ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA KHOA HỌC MÁY TÍNH

TRÍ TUỆ NHÂN TẠO (CS106.P21)

BÁO CÁO
CÀI ĐẶT THUẬT TOÁN DFS, BFS VÀ UCS CHO TRÒ CHƠI SOKOBAN

GIẢNG VIÊN HƯỚNG DẪN: TS. LƯƠNG NGỌC HOÀNG

STT	Họ tên SV	MSSV
1	Trần Minh Tiến	23521587

Nội dung

1 Mô hình hóa trò chơi Sokoban

1.1 Trạng thái của bài toán

Trạng thái trong Sokoban được định nghĩa bởi hai thành phần chính:

- Vị trí của người chơi: Một tọa độ (x,y) xác định vị trí hiện tại của người chơi trên bản đồ.
- Vị trí của các thùng: Một tập hợp các tọa độ biểu diễn vị trí của tất cả các thùng trên bản đồ.

1.2 Trạng thái khởi đầu và trạng thái kết thúc

- Trạng thái khởi đầu: Là vị trí ban đầu của người chơi và các thùng trên bản đồ.
- Trạng thái kết thúc: Khi tất cả các thùng đều được đặt đúng vào các vị trí đích được xác định trước trên bản đồ.

1.3 Không gian trạng thái

Mỗi lần người chơi thực hiện một hành động hợp lệ (di chuyển hoặc đẩy thùng), trạng thái của trò chơi sẽ thay đổi, tạo ra một trạng thái mới. Toàn bộ các trạng thái này kết hợp lại tạo thành không gian trạng thái, và thuật toán tìm kiếm sẽ duyệt qua không gian này để tìm ra lời giải.

1.4 Các hành động hợp lệ

Mỗi trạng thái có thể chuyển đổi sang một trạng thái khác thông qua một tập hợp các hành động hợp lệ:

- Người chơi có thể di chuyển lên, xuống, trái, phải.

- Nếu người chơi đứng cạnh một thùng hàng và di chuyển theo hướng đẩy nó, thùng sẽ di chuyển theo cùng hướng đó.
- Hành động hợp lệ: chỉ được đẩy một thùng mỗi lượt. Và sau khi thực hiện, thùng hàng không vượt ra ngoài bản đồ và không bị kẹt (giữa các bức tường hoặc giữa tường và thùng khác).

1.5 Hàm tiến triển (successor function)

Hàm tiến triển updateState sẽ trả về tập hợp các trạng thái mới nếu hành động của người chơi là một hành động hợp lệ.

2 Áp dụng các thuật toán tìm kiếm DFS, BFS, UCS cho Sokoban

2.1 Thuật toán DFS

- Đầu tiên, ta lấy vị trí của các thùng và người chơi lúc bắt đầu bằng hàm PosOfBoxes(gameState) và PosOfPlayer(gameState). Trạng thái bắt đầu của trò chơi được biểu diễn dưới dạng cặp (beginPlayer, beginBox), trong đó beginPlayer là vị trí của người chơi và beginBox là tập hợp vị trí các thùng trên bản đồ.
- Khởi tạo một frontier dưới dạng collections.deque([[startState]]), đóng vai trò như một ngăn xếp lưu trữ các trạng thái cần kiểm tra. Sau đó, khởi tạo một tập đóng exploredSet để theo dõi các trạng thái đã được xét. Tạo một danh sách actions = [[0]] để lưu lại chuỗi hành động tương ứng với từng trạng thái trong frontier, giúp biết được đường đi khi tìm thấy lời giải.
- Ta thực hiện một vòng lặp while để kiểm tra liệu frontier có còn trạng thái nào cần xét hay không. Ở mỗi bước, trạng thái hiện tại được lấy từ cuối frontier bằng phương thức pop(), đây là nguyên tắc LIFO, từ đó ta thấy, cấu trúc dữ liệu mà thuật toán DFS sử dụng là Stack. Cùng với việc lấy trạng thái hiện tại, chuỗi hành động tương ứng cũng được lấy từ actions.pop(). Nếu trạng thái hiện tại thỏa mãn điều kiện đích (isEndState(node[-1][-1])) (tất cả các thùng hàng đã được đặt vào đúng vị trí), thì chuỗi hành động được lưu vào temp và thuật toán kết thúc.
- Nếu trạng thái hiện tại chưa được xét trước đó (không nằm trong tập đóng), nó được thêm vào exploredSet để đánh dấu đã duyệt. Sau đó, thuật toán xác định tất cả các hành động hợp lệ từ trạng thái hiện tại bằng cách gọi legalActions(node[-1][0], node[-1][1]). Với mỗi hành động hợp lệ, một trạng thái mới được tạo ra bằng hàm updateState(node[-1][0], node[-1][1], action), trong đó newPosPlayer và newPosBox là vị trí mới của người chơi và của các thùng sau khi thực hiện hành động đó.
- Nếu trạng thái mới rơi vào tình huống thất bại (isFailed(newPosBox)), chẳng hạn như một thùng bị kẹt hoặc không thể di chuyển tiếp, thì thuật toán sẽ bỏ qua trạng thái này để tránh tìm kiếm vào những nhánh vô ích. Ngược lại, trạng thái hợp lệ sẽ được thêm vào cuối frontier (frontier.append(node + [(newPosPlayer, newPosBox)])) để tiếp tục thuật toán. Đồng thời, danh sách hành động cũng được cập nhật (actions.append(node_action + [action[-1]])) để lưu lại đường đi của người chơi.

2.2 Thuật toán BFS

- Đầu tiên, ta lấy vị trí của các thùng và người chơi lúc bắt đầu tương tự như thuật toán DFS.
- Việc khởi tạo frontier để lưu trữ các trạng thái cần kiểm tra và exploredSet theo dõi các trạng thái đã xét tương tự như thuật toán DFS. Tuy nhiên, danh sách actions lưu chuỗi hành động tương ứng với từng trạng thái sẽ được biểu diễn dưới dạng collections.deque([[startState]]), vì BFS cần sử dụng hàm popleft() và hàm popleft() trong deque nhanh hơn pop(0) trong list (O(1) so với O(n)).
- Ta thực hiện một vòng lặp while để kiểm tra liệu frontier có còn trạng thái nào cần xét hay không. Ở mỗi bước, trạng thái hiện tại được lấy từ đầu frontier bằng phương thức popleft(), đây là nguyên tắc FIFO, từ đó ta thấy, cấu trúc dữ liệu mà thuật toán BFS sử dụng là Queue. Đây chính là điểm khác biệt cơ bản giữa BFS và DFS. Đồng thời, chuỗi hành động tương ứng với trạng thái đó cũng được lấy từ actions.popleft(). Nếu trạng thái hiện tại thỏa mãn điều kiện đích (isEndState(node[-1][-1])), tức là tất cả các thùng đã được đặt vào đúng vị trí mục tiêu, thì chuỗi hành động được lưu vào temp và thuật toán kết thúc.
- Nếu trạng thái hiện tại chưa được xét trước đó (không nằm trong tập đóng), nó được thêm vào exploredSet để đánh dấu đã duyệt. Sau đó, thuật toán xác định tất cả các hành động hợp lệ từ trạng thái hiện tại bằng cách gọi legalActions(node[-1][0], node[-1][1]). Với mỗi hành động hợp lệ, một trạng thái mới được tạo ra bằng hàm updateState(node[-1][0], node[-1][1], action), trong đó newPosPlayer và newPosBox là vị trí mới của người chơi và của các thùng sau khi thực hiện hành động đó.
- Nếu trạng thái mới rơi vào tình huống thất bại (isFailed(newPosBox)), chẳng hạn như một thùng bị kẹt hoặc bị chặn không thể di chuyển tiếp, thì trạng thái đó bị loại bỏ để tránh lãng phí tài nguyên tìm kiếm. Ngược lại, nếu trạng thái hợp lệ, nó sẽ được thêm vào cuối frontier (frontier.append(node + [(newPosPlayer, newPosBox)])) để tiếp tục thuật toán. Đồng thời, danh sách hành động cũng được cập nhật (actions.append(node_action + [action[-1]])) để lưu lại đường đi của người chơi.

2.3 Thuật toán UCS

- Đầu tiên, ta lấy vị trí của các thùng và người chơi lúc bắt đầu tương tự như thuật toán DFS và BFS.
- Tiếp đó, khởi tạo frontier để lưu trữ các trạng thái cần kiểm tra và exploredSet để theo dõi các trạng thái đã xét tương tự như thuật toán DFS và BFS. Tuy nhiên, khác với BFS sử dụng hàng đợi (FIFO) và DFS sử dụng ngăn xếp (LIFO), thuật toán UCS sử dụng PriorityQueue, trong đó các trạng thái được sắp xếp theo chi phí tích lũy của đường đi đến trạng thái đó. Do đó, danh sách actions lưu chuỗi hành động tương ứng với từng trạng thái cũng được lưu trong PriorityQueue, đảm bảo các trạng thái có chi phí thấp nhất được xét trước.
- Ta thực hiện một vòng lặp while để kiểm tra liệu frontier có còn trạng thái nào cần xét hay không. Ở mỗi bước, trạng thái có chi phí thấp nhất sẽ được lấy ra bằng phương thức pop(), từ đó ta thấy, cấu trúc dữ liệu mà thuật toán UCS sử dụng là PriorityQueue. Đồng thời, chuỗi hành động tương ứng với trạng thái đó cũng được lấy ra. Nếu trạng thái hiện

tại thỏa mãn điều kiện đích (isEndState(node[-1][-1])), tức là tất cả các thùng đã được đặt vào đúng vị trí mục tiêu, thì chuỗi hành động được lưu vào temp và thuật toán kết thúc.

- Nếu trạng thái hiện tại chưa được xét trước đó (không nằm trong tập đóng), nó được thêm vào exploredSet để đánh dấu đã duyệt. Sau đó, thuật toán xác định tất cả các hành động hợp lệ từ trạng thái hiện tại bằng cách gọi legalActions(node[-1][0], node[-1][1]). Với mỗi hành động hợp lệ, một trạng thái mới được tạo ra bằng hàm updateState(node[-1][0], node[-1][1], action), trong đó newPosPlayer và newPosBox là vị trí mới của người chơi và của các thùng sau khi thực hiện hành động đó.
- Nếu trạng thái mới rơi vào tình huống thất bại (isFailed(newPosBox)), chẳng hạn như một thùng bị kẹt hoặc bị chặn không thể di chuyển tiếp, thì trạng thái đó bị loại bỏ để tránh lãng phí tài nguyên tìm kiếm. Ngược lại, nếu trạng thái hợp lệ, thuật toán tính toán chi phí mới dựa trên số bước đi (cost(actions + [action[-1]])) và đưa trạng thái vào PriorityQueue.

3 Bảng thống kê về số bước đi

Map	DFS	BFS	UCS
1	79	12	12
2	24	9	9
3	403	15	15
4	27	7	7
5	-	20	20
6	55	19	19
7	707	21	21
8	323	97	97
9	74	8	8
10	37	33	33
11	36	34	34
12	109	23	23
13	185	31	31
14	865	23	23
15	291	105	105
16	-	34	34
17	x	x	x
18	x	x	x

Bảng 1: Bảng thống kê về độ dài đường đi tìm được bởi 3 thuật toán DFS, BFS, UCS tại tất cả các bản đồ game Sokoban.

4 Nhận xét

4.1 So sánh hiệu suất giữa các thuật toán

- Cả ba thuật toán đều có thể tìm ra lời giải dẫn đến chiến thắng trong phần lớn các màn chơi. Tuy nhiên, DFS thường không tối ưu do xu hướng mở rộng theo chiều sâu, dễ rơi vào các nhánh không có lời giải hoặc mất nhiều thời gian để quay lui, dẫn đến đường đi dài hơn đáng kể so với BFS và UCS.

- BFS và UCS cho ra lời giải có số bước di chuyển giống nhau trong hầu hết các trường hợp, do mỗi bước di chuyển trong Sokoban có chi phí như nhau. Tuy nhiên, sự khác biệt giữa chúng nằm ở cách triển khai cấu trúc dữ liệu: UCS sử dụng PriorityQueue, trong khi BFS sử dụng Deque, giúp xử lý nhanh hơn nhưng đôi khi phải xét toàn bộ trạng thái trong một độ sâu trước khi tìm thấy lời giải.
- UCS có thể chậm hơn BFS trong một số trường hợp do chi phí quản lý PriorityQueue và tính toán chi phí của từng trạng thái. Tuy nhiên, với các bản đồ khó, UCS có khả năng tìm ra lời giải tối ưu nhanh hơn BFS, nhờ đặc tính ưu tiên mở rộng các trạng thái có chi phí thấp trước.

4.2 Đánh giá các bản đồ khó

- Một số bản đồ như Lv.5, Lv.16, Lv.17 và Lv.18 có thể được xem là các bản đồ khó giải. Đặc biệt, bản đồ 5 có không gian trạng thái rất lớn do khu vực trung tâm trống, làm tăng số lượng trạng thái cần xét. Trong trường hợp này, cả ba thuật toán đều gặp khó khăn vì phải mở rộng rất nhiều nhánh tìm kiếm.
- Hai bản đồ 17 và 18 không có lời giải, do đó cả ba thuật toán đều không thể tìm ra lời giải.