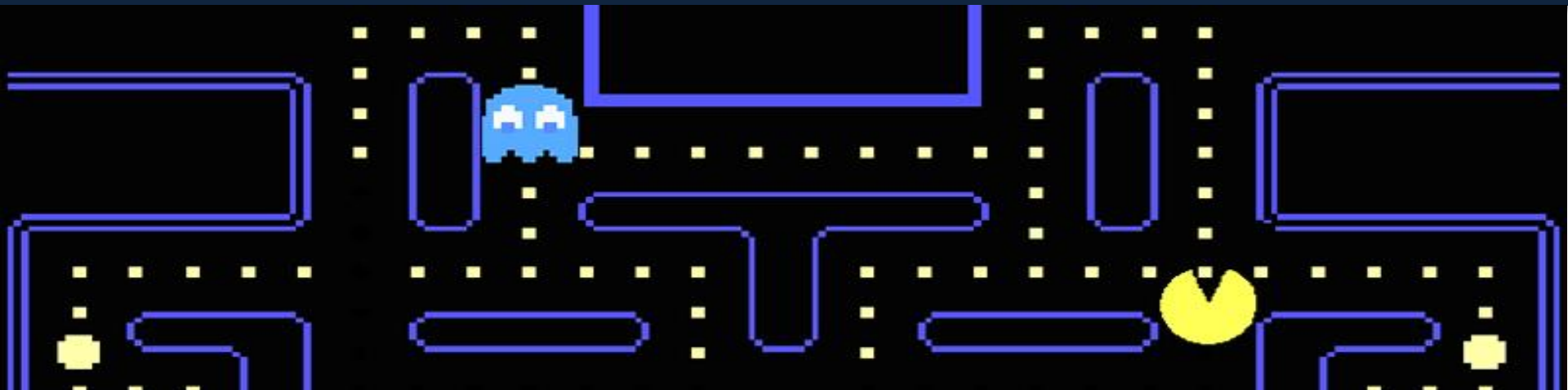
ĐỒ ÁN GIỮA KỲ

NHẬP MÔN TRÍ TUỆ NHÂN TẠO



THÀNH VIÊN NHÓM 4

- Nguyễn Quốc Anh – 52100871 – 52100871@student.tdtu.edu.vn
- Nguyễn Vũ Tường – 52100944 – 52100944@student.tdtu.edu.vn
- Võ Phú Vinh – 52100947 – 52100947@student.tdtu.edu.vn
- Trần Thị Vẹn – 52100674 – 52100674@student.tdtu.edu.vn

UNINFORMED SEARCH

TỔNG QUAN CÂU 1

```
├── problems.py
│   ├── SingleFoodSearchProblem
│   │   ├── State
│   │   ├── Node
│   │   ├── Initial state
│   │   ├── Successor function
│   │   ├── Goal-test function
│   │   ├── Path-cost function
│   │   ├── read maze function
│   │   └── print maze function
│   └── MultiFoodSearchProblem
│       ├── State
│       ├── Node
│       ├── Initial state
│       ├── Successor function
│       ├── Goal-test function
│       ├── Path-cost function
│       ├── read maze function
│       ├── print maze function
│       └── animate function
├── fringes.py
│   ├── Stack
│   ├── Queue
│   └── PriorityQueue
└── searchAgents.py
    ├── bfs(problem)
    ├── dfs(problem)
    ├── ucs(problem)
    ├── astar(problem, fn_heuristic)
    ├── gbfs(problem, fn_heuristic)
    ├── SingleFoodSearchProblem.animate(actions)
    ├── MultiFoodSearchProblem.get_successors(state)
    ├── MultiFoodSearchProblem.get_cost_of_actions(actions)
    ├── MultiFoodSearchProblem.is_goal_state(state)
    ├── MultiFoodSearchProblem.get_initial_state()
    ├── MultiFoodSearchProblem.heuristic(state)
    └── MultiFoodSearchProblem.animate(actions)
```

TÓM TẮT

Tóm tắt các phần sẽ có trong bài toán này:

- Vấn đề tìm kiếm đường đi trong mê cung.
- Giới thiệu về trò chơi Pacman và mục tiêu giải quyết vấn đề tìm kiếm đường đi.
- Các giải thuật tìm kiếm mù được sử dụng: tìm kiếm theo chiều rộng, tìm kiếm theo chiều sâu và tìm kiếm đồng nhất.
- Các lớp và hàm được sử dụng trong sơ đồ giải quyết vấn đề tìm kiếm đường đi trong mê cung.
- Cách thức áp dụng các lớp và hàm để giải quyết vấn đề tìm kiếm đường đi trong mê cung.

VẤN ĐỀ TÌM KIẾM

Giới thiệu tổng quan về bài toán tìm kiếm đường đi trong mê cung:

- Bài toán tìm kiếm đường đi trong mê cung là bài toán phổ biến trong lĩnh vực trí tuệ nhân tạo và khoa học máy tính.
- Mục tiêu của bài toán là tìm đường đi ngắn nhất từ một vị trí bắt đầu đến một điểm kết thúc trong mê cung.

Những ứng dụng thực tiễn của bài toán:

- Bài toán tìm kiếm đường đi trong mê cung có rất nhiều ứng dụng thực tiễn trong đời sống, như: hệ thống dẫn đường trên ô tô, máy bay, tàu hỏa; hệ thống điều khiển robot tự động; hệ thống tìm kiếm tuyến đường giao thông công cộng, v.v.

PACMAN & MỤC TIÊU GIẢI QUYẾT VẤN ĐỀ

Các thông tin cần biết về Pacman:

- Giới thiệu về trò chơi Pacman và cách chơi
- Liên kết giữa trò chơi Pacman và bài toán tìm kiếm đường đi trong mê cung
- Mục tiêu giải quyết bài toán tìm kiếm đường đi trong mê cung trong trò chơi Pacman

GIẢI THUẬT

Các giải thuật sử dụng:

- Tìm kiếm theo chiều rộng (BFS).
- Tìm kiếm theo chiều sâu (DFS)
- Tìm kiếm đồng nhất (UCS).

Uniform Cost Search

Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^m)$	$O(bm)$
BFS		Y	N	$O(b^s)$	$O(b^s)$
UCS		Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$

GIẢI THUẬT DFS

```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
  node <- α node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  if problem INSTANCEOF MultiFoodSearchProblem AND LEN(problem.all_dot) == 0 then return node['STATE']

  frontier <- a LIFO stack with node as the only element
  explored <- an empty set
  loop do
    if EMPTY ? (frontier) then return failure
    node <- POP(frontier) /*chooses the deepest 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 or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier <- PUSH(child, frontier)
```

GIẢI THUẬT BFS

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node <- α node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  if problem INSTANCEOF MultiFoodSearchProblem AND LEN(problem.all_dot) == 0 then return node['STATE']

  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 or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier <- INSERT(child, frontier)
```

GIẢI THUẬT UCS

```
function UNIFORM-COST-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)
  if problem INSTANCEOF MultiFoodSearchProblem AND LEN(problem.all_dot) == 0 then return node['STATE']
  frontier <- a priority queue ordered by PATH-COST, with node as the element
  explored <- an empty set
  loop do
    if EMPTY ? (frontier) then return failure
    node <- POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    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 or frontier then
        frontier <- INSERT(child,frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Class & Method

Các lớp được sử dụng:

Class	Property	Method
SingleFoodSearchProblem	state, node, initialState, successorFn, goalTestFn, pathCostFn	readMaze(), printMaze()
MultiFoodSearchProblem	state, node, initialState, successorFn, goalTestFn, pathCostFn	readMaze(), printMaze(), animate()

Các lớp được sử dụng:

- bfs(problem)
- dfs(problem)
- ucs(problem)

DEMO

Link chạy demo:

https://qiao.github.io/PathFinding.js/visual/

Best-First Search

TỔNG QUAN CÂU 2

```
├── problems.py
│   ├── SingleFoodSearchProblem
│   │   ├── State
│   │   ├── Node
│   │   ├── Initial state
│   │   ├── Successor function
│   │   ├── Goal-test function
│   │   ├── Path-cost function
│   │   ├── read maze function
│   │   └── print maze function
│   └── MultiFoodSearchProblem
│       ├── State
│       ├── Node
│       ├── Initial state
│       ├── Successor function
│       ├── Goal-test function
│       ├── Path-cost function
│       ├── read maze function
│       ├── print maze function
│       └── animate function
├── fringes.py
│   ├── Stack
│   ├── Queue
│   └── PriorityQueue
└── searchAgents.py
    ├── bfs(problem)
    ├── dfs(problem)
    ├── ucs(problem)
    ├── astar(problem, fn_heuristic)
    ├── gbfs(problem, fn_heuristic)
    ├── SingleFoodSearchProblem.animate(actions)
    ├── MultiFoodSearchProblem.get_successors(state)
    ├── MultiFoodSearchProblem.get_cost_of_actions(actions)
    ├── MultiFoodSearchProblem.is_goal_state(state)
    ├── MultiFoodSearchProblem.get_initial_state()
    ├── MultiFoodSearchProblem.heuristic(state)
    └── MultiFoodSearchProblem.animate(actions)
```

GIỚI THIỆU

Những sẽ có trong bài toán này:

- Best-First-Search
- SingleFoodSearchProblem
- MultiFoodSearchProblem
- Hàm heuristic

HEURISTIC & SINGLEFOODSEARCHPROBLEM

Hàm heuristic và vai trò của nó trong bài toán SingleFoodSearchProblem:

- **manhattan_distances():** Ước lượng khoảng cách Manhattan (tổng khoảng cách theo chiều ngang và chiều dọc) từ vị trí hiện tại của Pacman đến vị trí thức ăn gần nhất trên bản đồ.
- **euclidean_distances():** Ước lượng khoảng cách Euclidean (khoảng cách thẳng đo từ điểm hiện tại đến điểm đích) từ vị trí hiện tại của Pacman đến vị trí thức ăn gần nhất trên bản đồ.

```python
def manhattan_distances(state: tuple, problem: SingleFoodSearchProblem) -> float:
    manhattan = lambda c_x, c_y, d_x, d_y : abs(c_x - d_x) + abs(c_y - d_y)
    cur_x, cur_y = state
    all_dots = problem.all_dots

    dot_x, dot_y = all_dots[0]

    return manhattan(cur_x, cur_y, dot_x, dot_y)

def euclidean_distances(state: tuple, problem: SingleFoodSearchProblem) -> float:
    euclidean = lambda c_x, c_y, d_x, d_y : ((c_x - d_x)**2 + (c_y - d_y)**2)**0.5
    cur_x, cur_y = state
    all_dots = problem.all_dots

    dot_x, dot_y = all_dots[0]

    return euclidean(cur_x, cur_y, dot_x, dot_y)
```

HEURISTIC & MULTIFOODSEARCHPROBLEM

Hàm heuristic và vai trò của nó trong bài toán MultiFoodSearchProblem:

- shortest_manhattan_distances_multi(): Hàm này sẽ tính toán khoảng cách Manhattan ngắn nhất từ trạng thái hiện tại đến điểm ăn gần nhất trên bản đồ. Nếu không có điểm ăn nào còn lại, hàm trả về 0.

```
def shortest_manhattan_distances_multi(state: tuple, problem : MultiFoodSearchProblem):
    manhattan = lambda c_x, c_y, d_x, d_y : abs(c_x - d_x) + abs(c_y - d_y)
    cur_x, cur_y = state
    all_dots = problem.all_dots

    if len(all_dots) == 0:
        return 0
    elif len(all_dots) == 1:
        dot_x, dot_y = all_dots[0]
        return manhattan(cur_x, cur_y, dot_x, dot_y)
    else:
        dot_x, dot_y = all_dots[0]
        res = manhattan(cur_x, cur_y, dot_x, dot_y)
        for dot in all_dots:
            temp = manhattan(cur_x, cur_y, dot[0], dot[1])
            if res > temp:
                res = temp
        return res
```

A_STAR WITH SINGLE, MULTI FOODSEARCHPROBLEM

Hàm heuristic và vai trò của nó trong cả 2 bài toán Single, Multi FoodSearchProblem:

astar(problem : FoodSearchProblem, fn_heuristic): có chức năng tìm kiếm đường đi tối ưu từ trạng thái hiện tại đến trạng thái đích dựa trên hàm heuristic và giải thuật A*. Hàm trả về đường đi tối ưu nếu tồn tại hoặc None nếu không tìm được đường đi.

```python
def astar(problem : FoodSearchProblem, fn_heuristic) -> list:
    node = problem.node

    if (problem.goal_test(node['state'])):
        return ['Stop']

    if (isinstance(problem, MultiFoodSearchProblem) and len(problem.all_dots) == 0):
        return ['Stop']

    q = PriorityQueue()
    q.enqueue((fn_heuristic(node['state'], problem), node))
    explored = []
    visited = [node['state']]

    while not q.empty():
        weight, node = q.dequeue()
        explored.append(node['state'])

        all_actions = problem.getSuccessors(node['state'])

        for action in all_actions:
            state_x, state_y = node['state']
            child = {'state': problem.all_actions[action](state_x, state_y), 'parent': node, 'action': action}
            if not ((child['state'] in explored) or (child['state'] in visited)):
                visited.append(child['state'])

                if isinstance(problem, MultiFoodSearchProblem):
                    if problem.goal_test(child['state']):
                        node = make_node(child['state'], None, None)
                        # Remove a dot we that Pacman ate
                        problem.all_dots.remove(child['state'])
                        return solution(child)[:-1] + astar(MultiFoodSearchProblem(child['state'], node, problem.initial_state, problem.all_dots), fn_heuristic)
                    else:
                        q.enqueue((fn_heuristic(child['state'], problem) + weight - fn_heuristic(node['state'], problem) + 1, child))
                elif isinstance(problem, SingleFoodSearchProblem):
                    if problem.goal_test(child['state']):
                        return solution(child)
                    else:
                        q.enqueue((fn_heuristic(child['state'], problem) + weight - fn_heuristic(node['state'], problem) + 1, child))

    return None
```

GBFS

Hàm heuristic và vai trò của nó trong cả 2 bài toán Single, Multi FoodSearchProblem:

gbfs(problem : FoodSearchProblem, fn_heuristic): hàm gbfs này là một thuật toán tìm kiếm theo chiều rộng đầu tiên. Nó hoạt động bằng cách lặp lại việc duyệt qua các trạng thái con của các trạng thái hiện tại, tạo ra các nút mới cho đến khi đạt được trạng thái đích. Trong quá trình duyệt, gbfs sử dụng một hàng đợi ưu tiên để theo dõi trạng thái mới được thêm vào và sắp xếp chúng theo ước tính độ lớn của hàm heuristic.

```
def gbfs(problem : FoodSearchProblem, fn_heuristic) -> list:
    node = problem.node

    if (problem.goal_test(node['state'])):
        return ['Stop']

    if (isinstance(problem, MultiFoodSearchProblem) and len(problem.all_dots) == 0):
        return ['Stop']

    q = PriorityQueue()
    q.enqueue((fn_heuristic(node['state'], problem), node))
    explored = []
    visited = [node['state']]

    while not q.empty():
        weight, node = q.dequeue()
        explored.append(node['state'])

        all_actions = problem.getSuccessors(node['state'])

        for action in all_actions:
            state_x, state_y = node['state']
            child = {'state': problem.all_actions[action](state_x, state_y), 'parent': node, 'action': action}
            if not ((child['state'] in explored) or (child['state'] in visited)):
                visited.append(child['state'])

                if isinstance(problem, MultiFoodSearchProblem):
                    if problem.goal_test(child['state']):
                        node = make_node(child['state'], None, None)
                        # Remove a dot we that Pacman ate
                        problem.all_dots.remove(child['state'])
                        return solution(child)[:-1] + astar(MultiFoodSearchProblem(child['state'], node, problem.initial_state, problem.all_dots), fn_heuristic)
                    else:
                        q.enqueue((fn_heuristic(child['state'], problem), child))
                elif isinstance(problem, SingleFoodSearchProblem):
                    if problem.goal_test(child['state']):
                        return solution(child)
                    else:
                        q.enqueue((fn_heuristic(child['state'], problem), child))

    return None
```

LOCAL SEARCH

TỔNG QUAN CÂU 3

```python
class EightQueenProblem():

    def __init__(self, state = [], n = 8):···

    def __str__(self) -> str:···

    # YC3-1 print
    def print(self) -> None:···

    # YC3-1 load from file
    def load_from_file(self, filename: str) -> None:···

    def successor_for_col(self, col: int) -> list:···

    def best_successor(self):···
    # YC3-2
    def hill_climbing_search(self):···

def attack(queen: tuple, enemy: tuple) -> bool:···

# YC3-1 h function
def h(state: EightQueenProblem) -> int:···
```

GIẢI THUẬT

Khởi tạo trạng thái cũng như in ra chuỗi đại diện cho bàn cờ:

- Hàm **init(self, state = [], n = 8):** Hàm khởi tạo của class EightQueenProblem, mặc định state là một list rỗng và n = 8. state chứa các vị trí của các quân hậu trên bàn cờ.
- Hàm **str(self) -> str**: Hàm trả về một chuỗi đại diện cho bàn cờ với các quân hậu đã được đặt ở các vị trí tương ứng.
- Hàm **print(self) -> None**: Hàm này in ra chuỗi đại diện cho bàn cờ.

```python
def __init__(self, state = [], n = 8):
    self.state = state
    self.state.sort(key = lambda x: x[1])
    self.number_of_queen = n

def __str__(self) -> str:
    temp = [['0']*self.number_of_queen for j in range(self.number_of_queen)]

    for x_q, y_q in self.state:
        temp[x_q][y_q] = 'Q'

    res = [' '.join(line) for line in temp]

    return '\n'.join(res)

def print(self) -> None:
    print(str(self))
```

LOAD FILE

Nếu file tồn tại, hàm sẽ đọc nội dung file đó và lưu vào biến self.state dưới dạng một **danh sách các list**.

Tiếp theo, hàm tìm tất cả các quân hậu trong trò chơi 8 quân hậu dựa trên nội dung đã đọc được từ file và lưu chúng vào danh sách **all_queens.**

Sau đó, **hàm cập nhật lại self.state** thành danh sách tọa độ của tất cả các quân hậu và sắp xếp chúng theo thứ tự tăng dần của cột.

```
def load_from_file(self, filename: str) -> None:
    if os.path.exists(filename):
        with open(filename) as f:
            self.state = [line.rstrip().split() for line in f]
    all_queens = []
    self.number_of_queen = len(self.state)
    for i in range(self.number_of_queen):
        for j in range(self.number_of_queen):
            if self.state[i][j] == 'Q':
                all_queens.append((i,j))
    self.state = all_queens
    self.state.sort(key = lambda x: x[1])
```

HÀM TRẢ VỀ TẤT CẢ TRẠNG THÁI

Hàm successor_for_col trả về tất cả các trạng thái con của trạng thái hiện tại mà có thể tạo ra bằng cách di chuyển quân hậu ở cột được chỉ định đến một hàng mới.

- Lặp qua tất cả các cặp tọa độ (x_q, y_q) của các quân hậu trong trạng thái hiện tại. Nếu y_q bằng với cột được chỉ định, gán selected_q thành quân hậu có tọa độ (x_q, y_q). Nếu không, thêm quân hậu có tọa độ (x_q, y_q) vào danh sách all_qs_except_selected.
- Lặp qua tất cả các hàng có thể chứa quân hậu (hàng từ 0 đến 7), thêm một bộ giá trị mới vào danh sách res. Bộ giá trị này bao gồm tất cả các quân hậu trong danh sách all_qs_except_selected và thêm một quân hậu mới ở cột được chỉ định vào hàng hiện tại.

```python
def successor_for_col(self, col: int) -> list:
    res = []
    all_qs_except_selected = []
    selected_q = None
    for x_q, y_q in self.state:
        if (y_q == col):
            selected_q = (x_q, y_q)
        else:
            all_qs_except_selected.append((x_q, y_q))

    for i in range(8):
        res.append(all_qs_except_selected + [(i, selected_q[1])])

    return res
```

HÀM TRẢ VỀ TẤT CẢ TRẠNG THÁI

Hàm best_successor tìm trạng thái con tốt nhất của trạng thái hiện tại.

- Hàm tìm tất cả các trạng thái kế tiếp có thể được tạo ra bằng cách di chuyển quân hậu và tính giá trị hàm heuristic của từng trạng thái kế tiếp. Giá trị heuristic được sử dụng để ước lượng độ tốt của một trạng thái, bằng cách tính toán số lượng xung đột giữa các quân hậu trong trạng thái đó.
- Sau khi tính toán giá trị heuristic của tất cả các trạng thái kế tiếp, hàm chọn trạng thái kế tiếp tốt nhất bằng cách chọn trạng thái có giá trị heuristic nhỏ nhất (tức là ít xung đột nhất giữa các quân hậu). Nếu có nhiều trạng thái có cùng giá trị heuristic nhỏ nhất, hàm chọn một trạng thái bất kỳ trong số đó.
- Cuối cùng, hàm cập nhật trạng thái hiện tại của bàn cờ thành trạng thái kế tiếp tốt nhất và sắp xếp lại các quân hậu trong trạng thái mới theo thứ tự cột tăng dần.

```
def best_successor(self):
    for i in range(self.number_of_queen):
        min_successor = self.number_of_queen ** 2
        min_successor_state = []
        successor_i = self.successor_for_col(i)
        for successor in successor_i:
            heuristic = h(EightQueenProblem(successor))
            if ( heuristic <= min_successor):
                min_successor = heuristic
                min_successor_state = successor
        self.state = min_successor_state
        self.state.sort(key = lambda x: x[1])
```

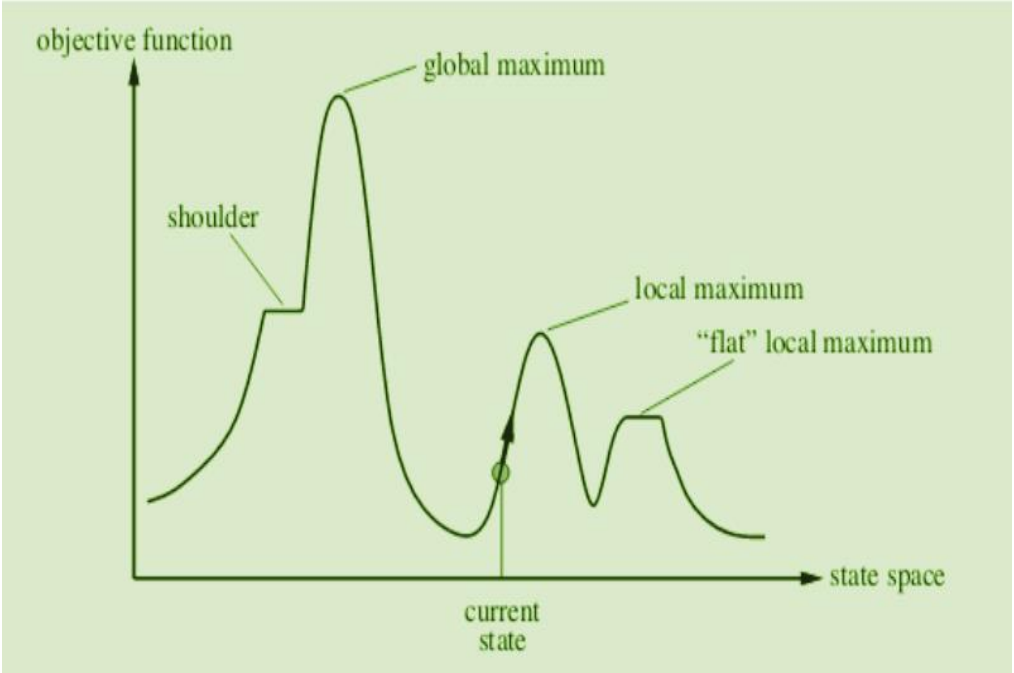
GIẢI THUẬT

- Hàm attack kiểm tra xem có tấn công giữa 2 quân hậu hay không. Nó trả về True nếu các quân hậu tấn công lẫn nhau và False nếu không.
- Hàm h tính toán giá trị hàm heuristic cho trạng thái hiện tại trong EightQueenProblem. Nó duyệt qua tất cả các cặp quân hậu và tăng giá trị của biến res lên 1 nếu chúng tấn công lẫn nhau. Tuy nhiên, mỗi cặp quân hậu sẽ được tính toán 2 lần nên kết quả được trả về sẽ chia đôi.

```python
def attack(queen: tuple, enemy: tuple) -> bool:
    x_q, y_q = queen
    x_e, y_e = enemy

    return False if queen == enemy else abs(x_q - x_e) == abs(y_q - y_e) or (x_e == x_q) or (y_e == y_q)

# YC3-1 h function
def h(state: EightQueenProblem) -> int:
    all_queens = state.state
    res = 0
    for queen in all_queens:
        res += [attack(queen, enemy) for enemy in all_queens].count(True)

    return res // 2
```

Diagram for Hill Climbing

- Local Maximum: một trạng thái tốt hơn các trạng thái hàng xóm của nó, nhưng cũng có một trạng thái khác cao hơn nó.
- Global Maximum: trạng thái tốt nhất có thể có trên địa hình không gian trạng thái. Nó có giá trị hàm mục tiêu cao nhất.
- Current State: Đó là trạng thái trong biểu đồ địa hình mà một tác nhân đang hiện diện.
- Flat local maximum: một không gian bằng phẳng trong địa hình, trong đó tất cả các trạng thái hàng xóm của trạng thái hiện tại đều có cùng một giá trị.
- Shoulder: Nó là một khu vực bằng phẳng có một mép dốc.



GIẢI THUẬT

function HILL-CLIMBING(*problem*) **returns** a solution state
  inputs: *problem*, a problem
  static: *current*, a node
    next, a node

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  loop do
    next ← a highest-valued successor of *current*
    if VALUE[next] < VALUE[current] **then return** *current*
    current ← *next*
  end

```
def hill_climbing_search(self):
        p_state = self.state
        while True:
            self.best_successor()
            if (p_state == self.state):
                break
            else:
                p_state = self.state
```

THUẬN LỢI VÀ KHÓ KHĂN

Thuận lợi	Khó khăn
Do có được một số mã giả mà giảng viên cung cấp nên việc giải quyết vấn đề dễ dàng hơn.	Khó khăn trong việc tìm lời giải tối ưu: Tìm ra lời giải tối ưu cho các thuật toán trên có thể đòi hỏi bạn phải thử nhiều phương pháp khác nhau và sử dụng nhiều kỹ thuật khác nhau để tối ưu hóa giải thuật của nhóm.
Có được một số nguồn tài liệu liên quan đến testcase giúp dễ dàng hơn trong việc đoán kết quả của các thuật toán.	Có rất nhiều ý kiến khác nhau về đường đi của pacman trong giải thuật gbfs về multiFoodSearch.
Do có thể chạy tay các thuật toán cũng như in ra từng kết quả trong lúc làm nên hạn chế được nhiều sai sót.	Khó khăn trong việc đánh giá hàm chi phí: Việc tìm kiếm giải pháp tốt trong bài toán 8 con hậu đòi hỏi phải đánh giá hàm chi phí của từng trạng thái. Đánh giá hàm chi phí này có thể rất phức tạp và

TÀI LIỆU THAM KHẢO

1. Russell, S., & Norvig, P. (2010). Artificial intelligence: A modern approach (3rd ed.). Prentice Hall Press.
2. Neller, T. (2010). Pac-Man and the Ghostly Adventures of DFS, BFS, and UCS. Computer Science Education, 20(1), 34-47. doi: 10.1080/08993408.2010.528028.
3. Zaremba, K. (2018). An Exploration of Pacman Agent Implementation. In Proceedings of the 2018 ACM SIGCSE Technical Symposium on Computer Science Education (pp. 769-774). doi: 10.1145/3159450.3159527.
4. Korf, R. E. (1999). Artificial intelligence search algorithms. Springer.
5. A. M. Mafoudi and N. Benouhiba, "A modified genetic algorithm for the eight queens problem," 2014 IEEE International Conference on Electronics, Energy and Measurement (ICEEM), Islamabad, 2014.
6. S. Maheshwari and A. Agarwal, "Solving the eight queens problem using genetic algorithm," 2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT), Coimbatore, 2015, pp. 1-5.

BẢNG PHÂN CÔNG NHIỆM VỤ

Họ và tên	Nhiệm vụ	Mức độ hoàn thành
Nguyễn Quốc Anh	• Hoàn thành các file code cho câu 2 • Hoàn thành code cho ý 3.1 • Soạn PowerPoint cho 2 ý trên	100%
Nguyễn Vũ Tường	• Hoàn thành các file code cho câu 1 • Hoàn thành code cho ý 3.2 • Soạn PowerPoint cho 2 ý trên	100%
Võ Phú Vinh	• Hoàn thành các file code cho câu 1 • Hoàn thành code cho ý 3.2 • Soạn PowerPoint cho 2 ý trên	100%
Trần Thị Vẹn	• Hoàn thành các file code cho câu 2 • Hoàn thành code cho ý 3.1 • Soạn PowerPoint cho 2 ý trên	100%

NHÓM 4

CẢM ƠN MỌI NGƯỜI ĐÃ LẮNG NGHE BÀI THUYẾT TRÌNH!