Ho Chi Minh University of Science
Information Technology (HQ Program)

CƠ SỞ TRÍ TUỆ NHÂN TẠO

PROJECT 1 – TÌM ĐƯỜNG ĐI TỐI ƯU

LỚP: K19CLC6

July 5, 2021
PROGRESS: 100%

NHÓM: WLTD
Ngô Huy Anh - **19127095**
Triệu Nguyên Phát – **19127505**

GIẢNG VIÊN
Châu Thành Đức
Phan Thị Phương Uyên
Ngô Đình Hy

Mục lục

Lý thuyết

Adversarial Search

1. Đặt vấn đề

Các thuật toán áp dụng chiến lược tìm kiếm có thông tin hay không có thông tin (BFS, DFS, A*,...) được ứng dụng để giải quyết một số bài toán mang tính độc lập như: tìm đường đi ngắn nhất, tìm đường trong mê cung, xây dựng game 8-puzzle, 8-queens, ... Số 'agent' được sử dụng để giải quyết các bài toán trên chỉ là 1.

Trong trường hợp các bài toán được đặt ra có số lượng 'agent' lớn hơn 1 như các trò chơi đối kháng như cờ vua, cờ vây, cờ caro,.. Những thuật toán trên không thể hiện được sự đột phá trong việc tìm kiếm. Các mẫu dữ liệu liên tục thay đổi và không tạo cơ hội để các giải thuật trên có thể tối ưu hóa được nước đi. Như vậy, cần phải có một chiến thuật tìm kiếm mạnh mẽ hơn cho các bài toán như đã nêu – **Tìm kiếm có đối thủ (Adversarial Search)**.

Chiến lược tìm kiếm có đối thủ nhận xét thông tin của đối thủ sau mỗi nước đi (chất lượng của nước đi, số lượng quân cờ còn lại, cách di chuyển, ...), sau đó đưa ra một nước đi tốt nhất để dần có thể chiếm ưu thế và giành chiến thắng (hoặc trường hợp tệ nhất là hòa).

2. Thuật toán Minimax

Minimax (ngược lại là **Maximin**) là một phương pháp trong lý thuyết quyết định có mục đích là tối thiểu hóa tổn thất vốn được dự tính có thể là tối đa[1]. **Minimax** là một trong những thuật toán được sử dụng rộng rãi trong nhánh tìm kiếm có đối thủ. Ngụ ý rằng đối thủ là một người chơi hoàn hảo, gần như sẽ không phạm bất kỳ lỗi nào trong mỗi nước đi, thuật toán **Minimax** vẫn dự đoán trước được '**tất cả**' các trường hợp có thể xảy ra trong tương lai của bàn cờ, rồi chọn ra một nước đi theo hướng có lợi nhất ở thời điểm hiện tại, cũng như gây thiệt hại lớn nhất cho đối thủ đó.

Để tránh trường hợp '**tất cả**' như nêu trên gây hiện tượng tràn bộ nhớ trong việc xử lý đệ quy, một số phiên bản của Minimax được cài thêm biến depth, mục đích để giới hạn số lần đệ quy tại mỗi state trong bàn cờ.

[1] https://vi.wikipedia.org/wiki/Minimax

```
function minimax(pos, depth, isMaximizing):
        if depth == 0 or game over at pos:
                return đánh giá nước đi tại pos

        if isMaximizing:
                maxEval = -∞  // giá trị tồi nhất mà Max phải nhận
                for từng child của pos:
                        nextEval = minimax(child, depth -1, false)
                        maxEval  = max(maxEval, nextEval)
                return maxEval

        else:
                minEval = +∞  // giá trị tồi nhất mà Min phải nhận
                for từng child của pos:
                        nextEval = minimax(child, depth -1, true)
                        minEval  = min(minEval, nextEval)
                return minEval
```

Mã giả thuật toán Minimax

Lưu ý rằng, việc đánh giá nước đi tại một state trong bàn cờ hoàn toàn phụ thuộc vào cách chơi hay hình thức của trò chơi đó. Không có một khái niệm nhất định cho việc đánh giá này. Song, trong các trò chơi cờ, người ta thường lấy số quân trắng trừ đi số quân đen như một cách đánh giá 'miễn cưỡng' (tạm chấp nhận được).

3. Giải thuật Alpha – Beta Pruning

Giải thuật Alpha – Beta Pruning là một nhánh phát triển của Minimax nhằm hỗ trợ giảm bớt các không gian trạng thái dư thừa của cây để tối ưu việc tìm kiếm tốt hơn cũng như tiết kiệm chi phí hơn. Thuật toán dựa vào 2 biến **alpha**, **beta** để so sánh những giá trị min và max cần tìm.

- **Nguyên tắc hoạt động:**

  Với alpha/beta lần lượt giữ và cập nhật các giá trị max/min.

- Xuất phát từ nút gốc, ta sử dụng thuật toán tìm kiếm theo chiều sâu (DFS) để duyệt cây, thay đổi và cập nhật dần dần các giá trị bên trong của alpha và beta.
- Nếu giá trị alpha ≥ beta thì giải thuật sẽ prune các nhánh con chưa duyệt tới.
- **Đối với các node yêu cầu trả về max:**

  Do phải chọn giá trị max trong tất cả các giá trị của node con min, nên khi alpha được cập nhật thì các node con có giá trị nhỏ hơn alpha không cần xét đến nữa.

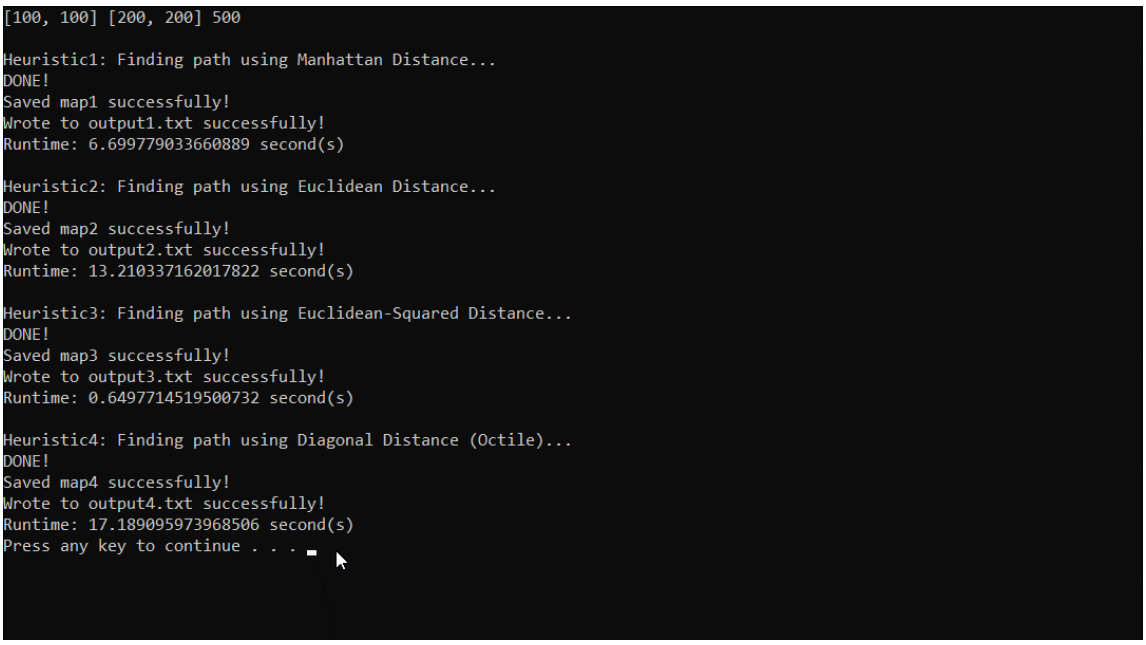
- **Đối với các node yêu cầu trả về min:**

  Do phải chọn giá trị min trong tất cả các giá trị của node con max, nên khi beta được cập nhật và alpha ≥ beta thì các node con có giá trị lớn hơn beta không cần xét đến nữa.

Vận dụng

Cách biên dịch và chạy mã nguồn

1. Tạo solution mới bằng **IDE Visual Studio Code** (ngôn ngữ **Python**).
2. Chuột phải vào **Project**, chọn **Add**, sau đó chọn hai file **Algorithms.py** và **main.py** có trong folder.
3. Đưa file **map.bmp** và **input.txt** vào đúng địa chỉ **Project** (Chuột phải vào **Project**, chọn **Open Folder in File Explorer**).
4. Quay về màn hình **Visual Studio Code**, nhấn **F5**.
5. Khi chương trình báo '**Press any key to continue . . .**' tức là đã hoàn thành.



Ví dụ của chương trình khi hoàn thành.

Các dữ liệu trên màn hình console:

Heuristic1: Số thứ tự của hàm Heuristic được áp dụng.

Fiding path using **Manhattan Distance**... : Tên của thuật toán heuristic được sử dụng.

DONE! : Thông báo đã chạy xong thuật toán.

2 dòng tiếp theo thể hiện việc **lưu output**.

Runtime: Thời gian chạy thuật toán.

Giải thuật tìm kiếm A*

```
1  function A_Star_Algorithm(map, startPoint, endPoint, m):
2      interactedPoints = set()
3
4      openSet = PriorityQueue()
5      openSet.put(0, startPoint)
6
7      gScores = 2D_Array(map.width, map.height)
8      set all value in gScores to ∞
9      gScores[startPoint.x][startPoint.y] = 0
10
11     path = Dictionary()
12     path[startPoint] = None (null)
13
14     while openSet is not empty:
15         current = openSet.get()
16
17         if current == endPoint:
18             path is found!
19             visualizePath(map, path, current)
20             return True
21
22         neighbors[] = getNeighbors(map, current)
23         interactedPoints.add(neighbors[])
24
25         for neighbor in neighbors[]:
26             if (|Δa(current, neighbor)| > m)
27                 skip this neighbor
28             gNeighbor = gScores[current] + distance(current, neighbor)
29
30             if gNeighbor < gScores[neighbor]:
31                 gScores[neighbor] = gNeighbor
32                 path[neighbor] = current
33                 f = gNeighbor + heuristic(neighbor, endPoint)
34                 openSet.put(f, neighbor)
35
36     cannot find path!
37     return False
```

*Mã giả giải thuật tìm kiếm A**

Như các dòng 2, 4, 7, 11. Phiên bản giải thuật A* này sử dụng 4 loại data structure: *Set* (tập hợp các unique), *Priority queue* (hàng đợi ưu tiên), *2D array* (mảng 2 chiều) và *Dictionary* (từ điển).

Các loại data structure này hỗ trợ nhau và thực hiện các nhiệm vụ riêng biệt nhau, cụ thể:

Tên biến		interactedPoints	openSet	gScores	path
Kiểu dữ liệu		**Set**	**PriorityQueue**	**2D_Array**	**Dictionary**
Tính chất của kiểu dữ liệu		Các giá trị không được trùng lắp.	Là Min Heap, không thể truy xuất ngẫu nhiên nhưng phần tử có thể truy xuất luôn có giá trị nhỏ nhất trong tập dữ liệu hiện tại.	Không gian 2 chiều, có thể truy xuất ngẫu nhiên.	Ánh xạ các phần tử từ vế trái đến vế phải. Còn có thể gọi là 'dịch'.
Chức năng		Lưu các vị trí đã đi qua.	Lưu các vị trí tiếp theo cần khám phá (lại).	Lưu giá trị g ở từng vị trí trên bản đồ. $g[i][j]$ được cập nhật khi có một đường khác ngắn hơn đến được $g[i][j]$.	Liên tục cập nhật quãng đường ngắn nhất đi được từ điểm xuất phát đến đích.
Khởi tạo		Tập rỗng. dòng 2	Gồm $startPoint$ với giá trị $priority$ là thấp nhất. dòng 5	Toàn bộ các phần tử đều bằng vô cùng, riêng phần tử tại $startPoint$ = 0 (tốn 0 để đi từ start đến start). dòng 8, 9	$startPoint$ dịch ra $None$ (không có phần tử nào đứng sau start). dòng 12
Chi phí	Add	O(1)			
	Remove	O(1)			

Nhận thấy rằng, dù cho chi phí thêm, tìm kiếm và xóa của cả 4 cấu trúc dữ liệu trên đều xấp xỉ **O(1)**, nhưng trên thực tế mảng 2 chiều vẫn nhanh hơn các cấu trúc dữ liệu còn lại do không phải tốn thêm chi phí băm (*Set* và *Dictionary*) hay sắp xếp (*Priority queue*).

Sau khi khởi tạo 4 bộ nhớ cần thiết, ta thực thi vòng lặp chính. Vòng lặp này được lặp liên tục và chỉ thoát khi không còn node mới trong **openSet** (một số phiên bản gọi là **frontier**, **minHeap**, **waitingList**, ...) được thêm vào, đồng nghĩa với việc không có cách nào để đến được **endPoint**.

Trong mỗi vòng lặp, việc đầu tiên mà thuật toán thực hiện là lấy được điểm nhỏ nhất trong **openSet**, tương đương phần tử có độ ưu tiên cao nhất. Việc này không có nhiều ý nghĩa khi điểm đang xét là **startPoint**, nhưng lệnh này sẽ mang nhiều giá trị hơn khi **openSet** đã thu thập được một danh sách lớn các điểm đang chờ được khám phá (explore). dòng 15

Tiếp đến, ta kiểm tra xem điểm được lấy từ **openSet** có phải là **endPoint** hay không, nếu phải, ta nhận định thuật toán A* đã kết thúc và chuyển qua bước **visualizePath**. dòng 17

Ngược lại, vòng lặp cần được tiếp tục như một phần '**chính thức**' trong giải thuật.

Từ dòng 22 và 23, hàm **getNeighbors** thu về một danh sách các neighbor liền kề với vị trí hiện tại, rồi cập nhật các điểm đó qua **interactedPoints** với ý nghĩa rằng đã tương tác với các điểm đó. Lưu ý rằng, **interactedPoints** là một set và không nhận giá trị duplicated nên ta không cần quan tâm về việc có lỡ thêm một điểm đã được thêm vào **interactedPoints** hay không. dòng 22, 23

Tiếp đến, một vòng lặp khác duyệt từng **neighbor** trong danh sách các **neighbor**. Với mỗi **neighbor**, các hành động sau cần được thực hiện:

1. Kiểm tra xem độ cao giữa **neighbor** và **current** có thỏa mãn điều kiện nhỏ hơn **m** hay không, tức yêu cầu của bài toán. dòng 26
2. Nếu thỏa, nhận giá trị **g** tạm thời mới của **neighbor**, áp dụng định nghĩa **g** của **neighbor** là độ dài quãng đường đi từ **startPoint** đến **current**, cộng cho quãng đường đi từ **current** đến **neighbor**. dòng 28
3. Nếu giá trị **g** mới này nhỏ hơn giá trị **g** cũ, ngụ ý rằng đã tìm được một đường đi khác ngắn hơn để đến **neighbor** này, ta tiếp tục thực hiện các hành động sau:
    a. Cập nhật giá trị **g** trong **gScores** của **neighbor**. dòng 31
    b. Cập nhật đường đi đến **neighbor** là từ **current**. dòng 32
    c. Đưa **neighbor** đó vào **openSet**, với giá trị ưu tiên đúng bằng **f** với:

$$\boldsymbol{f = g + h(neighbor, end)}$$ dòng 33, 34

Các hàm hỗ trợ

Hàm visualizePath

Hàm **visualizePath** thực hiện hành động trực quan hóa quãng đường sau khi đã đến được vạch đích (**endPoint**). Từ điểm **endPoint**, các điểm trước đó liên tục được backtrack thông qua dictionary đã được tạo trước đó, tức biến **path**. Các điểm ở trước đều đã được dịch ở dòng 32, và tạo thành một xâu các điểm liên tiếp đi từ **startPoint** đến **endPoint**. Như vậy, đó chính là quãng đường mà ta thu được.

```
function visualizePath(map, path, current):
map = changeColorChannels(grayscale to RGB)

while current in path is not None:

    map[current.x][current.y] = Yellow
    for neighbor in getNeighbors(map, current):
        map[neighbor.x][neighbor.y] = Yellow

    current = path[current]
save to file
```

Mã giả hàm visualizePath

Trước tiên, bản đồ địa hình cần được **reopen** dưới dạng **RGB** để có thể tô màu khác trắng, xám, đen, vốn đã được **open** dưới dạng **grayscale** trước khi thực hiện hàm A_Star_Algorithm. Sau đó, sau khi tô màu từng phần tử **current**, ta tiếp tục tô các ô xung quanh **current**, điều này có thể thực hiện nhờ hàm **getNeighbors** mà ta đã sử dụng trong giải thuật **A*** trên.

Sau khi tô các ô xung quanh, ta **backtrack** ô đã '**dịch**' ra điểm **current**, tức ô đứng trước **current** trong quãng đường từ **startPoint** đến **endPoint** như đã đề cập ở trên. Việc này được thực hiện liên tục cho đến khi nhận được **phần tử dịch từ current** là **None**, tức **current** là **startPoint**.

Hàm getNeighbors

Hàm **getNeighbors** xác định các trường hợp '**ngoài giới hạn**' (**out of bounds**) của bản đồ và thêm vào danh sách các điểm có thể thêm.

Thay cho mã giả, ta hoàn toàn có thể liệt kê các trường hợp **out of bounds** dưới dạng ô để nhận thấy ý tưởng của hàm một cách trực quan hơn.

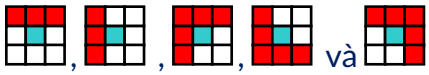
if not								
then add								

Các trường hợp cần kiểm tra của hàm getNeighbors

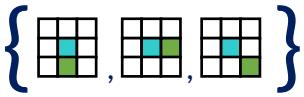
Trong đó:

- : Vị trí hiện tại đang được kiểm tra.
- : Vùng ngoài rìa bản đồ (không có trong bản đồ).
- : Vị trí sẽ được thêm vào danh sách các neighbor của điểm hiện tại.

Ví dụ, tại điểm **map[0][0]**, ta nhận thấy rìa của bản đồ là phần trên và bên trái của điểm được xét, ta loại **bất cứ** trường hợp nào có chạm đến các rìa được nêu trên, tức:

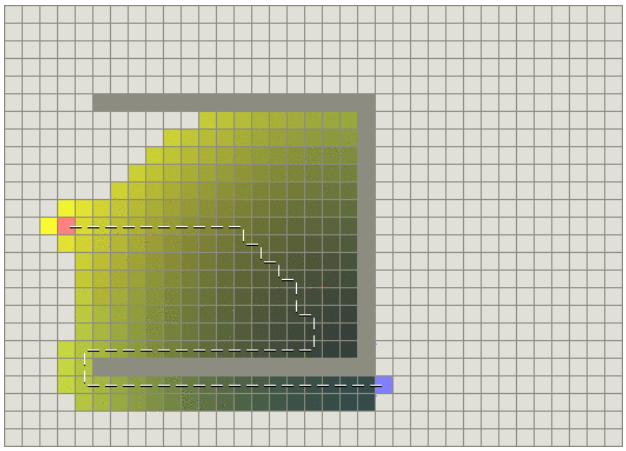


Như vậy, danh sách các **neighbor** thu được tại vị trí **map[0][0]** sẽ là:

getNeighbors(map[0][0]) = 

Các hàm heuristic

Trong đồ án này, hàm định nghĩa độ cao $\Delta a = a(x_1, y_1) - a(x_2, y_2)$ còn có thể được hiểu là dữ liệu dùng để tạo ra "chướng ngại vật". Ở đây, khi muốn di chuyển từ điểm (x_1, y_1) sang (x_2, y_2) trên bản đồ, ràng buộc $\Delta a \leq m$ (với m là dữ liệu input đầu vào) cần phải được thỏa mãn. Với góc nhìn này, bài toán có thể được quy thành một trò chơi quen thuộc – Giải mã mê cung. Như vậy các hàm heuristic dưới có thể được xây dựng như trong bối cảnh của không gian hai chiều.



Ví dụ về một bài toán tìm đường đi trong mê cung.[2]

Tuy nhiên, ta có thể thêm z_1 là độ cao tại điểm (x_1, y_1) trên bản đồ, hàm ý rằng điểm (x_1, y_1) ở đây sẽ trở thành (x_1, y_1, z_1). Điều này xây dựng bài toán tìm kiếm trong không gian hai chiều đơn thuần trở thành việc tìm đường đi trong không gian ba chiều.

Khi xây dựng bài toán ở góc nhìn này, việc kiểm tra các điều kiện thật sự được tối ưu hơn do còn dựa trên yếu tố độ cao địa hình. Việc này tối ưu hóa không chỉ số điểm chạm mà còn thời gian thực hiện bài toán.

```
Heuristic1: Finding path using Manhattan Distance...
DONE!
317.5391052434012
10566
Runtime: 0.7734978199005127 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
317.5391052434012
17411
Runtime: 1.2200696468353271 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
444.85281374238616
573
Runtime: 0.019001245498657227 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
317.5391052434012
18553
Runtime: 1.3947358131408691 second(s)
```

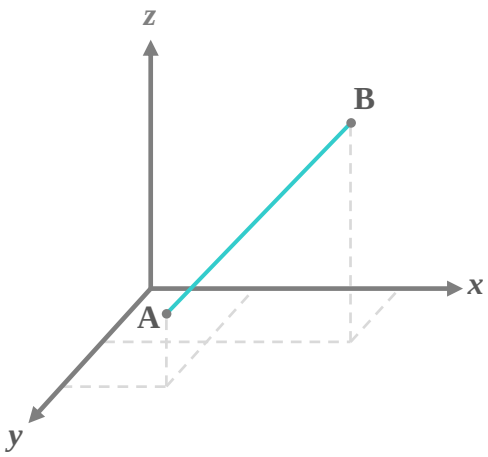
Không gian 3 chiều

```
Heuristic1: Finding path using Manhattan Distance...
DONE!
317.5391052434012
15876
Runtime: 1.1500096321105957 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
317.5391052434012
19661
Runtime: 1.3592090606689453 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
549.1959594928937
403
Runtime: 0.011000394821166992 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
317.5391052434012
18897
Runtime: 1.3269703388214111 second(s)
```

Không gian 2 chiều

[2] http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

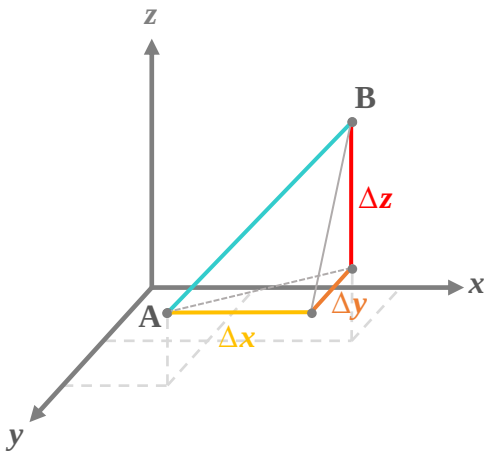
Các hàm heuristic dưới được xây dựng trong không gian ba chiều. Với dòng thứ i, cột thứ j là tọa độ x, y của một điểm, và chiều cao tại điểm đó là tọa độ z trong không gian.



Khoảng cách giữa 2 điểm A, B là đường thẳng màu xanh bạc hà

Hàm heuristic 1 (3D Manhattan Distance)

Hàm heuristic 1 được áp dụng dựa trên phép tính 3D Manhattan Distance, trong bối cảnh mà ta chỉ có thể đi trên các đường song song với trục tọa độ. Đây là một phương pháp tương đối ổn do phù hợp với tình huống bài toán. Việc đo đạc trên từng cặp pixel và hiệu giữa hai độ cao tương đối là như nhau (đều là các số nguyên, hiệu càng lớn tức khoảng cách càng lớn). Lưu ý rằng đường đi từ A đến B không phải là một đường thẳng mà luôn là các đường song song với trục tọa độ, tổng của 3 khoảng cách này đúng bằng khoảng cách để đi từ A đến B trong thực tế khi không có chướng ngại vật.

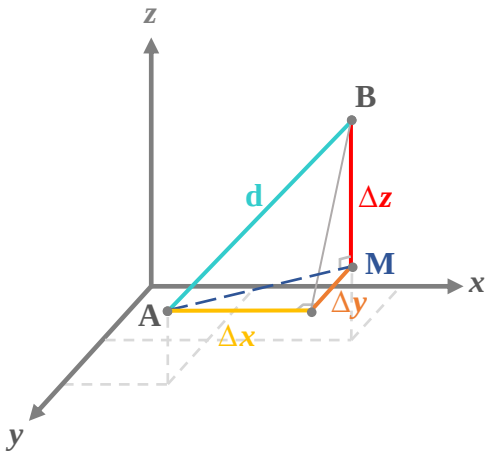



```
function heuristic1(map, A, B):
    Δx = |A.x - B.x|
    Δy = |A.y - B.y|
    Δz = |map[A.x][A.y] - map[B.x][B.y]|
    return Δx + Δy + Δz
```

Mã giả của hàm heuristic 1

Hàm heuristic 2 (Euclidean Distance)

Hàm heuristic 2 áp dụng phương pháp khoảng cách Euclidean, một phương pháp phổ thông trong hình học để xác định khoảng cách giữa 2 điểm bằng chính đoạn thẳng nối 2 điểm đó. Việc này có thể thực hiện được bằng cách áp dụng định lý Pytago 2 lần, cụ thể như sau:



$$AM^2 = \Delta x^2 + \Delta y^2$$

$$AB^2 = AM^2 + \Delta z^2 = d^2$$

Vậy:

$$d = \sqrt{AM^2 + \Delta z^2} = \sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}$$

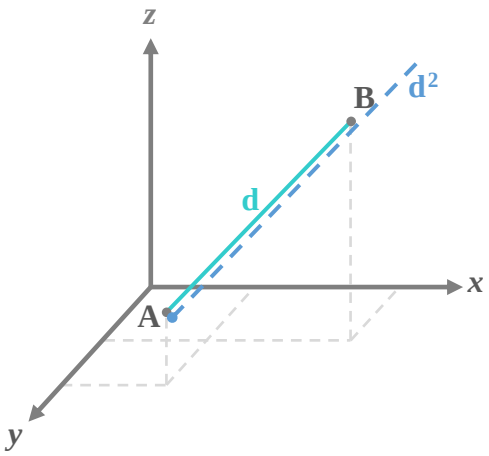
```
function heuristic2(map, A, B):
    Δx = |A.x - B.x|
    Δy = |A.y - B.y|
    Δz = |map[A.x][A.y] - map[B.x][B.y]|
    return √(Δx² + Δy² + Δz²)
```

Mã giả của hàm heuristic 2

Hàm heuristic 3 (Euclidean-Squared Distance)

Việc sử dụng khoảng cách Euclid ở hàm heuristic 2 cho ra một kết quả đúng bằng khoảng cách từ A đến B. Tuy nhiên việc này là không cần thiết do tính thực tế không cao (ta chỉ cần tương đối, không cần tuyệt đối). Mặt khác chi phí của việc lấy căn bậc hai cũng tương đối lớn. Để giảm thiểu các nhược điểm trên, ta loại bỏ hàm căn và chỉ thu về tổng của 3 độ lớn bình phương.

Việc này có nghĩa rằng, giá trị heuristic được trả về sẽ nới giãn ra một khoảng bình phương, khiến cho chi phí của nó áp đảo **g(x)** và khiến **g(x)** hầu như còn rất ít giá trị (khoảng 25% trong **f(x)**). Điều này khiến cho hàm **A*** gần như trở thành **Greedy Best First Search**, độ dài đường đi sẽ tương đối lớn, nhưng thời gian thuật toán được giảm đáng kể (trừ một số trường hợp không cụ thể).




```
function heuristic4(map, A, B):
    Δx = |A.x – B.x|
    Δy = |A.y – B.y|
    Δz = |map[A.x][A.y] – map[B.x][B.y]|
    return Δx² + Δy² + Δz²
```

Mã giả của hàm heuristic 3

Hàm heuristic 4 (Octile Distance)

Hàm heuristic 4 là áp dụng phương pháp khoảng cách Octile – sử dụng phỏng đoán để ước tính khoảng cách giữa 2 ô. Khoảng cách Octile là một biến thể khác của khoảng cách Euclidean – ưu tiên di chuyển theo đường chéo với kỳ vọng là sẽ tối ưu về số điểm chạm hơn Euclidean.

```
function heuristic4(map, A, B):
    Δx = |A.x – B.x|
    Δy = |A.y – B.y|
    Δz = |map[A.x][A.y] – map[B.x][B.y]|
    Max = max(Δx, Δy, Δz)
    Min = min(Δx, Δy, Δz)
    Return Max + (√2-1)*Min
```

Mã giả của hàm heuristic 4

Mở rộng

So sánh các hàm Heuristic dựa trên các kết quả thực nghiệm

Bản đồ địa hình sử dụng: map.bmp.

Với mỗi test case, kết quả được chia thành 2 trường hợp:

1. Trường hợp m là 255 – Tức là có thể di chuyển thoải mái trên bản đồ.
2. Trường hợp m nằm trong đoạn [1,10] – Tức là sẽ bị giới hạn đường đi lại đến mức tối đa, m nằm trong đoạn [0,10] để linh hoạt thay đổi nếu gặp trường hợp không tìm ra được đường đi.

Test case 1: (74;213) đến (96;311) – Test case mặc định

```
(74, 213) (96, 311) 255

Heuristic1: Finding path using Manhattan Distance...
DONE!
Saved map1 successfully!
317.5391052434012
10566
Runtime: 0.7599411010742188 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
Saved map2 successfully!
317.5391052434012
17411
Runtime: 1.2381200790405273 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
Saved map3 successfully!
444.85281374238616
573
Runtime: 0.01900005340576172 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
Saved map4 successfully!
317.5391052434012
18553
Runtime: 1.4660441875457764 second(s)
```

m = 255

```
(74, 213) (96, 311) 3

Heuristic1: Finding path using Manhattan Distance...
DONE!
Saved map1 successfully!
404.4091629284902
9183
Runtime: 0.41268062591552734 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
Saved map2 successfully!
404.4091629284902
13830
Runtime: 0.6085529327392578 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
Saved map3 successfully!
787.5046173579958
6158
Runtime: 2.2571163177490234 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
Saved map4 successfully!
404.4091629284902
14716
Runtime: 0.6049726009368896 second(s)
```

m = 3

Test case 2: (0;0) đến (511;511) – Di chuyển theo đường chéo

```
(0, 0) (511, 511) 255

Heuristic1: Finding path using Manhattan Distance...
DONE!
Saved map1 successfully!
1264.965078838708
61640
Runtime: 5.239453554153442 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
Saved map2 successfully!
1264.1366517139618
146532
Runtime: 11.043800115585327 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
Saved map3 successfully!
2677.624025129231
3096
Runtime: 0.05600404739379883 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
Saved map4 successfully!
1264.1366517139618
158509
Runtime: 11.41411304473877 second(s)
```

m = 255

```
(0, 0) (511, 511) 5

Heuristic1: Finding path using Manhattan Distance...
DONE!
Saved map1 successfully!
1269.8940110268425
60938
Runtime: 4.114989280700684 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
Saved map2 successfully!
1268.4797974644694
145059
Runtime: 8.504738807678223 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
Saved map3 successfully!
2536.913347010579
3265
Runtime: 0.062004804611206055 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
Saved map4 successfully!
1268.4797974644694
155310
Runtime: 9.145047664642334 second(s)
```

m = 5

Test case 3: (0;200) đến (511;200) – Di chuyển theo đường ngang

```
(0, 200) (511, 200) 255

Heuristic1: Finding path using Manhattan Distance...
DONE!
Saved map1 successfully!
1178.5828278447989
119588
Runtime: 9.995380640029907 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
Saved map2 successfully!
1162.0285706997497
200846
Runtime: 16.168599605560303 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
Saved map3 successfully!
2081.5361466505738
4424
Runtime: 0.48356032371520996 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
Saved map4 successfully!
1162.0285706997497
216752
Runtime: 16.77854585647583 second(s)
```

m = 255

```
(0, 200) (511, 200) 3

Heuristic1: Finding path using Manhattan Distance...
DONE!
Saved map1 successfully!
1253.5533905932762
101542
Runtime: 5.385876655578613 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
Saved map2 successfully!
1251.2102448427686
144037
Runtime: 6.77791690826416 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
Saved map3 successfully!
2926.638743754989
12390
Runtime: 5.562391519546509 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
Saved map4 successfully!
1251.2102448427686
148845
Runtime: 6.9135658740997314 second(s)
```

m = 3

Test case 4: (200;0) đến (200;511) – Di chuyển theo đường dọc

```
(200, 0) (200, 511) 255

Heuristic1: Finding path using Manhattan Distance...
DONE!
Saved map1 successfully!
1087.5239533417537
121166
Runtime: 9.876083612442017 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
Saved map2 successfully!
1083.8670990922612
177870
Runtime: 14.266540288925171 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
Saved map3 successfully!
1834.6782822743046
2334
Runtime: 0.05200386047363281 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
Saved map4 successfully!
1083.8670990922612
192276
Runtime: 14.891162395477295 second(s)
```

m = 255

```
(200, 0) (200, 511) 4

Heuristic1: Finding path using Manhattan Distance...
DONE!
Saved map1 successfully!
1096.6955262170075
115568
Runtime: 6.701682090759277 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
Saved map2 successfully!
1093.038671967515
171396
Runtime: 9.993539094924927 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
Saved map3 successfully!
1858.2935059634542
3023
Runtime: 0.09700942039489746 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
Saved map4 successfully!
1093.038671967515
185242
Runtime: 10.893048524856567 second(s)
```

m = 4

Test case 5: (511;511) đến (200;180) – Di chuyển có chênh lệch độ cao (thấp đến cao)

```
(511, 511) (200, 180) 255

Heuristic1: Finding path using Manhattan Distance...
DONE!
Saved map1 successfully!
816.6564209736049
30985
Runtime: 11.148430585861206 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
Saved map2 successfully!
785.1417023478435
127455
Runtime: 14.551811933517456 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
Saved map3 successfully!
1358.5214280248113
3464
Runtime: 0.3623661994934082 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
Saved map4 successfully!
785.1417023478435
143503
Runtime: 15.256705522537231 second(s)
```

m = 255

```
(511, 511) (200, 180) 4

Heuristic1: Finding path using Manhattan Distance...
DONE!
Saved map1 successfully!
933.7447327281742
112826
Runtime: 19.945919275283813 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
Saved map2 successfully!
916.7396820942905
178981
Runtime: 13.268605947494507 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
Saved map3 successfully!
1612.9722215136446
5035
Runtime: 0.5351054668426514 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
Saved map4 successfully!
916.7396820942905
189560
Runtime: 14.110506534576416 second(s)
```

m = 4

Test case 6: (200;180) đến (0;511) – Di chuyển có chênh lệch độ cao (cao đến thấp)

```
(200, 180) (0, 511) 255

Heuristic1: Finding path using Manhattan Distance...
DONE!
Saved map1 successfully!
862.6147904132631
29059
Runtime: 2.1798441410064697 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
Saved map2 successfully!
861.7863632885169
64383
Runtime: 5.1552581787109375 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
Saved map3 successfully!
1396.0117600329336
1526
Runtime: 0.030999422073364258 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
Saved map4 successfully!
861.7863632885169
75118
Runtime: 6.114228248596191 second(s)
```

m = 255

```
(200, 180) (0, 511) 4

Heuristic1: Finding path using Manhattan Distance...
DONE!
Saved map1 successfully!
892.0290039756362
30982
Runtime: 1.5587942600250244 second(s)

Heuristic2: Finding path using Euclidean Distance...
DONE!
Saved map2 successfully!
891.7863632885169
63545
Runtime: 3.2877726554870605 second(s)

Heuristic3: Finding path using Euclidean-Squared Distance...
DONE!
Saved map3 successfully!
1417.5092347159923
6562
Runtime: 4.682286262512207 second(s)

Heuristic4: Finding path using Diagonal Distance (Octile)...
DONE!
Saved map4 successfully!
891.7863632885169
72797
Runtime: 3.768085479736328 second(s)
```

m = 4

Từ những trường hợp trên, có thể rút ra kết luận về các hàm Heuristic.

Heuristic	Manhattan	Euclidean	Squared-Euclidean	Octile
Ưu điểm	▪ Tối ưu hóa được số điểm chạm so với Euclidean hay Octile. ▪ Thời gian chạy thuật toán tốt hơn Euclidean và Octile.	▪ **Tối ưu hóa đường đi tốt nhất.**	▪ **Tối ưu tốt nhất số điểm chạm.** ▪ Thời gian chạy thuật toán nhanh nhất.	▪ Là một biến thể khác của Euclidean nên vẫn tối ưu hóa được đường đi nhất.
Nhược điểm	▪ Không thật sự ổn định, trong 1 số trường hợp đường đi tìm được có chi phí tốt hơn hoặc bằng với Euclidean. ▪ Trong 1 số trường hợp thì thời gian lớn hơn Euclidean.	▪ Thời gian chạy thuật toán và số điểm chạm không tối ưu ▪ **Chi phí tính toán cao** (hàm căn)	▪ **Đường đi tìm được có chi phí quá cao.** ▪ Nếu bản đồ có kích thước quá lớn, có thể bị tràn dữ liệu vì phải tính toán số lớn.	▪ Không tối ưu được số điểm chạm với thời gian chạy so với Euclidean.

Phương pháp mà thế giới sử dụng

Từ thuật toán A* search, tính đến nay thì trên thế giới đã phát triển ra nhiều thuật toán mới như: IDA*, Theta*, HPA*, D*, … Dù đã được tối ưu nhưng mỗi thuật toán đều có ưu nhược điểm riêng so với A* và được áp dụng vào những bài toán riêng biệt. Bên cạnh đó, có một thuật toán có thể tối ưu hơn A* nếu đáp ứng được đủ các điều kiện ràng buộc – **Jump Point Search.**

Trong báo báo này đã đề cập phần **Adversarial Search** – với thuật toán Minimax và chiến lược Alpha-beta Pruning. Về cơ bản, **Jump Point Search (JPS)** được phát triển theo mô hình như vậy, tức là với một đồ thị hoặc ma trận với một hàm heuristic cho trước – Trong quá trình expand, tìm kiếm đường đi thì **JPS** sẽ lược bỏ đi các nút không cần thiết, giúp cho việc di chuyển từ điểm bắt đầu đến điểm kết thúc được diễn ra nhanh hơn mà không cần mở rộng ra các nút không cần thiết. **JPS** vẫn giữ nguyên phần lõi của thuật toán A*, nhưng nó được thêm một hàm với chức năng quét hết tất cả điểm xung quanh, sau đó tìm ra điểm nhảy và di chuyển tới đó. Do chức năng quét các điểm xung quanh và tìm ra điểm nhảy có thể được nới rộng ra hơn 8 điểm xung quanh điểm đó, tức có thể **di chuyển nhiều hơn 1 nút trên đồ thị hoặc ma trận**, điều mà không thể được áp dụng trong phạm vi của đồ án để tối ưu giải thuật A* này. Dù sao đi nữa, thuật toán **JPS** có thể được xem là thuật toán tối ưu nhất so với các thuật toán được liệt kê ở trên, nó tối ưu được thời gian cùng với số điểm chạm hơn thuật toán **A*** vì hạn chế được việc tìm kiếm và cập nhật các nút vào trong Priority Queue.

Hiện nay, ứng dụng tìm kiếm đường đi tốt nhất và phổ biến nhất chính là Google Maps – ứng dụng có thể tìm được hai đến ba đường đi phù hợp cho người dùng với thời gian chưa tới 2 giây, thậm chí có những kết quả sẽ đi kèm với những thông tin lưu ý như: Ùn tắc giao thông ở thời điểm hiện tại, công trình đang được thi công, ….

Thuật toán tìm kiếm đường đi của Google Maps được xây dựng và phát triển dựa trên thuật toán A* search với một trong các hàm heuristic được áp dụng công thức **tính khoảng cách Haversine** – Tính khoảng cách giữa 2 điểm trên một mặt cầu, Trên lý thuyết hàm heuristic Haversine hoàn toàn phù hợp với đặc điểm địa hình của Trái Đất, nhưng về mặt thực tế, không phải hai điểm bất kì nào cũng sẽ có bề mặt bằng phẳng để dễ dàng di chuyển, do đó việc tính toán khoảng cách sai lệch dẫn đến thông tin sai là điều khó tránh khỏi.

References

Programming

Tech With Tim - A* Pathfinding Visualization

The Coding Train - A* Pathfinding Algorithm - Part 1

Adrian Rosebrock - OpenCV Getting and Setting Pixels

tutorialkart - OpenCV Python Save Image

hafeezulkareem - Calculate Time taken by a Program to Execute in Python

Heuristic Function

Maarten Grootendorst – 9 Distance Measures in Data Science

Wikipedia – Harversine formula

Amit Patel – Amit's A* Pages

Bobby Anguelov – Optimizing the A* algorithm

An Zhang, Chong Li, Wenhao Bi - Rectangle expansion A∗ pathfinding for grid maps

Daniel Harabor, Alban Grastien - Online Graph Pruning for Pathfinding on Grid Maps

https://stackoverflow.com/questions/6937459/which-algorithm-does-google-maps-use-to-compute-the-direction-between-2-points