TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

fit@hcmus

Đồ án 1
ARES's ADVENTURE

Nhóm sinh viên thực hiện:
Lê Gia Bảo **MSSV**:23127325
Vũ Anh **MSSV**:23127321
Hồ Gia Huy **MSSV**:23127376
Nguyễn Phan Thế Vinh
MSSV:23127520

Môn: Cơ Sở Trí Tuệ Nhân Tạo
Năm học: 2024-2025

TP.HCM, tháng 3 năm 2025

Mục Lục

I) Phân công công việc

Họ và tên	Công việc
Nguyễn Phan Thế Vinh	GBFS(100%), GUI(30%), Input(100%)
Lê Gia Bảo	A*(100%), DFS(100%)
Hồ Gia Huy	GUI(70%), output file(100%)
Vũ Anh	BFS(100%), UCS(100%), Dijkstra(100%), report(100%)

II) Self-evaluation

1. Mức độ hoàn thành:
- Các thuật toán hoạt động đúng với yêu cầu đồ án đưa ra.
- Có đầy đủ các test case của các trường hợp có thể xảy ra.
- Hệ thống GUI được tối ưu với các hoạt động di chuyển, âm thanh trò chơi, trọng lượng đánh dấu mỗi hộp…
- Link hoàn thành đồ án: https://github.com/vanh1012/Ares-s-Adventure.git
- Video hướng dẫn đồ án: https://youtu.be/ExsUIHFz9fo

2. Hạn chế:
- Yêu cầu đề bài là một bài toán NP-Hard, nên việc tìm giải pháp để làm đúng yêu cầu không đơn giản.
- Cần nhiều thời gian để tối ưu thời gian chạy của thuật toán
- Thiết kế heuristic hiệu quả không đơn giản.

3. Cải thiện:
- Để được tối ưu thời gian đã sử dụng deadlock để loại những đường đi không tìm thấy
- Hàm heuristic đã sử dụng 2 thuật toán là Mahaatan và Hungarian (có thể các map có độ phức tạp cao thì heuristic có thể không hoàn toàn admissible)

III) Search Algorithms

Các thông số với các thuật toán:

- *Grid: danh sách chuỗi, mỗi chuỗi là một hàng của map*
- *Player_pos: vị trí của Ares*
- *Box_position(boxes_set): list các vị trí ban đầu của hộp*
- *Goals: set các vị trí của đích*
- *Weight_of_box: list các trọng số của hộp (theo cùng thứ tự trong Box_positions) hoặc dict (box_positions : weight of Box) tùy các thuật toán được triển khai*

1.DFS:

- Duyệt sâu đến khi gặp trạng thái đích hoặc gặp deadlock/hết đường đi rồi mới lùi lại (Sử dụng đệ quy xử lý)
- Lưu dấu đã duyệt bằng cách dùng cặp: (*player_pos, tuple_sorted(boxes))* để đảm bảo có thứ tự nhất quán trên *boxes*.
- Nếu *boxes == goals,* trả về *road* (đường đi tìm được)
- Lần lượt thử 4 hướng (U, D, L, R): sinh ra trạng thái mới, nếu chưa duyệt thì đánh dấu và gọi đệ quy
- Nếu tất cả nhánh đều thất bại, trả "NoSol"

```python
def dfs(self, positionPlayer, road):
        self.countNode += 1
        if self.boxes == self.goals:
            return road
        if self.checkDeadlock(self.boxes):
            return
        for move, (dx, dy) in MOVE.items():
            newPositionPlayer = [positionPlayer[0] + dx, positionPlayer[1] + dy]
            if (newPositionPlayer[0], newPositionPlayer[1]) not in self.boxes:
                new_State = (tuple(newPositionPlayer), tuple(sorted(self.boxes)))
                if (
                    self.isNextValid(newPositionPlayer)
                    and new_State not in self.visited
                ):
                    self.visited.add(new_State)
                    result = self.dfs(newPositionPlayer, road + [[move.lower(), 1]])
                    if result:
                        return result
            else:
                newPositionBox = [newPositionPlayer[0] + dx, newPositionPlayer[1] + dy]
                if self.isNextValid(newPositionPlayer, newPositionBox):
                    self.boxes.remove((newPositionPlayer[0], newPositionPlayer[1]))
                    self.boxes.add((newPositionBox[0], newPositionBox[1]))
                    new_State = (tuple(newPositionPlayer), tuple(sorted(self.boxes)))
                    if new_State not in self.visited:
                        self.visited.add(new_State)
                        old_box_pos = tuple(newPositionPlayer)
                        cost = self.box_weights[old_box_pos]
                        self.box_weights.pop(old_box_pos)
                        self.box_weights[tuple(newPositionBox)] = cost
                        result = self.dfs(newPositionPlayer, road + [[move, cost]])
                        if result:
                            return result
                        self.box_weights.pop(tuple(newPositionBox))
                        self.box_weights[old_box_pos] = cost
                    self.boxes.remove((newPositionBox[0], newPositionBox[1]))
                    self.boxes.add((newPositionPlayer[0], newPositionPlayer[1]))
        return "NoSol"
```

2.BFS:

- Xác định các thông số
- Xác định hàm *move_state* để từ 1 trạng thái sinh ra tối đa 4 trạng thái con, có kiểm tra deadlock(tính hợp lệ).
- Sử dụng queue để duyệt rộng.
- Lưu các node đã duyệt *visited* để tránh lặp lại.
- Khi gặp đích, dừng và tái dựng đường đi từ Node parents, không thì trả về "NoSol".

3.A*

- Hàm *MinimumCostFromBoxToGoal:*

+ heuristic h(n) = $\sum_{i=1}^{n} \left(\text{trọng số hộp}_i \times \text{ManhattanDist}\left(\text{hộp}_i, \text{goal}_j\right)\right)$

+ Hàm sử dụng thêm Hungarian Algorithm để tìm ánh xạ tối ưu hộp-goal

```python
def MinimumCostFromBoxToGoal(self, box_weight):
        cost = []
        for box, w in box_weight.items():
            row_cost = []
            for goal in self.goals:
                row_cost.append(w * self.manhattan_distance(box, goal))
            cost.append(row_cost)

        np_cost = np.array(cost)
        row_ind, col_ind = linear_sum_assignment(np_cost)
        return np_cost[row_ind, col_ind].sum()
```

- g(n) = tổng cost đã bỏ ra (di chuyển thường + 1, đẩy + trọng số hộp)
- Mỗi lần pop trạng thái có f = g + h nhỏ nhất khỏi hàng đợi ưu tiên
- Dùng deadlock loại bỏ đường vô ích

4. GBFS

- Heuristic h(n) = tổng Manhattan distance từ hộp tới goal gần nhất

```python
def calculate_heuristic(self, boxes):
        if not boxes or not self.goals:
            return float('inf')

        total = 0
        remaining_goals = list(self.goals)

        for box in boxes:
            if box in self.goals:
                if box in remaining_goals:
                    remaining_goals.remove(box)
                total += 0
            else:
                if remaining_goals:
                    closest_dist = min(self.manhattan_distance(box, goal) for goal in remaining_goals)

                    closest_goal = min(remaining_goals, key=lambda g: self.manhattan_distance(box, g))

                    total += closest_dist
                    if closest_goal in remaining_goals:
                        remaining_goals.remove(closest_goal)
                else:
                    total += 99

        return total
```

- Mỗi trạng thái gồm: *(heuristic, steps, player_pos, boxes_frozen, path)*
- Sử dụng hàng đợi ưu tiên sắp xếp heuristic tăng dần. Mỗi lần ,GBFS lấy trạng thái có giá trị heuristic nhỏ nhất để mở rộng trước.
- Kiểm tra deadlock loại bỏ đường vô ích.
- Khi mở rộng tính lại heuristic cho cấu hình hộp mới rồi mới thêm vào hàng đợi.
- Nếu tất cả hộp nằm trên đích thì lấy *path* xây dựng lời giải, nếu duyệt xong mà không tìm ra lời giải trả về "NoSol"

5. UCS:

- Thuật toán này có hàm *can_player_reach(start, target, boxes_set)*: để kiểm tra

người chơi có thể đi từ *start* đến *target* mà không đi xuyên tường hay xuyên hộp trong *boxes_set* để triển khai đã sử dụng thuật toán BFS.

- Sử dụng hàng đợi ưu tiên (Priority Queue) để lưu các trạng thái (Trạng thái bắt đầu có cost = 0, path rỗng và weight = 0).
- Pop ra khỏi hàng đợi ưu tiên trạng thái có cost nhỏ nhất, kiểm tra goal (Nếu tất cả hộp trong goals, ta trả về đường đi), kiểm tra deadlock (nếu có bỏ qua).
- Mỗi trạng thái mới tính là *new_cost*. Nếu *new_cost* < *best_cost.get(new_state, math.inf)* thì cập nhật trạng thái và đưa vào hàng đợi ưu tiên.
- Nếu hàng đợi cạn mà chưa về đích thì trả về "NoSol".

```python
def ucs(self):
        start_state = (self.start_player, self.start_boxes, self.start_box_weights)
        pq = [(0, start_state, "", 0)]
        best_cost = {start_state: 0}
        node_generated = 0

        while pq:
            cost, (player, boxes_tuple, box_weights_fro), path, total_weight = heapq.heappop(pq)
            node_generated += 1

            if cost > best_cost[(player, boxes_tuple, box_weights_fro)]:
                continue

            boxes_set = set(boxes_tuple)

            if self.is_deadlock(boxes_set):
                continue

            if boxes_set == self.goals:
                return node_generated, total_weight, path

            box_weights = dict(box_weights_fro)
            px, py = player

            for mv, (dx, dy) in MOVE.items():
                nx, ny = px + dx, py + dy
                if (nx, ny) not in boxes_set and not self.is_wall(nx, ny):
                    if self.can_player_reach((px, py), (nx, ny), boxes_set):
                        new_cost = cost + 1
                        new_state = ((nx, ny), boxes_tuple, box_weights_fro)
                        if new_cost < best_cost.get(new_state, math.inf):
                            best_cost[new_state] = new_cost
                            new_path = path + mv.lower()
                            heapq.heappush(pq, (new_cost, new_state, new_path, total_weight))
                    elif (nx, ny) in boxes_set:
                        bx, by = nx + dx, ny + dy
                        if (bx, by) not in boxes_set and not self.is_wall(bx, by):
                            if self.can_player_reach((px, py), (nx, ny), boxes_set - {(nx, ny)}):
                                w = box_weights.get((nx, ny), 0)
                                new_cost = cost + w
                                new_boxes = set(boxes_set)
                                new_boxes.remove((nx, ny))
                                new_boxes.add((bx, by))
                                new_boxes_tuple = tuple(sorted(new_boxes))
                                new_dict = dict(box_weights)
                                if (nx, ny) in new_dict:
                                    tmp = new_dict.pop((nx, ny))
                                    new_dict[(bx, by)] = tmp
                                nw_fro = frozenset(new_dict.items())
                                new_state = ((nx, ny), new_boxes_tuple, nw_fro)
                                new_total_weight = total_weight + w
                                if new_cost < best_cost.get(new_state, math.inf):
                                    best_cost[new_state] = new_cost
                                    new_path = path + mv.upper()
                                    heapq.heappush(pq, (new_cost, new_state, new_path, new_total_weight))
        return node_generated, 0, "NoSol"
```

6. Dijkstra (tương tự như UCS)

IV) Output file and GUI

1. GUI

- GUI có thể cho phép người dùng có thể chơi bằng cách điều khiển Ares theo các mũi tên.

Giao diện khi vào trong trò chơi

- Có tích hợp âm thanh giúp người dùng cảm thấy giải trí.
- Để sử dụng các thuật toán giải, người dùng ấn các số từ 1 – 6 với tương ứng các thuật toán A*, BFS, DFS, Dijkstra, GBFS, UCS.
- Cách sử dụng giao diện như restart màn chơi, chọn level,… có hướng dẫn.

2. Output file

- Khi ấn số trong trò chơi, thuật toán tương ứng sẽ giải và sẽ in ra file output trong folder Outputs với tên output-level.txt
- Khi đã có file output nếu sử dụng các thuật toán giải lại level đó sẽ ghi thêm lời giải cho file chứ không ghi đè
- Cấu trúc file output:

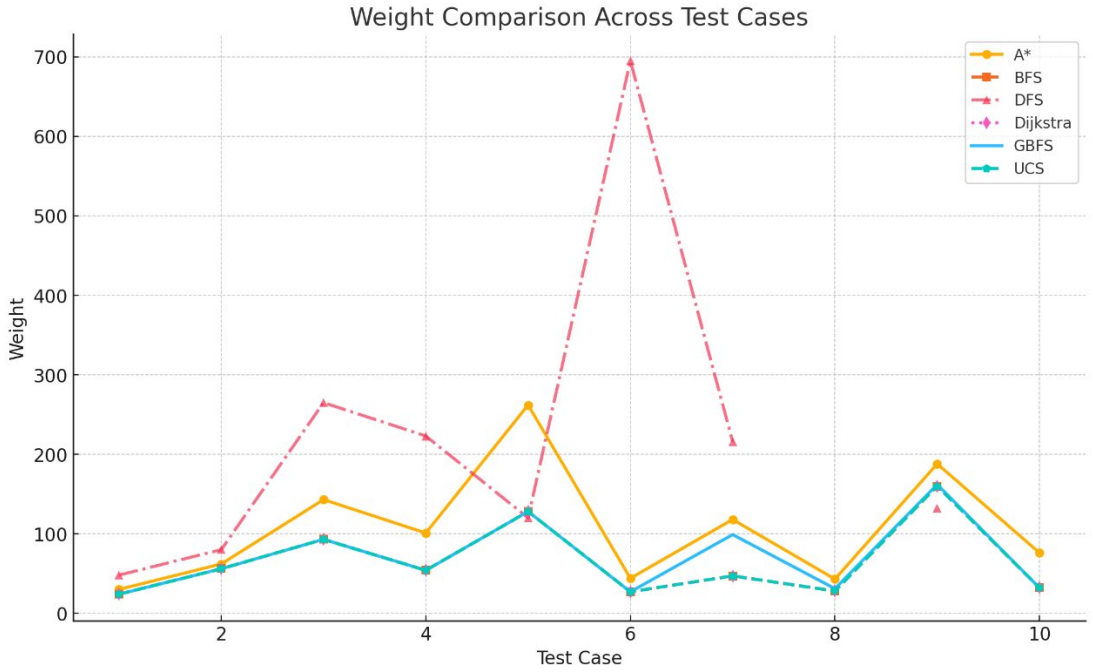
```
Algorithm: ASTAR
Steps: 10
Weight: 30
Node: 67
Time (ms) : 7.030
Memory (MB): 0.06
Solution: dlUruLLulD
```

Với:

+ Steps: tổng số lần đi trong lời giải

+ Weight: tổng các trọng số của thùng khi được đẩy (mỗi lần đẩy sẽ được cộng trọng số thùng được đẩy tương ứng)
+ Node: số lần thuật toán duyệt
+ Time: Thời gian giải của thuật toán
+ Memory: Dung lượng mà thuật toán sử dụng
+ Solution: Lời giải (ghi hoa là đẩy thùng, ghi thường là di chuyển)

V)Biểu đồ của các thuật toán



Biểu đồ tính trọng số mà ares đã đẩy trong các thuật toán

Nhận xét:
- DFS có trọng số rất cao và khi có độ phức tạp cao thì không thể giải -> thuật toán tệ trong việc tối ưu đường đi
- BFS, Dijkstra và UCS khá tương đồng nhau vì BFS tìm đường đi ngắn nhất mà input file các trọng số của hộp chênh lệch nhau ít dẫn đến việc khá tương đồng
- Dijkstra và UCS có bản chất giống nhau nên cho kết quả như nhau
- A* khá cao vì heuristic có tính thêm Hungarian Algorithm dẫn đến việc không admissible
- GBFS có heuristic tương đồng với A* nên chưa tìm được đường đi tối ưu hơn Dijkstra và UCS

VI) References

Giải trò chơi Sokoban - Nhập môn trí tuệ nhân tạo

AI Solver: Weighted Sokoban Puzzle (A*) — Asier Goni

Transposition Tables & Zobrist Keys - Advanced Java Chess Engine Tutorial

[30]

[AI in Game Playing: Sokoban Solver]

[Simple implementation of Sokoban in Python]