부산정보산업진흥원
Busan IT Industry Promotion Agency

동아대학교

Practice: Pathfinding and Graph Search Algorithms

동아대학교 컴퓨터AI공학부
천세진

PATHFINDING AND GRAPH SEARCH ALGORITHMS WITH NEO4J

Importing the Data into Neo4j

- 데이터베이스 생성 및 데이터베이스 선택

```
:use system
CREATE DATABASE chapter4;
:use chapter4
```

transport-nodes.csv

	id	latitude	longitude	population
1	Amsterdam	52.379189	4.899431	821752
2	Utrecht	52.092876	5.104480	334176
3	Den Haag	52.078663	4.288788	514861
4	Immingham	53.61239	-0.22219	9642
5	Doncaster	53.52285	-1.13116	302400
6	Hoek van Holland	51.9775	4.13333	9382
7	Felixstowe	51.96375	1.3511	23689
8	Ipswich	52.05917	1.15545	133384
9	Colchester	51.88921	0.90421	104390
10	London	51.509865	-0.118092	8787892
11	Rotterdam	51.9225	4.47917	623652
12	Gouda	52.01667	4.70833	70939

Importing the Data into Neo4j

- 데이터베이스 생성 및 데이터베이스 선택

```
WITH "https://github.com/chunsejin/KGProjects_DAU/raw/master/data/" AS base
WITH base + "transport-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (place:Place {id:row.id})
SET place.latitude = toFloat(row.latitude),
place.longitude = toFloat(row.longitude),
place.population = toInteger(row.population);
```

transport-relationships.csv

	src	dst	relationship	cost
1	Amsterdam	Utrecht	EROAD	46
2	Amsterdam	Den Haag	EROAD	59
3	Den Haag	Rotterdam	EROAD	26
4	Amsterdam	Immingham	EROAD	369
5	Immingham	Doncaster	EROAD	74
6	Doncaster	London	EROAD	277
7	Hoek van Holland	Den Haag	EROAD	27
8	Felixstowe	Hoek van Holland	EROAD	207
9	Ipswich	Felixstowe	EROAD	22
10	Colchester	Ipswich	EROAD	32
11	London	Colchester	EROAD	106
12	Gouda	Rotterdam	EROAD	25
13	Gouda	Utrecht	EROAD	35
14	Den Haag	Gouda	EROAD	32
15	Hoek van Holland	Rotterdam	EROAD	33

Importing the Data into Neo4j

- 데이터베이스 생성 및 데이터베이스 선택

```
WITH "https://github.com/chunsejin/KGProjects_DAU/raw/master/data/" AS base
WITH base + "transport-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (origin:Place {id: row.src})
MATCH (destination:Place {id: row.dst})
MERGE (origin)-[:EROAD {distance: toInteger(row.cost)}]->(destination);
```

Database Information

Use database

chapter4

Node Labels

*(12) Place

Relationship Types

*(15) EROAD

Property Keys

distance id latitude longitude population

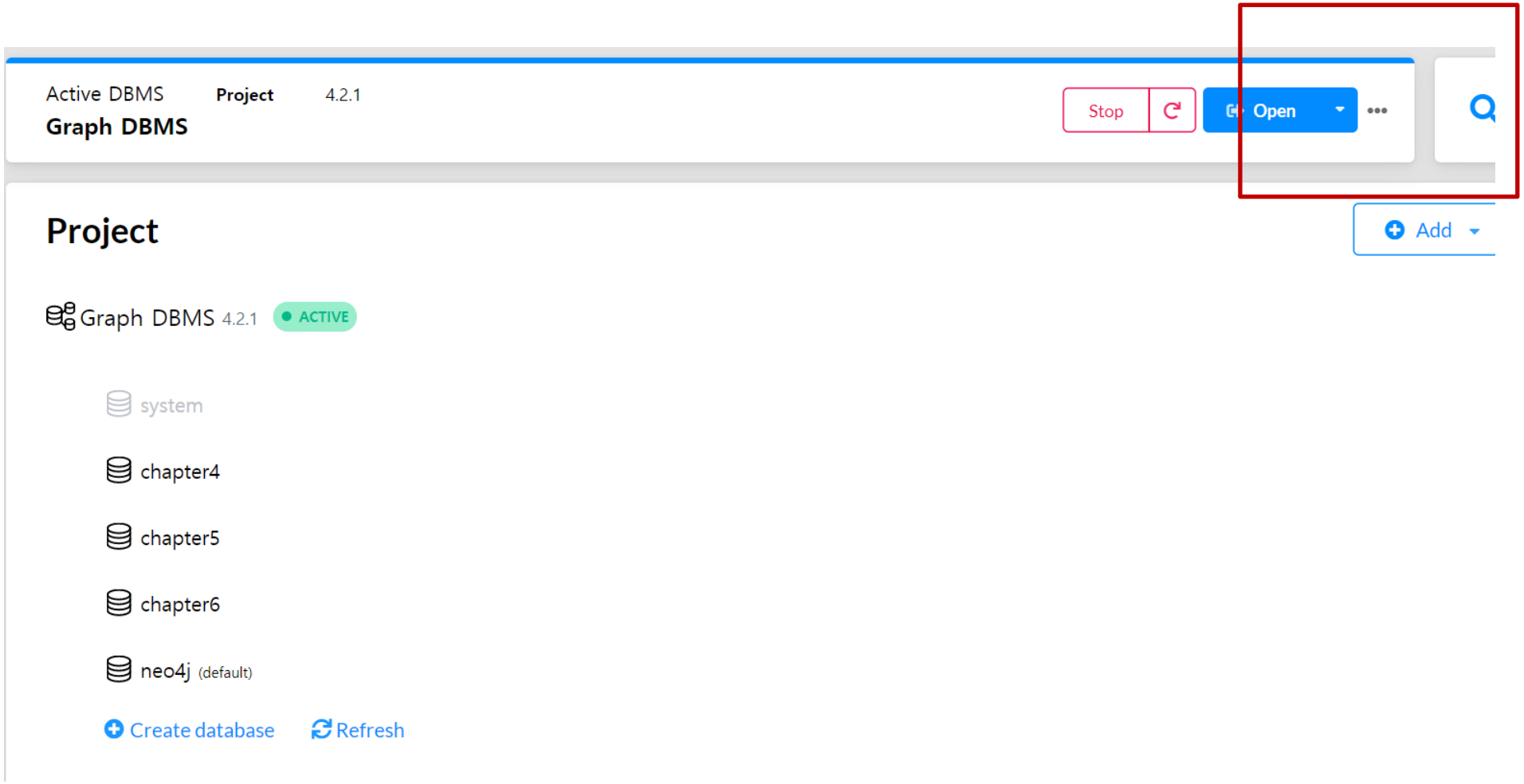
Connected as

Username: neo4j
Roles: admin, PUBLIC
Admin: :server user list
:server user add
Disconnect: :server disconnect

DBMS

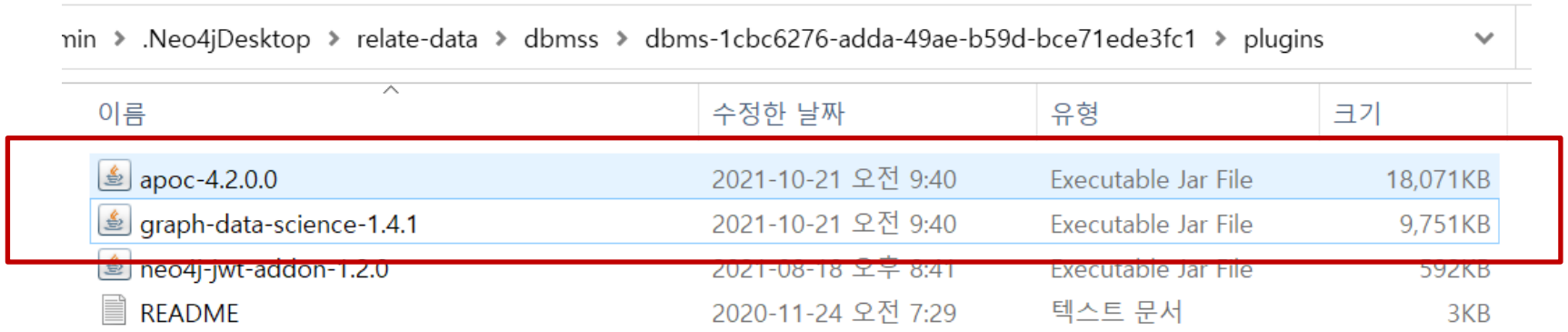
Version: 4.2.1
Edition: Enterprise
Name: chapter4
Databases: :dbs
Information: :sysinfo
Query List: :queries

Settings


Active DBMS
Project
4.2.1
Graph DBMS
Stop
Open
Project
Add
Graph DBMS 4.2.1
ACTIVE
system
chapter4
chapter5
chapter6
neo4j (default)
Create database
Refresh

Settings

- Downloads files
  - https://github.com/chunsejin/KGProjects_DAU/raw/master/jar



Settings

Edit settings

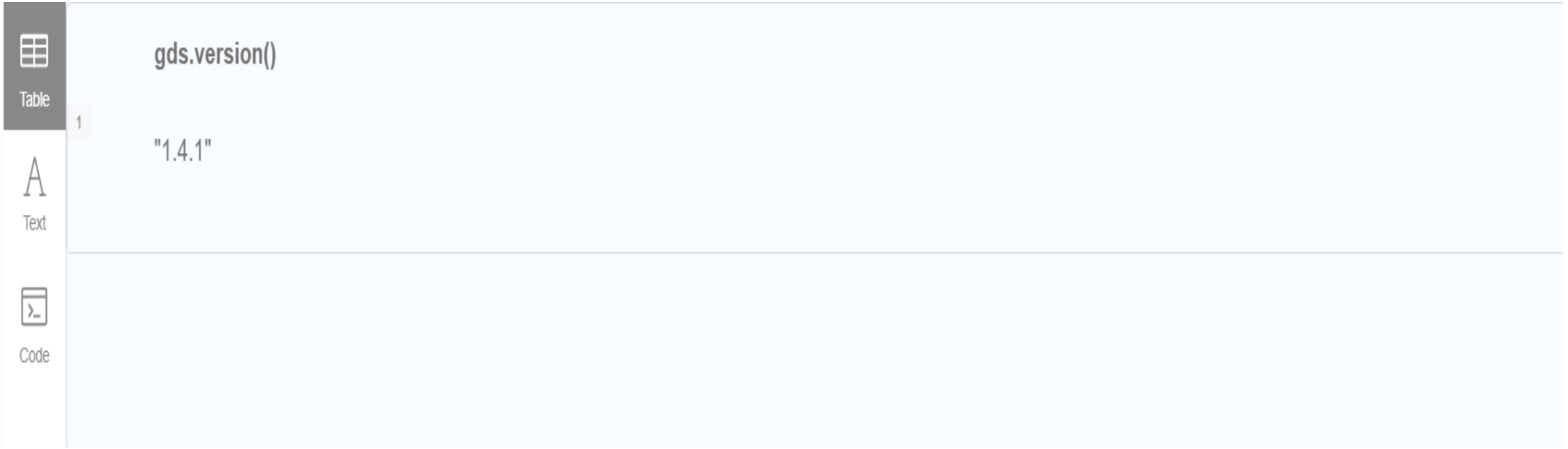
unrestricted

```
# this way will leave more bandwidth in the IO subsystem to service random-read IOs,
# which is important for the response time of queries when the database cannot fit
# entirely in memory. The only drawback of this setting is that longer checkpoint times
# may lead to slightly longer recovery times in case of a database or system crash.
# A lower number means lower IO pressure, and consequently longer checkpoint times.
# The configuration can also be commented out to remove the limitation entirely, and
# let the checkpointer flush data as fast as the hardware will go.
# Set this to -1 to disable the IOPS limit.
# dbms.checkpoint.iops.limit=600

# Only allow read operations from this Neo4j instance. This mode still requires
# write access to the directory for lock purposes.
#dbms.read_only=false

# Comma separated list of JAX-RS packages containing JAX-RS resources, one
# package name for each mountpoint. The listed package names will be loaded
# under the mountpoints specified. Uncomment this line to mount the
# org.neo4j.examples.server.unmanaged.HelloWorldResource.java from
# neo4j-server-examples under /examples/unmanaged, resulting in a final URL of
# http://localhost:7474/examples/unmanaged/helloworld/{nodeId}
#dbms.unmanaged_extension_classes=org.neo4j.examples.server.unmanaged=/examples/unmanaged

# A comma separated list of procedures and user defined functions that are allowed
# full access to the database through unsupported/insecure internal APIs.
dbms.security.procedures.unrestricted=jwt.security.*,apoc.*,gds.*

# A comma separated list of procedures to be loaded by default.
# Leaving this unconfigured will load all procedures found.
```

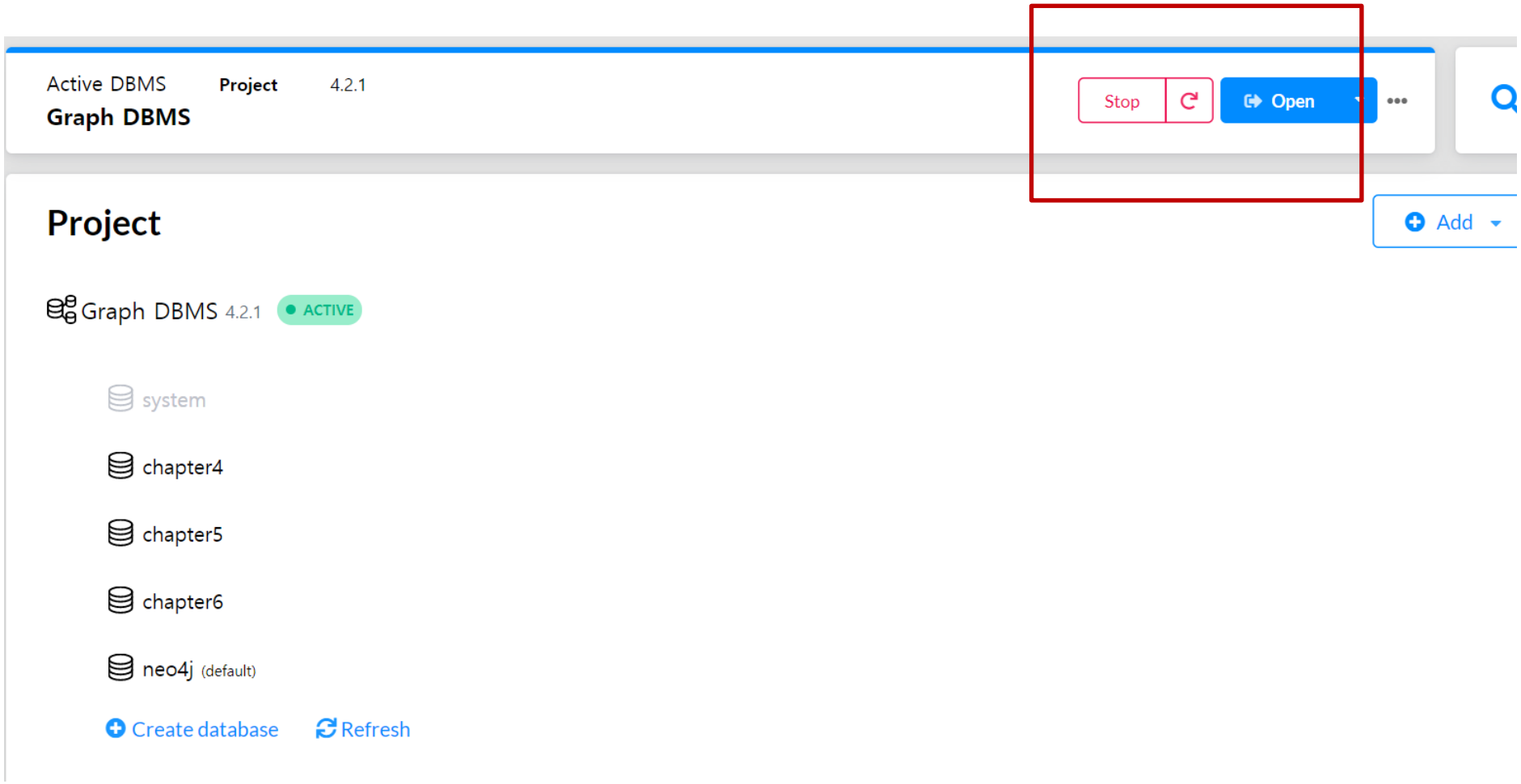
Reset to defaults | Undo | Apply | Close

Check

- RETURN gds.version()



Settings: Restart



Shortest Path with Neo4j

- startNode
- endNode
- nodeProjection
  - 특정 노드의 일부를 in-memory 그래프에 매핑
- relationshipProjection
  - 특정 관계의 일부를 in-memory 그래프 에 매핑
- relationWeightProperty
  - 노드의 쌍 사이에 순환 비용

암스테르담 부터 런던까지 최단거리 경로 계산

```
MATCH (source:Place {id: "Amsterdam"}),
      (destination:Place {id: "London"})

CALL gds.alpha.shortestPath.stream({
  startNode: source,
  endNode: destination,
  nodeProjection: "*",
  relationshipProjection: {
    all: {
      type: "*",
      properties: "distance",
      orientation:  "UNDIRECTED"
    }
  },
  relationshipWeightProperty: "distance"
})
YIELD nodeId, cost
RETURN gds.util.asNode(nodeId).id AS place, cost;
```

place	cost
Amsterdam	0.0
Immingham	1.0
Doncaster	2.0
London	3.0

가중치가 없는 최단경로 계산

```
MATCH (source:Place {id: "Amsterdam"}),
    (destination:Place {id: "London"})
CALL gds.alpha.shortestPath.astar.stream({
    startNode: source,
    endNode: destination,
    nodeProjection: {
        Place: {
            properties: ['longitude', 'latitude']
        }
    },
    relationshipProjection: {
        all: {
            type: "*",
            properties: "distance",
            orientation: "UNDIRECTED"
        }
    },
        relationshipWeightProperty: "distance",
        propertyKeyLat: "latitude",
        propertyKeyLon: "longitude"
})
YIELD nodeId, cost
RETURN gds.util.asNode(nodeId).id AS place, cost;
```

place	cost
Amsterdam	0.0
Immingham	369.0
Doncaster	443.0
London	720.0

가중치가 포함된 최단거리 경로

```
MATCH (start:Place {id:"Gouda"}),
      (end:Place {id:"Felixstowe"})

CALL gds.alpha.kShortestPaths.stream({
  startNode: start,
  endNode: end,
  nodeProjection: "*",
  relationshipProjection: {
    all: {
      type: "*",
      properties: "distance",
      orientation:  "UNDIRECTED"
    }
  },
  relationshipWeightProperty: "distance",
  k: 5
})

YIELD index, sourceNodeId, targetNodeId, nodeIds, costs, path
RETURN index,
       [node in gds.util.asNodes(nodeIds[1..-1]) | node.id] AS via,
       reduce(acc=0.0, cost in costs | acc + cost) AS totalCost;
```

place	cost
Amsterdam	0.0
Den Haag	59.0
Hoek van Holland	86.0
Felixstowe	293.0
Ipswich	315.0
Colchester	347.0
London	453.0

A* with Neo4j

- startNode
- endNode
- nodeProjection
- relationshipProjection
- relationshipWeightProperty
- propertyKeyLat
  - 위도 정보에 대해서 계산 (geospatial)
- propertyKeyLon
  - 경도 정보에 대해서 계산 (geospatial)

Den Haag와 London 사이의 거리 계산

```
MATCH (source:Place {id: "Den Haag"}),
(destination:Place {id: "London"})

CALL gds.alpha.shortestPath.astar.stream({
startNode: source,
endNode: destination,
nodeProjection: {
        Place: {
            properties: ['longitude', 'latitude']
        }
    },
relationshipProjection: {
all: {
type: "*",
properties: "distance",
orientation: "UNDIRECTED"
}
},
relationshipWeightProperty: "distance",
propertyKeyLat: "latitude",
propertyKeyLon: "longitude"
})
YIELD nodeId, cost
RETURN gds.util.asNode(nodeId).id AS place, cost;
```

place	cost
Den Haag	0.0
Hoek van Holland	27.0
Felixstowe	234.0
Ipswich	256.0
Colchester	288.0
London	394.0

Yen's with Neo4j

- startNode
- endNode
- nodeProjection
- relationshipProjection
- relationshipWeightProperty
- k
  - The maximum number of shortest paths to find

Yen's with Neo4j

```
MATCH (start:Place {id:"Gouda"}),
      (end:Place {id:"Felixstowe"})

CALL gds.alpha.kShortestPaths.stream({
  startNode: start,
  endNode: end,
  nodeProjection: "*",
  relationshipProjection: {
    all: {
      type: "*",
      properties: "distance",
      orientation:  "UNDIRECTED"
    }
  },
  relationshipWeightProperty: "distance",
  k: 5
})

YIELD index, sourceNodeId, targetNodeId, nodeIds, costs, path
RETURN index,
       [node in gds.util.asNodes(nodeIds[1..-1]) | node.id] AS via,
       reduce(acc=0.0, cost in costs | acc + cost) AS totalCost;
```

index	via	totalCost
0	[Rotterdam, Hoek van Holland]	265.0
1	[Den Haag, Hoek van Holland]	266.0
2	[Rotterdam, Den Haag, Hoek van Holland]	285.0
3	[Den Haag, Rotterdam, Hoek van Holland]	298.0
4	[Utrecht, Amsterdam, Den Haag, Hoek van Holland]	374.0

All Pairs Shortest Path with Neo4j

- APSP 알고리즘에 대한 병렬 구현
  - nodeProjection
  - relationshipProjection
  - relationshipWeightProperty

All Pairs Shortest Path with Neo4j

```
CALL gds.alpha.allShortestPaths.stream({
  nodeProjection: "*",
  relationshipProjection: {
    all: {
      type: "*",
      properties: "distance",
      orientation: "UNDIRECTED"
    }
  }
})
YIELD sourceNodeId, targetNodeId, distance
WHERE sourceNodeId < targetNodeId
RETURN gds.util.asNode(sourceNodeId).id AS source,
       gds.util.asNode(targetNodeId).id AS target,
       distance
ORDER BY distance DESC
LIMIT 10;
```

source	target	distance
Colchester	Utrecht	5.0
London	Rotterdam	5.0
London	Gouda	5.0
Ipswich	Utrecht	5.0
Colchester	Gouda	5.0
Colchester	Den Haag	4.0
London	Utrecht	4.0
London	Den Haag	4.0
Colchester	Amsterdam	4.0
Ipswich	Gouda	4.0

All Pairs Shortest Path with Neo4j

```
CALL gds.alpha.allShortestPaths.stream({
  nodeProjection: "*",
  relationshipProjection: {
    all: {
      type: "*",
      properties: "distance",
      orientation: "UNDIRECTED"
    }
  },
  relationshipWeightProperty: "distance"
})
YIELD sourceNodeId, targetNodeId, distance
WHERE sourceNodeId < targetNodeId
RETURN gds.util.asNode(sourceNodeId).id AS source,
       gds.util.asNode(targetNodeId).id AS target,
       distance
ORDER BY distance DESC
LIMIT 10;
```

source	target	distance
Doncaster	Hoek van Holland	529.0
Rotterdam	Doncaster	528.0
Gouda	Doncaster	524.0
Felixstowe	Immingham	511.0
Den Haag	Doncaster	502.0
Ipswich	Immingham	489.0
Utrecht	Doncaster	489.0
London	Utrecht	460.0
Colchester	Immingham	457.0
Immingham	Hoek van Holland	455.0

Single Source Shortest Path with Neo4j

- startNode
- nodeProjection
- relationshipProjection
- relationshipWeightProperty
- delta
  - 동시성(Concurrency)의 수준

Single Source Shortest Path with Neo4j

- Delta-Stepping Algorithm: 병렬 실행이 되도록 다익스트라 알고리즘을 페이즈(Phase)의 수로 나눈 방법

```
MATCH (n:Place {id:"London"})
CALL gds.alpha.shortestPath.deltaStepping.stream({
  startNode: n,
  nodeProjection: "*",
  relationshipProjection: {
    all: {
      type: "*",
      properties: "distance",
      orientation:  "UNDIRECTED"
    }
  },
  relationshipWeightProperty: "distance",
  delta: 1.0
})
YIELD nodeId, distance
WHERE gds.util.isFinite(distance)
RETURN gds.util.asNode(nodeId).id AS destination, distance
ORDER BY distance;
```

destination	distance
London	0.0
Colchester	106.0
Ipswich	138.0
Felixstowe	160.0
Doncaster	277.0
Immingham	351.0
Hoek van Holland	367.0
Den Haag	394.0
Rotterdam	400.0
Gouda	425.0
Amsterdam	453.0
Utrecht	460.0

Minimum Spanning Tree with Neo4j

- startNodeId
- nodeProjection
- relationshipProjection
- relationshipWeightProperty
- writeProperty
  - 결과로 쓰여질 관계타입의 이름
- weightWriteProperty
  - writeProperty 관계에서 가중치 속성의 이름

Minimum Spanning Tree with Neo4j

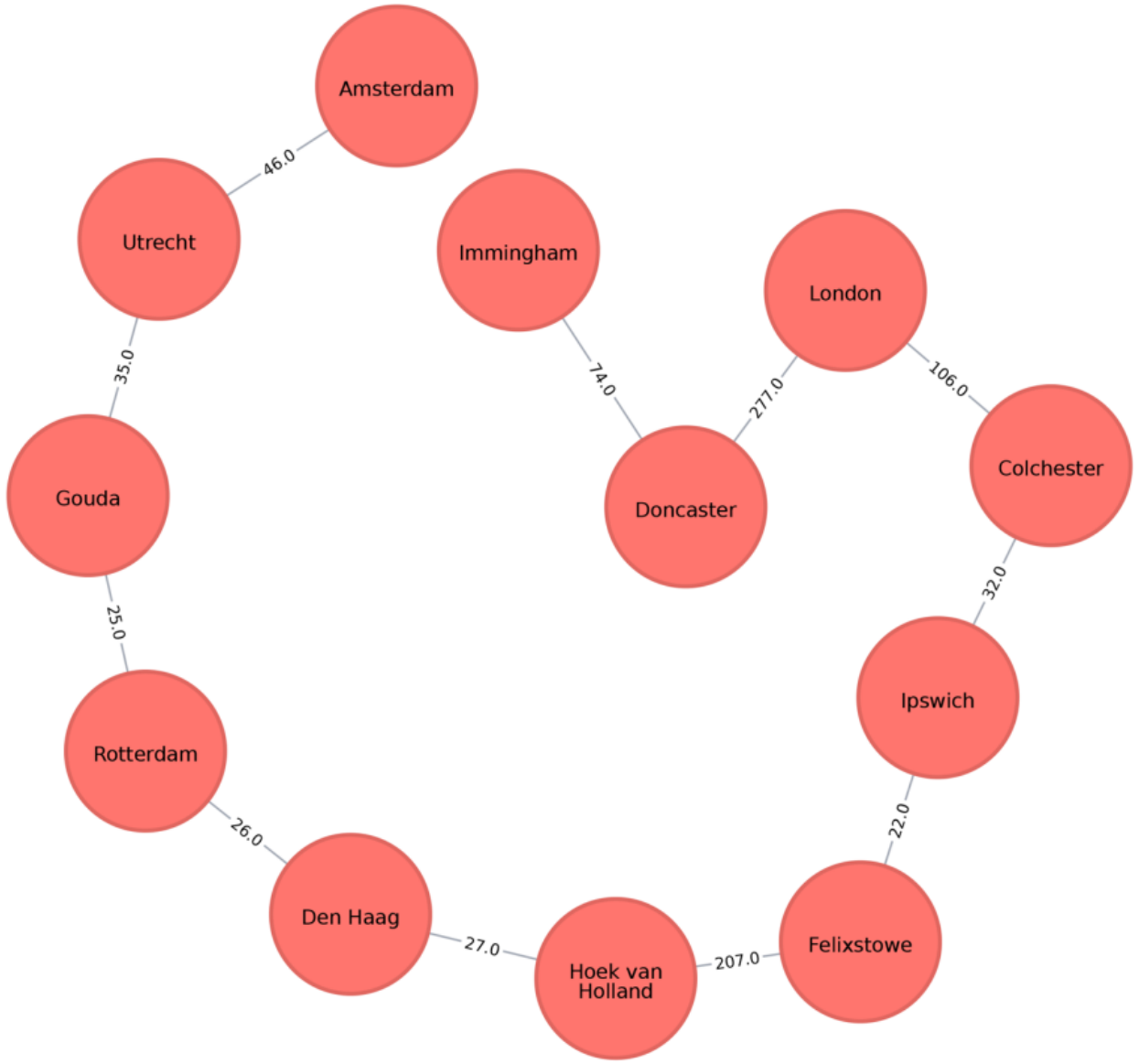
```
MATCH (n:Place {id:"Amsterdam"})
CALL gds.alpha.spanningTree.minimum.write({
  startNodeId: id(n),
  nodeProjection: "*",
  relationshipProjection: {
    EROAD: {
      type: "EROAD",
      properties: "distance",
      orientation:  "UNDIRECTED"
    }
  },
  relationshipWeightProperty: "distance",
  writeProperty: 'MINST',
  weightWriteProperty: 'cost'
})
YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN createMillis, computeMillis, writeMillis, effectiveNodeCount;
```

Minimum Spanning Tree with Neo4j

```
MATCH path = (n:Place {id:"Amsterdam"})-[:MINST*]-()
WITH relationships(path) AS rels
UNWIND rels AS rel
WITH DISTINCT rel AS rel
RETURN startNode(rel).id AS source,
       endNode(rel).id AS destination,
       rel.cost AS cost;
```

source	destination	cost
Amsterdam	Utrecht	46.0
Utrecht	Gouda	35.0
Gouda	Rotterdam	25.0
Rotterdam	Den Haag	26.0
Den Haag	Hoek van Holland	27.0
Hoek van Holland	Felixstowe	207.0
Felixstowe	Ipswich	22.0
Ipswich	Colchester	32.0
Colchester	London	106.0
London	Doncaster	277.0
Doncaster	Immingham	74.0

Minimum Spanning Tree with Neo4j



Random Walk with Neo4j

- 알고리즘 선택
  - Random
  - Node2vec
- Random Walk
  - Start
  - nodeProjection
  - relationshipProjection
  - steps
    - 이동의 수
  - walks
    - 반환되는 경로의 수

Random Walk with Neo4j

```
MATCH (source:Place {id: "London"})
CALL gds.alpha.randomWalk.stream({
  start: id(source),
  nodeProjection: "*",
  relationshipProjection: {
    all: {
      type: "*",
      properties: "distance",
      orientation:  "UNDIRECTED"
    }
  },
  steps: 5,
  walks: 1
})
YIELD nodeIds
UNWIND gds.util.asNodes(nodeIds) as place
RETURN place.id AS place
```

place
London
Doncaster
Immingham
Amsterdam
Utrecht
Amsterdam

Random Walk with Neo4j

WORLD CITIES

- Settings >> Java heap
  - 512m -> 4G

Import All World Cities into Neo4j

- World cities를 Neo4j 에 새로운 데이터베이스 Import
  - City_ascii를 id로
    lat, lon, country, population은 추가할 것

```
WITH "https://github.com/chunsejin/KGProjects_DAU/raw/master/data/" AS base
WITH base + "worldcities.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
```

```
CREATE CONSTRAINT node_id ON (n:City) ASSERT n.id IS UNIQUE
```

모든 노드간의 거리관계 생성하기

```
MATCH (c:City { name: “Busan”)
WITH point({latitude: c1.lat, longitude: c1.lon}) as p1,
        point({latitude: c2.lat, longitude: c2.lon}) as p2
WHERE c1 <> c2 and c1.country <> c2.country
MERGE ??????????????
```

distance(p1, p2) 는 좌표 값 사이의 거리를 계산

Again

- Shortest Path
- ASPS
- SSSP
- Random Walk
- Minimum Spanning Tree

Contents

- PATHFINDING AND GRAPH SEARCH ALGORITHMS WITH APACHE SPARK
- PATHFINDING AND GRAPH SEARCH ALGORITHMS WITH NEO4J

PATHFINDING AND GRAPH SEARCH ALGORITHMS WITH APACHE SPARK

Importing the Data into Apache Spark

- Spark와 GraphFrame을 사용

```
from pyspark.sql.types import *
from graphframes import *
```

Importing the Data into Apache Spark

- CSV로부터 GraphFrame을 생성

```
def create_transport_graph():
    node_fields = [
        StructField("id", StringType(), True),
        StructField("latitude", FloatType(), True),
        StructField("longitude", FloatType(), True),
        StructField("population", IntegerType(), True)
    ]
    nodes = spark.read.csv("data/transport-nodes.csv", header=True,
                           schema=StructType(node_fields))

    rels = spark.read.csv("data/transport-relationships.csv", header=True)
    reversed_rels = (rels.withColumn("newSrc", rels.dst)
                     .withColumn("newDst", rels.src)
                     .drop("dst", "src")
                     .withColumnRenamed("newSrc", "src")
                     .withColumnRenamed("newDst", "dst")
                     .select("src", "dst", "relationship", "cost"))

    relationships = rels.union(reversed_rels)

    return GraphFrame(nodes, relationships)
```

Importing the Data into Apache Spark

- 함수 create_transport_graph() 호출

```
g = create_transport_graph()
```

BFS with Apache Spark

```
(g.vertices
.filter("population > 100000 and population < 300000")
.sort("population")
.show())
```

id	latitude	longitude	population
Colchester	51.88921	0.90421	104390
Ipswich	52.05917	1.15545	133384

BFS with Apache Spark

- Den Hagg로부터 중간규모의 도시까지 최단거리

```
from_expr = "id='Den Haag'"
to_expr = "population > 100000 and population < 300000 and id <> 'Den Haag'"
result = g.bfs(from_expr, to_expr)

print(result.columns)
```

```
['from', 'e0', 'v1', 'e1', 'v2', 'e2', 'to']
```

BFS with Apache Spark

- e로 시작하는 컬럼들을 필터링

```
columns = [column for column in result.columns if not column.startswith("e")]
result.select(columns).show()
```

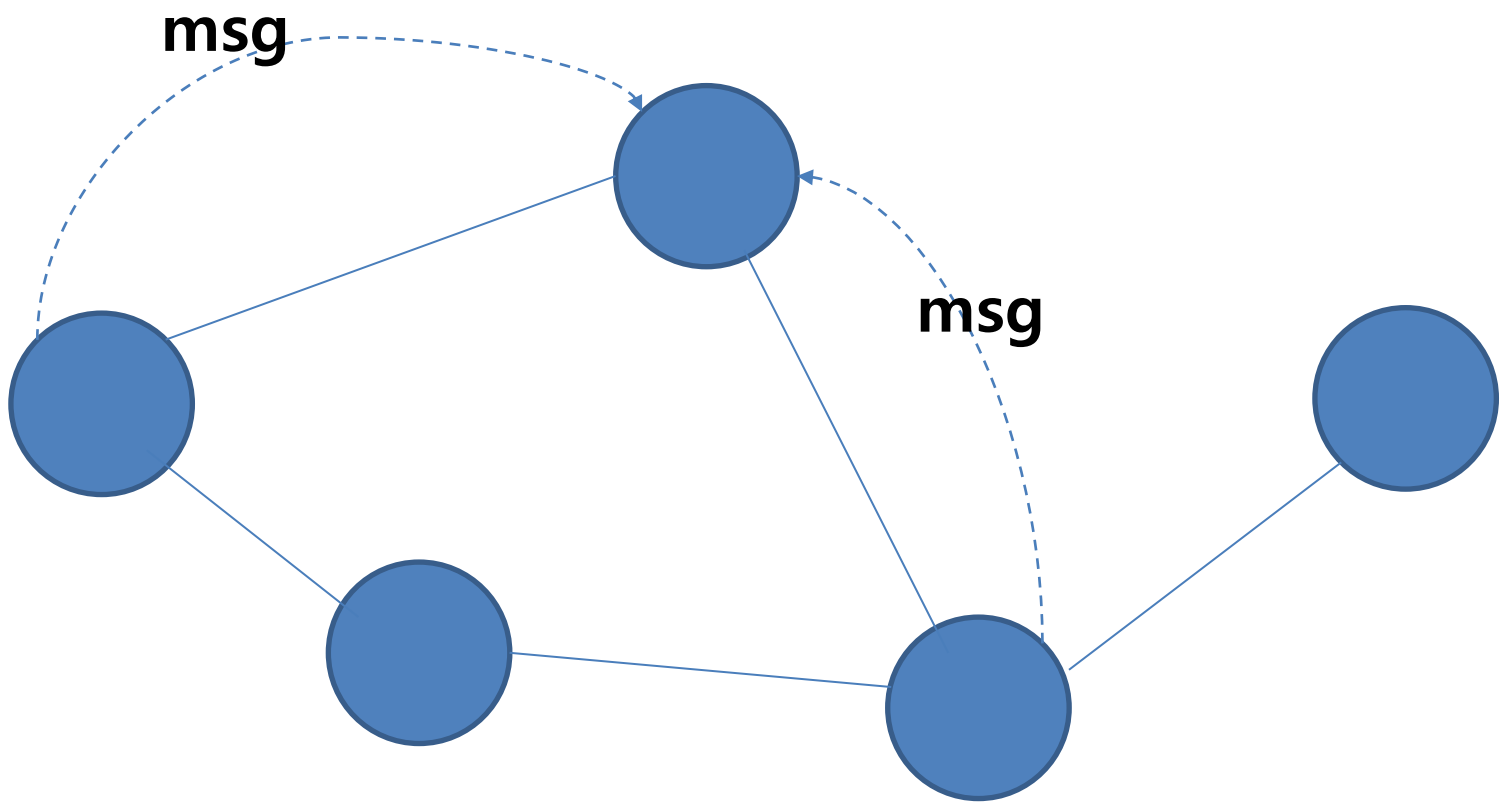
from	v1	v2	to
[Den Haag, 52.078…	[Hoek van Holland…	[Felixstowe, 51.9…	[Ipswich, 52.0591…

Shortest Path (Weighted)

```
from graphframes.lib import AggregateMessages as AM
from pyspark.sql import functions as F
```

Message passing via AggregateMessages(SPARK 특징)

- aggregateMessages: Send messages between vertices, and aggregate messages for each vertex



Shortest Path (Weighted)

- user define function 설정(udf): 출발지와 목적지간의 경로

```
add_path_udf = F.udf(lambda path, id: path + [id], ArrayType(StringType()))
```

Shortest Path (Weighted)

- 초기값, 출발지 기반 경로 레퍼런스를 설정

```
def shortest_path(g, origin, destination, column_name="cost"):
    if g.vertices.filter(g.vertices.id == destination).count() == 0:
        return (spark.createDataFrame(sc.emptyRDD(), g.vertices.schema)
                .withColumn("path", F.array()))

    vertices = (g.vertices.withColumn("visited", F.lit(False))
                .withColumn("distance", F.when(g.vertices["id"] == origin, 0)
                                         .otherwise(float("inf")))
                .withColumn("path", F.array()))
    cached_vertices = AM.getCachedDataFrame(vertices)
    g2 = GraphFrame(cached_vertices, g.edges)
```

Shortest Path (Weighted)

- 다음 노드을 탐색하면서 경로에 추가
  - Msg를 통해서 관련 경로 및 새로운 비용을 알립니다.

```
while g2.vertices.filter('visited == False').first():
    current_node_id = g2.vertices.filter('visited == False').sort
                                        ("distance").first().id

    msg_distance = AM.edge[column_name] + AM.src['distance']
    msg_path = add_path_udf(AM.src["path"], AM.src["id"])
    msg_for_dst = F.when(AM.src['id'] == current_node_id,
                         F.struct(msg_distance, msg_path))
    new_distances = g2.aggregateMessages(F.min(AM.msg).alias("aggMess"),
                                         sendToDst=msg_for_dst)

    new_visited_col = F.when(
        g2.vertices.visited | (g2.vertices.id == current_node_id),
                                           True).otherwise(False)
```

Shortest Path (Weighted)

- AM.getCachedDataFrame을 통해 경로 레퍼런스 저장

```
new_distance_col = F.when(new_distances["aggMess"].isNotNull() &
                          (new_distances.aggMess["col1"]
                          < g2.vertices.distance),
                          new_distances.aggMess["col1"])
                          .otherwise(g2.vertices.distance)
new_path_col = F.when(new_distances["aggMess"].isNotNull() &
               (new_distances.aggMess["col1"]
               < g2.vertices.distance), new_distances.aggMess["col2"]
               .cast("array<string>")).otherwise(g2.vertices.path)

new_vertices = (g2.vertices.join(new_distances, on="id",
                                 how="left_outer")
                .drop(new_distances["id"])
                .withColumn("visited", new_visited_col)
                .withColumn("newDistance", new_distance_col)
                .withColumn("newPath", new_path_col)
                .drop("aggMess", "distance", "path")
                .withColumnRenamed('newDistance', 'distance')
                .withColumnRenamed('newPath', 'path'))
cached_new_vertices = AM.getCachedDataFrame(new_vertices)
g2 = GraphFrame(cached_new_vertices, g2.edges)
```

Shortest Path (Weighted)

- 목적지에 도착하면 최종 경로를 반환
  - Resilient Distributed Datasets(RDD): 그래프 데이터가 한 곳에 저장된 것이 아니다. 나누어 저장되어 있다. 따라서 경로정보 업데이트를위한 메시지 교환이 필요하다. -> aggregateMessages

```
    if g2.vertices.filter(g2.vertices.id == destination).first().visited:
        return (g2.vertices.filter(g2.vertices.id == destination)
                .withColumn("newPath", add_path_udf("path", "id"))
                .drop("visited", "path")
                .withColumnRenamed("newPath", "path"))
return (spark.createDataFrame(sc.emptyRDD(), g.vertices.schema)
        .withColumn("path", F.array()))
```

All Pairs Shortest Path with Apache Spark

- 랜드마크(landmakrs)에 대한 최단경로 탐색
  - g.shortestPaths(landmarks=[ ])

```
result = g.shortestPaths(["Colchester", "Immingham", "Hoek van Holland"])
result.sort(["id"]).select("id", "distances").show(truncate=False)
```

Single Source Shortest Path with Apache Spark

- Spark's aggregateMessages framework를 사용함

```
from graphframes.lib import AggregateMessages as AM
from pyspark.sql import functions as F
```

Single Source Shortest Path with Apache Spark

- 경로 추가를 위한 udf 사용

```
add_path_udf = F.udf(lambda path, id: path + [id], ArrayType(StringType()))
```

Single Source Shortest Path with Apache Spark

- 경로 추가를 위한 udf 사용

```
def sssp(g, origin, column_name="cost"):
    vertices = g.vertices \
        .withColumn("visited", F.lit(False)) \
        .withColumn("distance",
            F.when(g.vertices["id"] == origin, 0).otherwise(float("inf"))) \
        .withColumn("path", F.array())
    cached_vertices = AM.getCachedDataFrame(vertices)
    g2 = GraphFrame(cached_vertices, g.edges)
```

Single Source Shortest Path with Apache Spark

- 경로 추가를 위한 udf 사용

```
while g2.vertices.filter('visited == False').first():
    current_node_id = g2.vertices.filter('visited == False')
                          .sort("distance").first().id

    msg_distance = AM.edge[column_name] + AM.src['distance']
    msg_path = add_path_udf(AM.src["path"], AM.src["id"])
    msg_for_dst = F.when(AM.src['id'] == current_node_id,
                  F.struct(msg_distance, msg_path))
    new_distances = g2.aggregateMessages(
        F.min(AM.msg).alias("aggMess"), sendToDst=msg_for_dst)

    new_visited_col = F.when(
        g2.vertices.visited | (g2.vertices.id == current_node_id),
                              True).otherwise(False)
    new_distance_col = F.when(new_distances["aggMess"].isNotNull() &
                              (new_distances.aggMess["col1"] <
                              g2.vertices.distance),
                              new_distances.aggMess["col1"]) \
                              .otherwise(g2.vertices.distance)
    new_path_col = F.when(new_distances["aggMess"].isNotNull() &
                          (new_distances.aggMess["col1"] <
                          g2.vertices.distance),
                          new_distances.aggMess["col2"]
                          .cast("array<string>")) \
                          .otherwise(g2.vertices.path)
```

Single Source Shortest Path with Apache Spark

```
        new_vertices = g2.vertices.join(new_distances, on="id",
                                        how="left_outer") \
            .drop(new_distances["id"]) \
            .withColumn("visited", new_visited_col) \
            .withColumn("newDistance", new_distance_col) \
            .withColumn("newPath", new_path_col) \
            .drop("aggMess", "distance", "path") \
            .withColumnRenamed('newDistance', 'distance') \
            .withColumnRenamed('newPath', 'path')
        cached_new_vertices = AM.getCachedDataFrame(new_vertices)
        g2 = GraphFrame(cached_new_vertices, g2.edges)

    return g2.vertices \
                .withColumn("newPath", add_path_udf("path", "id")) \
                .drop("visited", "path") \
                .withColumnRenamed("newPath", "path")
```

Single Source Shortest Path with Apache Spark

```
result = sssp(g, "Amsterdam", "cost")
(result
 .withColumn("via", via_udf("path"))
 .select("id", "distance", "via")
 .sort("distance")
 .show(truncate=False))
```

PATHFINDING AND GRAPH SEARCH ALGORITHMS WITH COLAB

RANDOM WALK

```
# 2D random walk.
import numpy
import pylab
import random

# 스텝의 수
n = 100000

# X와 Y에 대한 0으로 채워진 공간 생성
x = numpy.zeros(n)
y = numpy.zeros(n)
```

RANDOM WALK

RIGHT = 1

LEFT = 2

UP = 3

DOWN = 4

RANDOM WALK

```
for i in range(1, n):
    val = random.randint(1, 4)
    if val == RIGHT:
        x[i] = x[i - 1] + 1
        y[i] = y[i - 1]
    elif val == LEFT:
        x[i] = x[i - 1] - 1
        y[i] = y[i - 1]
    elif val == UP:
        x[i] = x[i - 1]
        y[i] = y[i - 1] + 1
    else: # DOWN
        x[i] = x[i - 1]
        y[i] = y[i - 1] - 1
```

RANDOM WALK

```
# plotting
pylab.title("Random Walk ($n = " + str(n) + "$ steps)")
pylab.plot(x, y)
pylab.savefig("rand_walk"+str(n)+".png",bbox_inches="tigh
t",dpi=600)
pylab.show()
```

PATHFINDING AND GRAPH SEARCH ALGORITHMS WITH OSMNX

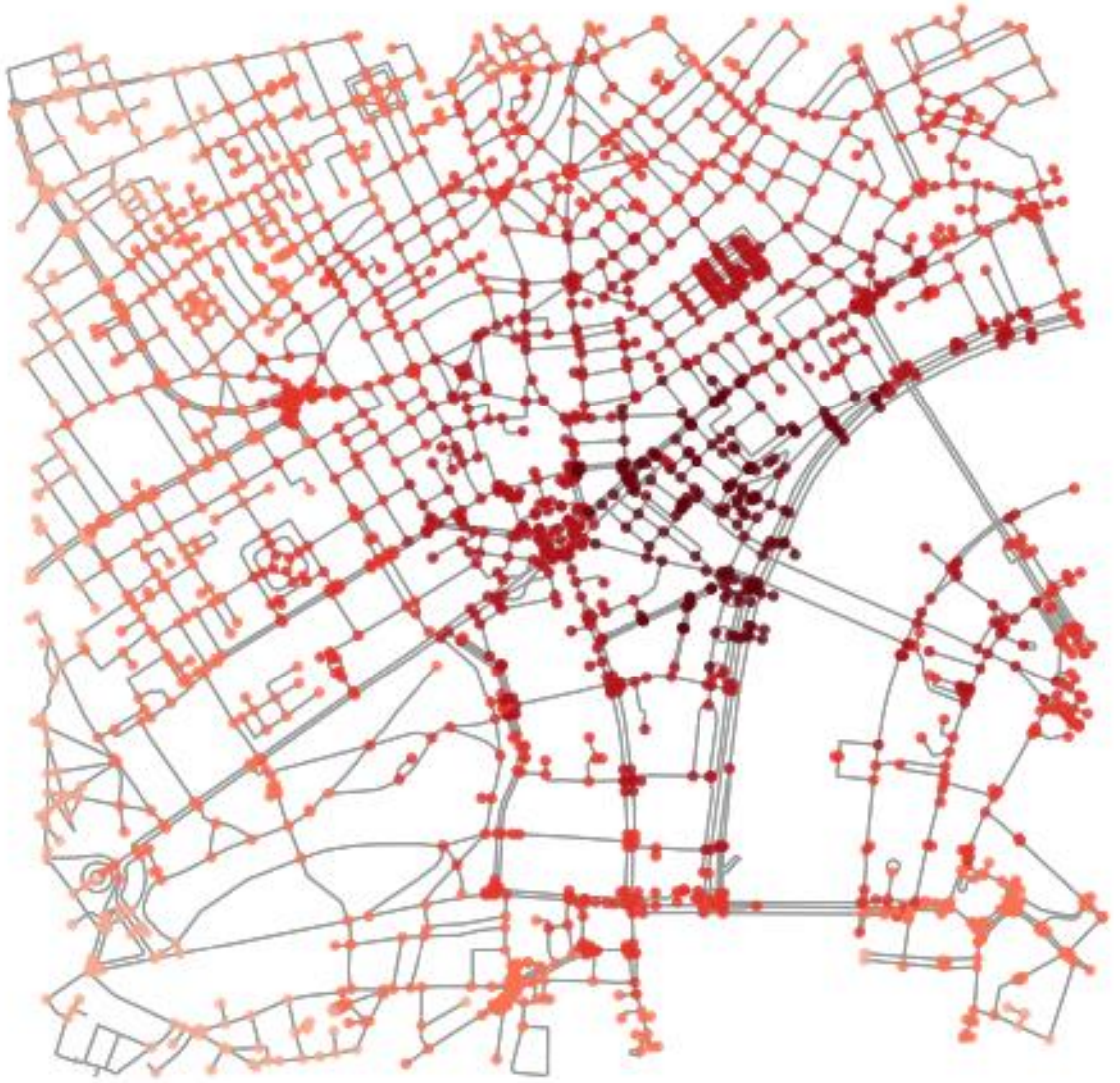
OSMNX: OpenStreetMap + NetworkX

- 주어진 지도를
  그래프 모델로 모델링
  - Nodes: 교차점
  - Edges: 도로

Singapore's Raffles Place MRT

OSMNX: OpenStreetMap + NetworkX

- 왜 좋은가?
  - 도시 계획/문제점 분석 시 매우 유용함
  - 노드/엣지 중심으로 분석
    - 데이터 수집/매핑/계산/분석을 노드(OR 노드 그룹) 위에서 수행
  - 예) 해운대구 물류망 분석
    - 노드: 해운대구 내 소속되는 노드들
    - 엣지: 해당 노드들에 관련된 엣지
    - 가중치에 대한 고려
      - 노드의 중요 값, 엣지에 대한 비용



Central London

Code: Setup

```
!pip install osmnx
!python -m pip uninstall matplotlib
!pip install matplotlib==3.1.3
```

matplotlib==3.1.3 버전에 맞춰야함

```
import networkx as nx
import osmnx as ox
import matplotlib.pyplot as plt
```

Code: Setup

1.ox.graph_from_place: 지역명
2.ox.graph_from_address: 주소정보
3.ox.graph_from_point: 위경도정보
4.ox.graph_from_bbox: 동서남북좌표

network_type: ‘walk’, ‘bike’, ‘drive’, ‘drive_se
rvice’, ‘all’, or ‘all_private’ 중 하나

```
# download/model a street network for some city then visualize it
place = "Saha-gu, Busan, Korea"
G = ox.graph_from_place(place, network_type="drive")
fig, ax = ox.plot_graph(G)
```

매우 중요한 건 G는 그래프 모델이라는 점

Codes: 도로타입 표시하기

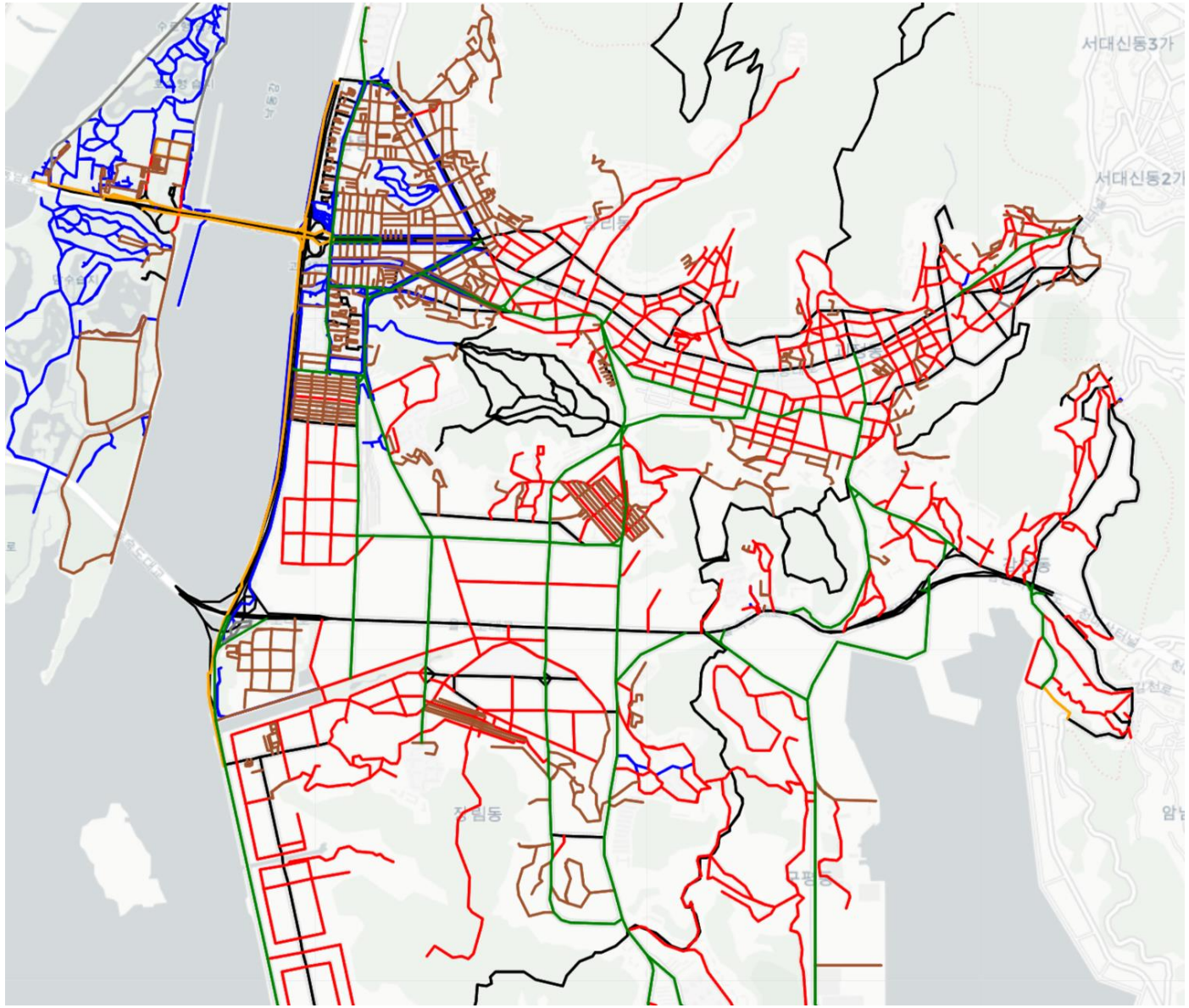
```
# edge의 타입 따라서 도로에 대한 색깔 다르게 표시하기
hwy_colors = {'footway': 'skyblue',
              'residential': 'paleturquoise',
              'cycleway’: lightgreen',
              'service': 'sienna',
              'living street': ’orange',
              'secondary': 'black',
              'pedestrian': 'lightskyblue'}
```

Codes: 도로타입 표시하기

```
# 엣지를 탐색
def find_edges(G, hwys):
    edges = []
    for u, v, k, data in G.edges(keys=True, data='highway'):
        check1 = isinstance(data, str) and data not in hwys
        check2 = isinstance(data, list) and all([d not in hwys for d in data])
        if check1 or check2:
            edges.append((u, v, k))
    return set(edges)
```

Codes: 도로타입 표시하기

```
# hwy 이 정해지지 않은 경우는 black
G_tmp = G.copy()
G_tmp.remove_edges_from(G.edges - find_edges(G, hwy_colors.keys()))
m = ox.plot_graph_folium(G_tmp, popup_attribute='highway', weight=5, color='black')

# 지도 위에서 Hwy_colors에서 정해진 타입에 따라 추가적인 edge를 그림
for hwy, color in hwy_colors.items():
    G_tmp = G.copy()
    G_tmp.remove_edges_from(find_edges(G_tmp, [hwy]))
    if G_tmp.edges:
        m = ox.plot_graph_folium(G_tmp,
                                 graph_map=m,
                                 popup_attribute='highway',
                                 weight=5,
                                 color=color)
```


서대신동3가
서대신동2가

Codes: 최단거리

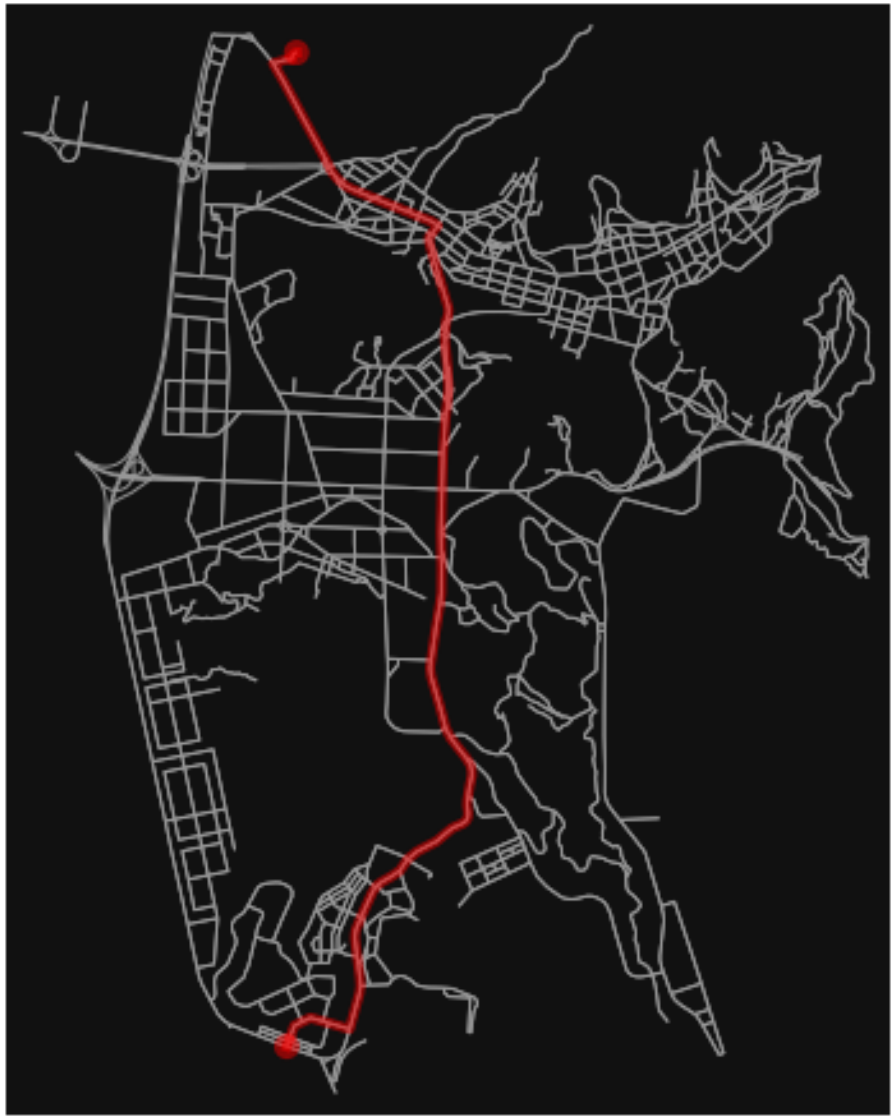
```
# impute missing edge speeds and calculate
edge travel times with the speed module
G = ox.speed.add_edge_speeds(G)
G = ox.speed.add_edge_travel_times(G)
```

Codes: 최단거리

```
# get the nearest network nodes to two lat/lng points with the distance module
orig = ox.distance.nearest_nodes(G, X=128.96755631796773, Y=35.116015941374444)
dest = ox.distance.nearest_nodes(G, X=128.96517223758627, Y=35.046698756214056)
```

Codes: 최단거리

```
# find the shortest path between nodes, minimizing travel time, then plot i
t
route = ox.shortest_path(G, orig, dest, weight="travel_time")
fig, ax = ox.plot_graph_route(G, route, node_size=0)
```

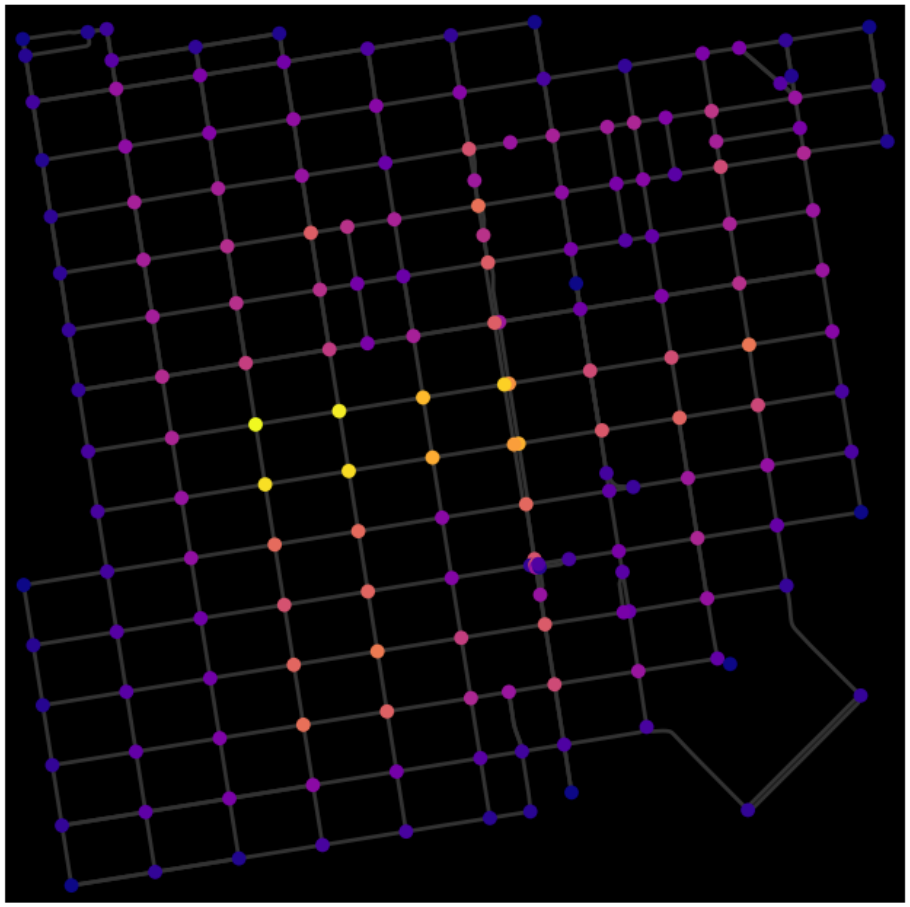


Codes: K-Shortest

```
routes  = ox.k_shortest_paths(G, orig, dest, 3, weight="travel_time")
paths = [r for r in routes]
```

```
# find the k-
shortest paths between nodes, minimizing travel time, then plot it
fig, ax = ox.plot_graph_routes(G, paths, route_colors=['r','b','c']
, route_linewidth=3)
```

그래프를 통한 도시계획 분석: 잠재 상권 분석



entertainments(V):
주위 편의시설

travel_time(E): 이동시간

traffic(V): 주위 교통혼잡

cost(V): 주위 부동산 가격

PotentialCommercialArea(G)
= w1*entertainments(V) + w2*travel_time(E) + w3*traffic(V) + w4*cost(V)

그래프를 통한 도시계획 분석: 동아대 인근

'인근'은 '이웃한 가까운 곳'을
뜻하고 비슷한말로
'가까이, 근방, 근처' 등이 있습니다

추천 원룸·오픈형 원룸
월세 100/15
26㎡·1층
사하구 하단동
ㅁ 대학로최저가 ㅁ 풀옵션방 ㅁ …

추천 원룸·오픈형 원룸
월세 300/38
19㎡·7층
사하구 하단동
🍒하단역 근처에 이런 집이🍒…

추천 원룸·오픈형 원룸
월세 200/20
29㎡·6층
사하구 하단동
✅큰대로변✅가성비갑✅큰방…

추천 원룸·분리형 원룸
월세 300/32
26㎡·1층
사하구 하단동
💟투룸급💟올리모델링💟

추천 원룸·오픈형 원룸
월세 500/40
26㎡·4층
사하구 하단동
💟큰길위치💟놀라운방💟

추천 원룸·오픈형 원룸
월세 100/18
23㎡·2층
사하구 하단동
❎가성비갑❎보증금낮게❎학…

추천 원룸·오픈형 원룸
전세 6,500
14㎡·7층
사하구 하단동
♥첫입주♥동아대인근♥전세…

그래프를 통한 도시계획 분석: 동아대 인근

1분,5분,10분, 20분, 30분내로
걷거나 뛰어갈 수 있는 지역을 찾아보자


추천 원룸·오픈형
월세 300/3
19㎡·7층
사하구 하단동
하단역 근처


추천 원룸·오픈형
월세 200/2
29㎡·6층
사하구 하단동
큰대로변


추천 원룸·분리형
월세 300/3
26㎡·1층
사하구 하단동
투룸급


추천 원룸·오픈형
월세 500/4
26㎡·4층
사하구 하단동
큰길위치


추천 원룸·오픈형
월세 100/1
23㎡·2층
사하구 하단동
가성비갑


추천 원룸·오픈형
전세 6,500
14㎡·7층
사하구 하단동
첫입주 동아대인근 전세…

시간범위와 이동속도

- `trip_times = [1, 5, 10, 20, 30]`

- `travel_speed = 4.5 # (km/hour)`
- `travel_Speed = 13 # (km/hour)` 만약 뛸 때
- `travel_Speed = 40 # (km/hour)` 계주선수라면
- `travel_Speed = 60 # (km/hour)` 자동차 보유 중
- `travel_Speed = 100 # (km/hour)` 레이서 라면

시간당 km -> 분당 meter 로 변환

```
meters_per_minute = travel_speed * 1000 / 60
# km per hour to m per minute
# 4.5 km/h -> 75 m/m
```

Code: Setup

```
!pip install osmnx
!python -m pip uninstall matplotlib
!pip install matplotlib==3.1.3
```

```
import networkx as nx
import osmnx as ox
import matplotlib.pyplot as plt
```

Code: Setup

```
import geopandas as gpd
from descartes import PolygonPatch
from shapely.geometry import LineString
from shapely.geometry import Point
from shapely.geometry import Polygon
```

```
# configure the place, network type, trip times, and travel speed
address = "Hadan-dong, Saha-gu, Busan, Korea"
network_type = "walk"
trip_times = [1, 5, 10, 15, 20, 25, 30]  # 분 단위의 여행시간, 5분거리내
travel_speed = 4.5  # 걷는 속도 (km/hour
```

```
# 위치에 따른 지도 가져오기
G = ox.graph_from_address( address, network_type=network_type)
```

```
# 원하는 지점과 해당 지도에 대해서 그래프
gdf_nodes = ox.graph_to_gdfs(G, edges=False)
# x, y = gdf_nodes["geometry"].unary_union.centroid.
xy # 지도 중심점 좌표
# 동아대학교의 위경도 좌표: lat -> 위도 x , 경도->lon y
x = 128.96817249950897
y = 35.11755694483541
center_node = ox.distance.nearest_nodes(G, x, y)
G = ox.project_graph(G)
```

그래프에 가중치 넣기

```
# 계산을 위한 edges에 속성값을 넣습니다.
meters_per_minute = travel_speed * 1000 / 60  # km per hour to m per minute
for _, _, _, data in G.edges(data=True, keys=True):
    data["time"] = data["length"] / meters_per_minute
```

```
{'osmid': 59528585, 'bridge': 'yes', 'oneway': False, 'ref': '2', 'name': '낙동남로', 'highway': 'primary', 'length': 98.955, 'geometry': <shapely.geometry.linestring.LineString o
{'osmid': [59528596, 157995253], 'bridge': 'yes', 'oneway': False, 'highway': 'primary_link', 'length': 182.09500000000003, 'lanes': '1', 'geometry': <shapely.geometry.linestring.
{'osmid': 59528585, 'bridge': 'yes', 'oneway': False, 'ref': '2', 'name': '낙동남로', 'highway': 'primary', 'length': 98.955, 'geometry': <shapely.geometry.linestring.LineString o
{'osmid': [59528585, 564201227, 942931998], 'bridge': 'yes', 'oneway': False, 'ref': '2', 'name': '낙동남로', 'highway': 'primary', 'length': 419.789, 'lanes': '4', 'geometry': <s
{'osmid': [157995241, 157995221], 'bridge': 'yes', 'oneway': False, 'highway': 'primary_link', 'length': 268.84099999999995, 'lanes': '1', 'geometry': <shapely.geometry.linestring
{'osmid': [59528596, 157995253], 'bridge': 'yes', 'oneway': False, 'highway': 'primary_link', 'length': 182.09500000000003, 'lanes': '1', 'geometry': <shapely.geometry.linestring.
{'osmid': 157995276, 'oneway': False, 'name': '강변대로', 'highway': 'primary', 'length': 360.35699999999997, 'geometry': <shapely.geometry.linestring.LineString object at 0x7f7c1
{'osmid': 164881861, 'oneway': False, 'name': '낙동대로450번길', 'highway': 'residential', 'length': 78.86, 'geometry': <shapely.geometry.linestring.LineString object at 0x7f7c1fa
{'osmid': 620726366, 'oneway': False, 'ref': '2', 'name': '낙동대로', 'highway': 'primary', 'length': 57.065000000000005, 'lanes': '6', 'geometry': <shapely.geometry.linestring.Li
{'osmid': 164881854, 'oneway': False, 'name': '승학로2번길', 'highway': 'residential', 'length': 40.408, 'geometry': <shapely.geometry.linestring.LineString object at 0x7f7c1e7c49
{'osmid': 164881861, 'oneway': False, 'name': '낙동대로450번길', 'highway': 'residential', 'length': 78.86, 'geometry': <shapely.geometry.linestring.LineString object at 0x7f7c1e7
{'osmid': 164881861, 'oneway': False, 'name': '낙동대로450번길', 'highway': 'residential', 'length': 72.544, 'geometry': <shapely.geometry.linestring.LineString object at 0x7f7c1e
{'osmid': 620726366, 'oneway': False, 'ref': '2', 'name': '낙동대로', 'highway': 'primary', 'length': 7.058, 'lanes': '6', 'geometry': <shapely.geometry.linestring.LineString obje
{'osmid': 665487678, 'oneway': False, 'highway': 'service', 'length': 71.195, 'geometry': <shapely.geometry.linestring.LineString object at 0x7f7c1f4c0d10>, 'time': 0.949266666666
{'osmid': 620726366, 'oneway': False, 'ref': '2', 'name': '낙동대로', 'highway': 'primary', 'length': 57.065, 'lanes': '6', 'geometry': <shapely.geometry.linestring.LineString obj
{'osmid': 37398620, 'oneway': False, 'highway': 'tertiary', 'length': 93.363, 'geometry': <shapely.geometry.linestring.LineString object at 0x7f7c1fa1a0d0>, 'time': 1.24484}
{'osmid': 551375433, 'oneway': False, 'name': '낙동대로', 'highway': 'primary', 'length': 53.247, 'geometry': <shapely.geometry.linestring.LineString object at 0x7f7c1fa1ae10>, 't
{'osmid': 37398620, 'oneway': False, 'highway': 'tertiary', 'length': 35.271, 'geometry': <shapely.geometry.linestring.LineString object at 0x7f7c1fa1a7d0>, 'time': 0.470280000000
{'osmid': 37398620, 'oneway': False, 'highway': 'tertiary', 'length': 93.363, 'geometry': <shapely.geometry.linestring.LineString object at 0x7f7c1fa1af90>, 'time': 1.24484}
```

등시성(isochrone) 맵 구현

```
# 등시성의 polygon 구현
isochrone_polys = []
for trip_time in sorted(trip_times, reverse=True):
    subgraph = nx.ego_graph(G, center_node, radius=trip_time, distance="time")
    node_points = [Point((data["x"], data["y"])) for node, data in subgraph.node
s(data=True)]
    bounding_poly = gpd.GeoSeries(node_points).unary_union.convex_hull
    isochrone_polys.append(bounding_poly)


# 등시성에 따른 네트워크를 표기
fig, ax = ox.plot_graph(
    G, show=False, close=False, edge_color="#999999", edge_alpha=0.2, node_size
=0
)
for polygon, fc in zip(isochrone_polys, iso_colors):
    patch = PolygonPatch(polygon, fc=fc, ec="none", alpha=0.6, zorder=-1)
    ax.add_patch(patch)
plt.show()
```

등시성(isochrone) 맵 구현

Polyline 기반 등시성(isochrone) 맵 구현

```
  # isochrone map을 위한 컬러값을 가져옴
  iso_colors = ox.plot.get_colors(n=len(trip_times), cmap="plasma", start=0, return_h
  ex=True)


# 등시성에 따라 노드에 색칠
node_colors = {}
for trip_time, color in zip(sorted(trip_times, reverse=True), iso_colors):
    subgraph = nx.ego_graph(G, center_node, radius=trip_time, distance="time")
    for node in subgraph.nodes():
        node_colors[node] = color
nc = [node_colors[node] if node in node_colors else "none" for node in G.nodes()]
ns = [15 if node in node_colors else 0 for node in G.nodes()]
fig, ax = ox.plot_graph(
    G,
    node_color=nc,
    node_size=ns,
    node_alpha=0.8,
    edge_linewidth=0.2,
    edge_color="#999999",
)
```

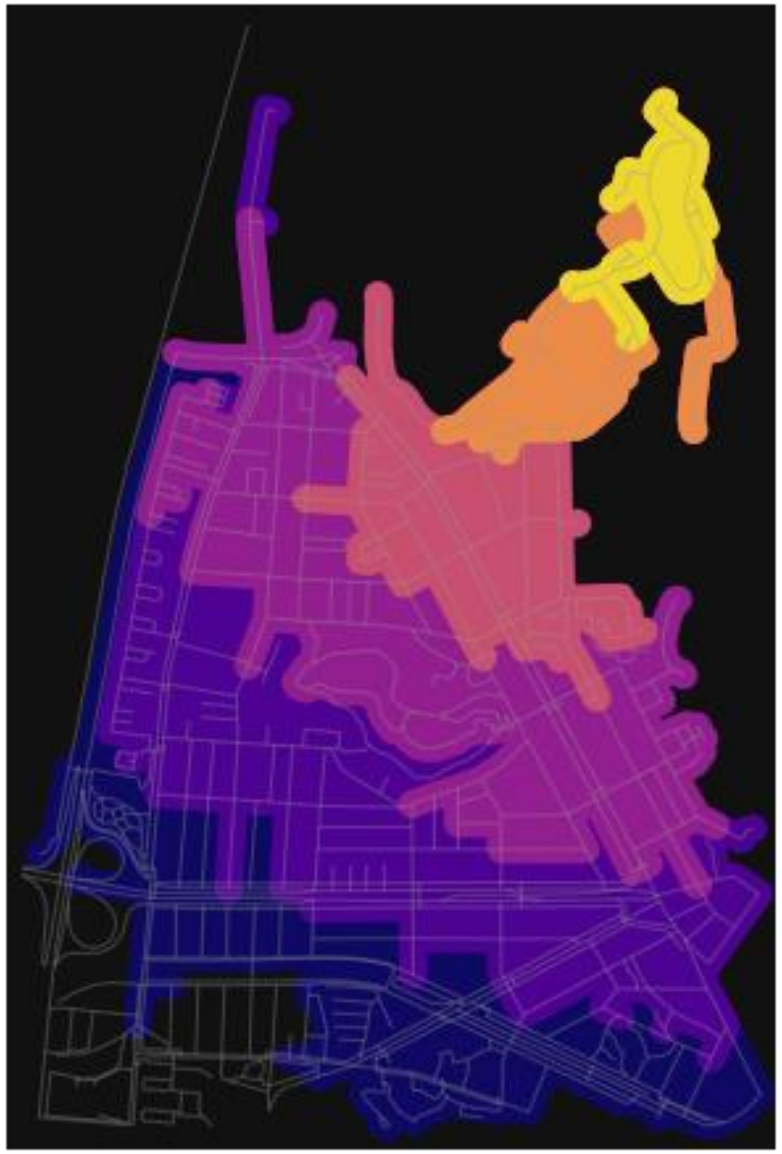
Polyline 기반 등시성(isochrone) 맵 구현

Isolated Polyline 기반 등시성(isochrone) 맵 구현

```
def make_iso_polys(G, edge_buff=25, node_buff=50, infill=False):
    isochrone_polys = []
    for trip_time in sorted(trip_times, reverse=True):
        subgraph = nx.ego_graph(G, center_node, radius=trip_time, distance="time")

        node_points = [Point((data["x"], data["y"])) for node, data in subgraph.nodes(data=True)]
        nodes_gdf = gpd.GeoDataFrame({"id": list(subgraph.nodes)}, geometry=node_points)
        nodes_gdf = nodes_gdf.set_index("id")

        edge_lines = []
        for n_fr, n_to in subgraph.edges():
            f = nodes_gdf.loc[n_fr].geometry
            t = nodes_gdf.loc[n_to].geometry
            edge_lookup = G.get_edge_data(n_fr, n_to)[0].get("geometry", LineString([f, t]))
            edge_lines.append(edge_lookup)

        n = nodes_gdf.buffer(node_buff).geometry
        e = gpd.GeoSeries(edge_lines).buffer(edge_buff).geometry
        all_gs = list(n) + list(e)
        new_iso = gpd.GeoSeries(all_gs).unary_union

        if infill:
            new_iso = Polygon(new_iso.exterior)
        isochrone_polys.append(new_iso)
    return isochrone_polys
```

Isolated Polyline 기반 등시성(isochrone) 맵 구현

```
isochrone_polys = make_iso_polys(G, edge_buff=25, node_buff=0, infill=True)
fig, ax = ox.plot_graph(
    G, show=False, close=False, edge_color="#999999", edge_alpha=0.2, node_size=0
)
for polygon, fc in zip(isochrone_polys, iso_colors):
    patch = PolygonPatch(polygon, fc=fc, ec="none", alpha=0.7, zorder=-1)
    ax.add_patch(patch)
plt.show()
```



HyperLocal

- Definition: information oriented around a well-defined community with its primary focus directed toward the concerns of the population in that community.

- [슬세권]이란 슬리퍼와 같은 편한 복장으로 각종 여가·편의시설을 이용할 수 있는 주거 권역을 가리킨다. 일상생활에 필요한 대부분을 인근에서 해결할 수 있어 주거 편의성이 높은 만큼 젊은 세대가 주 수요층인 오피스텔 시장에서 선호도가 높게 나타나고 있다.