ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

ĐỒ ÁN TỔNG HỢP -
HƯỚNG TRÍ TUỆ NHÂN TẠO (CO3031)

Ứng dụng trí tuệ nhân tạo trong tìm đường mê cung

GVHD:	Trần Ngọc Bảo Duy	
Lớp:	L02	
SV thực hiện:	Phạm Đại Hoàng An	1912539
	Nguyễn Đăng Hải	1913254
	Trần Lê Viết Khánh	1913758
	Lương Đoàn Việt Hoàng	1913430
	Trần Khắc Huy	1913560

Tp. Hồ Chí Minh, Tháng 10/2021

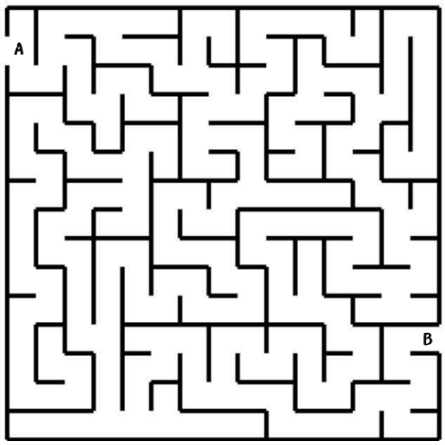
Mục lục

Danh sách hình vẽ

I. Giới thiệu đề tài

Trong khoa học máy tính, việc sử dụng một giải thuật phù hợp cho một bài toán cụ thể sẽ mang lại hiệu quả cao. Thiết kế một thuật toán tốt sẽ sử dụng ít tài nguyên hơn, thời gian xử lý và không gian bộ nhớ tốt hơn. Các giải thuật khác nhau cho một bài toán sẽ cho kết quả có thể giống nhau nhưng thời gian và tài nguyên sử dụng của mỗi thuật toán sẽ khác nhau. Để có thể đi sâu và nắm vững những kiến thức đã học, nhóm em chọn đề tài "***Ứng dụng trí tuệ nhân tạo trong tìm đường mê cung***" để tìm hiểu và nghiên cứu các giải thuật tìm kiếm từ đó đánh giá được giải thuật phù hợp trong việc tìm đường mê cung.

Mê cung là một hệ thống chứa rất nhiều nhánh, ngã rẽ đường khác nhau trong đó có cả ngõ cụt và chỉ có một đường thoát duy nhất. Bài toán tìm lối ra trong mê cung là một bài toán rất hiệu quả để đánh giá hiệu suất của một giải thuật tìm kiếm bất kì.



Hình 1: Mê cung biết trước lối vào và ra

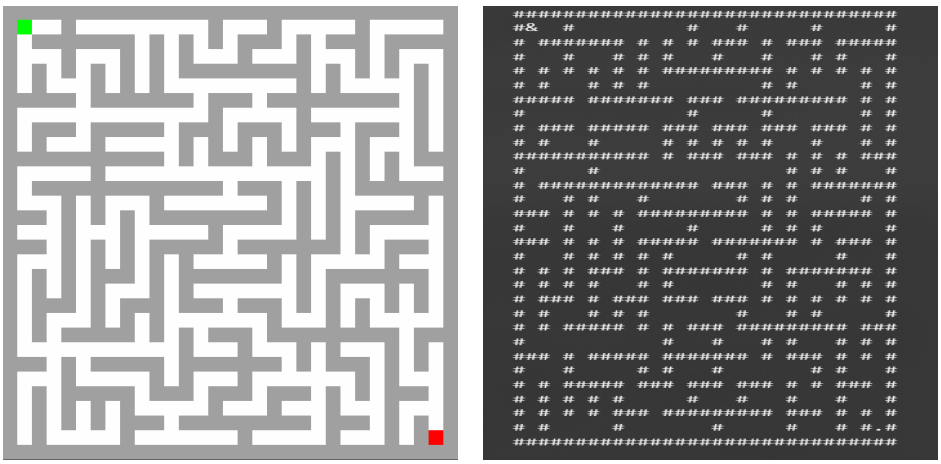
Tính chất: Có duy nhất một lối vào và một lối ra, có thể có nhiều cách đi khác nhau tùy theo thiết kế của mê cung và giải thuật chọn đường đi.

Ý nghĩa khoa học: Mê cung gắn với những câu chuyện thần thoại hay thực tế đã hấp dẫn rất nhiều nhà toán học từ đó hình thành nên những bài toán tìm đường mê cung và các giải thuật để tìm đường.

Phạm vi ứng dụng: Xây dựng các mật đạo, các mê cung giải trí, ứng dụng trong các trò chơi điện tử,...

II. Mô tả bài toán

Yêu cầu bài toán: Xây dựng 1 mê cung 2 chiều có một điểm đầu và điểm đích cố định, giả sử người tìm đường biết trước vị trí bắt đầu và điểm đích. Biểu diễn ma trận mê cung 2 chiều với giá trị "#" là tường, " " là đường đi, "&" là điểm bắt đầu và "." là điểm cần đến.

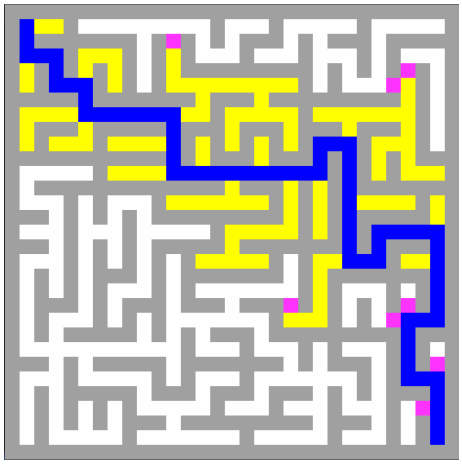


Hình 2: Ma trận mê cung

Sử dụng các giải thuật tìm kiếm như BFS, A*, bidirectional A* và greedy để giải quyết vấn đề tìm đường đi từ điểm bắt đầu đến đích và tránh vật cản. Với mỗi giải thuật khác nhau sẽ cho ra đường đi khác nhau cùng với thời gian tương ứng, và những ưu điểm/nhược điểm của mỗi giải thuật.

Input: Ma trận mê cung, tọa độ đầu và tọa độ cuối.

Output: Đường đi thoát khỏi mê cung, các đường đi có thể xảy ra, thời gian thực thi chương trình.



Hình 3: Minh họa output

III. Cơ sở lý thuyết

1. Giới thiệu giải thuật tìm kiếm

Trong ngành khoa học máy tính, một giải thuật tìm kiếm là một thuật toán lấy đầu vào là một bài toán và trả về kết quả là một lời giải cho bài toán đó, thường là sau khi cân nhắc giữa một loạt các lời giải có thể. Hầu hết các thuật toán được nghiên cứu bởi các nhà khoa học máy tính để giải quyết các bài toán đều là các thuật toán tìm kiếm.

Tập hợp tất cả các lời giải có thể đối với một bài toán được gọi là không gian tìm kiếm. Thuật toán thử sai (brute-force search) hay các thuật toán tìm kiếm "sơ đẳng" không có thông tin sử dụng phương pháp đơn giản nhất và trực quan nhất. Trong khi đó, các thuật toán tìm kiếm có thông tin sử dụng heuristics để áp dụng các tri thức về cấu trúc của không gian tìm kiếm nhằm giảm thời gian cần thiết cho việc tìm kiếm.

Thuật toán tìm kiếm thích hợp thường phụ thuộc vào cấu trúc dữ liệu được tìm kiếm và cũng có thể bao gồm kiến thức trước đó về dữ liệu. Một số cấu trúc cơ sở dữ liệu được xây dựng đặc biệt để làm cho các thuật toán tìm kiếm nhanh hơn hoặc hiệu quả hơn, chẳng hạn như search tree, hash map hoặc database index.

Thuật ngữ Search Algorithm:

- **Search:** Tìm kiếm là quy trình từng bước để giải quyết vấn đề tìm kiếm trong một không gian tìm kiếm nhất định. Một vấn đề tìm kiếm có thể có ba yếu tố chính:
  - **Search Space:** đại diện cho một tập hợp các giải pháp khả thi mà một hệ thống có thể có.
  - **Start State:** Là trạng thái mà từ đó tác nhân bắt đầu tìm kiếm.
  - **Goal test:** Là một chức năng quan sát trạng thái hiện tại và trả về trạng thái mục tiêu có đạt được hay không.
- **Search tree:** Cây biểu diễn bài toán tìm kiếm được gọi là cây Tìm kiếm. Gốc của cây tìm kiếm là nút gốc tương ứng với trạng thái ban đầu.
- **Actions:** Nó cung cấp mô tả về tất cả các hành động có sẵn cho tác nhân.
- **Transition model:** Mô tả về những gì mỗi hành động thực hiện, có thể được biểu diễn dưới dạng mô hình chuyển tiếp.
- **Path Cost:** Nó là một hàm chỉ định chi phí số cho mỗi đường dẫn.
- **Solution:** Đây là một chuỗi hành động dẫn từ nút bắt đầu đến nút mục tiêu.
- **Optimal Solution:** Một giải pháp có chi phí thấp nhất trong tất cả các giải pháp.

Thuộc tính của thuật toán tìm kiếm:

Tính đầy đủ: Thuật toán tìm kiếm được cho là hoàn chỉnh nếu nó đảm bảo trả về một giải pháp nếu có ít nhất bất kỳ giải pháp nào tồn tại cho bất kỳ đầu vào ngẫu nhiên nào.

Tính tối ưu: Nếu một giải pháp được tìm thấy cho một thuật toán được đảm bảo là giải pháp tốt nhất (chi phí đường dẫn thấp nhất) trong số tất cả các giải pháp khác, thì một giải pháp như vậy được cho là một giải pháp tối ưu.

Độ phức tạp về thời gian: Độ phức tạp về thời gian là thước đo thời gian để một thuật toán hoàn thành nhiệm vụ của nó.

Độ phức tạp về không gian: Đây là không gian lưu trữ tối đa cần thiết tại bất kỳ thời điểm nào trong quá trình tìm kiếm, tuỳ vào mức độ phức tạp của vấn đề.

2. Breadth-First Search(BFS)

Breadth-first search (BFS) là một thuật toán để duyệt qua hoặc tìm kiếm cấu trúc dữ liệu dạng cây hoặc đồ thị. Nó bắt đầu ở gốc cây (hoặc một số nút tùy ý của đồ thị, đôi khi được gọi là "khóa tìm kiếm") và khám phá các nút lân cận trước khi chuyển sang các nút lân cận cấp tiếp theo.

Thuật toán

Một số quy ước:

- *Open:* là tập hợp các đỉnh chờ được xét ở bước tiếp theo theo hàng đợi (hàng đợi: dãy các phần tử mà khi thêm phần tử vào sẽ thêm vào cuối dãy, còn khi lấy phần tử ra sẽ lấy ở phần tử đứng đầu dãy).
- *Close:* là tập hợp các đỉnh đã xét, đã duyệt qua.
- *s:* là đỉnh xuất phát, đỉnh gốc ban đầu trong quá trình tìm kiếm.
- *g:* đỉnh đích cần tìm.
- *p:* đỉnh đang xét, đang duyệt.

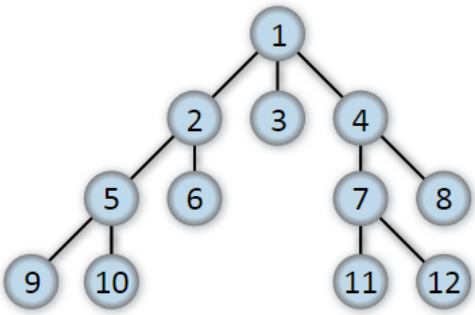
Trình bày thuật giải:

- Bước 1: Tập *Open* chứa đỉnh gốc *s* chờ được xét.
- Bước 2: Kiểm tra tập *Open* có rỗng không.
  - + Nếu tập *Open* không rỗng, lấy một đỉnh ra khỏi tập *Open* làm đỉnh đang xét *p*. Nếu *p* là đỉnh *g* cần tìm, kết thúc tìm kiếm.
  - + Nếu tập *Open* rỗng, tiến đến bước 4.
- Bước 3: Đưa đỉnh *p* vào tập *Close*, sau đó xác định các đỉnh kề với đỉnh *p* vừa xét. Nếu các đỉnh kề không thuộc tập *Close*, đưa chúng vào cuối tập *Open*. Quay lại bước 2.

– Bước 4: Kết luận không tìm ra đỉnh đích cần tìm.

Mô phỏng trên đồ thị

Biểu đồ sau đây cho thấy thứ tự mà các nút được phát hiện trong BFS:



Hình 4: Tìm kiếm theo chiều rộng

Ưu điểm:

– Xét duyệt tất cả các đỉnh để trả về kết quả.

– Nếu số đỉnh là hữu hạn, thuật toán chắc chắn tìm ra kết quả.

– Thuật toán luôn trả về đường đi ngắn nhất

Khuyết điểm:

– Mang tính chất vét cạn, không nên áp dụng nếu duyệt số đỉnh quá lớn.

– Mang tính chất mù quáng, duyệt tất cả đỉnh, không chú ý đến thông tin trong các đỉnh để duyệt hiệu quả, dẫn đến duyệt qua các đỉnh không cần thiết.

3. A* Search

A* là giải thuật tìm kiếm trong đồ thị, tìm đường đi từ một đỉnh hiện tại đến đỉnh đích có sử dụng hàm để ước lượng khoảng cách hay còn gọi là hàm Heuristic.

Từ trạng thái hiện tại A* xây dựng tất cả các đường đi có thể đi dùng hàm ước lược khoảng cách (hàm Heuristic) để đánh giá đường đi tốt nhất có thể đi. Tùy theo mỗi dạng bài khác nhau mà hàm Heuristic sẽ được đánh giá khác nhau. A* luôn tìm được đường đi ngắn nhất nếu tồn tại đường đi như thế.

Thứ tự ưu tiên cho một đường đi đươc quyết định bởi hàm Heuristic được đánh giá $f(x) = g(x) + h(x)$

– g(x) là chi chi phí của đường đi từ điểm xuất phát cho đến thời điểm hiện tại.

- h(x) là hàm ước lượng chi phí từ đỉnh hiện tại đến đỉnh đích f(x) thường có giá trị càng thấp thì độ ưu tiên càng cao.

Thuật toán

Một số quy ước:

- *Open*: tập các trạng thái đã được sinh ra nhưng chưa được xét đến.
- *Close*: tập các trạng thái đã được xét đến.
- *p*: đỉnh đang xét, đang duyệt.
- *q*: đỉnh đích đến tiếp theo.
- *Cost(p,q)*: Khoảng cách từ điểm p đến điểm q.

Trình bày thuật giải:

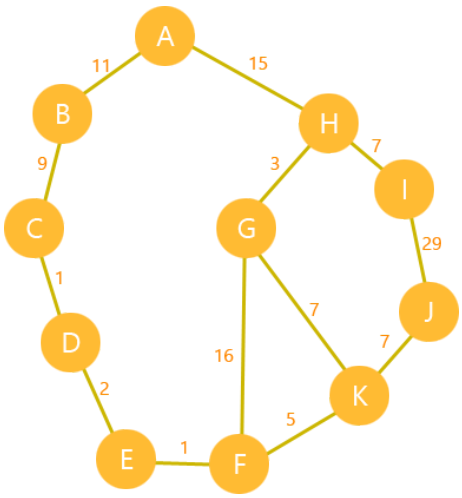
- Bước 1: Tập *Open* chứa điểm gốc là điểm bắt đầu, lúc này điểm bắt đầu là điểm đang xét.
  + Open:= {p}
  + Close : ={ }
- Bước 2: Kiểm tra *Open* có rỗng không, nếu rỗng tiến đến bước 3.
  + Chọn trạng thái (đỉnh) tốt nhất p trong Open (xóa p khỏi Open).
  + Nếu p là trạng thái kết thúc thì thoát.
  + Chuyển p qua Close và tạo ra các trạng thái kế tiếp q sau p.
    * Nếu q đã có trong Open
      · Nếu g(q) > g(p) + Cost(p, q):
      g(q) = g(p) + Cost(p, q)
      f(q) = g(q) + h(q)
      prev(q) = p (đỉnh cha của q là p)
    * Nếu q chưa có trong Open:
      · g(q) = g(p) + cost(p, q)
      · f(q) = g(q) + h(q)
      · prev(q) = p
      · Thêm q vào Open
    * Nếu q có trong Close:

· Nếu g(q) > g(p) + Cost(p, q):
Bỏ q khỏi Close
Thêm q vào Open

– Bước 3: Không tìm được điểm đích.

Mô phỏng trên đồ thị

h(A) = 60 / h(B) = 53 / h(C) = 36 / h(D) = 35 / h(E) = 35 / h(F) = 19 / h(G) = 16 / h(H) = 38 / h(I) = 23 / h(J) = 0 / h(K) = 7



Hình 5: Mô phỏng đồ thị

– Đỉnh bắt đầu A.

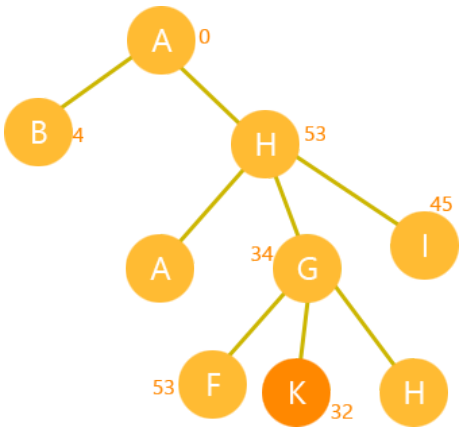
– Đỉnh kết thúc K.

– Ước lượng khoảng cách từ đỉnh hiện tại cho đến đỉnh kết thúc f(x)=g(x)+h(x) trong đó g là khoảng cách ngắn nhất từ đỉnh hiện tại đến đích. Ví dụ: f(A) = 0 + 60

Bước	P	Các đỉnh nối với P	Open	Close
0			A60	
1	A	B, H	B64, H53	A
2	H	G, I, A	B64, G34, I45	A, H
3	G	H, K, F	B64, I45, K32, F53	A, H, G
4	K	G, F, J	B64, J32, F49, I45	A, H, G
5	K(dừng)			

Cây tìm kiếm ứng với đồ thị trên:

Ưu điểm



Hình 6: A* Tree

Một thuật giải linh động, tổng quát, trong đó hàm chứa cả tìm kiếm chiều sâu, tìm kiếm chiều rộng và những nguyên lý Heuristic khác. Nhanh chóng tìm đến lời giải với sự định hướng của hàm Heuristic. Chính vì thế mà người ta thường nói A* chính là thuật giải tiêu biểu cho Heuristic.

Nhược điểm

A* rất linh động nhưng vẫn gặp một khuyết điểm cơ bản giống như chiến lược tìm kiếm chiều rộng đó là tốn khá nhiều bộ nhớ để lưu lại những trạng thái đã đi qua.

4. Bidirectional A* Search

Tìm kiếm hai chiều là một thuật toán tìm kiếm đồ thị mà tìm thấy một con đường ngắn nhất từ ban đầu đỉnh đến một đỉnh mục tiêu trong một đồ thị có hướng . Nó chạy đồng thời hai tìm kiếm: một tìm kiếm về phía trước so với trạng thái ban đầu và một tìm kiếm lùi lại từ mục tiêu, dừng lại khi cả hai gặp nhau.

Trong nhiều trường hợp sử dụng tìm kiếm hai chiều sẽ nhanh hơn, làm giảm đáng kể số lượng thăm dò cần thiết.Giả sử nếu hệ số phân nhánh của cây là b và khoảng cách của đỉnh mục tiêu từ nguồn là d , thì độ phức tạp tìm kiếm BFS / DFS thông thường sẽ là $O(b^d)$. Mặt khác, nếu chúng ta thực hiện hai thao tác tìm kiếm thì độ phức tạp sẽ là $O(b^{d/2})$ cho mỗi lần tìm kiếm và tổng độ phức tạp sẽ là $O(b^{d/2}+b^{d/2})$ nhỏ hơn nhiều so với $O(b^d)$.

Để xem xét sử dụng tìm kiếm hai chiều thì:

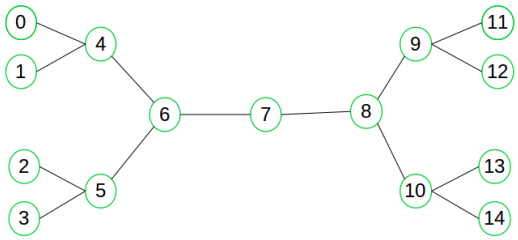
– Cả hai trạng thái ban đầu và mục tiêu là duy nhất và hoàn toàn được xác định.

– Hệ số phân nhánh hoàn toàn giống nhau theo cả hai hướng.

Thuật toán tìm kiếm hai chiều

Tìm kiếm hai chiều cũng dùng hàm ước lượng khoảng cách(Heuristic) để tìm đường đi ngắn nhất có thể. Nó tiến hành giải thuật tìm kiếm bắt đầu từ cả hai phía nguồn và mục tiêu, khi con đường đi qua nguồn và mục tiêu giao nhau thì tìm kiếm dùng lại. Đây là con đường ngắn nhất, nó làm giảm đáng kể thời gian tìm kiếm.

Xét ví dụ đơn giản sau:



Hình 7: Tìm kiếm hai chiều

- **Bước 1:** Giả sử, 0 là nút ban đầu và 14 là nút mục tiêu, và 7 là nút giao.
- **Bước 2:** Chúng ta sẽ bắt đầu tìm kiếm đồng thời từ nút bắt đầu đến nút mục tiêu và quay ngược lại từ mục tiêu đến nút bắt đầu.
- **Bước 3:** Bất cứ khi nào tìm kiếm tiến và tìm kiếm lùi giao nhau tại một nút, thì việc tìm kiếm sẽ dừng lại.

Ưu điểm

- Một trong những lợi thế chính của tìm kiếm hai chiều là tốc độ mà chúng ta nhận được kết quả mong muốn.
- Nó làm giảm đáng kể thời gian thực hiện tìm kiếm bằng cách tìm kiếm đồng thời.
- Nó cũng tiết kiệm tài nguyên cho người dùng vì nó yêu cầu ít dung lượng bộ nhớ hơn để lưu trữ tất cả các tìm kiếm.

Nhược điểm

- Vấn đề cơ bản với tìm kiếm hai chiều là người dùng nên biết trạng thái mục tiêu để sử dụng tìm kiếm hai chiều và do đó giảm đáng kể các trường hợp sử dụng của nó.
- Việc triển khai là một thách thức khác vì cần có mã và hướng dẫn bổ sung để triển khai thuật toán này, đồng thời cũng phải cẩn thận khi từng nút và từng bước để thực hiện các tìm kiếm như vậy.

- Thuật toán phải đủ mạnh để hiểu được điểm giao nhau khi tìm kiếm kết thúc, nếu không sẽ có khả năng xảy ra vòng lặp vô hạn.
- Cũng không thể tìm kiếm ngược qua tất cả các trạng thái.

5. Greedy Search

Greedy Search hay tìm kiếm tham lam là giải thuật tối ưu hóa tổ hợp. Giải thuật tìm kiếm, lựa chọn giải pháp tối ưu địa phương ở mỗi bước với hi vọng tìm được giải pháp tối ưu toàn cục. Thuật toán tham lam là một cách tiếp cận để giải quyết vấn đề bằng cách chọn phương án tốt nhất hiện có mà không cần quan tâm về kết quả trong tương lai mà nó sẽ mang lại.

Thuật toán

Một số quy ước:

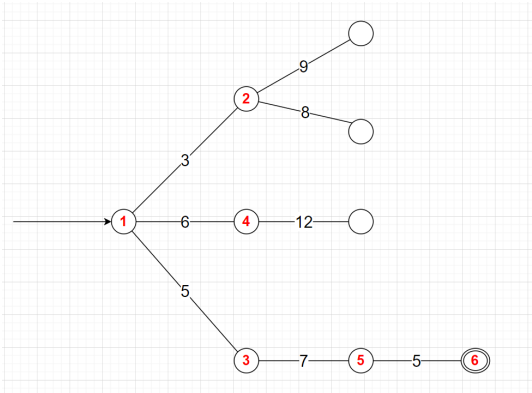
- *Open*: tập các trạng thái đã được sinh ra nhưng chưa được xét đến.
- *Close*: tập các trạng thái đã được xét đến.
- *p*: đỉnh đang xét, đang duyệt.
- *q*: đỉnh đích đến tiếp theo.
- *g(x)*: Khoảng cách đường đi đến điểm tiếp theo.

Trình bày thuật giải:

- Bước 1: Tập *Open* chứa điểm gốc là điểm bắt đầu, lúc này điểm bắt đầu điểm đang xét, đang duyệt.
  + Open:= {p}
  + Close : ={ }
- Bước 2: Kiểm tra *Open* có rỗng không, nếu rỗng tiến đến bước 3.
  + Nếu p là điểm cần đến thì kết thúc trạng thái.
  + Chuyển p qua *Close* và tạo ra các trạng thái kế tiếp q sau p (Xóa p khỏi *Open*).
  + Chọn đỉnh có *g(x)* thấp nhất trong *Open* là p.
  + Lặp lại bước 2.
- Bước 3: Không tìm được điểm đích.

Mô phỏng trên đồ thị

Biểu đồ sau đây cho thấy thứ tự mà các nút được phát hiện trong bằng Greedy Search:



Hình 8: Tìm kiếm bằng giải thuật tham lam

Ưu điểm

– Dễ dàng đưa ra phương án tốt nhất. Thông thường yêu cầu các phương án sắp xếp.

– Thuật toán này có thể hoạt động tốt hơn các thuật toán khác (nhưng không phải trong mọi trường hợp).

Nhược điểm

– Khó thiết kế: Khi chúng ta đã tìm ra cách tiếp cận giải thuật tham lam phù hợp, việc thiết kế các thuật toán tham lam có thể dễ dàng. Tuy nhiên, việc tìm ra cách tiếp cận phù hợp có thể sẽ khó.

– Khó xác định: Việc biểu diễn một thuật toán tham lam phù hợp thường đòi hỏi một lập luận sắc bén.

IV. Thiết kế và hiện thực

1. Thiết kế code

Để có thể thiết kế code một cách hoàn chỉnh, nhóm đã sử dụng **Python 3.7.1** trên hệ điều hành **Window 10** kết hợp với module **pygame** để thiết kế giao diện mê cung và hiện thực giải thuật. Quá trình thực hiện code bao gồm các hàm sau:

1.1 Thiết kế hàm

Thiết kế class Point:

```
class Point:
        def __init__(self,pos,piority=-1,parent=None,step :int = 0) -> None:
            self.pos = pos
            self.piority = piority
            self.parent = parent
            self.step = step
        def __str__(self) -> str:
            return "x: {0}, y: {1}".format(self.pos[0],self.pos[1])

        def __eq__(self, o: object) -> bool:
            return self.pos == o
        def update_piority(self, h):
            self.piority = self.piority + h
```

class Point : gồm vị trí hiên của điểm đó, priority, parent và số bước đã đi

Hàm transfer:

```
def transfer(layout):
    """Transfer the layout of initial puzzle"""
    layout = [x.replace('\n', '') for x in layout]
    layout = [','.join(layout[i]) for i in range(len(layout))]
    layout = [x.split(',') for x in layout]
    maxColsNum = max([len(x) for x in layout])
    for irow in range(len(layout)):
        for icol in range(len(layout[irow])):
            if layout[irow][icol] == ' ':
                layout[irow][icol] = 0   # free space
            elif layout[irow][icol] == '#':
                layout[irow][icol] = -2  # wall
            elif layout[irow][icol] == '&':
                layout[irow][icol] = 1  # player
            elif layout[irow][icol] == '.':
                layout[irow][icol] = -1  # goal
        colsNum = len(layout[irow])
        if colsNum < maxColsNum:
            layout[irow].extend([1 for _ in range(maxColsNum-colsNum)])
    return np.array(layout)
```

transfer() : dùng để chuyển các ký hiệu trong file txt thành các con số cho phù hợp với việc tính toán và so sánh

Hàm readCommand:

```
def readCommand(argv):
    layout = []
    f = open("map/"+ str(argv[0]) , "r")
    for i in f:
        layout.append(i)
    return layout
```

readCommand() : dùng để đọc map từ command line

Hàm getDestination:

```
def getDestination(gameMap):
    return tuple(np.argwhere(gameMap==-1)[0])
```

getDestination() : lấy vị trí của đích đến

Hàm getWall:

```
def getWall(gameMap):
    return tuple(tuple(i) for i in np.argwhere(gameMap == -2))
```

getWall() : lấy danh sách các bị trí của tường

Hàm getStartPoint:

```
def getStartPoint(gameMap):
    return tuple(np.argwhere(gameMap==1)[0])
```

getStartPoint() : lấy vị trí của điểm xuất phát

Hàm check_wall:

```
def check_wall(action, visited):
    if action not in walls + tuple(visited):
        return True
    else:
        return False
```

check_Wall() : kiểm tra vị trí mới có nằm trong visited list hoặc chạm tường hay không

Hàm action:

```
def action(posplayer, visited):
    allAction = [[-1,0],[1,0],[0,-1],[0,1]]
    convert = list(posplayer)
    list_action = []
    for action in allAction:
        xNew, yNew = convert[0] + action[0] , convert[1] + action[1]
        if yNew < 0:
            continue

        if check_wall(tuple((xNew,yNew)), visited):
            list_action.append(tuple((xNew,yNew)))
    return list_action
```

action(): trả về các vị trí có thể di chuyển từ một vị trí hiện tại

Hàm h:

```
def h(point):
    return abs(point[0] - getdespoint[0]) + abs(point[1] - getdespoint[1])
```

h() : tính khoảng cách từ điểm hiện tại đến điểm đích

Hàm h2:

```
def h2(point):
    return abs(point[0] - getstartPoint[0]) + abs(point[1] - getstartPoint[1])
```

h2(): tính khoảng cách từ điểm xuất phát tới điểm hiện tại.

Hàm checkEndState:

```
def checkEndState(point1, visit1, point2, visit2):
    if point1 in visit2:
        return True
    if point2 in visit1:
        return True

    return False
```

checkEndState(): được sử dụng trong giải thuật bidireactional, dùng để kiểm tra một điểm có đi vào vùng visited của điểm kia hay chưa.

Hàm Step:

```
def step(point : Point):
    return point.step + 1
```

Step(): Tăng thêm 1 bước đã đi.

Hàm getpath_nonRecur:

```
def getpath_nonRecur(finalnode:Point,screen,clock):
    temp = finalnode
    length = 0
    while temp != None:
        length +=1
        draw_color(screen,temp.pos,BLUE,32)
        temp = temp.parent
        clock.tick(60)
        pygame.display.update()
    return length
```

getPath_nonRecur() : tô màu cho đường đi đến đích và trả về số bước sẽ phải đi quap

Hàm getpath2:

```
def getpath2(start:Point,des:Point,screen,clock):
    length = 0
    finalnode1 = start
    finalnode2 = des
    while finalnode1 != None or finalnode2 != None:
        if finalnode1 != None:
            draw_color(screen,finalnode1.pos,BLUE,32)
        if finalnode2 != None:
            draw_color(screen,finalnode2.pos,BLUE,32)
        clock.tick(60)
        pygame.display.update()
        if finalnode1 != None and finalnode2 != None:
            length +=2
            finalnode1 = finalnode1.parent
            finalnode2 = finalnode2.parent
        elif finalnode1 != None and finalnode2 == None:
            length += 1
            finalnode1 = finalnode1.parent
        elif finalnode2 != None and finalnode1 == None:
            length += 1
            finalnode2 = finalnode2.parent
    return length
```

getPath2() : tô màu cho đường đi đến đích và trả về số bước sẽ phải đi qua

Hàm draw_color:

```
def draw_color(screen,point,color,size):
    pygame.draw.rect(screen,color,(point[1] * size,point[0] * size, size, size))
```

draw_color() : tô màu cho ô vuông được chọn

Hàm draw_button:

```
def draw_button(screen, point, color):
    pygame.draw.rect(screen,color, (point[0], point[1],80,20))
```

draw_button() : vẽ button

Hàm draw_text:

```
def draw_text(screen,font,point,color,data):
    text = font.render(data, True, color)
    screen.blit(text,(point))
```

draw_text() : viết chữ

Hàm pause:

```
def pause():
    paused = True
    while paused:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1:
                    paused = False
```

pause() : tạm dừng chương trình.

1.2 Thiết kế giải thuật tìm đường mê cung

Breadth-First Search:

```
def bfs(gameMap,screen,clock):
    """Thực thi BFS"""
    startPoint = Point(getstartPoint)
    desPoint = Point(getdespoint)

    visited = []
    opended = []

    opended.append(startPoint)

    while len(opended) > 0:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
        visit_point = opended.pop(0)
        # kiểm tra vị trí hiện tại có phải là vị trí đích hay chưa
        if visit_point == desPoint:
            return visit_point
        #kiểm tra vị trí hiện tại đã đi qua hay chưa
        if visit_point not in visited:
            visited.append(visit_point)

            # Lấy danh sách các bước đi hợp lệ
            actions = action(visit_point.pos,visited)

            for neighbor in actions:
                newPos = Point (neighbor,-1,visit_point)
                opended.append(newPos)
                # Tô màu cho những node con
                draw_color(screen,neighbor,PINK,32)

            draw_color(screen,visit_point.pos,YELLOW,32)
            clock.tick(60)
            pygame.display.update()
```

- Đầu tiên ta push điểm xuất phát vào opened queue.
- Kiểm tra opened queue có rỗng hay không.
- Nếu rỗng:
  + Kết thúc chương trình
- Nếu không rỗng:
  + visit_point = opened.pop
  + Kiểm tra visit_point đã đến đích hay chưa. Nếu đã đến đích -> return visit_point
  + Kiểm tra visit_point có nằm trong visited hay không. Nếu có thì thực hiện lại vòng lặp, kiểm tra opened queue có rỗng hay không.
  + Bỏ visit_point vào visited list

+ Lấy các vị trí có thể đi của visit_point và push vào opened queue. Thực hiện lại vòng lặp.

A* Search:

```
def astar(gameMap,screen,clock):
    # count dùng để phân biệt 2 node con có piority bằng nhau
    # node nào được tìm thấy trước sẽ có giá trị nhỏ hơn
    count = 0

    startPoint = Point(getstartPoint)
    desPoint = Point(getdespoint)

    visited = []
    pioQueue = PriorityQueue()
    # bỏ điểm xuất phát vào piority queue
    pioQueue.put((startPoint.piority,count,startPoint))

    while not pioQueue.empty():
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()

        # lấy node đầu của pioQueue (node có piority thấp nhất)
        # nếu piority bằng nhau -> node có count thấp nhất
        visit_point = pioQueue.get()[2]

        #kiểm tra điểm hiện tại có phải là điểm đích
        if visit_point == desPoint:
            return visit_point

        # Kiểm tra trong visited
        if visit_point not in visited:
            visited.append(visit_point)

            # cập nhật số bước đã đi
            newStep = step(visit_point)

            actions = action(visit_point.pos, visited)
            for neighbor in actions:
                count += 1

                # node con có piority = số step + khoảng cách đến đích
                newPos = Point(neighbor,h(neighbor) + newStep,visit_point,newStep)

                # đặt node con vào trong piority queue
                pioQueue.put((newPos.piority,count,newPos))
                # tô màu cho node con
                draw_color(screen,neighbor,PINK,32)

            # tô màu cho node đã đi qua
            draw_color(screen,visit_point.pos,YELLOW,32)
            clock.tick(60)
            pygame.display.update()
```

– Đầu tiên ta push điểm xuất phát vào priority queue (priority được thiết kế giống min Heap)

– Kiểm tra priority queue có rỗng hay không

– Nếu rỗng

+ Kết thúc chương trình

– Nếu không rỗng

+ visit_point = priority.pop

+ Kiểm tra visit_point đã đến đích hay chưa. Nếu đã đến đích -> return visit_point
+ kiểm tra visit_point có nằm trong visited hay không. Nếu có thì thực hiện lại vòng lặp, kiểm tra priority có rỗng hay không.
+ Bỏ visit_point vào visited list
+ Tính giá trị của step
+ Lấy các vị trí có thể đi của visit_point
+ Tính các giá trị priority cho mỗi vị trí con và push vào priority queue
+ Quay lại thực hiện vòng lặp.

Bidirectional A* Search:

```
def bidirectional(gameMap,screen,clock):
    count = 0
    startPoint = Point(getstartPoint)
    desPoint = Point(getdespoint)
    visited = []
    visited2 = []
    pioQueue = PriorityQueue()
    pioQueue2 = PriorityQueue()
    pioQueue.put((startPoint.piority,count,startPoint))
    pioQueue2.put((desPoint.piority,count,desPoint))
    while not pioQueue.empty() and not pioQueue2.empty():
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
        visit_point = pioQueue.get()[2]
        visit_point2 = pioQueue2.get()[2]
        if visit_point == visit_point2 or checkEndState(visit_point,visited,visit_point2,visited2):
            if visit_point == visit_point2:
                return visit_point,visit_point2
            elif visit_point in visited2:
                return visit_point, visited2[visited2.index(visit_point)]
            elif visit_point2 in visited:
                return visited[visited.index(visit_point2)], visit_point2
        if visit_point not in visited:
            visited.append(visit_point)
            newStep = step(visit_point)
            actions = action(visit_point.pos, visited)
            for neighbor in actions:
                count += 1
                newPos = Point(neighbor,h(neighbor) + newStep,visit_point,newStep)
                pioQueue.put((newPos.piority,count,newPos))
                draw_color(screen,neighbor,PINK,32)
        if visit_point2 not in visited2:
            visited2.append(visit_point2)
            newStep = step(visit_point2)
            actions = action(visit_point2.pos, visited2)
            for neighbor in actions:
                count += 1
                newPos = Point(neighbor,h2(neighbor) + newStep,visit_point2,newStep)
                pioQueue2.put((newPos.piority,count,newPos))
                draw_color(screen,neighbor,PINK,32)
            draw_color(screen,visit_point.pos,YELLOW,32)
            draw_color(screen,visit_point2.pos,YELLOW,32)
            clock.tick(60)
            pygame.display.update()
```

- Đầu tiên ta push điểm xuất và điểm kết thúc phát vào hai priority queue
- Kiểm tra cả 2 priority queue có đều rỗng hay không
- Nếu cả hai đều rỗng
  + Kết thúc chương trình
- Nếu không rỗng
  + visit_point1 = priority1.pop
  + visit_point2 = priority2.pop
  + Kiểm tra visit_point1 và visit_point2 đã gặp nhau hay chưa hoặc điểm này đi vào vùng đã đi qua của điểm kia. Nếu có thì trả về vị trí tương ứng đối với 2 cây
  + Nếu visit_point1 không nằm trong visited
    * Push visit_point1 vào visited
    * Push các vị trí con vào priority queue
  + Nếu visit_point2 không nằm trong visited2
    * Push visit_point2 vào visited2
    * Push các vị trí con vào priority queue
  + Quay lại thực hiện vòng lặp, kiểm tra điều kiện.

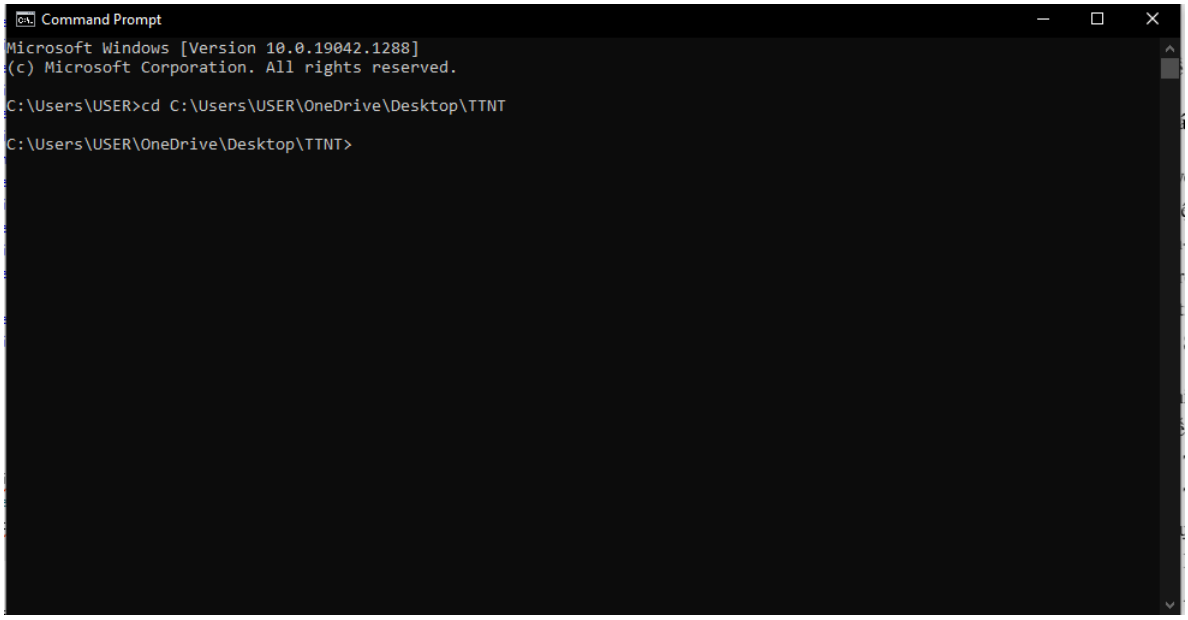
Greedy Search:

```
def greedy(gameMap,screen,clock):
    count = 0
    startPoint = Point(getstartPoint)
    desPoint = Point(getdespoint)

    visited = []
    pioQueue = PriorityQueue()
    pioQueue.put((startPoint.piority,count,startPoint))
    while not pioQueue.empty():
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
        visit_point = pioQueue.get()[2]
        # print(type(visit_point))
        if visit_point == desPoint:
            return visit_point
        if visit_point not in visited:
            visited.append(visit_point)
            newStep = 0
            actions = action(visit_point.pos, visited)
            # print(visit_point.pos)
            for neighbor in actions:
                count += 1
                newPos = Point(neighbor,h(neighbor) + newStep,visit_point,newStep)
                pioQueue.put((newPos.piority,count,newPos))
                draw_color(screen,neighbor,PINK,32)

            draw_color(screen,visit_point.pos,YELLOW,32)
            clock.tick(60)
            pygame.display.update()
```

- Đầu tiên ta push điểm xuất phát vào priority queue (priority được thiết kế giống min Heap).
- Kiểm tra priority queue có rỗng hay không.
- Nếu rỗng:
  + Kết thúc chương trình
- Nếu không rỗng:
  + visit_point = priority.pop
  + Kiểm tra visit_point đã đến đích hay chưa. Nếu đã đến đích -> return visit_point
  + Kiểm tra visit_point có nằm trong visited hay không. Nếu có thì quay lại thực hiện vòng lặp.
  + Bỏ visit_point vào visited list
  + Lấy các vị trí có thể đi của visit_point
  + Tính các giá trị priority cho mỗi vị trí con và push vào priority queue
  + Thực hiện lại vòng lặp.

2. Hiện thực chương trình

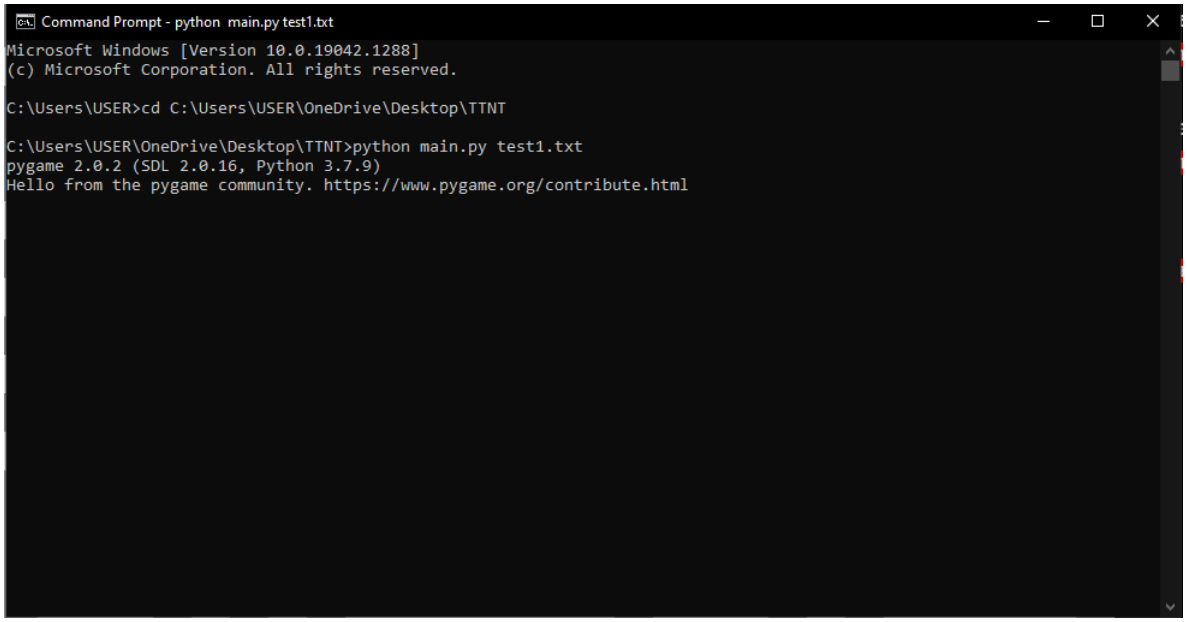
– Bước 1: Ta mở terminal lên và di chuyển tới thực mục lưu trữ source code.



Hình 9: Terminal thực hiện

Ở đây source code được lưu trữ tại thư mục TTNT.

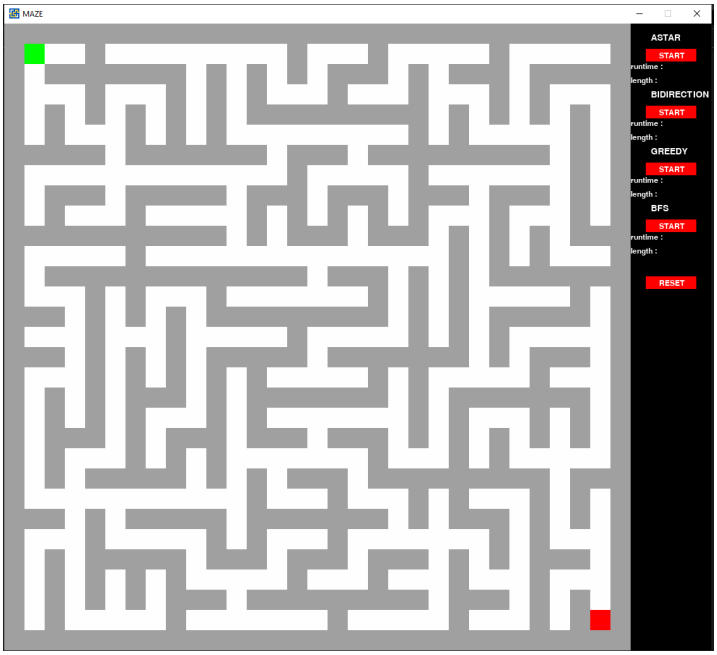
– Bước 2: Hiện thực chương trình bằng câu lệnh "python main.py test1.txt" , terminal sẽ chạy giống như hình:



Hình 10: Hiện thực chương trình

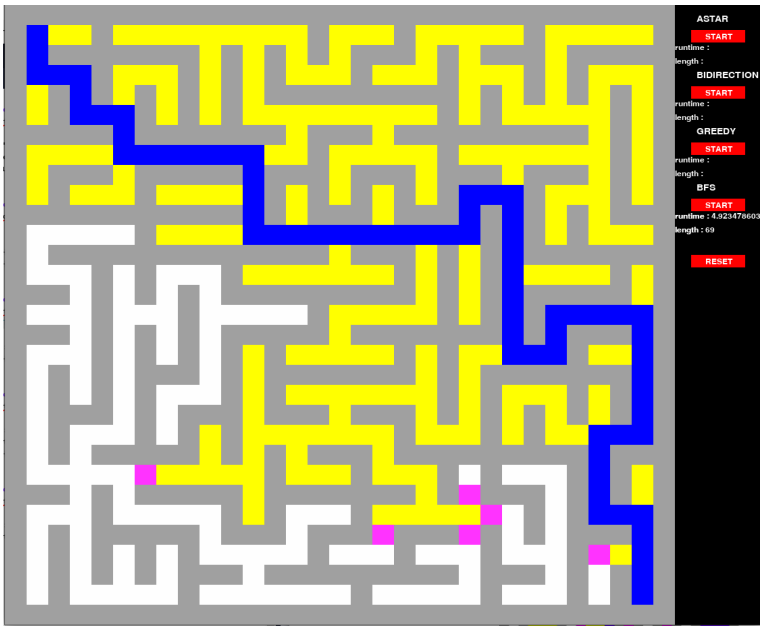
Trong đó: "main.py" là source code python của chương trình và "test1.txt" là file

lưu trữ map mê cung. Khi chạy chương trình, giao diện của mê cung được hiện lên:



Hình 11: Giao diện thực thi

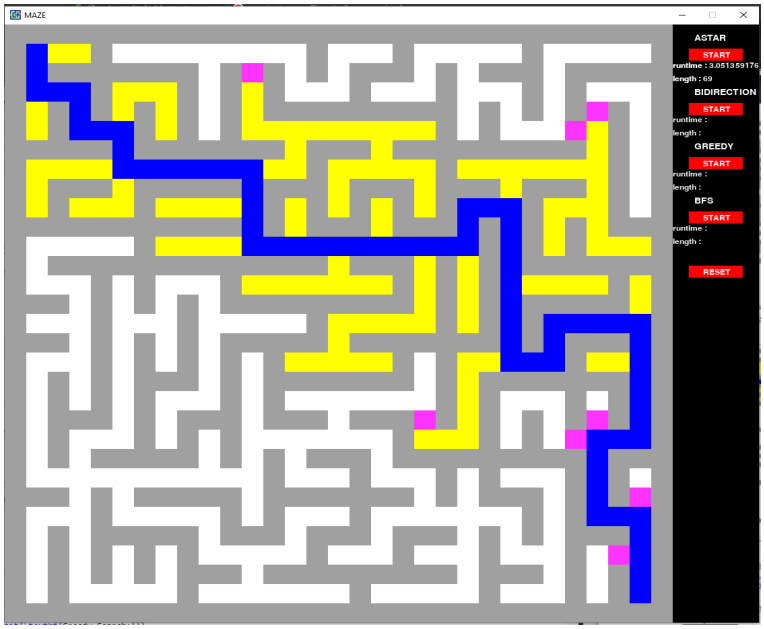
– Bước 3: Chọn giải thuật mà mình muốn áp dụng cho mê cung này:
Breadth-First Search:



Hình 12: Giải thuật BFS

Thời gian thực thi: 4.923s và mất 69 ô để đến đích.

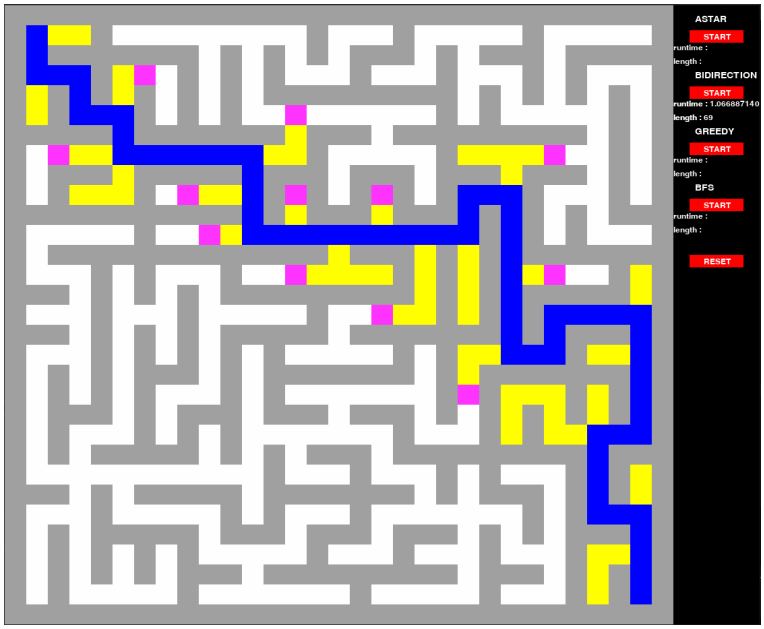
A* Search:



Hình 13: Giải thuật A* Search

Thời gian thực thi: 3.051s và mất 69 ô để đến đích.

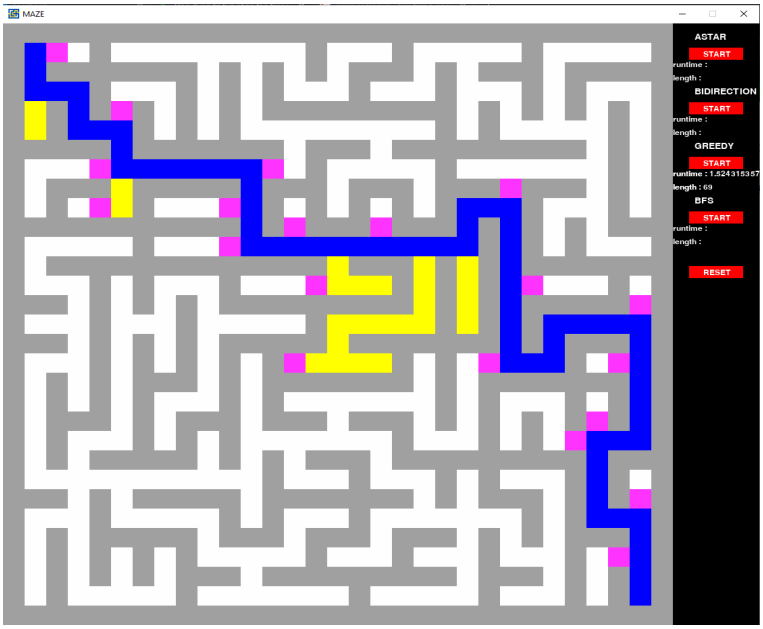
Bidirectional A* Search:



Hình 14: Giải thuật Bidirectional

Thời gian thực thi: 1.067s và mất 69 ô để đến đích.

Greedy Search:



Hình 15: Giải thuật Greedy Search

Thời gian thực thi: 1.524s và mất 69 ô để đến đích.

– Bước 4: Ấn RESET sau khi giải thuật đã chạy xong để có thể chọn lại giải thuật khác hoặc thoát khỏi chương trình.

3. Đánh giá

Trong cùng một mê cung, khi ta sử dụng các giải thuật khác nhau sẽ cho ra thời gian xử lý khác nhau:

– BFS: 4.923s

– A* Search: 3.051s

– Bidirectional A* Search: 1.067s

– Greedy Search: 1.524s

Thời gian xử lý của giải thuật Birectional A* Search là ngắn nhất, phù hợp cho bài toán tìm đường ngắn nhất trong mê cung nhưng độ phức tạp của giải thuật là lớn nhất vì phải thực hiện giải thuật tìm kiếm A* đồng thời cho điểm bắt đầu và điểm kết thúc.

Cả 3 giải thuật A* Search, Bidirectional A* Search và Greedy Search đều nhanh hơn so với giải thuật tìm kiếm logic thông thường BFS.

TÀI LIỆU THAM KHẢO

[1] Duy Đặng.(2018). Giới thiệu giải thuật tìm đường đi ngắn nhất trong mê cung. http://arduino.vn/bai-viet/5553-gioi-thieu-thuat-toan-tim-duong-di-ngan-nhat-trong-me-cung.

[2] Shichiki Lê.(2020). Thuật giải A*. https://www.stdio.vn/giai-thuat-lap-trinh/thuat-giai-a-DVnHj.

[3] Karleigh Moore, Jimin Khim, and Eli Ross.(2021). Greedy Algorithms. https://brilliant.org/wiki/greedy-algorithm/.

[4] Trần Minh Thắng. (2020). Thuật toán Breadth First Search. https://www.stdio.vn/giai-thuat-lap-trinh/thuat-toan-breadth-first-search-sBPnH?fbclid=IwAR29F0IQyQ9DG6pWOD5PcHXpgvAnjIDdW_cBcDbqLPoXG63k5aQprTUgyk0.

[5] Techie Delight.(2021).Breadth-First Search (BFS) – Iterative and Recursive Implementation. https://www.techiedelight.com/breadth-first-search/?fbclid=IwAR1s6GL5JPqX7ZErz7WbmLCTt2yQ.

[6] Priya Pedamkar.(2021).Bidirectional Search. https://www.educba.com/bidirectional-search/?fbclid=IwAR3_Xnswd82GaOZEs3vJkBP6Adb5XndBXTonFRlPO1cnRpsqx6rB1t8ID5M