

**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN  
ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**



## **BÁO CÁO ĐỒ ÁN**

**MÔN HỌC: CƠ SỞ TRÍ TUỆ NHÂN TẠO  
LỚP: CQ2017/4**

**CHỦ ĐỀ: ROBOT TÌM ĐƯỜNG**

**Giảng viên hướng dẫn: Lê Ngọc Thành  
Nguyễn Ngọc Thảo**

**Thành viên nhóm:**

1712358: Nguyễn Minh Đức

1712379: Đặng Thành Duy

1712403: Nguyễn Thành Giang



## **Mục lục**

I.	Tổ chức thiết kế đồ án & đánh giá mức độ hoàn thành.....	3
II.	Chi tiết các thuật toán và quá trình chạy thử.....	7
III.	Các nguồn tài liệu tham khảo.....	31



# I. Tổ chức thiết kế đồ án & đánh giá mức độ hoàn thành

Mức độ hoàn thành:

Mức 1	100%
Mức 2	100%
Mức 3	100%
Mức 4	0%
Mức 5	0%

Thư viện đã dùng:

Module	Dùng để
matplotlib	Thể hiện đồ họa
numpy	Khởi tạo các mảng nhiều chiều
time	Đo thời gian chạy của thuật toán
heapq	Triển khai 1 hàng đợi ưu tiên (tăng tốc thuật toán)
itertools	Dùng hàm permutations (tạo các hoán vị ở mức 3)

Tổ chức chương trình:

Ở mỗi mức, chương trình đều được viết thành file riêng để thực thi riêng biệt.

Mức	Thuật toán	File chứa hàm main thực thi	Các hàm thuật toán
Mức 1	A*	astar.py	A_star_search, heuristic_search, shortest_h
Mức 2	A*	astar.py	A_star_search, heuristic_search, shortest_h
	Best first search	bestfirstsearch.py	bestfirstsearch, shortest_h
	Dijkstra	dijkstra.py	dijkstra, heuristic_search, zero

Mức 3	Nhóm em tự thiết kế	level3.py	lv3_search, find_all_path, find_node, heuristic_search
-------	---------------------	-----------	---

Các hàm thuật toán chính của cả 3 mức đều được viết trong file ‘graph.py’. File ‘graph.py’ này đóng vai trò như 1 thư viện cung cấp các hàm cho các file run ở trên.

Tất cả đường đi nước bước của các hàm thuật toán đều được ghi lại trong biến recursive\_display. Vì vậy chương trình mới có khả năng “trình chiếu” lại quá trình tìm kiếm.

#### **Chức năng các hàm trong ‘graph.py’:**

init_world	Xử lý input
find_edge	Dựng các cạnh từ các đỉnh của đa giác
shortest_h	Hàm heuristic trả về khoảng cách ngắn nhất giữa 2 điểm
zero	Trả về không
display_world	Thể hiện đồ họa mô tả bởi 1 ma trận world
display_process	Dựa vào biến trả về từ hàm thuật toán, thay đổi đồ họa liên tục để thể hiện quá trình tìm kiếm
display_process_lv3	Thể hiện quá trình tìm kiếm cho mức 3
check	Trả về True nếu 2 list 1 chiều giống nhau
heuristic_search	Tìm kiếm đường đi ngắn nhất dựa trên độ ưu tiên của 1 hàm $f = h + g$ , trong đó h là hàm heuristic, g là cost từ node xuất phát tới node đang xét. Ở đây nếu $h=0$ thì ta có thuật toán dijkstra, $h = \text{shortest\_h}$ thì ta có thuật toán A*.
A_star_search	Tìm đường đi ngắn nhất theo thuật toán A*
dijkstra	Tìm đường đi ngắn nhất theo thuật toán dijkstra
bestfirstsearch	Tìm đường đi ngắn nhất theo thuật toán bestfirstsearch
find_node	trả về vị trí đứng của node A trong list các middle_node

find_all_paths	Tìm tất cả các đường đi trực tiếp ngắn nhất giữa S,G và các middle_node, chỉ trừ đường đi trực tiếp từ S tới G
lv3_search	Tìm đường đi ngắn nhất giữa S,G thông qua các middle_node

Quá trình thực thi:

- Xử lý input:

Gọi hàm `init_world()`, hàm này xử lý file input, đọc giới hạn bản đồ, tọa độ các điểm S, G sau đó dựng các đa giác (theo từng cạnh một thông qua hàm `find_edge()`) rồi trả về 2 ma trận thể giới. Một là world dùng cho đồ họa, hai là ma trận `emap` dùng cho các hàm thuật toán.

- Tìm kiếm đường đi:

Gọi hàm thuật toán tìm đường đi, đồng thời đặt timer để đo thời gian chạy. Lấy ra chi phí, chi tiết đường đi và quá trình tìm kiếm.

- Thể hiện kết quả:

In ra console kết quả tìm kiếm, chi phí đường đi, thời gian chạy, số node đã đi qua. Sau đó gọi hàm `display_process()` để thể hiện chi tiết quá trình tìm kiếm (nhờ vào biến tên `recursive_display` được trả về khi gọi hàm thuật toán).

### Thể hiện kết quả:

Các kết quả được in ra ở console có dạng như sau

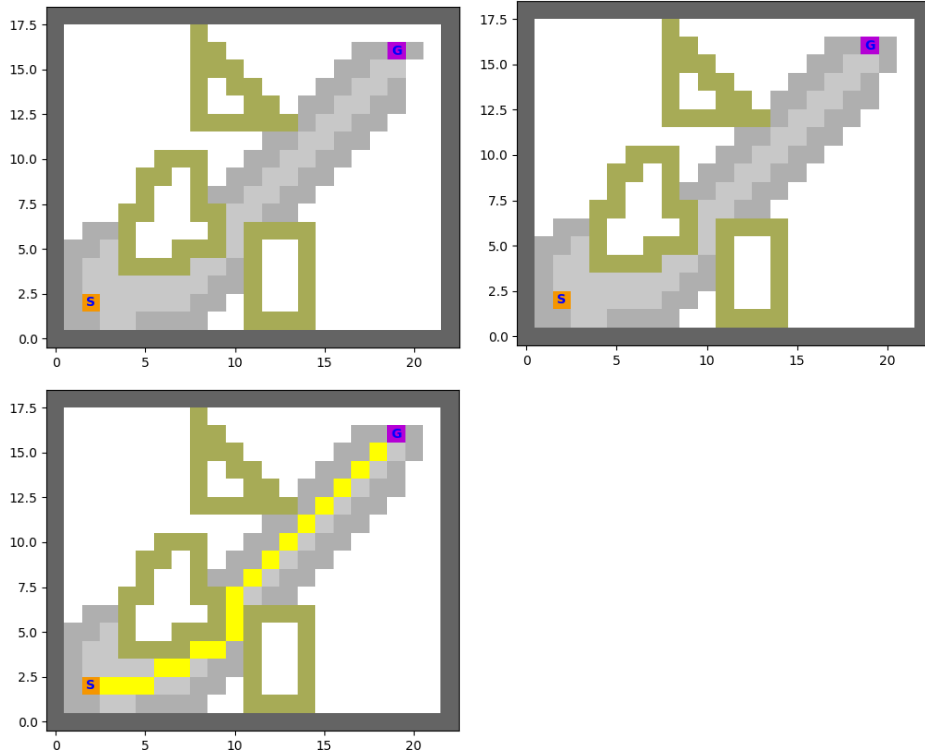
```
A* algorithm
Size map:  22 x 18
searching...
time searching:  0.0030007362365722656 (s)
total cost: 25.0
total nodes expand (opened and closed, maybe repeated): 138
□
```

Gồm tên thuật toán, kích thước thể giới, thời gian tìm kiếm, chi phí đường đi và tổng số node đã xét.

Việc thể hiện đồ họa đồ quá trình tìm kiếm nhóm em đã thực hiện chiếu các hình ảnh liên tiếp của thể giới khi thuật toán đi tới 1 node, đóng 1 node và chạm tới đích (trông giống như một video chuyển động). Tường thành có màu xanh đen, các đa giác có màu xanh xám nhạt. Các node ở frontier có màu xám đậm, với node đã đóng có màu xám nhạt, khi chạm đích sẽ thể hiện đường đi tìm được bằng 1 dải ô màu vàng.

Thể giới khi mới khởi tạo:





Ở mức 3 có thể hiện đồ họa đồ họa hơi khác một chút, chi tiết xin thầy coi tại phần II mức 3 bên dưới.

## II. Chi tiết các thuật toán và quá trình chạy thử:

### Mức 1

#### Giải thuật A\*

A\* là một thuật toán tìm kiếm trong đồ thị thuộc loại informed search (tức có sử dụng thông tin heuristic) và hiển nhiên là một ví dụ điển hình của loại tìm kiếm theo lựa chọn tốt nhất (best-first search).

Trong quá trình tìm kiếm, A\* mở rộng các node cho đến khi chạm được tới node đích. Nhưng do có thêm thông tin heuristic (là khoảng từ node hiện tại tới đích) nên A\* chỉ mở rộng đối với các node gần node đích hơn về mặt khoảng và các node có quãng đường di chuyển từ node khởi hành tới node hiện tại ngắn hơn.

Khi mở rộng, A\* đưa các node mới đánh giá vào frontier, đóng các node đã đi qua, sau đó chọn một node khả thi nhất từ frontier để mở, đưa các node lân cận với node mở này vào frontier,.. quá trình này cứ tiếp diễn cho tới khi mở được node đích.

Mỗi node đều có 1 giá trị  $f = h + g$  để đánh giá khả năng đường đi ngắn nhất sẽ đi qua node này. Giá trị  $f$  càng nhỏ thì càng dễ được chọn. Trong đó  $g$  là chi phí đường đi từ lúc khởi hành tới node mở hiện tại.

Còn  $h$  nhóm em chọn là chi phí đường đi ngắn nhất khi không có vật cản giữa 2 node.

Vì hàm  $h$  này luôn cho đường đi ngắn nhất nhỏ hơn hoặc bằng đường đi tối ưu nên thuật toán luôn trả về kết quả tối ưu.

Việc có thông tin heuristic đảm bảo cho  $A^*$  giữ được tốc độ tìm kiếm khá nhanh mà lại luôn cho kết quả tối ưu (trong trường hợp hàm  $h$  của nhóm em), chính là lý do khiến  $A^*$  trở thành một trong những thuật toán tìm kiếm nổi tiếng nhất thời điểm hiện tại.

Mô tả sơ lược giải thuật:

```
frontier = []
closed_node = []
thêm S vào frontier
while (khi trong frontier vẫn còn node):
    current_node = node có giá trị h nhỏ nhất trong frontier
    thêm current_node vào closed_node
    if là node G:
        thoát khỏi vòng lặp
    tìm các node lân cận với current_node
    với mỗi node lân cận:
        if node lân cận nằm trong closed_node:
            continue
        tính g,h. Rồi tính  $f=g+h$ 
        ghi nhớ lại node cha của node lân cận
        if node lân cận đã có trong frontier và  $f$  mới <  $f$  cũ:
            cập nhật node trong frontier, node cha
        if node lân cận không có trong frontier:
            thêm node lân cận vào frontier
```

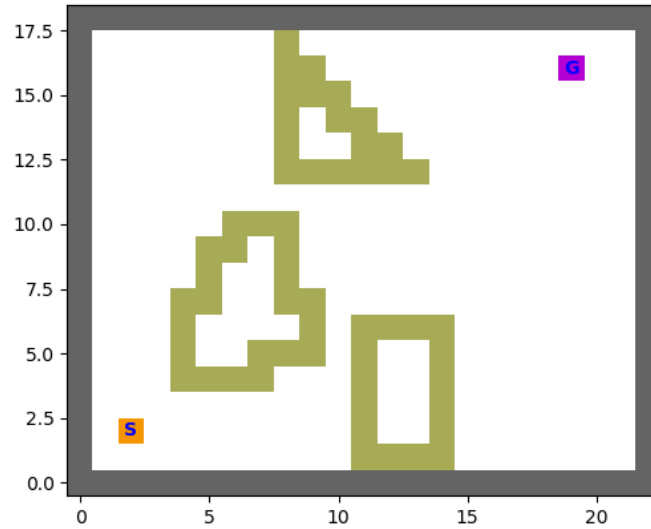
Từ node cha của G, dựng lại đường đi ngắn nhất

Trong các thao tác với frontier, để tối ưu thời gian tìm kiếm, nhóm em đã triển khai frontier thành một hàng đợi ưu tiên (thực hiện bởi min heap có sẵn trong python). Việc kiểm tra một node có nằm trong closed\_node hay không cũng được thực hiện nhanh chóng nhờ lưu kết quả luôn trong ma trận emap, nên khi cần kiểm tra chỉ cần tham chiếu tới emap với index là tọa độ của node đang xét.



### Quá trình chạy thử:

#### - Trường hợp địa hình đơn giản:



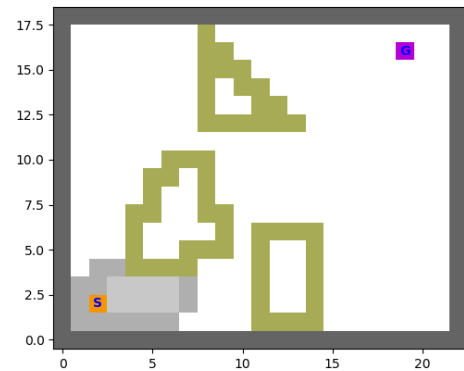
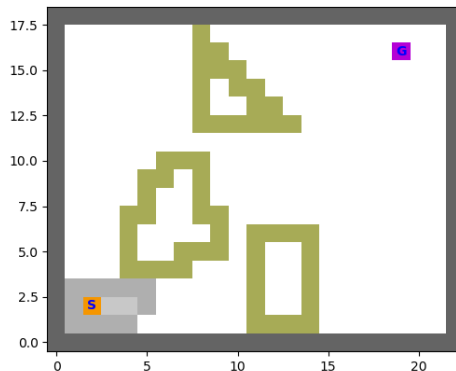
2 điểm S và G nằm ở góc của bản đồ, ở giữa bị chặn bởi 3 đa giác.

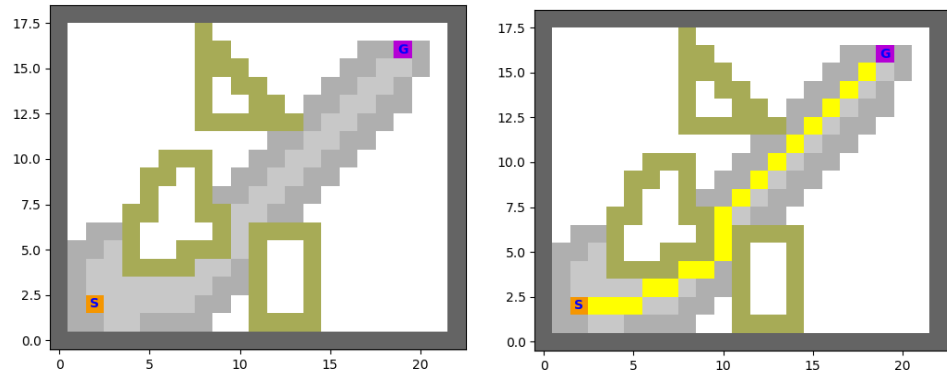
Sau khi chạy thuật toán

Kết quả ở màn hình console:

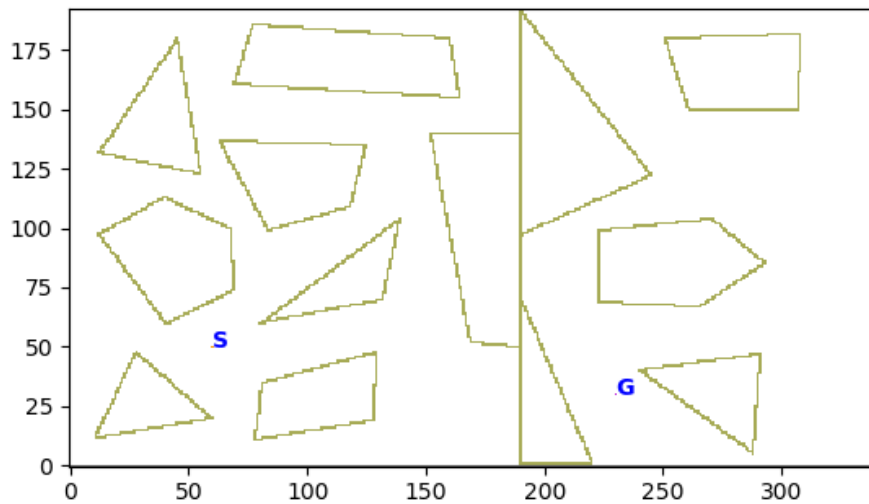
```
A* algorithm
Size map: 22 x 18
searching...
time searching: 0.0030007362365722656 (s)
total cost: 25.0
total nodes expand (opened and closed, maybe repeated): 138
[]
```

Thể hiện đồ họa ở 1 vài bước đầu tiên và cuối cùng (chi tiết xin thầy hãy chạy thử thuật toán để xem toàn bộ quá trình)





- **Trường hợp không tìm được đường đi:**



Các đa giác đã chặn hết lối đi từ S tới G

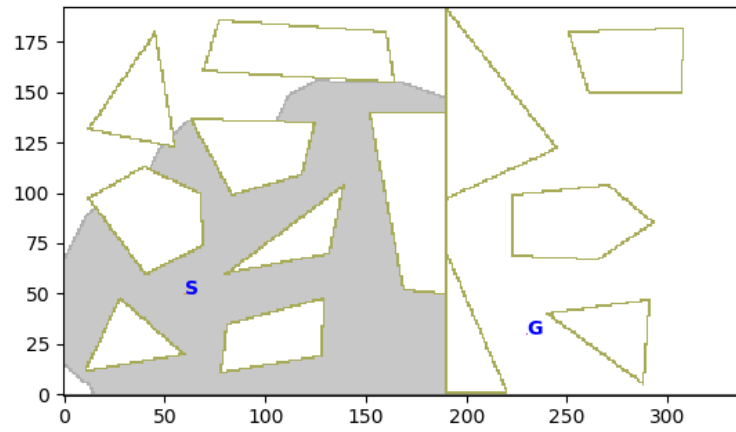
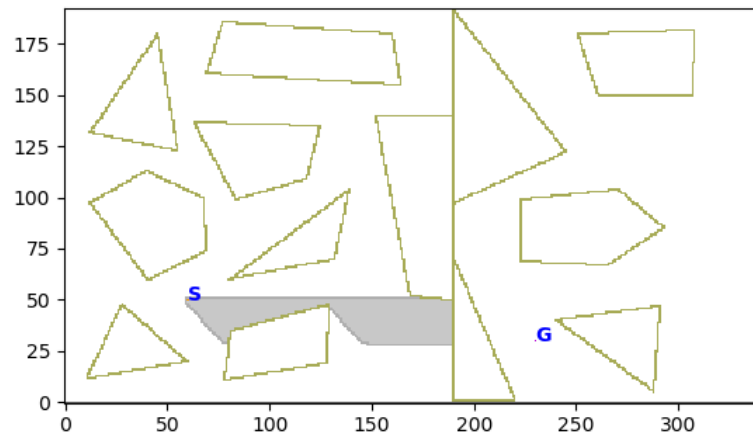
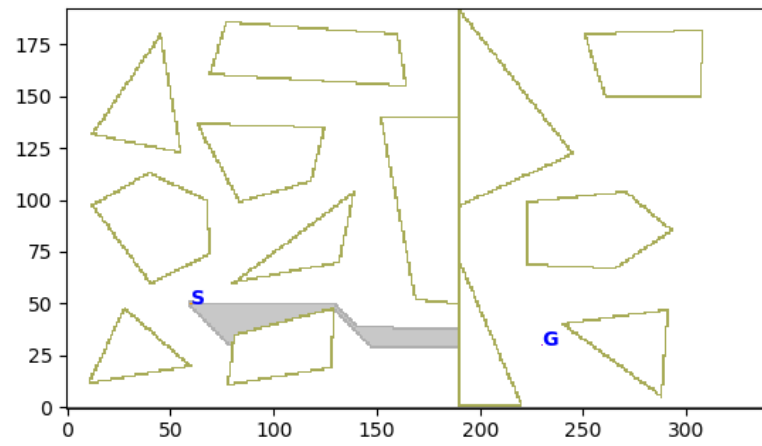
Sau khi chạy thuật toán

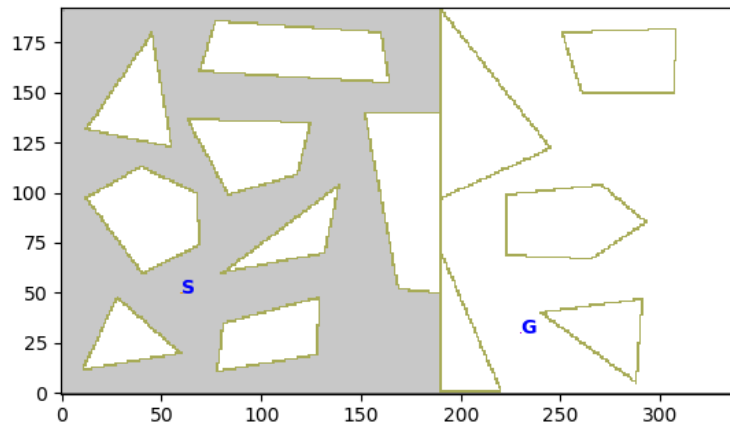
Kết quả ở màn hình console

```
A* algorithm
Size map: 340 x 192
searching...
time searching: 1.146338939666748 (s)
can not find any path from S to G
total nodes expand (opened and closed, maybe repeated): 61037
□
```

Dòng thông báo “can not find any path from S to G” thể hiện không tìm được đường đi.

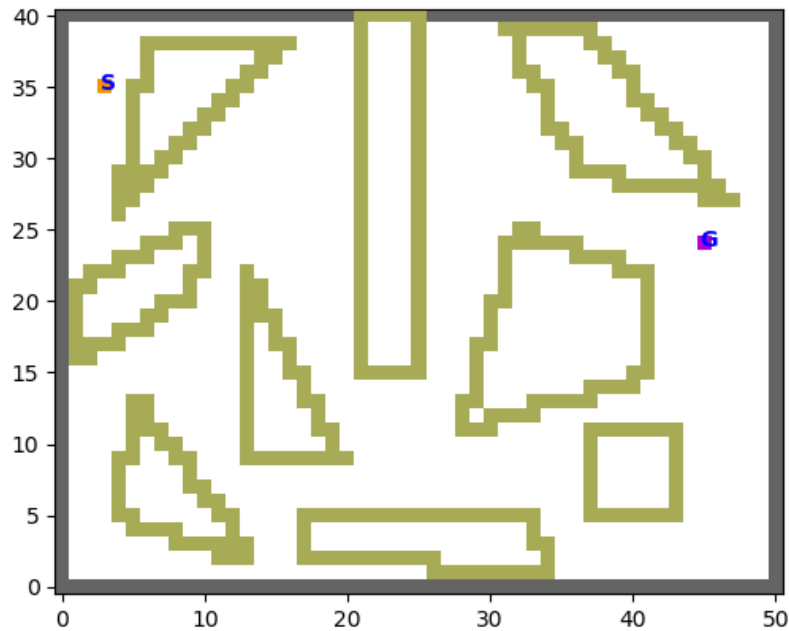
Thể hiện đồ họa ở 1 vài bước đầu tiên và cuối cùng (chi tiết xin thầy hãy chạy thử thuật toán để xem toàn bộ quá trình)





Các node trải rộng ra tất cả các nơi mà từ S có thể đi tới

**-Trường hợp có địa hình vừa phải:**

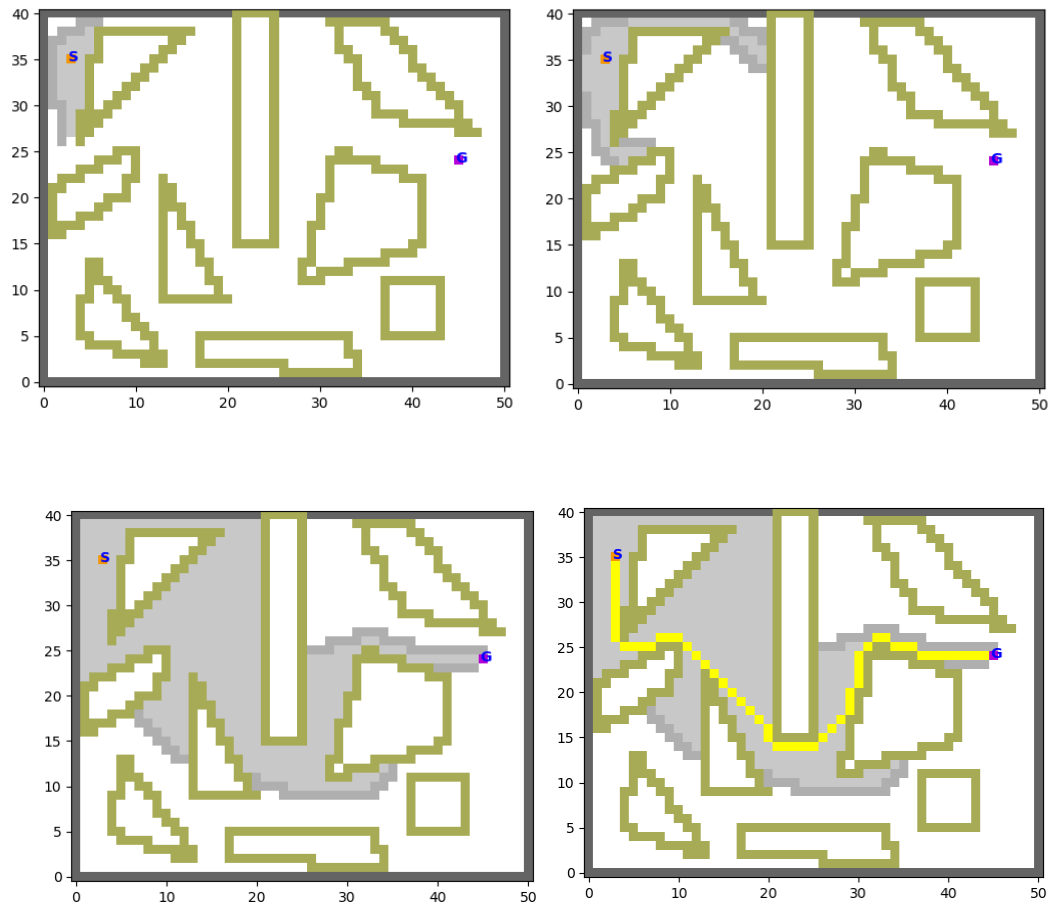


Sau khi chạy thuật toán

Kết quả ở màn hình console

```
A* algorithm
Size map: 50 x 40
searching...
time searching: 0.016000986099243164 (s)
total cost: 68.0
total nodes expand (opened and closed, maybe repeated): 1131
```

Thể hiện đồ họa ở 1 vài bước đầu tiên và cuối cùng (chi tiết xin thầy  
hãy chạy thử thuật toán để xem toàn bộ quá trình)



## Mức 2

### Giải thuật A\*

Như đã trình bày ở mức 1

### Giải thuật best-first search

Tên gọi của thuật toán này trùng khớp với lớp thuật toán mà nó thuộc về - best first search.

Best-first search cũng sử dụng một hàm đánh giá  $f$  để đánh giá độ ưu tiên mở các node trong frontier.

Hàm đánh giá  $f = h$ , best-first search chỉ dùng thông tin heuristic để lượng giá các node mà không quan tâm đến chi phí  $g$  (không quan tâm mình đã đi xa tới đâu, cứ node nào gần  $G$  nhất là đi tới). Điều này khiến cho đường đi của thuật toán không đạt được giá trị tối ưu.

Nhưng theo nhóm em thấy hình dáng đường đi của best-first search trông rất tự nhiên, các node mở rộng rất ít, thời gian chạy cực kỳ nhanh và chi phí đường đi cũng chênh lệch không quá xa so với đường đi tối ưu.

Trong thực tế khi yêu cầu về thời gian tìm kiếm thực sự gắt gao thì ta nên chọn best-first search, giải thuật cho ra một kết quả không hề tệ trong thời gian chớp mắt.

Mô tả sơ lược giải thuật:

```
frontier = []
closed_node = []
thêm S vào frontier
while (khi trong frontier vẫn còn node):
    current_node = node có giá trị h nhỏ nhất trong frontier
    thêm current_node vào closed_node
    nếu là node G:
        | thoát khỏi vòng lặp
    tìm các node lân cận với current_node
    với mỗi node lân cận:
        | nếu node lân cận nằm trong closed_node:
        |     continue
        | tính h
        | ghi nhớ lại node cha của node lân cận
        | thêm node lân cận vào frontier
```

Từ node cha của G, dựng lại đường đi ngắn nhất

Ở đây nhóm em không cập nhật giá trị  $f$  của node lân cận là do giá  $f$  này chỉ gồm  $h$ , mà  $h$  thì luôn không đổi. Việc cập nhật node cha cũng không có ý nghĩa do thuật toán không phân biệt được đi từ node cha nào sẽ ngắn hơn.

## Giải thuật Dijkstra:

Dijkstra là một thuật toán tương đương với giải thuật duyệt cây uniform-cost search (tìm kiếm chi phí đồng nhất).

Dijkstra là một giải thuật tìm kiếm mù (tức là việc biết trước tọa độ G không mang lại lợi ích gì cho thuật toán).

Dijkstra cũng chọn ra các node đang xét bỏ vào frontier, sau đó chọn ra node có tổng chi phí đường đi từ S tới node đó là ngắn nhất, rồi từ node này ta thêm các node lân cận vào frontier,.. cứ như vậy cho tới khi mở được node G.

Giải thuật có cách triển khai rất giống với A\*, nếu lấy hàm  $h=0$  thì ta có  $f=g$ , khi đó việc chọn node có  $f$  nhỏ nhất trong frontier cũng giống như việc chọn node có chi phí từ S tới là nhỏ nhất.

Vì không sử dụng bất kỳ thông tin heuristic nào nên Dijkstra có số node mở rộng rất lớn trước khi có thể chạm tới được node đích (đồng nghĩa với thời gian tìm kiếm tương đối lâu).

Dù rằng bị so sánh là chậm hơn A\* trong hầu hết các trường hợp những Dijkstra có khả năng tìm được đường đi ngắn nhất từ một đỉnh tới tất cả các đỉnh khác trong đồ thị trong thời gian chỉ hơn thời gian tìm đường đi giữa đỉnh một chút nên giải thuật vẫn có chỗ đứng rất vững chắc trong nhiều bài toán phức tạp hơn.

Mô tả sơ lược giải thuật:

```
frontier = []
closed_node = []
thêm S vào frontier
while (khi trong frontier vẫn còn node):
    current_node = node có giá trị h nhỏ nhất trong frontier
    thêm current_node vào closed_node
    if là node G:
        thoát khỏi vòng lặp
    tìm các node lân cận với current_node
    với mỗi node lân cận:
        if node lân cận nằm trong closed_node:
            continue
        tính g
        ghi nhớ lại node cha của node lân cận
        if node lân cận đã có trong frontier và g mới < g cũ:
            cập nhật node trong frontier, node cha
        if node lân cận không có trong frontier:
            thêm node lân cận vào frontier
```

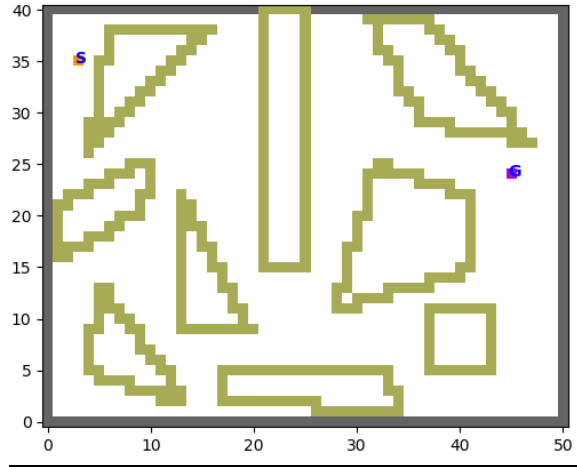
Từ node cha của G, dựng lại đường đi ngắn nhất

Vì mã nguồn A\* và Dijkstra rất giống nhau nên nhóm em đã gộp 2 thuật toán làm một và đặt trong hàm `heuristic_search()`, khi cần thực thi A\* thì

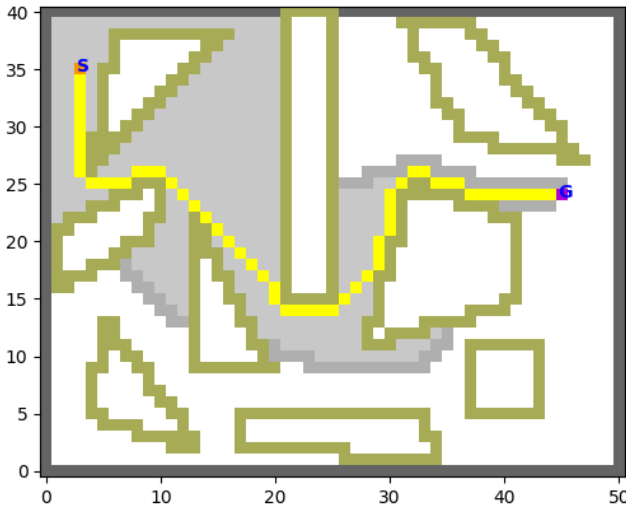
truyền vào hàm h là hàm shortest\_h, còn nếu cần Dijkstra thì truyền vào zero

## Chạy thử, so sánh:

### +MAP 1: Địa hình không quá phức tạp

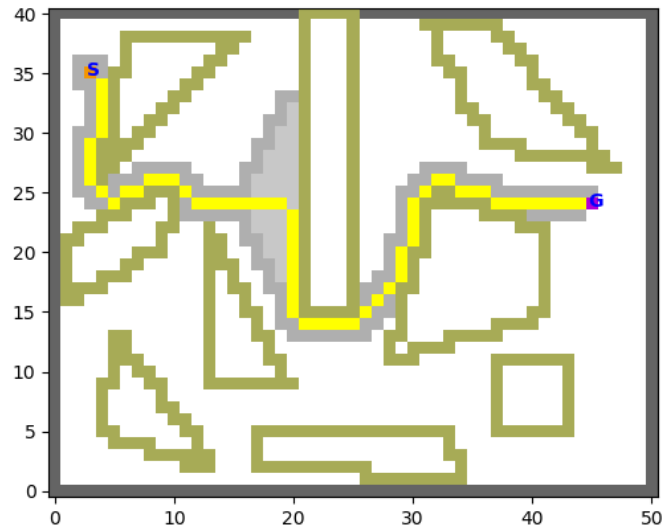


Kết quả:

Thuật toán	Kết quả hiển thị
A*	<div></div> <div>A* algorithm Size map: 50 x 40 searching... time searching: 0.0219876766204834 (s) total cost: 68.0 total nodes expand (opened and closed, maybe repeated): 1131</div>

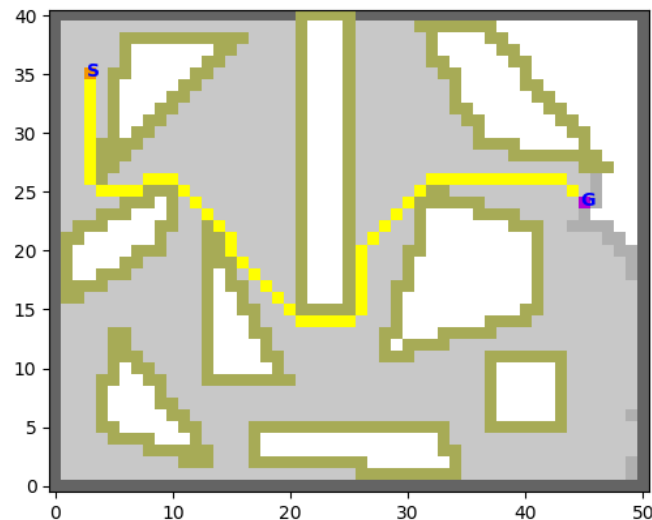


### Best first search



```
Best first search algorithm
Size map: 50 x 40
searching...
time searching: 0.001987934112548828 (s)
total cost: 73.5
total nodes expand (opened and closed, maybe repeated): 313
```

### Dijkstra



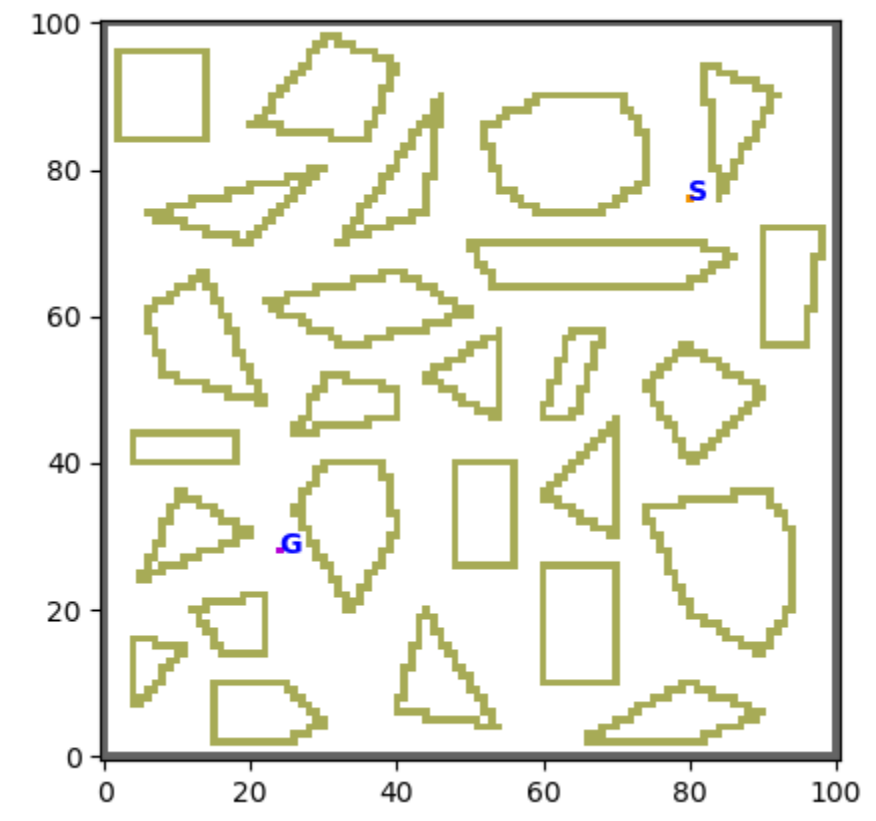
```
Dijkstra algorithm
Size map: 50 x 40
searching...
time searching: 0.03897666931152344 (s)
total cost: 68.0
total nodes expand (opened and closed, maybe repeated): 2048
```

Thuật toán	Total Cost	Time Searching
A*	68	0.0219876766204834 (s)

BestFirstSearch	73.5	0.001987934112548828 (s)
Dijkstra	68	0.03897666931152344 (s)

Thuật toán nhanh nhất là best-first search (nhanh gấp hơn 10 lần A\*, gần 20 lần Dijkstra), nhưng lại không có kết quả tối ưu. A\* vừa cho kết quả tối ưu vừa nhanh hơn Dijkstra (nhanh hơn không nhiều trong trường hợp này).

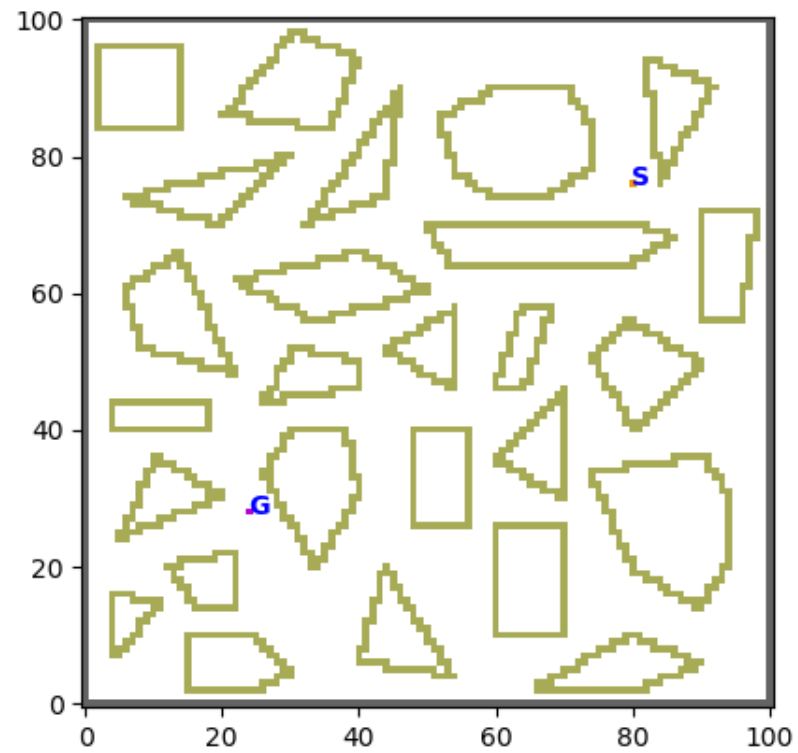
### **+MAP 2: Bản đồ rộng lớn, địa hình phức tạp**



**Kết quả:**

Thuật toán	Kết quả hiển thị
------------	------------------

A\*



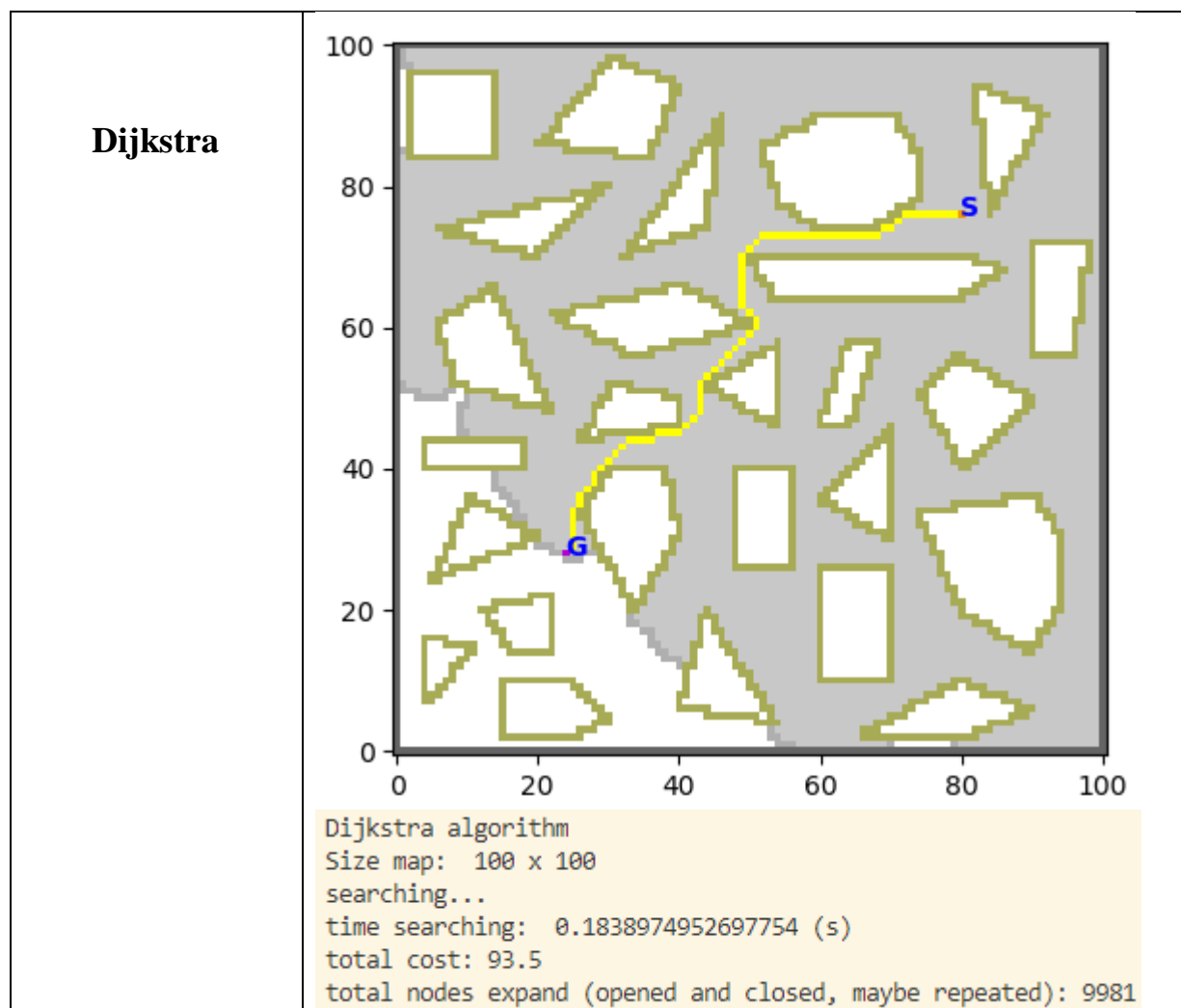
```
A* algorithm
Size map: 100 x 100
searching...
time searching: 0.0399775505065918 (s)
total cost: 93.5
total nodes expand (opened and closed, maybe repeated): 1819
```

**Best first search**

```

Best first search algorithm
Size map: 100 x 100
searching...
time searching: 0.0029931068420410156 (s)
total cost: 98.5
total nodes expand (opened and closed, maybe repeated): 336
    
```

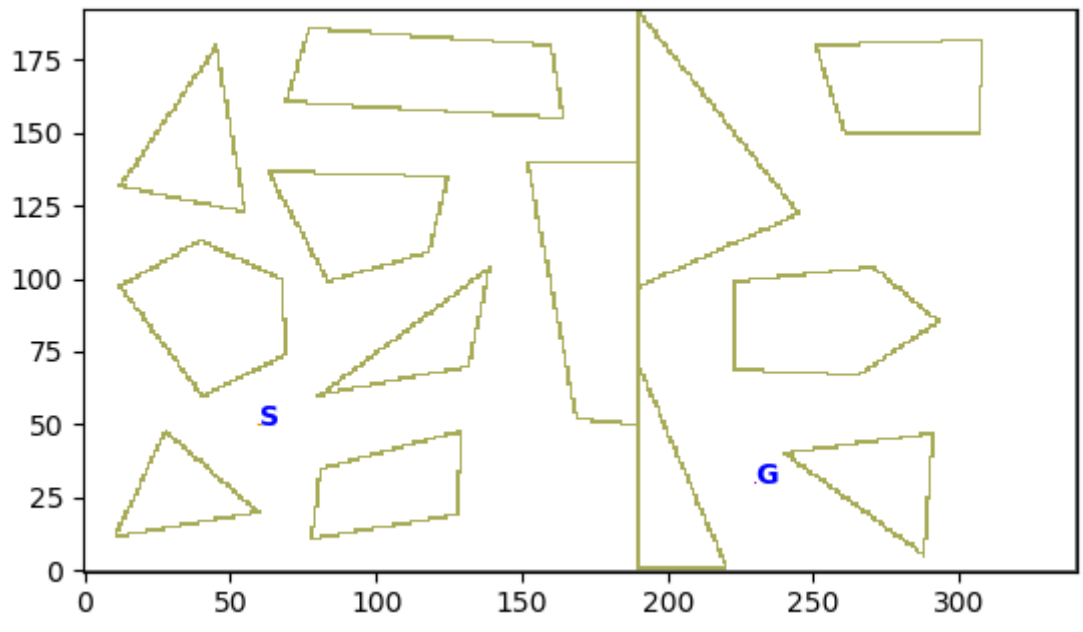
```
Best first search algorithm
Size map: 100 x 100
searching...
time searching: 0.0029931068420410156 (s)
total cost: 98.5
total nodes expand (opened and closed, maybe repeated): 336
```



Thuật toán	Total Cost	Time Searching
A*	93.5	0.0399775505065918 (s)
BestFirstSearch	98.5	0.0029931068420410156 (s)
Dijkstra	93.5	0.1838974952697754 (s)

Với những map rộng lớn như trường hợp này, Dijkstra lộ rõ tốc độ hạn chế của mình, các node expand trải rộng gần như toàn bộ bản đồ. A\* cho kết quả tối ưu và nhanh gấp 5 lần Dijkstra. Best first search vẫn luôn giữ phong độ là thuật toán nhanh nhất (gấp 13 lần A\* và gấp 65 lần Dijkstra, 1 tốc độ nhanh đến kinh khủng) với kết quả không quá lệch với kết quả tối ưu.

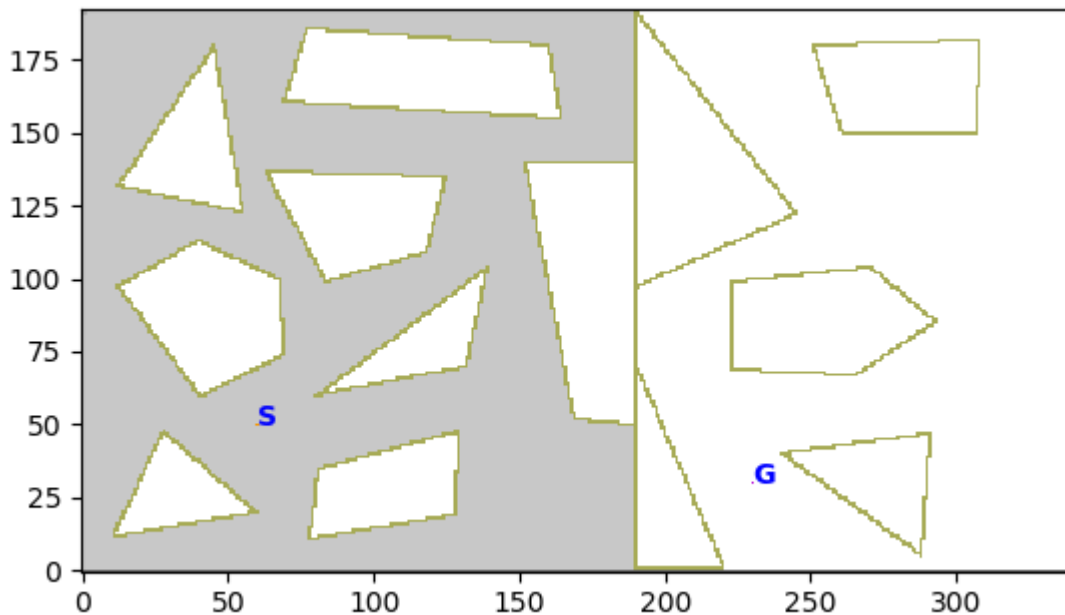
### **+MAP 3: Không tìm được đường đi từ S tới G**



**Kết quả:**

Thuật toán	Kết quả hiển thị
<b>A*</b>	<pre> A* algorithm Size map: 340 x 192 searching... time searching: 1.2142980098724365 (s) can not find any path from S to G total nodes expand (opened and closed, maybe repeated): 61037 </pre>
<b>Best first search</b>	<pre> Best first search algorithm Size map: 340 x 192 searching... time searching: 0.5316934585571289 (s) can not find any path from S to G total nodes expand (opened and closed, maybe repeated): 45951 </pre>
<b>Dijkstra</b>	<pre> Dijkstra algorithm Size map: 340 x 192 searching... time searching: 0.8794918060302734 (s) can not find any path from S to G total nodes expand (opened and closed, maybe repeated): 45955 </pre>

Cả 3 thuật toán đều có hiển thị đồ họa vào bước cuối cùng như sau:



Thuật toán	Total Cost	Time Searching
A*	$\infty$	1.2142980098724365 (s)
BestFirstSearch	$\infty$	0.5316934585571289 (s)
Dijkstra	$\infty$	0.8794918060302734 (s)

Cả 3 đều có kết luận đúng về đường đi (không tìm được đường đi) từ S tới G. Về mặt thời gian bây giờ Dijkstra đã chạy nhanh hơn A\*, do có số node expand tương đương mà lại không mất thời gian tính hàm heuristic. Best first search vẫn có thời gian chạy nhanh nhất nhưng cũng không hơn A\* và Dijkstra là bao.

## MỨC 3:

Đề bài yêu cầu tìm đường đi ngắn nhất từ S tới G nhưng trước khi tới G phải đi qua các điểm đón. Bài toán này sẽ được giải quyết dễ dàng nếu các điểm S, G, điểm đón tạo thành 1 đồ thị. Khi đó bài toán trở thành “tìm đường đi ngắn nhất qua tất cả các đỉnh của đồ thị”, rất giống bài toán người giao hàng chỉ khác là đã biết trước cả điểm khởi hành và điểm kết thúc. Bài toán người giao hàng được chứng minh là một bài toán thuộc lớp np-hard (tức hiện tại vẫn chưa có một giải thuật nào hiệu quả, ít nhất

là trong thời gian đa thức, giải quyết được bài toán này), nên nhóm em đã không câu nệ việc giải thuật kém hiệu quả, đi theo một ý tưởng đơn giản:

Nhờ Dijkstra tìm tất cả các con đường ngắn nhất giữa S, G, các điểm đón (chỉ trừ đường đi trực tiếp giữa S và G là không cần tìm).

Lập một đồ thị đầy đủ từ những con đường này

Hoán vị các điểm đón để tìm ra con đường ngắn nhất

Vì phải đi qua tất cả các điểm nên việc tìm đường đi tối ưu giữa các điểm là hoàn toàn cần thiết. Và vì sau đó nhóm em đã tính hết tất cả những con đường ngắn nhất có thể đi nên đường đi (nếu tồn tại) sẽ là đường đi tối ưu (ngắn nhất từ S tới G thông qua các điểm đón).

Mô tả sơ lược giải thuật:

```
middle_node
Đánh dấu tất cả các điểm đón đều là điểm đích trên emap
#tìm đường đi từ S tới các điểm đón
for i in range(số điểm đón trong middle_node):
    cho chạy thuật toán dijkstra tìm đường đi từ S tới đỉnh gần nhất
    lưu lại kết quả
#tìm đường đi giữa các điểm đón với nhau và với G
thêm G vào cuối của middle_node
for node in middle_node[:-1]:
    bỏ đánh dấu đích của node trên emap
    for i in range(số điểm đón trong middle_node từ sau node trở đi):
        cho chạy thuật toán dijkstra tìm đường đi từ node tới đỉnh gần nhất
        lưu lại kết quả
gán total_g là 1 số rất lớn
gán min_p là 1 hoán vị ngẫu nhiên
tạo hoán vị các số từ 0 tới số điểm đón - 1:
    với mỗi hoán vị tính tổng chi phí đi từ S->các điểm đón theo thứ tự hoán vị->G
    if tổng chi phí < total_g:
        total_g = tổng chi phí
        min_p = hoán vị hiện tại
từ bộ hoán vị min_p ta dựng lại đường đi ngắn nhất
```

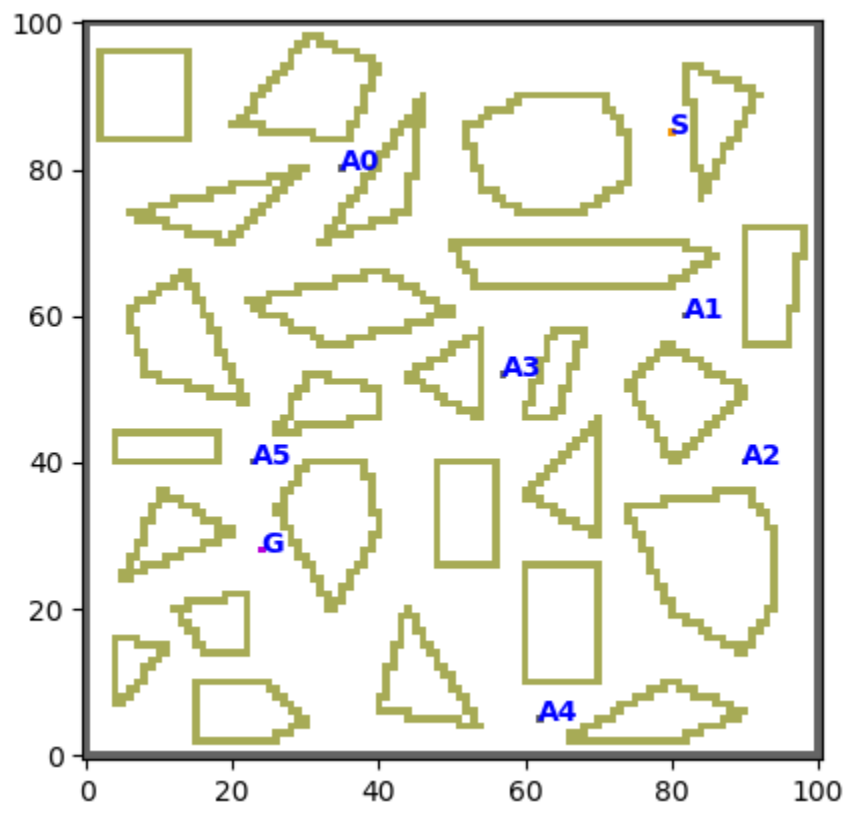
Việc dùng Dijkstra liên tục trên tất cả các đỉnh tốn rất nhiều thời gian (mặc dù nhóm em đã hạn chế bằng cách không tìm kiếm lại các con đường đã có kết quả). Bước hoán vị cũng sẽ tốn nhiều thời gian nếu số điểm đón đủ lớn, nhưng do số điểm đón chỉ khoảng dưới 10 điểm (nhỏ so với kích thước ma trận) nên chi phí thời gian gian đoạn sau coi như không đáng kể.

**Quá trình chạy thử:**



File input chứa thông tin các điểm đón dưới dạng như sau: ngay sau dòng cuối cùng phần thông tin các đa giác là số n thể hiện số lượng điểm đón. Sau đó là n dòng, mỗi dòng ghi hoành độ và tung độ của điểm đón, cách nhau bởi dấu phẩy.

### **MAP 1:**

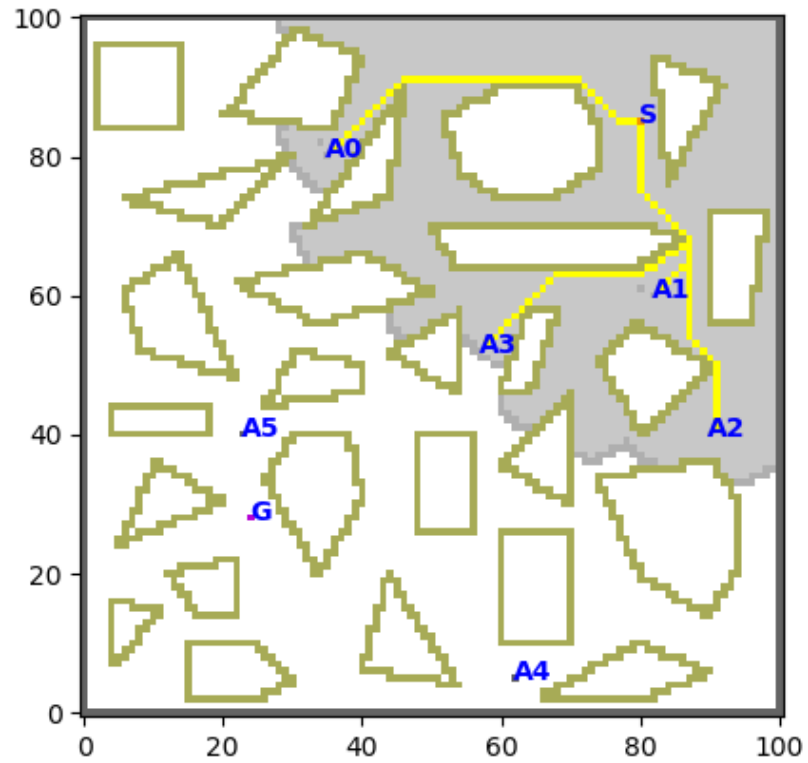


Bản đồ 100x100, điểm S(80,85) G(24,28) với 26 đa giác và 6 điểm đón (từ A0 tới A5)

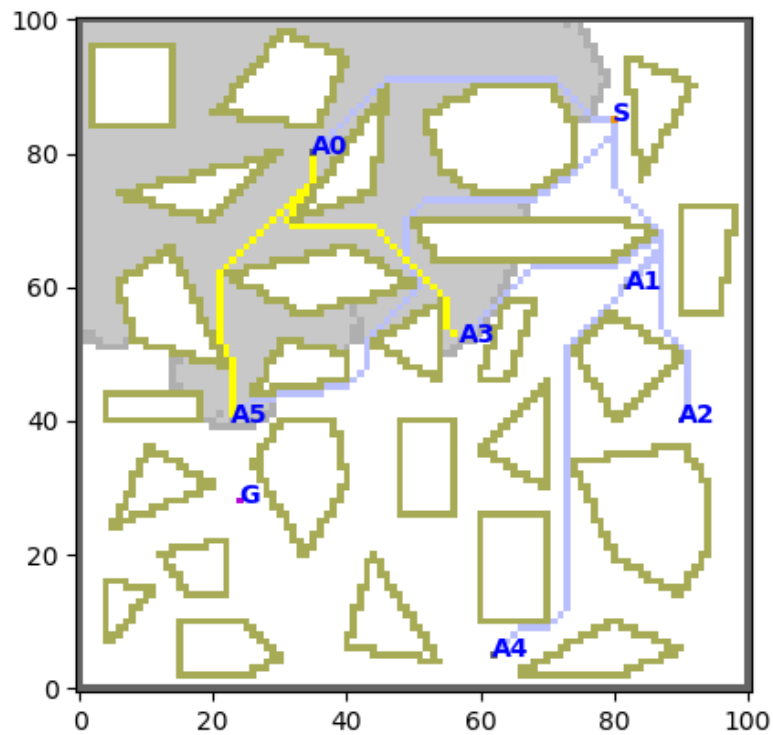
### **Kết quả :**

Bước thiết lập đồ thị

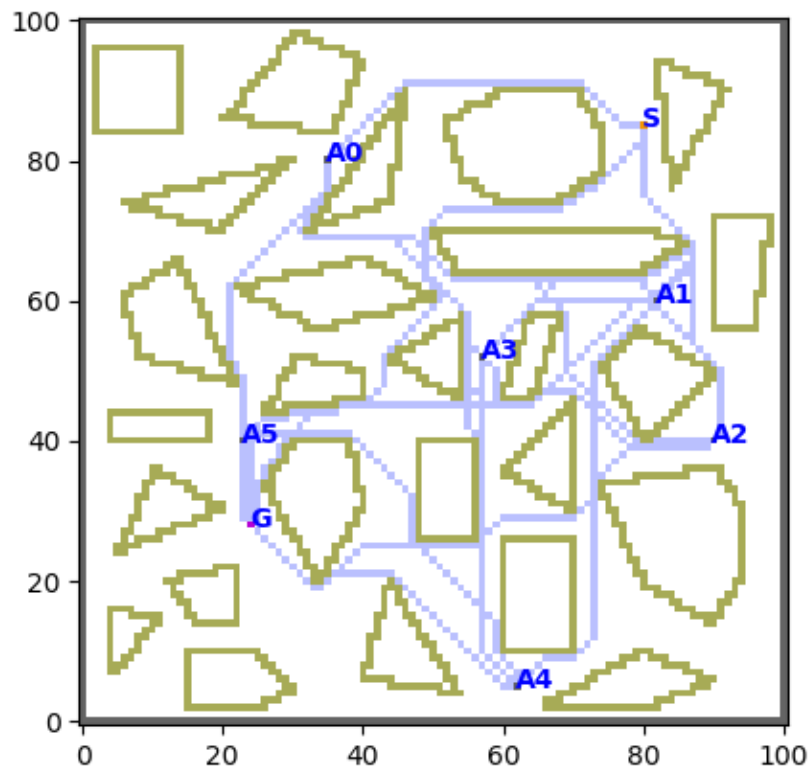
Tìm tất cả đường đi ngắn nhất từ S tới các điểm đón  
Đường đi thể hiện bằng đường màu vàng



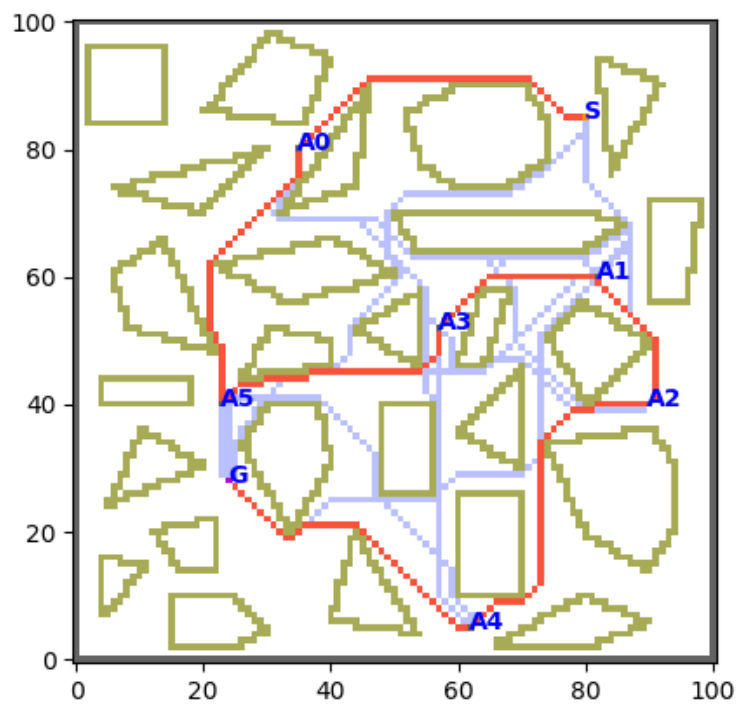
Sau đó tìm đường đi ngắn nhất từ A0 tới các điểm khác (trừ S, do đường đi giữa S và A0 đã được tìm từ trước). Các đường đã tìm ra cũ có màu xanh nhạt, các đường vừa tìm được ở đỉnh đang xét có màu vàng



Quá trình trên cứ tiếp diễn, cuối cùng đạt được đồ thị sau (gồm các đường màu xanh)



Sau đó kết quả được tính toán và hiển thị dưới dạng một đường đi màu đỏ



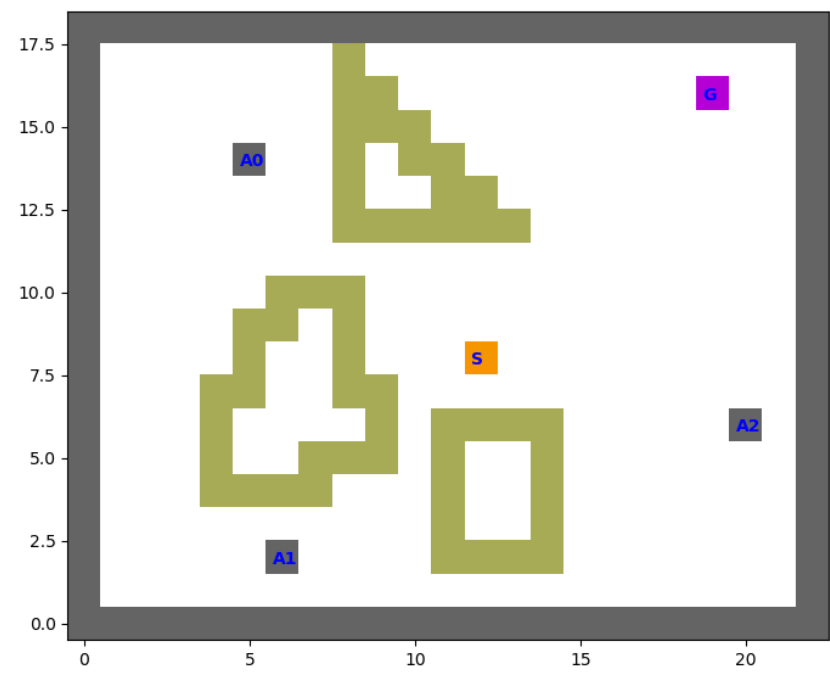
Chi tiết quá trình tìm kiếm mong thầy hãy chạy thử thuật toán để xem hết tất cả các bước

### Hiển thị ở console

```
Level 3 searching
Size map: 100 x 100
searching...
(0, 5, 3, 1, 2, 4)
time searching: 0.9384593963623047 (s)
total cost: 304.5
via S -> A0(80;35) -> A5(40;23) -> A3(52;57) -> A1(60;82) -> A2(40;90) -> A4(5;62) -> G
total nodes expand (opened and closed, maybe repeated): 51373
```

Thuật toán đã tìm ra đường đi tối ưu trong thời gian có thể chấp nhận được (dưới một giây).

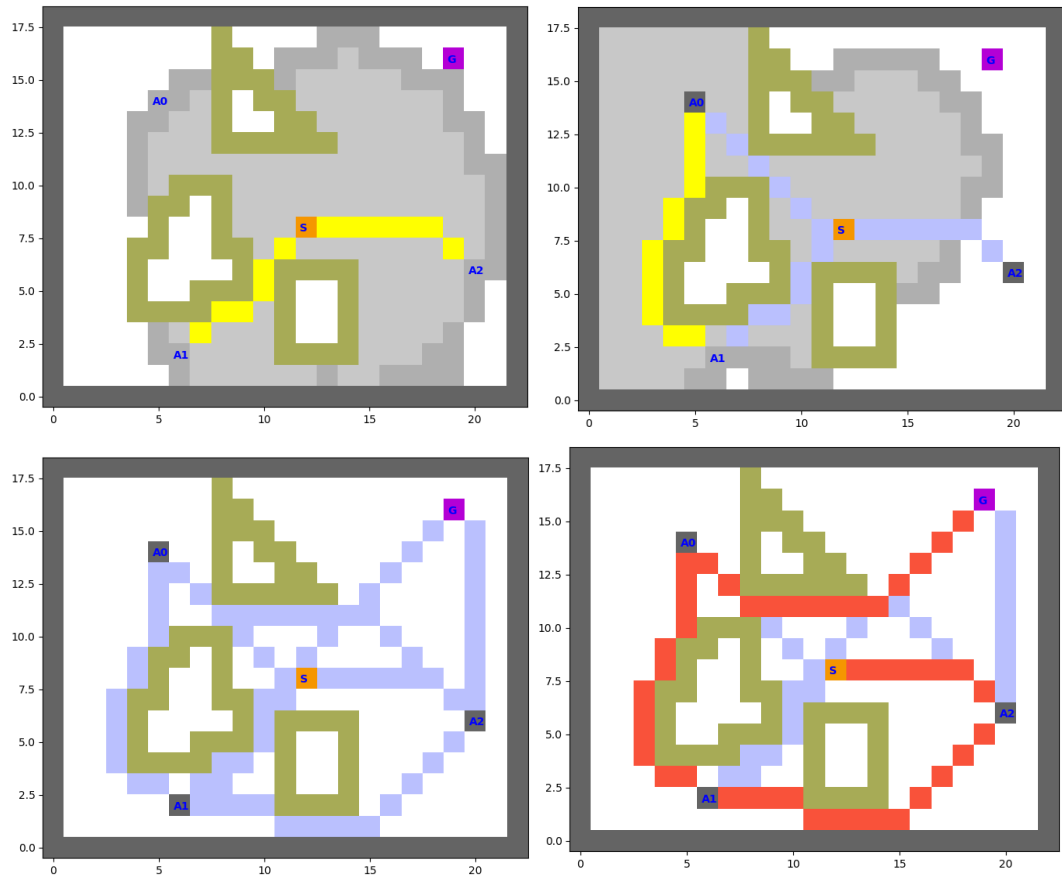
### MAP 2:



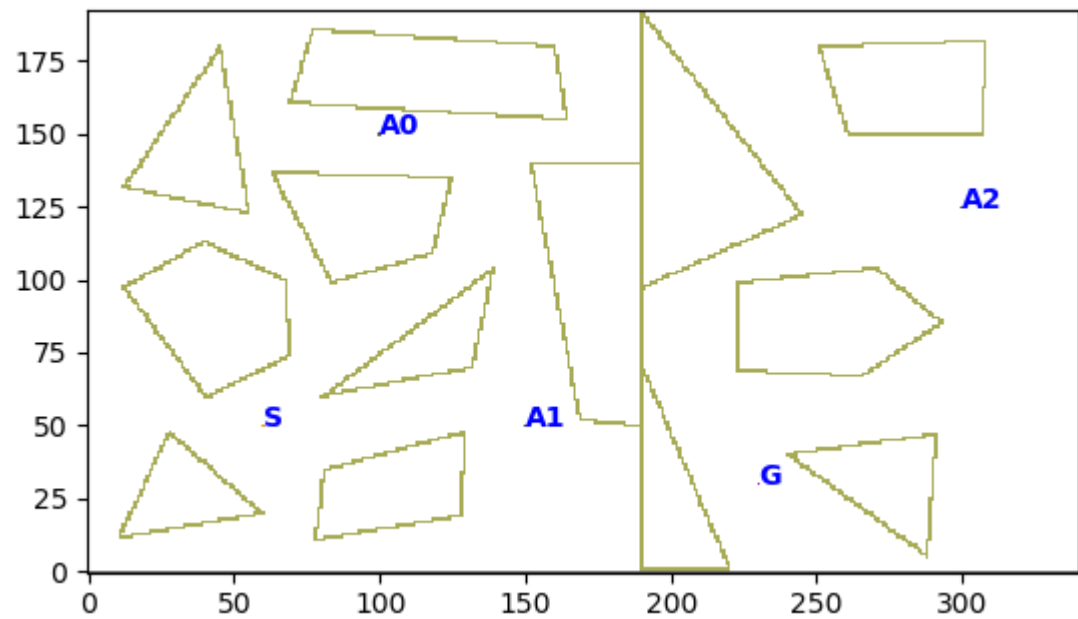
Bản đồ 22x18, điểm S(12,8) G(19,16) với 3 đa giác và 3 điểm đón

### Kết quả :

Một số thể hiện đồ họa của quá trình tìm kiếm và hiển thị kết quả cuối cùng



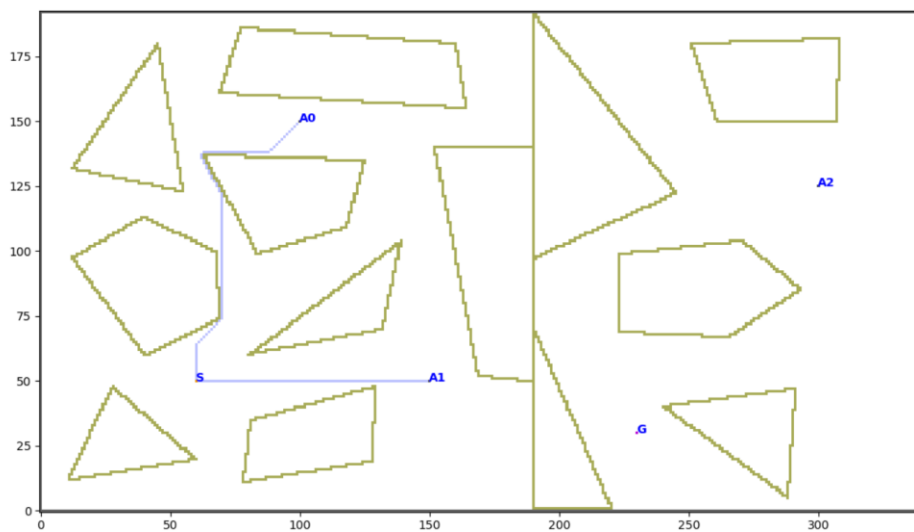
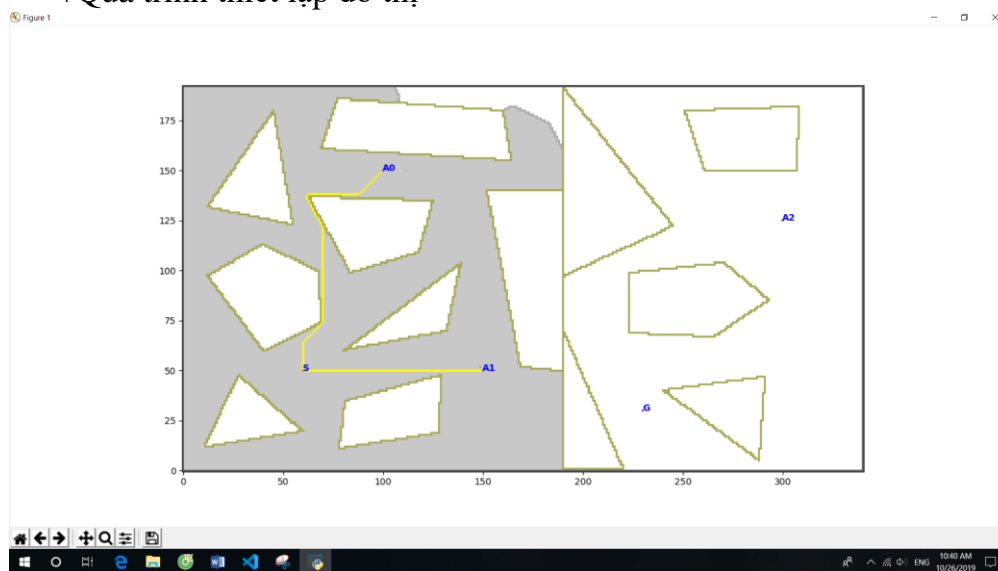
### MAP 3:



Bản đồ 340x192, điểm S(60,50) G(230,30) với 13 địa hình chướng ngại vật và 3 điểm đón. Bản đồ bị chia và không có đường đi nào giữa 1 số node.

## Kết quả

+Quá trình thiết lập đồ thị



Hiện thị ở console

```
Level 3 searching
Size map: 340 x 192
searching...
time searching: 0.889491081237793 (s)
can not find any path from S to G via all middle nodes
total nodes expand (opened and closed, maybe repeated): 45955
```

Trong bước thiết lập đồ thị, thuật toán chỉ tìm thấy đường giữa S và A0, A1. Khi không thể tìm thấy đường đi giữa 2 node bất kỳ (chứng tỏ bản đồ không liên thông) thì thuật toán kết thúc sớm và thể hiện đồ họa lại quá trình tìm kiếm tới khi dừng lại.

### III. Các nguồn tài liệu tham khảo:

#### Về giải thuật

[https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt\\_to%C3%A1n\\_Dijkstra](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_to%C3%A1n_Dijkstra)

[https://vi.wikipedia.org/wiki/T%C3%ACm\\_ki%E1%BA%BFm\\_theo\\_l%E1%BB%Bl%C3%A0\\_ch%E1%BB%8Dn\\_t%E1%BB%91t\\_nh%E1%BA%A5t](https://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_l%E1%BB%Bl%C3%A0_ch%E1%BB%8Dn_t%E1%BB%91t_nh%E1%BA%A5t)

<https://viblo.asia/p/a-pathfinding-nwmkyEjkoW>

[https://en.wikipedia.org/wiki/A\\*\\_search\\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)

[https://en.wikipedia.org/wiki/Best-first\\_search](https://en.wikipedia.org/wiki/Best-first_search)

[https://en.wikipedia.org/wiki/Travelling\\_salesman\\_problem](https://en.wikipedia.org/wiki/Travelling_salesman_problem)

<https://brilliant.org/wiki/johnsons-algorithm/>

slide môn học cơ sở trí tuệ nhân tạo, chương 3, 4

#### Về python

<https://matplotlib.org/>

[https://matplotlib.org/3.1.1/api/as\\_gen/matplotlib.pyplot.plot.html](https://matplotlib.org/3.1.1/api/as_gen/matplotlib.pyplot.plot.html)

[https://matplotlib.org/3.1.1/api/as\\_gen/matplotlib.pyplot.pause.html](https://matplotlib.org/3.1.1/api/as_gen/matplotlib.pyplot.pause.html)

<https://wiki.python.org/moin/TimeComplexity>

<https://devdocs.io/python/>

[https://www.tutorialspoint.com/python/python\\_date\\_time.htm](https://www.tutorialspoint.com/python/python_date_time.htm)

[https://matplotlib.org/3.1.1/api/as\\_gen/matplotlib.pyplot.imshow.html](https://matplotlib.org/3.1.1/api/as_gen/matplotlib.pyplot.imshow.html)

<https://docs.python.org/2/library/heapq.html>

[https://www.w3schools.com/python/python\\_lists.asp](https://www.w3schools.com/python/python_lists.asp)

#### Tham khảo implementing

<https://www.redblobgames.com/pathfinding/a-star/implementation.html>

<https://qiao.github.io/PathFinding.js/visual/>

