BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN, ĐHQG-HCM
KHOA CÔNG NGHỆ THÔNG TIN

MÔN HỌC: CƠ SỞ TRÍ TUỆ NHÂN TẠO
PROJECT 01: SEARCH

Giảng viên lý thuyết: **Bùi Duy Đăng**
Giảng viên thực hành: **Trần Quốc Huy**

Thành phố Hồ Chí Minh, 2023

MỤC LỤC

I. Giới thiệu:

STT	MSSV	Họ tên
1	21120439	Bùi Minh Duy
2	21120447	Nguyễn Nhật Hào
3	21120453	Tô Phương Hiếu
4	21120457	Lê Minh Hoàng

II. Phân bổ công việc:

Phần công việc	Công việc	Người thực hiện
Phần thuật toán	Cài đặt thuật toán level 1	Bùi Minh Duy Tô phương Hiếu
	Cài đặt thuật toán level 2	
	Cài đặt thuật toán level 3	Lê Minh Hoàng
	Cài đặt thuật toán level 4	Nguyễn Nhật Hào
Phần giao diện	Menu, màn hình bắt đầu, kết thúc game	Tô Phương HIếu
	Màn hình game chính, chọn map	Nguyễn Nhật Hào Lê Minh Hoàng
Thiết kế map	Map level 1, level 2	Bùi Minh Duy
	Map level 3, level 4	Nguyễn Nhật Hào
Báo cáo	Tổng hợp, trình bày	Tô Phương Hiếu

III. Môi trường:

- Ngôn ngữ: Python (version ≥ 3.0)

- Đồ họa: thư viện Pygame.

IV. Cấu trúc thư mục và hướng dẫn thực thi:

1. Cấu trúc thư mục:

```
./Group_6
    |_/Document
            |_/Report.pdf
    |_/Input
            |_/Level1
                    |_/map1.txt
                    |_/…
            |_/Level2
                    |_/map1.txt
                    |_/…
            |_/Level3
                    |_/map1.txt
                    |_/…
            |_/Level4
                    |_/map1.txt
                    |_/…
    |_/Source
            |_/Algorithms
                    |_/…
            |_/images
                    |_/…
            |_/Object
                    |_/…
            |_/Utils
                    |_/utils.py

            main.py
```

Thư mục **Input** chứa thông tin các map của trò chơi.

Thư mục **Algorithms** chứa các thuật toán cho các level với

- Level 1 và Level 2 là file **BFS.py**
- Level 3 là file **LocalSearch.py**
- Level 4 là file **Minimax.py** và file **Monster_Move.py**
- File **SearchAgent.py** sẽ là file quản lí các thuật toán cho các level và là file sẽ giao tiếp với file **main.py**

File **main.py** là file chạy chính của project.

2. Thực thi chương trình:

Bước 1: Bật console **cùng cấp** với file **main.py** (trong thư mục **Source**).

Bước 2: Nếu đã cài đặt python và pygame thì bỏ qua bước này. Cài đặt Python trên trang chủ **python.org**. Nếu chưa cài đặt pygame thì có thể dùng command line thực thi lệnh sau: **pip install pygame** hoặc **pip install –r requirements.txt .**

Bước 3: Để chạy chương trình dùng lệnh **py main.py** hoặc **python main.py.**

V. Đánh giá:

STT	Yêu cầu	Thực hiện	Hoàn thành
1	Level 1: Pacman biết vị trí của thức ăn trong bản đồ, không có Monster. Chỉ có một thức ăn tồn tại trên bản đồ	Sử dụng thuật toán Breadth-first search để tìm đường đi ngắn nhất đến thức ăn.	100%
2	Level 2: Monster không thể di chuyển, nếu Pacman và Monster va chạm thì trò chơi kết thúc. Vẫn chỉ có một thức ăn tồn tại trên bản đồ và Pacman biết vị trí của nó.		100%
3	Level 3: Tầm nhìn của Pacman bị giới hạn chỉ còn 3 đơn vị. Tức Pacman chỉ có thể "nhìn thấy" phạm vi 8 đơn vị xung quanh và mở rộng ra 3 đơn vị. Có nhiều thức ăn tồn tại trên bản đồ. Monster di chuyển từng bước một xung quanh vị trí bắt đầu. Với mỗi bước Pacman di chuyển, Monster cũng di chuyển.	Sử dụng thuật toán heuristic local search để tìm đường đi cho Pacman	100%

4	Level 4: Bản đồ kín. Monster sẽ truy đuổi nhằm tiêu diệt Pacman. Pacman sẽ cố gắng ăn nhiều thức ăn nhất có thể. Pacman sẽ thua cuộc nếu va chạm phải Monster. Monster có thể đi xuyên qua nhau. Với mỗi bước Pacman di chuyển, Monster cũng di chuyển. Có rất nhiều thức ăn.	Sử dụng thuật toán minimax để tìm đường đi cho Pacman. Sử dụng thuật toán A* để di chuyển Monster	100%
5	Biểu diễn đồ họa mỗi bước.	Sử dụng Pygame	100%
6	Tạo ít nhất 5 bản đồ với tường, Monster và thức ăn khác nhau.		100%
7	Báo cáo thuật toán		100%

VI. Thuật toán:

1. Level 1:

1.1. Mô tả:

- Pacman biết rõ vị trí thức ăn trên map nên vấn đề của level 1 là tìm đường đi ngắn nhất tới vị trí của thức ăn. Bởi vì chi phí cho mỗi bước đi là cố định (-1) nên không cần sử dụng thuật toán Uniform Cost Search hay bất kì thuật toán nào sử dụng giá trị heuristic.

- Ý tưởng ở đây là dùng thuật toán **BFS** để tìm ra đường đi ngắn nhất với trạng thái bắt đầu là vị trí xuất phát của pacman và đích của nó chính là thức ăn.

1.2. Ý tưởng thực hiện:

- Khởi tạo một ma trận visited để theo dõi các ô đã được duyệt qua và một ma trận trace để lưu vết theo dõi.

- Tìm vị trí của thức ăn gần nhất dựa trên khoảng cách **Manhattan** giữa các thức ăn và điểm bắt đầu bằng **find_nearest_food** (chương tình xây dựng cho trường hợp nếu có nhiều thức ăn trên bản đồ).

- Nếu không còn thức ăn nào cần ăn, trả về danh sách rỗng.

- Khởi tạo một danh sách (queue) và biến kiểm tra để duyệt BFS.

- Bắt đầu vòng lặp chạy BFS:

    + Loại bỏ điểm đầu tiên từ danh sách (queue).

+ Kiểm tra xem vị trí hiện tại có trùng với điểm đích (thức ăn gần nhất) không. Nếu có, đánh dấu đã tìm thấy đường đi và thoát khỏi vòng lặp.

+ Duyệt qua các hướng di chuyển có thể (4 ô chung cạnh với vị trí hiện tại), nếu là ô hợp lệ thì thêm vào cuối hàng đợi, đồng thời lưu vết của vị trí vừa đi qua.

- Nếu không tìm thấy đường đi đến thức ăn gần nhất, loại bỏ thức ăn này khỏi danh sách và gọi lại hàm BFS đệ quy để tìm thức ăn gần nhất còn lại. (Trong trường hợp này chỉ có 1 thức ăn duy nhất)

- Nếu tìm thấy đường đi đến thức ăn gần nhất, tái tạo đường đi thông qua ma trận lưu vết trace.

- Trả về đường đi từ điểm bắt đầu đến thức ăn gần nhất mà thuật toán tìm được.

1.3. Mã giả:

Breadth-first search on a graph

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?( frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored and not in frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

10

2. Level 2:

2.1 Mô tả:

- Yêu cầu tương tự level 1, chỉ thêm Monsters nhưng mà đứng yên tại chỗ. Để giải quyết thì ta xem Monster như tường vì chúng không di chuyển.

- Thuật toán **tương tự như level 1**. (BFS với trạng thái bắt đầu là vị trí xuất phát của Pacman và đích của nó chính là thức ăn)

2.2. Ý tưởng thực hiện: (tương tự level 1)

2.3. Mã giả: (tương tự level 1)

3. Level 3:

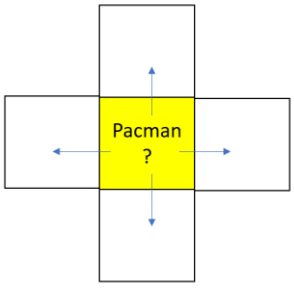
3.1. Mô tả:

- Ở level 3, Pacman không thể nhìn thấy thức ăn nếu ở phạm vi ngoài 3 bước đi. Pacman và Monsters di chuyển mỗi lần 1 bước thay phiên nhau.

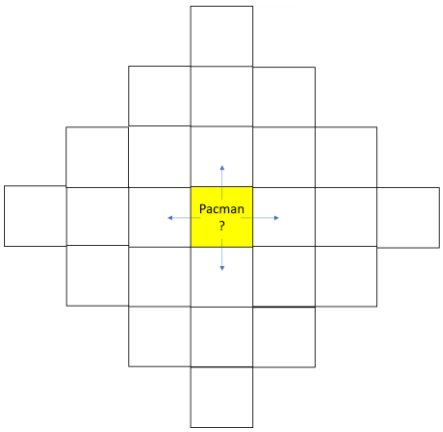
- Trong trường hợp này, Pacman phải ăn được thức ăn và tránh được Monsters. Ý tưởng ở đây dùng **heuristic local search**.

3.2. Ý tưởng thực hiện:

- Local search giúp xác định ô tiếp theo mà Pacman sẽ di chuyển tiếp.



- Tầm nhìn của Pacman bị giới hạn trong ba bước đi nên ở đây sẽ sử dụng thuật toán 3 level of Depth Limited Search (DLS) kể từ ô Pacman đang đứng.

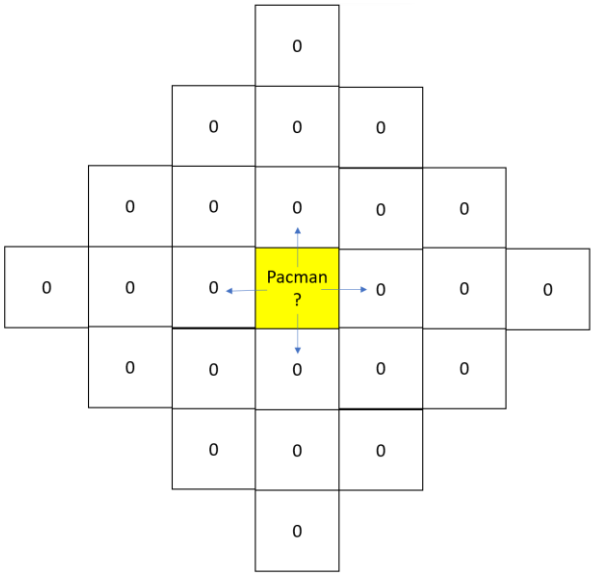


- Mỗi ô sẽ có 2 thuộc tính chính:

  + cost : khởi tạo ban đầu bằng 0 cho mỗi lần gọi hàm **calc_heuristic.**

  + _visited : khởi tạo ban đầu bằng 0 và sẽ tăng lên 1 đơn vị cho mỗi lần Pacman di chuyển qua ô đó.

- Để giúp Pacman quyết định được ô di chuyển tiếp theo, chúng ta sẽ chạy hàm **calc_heuristic** để tính cost lại mỗi ô. Ban đầu, cost lại mỗi ô đều có giá trị bằng 0.

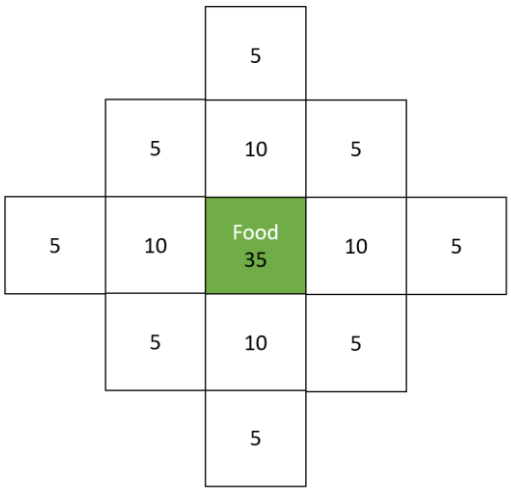


- Các bước thực thi:

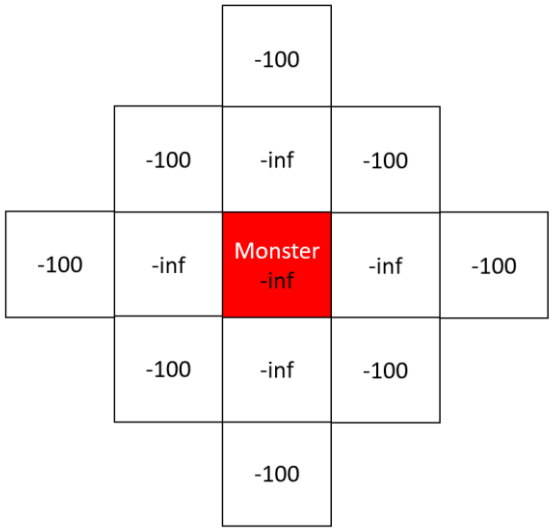
- **Bước 1:** Khởi tạo mảng 2 chiều _visited, cost với tất cả phần tử có giá trị bằng 0
- **Bước 2:** Ta thực hiện duyệt qua các ô theo DLS với depth bằng 3 (tính từ ô hiện tại) (main)
- **Bước 3:** Tại mỗi ô được duyệt, ta kiểm tra xem vị trí đó có thức ăn hay là monster không. Nếu vị trí đó là:

  + Ô trống thì quay lại **bước 2** (main)

  + Thức ăn: thực hiện DLS (sub) với depth bằng 2 duyệt qua các ô, tính toán lại giá trị heuristic tại các ô đó. Tính từ ô đang xét, giá trị biến depth càng giảm thì giá trị heuristic cộng vào càng giảm.



  + Monster: thực hiện DLS (sub) với depth bằng 2 duyệt qua các ô , tính toán lại giá trị heuristic tại các ô đó. Độ sâu bằng 0 (ô đang xét) và độ sâu bằng 1 (ô kề cận) thì giá trị tại đó bằng –inf (nhằm tránh Monsters ở gần Pacman), còn độ sâu bằng 2 thì giá trị tại vị trí đó bị trừ đi 100.



- **Bước 4:** Thực hiện tính toán và cập nhật giá trị heuristic tại các ô cho đến khi duyệt qua hết tất cả các ô theo DLS (main)
- **Bước 5:** Sau khi tính toán giá trị heuristic cho tất cả các ô trong phạm vi Pacman nhìn thấy, thực hiện tìm ô mà có giá trị **f(n)** cao nhất (ô kề cận với vị trí Pacman hiện tại) và trả về vị trí của ô đó nếu có giá trị lớn hơn giá trị Pacman đang đứng.

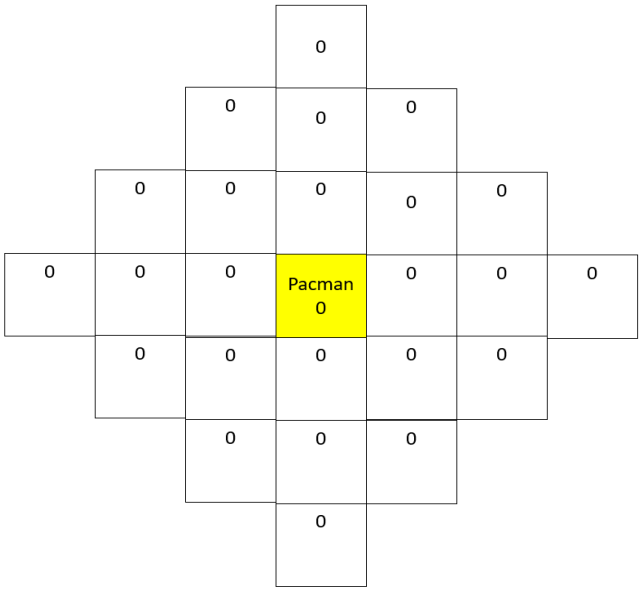
$$f(n) = h(n) - v(n)$$

Trong đó: h(n) là hàm heuristic, v(n) là _visited

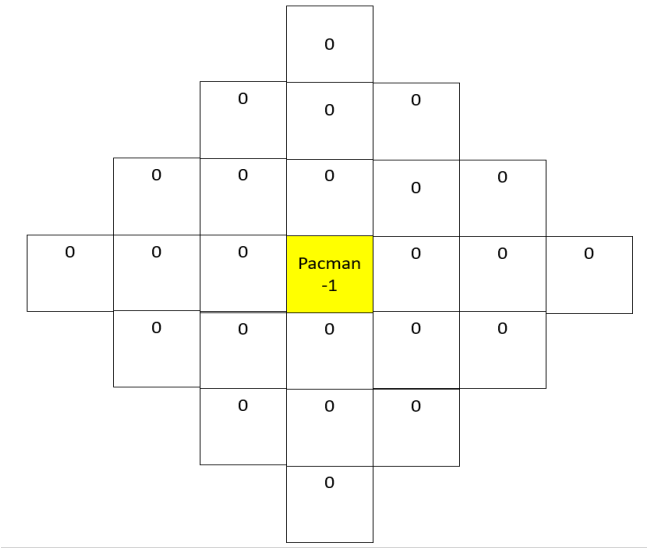
- **Bước 6:** Sau mỗi lần di chuyển qua 1 vị trí (row,col) thì thực hiện cộng dồn vào _visited[row][col] lên 1 đơn vị nhằm tránh cho Pacman đi qua đi lại một vị trí nhiều lần mà không cần thiết.

- Minh họa bằng hình vẽ các trường hợp:

+**TH1:** không có bất cứ thức ăn hay Monsters trong phạm vi 3 bước đi

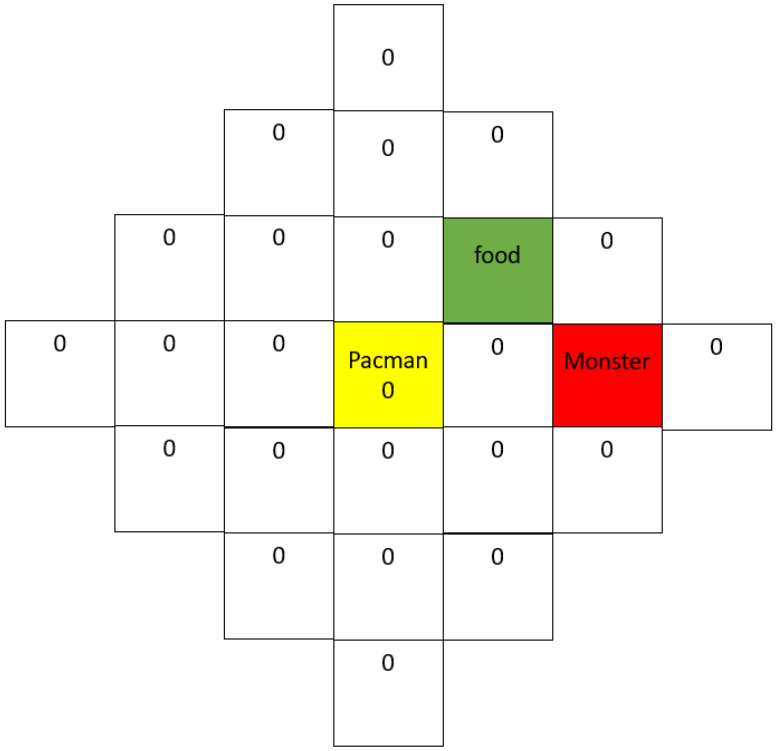


Ta thấy, sau khi tính toán xong các giá trị, _vistited tại vị trí Pacman tăng lên 1

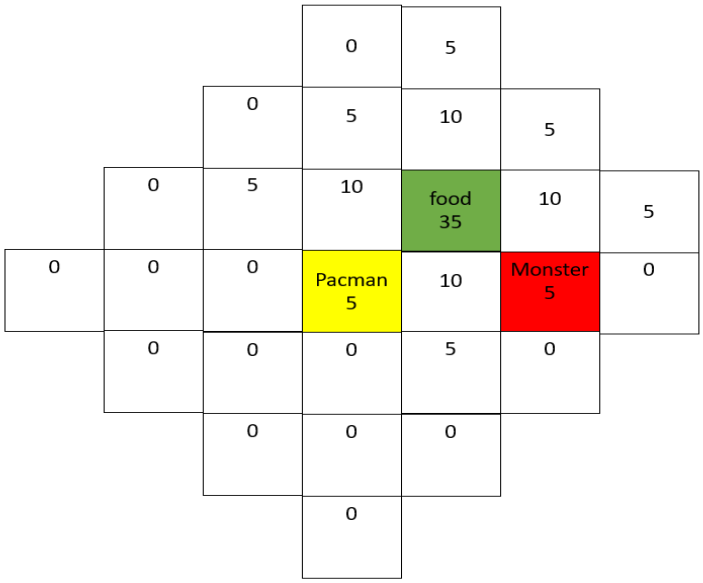


Lúc này, Pacman sẽ di chuyển qua bất kì một vị trí lân cận mà hợp lệ và không phải là tường.

+ **TH2:** có thức ăn hay Monster trong phạm vi 3 bước đi



Giả sử, khi duyệt qua thức ăn, lúc này tại vị trí thức ăn sẽ thực hiện duyệt DLS (depth=2) để cập nhật lại giá trị tại các ô được duyệt qua.



Tiếp theo, khi duyệt qua vị trí Monster (DLS main) , tại vị trí này sẽ thực hiện duyệt DLS (sub, depth=2) để cập nhật lại giá trị tại các ô được duyệt qua.

Sau khi thực hiện duyệt qua các vị trí, giá trị heuristic được cập nhật xong. Ta di chuyển Pacman đến ô kế tiếp có giá trị **f(n)** lớn nhất.

3.3. Mã giả:

function LocalSearch(*problem*) **returns** the state that is the most effective

current ← MAKE-NODE(problem.INITIALSTATE)

calc_heuristic(current)

loop do *neighbor* ← a highest cost successor of *current*

if *neighbor*.VALUE ≤ current.VALUE **then return** *current*.STATE

current ← *neighbor*

3.4. Nhận xét:

- Vì Pacman có tầm nhìn giới hạn nên không tránh khỏi việc phải đi qua đi lại một cung đường nhiều lần để tìm nước đi đến thức ăn tiếp theo. Song khả năng di chuyển của Monsters là **ngẫu nhiên** nên sẽ có trường hợp Pacman va chạm vào Monster vì tính ngẫu nhiên này + tầm nhìn giới hạn của Pacman.

- Ưu điểm:

    + Heuristic phù hợp giúp thuật toán tìm kiếm cục bộ tìm được lời giải tốt trong vùng lân cận.

- Khuyết điểm:

    + Nếu heuristic không đủ tốt hoặc không chính xác thì thuật toán có thể tìm ra giải pháp không tối ưu.

4. Level 4:

4.1. Mô tả:

- Monsters sẽ truy đuổi nhằm tiêu diệt Pacman. Pacman sẽ cố gắng ăn nhiều thức ăn nhất có thể. Pacman sẽ thua cuộc nếu va chạm phải Monster. Monsters có thể đi xuyên qua nhau. Với mỗi bước Pacman di chuyển, Monsters cũng di chuyển. Có rất nhiều thức ăn.

- Trong trường hợp này, Pacman phải ăn được thức ăn và tránh được Monsters. Ý tưởng ở đây dùng thuật toán minimax nhằm mô phỏng các trường hợp có thể.

4.2. Thuật toán di chuyển của Pacman:

a) Thuật toán: minimax

b) Mã giả:

```
function minimax(node, depth, maximizingPlayer):
    if depth is 0 or node is a terminal node:
        return the heuristic value of node

    if maximizingPlayer then
        bestValue <- −∞
        for each child in node:
            value <- minimax(child, depth - 1, False)
            bestValue <- max(bestValue, value)
        return bestValue
    else
        bestValue <- +∞
        for each child in node:
            value <- minimax(child, depth - 1, True)
            bestValue <- min(bestValue, value)
        return bestValue
```

```
# Trong hàm gọi đầu tiên, bạn sẽ gọi minimax với maximizingPlayer là True và depth tùy thuộc vào độ sâu tối đa bạn muốn xem xét.

# Hàm minimax sẽ tìm giá trị tối ưu cho maximizingPlayer ở đây là Pacman, tùy thuộc vào các giá trị của các nút con.
```

c) Mô tả: Minimax sẽ được tách thành hai hàm nhỏ là max_value và min_value.

- Hàm max_value dùng để tìm phương án đi tối đa cho **Pacman** theo bốn ô (hợp lệ) chung cạnh với vị trí đang xét.

- Hàm min_value dùng để giả định phương án đi tối đa cho **Monsters**.

- Việc xác định thế nào là phương án tối đa sẽ phụ thuộc vào hàm evaluationFunction được thiết lập theo các tiêu chí nhất định (điểm số hiện tại, khoảng cách đến các đối tượng khác).

d) Cài đặt:

- Hàm minimaxAgent:

Input: _map là mảng hai chiều với N hàng M cột để lưu thông tin của bản đồ hiện tại; pac_row và pac_col sẽ lưu vị trí hiện tại của Pacman; depth lưu độ sâu tối đa của thuật toán minimax; Score lưu thông tin điểm số mà Pacman có được đến thời điểm hiện tại.

Output: Vị trí đi tiếp theo (trong bốn ô chung cạnh) của Pacman.

Mô tả: Hàm sẽ thực hiện việc tính toán minimax theo evaluationFunction đã thiết lập trước để trả về vị trí đi tiếp theo của Pacman (tối ưu nhất có thể).

- Hàm terminal:

Input: _map là mảng hai chiều với N hàng M cột để lưu thông tin của bản đồ hiện tại; pac_row và pac_col sẽ lưu vị trí hiện tại của Pacman; depth lưu độ sâu tối đa của thuật toán minimax;

Output: trả về **True** nếu trạng thái hiện tại của trò chơi đã có kết quả (hết thức ăn, depth bằng 0, Monster đã bắt được Pacman). Ngược lại trả về **False**.

Mô tả: Hàm sẽ thực hiện kiểm tra trò chơi đã có kết quả hay chưa dựa vào các thông tin được cung cấp khi gọi hàm (vị trí Pacman, thông tin map).

Trò chơi được coi là kết thúc khi:

- Monster và Pacman va chạm nhau.

- Depth của thuật toán minimax bằng 0.
- Pacman đã ăn hết thức ăn có trên bản đồ.

- Hàm evaluationFunction:

Input: : _map là mảng hai chiều với N hàng M cột để lưu thông tin của bản đồ hiện tại; pac_row và pac_col sẽ lưu vị trí hiện tại của Pacman; score lưu thông tin điểm số mà Pacman có được đến thời điểm hiện tại.

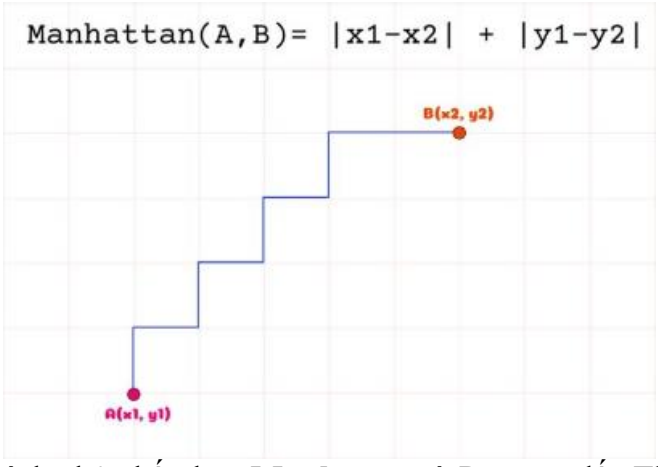
Output: Hàm trả về điểm số theo công thức được thiết lập trước, giá trị này dùng để xác định vị trí tiếp theo của Pacman.

Mô tả:

- Công thức để tính điểm số theo heuristic (chấp nhận được):

$$Điểm\ số\ hiện\ tại + \frac{WEIGHT\_FOOD}{Max(1,\ \ khoảng\ cách\ đến\ thức\ ăn\ gần\ nhất)} + \sum_{k=0}^{n} \frac{WEIGHT\_GHOST}{khoảng\ cách\ đến\ Ghost\ thứ\ k}$$

với **WEIGHT_FOOD=100**, **WEIGHT_GHOST=-150** và khoảng cách được tính theo công thức **Manhattan.**



- Đầu tiên ta sẽ tìm khoảng cách nhỏ nhất theo **Manhattan** từ Pacman đến Thức ăn rồi cộng vào điểm số hiện tại theo công thức trên.
- Tiếp theo ta tìm tất cả khoảng cách từ Pacman đến Monsters rồi thế vào công thức để tính điểm. Trong trường hợp khoảng cách từ Pacman đến Monster là 0 (Pacman và Monster va chạm nhau) thì trả về điểm số -inf và kết thúc hàm.

- Việc xác định giá trị của **WEIGHT_FOOD** và **WEIGHT_GHOST** trong công thức được xác định thông qua quá trình chạy thuật toán (thử nghiệm).

Sở dĩ **|WEIGHT_FOOD| < |WEIGHT_GHOST|** là do ta ưu tiên cho việc tránh chạm mặt Monster thay vì ưu tiên ăn thức ăn.

- Hàm max_value:

Input: _map là mảng hai chiều với N hàng M cột để lưu thông tin của bản đồ hiện tại; pac_row và pac_col sẽ lưu vị trí hiện tại của Pacman; depth lưu độ sâu tối đa của thuật toán minimax; score lưu thông tin điểm số mà Pacman có được đến thời điểm hiện tại.
Output: Điểm số tối đa mà Pacman có thể kiếm được theo thông tin bản đồ và vị trí hiện tại.
Mô tả: Hàm max_value dùng để xác định điểm tối đa mà Pacman có thể kiếm được (nhằm chọn vị trí đi tiếp theo) theo thông tin bản đồ và vị trí hiện tại của Monsters, Pacman và thức ăn.

- Đầu tiên ta giả định điểm số có thể đạt được sẽ là **âm vô cùng** và được lưu trong biến **v**.

- Tại vị trí Pacman hiện tại, ta mở rộng ra bốn ô chung cạnh. Nếu là ô hợp lệ (không phải tường và không có Monster), Pacman sẽ thử đi đến ô đấy, đồng thời cập nhật thông tin bản đồ, điểm số trò chơi và vị trí mới của Pacman.

- Khi Pacman đi đến ô mới, giá trị **v** sẽ được cập nhật bằng max của giá trị **v** hiện tại và điểm số của Monsters đi (hàm **min_value**).

$$v = \max(v, \min\_value(Monsters))$$

- Tuy nhiên khi hàm max_value được gọi ta sẽ phải kiểm tra trạng thái của game đã có kết quả hay chưa (tránh lặp vô tận) bằng việc gọi hàm terminal đã mô tả ở trên. Nếu đã kết thúc thì trả về kết quả của hàm evaluationFunction. Ngược lại thì tiếp tục như mô tả.

- Hàm min_value:

Input: _map là mảng hai chiều với N hàng M cột để lưu thông tin của bản đồ hiện tại; pac_row và pac_col sẽ lưu vị trí hiện tại của Pacman; depth lưu độ sâu tối đa của thuật toán minimax; score lưu thông tin điểm số mà Pacman có được đến thời điểm hiện tại.

Output: Điểm số tối đa mà đối tượng Monster có thể kiếm được theo thông tin bản đồ và vị trí hiện tại (giá trị này là số âm nhỏ nhất nhằm gây ảnh hưởng đến việc tính toán hàm max_value của Pacman).

Mô tả: Hàm min_value dùng để giả định cách đi tối đa cho Monsters (nhằm giúp Pacman chọn nước đi tối ưu nhất có thể) theo thông tin bản đồ, vị trí hiện tại của Monsters và Pacman.

- Đầu tiên ta giả định điểm số có thể đạt được sẽ là **dương vô cùng** và được lưu trong biến **v**.

- Xét tất cả các vị trí của Monsters hiện có trên bản đồ, với mỗi Monster ta mở rộng ra bốn ô chung cạnh. Nếu là ô hợp lệ (không phải tường), Monster sẽ thử đi đến ô đấy, đồng thời cập nhật thông tin bản đồ, vị trí mới của Monster đó.

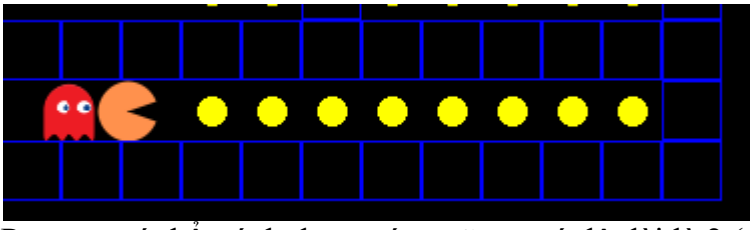
- Khi Monster đi đến ô mới, giá trị **v** sẽ được cập nhật bằng min của giá trị **v** hiện tại và điểm số của Pacman đi (hàm **max_value**).

$$v = \min(v, \max\_value(Pacman))$$

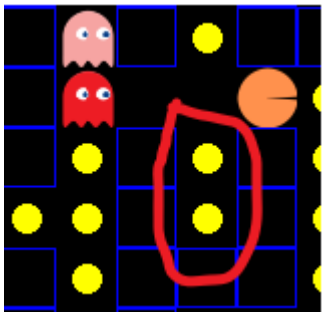
- Tuy nhiên khi hàm min_value được gọi ta sẽ phải kiểm tra trạng thái của game đã có kết quả hay chưa (tránh lặp vô tận) bằng việc gọi hàm terminal đã mô tả ở trên. Nếu đã kết thúc thì trả về kết quả của hàm evaluationFunction. Ngược lại thì tiếp tục như mô tả.

e) Nhận xét:

- Thuật toán minimax thực hiện xét tất cả các trạng thái đi lần lượt của Pacman và Monster nên khi bản đồ có số hàng và số cột lớn hay số lượng Monster nhiều thì sẽ làm cho thuật toán chạy chậm. Để thời gian chạy thuật toán là chấp nhận được (có thể quan sát quá trình chạy ngay) ta sẽ dùng thêm một biến **depth=4** để giới hạn độ sâu của cây trò chơi, mỗi lần gọi hàm **min_value** hay **max_value** thì giá trị **depth** sẽ giảm đi một đơn vị, khi giá trị **depth** về **không** thì ta sẽ dừng việc mở rộng xuống cây trò chơi và trả về kết quả cho node cha.
- Vì với thông tin bản đồ hiện tại, thuật toán không thể mô phỏng tất cả các trường hợp tiếp theo của trò chơi (do biến **depth** ở trên) nên sẽ có các trường hợp nước đi tiếp theo của **Pacman** sẽ thật sự không tối ưu nhất. Một trường hợp trong đó là khi bản đồ có một lối đi dài vào ngõ cụt (như hình):



- Tuy nhiên với **depth=4** Pacman có thể tránh được các ngõ cụt có độ dài là 2 (như hình):



- Với thuật toán này để có thể quan sát được quá trình chạy ngay, thì số Monsters tối đa là **5** với kích thước bản đồ là **tùy chọn**.

4.3. Thuật toán di chuyển của Monster:

*a) Thuật toán: A**

b) Mã giả:

```
1. function A_Star(start, goal, h)
2.       openSet <- {start}
3.       cameFrom <- an empty map
4.       gScore <- map with default value of Infinity
5.       gScore[start] <- 0
6.
7.       fScore <- map with default value of Infinity
8.       fScore[start] <- h(start)
9.
10.     while openSet is not empty
11.         current <- the node in openSet having the lowest
    fScore[] value
12.         if current = goal
13.             return reconstruct_path(cameFrom, current)
14.
15.         openSet.Remove(current)
16.         for each neighbor of current
```

```
17.             tentative_gScore <- gScore[current] + d(current,
    neighbor)
18.             if tentative_gScore < gScore[neighbor]
19.                 cameFrom[neighbor] <- current
20.                 gScore[neighbor] <- tentative_gScore
21.                 fScore[neighbor] <- tentative_gScore +
    h(neighbor)
22.                 if neighbor not in openSet
23.                   openSet.add(neighbor)
24.
25.       // Open set is empty but goal was never reached
26.       return failure
```

c) Mô tả:

- Ở **level 4** Monsters không ngừng truy đuổi Pacman, để hiện thực việc đó ở mỗi bước di chuyển của Pacman, Monsters sẽ chạy lại thuật toán **A*** để tìm đường đi tối ưu. Thay vì trả về toàn bộ đường đi đến Pacman ta sẽ trả về nước đi đầu tiên (do mỗi lần Monster di chuyển thì Pacman cũng di chuyển) cho Monsters.

d) Cài đặt:

- Hàm Monster_move_level4:

Input: _map là mảng hai chiều với N hàng M cột để lưu thông tin của bản đồ hiện tại; start_row và start_col sẽ lưu vị trí hiện tại của Monster đang xét; end_row và end_col sẽ lưu vị trí hiện tại của Pacman.

Output: Tọa độ ô tiếp theo (trong bốn ô chung cạnh với vị trí hiện tại) mà Monster sẽ di chuyển đến.

Mô tả:

- **Bước 1:** Khởi tạo các biến trace (lưu vết đường đi), cost (lưu chi phí đường đi theo heuristic), path (lưu đường đi), queue là một hàng đợi ưu tiên, visited là mảng hai chiều lưu thông tin đã thăm ô đó hay chưa (ban đầu tất cả bằng False). Thêm vị trí hiện tại của Monster vào hàng đợi với **độ ưu tiên** là khoảng cách từ Monster đến Pacman theo công thức **Manhattan** (khoảng cách nhỏ nhất sẽ ở đầu hàng đợi)**.**

- **Bước 2:** Nếu trong hàng đợi queue có phần tử:

+ Lấy phần tử đầu hàng đợi ra. Đánh dấu là đã thăm vị trí này.

+ Nếu phần tử này là vị trí đích thì ta thực hiện truy vết đường đi rồi trả về phần tử đầu đường đi (khác vị trí start_row, start_col).

+ Nếu không, thực hiện **bước 3**.

- **Bước 3:** Xét 4 ô chung cạnh với vị trí phần tử đang xét, nếu ô đấy là vị trí đi hợp lệ (khác tường):

+ Tính toán giá trị cost của vị trí đó theo heuristic sau:

$$cost(\text{vị trí mới}) = cost(\text{vị trí cũ}) + 1 + Manhattan(\text{vị trí mới}, \text{vị trí } Pacman)$$

+ Thêm vị trí mới này vào hàng đợi ưu tiên theo giá trị cost vừa tính được, đồng thời cũng lưu lại vết đường đi.

- Quay lại **Bước 2**.

e) Nhận xét:

- Vì Monster có thể đi qua các Monster khác nên sẽ có trường hợp hai Monster di chuyển trùng nhau (quá trình chạy thuật toán hoàn toàn giống nhau).

- Vì sau mỗi bước Monsters di chuyển thì Pacman cũng di chuyển đến vị trí mới nên thuật toán chỉ cần vị trí tiếp theo mà Monsters sẽ đi (không phải toàn bộ đường đi đến vị trí Pacman đang xét).

Tài liệu tham khảo

- UC-Berkeley-AI-Pacman-Project (https://github.com/karlapalem/UC-Berkeley-AI-Pacman-Project).
- Pygame docs (https://www.pygame.org/docs/).