TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

----------ꕥ----------

CƠ SỞ TRÍ TUỆ NHÂN TẠO – CQ2017/21

Project 1

ROBOT TÌM ĐƯỜNG

Giáo viên hướng dẫn:

Sinh viên thực hiện:

1. Lê Cát Phương – 1712680
2. Hoàng Minh Quân - 1712688
3. Phan Thanh Quan - 1712686

Lớp: CQ2017/21

TP.Hồ Chí Minh, tháng 10 năm 2019

Mục lục

1. Thông tin nhóm

1.1. Thông tin thành viên

Họ tên	MSSV
Lê Cát Phương	1712680
Phan Thanh Quan	1712686
Hoàng Minh Quân	1712688

1.2. Bảng phân công

STT	Công việc	Mô tả	Người thực hiện
1	Greedy algorithm		Phan Thanh Quan
2	Dijkstra algorithm		Hoàng Minh Quân
3	A Star algorithm		Lê Cát Phương
4	Input file, từ các điểm input, tạo list vật cản	Nhận dữ liệu từ file txt và lưu các thông số vào cấu trúc	Hoàng Minh Quân
5	Thiết kế menu trong main, từ các lựa chọn, gọi hàm tương ứng.		Phan Thanh Quan
6	Vật cản di chuyển	Tìm đường đi cho trường hợp vật cản di chuyển	Phan Thanh Quan
7	Nhận xét, báo cáo		Lê Cát Phương

2. Thông tin project

2.1. Mô tả chung project

Ngôn ngữ lập trình: Python 3.7.3

Thư viện ngoài: keyboard 0.13.4 (https://pypi.org/project/keyboard/)

Project tìm đường đi hợp lệ với đầu vào là kích thước bản đồ. điểm bắt đầu, kết thúc, điểm đón, vật cản. Subject có thể di chuyển chéo.

Dữ liệu đầu vào lưu dạng .txt và có tên: “map1.txt”, “map2.txt”, “map3.txt”

Folder project bao gồm:

- Source code
- Huongdansudung.txt
- Baocao.pdf
- Video chạy thử

2.2. Mức độ hoàn thành công việc

Mức độ yêu cầu	Nội dung	Tiến độ	Ghi chú
Mức 1	Cài đặt thành công 1 thuật toán.	100%	
Mức 2	Báo cáo nhận xét 3 thuật toán khác nhau.	100%	Greedy algorithm Dijkstra algorithm A Star algorithm
Mức 3	Điểm đón.	100%	
Mức 4	Đa giác di động.	100%	
Mức 5	Không gian 3D	0%	

3. Chi tiết thuật toán, mức độ công việc, hàm bổ trợ

3.1. A Star algorithm

3.1.1. Mô tả

Một vị trí di chuyển được lưu dạng cấu trúc State gồm các thuộc tính chính:

position(x,y): toạ độ của điểm.

parent: con trỏ trỏ đến vị trí (State) trước khi đi đến vị trí hiện tại.

g: là số bước đã đi của vị trí.

f = g+h trong đó h là khoảng cách manhattan từ vị trí đến goal.

Chú thích 1 số tên biến chính:

open_list: list chứa các vị trí (State) điểm mở, luôn được sắp xếp tăng dần theo đại lượng f.

close_list: list chứa các vị trí đã xét.

current_node: là vị trí hiện tại (State).

path: list tập hợp các vị trí tạo thành đường đi từ S đến G

Chọn điểm start là điểm bắt đầu, khởi tạo open_list chứa điểm start, close_list chứa rỗng.

Vòng lặp: Current_note bằng phần từ đầu tiên của open_list (có giá trị f nhỏ nhất), sau khi chọn, close_list sẽ lưu giá trị của phần tử được chọn. Kiểm tra current_note có trùng với điểm kết thúc hay không. Ở trường hợp không đúng, khởi tạo 1 list chứa 8 bước có thể đi tiếp theo (lên, xuống, sang trái, sang phải,… không phải tường hoặc ra khỏi bản đồ), trong đó thuộc tính parent của từng bước đi trỏ đến current_note. Kiểm tra lần lượt từng bước có nằm trong close_list hoặc open_list hay không. Nếu có, cờ check sẽ = 0. Ở trường hợp current_note có nằm trong open_list, so sánh f của vị trí bước đi và vị trí chứa trong open_list. Nếu f của bước đi nhỏ hơn, xoá vị trí cũ có trong open_list, thêm giá trị của bước đi vào đầu open_list rồi sắp xếp lại open_list tăng dần theo f.

Trong trường hợp current_note trùng với vị trí kết thúc. Truy xuất lần lượt từ current_note về lại vị trí bắt đầu thông qua thuộc tính parent. Lưu vị trí của các bước đi vào path. Return kết quả path là list tập hợp đường đi từ S đến G

3.1.2. Nhận xét

Thuật toán mở các phần từ theo hai điều kiện nên số phần từ trong tập hợp mở sẽ ít hơn so với thuật toán Dijkstra đồng thời đường đi cũng là đường tối ưu nhất.

Điều kiện dừng của vòng lặp là khi gặp được điểm kết thúc hoặc open_list không còn phần tử nào nghĩa là không có đường đi từ điểm bắt đầu đến kết thúc.

3.2. Dijkstra algorithm

3.2.1. Mô tả

Mỗi vị trí được lưu dạng cấu trúc Node_Dijkstra có các thuộc tính chính:

position: toạ độ của vị trí đó .

parent: chứa con trỏ, trỏ đến vị trí trước khi đi đến bước hiện tại.

f: giá trị distance mang ý nghĩa là cần bao nhiêu bước từ S để đi đến vị trị này.

Chú thích một số biến chính:

unexplored_list: Tập hợp tất cả các vị trí trong bản đồ, luôn được sắp xếp tăng dần theo f. Ban đầu giá trị f của tất cả vị trí là ∞ (1000) sau đó từng vị trí sẽ được xét đến và điều chỉnh f cho phù hợp.

explored_list: Tập hợp các vị trí đã được xét đến.

Khởi tạo giá trị f cho vị trí start = 0, tất cả các điểm còn lại là 1000 (= ∞). Khởi tạo unexplored_list chứa tất cả các vị trí trong bản đồ không phải là tường, xếp vị trí start lên đầu (unexplored_list[0] = start). Khởi tạo explored_list là list rỗng.

Vòng lặp: current_node là giá trị đầu tiên unexplored_list cũng là vị trí có f nhỏ nhất. Kiểm tra vị trí này có phải là goal hay không.

Trong trường hợp không phải goal, khởi tạo 1 list chứa 8 bước có thể đi tiếp theo (lên, xuống, sang trái, sang phải,… không phải tường hoặc ra khỏi bản đồ), Trong đó thuộc tính parent của từng bước đi sẽ trỏ đến current_note. Tăng giá trị f của 8 bước này thêm 1 so với current_node. Kiểm tra lần lượt từng bước có nằm trong explored_list, nếu có, không xét bước đó nữa mà xét bước tiếp theo. Nếu không nằm trong explored_list, so sánh f của vị trí bước đi này với f của vị trí chính nó trong unexplored_list. Nếu f của bước đi nhỏ hơn trong unexplored_list, thay đổi giá trị và sắp xếp lại unexplored_list theo thứ tự tăng dần của f.

Trong trường hợp current_note trùng với vị trí kết thúc. Truy xuất lần lượt từ current_note về lại vị trí bắt đầu thông qua thuộc tính parent. Lưu vị trí của các bước đi vào path. Return kết quả path là list tập hợp đường đi từ S đến G.

3.2.2. Nhận xét

Thuật toán khá giống với A star nhưng chỉ xét giá trị khoảng cách nên số tập phần tử trong tập mở sẽ nhiều hơn so với A star nên sẽ tốn không gian lưu trữ hơn và chạy chậm hơn do phải xét nhiều vị trí nhưng kết quả cho ra được đường đi tối ưu.

Điều kiện dừng của vòng lặp là khi gặp được điểm kết thúc hoặc open_list không còn phần tử nào nghĩa là không có đường đi từ điểm bắt đầu đến kết thúc.

3.3. Greedy algorithm

3.3.1. Mô tả

Ý tưởng khá giống Dijkstra algorithm và A star alogorithm tuy nhiên vì thuật toán này không phụ thuộc vào open_list mà chỉ quan tâm vị trí hiện tại và các bước đi tiếp theo nên chương trình sẽ thực hiện đệ quy.

Chú thích một số biến:

neighbors: list chứa tập hợp các bước có thể đi tiếp tính từ vị trị hiện tại và được sắp xếp theo vị trí tăng dần theo đại lượng f (khoảng cách từ vị trí đang xét đến goal)

checked_list: list chứa những vị trí đã xét đến, vì chương trình chạy đệ qui nên nên hàm đệ quy này có thể gọi đến những điểm mà hàm đệ qui khác đã từng gọi (đường đi không đến được goal). Nên check_list sẽ lưu những vị trí đã gọi đệ qui đến để không bị lặp vô hạn.

Đầu vào hàm đệ quy gồm:

Checked_list: List những vị trí đã check qua để nếu đường đi không đúng, những vòng đệ quy sau sẽ không gọi đến vị trí này nữa.

start_node: vị trí bắt đầu

current_node: vị trí hiện tại

goal_note: vị trí kết thúc

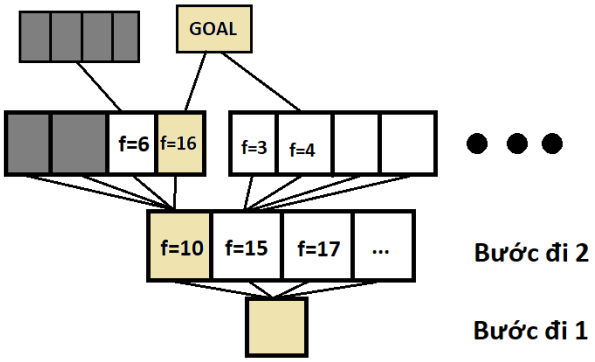
Map: Tập hợp các điểm trên bản đồ

Đệ qui: Sau khi truyền vào hàm đệ quy một current_node (vị trí hiện tại) chương trình sẽ kiểm tra đây có phải điểm kết thúc không? Nếu không phải thì sẽ tạo một list neighbor là tập hợp các bước đi có thể đi tiếp theo (không phải tường hoặc ra khỏi bản đồ). Các bước này không nằm trong checked_list và với mỗi các bước hợp lệ được lưu trong neighbor có

trỏ đến vị trí cha (vị trí trước khi đi đến). Lần lượt gọi đệ qui với tham số đầu vào là các vị trí trong list neighbor.

Trong trường hợp tập hợp neighbor của current_node là rỗng thì hàm đệ qui trả ra list path rỗng (path = []) thì khi đó hàm đệ qui cha sẽ tiếp tục gọi đệ qui con cho các vị trí tiếp theo trong list neighbor, kết quả trả vào biến path. Nếu path ở một hàm đệ qui con nào đó khác rỗng (nghĩa là đã tìm được kết quả) thì hàm đó trả return chính path đó cho hàm cha cho tới khi hết các hàm đệ qui. Vì thế nên nếu đã kiếm được đường đi hợp lệ từ S đến G thì hàm sẽ không gọi thêm bất kỳ hàm đệ qui nào nữa.

Trong trường hợp current_note gặp vị trí kết thúc (goal) thì chương trình sẽ truy xuất ngược từ vị trí hiện tại đến vị trí bắt đầu thông qua con trỏ parent và trả ra list path là tập hợp các bước đi từ S đến G.

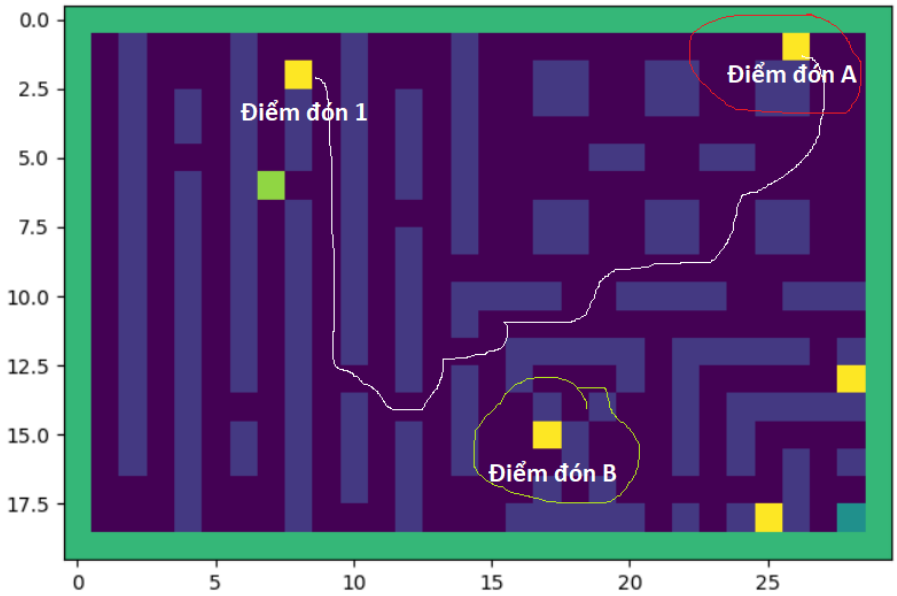


3.3.2. Nhận xét

Như hình trên, thuật toán chỉ chọn bước đi tiếp theo mà gần với điểm đến nhất chứ không xét chi phi di chuyển cho toàn bài toán nên đường đi tìm được không phải đường đi tối ưu nhất.

Điều kiện dừng của vòng lặp là khi gặp được điểm kết thúc hoặc tất cả các điểm có thể quét tới đều nằm trong check_list, khi đó current_note sẽ không tạo được list neighbor, hàm sẽ trả path = [] cho tới khi ra khỏi tất cả hàm đệ qui.

3.4. Điểm đón

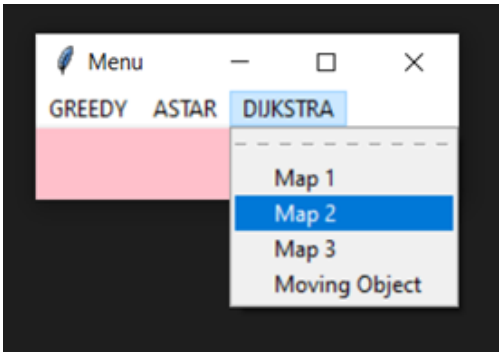


Bản chất việc giải bài toán điểm đón cũng là một bài toán nhỏ có điểm bắt đầu là S hoặc là điểm đón trước và điểm kết thúc là điểm đón tiếp theo hoặc là G.

Trong list điểm đón, chương trình sẽ tính toán từ vị trí hiện tại (vị trí bắt đầu, vị trí điểm đón trước) đến điểm cần đón. List Point sẽ chứa sắp xếp lần lượt [điểm bắt đầu, điểm đón 1, điểm đón 2,…, điểm kết thúc] trong đó các điểm đón được lựa chọn thứ tự dựa vào khoảng cách giữ điểm đón với vị trí hiện tại.

Ghi chú: Giả sử có 2 điểm A và B và khoản cách manhattan từ S đến A ngắn hơn từ S đến B. Vì thứ tự điểm đón xét trên khoảng cách của vị trí bắt đầu hoặc vị trí trường đó nên mặc dù để đi từ S đến A có đi qua B (hoặc gần đến B) nhưng chương trình sẽ đón A trước vì khoảng cách từ S đến A vẫn gần hơn từ S đến B như hình trên.

3.5. Menu



Khi chạy file, chương trình sẽ hiện lên bảng mục menu, người dùng sẽ lựa chọn thuật toán và các map lưu sẵn trong chương trình. Khi đó chương trình sẽ truyền vào hàm func các tham số khác nhau tuỳ thuộc yêu cầu người dùng.

Trong hàm func chương trình sẽ load map từ file .txt, thực hiện tạo list các vị trí vật cản từ các vị trí góc của vật cản, list điểm đón. Từ vị trí bắt đầu, kết thúc, điểm đón, chạy vòng lặp để kiểm tra việc nên đi theo thứ tự nào (đón điểm nào trước, điểm nào sau) cho kết quả vào list Point.

Từ list Point lần lượt chạy thuật toán theo yêu cầu người dùng cho tham số đầu vào là Point[i], Point[i+1] (điểm bắt đầu là Point[i], kết thúc là Point[i+1]). Hiện thị đường đi lên màn hình, thực hiện cho đến khi hết list Point.

Nếu người dùng yêu cầu vật cản di chuyển, chương trình sẽ gọi hàm riêng biệt (mục 3.7).

Nếu không thể tìm ra đường đi chương trình sẽ hiện ra dialog("không có đường đi") còn nếu tìm được đường đi, hiện dialog("chi phí đường đi:…").

3.6. Đọc file, tạo vật cản từ input

Sau khi người dùng chọn map muốn sử dụng, tham số tương ứng với tên map sẽ được truyền vào hàm đọc file.

3.6.1 Đọc nội dung file

Nội dung file được đọc bằng hàm readlines(), và trả ra danh sách chứa từng dòng của nội dung file txt. Từ đó hàm thực hiện split() trên từng dòng với tham số là dấu phẩy, để tách ra từng phần tử được ngăn cách bởi dấu phẩy, tạo thành một danh sách tọa độ. Nội dung mỗi dòng được đưa vào thành phần tương ứng của map theo quy định đề bài.

Ở bước lấy danh sách điểm đón, chương trình kiểm tra kích thước danh sách tọa độ ở dòng thứ 2, nếu có nhiều hơn 4 tọa độ (2 tọa độ x y của start và 2 tọa độ của goal) thì sẽ lấy các tọa độ của các điểm pickup, bắt đầu từ giá trị thứ 5 của dòng đó.

Ở bước lấy danh sách chướng ngại vật, đầu tiên chương trình lấy số lượng chướng ngại vật ở dòng thứ 3. Sau đó chương trình lấy lần lượt từng danh sách đỉnh của các chướng ngại vật và gọi hàm linePoints() cho từng cặp đỉnh để có danh sách các điểm tạo thành cạnh giữa 2 đỉnh đó.

3.6.2 Tính tọa độ các điểm tạo thành cạnh giữa 1 cặp đỉnh

Hàm linePoints() áp dụng thuật toán vẽ đường thẳng trên tọa độ số nguyên của Bresenham.

Đầu tiên, chương trình xác định độ dốc của đường thẳng bằng cách xác định độ chênh lệch giữa $\triangle y$ và $\triangle x$, sau đó kiểm tra tọa độ tương đối giữa 2 đỉnh để xác định hướng vẽ.

Hàm linePoints() dựa trên các điều kiện đã xác định ở trên để gọi 2 hàm phụ: lineLow() cho đường thoải (có độ dốc thấp hay $\triangle y < \triangle x$) và lineHigh() cho đường dốc (có độ dốc cao hay $\triangle y > \triangle x$).

Hàm lineLow() sẽ tăng dần giá trị của biến trục x và xét giá trị biến trục y tương ứng. Do độ chênh lệch tọa độ trên trục x lớn hơn nên việc tăng dần giá trị trên trục x sẽ đảm bảo đường biên cạnh được kín, không bị lủng lỗ. Áp dụng suy luận tương tự cho hàm lineHigh() với trục y chính.

3.7. Vật cản di chuyển

3.7.1. Thiết lập vật cản di chuyển

Thuộc tính trong map có list tập hợp các vị trí của vật cản.

Trong class map, tạo 4 phương thức tịnh tiến toàn bộ các điểm vật cản từ list thuộc tính 1 đơn vị: sang trái, phải, lên xuống. Nếu vật cản di chuyển ra khỏi biên ở một cạnh thì sẽ đến biên của cạnh đối diện.

3.7.2. Tìm đường đi khi vật cản di chuyển

Khi người dùng chọn vào mục Moving Object chương trình sẽ gọi vòng lặp cho tới khi người dùng nhấn "esc". Trong vòng lặp, random 1 số từ 1 đến 4 tương đương với việc vật cản di chuyển sang trái, phải, lên, xuống 1 bước. Với dữ liệu sau khi di chuyển vật cản, gọi hàm tìm đường đi (greedy, A star, Dijktra). Hiện thị đường đi lên màn hình qua thư viện matplotlib.pyplot.

Lưu ý: trong bài toán tìm đường đi khi vật cản di chuyển, chương trình sẽ không giải bài toán điểm đón, đường đi hiện trên màn hình là đường đi hợp lệ từ vị trí bắt đầu đến vị trí kết thúc.

Tài liệu tham khảo

Bresenham's line algorithm:
https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

A star algorithm:
https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2

Python GUI:
https://www.geeksforgeeks.org/python-gui-tkinter/