KNO-1301-5103

مسیریاب چند مقصدی: روشی برای محاسبه کوتاه‌ترین مسیر برای گذر یک‌باره از چندین نقطه بدون بازگشت به مبدا

محمدرضا سلمانی [۱] m.salmani@gmail.com

علیرضا محمودی‌فرد *[۲] alireza10.m10@gmail.com

[۱] دانشجوی کاردانی نرم‌افزار دانشگاه ملی مهارت، دانشکده فنی انقلاب اسلامی، تهران، ایران

[۲] پسادکترای حرفه‌ای آینده‌پژوهی و استاد مدعو دانشگاه ملی مهارت، دانشکده فنی انقلاب اسلامی، تهران، ایران

چکیده

یافتن کوتاه‌ترین مسیر ممکن برای گذر از چندین نقطه به شکلی که تنها یک‌بار از هر نقطه عبور کند، چالشی کلیدی در سیستم‌های حمل‌ونقل هوشمند و لجستیک شهری است که می‌تواند کاربردهای زیادی در حوزه‌های مختلف از زندگی روزمره گرفته تا سرویس‌های تحویل آنلاین کالا داشته باشد. روش‌های موجود مانند مسئله فروشنده دوره‌گرد (TSP) با وجود کارایی نظری، به‌دلیل الزام بازگشت به مبدا و عدم پشتیبانی از داده‌های پویا، کارایی محدودی در کاربردهای واقعی دارند. این مقاله الگوریتم مسیریاب چند مقصدی (MDR) را ارائه می‌کند که با استفاده از داده‌های زنده OpenStreetMap و ترکیب آن با بهینه‌سازی مبتنی بر گراف‌ها، این مسئله را به صورت کاربردی و با دقت صددرصدی حل کرده است. آزمایش‌های انجام‌شده روی داده‌های واقعی در کلان‌شهر تهران نشان‌دهنده کاهش ۱۶ درصدی مسافت و ۷ درصدی زمان محاسبات نسبت به روش‌های پایه است. این الگوریتم، به‌ویژه در زمینه سرویس‌های تحویل مدرن و برنامه‌ریزی شهری، گزینه‌ای عملی و کارآمد به شمار می‌آید.

کلید واژه‌ها: *مسیریاب چندمقصدی، الگوریتم MDR، OpenStreetMap، مسئله فروشنده دوره‌گرد (TSP)، سرویس‌های تحویل هوشمند*

۱ .مقدمه

در دنیای امروز، مسیریابی بهینه برای سیستم‌های حمل‌ونقل و لجستیک شهری یکی از مسائل کلیدی در بهبود کارایی و کاهش هزینه‌ها است. از آنجا که تعداد روزافزونی از داده‌های مختلف برای مسیریابی در دسترس قرار دارند، استفاده از این داده‌ها برای طراحی الگوریتم‌های بهینه اهمیت زیادی پیدا کرده است. در این راستا، مسائل مربوط به مسیریابی از جمله یافتن کوتاه‌ترین مسیر برای عبور از چندین نقطه، به‌طور خاص در زمینه‌های حمل‌ونقل، تحویل کالا و برنامه‌ریزی شهری از اهمیت ویژه‌ای برخوردار است [1].

یکی از مسائل کلاسیک در این زمینه، مسئله فروشنده دوره‌گرد (TSP) است که در آن هدف یافتن کوتاه‌ترین مسیری است که تمامی نقاط را بازدید کرده و در نهایت به نقطه شروع بازگردد [۲]. این مسئله به‌دلیل پیچیدگی بالا و زمان محاسباتی زیاد، به‌ویژه در مسائل واقعی که داده‌های پویا و نیازهای به‌روز در مسیر وجود دارند، مناسب نیست؛ علاوه‌بر این، در بسیاری از کاربردهای عملی مانند تحویل کالا یا مسیریابی در شبکه‌های شهری، بازگشت به نقطه آغازین نه‌تنها ضروری نیست، بلکه به‌طور عمده عملاً غیرضروری است [۳]. این ویژگی یکی از دلایلی است که الگوریتم MDR توسعه داده شده است، زیرا الگوریتم‌های سنتی همچون TSP برای این نوع کاربردها با محدودیت‌های زیادی روبرو هستند.

۲. توصیف

در این مقاله، الگوریتم *مسیریاب چند مقصدی* (MDR) معرفی شده‌است که با استفاده از داده‌های زنده OpenStreetMap و بهینه‌سازی مبتنی‌بر گراف‌ها، مسئله مسیریابی چند مقصدی را به‌شکلی کاربردی و با دقت صددرصدی حل می‌کند. این الگوریتم با حذف الزام بازگشت به مبدا، در مقایسه با روش‌های سنتی همچون TSP، زمان محاسباتی را به‌طور قابل توجهی کاهش می‌دهد و در مسائل حمل‌ونقل و سرویس‌های تحویل آنلاین کارایی بیشتری دارد.

ساختار مقاله به این صورت است: ابتدا در بخش ۳، روش‌های موجود و پیشینه تحقیق در زمینه مسیریابی چند مقصدی بررسی می‌شود؛ در بخش ۴، الگوریتم MDR معرفی شده و مراحل آن شرح داده می‌شود؛ سپس در بخش ۵، نتایج آزمایش‌های انجام شده و مقایسه‌های مربوطه با روش‌های موجود آورده خواهد شد؛ در پایان، بخش در بخش ۶ به محدودیت‌ها و کارهای آینده مروبط به این الگوریتم برسی می‌شود و در نهایت در بخش ۷ به نتیجه‌گیری اختصاص خواهد یافت.

۳. روش‌های موجود و پیشینه تحقیق

در این بخش، به مرور روش‌های مختلفی که در زمینه مسیریابی چند مقصدی و مشابه آن مانند مسئله فروشنده دوره‌گرد (TSP) مطرح شده‌اند، پرداخته می‌شود؛ بسیاری از این روش‌ها سعی در حل مسائل مسیریابی بهینه در شرایط مختلف دارند، اما هرکدام محدودیت‌های خاص خود را دارند.

۳,۱ مسئله فروشنده دوره‌گرد (TSP)

مسئله فروشنده دوره‌گرد (TSP) یکی از مهم‌ترین مسائل در زمینه مسیریابی است که هدف آن یافتن کوتاه‌ترین مسیر برای بازدید از تمامی نقاط و در نهایت بازگشت به نقطه شروع است [۲]. این مسئله از لحاظ پیچیدگی زمانی، به‌ویژه برای مقیاس‌های بزرگ، بسیار چالش‌برانگیز است و معمولاً برای حل آن از الگوریتم‌های تقریبی استفاده می‌شود. در دهه‌های اخیر، روش‌هایی مانند الگوریتم‌های فرابتکاری (مانند الگوریتم‌های ژنتیک و شبیه‌سازی تبرید) به‌طور گسترده برای حل TSP استفاده شده‌اند، که این روش‌ها در مسائل کوچکتر موثر هستند، اما همچنان با چالش‌های زیادی در مقیاس‌های بزرگ روبرو هستند [3][4].

از سوی دیگر، در مسائل واقعی مسیریابی، الزام به بازگشت به مبدا، که بخش اصلی مسئله TSP است، معمولاً وجود ندارد؛ در واقع، بسیاری از کاربردهای واقعی مانند سرویس‌های تحویل کالا به‌ویژه در حوزه‌های شهری، نیازی به بازگشت به نقطه شروع ندارند؛ این یکی از دلایلی است که چرا الگوریتم‌های TSP نمی‌توانند به‌طور کامل در این حوزه‌ها کارآمد باشند.

۳,۲ پیچیدگی مسیریابی مبتنی‌بر داده‌های زنده

مسیریابی در نقشه‌های واقعی به‌دلیل وجود وزن‌های متفاوت برای مسیرهای مختلف بین نودها پیچیده‌تر از مسائل ساده مسیریابی است. این وزن‌ها می‌توانند بسته به شرایط مختلفی نظیر وضعیت ترافیک، نوع جاده، زمان روز، یا حتی شرایط آب و هوایی تغییر کنند. این ویژگی‌ها باعث می‌شوند که وزن مسیرها در مسائل واقعی ثابت نباشد و به‌طور مداوم تغییر کند [۵]. به همین دلیل، الگوریتم‌های کلاسیک مانند TSP، که فرض می‌کنند وزن‌ها ثابت و همگن هستند، نمی‌توانند در مسائل واقعی مسیریابی با کارایی مطلوب عمل کنند. در این الگوریتم‌ها، تمام مسیرها باید یکسان فرض شوند و به همین دلیل، کاربرد آن‌ها در مسیریابی واقعی با چالش‌هایی مواجه است.

برای مسیریابی در نقشه‌های واقعی، الگوریتم‌هایی مانند دیکسترا برای یافتن کوتاه‌ترین مسیر بین دو نقطه به‌طور گسترده استفاده می‌شود؛ این الگوریتم در گراف‌های بدون وزن منفی به‌خوبی عمل می‌کند و یکی از پرکاربردترین الگوریتم‌ها در مسائل مسیریابی است [۶]؛ با این حال، در شرایطی که نیاز به مسیریابی از یک نقطه به چندین مقصد وجود داشته باشد، دیکسترا به‌طور مستقیم قادر به حل مسأله بهینه نمی‌باشد؛ در این موارد، بعضاً از دیکسترا به‌عنوان بخشی از یک روش توسعه‌یافته استفاده می‌شود که در آن کمترین فاصله از مبدا به هر مقصد به‌طور جداگانه محاسبه می‌شود؛ این رویکرد ممکن است در برخی موارد نتایج دقیقی ندهد، زیرا ممکن است مسیرهای پیچیده‌تر و روابط غیرمستقیم بین نقاط به‌درستی محاسبه نشوند و مسیر بهینه را نیابد [7].

۳,۳ مسیریابی چند مقصدی بدون بازگشت به مبدا

در مسیریابی چند مقصدی که نیاز به بازگشت به مبدا ندارد، چالش‌های متفاوتی وجود دارد. یکی از مهم‌ترین مسائلی که در این نوع مسیریابی باید در نظر گرفته شود، بهینه‌سازی زمان و مسافت در شرایطی است که تغییرات جغرافیایی و شهری به‌طور مستمر رخ می‌دهند. الگوریتم‌هایی که به‌طور خاص برای حل این نوع از مسائل طراحی شده‌اند، معمولاً از روش‌های جدیدتر بهینه‌سازی مانند بهینه‌سازی مسیر پویا و الگوریتم‌های بهینه‌سازی چندهدفه استفاده می‌کنند [8].

با پیشرفت فناوری و دسترسی به داده‌های زنده، الگوریتم‌های جدیدتری برای مسیریابی معرفی شده‌اند که تلاش می‌کنند محدودیت‌های داده‌های پویا و تغییرات در زمان واقعی را در نظر بگیرند. به‌عنوان مثال، در استفاده از داده‌های OpenStreetMap (OSM) برای مسیریابی، قابلیت به‌روزرسانی مسیرها به‌صورت لحظه‌ای و پاسخ به تغییرات محیطی اهمیت دارد [۵]. مطالعات مختلف نشان داده‌اند که ترکیب این داده‌ها با الگوریتم‌های بهینه‌سازی مبتنی‌بر گراف‌ها می‌تواند راهکارهایی مناسب برای حل مسائل مسیریابی در دنیای واقعی ارائه دهد [6].

۳,۴ محدودیت‌ها و چالش‌های موجود

با اینکه الگوریتم‌های موجود برای مسیریابی چند مقصدی به‌طور کلی تلاش کرده‌اند تا مسائل مختلف را حل کنند، همچنان در زمینه‌های مختلفی مانند دقت، زمان محاسباتی و امکان استفاده در مقیاس‌های بزرگ محدودیت‌هایی وجود دارد. الگوریتم‌هایی که به‌ویژه بر داده‌های زنده تکیه می‌کنند، نیازمند پردازش سریع و دقیق اطلاعات هستند، که این خود چالش‌های جدیدی را به‌همراه دارد؛ علاوه‌بر این، پیچیدگی مسیریابی در دنیای واقعی به‌ویژه در شرایطی که مسیرها پویا هستند و تغییرات محیطی مداوم رخ می‌دهند، سبب می‌شود که الگوریتم‌های کلاسیک نظیر TSP به‌خوبی در این زمینه عمل نکنند.

۴. الگوریتم مسیریاب چند مقصدی (MDR)

الگوریتم MDR (Multi Destination Router) برای محاسبه کوتاه‌ترین مسیر ممکن برای عبور از چندین نقطه بهطوری‌که از تمامی نقاط یکبار عبور کند طراحی شده است. این الگوریتم مشابه مسئله TSP(مسئله فروشنده دوره‌گرد) است، با این تفاوت که الزام به

بازگشت به مبدأ وجود ندارد. در این بخش، از داده‌های نقشه OpenStreetMap و ماژول‌های osmnx و networkx برای مدل‌سازی نقشه و محاسبه مسیرها استفاده شده‌است.

۴,۱ تعریف کلاس MDR (شکل ۱)

در ابتدا یک کلاس به‌نام MDR تعریف شده‌است که وظیفه پردازش داده‌های نقشه و اعمال الگوریتم‌ها روی آن‌ها را بر عهده دارد.

در این بخش، اطلاعات کلی از نقشه مشخص می‌شود. نوع شبکه مسیریابی به‌طور پیش‌فرض شامل تمامی مسیرهای ممکن است که می‌توان آن را برای مسیرهای خودرو، دوچرخه و پیاده‌روی تغییر داد. همچنین یک حاشیه برای نقشه در نظر گرفته‌شده است تا تمامی مسیرهای ممکن را شامل شود.

همچنین در این بخش از سیستم کش‌گذاری برای ذخیره‌ی داده‌ها برای افزایش سرعت استفاده شده است.

```python
class MDR:
    # Enable OSMnx cache to avoid redundant downloads
    ox.settings.use_cache = True

    def __init__(self, street_type='all', buffer=0.005):
        # Stores the OpenStreetMap-based graph
        self.graph_map = None
        # Stores user-defined destination coordinates
        self.nodes = []
        # Specifies the type of street network to retrieve (e.g., 'all', 'drive', etc.)
        self.street_type = street_type
        # Buffer (in degrees) to expand around destination points for graph coverage
        self.buffer = buffer
```

شکل ۱ – تعریف کلاس MDR

۴,۲ اضافه کردن و حذف گره‌ها

```python
# Adds coordinates to the list of destination nodes and resets the graph
    def add_nodes(self, nodes):
        self.nodes.extend(nodes)
        self.graph_map = None
        return self.nodes
```

شکل ۲ – تابع add_nodes برای اضافه کردن نقاط

در کلاس MDR توابعی برای افزودن (شکل ۲) و حذف گره‌ها (شکل ۳) از لیست self.nodes تعریف شده است.

این توابع اجازه می‌دهند که گره‌های خاصی را به گراف اضافه یا از آن حذف شود.

۴,۳ یافتن نزدیک‌ترین گره

```python
# Removes specified coordinates from the list of destination nodes
    def remove_nodes(self, nodes):
        self.nodes = [node for node in self.nodes if node not in nodes]
}
```

شکل ۳ - تابع remove_nodes برای حذف کردن نقاط

```python
# Finds the nearest node in the OSM graph to a given coordinate
    def find_nearest_node(self, node):
        return ox.nearest_nodes(G=self.graph_map, X=node[1], Y=node[0])
```

شکل ۴ - تابع find_nearest_node برای پیدا کردن نزدیک‌ترین نقطه به نقطه‌ی موردنظر روی گراف

از آنجایی که ممکن است نقطه‌ی ارسال شده دقیقا یک نقطه‌ی جغرافیایی در مسیرهای شهری نباشد، یک تابع برای پیدا کردن نزدیک‌ترین نقطه به آن تعریف شده‌است به‌نام find_nearest_node (شکل ۴).

این تابع از داده‌های OpenStreetMap برای محاسبه نزدیک‌ترین گره به مختصات جغرافیایی داده‌شده استفاده می‌کند.

۴,۴ ساخت گراف از نقشه

تابع create_graph_from_nodes برای ایجاد یک گراف شامل تمام نقاط مورد نظر است، استفاده می‌شود (شکل ۵).

```
# Creates a graph from the bounding box around the destination nodes
    def create_graph_from_nodes(self):
        lats = [pt[0] for pt in self.nodes]
        lons = [pt[1] for pt in self.nodes]

        # Calculate bounding box limits with buffer
        north = max(lats) + self.buffer
        south = min(lats) - self.buffer
        east = max(lons) + self.buffer
        west = min(lons) - self.buffer

        # Download the street network within the bounding box
        self.graph_map = ox.graph_from_bbox(north, south, east, west, network_type=self.street_type)
        # Add estimated speed and travel time attributes to graph edges
        self.graph_map = ox.add_edge_speeds(G=self.graph_map, fallback=30)
        self.graph_map = ox.add_edge_travel_times(G=self.graph_map)
        return self.graph_map
```

شکل ۵ – تابع create_graph_from_nodes برای ساخت یک گراف از نقشه‌ی با استفاده از داده‌های واقعی OSM

این تابع شمالی‌ترین، جنوبی‌ترین، شرقی‌ترین و غربی‌ترین نقاط مورد نظر را استخراج می‌کند و سپس با استفاده از داده‌های OpenStreetMap یک گراف کامل از محدوده (Bounding Box) با اعمال یک حاشیه (buffer) می‌سازد که شامل تمامی گره‌ها و یال‌ها(سرعت و زمان) است.

۴,۵ ساخت گراف گره‌ها

تابع graph_nodes یک گراف جدید از گره‌هایی که در self.nodes قرار دارند می‌سازد تا داده‌هایی که نیاز به آن‌ها پردازش آن‌ها نیست، دیگر استفاده نشوند و سرعت پردازش آن بیشتر شود (شکل ۶).

```
# Builds a complete weighted graph of destination nodes based on shortest OSM paths
    def graph_nodes(self):
        if not self.graph_map:
            self.graph_map = self.create_graph_from_nodes()
        graph_node = nx.Graph()
        nearest_nodes = {node: self.find_nearest_node(node) for node in self.nodes}

        for start, node_start in nearest_nodes.items():
            for end, node_end in nearest_nodes.items():
                if node_end != node_start:
                    length = nx.shortest_path_length(self.graph_map, source=node_start, target=node_end, weight='length')
                    time = nx.shortest_path_length(self.graph_map, source=node_start, target=node_end, weight='travel_time')
                    graph_node.add_edge(start, end, travel_time=time, length=length)

        return graph_node
```

شکل ۶ – تابع graph_nodes برای ساخت یک گراف فقط از نقاط مورد نظر

این تابع گراف جدیدی با گره‌های خاص که در self.nodes تعریف شده‌اند می‌سازد. برای هر جفت گره، طول مسیر و زمان سفر محاسبه می‌شود و به گراف افزوده می‌شود (شکل ۷).

۴,۶ پیدا کردن کوتاه‌ترین مسیر

تابع shortest_path برای یافتن کوتاه‌ترین مسیر از گره شروع به گره‌های دیگر استفاده می‌شود. در اینجا، این تابع به دنبال مسیرهایی می‌گردد که تمامی گره‌های موجود در self.nodes را از گره شروع عبور دهد.

```python
# Computes the shortest visiting sequence of nodes starting from a given point
    def shortest_path(self, start, weight='length'):
        shortest_length = float('inf')
        shortest_path = None
        graph = self.graph_nodes()

        for node in self.nodes:
            if node != start:
                for path in nx.all_simple_paths(graph, source=start, target=node):
                    if len(path) == len(graph.nodes):
                        weight_path = sum(
                            graph[path[i]][path[i + 1]][weight] for i in range(len(path) - 1)
                        )
                        if weight_path < shortest_length:
                            shortest_length = weight_path
                            shortest_path = path

        return shortest_path
```

شکل ۷ – تابع shortest_path برای پیدا کردن کوتاه‌ترین مسیر ممکن

- ابتدا متغیرهایی به‌نام‌های shortest_length و shortest_path تعریف می‌شود که به‌ترتیب برای ذخیره طول کوتاه‌ترین مسیر و خود مسیر استفاده می‌شوند.
- گراف از توابع قبلی ساخته می‌شود و سپس برای هر گره (غیر از گره شروع) تمامی مسیرهای ساده از گره شروع به آن گره بررسی می‌شود.
- برای هر مسیر ساده، جمع وزن‌ها (مسافت یا زمان) محاسبه می‌شود.
- در نهایت، اگر وزن مسیر پیدا شده کمتر از وزن قبلی باشد، مسیر و طول آن به‌روزرسانی می‌شود.

۴،۷ یافتن مسیر روی نقشه

پس از یافتن کوتاه‌ترین مسیر، تابع route_on_map مسیر به‌دست آمده را روی نقشه نمایش می‌دهد(شکل ۸).

```python
# Returns detailed OSM routes between ordered destination nodes
    def route_on_map(self, path, weight='length'):
        path = [self.find_nearest_node(node) for node in path]
        return [nx.shortest_path(self.graph_map, source=path[i], target=path[i + 1], weight=weight) for i in range(len(path) - 1)]
```

شکل ۸ – تابع route_on_map برای پیدا کردن مسیر مورد نظر روی نقشه

این تابع مسیر پیدا شده را به صورت پیکربندی شده به‌شکل یک لیست از مسیرها برمی‌گرداند که می‌توان آن را روی نقشه نمایشی مشاهده کرد.

خلاصه الگوریتم:

الگوریتم MDR مسیر کوتاه‌تری را برای عبور از چندین مقصد مشخص می‌کند. ابتدا گرافی از داده‌های OpenStreetMap ساخته می‌شود و سپس مسیرهای کوتاه میان گره‌ها محاسبه می‌شود. در اینجا تابع shortest_path با استفاده از تمامی مسیرهای ساده و مقایسه وزن آن‌ها، کوتاه‌ترین مسیر را پیدا می‌کند.

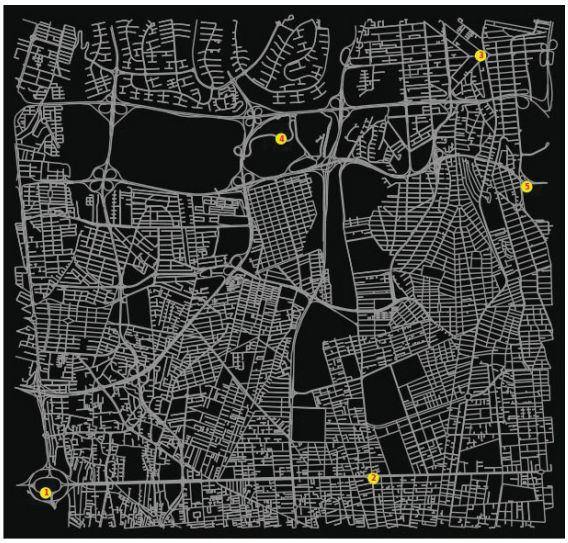
۵. آزمایش‌ها و نتایج

در این بخش الگوریتم MDR با داده‌های واقعی شهر تهران تست‌شده است تا اهمیت استفاده از آن به مشخص شود.

در این آزمایش از یک سیستم با پردازنده Core i5 10300H و رم 16 گیگابایت استفاده شده است.

۵،۱ مقایسه‌ی MDR با مسیر انتخاب شده‌ی دستی

در این مقایسه ۵ نقطه‌ی مختلف انتخاب شده است: میدان‌آزادی، میدان‌انقلاب‌اسلامی، میدان ونک، برج‌میلاد و میدان آرژانتین که مبدا در نظرگرفته‌شده میدان‌آزادی می‌باشد (شکل ۹).



شکل ۹ – نقاط مشخص شده روی نقشه

در مسیر انتخاب شده‌ی به‌صورت دستی نقاط به‌ترتیب ذکر شده انتخاب شده‌اند که نتیجه‌ی آن در شکل ۱۰ آمده‌است:

شکل ۱۰ – مسیری که به‌صورت دستی مشخص شده است

در نهایت مسیریابی با استفاده از الگوریتم MDR انجام شده است که نتیجه‌ی آن در شکل ۱۱ آمده است:

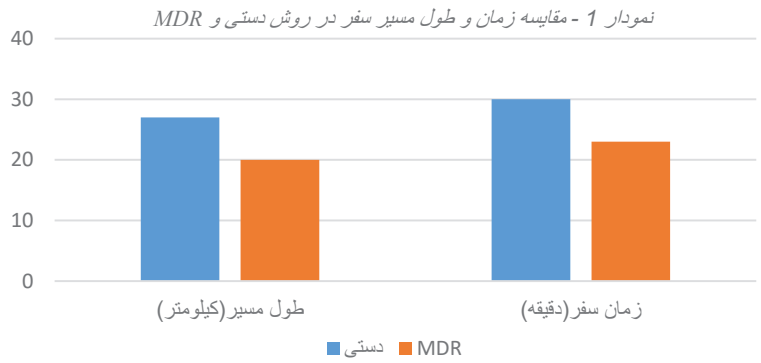
شکل ۱۱ – مسیری که با استفاده از MDR مشخص شده‌است

نتایج آماری از این آزمایش در جدول ۱ آمده‌است:

جدول ۱ – نتایج به‌دست آمده از آزمایش ۵،۱

روش مسیریابی	ترتیب مقصد	طول مسیر	زمان سفر
دستی	میدان‌آزادی – میدان‌انقلاب‌اسلامی – میدان‌ونک – برج‌میلاد – میدان‌آرژانتین	26.794km	30m
الگوریتم MDR	میدان‌آزادی – میدان‌انقلاب‌اسلامی – میدان‌آرژانین – میدان‌ونک – برج‌میلاد	19.787km	23m

طبق نتایج آماری به‌دست‌آمده از این آزمایش استفاده از الگوریتم MDR برای مسیریابی شاهد کاهش ۲۶ درصدی طول مسیر و کاهش ۲۳ درصدی زمان سفر می‌شود (نمودار ۱).



نمودار ۱ – مقایسه زمان و طول مسیر سفر در روش دستی و MDR

۵،۲ بررسی عملکرد زمانی الگوریتم MDR در تعداد نقاط مختلف

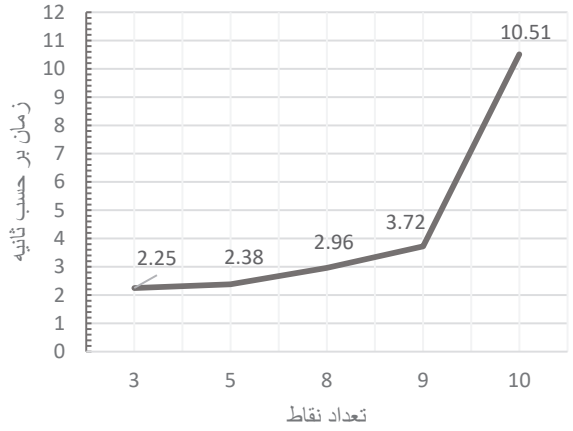
یکی از جنبه‌های کلیدی در ارزیابی الگوریتم‌های مسیریابی، بررسی مقیاس‌پذیری زمانی آن‌هاست؛ یعنی اینکه الگوریتم تا چه حد می‌تواند با افزایش تعداد نقاط مقصد بدون افت محسوس در کارایی، همچنان در زمان قابل‌قبولی اجرا شود. برای بررسی این موضوع، این آزمایش انجام شده‌است که در آن الگوریتم MDR روی مجموعه‌های مقصد با تعدادهای مختلف اجرا شده‌است.

نتایج به‌دست‌آمده در جدول ۲ آمده‌است:

جدول ۲ – نتایج به‌دست‌آمده از آزمایش ۵،۲

تعداد نقاط	سرعت الگوریتم
3	2.25
5	2.38
8	2.96
9	3.72
10	10.51

با توجه به نتایج فوق، می‌توان مشاهده کرد که زمان اجرا در حالت کلی با افزایش تعداد مقصد، افزایش می‌یابد، اما تا ۹ مقصد این رشد نسبتاً ملایم است و الگوریتم پاسخگو باقی می‌ماند. با این حال، در نقطه‌ی ۱۰ مقصد، یک جهش قابل‌توجه در زمان اجرا دیده می‌شود که ناشی از رشد ترکیبات مسیرهای ممکن و پیچیدگی پردازش در آن مقیاس است (نمودار ۲).



نمودار ۲ – مقایسه عملکرد زمان اجرایی الگوریتم MDR با تعداد نقاط مختلف

این نتایج نشان می‌دهند که الگوریتم MDR در مقیاس‌های کم تا متوسط، زمان اجرای مناسبی دارد و برای استفاده در سناریوهای عملیاتی با تعداد مقصد پایین، به‌ویژه در سامانه‌های مسیریابی شهری یا پلتفرم‌های تحویل، کاملاً مناسب است.

۵،۳ بررسی تأثیر نوع مسیر (خودرویی در برابر پیاده‌روی)

الگوریتم‌های مسیریابی در کاربردهای واقعی باید توانایی انطباق با انواع مختلف شبکه‌های حمل‌ونقل را داشته باشند. یکی از تفاوت‌های اساسی، نوع مسیر مورد استفاده است؛ مسیرهای پیاده‌روی و مسیرهای خودرویی نه‌تنها از نظر ساختاری بلکه از لحاظ محدودیت‌های دسترسی، جهت حرکت و قوانین راهنمایی و رانندگی نیز متفاوت‌اند.

برای بررسی عملکرد الگوریتم MDR در این شرایط، آزمایشی بر روی یک مجموعه‌ی ۵ نقطه‌ای در مرکز شهر تهران (محدوده میدان‌انقلاب‌اسلامی) انجام شده است(۱۲و۱۳). الگوریتم یک بار در حالت شبکه‌ی خودرو (network_type='drive') و یک بار در حالت شبکه‌ی پیاده‌روی (network_type='walk') اجرا گردیده است.

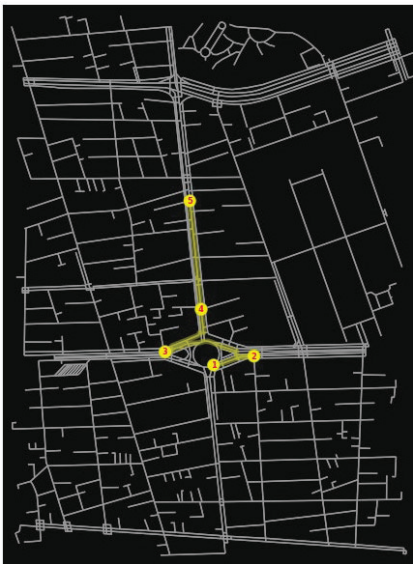
شکل ۱۲ – نقاط مشخص شده در شبکه خودرویی

شکل ۱۳ – نقاط مشخص شده در شبکه پیاده‌روی**نتایج به‌دست آمده به‌صورت زیر است:**

جدول ۳ – نتایج به‌دست‌آمده از آزمایش ۵،۳

نوع مسیر	طول مسیر	زمان سفر
خودرو	3.354km	4.3m
پیاده روی	0.916km	1.1m

شکل ۱۴ – مسیری که در شبکه خودرویی مشخص شده‌است



شکل ۱۵ – مسیری که در شبکه پیاده‌روی مشخص شده‌است

نتایج نشان می‌دهند که در این آزمایش، مسیر پیاده‌روی(شکل ۱۵) تقریباً یک‌سوم مسیر خودرویی (شکل ۱۴) طول داشته و در عین‌حال زمان سفر نیز به میزان قابل‌توجهی کمتر بوده است؛ این تفاوت به‌وضوح بیانگر توانایی الگوریتم MDR در استفاده از مسیرهای مناسب با نوع شبکه است. در مسیرهای پیاده، به‌دلیل امکان عبور از کوچه‌ها، پیاده‌روها و گذرگاه‌های خاص، طول و زمان بهینه‌تری حاصل شده است. در مقابل، مسیرهای خودرویی به‌دلیل محدودیت‌های عبور و جهت خیابان‌ها، مجبور به دور زدن و پیمودن مسیرهای طولانی‌تر شده‌اند.

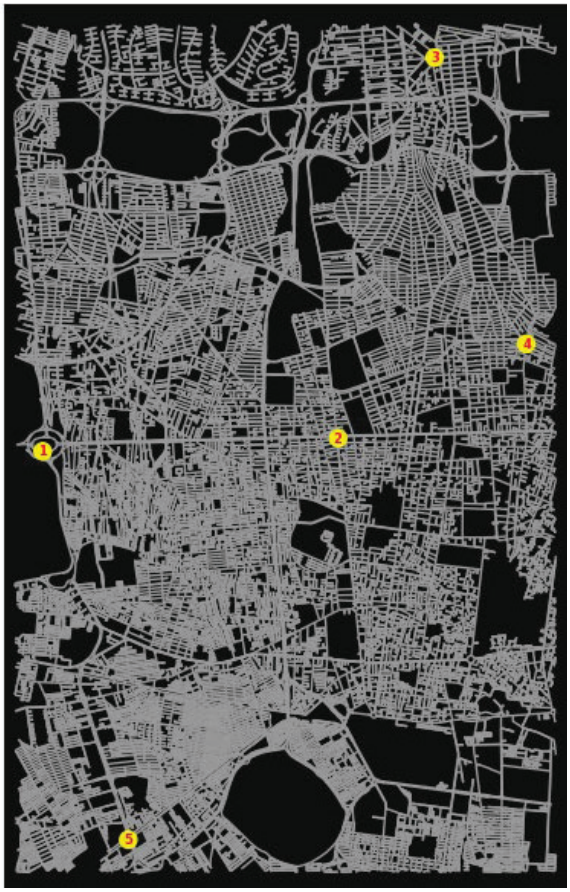
این توانایی انطباق، MDR را به ابزاری قابل اتکا برای استفاده در اپلیکیشن‌های ناوبری چندحالته، سیستم‌های مسیریابی هوشمند شهری، سامانه‌های تحویل پیک و کاربردهای موقعیت‌محور دیگر تبدیل می‌کند.

۵,۴ مقایسه با روش‌های پایه (Greedy و Brute-force)

برای ارزیابی دقت و کارایی الگوریتم MDR، آن را با دو روش پایه شامل Greedy و Brute-force مقایسه شده است. هدف این مقایسه، بررسی توانایی الگوریتم MDR در یافتن مسیر بهینه (از نظر طول مسیر و زمان سفر) و همچنین سنجش عملکرد زمان اجرای آن نسبت به روش‌های متداول است.

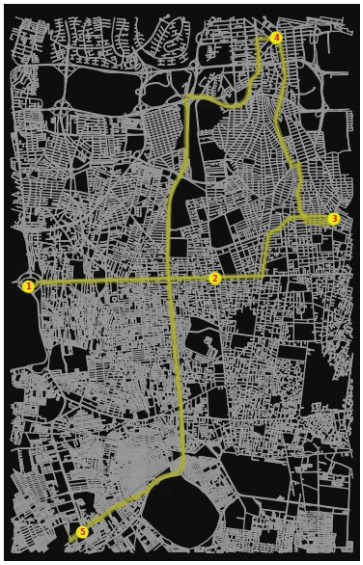
الگوریتم Greedy در هر مرحله، نزدیک‌ترین مقصد بعدی را انتخاب می‌کند و بدون در نظر گرفتن مسیر کلی، به‌صورت محلی تصمیم‌گیری می‌کند. این روش سریع است اما در بسیاری از موارد به مسیرهای غیر بهینه منجر می‌شود [9]. در مقابل، الگوریتم Brute-force تمامی ترتیب‌های ممکن برای بازدید از مقصدها را بررسی می‌کند و مسیر دقیق بهینه را با هزینه زمانی بسیار بالا پیدا می‌کند [8].

در این مقایسه ۵ نقطه‌ی مختلف انتخاب شده است: میدان‌آزادی، میدان‌انقلاب‌اسلامی، میدان ونک، میدان آرژانتین و بزرگراه آیت‌الله‌سعیدی که مبدا در نظرگرفته‌شده میدان‌آزادی می‌باشد (شکل ۱۶).



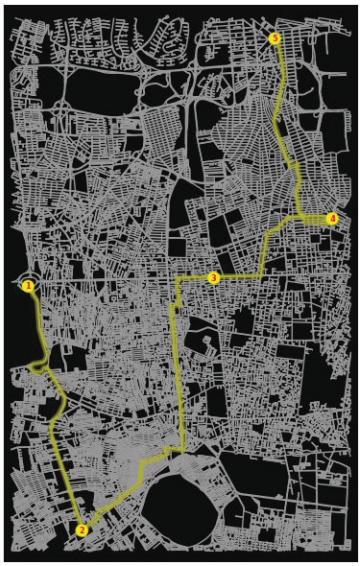
شکل ۱۶ – نقاط مشخص شده روی نقشه

نتایج به‌دست‌آمده در شکل ۱۷ و ۱۸ و ۱۹ آمده است:

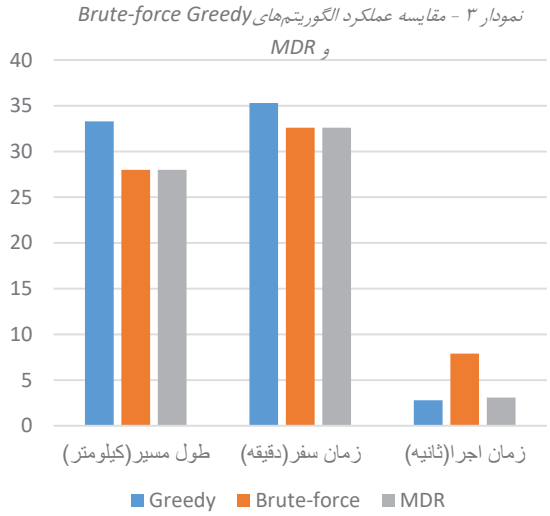


شکل ۱۷ – مسیری که با استفاده از **Greedy** مشخص شده است

شکل ۱۸ – مسیری که با استفاده از **Brute-force** مشخص شده است



شکل ۱۹ – مسیری که با استفاده از **MDR** مشخص شده است



نمودار ۳ - مقایسه عملکرد الگوریتم‌های Brute-force Greedy و MDR

جدول ۴ – نتایج به‌دست‌آمده از آزمایش ۵،۴

الگوریتم	طول مسیر	زمان سفر	زمان اجرا
Greedy	33.356km	35.3m	2.8s
Brute-force	28.208km	32.6m	7.9s
MDR	28.208km	32.6m	3.1s

نتایج جدول ۴ و نمودار ۳ نشان می‌دهد الگوریتم MDR همان مسیر بهینه‌ای را یافته که Brute-force با زمان اجرای بسیار بیشتر تولید کرده است. این بیانگر دقت بالای MDR است.

همچنین MDR در حالی زمان اجرای نزدیک به Greedy دارد که برخلاف آن، مسیر کاملاً بهینه ارائه می‌دهد. بنابراین، MDR با حفظ دقت، توازنی مؤثر میان سرعت اجرا و کیفیت مسیر برقرار کرده و عملکردی برتر نسبت به روش‌های پایه دارد.

۶ محدودیت‌ها و کارهای آینده

با وجود کارایی مناسب الگوریتم MDR در حل مسئله‌ی مسیریابی چندمقصدی بدون بازگشت به مبدا، این الگوریتم نیز دارای برخی محدودیت‌ها است. یکی از محدودیت‌های اصلی، رشد زمان اجرا در مقیاس‌های بسیار بزرگ (تعداد مقصدهای بالا) به دلیل افزایش ترکیبات ممکن و پیچیدگی ساخت گراف کامل بین مقاصد است. همچنین، در نسخه‌ی فعلی، الگوریتم تنها براساس طول مسیر یا زمان سفر تصمیم‌گیری می‌کند و قابلیت لحاظ‌کردن محدودیت‌های زمانی، اولویت‌های مقصد یا پنجره‌های زمانی تحویل را ندارد.

در ادامه این پژوهش، می‌توان نسخه‌ی بهینه‌تری از الگوریتم MDR طراحی کرد که از روش‌های ابتکاری یا یادگیری ماشین برای کاهش فضای جستجو استفاده کند. همچنین، امکان توسعه‌ی الگوریتم به‌گونه‌ای که از داده‌های بلادرنگ مانند ترافیک لحظه‌ای، مسدود بودن مسیر یا هزینه‌های متغیر بهره بگیرد، مسیر را برای استفاده در سامانه‌های ناوبری پویا هموار خواهد کرد. پشتیبانی از پارامترهای متنوع‌تر مانند محدودیت ظرفیت وسایل نقلیه یا زمان توقف نیز می‌تواند در نسخه‌های آتی اضافه شود.

نتیجه‌گیری

در این مقاله، الگوریتمی تحت‌عنوان "مسیریاب چندمقصدی" معرفی شد که قادر است مسئله‌ی مسیریابی از میان چند نقطه بدون الزام بازگشت به مبدا را با دقت بالا و زمان اجرای مناسب حل کند. این الگوریتم با استفاده از داده‌های نقشه‌ی OpenStreetMap و ساخت گراف‌های وزن‌دار، مسیر بهینه را میان مقاصد مختلف پیدا می‌کند.

آزمایش‌های انجام‌شده بر روی داده‌های واقعی شهر تهران نشان دادند که MDR توانسته با حفظ دقتی بالا معادل الگوریتم-Brute force، زمان اجرای بسیار پایین‌تری ارائه دهد و عملکردی به‌مراتب بهتر از روش‌های پایه مانند Greedy داشته باشد (نمودار ۳). توانایی انطباق با نوع مسیر (پیاده‌روی یا خودرو) و کاهش قابل‌توجه طول مسیر و زمان سفر از دیگر مزایای این الگوریتم به شمار می‌رود.

با توجه به نتایج به‌دست‌آمده، الگوریتم MDR می‌تواند به‌عنوان راهکاری عملی و موثر برای سیستم‌های مسیریابی، خدمات تحویل هوشمند و سامانه‌های مدیریت ناوگان شهری مورد استفاده قرار گیرد.

منابع

[1] Batty, M., & Xie, Y. (1999). From cells to cities: The automated geography of urban systems. *Environment and Planning B: Planning and Design*, 26(3), 343–358.

[2] Applegate, D., Bixby, R., Chvátal, V., & Cook, W. (2006). *The traveling salesman problem: A computational study*. Princeton University Press.

[3] Salazar, R., & Martínez, F. (2015). Solving the TSP using a hybrid genetic algorithm with local search. *Computational Intelligence*, 31(4), 671–688.

[4] Applegate, D., Bixby, R., Chvátal, V., & Cook, W. (2006). *The traveling salesman problem: A computational study*. Princeton University Press.

[5] Gendreau, M., & Potvin, J. Y. (2005). Metaheuristics for the traveling salesman problem. In F. Glover & G. A. Kochenberger (Eds.), *Handbook of metaheuristics* (pp. 257–287). Springer.

[6] Haklay, M., & Weber, P. (2008). OpenStreetMap: The intuitive and flexible map. In *Web 2.0 for transportation and urban planning* (pp. 105–126). Springer.

[7] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1), 269–271.

[8] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

[9] Johnson, D. S., & Papadimitriou, C. H. (1985). Performance guarantees for heuristics. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, & D. B. Shmoys (Eds.), *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization* (pp. 145–180). Wiley.