COMPARISONS

# Comparing Different Search Strategies: Maze

1. Giới thiệu bài toán

* Bài toán tìm đường trong mê cung (Maze solving Problem)
* Việc giải quyết bài toán này có thể được thực hiện bằng nhiều chiến lược tìm kiếm khác nhau. Mỗi thuật toán có ưu – nhược điểm riêng về thời gian chạy, bộ nhớ sử dụng và khả năng tìm ra lời giải tối ưu.
* Các chiến lược tìm kiếm:
  + BFS (Breadth - First Search) – Tìm kiếm theo chiều rộng
  + DFS (Depth – First Search) – Tìm kiếm theo chiều sâu
  + GBFS (Greedy Best – First Search) – Tìm kiếm tham lam
  + A\* - Phiên bản có trọng số của A\*
  + Weighted A\* - Phiên bản có trọng số của A\*
  + IDS (Iterative Deepening Search) – Tìm kiếm sâu dần

1. Cài đặt và biểu diễn thuật toán

A black background with white x and blue x

AI-generated content may be incorrect.

Mê cung được biểu diễn như hình trên `L\_maze.txt`:

* X đại diện cho tường (không thể đi qua).
* S là điểm bắt đầu
* G là điểm đích
* Khoảng trống là ô có thể đi

1. Các chiến lược tìm kiếm
   1. Breadth – First Search (BFS) – Tìm kiếm theo chiều rộng

Ý tưởng:

BFS duyệt từng lớp theo khoảng cách từ điểm bắt đầu. Nó sử dụng hàng đợi (queue) để đảm bảo các nút gần hơn được duyệt trước.

Ưu điểm:

* Đảm bảo tìm được đường đi ngắn nhất (nếu chi phí di chuyển bằng nhau).
* Dễ cài đặt

Nhược điểm:

* Tốn nhiều bộ nhớ do lưu tất cả các nút đã duyệt.
* Không tối ưu nếu mê cung lớn

Mã giả:

tree\_search.set\_order("NESW")

%time result = best\_first\_search(maze, strategy = "BFS", debug = False, vis = False)

show\_path(maze, result)

result['actions']

* Xác định thứ tự mở rộng nút (Bắt, Đông, Nam, Tây). Điều này ảnh hướng đến các BFS duyệt mê cung.
* Gọi hàm tìm kiếm theo chiến lược BFS.
* Hiển thị kết quả đường đi trong mê cung.
* Cho biết chuỗi các hành động (ví dụ: Lên – Phải – Xuống…) để đi từ S đến G.

Kết quả:

A screenshot of a computer game

AI-generated content may be incorrect.

* 1. Depth – First Search (DFS) – Tìm kiếm theo chiều sâu

Ý tưởng:

DFS đi sâu vào một nhánh cho đến khi không thể đi tiếp, sau đó quay lại (backtrack). Sử dụng ngăn xếp (stack) hoặc đệ quy.

Ưu điểm:

* Cài đặt đơn giản, dùng ít bộ nhớ.
* Tìm được đường đi (không nhất thiết ngắn nhất).

Nhược điểm:

* Có thể đi lạc vào nhánh sâu không có lời giải.
* Không đảm bảo đường đi ngắn nhất.

Mã giả:

tree\_search.set\_order("NESW")

#tree\_search.set\_order("SENW")

#tree\_search.set\_order("WSEN")

#tree\_search.set\_order(random=True)

%time result = best\_first\_search(maze, strategy = "DFS", debug = False, vis = False)

show\_path(maze, result)

Kết quả:

A screenshot of a game

AI-generated content may be incorrect.

* 1. Greedy Best – First Search – Tìm kiếm tham lam

Ý tưởng:

Ưu tiên mở rộng nút gần đích nhất dựa trên hàm heuristic h(n)

* Hàm tính khoảng cách Manhattan giữa 2 vị trí:

def manhattan(pos1, pos2):

    """returns the Manhattan distance between two positions"""

    return(np.sum(np.abs(np.subtract(pos1, pos2))))

print(manhattan((0,0), (1,1)))

print(euclidean([0,0], [1,1]))

Khoảng cách Manhattan là tổng độ lệch tuyệt đối trên từng trục tọa độ - hay nói cách khác là số bước di chuyển ngang + dọc cần thiết nếu chỉ đi theo 4 hướng (trên – dưới – trái – phải).

**Công thức:**

Đơn giản, nhanh và rất thường dùng làm hàm heuristic trong bài toán tìm đường.

* Hàm tính khoảng cách Euclid giữa hai điểm:

def euclidean(pos1, pos2):

    """returns the Euclidean distance between two positions"""

    return(np.sqrt(np.sum(np.square(np.subtract(pos1, pos2)))))

print(euclidean([0,0], [1,1]))

Khoảng cách Euclid là “độ dài đường thẳng” nối giữa pos1 và pos2

**Công thức:**

Cung cấp thông tin “thực” hơn về khoảng cách vật lý giữa hai điểm, nhưng tốn tính toán hơn một chút.

* + - Khi giải bài toán mê cungg với 4 hướng di chuyển, Manhattan thường là lựa chọn tối ưu. Nếu mô hình cho phép di chuyển chéo hoặc cần độ chính xác cao hơn, Eulicean sẽ phù hợp hơn.
  1. A\* Search – Tìm kiếm tối ưu có heuristic

Ý tưởng:

Kết hợp giữa chi phí đã đi (g(n)) và ước lượng còn lại (h(n)):

Ưu điểm:

* Tối ưu hoàn chỉnh (tìm được đường đi ngắn nhất nếu h chấp nhận được).
* Hiệu quả hơn BFS trong nhiều trường hợp.

Nhược điểm:

* Phức tạp hơn trong cài đặt.
* Tốn bộ nhớ hơn DFS.

Mã giả:

tree\_search.set\_order("NESW")

#tree\_search.set\_order("SENW")

#tree\_search.set\_order("WSEN")

#tree\_search.set\_order(random=True)

%time result = best\_first\_search(maze, strategy = "A\*", debug = False, vis = False)

show\_path(maze, result)

Kết quả:

A screenshot of a computer game

AI-generated content may be incorrect.

1. So sánh các thuật toán

* BFS và A\* là lựa chọn tối ưu nếu yêu cầu đường đi ngắn nhất.
* DFS phù hợp với không gian nhỏ hoặc khi cần giải nhanh không quan tâm tối ưu.
* Greedy Best – First cho kết quả nhanh nhưng không đảm bảo chính xác.
* Trong thực tế, A\* là thuật toán cân bằng giữa tốc độ và chất lượng đường đi.

# Comparing BFS and A\* Search

1. Giới thiệu

* BFS: là thuật toán tìm kiếm không thông tin (uninformed search), mở rộng tìm kiếm theo lớp (concentric circles) từ điểm bắt đầu cho đến khi tìm thấy mục tiêu. Nó đảm bảo tính tối ưu (optimal) nhưng có thể tốn nhiều bộ nhớ và thời gian do khám phá nhiều trạng thái.
* A\*: là thuật toán tìm kiếm có thông tin (informed search), sử dụng hàm heuristic để ước lượng khoảng cách đến mục tiêu, giúp tập trung vào các đường đi tiềm năng hơn. Nó cũng tối ưu nếu heuristic là admisssible (không ước lượng quá cao).

1. Phương pháp thực hiện
   1. Chuẩn bị dữ liệu và môi trường

* Đọc mê cung:

import numpy as np

import maze\_helper as mh

#f = open("small\_maze.txt", "r")

#f = open("medium\_maze.txt", "r")

#f = open("large\_maze.txt", "r")    # this has only one solution!

#f = open("open\_maze.txt", "r")

#f = open("empty\_maze.txt", "r")

#f = open("empty\_2\_maze.txt", "r")

#f = open("loops\_maze.txt", "r")

f = open("L\_maze.txt", "r")

maze\_str = f.read()

maze = mh.parse\_maze(maze\_str)

* Cài đặt heuristic và thứ tự mở rộng

from tree\_search\_solution import \*

import tree\_search\_solution as tree\_search

# used heuristic

tree\_search.heuristic = manhattan

#tree\_search.heuristic = euclidean

# order in which we add new states to the frontier

#tree\_search.set\_order("NESW")

tree\_search.set\_order(random=True)

* Tạo và hiển thị animation minh họa quá trình tìm kiếm đường đi trong mê cung

import numpy as np

from matplotlib import colors

from matplotlib import animation, rc

from IPython.display import HTML

rc('animation', html='html5')

# numpy comparison warnings

import warnings

warnings.simplefilter(action='ignore', category=FutureWarning)

def maze\_to\_matrix(maze):

    """convert a maze a numeric numpy array for visualization via imshow."""

    # make a deep copy first so the original maze is not changed

    maze = np.copy(maze)

    # Converts all tile types to integers

    maze[maze == ' '] = 0

    maze[maze == 'X'] = 1 # wall

    maze[maze == 'S'] = 2 # start

    maze[maze == 'G'] = 3 # goal

    maze[maze == 'P'] = 4 # position/final path

    maze[maze == '.'] = 5 # explored squares

    maze[maze == 'F'] = 6 # frontier

    maze = maze.astype(int)

    return(maze)

# Based on show\_maze but modified to generate animation (suggested by Troy Jeffrey McNitt)

# Sadly I can not embed the animations in the PDF I have to submit :(

def animate\_maze(result, repeat = False):

        """Build an animation from a list of mazes. Assumes that results has the elements:

           path, reached, actions and maze\_anim with a list of maze arrays."""

        if result['path'] != None:

            print(f"Path length: {len(result['path'])-1}")

            print(f"Reached squares: {len(result['reached'])}")

            print(f"Action sequence: {result['actions']}")

        else:

            print("No solution found.")

        mazes = result['maze\_anim']

        cmap = colors.ListedColormap(['white', 'black', 'blue', 'green', 'red', 'gray', 'orange'])

        goal = find\_pos(mazes[0], 'G')

        start = find\_pos(mazes[0], 'S')

        mazes = [maze\_to\_matrix(m) for m in mazes]

        fig, ax = plt.subplots()

        im = ax.imshow(maze\_to\_matrix(mazes[0]), cmap = cmap, norm = colors.BoundaryNorm(list(range(cmap.N + 1)), cmap.N))

        plt.text(start[1], start[0], "S", fontsize = 10, color = "white",

                horizontalalignment = 'center',

                verticalalignment = 'center')

        plt.text(goal[1], goal[0], "G", fontsize = 10, color = "white",

                horizontalalignment = 'center',

                verticalalignment = 'center')

        def step(i):

                im.set\_array(maze\_to\_matrix(mazes[i]))

                return([im])

        ani = animation.FuncAnimation(

            fig,

            step,

            frames = len(mazes),

            repeat = repeat

        )

        plt.close()

        return ani

* Hàm animation:
  + Sử dụng matplotlib.animation để tạo video từ danh sách mê cung (maze\_anim).
  + Phân tích: chuyển mê cung thành ma trận số (0: trống, 1: tường, 2: S, 3: G, 4: vị trí, 5: đã khám phá, 6: biên giới). Sử dụng colormap để hiển thị màu sắc (trắng, đen, xanh, xanh lá, đỏ, xám, cam).
  + Tạo animation với từng frame là một trạng thái mê cung, giúp visualize quá trình mở rộng.
  + Ưu điểm: Dễ quan sát sự khác biệt giữa BFS (mở rộng) và A\* (mở rộng hướng mục tiêu). Nhược điểm: không nhúng được và DFS.
* Các hàm phụ:
  + `maze\_to\_matrix(maze)`: chuyển mê cung thành ma trận số để vẽ.
  + `find\_pos(maze, char)`: tìm vị trí của ký tự (S hoặc G).
  1. Thực thi
* Chạy `%time` để đo thời gian.
* Mê cung: “L\_maze.txt” (mê cung hình L, chỉ có một giải pháp duy nhất).
* Kết quả được hiển thị ma animation và các chỉ số: độ dài đường đi, số ô khám phá chuỗi hành động.

1. Kết quả thực nghiệm
   1. Kết quả BFS

A white background with black text

AI-generated content may be incorrect.

BFS khám phá gần như toàn bộ mê cung (139/tổng số ô trống), vì nó mở rộng đồng đều mà không biết vị trí mục tiêu. Animation cho thấy sự mở rộng theo lớp từ S, tạo thành “các vòng tròng đồng tâm”. Lưu trữ toàn bộ cây (gray squares) dẫn để tốn bộ nhớ.

* 1. Kết quả A\*

A black text on a white background

AI-generated content may be incorrect.

A\* nhanh hơn gấp 4 – 5 lần nhờ heuristic hướng dẫn, chỉ khám phá 40 ô (giảm 71% so với BFS). Animation cho thấy sự mở rộng tập trung về phía G, heuristic giảm dần khi gần mục tiêu. Tuy nhiên, hiệu suất phụ thuộc vào heuristic; Manhattan rất tốt ở đây.

1. So sánh

* Hiệu suất:
  + Khám phá 142 trạng thái (bao gồm frontier), tốn thời gian và bộ nhớ do không có thông tin hướng dẫn.
  + A\*: Chỉ 42 trạng thái, nhanh hơn vì ưu tiên nút có f(n) = g(n) + h(n) thấp nhất. Tiết kiệm bộ nhớ nhờ cây nhỏ hơn.
* Ưu nhược điểm:
  + BFS đơn giản, không cần heuristic, nhưng không hiệu quả với không gian lớp (có thể gặp vấn đề bộ nhớ).
  + A\* hiệu quả hơn với heuristic tốt, nhưng cần thiết kế heuristic admissible. Nếu heuristic kém, có thể tệ hơn BFS.
* Tác động của heuristic: Manhattan phù hợp vì mê cung là lưới vuông góc. Nếu dùng Eulidean, có thể khác biệt nhỏ nhưng vẫn rất tốt.
* Gợi ý cải thiện: Với không gian lớn hơn, dùng Weighted A\* (tăng trọng số heuristic) hoặc Iterative Deepening để giảm bộ nhớ.

1. Kết luận:

Qua notebook thì ta thấy A\* vượt trội hơn BFS trong mê cung này nhờ giảm số trạng thái khám phá (từ 139 xuống còn 40) và thời gian chạy, trong khi vẫn giữ tính tối ưu. Điều này chứng minh lợi ích của informed search. Tuy nhiên, BFS vẫn hữu ích cho các vấn đề không có heuristic tốt.