Dự Án So Sánh Thuật Toán Tìm Kiếm

Game Pacman

Nhóm 6

April 19, 2025

Tóm tắt

Báo cáo này trình bày kết quả của dự án "Cơ sở trí tuệ nhân tạo - CSC14003" - một dự án lập trình trong đó nhóm chúng em đã triển khai bốn thuật toán tìm kiếm khác nhau (BFS, DFS, UCS, và A*) để điều khiển các con ma trong trò chơi Pac-Man. Dự án này nhằm mục đích phân tích và so sánh hiệu suất của các thuật toán tìm đường khác nhau trong một môi trường động.

Chúng em đã phát triển một mô phỏng trò chơi Pac-Man trong đó mỗi con ma được điều khiển bởi một thuật toán tìm kiếm riêng biệt: Blue Ghost sử dụng BFS, Pink Ghost sử dụng DFS, Orange Ghost sử dụng UCS, và Red Ghost sử dụng A*. Các thuật toán được đánh giá dựa trên ba tiêu chí: thời gian tìm kiếm, bộ nhớ sử dụng và số nút được mở rộng.

Kết quả thí nghiệm của chúng em cho thấy [tóm tắt kết quả chính, ví dụ: "BFS tìm ra đường đi ngắn nhất nhưng tiêu tốn nhiều bộ nhớ, trong khi A* cung cấp sự cân bằng tốt nhất giữa hiệu quả thời gian và tối ưu hóa đường đi"]. Ngoài ra, chúng em cũng đã triển khai thành công các tính năng nâng cao như điều khiển đồng thời nhiều con ma và cho phép người chơi điều khiển Pac-Man trong thời gian thực.

Báo cáo này mô tả chi tiết về thuật toán, triển khai, kết quả thí nghiệm, và các phân tích so sánh giữa các thuật toán, cung cấp cái nhìn sâu sắc về ứng dụng các thuật toán tìm kiếm trong game AI.

Contents

1 Kế Hoạch Đồ Án và Phân Công Nhiệm Vụ

1.1 Trách Nhiệm Nhóm

MSSV	Thành Viên	Nhiệm Vụ Được Giao	Tỷ Lệ Hoàn Thành
22120050	Hồ Mạnh Đào	Mã UCS, Cập nhật Readme file cho đồ án.	100%
22120113	Nguyễn Việt Hoàng	Mã BFS, Tạo Menu Game, Tính điểm trong Game, Phân tích thời gian và bộ nhớ.	100%
22120115	Đỗ Thái Học	Mã DFS, Tạo giao diện cho Game, Phân tích thời gian và bộ nhớ. Quay video.	100%
22120418	Huỳnh Trần Ty	Mã A*. Phân tích thời gian và bộ nhớ. Vẽ Biểu đồ, Đồ thị và Viết báo cáo.	100%

Table 1: Phân Phối Nhiệm Vụ và Tỷ Lệ Hoàn Thành

1.2 Link demo

- Demo: Link video demo

2 Mô Tả Thuật Toán

2.1 Thuật Toán Tìm Kiếm Theo Chiều Rộng (BFS)

Tìm kiếm theo chiều rộng là một thuật toán để duyệt hoặc tìm kiếm cấu trúc dữ liệu cây hoặc đồ thị. Nó bắt đầu từ một nút được chọn (gốc) và khám phá tất cả các nút lân cận ở độ sâu hiện tại trước khi chuyển sang các nút ở mức độ sâu tiếp theo.

2.1.1 Thuật Toán Ghost BFS Search

```
function Ghost_BFS_Search(vị_trí_bắt_đầu, đích, tiles, vị_trí_cấm):

    rows, cols <- kích thước của tiles
    visited <- tập rỗng
    queue <- hàng đợi chứa (vị_trí_bắt_đầu, [vị_trí_bắt_đầu])
    số_lượng_nodes_mở <- 0
    max_queue_size <- 1
    start_time <- thời gian hiện tại

    while queue không rỗng:
        current, path <- lấy phần tử đầu của queue
        tăng số_lượng_nodes_mở lên 1

        nếu current == goal:
```

```
            end_time <- thời gian hiện tại
            search_time <- end_time - start_time
            trả về (path, số_lượng_nodes_mở)

        nếu current đã nằm trong visited:
            tiếp tục

        thêm current vào visited

        (x, y) <- current

        với mỗi (dx, dy) trong {(-1,0), (1,0), (0,-1), (0,1)}:
            (nx, ny) <- (x+dx, y+dy)
            nếu (nx, ny) nằm trong lưới và không phải tường và
               chưa được visited và
               (vị_trí_cấm khác (nx, ny) hoặc current là vị_trí_bắt_đầu):

                thêm ((nx, ny), path + [(nx, ny)]) vào queue
                max_queue_size <- max(max_queue_size, độ dài queue)

    trả về ([], 0)
```

2.1.2 Độ Phức Tạp Thời Gian và Không Gian

- Độ Phức Tạp Thời Gian: $O(V + E)$ trong đó V là số đỉnh và E là số cạnh
- Độ Phức Tạp Không Gian: $O(V)$ để lưu trữ hàng đợi và tập hợp đã thăm
- Hình ảnh trong game, thuật toán BFS tìm kiếm Pacman:

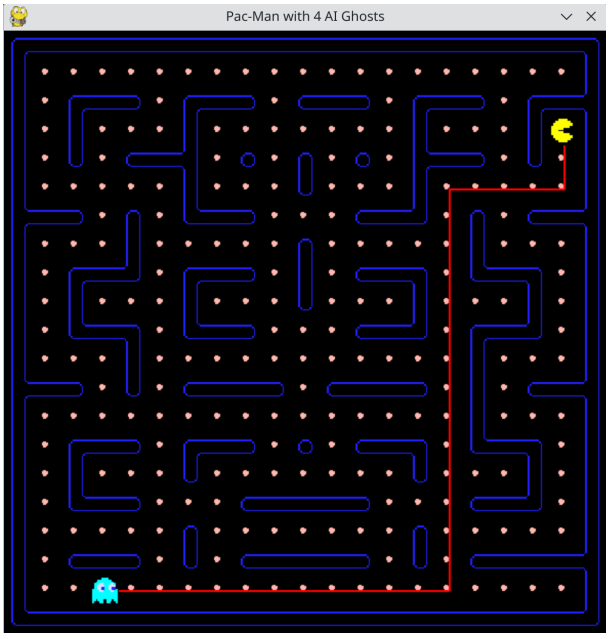

Figure 1: Thuật toán BFS tìm kiếm đường đi cho ghost tới Pacman

2.2 Thuật Toán Tìm Kiếm Theo Chiều Sâu (DFS)

Tìm kiếm theo chiều sâu là một thuật toán để duyệt hoặc tìm kiếm cấu trúc dữ liệu cây hoặc đồ thị. Thuật toán bắt đầu tại nút gốc và khám phá càng xa càng tốt dọc theo mỗi nhánh trước khi quay lại.

2.2.1 Thuật Toán Ghost DFS Search

```
function Ghost_DFS_Search(vị_trí_bắt_đầu, đích, tiles, vị_trí_cấm):

    rows, cols <- kích thước của tiles
    visited <- tập rỗng
    stack <- ngăn xếp chứa start
    trace <- ánh xạ rỗng
    số_lượng_nodes_mở <- 0

    while stack không rỗng:
        current <- lấy phần tử trên cùng của stack
        tăng số_lượng_nodes_mở lên 1

        nếu current == goal:
            path <- []
            trong khi current != start:
                thêm current vào path
                current <- trace[current]
            thêm start vào path
            đảo ngược path
            trả về (path, số_lượng_nodes_mở)

        (x, y) <- current

        với mỗi (dx, dy) trong {(0,1), (0,-1), (-1,0), (1,0)}:
            (next_x, next_y) <- (x+dx, y+dy)
            next_pos <- (next_x, next_y)

            nếu next_pos nằm trong lưới và không phải tường và
               chưa được visited và
               (next_pos khác vị_trí_cấm hoặc current là vị_trí_bắt_đầu):

                thêm next_pos vào stack
                đánh dấu next_pos đã visited
                trace[next_pos] <- current

    trả về ([], 0)
```

2.2.2 Độ Phức Tạp Thời Gian và Không Gian

- Độ Phức Tạp Thời Gian: $O(V + E)$ trong đó V là số đỉnh và E là số cạnh
- Độ Phức Tạp Không Gian: $O(V)$ để lưu trữ ngăn xếp và tập hợp đã thăm
- Hình ảnh trong game, thuật toán DFS tìm kiếm Pacman:

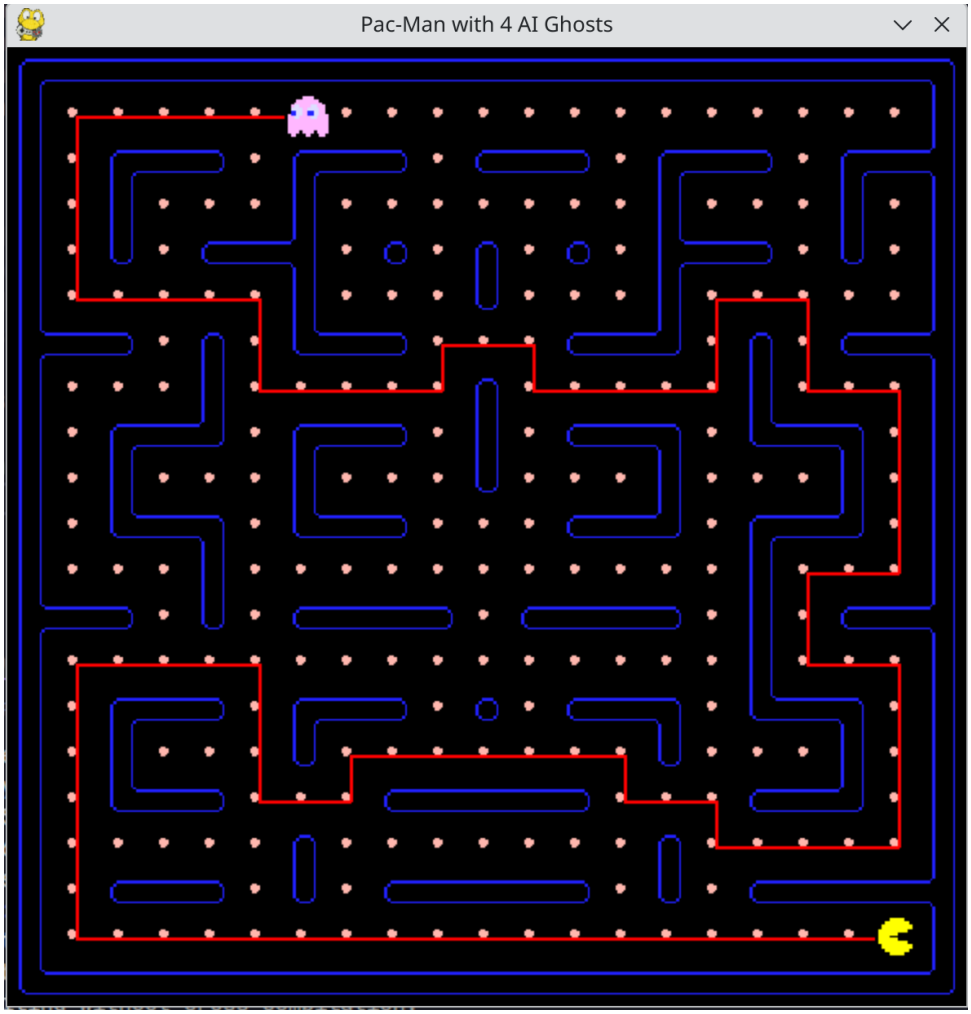

Figure 2: Thuật toán DFS tìm kiếm đường đi cho ghost tới Pacman

2.3 Thuật Toán Tìm Kiếm Chi Phí Đồng Nhất (UCS)

Tìm kiếm chi phí đồng nhất là một thuật toán tìm kiếm đồ thị dùng để tìm đường đi có chi phí thấp nhất từ một nút ban đầu đến nút đích. Thuật toán này mở rộng nút có chi phí tích lũy thấp nhất từ nút bắt đầu.

2.3.1 Thuật Toán Ghost Uniform Cost Search

```
function Ghost_Uniform_Cost_Search(vị_trí_bắt_đầu, đích, tiles, vị_trí_cấm):

    rows, cols <- kích thước của tiles
    visited <- tập rỗng
```

```
pq <- hàng đợi ưu tiên chứa (0, start, [start])
số_lượng_nodes_mở <- 0

while pq không rỗng:
    (cost, current, path) <- lấy phần tử có cost nhỏ nhất từ pq
    tăng số_lượng_nodes_mở lên 1

    nếu current == goal:
        trả về (path, số_lượng_nodes_mở)

    nếu current đã có trong visited:
        tiếp tục

    thêm current vào visited

    (x, y) <- current

    với mỗi (dx, dy) trong {(-1,0), (1,0), (0,-1), (0,1)}:
        (nx, ny) <- (x+dx, y+dy)

        nếu (nx, ny) nằm trong lưới và không phải tường và
           (nx, ny) chưa bị cấm hoặc current là vi_trí_bắt_đầu:

            thêm (cost+1, (nx, ny), path + [(nx, ny)]) vào pq

trả về ([], 0)
```

2.3.2 Cách tính chi phí trong thuật toán Uniform-Cost Search (UCS)

Cách tính chi phí

- Ban đầu, vị trí xuất phát có **chi phí bằng 0**.
- Mỗi khi di chuyển từ một ô hiện tại sang một ô kề cạnh (lên, xuống, trái, phải), **chi phí sẽ tăng thêm một đơn vị**.
- Tổng chi phí để đến một vị trí bất kỳ bằng **tổng chi phí của tất cả các bước di chuyển từ vị trí bắt đầu đến vị trí đó**.

Công thức tổng quát

Giả sử ta có một đường đi từ vị trí bắt đầu đến vị trí n gồm các bước di chuyển liên tiếp qua các ô:

$$start \to node_1 \to node_2 \to \cdots \to node_n$$

Tổng chi phí $g(n)$ để đến vị trí n được tính theo công thức:

$$g(n) = \sum_{i=0}^{n-1} cost(node_i, node_{i+1})$$

Trong đó:

- $cost(node_i, node_{i+1})$ là chi phí di chuyển từ ô $node_i$ đến ô $node_{i+1}$.
- Với bài toán mê cung, ta giả sử mỗi bước di chuyển đều có **chi phí bằng 1**, nên:

$$cost(node_i, node_{i+1}) = 1, \forall i$$

Khi đó, công thức trở thành:

$$g(n) = n$$

Tức là, chi phí để đến một vị trí bất kỳ bằng **số bước di chuyển** từ vị trí bắt đầu đến vị trí đó.

Chiến lược mở rộng

Tại mỗi bước lặp, thuật toán UCS sẽ:

- Chọn vị trí có **tổng chi phí nhỏ nhất** tính đến thời điểm hiện tại để mở rộng.
- Cập nhật chi phí cho các vị trí kề cạnh nếu đường đi mới có chi phí thấp hơn.
- Tiếp tục lặp cho đến khi tìm được đường đi đến vị trí đích hoặc không còn vị trí nào để mở rộng.

2.3.3 Độ Phức Tạp Thời Gian và Không Gian

- Độ Phức Tạp Thời Gian: $O(|E| + |V| \log |V|)$ với hàng đợi ưu tiên dựa trên heap
- Độ Phức Tạp Không Gian: $O(|V|)$ để lưu trữ hàng đợi ưu tiên, chi phí và đường đi
- Hình ảnh trong game, thuật toán UCS tìm kiếm Pacman:

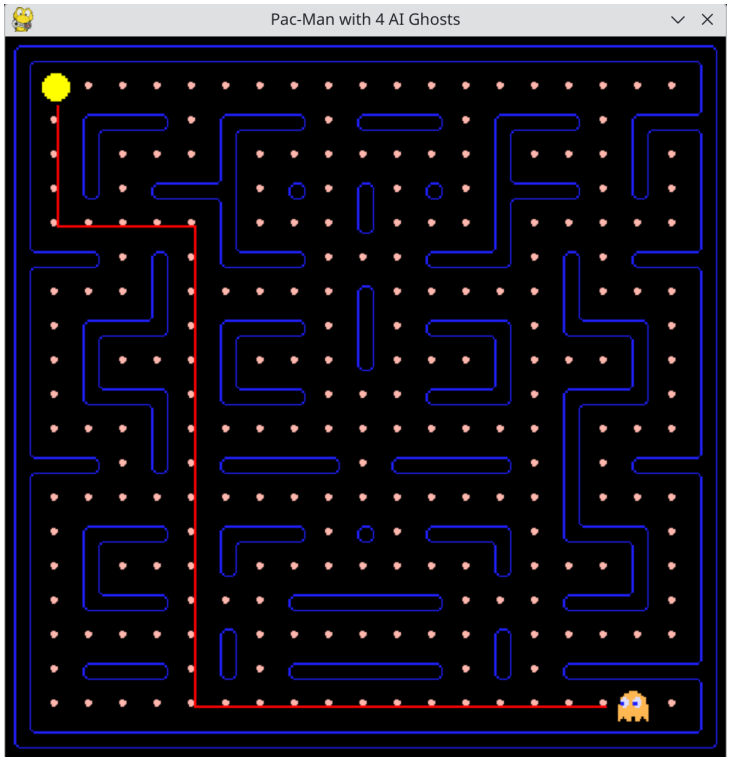

Figure 3: Thuật toán UCS tìm kiếm đường đi cho ghost tới Pacman

2.3.4 So Sánh với BFS

UCS là một phiên bản tổng quát hóa của BFS. Trong khi BFS tìm đường đi với số lượng cạnh ít nhất trong đồ thị không có trọng số, UCS tìm đường đi có chi phí tích lũy thấp nhất trong đồ thị có trọng số. Khi tất cả cạnh có cùng trọng số, UCS hoạt động giống hệt BFS.

2.4 Thuật Toán Tìm Kiếm A*

A* là một thuật toán tìm kiếm có thông tin, tìm đường đi có chi phí thấp nhất từ một nút ban đầu đến nút đích. Nó sử dụng hàm heuristic để ước tính chi phí của đường đi rẻ nhất từ nút hiện tại đến nút đích.

2.4.1 Thuật Toán Ghost A* Search (Mã giả)

```
function Ghost_AStar_Search(tiles, vị_trí_bắt_đầu, đích, vị_trí_cấm,
                            khu_vực_nguy_hiểm, game_state, ghost_index, weight):

    rows, cols <- kích thước của tiles
    open_set <- hàng đợi ưu tiên rỗng
    thêm (0, vị_trí_bắt_đầu) vào open_set
    came_from <- {}
    g_score[vị_trí_bắt_đầu] <- 0
    f_score[vị_trí_bắt_đầu] <- heuristic(vị_trí_bắt_đầu, đích, khu_vực_nguy_hiểm,
                                          weight, game_state, ghost_index)
    số_lượng_nodes_mở <- 0

    while open_set không rỗng:
        current <- lấy nút có f_score nhỏ nhất trong open_set
        tăng số_lượng_nodes_mở lên 1

        nếu current == goal:
            path <- []
            while current tồn tại trong came_from:
                thêm current vào path
                current <- came_from[current]
            thêm vị_trí_bắt_đầu vào path
            trả về (path đảo ngược,  số_lượng_nodes_mở)

        (x, y) <- current

        với mỗi (dx, dy) trong {(-1,0), (1,0), (0,-1), (0,1)}:
            neighbor <- (x+dx, y+dy)
            nếu neighbor nằm trong lưới và không phải tường:
                tentative_g_score <- g_score[current] + 1

                nếu (neighbor chưa có trong g_score hoặc
                    tentative_g_score < g_score[neighbor]) và
                   (vị_trí_cấm khác neighbor hoặc current là vị_trí_bắt_đầu):
```

```
                came_from[neighbor] <- current
                g_score[neighbor] <- tentative_g_score

                f_score[neighbor] <- tentative_g_score +
                                     heuristic(neighbor, đích, khu_vực_nguy_hiểm,
                                               weight, game_state, ghost_index)

                thêm (f_score[neighbor], neighbor) vào open_set

    trả về ([], số_lượng_nodes_mở)
```

2.4.2 Giải thích cách tính chi phí

Trong thuật toán A*, mỗi bước di chuyển được tính toán dựa trên hai thành phần chính:

- **G-Score (chi phí thực tế):** Là tổng chi phí từ vị trí bắt đầu đến vị trí hiện tại. Trong bài toán mê cung, mỗi bước di chuyển từ một ô sang một ô kề cạnh có chi phí cố định bằng 1.
- **F-Score (chi phí ước lượng):** Là tổng của chi phí thực tế và chi phí ước lượng từ vị trí hiện tại đến đích:
$$f(n) = g(n) + h(n)$$
Trong đó:
  - $g(n)$ là chi phí thực tế từ vị trí xuất phát đến vị trí n.
  - $h(n)$ là giá trị heuristic (ước lượng chi phí còn lại từ n đến đích).
- Với bài toán mê cung, mỗi bước di chuyển từ một ô sang một ô kề cạnh (lên, xuống, trái, phải) đều có chi phí bằng 1.
- Khi di chuyển từ vị trí *current* sang vị trí *neighbor*, chi phí được tính theo công thức:
$$g(neighbor) = g(current) + cost\_to\_move$$
- Trong trường hợp này:
$$cost\_to\_move = 1$$
nên:
$$g(neighbor) = g(current) + 1$$
- Giá trị khởi tạo:
$$g(start) = 0$$

2.4.3 Giải thích hàm heuristic

Hàm `heuristic` trong thuật toán A* này có vai trò ước lượng chi phí từ vị trí hiện tại đến đích, đồng thời kiểm tra tạo ra chiến thuật cho ghost. Cụ thể:

- ***Môi trường Pacman là lưới ô vuông với các hướng di chuyển hạn chế (lên, xuống, trái, phải). Khoảng cách Manhattan hoạt động hiệu quả nhất trong không gian này vì nó chính xác phản ánh số bước di chuyển tối thiểu cần thiết.***

- **Ước lượng khoảng cách:** Dùng công thức khoảng cách Manhattan:

$$base\_distance = |x_1 - x_2| + |y_1 - y_2|$$

để ước lượng số bước tối thiểu cần di chuyển đến đích.

- ***Bức tường là đặc điểm quan trọng của Pacman, và heuristic tối ưu phải xử lý được các đường đi bị chặn. Thêm wall_ penalty để phản ánh tình huống có tường giữa ghost và Pacman.***

- **Xử lý vùng nguy hiểm:** Nếu vị trí hiện tại nằm trong `danger_zones`, một khoản phạt (`penalty`) sẽ được cộng thêm vào heuristic. Ví dụ:

$$penalty = 10$$

- ***Trong Pacman truyền thống có nhiều ghost. Heuristic tốt nhất cân nhắc vị trí của các ghost khác để tạo ra chiến lược bao vây hiệu quả.***

- **Tính điểm chiến lược:**
  - Tập trung vào việc tính toán **điểm chiến lược** (`strategic_score`), nhằm khuyến khích các ghost phối hợp với nhau để bao vây Pac-Man. Điều này làm cho hành vi của ghost trở nên thông minh và khó đoán hơn.
  - Các bước tính toán:
    1. **Khởi tạo điểm chiến lược:** `strategic_score` được khởi tạo bằng `0`. Điểm này sẽ được điều chỉnh dựa trên vị trí của các ghost khác so với mục tiêu.
    2. **Kiểm tra trạng thái game và chỉ số ghost:** Đầu tiên, đoạn mã kiểm tra xem trạng thái game (`game_state`) có hợp lệ hay không và chỉ số của ghost hiện tại (`ghost_index`) có được cung cấp hay không. Nếu cả hai điều kiện này đều đúng, hàm sẽ tiếp tục tính toán.
    3. **Lấy vị trí của các ghost khác:** Hàm `game_state.get_ghost_positions()` được gọi để lấy danh sách vị trí của tất cả các ghost trong trò chơi. Điều này cho phép ghost hiện tại biết được vị trí của các ghost khác để phối hợp.
    4. **Phối hợp giữa các ghost:** Nếu có nhiều hơn một ghost trong trò chơi, đoạn mã duyệt qua danh sách vị trí của các ghost khác (ngoại trừ ghost hiện tại, được xác định bởi `ghost_index`). Với mỗi ghost khác, nếu khoảng cách Manhattan giữa vị trí của ghost đó và mục tiêu nhỏ hơn 5, điểm chiến lược (`strategic_score`) sẽ giảm đi 2. Điều này phản ánh rằng ghost hiện tại đang ở trong tình huống thuận lợi để phối hợp với ghost khác nhằm bao vây Pac-Man.

- **Tổng hợp heuristic:** Heuristic trả về theo công thức:

$$h(n) = weight \times base\_distance + penalty + strategic\_score$$

Ý nghĩa:

- Điểm chiến lược giảm khi ghost khác ở gần mục tiêu, điều này làm giảm giá trị heuristic. Kết quả là thuật toán A* sẽ ưu tiên các đường đi giúp ghost phối hợp hiệu quả hơn với các ghost khác để tạo áp lực lên Pac-Man.
- Tác động đến hành vi của ghost Điểm chiến lược giúp ghost không chỉ tập trung vào việc đuổi theo Pac-Man một cách độc lập mà còn cân nhắc đến vị trí của các ghost khác. Điều này làm cho hành vi của ghost trở nên chiến lược hơn, tăng độ khó cho người chơi.
- Việc kết hợp giữa khoảng cách ước lượng, kiểm tra vùng nguy hiểm và điểm chiến lược giúp thuật toán A* tìm được đường đi hợp lệ, tối ưu và an toàn trong không gian trạng thái phức tạp.

2.4.4 Độ Phức Tạp Thời Gian và Không Gian

- Độ Phức Tạp Thời Gian: $O(E \log V)$ với cài đặt đống nhị phân
- Độ Phức Tạp Không Gian: $O(V)$ để lưu trữ tập mở, bản đồ đến từ, và điểm số
- Hình ảnh trong game, thuật toán A* tìm kiếm Pacman:

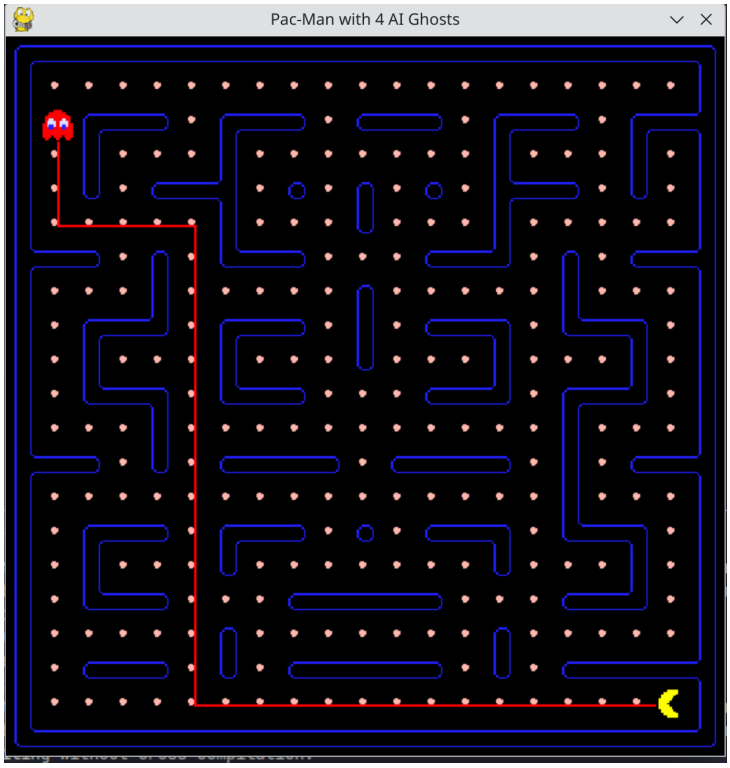

Figure 4: Thuật toán A* tìm kiếm đường đi cho ghost tới Pacman

2.4.5 So Sánh với UCS

A* là một phiên bản cải tiến của UCS, sử dụng thêm hàm heuristic để ước tính khoảng cách đến đích. Trong khi UCS chỉ xem xét chi phí đã biết từ nút bắt đầu đến nút hiện tại (g(n)), A* còn kết hợp ước tính chi phí từ nút hiện tại đến đích (h(n)). Nếu hàm heuristic h(n) luôn bằng 0, A* sẽ hoạt động giống hệt UCS.

3 Thí Nghiệm

3.1 Thiết Lập Thí Nghiệm

3.1.1 Môi Trường Kiểm Tra

- Phần cứng: Máy tính Lennovo Thinkbook G6+, AMD RYZEN7, 32GB GRAM, 1TB Ổ cứng SSD.
- Phần mềm: Ngôn ngữ lập trình Python
- Thư viện hỗ trợ thí nghiệm:
  - Đo thời gian: time
  - Đo bộ nhớ: tracemalloc
  - Đếm nodes mở: tạo biến đếm

3.1.2 Các Chỉ Số Hiệu Suất

- Thời Gian Tìm Kiếm: Đo bằng mili giây
- Sử Dụng Bộ Nhớ: Đo bằng Kilobyte
- Nút Mở Rộng: Số lượng nút được thăm trong quá trình tìm kiếm

3.1.3 Mô tả các Kịch bản thí nghiệm

1. **Ghost**: Góc trên bên trái, **Pacman**: Trung tâm mê cung
2. **Ghost**: Góc trên bên phải, **Pacman**: Trung tâm-dưới mê cung
3. **Ghost**: Góc dưới bên trái, **Pacman**: Bên phải-giữa mê cung
4. **Ghost**: Góc dưới bên phải, **Pacman**: Bên trái-trên mê cung
5. **Ghost**: Trung tâm mê cung, **Pacman**: Bên phải-dưới mê cung

3.2 Kết Quả và Phân Tích

3.2.1 So sánh Thời gian

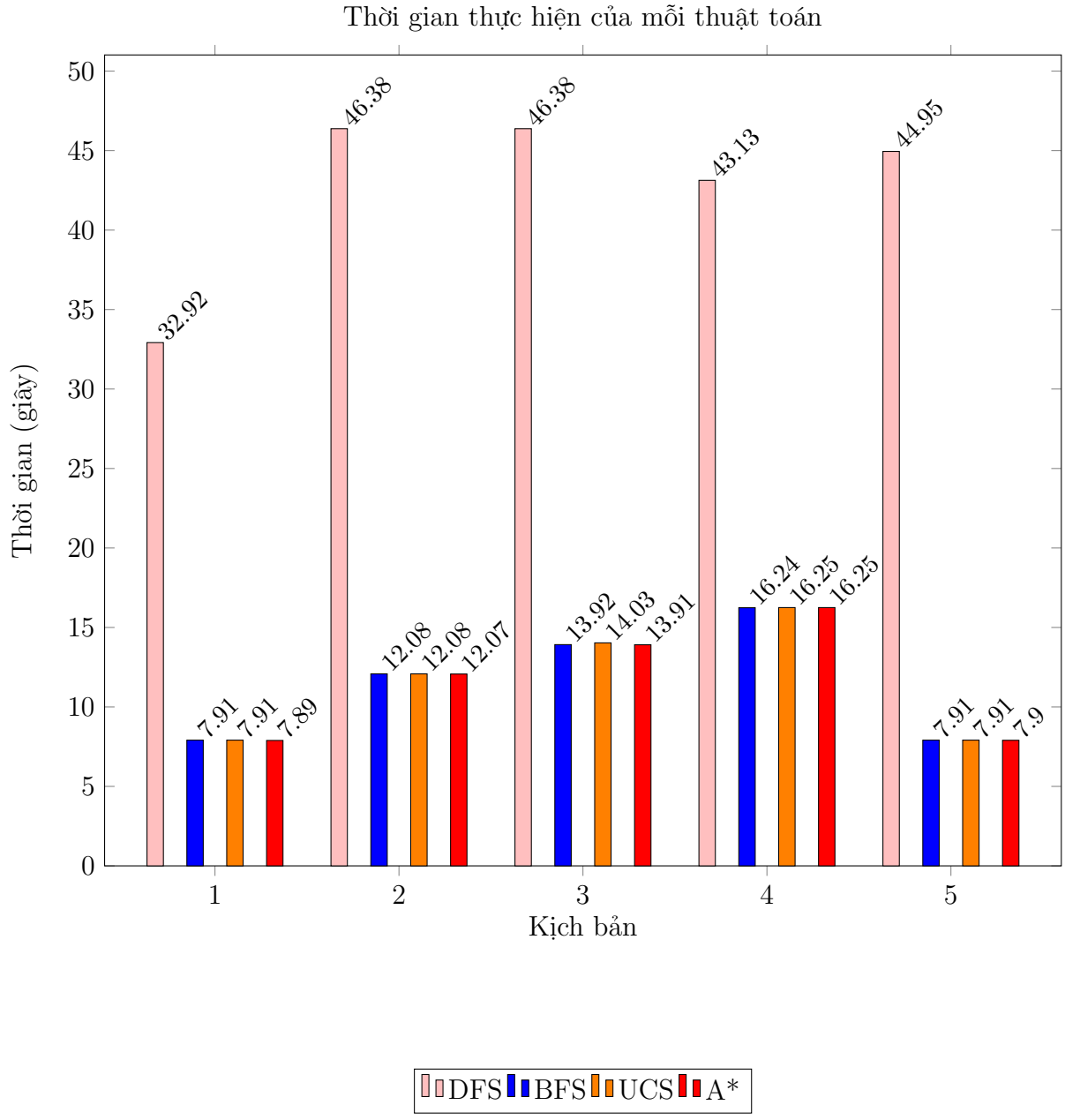

Figure 5: Thời gian thực hiện (tính bằng giây) của mỗi thuật toán qua năm kịch bản khác nhau.

3.2.2 So sánh Số lượng Nút Mở rộng

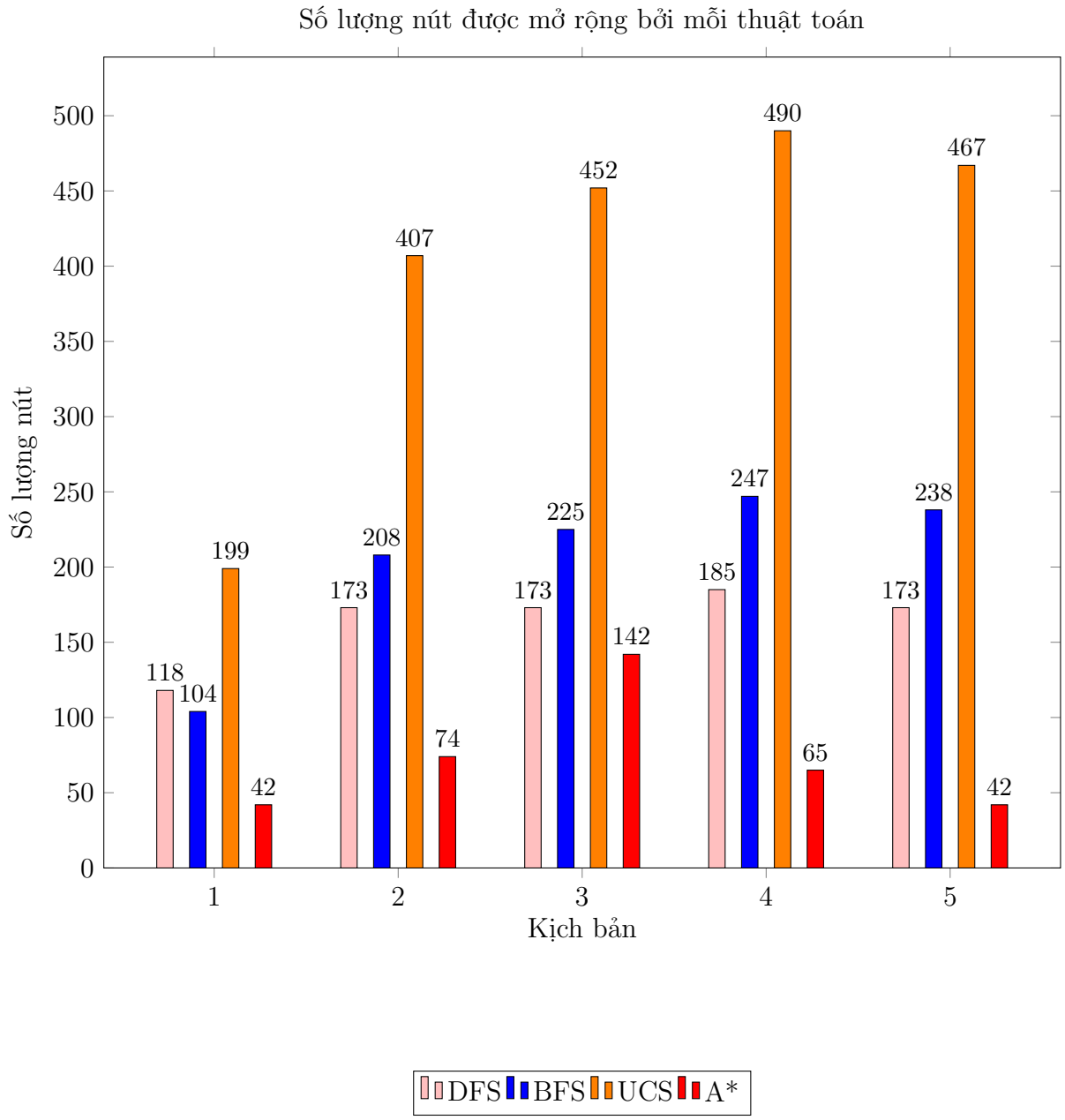

Figure 6: Số lượng nút được mở rộng bởi mỗi thuật toán qua năm kịch bản khác nhau.

3.2.3 So sánh Bộ nhớ Sử dụng

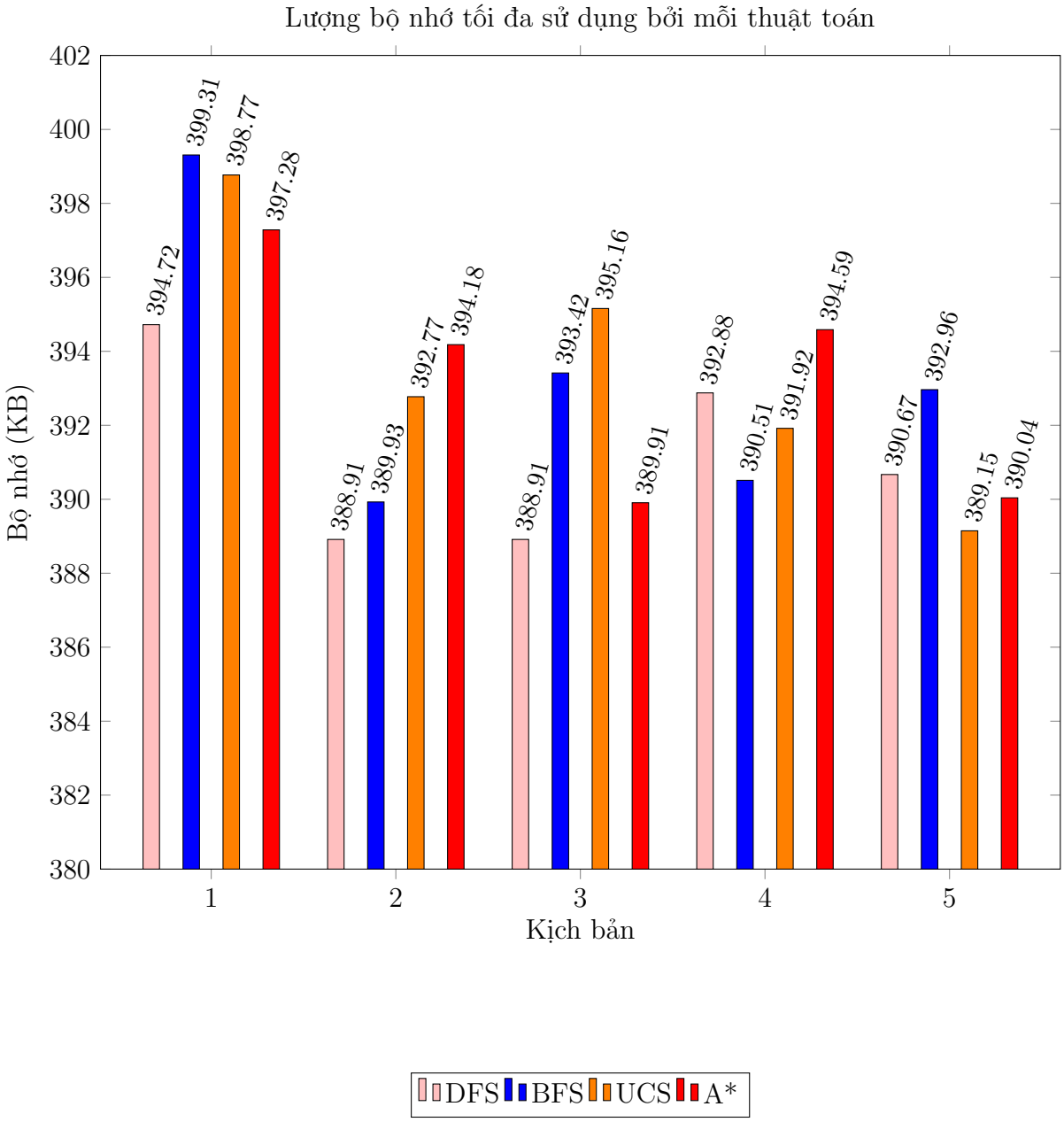

Figure 7: Lượng bộ nhớ tối đa (tính bằng KB) sử dụng bởi mỗi thuật toán qua 5 kịch bản khác nhau.

3.3 Tóm tắt So sánh Hiệu suất

- **DFS (Ma Hồng)**: Có thời gian thực hiện dài hơn đáng kể so với các thuật toán khác nhưng số lượng nút mở rộng ở mức trung bình. Điều này cho thấy thuật toán kém hiệu quả hơn cho việc tìm đường trong ứng dụng này.
- **BFS (Ma Xanh)**: Thể hiện sự cân bằng tốt giữa thời gian và số nút mở rộng, với hiệu suất ổn định qua các kịch bản khác nhau.

- **UCS (Ma Cam)**: Thời gian thực hiện tương tự như BFS nhưng mở rộng số lượng nút nhiều hơn đáng kể, cho thấy chi phí tính toán cao hơn ngay cả khi tìm được đường đi tối ưu.
- **A* (Ma Đỏ)**: Thể hiện hiệu suất tổng thể tốt nhất với thời gian thực hiện tương tự BFS/UCS nhưng số lượng nút mở rộng ít hơn đáng kể, nổi bật hiệu quả của thuật toán trong việc tìm đường đi tối ưu với sự hướng dẫn của heuristic.

3.4 Tổng kết

3.4.1 Phân Tích Hiệu Suất

Kết quả thí nghiệm của chúng em cho thấy A* liên tục vượt trội hơn các thuật toán khác về thời gian tìm kiếm và số lượng nút mở rộng, đặc biệt khi kích thước đồ thị tăng lên. UCS có hiệu suất tốt hơn BFS và DFS trong đồ thị có trọng số, nhưng không tối ưu bằng A* do thiếu thông tin heuristic để định hướng tìm kiếm về phía nút đích.

3.4.2 Hiệu Quả Bộ Nhớ

DFS cho thấy hiệu quả bộ nhớ tốt nhất trong tất cả các thuật toán, đặc biệt là đối với các đồ thị lớn hơn. BFS và UCS có mức sử dụng bộ nhớ tương tự nhau, trong khi A* tiêu thụ nhiều bộ nhớ nhất do phải lưu trữ thêm thông tin heuristic và cấu trúc dữ liệu phức tạp hơn.

3.4.3 Tính Đầy Đủ và Tối Ưu

- BFS đảm bảo giải pháp tối ưu khi tất cả các cạnh có trọng số bằng nhau
- DFS không đảm bảo giải pháp tối ưu nhưng yêu cầu ít bộ nhớ hơn
- UCS luôn tìm được đường đi có chi phí thấp nhất trong đồ thị có trọng số không âm
- A* đảm bảo giải pháp tối ưu khi sử dụng heuristic phù hợp (không vượt quá chi phí thực) và mở rộng ít nút hơn UCS

3.4.4 Lựa Chọn Thuật Toán Phù Hợp

Dựa trên các thí nghiệm của nhóm đã thực hiện, nhóm chúng em đề xuất:

- BFS cho đồ thị không có trọng số hoặc khi cần đường đi ngắn nhất về số cạnh
- DFS cho các bài toán có hạn chế về bộ nhớ hoặc khi tìm kiếm đường đi trong cấu trúc cây sâu
- UCS cho đồ thị có trọng số không âm khi không có heuristic tốt
- A* cho các bài toán mà việc tìm đường đi tối ưu là quan trọng và có sẵn một heuristic tốt

4 Kết Luận

Báo cáo này đã trình bày phân tích toàn diện về các thuật toán tìm kiếm, bao gồm chi tiết triển khai và đặc điểm hiệu suất của chúng. Kết quả thí nghiệm của chúng tôi cung cấp những hiểu biết giá trị về điểm mạnh và điểm yếu của mỗi thuật toán, có thể hướng dẫn việc lựa chọn thuật toán cho các lĩnh vực bài toán cụ thể.

Việc thêm UCS vào phân tích cho phép chúng em hiểu rõ hơn về mối quan hệ giữa các thuật toán tìm kiếm. UCS đóng vai trò như cầu nối giữa BFS (tìm kiếm không có thông tin trong đồ thị không trọng số) và A* (tìm kiếm có thông tin trong đồ thị có trọng số). Kết quả cho thấy việc lựa chọn thuật toán phù hợp phụ thuộc vào đặc điểm cụ thể của bài toán, bao gồm cấu trúc đồ thị, sự có sẵn của hàm heuristic, và các ràng buộc về tài nguyên.

Nhóm đã hoàn thành thành công tất cả các nhiệm vụ được giao với tỷ lệ hoàn thành cao, thể hiện sự hợp tác hiệu quả và phân phối nhiệm vụ. Kết quả thí nghiệm phù hợp với kỳ vọng lý thuyết và cung cấp nền tảng vững chắc cho công việc trong tương lai về tối ưu hóa thuật toán tìm kiếm.

5 Tài Liệu Tham Khảo

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

2. Russell, S. J., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.

3. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100-107.

4. Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. Numerische Mathematik, 1, 269-271.

5. PyGame: https://www.pygame.org/wiki/tutorials