پردیس دانشکده های فنی دانشگاه تهران

دانشکده مهندسی نقشه برداری و اطلاعات مکانی

تمرین دوم هندسه محاسباتی

کدنویسی تقاطع به روش خط جاروب

استاد:

سرکار خانم دکتر زهرا بهرامیان

گروه هشتم:

کاروان جلالی، فرشاد شمسی خانی

بهار ۱۴۰۴

فهرست مطالب

مقدمه ۴

الگوریتم خط جاروب ۴

مراحل و کارایی ۴

پیاده سازی ۵

۱) کلاس Segment برای نمایش قطعه خط ۵

۲) خواندن قطعه‌خط‌ها از فایل Shapefile و تبدیل به اشیاء Segment ۷

۳) تعریف ساختار رویداد (Event) و صف رویدادها (EventQueue) ۹

۴) توابع کمکی هندسی(Geometry Helpers) ۱۱

۵) ساختار وضعیت(StatusStructure) ۱۳

۶) تابع پردازش رویداد(handle_event_point) ۱۵

۷) بررسی و افزودن رویداد تقاطع جدید(find_new_event) ۱۸

۸) محاسبه نقطه تقاطع دقیق بین دو قطعه‌خط (compute_intersection_point) ۲۰

۹) تابع اصلی اجرای الگوریتم جاروب(FINDINTERSECTIONS) ۲۲

۱۰) تشخیص نوع نقطه‌ی تلاقی classify_intersection ۲۴

۱۱) ذخیره‌سازی نقاط در فایل شیپ (save_intersections_to_shapefile) ۲۵

۱۲) ترسیم خطوط و نقاط تلاقی plot_segments_and_intersections ۲۶

۱۳) اجرای اصلی برنامه Main Program Execution ۲۷

نتیجه‌گیری ۲۸

مقدمه

در بسیاری از کاربردهای هندسه محاسباتی، مانند سیستم‌های اطلاعات مکانی، گرافیک کامپیوتری، و برنامه‌ریزی مسیر، نیاز به شناسایی نقاط تقاطع بین قطعات خطی وجود دارد. الگوریتم خط جاروب یکی از روش‌های مؤثر برای حل این مسئله است. این الگوریتم با حرکت یک خط فرضی (معمولاً عمودی) از یک سمت صفحه به سمت دیگر، نقاط تقاطع را شناسایی می‌کند. در هر موقعیت از حرکت خط جاروب، تنها قطعات خطی که در نزدیکی آن قرار دارند بررسی می‌شوند، که این امر باعث افزایش کارایی الگوریتم می‌شود. در این گزارش، به بررسی الگوریتم خط جاروب (Sweep Line) برای یافتن نقاط تقاطع بین مجموعه‌ای از قطعات خطی (Line Segments) می‌پردازیم و بخش های مختلف کد را توضیح میدهیم.

الگوریتم خط جاروب

یافتن نقاط تقاطع بین قطعات خطی یکی از مسائل بنیادین در هندسه محاسباتی است. این مسئله در برنامه‌ریزی مسیر برای ربات‌ها، تحلیل داده‌های مکانی در GIS ، و طراحی گرافیک کامپیوتری بسیار حیاتی است. به عنوان مثال، در GIS، شناسایی نقاط تقاطع برای نقشه‌برداری و تحلیل داده‌های مکانی ضروری است.

الگوریتم خط جاروب (Sweep Line) یکی از روش‌های کارآمد برای حل این مسئله است. این الگوریتم، که نوعی از الگوریتم‌های جاروب صفحه محسوب می‌شود، با حرکت یک خط فرضی (معمولاً عمودی) از یک سمت صفحه به سمت دیگر، نقاط تقاطع را شناسایی می‌کند. در هر موقعیت، تنها قطعات خطی که با خط جاروب تقاطع دارند بررسی می‌شوند.

مراحل و کارایی

الگوریتم خط جاروب شامل دو بخش اصلی است:

۱. **وضعیت خط جاروب (Sweep Line Status) :** فهرستی از قطعات خطی فعال که در حال حاضر با خط جاروب تقاطع دارند و بر اساس ترتیب عمودی مرتب شده‌اند.

۲. **صف رویدادها (Event Queue) :** شامل نقاطی مانند شروع و پایان قطعات خطی و نقاط احتمالی تقاطع که خط جاروب باید در آنها متوقف شود.

در هر مرحله، الگوریتم بررسی می‌کند که آیا قطعات خطی مجاور در وضعیت خط جاروب با هم تقاطع دارند یا خیر. این روش باعث می‌شود که تنها زوج‌های نزدیک به هم بررسی شوند، که کارایی را افزایش می‌دهد.

پیاده سازی

در ادامه، بخش‌های مختلف کد ارائه و توضیح داده خواهد شد تا درک عمیق‌تری از نحوه عملکرد آن فراهم شود.

۱) کلاس Segment برای نمایش قطعه خط

کلاس Segment نمایانگر یک قطعه خطی در صفحه دوبعدی است که توسط دو نقطه انتهایی تعریف می‌شود. این کلاس شامل ویژگی‌ها و متدهایی است که برای پردازش و مقایسه قطعات خطی در الگوریتم خط جاروب ضروری هستند.

```
# ============================
# Segment class to represent a line segment
# ============================
class Segment:
    def __init__(self, p1, p2, id=None):
        # Determine which point is the upper (higher y, or left-most if equal y)
        if p1[1] > p2[1] or (p1[1] == p2[1] and p1[0] < p2[0]):
            self.upper = p1
            self.lower = p2
        else:
            self.upper = p2
            self.lower = p1
        self.id = id
        self.is_horizontal = abs(p1[1] - p2[1]) < 1e-5  # Check for nearly horizontal segments

    def get_x_at_y(self, y):
        # Returns x-coordinate of the segment at a specific y (used in status ordering)
        x1, y1 = self.upper
```

```
    x2, y2 = self.lower
    if self.is_horizontal:
        return (x1 + x2) / 2  # Return midpoint x if segment is horizontal
    if abs(y2 - y1) < 1e-6:
        return x1
    return x1 + (x2 - x1) * (y - y1) / (y2 - y1)
```

ویژگی‌ها:

upper و lower : این دو ویژگی به ترتیب نشان‌دهنده نقاط بالایی و پایینی قطعه خط هستند. ترتیب‌دهی این نقاط بر اساس مختصات y انجام می‌شود و در صورت تساوی، مختصات x مقایسه می‌شود. این ترتیب‌دهی برای ساده‌سازی مقایسه‌ها در طول اجرای الگوریتم مفید است.

Id: شناسه‌ای یکتا برای هر قطعه خط که برای ردیابی و شناسایی آن در طول اجرای الگوریتم استفاده می‌شود.

is_horizontal: این ویژگی بررسی می‌کند که آیا قطعه خط افقی است یا خیر، با مقایسه تفاوت مختصات y نقاط انتهایی.

متدها:

get_x_at_y(y): این متد مقدار x را در یک مقدار y مشخص برای قطعه خط محاسبه می‌کند. این محاسبه برای تعیین موقعیت قطعه خط نسبت به خط جاروب در طول اجرای الگوریتم استفاده می‌شود.

استفاده از این کلاس باعث ساختاردهی بهتر به داده‌ها و تسهیل در پیاده‌سازی الگوریتم خط جاروب می‌شود.

۲) خواندن قطعه‌خط‌ها از فایل Shapefile و تبدیل به اشیاء Segment

برای خواندن داده‌های برداری از فایل‌های Shapefile در پایتون، کتابخانه‌های مختلفی وجود دارند. در این پروژه، از کتابخانه geopandas استفاده شده است که بر پایه pandas و shapely ساخته شده و امکانات قدرتمندی برای پردازش داده‌های مکانی فراهم می‌کند.

مراحل اصلی:

۱. **وارد کردن کتابخانه‌ها**

ابتدا، کتابخانه‌های مورد نیاز را وارد می‌کنیم:

```
import geopandas as gpd
from shapely.geometry import LineString
```

۲. **خواندن فایل Shapefile**

با استفاده از تابع gpd.read_file()، فایل Shapefile را می‌خوانیم و داده‌ها را در یک GeoDataFrame ذخیره می‌کنیم:

```
gdf = gpd.read_file('path_to_shapefile.shp')
```

در اینجا، gdf شامل تمام ویژگی‌ها و هندسه‌های موجود در فایل است.

۳. **تبدیل هندسه‌ها به اشیای Segment**

برای هر ردیف در gdf، هندسه را بررسی می‌کنیم و در صورت تطابق با نوع LineString، نقاط ابتدایی و انتهایی را استخراج کرده و یک شیء Segment ایجاد می‌کنیم:

```
segments = []
for idx, row in gdf.iterrows():
    geom = row.geometry
    if isinstance(geom, LineString):
        coords = list(geom.coords)
        start_point = coords[0]
        end_point = coords[-1]
        segment = Segment(start_point, end_point, id=idx)
        segments.append(segment)
```

در اینجا، Segment کلاسی است که قبلاً تعریف شده و نمایانگر یک قطعه‌خط با ویژگی‌هایی مانند نقاط ابتدایی و انتهایی و شناسه یکتا است.

با انجام این مراحل، مجموعه‌ای از اشیاء Segment در اختیار داریم که می‌توانند در الگوریتم خط جاروب برای شناسایی نقاط تقاطع مورد استفاده قرار گیرند.

```python
# ============================
# Read segments from shapefile into a list of Segment objects
# ============================
def read_segments_from_shapefile(shapefile_path):
    gdf = gpd.read_file(shapefile_path)
    segments = []
    print("Geometry types in shapefile:", gdf.geometry.type.unique())
    print("Total features:", len(gdf))
    for index, row in gdf.iterrows():
        line = row.geometry
        if isinstance(line, LineString):
            coords = list(line.coords)
            for j in range(len(coords) - 1):
                p1 = coords[j]
                p2 = coords[j + 1]
                segment = Segment(p1, p2, id=f"{index}_{j}")
                segments.append(segment)
        else:
            print(f"Skipping unsupported geometry at index {index}: {type(line)}")
    print("Total segments created:", len(segments))
    return segments, gdf
```

۳) تعریف ساختار رویداد (Event) و صف رویدادها (EventQueue)

در الگوریتم خط جاروب، تحلیل و پردازش نقاط کلیدی که تغییراتی در وضعیت خط جاروب ایجاد می‌کنند، ضروری است. این نقاط که شامل نقاط آغاز، پایان، یا نقاط تقاطع قطعات خطی هستند، با نام رویداد (Event) شناخته می‌شوند.

کلاس Event

کلاس Event برای مدل‌سازی یک رویداد در صفحه طراحی شده است و شامل اطلاعات زیر است:

- **point:** مختصات نقطه‌ای که رویداد در آن رخ می‌دهد (به‌صورت (x, y)).
- **Type:** نوع رویداد، که می‌تواند یکی از موارد زیر باشد:
  - ✓ 'start' برای نقطه شروع یک قطعه‌خط (یعنی نقطه بالایی یا چپ‌تر)
  - ✓ 'end' برای نقطه پایانی یک قطعه‌خط
  - ✓ 'intersection' برای نقطه تقاطع بین دو یا چند قطعه‌خط
- **Segments:** لیستی از قطعات خطی که در این رویداد مشارکت دارند.

همچنین، پیاده‌سازی متد __lt__در این کلاس ترتیب رویدادها را در صف مشخص می‌کند. طبق الگوریتم جاروب، رویدادها باید به ترتیب نزولی y و سپس صعودی x پردازش شوند، چون خط جاروب از بالا به پایین حرکت می‌کند.

کلاس EventQueue

کلاس EventQueue وظیفه مدیریت صف رویدادها را بر عهده دارد. رویدادها در این صف به ترتیب وقوع ذخیره شده و به مرور پردازش می‌شوند.

ویژگی‌ها و عملکردهای اصلی این کلاس شامل موارد زیر است:

- events لیستی از رویدادهای موجود در صف.

- seen_points مجموعه‌ای برای جلوگیری از اضافه شدن رویدادهای تکراری (بر اساس مختصات گرد شده).
- add(event) افزودن یک رویداد جدید به صف در صورتی که تکراری نباشد.
- pop() حذف و بازگرداندن اولین رویداد از صف (با اولویت بالا).
- is_empty() بررسی تهی بودن صف.

اهمیت Event و EventQueue در الگوریتم

این ساختارها هسته اجرای الگوریتم هستند. هر زمان که رویدادی اتفاق می‌افتد (مثلاً تقاطع بین دو قطعه‌خط)، موقعیت آن به صف افزوده می‌شود و سپس در زمان مناسب، با بررسی وضعیت فعلی و به‌روزرسانی وضعیت خط جاروب، پردازش می‌شود.

```
# ============================
# Event class for sweep line events (segment endpoints and intersections)
# ============================
class Event:
    def __init__(self, point, event_type, segments):
        self.point = point  # (x, y) tuple
        self.type = event_type  # 'start', 'end', or 'intersection'
        self.segments = segments  # list of involved segments

    def __lt__(self, other):
        # Events are ordered top-to-bottom, then left-to-right
        if abs(self.point[1] - other.point[1]) > 1e-6:
            return self.point[1] > other.point[1]
        return self.point[0] < other.point[0]

    def __repr__(self):
        return f"{self.type} @ {self.point}"

# ============================
# Event queue to manage all upcoming sweep line events
# ============================
class EventQueue:
    def __init__(self):
        self.events = []
```

```
    self.seen_points = set()

  def add(self, event):
    # Avoid adding duplicate events
    point_tuple = (round(event.point[0], 8), round(event.point[1], 8))
    if point_tuple not in self.seen_points:
      self.events.append(event)
      self.events.sort()  # Keep events ordered
      self.seen_points.add(point_tuple)

  def pop(self):
    return self.events.pop(0)

  def is_empty(self):
    return len(self.events) == 0
```

۴) توابع کمکی هندسی(Geometry Helpers)

در این بخش از برنامه، دو تابع تعریف شده‌اند که نقش حیاتی در تشخیص تقاطع میان دو قطعه‌خط دارند. این توابع کاملاً برگرفته از اصول هندسه محاسباتی هستند و پایه‌ی تابع do_intersect را تشکیل می‌دهند که بارها در طول الگوریتم فراخوانی می‌شود.

تابع ccw(A, B, C)

این تابع بررسی می‌کند که سه نقطه A,B,CA, B, CA,B,C به صورت پادساعت‌گرد مرتب شده‌اند یا خیر. این بررسی با استفاده از ضرب برداری انجام می‌شود.

(پادساعتگرد : (Counter ClockWise - CCW)

فرمول به کار رفته

```
(C[1] - A[1]) * (B[0] - A[0]) > (B[1] - A[1]) * (C[0] - A[0])
```

کاربرد:

- بررسی راست یا چپ بودن جهت چرخش بین سه نقطه.

تابع do_intersect(s1, s2)

این تابع بررسی می‌کند که آیا دو قطعه‌خط s1 و s2 با هم تقاطع دارند یا خیر.

از چهار تست CCW برای نقاط انتهایی قطعات استفاده می‌شود:

```
ccw(A, C, D) != ccw(B, C, D) and ccw(A, B, C) != ccw(A, B, D)
```

این آزمون بر اساس اصل معروفی در هندسه است که می‌گوید:

دو قطعه‌خط در صورتی با هم تقاطع دارند که نقاط انتهایی یکی از آن‌ها در دو سمت مختلف قطعه دیگر قرار گیرند.

مزایای این روش

- سریع : فقط نیاز به محاسبات ساده جبری دارد.
- دقیق : برخلاف مقایسه‌های عددی ساده، وابسته به دقت اعشاری نیست.
- عمومی : برای همه انواع قطعات (به جز هم‌خطی کامل) قابل استفاده است.

```python
# ============================
# Geometry helpers
# ============================
def ccw(A, B, C):
    return (C[1] - A[1]) * (B[0] - A[0]) > (B[1] - A[1]) * (C[0] - A[0])

def do_intersect(s1, s2):
    A, B = s1.upper, s1.lower
    C, D = s2.upper, s2.lower
    return ccw(A, C, D) != ccw(B, C, D) and ccw(A, B, C) != ccw(A, B, D)
```

۵) ساختار وضعیت(StatusStructure)

در الگوریتم خط جاروب، لازم است در هر لحظه بدانیم کدام قطعات خطی توسط خط جاروب قطع شده‌اند و ترتیب افقی آن‌ها چگونه است. این اطلاعات در ساختاری به نام Status Structureیا «ساختار وضعیت» نگهداری می‌شود.

کلاس StatusStructure

این کلاس لیستی از قطعات فعال را نگه می‌دارد؛ منظور از قطعه فعال، قطعه‌ای است که در لحظه برخورد خط جاروب به یک رویداد، هنوز تقاطعش با خط جاروب خاتمه نیافته است. این قطعات به صورت افقی (بر اساس مقدار x آن‌ها) در محل خط جاروب (مرتب می‌شوند.

متدهای اصلی

- **__init__** مقداردهی اولیه و ایجاد لیست active_segmentsبه عنوان لیست خالی.
- **insert(segment, y)** اضافه‌کردن یک قطعه‌خط به لیست فعال‌ها. مکان درج به‌گونه‌ای تعیین می‌شود که ترتیب افقی آن حفظ شود. برای این منظور، ابتدا مختصات x مربوط به موقعیت y (نزدیک به رویداد) برای قطعه محاسبه شده و سپس بر اساس مقدار x آن در لیست قرار می‌گیرد.

(اگر قطعه افقی باشد (is_horizontal = True)، مستقیماً به انتهای لیست افزوده می‌شود.)

- **delete(segment)** حذف قطعه‌خط مشخص از لیست فعال‌ها در صورت وجود.
- **segments_containing_point(p)** بررسی این‌که آیا نقطه مورد نظر p روی بدنه یکی از قطعه‌خط‌های فعال قرار دارد یا خیر. اگر چنین باشد، آن قطعه‌خط به لیست خروجی افزوده می‌شود.

دلیل استفاده از ساختار وضعیت

یکی از اصول کلیدی در الگوریتم خط جاروب این است که تنها تقاطع‌های احتمالی بین قطعات مجاور در ساختار وضعیت بررسی شوند، زیرا سایر قطعات به دلیل ترتیب افقی‌شان نمی‌توانند در آینده نزدیک با هم برخورد داشته باشند.

نکات پیاده‌سازی

- ترتیب قطعات در لیست active_segments حیاتی است و باید دقیقاً بازتاب‌دهنده ترتیب آن‌ها روی خط جاروب باشد.
- قطعات افقی انتهای لیست قرار می‌گیرند تا با اولویت پایین‌تری بررسی شوند.
- استفاده از مقدار کمی کمتر از y (y - 1e-6) برای جلوگیری از تداخل با خود نقطه رویداد انجام می‌شود.

```
# ============================
# Sweep line status structure to track active segments
# ============================
class StatusStructure:
    def __init__(self):
        self.active_segments = []

    def insert(self, segment, y):
        # Insert segment into correct horizontal order based on x at sweep line y
        if segment.is_horizontal:
            self.active_segments.append(segment)
        else:
            x = segment.get_x_at_y(y - 1e-6)
            i = 0
            while i < len(self.active_segments) and (
                self.active_segments[i].is_horizontal or
                self.active_segments[i].get_x_at_y(y - 1e-6) < x
            ):
                i += 1
            self.active_segments.insert(i, segment)

    def delete(self, segment):
        # Remove a segment if present
```

```
    if segment in self.active_segments:
        self.active_segments.remove(segment)

def segments_containing_point(self, p):
    # Return segments that contain the point p
    result = []
    for seg in self.active_segments:
        if seg.upper != p and seg.lower != p:
            y_min = min(seg.upper[1], seg.lower[1])
            y_max = max(seg.upper[1], seg.lower[1])
            if y_min - 1e-6 <= p[1] <= y_max + 1e-6:
                x = seg.get_x_at_y(p[1])
                if seg.is_horizontal:
                    x_min = min(seg.upper[0], seg.lower[0])
                    x_max = max(seg.upper[0], seg.lower[0])
                    if x_min - 1e-6 <= p[0] <= x_max + 1e-6:
                        result.append(seg)
                elif abs(x - p[0]) < 1e-6:
                    result.append(seg)
    return result
```

۶) تابع پردازش رویداد (handle_event_point)

الگوریتم جاروب هنگام رسیدن به یک رویداد (نقطه شروع، پایان یا تقاطع)، باید وضعیت خط جاروب را به‌روزرسانی کند. این وظیفه به عهده تابع handle_event_point است.

این تابع در هر بار اجرا، مراحل زیر را انجام می‌دهد:

۱) بازیابی اطلاعات مربوط به رویداد

```
U = point_segment_map.get(('U', p), [])  # Segments starting at p
L = point_segment_map.get(('L', p), [])  # Segments ending at p
C = T.segments_containing_point(p)       # Segments passing through p
```

- U لیستی از قطعاتی که از نقطه p شروع می‌شوند.

- L لیستی از قطعاتی که در p پایان می‌یابند.
- C لیستی از قطعاتی که p در داخل آن‌ها قرار دارد (نه به عنوان ابتدا یا انتها).

۲) ثبت نقطه تقاطع (اگر بیش از یک قطعه در آن نقطه مشارکت دارد)

```
if len(involved) > 1 and point_tuple not in seen_points:
   ...
   output.append((p, involved))
```

- اگر بیش از یک قطعه در نقطه p حضور دارند، آن نقطه به‌عنوان نقطه تقاطع ثبت می‌شود (در خروجی و همچنین در مجموعه‌ای به نام seen_points برای جلوگیری از تکرار)

۳) حذف و درج مجدد قطعات در ساختار وضعیت

```
for s in L + C:
   T.delete(s)
for s in U + C:
   T.insert(s, p[1])
```

- قطعاتی که در p پایان می‌یابند (L) و آن‌هایی که از داخل p عبور می‌کنند (C) از ساختار وضعیت حذف می‌شوند.
- قطعاتی که در p آغاز می‌شوند (U) و C دوباره در ساختار وضعیت درج می‌شوند (با موقعیت جدید خط جاروب در ارتفاع y = p[1]).

۴) بررسی همسایه‌های جدید برای یافتن رویداد تقاطع آینده

```
for i in range(len(T.active_segments) - 1):
   s1 = T.active_segments[i]
   s2 = T.active_segments[i + 1]
   find_new_event(s1, s2, p, Q, seen_points)
```

- پس از به‌روزرسانی لیست قطعات فعال، همسایه‌های جدید در ساختار وضعیت بررسی می‌شوند.

- اگر بین آن‌ها تقاطعی وجود داشته باشد که پایین‌تر از نقطه فعلی p باشد، به صف رویداد اضافه می‌شود.

اهمیت این تابع

- این‌جا تصمیم‌گیری برای به‌روزرسانی $\mathcal{T}$(ساختار وضعیت) و ثبت تقاطع‌های جدید انجام می‌شود.
- همچنین تضمین می‌کند که هر تقاطع فقط یک‌بار گزارش شود (با کمک seen_points).

```
# ==========================
# Handle an event point
# ==========================
def handle_event_point(p, T, Q, point_segment_map, output, seen_points):
    U = point_segment_map.get(('U', p), [])  # Segments starting at p
    L = point_segment_map.get(('L', p), [])  # Segments ending at p
    C = T.segments_containing_point(p)      # Segments passing through p
    involved = list(set(U + L + C))
    point_tuple = (round(p[0], 8), round(p[1], 8))

    # Report intersection if more than one segment is involved
    if len(involved) > 1 and point_tuple not in seen_points:
        intersection_type = classify_intersection(p, involved)
        print(f"Intersection at {p} with segments: {[s.id for s in involved]} → type:
{intersection_type}")
        output.append((p, involved))
        seen_points.add(point_tuple)

    # Update the status structure
    for s in L + C:
        T.delete(s)
    for s in U + C:
        T.insert(s, p[1])

    # Check new intersections
    for i in range(len(T.active_segments) - 1):
        s1 = T.active_segments[i]
        s2 = T.active_segments[i + 1]
        find_new_event(s1, s2, p, Q, seen_points)
```

۷) بررسی و افزودن رویداد تقاطع جدید(find_new_event)

این تابع برای شناسایی رویدادهای تقاطع آتی بین دو قطعه‌خط (s1, s2) استفاده می‌شود. معمولاً این دو قطعه همسایگان جدید در ساختار وضعیت هستند و باید بررسی شود آیا در آینده‌ای نزدیک با هم برخورد می‌کنند یا نه.

ساختار کلی تابع:

```
def find_new_event(s1, s2, p, Q, seen_points):
    if s1 is None or s2 is None or s1 == s2:
        return
```

اگر یکی از ورودی‌ها None یا هر دو یکسان باشند، تابع سریعاً باز می‌گردد – این شرط برای اطمینان از درستی داده‌ها است.

مرحله دوم: بررسی تقاطع

```
    if do_intersect(s1, s2):
        ipt = compute_intersection_point(s1, s2)
```

از تابع do_intersect (که پیش‌تر توضیح دادیم) برای بررسی وجود تقاطع استفاده می‌شود.

در صورت وجود تقاطع، محل دقیق آن با compute_intersection_point محاسبه می‌شود (که در پیام بعدی دقیق بررسی می‌کنیم).

مرحله سوم: شرط برای اینکه تقاطع واقعاً «جدید» باشد

```
        if ipt and (
            ipt[1] < p[1] - 1e-6 or
            (abs(ipt[1] - p[1]) < 1e-6 and ipt[0] > p[0] + 1e-6) ):
```

- این شرط تضمین می‌کند که فقط رویدادهایی پایین‌تر از نقطه جاری p یا در همان y ولی سمت راست‌تر از p وارد صف رویداد شوند.

- این موضوع از این واقعیت سرچشمه می‌گیرد که خط جاروب به سمت پایین حرکت می‌کند، بنابراین فقط رویدادهایی که در آینده پیش‌رو هستند باید پردازش شوند.

افزودن به صف

```
Q.add(Event(ipt, 'intersection', [s1, s2]))
```

- اگر تمام شرایط بالا برقرار باشد، یک رویداد جدید از نوع 'intersection' ساخته شده و به صف رویدادها اضافه می‌شود.

اگرچه این تابع کوچک است، اما یکی از کلیدی‌ترین قسمت‌های الگوریتم Sweep Line محسوب می‌شود. بدون آن، بسیاری از تقاطع‌های بالقوه بین قطعات خطی شناسایی نمی‌شوند و خروجی ناقص خواهد بود.

```
# ==========================
# Check and enqueue new intersection event
# ==========================
def find_new_event(s1, s2, p, Q, seen_points):
    if s1 is None or s2 is None or s1 == s2:
        return
    if do_intersect(s1, s2):
        ipt = compute_intersection_point(s1, s2)
        if ipt and (ipt[1] < p[1] - 1e-6 or (abs(ipt[1] - p[1]) < 1e-6 and ipt[0] > p[0] + 1e-6)):
            Q.add(Event(ipt, 'intersection', [s1, s2]))
```

۸) محاسبه نقطه تقاطع دقیق بین دو قطعه‌خط (compute_intersection_point)

در الگوریتم خط جاروب، بعد از تشخیص اولیه اینکه دو قطعه ممکن است تقاطع داشته باشند با (do_intersect)، لازم است نقطه دقیق تقاطع آن‌ها را بیابیم. این وظیفه بر عهده تابع compute_intersection_point است.

مراحل گام‌به‌گام:

۱) بازیابی مختصات نقاط انتهایی دو قطعه‌خط

```
x1, y1 = s1.upper
x2, y2 = s1.lower
x3, y3 = s2.upper
x4, y4 = s2.lower
```

هر قطعه‌خط دارای دو نقطه است که مختصات آن‌ها استخراج می‌شود.

۲) محاسبه تعیین‌کننده (determinant) برای تشخیص موازی بودن

```
denom = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4)
if abs(denom) < 1e-6:
   return None
```

این مقدار مخرج معادلات خطی است. اگر برابر صفر (یا خیلی نزدیک به صفر) باشد، به این معنی است که خطوط یا موازی هستند یا منطبق. در این صورت، تابع None برمی‌گرداند و ادامه نمی‌دهد.

۳) محاسبه مختصات نقطه تقاطع

```
px = ((x1 * y2 - y1 * x2) * (x3 - x4) - (x1 - x2) * (x3 * y4 - y3 * x4)) / denom
py = ((x1 * y2 - y1 * x2) * (y3 - y4) - (y1 - y2) * (x3 * y4 - y3 * x4)) / denom
```

این دو فرمول از حل دستگاه معادلات دو خط به دست آمده‌اند و این روش بر پایه تبدیل دو خط به معادله پارامتری یا استفاده از ضرب برداری است.

۴) تأیید اینکه نقطه تقاطع واقعاً روی هر دو قطعه‌خط قرار دارد

```
def between(a, b, c):
return min(a, b) - 1e-6 <= c <= max(a, b) + 1e-6
```

و سپس:

```
if between(x1, x2, px) and between(y1, y2, py) and between(x3, x4, px) and between(y3, y4, py):
return (px, py)
```

بررسی می‌شود که مختصات نقطه تقاطع واقعاً در بازه بین دو نقطه انتهایی هر دو قطعه‌خط قرار دارد، در غیر این صورت، حتی اگر خطوط نامتناهی تقاطع داشته باشند، ولی تقاطع آن‌ها بیرون از قطعات باشد، None برمی‌گرداند.

```
# ============================
# Compute intersection point of two segments
# ============================
def compute_intersection_point(s1, s2):
    x1, y1 = s1.upper
    x2, y2 = s1.lower
    x3, y3 = s2.upper
    x4, y4 = s2.lower
    denom = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4)
    if abs(denom) < 1e-6:
        return None
    px = ((x1 * y2 - y1 * x2) * (x3 - x4) - (x1 - x2) * (x3 * y4 - y3 * x4)) / denom
    py = ((x1 * y2 - y1 * x2) * (y3 - y4) - (y1 - y2) * (x3 * y4 - y3 * x4)) / denom
    def between(a, b, c): return min(a, b) - 1e-6 <= c <= max(a, b) + 1e-6
    if between(x1, x2, px) and between(y1, y2, py) and between(x3, x4, px) and
between(y3, y4, py):
        return (px, py)
    return None
```

۹) تابع اصلی اجرای الگوریتم جاروب(FINDINTERSECTIONS)

این تابع، هسته اجرایی الگوریتم Sweep Line است و تمام ساختارها و توابعی که تاکنون بررسی کردیم را گرد هم می‌آورد تا نقاط تقاطع بین قطعات خطی را بیابد.

۱. ساخت Event Queue

با استفاده از کلاس EventQueue، تمامی نقاط ابتدایی و انتهایی پاره‌خط‌ها به عنوان رویدادهای اولیه به صف رویداد اضافه می‌شوند. هر پاره‌خط دو رویداد دارد:

- ✓ start نقطه‌ای که پاره‌خط از آن آغاز می‌شود (نقطه بالایی).
- ✓ end نقطه‌ای که پاره‌خط در آن به پایان می‌رسد (نقطه پایینی).

۲. ساخت Status Structure

از کلاس StatusStructure برای نگهداری وضعیت فعلی پاره‌خط‌هایی که خط جاروب در آن لحظه از آن‌ها عبور می‌کند، استفاده می‌شود. این ساختار مرتب‌سازی‌شده است تا بتوان بررسی کرد که کدام پاره‌خط‌ها ممکن است با هم تقاطع داشته باشند.

۳. ساخت نگاشت نقطه به پاره‌خط (point_segment_map)

این دیکشنری به ما اجازه می‌دهد تا سریعاً بفهمیم کدام پاره‌خط‌ها در یک نقطه خاص شروع یا تمام می‌شوند. کلیدها به صورت ('U', point) برای نقاط آغاز و ('L', point) برای نقاط پایان تعریف شده‌اند.

۴. حلقه پردازش رویدادها

تا زمانی که صف رویداد خالی نشده، به ترتیب از بالا به پایین (y بزرگ‌تر به y کوچک‌تر) و چپ به راست رویدادها بررسی می‌شوند. برای هر رویداد:

تابع handle_event_point فراخوانی می‌شود تا وضعیت را به‌روز کند، تقاطع‌ها را ثبت کند، و اگر لازم باشد رویدادهای جدیدی (از نوع intersection) به صف اضافه کند.

۵. بازگشت خروجی

در پایان، لیستی از نقاط تقاطع به همراه پاره‌خط‌های درگیر در هر تقاطع بازگردانده می‌شود.

- استفاده از ساختارهای داده‌ای مانند صف اولویت (EventQueue) و وضعیت جاروب (StatusStructure) امکان پیاده‌سازی کارآمد الگوریتم را فراهم می‌کند.
- با هر بار وقوع رویداد، پاره‌خط‌ها از وضعیت اضافه یا حذف می‌شوند و همسایگان جدید بررسی می‌شوند.
- تقاطع‌هایی که قبلاً دیده شده‌اند با کمک مجموعه seen_points نادیده گرفته می‌شوند تا از تکرار جلوگیری شود.

```
# ============================
# Main intersection algorithm using sweep line
# ============================
def FINDINTERSECTIONS(segments):
    Q = EventQueue()
    T = StatusStructure()
    output = []
    seen_points = set()
    point_segment_map = {}
    for s in segments:
        point_segment_map.setdefault(('U', s.upper), []).append(s)
        Q.add(Event(s.upper, 'start', [s]))
        point_segment_map.setdefault(('L', s.lower), []).append(s)
        Q.add(Event(s.lower, 'end', [s]))
    while not Q.is_empty():
        event = Q.pop()
        handle_event_point(event.point, T, Q, point_segment_map, output, seen_points)
    return output
```

۱۰) تشخیص نوع نقطه‌ی تلاقی classify_intersection

این تابع وظیفه دارد تعیین کند که یک نقطه‌ی تلاقی، از نوع نقطه‌ی پایانی (endpoint) است یا درونی.(interior)

- **ورودی‌های تابع شامل:**
  - point نقطه‌ای که باید بررسی شود.
  - segments لیستی از قطعاتی (خطوط) که در آن نقطه تلاقی دارند.
- **منطق عملکرد:**
  - اگر نقطه‌ی تلاقی دقیقا منطبق با یکی از نقاط پایانی (بالا یا پایین) یک قطعه باشد، آن تلاقی به عنوان "endpoint" دسته‌بندی می‌شود.
  - در غیر این صورت، "interior"محسوب می‌شود، یعنی تلاقی در داخل یکی از خطوط اتفاق افتاده و نه در انتهای آن.

کاربرد: این دسته‌بندی برای تعیین رنگ و نوع نمایش در خروجی گرافیکی و همچنین در shapefile خروجی ضروری است.

```
# ============================
# Classify intersection type: endpoint or interior
# ============================
def classify_intersection(point, segments):
   for s in segments:
      if (abs(point[0] - s.upper[0]) < 1e-6 and abs(point[1] - s.upper[1]) < 1e-6) or \
        (abs(point[0] - s.lower[0]) < 1e-6 and abs(point[1] - s.lower[1]) < 1e-6):
         return "endpoint"
   return "interior"
```

۱۱) ذخیره‌سازی نقاط در فایل شیپ (save_intersections_to_shapefile)

این تابع وظیفه دارد اطلاعات مربوط به نقاط تلاقی را در قالب یک فایل shapefile ذخیره کند تا بتوان آن را در نرم‌افزارهای GIS مانند QGIS یا ArcGIS مشاهده و تحلیل کرد.

- **ورودی‌ها:**

intersections لیستی از نقاط تلاقی به همراه خطوط درگیر.
output_path مسیر ذخیره‌ی فایل خروجی.
crs سیستم مختصات مکانی (Coordinate Reference System) فایل اصلی، برای تطبیق صحیح مکان نقاط.

برای هر تلاقی نوع آن (درونی یا پایانی) با استفاده از تابع classify_intersection تعیین می‌شود سپس شناسه‌ی خطوط درگیر جمع‌آوری می‌شود و نقطه و اطلاعات مربوطه به یک DataFrame جغرافیایی (GeoDataFrame) اضافه می‌شود. در نهایت فایل shapefile ایجاد و ذخیره می‌شود.

```
# ==========================
# Save intersection results to shapefile
# ==========================
def save_intersections_to_shapefile(intersections, output_path, crs):
    data = {'geometry': [], 'type': [], 'seg_ids': []}
    for point, segs in intersections:
        kind = classify_intersection(point, segs)
        seg_ids = [s.id for s in segs]
        data['geometry'].append(Point(point))
        data['type'].append(kind)
        data['seg_ids'].append(','.join(map(str, seg_ids)))
    gdf = gpd.GeoDataFrame(data, crs=crs)
    gdf.to_file(output_path)
    print(f"Intersections saved to {output_path}")
```

۱۲) ترسیم خطوط و نقاط تلاقی plot_segments_and_intersections

این تابع یک نمودار تصویری از خطوط و نقاط تلاقی ترسیم می‌کند.

مراحل اجرا: ابتدا خطوط موجود در shapefile ترسیم می‌شوند، سپس، نقاط تلاقی با توجه به نوعشان (درونی یا پایانی) به رنگ‌های مختلف (قرمز یا آبی) نمایش داده می‌شوند و نمودار حاصل ذخیره شده و نمایش داده می‌شود.

```
# ============================
# Plot segments and intersection points
# ============================
def plot_segments_and_intersections(gdf_segments, intersections):
    fig, ax = plt.subplots(figsize=(8, 6))
    gdf_segments.plot(ax=ax, color='#FFCC99', linewidth=0.7, label='Segments')
    points = []
    colors = []
    for point, segs in intersections:
        kind = classify_intersection(point, segs)
        points.append(Point(point))
        colors.append('red' if kind == 'interior' else 'blue')
    if points:
        gdf_points = gpd.GeoDataFrame(geometry=points, crs=gdf_segments.crs)
        gdf_points.plot(ax=ax, color=colors, markersize=40, label='Intersections')
    plt.title("Line Segments and Intersections")
    plt.legend()
    plt.axis('off')
    plt.tight_layout()
    plt.savefig("Exports/intersections_map.png", dpi=300)
    plt.show()
```

۱۳) اجرای اصلی برنامه Main Program Execution

این بخش بدنه‌ی اصلی اجرای برنامه را شامل می‌شود.

۱. **خواندن فایل shapefile**: خطوط از فایل "layers/FINAL.shp" خوانده می‌شوند.
۲. **یافتن نقاط تلاقی**: با استفاده از الگوریتم جاروی خطی (FINDINTERSECTIONS) نقاط تلاقی محاسبه می‌شوند.
۳. ذخیره خروجی‌ها:

- ✓ ابتدا اگر پوشه‌ی "Exports" وجود نداشته باشد، ساخته می‌شود.
- ✓ سپس نقاط تلاقی در فایل "Exports/intersections.shp" ذخیره می‌شوند.

۴. **نمایش اطلاعات تلاقی**: در خروجی کنسول اطلاعات هر تلاقی چاپ می‌شود.
۵. **ترسیم نمودار:** خطوط و نقاط تلاقی روی یک نقشه ترسیم می‌گردند.

اهمیت این بخش: این قسمت همه بخش‌های قبلی را به یکدیگر متصل کرده و فرآیند کلی را تکمیل می‌کند. از دریافت داده‌ها گرفته تا پردازش و نمایش نهایی همگی در این قسمت انجام می‌شود.

```
# ============================
# Main program execution
# ============================
shapefile_path = "layers/FINAL.shp"
segments, gdf_segments = read_segments_from_shapefile(shapefile_path)
intersections = FINDINTERSECTIONS(segments)
if not os.path.exists('Exports'):
    os.makedirs('Exports')
output_shapefile_path = "Exports/intersections.shp"
save_intersections_to_shapefile(intersections, output_shapefile_path,
gdf_segments.crs)
for point, segs in intersections:
    seg_ids = [f"S{s.id}" for s in segs]
    kind = classify_intersection(point, segs)
    print(f"Intersection at {point} between {seg_ids} → type: {kind}")
plot_segments_and_intersections(gdf_segments, intersections)
```

نتیجه‌گیری

در این پروژه، فرآیند شناسایی، طبقه‌بندی، ذخیره‌سازی و نمایش نقاط تلاقی بین خطوط یک فایل مکانی (Shapefile) با موفقیت انجام شد. ابتدا با استفاده از الگوریتم یافتن تلاقی (احتمالاً الگوریتم جاروی خطی یا مشابه آن) نقاط تلاقی بین خطوط استخراج شدند. سپس با بهره‌گیری از توابع کمکی نوع تلاقی (پایانی یا درونی) برای هر نقطه تعیین گردید، اطلاعات مکانی و توصیفی تلاقی‌ها در قالب فایل خروجی قابل استفاده در محیط‌های GIS ذخیره شد، و در نهایت، نمایش گرافیکی مناسبی از خطوط و نقاط تلاقی تهیه و ذخیره شد. این فرآیند می‌تواند در تحلیل‌های شبکه‌ای، نقشه‌برداری، مطالعات شهری، یا مدیریت زیرساخت‌ها کاربرد وسیعی داشته باشد؛ به ویژه در مواقعی که نیاز به درک موقعیت‌های حساس از نظر اتصال یا تقاطع در داده‌های مکانی وجود دارد.