ĐẠI HỌC QUỐC GIA
THÀNH PHỐ HỒ CHÍ MINH

BÁO CÁO ĐỒ ÁN 1

Nghiên cứu, cài đặt và trình bày các thuật toán tìm kiếm đường đi

19120622 – Nguyễn Minh Phụng

19120678 – Nguyễn Hoàng tiến

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHTN
TP. HỒ CHÍ MINH

Mục lục

Các hàm thành phần

Mỗi điểm trên bản đồ sẽ được gọi là một node, sử dụng class Node làm cấu trúc:

```
class Node:
    def __init__(self, x, y, reward=0, f=0, g=0, h=0, neighbors=[], previous=None, isWall=False, isVisited=False):
        self.x = x
        self.y = y
        self.reward = reward
        self.f = f
        self.g = g
        self.h = h
        self.neighbors = neighbors[:]
        self.previous = previous
        self.isWall = isWall
        self.isVisited = isVisited
```

Một node sẽ bao gồm:

- x, y: tọa độ của node trên bản đồ
- reward: giá trị điểm thưởng (nếu node là một điểm thưởng)
- f: có giá trị f = g + h
- g: chi phí đi từ điểm Start đến node
- h: chi phí ước lượng heuristic từ node đến điểm Exit
- neighbors: mảng các node kề với node hiện tại
- previous: tham chiếu đến node cha của node hiện tại
- isWall: biến boolean để phân biệt node đường đi và node là tường
- isVisited: biến boolean để phân biệt node đã được hoặc chưa được đi qua

```
def add_neighbors(self, graph):
    if self.isWall:
        return

    m = len(graph)
    n = len(graph[0])
    if self.x + 1 < m and not graph[self.x + 1][self.y].isWall:
        self.neighbors.append(graph[self.x + 1][self.y])

    if self.x - 1 > -1 and not graph[self.x - 1][self.y].isWall:
        self.neighbors.append(graph[self.x - 1][self.y])

    if self.y + 1 < n and not graph[self.x][self.y + 1].isWall:
        self.neighbors.append(graph[self.x][self.y + 1])

    if self.y - 1 > -1 and not graph[self.x][self.y - 1].isWall:
        self.neighbors.append(graph[self.x][self.y - 1])
```

Hàm `addNeighbors(self, graph)` dùng để thêm các node kề vào neighbors của node hiện tại, thứ tự các node được thêm là: dưới, trên, phải, trái

```
def getRoute(start, end):
    route = [end]
    while not route[-1].isEqual(start):
        route.append(route[-1].previous)
    route.reverse()
    return route
```

Về cơ bản, các thuật toán tìm kiếm hoạt động dựa trên các công đoạn nối node phục vụ mục đích tìm đường đi.
`getRoute(start, end)` trả về tập các node để biểu diễn đường đi cuối cùng của thuật toán.

```
def Manhattan(node, end):
    h = abs(node.x - end.x) + abs(node.y - end.y)
    return h

def Euclidean(node, end):
    h = math.sqrt((node.x - end.x)**2 + (node.y - end.y)**2)
    return h

def Distance(node, end, type):
    if type == 1:
        return abs(node.x - end.x) + abs(node.y - end.y)
    elif type == 2:
        return math.sqrt((node.x - end.x)**2 + (node.y - end.y)**2)

def DistanceX(start, end, node, type):
    return Distance(start, node, type) + Distance(node, end, type) + node.reward - Distance(start, end, type)

def Heuristic(node, end, type = 1):
    if type == 1:
        node.h = Manhattan(node, end)
    elif type == 2:
        node.h = Euclidean(node, end)
```

Các hàm hỗ trợ công đoạn tính toán Heristic của node:

- `Manhattan(node, end)` tính khoảng cách giữ 2 node bất kì dựa trên công thức Mannhattan.
- `Euclidean(node, end)` tính khoảng cách giữ 2 node bất kì dựa trên công thức Euclidean.
- `Distance(node, end, type)` trả về khoảng cách giữ 2 node bất kì, phụ thuộc vào đối số type mà dùng công thức bất kì (1: Manhattan, 2: Euclidean).
- `DistanceX(start, end, node, type)` tính toán sự chênh lệch chi phí tương đối giữa 2 lộ trình (lộ trình từ start đến end mà không đi qua node và đi qua node lấy điểm thưởng).
- `Heuristic(node, end, type = 1)` cập nhật thuộc tính Heuristic của node lấy end làm đối số.

Các thuật toán tìm kiếm

BFS (Breadth-first Search):

Định nghĩa:

- BFS là một thuật toán dùng để duyệt hoặc tìm kiếm trong đồ thị. Thuật toán bắt đầu ở 1 node gốc và xét tất cả những node ở độ sâu hiện tại trước khi xét tiếp những node ở độ sâu tiếp theo.
- Chúng ta thường sử dụng queue để lưu những node sẽ được xét.
- Trong đồ thị không có trọng số thì BFS luôn tìm ra đường đi ngắn nhất giữa 2 node.

Code:

```
def BFS(start, end, bonus_points = None):
    queue = []
    queue.append(start)
    while queue:
        node = queue.pop(0)
        node.isVisited = True
        if node.is_equal(end):
            route = get_route(start, end)
            return route, len(route)
        for neighbor in node.neighbors:
            if neighbor not in queue and not neighbor.isVisited:
                neighbor.previous = node
                queue.append(neighbor)
```

Giải thích: Thuật toán sẽ bắt đầu xét từ việc thêm node đầu tiên (node gốc) vào queue, sau đó lấy ra node đầu tiên (current node) trong queue và lần lượt thêm các node kề trực tiếp của current node vào sau queue nếu các node kề này chưa được xét. Do tính chất First In, First Out (FIFO) của queue nên những node kề gần nhất sẽ đảm bảo được duyệt trước.

Độ hoàn thiện: đảm bảo tìm được đường đi

Tính tối ưu: đảm bảo tìm được đường đi tốt nhất

Độ phức tạp không gian: $O(b^m)$ do có thể phải duyệt tất cả node trong đồ thị

Độ phức tạp thời gian $O(b^m)$ do có thể phải lưu tất cả node trong đồ thị

Trong đó:

b: số nhánh tối đa của cây, m: chiều sâu tối đa của cây

Khi nào nên sử dụng BFS:

- Không gian không bị hạn chế
- Cần thiết phải tìm lời giải tốt nhất

Khi nào không nên sử dụng BFS:

- Không gian bị hạn chế
- Lời giải nằm ở độ sâu lớn

DFS (Depth-first Search):

Định nghĩa:

- DFS là một thuật toán dùng để duyệt hoặc tìm kiếm trong đồ thị. Thuật toán bắt đầu ở 1 node gốc và duyệt theo chiều sâu nhất có thể (duyệt hết 1 nhánh chiều sâu có thể rồi mới quay lui để duyệt nhánh khác)
- Chúng ta thường sử dụng stack để lưu những node sẽ được xét.
- DFS không đảm bảo đường đi ngắn nhất giữa 2 node trong đồ thị.

Code:

```
def DFS(start, end, bonus_points = None):
    stack = []
    stack.append(start)
    while stack:
        node = stack.pop()
        node.isVisited = True
        if node.is_equal(end):
            route = get_route(start, end)
            return route, len(route)
        for neighbor in node.neighbors:
            if not neighbor.isVisited:
                neighbor.previous = node
                stack.append(neighbor)
```

Giải thích: Thuật toán sẽ bắt đầu xét từ việc thêm node đầu tiên (node gốc) vào stack, sau đó lấy ra node cuối cùng (current node) trong stack và lần lượt thêm các node kề trực tiếp của current node vào sau stack nếu các node kề này chưa được xét. Do tính chất Last In, First Out (LIFO) của stack nên những node kề xa nhất sẽ được đảm bảo duyệt trước.

Độ hoàn thiện: DFS đảm bảo tìm được đường đi đối với đồ thị có hữu hạn node

Tính tối ưu: DFS không đảm bảo tìm được đường đi tốt nhất

Độ phức tạp không gian: O(bm) do đối với mỗi node ở một độ sâu thì phải lưu b node kề

Độ phức tạp thời gian: $O(b^m)$ do có thể phải duyệt tất cả node trong đồ thị

Do thứ tự các node kề được thêm vào là dưới, trên, phải, trái nên khi sau khi được thêm vào stack, thứ tự các node kề được lấy ra để xét là trái, phải, trên, dưới => Thuật toán sẽ ưu tiên đi sâu nhất có thể theo thứ tự về phía bên trái, phải, trên, dưới.

Khi nào nên sử dụng DFS:

- Không gian bị hạn chế.
- Biết cách sắp xếp thứ tự thêm các node kề để tối ưu lời giải hơn.

Khi nào không nên sử dụng DFS:

- Đồ thị có vòng

- Các lời giải có độ chênh lệch chiều sâu lớn

Greedy Search (Best First Search)

Định nghĩa:

- Greedy là thuật toán toán dùng để duyệt hoặc tìm kiếm trong đồ thị. Đây là thuật toán tìm kiếm có thông tin, cụ thể dựa trên hàm Heuristic nhất định mà độ hiệu quả của đường đi phụ thuộc vào thiết kế của người lập trình.
- Mỗi node có các thuộc tính f, g, h. Trong Greedy g mọi node = 0, h là giá trị Heuristic, f = g + h
- Thuật toán bắt đầu ở node gốc và tham lam duyệt các node có chi phí so với node đích là ngắn nhất. Có thể xem là thuật toán BFS mở rộng.
- Thuật toán thường sử dụng queue để hỗ trợ việc xét và lưu các node.

Code:

```
def increasingSort(arr, end, type = 1): # Calculate the distance from each node to END point
    for i in range(len(arr)):           # Rearrange nodes due to it
        Heuristic(arr[i], end, type)
        arr[i].f = arr[i].g + arr[i].h

    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if j < len(arr):
                if arr[i].f > arr[j].f:
                    swapArrElement(arr,i,j)
```

```
def Greedy(start, end, type = 1, bonus_points = None):
    PriorityQueue = []
    PriorityQueue.append(start)

    while PriorityQueue:
        increasingSort(PriorityQueue, end)
        node = PriorityQueue.pop(0)
        node.isVisited = True
        if node.isEqual(end):
            route = getRoute(start,node)
            return route, len(route)

        for neighbor in node.neighbors:
            if neighbor not in PriorityQueue and not neighbor.isVisited:
                neighbor.previous = node
                PriorityQueue.append(neighbor)
```

Hàm `increasingSort(arr, end, type = 1)` sắp xếp lại mảng các node theo thứ tự từ bé đến lớn dựa trên giá trị Heuristic của từng node, lấy end làm đối số.

Giải thích:

- Thuật toán bắt đầu từ việc thêm node đầu (node gốc) vào queue. Sau đó, ta sắp xếp lại các node trong queue rồi lần lượt lấy phần tử đầu tiên của queue ra xét cho đến khi queue rỗng. Với mỗi node được lấy ra, ta xét rồi lại thêm các node kề trực tiếp của nó vào queue (nếu các node đó chưa duyệt).

- Với mỗi vòng lặp, ta sắp xếp lại queue rồi duyệt phần tử đầu của queue cũng là phần tử có chi phí ước tính thấp nhất cho đến khi tìm được node đích (thỏa mãn tính chất tham lam của **thuật** toán).

Độ hoàn thiện: đảm bảo tìm được đường đi.
Tính tối ưu: không đảm bảo tìm được đường đi ngắn nhất.
Độ phức tạp không gian: $O(b^m)$
Độ phức tạp thời gian: $O(b^m)$
Khi nào nên sử dụng Greedy Search:
Khi lộ trình mà thuật toán tham lam lập ra không bị cản ngăn nhiều, bởi tính tham lam nên thuật toán luôn cố gắng tìm đường đi mà nó ước tính chi phí là nhỏ nhất.
Khi nào không nên sử dụng Greedy Search:
Khi độ thoáng lối của lộ trình thiết lập dựa trên Heuristic không khả thi.

A* Search

- A* là thuật toán tìm kiếm được dùng dể duyệt và tìm kiếm trong đồ thị. Đây là thuật toán tìm kiếm có thông tin, dựa trên 2 yếu tố:
- g(n) chi phí đường đi từ điểm đầu đến node n.
- h(n): cũng là hàm Heuristic tính toán chi phí ước tính để đến được node n.
- f(n) = g(n) + h(n) mà f là yếu tố mà ta xét khi duyệt mỗi node.
- Thuật toán bắt đầu ở node gốc, qua xem xét giá trị f mà duyệt các node cùng các node kề lận cận đến khi chạm được node đích.
- Thuật toán sử dụng 2 list là openList và closedList để hỗ trợ xét duyệt các node, duy trì tính chính xác của thuật toán.
- Đây là thuật toán nhóm chọn để lập kế hoạch lộ trình đối với các đồ thị có điểm cộng.

Có 2 hướng:

- **Quét cạn**: ưu tiên lấy hết điểm cộng trước khi tìm đường đến đích.

```
def increasingSort(arr, end, type = 1):
    for i in range(len(arr)):
        Heuristic(arr[i], end, type)
        arr[i].g = arr[i].previous.g + 1
        arr[i].f = arr[i].g + arr[i].h

    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if j < len(arr):
                if arr[i].f > arr[j].f:
                    swapArrElement(arr,i,j)

def bonus_iSort(arr, end, type = 1):
    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if j < len(arr):
                if Distance(arr[i], end, type) > Distance(arr[j], end, type):
                    swapArrElement(arr, i, j)
```

- `increasingSort(arr, end, type = 1)` sắp xếp mảng các node theo thứ tự từ bé đến lớn dựa vào yếu tố f, lấy end làm đối số.
- `bonus_iSort(arr, end, type = 1)` sắp xếp mảng các node (cụ thể là các node điểm cộng) từ bé đến lớn dựa vào chi phí ước tính đi từ từng phần tử đến node end.

```python
def A_Star(start, end, bonus_points = [], type=1):
    if len(bonus_points) == 0:
        openList = []
        closedList = []
        openList.append(start)

        while openList:
            node = openList.pop(0)
            node.isVisited = True
            closedList.append(node)
            if node.isEqual(end):
                route = getRoute(start, node)
                return route, len(route)

            for neighbor in node.neighbors:
                if neighbor in closedList:
                    continue
                tempf = node.g + 1 + Distance(neighbor, end, type)
                if neighbor in openList and tempf >= openList[openList.index(neighbor)].f:
                    continue
                openList.append(neighbor)
                neighbor.previous = node

        increasingSort(openList, end, type)

    else:
        cost = 0
        for node in bonus_points:
            cost += node.reward
        bonus_iSort(bonus_points, start, type)
        node = bonus_points.pop(0)
        route = A_Star(start, node, [], type)[0]
        wayPaving(route)
        route.pop()
        route += A_Star(node, end, bonus_points, type)[0]
        return route, len(route) + cost
```

Giải thích:

- Trong trường hợp không có điểm cộng trên đồ thị, thuật toán bắt đầu từ việc thêm node đầu tiên (node gốc) vào openList (là list chứa các node sẽ được duyệt), sau đó duyệt phần tử đầu của openList đồng thời xem các node liền kề để lại thêm vào openList. Về cơ bản, các node trong A* luôn được cập nhật liên tục các giá trị g (do có nhiều lộ trình để đến được node bất kì), việc so sánh lại các giá trị f của node trong openList là cần thiết (tránh bỏ xót các đường đi ngắn). Các node khi được duyệt sẽ được đưa vào closedList và thuộc tính g của node cũng được xem là tối ưu nhất (ít nhất là trong đồ thị không trọng số như các mê cung). Sau mỗi lần duyệt sẽ sắp xếp lại openList ưu tiên bé hơn lên đầu để đảm báo tính đúng đắn của thuật toán. Các vòng lặp được thực hiện cho đến khi tìm được lộ trình đến đích.
- Trong trường hợp có điểm cộng, chiến lược lộ trình sẽ là phân nhỏ thành các lộ trình từ điểm gốc khi đi qua tất các điểm cộng sau đó đến đích. Đơn giản là cộng dồn các lộ trình thành phần, sau mỗi lần đi qua một điểm thì tái sắp xếp lại mảng điểm cộng để đưa các điểm cộng tiềm năng (có chi phí ước tính gần hơn so với điểm hiện hành) lên đầu cho lần cộng dồn sau.

- **Suy xét**: khả năng đi qua điểm cộng sẽ phụ thuộc vào thuật toán so sánh sự chênh lệch của chi phí ước tính (dựa trên một số điều kiện nhất định)

```
def increasingSort(arr, end, type = 1):
    for i in range(len(arr)):
        Heuristic(arr[i], end, type)
        arr[i].g = arr[i].previous.g + 1
        arr[i].f = arr[i].g + arr[i].h

    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if j < len(arr):
                if arr[i].f > arr[j].f:
                    swapArrElement(arr,i,j)

def bonus_iSort(arr, start, end, type = 1):
    i = len(arr) - 1
    while i >= 0:
        if (DistanceX(start, end, arr[i], type) >= 0):
            arr.remove(arr[i])
        i-= 1

    if len(arr) > 1:
        for i in range(len(arr)):
            for j in range(i+1, len(arr)):
                if j < len(arr):
                    if DistanceX(start, end, arr[i], type) > DistanceX(start, end, arr[j], type):
                        swapArrElement(arr, i, j)
```

- `increasingSort(arr, end, type = 1)` sắp xếp mảng các node theo thứ tự từ bé đến lớn dựa vào yếu tố f, lấy end làm đối số.
- `bonus_iSort(arr, start, end, type = 1)` sắp xếp mảng các node (cụ thể là các node điểm cộng) từ bé đến lớn dựa vào sự chênh lệch chi phí ước tính giữa 2 lộ trình giữa 2 node start end khi đi qua điểm cộng và khi không đi qua điểm cộng. Chênh lệch chi phí thấp sẽ đứng trước (chỉ giữ lại chi phí ước tính âm – tức đi từ start qua nhận điểm cộng rồi sang end sẽ ít tốn chi phí hơn đi từ start đến end).

```python
def A_Star_2(start, end, bonus_points = [], type=1):
    if len(bonus_points) == 0:
        openList = []
        closedList = []
        openList.append(start)

        while openList:
            node = openList.pop(0)
            node.isVisited = True
            closedList.append(node)
            if node.isEqual(end):
                route = getRoute(start, node)
                return route, len(route)

            for neighbor in node.neighbors:
                if neighbor in closedList:
                    continue
                tempf = node.g + 1 + Distance(neighbor, end, type)
                if neighbor in openList and tempf >= openList[openList.index(neighbor)].f:
                    continue
                openList.append(neighbor)
                neighbor.previous = node

        increasingSort(openList, end, type)

    else:
        cost = 0
        bonus_iSort(bonus_points, start, end, type)
        if len(bonus_points) == 0:
            return A_Star_2(start, end, [], type)
        node = bonus_points.pop(0)
        cost += node.reward
        route = A_Star_2(start, node, [], type)[0]
        wayPaving(route)
        route.pop()
        route += A_Star_2(node, end, bonus_points, type)[0]
        return route, len(route) + cost
```

Giải thích:

Đối với trường hợp không có điểm cộng thì sẽ tương tự như trên.

Đối với trường hợp có điểm cộng, lộ trình được phân tách thành các lộ trình nhỏ hơn đi qua các điểm cộng tiềm năng dựa vào chiến lược đã xây dựng trước đó trong bonus_iSort

Độ hoàn thiện: đảm bảo tìm được đường đi.

Tính tối ưu: đảm bảo tìm được đường đi ngắn nhất khi thỏa mãn hàm Heuristic hợp lý và nhất quán (đối với các đồ thị không điểm cộng).

Độ tạp không gian: $O(b^m)$

Độ phức tạp thời gian: $O(b^m)$

Khi nào nên sử dụng A* Search:

Dùng trong không gian nhỏ hoặc vừa sẽ ổn hơn, vì thường lưu hết các node trong bộ nhớ.

Lời giải tối ưu trong các đồ thị với các cạnh không trọng số.

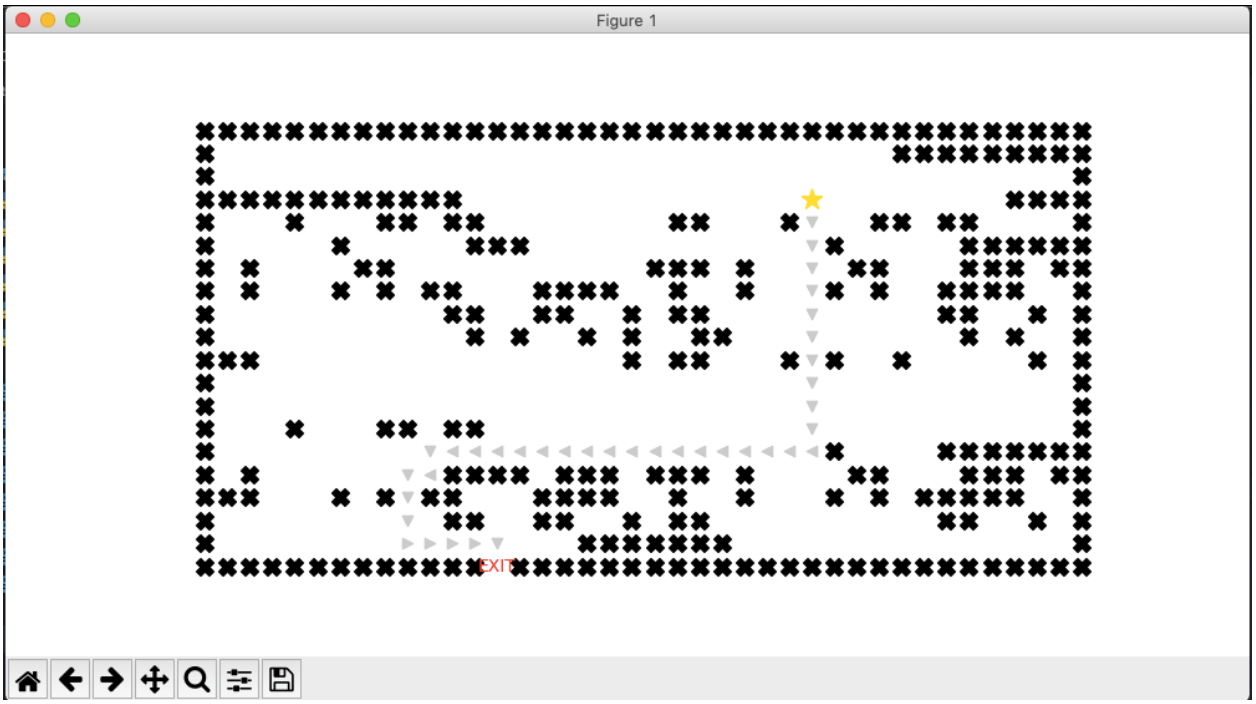
Khi nào không không nên sử dụng A* Search:

Khi kích thước đồ thị cần duyệt quá lớn, khiến quá tải bộ nhớ lưu trữ (A* thực hiện nhiều thao tác lưu node)

Lộ trình

Bản đồ không điểm thưởng

Map 1:



BFS cost 39

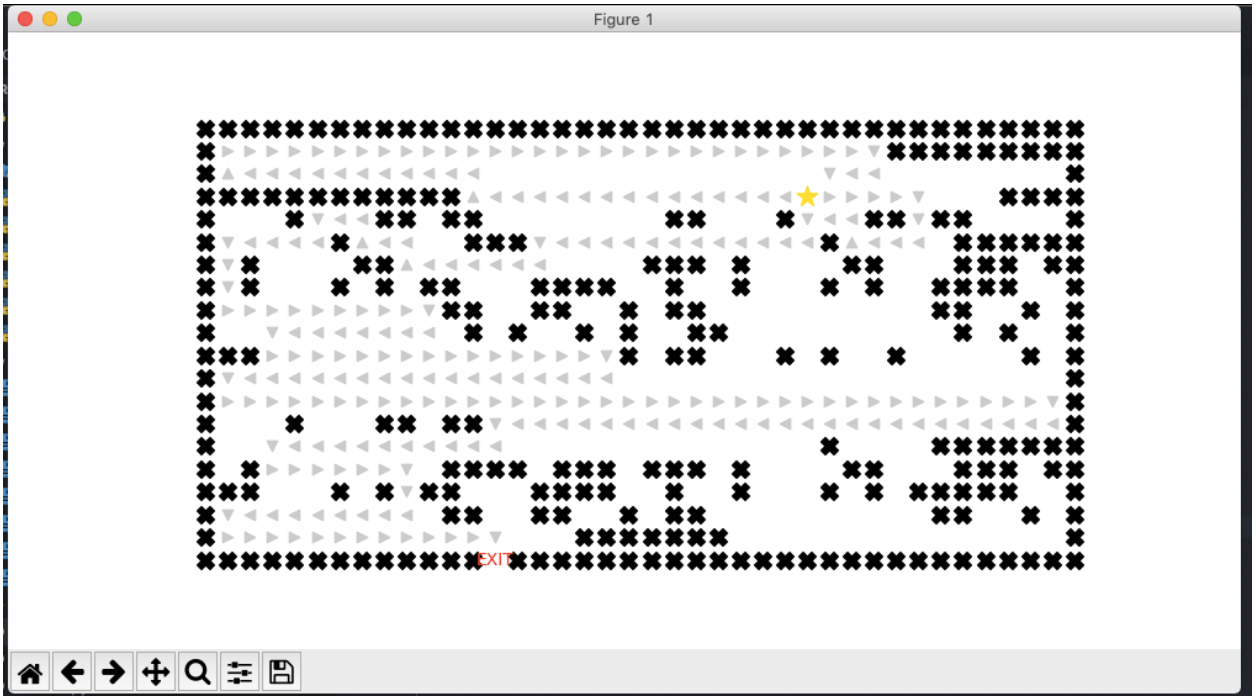
Độ hoàn thiện: tìm được đường đi

Tính tối ưu: tìm được đường đi ngắn nhất

Độ phức tạp về mặt thời gian: $O(b^m)$

Độ phức tạp về mặt không gian: $O(b^m)$

Nhận xét: độ phức tạp không gian, thời gian vẫn chưa tốt.



DFS cost 265

Độ hoàn thiện: tìm được đường đi

Tính tối ưu: không tìm được đường đi ngắn nhất

Độ phức tạp về mặt thời gian: $O(b^m)$

Độ phức tạp về mặt không gian: O(bm)

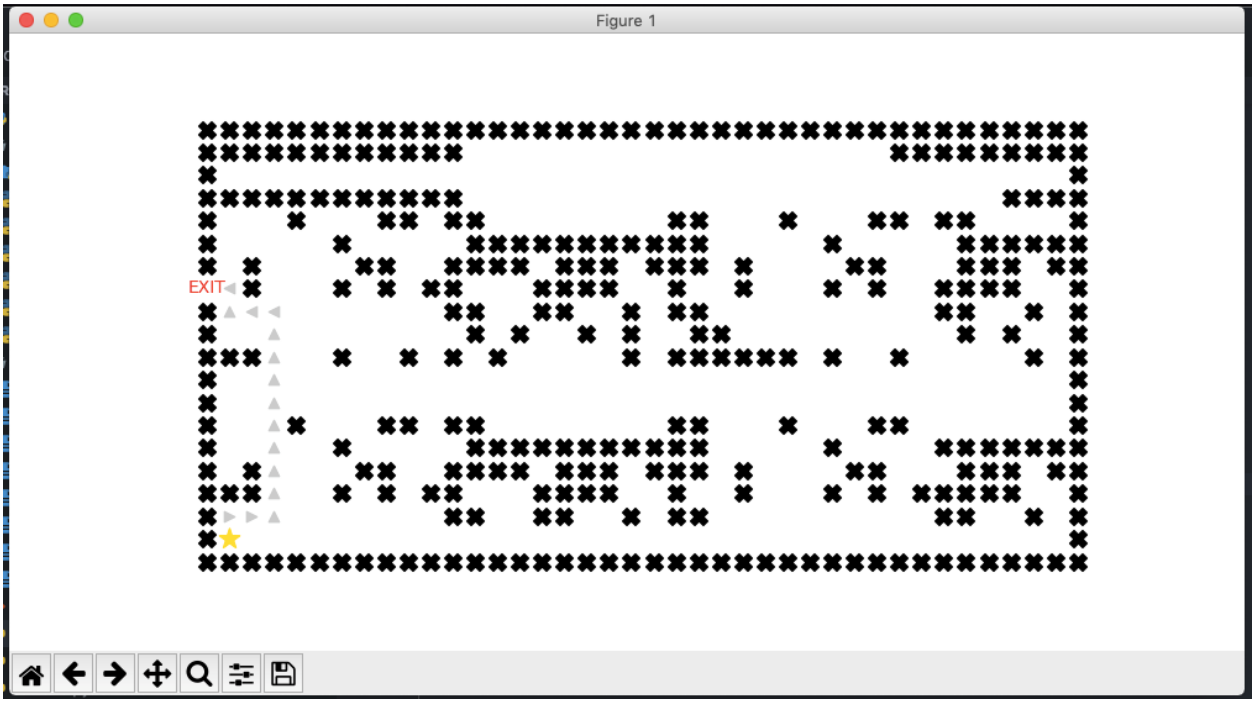
Nhận xét: độ phức tạp thời gian vẫn chưa tốt, đường đi rất tệ (rất dài).

A* Search cost 39 (Manhattan)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

A* Search cost 39 (Euclidean)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

Greedy Search cost 47 (Manhattan)
Nhận xét:
Tìm được đường đi, chưa tối ưu.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

Map 2:



BFS cost 17

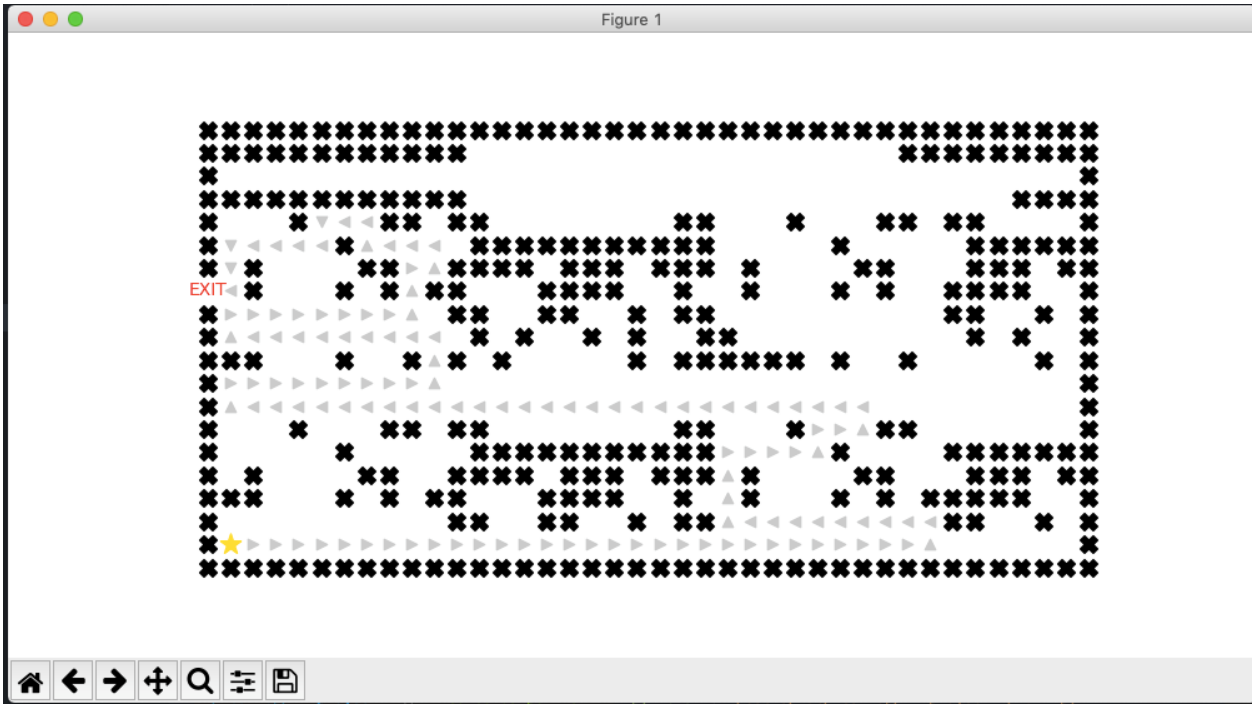
Độ hoàn thiện: tìm được đường đi

Tính tối ưu: tìm được đường đi ngắn nhất

Độ phức tạp về mặt thời gian: $O(b^m)$

Độ phức tạp về mặt không gian: $O(b^m)$

Nhận xét: độ phức tạp không gian, thời gian vẫn chưa tốt.



DFS cost 129

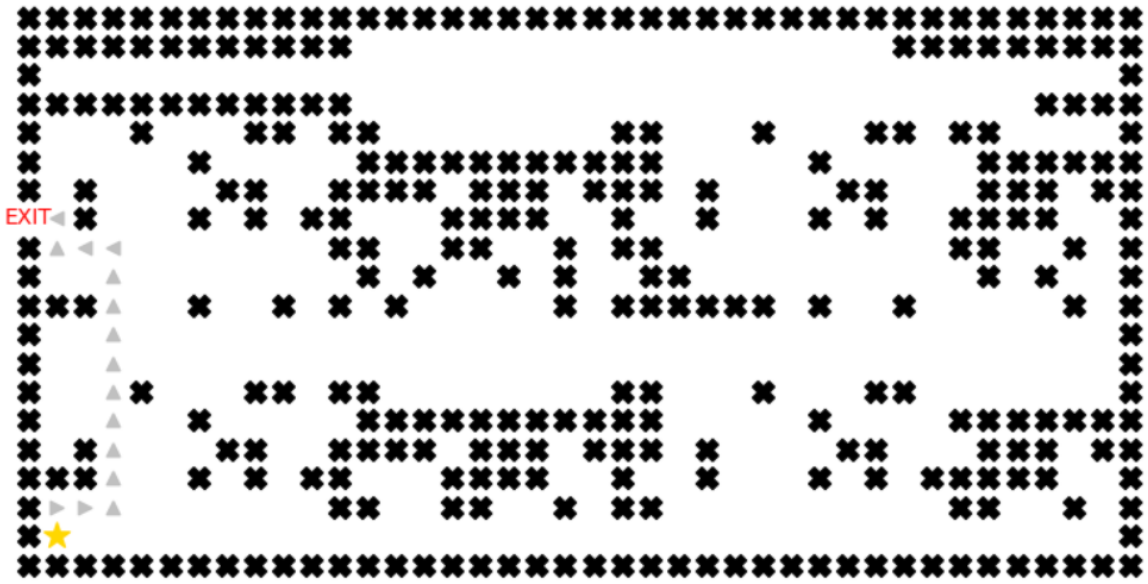
Độ hoàn thiện: tìm được đường đi

Tính tối ưu: không tìm được đường đi ngắn nhất

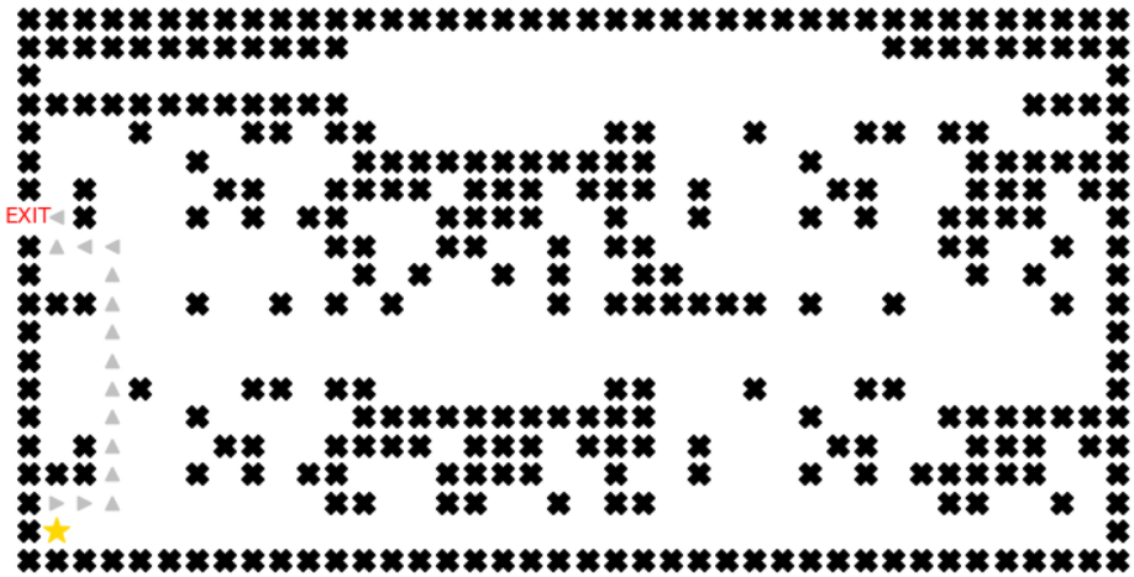
Độ phức tạp về mặt thời gian: $O(b^m)$

Độ phức tạp về mặt không gian: O(bm)

Nhận xét: độ phức tạp thời gian vẫn chưa tốt, đường đi rất tệ (rất dài).

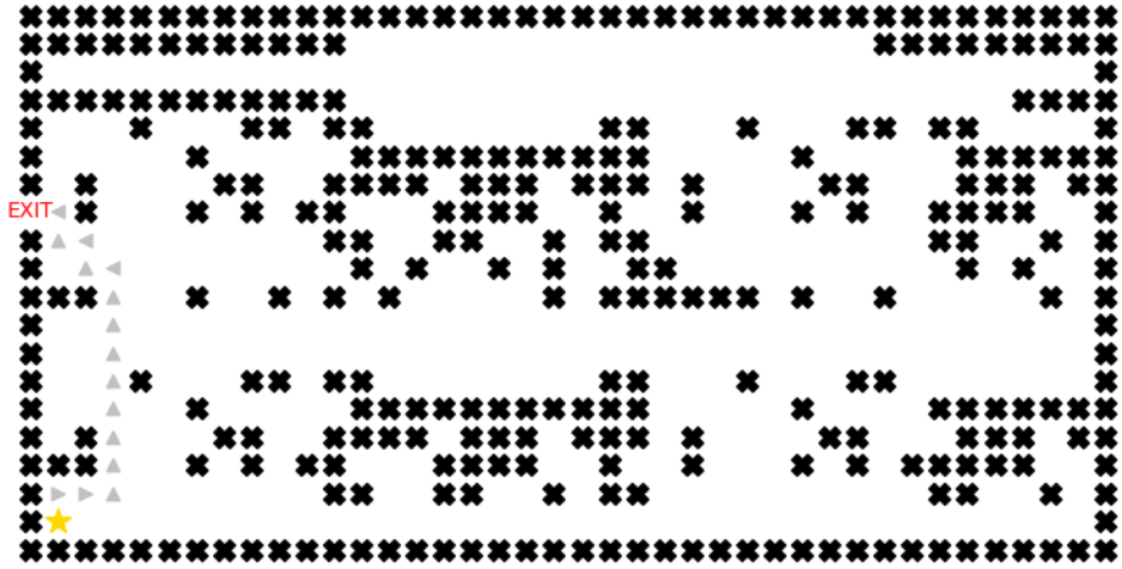


A* Search cost 17 (Manhattan)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.



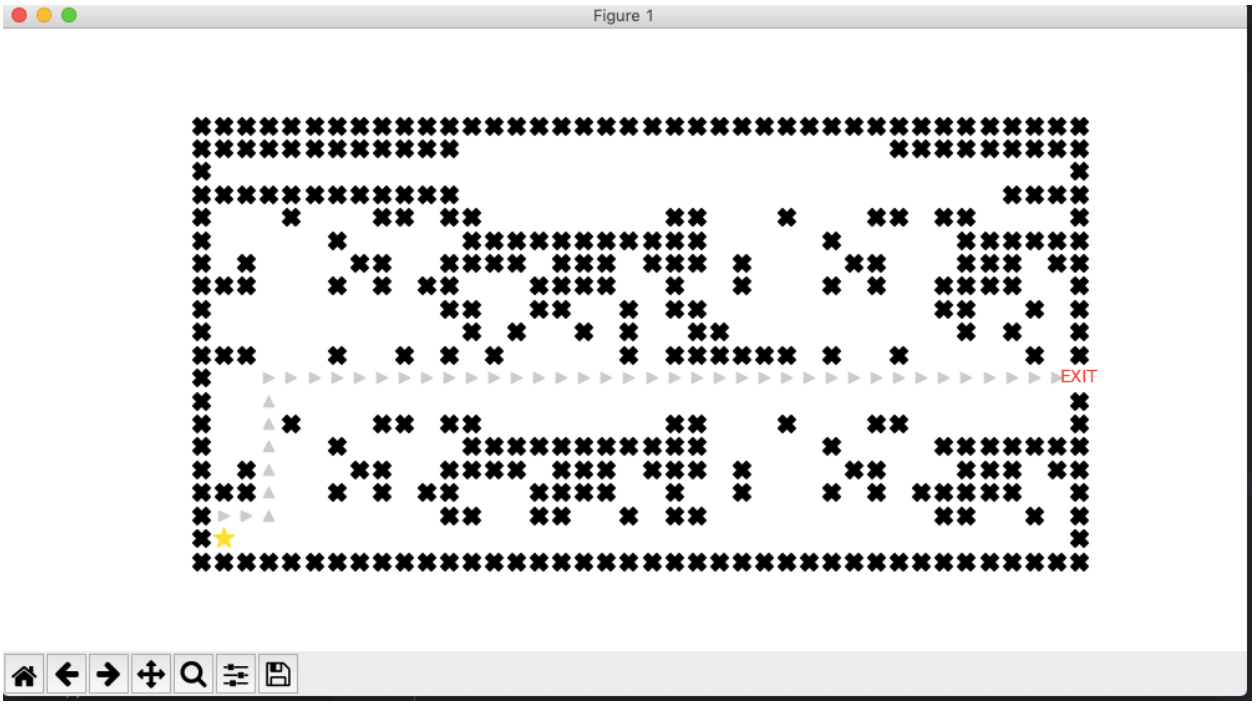
A* Search cost 17 (Euclidean)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

Greedy Search cost 17 (Manhattan)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.



Greedy Search cost 17 (Euclidean)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

Map 3:



BFS cost 46

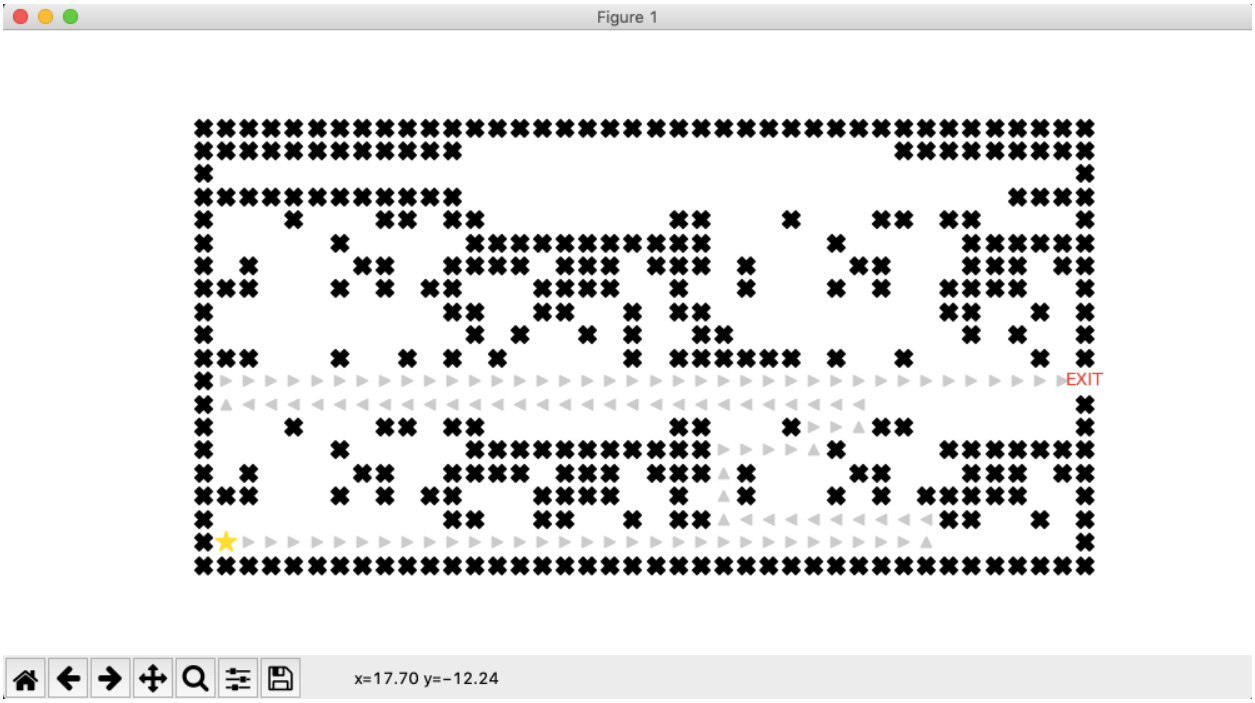
Độ hoàn thiện: tìm được đường đi

Tính tối ưu: tìm được đường đi ngắn nhất

Độ phức tạp về mặt thời gian: $O(b^m)$

Độ phức tạp về mặt không gian: $O(b^m)$

Nhận xét: độ phức tạp không gian, thời gian vẫn chưa tốt.



DFS cost 120

Độ hoàn thiện: tìm được đường đi

Tính tối ưu: không tìm được đường đi ngắn nhất

Độ phức tạp về mặt thời gian: $O(b^m)$

Độ phức tạp về mặt không gian: O(bm)

Nhận xét: độ phức tạp thời gian vẫn chưa tốt, đường đi tệ (dài).

A* Search cost 46 (Manhattan)

Nhận xét:

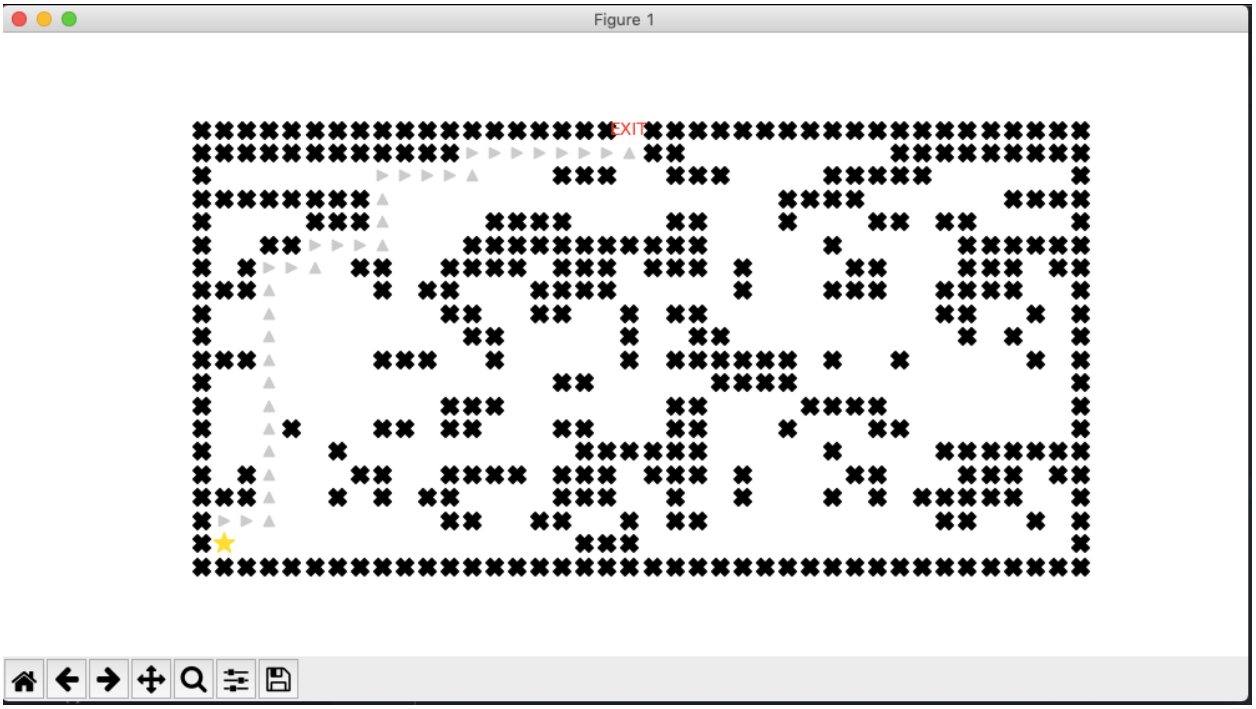
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

A* Search cost 46(Euclidean)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

Greedy Search cost 46 (Manhattan)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

Greedy Search cost 46(Euclidean)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

Map 4:



BFS cost 37

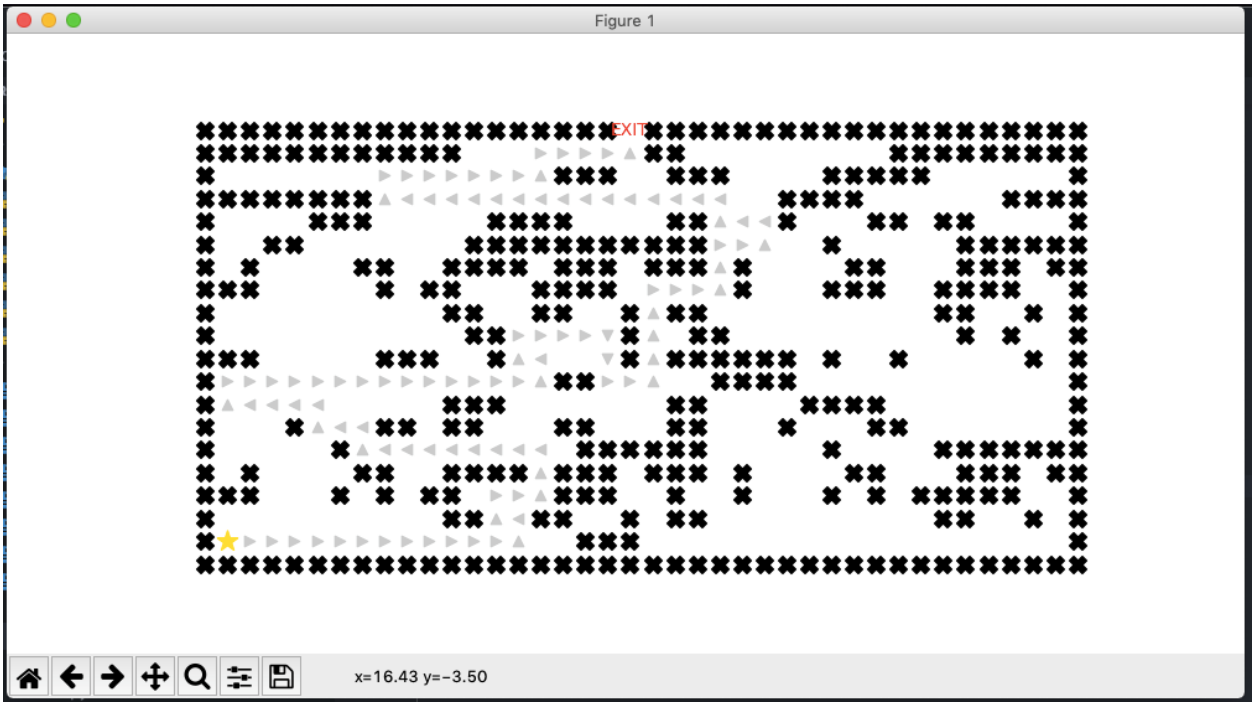
Độ hoàn thiện: tìm được đường đi

Tính tối ưu: tìm được đường đi ngắn nhất

Độ phức tạp về mặt thời gian: $O(b^m)$

Độ phức tạp về mặt không gian: $O(b^m)$

Nhận xét: độ phức tạp không gian, thời gian vẫn chưa tốt.



DFS cost 107

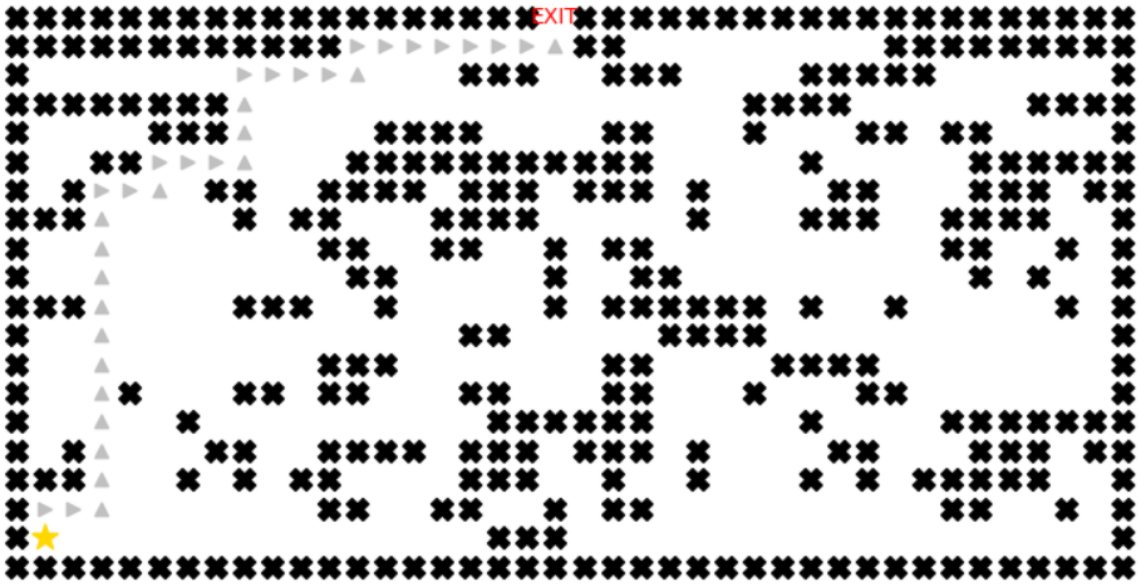
Độ hoàn thiện: tìm được đường đi

Tính tối ưu: không tìm được đường đi ngắn nhất

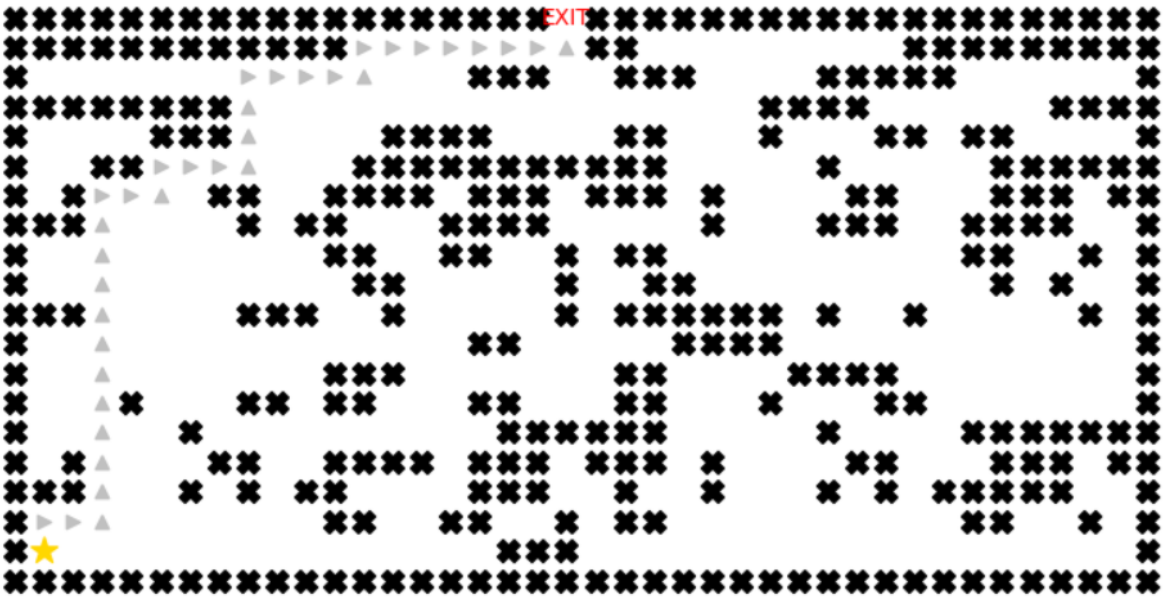
Độ phức tạp về mặt thời gian: $O(b^m)$

Độ phức tạp về mặt không gian: O(bm)

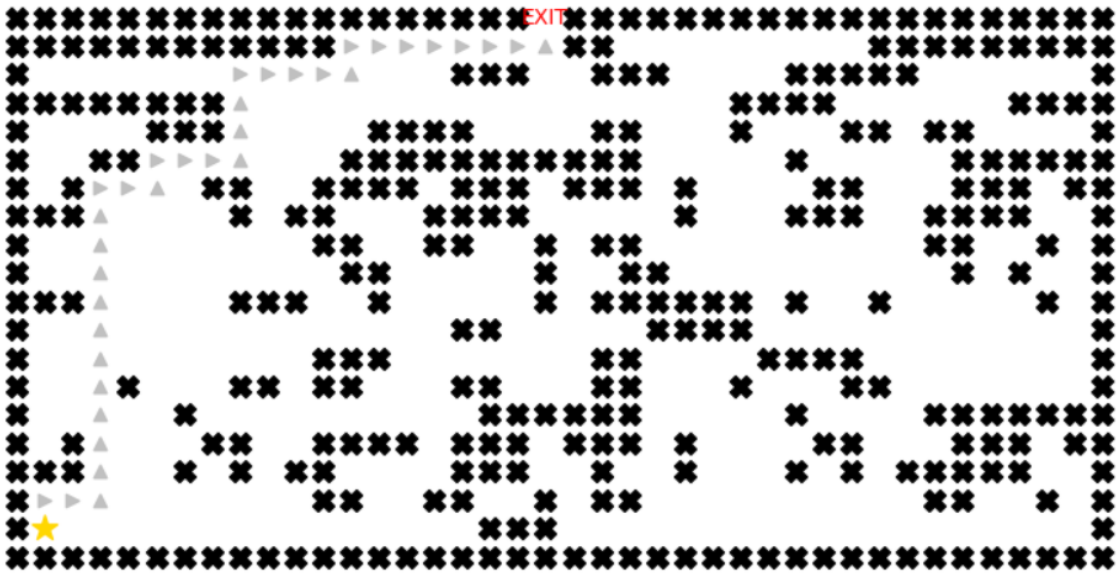
Nhận xét: độ phức tạp thời gian vẫn chưa tốt, đường đi tệ (dài).



A* Search cost 37 (Manhattan)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.



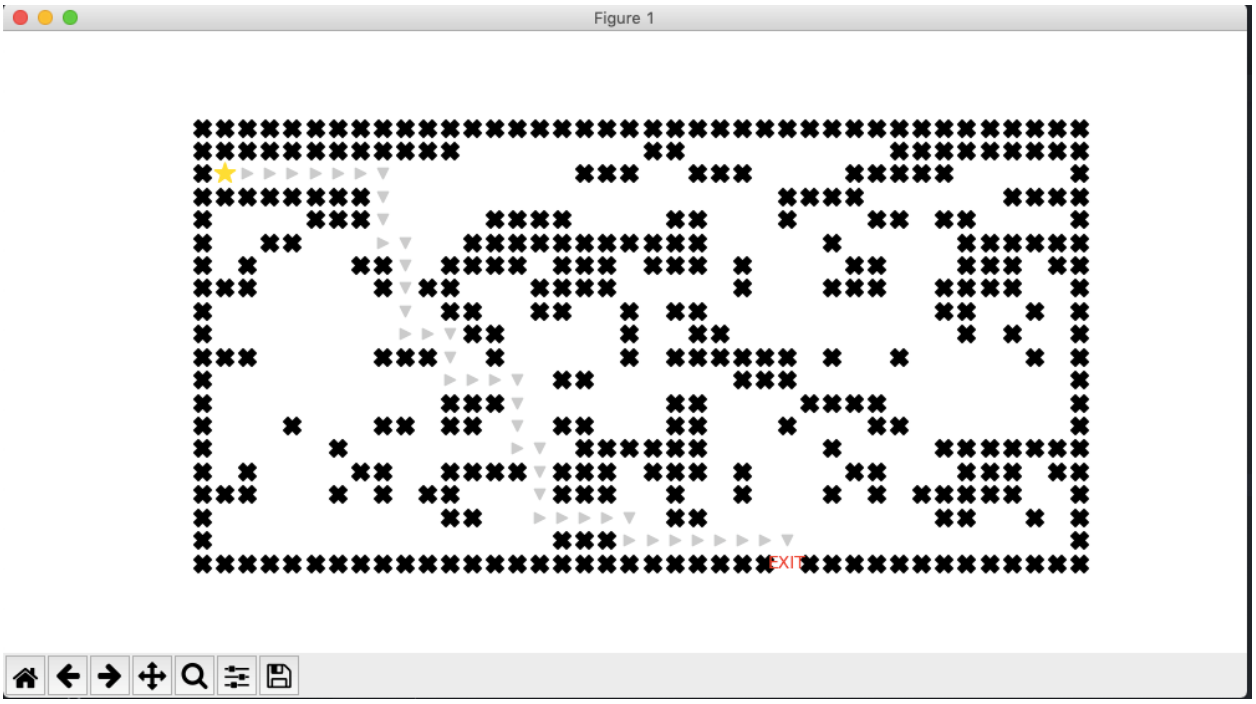
A* Search cost 37 (Euclidean)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.



Greedy Search cost 37 (Manhattan)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

Greedy Search cost 43 (Euclidean)
Nhận xét:
Tìm được đường đi, chưa tối ưu.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

Map 5:



BFS cost 43

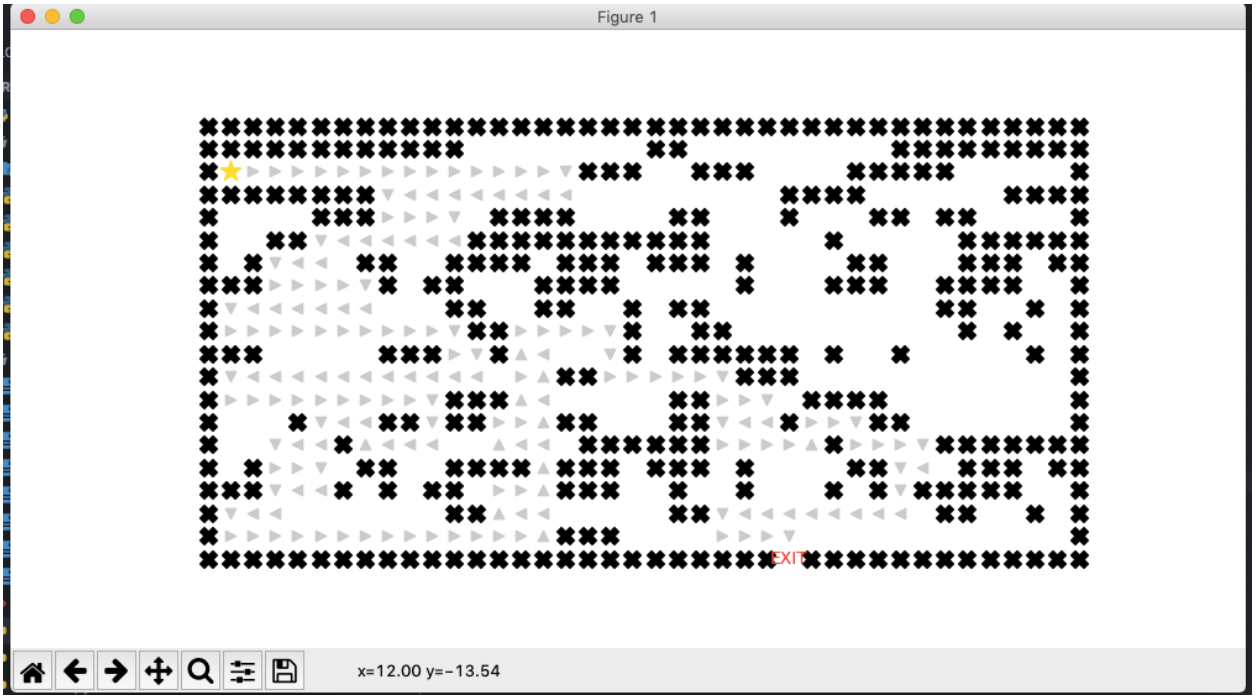
Độ hoàn thiện: tìm được đường đi

Tính tối ưu: tìm được đường đi ngắn nhất

Độ phức tạp về mặt thời gian: $O(b^m)$

Độ phức tạp về mặt không gian: $O(b^m)$

Nhận xét: độ phức tạp không gian, thời gian vẫn chưa tốt.



DFS cost 187

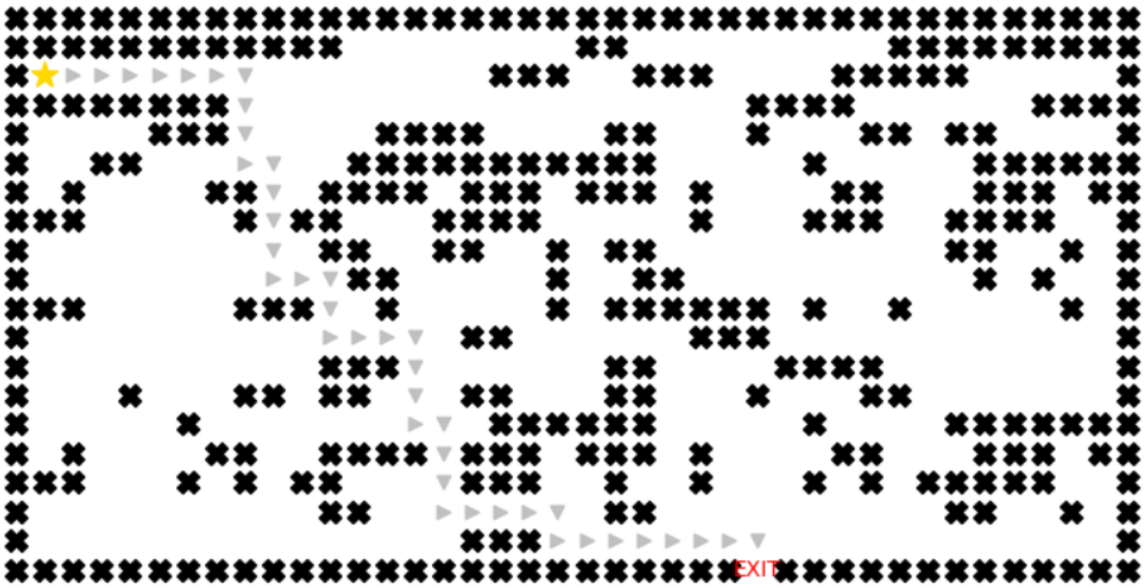
Độ hoàn thiện: tìm được đường đi

Tính tối ưu: không tìm được đường đi ngắn nhất

Độ phức tạp về mặt thời gian: $O(b^m)$

Độ phức tạp về mặt không gian: O(bm)

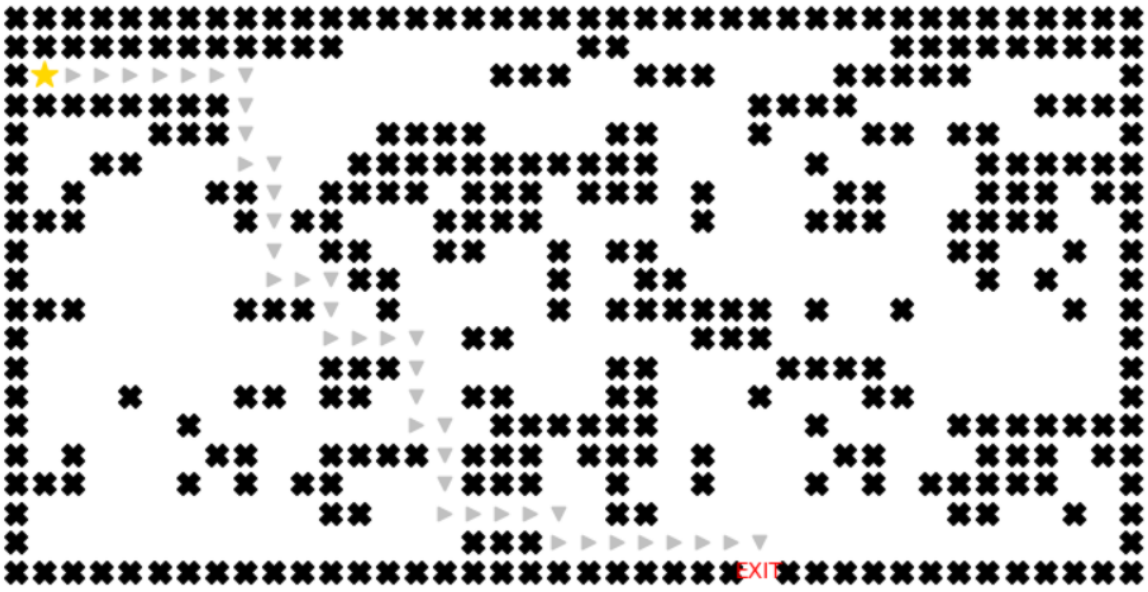
Nhận xét: độ phức tạp thời gian vẫn chưa tốt, đường đi tệ (dài).



A* Search cost 43 (Manhattan)

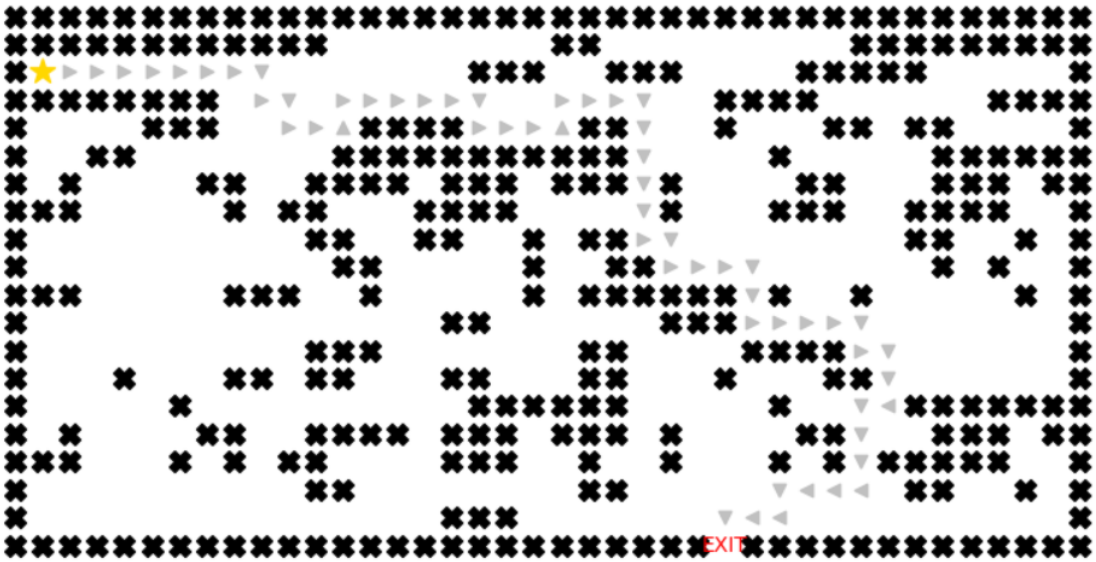
Nhận xét:

Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.



A* Search cost 43 (Euclidean)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

Greedy Search cost 43 (Manhattan)
Nhận xét:
Tìm được đường đi, tối ưu nhất.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

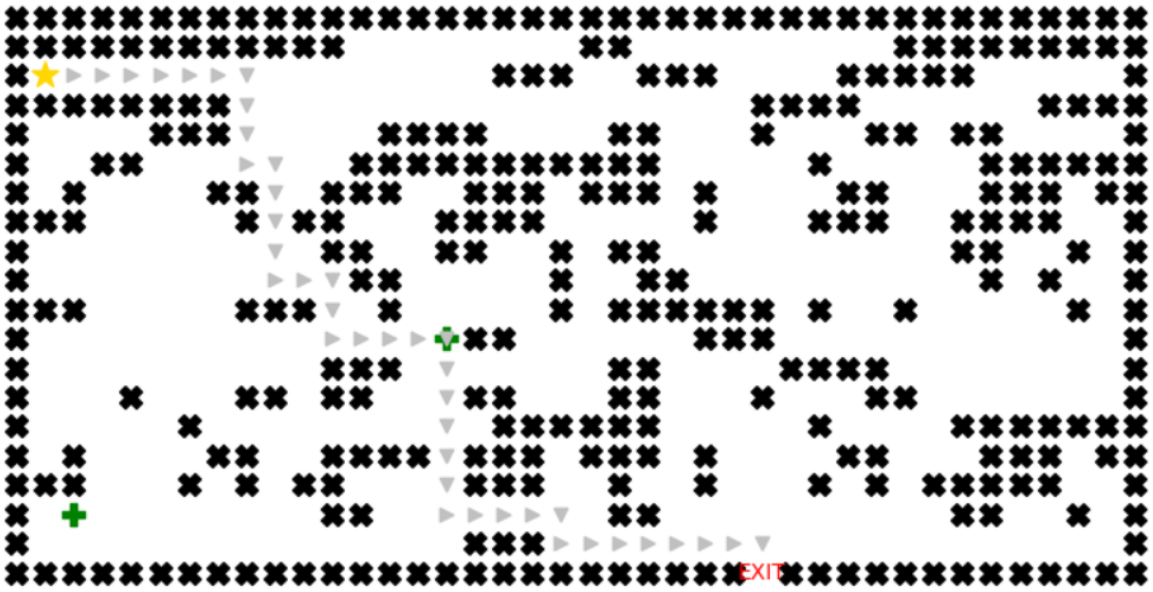


Greedy Search cost 59 (Euclidean)
Nhận xét:
Tìm được đường đi, chưa tối ưu.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

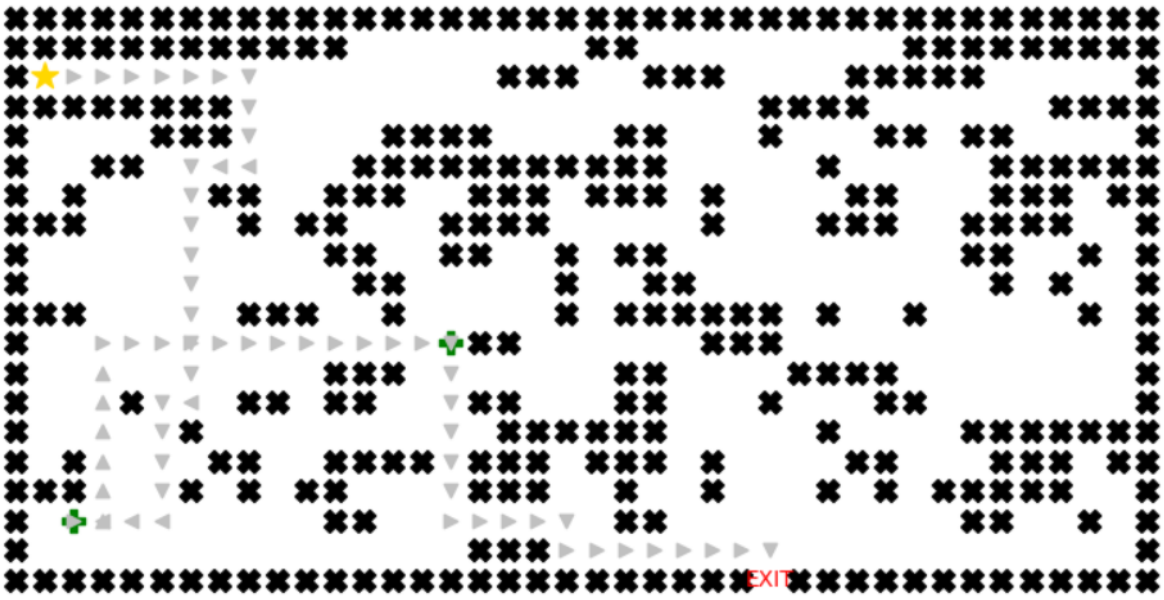
Bản đồ có điểm thưởng

Map 6 (chứa 2 điểm cộng):

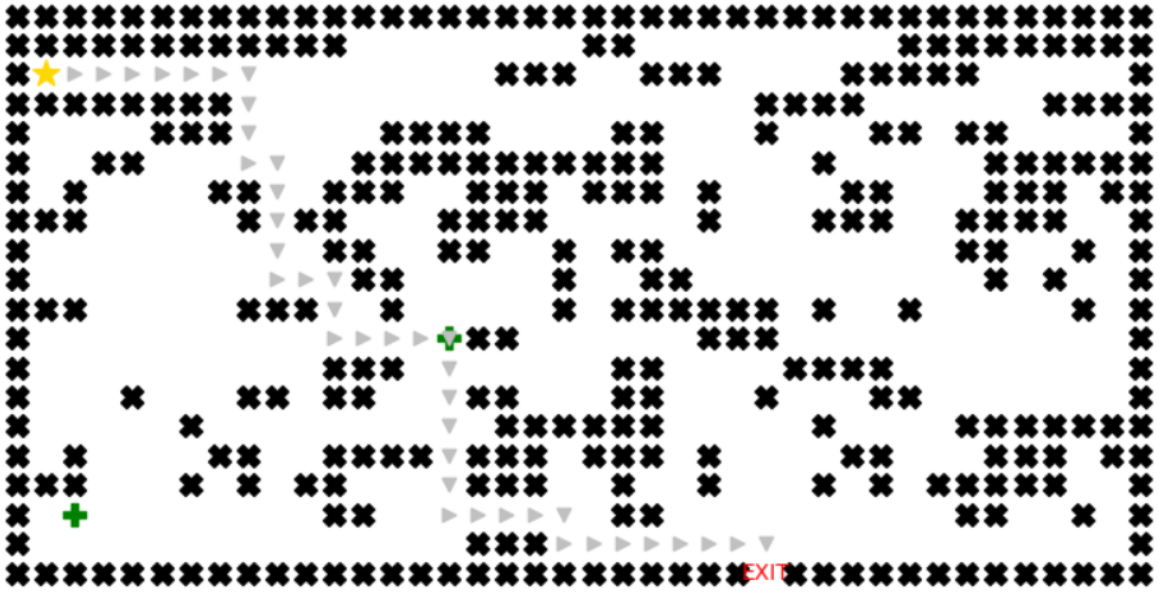
A* Search cost 52 (Manhattan – Quét cạn)
Lộ trình phân tách thành các lộ trình con: Start → (17, 2) → (11, 15) → End
Nhận xét:
Tìm được đường đi, khá tối ưu.
Độ phức tạp không gian, thời gian vẫn chưa tốt.



A* Search cost 33 (Manhattan – Suy xét)
Lộ trình phân tách thành các lộ trình con: Start → (11, 15) → End
Nhận xét:
Tìm được đường đi, tối ưu.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

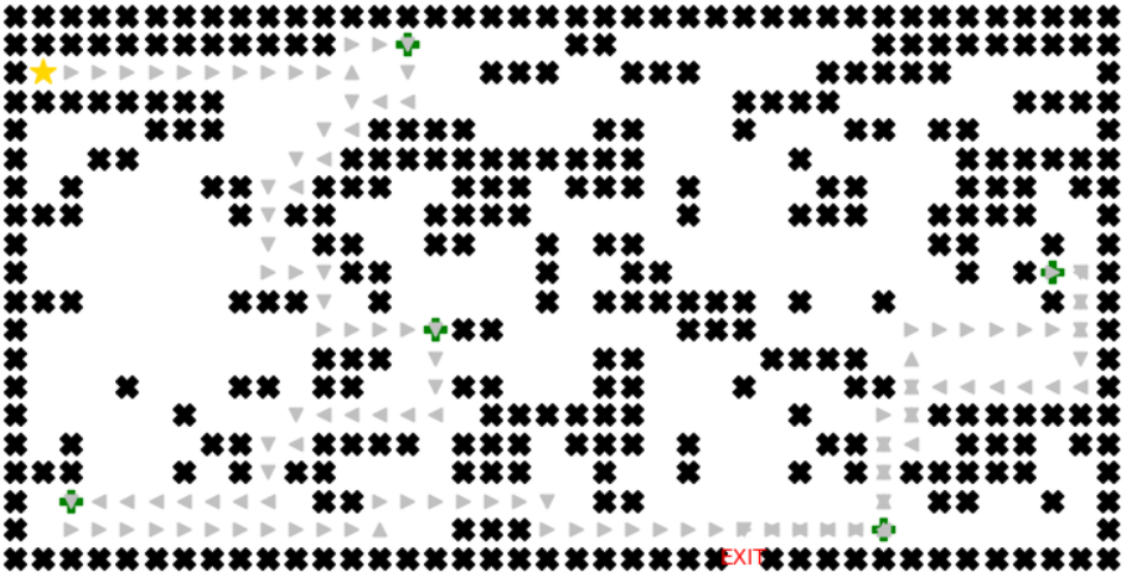


A* Search cost 52 (Euclidean – Quét cạn)
Lộ trình phân tách thành các lộ trình con: Start → (17, 2) → (11, 15) → End
Nhận xét:
Tìm được đường đi (dài), khá tối ưu.
Độ phức tạp không gian, thời gian vẫn chưa tốt.



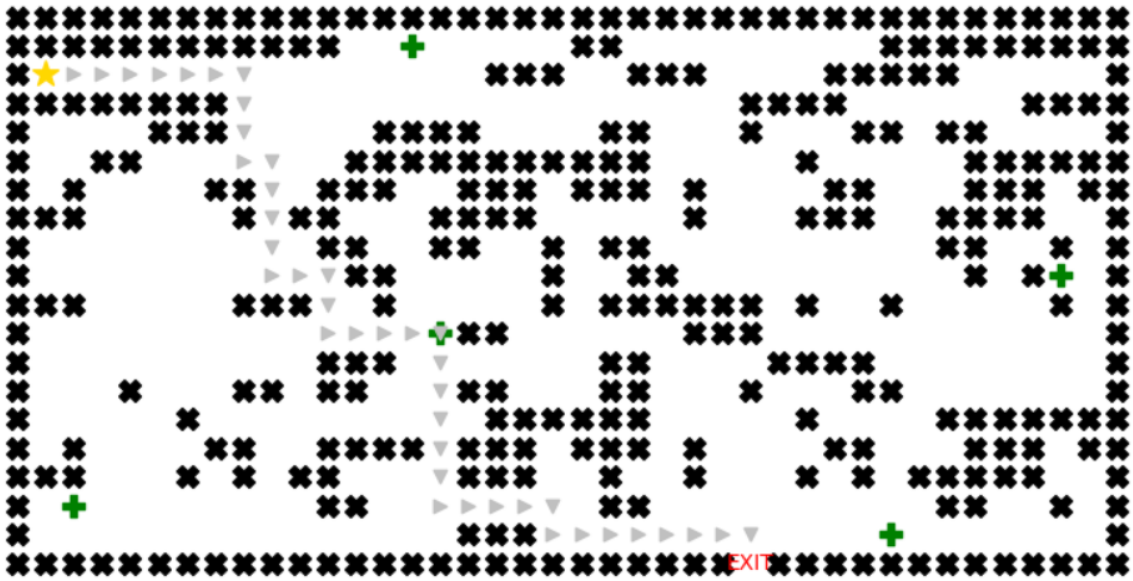
A* Search cost 33 (Euclidean – Suy xét)
Lộ trình phân tách thành các lộ trình con: Start → (11, 15) → End
Nhận xét:
Tìm được đường đi, tối ưu.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

Map 7 (chứa 5 điểm cộng):

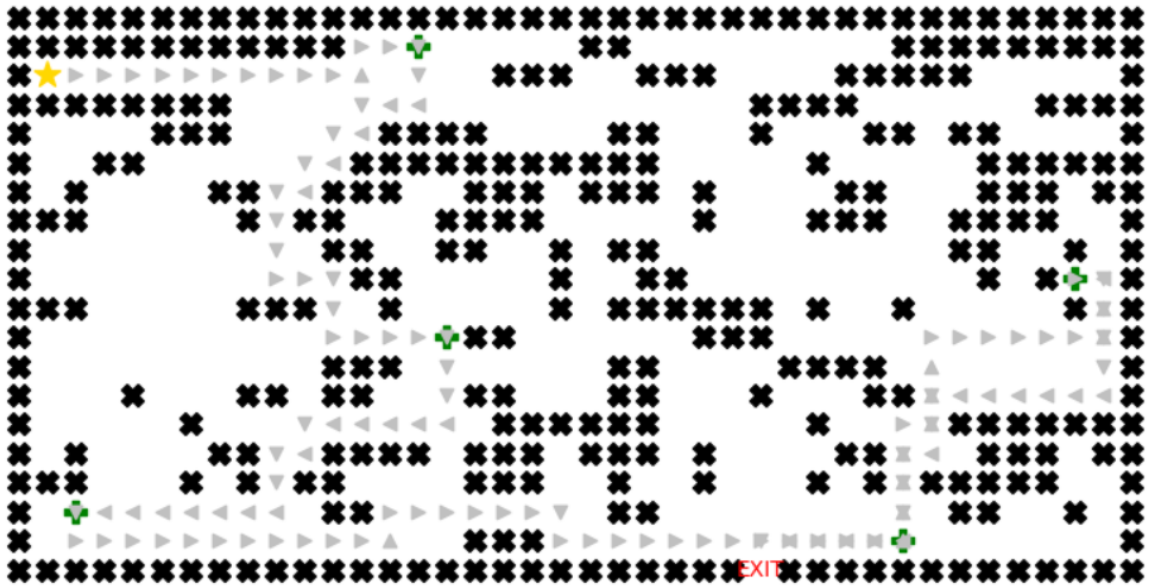


A* Search cost 97 (Manhattan – Quét cạn)

Lộ trình phân tách thành các lộ trình con: Start → (1, 14) → (11, 15) → (17, 2) → (18, 31) → (9, 37) → End
Nhận xét:
Tìm được đường đi (dài), không hiệu quả.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

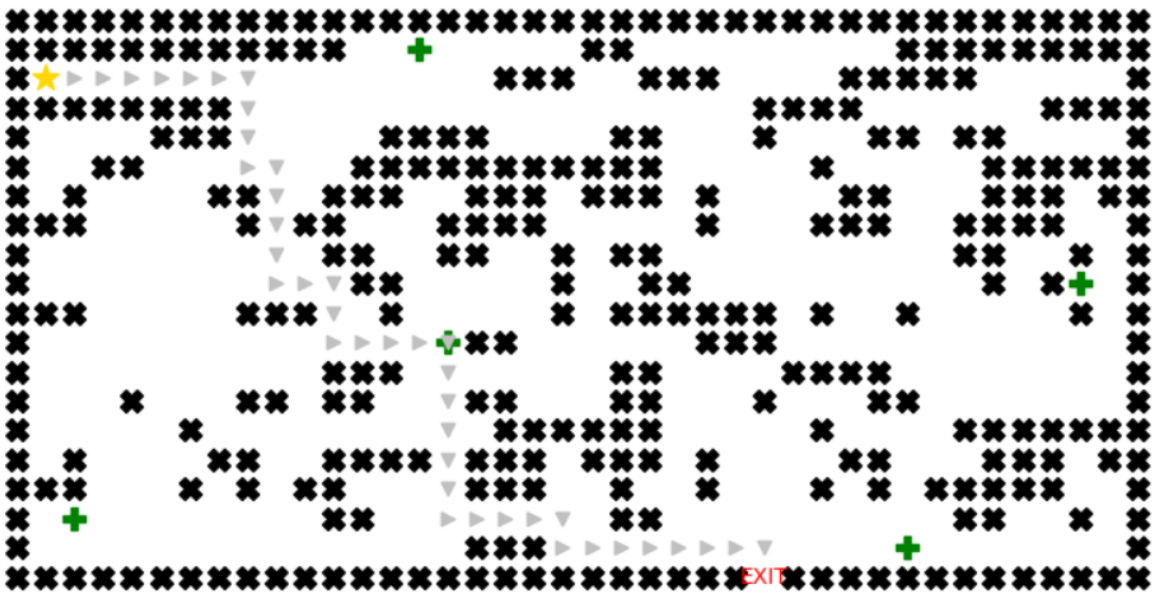


A* Search cost 33 (Manhattan – Suy xét)
Lộ trình phân tách thành các lộ trình con: Start → (11, 15) → End
Nhận xét:
Tìm được đường đi, tối ưu.
Độ phức tạp không gian, thời gian vẫn chưa tốt.



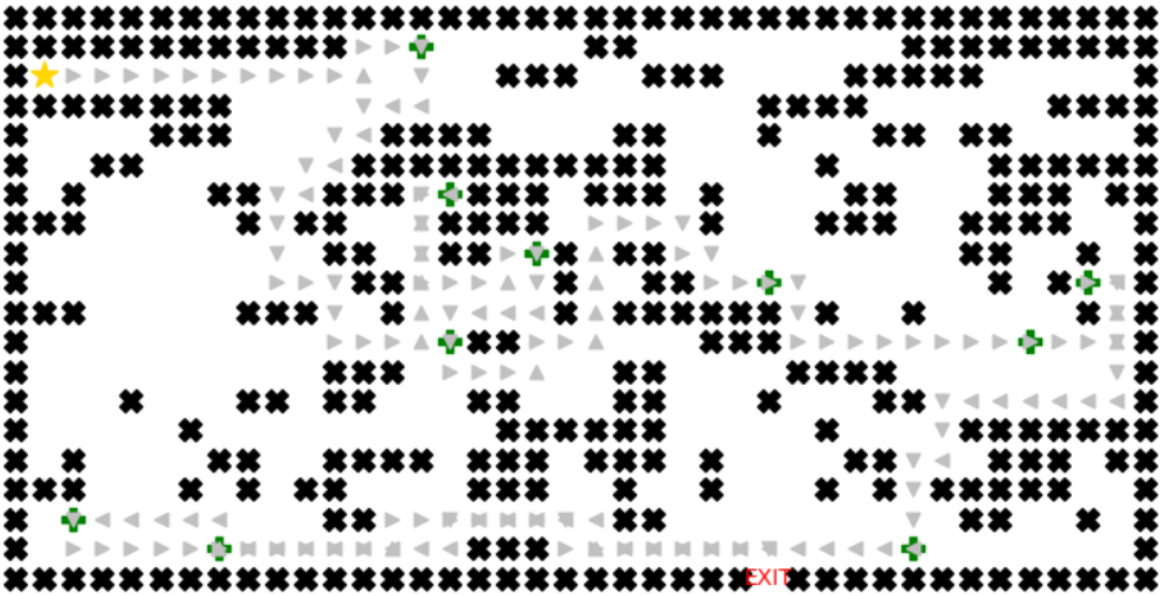
A* Search cost 97 (Euclidean – Quét cạn)

Lộ trình phân tách thành các lộ trình con: Start → (1, 14) → (11, 15) → (17, 2) → (18, 31) → (9, 37) → End
Nhận xét:
Tìm được đường đi (dài), không hiểu quả.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

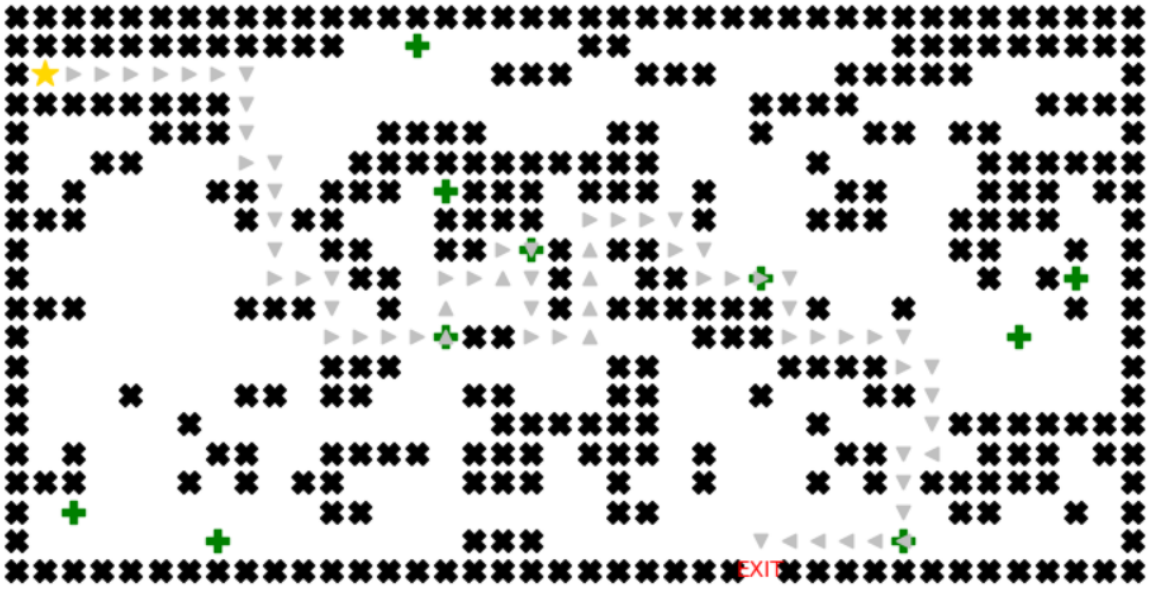


A* Search cost 33 (Euclidean – Suy xét)
Lộ trình phân tách thành các lộ trình con: Start → (11, 15) → End
Nhận xét:
Tìm được đường đi, tối ưu.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

Map 8 (chứa 10 điểm cộng):

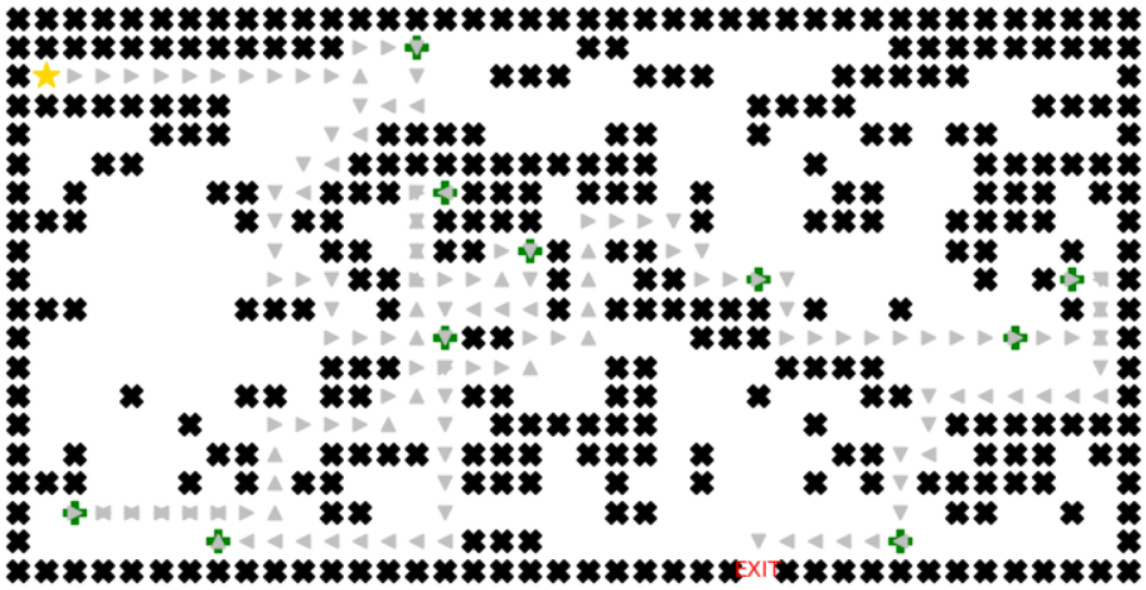


A* Search cost 112 (Manhattan – Quét cạn)
Lộ trình phân tách thành các lộ trình con: Start → (1, 14) → (6, 15) → (8, 18) → (11, 15) → (9, 26) → (11, 35) → (9, 37) → (18, 31) → (18, 7) → (17, 2) → End
Nhận xét:
Tìm được đường đi (dài), không hiệu quả.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

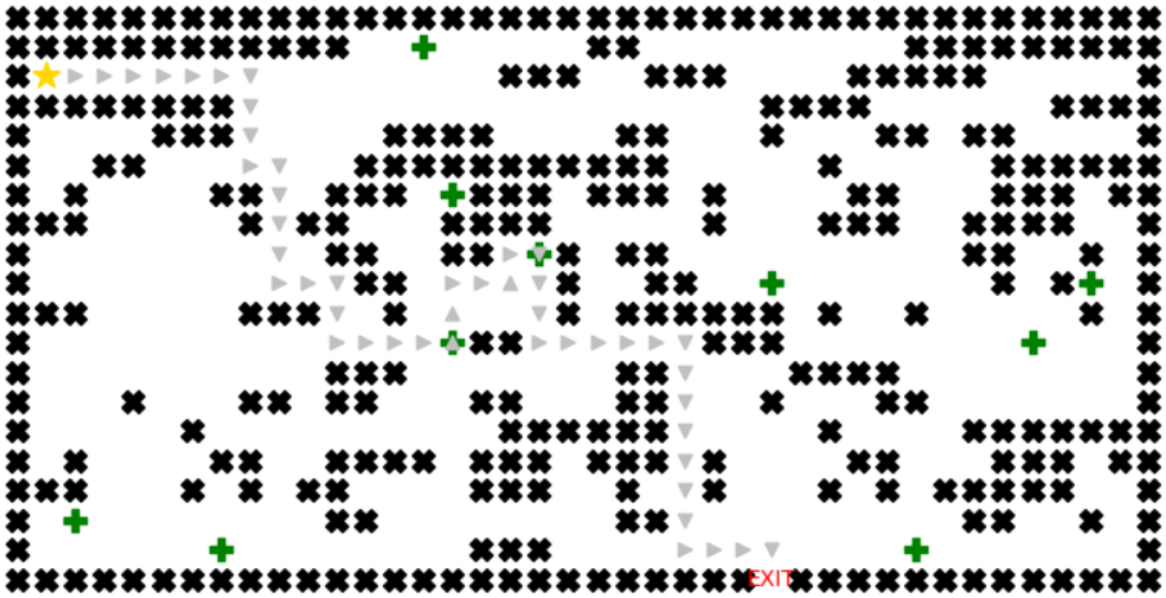


A* Search cost 59 (Manhattan – Suy xét)
Lộ trình phân tách thành các lộ trình con: Start → (11, 15) → (8, 18) → (9, 26) → (18, 31) → End

Nhận xét:
Tìm được đường đi, khá tối ưu.
Độ phức tạp không gian, thời gian vẫn chưa tốt.



A* Search cost 96 (Euclidean – Quét cạn)
Lộ trình phân tách thành các lộ trình con: Start → (1, 14) → (6, 15) → (8, 18) → (11, 15) → (18, 31) → (17, 2) → (9, 26) → (11, 35) → (9, 37) → (18, 31) → End
Nhận xét:
Tìm được đường đi (dài), không hiệu quả.
Độ phức tạp không gian, thời gian vẫn chưa tốt.



A* Search cost 39 (Euclidean – Suy xét)
Lộ trình phân tách thành các lộ trình con: Start → (11, 15) → (8, 18) → End
Nhận xét:
Tìm được đường đi, tối ưu.
Độ phức tạp không gian, thời gian vẫn chưa tốt.

So sánh

Giữa các thuật toán tìm kiếm trong đồ thị không điểm cộng

Tìm kiếm mù

DFS: không tối ưu bằng các thuật toán còn lại do cơ chế tìm kiếm theo chiều sâu, ưu tiên duyệt các node xa cho đến khi gặp node đích.
BFS: áp dụng cơ chế duyệt theo chiều rộng, trong đồ thị không trọng số như mê cung, có thể đưa ra lộ trình tối ưu nhất.

Tìm kiếm có thông tin

Các thuật toán tìm kiếm có thông tin xác định lối đi dựa trên việc xem xét thuộc tính f, giá trị thuộc tính f phụ thuộc vào yếu tố h (áp dụng trong A* và Greedy) đại diện cho các Heuristic và g (áp dụng trong A*).
Heuristic Manhattan: Ưu tiên chọn node gần hơn so với lộ trình tạo bởi 2 cạnh góc vuông từ node đến điểm đến.
Heuristic Euclidean: Ưu tiên chọn node gần hơn so với khoảng cách giữa node với điểm đến.
Đường đi trả về có tối ưu hay không dựa vào độ thoáng của lộ trình được đặt ra trước đó tùy vào loại Heuristic.

Greedy Search: được xem như bản mở rộng của BFS (hoặc gần hơn: một DFS chọn lọc). Tuy duyệt theo chiều rộng, Greedy khác với BFS là dùng queue ưu tiên thay cho queue tức các node trong đó sẽ được sắp xếp lại theo giá trị Heuristic, do tham lam duyệt các node có chi phí ước tính tốt nên Greedy hoạt động như một DFS chọn lọc nhánh. Thuật toán khá tối ưu đối với các đồ thị thoáng lối cho Heuristic.
A* Search: thuật toán tối ưu nhất (ngang với lại BFS) khi thao tác trên các đồ thị không trọng số. Thuật toán không chỉ áp dụng Heuristic mà còn liên tục cập nhật chi phí đã đi qua khiến thuật toán trở nên vẹn toàn. Bởi vì tính xét trọn vẹn của A* nên khi thao tác với các đồ thị có kích thước lớn hơn sẽ gặp hạn chế trong việc lưu trữ bộ nhớ.

Giữa các chiến lược trong đồ thị có điểm cộng

Chiến lược quét cạn: chiến lược tạo lập lộ trình để đi qua hết các điểm cộng, chính vì vậy chiến lược sẽ thường không hiệu quả bởi vì tính phụ thuộc vào giá trị của điểm cộng hơn là đồ thị.
Chiến lược suy xét: chiến lược này nhỉnh hơn so với quét cạn vì xem xét các chi phí ước tính, không phụ thuộc nhiều vào điểm cộng, phụ thuộc vào độ thoáng lối của lộ trình tạo lập bởi Heuristic.

Tham khảo

Depth-first search - Wikipedia
Breadth-first search - Wikipedia
Depth First Search or DFS for a Graph - GeeksforGeeks
Breadth First Search or BFS for a Graph - GeeksforGeeks
Best First Search (Informed Search) - GeeksforGeeks
A* Search Algorithm - GeeksforGeeks
Google Colab