TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

ĐẠI HỌC QUỐC GIA TP. HCM

Cơ sở trí tuệ nhân tạo

Đồ án: Ứng dụng các thuật toán tìm kiếm

19120615 – Hùng Ngọc Phát
19120621 – Lê Minh Phục

Ngày 14 tháng 11 năm 2021

Mục lục

Chương 1

Giới thiệu

1.1 Về thành viên nhóm và phân công

Nhóm có 2 thành viên:

- **Lê Minh Phục (19120629):** hoàn thành các thuật toán tìm kiếm không điểm thưởng và viết báo cáo cho phần này.
- **Hùng Ngọc Phát (19120615):** hoàn thành các thư viện bổ trợ và làm phần bản đồ có điểm thưởng và có cánh cổng. Thiết kế test cases.

Mức độ hoàn thành của từng hạng mục

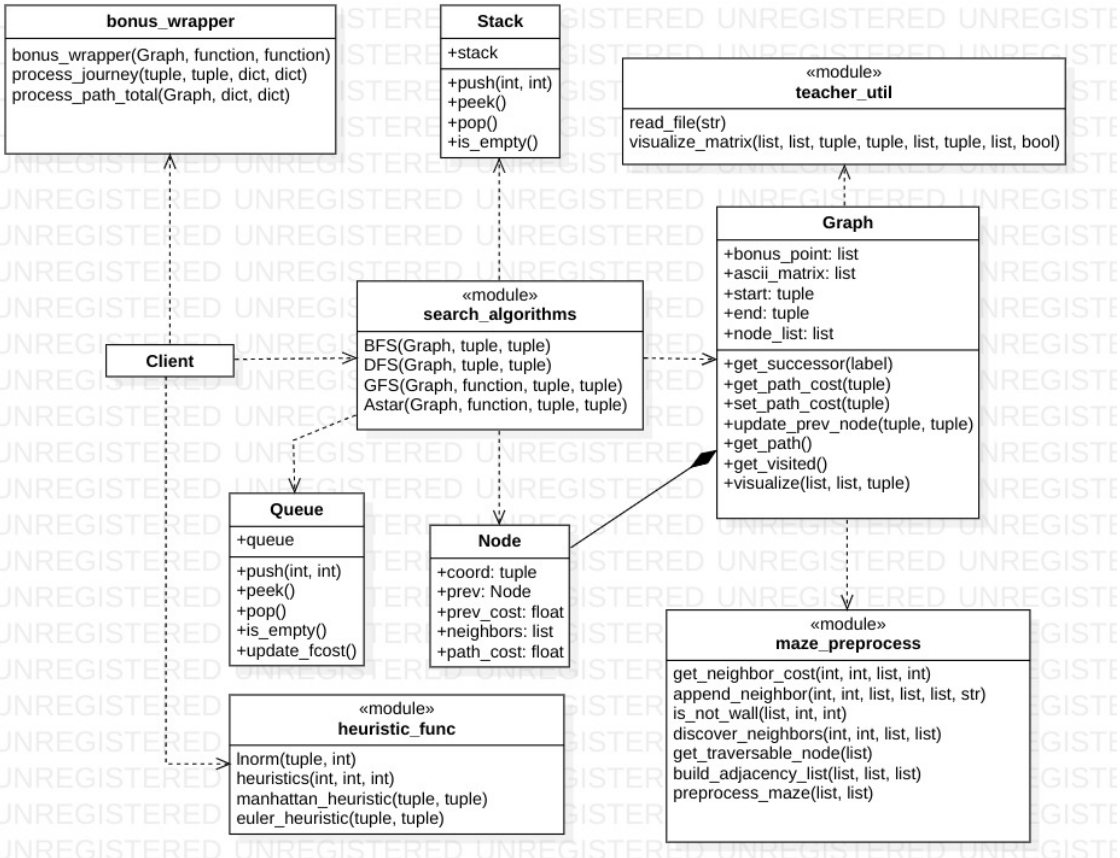
Hạng mục	Không điểm thưởng	Có điểm thưởng	Có cánh cổng
DFS	100%	-	
BFS	100%	-	
GBFS	100%	70%	60%
A*	100%	70%	60%

1.2 Về kiến trúc của chương trình, kiến trúc dữ liệu và processing pipeline

1.2.1 Kiến trúc chương trình

"Chương trình báo cáo" này được viết bằng Jupyter Notebook để dễ dàng visualize các hình ảnh cũng như tiện lợi cho việc tính toán, sau đó được xuất sang LaTeX. Thầy có thể thử chạy lại notebook báo cáo này trong thư mục `report-src`.

Chương trình gồm các lớp đối tượng sau:



Ghi chú: các "class" có gắn `<<module>>` không phải là class mà chỉ là tập hợp các hàm trong 1 file `*.py` (module).

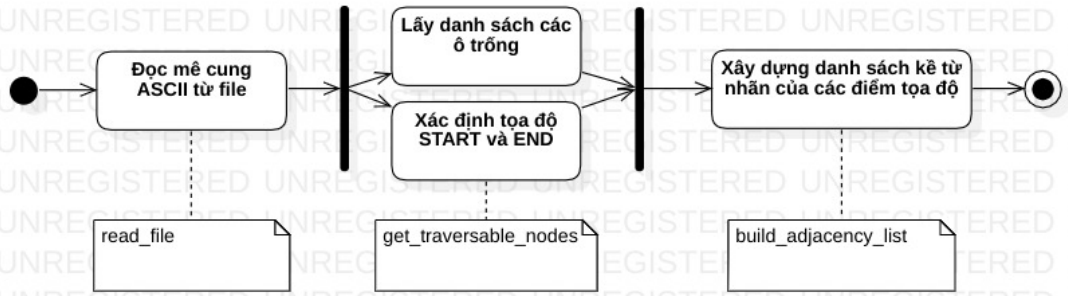
- `Node`: lớp đối tượng thể hiện một đỉnh (vertex) trên đồ thị.
- `Stack`: lớp đối tượng ngăn xếp.
- `Queue`: lớp đối tượng cho hàng đợi và hàng đợi ưu tiên.
- `Graph`: lớp đối tượng cho một đồ thị được sinh ra từ mê cung.
- `searching_algorithms`: module chứa các hàm cài đặt các thuật toán tìm kiếm.
- `maze_preprocessor`: module chứa các hàm tiền xử lý mê cung.
- `heuristic_func`: module chứa các hàm heuristic.
- `teacher_util`: module chứa 2 hàm mà thầy cung cấp trong notebook.

Các hàm tìm kiếm sẽ sử dụng lớp `Graph` để truy xuất các đỉnh kề, lấy, cập nhật các giá trị chi phí, ... mà không cần thao tác trực tiếp trên danh sách kề.

Mỗi hàm nhóm em viết đều có một docstring mô tả input/output và chức năng. Thầy có thể tham khảo source code để biết thêm.

1.2.2 Quá trình tiền xử lý mê cung

- Để các thuật toán tìm kiếm đường đi có thể chạy được, ta cần chuyển đổi mê cung dưới dạng ASCII như trong file `txt` thành một đồ thị. Ở đây nhóm em đã quyết định chọn một danh sách kề để biểu diễn một đồ thị sinh ra từ mê cung. Về kiến trúc của các hàm em có để ở phần tài liệu tham khảo (còn về implementation và bổ sung trọng số thì của nhóm em tự làm).
- Quá trình tiền xử lý mê cung được thực hiện bởi các hàm trong module `maze_preprocessor`, theo thứ tự sau:



1. Đọc mê cung từ file ASCII.
2. Lấy danh sách các ô trống & xác định tọa độ START & END.
3. Thiết lập danh sách kề: quét các ô đi được đã tìm được ở trên và tìm các ô kề với nó, xây dựng nên một *hữu hướng* đơn đồ thị (có dạng lưới). Từ đó xây dựng được danh sách kề của đồ thị này. Có nghĩa là, đồ thị được sinh ra này có các đỉnh là các ô đi được trên mê cung (không chứa các ô tường).

Về trọng số các cạnh:

- Với mê cung không điểm thưởng: đồ thị là vô hướng đơn đồ thị. Tất cả các cạnh có trọng số là 1.
- Với mê cung có điểm thưởng: đồ thị là hữu hướng đa đồ thị. Cạnh nối 2 điểm không có điểm thưởng sẽ có giá trị bằng 1 ở cả 2 chiều. Cạnh nào nối vào đỉnh điểm thưởng thì trọng số sẽ được cộng với giá trị của điểm thưởng đó (giá trị điểm thưởng là âm).

Chương 2

Bản đồ không có điểm thưởng

Ở phần này,chúng em sẽ cố gắng chọn các bản đồ càng đơn giản càng tốt, nhằm rút trích ra các điểm khác nhau chính giữa các chiến thuật của các thuật toán. Riêng 2 bản đồ lớn ở cuối thì em sinh ngẫu nhiên từ trang https://www.dcode.fr/maze-generator (có chỉnh sửa lại để làm rõ sự khác nhau giữa các thuật toán).

```
In [13]: %matplotlib inline
         %cd ../../source
```

```
[WinError 2] The system cannot find the file specified: '../../source'
e:\Học Tập\BTVN_Chuyên_ngành\CSTTNT\HCMUS-IntroAI-Projects\source
```

```
In [14]: from Graph import Graph
         from searching_algorithms import *
         from heuristic_func import euclide_heuristic, manhattan_heuristic
```

```
In [15]: # Hàm client để chạy thuật toán tìm kiếm
         def run_search_nobonus(g: Graph, algorithm, heuristic=None, figsize=(5, 3)):
             # g: đồ thị được đọc vào
             # algorithm: con trỏ đến hàm tìm kiếm cần chạy
             g.clear()
             if heuristic:
                 algorithm(g, heuristic)
             else:
                 algorithm(g)
             visited, path, cost = g.get_visited()
             g.visualize(path, visited=visited, figsize=figsize)
             print('Cost:', cost)
```

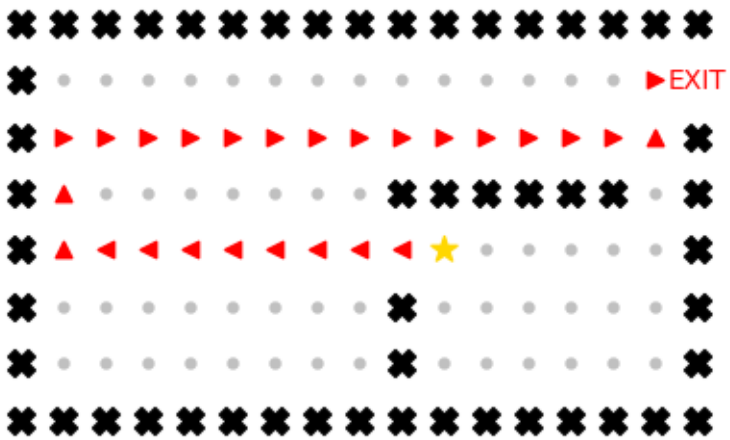
2.1 Bản đồ 1 (8x17)

```
In [40]: g1 = Graph('testcases/nobonus1.txt')
```

```
Graph initialized from maze with size 8 x 17
```

2.1.1 DFS

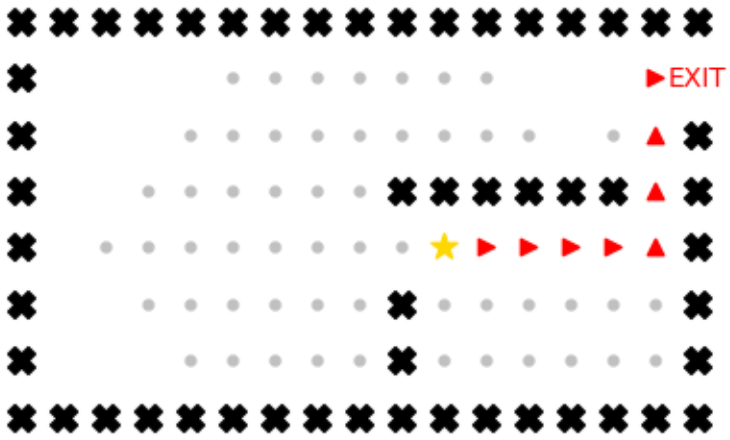
```
In [41]: run_search_nobonus(g1, DFS)
```




```
Starting point (x, y) = (4, 10)
Ending point (x, y) = (1, 16)
Cost: 27
```

2.1.2 BFS

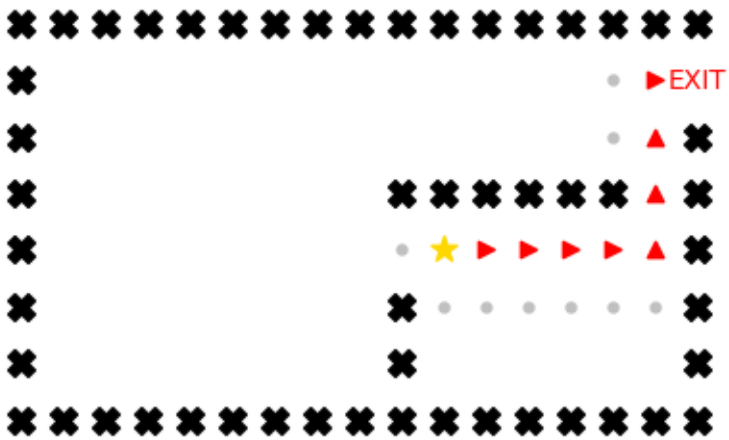
```
In [42]: run_search_nobonus(g1, BFS)
```




```
Starting point (x, y) = (4, 10)
Ending point (x, y) = (1, 16)
Cost: 9
```

2.1.3 Greedy Best-first Search

```
In [43]: run_search_nobonus(g1, GBFS, manhattan_heuristic)
```




```
Starting point (x, y) = (4, 10)
Ending point (x, y) = (1, 16)
Cost: 9
```

2.1.4 A*

```
In [44]: # Đồ thị 1, A*
         run_search_nobonus(g1, Astar, manhattan_heuristic)
```


```
Starting point (x, y) = (4, 10)
Ending point (x, y) = (1, 16)
Cost: 9
```

Chú thích: màu đỏ đại diện cho đường đi mà thuật toán đã tìm được, còn các hình tròn màu màu xám đại diện cho các ô mà thuật toán đã từng ghé thăm.

GBFS, BFS và A* đã tìm được đường đi ngắn nhất từ `START` đến `EXIT`. Còn DFS thì do chỉ "chăm chăm" vào đi một lối nên nó đã phải duyệt hết tất cả các hướng của bản đồ mới tìm được lối ra (điều này còn tùy thuộc vào implementation của DFS. Nhóm em tạo danh sách kề theo thứ tự: trên, dưới, phải, trái, nên DFS sẽ duyệt theo thứ tự ngược lại (vì nó dùng stack làm fringe)).

Ngoài ra, ta cũng thấy được trong map này GBFS và A* duyệt ít ô nhất (các ô màu xám) so với 2 thuật toán còn lại.

Thuật toán	Chi phí
DFS	27
BFS	9
GBFS (M)	9
A* (M)	9

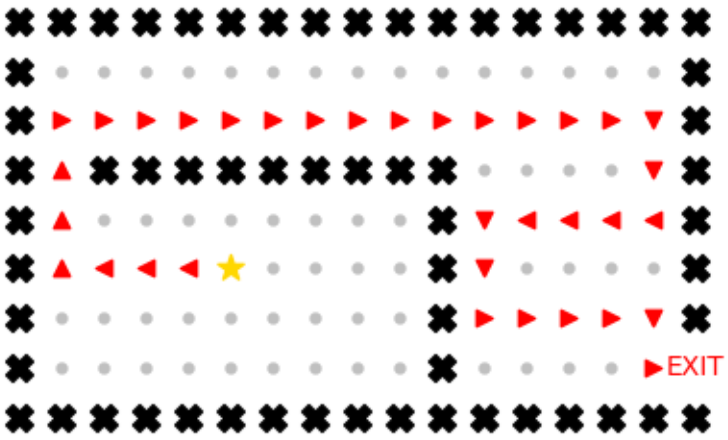
2.2 Bản đồ 2 (7x17)

```
In [45]: g2 = Graph('testcases/nobonus2.txt')

Graph initialized from maze with size 9 x 17
```

2.2.1 DFS

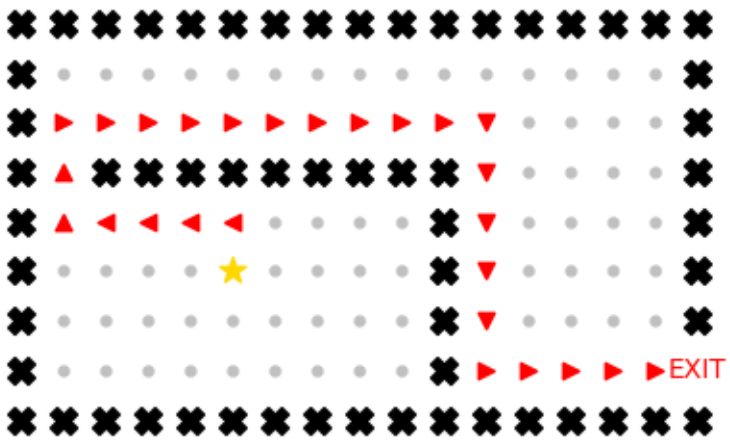
```
In [46]: run_search_nobonus(g2, DFS)
```




```
Starting point (x, y) = (5, 5)
Ending point (x, y) = (7, 16)
Cost: 35
```

2.2.2 BFS

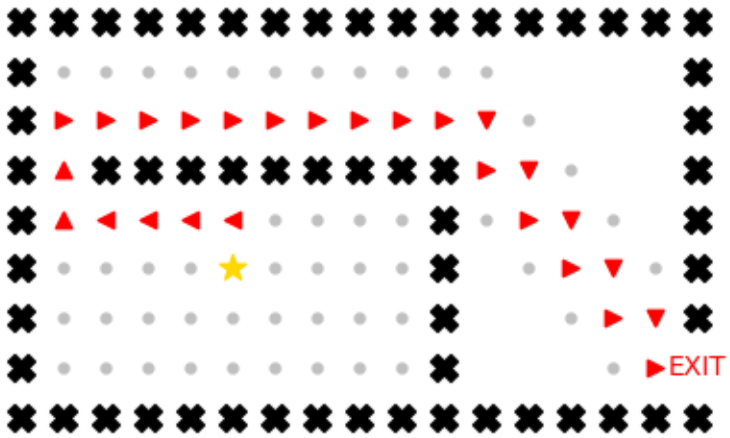
```
In [47]: run_search_nobonus(g2, BFS)
```




```
Starting point (x, y) = (5, 5)
Ending point (x, y) = (7, 16)
Cost: 27
```

2.2.3 Greedy Best-first Search

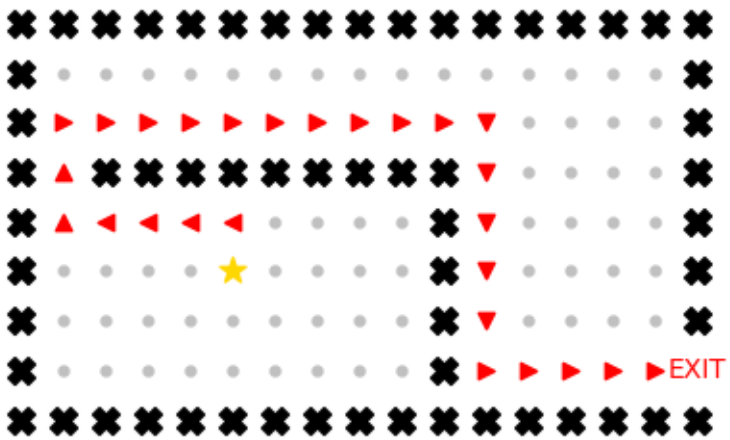
```
In [48]: run_search_nobonus(g2, GBFS, euclide_heuristic)
```




```
Starting point (x, y) = (5, 5)
Ending point (x, y) = (7, 16)
Cost: 27
```

2.2.4 A* (Manhattan)

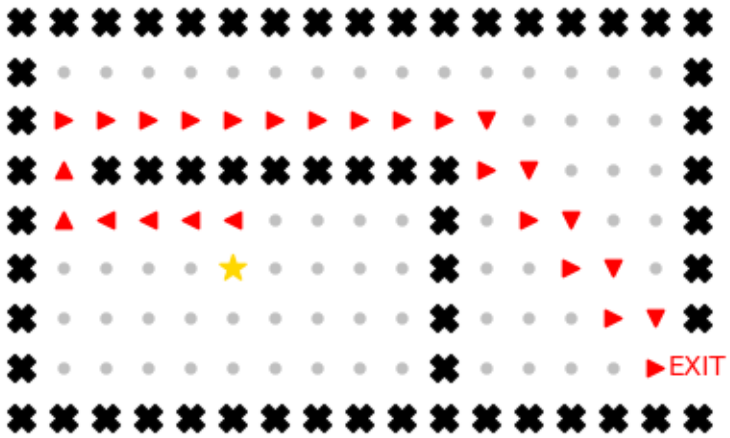
```
In [49]: run_search_nobonus(g2, Astar, manhattan_heuristic)
```




```
Starting point (x, y) = (5, 5)
Ending point (x, y) = (7, 16)
Cost: 27
```

2.2.5 A* (Euclide)

```
In [50]: run_search_nobonus(g2, Astar, euclide_heuristic)
```




```
Starting point (x, y) = (5, 5)
Ending point (x, y) = (7, 16)
Cost: 27
```

Cũng tương tự như trên, BFS, GBFS và A* đã tìm được đường đi ngắn nhất từ `START` đến `EXIT`, còn DFS thì lại đi lòng vòng. Ở đây chúng em cũng cho thử A* với 2 hàm heuristic là hàm khoảng cách Manhattan và Euclide. Chi phí đường đi trong 2 trường hợp là như nhau, nhưng khi sử dụng heuristic là hàm Euclide

thì thuật toán có xu hướng đi chéo (vì khoảng cách Euclide là khoảng cách "theo đường chéo" trên hệ trục tọa độ).

Chiến lược của A* khác GBFS ở chỗ GBFS sẽ chạy thẳng tới điểm mà nó cho là gần đích nhất, trong khi đó A* sẽ mở thêm các node lân cận để "xem thử" có điểm nào gần đích hơn là đường đi mà nó vừa tìm được hay không.

Thuật toán	Chi phí
DFS	35
BFS	27
GBFS (E)	27
A* (M)	27
A* (E)	27

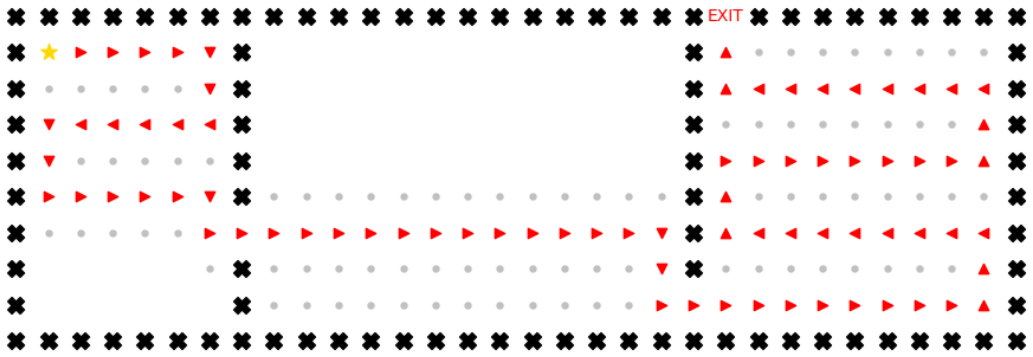
2.3 Bản đồ 3 (10x32)

```
In [51]: g3 = Graph('testcases/nobonus3.txt')

Graph initialized from maze with size 10 x 32
```

2.3.1 DFS

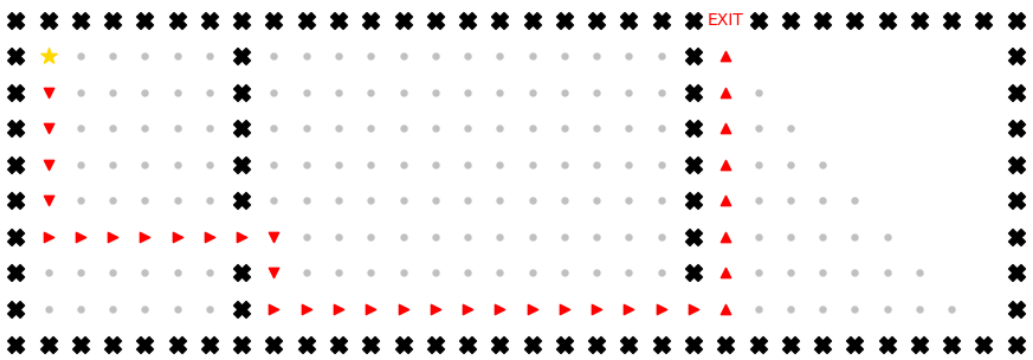
```
In [52]: run_search_nobonus(g3, DFS, figsize=(12, 4))
```




```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (0, 22)
Cost: 78
```

2.3.2 BFS

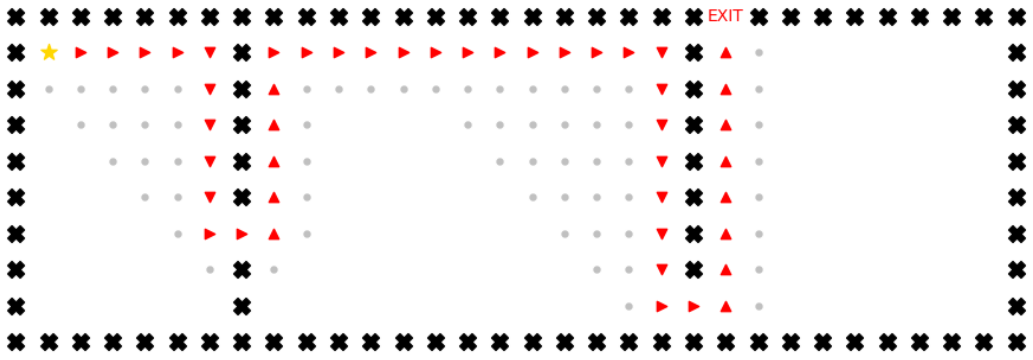
```
In [53]: run_search_nobonus(g3, BFS,figsize=(12, 4))
```




```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (0, 22)
Cost: 36
```

2.3.3 Greedy Best-first Search (Manhattan)

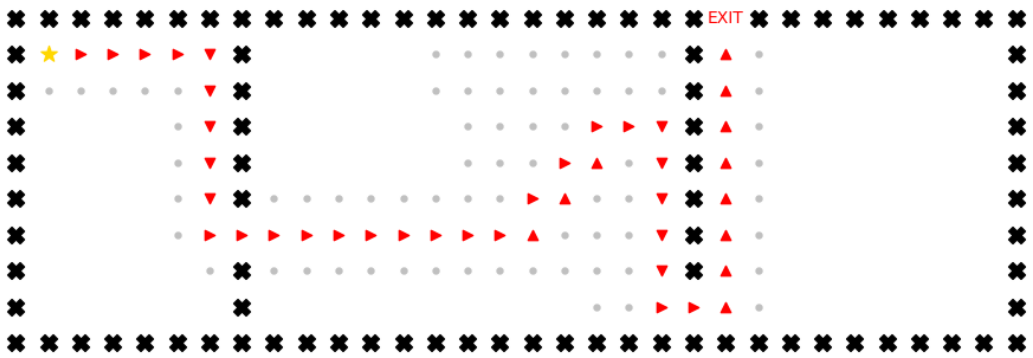
```
In [54]: run_search_nobonus(g3, GBFS,manhattan_heuristic,figsize=(12, 4))
```




```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (0, 22)
Cost: 46
```

2.3.4 Greedy Best-first Search (Euclide)

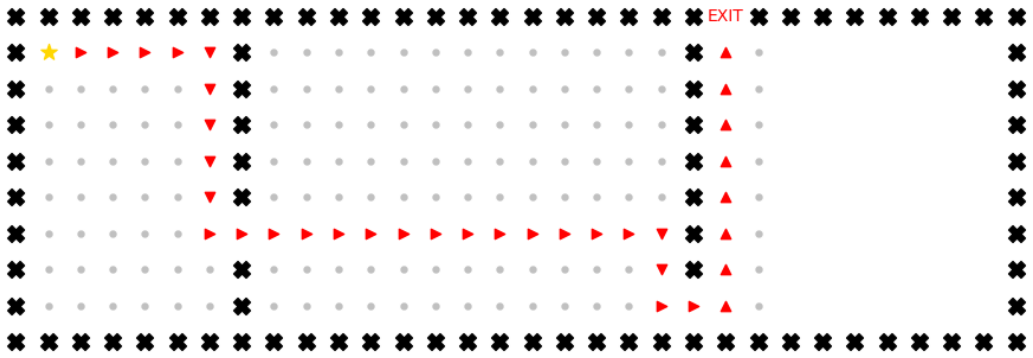
```
In [55]: run_search_nobonus(g3, GBFS, euclide_heuristic,figsize=(12,4))
```




```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (0, 22)
Cost: 42
```

2.3.5 A* (Manhattan)

```
In [56]: run_search_nobonus(g3, Astar,manhattan_heuristic,figsize=(12, 4))
```




```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (0, 22)
Cost: 36
```

BFS và A* đã tìm được đường đi ngắn nhất từ `START` đến `EXIT`. DFS vẫn là thuật toán tốn nhiều chi phí nhất khi đi lòng vòng.

Ở đây chúng em thử GBFS với 2 hàm heuristic là hàm khoảng cách Manhattan và Euclide, chi phí đường đi của 2 trường hợp này là khác nhau và không có trường hợp nào tìm được đường đi ngắn nhất do điểm `EXIT` "được" một bờ tường che, khiến cách tìm điểm gần nhất với đích của GBFS không hiệu quả (*"tham lam là không tốt"*). Tuy nhiên với hàm khoảng cách Euclide thì GBFS "nhận ra" được bờ tường sớm hơn.

Ngoài ra, ta cũng thấy được trong bản đồ này A* duyệt ít ô hơn (các ô màu xám) so với thuật toán BFS.

Thuật toán	Chi phí
DFS	78
BFS	36
GBFS (M)	46
GBFS (E)	42
A* (M)	36

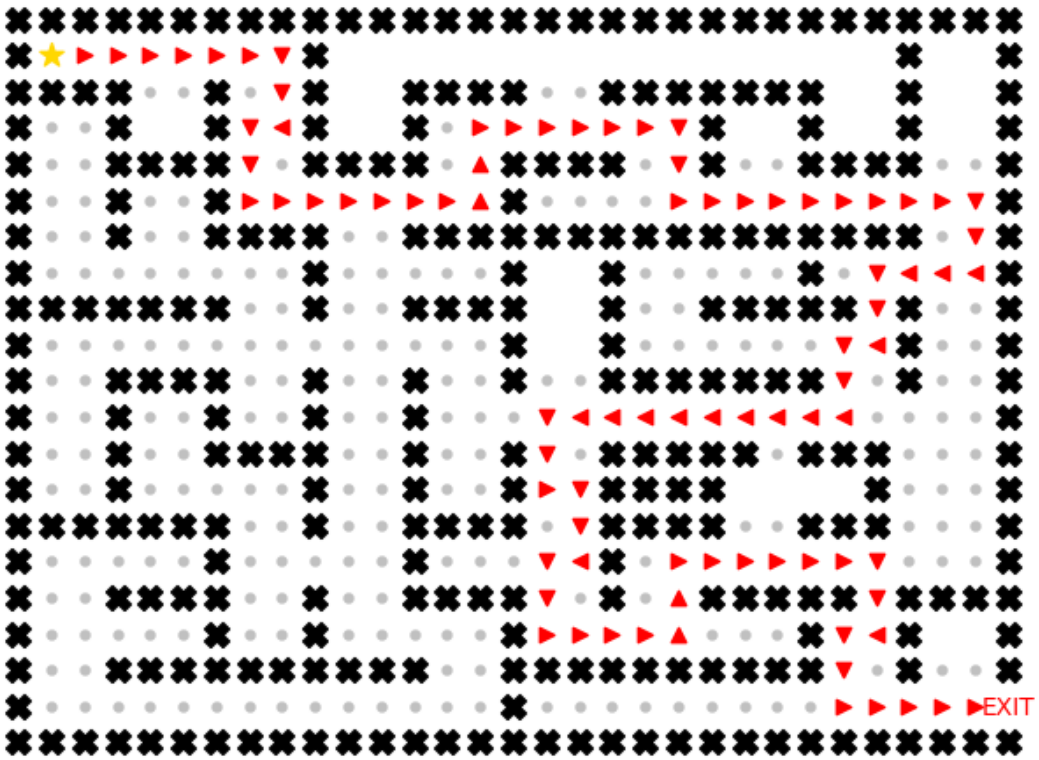
2.4 Bản đồ 4 (21x31)

```
In [6]: g4 = Graph('testcases/nobonus4.txt')
```

```
Graph initialized from maze with size 21 x 31
```

2.4.1 DFS

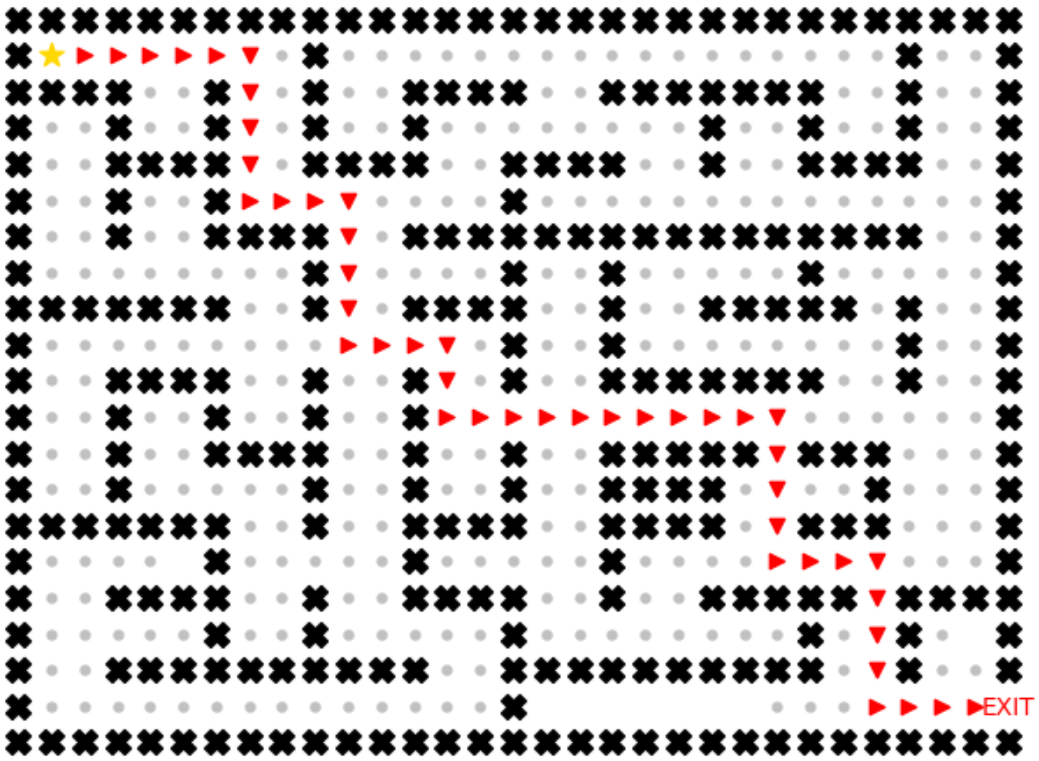
```
In [58]: run_search_nobonus(g4, DFS, figsize=(8, 6))
```




```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (19, 30)
Cost: 87
```

2.4.2 BFS

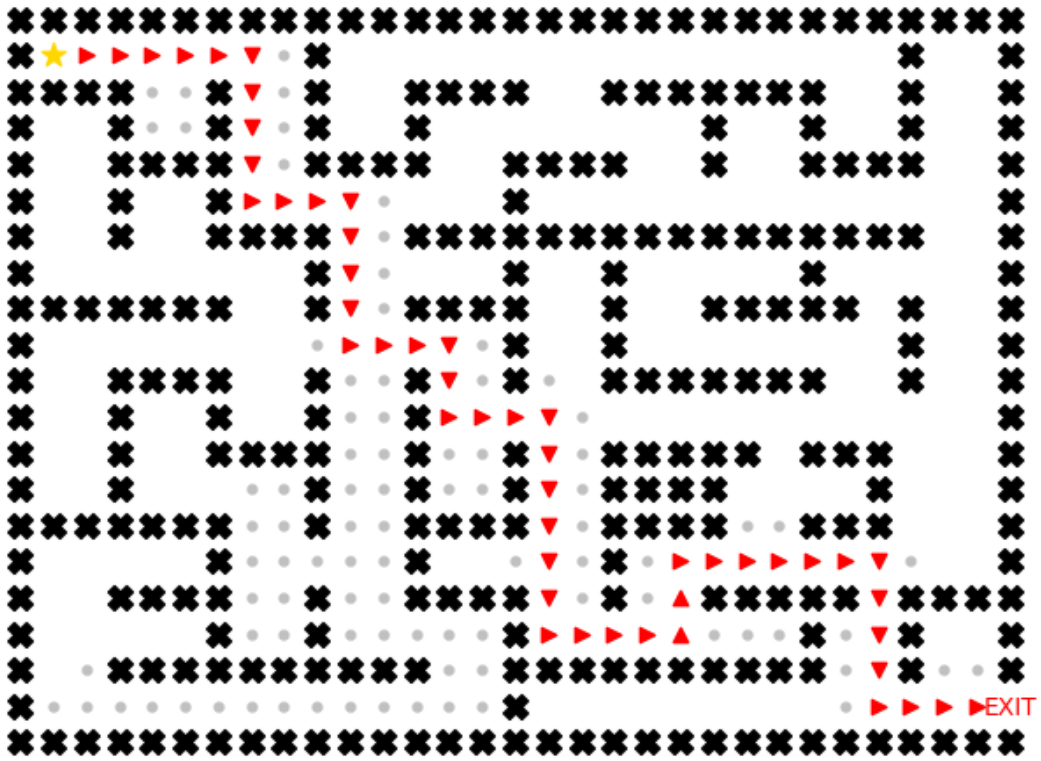
```
In [59]: run_search_nobonus(g4, BFS, figsize=(8, 6))
```




```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (19, 30)
Cost: 47
```

2.4.3 Greedy Best-first Search (Manhattan)

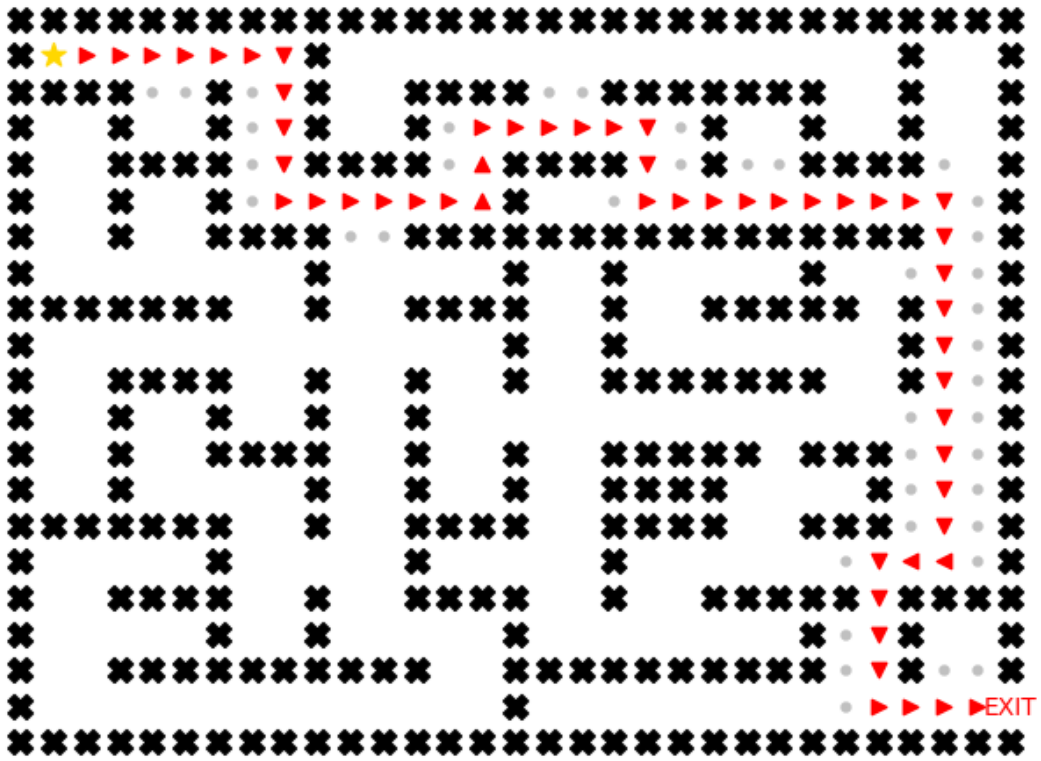
```
In [7]: run_search_nobonus(g4, GBFS,manhattan_heuristic,figsize=(8, 6))
```




```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (19, 30)
Cost: 51
```

2.4.4 Greedy Best-first Search (Euclide)

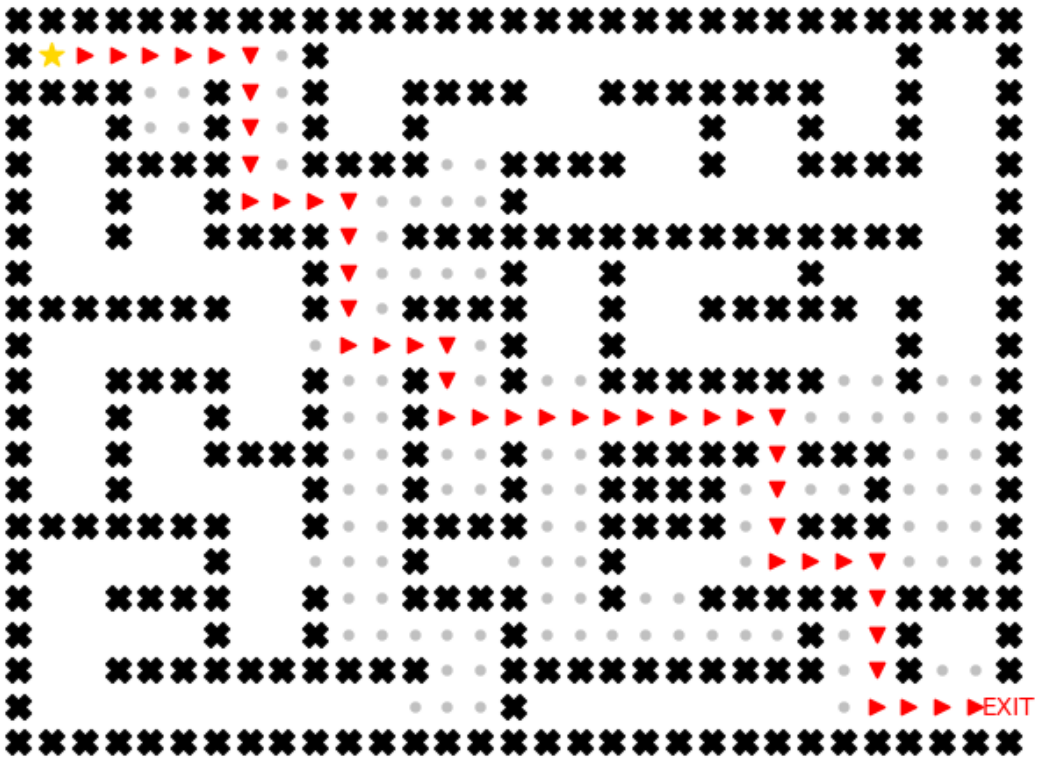
```
In [8]: run_search_nobonus(g4, GBFS, euclide_heuristic,figsize=(8,6))
```




```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (19, 30)
Cost: 55
```

2.4.5 A* (Manhattan)

```
In [ ]: run_search_nobonus(g4, Astar,manhattan_heuristic,figsize=(8, 6))
```




```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (19, 30)
Cost: 47
```

BFS và A* đã tìm được đường đi ngắn nhất từ `START` đến `EXIT`. DFS vẫn là thuật toán tốn nhiều chi phí nhất khi đi lòng vòng.

Ở đây chúng em thử GBFS với 2 hàm heuristic là hàm khoảng cách Manhattan và Euclide,chi phí đường đi của 2 trường hợp này là khác nhau và không có trường hợp nào tìm được đường đi ngắn nhất. Tuy nhiên khác với mê cung 3, trong mê cung này hàm khoảng cách Euclide tỏ ra kém hiệu quả hơn hàm khoảng cách Manhattan do đường đi ngắn nhất mê cung có ít sự ngăn cản từ các bức tường.

Ngoài ra, ta cũng thấy được trong bản đồ này A* duyệt ít ô hơn (các ô màu xám) so với thuật toán BFS.

Thuật toán	Chi phí
DFS	87
BFS	47
GBFS (M)	51
GBFS (E)	55
A* (M)	47

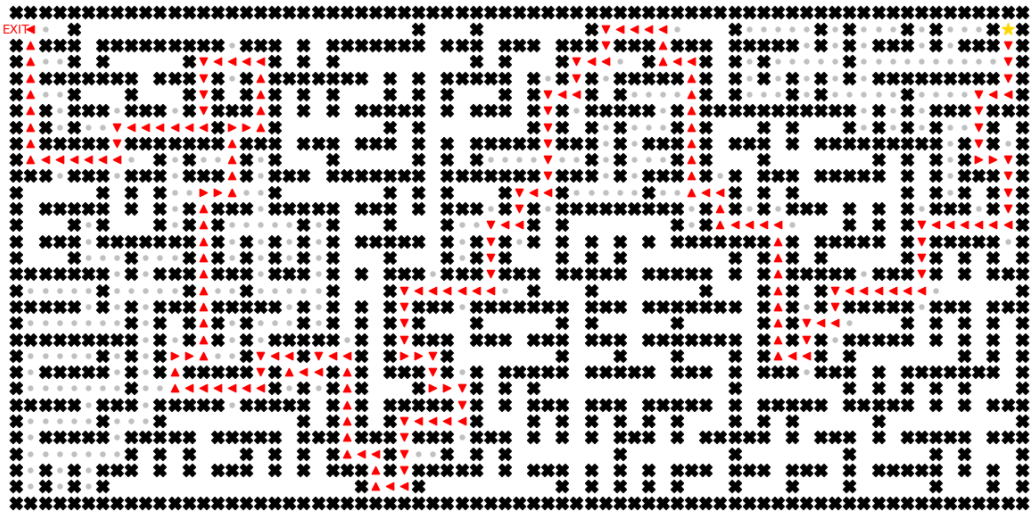
2.5 Bản đồ 5 (31x71)

```
In [63]: g5 = Graph('testcases/nobonus5.txt')
```

```
Graph initialized from maze with size 31 x 71
```

2.5.1 DFS

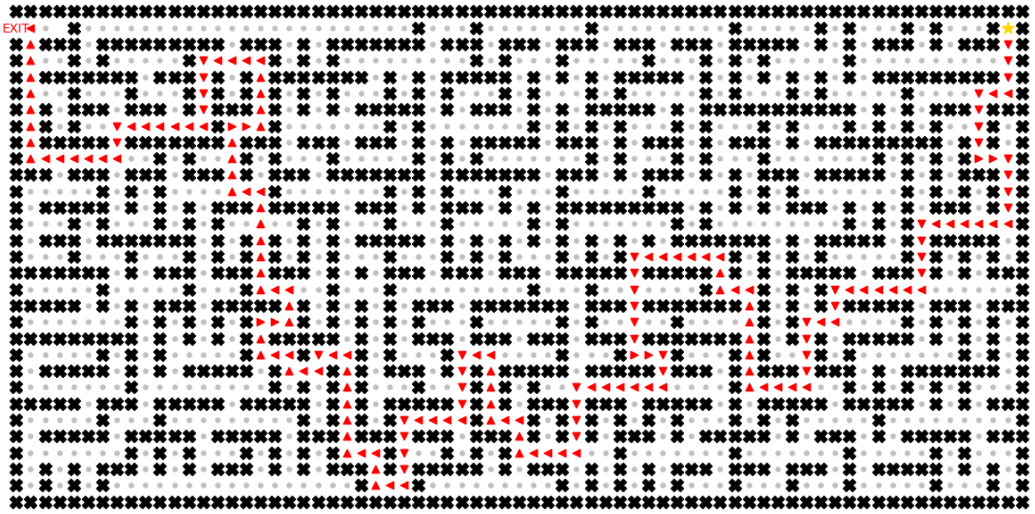
```
In [64]: run_search_nobonus(g5, DFS, figsize=(16, 8))
```



```
Starting point (x, y) = (1, 69)
Ending point (x, y) = (1, 0)
Cost: 207
```

2.5.2 BFS

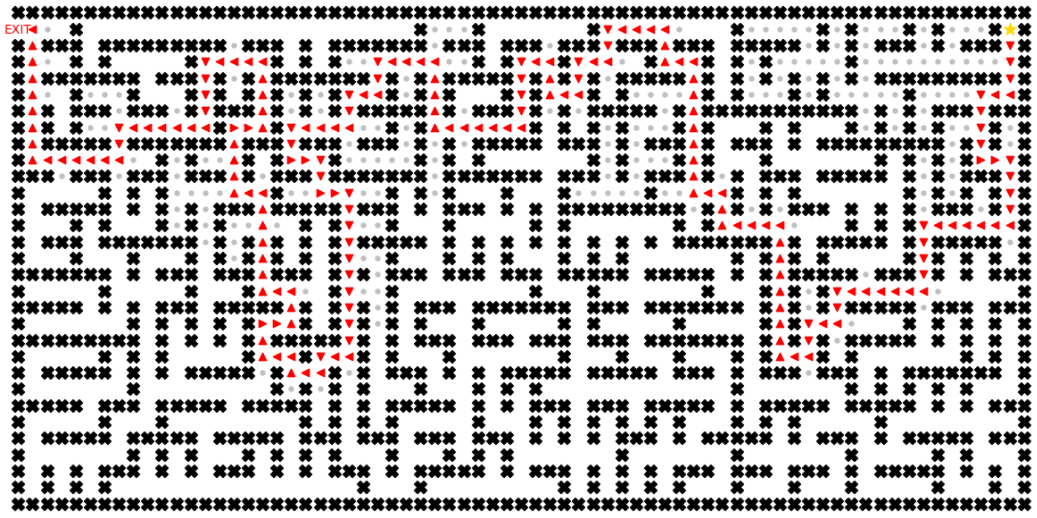
```
In [65]: run_search_nobonus(g5, BFS, figsize=(16, 8))
```



```
Starting point (x, y) = (1, 69)
Ending point (x, y) = (1, 0)
Cost: 183
```

2.5.3 Greedy Best-first Search (Euclide)

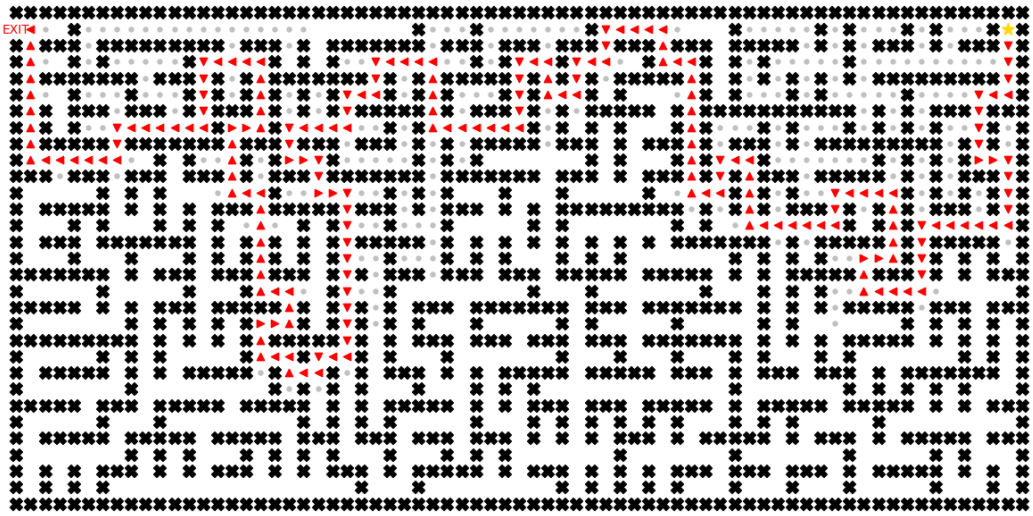
```
In [66]: run_search_nobonus(g5, GBFS, euclide_heuristic,figsize=(16,8))
```




```
Starting point (x, y) = (1, 69)
Ending point (x, y) = (1, 0)
Cost: 195
```

2.5.4 Greedy Best-first Search (Manhattan)

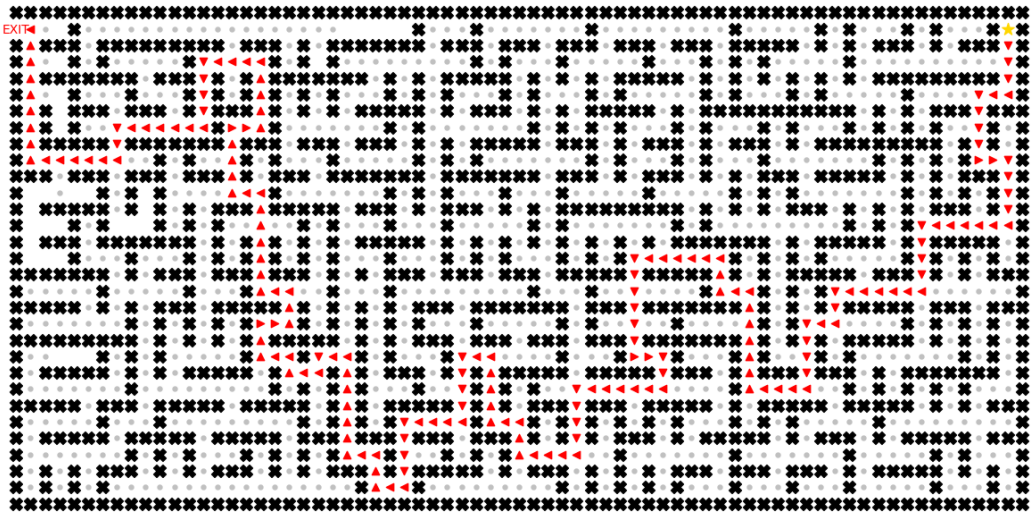
```
In [68]: run_search_nobonus(g5, GBFS,manhattan_heuristic,figsize=(16, 8))
```



```
Starting point (x, y) = (1, 69)
Ending point (x, y) = (1, 0)
Cost: 199
```

2.5.5 A* (Euclide)

```
In [67]: run_search_nobonus(g5, Astar, euclide_heuristic,figsize=(16,8))
```



```
Starting point (x, y) = (1, 69)
Ending point (x, y) = (1, 0)
Cost: 183
```

BFS và A* đã tìm được đường đi ngắn nhất từ `START` đến `EXIT`. DFS vẫn là thuật toán tốn nhiều chi phí nhất khi đi lòng vòng.

Ở đây chúng em thử GBFS với 2 hàm heuristic là hàm khoảng cách Manhattan và Euclide, chi phí đường đi của 2 trường hợp này là khác nhau và không có trường hợp nào tìm được đường đi ngắn nhất. Tuy nhiên khác với mê cung 4, trong mê cung này hàm khoảng cách Euclid tỏ ra hiệu quả hơn hàm khoảng cách Manhattan do đường đi ngắn nhất mê cung có nhiều sự ngăn cản từ các bức tường.

Ngoài ra, ta cũng thấy được trong bản đồ này A* duyệt ít ô hơn (các ô màu xám) so với thuật toán BFS.

Thuật toán	Chi phí
DFS	207
BFS	183
GBFS (E)	195
GBFS (M)	199
A* (M)	183

Chương 3

Bản đồ có điểm thưởng

```
In [1]: %matplotlib inline
        %cd ../../source
        from Graph import Graph
        from searching_algorithms import *
        from heuristic_func import euclide_heuristic, manhattan_heuristic
        from bonus_wrapper import *
```

```
/home/hiraki/source/IntroAI/Project01/source
```

Với trường hợp này, tụi em đã nghĩ ra một phương án đó là ta sẽ tìm các chặng đường đường đi ngang qua các điểm thưởng sao cho tại mỗi đầu mút của chặng, ta sẽ đi đến điểm thưởng tiếp theo mà có tổng đường đi từ điểm hiện tại đến điểm đó là thấp nhất (đi trên từng chặng).

Cụ thể, tụi em sẽ xây dựng một thuật toán wrapper như sau (nó sẽ gọi lại hàm tìm kiếm để thực hiện việc tìm kiếm trên từng chặng):

Duy trì một tập để chứa các ô bị cấm (ô ngõ cụt); một bảng tra chặng để biết được trước khi đi vào một node ta đã ở node nào (nhằm quay lại khi gặp ngõ cụt); một bảng tra để biết được từ một node nào đó, ta đã từng đi đến các node nào; và một tập để lưu đường đi trên từng chặng.

1. Gán node hiện tại `current = START`.
2. Nếu tất cả các điểm đi được đều đã bị cấm, kết thúc giải thuật.
3. Thêm các điểm đã đi của node liền trước vào danh sách các điểm đã đi của current (để không phát sinh chu trình).
4. Tìm ra điểm thưởng gần nhất (*) với `current`, gọi đó là `dest`, sao cho từ `current` chưa đi `dest` và `dest` không bị cấm và `dest` != `current`.
5. Nếu không tồn tại `dest` nào thỏa, thêm `current` vào danh sách cấm, gán node `current` bằng node liền trước của nó. Quay lại bước 2.
6. Gọi thuật toán (GBFS, A*) để tìm đường đi từ `current` đến `dest` sao cho* không trùng* với đường đi từ node liền trước vào `current` (để một ô không bị đi vào 2 lần). Thêm `dest` vào danh sách các node đã đi từ `current`.
7. Nếu không tồn tại đường đi, quay lại B2.
8. Nếu có đường đi từ `current` đến `dest`, ghi lại đường đi cũng như gán node liền trước `dest` là `current`.
9. Gán `current = dest`. Nếu `current` ≡ `end`, kết thúc. Ngược lại, quay lại B2.

(*) **Để đánh giá như thế nào là "gần nhất"**, tụi em sử dụng hàm đánh giá sau:

$$f(u,v) = h(u,v) + h(v,e) + b(v)$$

Trong đó:

- $f(u,v)$ là hàm đánh giá "độ gần" của đường đi từ u đến v.

- $h(u, v)$ là hàm đánh giá khoảng cách heuristic từ đỉnh u đến v.
- $h(v, e)$ là khoảng cách heuristic từ v đến điểm kết thúc.
- $b(v)$ là trị số điểm thưởng của đỉnh v. Nếu v không phải là đinhr điểm thưởng (vd: $v \equiv e$) thì $b(v) = b(e) = 0$.

Như vậy, thuật toán này (từ đây tụi em sẽ gọi là "thuật toán dẫn đường") sẽ cố gắng tìm đường đi tối ưu sao cho cân bằng giữa tổng trị số của điểm thưởng đã ăn và độ dài ước lượng của đường đi nếu đi ngang qua điểm thưởng nào đó. Còn việc tìm đường đi ngắn nhất trên từng chặng thì giao lại cho các thuật toán tương ứng là A* và GBFS.

Em sẽ không chạy bản đồ có điểm thưởng với DFS và BFS vì *về mặt lý thuyết* thì các thuật toán này không thể "nhìn trước" được bản đồ nên cũng không lên kế hoạch gì được (mặc dù nếu truyền vào thuật toán trên thì nó vẫn chạy được).

```
In [22]: # Hàm client để chạy thuật toán tìm kiếm
         def run_search_bonus(g: Graph, algorithm, heuristic, figsize=(5, 3)):
             # g: đồ thị được đọc vào
             # algorithm: con trỏ đến hàm tìm kiếm cần chạy

             path_tracker, prev_node_tracker = bonus_wrapper(g, algorithm, heuristic)
             path, total_cost = process_path_total(g, path_tracker, prev_node_tracker)
             g.visualize(path, figsize=figsize)
             print('Cost:', total_cost)
```

3.1 Bản đồ 1 (3 điểm, 8x17)

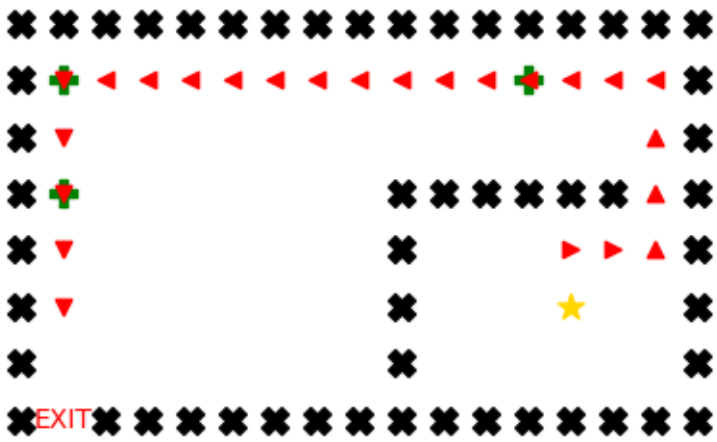
```
In [23]: g1 = Graph('testcases/bonus1.txt')

Graph initialized from maze with size 8 x 17
```

3.1.1 Greedy Best First Search (Manhattan)

```
In [24]: run_search_bonus(g1, GBFS, manhattan_heuristic)

Going from (5, 13) to (1, 1)
Found a path! Keep going...
Going from (1, 1) to (3, 1)
Found a path! Keep going...
Going from (3, 1) to (7, 1)
Found a path! Keep going...
Yay! Finished!
```




```
Starting point (x, y) = (5, 13)
Ending point (x, y) = (7, 1)
Bonus point at position (x, y) = (1, 1) with point -10
Bonus point at position (x, y) = (1, 12) with point -2
Bonus point at position (x, y) = (3, 1) with point -5
Cost: 9
```

3.1.2 Greedy Best First Search (Euclide)

```
In [25]: run_search_bonus(g1, GBFS, euclide_heuristic)
```

```
Going from (5, 13) to (1, 1)
Found a path! Keep going...
Going from (1, 1) to (3, 1)
Found a path! Keep going...
Going from (3, 1) to (7, 1)
Found a path! Keep going...
Yay! Finished!
```

```
Starting point (x, y) = (5, 13)
Ending point (x, y) = (7, 1)
Bonus point at position (x, y) = (1, 1) with point -10
Bonus point at position (x, y) = (1, 12) with point -2
Bonus point at position (x, y) = (3, 1) with point -5
Cost: 11
```

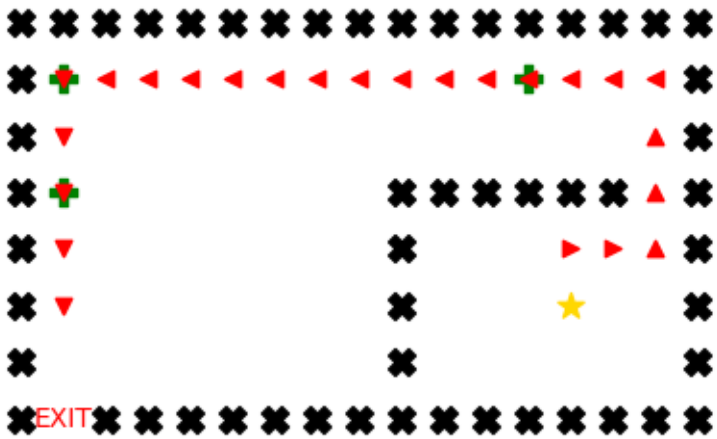
Với 2 hàm heuristic khác nhau, GBFS đã cho ra 2 kết quả khác nhau. Nhìn vào log của giải thuật dẫn đường, ta thấy nó đã chọn con đường: $S \rightarrow (1,1) \rightarrow (3,1) \rightarrow E$ (trong đó S, E là điểm bắt đầu và kết thúc).

Ở lần thứ nhất, GBFS đã "ăn may" khi đi qua được điểm thưởng (1, 12), vốn không được chọn bởi thuật toán dẫn đường. Ở lần thứ hai, do sử dụng khoảng cách Euclide nên nó có xu hướng đi đường "chéo", cho nên nó đã không "ăn may" như lần đầu tiên nữa. Lý do em nói GBFS "ăn may" là vì bản thân hàm heuristic của GBFS hoàn toàn không quan tâm đến trọng số trên đường đi nên nó không thể biết được chỗ nào thực sự là có điểm thưởng. Nó chỉ biết được nơi có điểm thưởng nhờ thuật toán dẫn đường này.

3.1.3 A* (Manhattan)

```
In [30]: run_search_bonus(g1, Astar, manhattan_heuristic)
```

```
Going from (5, 13) to (1, 1)
Found a path! Keep going...
Going from (1, 1) to (3, 1)
Found a path! Keep going...
Going from (3, 1) to (7, 1)
Found a path! Keep going...
Yay! Finished!
```




```
Starting point (x, y) = (5, 13)
Ending point (x, y) = (7, 1)
Bonus point at position (x, y) = (1, 1) with point -10
Bonus point at position (x, y) = (1, 12) with point -2
```

```
Bonus point at position (x, y) = (3, 1) with point -5
Cost: 9
```

3.1.4 A* (Euclide)

```
In [31]: run_search_bonus(g1, Astar, euclide_heuristic)
```

```
Going from (5, 13) to (1, 1)
Found a path! Keep going...
Going from (1, 1) to (3, 1)
Found a path! Keep going...
Going from (3, 1) to (7, 1)
Found a path! Keep going...
Yay! Finished!
```


```
Starting point (x, y) = (5, 13)
Ending point (x, y) = (7, 1)
Bonus point at position (x, y) = (1, 1) with point -10
Bonus point at position (x, y) = (1, 12) with point -2
Bonus point at position (x, y) = (3, 1) with point -5
Cost: 9
```

Với cả 2 hàm heuristic, A* đã chọn phương án ăn tất cả các điểm thưởng, và đương nhiên đây cũng chính là phương án tối thiểu hóa chi phí đường đi trong bản đồ này (tuy nhiên điều này có thể chỉ do trùng hợp). Hàm heuristic của A* cũng quan tâm đến trọng số của các cạnh, đó là lý do nó đã tìm được các điểm thưởng.

Lưu ý là với một số bản đồ khác, **tổng đường đi** mà nó tìm được có thể không phải là ngắn nhất vì nó còn phụ thuộc vào *sự hướng dẫn của thuật toán dẫn đường*, nên đường đi mà nó tìm được chỉ là *có vẻ là ngắn nhất* chứ không đảm bảo là ngắn nhất. Tuy nhiên, ta luôn luôn đảm bảo được rằng A* sẽ tìm được **đường đi ngắn nhất trên mỗi chặng**. Lý do là vì A* là một thuật toán *complete*, nên nó sẽ luôn tìm được đường đi ngắn nhất **giữa 2 điểm** nếu đường đi này có tồn tại.

Tổng kết cho bản đồ 1:

Thuật toán	Chi phí
GBFS (M)	9
GBFS (E)	11
A* (M)	9
A* (E)	9

3.2 Bản đồ 2 (5 điểm, 21x31)

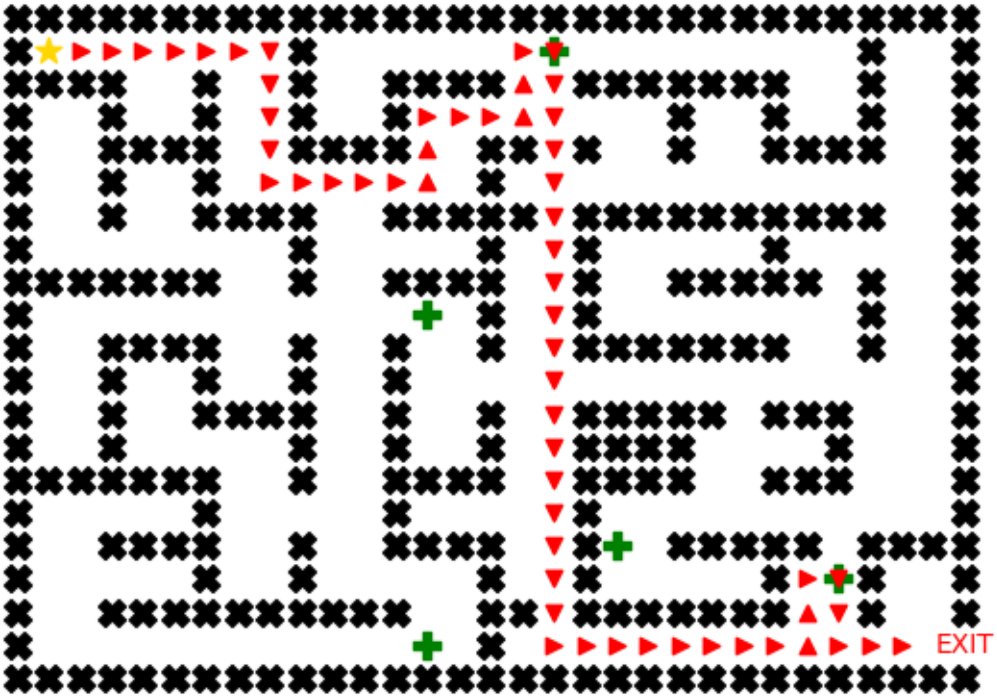
```
In [43]: g2 = Graph('testcases/bonus2.txt')
```

```
Graph initialized from maze with size 21 x 31
```

3.2.1 Greedy Best-first Search (Manhattan)

```
In [44]: run_search_bonus(g2, GBFS, manhattan_heuristic, figsize=(7, 5))
```

```
Going from (1, 1) to (1, 17)
Found a path! Keep going...
Going from (1, 17) to (9, 13)
Found a path! Keep going...
Going from (9, 13) to (19, 13)
Found a path! Keep going...
Going from (19, 13) to (19, 30)
Going from (19, 13) to (16, 19)
Going from (19, 13) to (17, 26)
Hmmm, this seems to be a dead end. Me go back then.
Going from (9, 13) to (16, 19)
Going from (9, 13) to (17, 26)
Going from (9, 13) to (19, 30)
Hmmm, this seems to be a dead end. Me go back then.
Going from (1, 17) to (16, 19)
Found a path! Keep going...
Going from (16, 19) to (17, 26)
Going from (16, 19) to (19, 30)
Hmmm, this seems to be a dead end. Me go back then.
Going from (1, 17) to (17, 26)
Found a path! Keep going...
Going from (17, 26) to (19, 30)
Found a path! Keep going...
Yay! Finished!
```



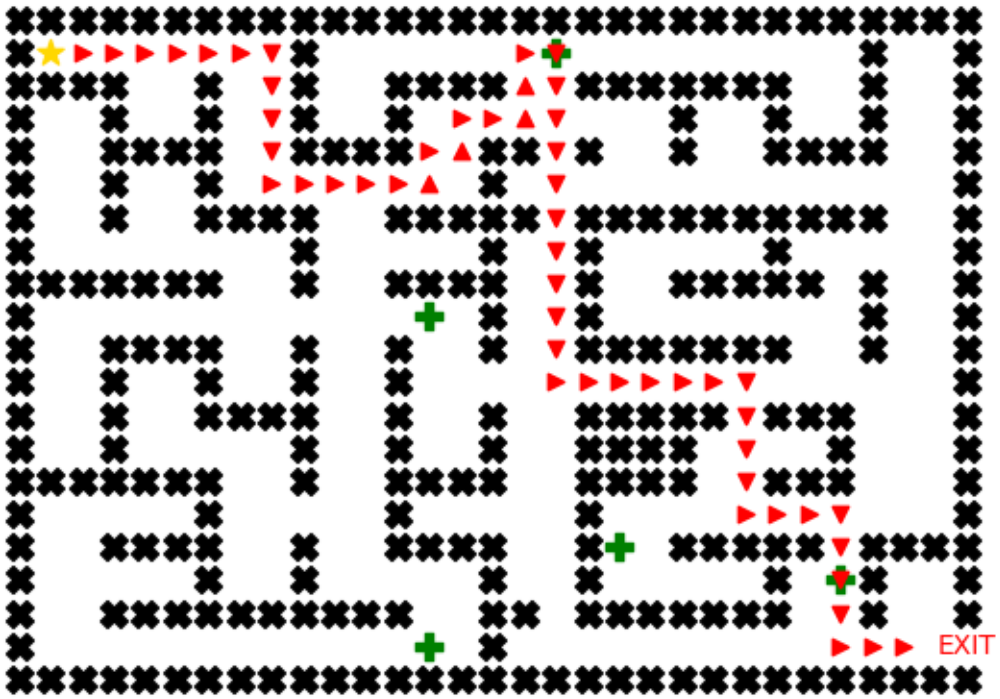

```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (19, 30)
Bonus point at position (x, y) = (19, 13) with point -10
Bonus point at position (x, y) = (17, 26) with point -1
Bonus point at position (x, y) = (9, 13) with point -15
Bonus point at position (x, y) = (16, 19) with point -5
Bonus point at position (x, y) = (1, 17) with point -20
Cost: 38
```

3.2.2 Greedy Best-first Search (Euclide)

```
In [45]: run_search_bonus(g2, GBFS, euclide_heuristic, figsize=(7, 5))
```

```
Going from (1, 1) to (1, 17)
Found a path! Keep going...
Going from (1, 17) to (9, 13)
Found a path! Keep going...
Going from (9, 13) to (16, 19)
Going from (9, 13) to (19, 13)
Found a path! Keep going...
Going from (19, 13) to (17, 26)
Going from (19, 13) to (19, 30)
Hmmm, this seems to be a dead end. Me go back then.
Going from (9, 13) to (17, 26)
Going from (9, 13) to (19, 30)
Hmmm, this seems to be a dead end. Me go back then.
```

```
Going from (1, 17) to (16, 19)
Found a path! Keep going...
Going from (16, 19) to (17, 26)
Going from (16, 19) to (19, 30)
Hmmm, this seems to be a dead end. Me go back then.
Going from (1, 17) to (17, 26)
Found a path! Keep going...
Going from (17, 26) to (19, 30)
Found a path! Keep going...
Yay! Finished!
```



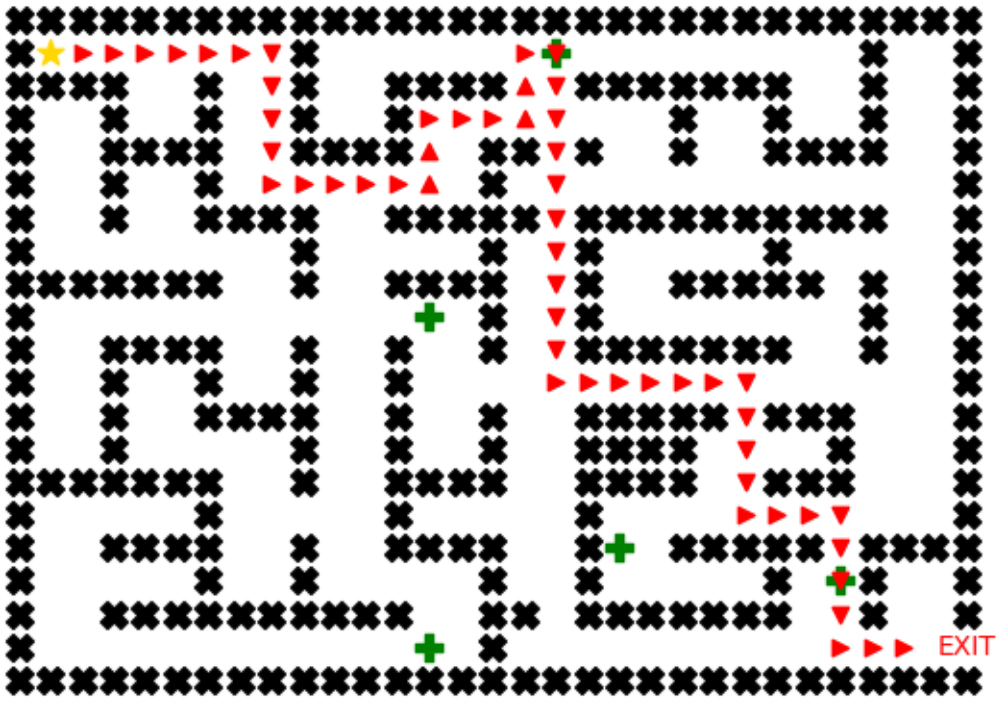

```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (19, 30)
Bonus point at position (x, y) = (19, 13) with point -10
Bonus point at position (x, y) = (17, 26) with point -1
Bonus point at position (x, y) = (9, 13) with point -15
Bonus point at position (x, y) = (16, 19) with point -5
Bonus point at position (x, y) = (1, 17) with point -20
Cost: 34
```

3.2.3 A* (Manhattan)

```
In [46]: run_search_bonus(g2, Astar, manhattan_heuristic, figsize=(7, 5))
```

```
Going from (1, 1) to (1, 17)
Found a path! Keep going...
```

```
Going from (1, 17) to (9, 13)
Found a path! Keep going...
Going from (9, 13) to (19, 13)
Found a path! Keep going...
Going from (19, 13) to (19, 30)
Going from (19, 13) to (16, 19)
Going from (19, 13) to (17, 26)
Hmmm, this seems to be a dead end. Me go back then.
Going from (9, 13) to (16, 19)
Going from (9, 13) to (17, 26)
Going from (9, 13) to (19, 30)
Hmmm, this seems to be a dead end. Me go back then.
Going from (1, 17) to (16, 19)
Found a path! Keep going...
Going from (16, 19) to (17, 26)
Going from (16, 19) to (19, 30)
Hmmm, this seems to be a dead end. Me go back then.
Going from (1, 17) to (17, 26)
Found a path! Keep going...
Going from (17, 26) to (19, 30)
Found a path! Keep going...
Yay! Finished!
```



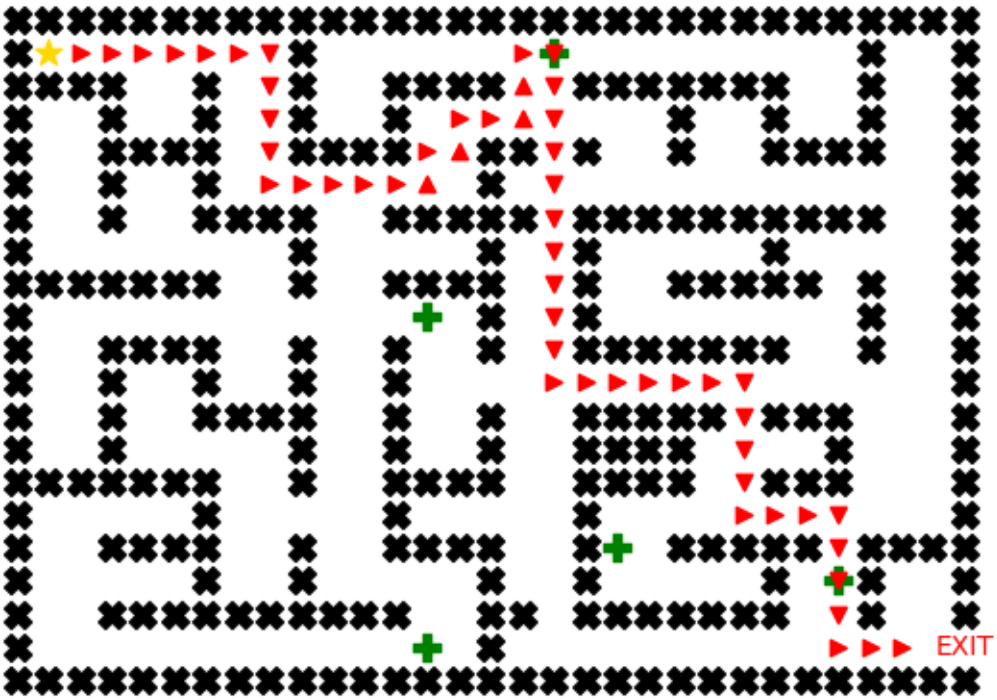

```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (19, 30)
Bonus point at position (x, y) = (19, 13) with point -10
```

```
Bonus point at position (x, y) = (17, 26) with point -1
Bonus point at position (x, y) = (9, 13) with point -15
Bonus point at position (x, y) = (16, 19) with point -5
Bonus point at position (x, y) = (1, 17) with point -20
Cost: 34
```

3.2.4 A* (Euclide)

```
In [47]: run_search_bonus(g2, Astar, euclide_heuristic, figsize=(7, 5))
```

```
Going from (1, 1) to (1, 17)
Found a path! Keep going...
Going from (1, 17) to (9, 13)
Found a path! Keep going...
Going from (9, 13) to (16, 19)
Going from (9, 13) to (19, 13)
Found a path! Keep going...
Going from (19, 13) to (17, 26)
Going from (19, 13) to (19, 30)
Hmmm, this seems to be a dead end. Me go back then.
Going from (9, 13) to (17, 26)
Going from (9, 13) to (19, 30)
Hmmm, this seems to be a dead end. Me go back then.
Going from (1, 17) to (16, 19)
Found a path! Keep going...
Going from (16, 19) to (17, 26)
Going from (16, 19) to (19, 30)
Hmmm, this seems to be a dead end. Me go back then.
Going from (1, 17) to (17, 26)
Found a path! Keep going...
Going from (17, 26) to (19, 30)
Found a path! Keep going...
Yay! Finished!
```




```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (19, 30)
Bonus point at position (x, y) = (19, 13) with point -10
Bonus point at position (x, y) = (17, 26) with point -1
Bonus point at position (x, y) = (9, 13) with point -15
Bonus point at position (x, y) = (16, 19) with point -5
Bonus point at position (x, y) = (1, 17) with point -20
Cost: 34
```

Ở cả 2 hàm heuristic, thuật toán dẫn đường đều cho cùng một kết quả. Nhưng kết quả tìm được giữa 2 thuật toán có sự khác biệt:

- GBFS hành xử khác nhau, tùy thuộc vào hàm heuristic là Euclide hay Manhattan. Với hàm heuristic là Euclide, nó đã tìm được đường đi ngắn nhất (do đường đi ngắn nhất thực sự là đi theo đường chéo, chứ không phải đi thẳng), trong khi Manhattan thì không.
- A* hành xử giống nhau với cả 2 hàm heuristic, và nó đã tìm được đường đi có vẻ là ngắn nhất trên bản đồ này.

Thuật toán	Chi phí
GBFS (M)	38
GBFS (E)	34
A* (M)	34
A* (E)	34

3.3 Bản đồ 3 (10 điểm, 31x71)

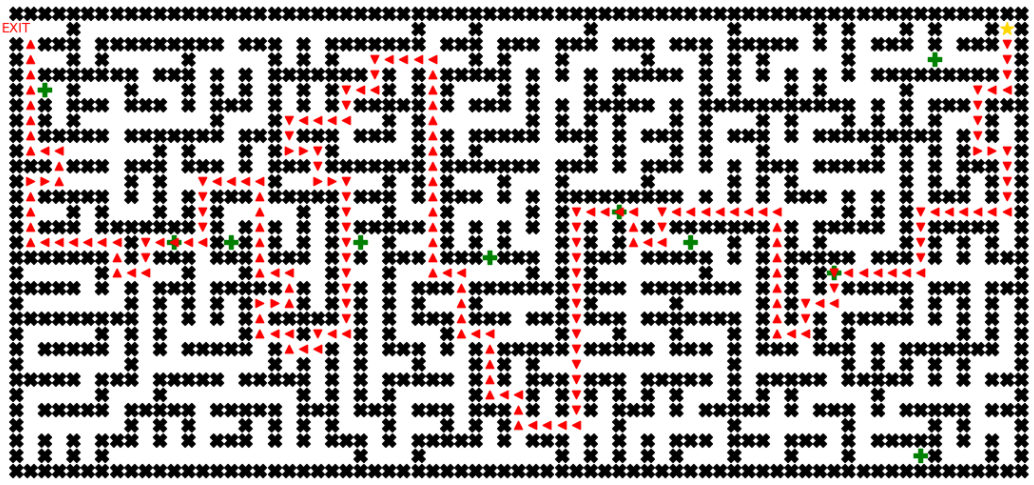
```
In [48]: g3 = Graph('testcases/bonus3.txt')

Graph initialized from maze with size 31 x 71
```

3.3.1 Greedy Best-first Search (Manhattan)

```
In [49]: run_search_bonus(g3, GBFS, manhattan_heuristic, figsize=(15, 7))

Going from (1, 69) to (5, 2)
Found a path! Keep going...
Going from (5, 2) to (1, 0)
Going from (5, 2) to (15, 11)
Going from (5, 2) to (15, 15)
Going from (5, 2) to (15, 24)
Going from (5, 2) to (13, 42)
Going from (5, 2) to (16, 33)
Going from (5, 2) to (15, 47)
Going from (5, 2) to (17, 57)
Going from (5, 2) to (3, 64)
Going from (5, 2) to (29, 63)
Hmmm, this seems to be a dead end. Me go back then.
Going from (1, 69) to (17, 57)
Found a path! Keep going...
Going from (17, 57) to (15, 11)
Found a path! Keep going...
Going from (15, 11) to (15, 15)
Going from (15, 11) to (1, 0)
Found a path! Keep going...
Yay! Finished!
```

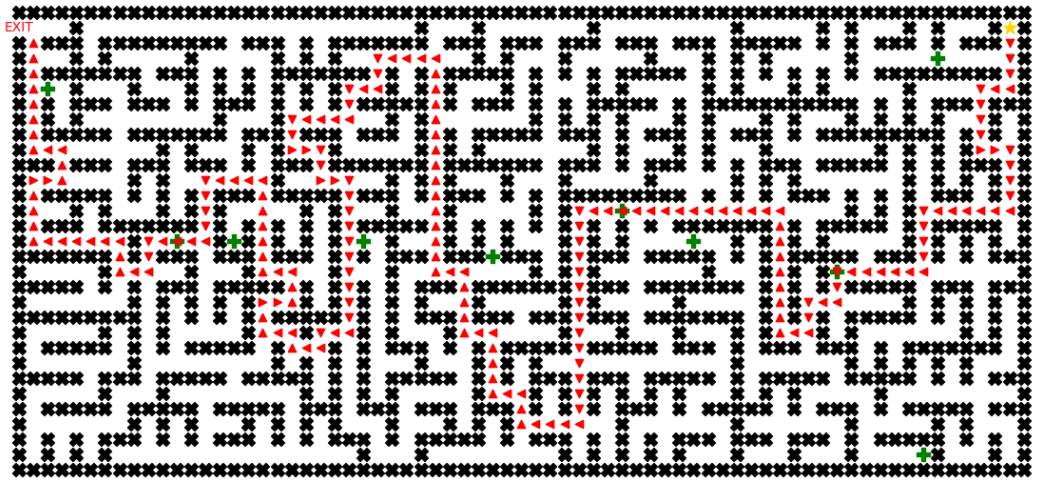


```
Starting point (x, y) = (1, 69)
Ending point (x, y) = (1, 0)
Bonus point at position (x, y) = (15, 15) with point -8
Bonus point at position (x, y) = (3, 64) with point -2
Bonus point at position (x, y) = (15, 47) with point -12
Bonus point at position (x, y) = (15, 24) with point -20
Bonus point at position (x, y) = (15, 11) with point -30
Bonus point at position (x, y) = (29, 63) with point -15
Bonus point at position (x, y) = (5, 2) with point -14
Bonus point at position (x, y) = (13, 42) with point -17
Bonus point at position (x, y) = (17, 57) with point -35
Bonus point at position (x, y) = (16, 33) with point -2
Cost: 129
```

3.3.2 Greedy Best-first Search (Euclide)

```
In [50]: run_search_bonus(g3, GBFS, euclide_heuristic, figsize=(15, 7))
```

```
Going from (1, 69) to (17, 57)
Found a path! Keep going...
Going from (17, 57) to (15, 11)
Found a path! Keep going...
Going from (15, 11) to (5, 2)
Found a path! Keep going...
Going from (5, 2) to (1, 0)
Going from (5, 2) to (15, 15)
Going from (5, 2) to (15, 24)
Going from (5, 2) to (16, 33)
Going from (5, 2) to (13, 42)
Going from (5, 2) to (15, 47)
Going from (5, 2) to (29, 63)
Going from (5, 2) to (3, 64)
Hmmm, this seems to be a dead end. Me go back then.
Going from (15, 11) to (15, 15)
Going from (15, 11) to (1, 0)
Found a path! Keep going...
Yay! Finished!
```




```
Starting point (x, y) = (1, 69)
Ending point (x, y) = (1, 0)
Bonus point at position (x, y) = (15, 15) with point -8
Bonus point at position (x, y) = (3, 64) with point -2
Bonus point at position (x, y) = (15, 47) with point -12
Bonus point at position (x, y) = (15, 24) with point -20
Bonus point at position (x, y) = (15, 11) with point -30
Bonus point at position (x, y) = (29, 63) with point -15
Bonus point at position (x, y) = (5, 2) with point -14
Bonus point at position (x, y) = (13, 42) with point -17
Bonus point at position (x, y) = (17, 57) with point -35
Bonus point at position (x, y) = (16, 33) with point -2
Cost: 125
```

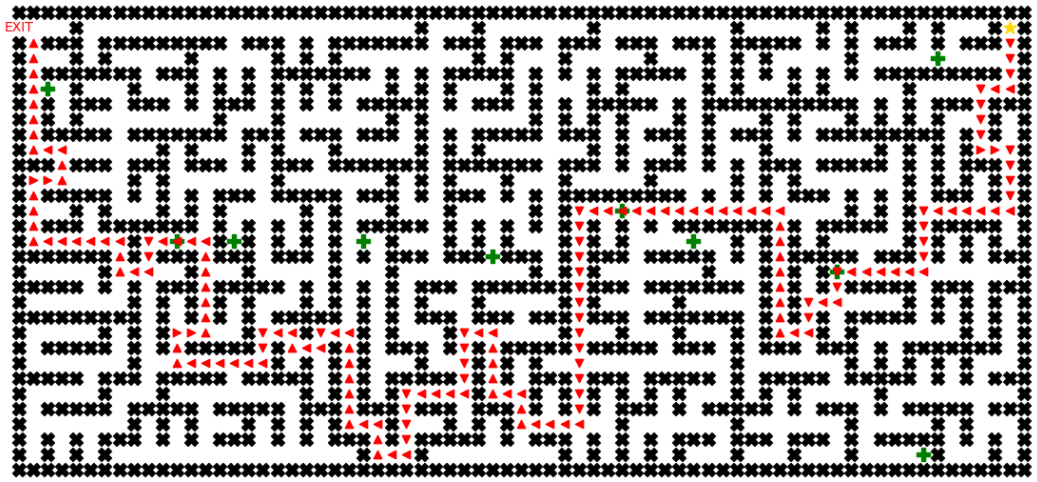
Với heuristic là Euclide, nó đã chọn một đường đi khác và lần này nó đã không ăn may được như lần trước nữa. Trên đường đi nó không vô tình ăn được điểm thưởng nào cả.

3.3.3 A* (Euclide)

```
In [51]: run_search_bonus(g3, Astar, euclide_heuristic, figsize=(15, 7))
```

```
Going from (1, 69) to (17, 57)
Found a path! Keep going...
Going from (17, 57) to (15, 11)
Found a path! Keep going...
Going from (15, 11) to (5, 2)
Found a path! Keep going...
Going from (5, 2) to (1, 0)
Going from (5, 2) to (15, 15)
Going from (5, 2) to (15, 24)
Going from (5, 2) to (16, 33)
Going from (5, 2) to (13, 42)
Going from (5, 2) to (15, 47)
Going from (5, 2) to (29, 63)
```

```
Going from (5, 2) to (3, 64)
Hmmm, this seems to be a dead end. Me go back then.
Going from (15, 11) to (15, 15)
Found a path! Keep going...
Going from (15, 15) to (15, 24)
Going from (15, 15) to (1, 0)
Going from (15, 15) to (16, 33)
Going from (15, 15) to (13, 42)
Going from (15, 15) to (15, 47)
Going from (15, 15) to (29, 63)
Going from (15, 15) to (3, 64)
Hmmm, this seems to be a dead end. Me go back then.
Going from (15, 11) to (1, 0)
Found a path! Keep going...
Yay! Finished!
```




```
Starting point (x, y) = (1, 69)
Ending point (x, y) = (1, 0)
Bonus point at position (x, y) = (15, 15) with point -8
Bonus point at position (x, y) = (3, 64) with point -2
Bonus point at position (x, y) = (15, 47) with point -12
Bonus point at position (x, y) = (15, 24) with point -20
Bonus point at position (x, y) = (15, 11) with point -30
Bonus point at position (x, y) = (29, 63) with point -15
Bonus point at position (x, y) = (5, 2) with point -14
Bonus point at position (x, y) = (13, 42) with point -17
Bonus point at position (x, y) = (17, 57) with point -35
Bonus point at position (x, y) = (16, 33) with point -2
Cost: 93
```

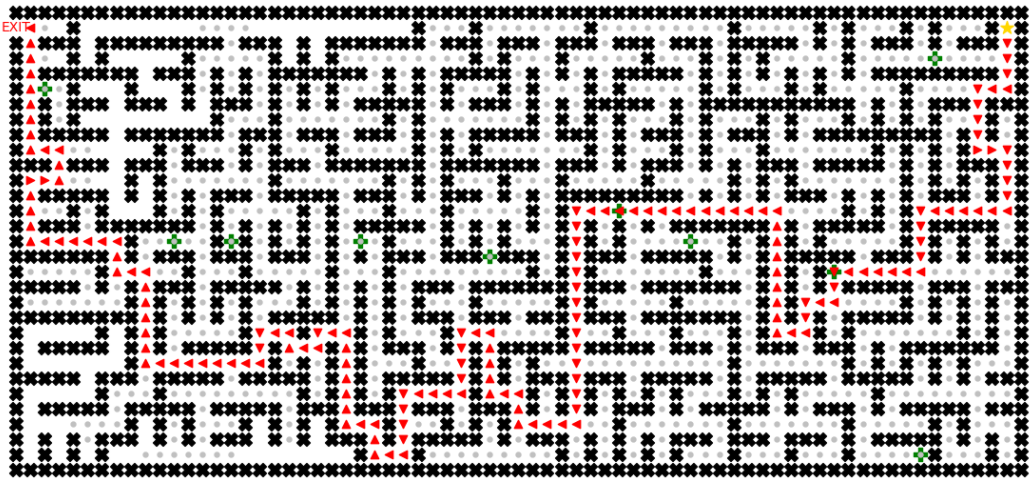
Nhận xét về hành vi của 2 thuật toán và thuật toán dẫn đường tương tự với các bản đồ trước. GBFS đã không tìm được đường đi ngắn nhất với cả 2 hàm heuristic. A* đã tìm được đường đi *có vẻ* là ngắn nhất.

Thuật toán	Chi phí
GBFS (M)	129
GBFS (E)	125
A* (E)	93

3.3.4 Bonus: A* mà không có thuật toán dẫn đường

Ở ví dụ này tụi em sẽ chạy lại A* với bản đồ 3 (**có** điểm thưởng) mà không có thuật toán dẫn đường để xem thử nếu không có thuật toán dẫn đường, A* sẽ hành xử như thế nào. Lý do mà tụi em muốn thử trường hợp này là do thực tế, A* cũng có thể nhìn thấy được các điểm thưởng (do nó có quan tâm đến trọng số của các cạnh).

```
In [56]: g3.clear()
         Astar(g3, euclide_heuristic)
         visited, path, cost = g3.get_visited()
         g3.visualize(path, visited=visited, figsize=(15, 7))
         print('Cost:', cost)
```




```
Starting point (x, y) = (1, 69)
Ending point (x, y) = (1, 0)
Bonus point at position (x, y) = (15, 15) with point -8
Bonus point at position (x, y) = (3, 64) with point -2
Bonus point at position (x, y) = (15, 47) with point -12
Bonus point at position (x, y) = (15, 24) with point -20
Bonus point at position (x, y) = (15, 11) with point -30
Bonus point at position (x, y) = (29, 63) with point -15
Bonus point at position (x, y) = (5, 2) with point -14
Bonus point at position (x, y) = (13, 42) with point -17
Bonus point at position (x, y) = (17, 57) with point -35
Bonus point at position (x, y) = (16, 33) with point -2
Cost: 115
```

Như vậy A* tìm được phương án tối ưu để ăn các điểm thưởng là nhờ vào thuật toán dẫn đường mà tụi em đã viết.

Thuật toán	Chi phí
A* (có dẫn đường)	93
A* (thông thường)	115

Tuy nhiên phương pháp này có một vấn đề mà tụi em sẽ trình bày trong phần tổng kết các thuật toán.

Chương 4

Bản đồ có cổng dịch chuyển

Ý tưởng: Nếu trong mê cung có cánh cổng đi từ điểm A đến B thì trên đồ thị chúng ta sẽ thêm một cạnh nối từ A đến B và gán trọng số cạnh đó bằng 1. Bằng cách này các thuật toán sẽ tự tìm được đường đi từ START đến END mà đi qua được cổng dịch chuyển.

Phương pháp này không thực sự chỉnh sửa hàm heuristic của thuật toán nên nó không chính xác lắm. Với phương pháp này, ta thấy khoảng cách heuristic giữa cánh cổng đi vào và điểm `EXIT` vẫn bằng với trường hợp khi không có cánh cổng này. Nếu muốn "thực sự" thêm cánh cổng vào bản đồ, ta có thể định nghĩa lại hàm heuristic $h'(x)$ như sau:

$$h'(x) = h(x) - t(x)$$

Trong đó $t(x)$ là khoảng cách giữa điểm đi vào cánh cổng và điểm đi ra khỏi cánh cổng *nếu như* x là điểm đi vào của một "đường hầm" teleport nào đó. Nếu x không phải cánh cổng thì $t(x) = 0$.

Tuy nhiên hàm heuristic này vẫn có một hạn chế là nếu ta tính khoảng cách giữa 2 điểm mà vô tình khoảng cách này có đi qua 2 cánh cổng (trường hợp heuristic là Manhattan sẽ gặp nhiều) thì nó vẫn chưa đủ. Tuy nhiên để đơn giản thì tụi em chỉ implement tới mức đó thôi vì thời gian cũng không còn nhiều.

4.1 Định nghĩa các hàm cần thiết

```
In [1]: %cd ../../source
        import matplotlib.pyplot as plt
        from teacher_utils import visualize_maze
        from Graph import Graph
        from searching_algorithms import *
        from heuristic_func import euclide_heuristic, manhattan_heuristic
```

```
/home/hiraki/source/IntroAI/Project01/source
```

```
In [2]: # Hàm để vẽ các cánh cổng dịch chuyển
        def draw_teleports(fig, port_list):
            ax = fig.gca()
            for port in port_list:
                port_in = port[0]
                port_out = port[1]
                port_color = port[2]
                ax.scatter(port_in[1], -port_in[0], marker='D', s=50, color=port_color)
                ax.scatter(port_out[1], -port_out[0], marker='o', s=50, color=port_color)
```

```
In [3]: # Hàm client để chạy thuật toán tìm kiếm
        def run_search_nobonus(g: Graph, algorithm, heuristic=None, figsize=(5, 3), ports=[]):
```

```
            # g: đồ thị được đọc vào
            # algorithm: con trỏ đến hàm tìm kiếm cần chạy
            g.clear()
            algorithm(g, heuristic)
            visited, path, cost = g.get_visited()
            print('Cost:', cost)
            fig = g.visualize(path, visited=visited, figsize=figsize, dont_show=True)
            draw_teleports(fig, ports)
```

```
In [4]: # Hàm định nghĩa heuristic h'
        def tunnel_wrapper(hf, coord1, coord2, port_list):
            # Nếu x là đường đi vào hầm
            for pair in port_list:
                if coord1 == pair[0]:
                    return hf(pair[1], coord2)
            # Trường hợp thông thường
            return hf(coord1, coord2)
```

```
In [5]: def create_tunnels(g: Graph, port_list):
            # Đường vào chỉ có kề với đường ra, không kề với các điểm kác xung quanh nữa
            for port in port_list:
                g.node_list[port[0]].neighbors.clear()
                g.node_list[port[0]].neighbors.append({
                    'coord': port[1],
                    'cost': 1,
                    'direction': ''
                })
```

4.2 Ví dụ với bản đồ không điểm thưởng số 4

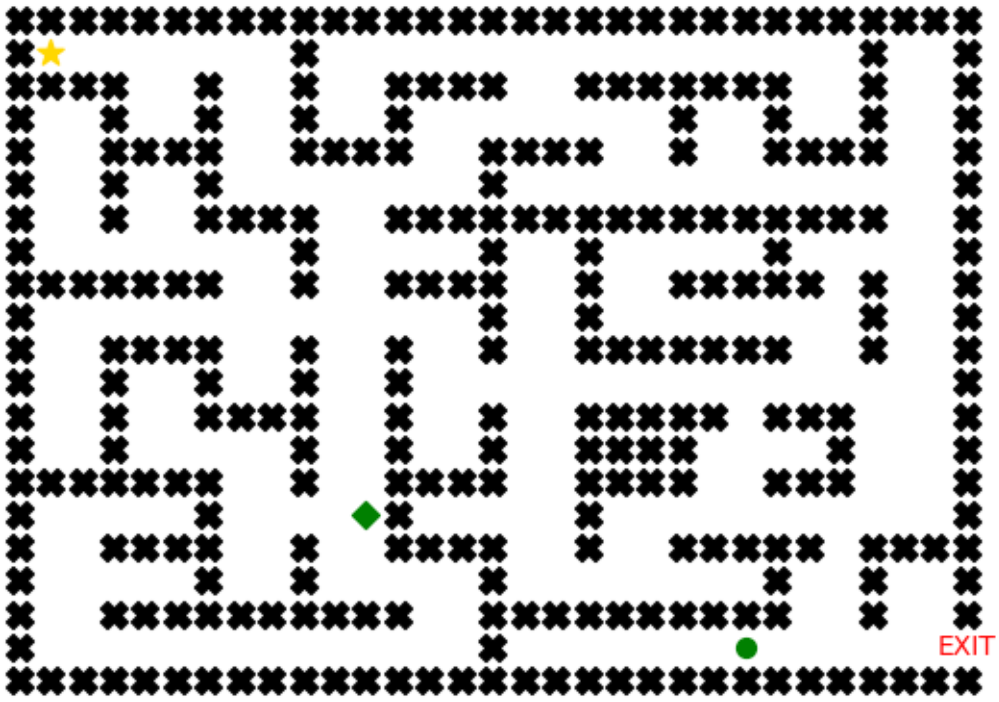
Giả sử em thêm một cổng dịch chuyển từ vị trí (15, 11) đến (19, 23) vào bản đồ không điểm thưởng số 4.

```
In [6]: g = Graph('testcases/nobonus4.txt')
```

```
Graph initialized from maze with size 21 x 31
```

```
In [7]: port_list = [((15, 11), (19, 23), 'green')]
        fig = g.visualize(figsize=(7, 5), dont_show=True)
        draw_teleports(fig, port_list)
```

```
Starting point (x, y) = (1, 1)
Ending point (x, y) = (19, 30)
```



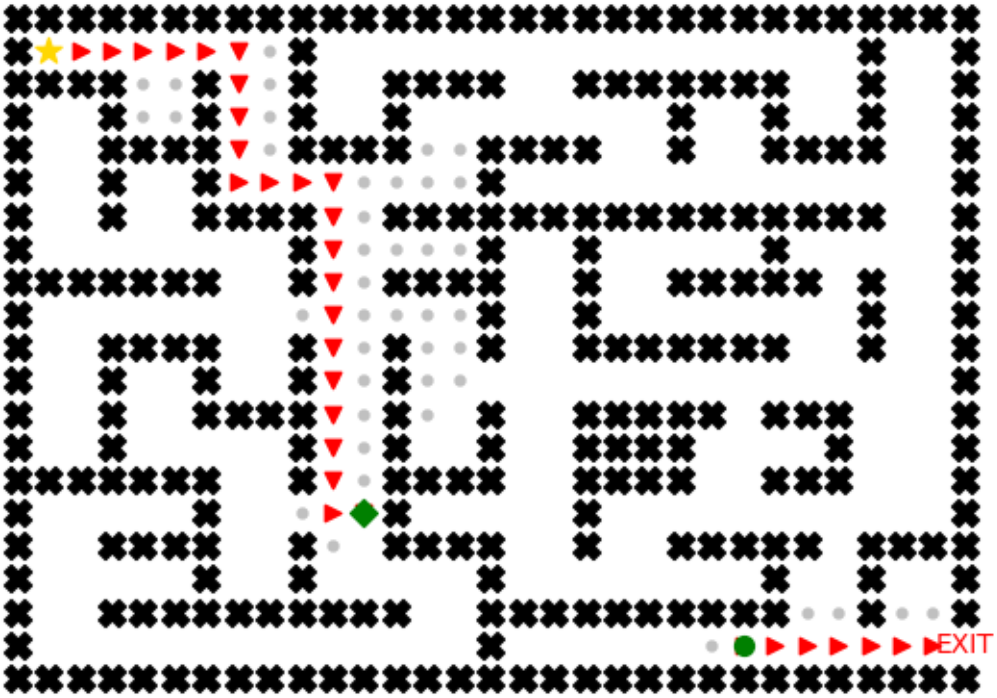

```
In [8]: # Tạo một cạnh nối từ (15, 11) -> (19, 23)
        create_tunnels(g, port_list)
```

```
In [9]: # Định nghĩa hàm heuristic Manhattan có xử lý cánh cổng
        tunnelled_manhattan = \
            lambda coord1, coord2: tunnel_wrapper(manhattan_heuristic, coord1, coord2, port_list)
```

4.2.1 A*

```
In [10]: run_search_nobonus(g, Astar,
                            tunnelled_manhattan, (7, 5), ports=port_list)
```

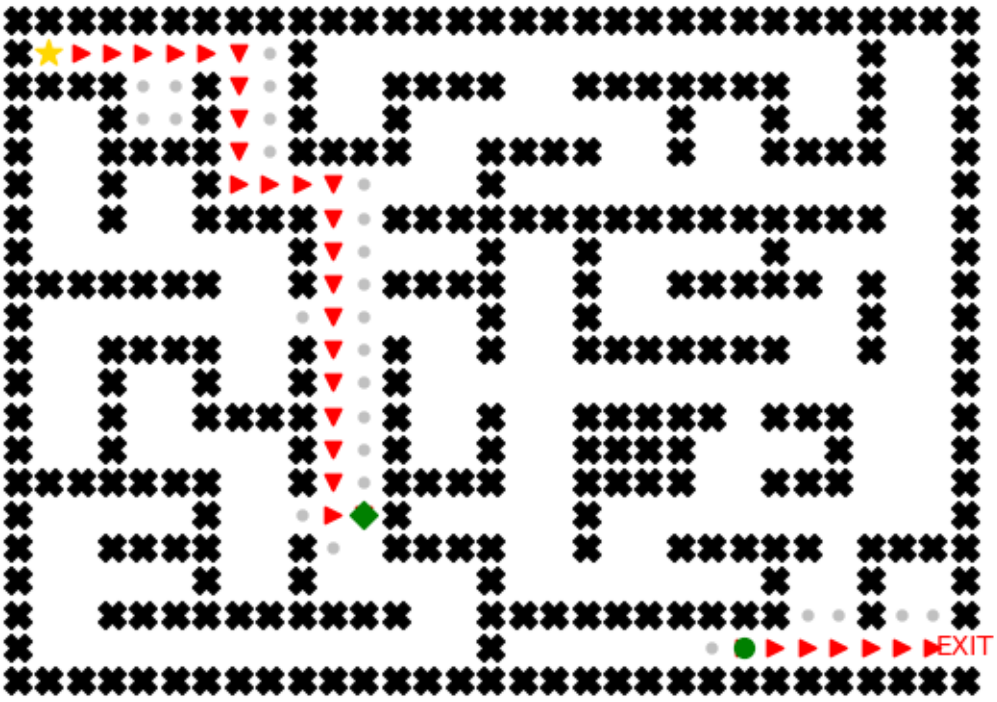
```
Cost: 32
Starting point (x, y) = (1, 1)
Ending point (x, y) = (19, 30)
```



4.2.2 Greedy Best-first Search

```
In [11]: run_search_nobonus(g, GBFS,
                            tunnelled_manhattan, (7, 5), ports=port_list)
```

```
Cost: 32
Starting point (x, y) = (1, 1)
Ending point (x, y) = (19, 30)
```



Cả 2 thuật toán đều đã tìm được cánh cổng trong tất cả các . Với 2 thuật toán tìm kiếm mù còn lại cũng tương tự.

Cần lưu ý là với hàm heuristic đã chọn thì cánh cổng chỉ thực sự phát huy tác dụng nếu nó vô tình nằm trên con đường duyệt của thuật toán (vì khi "dẫm chân" lên cánh cổng nó mới biết cánh cổng có tồn tại). Ngoài ra nó còn phụ thuộc vào chiến lược heuristic của từng thuật toán. Nếu muốn hoàn chỉnh ta có thể xét riêng hàng/cột thay vì xét hết tọa độ của điểm đang xét và tọa độ của đường vào cánh cổng. Tuy nhiên do thời gian không còn nhiều nên tụi em chỉ muốn trình bày proof-of-concept đại khái như vậy thôi ạ. Vấn đề này sẽ được em trình bày trong 2 ví dụ dưới.

4.3 Ví dụ với bản đồ không điểm thưởng số 3 (TH1)

4.3.1 Greedy Best-first Search

```
In [12]: g = Graph('testcases/nobonus3.txt')
```

```
Graph initialized from maze with size 10 x 32
```

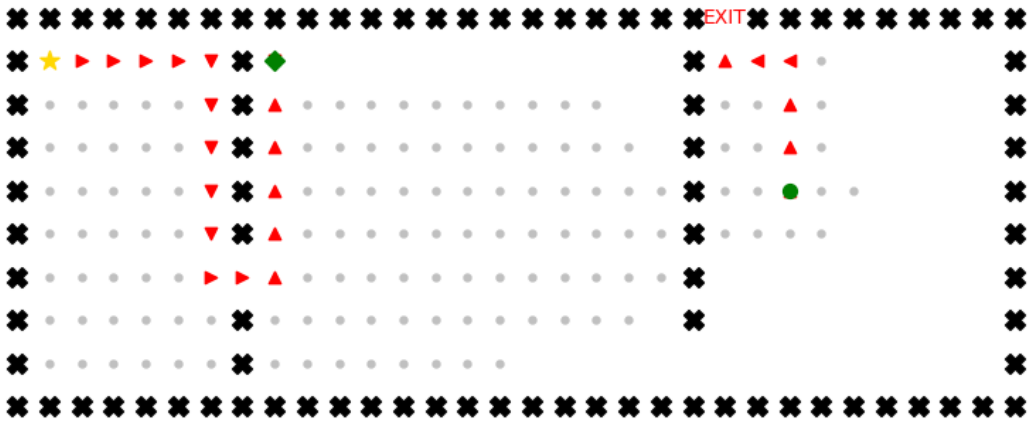
```
In [13]: port_list = [((1, 8), (4, 24), 'green')]
         create_tunnels(g, port_list)
```

```
In [14]: # Định nghĩa hàm heuristic Euclide có xử lý cánh cổng
         tunnelled_euclide = \
             lambda coord1, coord2: tunnel_wrapper(euclide_heuristic, coord1, coord2, port_list)
```

4.3.2 A*

```
In [15]: run_search_nobonus(g, Astar,
                            tunnelled_euclide, (10, 4), ports=port_list)
```

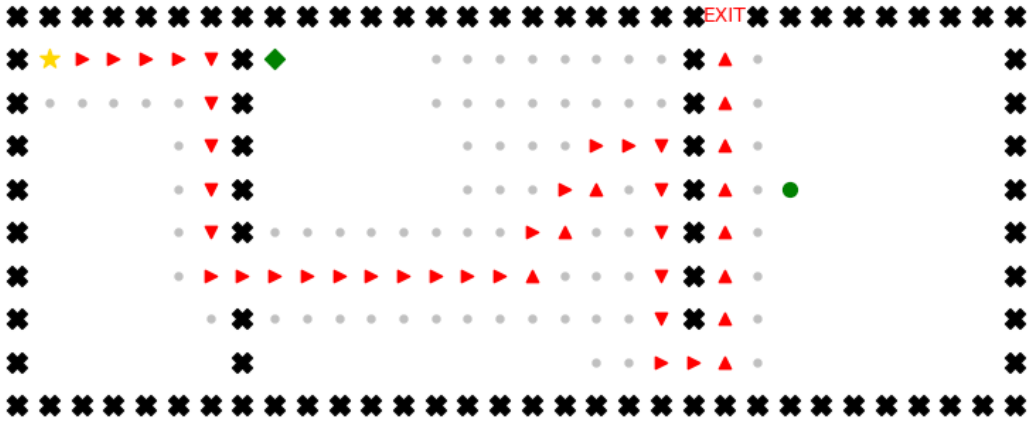
```
Cost: 24
Starting point (x, y) = (1, 1)
Ending point (x, y) = (0, 22)
```



4.3.3 Greedy Best-first Search

```
In [16]: run_search_nobonus(g, GBFS,
                            tunnelled_euclide, (10, 4), ports=port_list)
```

```
Cost: 42
Starting point (x, y) = (1, 1)
Ending point (x, y) = (0, 22)
```



Ở ví dụ này ta đã thấy được việc có tìm được cánh cổng hay không còn phụ thuộc vào chiến thuật heuristic của thuật toán. Do GBFS với heuristic là Euclide đã không quét được đường vào của cánh cổng nên nó đã không tìm được cánh cổng này.

4.4 Ví dụ với bản đồ không điểm thưởng số 4 (TH2)

4.4.1 Greedy Best-first Search

```
In [22]: g = Graph('testcases/nobonus3.txt')
```

```
Graph initialized from maze with size 10 x 32
```

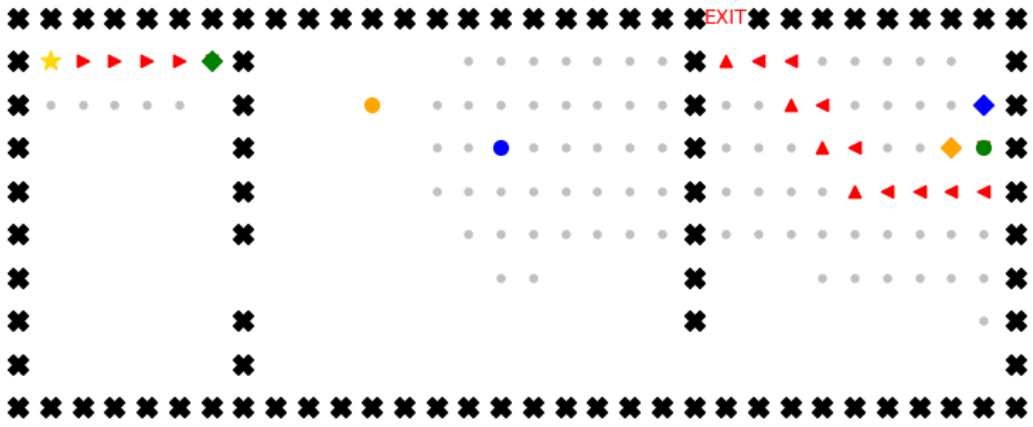
```
In [23]: port_list = [((1, 6), (3, 30), 'green'),
                      ((2, 30), (3, 15), 'blue'),
                      ((3, 29), (2, 11), 'orange')]
         create_tunnels(g, port_list)
```

```
In [24]: # Định nghĩa hàm heuristic Euclide có xử lý cánh cổng
         tunnelled_euclide = \
             lambda coord1, coord2: tunnel_wrapper(euclide_heuristic, coord1, coord2, port_list)
```

4.4.2 A*

```
In [25]: run_search_nobonus(g, Astar,
                            tunnelled_euclide, (10, 4), ports=port_list)
```

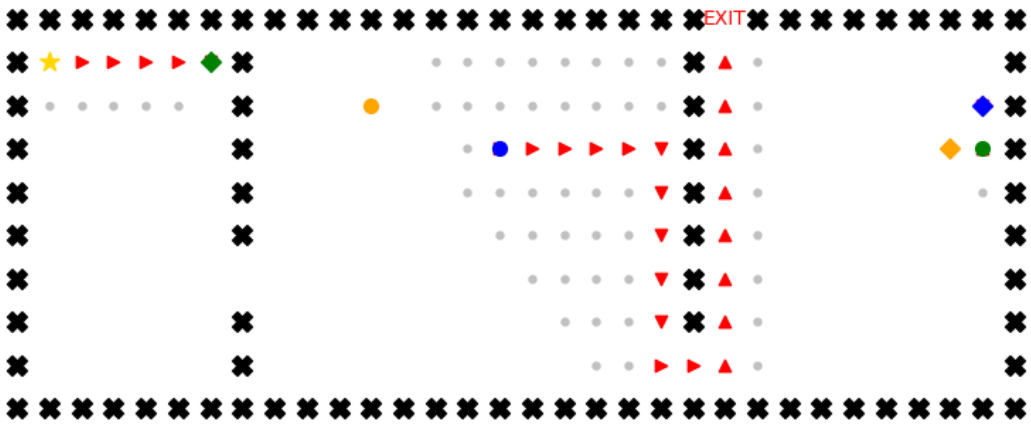
```
Cost: 19
Starting point (x, y) = (1, 1)
Ending point (x, y) = (0, 22)
```



4.4.3 Greedy Best-first Search

```
In [26]: run_search_nobonus(g, GBFS,
                            tunnelled_euclide, (10, 4), ports=port_list)
```

```
Cost: 28
Starting point (x, y) = (1, 1)
Ending point (x, y) = (0, 22)
```



GBFS một lần nữa đã bị "lừa" khi cánh cổng bị "bao vây" như trên. Do đường ra của cánh cổng xanh dương sẽ gần hơn là điểm ngay phía dưới của đường ra của cánh cổng xanh lá nên nó đã chọn phương án đi vào trong cánh cổng xanh dương.

Chương 5

Đánh giá các thuật toán

Tụi em nghĩ rằng tụi em không cần trình bày lại tất cả các đặc điểm của từng thuật toán, vì trong đề không có nói rõ ràng lắm về mục này, nên ở đây tụi em chỉ nên những nhận xét của bản thân kết hợp với các đặc điểm được nêu trong tài liệu khi quan sát kết quả output của các thuật toán tìm kiếm.

5.1 Trường hợp không điểm thưởng

- **Depth-First Search**: luôn luôn cố gắng mở rộng ô sâu nhất có thể (fringe là stack). Nếu bản đồ thông thoáng, nó sẽ chạy qua chạy lại trên bản đồ (như ở các bản đồ không điểm thưởng số 1, 2, 3). Lúc này nó giống như là *"Drunk-First Search"*. DFS chỉ có thể tìm được đường đi chứ không phải là đường đi ngắn nhất. Nếu trên đồ thị có chu trình thì có khả năng nó sẽ bị lọt vào một vòng lặp không hồi kết, và thuật toán sẽ không bao giờ dừng, do đó DFS là không hoàn chỉnh. Ngoài ra, nó cũng không tối ưu (do nó không quan tâm đến trọng số trên đường đi).
- **Breadth-First Search**: luôn luôn cố gắng mở rộng ô nông nhất có thể (fringe là queue). BFS luôn tìm được đường đi nếu đường đi này có tồn tại (có nghĩa cây tìm kiếm có chiều sâu hữu hạn), do đó nó là *hoàn chỉnh*. Nếu chi phí đường đi là bằng nhau, nó sẽ luôn tìm được đường đi ngắn nhất (lúc này nó trở thành một trường hợp đặc biệt của UCS, và lúc này nó sẽ vừa *hoàn chỉnh và tối ưu*). Nếu bản đồ trống nhiều và khoảng cách giữa điểm đầu và điểm đích xa, nó sẽ phải mở rộng rất nhiều ô. Trong hầu hết các trường hợp đã trình bày, BFS luôn luôn duyệt tất cả các ô trong bản đồ.

2 thuật toán trên chỉ có thể "mò" đường đi đến đích mà không có được cái nhìn tổng quan về bản đồ.

- **Greedy Best-First Search**: luôn luôn mở rộng ô mà có trị số heuristic thấp nhất, là vị trí mà nó tin là gần đích nhất (fringe là priority queue). GBFS là không hoàn chỉnh cũng như tối ưu, đặc biệt nếu hàm heuristic được chọn không tốt, nó sẽ đi sai hướng rất nhiều, đi vào các ô mà nó tin rằng là gần đích nhưng thực ra không phải. Nhìn chung hành vi của GBFS khá là khó đoán. Nhưng GBFS cũng có tốc độ thực thi khá nhanh vì nó không mở rộng quá nhiều ô.
- **A***: luôn luôn mở rộng ô mà có trị số heuristic thấp nhất, là vị trí mà nó tin là có* tổng chi phí* ngắn nhất (fringe là priority queue), với tổng chi phí là tổng chi phí trên toàn bộ đường đi từ điểm đầu đến điểm kết thúc. A* là sự kết hợp tuyệt vời giữa UCS và Greedy. A* vừa hoàn chỉnh vừa tối ưu, có nghĩa là nó luôn luôn tìm được đường đi ngắn nhất nếu có tồn tại, nếu hàm heuristic được chọn là phù hợp.

5.2 Trường hợp có điểm thưởng

Với ý tưởng của nhóm thì các điểm thưởng được biểu diễn bằng các cạnh có trọng số âm (do đó A* sẽ biết được đường đi nào là ngắn hơn). Tuy nhiên, A* lại không được sinh ra cho các đồ thị có trọng số âm. Lúc này heuristic của A* sẽ trở nên không nhất quán, vì nếu có cạnh có trọng số âm, sẽ tồn tại một đường đi làm

$f(n)$ giảm. Lúc này A* đột nhiên trở nên không tối ưu (!!), do đó thuật toán trong trường hợp này không còn là A* nữa (??). Tuy nhiên hiện tại thì tụi em chưa nghĩ ra được phương pháp để khắc phục điều này.

Chương 6

Tài liệu tham khảo

1. Slide bài giảng và lecture note trên Moodle.
2. From an ASCII Maze to an Adjacency List, https://mazes.readthedocs.io/en/latest/middle.html
3. A* search algorithm, https://en.wikipedia.org/wiki/A*_search_algorithm
4. Easy A star Pathfinding https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2