TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN TP.HCM

KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN MÔN HỌC: CƠ SỞ TRÍ TUỆ NHÂN TẠO

Giảng viên: Lê Hoài Bắc

THUẬT TOÁN TÌM KIẾM

Các thành viên trong nhóm:

Tên	MSSV
Dương Thanh Hiệp	19120505
Nguyễn Bá Ngọc	19120603

BÁO CÁO

Sau đây là báo cáo về đồ án Các thuật toán tìm kiếm đường đi do các thành viên trong nhóm thực hiện.

Đồ án này được chia ra thành 3 phần trong đó 2 phần chính và 1 phần là nâng cao:

1. Tìm kiếm đường đi trên bản đồ không có điểm thưởng:
    - Thuật toán tìm kiếm không có thông tin:
        - BFS (Breadth First Search).
        - DFS (Depth First Search).
    - Thuật toán tìm kiếm có thông tin:
        - Thuật toán tìm kiếm tham lam (Greedy Best First Search).
        - Thuật toán tìm kiếm A*.
2. Tìm kiếm đường đi trên bản đồ có điểm thưởng.
3. Tìm kiếm đường đi tối ưu nhất cho bản đồ đã được nâng cấp(có cánh cửa dịch chuyển, chướng ngại vật trên bản đồ có thể di chuyển...).

Đây là bảng tỉ lệ phần trăm hoàn thành và bảng đánh giá mức độ hoàn thành trên từng yêu cầu và toàn bộ đồ án(không tính phần điểm cộng):

Phần	Yêu cầu	Đã hoàn thành	Tỉ lệ hoàn thành
1	Breadth First Search	Rồi	100%
	Depth First Search	Rồi	100%
	Greedy Best First Search	Rồi	100%
	Thuật toán tìm kiếm A*	Rồi	100%
2	Tìm đường đi trên bản đồ có điểm thưởng	Rồi	100%
3	Nâng cấp bản đồ và tìm đường đi tối ưu	Chưa	0%
Đồ án			100%

MỤC LỤC

I. Demo Code:

```python
def init_variable(matrix):
    consider = []
    previouspoint = []
    queue = []
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] == 'x':
                consider.append(0)
            else:
                consider.append(1)
            previouspoint.append((0,0))
            queue.append(0)
    return consider, previouspoint, queue
```

- Đây là hàm khởi tạo các mảng cần dùng trong tất cả các thuật toán tìm kiếm tụi em trình bày bên dưới.
  - Mảng consider là mảng có tác dụng kiểm tra xem điểm x đã xét hay chưa. Nếu như điểm x đã xét thì consider[x] == 0 và ngược lại thì consider[x] == 1.
  - Mảng previouspoint là mảng có tác dụng lưu điểm ngay trước điểm x (có nghĩa là phải đi qua điểm đó thì mới đến được điểm x). Vì đây là hàm khởi tạo nên mảng này chưa có giá trị gì, ta lưu tất cả phần tử trong mảng là (0, 0).
  - Mảng queue là mảng có tác dụng như một hàng đợi, lưu các điểm kế tiếp mà ta phải xét theo thứ tự.

```python
def find_index(i, matrix):
    return len(matrix[0])*i[0] + i[1]
```

- Đây là hàm giúp ta trả về giá trị số nguyên n (n thuộc N) của 1 điểm (giống như là vị trí của điểm đó trên bản đồ, tính bằng cách đánh số thứ tự từ trái qua phải và từ trên xuống dưới tăng dần) đang có giá trị tọa độ là (x, y).

```
def find_coordinate(i, matrix):
    return (i//len(matrix[0]),i%len(matrix[0]))
```

- Hàm này ngược lại với hàm find_index, nó giúp tìm ra địa chỉ của tọa độ dạng (x, y) từ 1 số nguyên n trên bản đồ.

```
def calculate_distance(i, j):
    return math.sqrt(pow((i[0] - j[0]),2) + pow((i[1] - j[1]),2))

def calculate_cost(u , start, previouspoint, matrix):
    count = 1
    i = find_index(u, matrix)
    while previouspoint[i] != start:
        count = count + 1
        i = find_index(previouspoint[i], matrix)
    return count
```

- Đây là 2 hàm được dùng riêng cho các thuật toán tìm kiếm có thông tin:
  - Hàm calculate_distance có tác dụng tính khoảng cách lí thuyết giữa 2 điểm mang giá trị tọa độ (x1, y1) và (x2, y2) theo công thức tọa độ.
  - Hàm calculate_cost có tác dụng tính khoảng cách thực giữa điểm ta đang xét đến điểm start. Ở đây tụi em quy ước đi 1 bước thì có độ dài là 1.

```
def adjacencylist(i, matrix):
    s = find_index(i, matrix)
    m = len(matrix[0])
    list = []
    for i in (s-m,s-1,s+1,s+m):
        if i >= 0 and i < m*len(matrix) and matrix[i//m][i%m] != 'x':
            list.append(i)
    return list
```

- Hàm giúp chúng ta xác định được các đỉnh kề với điểm mà chúng ta đang xét sau đó kết quả trả về là 1 mảng các số nguyên là vị trí các điểm đó trên bảng đồ.

1. Breadth First Search (BFS):

```
def BFS_Search(matrix,start, end):
    F = 0
    L = 0
    consider,previouspoint,queue = init_variable(matrix)
    queue[F] = start
    consider[find_index(start, matrix)]=0;
    previouspoint[find_index(start, matrix)] = start
    while F <= L and consider[find_index(end,matrix)] != 0 :
        u=queue[F]
        F = F + 1
        for p in adjacencylist(u, matrix):
            if consider[p] == 1:
                L=L+1
                queue[L]= find_coordinate(p, matrix)
                consider[p]= 0
                previouspoint[p]= u
    return previouspoint, consider
```

- Thuật toán BFS là thuật toán tìm kiếm đường đi theo chiều rộng. Thuật toán này giúp chúng ta duyệt các tọa độ có thể đi từ điểm start vào một danh sách. Dựa vào các điểm đó ta sẽ tìm được đường đi từ điểm start đến điểm end (nếu điểm end thuộc danh sách), nếu không thì không có đường đi nào từ điểm start đến điểm end.
- Đầu tiên ta khởi tạo các giá trị đầu cho các mảng consider, previouspoint, queue.
- Ta lưu giá trị đầu tiên cũng như là giá trị cần xét vào queue[0]. Sau đó ta khởi tạo vòng lặp xét các điểm là điểm kề của điểm ta đang xét (gọi là x). Nếu consider của điểm đó (gọi là y) == 1 nghĩa là điểm đó chưa được xét đến, ta tăng giá trị của L và lưu điểm đó vào queue (hàng đợi) sau đó set giá trị consider của điểm đó về 0 và previous[y] = x.
- Kết quả cuối cùng của hàm ta trả về mảng previous và mảng consider.

2. Depth First Search (DFS):

```
def DFS_Search(matrix, previouspoint, consider, start, end):
    consider[find_index(start,matrix)] = 0
    if consider[find_index(end,matrix)] == 0:
        return
    for u in adjacencylist(start, matrix):
        if consider[u]== 1:
            previouspoint[u] = start
            DFS_Search(matrix,previouspoint,consider, find_coordinate(u, matrix), end)
```

- Thuật toán DFS là thuật toán tìm kiếm đường đi theo chiều sâu. Thuật toán này giúp chúng ta duyệt các tọa độ có thể đi từ điểm start vào một danh sách. Dựa vào các điểm đó ta sẽ tìm được đường đi từ điểm start đến điểm end (nếu điểm end thuộc danh sách), nếu không thì không có đường đi nào từ điểm start đến điểm end.
- Ta xây dựng 1 hàm đệ quy với điểm bắt đầu là điểm start (set consider của start về giá trị 0), điều kiện kết thúc là khi hàm đã xét đến điểm end (cổng ra của bản đồ).
- Ta khởi tạo vòng lặp dựa trên các điểm trong danh sách kề của điểm ta đang xét (gọi là y). Nếu điểm y có giá trị consider là 1, previous[y] = x (x là điểm start đang xét), sau đó t gọi lại hàm DFS_Search và thay điểm start bằng điểm y.

3. Greedy Best First Search (GBFS):

```
def sort_adjacencylistbyGBFS(queue, previouspoint, first, last):
    min = first
    for i in range(first + 1, last + 1):
        if calculate_distance(find_coordinate(queue[i], matrix), end) < calculate_distance(find_coordinate(queue[min], matrix), end):
            min = i
    queue[first], queue[min] = queue[min],queue[first]
```

- Hàm này dùng để sắp xếp các phần tử trong queue từ phần tử thứ first đến phần thử thứ last theo thứ tự tăng dần.

```
def GBFS_Search(matrix,start,end):
    first = 0
    last = 0
    consider,previouspoint,queue = init_variable(matrix)
    queue[first] = find_index(start, matrix)
    consider[find_index(start, matrix)]=0;
    previouspoint[find_index(start, matrix)] = start
    while first <= last and consider[find_index(end,matrix)] != 0  :
        u=queue[first]
        first = first + 1
        list = adjacencylist(find_coordinate(u, matrix), matrix)
        for p in list:
            if consider[p] == 1:
                last=last+1
                queue[last]= p
                consider[p]= 0
                previouspoint[p]= find_coordinate(u, matrix)
        sort_adjacencylistbyGBFS(queue,previouspoint, first, last)
    return previouspoint
```

- Thuật toán GBFS là thuật toán tìm kiếm đường đi với tri thức bổ sung từ việc sử dụng các tri thức cụ thể của bài toán. Nó sẽ sử dụng 1 hàm đánh giá heuristic để đánh giá xem điểm nào là có vẻ gần với điểm end nhất vì thuật toán này chỉ đi đến những điểm có ước lượng tốt nhất.
- Đầu tiên ta cũng khởi tạo các giá trị đầu cho các mảng và gán các giá trị như thuật toán BFS. Sau đó ta khởi tạo vòng lặp để xét các điểm kề với điểm đang xét (gọi là x). Nếu giá trị consider của điểm đó (gọi là y) là 1 thì ta sẽ thêm điểm đó vào queue, tăng giá trị last cho queue, set consider của điểm đó về giá trị 0 và ta gán giá trị previouspoint[y] = x. Cuối cùng ta sắp xếp mảng Queue để đặt lại thứ tự các điểm mà ta ưu tiên xét dựa trên khoảng cách ước tính từ điểm đó đến đích (sắp xếp tăng dần).
- Kết quả của hàm trả về mảng previouspoint.

4. Thuật toán tìm kiếm A*:

```
def sort_adjacencylistbyA(queue, previouspoint, first, last):
    min = first
    for i in range(first + 1, last + 1):
        if (calculate_cost(find_coordinate(queue[i], matrix), start,previouspoint, matrix) + calculate_distance(find_coordinate(queue[i], matrix), end)) < (calculate_cost(find_coordinate(queue[min], matrix), start,previous
            min = i
    queue[first], queue[min] = queue[min], queue[first]
```

- Hàm này có tác dụng sắp xếp mảng queue từ vị trí first đến vị trí last theo thứ tự tăng dần dựa trên đánh giá heuristic của A*.

```
def A_Search(matrix,start, end):
    first = 0
    last = 0
    list = []
    consider,previouspoint,queue = init_variable(matrix)
    queue[first] = find_index(start,matrix)
    consider[find_index(start, matrix)]=0
    previouspoint[find_index(start, matrix)] = start
    while first <= last and consider[find_index(end, matrix)] != 0  :
        u=queue[first]
        first = first + 1
        list = adjacencylist(find_coordinate(u, matrix), matrix)
        for p in list:
            if consider[p] == 1:
                last=last+1
                queue[last]= p
                consider[p]= 0
                previouspoint[p]= find_coordinate(u, matrix)
        sort_adjacencylistbyA(queue,previouspoint, first, last)
    return previouspoint
```

- Thuật toán A* là thuật toán tìm kiếm đường đi sử dụng 1 hàm đánh giá heuristic để tính tổng của 2 khoảng cách là từ điểm đang xét đến đích và từ điểm đó về lại vị trí bắt đầu (khoảng cách đã đi qua). Nhờ vậy nó có thể tìm được đường đi tối ưu hơn các thuật toán khác.
- Vì A* là một biến thể khác của thuật toán Greedy Best First Search nên hàm A_Search tụi em viết như hàm GBFS_Search. Nó chỉ khác duy nhất ở hàm sort_adjacencylistbyA do đánh giá heuristic của thuật toán A* khác với GBFS.

5. Tìm đường đi trên bản đồ có điểm thưởng:

```
def nearest_bonuspoint(u, bonus_points, consider):
    for i in range(0, len(bonus_points)):
        temp = adjacencylist(bonus_points[i], matrix)
        if len(temp) == 1:
            consider[find_index(bonus_points[i], matrix)] = 0
    nearest = 0
    for i in range(1, len(bonus_points)):
        if (calculate_distance(u,bonus_points[i]) <  calculate_distance(u,bonus_points[nearest])) and consider[find_index(bonus_points[i], matrix)] == 1:
            nearest = i
    if consider[find_index(bonus_points[nearest], matrix)] == 1:
        return nearest
    return -1
```

- Hàm này giúp cho ta tìm được điểm thưởng gần với điểm ta đang xét nhất

```
def best_way(u, bonus_points,consider, end):
    nearest = nearest_bonuspoint(u, bonus_points, consider)
    if nearest == -1:
        return end
    else:
        if (bonus_points[nearest][2] + calculate_distance(bonus_points[nearest],u) + calculate_distance(bonus_points[nearest],end) ) > (calculate_distance(u,end)):
            return end
        else:
            x = (bonus_points[nearest][0],bonus_points[nearest][1])
            return x
```

- Hàm này giúp ta đánh giá xem giữa đi tiếp theo thuật toán (gọi là đường x) và đi đến điểm thưởng rồi đi tiếp (gọi là đường y) thì con đường nào sẽ có ít chi phí di chuyển hơn. Nếu x < y thì kết quả của hàm sẽ trả về địa chỉ điểm end, ngược lại thì kết quả của hàm trả về địa chỉ của điểm thưởng.

```
def sort_adjacencylistbyBonus(list, previouspoint, bonus_points, consider, end):
    for i in range(0, len(list) ):
        way1 = best_way(find_coordinate(list[i],matrix),bonus_points,consider, end)
        for j in range(i + 1, len(list)):
            way2 = best_way(find_coordinate(list[j],matrix),bonus_points,consider, end)
            if (calculate_distance(find_coordinate(list[i], matrix), way1)) > calculate_distance(find_coordinate(list[j], matrix), way2):
                list[i], list[j] = list[j], list[i]
```

- Hàm này giúp sắp xếp danh sách kề của điểm đang xét sao cho khi xét hàng đợi thì luôn xét điểm hướng đến vị trí Bonus hoặc điểm end.

```
def Bonus_point_Search(matrix, previouspoint, consider, start, end):
    consider[find_index(start,matrix)] = 0
    if consider[find_index(end,matrix)] == 0:
        return
    list = adjacencylist(start, matrix)
    sort_adjacencylistbyBonus(list, previouspoint, bonus_points, consider, end)
    for u in list:
        if consider[u]== 1:
            previouspoint[u] = start
            Bonus_point_Search(matrix,previouspoint,consider, find_coordinate(u, matrix), end)
```

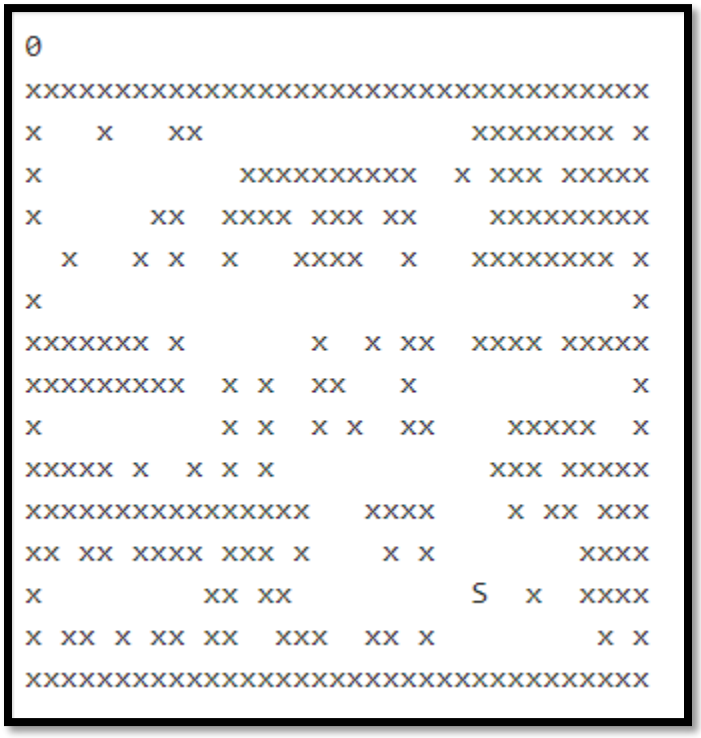
- Ta dùng hàm đệ quy ở thuật toán này, điểm dừng của hàm là khi đã xét đến điểm end. Ta sắp xếp các điểm kề của điểm ta xét (gọi là x) sao cho đúng với hàm đánh giá heuristic. Sau đó ta bắt đầu vòng lặp trên danh sách kề đã sắp xếp, nếu điểm kề đó (gọi là y) có giá trị consider là 1 thì ta set giá trị previouspoint[y] = x rồi gọi là hàm Bonus_point_Search.

II. Kết quả của từng thuật toán khi thực hành trên bản đồ (không có điểm thưởng):

❖ Nói sơ lược về cách tìm đường đi của các thuật toán:

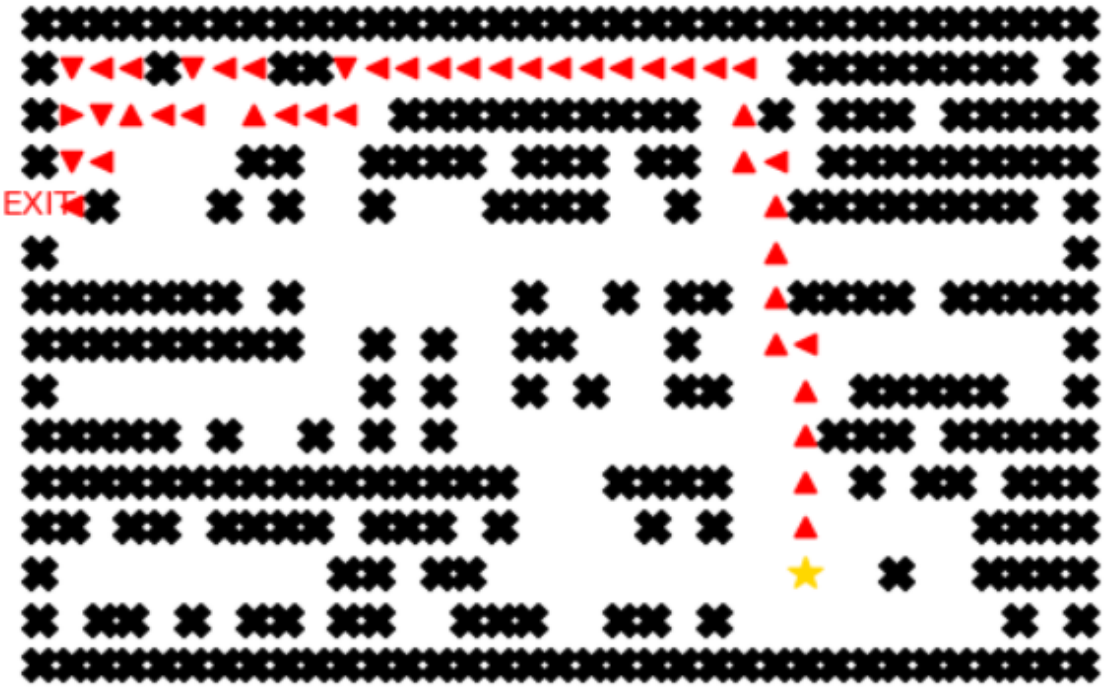
- *Thuật toán DFS:* là thuật toán duyệt theo chiều sâu, theo chiến lược tụi em viết thì thuật toán sẽ xét điểm kề có giá trị nhỏ nhất với điểm đang xét trước. Từ điểm đó, thuật toán này sẽ duyệt đến khi nào không còn đường đi nữa hay là đã duyệt đến đích thì mới chuyển sang điểm tiếp theo hoặc dừng lại.
- *Thuật toán BFS:* là thuật toán duyệt theo chiều rộng. Nó sẽ duyệt hết các điểm kề xung quanh của điểm đang xét theo chiến lược cho tới khi nào gặp được điểm đích hoặc không còn điểm để duyệt nữa thì dừng lại.
- *Thuật toán GBFS:* là thuật toán duyệt đường đi dựa trên đánh giá heuristic (khoảng cách ước tính từ điểm đang xét đến đích). Ta sắp xếp các điểm đấy vào hàng đời queue theo thứ tự tăng dần. Thuật toán sẽ xét đến khi nào gặp đích hoặc không còn điểm để xét nữa.
- *Thuật toán A*:* tương tự như thuật toán GBFS nhưng hàm heuristic của A* sẽ đánh giá tổng của 2 khoảng cách. Một là khoảng cách ước tính từ điểm đang xét đến đích. Hai là khoảng cách điểm đó đã đi qua kể từ điểm bắt đầu.

1. Bản đồ 1:



- *Thuật toán Breadth First Search:* do chiều rộng có tối đa là 4 nút là chi phí mỗi bước đi là như nhau (chi phí các cạnh là tương đương) nên thuật toán BFS ở trường hợp này có thể nói là trường hợp tốt của BFS.

- *Thuật toán Deapth First Search:* theo chiến lược, thuật toán chỉ đơn giản tìm độ sâu của nút kề phía trên (hoặc nút kề có giá trị nhỏ nhất). Cho nên ta thấy được nó luôn đi bám lên phía trên của bản đồ.

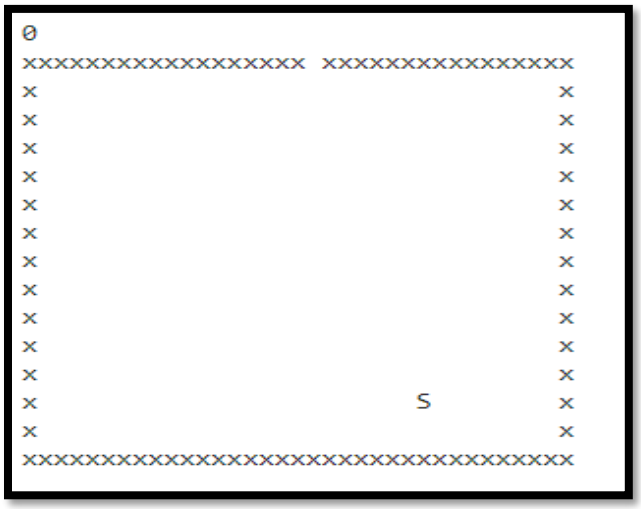


- *Thuật toán Greedy Best First Search:* thuật toán này luôn chọn đường đi có điểm mà nó cho là gần với đích nhất nên không thể đoán được kịch bản đánh lừa của bản đồ. Nhưng ở bản đồ này, thuật toán tìm ra được đường đi tốt nhất.

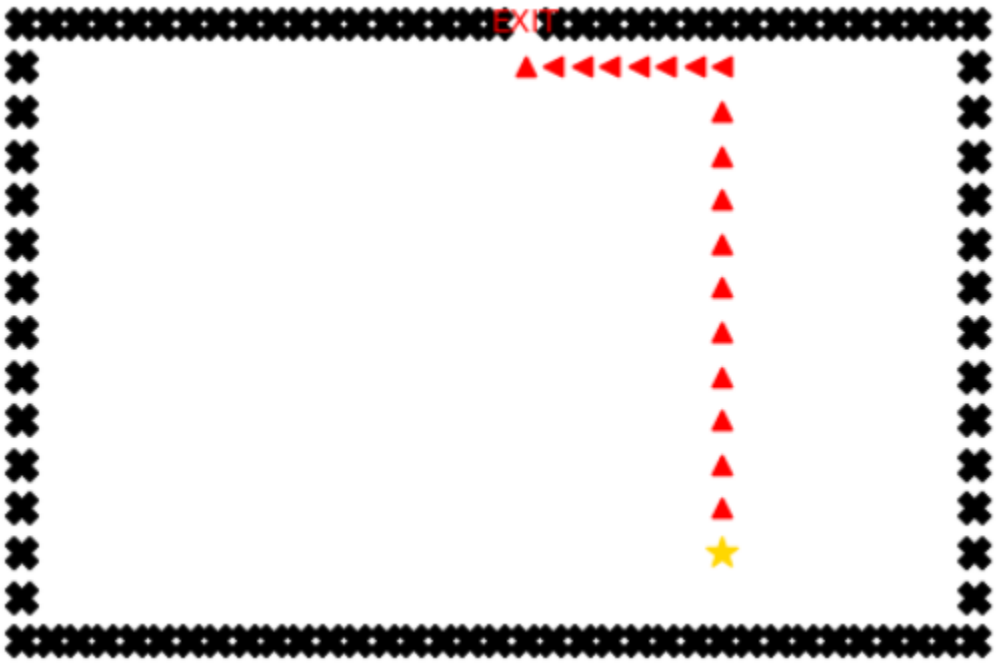
- *Thuật toán A*:* Đây là thuật toán có thể gọi là tốt nhất trong hầu hết trường hợp do nó có sự kết hợp giữ sự hoàn chỉnh và tối ưu của nhiều thuật toán.

➔ Kết luận: Trong bản đồ này các tìm đường đi của DFS là tệ nhất, các thuật toán BFS, A*, GBFS tuy có đường đi khác nhau nhưng chi phí là như nhau. Tuy nhiên GBFS và A* nhỉnh hơn BFS do BFS cần nhiều không gian để lưu trữ các điểm cần duyệt.

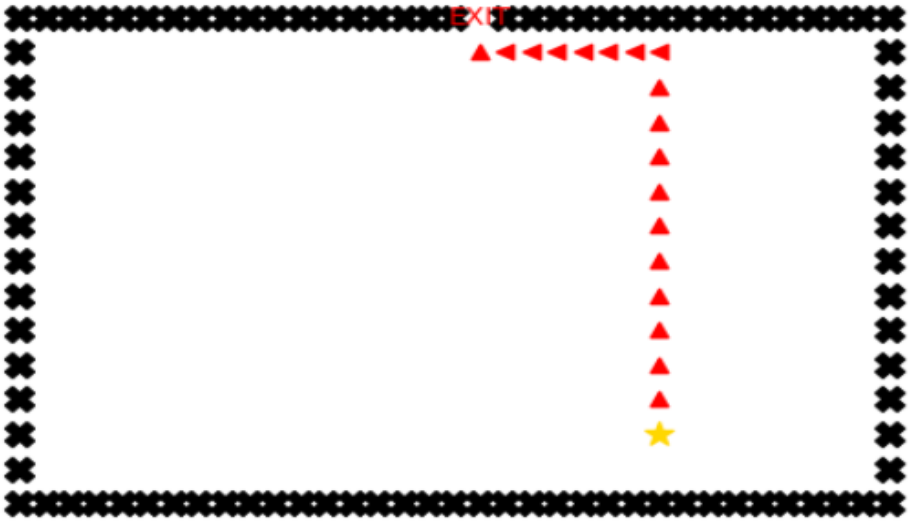
2. Bản đồ 2:



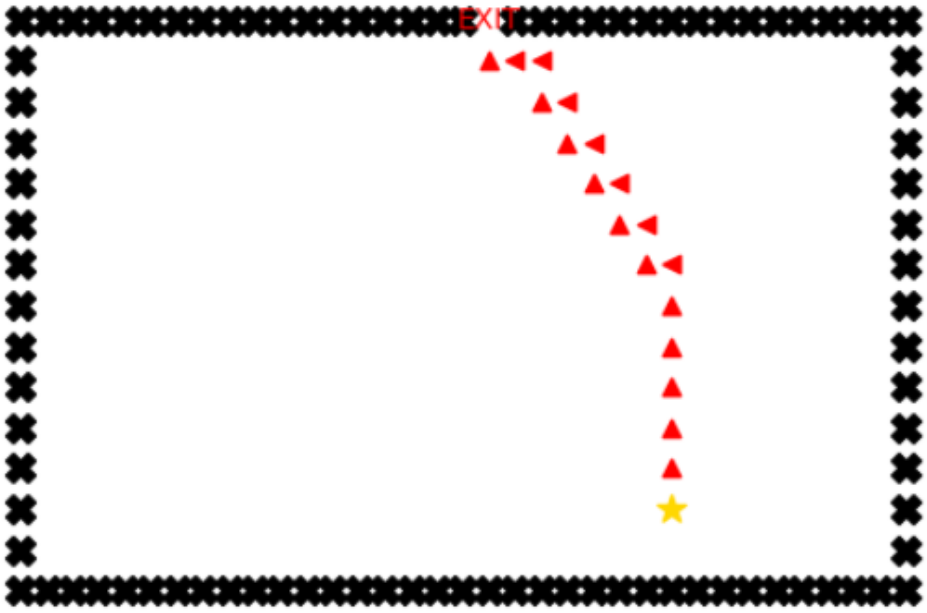
- *Thuật toán Breadth First Search:* chỉ có tối đa 4 đỉnh kề và chi phí các bước đi là như nhau nên cũng là trường hợp tốt của BFS. Nhưng sẽ tốn không gian lưu trữ do lúc nào cũng phải duyệt hết các đỉnh cho đến khi tìm được đích đến. Số điểm thuật toán này phải duyệt là có thể lên đến toàn bản đồ.



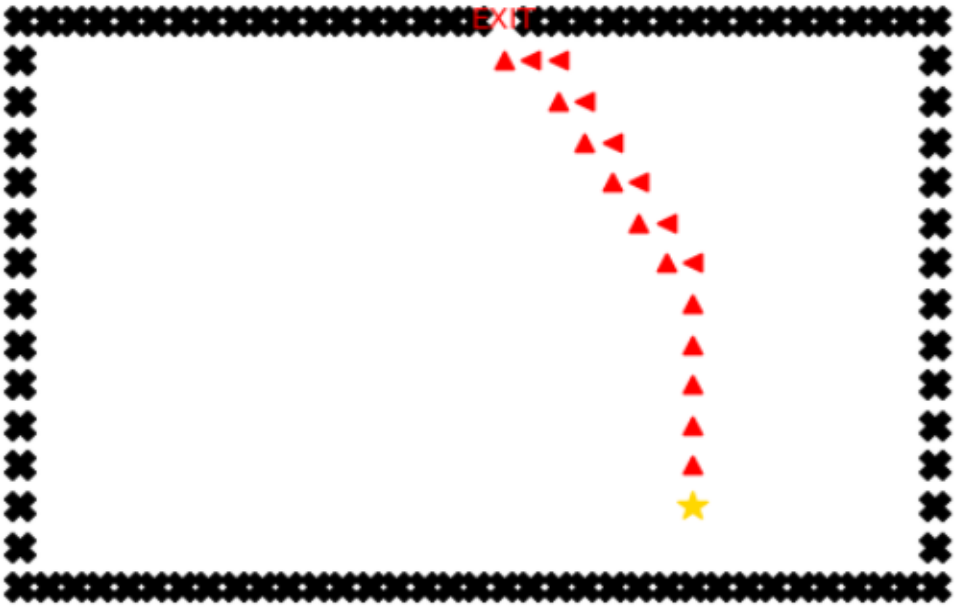
- *Thuật toán Deapth First Search:* về cơ bản thuật toán này tìm giải pháp từ "điểm có giá trị nhỏ nhất"(đỉnh kề nằm ngay phía trên đỉnh đang xét) mà không quan tâm chi phí. Nhưng trong trường hợp này do không có chướng ngại vật và đích đến nằm bên trên nên đây là trường hợp tốt đối với DFS. Trong TH này, nó cũng tìm được đường đi ngắn nhất.



- *Thuật toán Greedy Best First Search:* chọn đỉnh để duyệt kế tiếp dựa trên khoảng cách ước tính từ đỉnh đó đến đích là ngắn nhất. Thuật toán này cho ra đường đi tốt và các điểm được duyệt ít nhất.

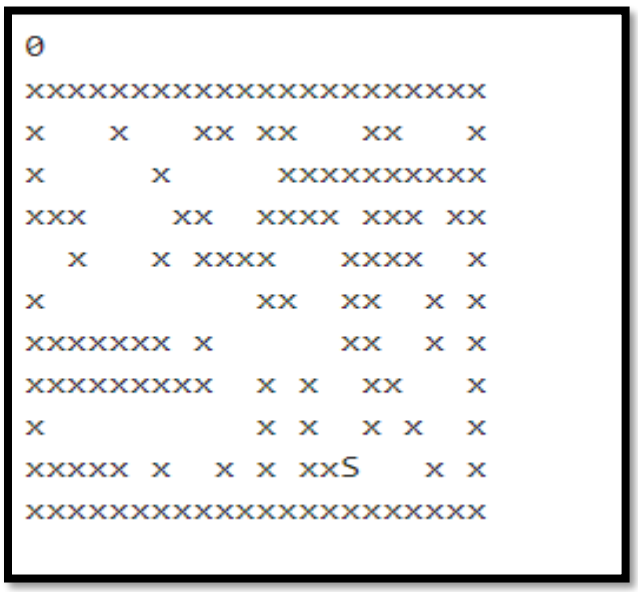


- *Thuật toán A*:* duyệt dựa trên tổng 2 khoảng cách. Một là khoảng cách từ điểm đang xét đến đích. Hai là khoảng cách đã đi qua. A* sẽ duyệt điểm nào có tổng đấy nhỏ nhất và cho ra đường đi tối ưu.

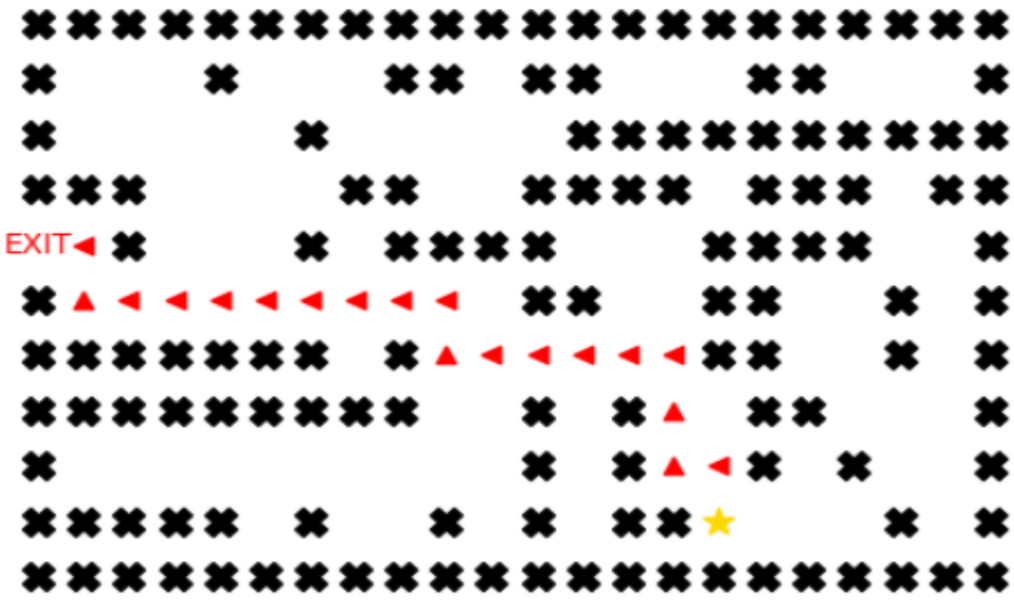


➔ Kết luận: Mặc dù ở cả 4 thuật toán đều cho ta đường đi đến được đích nhưng BFS tốn không gian lưu trữ nhất nên sẽ ít được sử dụng trong trường hợp này. Trong bản đồ này, DFS, GBFS, A* đều cho ra đường đi tối ưu, đỡ tốn bộ nhớ do có ít điểm cần duyệt và đây là trường hợp tốt nhất cho DFS.

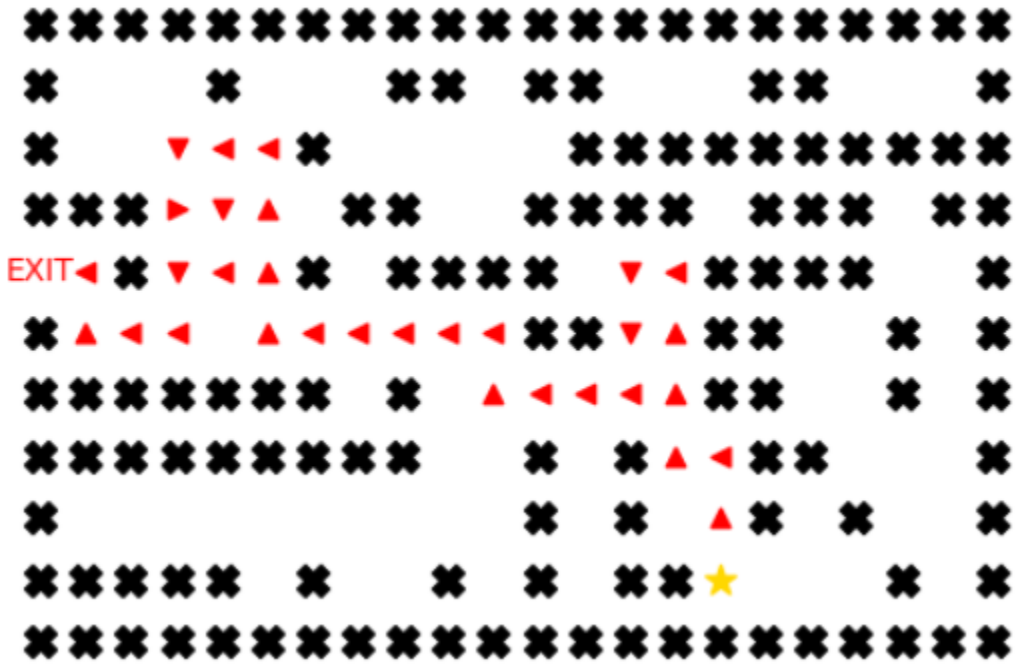
3. Bản đồ 3:



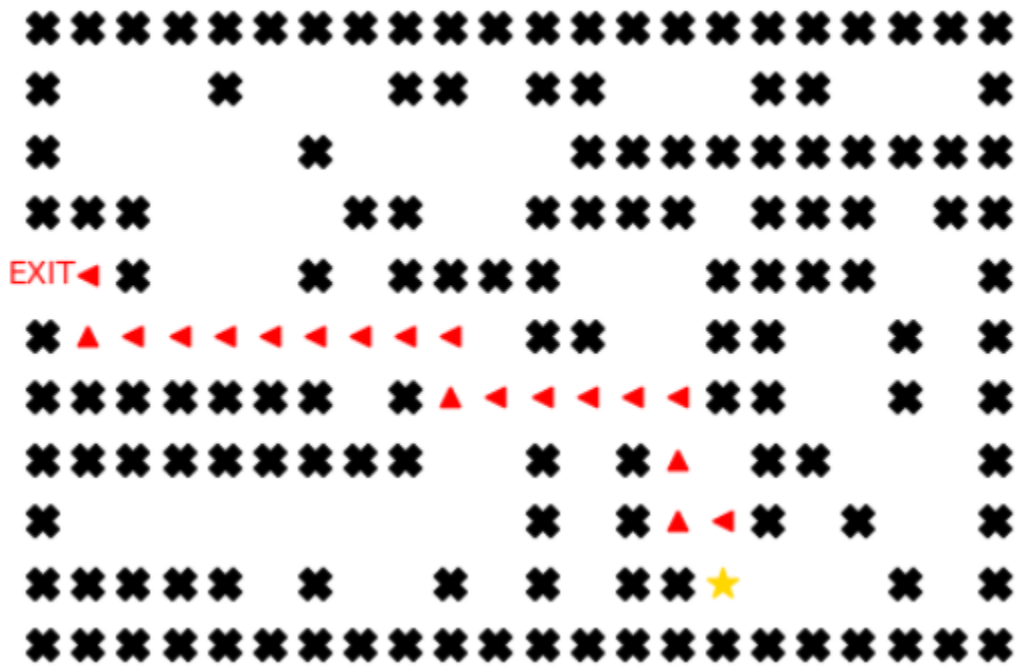
- *Thuật toán Breadth First Search:* cho được đường đi tối ưu nhưng theo cách duyệt của thuật toán thì vẫn gây tốn bộ nhớ.



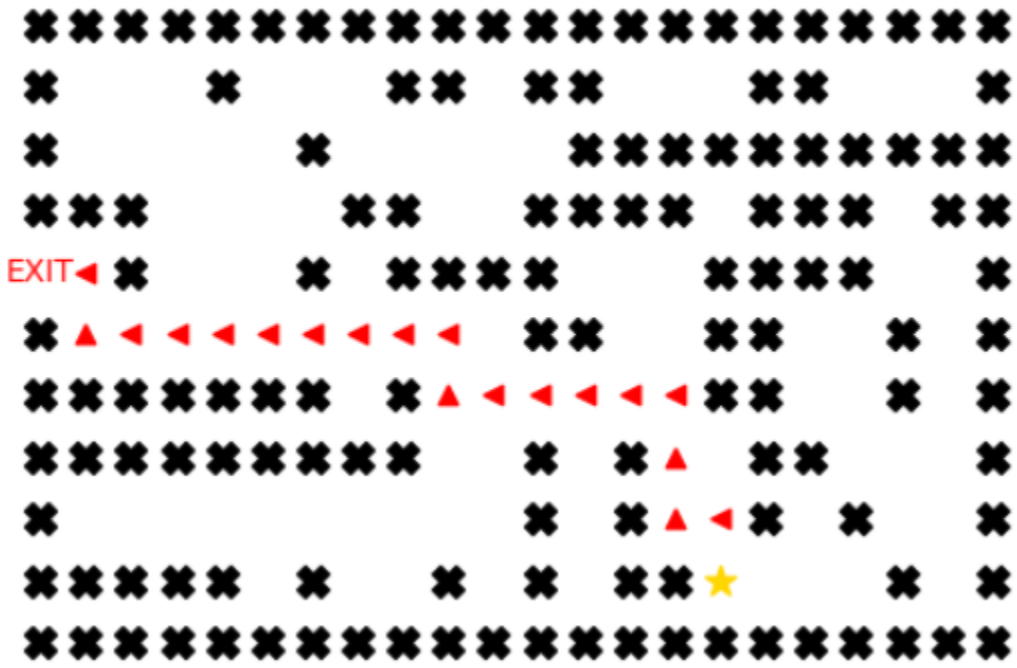
- *Thuật toán Deapth First Search:* do cách duyệt của thuật toán nó vẫn sẽ ưu tiên đi đường đi từ "điểm có giá trị nhỏ nhất" cho tới khi tìm thấy đích. Nhưng nó vẫn xét những điểm không cần thiết vì vậy cho nên chưa tối ưu trong việc xét các điểm cần duyệt và chưa đưa ra được đường đi tối ưu.



- *Thuật toán Greedy Best First Search:* ưu tiên xét các điểm mà nó cho là khoảng cách ước tính từ điểm đó đến đích là gần nhất. Các điểm duyệt không bị dư thừa. Cho ra đường đi tối ưu. Chi phí là 20.

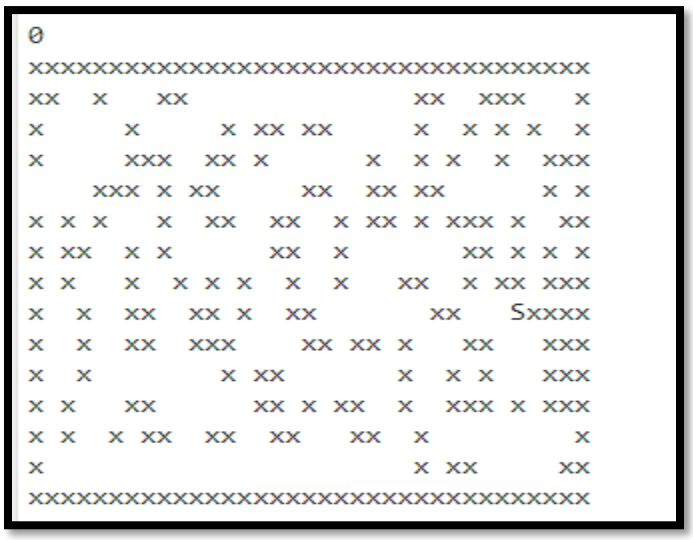


- *Thuật toán A*:* thuật toán này cũng như GBFS, đều cho ra được đường đi tối ưu với chi phí ở bản đồ này là 20.

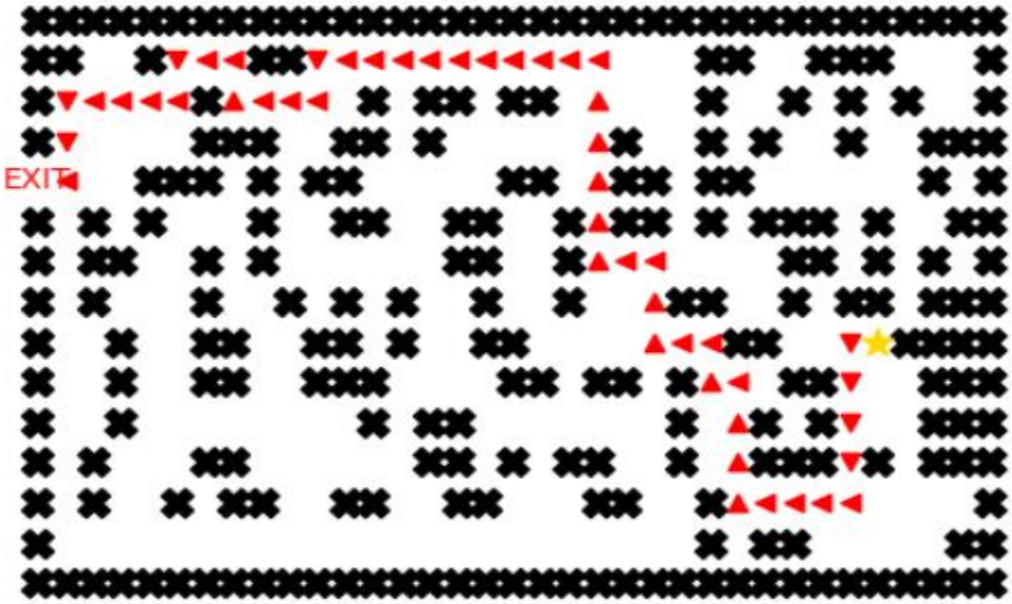


➔ Kết luận: Ở bản đồ này, thuật toán DFS không cho ra được đường đi tối ưu do chiến lược duyệt điểm của nó, thuật toán BFS cho ra được đường đi tối ưu nhưng lại tốn không gian để lưu trữ những điểm đã duyệt (do nó phải duyệt tất cả các điểm kề). Nên thuật toán A* và GBFS là tối ưu đối với trường hợp này, vừa đưa ra được đường đi tối ưu vừa ít tốn không gian lưu trữ do các điểm cần duyệt ít.

4. Bản đồ 4:



- *Thuật toán Breadth First Search:* cho ra được đường đi tối ưu với chi phí đường đi là 50 nhưng các điểm được duyệt lại nhiều nhất trong 4 thuật toán ta xét. Vậy nên thuật toán này tốn không gian lưu trữ.



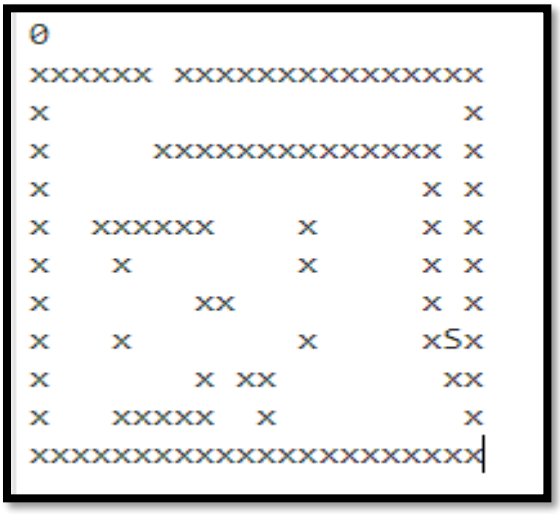
- *Thuật toán Deapth First Search:* đưa ra được giải phải đường đi nhưng chưa phải tối ưu nhất với chi phí đường đi là 60. Tuy nhiên thuật toán này lại duyệt ít đỉnh hơn thuật toán BFS nên được lợi về không gian lữu trữ.

- *Thuật toán Greedy Best First Search:* ở trường hợp này, thuật toán GBFS chưa đưa ra được đường đi tối ưu với chi phí 52. Nhưng được lợi là điểm cần duyệt ít hơn cả BFS và DFS nhờ có hàm đánh giá heuristic tối ưu hơn mặc dù chi phí cao hơn của BFS.

- *Thuật toán A*:* cho ta được đường đi tối ưu với chi phí thấp nhất. Nhờ hàm đánh giá heuristic tối ưu và hoàn thiện của A* nên số điểm cần duyệt của A* là ít nhất.

➔ Kết luận: Thuật toán A* là tối ưu nhất với bản đồ này, vừa ít chi phí, vừa đỡ tốn không gian lưu trữ.

5. Bản đồ 5:



- *Thuật toán Breadth First Search:* cho được đường đi tối tối ưu với chi phí thấp nhất. Ở trường hợp này, các điểm cần duyệt của A cũng thấp nhất (có 2 điểm kề nhưng đã có 2 điểm duyệt rồi nên chỉ cần duyệt điểm còn lại). Đây là trường

hợp tốt nhất về mọi mặt cho thuật toán này.

- *Thuật toán Deapth First Search:* cho được đường đi tối ưu. Về cơ bản, đây là thuật toán tìm giải pháp từ "điểm có giá trị nhỏ nhất". Trong trường hợp này cũng là tốt nhất cho DFS.

- *Thuật toán Greedy Best First Search:* tìm kiếm đường đi theo đánh giá heuristic, thuật toán sẽ đi theo điểm mà nó cho là gần đích nhất.

- *Thuật toán A*:* là thuật toán vừa tối ưu vừa hoàn thiện sẽ đảm bảo tìm được đường đi tốt nhất.

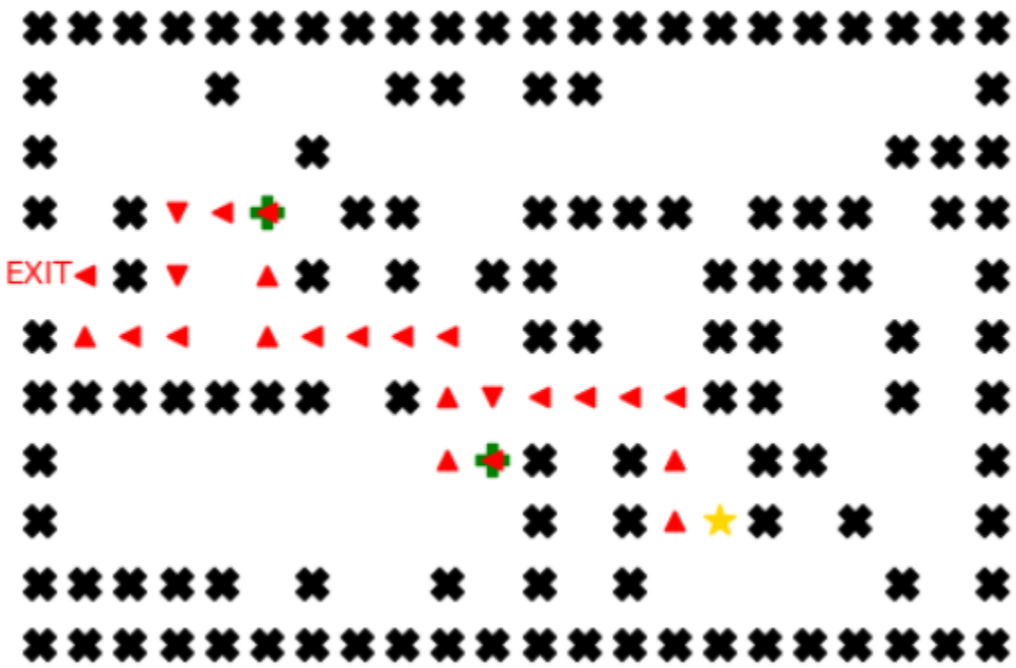
➔ Kết luận: Ở bản đồ này, các thuật toán đều tìm được đường đi tốt nhất có chi phí thấp nhất và xét số điểm cần xét ít nhất.

III. Kết quả thực thi trên bản đồ (có điểm thưởng):

1. 2 điểm thưởng:

```
2
3  5  -8
7  10  -20
xxxxxxxxxxxxxxxxxxxxxxx
x    x    xx xx          x
x       x                xxx
x x   + xx   xxxx xxx xx
  x    x x xx    xxxx   x
x             xx   xx   x x
xxxxxxx x           xx   x x
x            +x x   xx    x
x             x x Sx x    x
xxxxx x   x x x       x x
xxxxxxxxxxxxxxxxxxxxxxxx
```

- Kết quả khi chạy thuật toán:



➔ Nhận xét: Thuật toán này cho ra được đường đi tối ưu và ít chi phí nhất. Nó dựa vào hàm đánh giá heuristic để xác định được điểm thưởng nào gần nhất và có nên đi qua điểm thưởng đó hay không. Ở đây ta có 2 điểm thưởng 1 điểm 8 và

1 điểm 20, thuật toán đã đi qua cả 2 điểm do cả 2 điểm đều giúp rút ngắn chi phí đường đi.

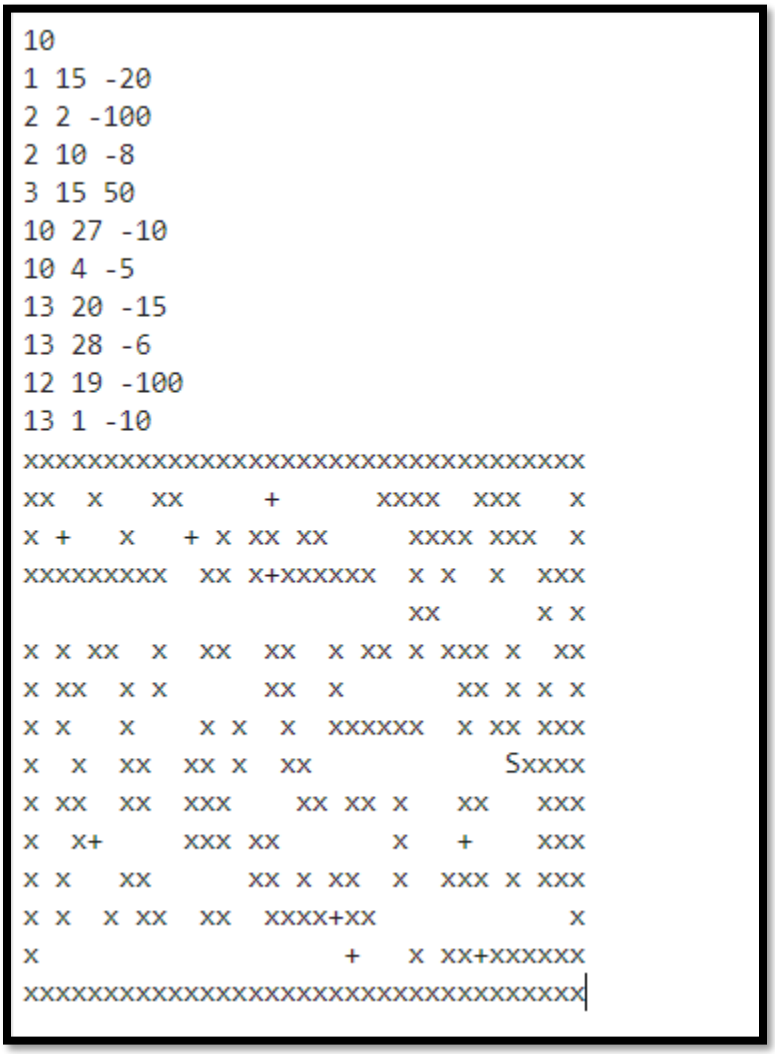
2. 5 điểm thưởng:

```
5
3 5 -8
2 3 -1
7 10 -20
7 6 -1
5 14 -8
xxxxxxxxxxxxxxxxxxxxxx
x    x    xx xx          x
x   +   x               xxx
x x   + xx   xxxx xxx xx
   x    x x xx    xxxx   x
x             xx +xx   x x
xxxxxxx x        xx   x x
x       +    +x x   xx    x
x             x x Sx x   x
xxxxx x   x x x       x x
xxxxxxxxxxxxxxxxxxxxxx
```

- Kết quả khi chạy thuật toán:

➔ Nhật xét: Thuật toán cho ra được đường đi tối ưu và có chi phí nhỏ nhất (-14). Chương trình sẽ duyệt điểm thưởng nào gần điểm đang xét nhất và chi phí khi đến đích sẽ giảm khi đi qua điểm thưởng đó thì sẽ hướng đường đi về điểm thưởng đã chọn. Theo kết quả bài toán, có 1 điểm thưởng không được đi qua do điểm của điểm thưởng là 1, nếu thuật toán cho đường đi đi qua điểm đó thì sẽ phải vòng lại tốn nhiều chi phí hơn là không đi qua điểm thưởng đấy.

3. 10 điểm thưởng:



- Kết quả bài toán:

➔ Nhận xét: Ta nâng cấp bài toán điểm thưởng từ thuật toán DFS kết hợp với hàm đánh giá đường đi heuristic nên trong số ít trường hợp thuật toán không đưa được ra dạng đường đi tối ưu và có chi phí thấp nhất. Tuy ở bản đồ này thuật toán đã đi đến những điểm thưởng có lợi nhưng cách tìm đường đi ở một số đoạn vẫn còn chưa hợp lí gây mất thêm chi phí không đáng có. Đây là trường hợp mà thuật toán không đưa ra được đường đi tối ưu có chí phí nhỏ nhất.

IV. Tham khảo:

1. Tham khảo từ giáo trình:

- Slide bài giảng trên mooddle.

2. Tham khảo từ internet:

- https://simplecodecjava.blogspot.com/2015/09/thuat-toan-tim-uong-i-giua-hai-inh-cua.html