

ĐẠI HỌC BÁCH KHOA HÀ NỘI

Hanoi University of Science and Technology



BÁO CÁO BÀI TẬP LỚN

Học phần: Nhập môn trí tuệ nhân tạo
Đề tài: Tìm đường đi ngắn nhất trên bản đồ

Nhóm 13

Nguyễn Kiều Linh	20215078
Trần Mạnh Toàn	20215149
Trần Đức Kiên	20215071
Lê Bá Trọng	20215153
Khúc Duy Hòa	20210357

Hà Nội, ngày 06 tháng 08 năm 2024

Mục lục

I. Phân công công việc.....	2
II. Đề tài.....	3
1. Giới thiệu bài toán.....	3
2. Biểu diễn không gian bài toán.....	3
3. Phương pháp tìm kiếm lời giải.....	4
3.1 Áp dụng giải thuật A*	4
3.2 Áp dụng giải thuật Dijkstra.....	4
3.3 Áp dụng giải thuật Bellman-Ford.....	5
3.4 So sánh 3 giải thuật.....	5
4. Cài đặt chương trình.....	6
5. Kết quả.....	10

I. Phân công công việc

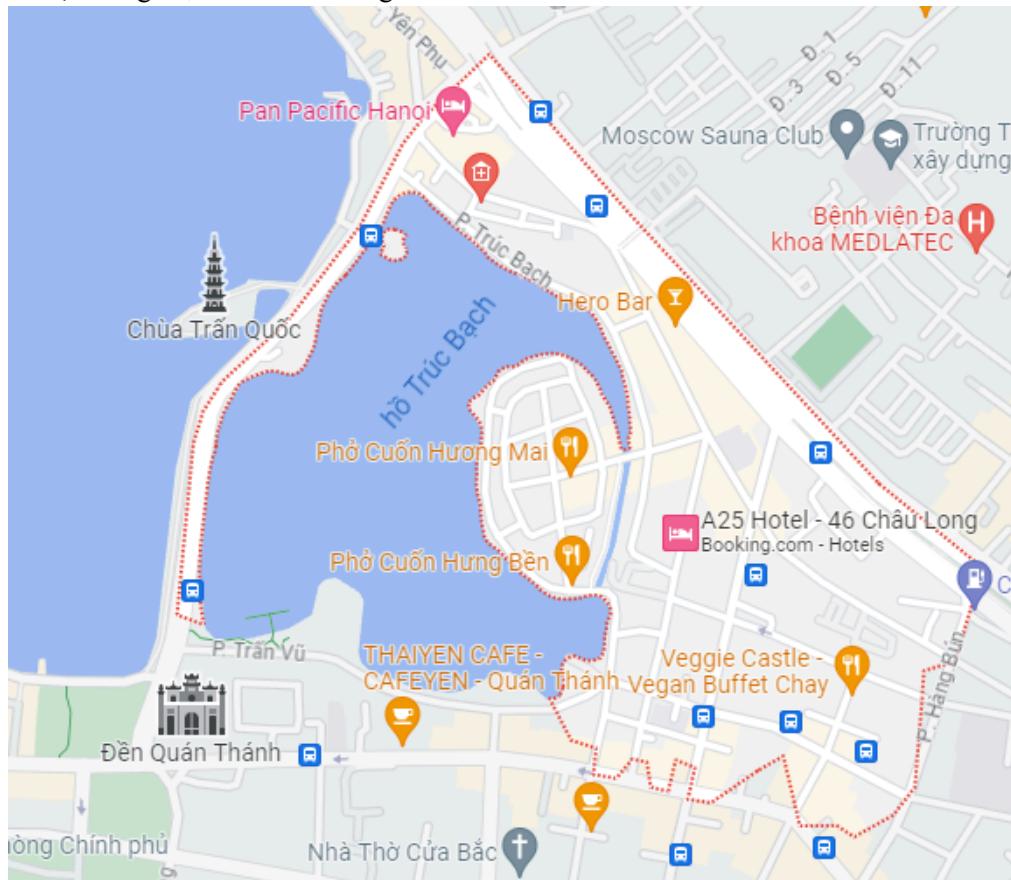
Họ và tên	MSSV	Công việc thực hiện	Điểm
Nguyễn Kiều Linh	20215078	Phân chia nhiệm vụ cho các thành viên Phân tích yêu cầu nghiệp vụ Thiết kế UI, UX, tổng hợp báo cáo	10
Trần Đức Kiên	20215071	Khởi tạo dữ liệu cho bản đồ, các đoạn đường 1 chiều và 2 chiều, gắn tọa độ cho các điểm trên bản đồ Làm slide	10
Khúc Duy Hòa	20210357	Triển khai thuật toán A* Viết báo cáo về thuật toán A*	10
Lê Bá Trọng	20215153	Triển khai thuật toán Dijkstra Viết báo cáo về thuật toán Dijkstra	10
Trần Mạnh Toàn	20215149	Triển khai thuật toán Bellman-Ford Viết báo cáo về thuật toán Bellman-Ford	10

III. Đề tài

1. Giới thiệu bài toán

Cho bản đồ phường Trúc Bạch, tìm và biểu diễn đường đi từ một vị trí xuất phát đến điểm đích trên bản đồ. Bản đồ sẽ gồm các đường phố, ngõ hẻm, các địa điểm quan trọng khác trong phường.

Đường phố cũng được phân loại thành các loại đường: đường có dải ngăn cách giữa và không có ngăn cách, đường một chiều và đường hai chiều.



2. Biểu diễn không gian bài toán

Không gian bài toán: $N = \{(x, y) \mid 0 \leq x \leq 980, 0 \leq y \leq 890, (x, y) \in F\}$

- Trạng thái ban đầu: $N_0 = \{(x_0, y_0) \mid (x_0, y_0) \in N\}$ với (x_0, y_0) là toạ độ vị trí xuất phát được chọn
- Đích $N_n = \{(x_n, y_n) \mid (x_n, y_n) \in N\}$ với (x_n, y_n) là toạ độ vị trí đích được chọn
- Gọi $K((x, y))$ là tập các cạnh kề của điểm (x, y) . Ta có:

$$K((x, y)) = \{(x_{k1}, y_{k1}), (x_{k2}, y_{k2}), \dots, (x_{kn}, y_{kn})\}$$

- Ta có tập biến đổi trạng thái:

$$A = \{(x, y) \rightarrow (x_{k1}, y_{k1}), (x, y) \rightarrow (x_{k2}, y_{k2}), \dots, (x, y) \rightarrow (x_{kn}, y_{kn})\}$$

3. Phương pháp tìm kiếm lời giải

Bước đầu, cần xác định toạ độ điểm xuất phát (A) và điểm đích (B). Sau đó, ta xây dựng đồ thị và tính toán hàm đánh giá dựa trên thông tin hệ thống đường trên bản đồ, xây dựng đồ thị biểu diễn các điểm và đường đi giữa chúng.

Mỗi điểm trên bản đồ sẽ tương ứng với một đỉnh trong đồ thị, và các đường nối giữa các điểm sẽ tương ứng với các cạnh trong đồ thị.

Với mỗi cạnh, tính toán độ dài (theo khoảng cách Euclid) và hàm đánh giá (gồm giá trị khoảng cách thực tế từ điểm xuất phát đến điểm hiện tại và giá trị ước tính từ điểm hiện tại tới điểm đích)

3.1 Áp dụng giải thuật A*

Khởi tạo tập đóng, tập mở để lưu trữ các điểm và ưu tiên dựa trên giá trị hàm $f(n)$ ước lượng chi phí từ A đến B thông qua:

- $g(n)$: chi phí thực tế để đi từ A đến điểm đang xét
- $h(n)$ là giá trị ước lượng của chi phí còn lại từ điểm đang xét đến B

$$f(n) = g(n) + h(n)$$

Ban đầu, đặt điểm xuất phát làm trạng thái ban đầu, đưa điểm xuất phát vào tập mở

Lặp cho đến khi tập mở trống hoặc tìm thấy đường đi tới B.

Lấy điểm M có giá trị hàm đánh giá nhỏ nhất từ tập mở.

- Xóa điểm M khỏi tập mở, thêm M vào tập đóng.
- Kiểm tra với mọi điểm N lân cận với M, xem N có trùng với điểm đích không:
 - Nếu đúng, tìm kiếm kết thúc và đưa ra lời giải.
 - Nếu sai, đồng thời N không nằm trong tập đóng, khi đó N được thêm vào tập mở và tính toán các giá trị hàm đánh giá và chi phí từ M.

Hàm heuristic:

$$h(n) = \sqrt{(x_n - x_k)^2 + (y_n - y_k)^2}$$

Trong đó

(x_k, y_k) là toạ độ điểm đang xét

(x_n, y_n) là toạ độ điểm kết thúc

Như vậy, ta có:

$$f(n) = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2} + \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} + \dots + \sqrt{(x_k - x_{k-1})^2 + (y_k - y_{k-1})^2} + \sqrt{(x_n - x_k)^2 + (y_n - y_k)^2}$$

3.2 Áp dụng giải thuật Dijkstra

Khởi tạo

- Tạo hai tập hợp: tập mở (open list) và tập đóng (closed list).
- Tập mở: Hàng đợi ưu tiên để lưu trữ các điểm cần xử lý. Độ ưu tiên dựa trên chi phí từ điểm xuất phát đến điểm hiện tại ($g(n)$).
- Tập đóng: để lưu trữ các điểm đã được xử lý.
- Ban đầu, thêm điểm xuất phát vào tập mở với chi phí $g(n) = 0$.

Duyệt các điểm

- Chọn điểm hiện tại bằng cách lấy điểm có chi phí $g(n)$ nhỏ nhất trong tập mở. Kiểm tra nếu điểm hiện tại đã xử lý thì bỏ qua. Nếu chưa thì thêm điểm hiện tại vào tập đóng.
- Kiểm tra nếu điểm hiện tại là điểm đích, xây dựng đường đi từ điểm đích về điểm xuất phát bằng cách theo dõi các điểm cha và trả về đường đi tìm được.
- Duyệt qua các điểm kè của điểm hiện tại. Nếu chi phí mới nhỏ hơn chi phí hiện tại, cập nhật chi phí và điểm cha, sau đó thêm điểm kè vào tập mở.

Ví dụ điểm hiện tại là M, xét điểm N là điểm kè của M ta có:

$$g(N) = \min(g(N), g(M) + d(M,N))$$
, Trong đó:

- $g(N)$ là chi phí từ điểm xuất phát đến N
- $g(M)$ là chi phí từ điểm xuất phát đến M
- $d(M,N)$ là chi phí đi từ điểm M đến N

Chương trình sẽ trả về null nếu không tìm được đường đi.

3.3 Áp dụng giải thuật Bellman-Ford

Khởi tạo tất cả các điểm trên bản đồ với khoảng cách từ điểm xuất phát là vô cực (INFINITY), ngoại trừ điểm xuất phát có khoảng cách là 0. Tạo một tập hợp các điểm và các cạnh (đường) nối giữa chúng dựa trên thông tin hệ thống đường trên bản đồ. Đặt điểm xuất phát làm trạng thái ban đầu với khoảng cách bằng 0.

Lặp lại $|V| - 1$ lần (với $|V|$ là số lượng điểm trên bản đồ). Trong mỗi lần lặp, thực hiện nói lồng tất cả các cạnh. Đối với mỗi cạnh từ điểm u đến điểm v có độ dài w, nếu khoảng cách hiện tại đến u cộng với độ dài w nhỏ hơn khoảng cách hiện tại đến v, cập nhật khoảng cách và lưu điểm cha của v là u.

Sau khi thực hiện nói lồng tất cả các cạnh $|V| - 1$ lần, kiểm tra chu trình trọng số âm bằng cách lặp lại tất cả các cạnh một lần nữa. Nếu có cạnh nào có thể nói lồng thêm, điều đó có nghĩa là đồ thị có chứa chu trình trọng số âm, báo lỗi và kết thúc giải thuật.

Nếu không có chu trình trọng số âm, bắt đầu từ điểm đích, truy ngược lại qua các điểm cha để tìm đường đi từ điểm xuất phát đến điểm đích. Lưu đường đi này trong một danh sách và trả về kết quả, bao gồm danh sách các cạnh tạo nên đường đi ngắn nhất, danh sách các điểm đã mở trong quá trình tìm đường, và danh sách các điểm đã đóng.

3.4 So sánh 3 giải thuật

Yếu tố	Dijkstra	Bellman-Ford	A*
Tính hoàn chỉnh	Có, nếu đồ thị không có chu trình trọng số âm.	Có, ngay cả khi có trọng số âm.	Có, nếu hàm heuristic là chấp nhận được và không đánh giá thấp.
Độ phức tạp thời gian	$O((V+E)\log V)$ với hàng đợi ưu tiên tốt (như Fibonacci heap).	$O(V \cdot E)$	$O((V+E)\log V)$ với hàng đợi ưu tiên tốt, phụ thuộc vào chất lượng của hàm

			heuristic.
Độ phức tạp bộ nhớ	O(V) (lưu trữ các khoảng cách và điểm cha).	O(V) (lưu trữ các khoảng cách và điểm cha và giá trị heuristic).	O(V) (lưu trữ các khoảng cách, điểm cha và giá trị heuristic).
Tính tối ưu	Có, nếu không có trọng số âm.	Có, ngay cả khi có trọng số âm.	Có, nếu hàm heuristic là chấp nhận được và không đánh giá thấp.
Sự khác biệt trong việc tìm đường đi	<ul style="list-style-type: none"> - Tìm đường đi ngắn nhất trong đồ thị với các cạnh có trọng số không âm.
 - Hiệu quả cho đồ thị không trọng số âm. 	<ul style="list-style-type: none"> - Tìm đường đi ngắn nhất ngay cả khi có cạnh có trọng số âm. - Có thể phát hiện chu trình trọng số âm. 	<ul style="list-style-type: none"> - Tìm đường đi ngắn nhất sử dụng hàm heuristic để tăng tốc tìm kiếm. - Hiệu quả nhất khi hàm heuristic tốt, dẫn đến thời gian tìm kiếm nhanh hơn so với Dijkstra.

4. Cài đặt chương trình

Các cặp tọa độ (x, y) định nghĩa các đoạn đường hai chiều và một chiều trên bản đồ

```

TWO_WAY_ROADS = [
    ((411, 172), (374, 141)),
    ((673, 448), (607, 468)),
    ((441, 458), (443, 509)),
    ((673, 448), (637, 608)),
    ((753, 434), (726, 535)),
    ((753, 434), (762, 426)),
    ((762, 426), (773, 416)),
    ((623, 676), (607, 740)),
    ((566, 328), (551, 303)),
    ((735, 874), (769, 885)),
    ((673, 762), (665, 800)),
    ((278, 789), (283, 756)),
    ((276, 818), (278, 789)),
    ((597, 778), (607, 740)),
    ((726, 535), (673, 448)),
]
ONE_WAY_ROADS = [
    ((35, 825), (30, 808)),
    ((30, 808), (35, 779)),
    ((35, 779), (52, 743)),
    ((52, 743), (69, 694)),
    ((69, 694), (89, 542)),
    ((89, 542), (107, 418)),
    ((107, 418), (155, 359)),
    ((155, 359), (226, 288)),
    ((226, 288), (276, 223)),
    ((276, 223), (319, 145)),
    ((319, 145), (345, 95)),
    ((345, 95), (360, 79)),
    ((360, 79), (393, 54)),
    ((35, 825), (30, 808)),
    ((30, 808), (25, 779))
]

```

Lớp Point khởi tạo các đối tượng “điểm” lưu vị trí tọa độ (x, y) và các điểm lân cận, các phương thức tính khoảng cách

```

class Point:
    """Class used for point objects"""
    def __init__(self, pos, pos2=None):
        #receive pos as (x,y)
        if pos2 is None:
            from road import Road
            self.pos = pos
            self.x, self.y = pos
            self.adjacents: list[tuple[Road, Point]] = []
        else:
            self.pos = (pos, pos2)
            self.x = pos
            self.y = pos2
            self.adjacents = []

    def _calc_dist(self, pos: tuple):
        """Calculate distance from this position to another position

        Parameters:
        :param pos(Tuple[int, int]): A tuple containing the (x,y) coordinates of the 'pos' position
        """
        return math.dist(self.pos, pos)

```

Lớp Road thể hiện cho các đoạn đường một chiều. Ngoài ra còn có phương thức tính khoảng cách từ một vị trí tọa độ đến đoạn đường đi, xác định tọa độ hình chiếu từ một điểm lên đoạn

```

class Road:
    """Class used for one-way roads. For two-way road, see `TwoWayRoad` class
    def __init__(self, from_point: Point, to_point: Point):
        self.from_point = from_point
        self.to_point = to_point
        self.from_pos = from_point.pos
        self.to_pos = to_point.pos

        self.length = math.dist(from_point.pos, to_point.pos)
        self.from_point.adjacents.append((self, to_point))

    def _calc_dist(self, pos: tuple):
        """Calculate distance from a position to the road

        Parameters:
        :param pos(Tuple[int, int]): A tuple containing the (x,y) coordinates
        """
        p1 = np.asarray(self.from_pos)
        p2 = np.asarray(self.to_pos)
        p3 = np.asarray(pos)

        return np.abs(np.cross(p2 - p1, p1 - p3)) / norm(p2 - p1)

    def _is_look(self, pos: tuple) -> bool:
        """Check if this road look to the 'pos' position

        Parameters:
        :param pos(Tuple[int, int]): A tuple containing the (x,y) coordinates
        """
        x, y = self._perpendicular_pos(pos)
        x1, y1 = self.from_pos
        x2, y2 = self.to_pos

```

Cài đặt thuật toán A*:

```

def find_path(self, start_pos: tuple, end_pos: tuple):
    process_point: dict[str, ProcessPoint] = {}
    for key in self.map_points:
        process_point[key] = ProcessPoint(self.map_points[key])

    if self.map_points[str(start_pos)] is None:
        process_point[str(start_pos)] = ProcessPoint(start_pos)

    if self.map_points[str(end_pos)] is None:
        process_point[str(end_pos)] = ProcessPoint(end_pos)

    open_lst = {str(start_pos)}
    closed_lst = set()
    process_point[str(start_pos)].g = 0

    found = False

    while len(open_lst) > 0:
        # find node with the least f on the open list -> q
        minf = min([process_point[key].f for key in open_lst])
        q = None
        for key in open_lst:
            if minf == process_point[key].f:
                q = key
                break

        # pop q from open list
        open_lst.remove(q)
        closed_lst.add(q)
        # generate q's successors and set their parent to q
        current_point = process_point[q]

        for road, to_point in current_point.point.adjacents:

            # if successor is the goal, stop searching and output
            process_to_point = process_point[str(to_point)]
            current_f = process_to_point.f
            if process_to_point.pos == end_pos:
                # FOUND
                process_to_point.parent = current_point
                stack: list[Road] = []
                child_node = process_to_point
                parent_node = child_node.parent
                while parent_node is not None:
                    for road_, point in parent_node.point.adjacents:
                        if str(point) == str(child_node):
                            stack.insert(0, road_)
                            break
                    child_node = parent_node
                    parent_node = parent_node.parent

                found = True
                return (stack,
                        [process_point[key].point for key in open_lst],
                        [process_point[key].point for key in closed_lst])

            elif not str(to_point) in closed_lst:
                g_new = current_point.g + road.length
                h_new = process_to_point._calc_dist(end_pos)
                f_new = g_new + h_new

                if str(to_point) not in open_lst or current_f > f_new:
                    open_lst.add(str(to_point))

```

Cài đặt thuật toán Dijkstra

```

def find_path_dijicktra(self, start_pos: tuple, end_pos: tuple):
    import heapq
    process_point: dict[str, ProcessPoint] = {}
    for key in self.map_points:
        process_point[key] = ProcessPoint(self.map_points[key])

    if self.map_points.get(str(start_pos)) is None:
        process_point[str(start_pos)] = ProcessPoint(start_pos)

    if self.map_points.get(str(end_pos)) is None:
        process_point[str(end_pos)] = ProcessPoint(end_pos)
    open_lst = []
    closed_lst = set()
    process_point[str(start_pos)].g = 0
    heapq.heappush(open_lst, (0, str(start_pos)))
    while open_lst:
        current_g, q = heapq.heappop(open_lst)
        if q in closed_lst:
            continue
        current_point = process_point[q]
        closed_lst.add(q)

        if current_point.pos == end_pos:
            # Found the destination
            stack = []
            child_node = current_point
            parent_node = child_node.parent

            while parent_node is not None:
                for road, point in parent_node.point.adjacents:
                    if str(point) == str(child_node):
                        stack.insert(0, road)
                        break
                child_node = parent_node
                parent_node = child_node.parent

            return (stack,
                    [process_point[key].point for _, key in open_lst],
                    [process_point[key].point for key in closed_lst])

        for road, to_point in current_point.point.adjacents:
            process_to_point = process_point[str(to_point)]
            if str(to_point) not in closed_lst:
                g_new = current_g + road.length

                if g_new < process_to_point.g:
                    process_to_point.g = g_new
                    process_to_point.parent = current_point
                    heapq.heappush(open_lst, (g_new, str(to_point)))

    # If the loop completes without finding the end_pos
    return (None, [process_point[key].point for _, key in open_lst],
            [process_point[key].point for key in closed_lst])

```

Cài đặt thuật toán Bellman-Ford

```

def find_path_Bellmanford(self, start_pos: tuple, end_pos: tuple):
    process_point: dict[str, ProcessPoint] = {}
    for key in self.map_points:
        process_point[key] = ProcessPoint(self.map_points[key])

    if self.map_points.get(str(start_pos)) is None:
        process_point[str(start_pos)] = ProcessPoint(start_pos)

    if self.map_points.get(str(end_pos)) is None:
        process_point[str(end_pos)] = ProcessPoint(end_pos)
    process_point[str(start_pos)].g = 0
    # Number of vertices
    num_vertices = len(process_point)
    # Relax all edges num_vertices - 1 times
    for _ in range(num_vertices - 1):
        for key, current_point in process_point.items():
            for road, to_point in current_point.point.adjacents:
                process_to_point = process_point[str(to_point)]
                g_new = current_point.g + road.length
                if g_new < process_to_point.g:
                    process_to_point.g = g_new
                    process_to_point.parent = current_point
    # Check for negative weight cycles
    for key, current_point in process_point.items():
        for road, to_point in current_point.point.adjacents:
            process_to_point = process_point[str(to_point)]
            if current_point.g + road.length < process_to_point.g:
                raise ValueError("Graph contains a negative weight cycle")

    # Build the path and collect open and closed points
    stack = []
    closed_lst = set()
    current_point = process_point[str(end_pos)]
    parent_node = current_point.parent

    if parent_node is not None or start_pos == end_pos:
        while parent_node is not None:
            for road, point in parent_node.point.adjacents:
                if str(point) == str(current_point):
                    (variable) closed_lst: set
                    closed_lst.add(str(current_point))
                    current_point = parent_node
                    parent_node = parent_node.parent

            closed_lst.add(str(current_point))

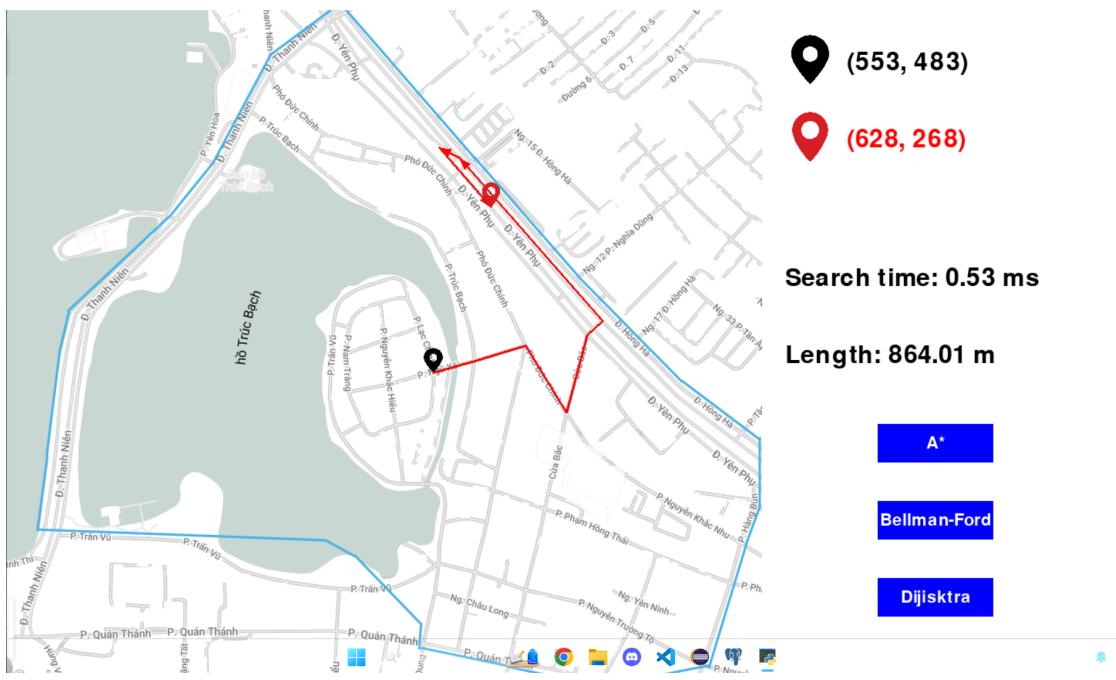
            open_lst = [process_point[key].point for key in process_point if key not in closed_lst]

        return (stack, open_lst, [process_point[key].point for key in closed_lst])
    else:
        return (None, [process_point[key].point for key in process_point if key != str(start_pos)],
                [process_point[key].point for key in closed_lst])

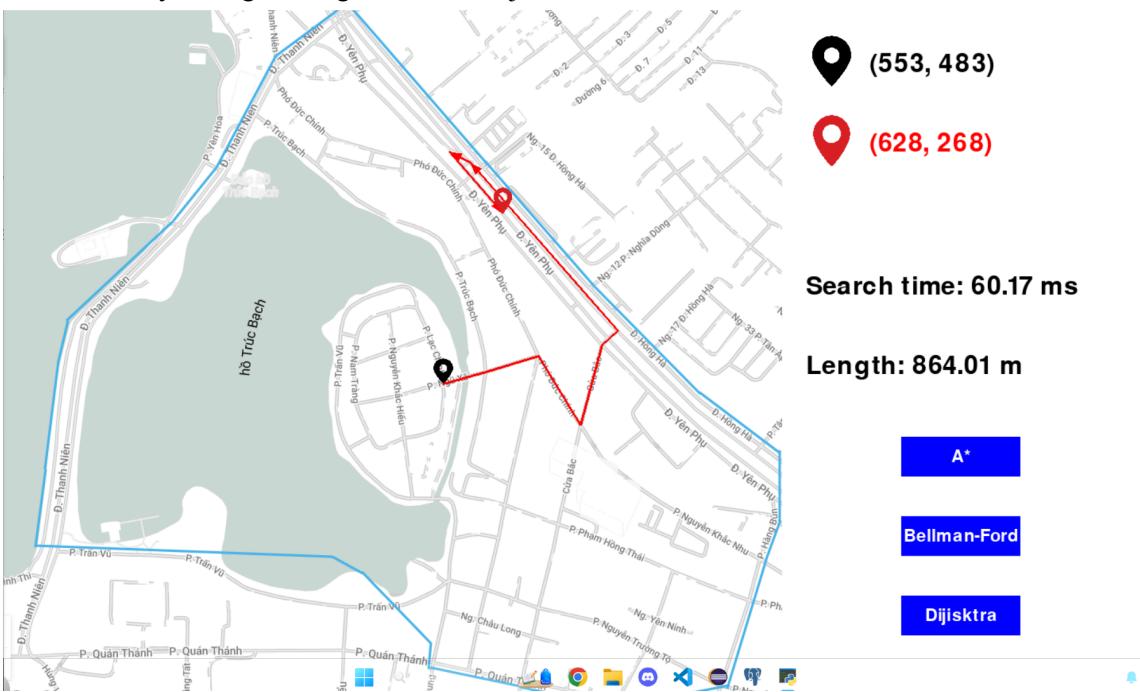
```

5. Kết quả

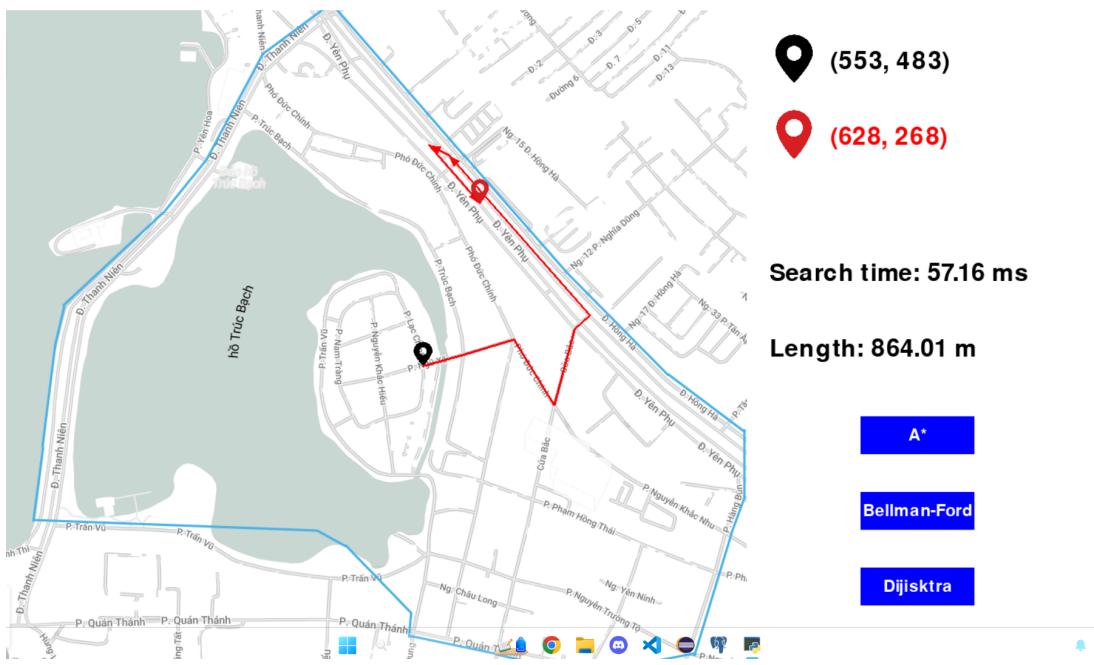
Tìm thấy đường đi bằng thuật toán A*:



Tìm thấy đường đi bằng thuật toán Dijkstra:



Tìm thấy đường đi bằng thuật toán Bellman-Ford:



Không tìm thấy đường đi:

