Importing data into Neo4j
APOC library

Neo4j conf file

- Stop neo4j
- neo4j.conf file is in the conf directory of your $NEO4J_HOME.

- You can create a new database called transport by editing the option

  `dbms.default_database`
- Uncomment the line and replace neo4j with transport, and save the file.
- Start neo4j again.

- Lots of other configuration options

Many ways of importing data to Neo4j

1.LOAD CSV Cypher command: this command is a great starting point and handles small- to medium-sized data sets (up to 10 million records).

2.neo4j-admin bulk import tool: command line tool useful for straightforward loading of large data sets.

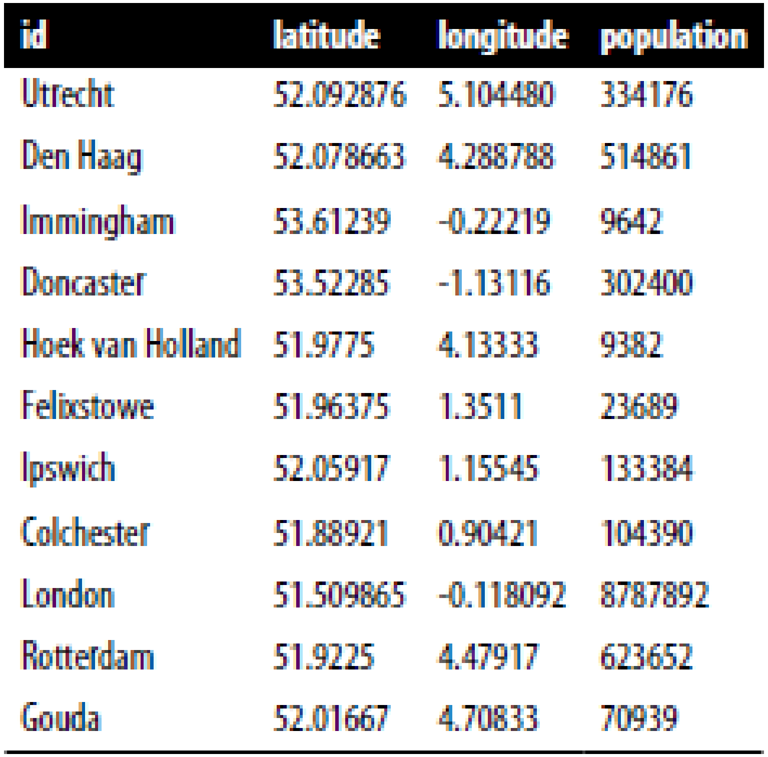
3.Kettle import tool

- Import from CSV using cypher
  - Files can be placed in the import folder within $NEO4J_HOME or hosted on the web
    - On OSX/UNIX, you would need to use the following for a local file “file:///path/to/data.csv” whereas the same url on Windows would be “file:c:/path/to/data.csv”.
  - Load csv with headers from (path) as alias

The transport Graph

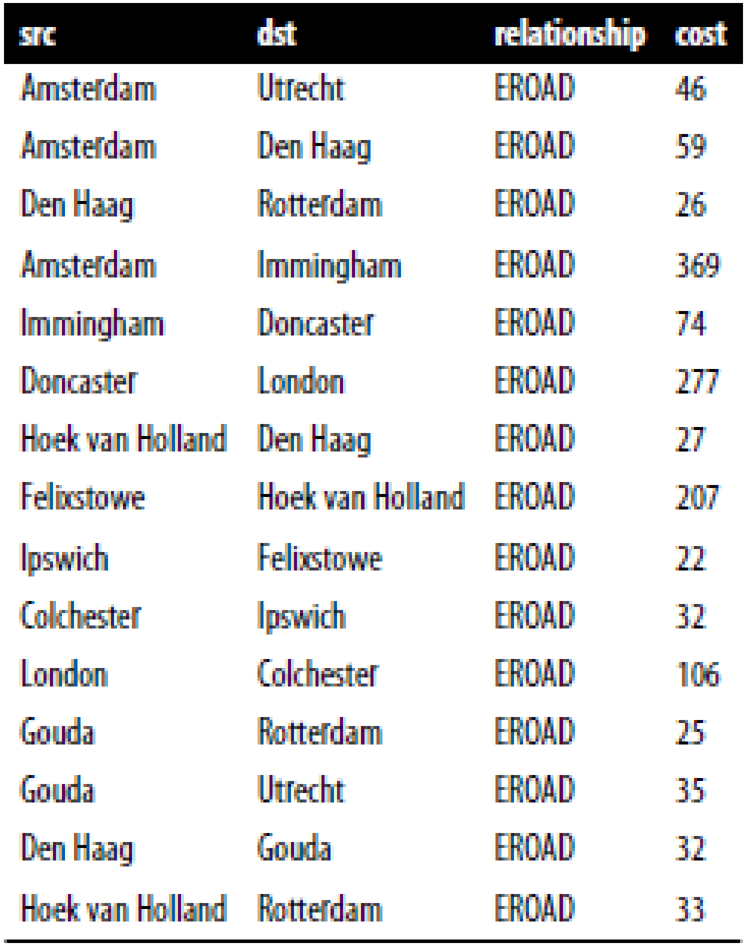
- Graph containing a subset of the European road network

Nodes



id	latitude	longitude	population
Utrecht	52.092876	5.104480	334176
Den Haag	52.078663	4.288788	514861
Immingham	53.61239	-0.22219	9642
Doncaster	53.52285	-1.13116	302400
Hoek van Holland	51.9775	4.13333	9382
Felixstowe	51.96375	1.3511	23689
Ipswich	52.05917	1.15545	133384
Colchester	51.88921	0.90421	104390
London	51.509865	-0.118092	8787892
Rotterdam	51.9225	4.47917	623652
Gouda	52.01667	4.70833	70939

Relationships



src	dst	relationship	cost
Amsterdam	Utrecht	EROAD	46
Amsterdam	Den Haag	EROAD	59
Den Haag	Rotterdam	EROAD	26
Amsterdam	Immingham	EROAD	369
Immingham	Doncaster	EROAD	74
Doncaster	London	EROAD	277
Hoek van Holland	Den Haag	EROAD	27
Felixstowe	Hoek van Holland	EROAD	207
Ipswich	Felixstowe	EROAD	22
Colchester	Ipswich	EROAD	32
London	Colchester	EROAD	106
Gouda	Rotterdam	EROAD	25
Gouda	Utrecht	EROAD	35
Den Haag	Gouda	EROAD	32
Hoek van Holland	Rotterdam	EROAD	33

Merge and With clauses

https://neo4j.com/docs/cypher-manual/current/clauses/merge/

The MERGE clause ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.

- Like a combination of MATCH and CREATE

https://neo4j.com/docs/cypher-manual/current/clauses/with/#query-with

The WITH clause allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.

Import data using cypher

- We’ll start by loading the nodes:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/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.latitude),
place.population = toInteger(row.population)
```

- And now the relationships:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/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)
```

Some queries

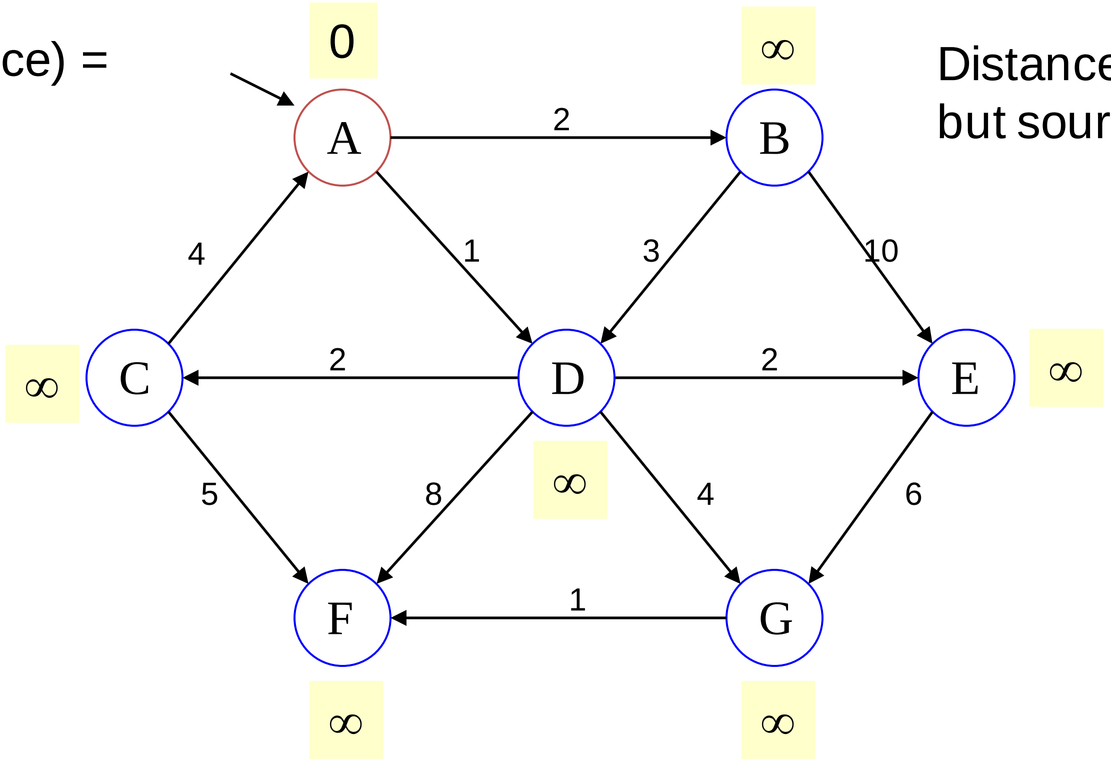
```
MATCH (p:Place)
WITH max(p.population)  as highestPop
MATCH (p2:Place)
where p2.population=highestPop
return p2;

MATCH (source:Place {id: "Amsterdam"}),
(destination:Place {id: "London"}),
p=shortestPath((source)-[*]-(destination))
RETURN p;
```

Dijkstra pseudocode

```
Dijkstra(v1, v2):
  for each vertex v:                    // Initialization
     v's distance := infinity.
     v's previous := none.
  v1's distance := 0.
  List := {all vertices}.

  while List is not empty:
     v := remove List vertex with minimum distance.
     mark v as known.
     for each unknown neighbor n of v:
        dist := v's distance + edge (v, n)'s weight.

        if dist is smaller than n's distance:
           n's distance := dist.
           n's previous := v.

  reconstruct path from v2 back to v1,
  following previous pointers.
```

Example: Initialization

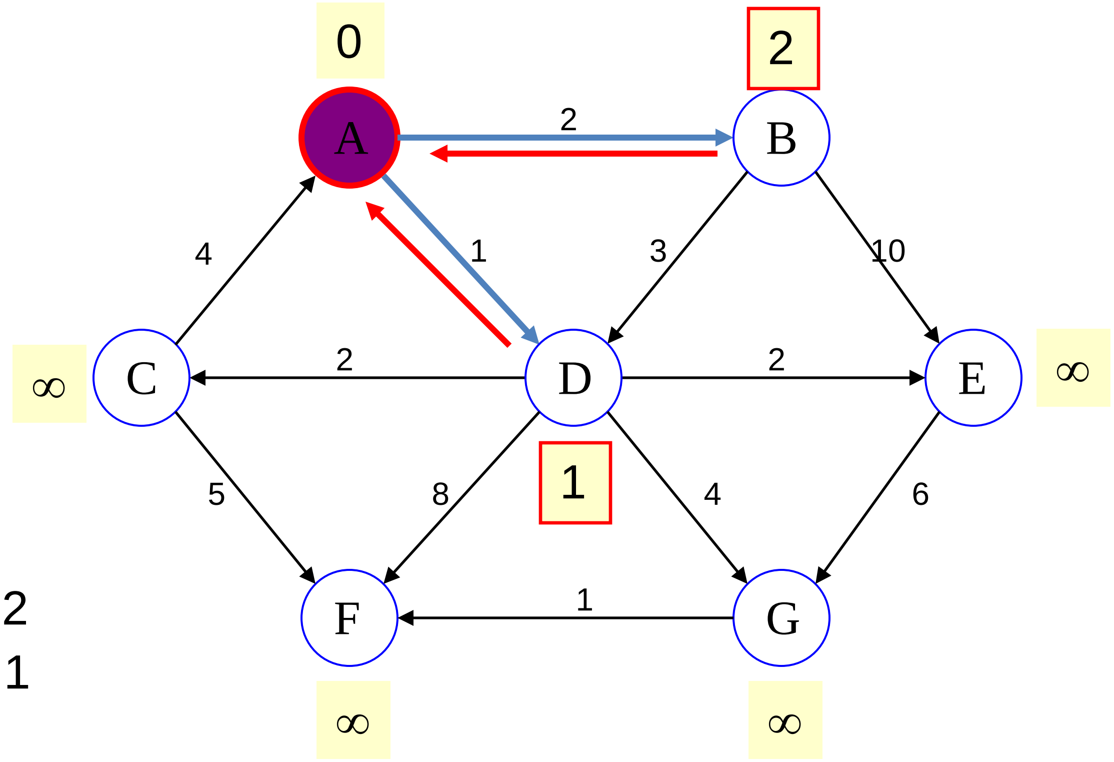
Distance(source) = 0



Distance (all vertices but source) = ∞

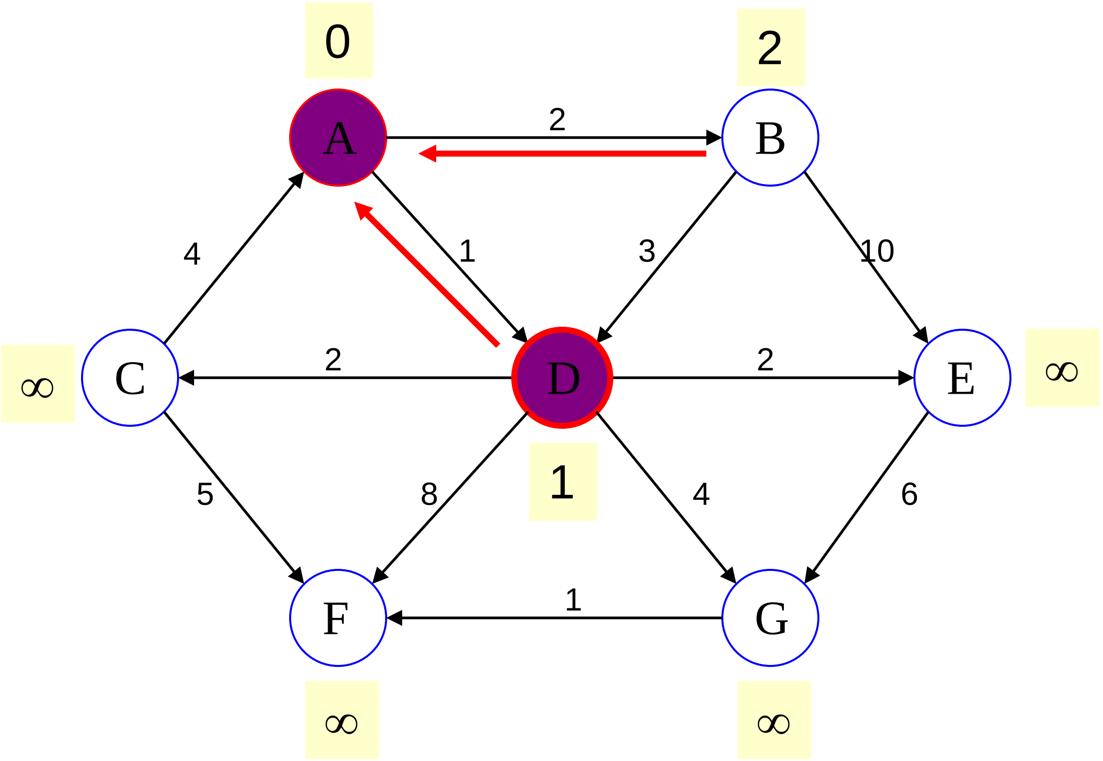
Pick vertex in List with minimum distance.

Example: Update neighbors' distance



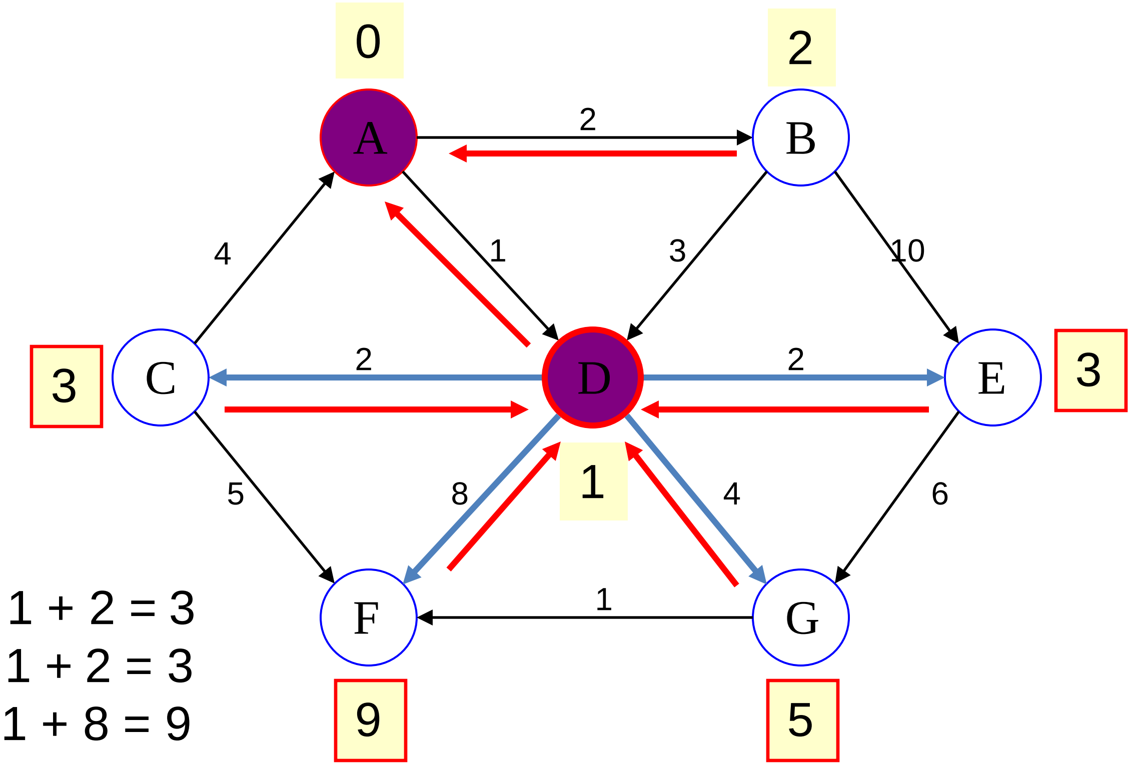
Distance(B) = 2
Distance(D) = 1

Example: Remove vertex with minimum distance



Pick vertex in List with minimum distance, i.e., D

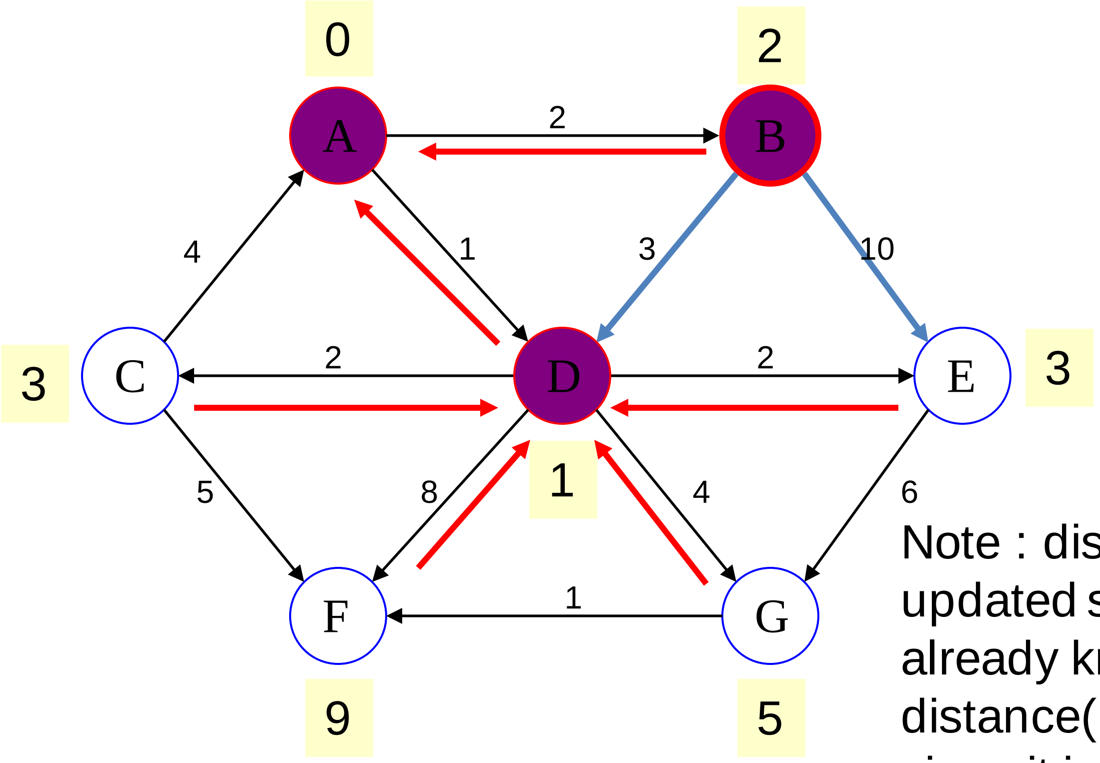
Example: Update neighbors



Distance(C) = 1 + 2 = 3
Distance(E) = 1 + 2 = 3
Distance(F) = 1 + 8 = 9
Distance(G) = 1 + 4 = 5

Example: Continued...

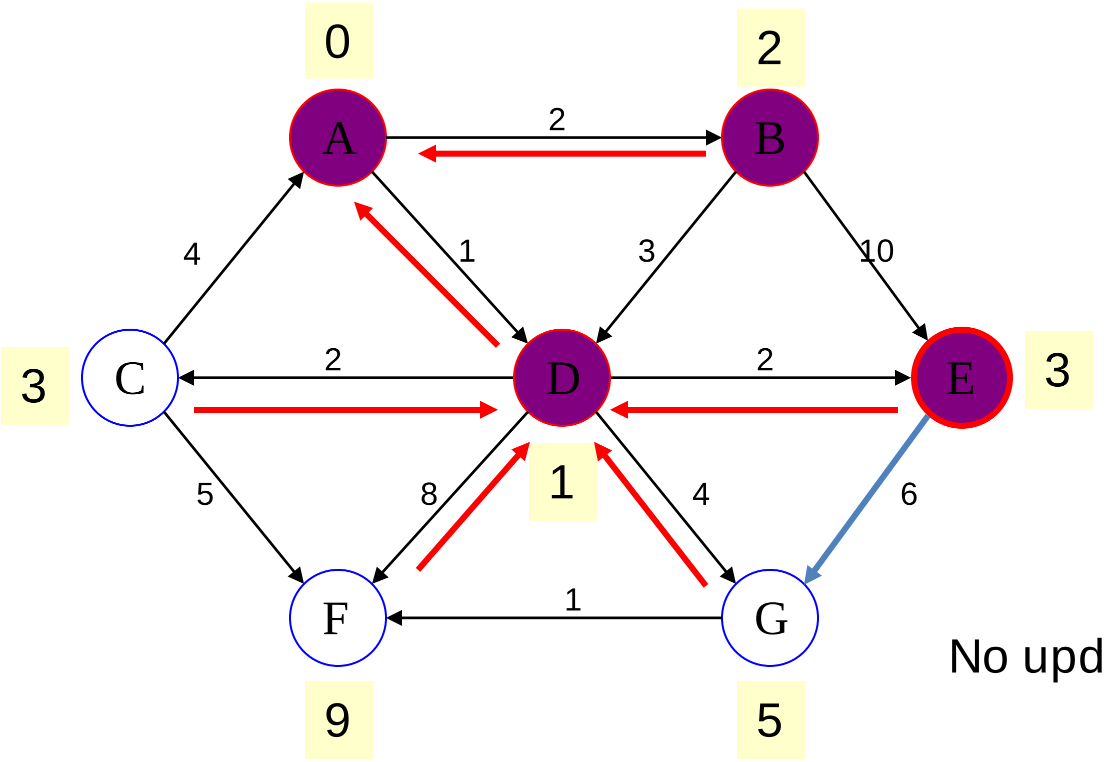
Pick vertex in List with minimum distance (B) and update neighbors



Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

Example: Continued...

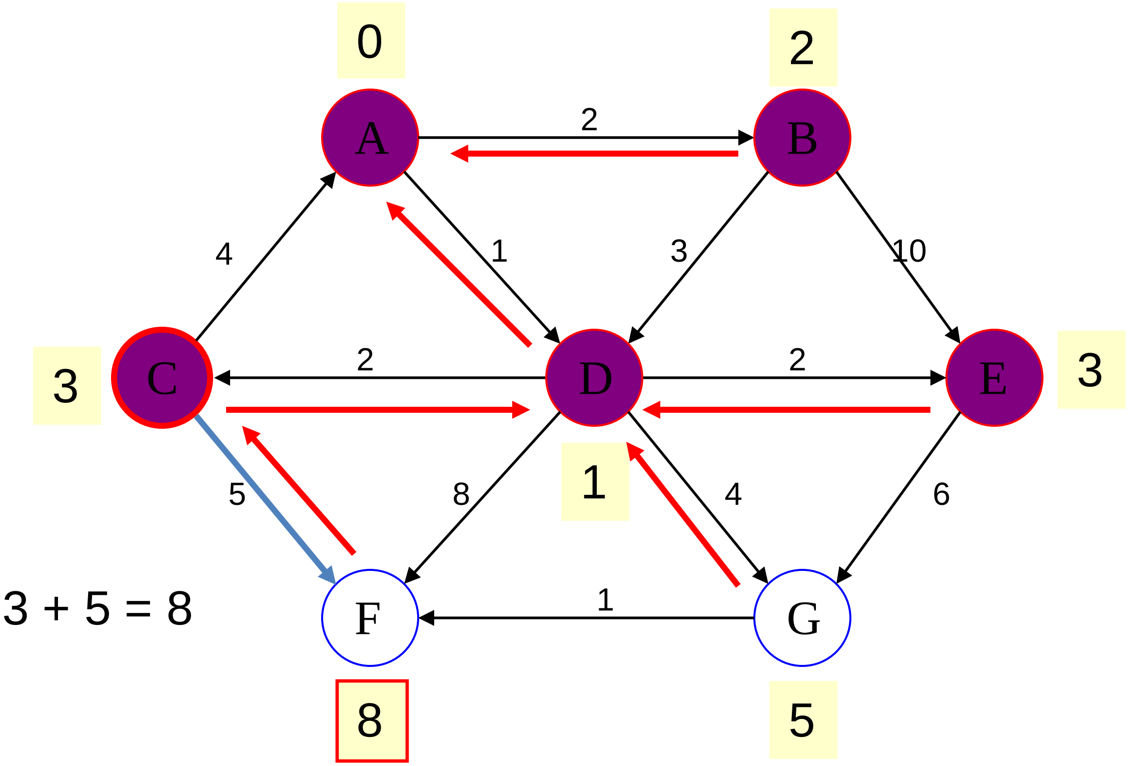
Pick vertex List with minimum distance (E) and update neighbors



No updating

Example: Continued...

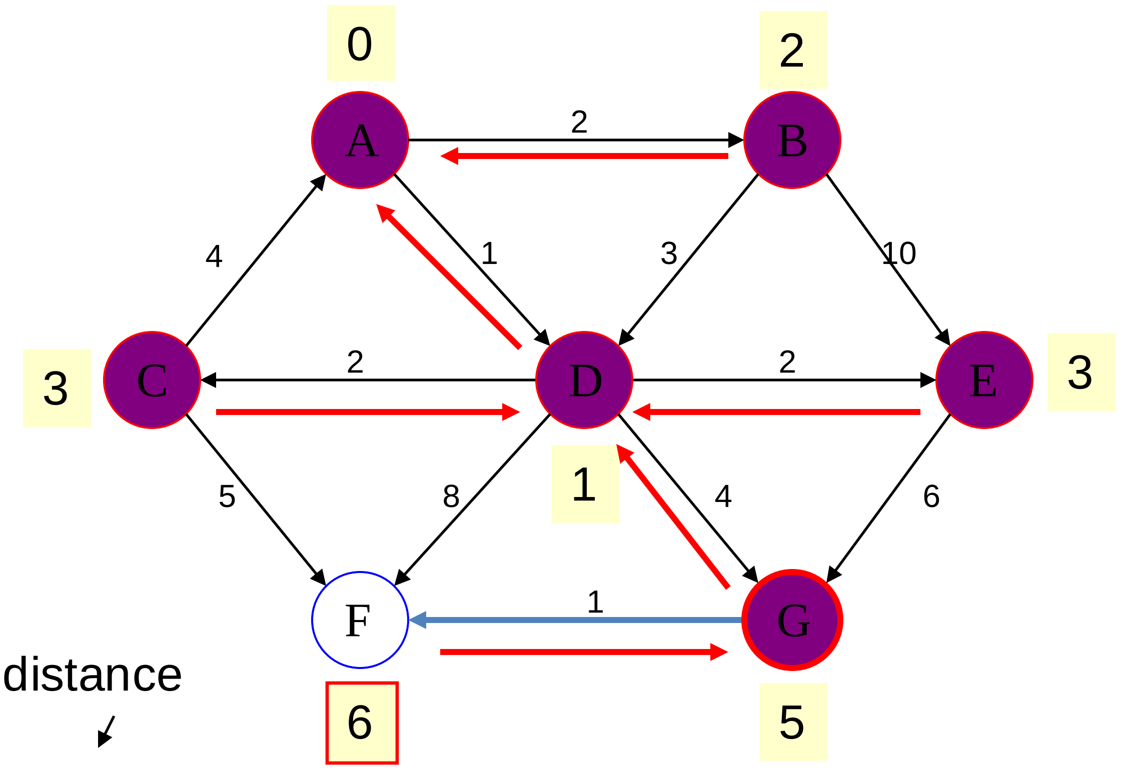
Pick vertex List with minimum distance (C) and update neighbors



Distance(F) = 3 + 5 = 8

Example: Continued...

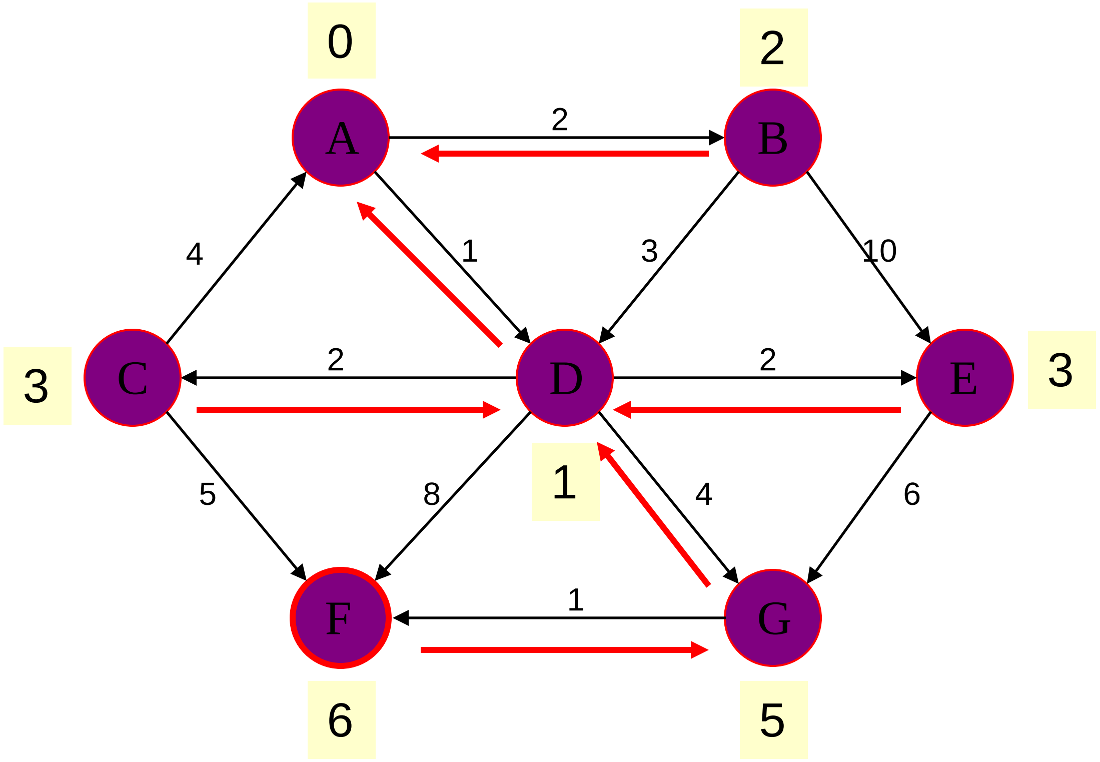
Pick vertex List with minimum distance (G) and update neighbors



Previous distance

Distance(F) = min (8, 5+1) = 6

Example (end)



Pick vertex not in S with lowest cost (F) and update neighbors

APOC library

- https://neo4j.com/labs/apoc/4.1/

  APOC Core can be installed by moving the APOC jar file from the $NEO4J_HOME/labs directory to the $NEO4J_HOME/plugins directory and restarting Neo4j.

- https://neo4j.com/labs/apoc/4.1/import/

  -
    ```
    WITH 'https://raw.githubusercontent.com/neo4j-contrib/neo4j-apoc-procedures/4.1/src/test/resources/person.json' AS url CALL apoc.load.json(url) YIELD value as person MERGE (p:Person {name:person.name}) ON CREATE SET p.age = person.age, p.children = size(person.children)
    ```

Dijkstra’s shortest path

-
  ```
  MATCH (source:Place {id: "Amsterdam"}),
  (destination:Place {id: "London"})
  CALL apoc.algo.dijkstra(source, destination, 'EROAD',
  'distance')
  YIELD path, weight as cost
  RETURN path, cost
  ```

-
  ```
  MATCH (source:Place {id: "Amsterdam"}),
  (destination:Place {id: "London"})
  CALL apoc.algo.dijkstraWithDefaultWeight(source,
  destination, 'EROAD', 'distance',10)
  YIELD path, weight as cost
  RETURN path, cost
  ```

Install and configure the Graph Data Science Library

On a standalone Neo4j Server, the library will need to be installed and configured manually.

1.Download neo4j-graph-data-science-[version]-standalone.jar from the Neo4j Download Center and copy it into the $NEO4J_HOME/plugins directory.

2.Add the following to your $NEO4J_HOME/conf/neo4j.conf file:

```
dbms.security.procedures.unrestricted=gds.*
```

This configuration entry is necessary because the GDS library accesses low-level components of Neo4j to maximize performance.

3.Check if the procedure whitelist is enabled in the $NEO4J_HOME/conf/neo4j.conf file and add the GDS library if necessary:

```
dbms.security.procedures.whitelist=gds.*
```

4.Restart Neo4j

For today…

After configuring gds, load the result of this query in the transport database to ICON:

```
MATCH (source:Place {id: "Amsterdam"}),
(destination:Place {id: "London"})
CALL gds.algo.shortestPath.stream(source, destination, null)
YIELD nodeId, cost
RETURN gds.algo.getNodeById(nodeId).id AS place, cost
```