ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO ĐỒ ÁN 01

Search: Pacman Game

Môn học: Cơ sở Trí tuệ Nhân tạo

Giảng viên hướng dẫn

Bùi Duy Đăng
Trần Quốc Huy
Nguyễn Trần Duy Minh

Nhóm 15

MSSV	Thành viên
21120291	Nguyễn Đức Nam (NT)
21120174	Nguyễn Thị Gái
21120178	Văn Bá Bảo Huy
21120180	Nguyễn Bích Khuê

Thành phố Hồ Chí Minh, 2023

Mục lục

Trang

Danh sách bảng

Danh sách hình vẽ

Chapter I

Thông tin đồ án

Cài đặt mô phỏng game Pacman, trong đó bạn sẽ sử dụng thuật toán học cho Pacman để tìm kiếm thức ăn mà không bị giết bởi quái vật.
Cả Pacman và quái vật đều liên tục đi với 4 hướng: trái, trên, phải dưới, không được đi xuyên tường, có 4 cấp độ như sau:

- Cấp 1: Bản đồ chỉ có Pacman, 1 thức ăn, và tường. Pacman nhận biết được vị trí của 1 thức ăn đó.
- Cấp 2: Bản đồ có Pacman, 1 thức ăn, (có thể nhiều) quái vật và tường. Ở đây, quái vật chỉ đứng im, và Pacman nhận biết được vị trí 1 thức ăn đó.
- Cấp 3: Bản đồ gồm Pacman, (có thể nhiều) thức ăn, quái vật, và tường. Pacman bị giới hạn tầm nhìn trong khoảng 3 bước đi (8 ô x 3 dãy tức 24 ô xung quanh), còn quái vật chỉ có thể đi những ô xung quanh nó (rồi trở lại). Cả pacman và quái vật đều di chuyển 1 lần mỗi lượt.
- Cấp 4: Bản đồ gồm Pacman, (có thể nhiều) thức ăn, quái vật, và tường. Quái vật sẽ liên tục truy đuổi Pacman, trong khi đó Pacman cố gắng ăn được càng nhiều thức ăn càng tốt mà không để chết, các con quái có thể di chuyển đè lên nhau. Cả hai bên đều di chuyển 1 lần mỗi lượt.

Thuật toán được so sánh trên nhiều map, và đánh giá dựa vào các tiêu chí sau:

- Thời gian hoàn thành.
- Chiều dài đường đi.
- Điểm số (đi mỗi bước Pacman bị trừ 1 điểm, khi ăn được thức ăn được cộng 20 điểm).

1. Mục tiêu đồ án

Mô phỏng được game Pacman, sử dụng được các thuật toán học (ở đây là tìm kiếm) để giúp Pacman tìm thức ăn, đồng thời tránh bị giết bởi quái vật qua nhiều cấp độ. Đồng thời cũng là để:

- Phát triển kĩ năng làm việc nhóm.
- Áp dụng được kiến thức lý thuyết vào một chương trình cụ thể.
- Giảng viên đánh giá được mức độ kiến thức của sinh viên qua cài đặt, báo cáo v,v,..

2. Bảng phân công công việc

STT	Công việc	Thành viên tham gia
1	Đồ họa	Văn Bá Bảo Huy Nguyễn Thị Gái Nguyễn Bích Khuê
2	Level 1,2	Cả 4 thành viên
3	Level 3	Văn Bá Bảo Huy Nguyễn Đức Nam
4	Level 4	Nguyễn Thị Gái Nguyễn Đức Nam
5	Viết báo cáo	Cả 4 thành viên

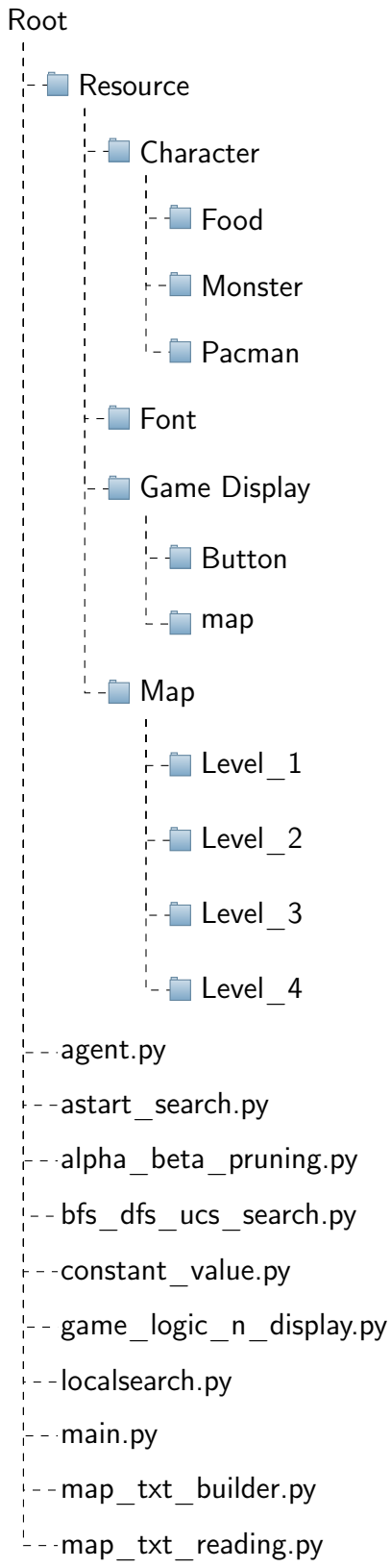
Bảng I.1: Bảng phân công công việc

3. Đánh giá mức độ hoàn thành đồ án

STT	Mô tả	Ghi chú	Hoàn thành
1	**Level 1**: Pac-Man biết được vị trí của thức ăn. Không có Monster và chỉ có 1 thức ăn.	Dùng 1 trong 4 thuật toán tìm kiếm A*, DFS, BFS, UCS để tìm đường đi tới thức ăn.	100%
2	**Level 2**: Tương tự level 1, level 2 có thêm những con quái vật, nhưng quái vật chỉ đứng im một chỗ. Nếu Pacman đi vào ô có quái vật thì trò chơi kết thúc.	Tương tự level 1, nhưng ta sẽ xem như quái vật là tường. Nếu quái vật chặn mọi đường đi thì Pacman sẽ thua.	100%
3	**Level 3**: Tầm nhìn của Pac-Man bị giới hạn ở ba bước gần nhất, Pac-Man chỉ có thể quét các ô liền kề trong phạm vi 8 ô và 3 bước. Có nhiều thức ăn trải rộng khắp bản đồ. Quái vật có thể di chuyển một bước theo bất kỳ hướng hợp lệ nào xung quanh vị trí ban đầu của chúng khi bắt đầu trò chơi.	Dùng Heuristic Local Search để tìm ra bước đi tốt nhất tại mỗi thời điểm.	100%
4	**Level 4**: bao gồm một bản đồ nơi quái vật không ngừng truy đuổi Pac-Man. Pac-Man phải thu thập càng nhiều thức ăn càng tốt đồng thời tránh bị bất kỳ Monster nào bắt được. Những Monster có khả năng đi xuyên qua nhau. Cả Pac-Man và quái vật đều di chuyển một bước mỗi lượt và bản đồ chứa vô số thức ăn.	Dùng thuật toán Alpha-Beta (cải tiến Minimax) để xác định bước đi tốt nhất của Pacman mỗi lượt, quái vật sử dụng A* (kết hợp với random) để truy đuổi Pacman.	100%
5	Trình bày đồ họa		100%
6	Thiết kế 5 bản đồ theo độ khó tăng dần		100%
7	Viết báo cáo		100%

Bảng I.2: Bảng đánh giá mức độ hoàn thành đồ án

4. Cấu trúc thư mục



Chapter II

Level 1, 2

1. Ý tưởng thuật toán

1.1 Level 1

Nhắc lại bài toán: Pacman biết được vị trí của thức ăn (goal). Không có quái vật (monster) và chỉ có 1 thức ăn trên toàn bộ bản đồ. Mục tiêu của Pacman là tìm đường đến thức ăn và giành chiến thắng. Ở level này, Pacman nhìn thấy toàn bộ bản đồ bao gồm thức ăn và mọi đường đi.

Ý tưởng thuật toán: Theo yêu cầu bài toán, ta cần phải tìm đường đi để đưa Pacman đến vị trí thức ăn và giành chiến thắng. Vì vậy, với dữ liệu bản đồ trò chơi đã đọc được, ta sử dụng các thuật toán sau đây để thực hiện tìm kiếm: A* (A-star), UCS (Uniform Cost Search), BFS (Breadth-First Search), và DFS (Depth-First Search).

1.2 Level 2

Nhắc lại bài toán: Giống level 1 nhưng có thêm (1 hoặc nhiều) quái vật. Các quái vật không thể di chuyển được.

Ý tưởng thuật toán: Coi các quái vật như là tường và thực hiện tìm kiếm như ở level 1. Nếu các quái vật chặn hết mọi đường, Pacman sẽ vẫn tìm đường đến nhưng khi đi vào ô có quái vật, trò chơi kết thúc.

1.3 Thuật toán DFS

- DFS (Depth-First Search), gọi là tìm kiếm theo chiều sâu. Với mục đích là tìm được đường đi tới điểm kết thúc (ở đây là thức ăn), từ ô bắt đầu (ở đây là pacman_pos).
- Thuật toán cố gắng tìm theo một đường càng sâu càng tốt, trước khi quay lại (không sâu thêm được nữa)

```
function dfs(node):
Mark node as visited
for each neighbor in neighbors of node:
```

```
if neighbor is not visited:
    dfs(neighbor)
```

- Từ điểm bắt đầu (pacman_pos),ta sẽ cho là điểm này đã đi,rồi chọn một node mà nó có thể đi được (hàng xóm), rồi lại tiến hành DFS cho điểm mới này. Cho đến khi không còn node (hàng xóm) nào để thăm (tức DFS hết mà không có đường đi đến điểm đích), thì ta quay lại điểm trước đó (tất nhiên là cho điểm này thành chưa thăm lại), và tìm kiếm một con đường khác bắt đầu từ ô này.

- Lặp lại quá trình cho đến khi không còn node nào chưa thăm, nếu tìm được điểm đích thì dừng, còn vẫn không tìm được thì nghĩa là không có đường đi từ điểm bắt đầu, đến đích (pacman_pos đến đích food_pos).

1.4 Thuật toán BFS

- BFS (Breadth-first Search) là thuật toán tìm kiếm theo chiều rộng.

- Hàm bfs sử dụng hàng đợi để quản lý các nút. Ban đầu, hàng đợi chỉ chứa vị trí bắt đầu (pacman). Thuật toán sau đó lặp đi lặp lại loại bỏ một nút khỏi hàng đợi, đánh dấu nó là đã được truy cập và thêm các nút kề chưa được truy cập của nó vào hàng đợi. Quá trình này tiếp tục cho đến khi nút đích(food) được tìm thấy hoặc hàng đợi trống.

1.5 Thuật toán UCS

- Vị trí bắt đầu là Pacman và vị trí đích là Food.

- Sử dụng hàng đợi ưu tiên để lưu trữ các vị trí Pacman có thể đi qua để đến Food.

- Chi phí của điểm bắt đầu (ví trí của Pacman) là 0, ta sẽ duyệt qua các điểm lân cận của vị trí bắt đầu. Những điểm này đều được gắn với chi phí đường đi là 1 và lưu vào hàng đợi ưu tiên.

- Từ hàng đợi, ta xét điểm đầu tiên có phải Food hay không, nếu là Food sẽ kết thúc thuật toán.

- Ngược lại nếu không phải Food, ta sẽ xem như Pacman di chuyển đến điểm này và cập nhật điểm đó là điểm xét duyệt hiện tại. Bắt đầu duyệt các điểm lân cận của điểm hiện tại để cập nhật vào hàng đợi. Chi phí đường đi của các điểm này sẽ bằng chi phí đường đi của điểm xét duyệt hiện tại cộng với 1.

- Tương tự như vậy ta sẽ duyệt đến khi nào đến vị trí đích tức là đã tìm được Food thì sẽ kết thúc thuật toán.

1.6 Thuật toán A*

- A* là một thuật toán tìm kiếm trong đồ thị được cải thiện từ thuật toán greedy best-first search sẽ tìm đường từ 1 nút ban đầu đến 1 nút đích cho trước sao cho chi phí là tốt nhất và số bước duyệt là ít nhất. Trong A*, chi phí đường đi được tính là tổng của chi phí

đường đi từ nút bắt đầu đến trạng thái hiện tại với tổng chi phí ước tính (bằng heuristic) đến nút đích.

- Thuật toán sẽ tìm kiếm những đường đi có chi phí tối ưu nhất.

```
function Heuristic(state, goal):
    return max(distance in x axis, distance in y axis)

function Astar(graph, start, goal):
    Visited  = set list
    Frontier = priority queue
    start    = Node contains: start state, path cost, parent state.

While frontier is not empty:
    get Node from frontier list
    return pathway if Node is goal
    for each neighbor of Node:
        create Node with pathcost = pathcost from start to state + heuristic
        if Node is not visited:
            add to Frontier
            add to Visited
```

- Đầu tiên, ta tạo ra các dãy, gồm: Visited (kiểu dữ liệu set) dùng để lưu trữ các nút đã duyệt qua và Frontier là hàng đợi ưu tiên để lấy các nút có chi phí thấp nhất.
- Ta duyệt qua nút đầu tiên và đưa nó vào danh sách Visited và hàng đợi Frontier.
- Với mỗi nút lân cận của nút hiện tại, ta tính chi phí đường đi bằng công thức $f(n) = g(n) + h(n)$, nghĩa là tổng chi phí đường đi từ nút đầu tiên đến nút hiện tại với chi phí heuristic ước tính từ nút hiện tại đến nút đích.
- Lặp lại quá trình trên đến khi nút hiện tại là nút đích và trả về đường đi là một danh sách các nút từ nút bắt đầu đến nút đích.

Cả bốn thuật toán đều sẽ tìm kiếm đường đi từ nút đầu tiên đến nút đích và trả về là một danh sách các nút từ nút đầu tiên dẫn đến nút đích.

2. Mô tả thuật toán

2.1 Xử lí dữ liệu đầu vào và hiển thị

Đầu tiên, từ dữ liệu đầu vào, ta sử dụng hàm đọc dữ liệu và xử lí để trả về các dữ liệu sau:

- graph_map: là dữ liệu dạng dictionary lưu trữ lại các nút lân cận của một nút. Có được bằng cách duyệt qua ma trận đầu vào và tìm kiếm những nút lân cận có thể đi đến được từ mỗi nút.
- pacman_pos: là dữ liệu dạng tuple (x, y) là vị trí ban đầu của Pacman trong hệ tọa độ Oxy.
- foods_pos: là dữ liệu dạng tuple (x, y) là vị trí của thức ăn trong hệ tọa độ Oxy.

```
graph_map, pacman_pos, food_pos =
readmap.map_level1(s_map_txt_path[self.level-1][self.map_index])
```

Khi có được vị trí của Pacman và thức ăn, ta hiển thị hai đối tượng đó lên trên bản đồ.

```
# Call Pacman
pacman = ag.Pacman(self, pacman_pos)
pacman.pacman_call()
# Call Food
food = ag.Food(self, food_pos)
food.food_display()
```

Sau đó, ta đưa các dữ liệu có được từ việc đọc dữ liệu đầu vào qua một trong bốn thuật toán: BFS, DFS, UCS hoặc A* để tìm kiếm đường đi cho Pacman từ vị trí bắt đầu đến vị trí có thức ăn. Khi đó, ta sẽ được một danh sách là danh sách các nút để đi từ nút ban đầu (vị trí Pacman) đến nút kết thúc (vị trí thức ăn) gọi là **pathway**. Để thực hiện 4 thuật toán này, ta sẽ xây dựng một lớp Node:

```
class Node:
    def __init__(self, state, path_cost, parent):
        self.state     = state
        self.path_cost = path_cost
        self.parent    = parent

    def __lt__(self, other):
        return self.path_cost < other.path_cost
```

Lớp Node (nút) bao gồm các dữ liệu:

- state: trạng thái của nút, mang kiểu dữ liệu tuple (x, y) là vị trí nút trong hệ tọa độ Oxy.
- path_cost: chi phí đường đi, bao gồm $f(n) = g(n) + h(n)$ bao gồm chi phi đường đi từ nút bắt đầu đến nút đó và chi phí heuristic ước tính từ nút đó đến nút đích.
- parent: là nút cha của nút hiện tại.

Chi tiết các thuật toán sẽ được nêu ở phần dưới.

Sau khi đã sử dụng các thuật toán trên và lấy được danh sách đường đi pathway, ta tiến hành kiểm tra, nếu danh sách rỗng, nghĩa là đường đi đến thức ăn bị chặn, trò chơi kết thúc và thông báo thất bại. Ngược lại, Pacman sẽ thực hiện di chuyển sử dụng các hàm đồ họa để di chuyển trên màn hình đến vị trí đích. Khi Pacman đến vị trí thức ăn, trò chơi kết thúc và thông báo chiến thắng.

2.2 DFS

Trong hàm DFS, ta dùng biến visited (lưu các đỉnh đã thăm), path(lưu lại đường đi - chính là giá trị trả về), rồi gọi hàm DFS_until tìm. Nếu có thì trả về được output pathway, nếu không thì trả về None, tức không có đường đi.

```
def DFS(graph,pacman,food):
    #Set of visited nodes and path for run
    visited = set()
    path = []
    DFS_until(graph,pacman,food,visited,path)
    if not path:
        return None
    return path
```

Trong hàm DFS_until (gọi điểm pacman đi là start, end chính là vị trí kết thúc, food_pos), lần lượt thêm vào visited,path điểm hiện tại.
Nếu điểm hiện tại chính là điểm end (food), thì trả về True (tức path là đường đi đúng), nếu chưa ta xét các điểm mà tại start (hiện tại), có thể đi được (có liên kết, và chưa được thăm), sau đó lại dùng DFS_until cho điểm này.
Chú ý vì list trong python truyền vào hàm, xong ta thay đổi các giá trị trong hàm, nó sẽ thay đổi cả bên ngoài (tức toàn cục). Nếu trong quá trình DFS_until tìm tới được điểm end thì trả về True, tức đã tìm được đường đi path.
Nếu không, ta sẽ loại điểm này ra khỏi path (vì đường đi tới điểm này, xét tiếp thì không đi được đến đích) (không loại khỏi visited, vì có thăm điểm này nữa thì cũng không tới đích được), để còn quay lại xét con đường khác.

```
    def DFS_until(graph,start,end,visited,path):
        visited.add(start)
        path.append(start)

        if start == end:
            return True

        for neighbor in graph[start]:
            if neighbor not in visited:
                if DFS_until(graph,neighbor,end,visited,path):
                    return True

        path.pop()
        return False
```

2.3 BFS

Hàm bfs sử dụng hàng đợi để quản lý các nút. Ban đầu, hàng đợi chỉ chứa vị trí bắt đầu (pacman). Thuật toán sau đó lặp đi lặp lại loại bỏ một nút khỏi hàng đợi, đánh dấu nó là đã được truy cập và thêm các nút kề chưa được truy cập của nó vào hàng đợi. Quá trình này tiếp tục cho đến khi nút đích(food) được tìm thấy hoặc hàng đợi trống.
Khởi tạo:

- visited: Một tập để theo dõi các trạng thái đã được truy cập.
- frontier: Một hàng đợi để lưu trữ các nút cần được khám phá.

Tạo nút bắt đầu là None và thêm vào frontier đồng thời đánh dấu là đã thăm.
Nếu hàng đợi frontier không trống thì liên tục lặp:

- Lấy 1 nút khỏi frontier
- Kiểm tra xem phải nút đích không. Nếu có thì trả về pathway
- Thực hiện vòng lặp for để duyệt các các nút kế bên, nếu nút chưa được thăm thì tiến hành thêm vào frontier và đánh dấu là đã được thăm.

Vòng lặp liên tục thực hiện cho đến khi tìm thấy nút đích hoặc là đi qua hết tất cả các nút.

```
def bfs(graph, pacman, food):
    visited = set()
    frontier = queue.Queue()
```

```
    start_node = Node(pacman, 0, None)

    frontier.put(start_node)
    visited.add(start_node.state)

    while not frontier.empty():
        node = frontier.get()

        if node.state == food:
            return pathway_bfs(node)

        for child_state in graph[node.state]:
            child_node = Node(child_state, node.path_cost + 1, node)

            if child_node.state not in visited:
                frontier.put(child_node)
                visited.add(child_node.state)

    return None
```

Hàm pathway_bfs xây dựng đường đi từ nút đích(food) đến nút bắt đầu(pacman) bằng cách theo dõi các liên kết cha của mỗi nút.

```
def pathway_bfs(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    path.reverse()
    return path
```

2.4 UCS

- Hàm **ucs**

```
    def ucs(graph, pacman, food):
        visited = set()
        frontier = queue.PriorityQueue()
        start_node = Node(pacman, 0, None)
```

```
    frontier.put(start_node)
    visited.add(start_node.state)

    while not frontier.empty():
        node = frontier.get()

        if node.state == food:
            return pathway_ucs(node)

        for child_state in graph[node.state]:
            child_node = Node(child_state, node.path_cost + 1, node)

            if child_node.state not in visited:
                frontier.put(child_node)
                visited.add(child_node.state)
    return None
```

- Hàm **def ucs(graph, pacman, food)** nhận vào ba đối số:
  - **graph**: Đồ thị biểu diễn bản đồ của trò chơi dưới dạng danh sách kề các trạng thái và các trạng thái lân cận.
  - **pacman**: Vị trí ban đầu của Pacman
  - **food**: Vị trí mục tiêu (Food) mà Pacman cần đến.
- **visited = set()**: Tạo một tập hợp rỗng lưu các vị trí đã được thăm.
- **frontier = queue.PriorityQueue()**: Tạo một hàng đợi ưu tiên sắp xếp các nút dựa trên chi phí đường đi, với nút có chi phí thấp hơn được ưu tiên trước.
- Sau đó ta sẽ tạo nút bắt đầu **start_node**, thêm nút bắt đầu vào hàng đợi ưu tiên và đánh giá trạng thái ban đầu là đã thăm bằng cách thêm vào tập hợp **visited**.
- Trong vòng lặp tiếp theo khi hàng đợi ưu tiên không trống, chúng ta sẽ lặp qua các nút trong hàng đợi ưu tiên và kiểm tra xem nút hiện tại có phải là nút mục tiêu (Food) không. Nếu là Food, hàm **pathway_ucs** được gọi để truy vết đường đi và trả về đường đi từ Pacman đến Food.

– Nếu không phải nút mục tiêu, tạo các nút con cho các vị trí lân cận chưa được thăm, cập nhật chi phí đường đi và thêm chúng vào hàng đợi ưu tiên và tập hợp **visited**.

- Hàm **pathway_ucs**

```
def pathway_ucs(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    path.reverse()
    return path
```

Hàm này nhận một nút và truy vết từ nút cuối lên đến nút ban đầu để tạo một danh sách đường đi. Đường đi này sau đó được đảo ngược để có thứ tự từ ban đầu đến cuối.

2.5 A* search

Thuật toán A* nhằm tìm kiếm đường đi ngắn nhất (có chi phí thấp nhất) từ vị trí ban đầu đến vị trí đích. Sử dụng lớp Node, danh sách Visited có cấu trúc Set và hàng đợi ưu tiên để thực hiện tìm kiếm:

- Visited: là một Set lưu trữ trạng thái (kiểu tuple, là vị trí trong hệ tọa độ Oxy) của nút đã duyệt qua.
- Frontier: hàng đợi ưu tiên, lưu trữ danh sách các đối tượng của lớp Node với giá trị ưu tiên là chi phi đường đi thấp nhất của các nút.

```
def astar_search(graph, pacman, food):
    def heuristic(state, goal):
        dx = abs(state[0] - goal[0])
        dy = abs(state[1] - goal[1])
        return dx + dy

    visited    = set()
    frontier   = queue.PriorityQueue()
    start_node = Node(pacman, heuristic(pacman, food), None)

    frontier.put(start_node)
    visited.add(start_node.state)

    while not frontier.empty():
        node = frontier.get()
```

```
        if node.state == food:
            return pathway(node)
        for child_state in graph[node.state]:
            child_node = Node(child_state, node.path_cost - heuristic(node.state, food) + 1
                                    + heuristic(child_state, food), node)

            if child_node.state not in visited:
                frontier.put(child_node)
                visited.add(child_node.state)
```

Đầu tiên, ta xây dựng hàm Heuristic để ước lượng chi phí đường đi từ một nút đến nút đích. Trong trường hợp này, sử dụng khoảng cách diagonal.

Tiếp theo, như đã đề cập, để thực hiện thuật toán tìm kiếm A*, ta sử dụng 2 danh sách là Visited và Frontier. Ta đặt vào 2 danh sách đó nút ban đầu (vị trí ban đầu của Pacman).

Thực hiện vòng lặp while:

- Lấy Node ra từ hàng đợi ưu tiên Frontier (là nút có chi phí đường đi thấp nhất)
- Nếu nút đó là nút đích, trả về đường đi là một danh sách được tạo bằng hàm pathway.
- Ngược lại, tiến hành duyệt đến các nút lân cận của nút hiện tại. Tạo các đối tượng của lớp Node với trạng thái của nút đang duyệt, chi phí đường đi là tổng chi phí từ nút bắt đầu đến nút đang duyệt và chi phí heuristic ước tính từ nút đang duyệt đến nút đích. Nếu nút đó chưa có trong danh sách đã duyệt thì thêm vào Frontier và Visited.

Hàm **pathway** tạo và trả về một danh sách các tuple là vị trí các nút từ nút bắt đầu đi đến nút đích.

```
def pathway(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    path.reverse()
    return pat
```

3. Đánh giá

	DFS		BFS		UCS		A*	
	Time	Score	Time	Score	Time	Score	Time	Score
Map 1	$5.73 * 10^{-5}$	-13	0.000652	-13	0.000633	-13	0.00047	-15
Map 2	$9.57 * 10^{-5}$	-110	0.000918	-16	0.001347	-16	0.0011111	-16
Map 3	0.000126	-112	0.000832	-20	0.000962	-20	0.001279	-20
Map 4	0.00037	-381	0.001838	-31	0.006221	-31	0.001854	-31
Map 5	0.000234	-236	0.00228	-30	0.001912	-30	0.002399	-30
Average	0.0001766	-170.4	0.001304	-22	0.002215	-22	0.00142262	-22.4

Bảng II.1: Bảng so sánh các thuật toán tìm kiếm

Nhận xét:

- DFS thường tìm đường đi nhanh nhất, nhưng có điểm số thấp hơn. Điều này có nghĩa là nó có thể tìm thấy đường đi đến mục tiêu một cách nhanh chóng, nhưng đường đi này có thể dài và không tốt.
- BFS thường tìm đường đi nhanh nhưng có chi phí cao hơn. Bởi vì nó tìm kiếm gần như toàn bộ không gian trạng thái trước khi tìm thấy đường đi ngắn nhất.
- UCS tìm kiếm đường đi với chi phí thấp nhất. Bởi vì đây là thuật toán tìm kiếm đường đi có chi phí thấp nhất, nhưng không hẳn là nhanh nhất.
- A* là một phương pháp tìm kiếm thông minh kết hợp giữa việc tối ưu hóa chi phí và tốc độ tìm kiếm. Thuật toán này thường tìm kiếm đường đi có chi phí thấp và tốc độ nhanh hơn so với UCS.

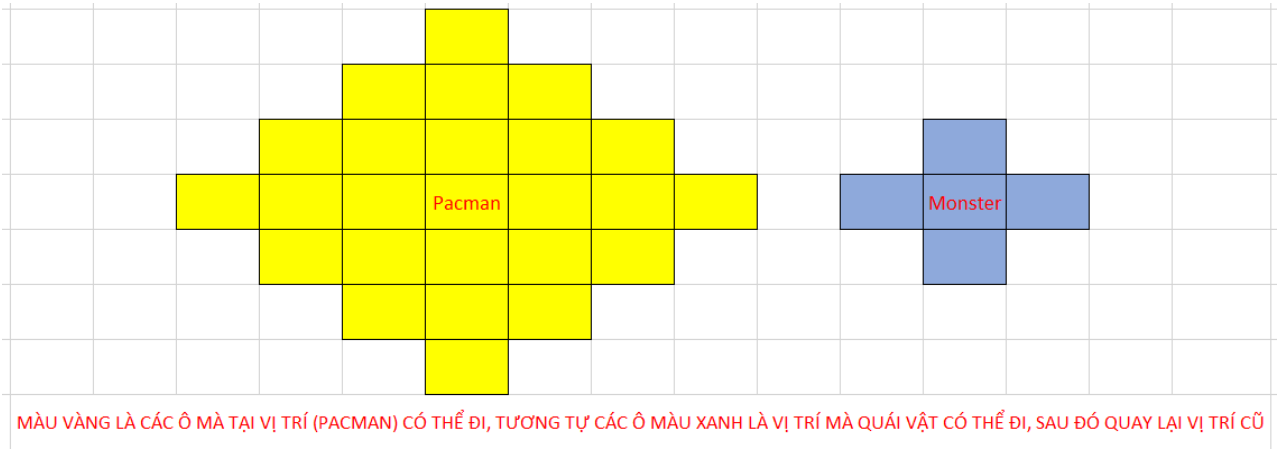
Dựa vào những đặc điểm trên chúng ta có thể lựa chọn thuật toán phù hợp tùy thuộc vào ưu tiên của chúng ta đối với tốc độ tìm kiếm và chi phí đường đi trong các tình huống cụ thể.

Chapter III

Level 3

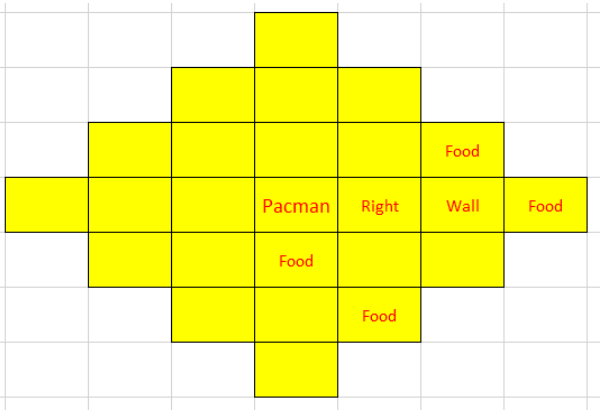
1. Ý tưởng thuật toán

Nhắc lại bài toán: Pacman bị giới hạn tầm nhìn, chỉ có thể nhận biết được các ô trong khoảng 3 bước đi (8 ô x 3 dãy), có nhiều thức ăn trên bản đồ, còn quái vật thì chỉ có thể di chuyển xung quanh 1 nước đi của mình và quay trở lại. Mỗi lượt pacman và quái vật di chuyển một bước đi.



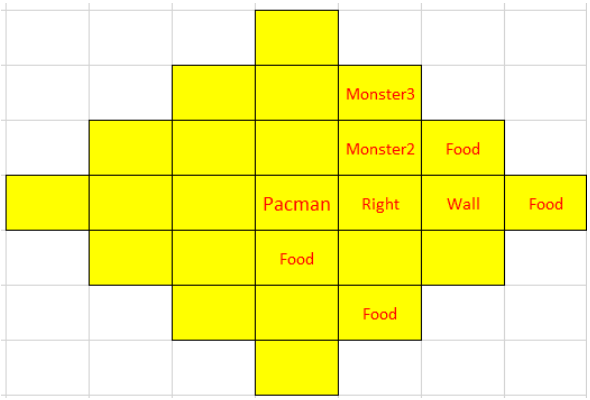
Hình III.1: Ảnh minh họa vị trí đi của Pacman và Monster

Ý tưởng: Đầu tiên ta xét pacman tại vị trí (x,y) bất kì, sẽ có 4 hướng trái, trên, phải, dưới cho pacman đi, nhiệm vụ của ta là quyết định xem pacman nên đi hướng nào trong 4 hướng trên, nên ta sẽ xét từng hướng (chỉ xét các hướng đi được), tính giá trị heuristic, sau đó so sánh và lấy giá trị heuristic cao nhất để đi, việc tính được làm như sau:
Đầu tiên, mỗi nước đi của Pacman sẽ bị trừ 1 điểm, để ưu tiên cho các ô chưa đi.
Thứ hai, mỗi nước đi của Pacman, nếu lặp lại nước đi trước đó (như đi từ (1,1) đến (1,2) rồi quay lại (1,1)), sẽ bị trừ 2 điểm, vì khi đi lại như vậy thì không tạo thêm điều gì mới cả, lặp lại nước đi ban đầu.
Công thức tính Heuristic cho từng hướng đi (trong trường hợp chưa có quái vật xung quanh) sẽ là, tổng số điểm thức ăn mà nó có thể đi đến được trong 2 nước đi (vì Pacman 3 bước, đi một bước tới ô này), không tính quay lại ô ban đầu. Ví dụ trong hình bên dưới, đi sang phải,tại vị trí Right, chỉ xét thức ăn trong phạm vi 2 nước đi của nó (không tính đi về phía vị trí Pacman cũ), Heuristic ở đây là 3*(điểm cho mỗi thức ăn), vì vị trí thức ăn sau tường thì cần nhiều nước đi hơn 2.



Hình III.2: Ảnh minh họa tính Heuristic khi không có quái vật.

Nếu trường hợp phát hiện có Monster xung quanh, thì sẽ bị trừ nhiều điểm hơn để tránh đi vào những ô này, ưu tiên đi các ô có điểm Heuristic cao hơn. Ví dụ hình minh họa dưới, khi đi sang phải, vị trí Right, nếu Right là vị trí của quái vật (1 bước đi từ Pacman), thì bị trừ ABC điểm, còn nếu quái vật là vị trí Monster2 (2 bước đi) thì bị trừ XYZ điểm, còn lại, quái vật ở vị trí Monster3, bị trừ TZ điểm, nếu có nhiều quái thì bị trừ cộng dồn lại như thường (tức không nên đi vào vùng nhiều quái dễ chết).



Hình III.3: Ảnh minh họa tính Heuristic khi có quái vật

Có sự chênh lệch giữa các điểm vì, quái vật cách 1 bước đi thì sẽ nguy hiểm hơn cách 2 bước, tương tự nguy hiểm hơn với 3 bước đi.
Những điểm này tìm được thông qua nhiều lần chạy. Vì khi đã tránh vào vùng có Monster, rồi đi ăn các ô còn lại, bị trừ điểm đủ nhiều (tức đã khám phá hoàn toàn các vùng không có quái vật, ăn hết rồi), thì số điểm các ô xung quanh đã bị trừ nhiều mức ABC,XYZ,TZ đã xét, bắt đầu Pacman phải mạo hiểm để vào vùng nguy hiểm.
Ví dụ Pacman đã ăn hết thức ăn bên ngoài, và đi nhiều lần các ô bên ngoài (vì các ô gần quái bị trừ điểm cao nên không đi), tới một lúc, giả sử điểm các ô bên ngoài đã đều lên ABC+ điểm, thì các ô gần Monsters (bị trừ ABC), đã cao hơn các ô xung quanh, nên Pacman sẽ bắt đầu chạy vào các ô nguy hiểm này.
Chương trình sẽ kết thúc khi Pacman ăn hết thức ăn, hoặc chết khi bị bắt bởi quái vật (cùng một ô).

2. Mô tả thuật toán

2.1 Đọc dữ liệu và hiển thị

Đầu tiên, đọc dữ liệu, ta đọc được dữ liêu về các biến sau:

- graph_map: Lưu danh sách các ô liền kề với ô hiện dưới dạng dictionary. Keys là vị trí ô, values là các ô liền kề (có thể đi được từ ô này).
- pacman_pos: Vị trí pacman ban đầu dưới dạng tuple.
- foods_pos: Danh sách vị trí các thức ăn trên bản đồ, mỗi thành phần trong danh sách này là 1 tuple.
- monster_list: Lưu danh sách các Monsters và nước đi liền kề hợp lệ của nó. (Keys là vị trí Monster ban đầu, values là ô liền kề với keys hợp lệ đi được).

```
graph_map, pacman_pos, foods_pos, monster_list =
readmap.map_level3(s_map_txt_path[self.level - 1][self.map_index])
```

2.2 Giải thích thuật toán

Tiếp theo, đơn giản là ta cho in hết toàn bộ Map, Pacman, Food, Monster lên.

```
# Call Pacman
pacman_pos_draw = [pacman_pos[1], pacman_pos[0]]
pacman = ag.Pacman(self, pacman_pos_draw)
pacman.pacman_call()
# Call Food
for foods_pos_t in foods_pos:
    foods_pos_draw_t = [foods_pos_t[1], foods_pos_t[0]]
    food = ag.Food(self, foods_pos_draw_t)
    food.food_display()
# Call Monster
monsters = [ag.Monster(self, [pos[1], pos[0]]) for pos in list(monster_list.keys())]
for mons in monsters:
    mons.monster_call()
```

Ta cũng có thêm các biến để lưu các giá trị ở trạng thái hiện tại, như:

- pre_pos (bước trước đó, để -2 khi lặp lại)
- time_visits (để lưu số lần đi đến ô này, vì mỗi lần đi đều trừ điểm) (chú ý: time_visits khống có nghĩa là Pacman biết toàn bản đồ để lưu lại số lần đi tại mỗi điểm, mà là nó

nhớ lại những nơi mà nó đã đi qua)

- parity_index_monster (dùng để giữ quái trong phạm vi 5 ô, nếu có giá trị lẻ thì Monsters sẽ đi 1 trong 4 hướng, chẵn thì sẽ quay trở lại ban đầu)
- old_pos (vị trí hiện tại của Monster)
- new_pos (Vị trí mới của Monster, ban đầu là old new trùng nhau, khi di chuyển new sẽ đổi trước, dùng lại biến old để xử lý ô cũ, sau đó gán old_pos = new_pos)
- keys_monster (lưu vị trí ban đầu của các Monsters cho dễ xử lý)
- heuristic_call (Trả về giá trị Heuristic của các ô xung quanh, tức các ô 1 nước đi của Pacman, sẽ được sắp theo thứ tự giảm dần, lấy ô đầu tiên tức ô có Heuristic cao nhất)
- Hai biến catch (bắt trạng thái quát vật gặp Pacman), go_home (bắt trạng thái quay về màn hình chính)

```
# Preprocessing
pre_pos = None
time_visits = {}
for key in list(graph_map.keys()):
    time_visits[key] = 0
parity_index_monster = 0  # to keep monster in this 5 cells
# Position of monsters
old_pos = []
for name in list(monster_list.keys()):
    old_pos.append(name)
new_pos = old_pos.copy()
keys_monster = list(monster_list.keys())
# Heuristics
heuristic_call = {}
for pos in graph_map[pacman_pos]:
    heuristic_call[pos] = ls.heuristic_lv3_2around(pacman_pos, graph_map, pos, foods_pos, old_pos)
#Sort
heuristic_call = dict(sorted(heuristic_call.items(), key=lambda item: item[1], reverse=True))
#Catch event
catch = 0
go_home = 0
```

Ta bắt đầu vào game (tức vào vòng lặp While True, đến khi game dừng thì thôi, hoặc bắt trở

về màn hình chính).
Đầu tiên là cho pacman di chuyển, bao gồm các bước sau:

- Lấy vị trí ô xung quanh với Heuristic cao nhất (index 0)
- Kiểm tra nếu ô đó là Food, cộng điểm, cho thức ăn biến mất. Kiểm tra đã ăn toàn bộ thức ăn thì cho game dừng lại (đã thắng).
- Cho pacman hiển thị đã di chuyển (lưu lại trạng thái cho pre_pos để lần sau kiểm tra có lặp lại để -2 điểm, đồng thời -1 điểm vì di chuyển 1 bước - đề)

```
# Get new move and check Foods
new_pacman_pos = list(heuristic_call.keys())[0]
if new_pacman_pos in foods_pos:
    count += 1
    food = ag.Food(self, (new_pacman_pos[1], new_pacman_pos[0]))
    food.food_disappear()
    self.pacman_scoring(s_score_gift)
    foods_pos.remove(new_pacman_pos)
#Check Win
if len(foods_pos) == 0:
    self.stage = s_display_f_vic
    break
#Pacman move
pre_pos = pacman_pos
pacman_pos = new_pacman_pos
pacman.pacman_control([pacman_pos[1], pacman_pos[0]])
self.pacman_scoring(s_score_move)
```

Ta kiểm tra xem nếu Pacman di chuyển vào vị trí quái vật nào không, nếu có thì game kết thúc, nếu không, ta tăng 1 đơn vị cho biến parity_index_monster cho Monsters di chuyển.

```
for pos in new_pos:
    if (pacman.co_or_pos[1], pacman.co_or_pos[0]) == pos:
        catch = 1
        break


parity_index_monster += 1
```

Bắt đầu cho quái vật di chuyển (ở đây khi code thì ta thấy Pacman chạy trước, Monster chạy sau, nhưng khi chạy tốc độ cao ta thấy hai vật di chuyển cùng một lúc, nếu cả hai cùng đâm

nhau thì cũng tương tự như kết thúc game).
Đầu tiên ta duyệt qua từng con quái vật, lấy vị trí gốc (khi khởi tạo của quái vật) gán vào biến element.

```
    for i in range(len(monster_list)):
        element = keys_monster[i]
```

Kiểm tra biến parity_index_monster, nếu:
parity_index_monster lẻ, tức con quái vật này sẽ di chuyển 1 ô xung quanh vị trí gốc (ngẫu nhiên), tương tự ta cũng xử lý cho quái vật di chuyển (xử lý đồ họa không cho mất Food, kiểm tra nếu đã bắt được,..)

```
    if (parity_index_monster % 2 == 1):
        #Save old position monster
        old_pos[i] = element
        size = len(monster_list[element])
        num = random.randint(0, size - 1)
        new_pos[i] = (monster_list[element][num][0], monster_list[element][num][1])
        #Don't get confict display food
        if ((old_pos[i][0], old_pos[i][1]) in foods_pos):
            foods_pos_draw_t = [old_pos[i][1], old_pos[i][0]]
            food = ag.Food(self, foods_pos_draw_t)
            food.food_display()
        #Monster move
        monster = ag.Monster(self, (old_pos[i][1], old_pos[i][0]))
        monster.monster_control([new_pos[i][1], new_pos[i][0]])
        #Check Failure (Catched)
        if monster.co_or_pos == pacman.co_or_pos:
            catch = 1
            break
        #Save old to new
        old_pos[i] = new_pos[i]
```

Ngược lại parity_index_monster chẵn, Monster sẽ quay lại trạng thái ban đầu (vị trí khi khởi tạo):

```
    else:
        if ((old_pos[i][0], old_pos[i][1]) in foods_pos):
            foods_pos_draw_t = [old_pos[i][1], old_pos[i][0]]
            food = ag.Food(self, foods_pos_draw_t)
            food.food_display()
```

```
        new_pos[i] = element
        monster = ag.Monster(self, (old_pos[i][1], old_pos[i][0]))
        monster.monster_control([new_pos[i][1], new_pos[i][0]])

        if monster.co_or_pos == pacman.co_or_pos:
            catch = 1
            break

        old_pos[i] = new_pos[i]
```

Cuối cùng, nếu vẫn chưa kết thúc, ta tính toán lại heuristic_call cho Pacman (để tiếp tục cho Pacman di chuyển tiếp), đến khi nào kết thúc game thì thôi. (Chú ý lúc này bắt đầu tính toán, nếu đi lặp lại ô sẽ bị trừ 2 điểm, đi bình thường trừ 1 điểm (+=1 sau đó nhân với (-1)).

```
    # Calculate heuristic for next move
    heuristic_call = {}
    for pos in graph_map[pacman_pos]:
        if pos == pre_pos:
            time_visits[pos] += 2
        else:
            time_visits[pos] += 1
        heuristic_call[pos] = (-1) * time_visits[pos] +
        ls.heuristic_lv3_2around(pacman_pos, graph_map, pos, foods_pos, old_pos)

    heuristic_call = dict(sorted(heuristic_call.items(), key=lambda item: item[1], reverse=True))
```

Giải thích một số hàm tính toán

```
    ls.heuristic_lv3_2around(pacman_pos, graph_map, pos, foods_pos, old_pos)
```

Hàm trên sẽ trả về giá trị Heuristic như phần ý tưởng, tổng số điểm Food có thể đi được (trong 2 bước) từ ô này, và cộng thêm với hệ số nếu có Monster.

```
    def heuristic_lv3_2around(pacman_pos,graph_map,position,foods_pos,monster_pos):
        '''Input: pacman position, graphmap, postion: vị trí kế của pacman_pos đang xét (trái, trên, phải, dưới)
        foods_pos (list of food pos[(),(),..]),prepos(position haved gone),
        monster_pos (list of monster post) [(),(),..]
        Output: Giá trị heuristic
        Check xem vị trí kế cận (1 bước đi) của pacman có phải là monster không, nếu có thì lấy
        -60, sau đó từ vị trí này lại xét thêm 2 bước nữa (vì pacman nhận biết được 3 bước xung quanh)
```

```
        Nếu vị trí đó là.
        Tất nhiên là mình sẽ không xét lại vị trí trước đó (pacman_pos, đi tới vị trí này).'''
        heu = 0
        if position in foods_pos:
            heu+=2
        if position in monster_pos:
            heu -= 60
        pre_pos = []
        pre_pos.append(position)
        pre_pos.append(pacman_pos)
        for pos in graph_map[position]:
            if pos == pacman_pos:
                continue
            heu = heu + heuristic_lv3_around(graph_map,pos,foods_pos,pre_pos,monster_pos)
        return heu
```

Như phần command giải thích, hàm trên chỉ kiểm trong phạm vi 1 ô đi từ ô này, tiếp tục kiểm tra thêm phạm vi 1 ô đi từ ô kế tiếp (hàm heuristic_lv3_around) (dùng pre_pos để tránh đi ngược lại)

```
    def heuristic_lv3_around(graph_map,position,foods_pos,pre_pos,monster_pos):
        '''Input: graphmap, position (pacman), foods_pos (list of food pos[(),(),..]),prepos(position haved gone),
        monster_pos (list of monster post) [(),(),..]
        Output: Giá trị heuristic
        Khi gặp monster cách pacman 2,3 bước đi thì giá trị sẽ lần lượt nhỏ hơn, -30, và -20'''
        heu = 0
        if position in foods_pos:
            heu+=2
        if position in monster_pos:
            heu -= 40
        for pos in graph_map[position]:
            if pos in pre_pos:
                continue
            if pos in foods_pos and pos not in pre_pos:
                pre_pos.append(pos)
                heu+=2
            elif pos in monster_pos:
                heu-=20
        return heu
```

3. Đánh giá

Bài làm đã đạt được:

- Đầu tiên, thuật toán đã giúp Pacman đi được trong điều kiện thiếu tầm nhìn. Với mục đích ăn được nhiều thức ăn nhất có thể, đồng thời tránh chết bởi những con quái vật.
- Đã áp dụng được Heuristic Local Search vào bài toán.

Tuy nhiên, bài làm vẫn có một số vấn đề như:

- Các hệ số điểm, được cho thủ công, qua nhiều lần thử (được tạo như là kinh nghiệm).
- Sau khi ăn hết thức ăn ở khu an toàn, thì sẽ phải đi lại rất lâu mới đi vào khu vực có quái vật (nguy hiểm), vì vậy điểm số thấp, bù lại ăn được nhiều thức ăn hơn.

Chapter IV

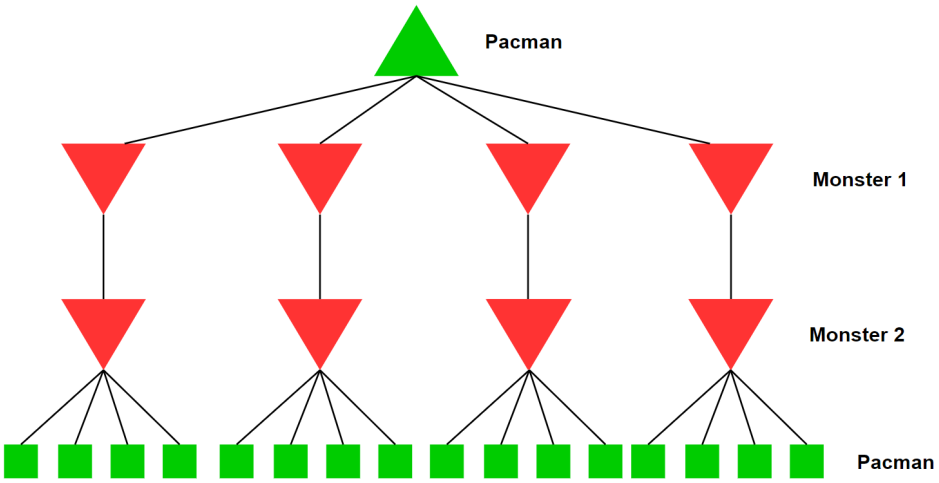
Level 4

1. Ý tưởng thuật toán

Nhắc lại bài toán: Bản đồ gồm Pacman, (có thể nhiều) thức ăn, quái vật, và tường. Quái vật sẽ liên tục truy đuổi Pacman, trong khi đó Pacman cố gắng ăn được càng nhiều thức ăn càng tốt mà không để chết, các con quái có thể di chuyển đè lên nhau. Cả hai bên đều di chuyển 1 lần mỗi lượt.

Ý tưởng: Để giải quyết bài toán thì ta sẽ áp dụng thuật toán Alpha-beta pruning. Đầu tiên chúng ta sẽ chọn depth = 16, đồng nghĩa với việc Pacman có thể dự đoán đường đi của 16 lần di chuyển tiếp theo bao gồm cả đường đi của Monsters. Thuật toán Alpha-beta pruning sẽ trả về đường đi tốt nhất trong vòng 20 lần di chuyển đó.

Thuật toán sẽ xét tất cả trường hợp có thể xảy ra của 16 nước đi tiếp theo và chọn ra đường đi có điểm lớn nhất và di chuyển 1 bước theo đường đi đó. Vì độ phức tạp của 1 lần di chuyển khi backtracking tất cả các trường hợp sẽ là $(\text{số monsters} + 1)^4$. Để giảm độ phức tạp thì nhóm em quyết định thay vì rẽ Monsters theo 4 nhánh thì sẽ cho Monsters chạy theo thuật toán search A*. Đường đi sẽ có dạng cây như hình sau.



Hình IV.1: Cấu trúc đường đi dạng cây

Từ vị trí ban đầu, Pacman sẽ di chuyển rẽ nhánh và khi depth = 0 (đi đến nút lá), thì sẽ tiến hành tính điểm và chọn được đường đi tốt nhất. Để giảm độ phức tạp, chúng ta sẽ

tiến hành so sánh và lưu lại alpha, beta cho mỗi nút, việc cắt nhánh sẽ được tiến hành khi $beta \leq alpha$.

2. Mô tả thuật toán

2.1 Đọc dữ liệu và hiển thị

Thực hiện tương tự câu 3

2.2 Bắt đầu vào game

Thực hiện vòng lặp While True cho đến khi gặp các trường hợp sau:

- Tất cả thức ăn hết và Pacman chiến thắng.

```
if len(foods_pos) == 0:
    self.stage = s_display_f_vic
    break
```

- Vị trí Pacman và vị trí Monster giống nhau. Suy ra Monters đã bắt được Pacman.

```
if monster.co_or_pos == pacman.co_or_pos:
    catch = 1
    self.stage = s_display_f_ove
    break
```

- Người điều khiển bấm nút trở về màn hình chính (go_home)

```
if go_home == 1:
    self.stage = s_display_home
```

2.3 Pacman di chuyển

Pacman sẽ tìm đường đi dựa trên hàm alpha-beta . Các tham số đầu vào của apha-beta lần lượt là:

- graph_map: lưu danh sách các ô liền kề (không tính tường) với ô hiện tại dưới dạng dictionary.
- game_state: bao gồm vị trí của Pacman(pacman_pos), vị trí Monsters(monsters_pos), vị trí của Foods(foods_pos), điểm (points) và số lượt Pacman đi qua ô đó (time_visits)
- depth: số lượng nước đi mà Pacman có thể dự đoán được
- player: nếu là lượt đi của Pacman thì có giá trị True, ngược lại nếu là lượt đi của Monsters thì có giá trị False
- alpha: giá trị lớn nhất mà maximizer có thể đảm bảo từ mức đó trở lên
- beta: giá trị nhỏ nhất mà minimizer có thể đảm bảo từ mức đó trở xuống

Giá trị player ban đầu sẽ là True, chúng ta sẽ giả định đường đi của Pacman trong vòng 1 bước đi.

Dùng hàm get_moves để có thể tìm được đường đi hợp lệ. Sau đó chuyển lượt đi cho Monsters bằng cách gọi lại hàm alpha-beta với giá trị player bằng False và depth giảm đi 1 đơn vị. Điều này có nghĩa là sau lượt 1 Pacman di chuyển chúng ta sẽ dự đoán nước đi của Monsters tương ứng với nước đi đó.
Ở lượt đi của Monster thay vì lấy hết đường đi có thể có thì chúng em sẽ lấy đường đi dựa trên thuật toán A*, để tránh việc cây rẽ nhánh nhiều nhất có thể.

```
        if player:  # Pacman
        max_eval = float('-inf')
        best_move = None
        list_states = get_moves(graph_map, game_state, True)
        if list_states == []:
            return evaluation_function(graph_map, game_state), None
        for sub_state in list_states:
            eval, temp = alpha_beta(graph_map, sub_state, depth - 1, False, alpha, beta)
            if eval > max_eval:
                max_eval = eval
                best_move = sub_state.pacman_pos
            alpha = max(alpha, max_eval)
            if beta <= alpha:
                break
        return max_eval, best_move
    else:
        min_eval = float('inf')
        best_move = None
        list_states = get_moves(graph_map, game_state, False)
        # for sub_state in list_states:
        eval, temp = alpha_beta(graph_map, list_states, depth - 1, True, alpha, beta)
        if eval < min_eval:
            min_eval = eval
            best_move = list_states.monsters_pos
        beta = min(beta, min_eval)
        return min_eval, best_move
```

Với mỗi lần get_moves chúng ta sẽ cập nhật điểm như sau:

- Nếu gặp thức ăn sẽ cộng 20 điểm vào new_points nếu không sẽ trừ đi 1 (cách tính điểm thông thường).

- Nếu vị trí mới của Pacman trùng với vị trí cũ trước đó của nó thì trừ đi 10 điểm. Việc này sẽ giúp Pacman tránh lặp đi lặp lại giữa các nhánh có điểm bằng nhau.

- Trừ đi time_visits của ô đó, giúp Pacman tránh lựa chọn những ô đã đi qua.

```
if player:  # pacman
  # Directions (left, up, right, down)
  directions = [(0, -1), (-1, 0), (0, 1), (1, 0)]

  for direction in directions:
      new_pacman_pos = (pacman_pos[0] + direction[0], pacman_pos[1] + direction[1])
      if is_valid_pos(graph_map, new_pacman_pos) and new_pacman_pos not in monsters_pos:
          new_points = points
          if new_pacman_pos in foods_pos:
              new_points += 20
          else:
              new_points -= 1
          if new_pacman_pos == pre_pos:
              new_points -=10
          new_points -= time_visits[new_pacman_pos]
          new_foods_pos = [pos for pos in foods_pos if pos != new_pacman_pos]
          new_game_state = GameState(new_pacman_pos, new_foods_pos,
          monsters_pos, new_points,time_visits,pre_pos)
          possible_moves.append(new_game_state)
  return possible_moves
```

Với mỗi lần duyệt đến depth = 0, chúng ta sẽ thực hiện tính điểm cho đường đi của nó dựa trên hàm evaluation_function.
Nếu đường đi của Pacman dự đoán sẽ gặp Monsters thì điểm sẽ bị -500. Điều này giúp cho Pacman tránh được việc đụng độ với Monsters và tìm kiếm đường có điểm cao hơn.
Hàm *get_extension* sẽ giúp cộng điểm cho những nước đi có độ mở rộng cao hơn (ít bị chắn bởi tường).

```
def evaluation_function(graph_map, game_state):
if game_state.pacman_pos in game_state.monsters_pos:
    game_state.points -= 500
game_state.get_extension(graph_map)
return game_state.points - game_state.time_visits[game_state.pacman_pos]
```

2.4 Monsters di chuyển

Bởi vì yêu cầu của đề là Monsters có thể đuôi theo Pacman nên chúng em thiết kế Monster theo thuật toán A* và random. Nếu chỉ sử dụng A* thì có thể các Monsters sẽ hợp lại với nhau và đuổi theo Pacman liên tục nên việc tạo thêm bước đi ngẫu nhiên sẽ phù hợp hơn. Với bước đi ngẫu nhiên, các Monsters cũng có thể ép Pacman vào góc thay vì truy đuổi từ 1 phía như A* đơn thuần.

- best_move_monster: Tìm đường đi đến Pacman bằng A*.
- random_move_monster: Random 1 đường đi bất kỳ trong số những đường đi hợp lệ.

3. Đánh giá

- Giúp Pacman dự đoán được đường đi tìm được nhiều thức ăn nhất và tránh khỏi Monsters.
- Độ phức tạp quá lớn, với mỗi bước đi, Pacman phải duyệt hết tất cả những trường hợp xảy ra.
- Đối với depth có giá trị nhỏ thì thuật toán có vẻ không hiệu quả và sẽ có thể đi đến những góc chết nguy hiểm.
- Các hệ số điểm, được cho thủ công, qua nhiều lần thử (được tạo như là kinh nghiệm).

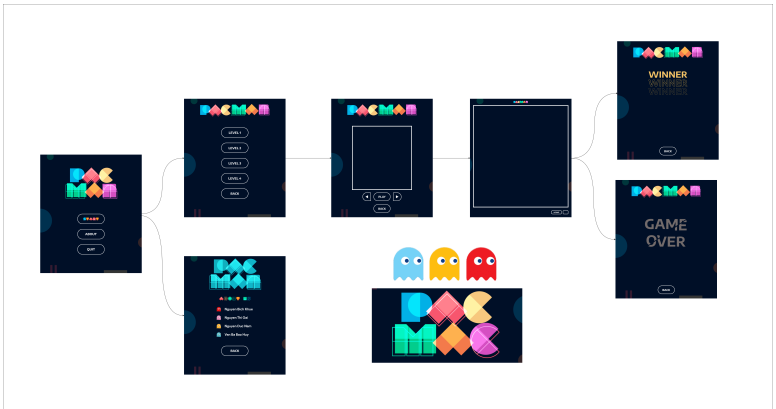
Chapter V

Graphic

1. Display

Class pacman_game chịu trách nhiệm khởi tạo và quản lý các yếu tố đồ họa của trò chơi Pacman. Hàm __init__ thực hiện các nhiệm vụ sau:

- Khởi tạo Pygame: Khởi tạo thư viện Pygame, cần thiết để tạo và quản lý các yếu tố đồ họa.
- Thiết lập cửa sổ: Tạo cửa sổ trò chơi bằng pygame.display.set_mode(). Điều này đặt kích thước, tiêu đề và biểu tượng của cửa sổ.
- Load font: Tải font từ đường dẫn được chỉ định bằng pygame.freetype.Font(). Font này được sử dụng để hiển thị văn bản trên màn hình.
- Load các giai đoạn hiển thị: Tải hình ảnh nền cho các giai đoạn khác nhau của trò chơi, chẳng hạn như màn hình chính, màn hình về, lựa chọn cấp độ, bản đồ, chơi game, chiến thắng và thua cuộc.
- Load bóng mờ của nút: Tải các phiên bản bóng mờ và không bóng mờ của hình ảnh nút, được sử dụng để chỉ ra liệu nút nào đó có đang được nhấp hay không.
- Khởi tạo biến trò chơi: Khởi tạo các biến trò chơi khác nhau, bao gồm giai đoạn hiện tại, vị trí chuột, chỉ số bản đồ, hình ảnh bản đồ, điểm số, cấp độ và tốc độ.



2. Map Builder

Đọc dữ liệu đầu vào, gồm có kích thước bản đồ và ma trận của bản đồ. Ở phần xây dựng bản đồ, ta chỉ quan tâm đến dữ liệu bao gồm tường và đường đi (đường đi bao gồm cả các vị trí của thức ăn, quái vật). Trong phần này, ta cho kích thước của mỗi ô là 20 pixels. Từ đó, ta tạo một bức ảnh trống với kích thước bằng kích thước của ma trận mỗi chiều nhân cho 20 (mỗi ô trong ảnh bao gồm 400 pixel tương ứng với một ô trong trò chơi) Từ đó, ta duyệt qua từng ô trong ma trận bản đồ và tô màu cho mỗi ô pixel để tạo thành bản đồ hoàn chỉnh.

```
def draw_maze_image(maze, maze_size, block_size):
    wall_color  = s_color_rose      # Color for walls
    empty_color = s_color_dblue     # Color for empty spaces

    width, height = maze_size[1] * block_size, maze_size[0] * block_size
    maze_image = Image.new("RGB", (width, height), empty_color)

    pixels = maze_image.load()
    for y in range(maze_size[0]):
        for x in range(maze_size[1]):
            for i in range(block_size):
                for j in range(block_size):
                    if x * block_size + i < width and y * block_size + j < height and maze[y][x] == 1:
                        pixels[x * block_size + i, y * block_size + j] = wall_color

    return maze_image
```

3. Agent

3.1. Pacman

Xây dựng class Pacman đại diện cho Pacman:

```
class Pacman:
    def __init__(self, app, start_position, cell=None):
        self.app = app
        #Start position
        self.co_or_pos = start_position
        self.pixel_pos = self.coor_to_pixel()
        self.direction = s_direction_R
        #Pacman
        self.character_image = pygame.image.load(s_character_path)
        self.character_img_R = pygame.image.load(s_character_R)
        self.character_img_L = pygame.image.load(s_character_L)
        self.blank_space     = pygame.image.load(s_blank_space)

        #Cell
        self.cell = cell

        #Pacman Memory
        self.detected_food = []             #Save all food detected by pacman
        self.path_to_detected_food = []     #Save pathway to detected food in memory

        # Limited visibility
        self.vision_food_list = []          # all food in pacman's threestep vision
        self.vision_mons_list = []          # all monsters in pacman's threestep vision
```

Tiếp theo chúng ta sẽ hiển thị Pacman ra màn hình với hướng di chuyển tương ứng và chuyển đổi tọa độ Pacman từ tọa độ bản đồ sang tọa độ Pixel.

```
#From Co-ordinare -> Pixel position to display
    def coor_to_pixel(self):
        pixel = ((self.co_or_pos[0] + 1) * 20, (self.co_or_pos[1] + 2) * 20)
        return pixel

    #Call pacman to display
    def pacman_call(self):
        #Call pacman for the first time
        self.display_character()
```

```
    #Display pacman in map with facing direction
    def display_character(self):
        #Display pacman
        if self.direction == s_direction_R:
            pygame.display.update(self.app.screen.blit(self.character_img_R, self.pixel_pos))
        elif self.direction == s_direction_L:
            pygame.display.update(self.app.screen.blit(self.character_img_L, self.pixel_pos))
```

- **def coor_to_pixel(self)**: hàm này dùng để chuyển đổi tọa độ $(x; y)$ của Pacman từ tọa độ bản đồ sang tọa độ pixel trên màn hình.
- **def pacman_call(self)**: "gọi" Pacman để hiển thị lên màn hình. Thường sẽ gọi lần đầu khi trò chơi bắt đầu.
- **def display_character(self)**: hàm này được sử dụng để hiển thị Pacman trên màn hình với hướng di chuyển tương ứng bao gồm:
  - Kiểm tra hướng di chuyển của Pacman (phải hoặc trái).
  - Sử dụng thư viện **pygame** để cập nhật hình ảnh của Pacman theo hướng di chuyển tương ứng.

Tiếp theo, chúng ta sẽ thực hiện các thaoc tác để điều khiển và cập nhật trạng thái của Pacman khi di chuyển trên bản đồ

```
#Pacman control
    def pacman_control(self, togo_pos):
        self.update(togo_pos)
        self.display_character()

    #Update pacman's data with new position
    def update(self, togo_pos):
        #Fill blank space to old position
        pygame.display.update(self.app.screen.blit(self.blank_space, self.pixel_pos))
        #Get direction data
        self.get_direction(togo_pos)
        #Get position data
        self.co_or_pos = togo_pos
        self.pixel_pos = self.coor_to_pixel()
```

```
    #Get pacman direction
    def get_direction(self, togo_pos):
        if togo_pos[0] - self.co_or_pos[0] == 1:      #Move Right
            self.direction = s_direction_R
        elif togo_pos[0] - self.co_or_pos[0] == -1:   #Move Left
            self.direction = s_direction_L
```

- **def pacman_control(self, togo_pos)**: Sử dụng để điều khiển Pacman đến vị trí mới **togo_pos**. Gồm hai bước chính: cập nhật trạng thái của Pacman với vị trí mới và hiển thị Pacman trên màn hình.

- **def update(self, togo_pos)**: Sử dụng để cập nhật trạng thái của Pacman sau khi di chuyển đến vị trí mới bao gồm:

  - Xóa bỏ hình ảnh Pacman ở vị trí cũ trước khi di chuyển đến vị trí mới.

  - Xác định hướng di chuyển của Pacman dựa trên vị trí mới.

  - Cập nhật trạng thái của Pacman với vị trí mới và tính toán tọa độ pixel tương ứng với vị trí mới.

- **def get_direction(self, togo_pos)**: Sử dụng để xác định hướng di chuyển của Pacman dựa trên vị trí mới.

3.2. Food:

Xây dựng class Food đại diện cho Food:

```
class Food:
    def __init__(self, app, goal_pos, cell=None):
        self.app = app
        self.co_or_pos = goal_pos
        self.pixel_pos = self.coor_to_pixel()
        #Food
        self.icon_image = pygame.image.load(s_food_path)
        self.blank_space = pygame.image.load(s_blank_space)
        #
        self.cell = cell
        #


    def coor_to_pixel(self):
```

```
        pixel = ((self.co_or_pos[0] + 1) * 20, (self.co_or_pos[1] + 2) * 20)
        return pixel

    def food_display(self):
        pygame.display.update(self.app.screen.blit(self.icon_image, self.pixel_pos))

    def food_disappear(self):
        pygame.display.update(self.app.screen.blit(self.blank_space, self.pixel_pos))
```

Định nghĩa một lớp có tên **Food**, đại diện cho thức ăn trong trò chơi. Lớp này chứa các phương thức và biến liên quan đến việc hiển thị và quản lý thức ăn.

3.3. Monster

Xây dựng lớp Monster để biểu thị một quái vật trong trò chơi và điều khiển vị trí, hình ảnh và hướng di chuyển của nó:

```
class Monster():
    def __init__(self, app, start_position, cell=None):
        self.app = app
        # Start position
        self.co_or_pos = start_position
        self.pixel_pos = self.coor_to_pixel()
        self.direction = s_direction_R
        # Monster
        self.character_image = pygame.image.load(s_character_monster)
        self.character_img_R = pygame.image.load(s_character_monster)
        self.character_img_L = pygame.image.load(s_character_monster)
        self.blank_space = pygame.image.load(s_blank_space)
        #
        self.cell = cell
        self.initial_cell = cell

    # Monster Function
    def coor_to_pixel(self):
        pixel = ((self.co_or_pos[0] + 1) * 20, (self.co_or_pos[1] + 2) * 20)
        return pixel

    def monster_call(self):
        # Call pacman for the first time
        self.display_character()
```

```
    def display_character(self):
        # Display pacman
        if self.direction == s_direction_R:
            pygame.display.update(self.app.screen.blit(self.character_img_R, self.pixel_pos))
        elif self.direction == s_direction_L:
            pygame.display.update(self.app.screen.blit(self.character_img_L, self.pixel_pos))

        # Pacman Control

    def monster_control(self, togo_pos):
        self.update(togo_pos)
        self.display_character()

    def update(self, togo_pos):
        pygame.display.update(self.app.screen.blit(self.blank_space, self.pixel_pos))
        self.get_direction(togo_pos)
        self.co_or_pos = togo_pos
        self.pixel_pos = self.coor_to_pixel()

    def get_direction(self, togo_pos):
        if togo_pos[0] - self.co_or_pos[0] == 1:  # Move Right
            self.direction = s_direction_R
        elif togo_pos[0] - self.co_or_pos[0] == -1:  # Move Left
            self.direction = s_direction_L

    def get_around_cells_of_initial_cell(self, graph_map):
        return graph_map[self.initial_cell]

    def get_around_cells(self, graph_map):
        return graph_map[self.cell]

    def monster_disappear(self):
        pygame.display.update(self.app.screen.blit(self.blank_space, self.pixel_pos))
```

Ở lớp này chúng ta sẽ thực hiện các thao tác sau:

- Khởi tạo Monster với vị trí ban đầu, hình ảnh, và môi trường ứng dụng.
- Chuyển đổi tọa độ của quái vật từ tọa độ trên bản đồ sang tọa độ pixel trên màn hình.

- Hiển thị hình ảnh của quái vật trên màn hình với hướng di chuyển tương ứng.
- Quản lý di chuyển của quái vật, cập nhật vị trí và hình ảnh của quái vật khi nó di chuyển.
- Xác định hướng di chuyển của quái vật dựa trên vị trí mới.
- Lấy các ô xung quanh quái vật trên bản đồ.
- Xóa hình ảnh của quái vật khỏi màn hình khi cần.