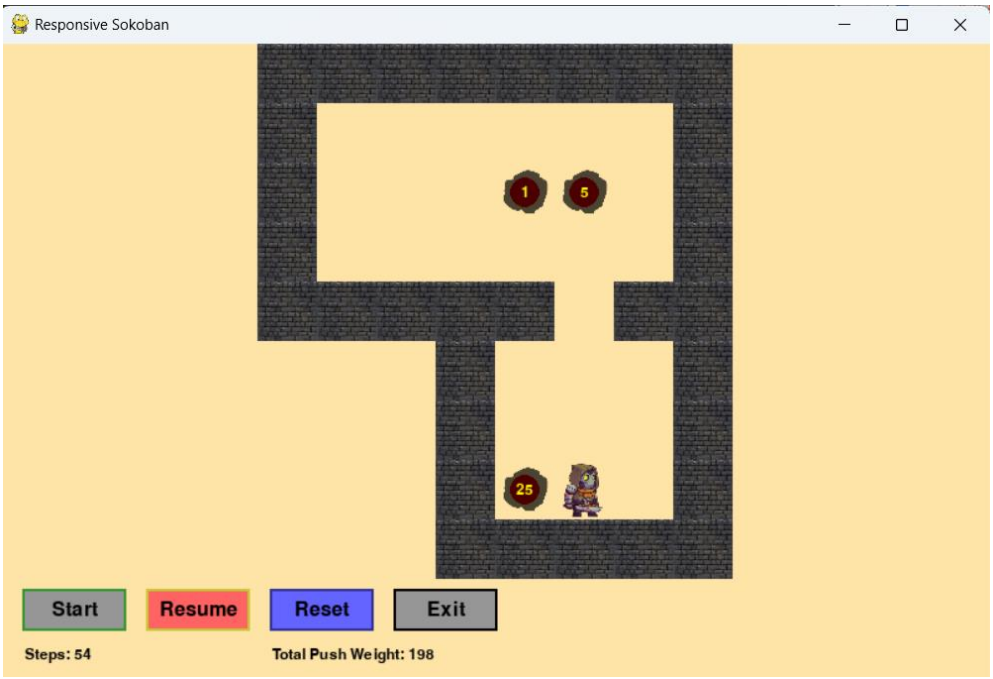
- Output:
  - BFS:



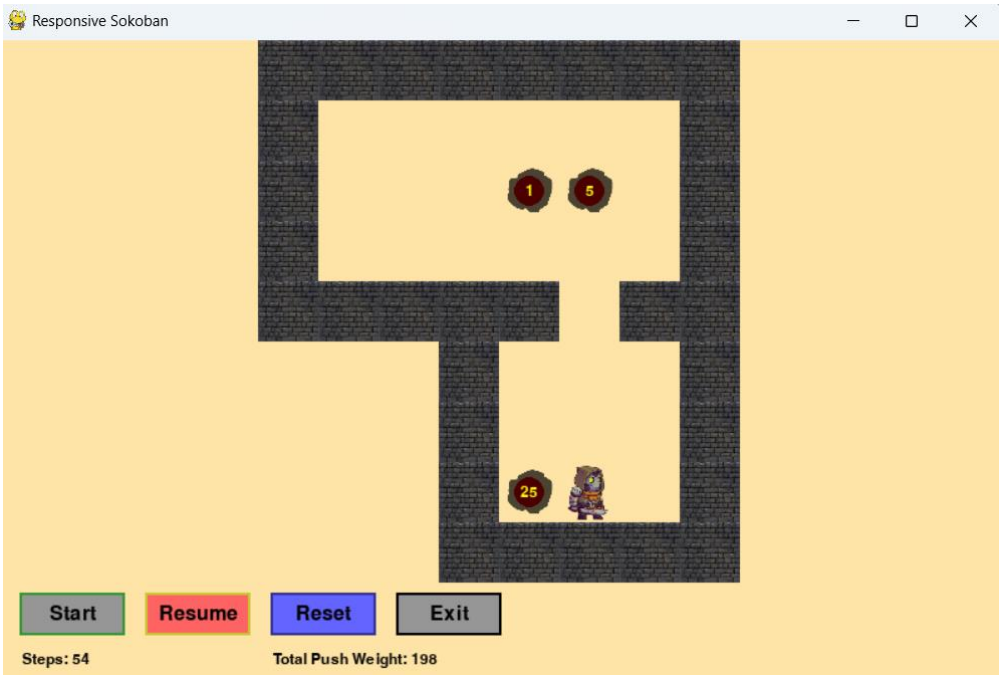
- DFS:


Responsive Sokoban
1
25
5
Start
Resume
Reset
Exit
Steps: 271
Total Push Weight: 1210

- UCS:


Responsive Sokoban
1
5
25
Start
Resume
Reset
Exit
Steps: 54
Total Push Weight: 198

- A*:

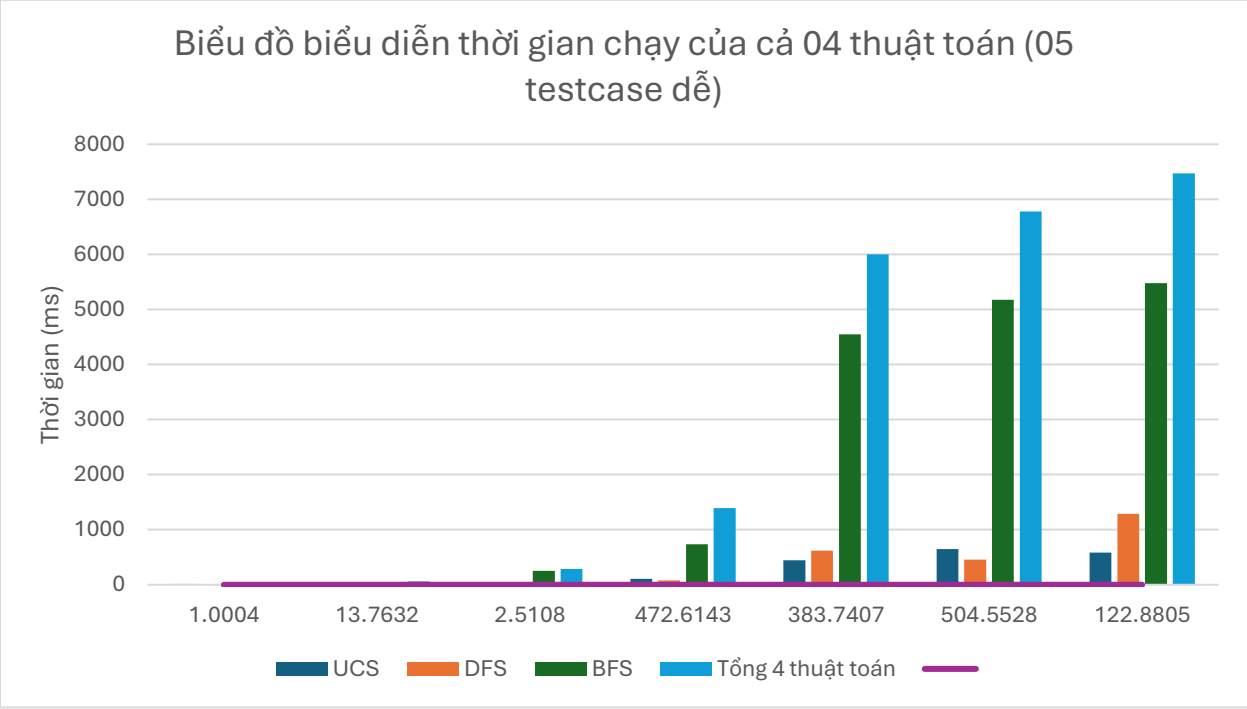

Responsive Sokoban
1
5
25
Start
Resume
Reset
Exit
Steps: 54
Total Push Weight: 198

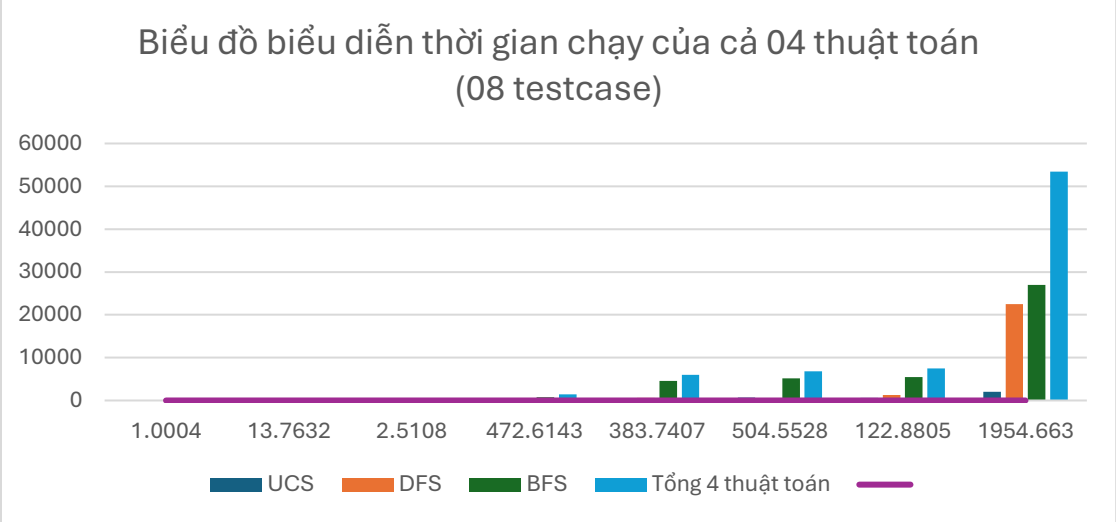
❖ **Thống kê và phân tích:**

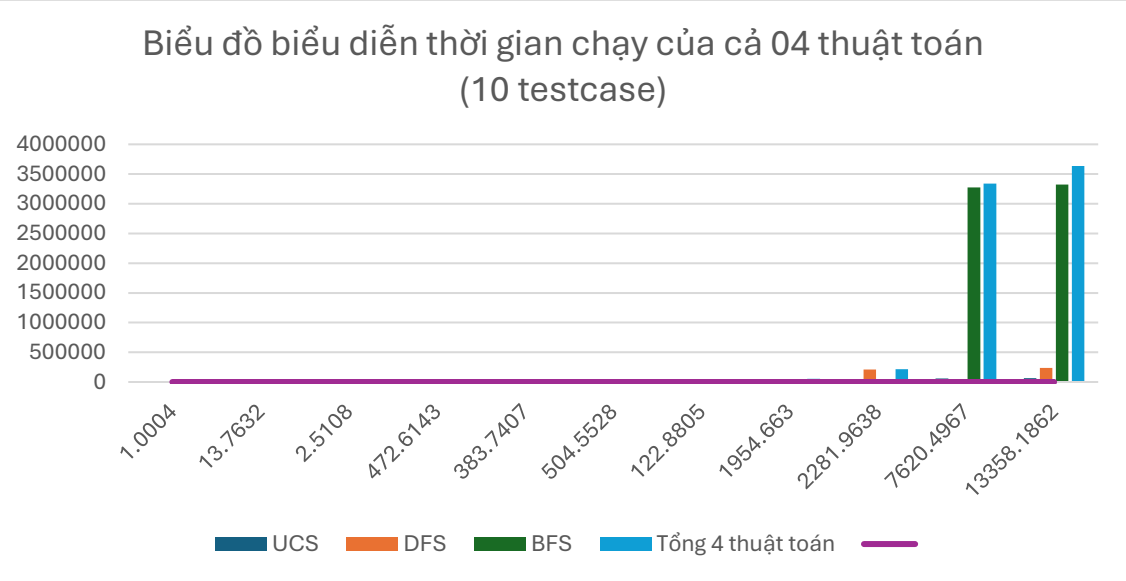
- **Thời gian:**

	BFS	DFS	UCS	A*	Tổng 04 thuật toán
Testcase 04	7.6149	2.9998	14.2808	1.0004	25.8959
Testcase 01	21.5011	22.2361	4.3433	13.7632	61.8437
Testcase 08	254.5679	24.6251	3.5195	2.5108	285.2233
Testcase 03	737.3531	78.7003	104.9597	472.6143	1393.6274
Testcase 07	4549.2289	619.8366	445.0438	383.7407	5997.85
Testcase 06	5177.4602	453.7578	645.9711	504.5528	6781.7419
Testcase 05	5479.9967	1285.3944	581.2829	122.8805	7469.5545
Testcase 10	26943.9905	22508.873	2018.9652	1954.663	53426.4917
Testcase 02	4442.1585	209341.8877	1940.5277	2281.9638	218006.5377
Testcase 09	3274298.77	434.8896	58070.4198	7620.4967	3340424.575
Tổng 10 Testcase	47613.8718	234338.3108	5758.894	5737.6895	293448.7661

Bảng giá trị thời gian chạy của 4 thuật toán theo từng Testcase (sắp xếp theo tổng thời gian của cả 4 thuật toán giảm dần) – đơn vị: ms.





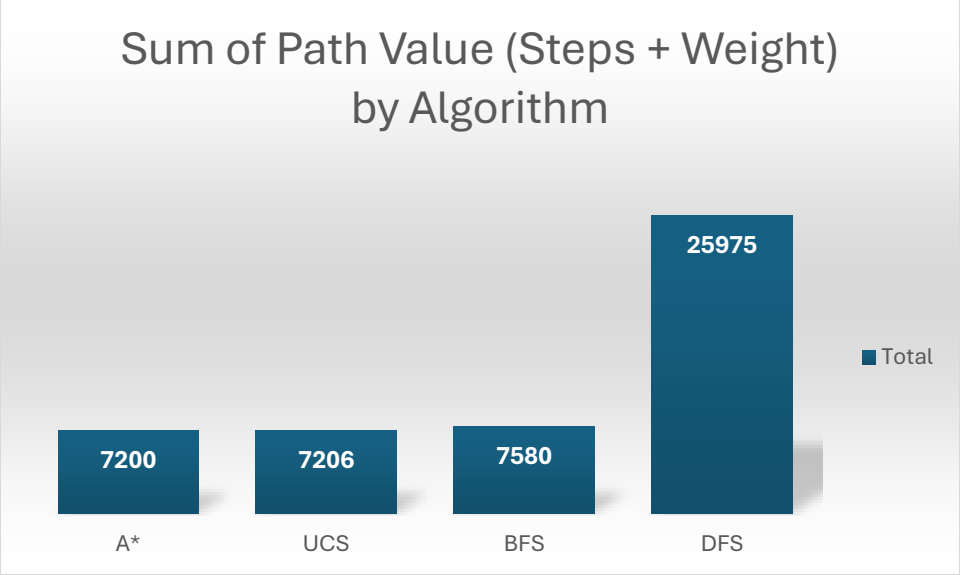


- Thuật toán có tổng thời gian nhỏ nhất: A*.
- Thuật toán có tổng thời gian lớn nhất: BFS.

- **Chi phí (số bước Ares đi + khối lượng đá đã đẩy):**

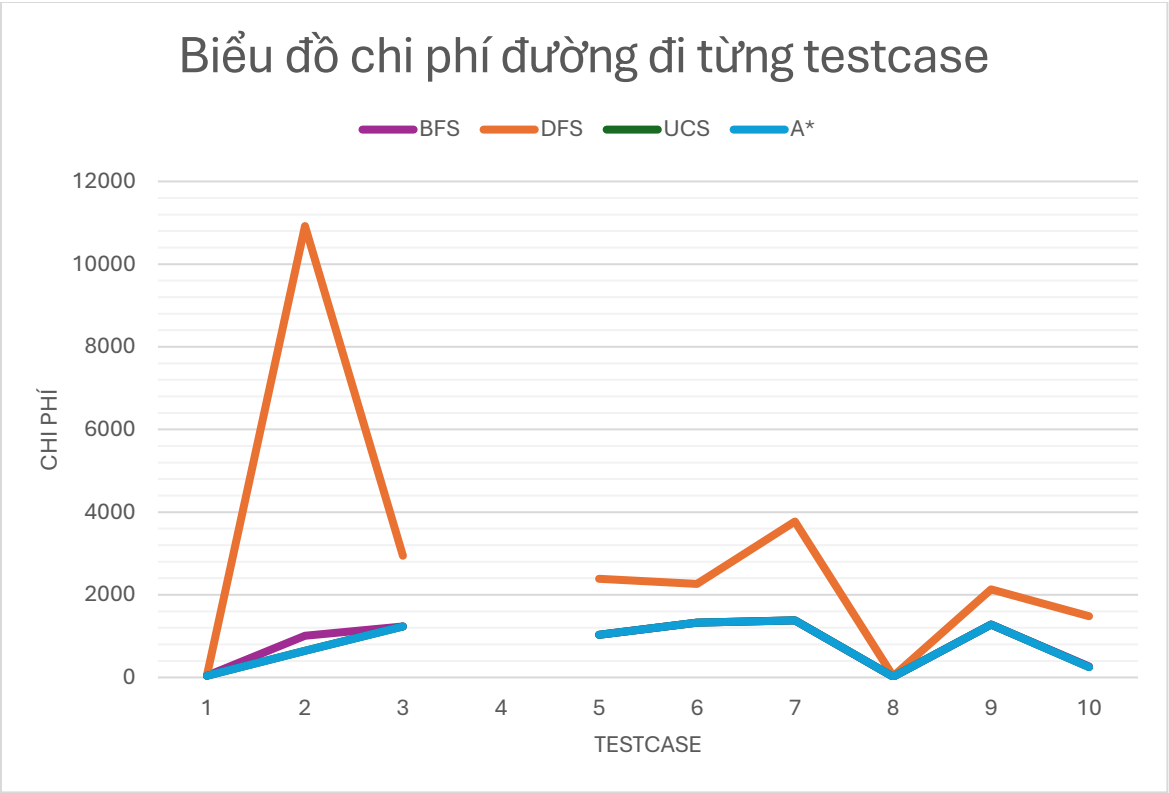
Algorithm	Sum of Path Value (Steps + Weight)
A*	7200
BFS	7580
DFS	25975
UCS	7206

Bảng giá trị tổng chi phí đường đi của 04 thuật toán của cả 10 testcase.



Testcase	BFS	DFS	UCS	A*
1	40	50	40	40
2	1011	10921	647	643
3	1231	2942	1231	1231
4				
5	1031	2383	1031	1031
6	1327	2265	1327	1327
7	1380	3774	1382	1380
8	18	29	18	18
9	1278	2130	1278	1278
10	264	1481	252	252

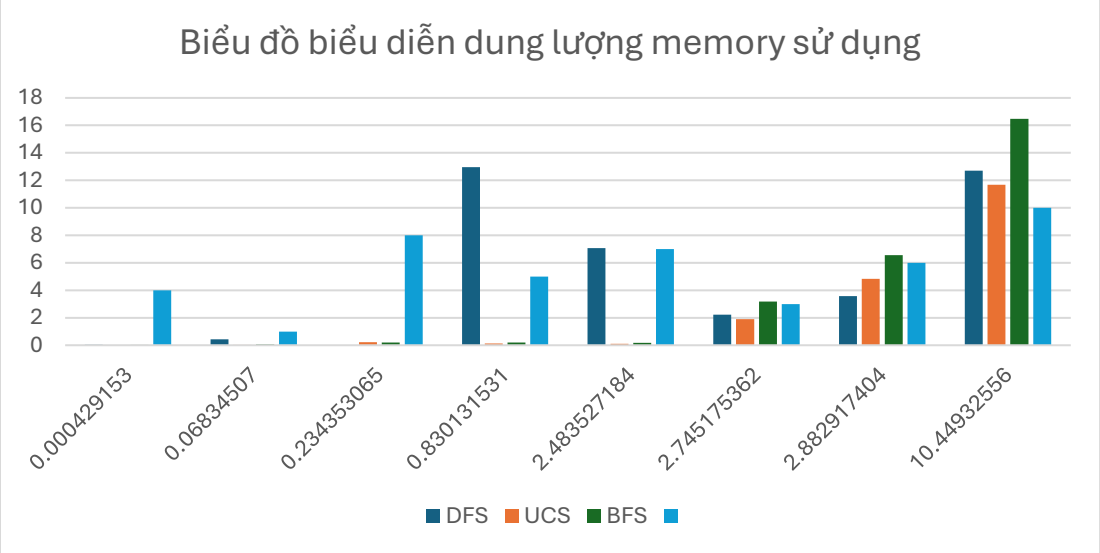
Bảng giá trị chi phí đường đi

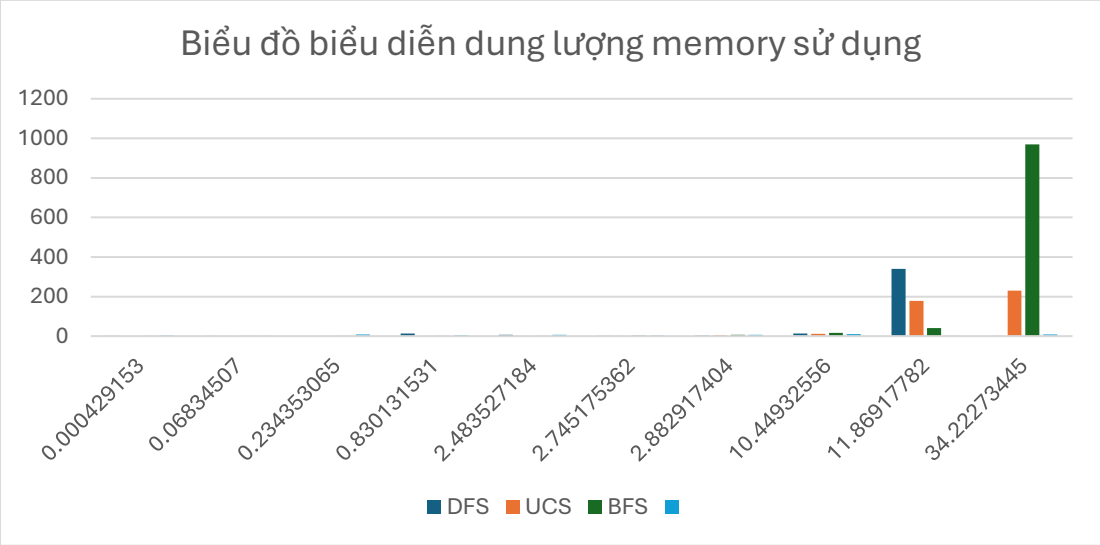


- Dung lượng Memory sử dụng:

Testcase	BFS	DFS	UCS	A*
1	0.05469	0.44141	0.00391	0.06835
2	40.375	339.973	177.523	11.8692
3	3.1875	2.21875	1.89844	2.74518
4	0.01563	0.03125	0	0.00043
5	0.1933	12.9531	0.13436	0.83013
6	6.55859	3.56641	4.83203	2.88292
7	0.18319	7.07031	0.10818	2.48353
8	0.205	0	0.22074	0.23435
9	969.543	1.19531	230.797	34.2227
10	16.4883	12.6992	11.6836	10.4493

Bảng giá trị dung lượng memory các thuật toán sử dụng tương ứng 10 testcase – đơn vị MB.

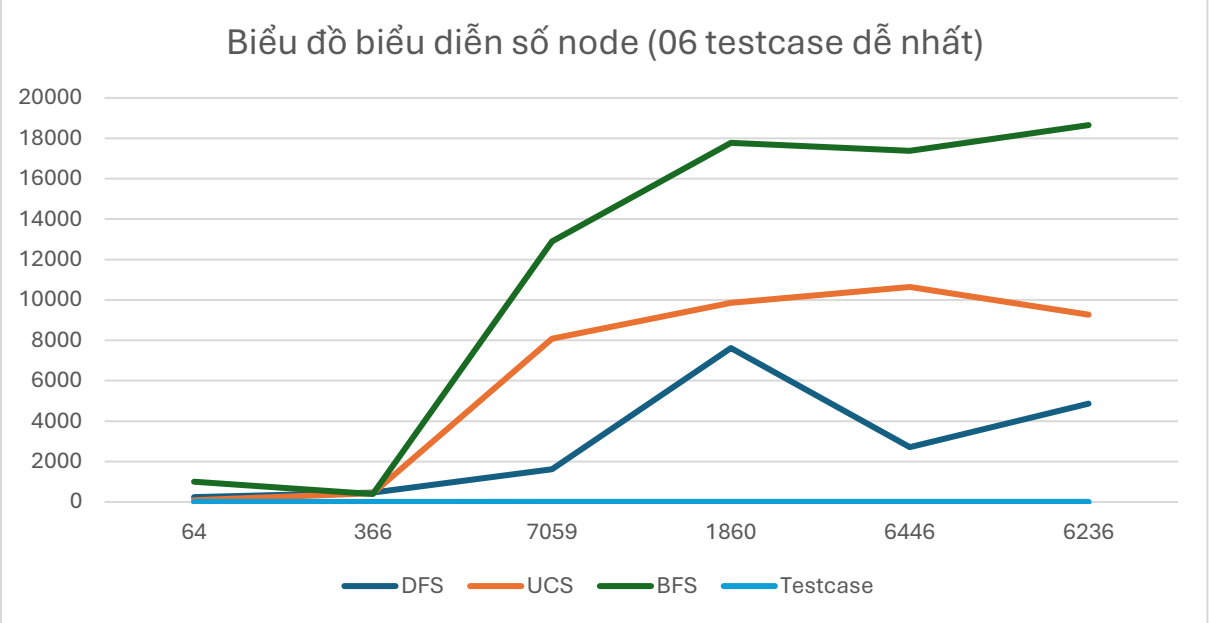


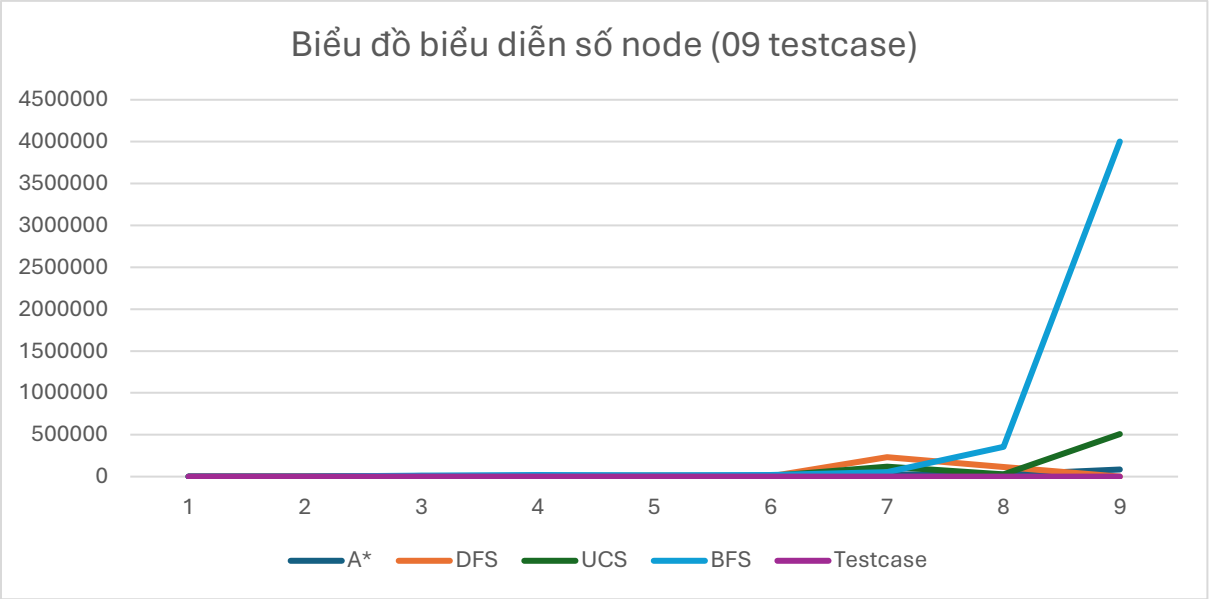


- Số lượng Node:

Testcase	A*	BFS	DFS	UCS
8	64	996	239	85
1	366	391	455	427
3	7059	12894	1622	8080
5	1860	17780	7617	9849
6	6446	17377	2709	10641
7	6236	18653	4869	9274
2	27218	53996	232131	117644
10	24806	356758	114504	27935
9	85000	4001609	2160	508080

Bảng giá trị số node từng thuật toán triển khai tương ứng từng testcase, không tính testcase 04 – bài toán sai (deadlock).


Biểu đồ biểu diễn số node (06 testcase dễ nhất)
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
64
366
7059
1860
6446
6236
DFS
UCS
BFS
Testcase


Biểu đồ biểu diễn số node (09 testcase)
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
1
2
3
4
5
6
7
8
9
A*
DFS
UCS
BFS
Testcase

Tổng kết:

Với từng tiêu chí, thứ tự sắp xếp các thuật toán theo hướng giảm dần tính hiệu quả từ trái sang phải:

- Thời gian:

A* > UCS > DFS > BFS

- Số node:

A* > DFS > UCS > BFS

- Chi phí đường đi:

A* > UCS > BFS > DFS

- Memory:

A* > DFS > UCS > BFS

--HẾT--