Báo cáo đồ án môn học

Cơ sở trí tuệ nhân tạo

Thuật toán tìm kiếm

Thành viên:

19120387 – Lê Sỹ Thuần

19120328- Võ Trọng Phú

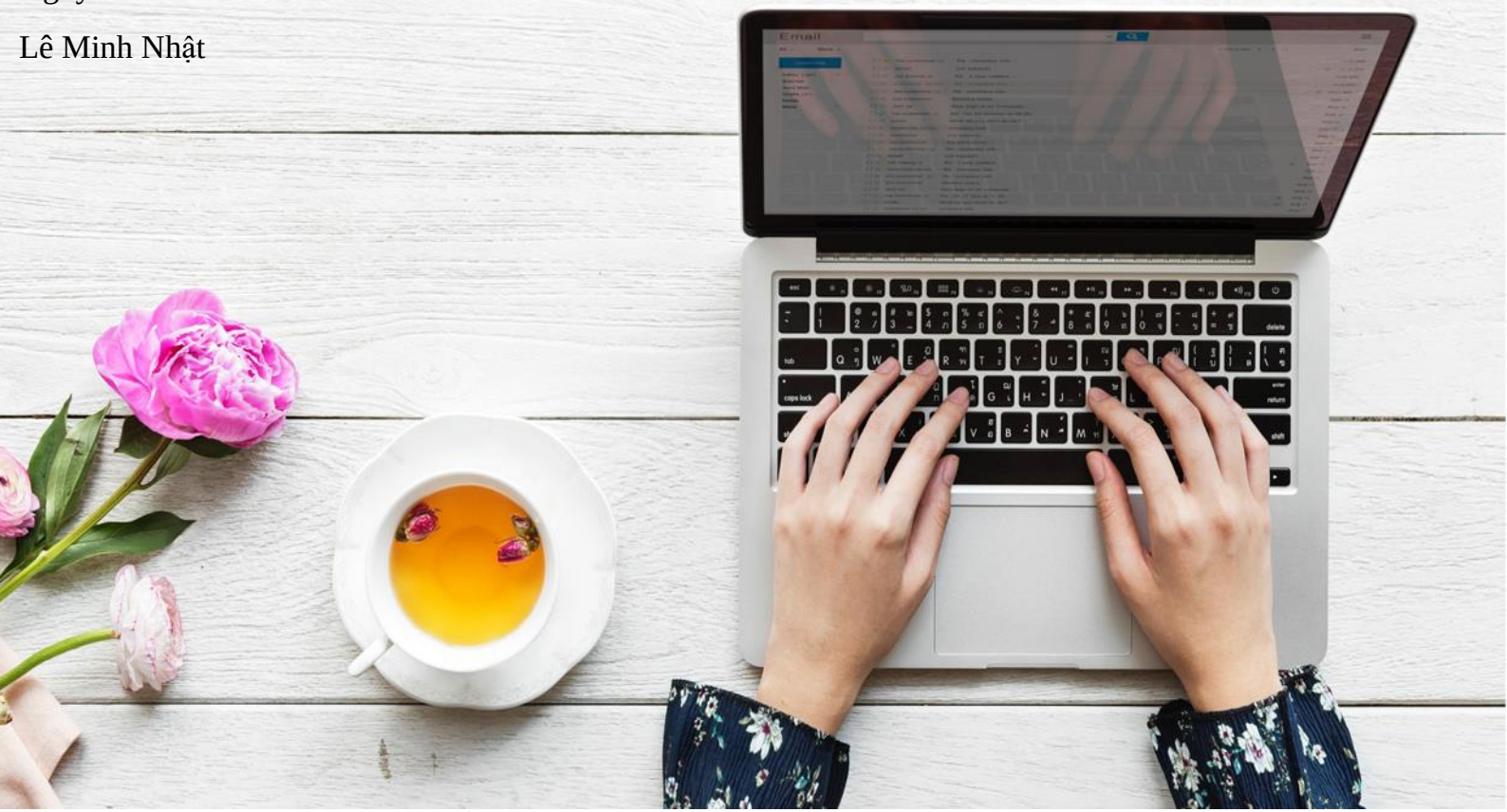
Bảng phân công công việc:

Lê Sỹ Thuần	Thiết kế, cài đặt DFS, A*, thuật toán cho bản đồ có điểm thưởng, thiết kế bản đồ, tổng hợp code, viết báo cáo
Võ Trọng Phú	Thiết kế, cài đặt BFS, Greedy BFS, thiết kế bản đồ, viết báo cáo

Giáo viên hướng dẫn:

Nguyễn Khánh Toàn

Lê Minh Nhật



MỤC LỤC

GIỚI THIỆU

Nghiên cứu, cài đặt và trình bày các thuật toán tìm kiếm: Depth First Search (DFS) , Breath First Search(BFS), A* Search, Greedy Best First Search ứng với 5 bản đồ không có điểm thưởng, 3 bản đồ có điểm thưởng

I. TỔNG QUAN VỀ CÁC THUẬT TOÁN:

1. THUẬT TOÁN TÌM KIẾM THEO CHIỀU SÂU:

Tìm kiếm ưu tiên chiều sâu hay tìm kiếm theo chiều sâu (DFS) là một thuật toán duyệt hoặc tìm kiếm trên một cây hoặc một đồ thị. Thuật toán khởi đầu tại gốc (hoặc chọn một đỉnh nào đó coi như gốc) và phát triển xa nhất có thể theo mỗi nhánh.

DFS là một dạng tìm kiếm thông tin không đầy đủ mà quá trình tìm kiếm được phái triển tới đỉnh con đầu tiên của nút đang tìm kiếm cho tới khi gặp được đỉnh cần tìm hoặc tới một nút không có con. Khi đó, giải thuật quay lui về đỉnh vừa mới tìm kiếm ở bước trước.

2. THUẬT TOÁN TÌM KIẾM THEO CHIỀU RỘNG:

Thuật toán tìm kiếm theo chiều rộng (BFS) là một thuật toán tìm kiếm trong đồ thị, trong đó việc tìm kiếm bao gồm các thao tác: cho trước một đỉnh của đồ thị, sau đó them các đỉnh kề với đỉnh vừa cho vào danh sách có thể hướng tới tiếp theo.

Thuật toán BFS bắt đầu từ đỉnh gốc và lần lượt nhìn các đỉnh kề với đỉnh gốc. Sau đó với mỗi đỉnh trong số đó, thuật toán lại lần lượt nhìn trước các đỉnh kề với nó mà chưa được quan sát trước đó và lặp lại. Thuật toán tìm kiếm theo chiều rộng luôn tìm ra đường đi ngắn nhất có thể.

3. THUẬT TOÁN TÌM KIẾM THAM LAM:

Giải thuật tham lam (Greedy algorithm) là một thuật toán giải quyết một bài toán theo kiểu metaheuristic để tìm kiếm lựa chọn tối ưu ở mỗi bước đi với hy vọng tìm được tối ưu toàn cục.

Chúng ta có thể lựa chọn giải pháp nào được cho là tốt nhất ở thời điểm hiện tại và sau đó giải bài toán con nảy sinh từ việc thực hiện lựa chọn vừa rồi. Lựa chọn của thuật toán tham lam có thể phụ thuộc vào các lựa chọn trước đó.
Đối với nhiều bài toán, giải thuật tham lam hầu như không cho ra lời giải tối ưu toàn cục vì chúng thương không chạy trên tất cả các trường hợp.

4. THUẬT TOÁN TÌM KIẾM A*:

Giải thuật tìm kiếm A* (A-star search) là giải thuật tìm một đường đi từ một nút khởi đầu tới một nút đích cho trước. Thuật toán này sử dụng một "đánh giá heuristic" để xếp loại từng nút theo ước lượng về tuyến đường tốt nhất đi qua nút đó và duyệt các nút theo thứ tự của đánh giá heuristic này.

Cũng như BFS, A* là thuật toán đầy đủ theo nghĩa rằng nó sẽ luôn luôn tìm thấy lời giải nếu bài toán có lời giải.

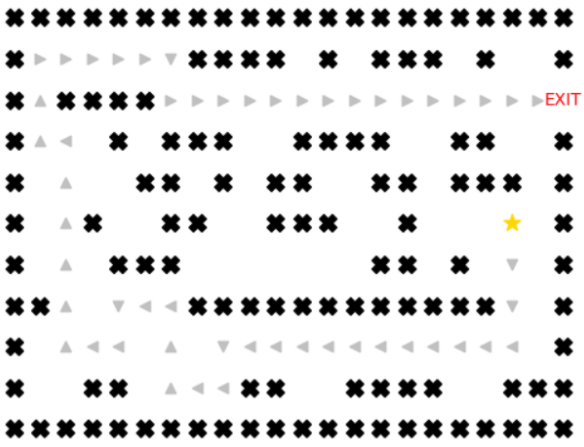
A* còn có tính chất hiệu quả một cách tối ưu với mọi hàm heuristic h, nghĩa là không có thuật toán nào cũng sử dụng hàm heuristic đó mà chỉ phải mở rộng ít nút hơn A*, trừ khi có một số lời giải chưa đầy đủ mà tại đó h dự đoán chính xác chi phí của đường đi tối ưu.

II. BÁO CÁO TESTCASE CỦA CÁC THUẬT TOÁN:

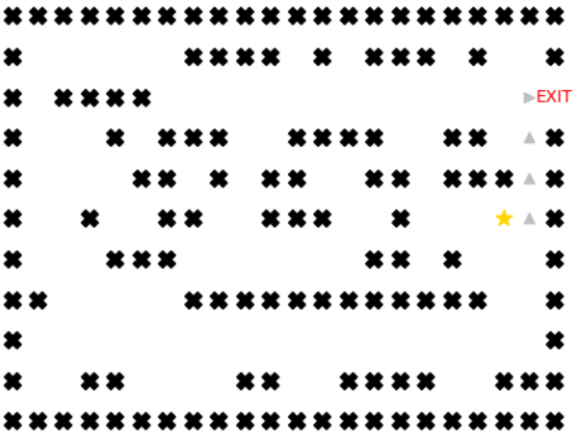
1. BẢN ĐỒ KHÔNG CÓ ĐIỂM THƯỞNG:

Bản đồ 1: maze_mapBFS.txt

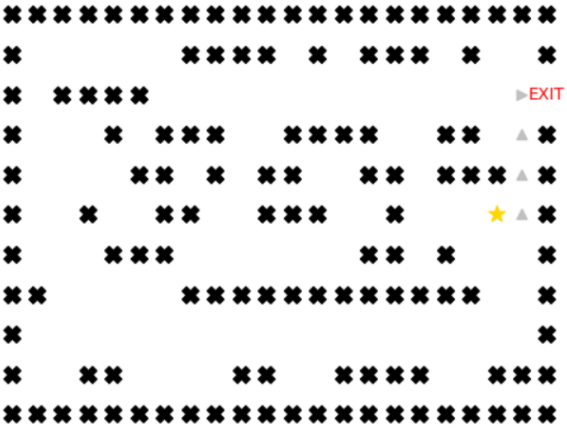
Depth first search / maze_mapBFS.txt



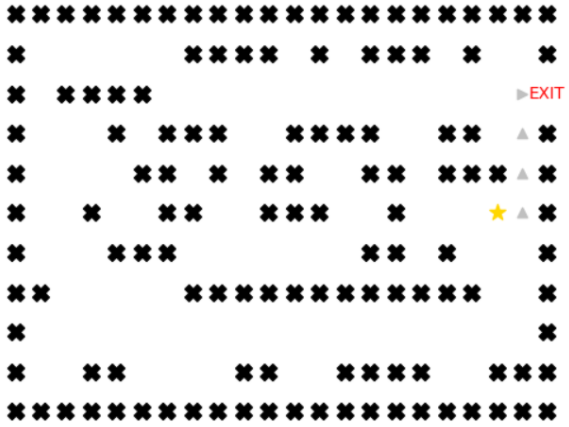
Breadth first search / maze_mapBFS.txt



Greedy search / maze_mapBFS.txt



A star search / maze_mapBFS.txt




```
DEPTH FIRST SEARCH
Cost from start to end: 54
------------------------------
BREADTH FIRST SEARCH
Cost from start to end: 6
------------------------------
GREEDY SEARCH
Cost from start to end: 6
------------------------------
A STAR SEARCH
Cost from start to end: 6
```

➢ *Nhận xét về các thuật toán*

- Thuật toán tìm kiếm mù DFS và BFS:

- DFS sẽ tìm kiếm theo chiều sâu của ma trận nên sẽ cho ra đường đi dài hơn BFS.

- BFS duyệt tất cả các nút con đến khi tìm được đích thì dừng nên đường đi luôn là tối ưu nếu chi phí di chuyển giữa các nút bằng nhau và không đổi.

- Nhưng DFS sẽ cần không gian ít hơn, chỉ cần lưu các nút từ điểm bắt đầu tới đích, còn BFS sẽ phải lưu tất cả các nút cho đến khi gặp đích. Như vậy trong thực tế cần phải cân nhắc việc chọn lựa BFS hay DFS mặc dù BFS cho đường đi ngắn nhất nhưng tốn nhiều không gian

- Trong ma trận đầu tiên ta thấy điểm đích ở gần với điểm bắt đầu nên BFS sẽ cho lời giải tốt nhất trong ma trận này.

- Thuật toán tìm kiếm có thông tin Greedy Best First Search và A* search:

- Cả hai đều tìm được đường đi ngắn nhất với heuristic là khoảng cách euclid giữa điểm và điểm đích

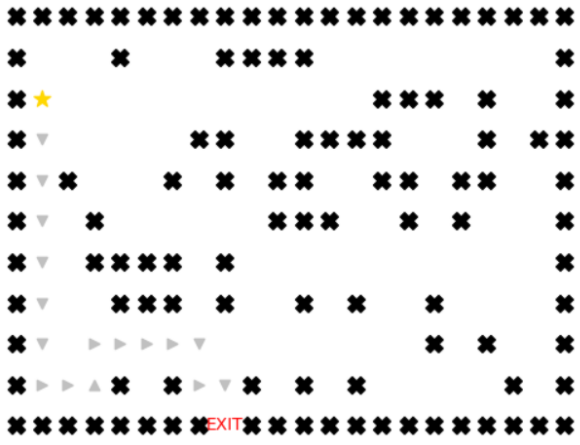
- Vì chi phí di chuyển đến các điểm trong ma trận là 1 nên BFS sẽ cho ra được đường đi ngắn nhất tới điểm đích.

→ Các thuật toán đầy đủ khi đường đi là hữu hạn và có đường đi từ điểm bắt đầu đến điểm kết thúc

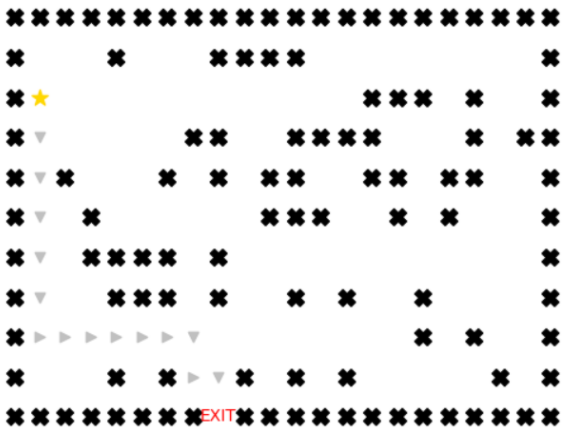
→ Hàm heuristc với khoảng cách euclid giữa điểm hiện tại và điểm đích là nhất quán trong tìm đường đi các ma trận vì khoảng cách giữa điểm hiện tại tới điểm kết thúc <= khoảng cách giữa điểm hiện tại tới các nút con + khoảng cách từ các nút con đến điểm kết thúc

Bản đồ 2: maze_mapDFS.txt

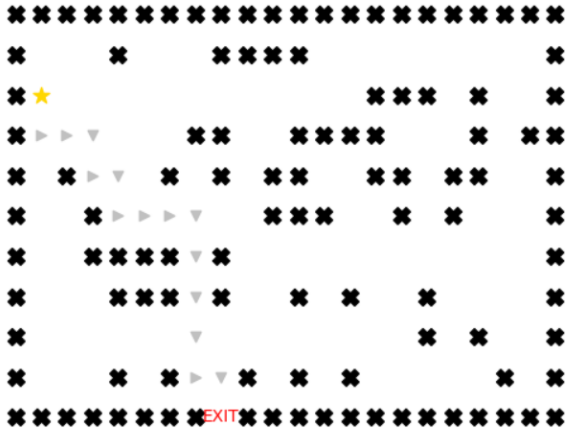
Depth first search / maze_mapDFS.txt



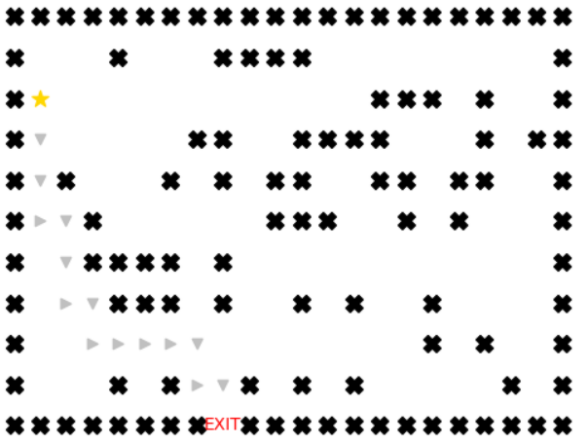
Breadth first search / maze_mapDFS.txt



Greedy search / maze_mapDFS.txt



A star search / maze_mapDFS.txt




```
DEPTH FIRST SEARCH
Cost from start to end: 18
------------------------------
BREADTH FIRST SEARCH
Cost from start to end: 16
------------------------------
GREEDY SEARCH
Cost from start to end: 16
------------------------------
A STAR SEARCH
Cost from start to end: 16
------------------------------
```

- Nhận xét về các thuật toán:

- Thuật toán DFS sẽ duyệt từ điểm bắt đầu tới các nút có chiều sâu từ trên xuống dưới trước cho đến khi gặp được đích

- Thuật toán BFS duyệt tất cả các điểm là nút con của nút cha cho đến khi gặp điểm kết thúc thì dừng.

- Trong ma trận 2, điểm kết thúc có chiều sâu so với điểm bắt đầu, và chi phí đến đích không chênh lệch nhau quá nhiều nên để tối ưu về mặt không gian thì DFS sẽ tốt hơn so với BFS

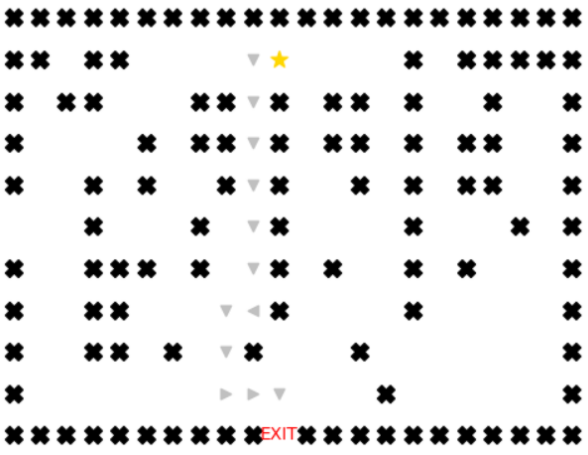
- Greedy Best First Search và A Star Search đều cho kết quả giống nhau nhưng đường đi lại khác nhau vì:

- Greedy Best First Search quyết định đường đi dựa trên h(n): hàm heuristic. Nhóm em sử dụng heuristic là khoảng cách euclid giữa điểm đến điểm kết thúc. Nên tại một thời điểm sẽ có thể có 2 điểm bằng heuristic nhau nên phải lựa chọn ngẫu nhiên, đi theo 1 hướng dài hơn sẽ cho ra đường đi dài hơn

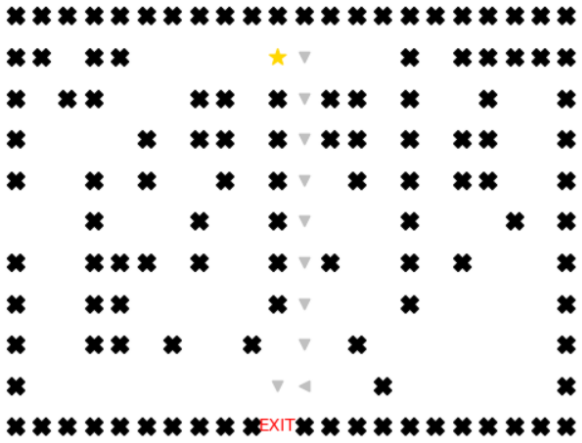
- A* Search quyết định đường đi dựa trên g(n)+h(n) trong đó g(n) là chi phí đường đi từ điểm bắt đầu tới điểm hiện tại. Vì lúc duyệt các nút có heuristic bằng nhau sẽ luôn so sánh g(n)+h(n) giữa các hướng đi nên sẽ tìm được đi tối ưu nhất nhưng tốn nhiều không gian hơn

Bản đồ 3: maze_mapAStar.txt

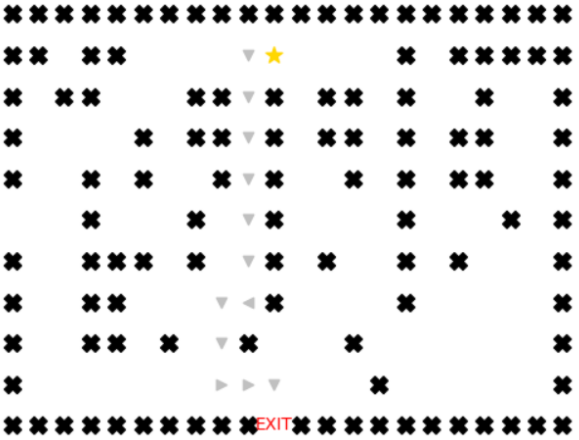
Depth first search / maze_mapAStar.txt



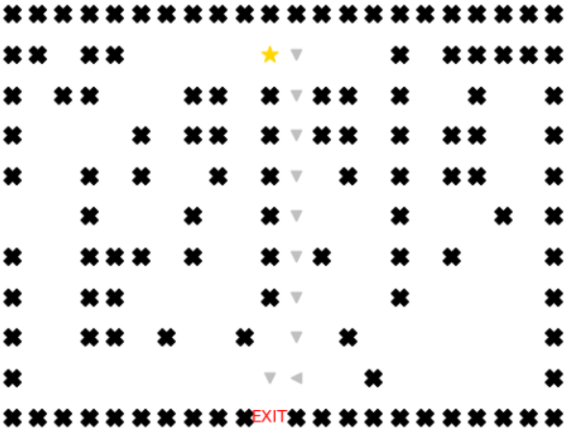
Breadth first search / maze_mapAStar.txt



Greedy search / maze_mapAStar.txt



A star search / maze_mapAStar.txt



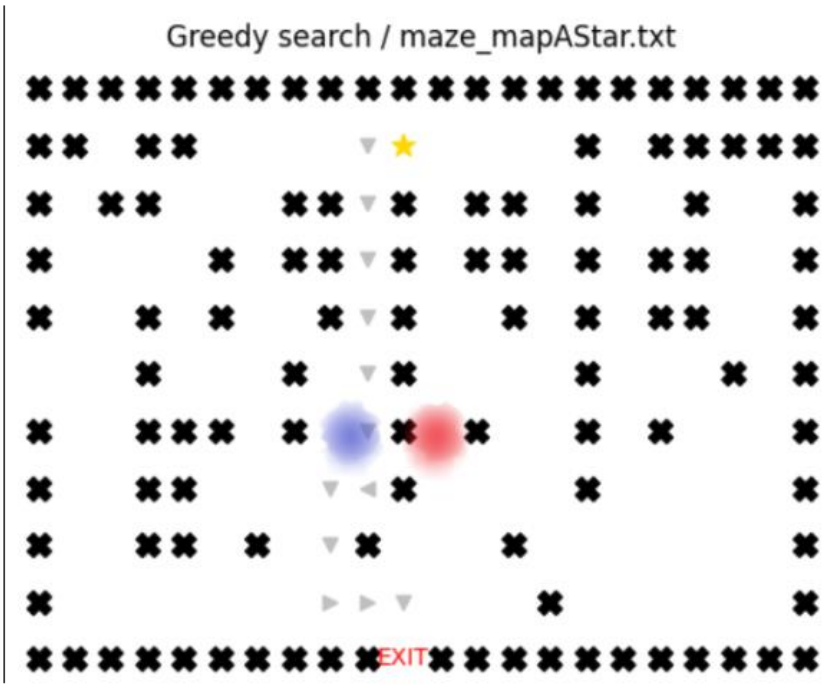

```
------------------------------
DEPTH FIRST SEARCH
Cost from start to end: 14
------------------------------
BREADTH FIRST SEARCH
Cost from start to end: 12
------------------------------
GREEDY SEARCH
Cost from start to end: 14
------------------------------
A STAR SEARCH
Cost from start to end: 12
------------------------------
```

➢ *Nhận xét về các thuật toán:*

- Trong ma trận 3, điểm kết thúc có chiều sâu so với điểm bắt đầu, và chi phí đến đích không chênh lệch nhau quá nhiều nên để tối ưu về mặt không gian thì DFS sẽ tốt hơn so với BFS

- Greedy Best First Search không tối ưu bằng A * khi chi phí là 14 so với chỉ 12 là chi phí tối ưu

- Cả hai đều dùng chung một hàm heuristic và Greedy Best First Search sẽ duyệt theo nhánh bên trái(vì trong code sẽ duyệt bên trái trước)



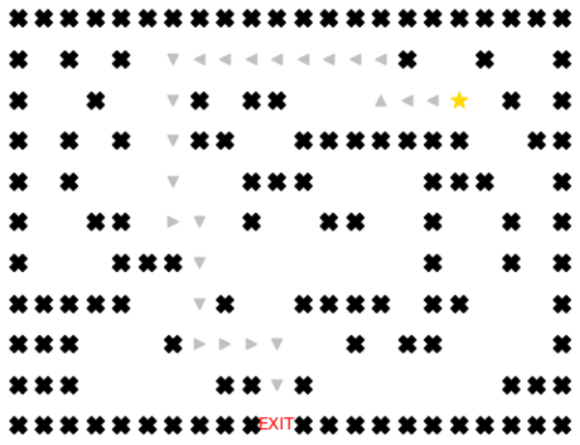
- A* search sẽ duyệt các nút con từ cả 2 hướng và tìm được đường ngắn nhất dựa trên chi phí từ điểm bắt đầu tới hiện tại cộng khoảng cách từ điểm hiện tại tới điểm kết thúc

- BFS và A* Search trong ma trận này cho đường đi ngắn nhất

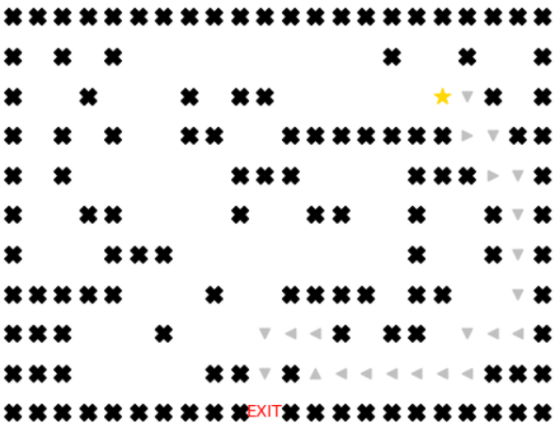
- Nhưng nếu thuật toán Greedy Best First Search duyệt con bên phải trước thì trong ma trận này sẽ cho kết quả tối ưu.

Bản đồ 4: maze_mapRandom1.txt

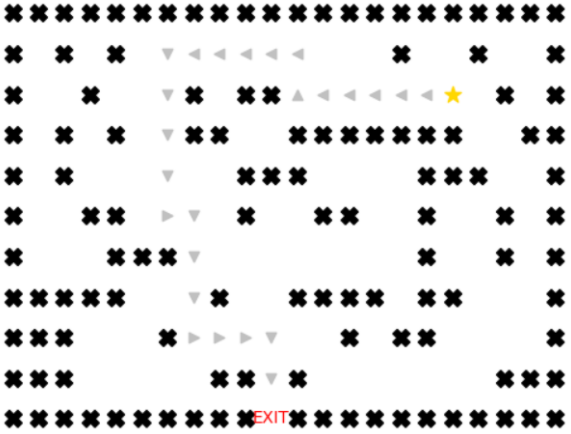
Depth first search / maze_mapRandom1.txt



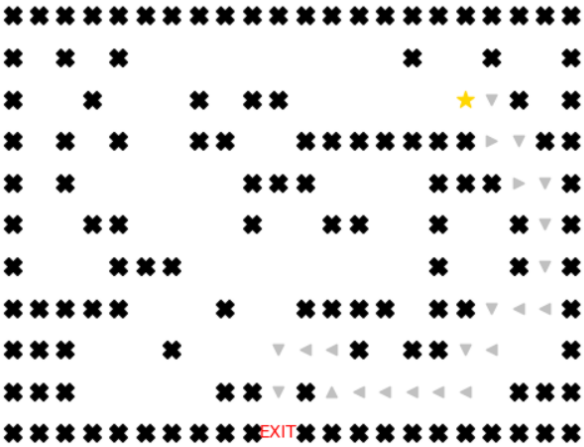
Breadth first search / maze_mapRandom1.txt



Greedy search / maze_mapRandom1.txt



A star search / maze_mapRandom1.txt




```
DEPTH FIRST SEARCH
Cost from start to end: 26
------------------------------
BREADTH FIRST SEARCH
Cost from start to end: 24
------------------------------
GREEDY SEARCH
Cost from start to end: 26
------------------------------
A STAR SEARCH
Cost from start to end: 24
------------------------------
```

➢ *Nhận xét về các thuật toán:*

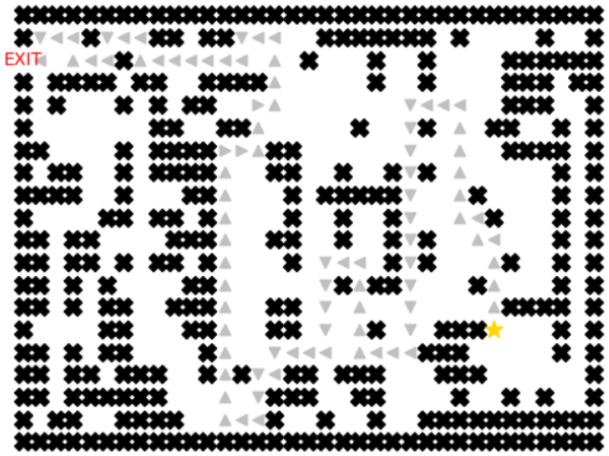
- Trong ma trận 4, điểm kết thúc có chiều sâu so với điểm bắt đầu, và chi phí đến đích không chênh lệch nhau quá nhiều nên để tối ưu về mặt không gian thì DFS sẽ tốt hơn so với BFS

- BFS và A* search cho ra đường đi ngắn nhất

- Hàm heuristic được sử dụng không cho kết quả tối ưu cho Greedy Best First Search nhưng tối ưu cho A* search

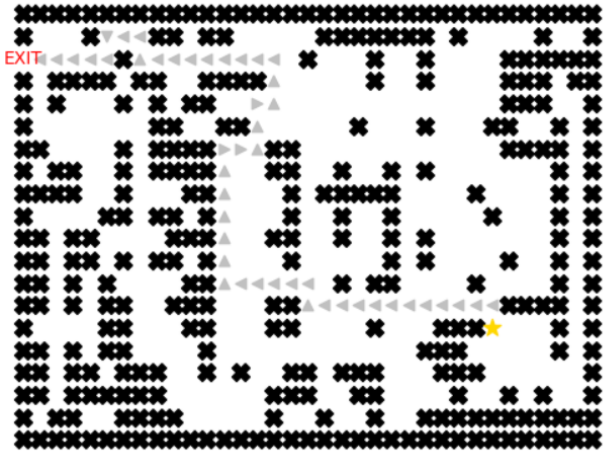
Bản đồ 5: bản đồ lớn maze_mapRandom2.txt

Depth first search / maze_mapRandom2.txt

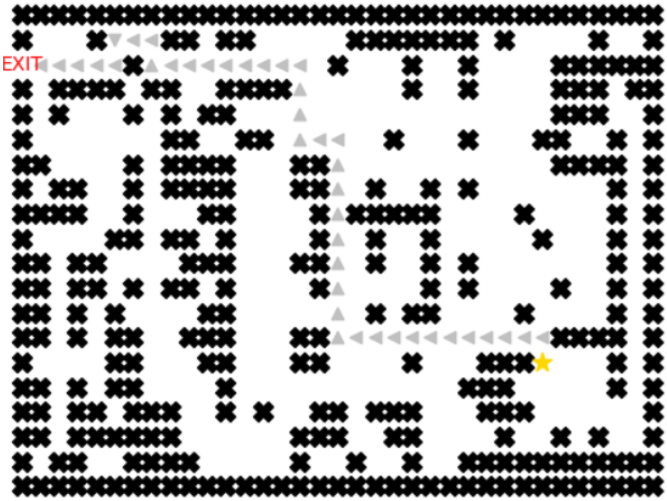


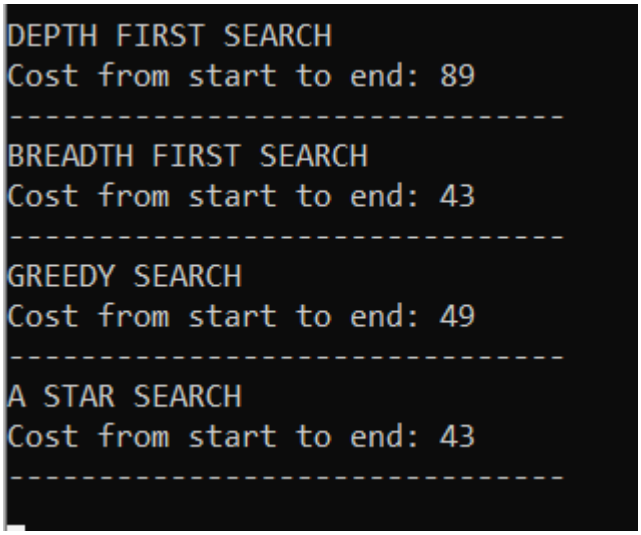
Breadth first search / maze_mapRandom2.txt

Greedy search / maze_mapRandom2.txt



A star search / maze_mapRandom2.txt





➢ *Nhận xét về các thuật toán:*

- Thuật toán DFS sẽ tìm đường đi dựa theo chiều sâu nên với 1 bản đồ lớn nên sẽ tìm ra đường đi rất dài.

- Thuật toán BFS sẽ tìm đường đi dựa trên tất cả các nút con xung quanh các điểm nên sẽ tìm được đường đi ngắn nhất nhưng phải tốn không gian gấp nhiều lần thuật toán DFS nên trong trường hợp bản đồ lớn nên sử dụng DFS

- Thuật toán tìm kiếm có thông tin Greedy Best First Search và A* search:

- Có sự chênh lệch nhiều về đường đi giữa 2 thuật toán
- Bản đồ càng lớn thì Greedy Best First Search sử dụng heuristic sẽ không thể tìm được đường đi ngắn nhất

2. BẢN ĐỒ CÓ ĐIỂM THƯỞNG:

Ý tưởng của thuật toán:

❖ Các hàm bổ trợ:

- Nhóm em xây dựng thuật toán dựa trên A* search sử dụng hàng đợi ưu tiên.

- Sử dụng hàm bonusHeuristic là khoảng cách manhattan tới điểm thưởng gần nhất nhân 1.1 vì nếu muốn ăn điểm thưởng phải giảm giá trị của f(n) xuống bằng cách tăng h(n) lên vì g(n) là không đổi.

```
#Heuristic la kc đến điểm thưởng mới bằng khoảng cách manhattan nhân với 1.1
def bonusHeuristic(current:tuple,end:tuple,newbonus:tuple,oldbonus:tuple)->float:
    toNewBonus= abs(newbonus[0]-current[0]) + abs(newbonus[1]-current[1])
    toEnd= abs(end[0]-current[0]) + abs(end[1]-current[1])
    if (toEnd <toNewBonus+newbonus[2]):#Neu kc tu diem toi end < kc tu diem toi diem thuong + gia tri diem thuong
        return toEnd #Return kc toi diem End
    result=toNewBonus*1.1
    return result
```

- Sử dụng thêm hàm manhattanBonusHeuristic để hỗ trợ thêm trong việc lựa chọn các nút khi chuyển giao giữa điểm thưởng mới và điểm thưởng cũ. H(n) = khoảng cách tới điểm thưởng mới nhân 1.2 cộng khoảng cách tới điểm thưởng cũ, sẽ làm giảm f(n) trong hàng đợi ưu tiên

```
def manhattanBonusHeuristic(current:tuple,end:tuple,newbonus:tuple,oldbonus:tuple)->float:
    toNewBonus= abs(newbonus[0]-current[0]) + abs(newbonus[1]-current[1])
    toOldBonus=abs(oldbonus[0]-current[0]) + abs(oldbonus[1]-current[1])
    toEnd= abs(end[0]-current[0]) + abs(end[1]-current[1])
    if (toEnd <toNewBonus+newbonus[2]):#Neu kc tu diem toi end < kc tu diem toi diem thuong + gia tri diem thuong
        return toEnd #Return kc toi diem End
    result=toNewBonus*1.2+toOldBonus#Nhân 1.2 và cộng thêm khoảng cách tới bonus cũ sẽ giảm giá trị f(n)
    return result
```

❖ Hàm chính chưa thuật toán:

```
def aStarSearchWithBonusPoints(start :tuple,end:tuple,matrix,bonus)->Optional[Node]:
    tempPoint1=[]
    #Tạo 1 list chứa bonus points
    for point in bonus:
        tempPoint1.append(point)
    # Sort theo khoang cach tu diem hien tai toi diem thuong tang dan
    tempPoint1.sort(key=lambda point: euclidHeuristic(tuple((point[0],point[1])),start))
    newBonus=tempPoint1.pop(0) #Lấy bonus gan nhat
    oldBonus=tuple((start[0],start[1],0)) #Chưa có old bonus, đặt là start
    openPoints=PriorityQueue()
    openPoints.push(Node(start,None,0.0,bonusHeuristic(start,end,newBonus,oldBonus)))
    closedPoints:dict[tuple,float]={start:0.0}
```

- Ban đầu, tất cả các bonus points sẽ được sắp xếp dựa trên khoảng cách euclid giữa điểm bắt đầu và các bonus points theo thứ tự tăng dần

- Tạo new bonus và old bonus để quyết định đường đi

- Khởi tạo openpoint và closedpoint như trong A* sử dụng bonusHeuristic

```
#Khi đụng tới new bonus hoac vuot qua new bonus mà không ăn bonus sẽ chuyển tới vị trí có bonus mới
startToBonus=euclidHeuristic(tuple((newBonus[0],newBonus[1])),start)
startToCurrent=euclidHeuristic(currentPoint.item,start)
if(currentPoint.item==tuple((newBonus[0],newBonus[1])) or startToBonus <startToCurrent):
    #Sap xep lai list theo khoang cach toi diem thuong
    tempPoint1.sort(key=lambda point: euclidHeuristic(tuple((point[0],point[1])),currentPoint.item))
    if len(tempPoint1)>0:
        oldBonus=newBonus
        newBonus=tempPoint1.pop(0)
        #Cap nhat lai heuric với new bonus va old bonus
        openPoints.updateStatus(end,newBonus,oldBonus)
    else: #Nêu kh còn bonus,bonus sẽ là end
        oldBonus=newBonus
        newBonus=tuple((end[0],end[1],0))
        openPoints.updateStatus(end,newBonus,oldBonus)
```

- Duyện đường đi như A* cho đến khi ăn được 1 bonus point hoặc vượt qua bonus point. Sắp xếp lại bonus points theo khoảng cách tới điểm hiện tại. Nếu như ăn được 1 bonus hoặc vượt quá sẽ xóa bonus đó khỏi bonus point

- Cập nhật lại open points dựa trên heuristic với new bonus và old bonus

```
def updateStatus(self,end,newBonus,oldBonus):
    #Update heuristic cho những điểm có thể đi, để di chuyển qua tới bonus mới từ bonus cũ
    for i in self.items:
        i.heuristic=manhattanBonusHeuristic(i.item,end,newBonus,oldBonus)
    #Sắp xếp lại theo khoảng cách đến bonus mới
    for i in range(len(self.items)-1):
        for j in range(i+1,len(self.items)):
            if (self.items[i].__lt__(self.items[j])):
                temp=self.items[i]
                self.items[i]=self.items[j]
                self.items[j]=temp
```

- Sau đó vì muốn ưu tiên di chuyển theo con đường đã ăn bonus point nên sẽ ưu tiên duyệt những nút con nút hiện tại

```
#Tạo hàng đợi ưu tiên cho các nút con của bonus
tempPoints=PriorityQueue()
for child in children(matrix,currentPoint.getItem()):
    nextCost=currentPoint.pathCost+1
    #if(child not in closedPoints ):
    closedPoints[child]=nextCost
    tempPoints.push(Node(child,currentPoint,nextCost,bonusHeuristic(child,end,newBonus,oldBonus)))
#Ghép vào openpoints
for i in tempPoints.items:
    openPoints.items.append(i)
```

Những trường hợp thuật toán chưa tối ưu:

- Khi đi vào đường chứa 2 bonus point liên tiếp mà không còn đường tiếp. Vì khi lấy bonus point sẽ xóa bonus point đó khỏi danh sách bonus points nên sẽ không duyệt ngược lại bonus point 1 mặc dù ở đó có đường đi khác tối ưu

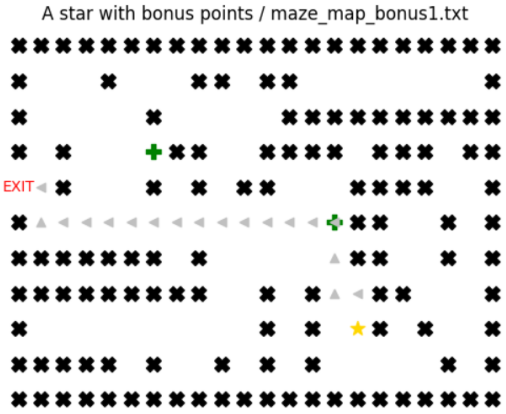
- Đi qua các nút nhiều lần để có đường đi tối ưu

- Các nút điểm thưởng nằm ở vị trí khó ăn hoặc giá trị lớn vì chỉ ăn theo khoảng cách tới điểm thưởng

Bản đồ 1:

Điểm thưởng 1: (3,6): -3

Điểm thưởng 2: (5,14): -1




```
A STAR WITH BONUS POINTS
Cost from start to end: 19
------------------------------
```

+ Nhận xét:

- Tìm được đường đi ngắn nhất từ điểm bắt đầu tới điểm kết thúc.

- Thuật toán cho đường đi tối ưu trong ma trận này, ăn được một điểm thưởng.

Bản đồ 2:

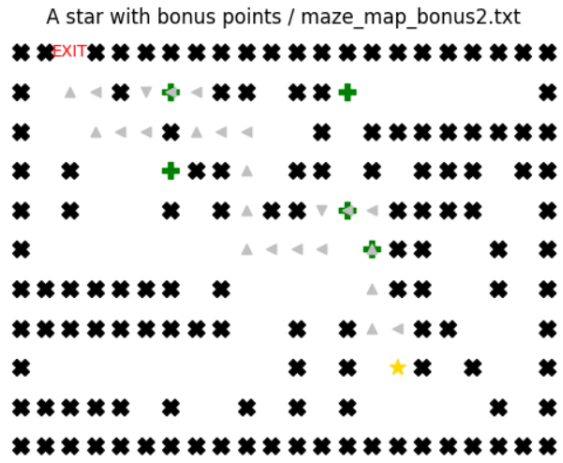
Điểm thưởng 1: (1,6): -1

Điểm thưởng 2: (1,13): -4

Điểm thưởng 3: (3,6): -3

Điểm thưởng 4: (4,13): -5

Điểm thưởng 5: (5,14): -2




```
A STAR WITH BONUS POINTS
Cost from start to end: 18
------------------------------
```

➢ *Nhận xét:*

- Đi qua được 3 trên 5 điểm thưởng

- Với các giá trị điểm thưởng thì chưa tối ưu, ăn điểm thưởng 3 thay vì 1 sẽ tối ưu hơn.

- Đi qua những điểm thưởng theo thứ tự khoảng cách.

- Thuật toán không cho đi lại những điểm đã đi qua nên không tìm được đường đi tốt hơn (đi qua điểm thưởng 3 sẽ phải đi lại 2 lần qua 1 điểm)

Bản đồ 3:

Các điểm thưởng:

(1,26)	-4
(2,7)	-1
(2,10)	-5
(3,15)	-10
(4,17)	-6
(4,22)	-9
(6,14)	-8
(6,26)	-7
(10,12)	-11
(14,18)	-2

A star with bonus points / maze_map_bonus3.txt


```
A STAR WITH BONUS POINTS
Cost from start to end: 12
--------------------------------
```

➢ *Nhận xét:*

- Đi qua được 5/10 điểm thưởng.

- Thuật toán không cho đi lại những điểm đã đi qua nên không tìm được đường đi tốt hơn

- Đi qua những điểm thưởng theo thứ tự khoảng cách.

III. TỔNG KẾT VỀ CÁC THUẬT TOÁN:

Qua những cơ sở lý thuyết và kiểm thử của 4 thuật toán trên, ta có thể kết luận về ưu nhược điểm của từng thuật toán như sau:

THUẬT TOÁN	ƯU ĐIỂM	KHUYẾT ĐIỂM
DEPTH FIRST SEARCH	Vì thuật toán sẽ đi duyệt hết tất cả các nút để tìm ra kết quả nên nếu số lượng nút là hữu hạn thì bài toán chắc chắn sẽ tìm được lời giải.	Mang tính chất vét cạn, không nên áp dụng nếu duyệt số nút quá lớn. Chỉ duyệt một cách mù quáng là duyệt tất cả các điểm, không quan tâm đến việc tối ưu thông tin chi phí ở các nút nên sẽ duyệt qua những điểm không cần thiết.
BREADTH FIRST SEARCH	BFS duyệt tất cả các nút con đến khi tìm được đích thì dừng nên thuật toán luôn tìm đường đi tối ưu nhất Cũng giống như DFS, BFS luôn tìm ra lời giải của bài toán	Mang khuyết điểm "vét cạn" và "mù quáng" giống như DFS. Tuy BFS luôn tìm ra đường đi tối ưu nhất nhưng chính điều này cũng làm cho nó tiêu tốn chi phí không gian rất nhiều, nhiều hơn so với DFS.
GREEDY	Thuật toán tham lam thường dễ tìm ra và chạy nhanh hơn các thuật toán khác (không phải tất cả).	Việc tìm kiếm một thuật toán tham lam phù hợp đòi hỏi một lập luận sắc bén.

	Không tồn tại một công thức chung nào cho thuật toán này nhưng ta có thể nhìn ra bằng việc phân tích tính chất bài toán và kinh nghiệm.	Thuật toán greedy có thể có nhiều cách giải quyết khác nhau trong một bài toán, nhưng chỉ số ít trong đó là giải quyết chính xác.
A STAR SEARCH	Thuật toán linh động, tổng quát, khi có hàm heuristic phù hợp thì nhanh chóng tìm ra lời giải cho bài toán. A* là thuật giải tiêu biểu cho heuristic	Mỗi bài toán khi áp dụng A* đều phải tìm ra heuristic phù hợp thì mới giải được. A* vẫn mang những khuyết điểm giống BFS như tiêu tốn khá nhiều bộ nhớ để lưu thông tin.

TÀI LIỆU THAM KHẢO:

Solving Mazes in Python: Depth-First Search, Breadth-First Search, & A* - YouTube

algorithm - When is it practical to use Depth-First Search (DFS) vs Breadth-First Search (BFS)? - Stack Overflow

https://cs.adelaide.edu.au/~dsuter/Harbin_course/UninformedSearch.pdf

https://vi.wikipedia.org/

https://www.stdio.vn/giai-thuat-lap-trinh/thuat-toan-breadth-first-search-sBPnH

https://www.stdio.vn/giai-thuat-lap-trinh/thuat-toan-depth-first-search-APnHi1

https://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_r%E1%BB%99ng

https://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u

https://vi.wikipedia.org/wiki/Gi%E1%BA%A3i_thu%E1%BA%ADt_tham_lam

https://vi.wikipedia.org/wiki/Gi%E1%BA%A3i_thu%E1%BA%ADt_t%C3%ACm_ki%E1%BA%BFm_A*

----Hết----