Báo cáo đồ án 1:

Robot Tìm Đường

Môn: Cơ sở trí tuệ nhân tạo

Sinh viên thực hiện:

Nguyễn Ngọc Băng Tâm - 1712747

Bùi Thị Cẩm Nhung - 1712645

26/10/2019

1. Tổng quan

1.1. Mô tả đồ án

Bài toán Đường đi ngắn nhất (Shortest path problem) là một trong những bài toán kinh điển trong lý thuyết đồ thị. Phát biểu một cách đơn giản, bài toán yêu cầu tìm đường đi giữa hai đỉnh (hoặc nhiều đỉnh) trong đồ thị sao cho tổng trọng số của các cạnh tạo nên đường đi đó là nhỏ nhất.

Có nhiều biến thể của bài toán đường đi ngắn nhất:

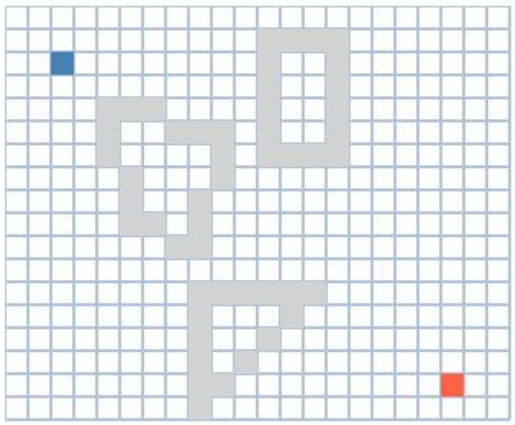
- Tìm đường đi ngắn nhất nguồn đơn (single source).
- Tìm đường đi ngắn nhất đích đơn (single destination).
- Tìm đường đi ngắn nhất cặp đơn (single pair).
- Tìm đường đi ngắn nhất giữa mọi cặp đỉnh (all pair).

Đồ thị được cho có thể là đồ thị vô hướng hoặc có hướng, có trọng số hoặc không có trọng số, tồn tại chu trình hoặc không.

Trong khuôn khổ đồ án này, sinh viên nghiên cứu và cài đặt một số thuật toán tìm đường đi ngắn nhất cặp đơn trên đồ thị là một bản đồ (ma trận) có kích thước cho trước.

Trên ma trận có chứa một số vật cản là các đa giác, mỗi đa giác được xác định bằng tập hợp các đỉnh của đa giác đó. Không có bất cứ đa giác nào nằm chồng lên nhau và nằm chồng lên điểm xuất phát và điểm đích.

Tìm đường đi ngắn nhất từ đỉnh xuất phát S đến điểm đích G sao cho không va phải hay xuyên qua bất cứ vật cản nào.



Hình 01. Minh họa bản đồ

1.2. Đánh giá mức độ hoàn thành

- Đánh giá tổng thể: 90%
- Chi tiết từng yêu cầu:

Yêu cầu	Hoàn thành	Người thực hiện
Mức 1 (40%): Tìm đường đi từ S đến G.	x	Nhung
Mức 2 (30%): Tìm đường đi có chi phí nhỏ nhất từ S đến G.	x	Tâm
Mức 3 (20%): Tìm đường đi từ S đến G, đồng thời đi qua các điểm đón sao cho tổng chi phí là nhỏ nhất.	x	Nhung
Mức 4 (10%): Vật cản là các đa giác động. Tìm đường đi từ S đến G.		
Mức 5 (+ 10%): Thể hiện mô hình trên không gian ba chiều.		

1.3. Tóm tắt thuật toán đã sử dụng

- Vẽ đa giác: sử dụng thuật toán vẽ đường thẳng Bresenham để nối giữa các đỉnh trong một đa giác.
- Tìm đường đi: sử dụng thuật toán DFS và BFS.
- Tìm đường đi ngắn nhất: sử dụng thuật toán Dijkstra, Greedy Best First Search và A*.
- Tìm đường qua các điểm đón và kết thúc tại điểm đích: Sử dụng thuật toán Simulated Annealing + Dijkstra.
- Tìm đường đi qua các vật cản động: sử dụng thuật toán D* Lite.

1.4. Các ràng buộc của bài toán

Tập hợp các tọa độ điểm là các tọa độ nguyên.

Để hạn chế việc đi xuyên qua đa giác, tại một ô chỉ cho phép đến 4 khác nằm ở các hướng trên, dưới, trái, phải mà không cho đi chéo. Lúc này chi phí đi từ ô đó đến 4 ô kề là như nhau và bằng 1.

2. Cơ sở lý thuyết

2.1. Không gian trạng thái

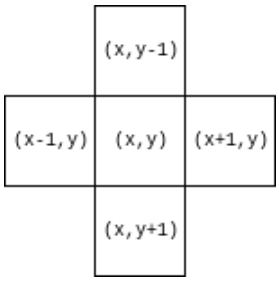
- Trạng thái: vị trí (x,y) trên ma trận
- Hành động: di chuyển trên, dưới, trái, phải
- Hàm kế nhiệm: cập nhật vị trí
- Mục tiêu: đến vị trí (x',y') mong muốn

Kích thước không gian trạng thái của bài toán: $M * N$.

2.2. Xây dựng đồ thị

Để cài đặt các thuật toán tìm đường trên đồ thị được cho là ma trận kích thước $M * N$, ta cần xây dựng danh sách đỉnh kề G với G_u chứa danh sách các đỉnh kề v của u (cạnh nối từ u đến v) và trọng số của cạnh (u,v).

Xem mỗi ô trong ma trận là một đỉnh của đồ thị. Ta có thể ánh xạ mỗi ô (x,y) với một đỉnh i trong đồ thị bằng công thức $i * N + j$. Tuy nhiên việc làm này là không cần thiết vì ta có thể sử dụng mảng hai chiều để lưu giá trị của ô (x,y) tại dòng y và cột x.



Hình 02. Tọa độ 4 ô kề của (x,y)

Biết rằng mỗi ô (x,y) trong ma trận sẽ có tối đa 4 ô kề trên, dưới, trái, phải với tọa độ lần lượt là $(x,y-1)$, $(x,y+1)$, $(x-1,y)$ và $(x+1,y)$ và chi phí di chuyển từ ô (x,y) đến các ô này đều là như nhau và bằng 1. Ta có hai cách để xây dựng danh sách (ô kề, chi phí) G:

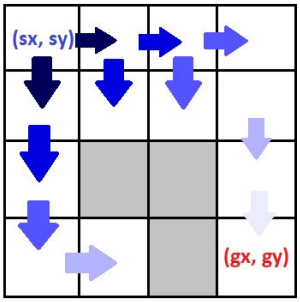
1. Duyệt qua từng ô (x,y) trong ma trận, thêm vào $G_{x,y}$ là cặp các (tọa độ ô kề, chi phí). Như vậy để lấy được danh sách các (ô kề, chi phí) của ô $u(x,y)$, ta chỉ cần truy cập trực tiếp vào $G_{x,y}$.
2. Gọi dx_i, dy_i, w_i lần lượt là độ lệch tọa độ x, y và chi phí tính từ u đến ô kề thứ i ($1 \le i \le 4$ tương ứng với các hướng trên, dưới, trái phải). Không trực tiếp xây dựng G. Khi cần lấy danh sách các (ô kề, chi phí) của ô $u(x,y)$, ta phát sinh ra tọa độ ô kề thứ i của u bằng công thức $(x+dx_i, y+dy_i)$ và chi phí tương ứng là w_i.

Trong đồ án này, nhóm thực hiện theo cách làm thứ hai.

2.3. Thuật toán BFS

Breadth-First Search (Tìm kiếm theo chiều sâu) là một trong những thuật toán sử dụng chiến lược Tìm kiếm mù (Blind Search hay Uninformed Search) để tìm kiếm trong một không gian trạng thái cho trước.

Áp dụng với bài toán đặt ra, thuật toán BFS cho phép từ 1 ô nguồn loang rộng đến các ô đích khác trong đồ thị dạng ma trận kích thước $M * N$. Đặc biệt khi trọng số mỗi cạnh nối - tương ứng với chi phí đi từ một ô đến ô kề nó, đều bằng nhau thì BFS cho ta đường đi có chi phí nhỏ nhất.



Hình 03. Mô tả đường đi của BFS

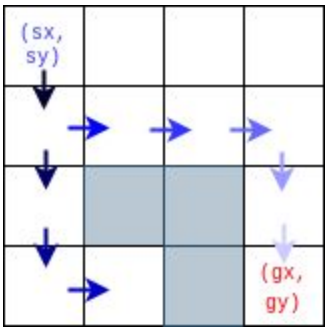
Tư tưởng thuật toán: Xuất phát từ một ô nguồn đã cho, di chuyển đồng thời tới tất cả các ô kề của nó (tương tự như vệt sóng lan) cho đến khi gặp ô đích hoặc không còn ô nào để xét.

Lưu ý về cài đặt: Để thể hiện tính loang rộng của thuật toán, ta sử dụng cấu trúc dữ liệu queue (hàng đợi) để chứa các ô kề được mở.

2.4. Thuật toán DFS

Depth-First Search (Tìm kiếm theo chiều sâu) là một trong những thuật toán sử dụng chiến lược Tìm kiếm mù (Blind Search hay Uninformed Search) để tìm kiếm trong một không gian trạng thái cho trước.

Áp dụng đối với bài toán đặt ra, thuật toán DFS cho phép tìm kiếm đường đi từ một ô nguồn đến các ô đích khác trong một đồ thị là ma trận kích thước $M * N$. Tuy nhiên không đảm bảo đường đi tìm được có chi phí nhỏ nhất.



Hình 02. Mô tả đường đi của DFS

Tư tưởng thuật toán: Xuất phát từ một ô nguồn đã cho, ta chọn một hướng để bắt đầu đi và men theo hướng đi đã chọn cho đến khi không thể đi được nữa. Lúc này, ta quay trở lại theo hướng vừa tới và chọn đi theo một hướng khác. Tiếp tục men theo hướng đi này cho đến khi tới được ô cần tìm hoặc không thể đi được nữa thì lặp lại quá trình trên.

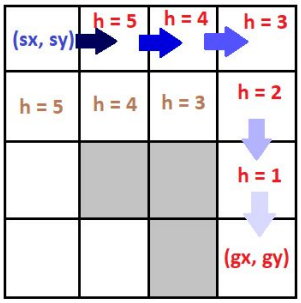
Lưu ý về cài đặt:

- Để thể hiện việc đi sâu vào một nhánh, có thể sử dụng đệ quy hoặc cấu trúc dữ liệu stack (ngăn xếp). Trong đồ án này, nhóm lựa chọn cài đặt DFS bằng stack.
- Để tránh việc đi vào chu kỳ vô tận, mỗi ô trong bài toán chỉ được xét một lần duy nhất. Việc xét lại một ô nhiều hơn một lần không mang nhiều ý nghĩa vì khi một ô được xét, tất cả các đường đi có thể đi qua ô đó đều đã được ghi nhận (đã xét hết tất cả các ô kề của ô hiện tại).

2.5. Thuật toán Greedy Best-First Search

Greedy Best-First Search (Tìm kiếm tối ưu kiểu tham lam) là một trong những thuật toán sử dụng chiến lược Tìm kiếm có thông tin (Informed Search) để tìm kiếm trong một không gian trạng thái cho trước.

Áp dụng với bài toán đặt ra, thuật toán Greedy BFS cho phép từ một đỉnh mở rộng tới một ô gần đích nhất với hy vọng việc mở rộng như vậy có thể dẫn đến lời giải một cách nhanh nhất. Do đó thuật toán này đánh giá chi phí các ô chỉ dựa trên chi phí từ ô đó đến ô đích: $f(n) = h(n)$ với $f(n)$ là hàm đánh giá và $h(n)$ là hàm heuristic.



Hình 04. Mô tả đường đi của Greedy BFS

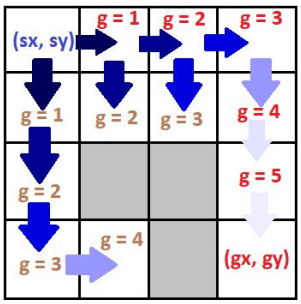
Tư tưởng thuật toán: Xuất phát từ ô nguồn cho trước, tính chi phí heuristic của tất cả các ô kề so với ô đích (lưu ý chỉ tiếp tục duyệt đến ô kề có chi phí heuristic tốt nhất). Lặp lại quá trình trên cho đến khi gặp ô đích hoặc không còn ô nào có thể xét.

Lưu ý về cài đặt: Sử dụng cấu trúc dữ liệu priority queue (hàng đợi ưu tiên) để chứa ô kề và chi phí heuristic so với ô đích. Ô có chi phí heuristic càng thì độ ưu tiên càng cao. Trường hợp 2 ô có cùng chi phí heuristic chọn ô có tọa độ nhỏ hơn.

- Với ràng buộc chỉ có thể di chuyển trong lân cận 4 hướng trên, dưới, trái, phải, hàm heuristic được chọn là khoảng cách manhattan giữa ô kề đang xét và ô đích.

2.6. Thuật toán Dijkstra

Thuật toán Dijkstra (Dijkstra's Algorithm) là một trong những thuật toán sử dụng chiến lược Tìm kiếm mù (Blind Search hay Uninformed Search) để tìm kiếm trong một không gian trạng thái cho trước.



Hình 05. Mô tả đường đi của Dijkstra

Áp dụng với bài toán đặt ra, thuật toán Dijkstra cho phép từ tìm đường đi ngắn nhất một ô nguồn đã cho đến các tất cả các ô khác trong đồ thị. Thuật toán này đánh giá chi phí các ô chỉ dựa trên chi phí từ ô gốc đến ô đó: $f(n) = g(n)$ với $f(n)$ là hàm đánh giá và $g(n)$ là hàm chi phí thực tế.

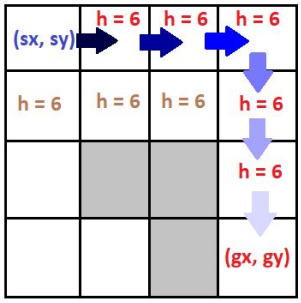
Tư tưởng thuật toán: Xuất phát từ một ô nguồn cho trước, mở các ô lân cận và cập nhật chi phí từ ô xuất phát đến ô đó. Lần lượt chọn các đường đi ngắn nhất hiện tại để phát triển tiếp. Thực hiện cho đến khi gặp ô đích hoặc không còn ô nào có thể đi.

Lưu ý về cài đặt: Để nhanh chóng chọn ra đường đi có chi phí thấp nhất ở thời điểm hiện tại, ta sử dụng priority queue (hàng đợi ưu tiên) chứa các ô kề kèm chi phí được mở.

2.7. Thuật toán A*

A* là một trong những thuật toán sử dụng chiến lược Tìm kiếm có thông tin (Informed Search) để tìm kiếm trong một không gian trạng thái cho trước. Thuật toán này là sự kết hợp giữa thuật toán Dijkstra và Greedy BFS để tối ưu hóa việc tìm kiếm đường đi.

Áp dụng với bài toán đặt ra, thuật toán A* cũng cho phép từ một ô mở rộng đến ô gần đích nhất với hy vọng việc mở rộng như vậy sẽ dẫn đến lời giải một cách nhanh nhất. Tuy nhiên, thuật toán này đánh giá chi phí các ô dựa trên tổng chi phí từ nút gốc đến nút đó và chi phí từ nút đó đến đích: $f(n) = g(n) + h(n)$ với $f(n)$ là hàm đánh giá, $g(n)$ là hàm chi phí thực và $h(n)$ là hàm heuristic.



Hình 06. Mô tả đường đi của A*

Tư tưởng thuật toán: Xuất phát từ một ô và tính chi phí ước lượng của tất cả các ô lân cận (lưu ý chỉ tiếp tục duyệt đến ô kề có chi phí ước lượng tốt nhất). Lặp lại quá trình trên cho đến khi gặp ô đích hoặc không còn ô nào có thể đi.

Lưu ý cài đặt: Sử dụng cấu trúc dữ liệu priority queue (hàng đợi ưu tiên) để chứa ô kề và chi phí ước lượng so với ô đích. Chi phí ước lượng của ô càng nhỏ thì độ ưu tiên càng cao. Trường hợp hai ô có cùng chi phí ước lượng, ta chọn ô có tọa độ nhỏ hơn.

- Với ràng buộc chỉ có thể di chuyển trong lân cận 4 hướng trên, dưới, trái, phải, hàm heuristic được chọn là khoảng cách manhattan giữa ô kề đang xét và ô đích.

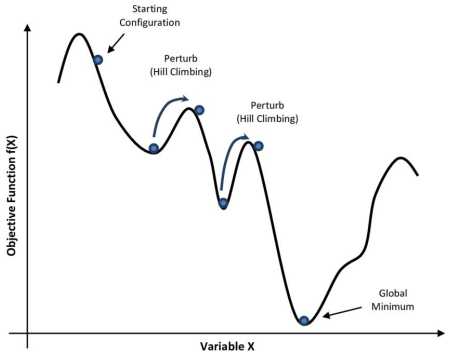
2.6. Thuật toán Simulated Annealing

Simulated Annealing (Thuật toán Mô phỏng luyện thép) là một thuật toán ngẫu nhiên (Randomized Algorithm) hỗ trợ tìm ra lời giải tối ưu toàn cục trong các bài toán tối ưu.

Áp dụng với bài toán được đặt ra, ta có thể xem việc tìm được đường đi ngắn nhất xuất phát từ điểm nguồn, đi qua các điểm đón và kết thúc tại điểm đích cho trước về bài toán Traveling Salesman (Bài toán Người giao hàng) bằng việc sinh ra một đường đi trực tiếp với chi phí 0 giữa điểm nguồn và điểm đích. Lúc này, ta có thể sử dụng Simulated Annealing để nhanh chóng tìm ra một lịch trình đủ tốt với tổng độ dài của các cạnh trên đường đi là nhỏ nhất.

Tư tưởng của thuật toán:

- Lấy ý tưởng từ quá trình luyện kim trong vật lý, thuật toán bắt đầu từ nhiệt độ T_0 rất lớn. Tại thời điểm bắt đầu này, cấu trúc các phân tử còn khá rời rạc và lỏng lẻo, tương tự như việc lời giải hiện tại của ta còn quá thiếu sót thông tin để có thể đánh giá đó có phải là một lời giải tối ưu hay không.
- Quá trình luyện kim bắt đầu. Nhiệt độ được giảm từ từ. Ở mỗi lần nhiệt độ giảm, ta sinh ra ngẫu nhiên một lời giải mới từ lời giải đã có. Nếu lời giải mới được sinh ra có "chất lượng" tốt hơn lời giải tốt nhất hiện tại (trong bài toán của ta, lời giải có chất lượng tốt hơn là một lịch trình với tổng độ dài các cạnh trên đường đi là nhỏ hơn), ta chấp nhận lời giải mới này. Ngược lại, ta biết rằng đây là một lời giải tồi hơn, nhưng ta vẫn lựa chọn chấp nhận nó với xác suất chấp nhận tỉ lệ nghịch với nhiệt độ. Việc chấp nhận này giúp ta không bị mắc kẹt trong một lời giải tối ưu cục bộ. Xác suất chấp nhận tỉ lệ nghịch với nhiệt độ cũng đồng nghĩa với việc càng về sau, một lời giải tồi càng khó được chấp nhận hơn.



Hình 07. Mô tả quá trình tìm kiếm của Simulated Annealing

- Quá trình luyện kim sẽ kết thúc nếu đạt tới nhiệt độ giới hạn hoặc khi tổng số bước đã vượt quá một ngưỡng đã định trước.

Lưu ý về cài đặt:

- Ở mỗi lần lặp, nhiệt độ giảm với hệ số $\alpha = 0.95$.
- Hàm xác suất chấp nhận tuân theo phân phối Boltzman trong vật lý học thống kê: $e^{-\frac{\Delta}{T}}$ với Δ là độ chênh lệch giữa đường đi hiện tại và đường đi tốt nhất hiện có, T là nhiệt độ hiện tại.

- Để xác định chi phí của đường đi hiện tại, ta sử dụng thuật toán Dijkstra tìm đường đi ngắn nhất giữa mọi cặp điểm mà ta quan tâm trong đồ thị, cụ thể gồm ô nguồn, ô đích và các ô đón. Như vậy đường đi với lịch trình S → B → C → G có chi phí sẽ bằng tổng chi phí đường đi ngắn nhất từ S → B, B → C, C → G.

3. Thực nghiệm

3.1. Hướng dẫn sử dụng chương trình

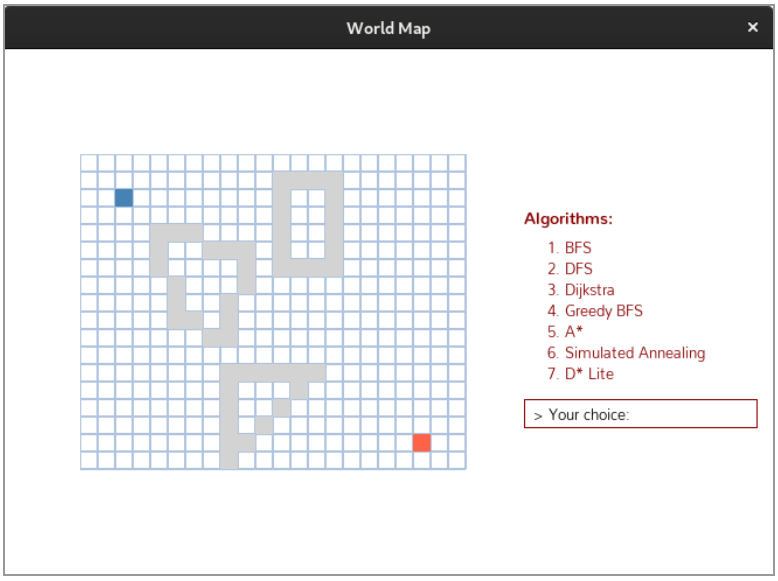
3.1.1. Cài đặt môi trường và các thư viện cần thiết (Xem file README.pdf)

3.1.2. Giao diện chương trình

Sau khi chương trình khởi động sẽ hiện lên một hộp thoại yêu cầu nhập đường dẫn đến file input. Đường dẫn nhập vào có thể là đường dẫn tuyệt đối hoặc tương đối.

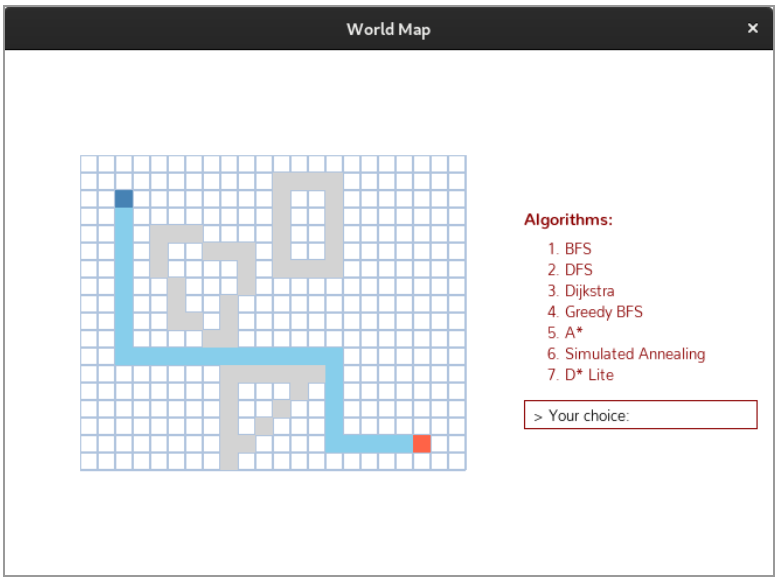
Hình 08. Hộp thoại nhập file input

Cửa sổ chính của chương trình được chia thành hai phần: bản đồ ứng với file input đã được nhập ở bước trước và danh sách các thuật toán bạn muốn thử nghiệm trên bản đồ này.



Hình 09. Chương trình chính

Để chạy một thuật toán, nhập vào một số nguyên từ 1 - 7 tương ứng với thứ tự của thuật toán muốn chạy. Kết quả được trả về ngay trên chính bản đồ đang hiển thị.



Hình 10. Kết quả chạy thử nghiệm BFS trên bản đồ được cho

Lưu ý: Chi phí của đường đi sẽ được hiển thị trên terminal sau khi thực hiện xong quá trình truy vết đường đi. Đặc biệt với thuật toán A*, chi phí đường đi đã được cộng dồn các giá trị heuristic trong quá trình tìm kiếm.

```
pygame 1.9.6
Hello from the pygame community. https://www.pygame.org/contribute.html
User enters path: ./sample/1.in
MAP: Successfully load map from './sample/1.in'
* Running BFS:
PATH: Tracing...
PATH: Finish tracing
PATH: 30.0000
* Running DFS:
PATH: Tracing...
PATH: Finish tracing
PATH: 232.0000
* Running Dijkstra:
PATH: Tracing...
PATH: Finish tracing
PATH: 30.0000
* Running Greedy BFS:
PATH: Tracing...
PATH: Finish tracing
PATH: 30.0000
* Running A*:
PATH: Tracing...
PATH: Finish tracing
PATH: 495.0000
* Running Simulated Annealing:
PATH: Tracing...
PATH: Finish tracing
PATH: 30.0000
```

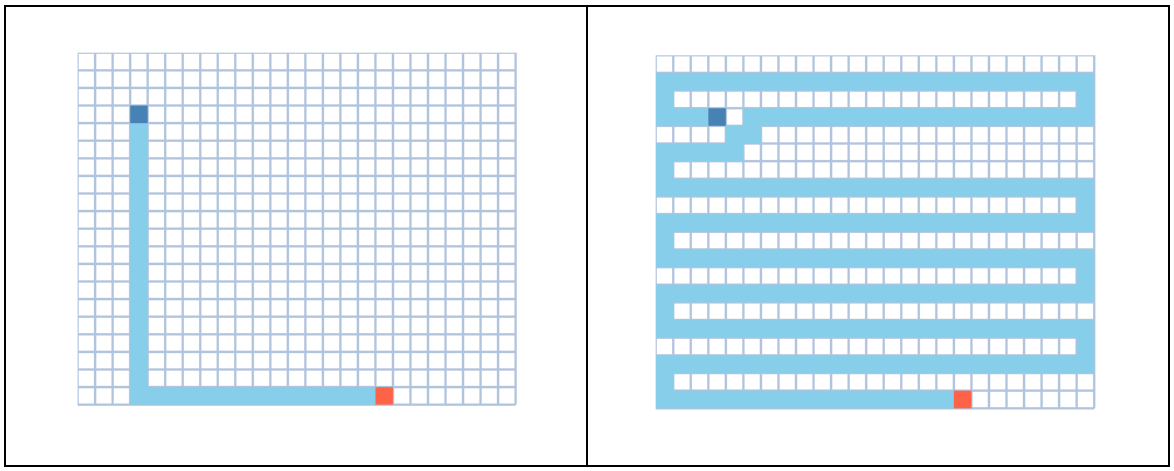
Hình 11. Chi phí đường đi

3.2. Các thử nghiệm

Danh sách các file input đã thử nghiệm được lưu trong thư mục sample. Thực hiện thử nghiệm trên 6 file input, cụ thể:

- 1.in: Không có vật cản
- 2.in: Không có đường đi
- 3.in: Có vật cản, không có điểm đón
- 4.in: Không có vật cản, có các điểm đón
- 5.in: Có vật cản, có các điểm đón
- 6.in: Hai đa giác tiếp xúc với nhau

Thử nghiệm 1:



Hình 12. Kết quả chạy thử nghiệm BFS (trái) và DFS (phải) trên bản đồ 01

Nhận xét:

Thuật toán BFS và DFS gần như đối lập nhau về tư tưởng: BFS chú trọng mở rộng theo chiều rộng trong khi DFS lại chăm chú mở rộng theo chiều sâu.

Chính vì DFS tập trung phát triển một nhánh duy nhất, với đồ thị không có vật cản hoặc rất ít vật cản, nó gần như chắc chắn sẽ tìm ra một đường đi hợp lệ tới đích ngay từ những lần thử đầu tiên. Đánh đổi lại, đường đi thu được lại là một đường đi rất dài.

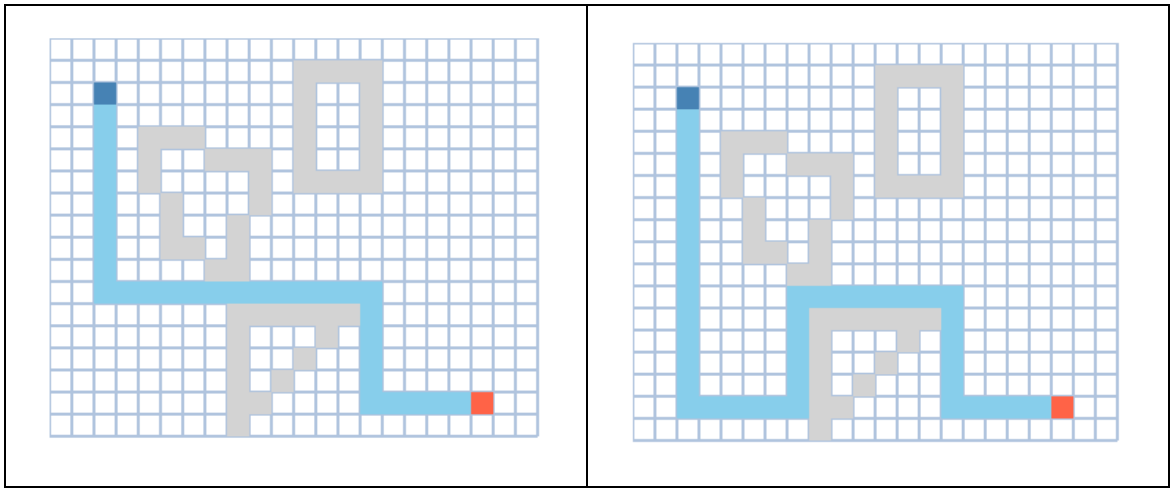
BFS làm tốt hơn trong vai trò tìm ra một đường đi tối ưu. Thế nhưng vì ở mỗi bước, nó phải đồng loại xét tất cả các ô lân cận, do đó thời gian để tìm ra đường đến đích lâu hơn DFS.

Trong ví dụ trên, BFS gần như phải duyệt qua toàn bộ ma trận để tìm được đường đi tới ô đích.

Chính vì những ưu và nhược điểm như vậy, BFS và DFS thường được sử dụng vào các mục tiêu khác nhau:

- BFS: sử dụng khi muốn tìm đường đi ngắn nhất trên đồ thị vô hướng hoặc có hướng, không trọng số hoặc trọng số bằng nhau ở tất cả các cạnh.
- DFS: sử dụng để kiểm tra có tồn tại đường đi từ ô nguồn tới ô đích hay không.

Thử nghiệm 2:



Hình 13. Kết quả chạy thử nghiệm Dijkstra (trái)
và Greedy Best First Search (phải) trên bản đồ 03

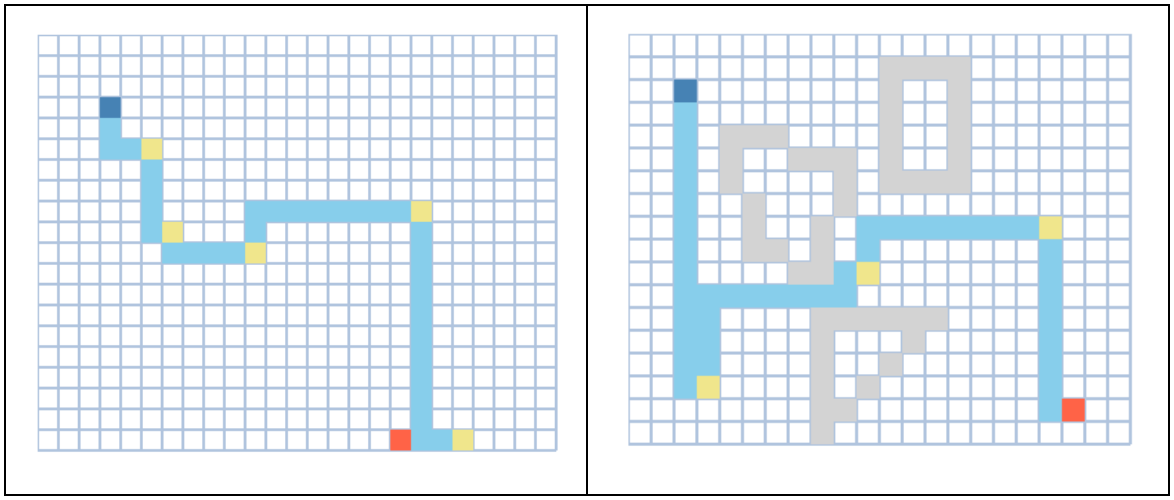
Nhận xét:

Cùng là các thuật toán áp dụng cho đồ thị có trọng số, hướng tiếp cận của Dijkstra và Greedy Best First Search lại hoàn toàn khác nhau. Dijkstra quan tâm đến việc tối ưu hóa chi phí trên đường đi, trong khi Greedy tập trung tối ưu hóa thời gian tìm đường đến đích.

Trong thử nghiệm trên, Dijkstra có vẻ chiếm ưu thế hơn về mặt chi phí đường đi cũng như thời gian tìm kiếm. Nguyên nhân chính là do Greedy Best First Search sử dụng hàm heuristic để định hướng quá trình tìm kiếm. Nhờ heuristic, nó "cảm nhận" được mình đang tiến về phía điểm đích theo con đường tối ưu nhưng khi gặp vật cản, heuristic được sử dụng lại không phát huy tốt tác dụng khi tiếp tục "đánh lừa" rằng khoảng cách manhattan càng ngắn thì càng gần tới đích. Trên thực tế, khoảng cách này không tính đến chi phí phải đi vòng qua các vật cản.

Trong khi đó, Dijkstra dựa vào chi phí đường đi thực để quyết định nước đi tiếp theo. Nhờ vậy, nó sẽ ngay lập tức phát hiện ra được vật cản và chọn được một đường đi khác tốt hơn trong danh sách các đường đi ngắn nhất hiện tại để phát triển.

Thử nghiệm 3:



Hình 14. Kết quả chạy thử nghiệm Simulated Annealing
trên bản đồ 04 (trái) và 05 (phải)

4. Tài liệu tham khảo

1. Stanford AStar Comparison
2. Stanford Heuristics
3. Redblob Pathfinding
4. CMU Robotic Motion Planning: A* and D* Search
5. Simulated Annealing Technique for Routing in a Rectangular Mesh Network