**VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY THE UNIVERSITY OF SCIENCE**



**FACULTY OF INFORMATION TECHNOLOGY**

BÁO CÁO ĐỒ ÁN

NHẬP MÔN TRÍ TUỆ NHÂN TẠO

LAB01

**Thành viên**:

Cao Anh Khoa - 22120161

Trần Anh Khoa - 22120164

Nguyễn Thanh Phong - 22120265

Điểu Kham - 19120441

**MỤC LỤC**

[1.Mô tả đồ án 4](#_Toc181825349)

[2.Đánh giá cá nhân 6](#_Toc181825350)

[3.Mô tả thuật toán 6](#_Toc181825351)

[a. BFS ( Breadth - First Search ) 6](#_Toc181825352)

[b. DFS ( Depth - First Search ) 8](#_Toc181825353)

[c. UCS ( Uniform - Cost Search ) 11](#_Toc181825354)

[d. A\* 12](#_Toc181825355)

[4.Testcase và kết quả 15](#_Toc181825356)

[a. Testcase 1 15](#_Toc181825357)

[b. Testcase 2 16](#_Toc181825358)

[c. Testcase 3 17](#_Toc181825359)

[d. Testcase 4 19](#_Toc181825360)

[e. Testcase 5 19](#_Toc181825361)

[f. Testcase 6 21](#_Toc181825362)

[g. Testcase 7 22](#_Toc181825363)

[h. Testcase 8 23](#_Toc181825364)

[i. Testcase 9 24](#_Toc181825365)

[j. Testcase 10 25](#_Toc181825366)

[Tổng Kết : 26](#_Toc181825367)

[5. Tài liệu tham khảo 26](#_Toc181825368)

[6. Video demo : 26](#_Toc181825369)

**MỤC LỤC HÌNH ẢNH**

[**Figure 1-Trò chơi hoàn chỉnh** 4](#_Toc181825234)

[**Figure 2-Giao diện đầu vào** 5](#_Toc181825235)

[**Figure 3-Giao diện chọn map** 5](#_Toc181825236)

[**Figure 4-Giao diện chọn thuật toán** 6](#_Toc181825237)

[**Figure 5-Lưu đồ thuật toán BFS** 8](#_Toc181825238)

[**Figure 6-Lưu đồ thuật toán DFS** 10](#_Toc181825239)

[**Figure 7-Lưu đồ thuật toán UCS** 12](#_Toc181825240)

[**Figure 8-Lưu đồ thuật toán A\*** 14](#_Toc181825241)

[**Figure 9-Input & Output testcase 1** 15](#_Toc181825242)

[**Figure 10 - Map 1** 15](#_Toc181825243)

[**Figure 11-Input & Output testcase 2** 16](#_Toc181825244)

[**Figure 12-Map2** 17](#_Toc181825245)

[**Figure 13-Input & Output testcase 3** 18](#_Toc181825246)

[**Figure 14-Map3** 18](#_Toc181825247)

[**Figure 15-Input & Output testcase 4** 19](#_Toc181825248)

[**Figure 16- Map4** 19](#_Toc181825249)

[**Figure 17-Input & Output testcase 5** 20](#_Toc181825250)

[**Figure 18-Map5** 20](#_Toc181825251)

[**Figure 19-Input & Output testcase 6** 21](#_Toc181825252)

[**Figure 20-Map6** 21](#_Toc181825253)

[**Figure 21-Input & Output testcase 7** 22](#_Toc181825254)

[**Figure 22-Map7** 22](#_Toc181825255)

[**Figure 23-Input & Output testcase 8** 23](#_Toc181825256)

[**Figure 24-Map8** 23](#_Toc181825257)

[**Figure 25-Input & Output testcase 9** 24](#_Toc181825258)

[**Figure 26-Map9** 24](#_Toc181825259)

[**Figure 27-Input & Output testcase 10** 25](#_Toc181825260)

[**Figure 28-Map10** 26](#_Toc181825261)

# 1.Mô tả đồ án

Ares, một nhà thám hiểm trẻ, quyết định đối mặt với một mê cung đầy thách thức để mở cánh cổng kho báu bí ẩn. Để làm được điều này, anh phải di chuyển các tảng đá lớn vào vị trí các công tắc bí mật trong mê cung phức tạp. Nhiệm vụ yêu cầu sử dụng các thuật toán tìm kiếm từ Trí tuệ Nhân tạo để hướng dẫn Ares vượt qua mê cung và đặt các tảng đá đúng chỗ để mở cánh cổng kho báu.

Mục tiêu là đặt tất cả tảng đá lên các công tắc. Để đạt được điều này, chúng em phải triển khai các thuật toán tìm kiếm như BFS, DFS, UCS, và A\* với heuristic, nhằm tìm ra cách tốt nhất giúp Ares vượt qua mê cung và hoàn thành nhiệm vụ.

Ở đồ án này, chúng em sử dụng :   
Python 3.12.2

Pygame 2.5.2

A screenshot of a video game

Description automatically generated

**Figure 1-Trò chơi hoàn chỉnh**

Giao diện đầu vào :

A screenshot of a video game

Description automatically generated

**Figure 2-Giao diện đầu vào**

Giao diện chọn map :

A pink screen with black text

Description automatically generated

**Figure 3-Giao diện chọn map**

Giao diện chọn thuật toán :

A screenshot of a pink screen

Description automatically generated

**Figure 4-Giao diện chọn thuật toán**

# 2.Đánh giá cá nhân

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Task** | **Tên** | **Mức độ hoàn thành** |
| 1 | BFS | Nguyễn Thanh Phong | 100% |
| 2 | DFS | Cao Anh Khoa | 100% |
| 3 | UCS | Nguyễn Thanh Phong | 100% |
| 4 | A\* | Điểu Kham | 100% |
| 5 | Testcase | Trần Anh Khoa | 100% |
| 6 | Result | Trần Anh Khoa | 100% |
| 7 | Video demo | Điểu Kham | 100% |
| 8 | Report | Cao Anh Khoa | 100% |
| Total |  |  | 100% |

# 3.Mô tả thuật toán

## BFS ( Breadth - First Search )

Tìm kiếm theo chiều rộng (BFS) là một thuật toán được sử dụng để duyệt đồ thị theo chiều rộng, khám phá tất cả các Node tiếp theo ở mức độ sâu hiện tại trước khi chuyển sang các Node ở mức độ sâu tiếp theo. Mục tiêu của BFS là tìm Node đích bằng cách khám phá các Node láng giềng gần nhất trước.

Các bước thực hiện thuật toán :

Bước 1. Khởi tạo :

- Bắt đầu bằng việc kiểm tra thời gian, nếu hết thời gian sẽ dừng BFS và trả về kết quả là "Timeout".

- Tạo một luồng phụ (`timeout\_thread`) để kiểm tra thời gian, sau đó khởi động luồng này.

- Tạo một nút bắt đầu (`start\_node`) từ trạng thái ban đầu của bài toán.

- Khởi tạo hàng đợi (`queue`) chứa các nút cần duyệt theo thứ tự BFS và một tập (`visited\_nodes`) để lưu các nút đã duyệt.

- Bắt đầu theo dõi bộ nhớ (`tracemalloc.start()`) và ghi lại thời gian bắt đầu.

Bước 2.Vòng lặp duyệt BFS:

- Trong vòng lặp, BFS kiểm tra lại thời gian để đảm bảo không vượt quá thời hạn quy định.

- Lấy nút đầu tiên từ hàng đợi (`queue.pop(0)`) để xử lý.

- Kiểm tra xem trạng thái của nút hiện tại có phải là trạng thái đích (mục tiêu) không:

+Nếu đúng, dừng luồng kiểm tra thời gian và xử lý kết quả thông qua hàm `process\_solution`, sau đó trả về kết quả.

+Nếu không phải là trạng thái đích, thêm nút hiện tại vào `visited\_nodes`.

- Khám phá các nút kế tiếp:

* Lặp qua tất cả các hành động có thể từ trạng thái hiện tại của `current\_node`.
* Tạo nút con (`new\_node`) bằng cách áp dụng mỗi hành động.
* Tăng bộ đếm số lượng nút đã được tạo ra (`generated\_node\_count`).
* Kiểm tra nếu `new\_node` chưa được duyệt (`visited\_nodes`) và chưa có trong hàng đợi (`queue`):
  + Nếu `new\_node` đạt trạng thái đích, dừng luồng kiểm tra thời gian và xử lý kết quả thông qua `process\_solution`, sau đó trả về kết quả.
  + Nếu không, thêm `new\_node` vào hàng đợi để tiếp tục duyệt trong vòng lặp tiếp theo.

Bước 3. Kết thúc:

- Nếu hàng đợi trống mà chưa tìm thấy trạng thái đích, BFS sẽ trả về `None`, nghĩa là không có đường đi nào tới mục tiêu được tìm thấy trong thời gian giới hạn.

A diagram of a software flowchart

Description automatically generated

**Figure 5-Lưu đồ thuật toán BFS**

## DFS ( Depth - First Search )

Tìm kiếm theo chiều sâu (Depth-First Search - DFS) là một thuật toán được sử dụng để duyệt đồ thị theo chiều sâu, khám phá càng xa càng tốt dọc theo mỗi nhánh. Mục tiêu của DFS là tìm một Node đích bằng cách đi sâu vào từng đường đi trước khi quay lại.

*Lưu ý:* DFS dừng lại khi tìm thấy Node đích.

Các bước thực hiện thuật toán :

Bước 1. Khởi tạo:

- Bắt đầu bằng việc kiểm tra thời gian; nếu hết thời gian thì dừng DFS và trả về kết quả "Timeout".

- Tạo một luồng phụ (`timeout\_thread`) để kiểm tra thời gian và khởi động luồng này.

- Tạo một nút bắt đầu (`start\_node`) từ trạng thái ban đầu của bài toán.

- Khởi tạo ngăn xếp (`stack`) để lưu các nút cần duyệt theo thứ tự DFS và tập hợp (`visited\_nodes`) để lưu các nút đã được khám phá.

- Bắt đầu theo dõi bộ nhớ (`tracemalloc.start()`) và ghi lại thời gian bắt đầu.

Bước 2. Vòng lặp duyệt DFS:

- Trong vòng lặp, kiểm tra thời gian để đảm bảo không vượt quá thời gian giới hạn.

- Lấy nút cuối cùng từ ngăn xếp (`stack.pop()`) để xử lý (theo cơ chế LIFO - Last In First Out).

- Kiểm tra xem trạng thái của `current\_node` có phải là trạng thái đích hay không:

+ Nếu đúng, dừng luồng kiểm tra thời gian và gọi hàm `process\_solution` để xử lý và trả về kết quả.

+ Nếu không phải trạng thái đích, thêm nút hiện tại vào `visited\_nodes`.

- Khám phá các nút con:

* Lặp qua tất cả các hành động có thể từ trạng thái hiện tại của `current\_node`.
* Tạo nút con (`new\_node`) bằng cách áp dụng mỗi hành động.
* Tăng bộ đếm số lượng nút được tạo ra (`generated\_node\_count`).
* Kiểm tra nếu `new\_node` chưa được duyệt (`visited\_nodes`) và chưa có trong ngăn xếp (`stack`):
  + Nếu `new\_node` đạt trạng thái đích, dừng luồng kiểm tra thời gian và gọi `process\_solution` để xử lý kết quả, sau đó trả về.
  + Nếu không, thêm `new\_node` vào ngăn xếp để tiếp tục duyệt sâu hơn trong vòng lặp tiếp theo.

Bước 3. Kết thúc:

- Nếu ngăn xếp trống mà vẫn chưa tìm thấy trạng thái đích, DFS sẽ trả về `None`, nghĩa là không có đường đi nào tới mục tiêu trong thời gian giới hạn.

**A diagram of a flowchart

Description automatically generated**

**Figure 6-Lưu đồ thuật toán DFS**

## UCS ( Uniform - Cost Search )

Tìm kiếm chi phí (Uniform-Cost Search - UCS) tìm đường đi có chi phí thấp nhất từ một Node bắt đầu (nút nguồn) đến một Node đích (nút mục tiêu) trong một đồ thị có trọng số. Đường đi được tìm thấy là đường đi tối ưu với chi phí thấp nhất dựa trên tổng chi phí của đường đi.

*Lưu ý:* UCS dừng lại khi nút đích được mở rộng (khi nút đích là nút được lấy ra khỏi hàng đợi ưu tiên).

Các bước thực hiện thuật toán :

Bước 1. Khởi tạo:

- Kiểm tra nếu hết thời gian, dừng UCS và trả về kết quả "Timeout".

- Tạo một luồng phụ (`timeout\_thread`) để kiểm tra thời gian, sau đó khởi động luồng này.

- Tạo một nút bắt đầu (`start\_node`) từ trạng thái ban đầu của bài toán.

- Khởi tạo một hàng đợi ưu tiên (`priority\_queue`) để quản lý các nút theo chi phí (tổng chi phí từ nút gốc đến nút hiện tại).

- Tập hợp `visited\_nodes` để lưu trữ các nút đã khám phá cùng với chi phí nhỏ nhất đến các nút này và nút cha trước đó.

- Bắt đầu theo dõi bộ nhớ (`tracemalloc.start()`) và ghi lại thời gian bắt đầu.

Bước 2. Vòng lặp tìm kiếm UCS:

- Trong vòng lặp, kiểm tra lại thời gian để đảm bảo không vượt quá thời gian giới hạn.

- Lấy nút có chi phí thấp nhất từ hàng đợi ưu tiên (`heapq.heappop(priority\_queue)`).

- Kiểm tra xem trạng thái của `current\_node` có phải là trạng thái đích hay không:

+ Nếu đúng, dừng luồng kiểm tra thời gian và xử lý kết quả thông qua `process\_solution`, sau đó trả về kết quả.

+ Nếu sai -> khám phá các nút con:

* Lặp qua tất cả các hành động có thể từ trạng thái hiện tại của `current\_node`.
* Tạo nút con (`new\_node`) bằng cách áp dụng mỗi hành động.
* Tăng bộ đếm số lượng nút đã được tạo (`generated\_node\_count`).
* Kiểm tra nếu `new\_node` chưa được khám phá hoặc nếu chi phí của `new\_node` nhỏ hơn chi phí hiện tại trong `visited\_nodes`:
* Nếu có chi phí nhỏ hơn, cập nhật `visited\_nodes` với chi phí mới và cha của nút (`current\_node`).
* Thêm `new\_node` vào hàng đợi ưu tiên (`priority\_queue`) để tiếp tục mở rộng trong vòng lặp tiếp theo.

Bước 3. Kết thúc :

- Nếu hàng đợi ưu tiên trống mà chưa tìm thấy trạng thái đích, UCS sẽ trả về `None`, nghĩa là không có đường đi nào tới mục tiêu được tìm thấy trong thời gian giới hạn.

A diagram of a process

Description automatically generated

**Figure 7-Lưu đồ thuật toán UCS**

## A\*

Bước 1. Khởi tạo:

- Kiểm tra nếu hết thời gian, dừng A\* và trả về kết quả "Timeout".

- Tạo một luồng phụ (`timeout\_thread`) để kiểm tra thời gian và khởi động luồng này.

- Tạo một nút bắt đầu (`start\_node`) từ trạng thái ban đầu của bài toán.

- Khởi tạo hàng đợi ưu tiên (`priority\_queue`) để quản lý các nút theo độ ưu tiên, dựa trên chi phí dự đoán của nút (tổng chi phí từ gốc đến đích).

- Tập hợp `visited\_nodes` lưu trữ các nút đã khám phá, với chi phí nhỏ nhất và nút trước đó.

- Bắt đầu theo dõi bộ nhớ (`tracemalloc.start()`) và ghi lại thời gian bắt đầu.

Bước 2. Vòng lặp tìm kiếm A\*:

- Kiểm tra lại thời gian trong vòng lặp để đảm bảo không vượt quá thời gian giới hạn.

- Lấy nút có chi phí ưu tiên thấp nhất từ hàng đợi (`heapq.heappop(priority\_queue)`).

- Kiểm tra nếu `current\_node` là trạng thái đích:

+ Nếu đúng, dừng luồng kiểm tra thời gian và xử lý kết quả qua `process\_solution`, sau đó trả về.

+ Nếu sai -> Khám phá các nút con:

* Lặp qua tất cả các hành động có thể từ trạng thái hiện tại của `current\_node`.
* Tạo nút con (`new\_node`) bằng cách áp dụng mỗi hành động, kèm theo chi phí dự đoán (`use\_heuristic=True`).
* Tăng bộ đếm số lượng nút được tạo (`generated\_node\_count`).
* Kiểm tra nếu `new\_node` chưa được khám phá hoặc nếu chi phí của nó thấp hơn chi phí hiện tại trong `visited\_nodes`:
* Nếu đúng, cập nhật `visited\_nodes` với chi phí và cha của `new\_node`.
* Thêm `new\_node` vào hàng đợi ưu tiên (`priority\_queue`) để tiếp tục mở rộng trong vòng lặp tiếp theo.

Bước 3. Kết thúc:

- Nếu hàng đợi ưu tiên trống mà chưa tìm thấy trạng thái đích, A\* sẽ trả về `None`, nghĩa là không có đường đi nào tới mục tiêu trong thời gian giới hạn.

A diagram of a flowchart

Description automatically generated

**Figure 8-Lưu đồ thuật toán A\***

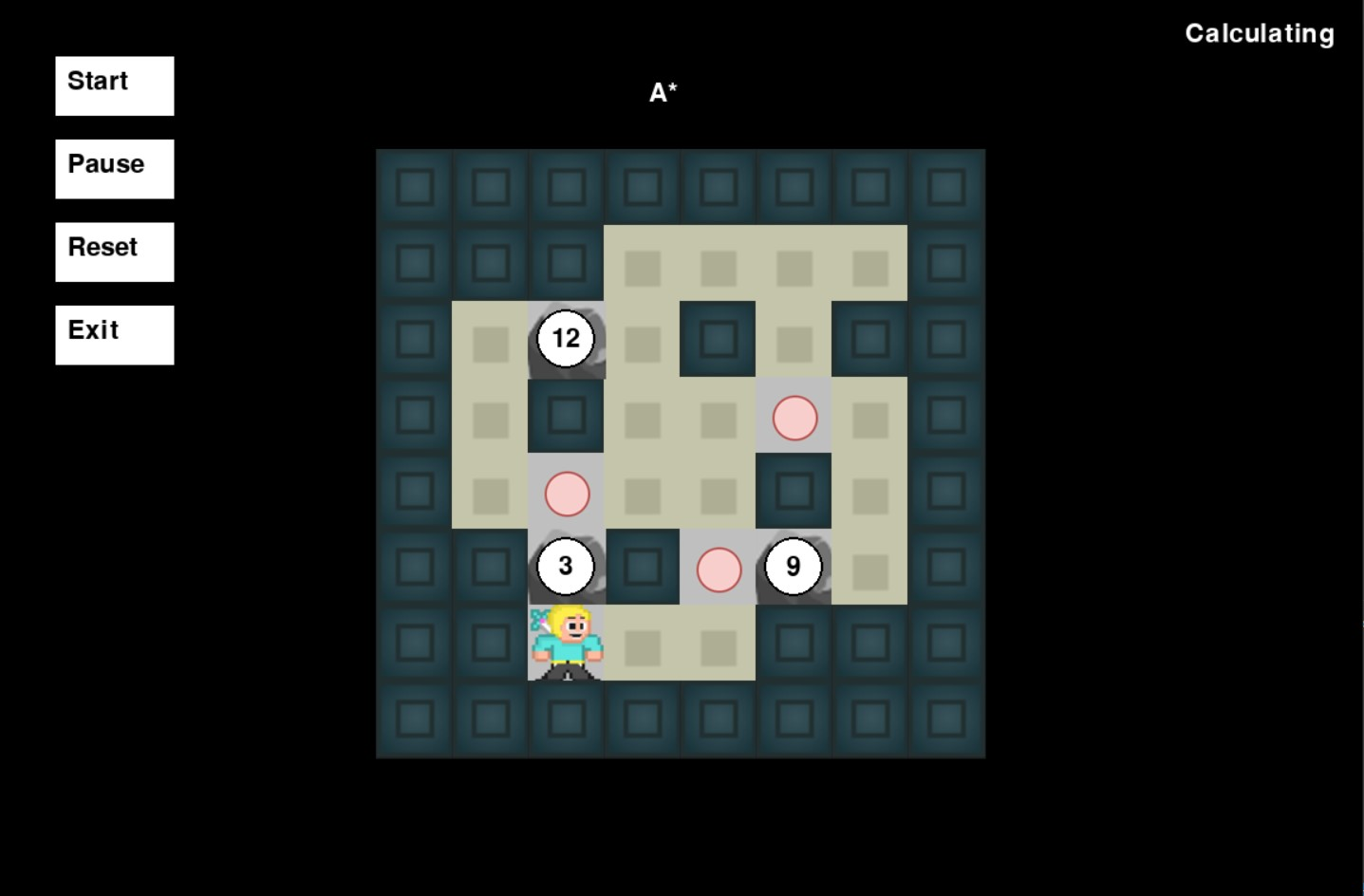
# 4.Testcase và kết quả

## Testcase 1

A screenshot of a computer program

Description automatically generated

**Figure 9-Input & Output testcase 1**



**Figure 10 - Map 1**

Nhận xét :

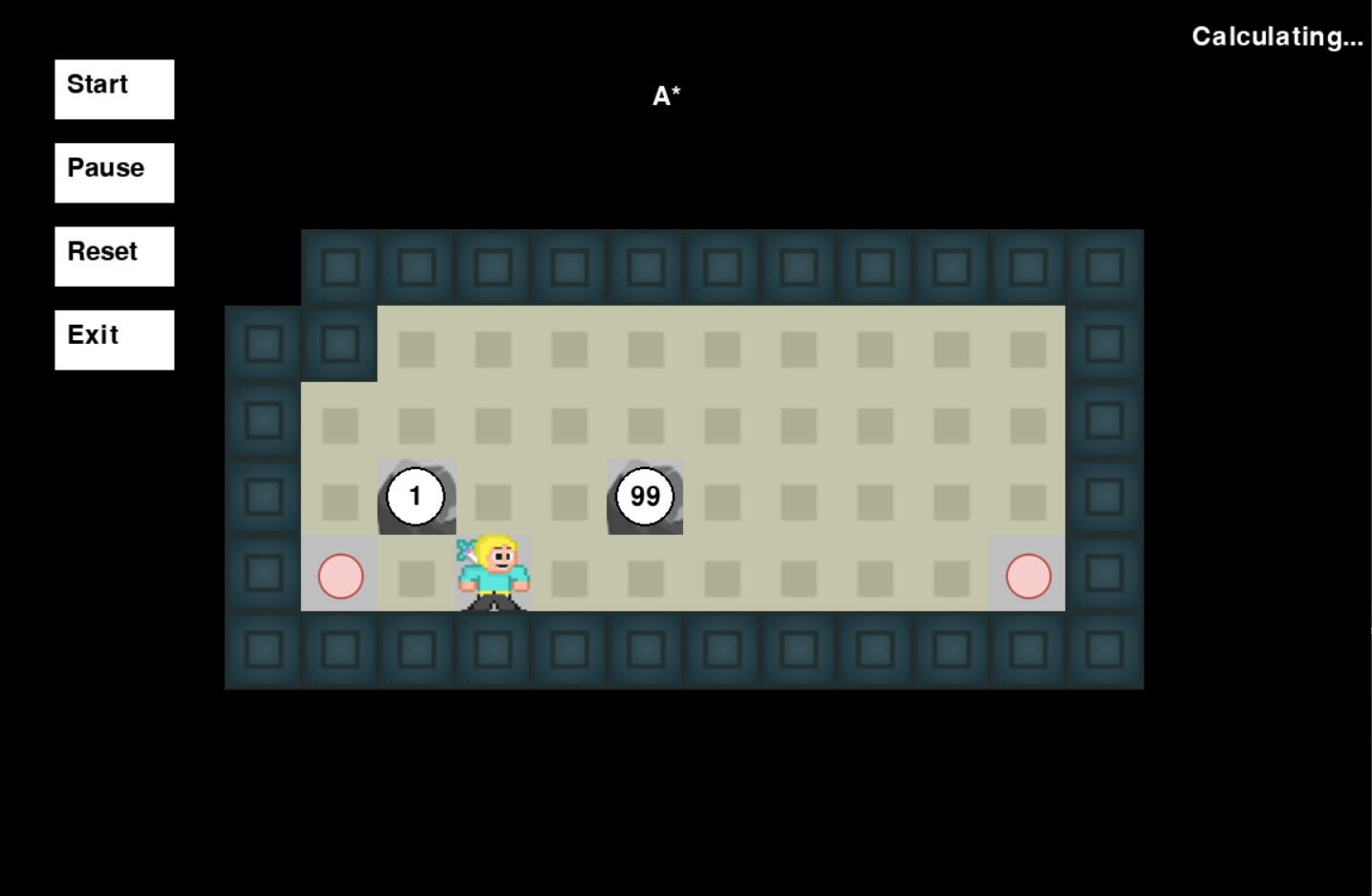
* A\* là thuật toán tốt nhất trong trường hợp này, vì nó tìm ra đường đi tối ưu về chi phí và hiệu quả hơn trong việc sử dụng tài nguyên (thời gian và bộ nhớ).
* UCS cũng tìm ra đường đi tối ưu nhưng mở rộng nhiều nút hơn A\*, nên kém hiệu quả hơn.
* BFS tìm được đường đi ngắn nhất về số bước nhưng không tối ưu về chi phí và tốn nhiều tài nguyên hơn A\*.
* DFS là thuật toán kém hiệu quả nhất trong trường hợp này, với số nút mở rộng lớn và chi phí cao nhất.

## Testcase 2

A screenshot of a computer program

Description automatically generated

**Figure 11-Input & Output testcase 2**



**Figure 12-Map2**

* **A**\* là thuật toán hiệu quả nhất trong trường hợp này, với số bước ít và chi phí tổng thấp hơn so với UCS.
* **UCS** tìm được đường đi tối ưu về số bước nhưng tiêu tốn nhiều tài nguyên (thời gian và bộ nhớ).
* **BFS** có số bước thấp nhưng chi phí cao hơn và mở rộng nhiều nút.
* **DFS** sử dụng ít bộ nhớ và thời gian, nhưng không tìm được đường đi tối ưu.

## Testcase 3

A screenshot of a computer program

Description automatically generated

**Figure 13-Input & Output testcase 3**

A screenshot of a video game

Description automatically generated

**Figure 14-Map3**

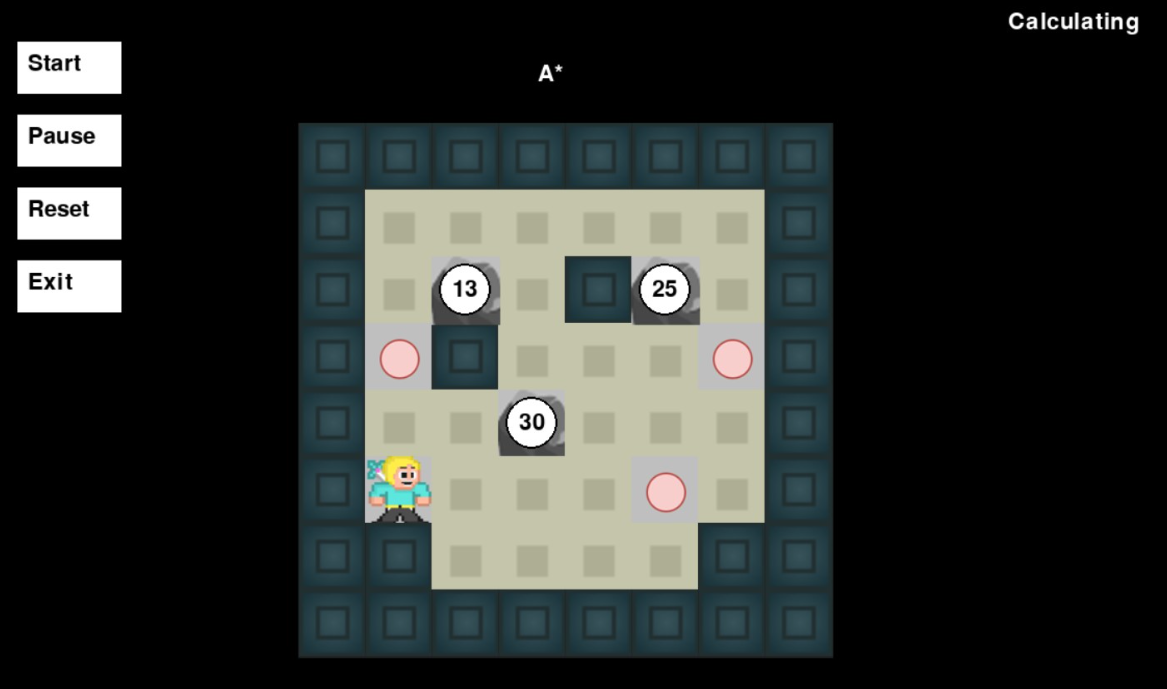
* **A**\* là lựa chọn tốt nhất trong trường hợp này, vì nó tìm ra đường đi tối ưu về chi phí với số nút mở rộng ít và thời gian xử lý ngắn nhất.
* **UCS** và **BFS** tìm được con đường tối ưu về số bước và chi phí nhưng mở rộng quá nhiều nút và tiêu tốn thời gian đáng kể.
* **DFS** không hiệu quả trong việc tìm kiếm đường đi tối ưu, với chi phí và số bước lớn nhất.

## Testcase 4

A computer screen with white text

Description automatically generated

**Figure 15-Input & Output testcase 4**



**Figure 16- Map4**

* **UCS** tìm được đường đi tối ưu với số bước ít và chi phí tối thiểu, nhưng không hiệu quả về thời gian và bộ nhớ do mở rộng quá nhiều nút.
* **BFS** và **DFS** đều không thành công trong việc tìm kiếm, dẫn đến timeout.
* **A**\* là thuật toán duy nhất tìm thấy giải pháp, với chi phí tối ưu và thời gian xử lý ngắn.

## Testcase 5

A screenshot of a computer program

Description automatically generated

**Figure 17-Input & Output testcase 5**

A screenshot of a video game

Description automatically generated

**Figure 18-Map5**

* **UCS** và **BFS** có số bước đi và trọng số tương đương, nhưng UCS có số nút tìm kiếm lớn hơn và thời gian thực hiện nhanh hơn.
* **DFS** có số bước đi cao nhất và thời gian tìm kiếm lâu nhất, cho thấy rằng nó có thể không phải là lựa chọn tối ưu cho không gian trạng thái lớn.
* **A**\* là thuật toán hiệu quả nhất về thời gian (1001.32 ms) và bộ nhớ (5.66 MB), đồng thời có số bước đi thấp hơn UCS và DFS.

## Testcase 6

A screenshot of a computer

Description automatically generated

**Figure 19-Input & Output testcase 6**

A screenshot of a video game

Description automatically generated

**Figure 20-Map6**

* **Timeout** cho UCS, BFS và DFS cho thấy rằng trong không gian trạng thái này, các thuật toán này gặp khó khăn trong việc tìm kiếm, có thể do độ phức tạp cao hoặc không gian tìm kiếm lớn.
* **A**\* là thuật toán duy nhất hoạt động thành công, với 42 bước đi. Mặc dù thời gian tìm kiếm (7034.69 ms) và bộ nhớ (238.58 MB) tương đối cao, nó vẫn có thể tìm ra đường đi.

## Testcase 7

A screenshot of a computer program

Description automatically generated

**Figure 21-Input & Output testcase 7**

A screenshot of a video game

Description automatically generated

**Figure 22-Map7**

* **BFS** có số bước thấp nhất (70) nhưng lại tốn nhiều thời gian nhất (8040.87 ms). Điều này có thể cho thấy rằng trong khi BFS khám phá tất cả các nút theo từng lớp, nó có thể mất nhiều thời gian hơn trong các lưới lớn hoặc phức tạp.
* **DFS** có số bước cao thứ hai nhưng sử dụng bộ nhớ ít nhất (8.14 MB), cho thấy nó khám phá sâu hơn mà không giữ nhiều thông tin trạng thái.
* **UCS** và A\* có hiệu suất tương tự về số bước, nhưng A\* có hồ sơ bộ nhớ tốt hơn so với UCS.

## Testcase 8

A screenshot of a computer program

Description automatically generated

**Figure 23-Input & Output testcase 8**

A screenshot of a video game

Description automatically generated

**Figure 24-Map8**

* **UCS**, **BFS**, và **A**\* đều có số bước giống nhau (26), nhưng khác nhau về số lượng nút đã khám phá và bộ nhớ sử dụng.
* **UCS** và **A\*** có cùng trọng số (203) và thời gian thực hiện gần giống nhau, trong khi A\* tiêu tốn ít bộ nhớ nhất (2.05 MB).
* **DFS** có số bước cao hơn (53) và trọng số lớn hơn (375), cho thấy nó khám phá nhiều nút hơn mà không tối ưu hóa chi phí.

## Testcase 9

A screenshot of a computer program

Description automatically generated

**Figure 25-Input & Output testcase 9**

**A screenshot of a video game

Description automatically generated**

**Figure 26-Map9**

* **UCS** và **A**\* có số bước giống nhau (54) và cùng trọng số (199), nhưng A\* tiêu tốn ít bộ nhớ hơn (1.81 MB) và thời gian nhanh hơn (1000.58 ms) so với UCS (2005.45 ms).
* **DFS** có số bước cao hơn đáng kể (148) và trọng số lớn hơn (757), cho thấy nó đã khám phá nhiều nút hơn nhưng cũng tiêu tốn nhiều thời gian và bộ nhớ hơn.
* **BFS** đã bị timeout, có nghĩa là thuật toán không hoàn thành trong thời gian cho phép, điều này có thể xảy ra trong các bài toán lớn hoặc phức tạp.

## Testcase 10

A screenshot of a computer

Description automatically generated

**Figure 27-Input & Output testcase 10**

A screenshot of a video game

Description automatically generated

**Figure 28-Map10**

* **UCS, BFS,** và **DFS** không thành công trong việc tìm kiếm do timeout, cho thấy rằng trong các tình huống phức tạp, các thuật toán này có thể không hiệu quả.
* **A**\* là thuật toán duy nhất tìm thấy giải pháp, mặc dù nó tiêu tốn nhiều tài nguyên. Tuy nhiên, điều này cho thấy **A**\* có khả năng hoạt động tốt hơn trong không gian tìm kiếm phức tạp.

## Tổng Kết :

- **A\*:** Hiệu quả nhất, tìm đường nhanh, tối ưu chi phí với ít bộ nhớ. Phù hợp cho bài toán phức tạp.

- **UCS**: Tìm đường tối ưu nhưng chậm và dễ bị timeout khi bài toán phức tạp.

- **BFS**: Đảm bảo đường đi ngắn nhất trong lưới không trọng số nhưng tốn bộ nhớ, dễ timeout.

- **DFS**: Tiết kiệm bộ nhớ, không đảm bảo tối ưu và dễ rơi vào đường cụt hoặc timeout.

# Tài liệu tham khảo

GeeksForGeek:<https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/>

ChatGPT : [open.ai](file:///C:\Users\KitOfoso\Downloads\open.ai)

ViettelIDC: <https://viettelidc.com.vn/tin-tuc/cam-nang-ai-cac-thuat-toan-tim-kiem-pho-bien-trong-ai>

# Video demo :

<https://www.youtube.com/watch?v=vjQOhejXR_o>