Betweenness centrality

introduction

Ing. Robert Skopal, Ing. Jiří Hanzelka

IT4Innovations
national01$#&0
supercomputing
center@#01%101

Motivation

- Betweenness = indicator of node/edge “importance” in a network

- Social networks
  - identify “important” users that connects other users (celebrities, politicians etc.)

- Traffic
  - Identify traffic bottlenecks
  - Monitor how these bottlenecks change during the day
  - Compute betweenness for different types of factors – distance, speed etc.
  - Use information about bottlenecks as input for routing alogirthms

Definition (Freeman, 1977)

G(V,E) … graph
V … set of vertices (nodes)
E … set of edges

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

"On how many shortest paths the node *v* is on?"

σ_{st} - number of shortest paths between *s* and *t*

$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*

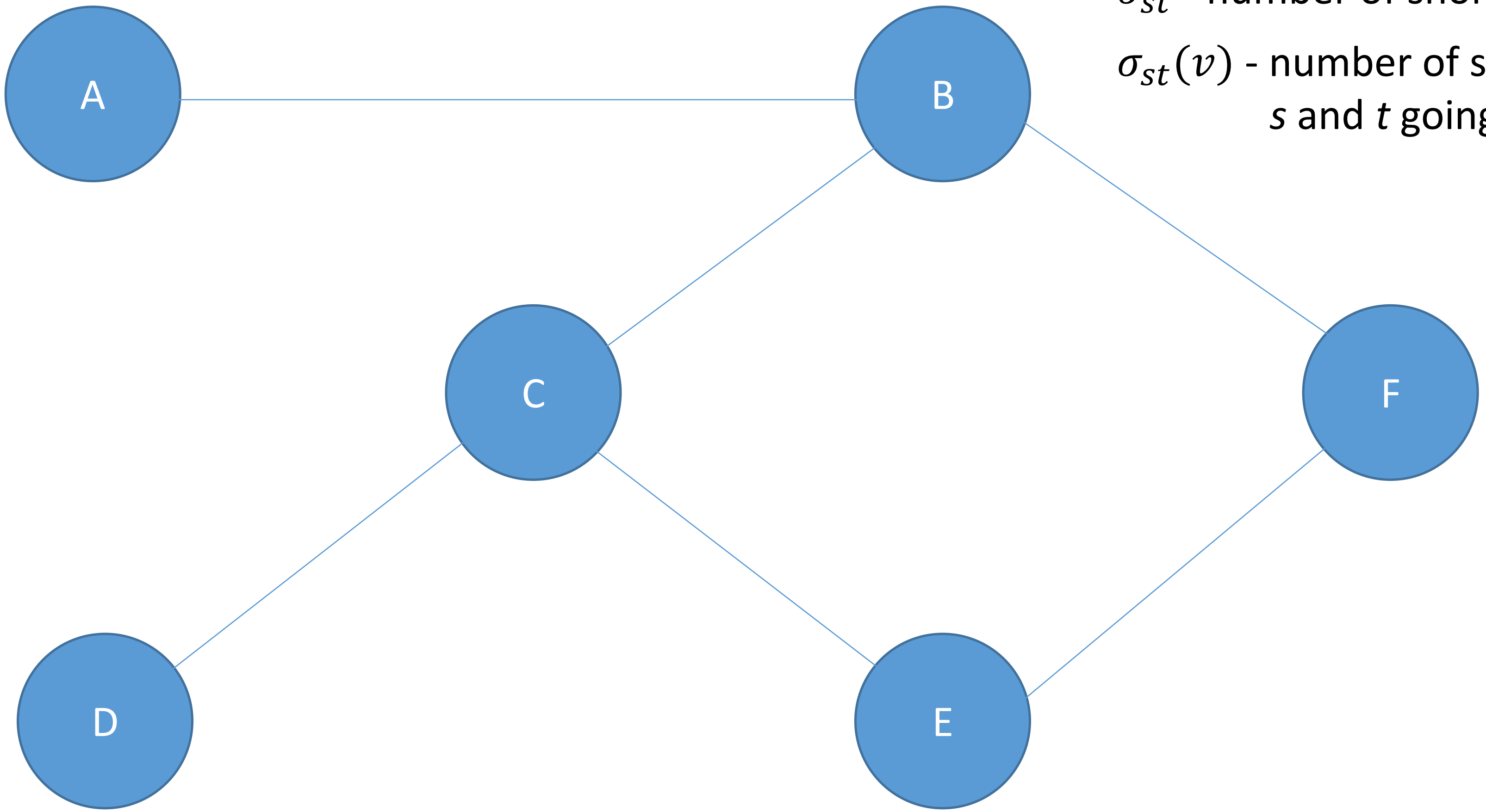
- Indicator of node/edge "importance" in a network.
- Node with high betweenness centrality score can reach others on relatively short paths.

Computation (naive)

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*

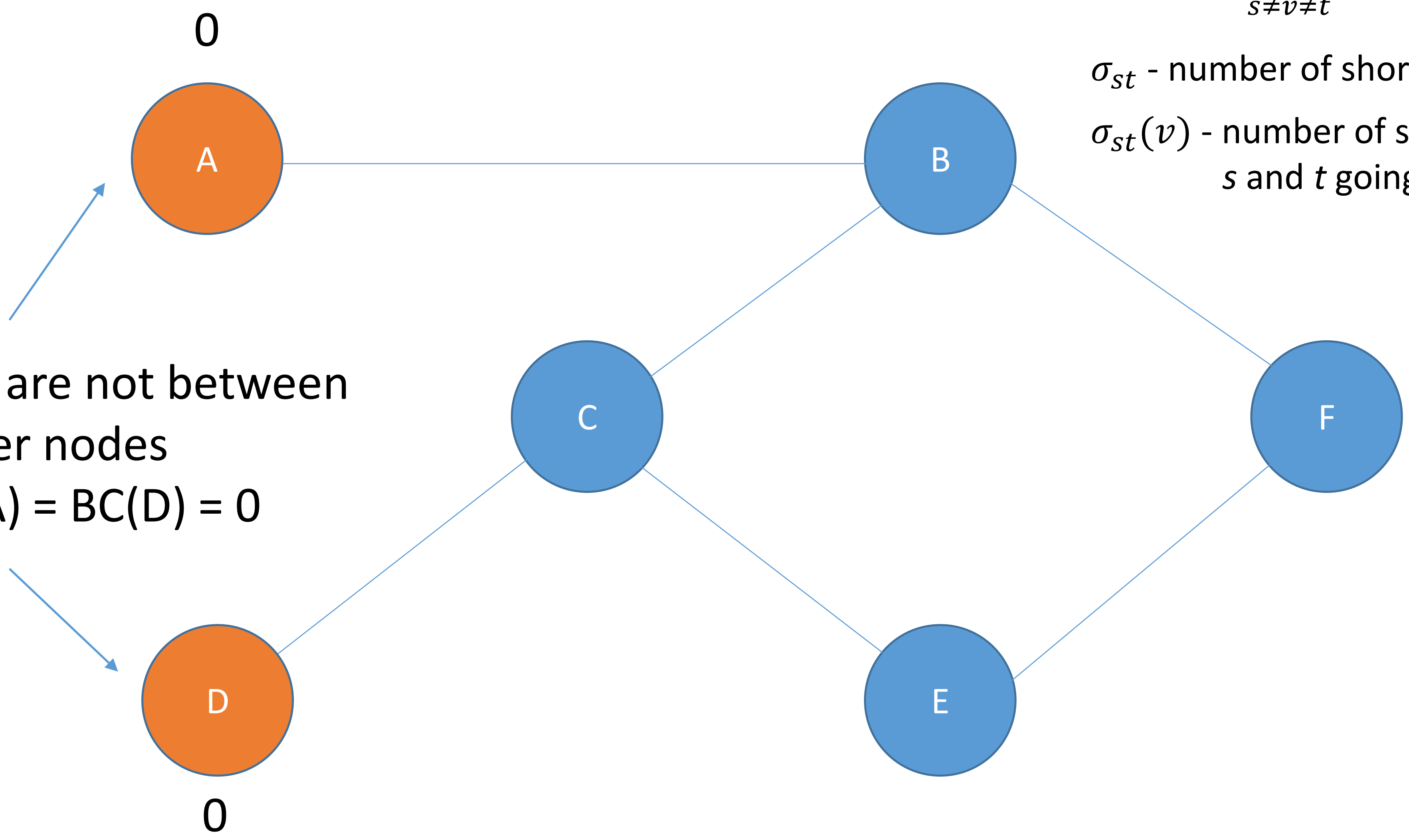


Computation (naive)

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*



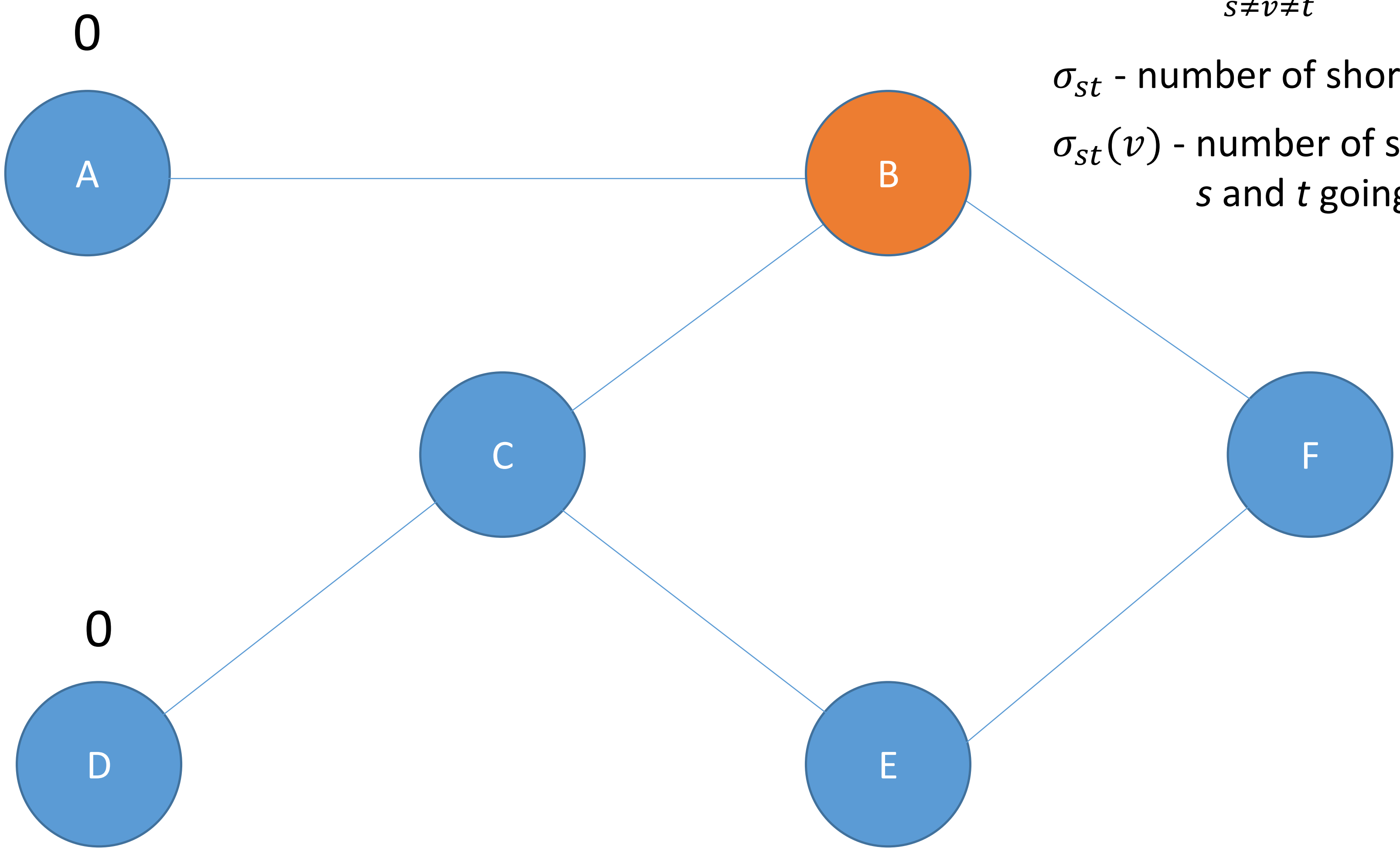
A and D are not between any other nodes
=> BC(A) = BC(D) = 0

Computation (naive)

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*



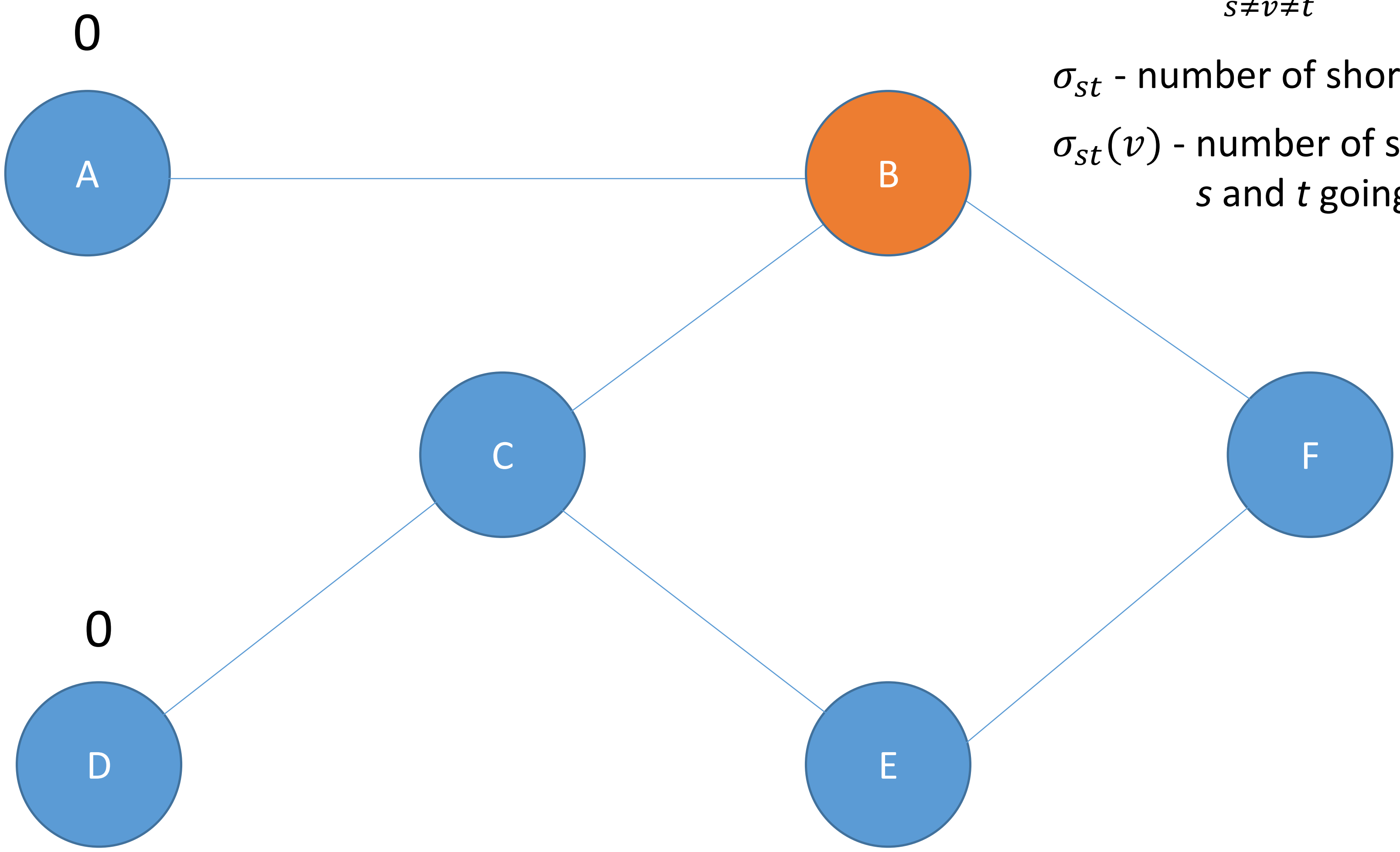
BC(B) = ?

Computation (naive)

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

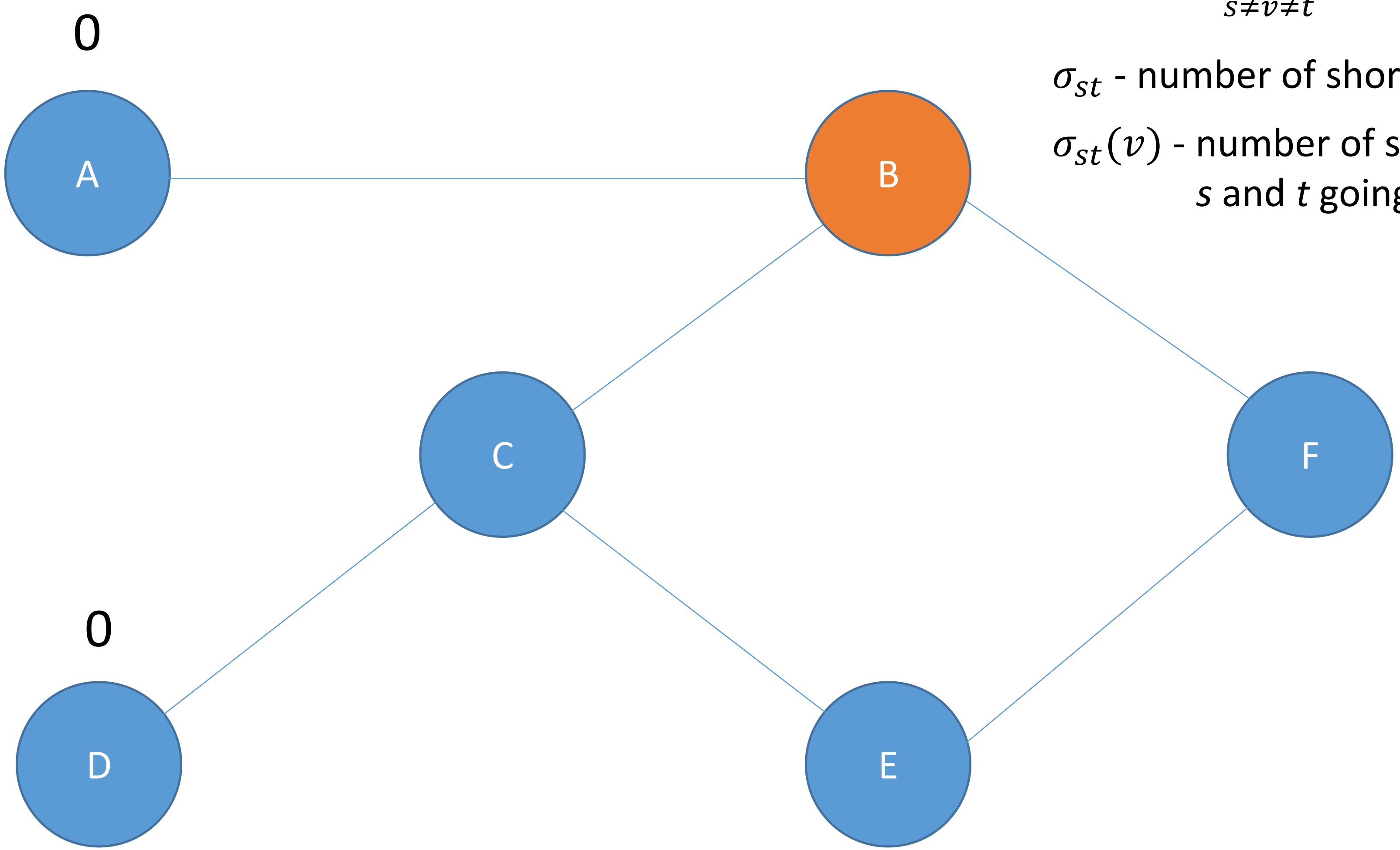
$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*



AC: A-B-C => 1/1

BC(B) = ?

Computation (naive)



$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*

AC: A-B-C => 1/1
AD: A-B-C-D => 1/1

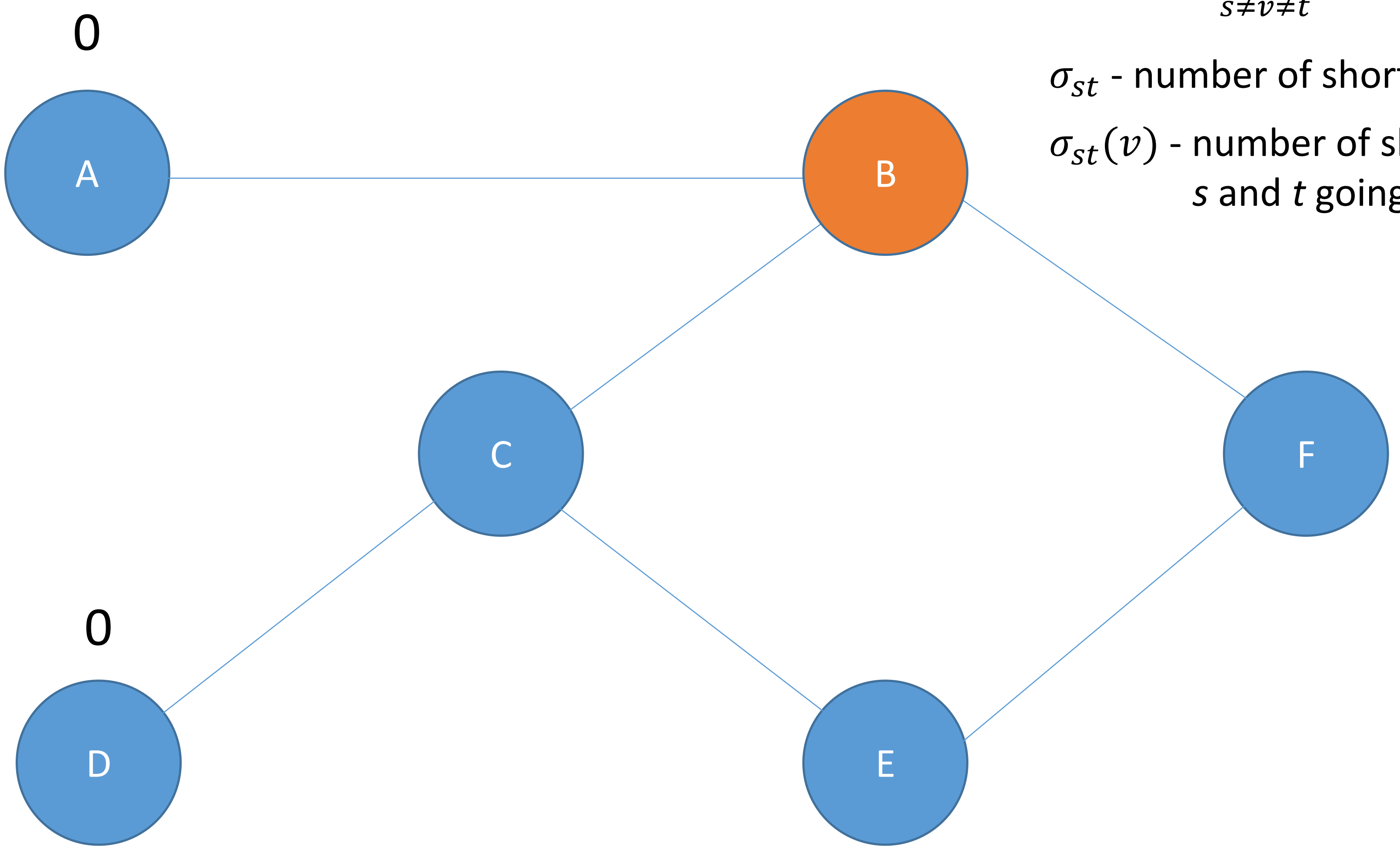
BC(B) = ?

Computation (naive)

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*



AC: A-B-C => 1/1
AD: A-B-C-D => 1/1
AE: A-B-C-E
A-B-F-E => 2/2

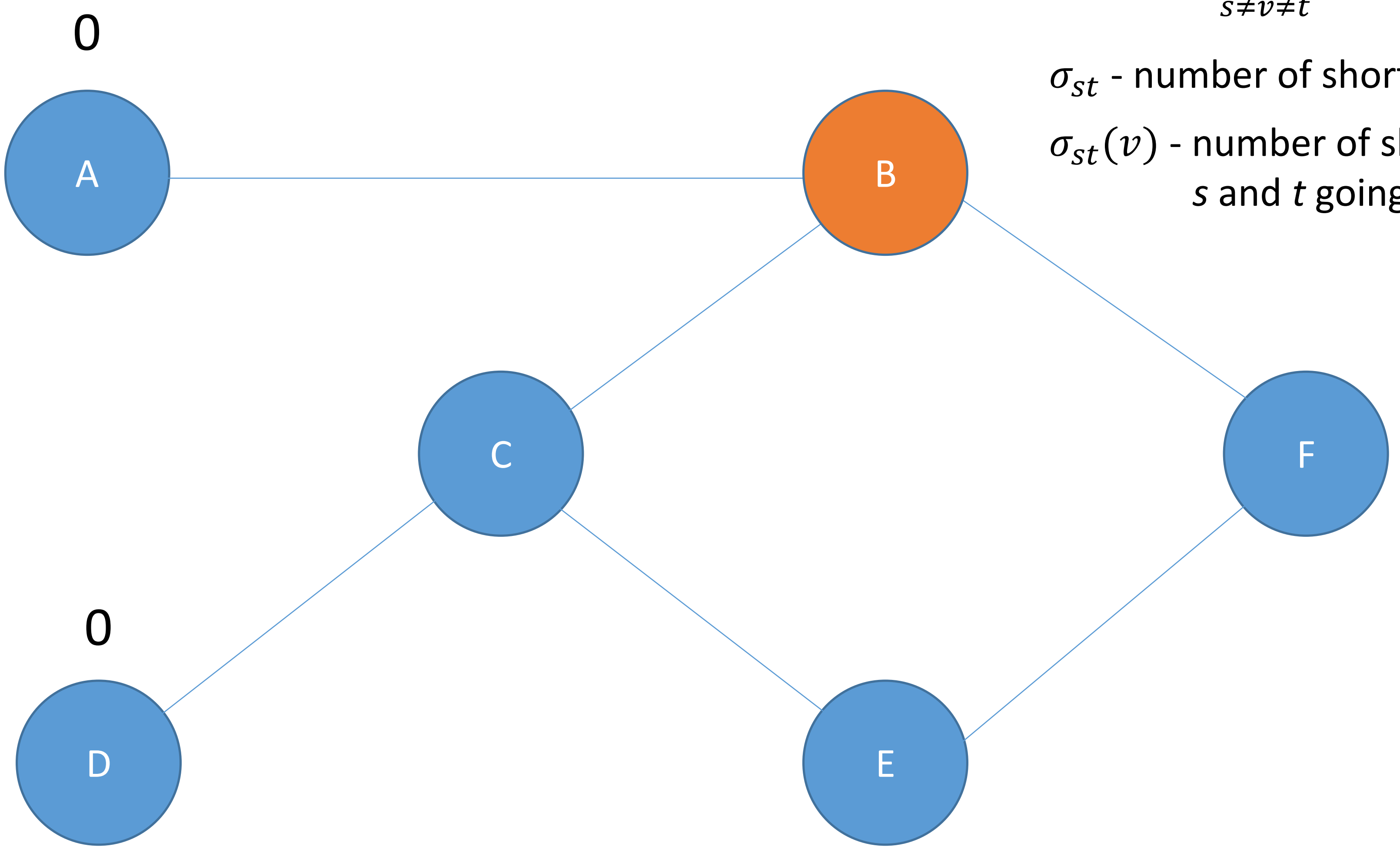
BC(B) = ?

Computation (naive)

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*



AC: A-B-C => 1/1
AD: A-B-C-D => 1/1
AE: A-B-C-E
A-B-F-E => 2/2
AF: A-B-F => 1/1

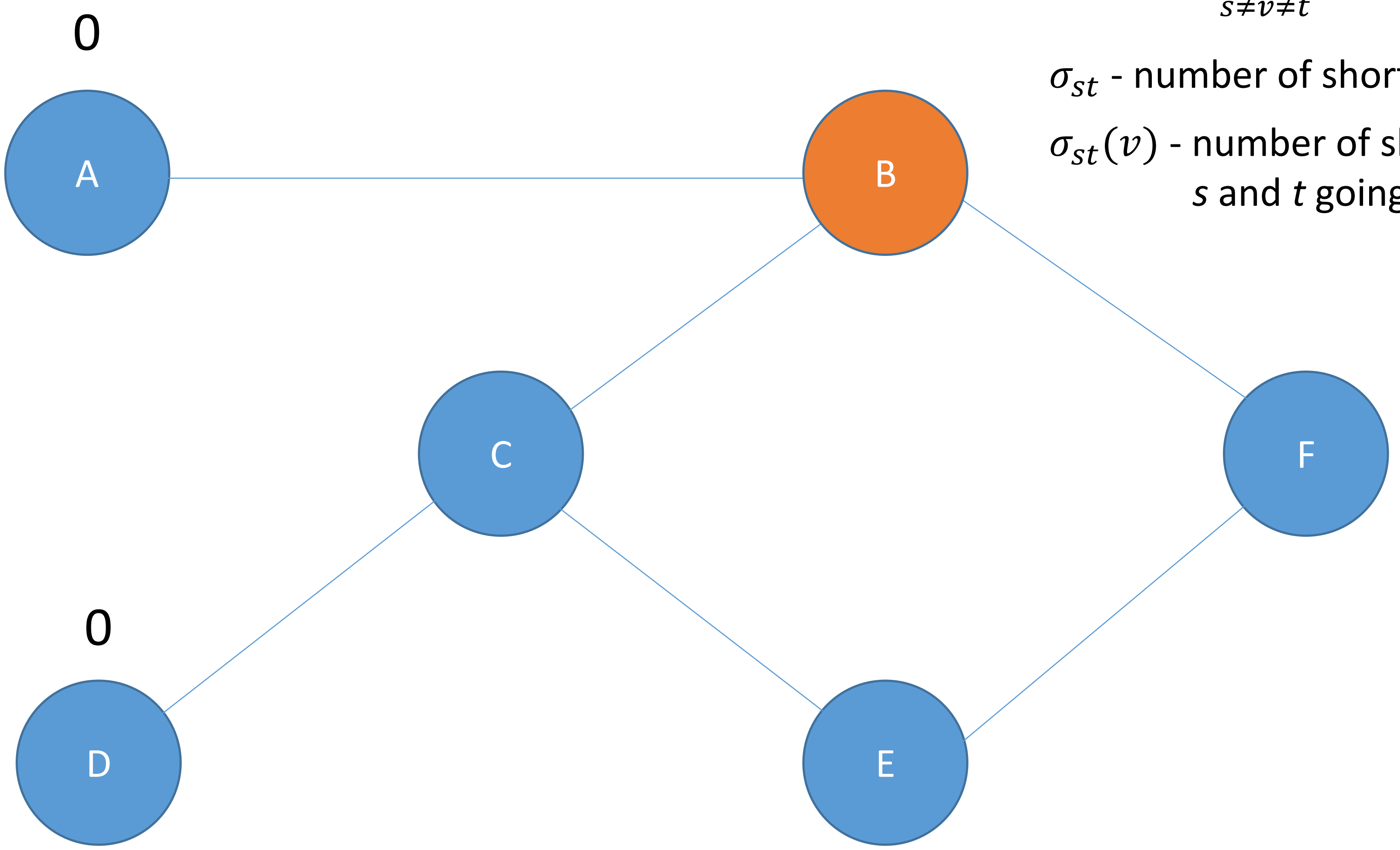
BC(B) = ?

Computation (naive)

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*



AC: A-B-C => 1/1
AD: A-B-C-D => 1/1
AE: A-B-C-E
A-B-F-E => 2/2
AF: A-B-F => 1/1
CD: X

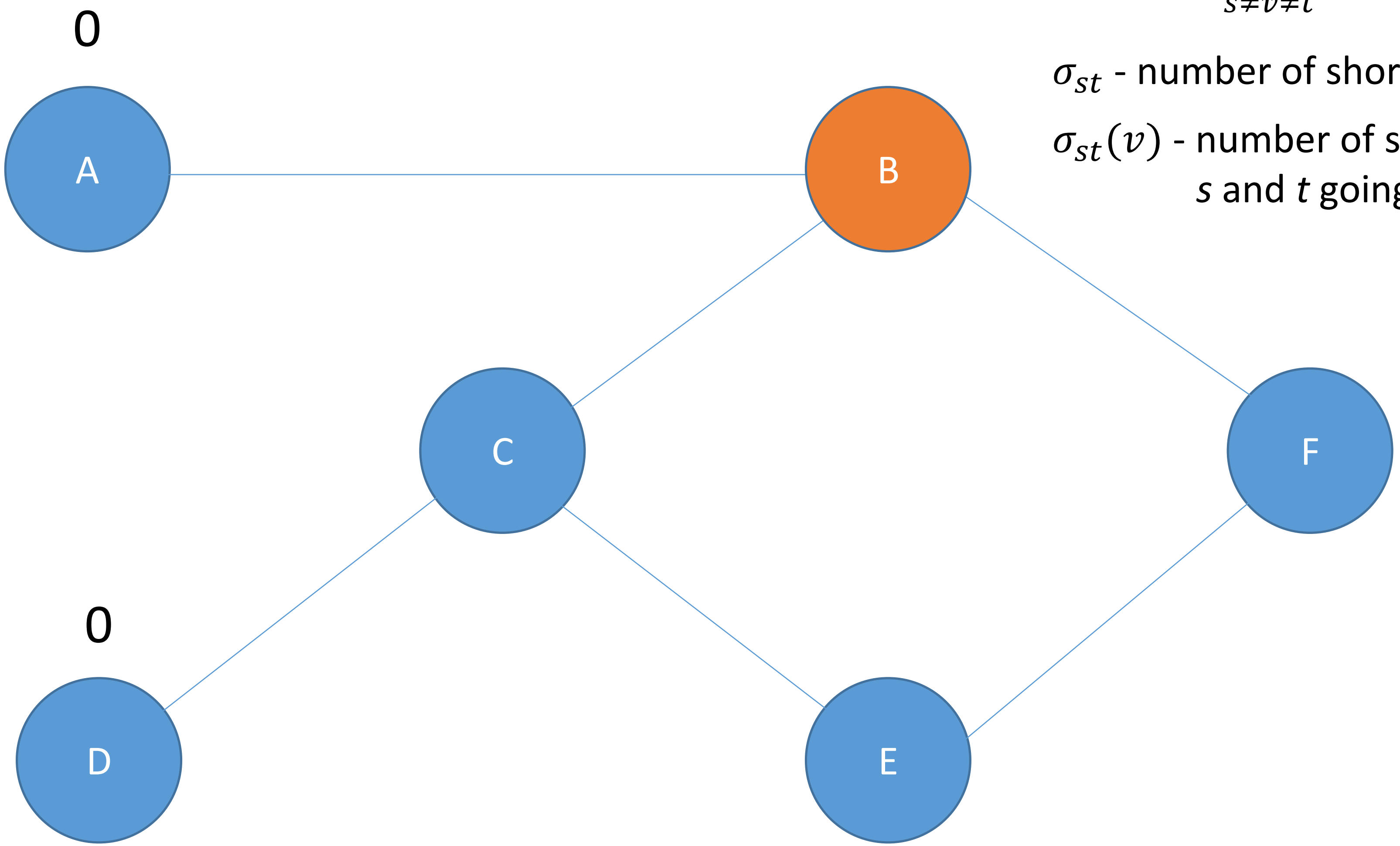
BC(B) = ?

Computation (naive)

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

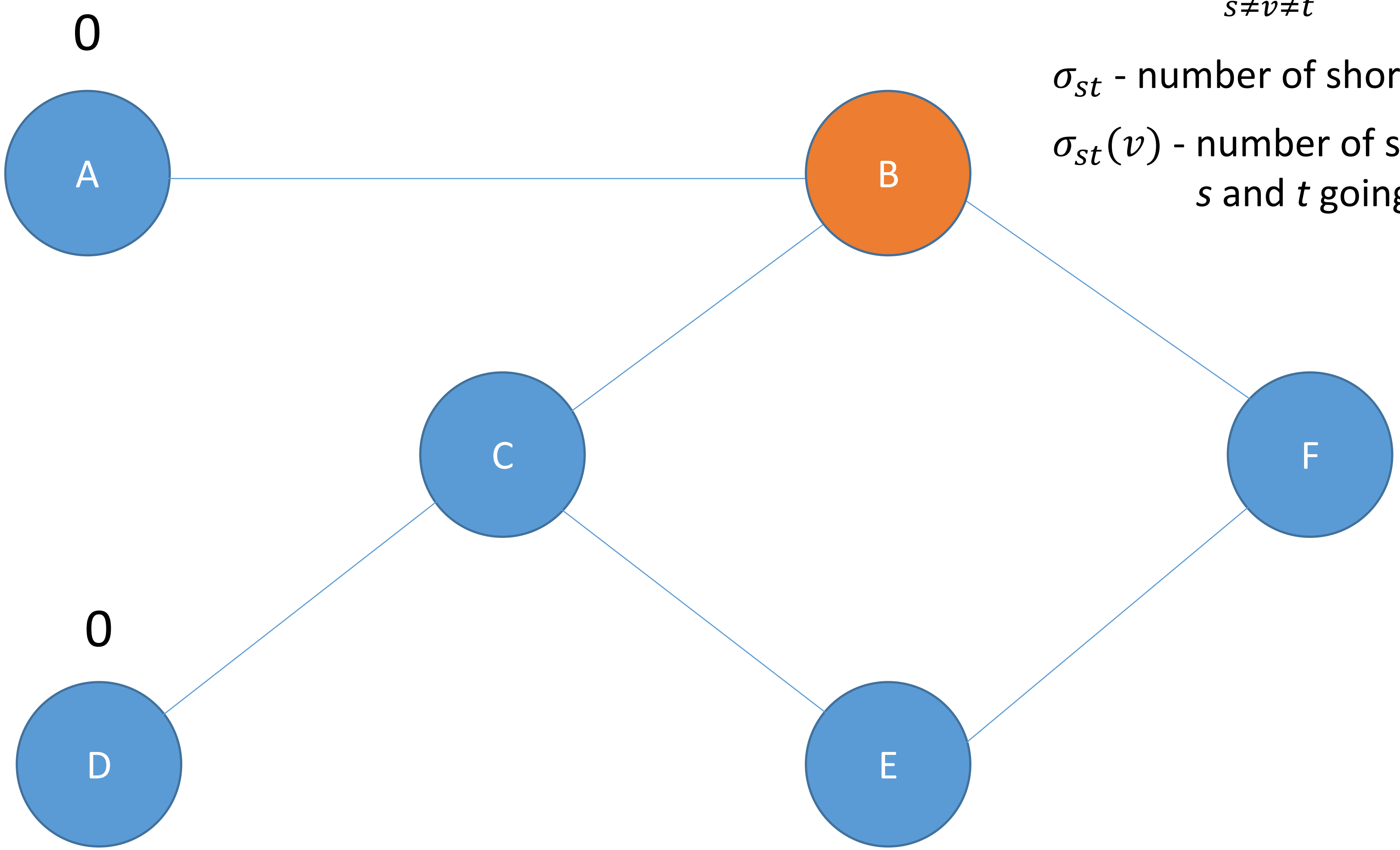
$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*



AC: A-B-C => 1/1
AD: A-B-C-D => 1/1
AE: A-B-C-E
A-B-F-E => 2/2
AF: A-B-F => 1/1
CD: X
CE: X

BC(B) = ?

Computation (naive)



$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*

AC: A-B-C => 1/1
AD: A-B-C-D => 1/1
AE: A-B-C-E
A-B-F-E => 2/2
AF: A-B-F => 1/1
CD: X
CE: X
CF: C-B-F => 1/2

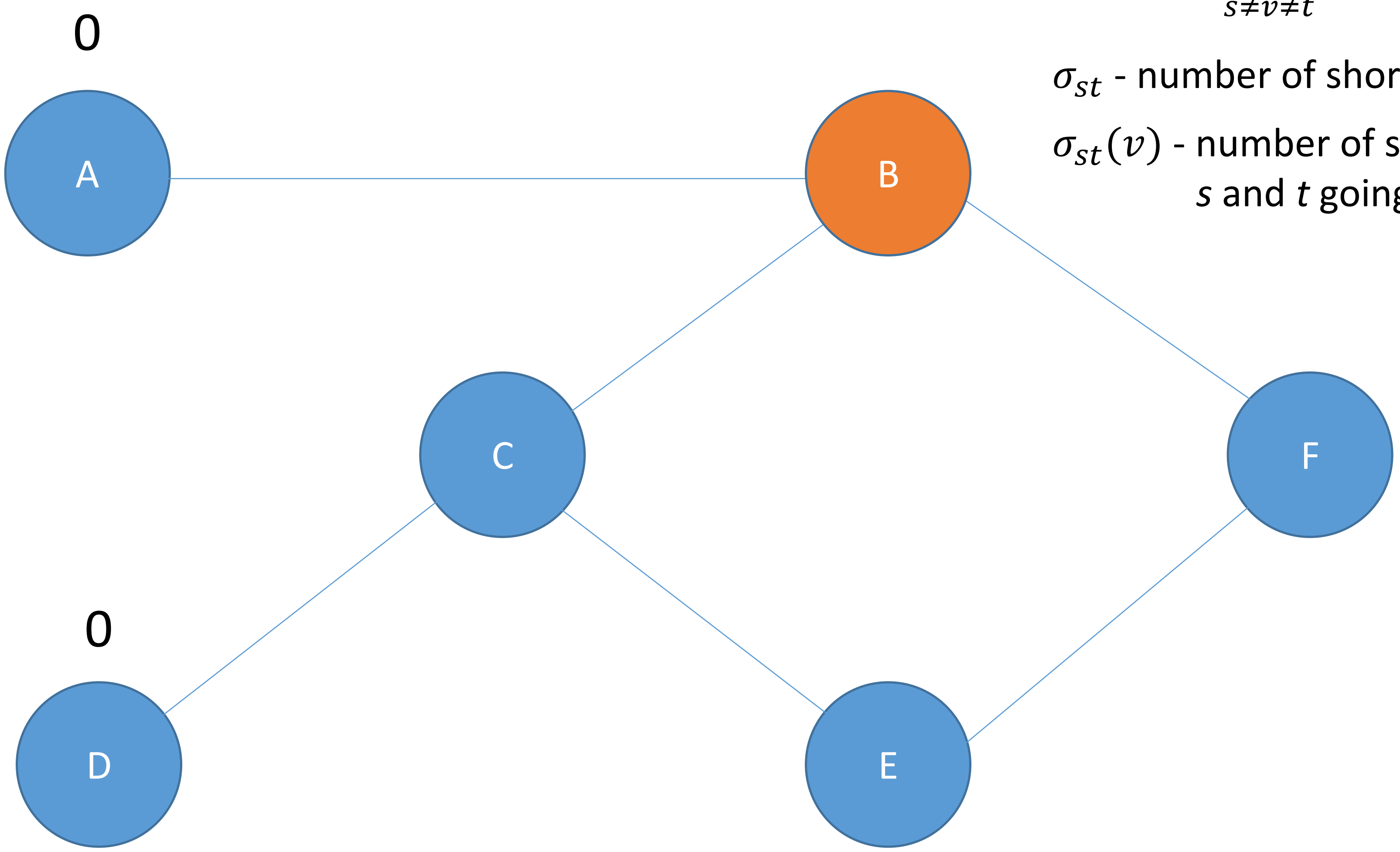
BC(B) = ?

Computation (naive)

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*



AC: A-B-C => 1/1
AD: A-B-C-D => 1/1
AE: A-B-C-E
A-B-F-E => 2/2
AF: A-B-F => 1/1
CD: X
CE: X
CF: C-B-F => 1/2
DE: X

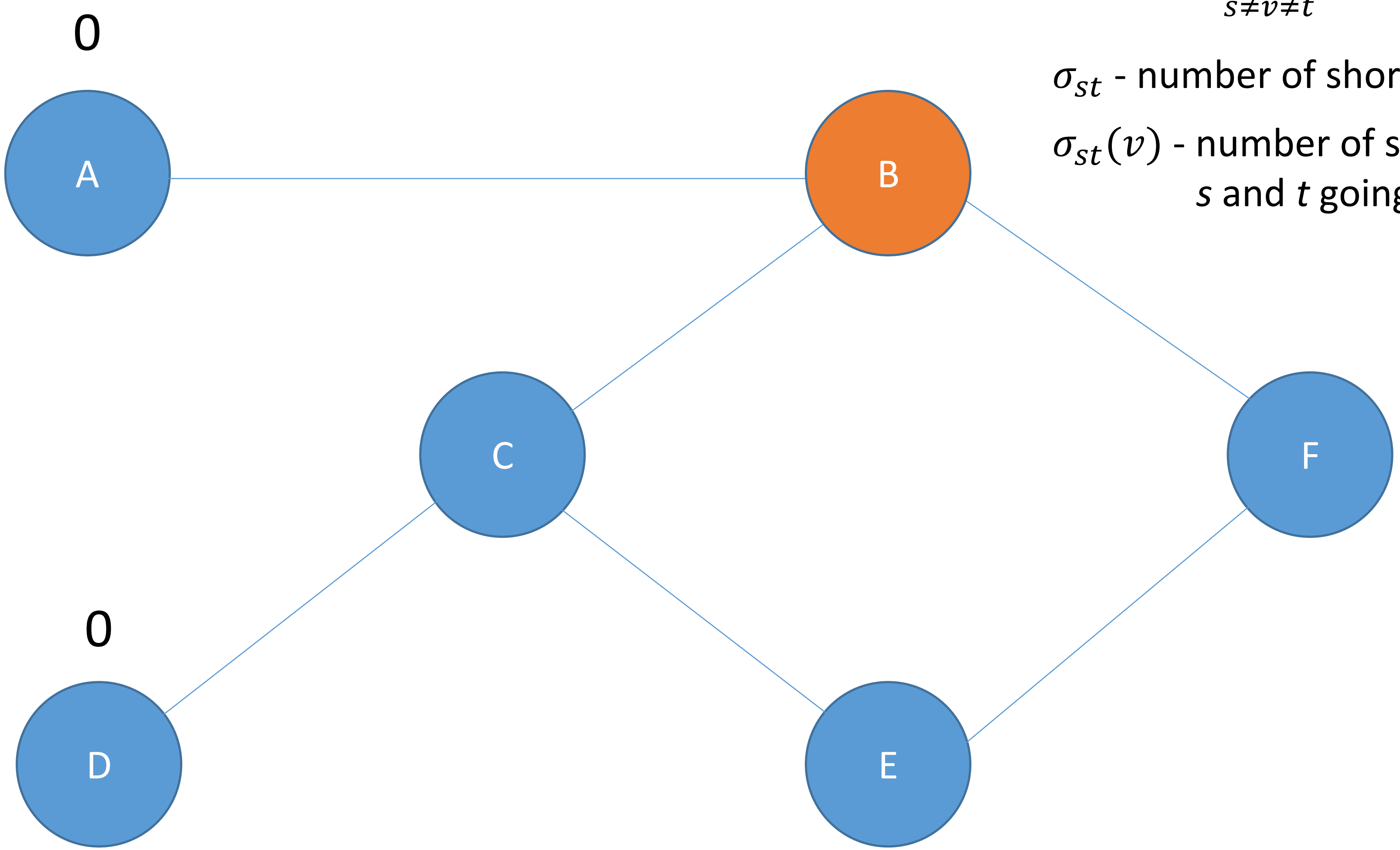
BC(B) = ?

Computation (naive)

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*



AC: A-B-C => 1/1
AD: A-B-C-D => 1/1
AE: A-B-C-E
A-B-F-E => 2/2
AF: A-B-F => 1/1
CD: X
CE: X
CF: C-B-F => 1/2
DE: X
DF: D-C-B-F => 1/2

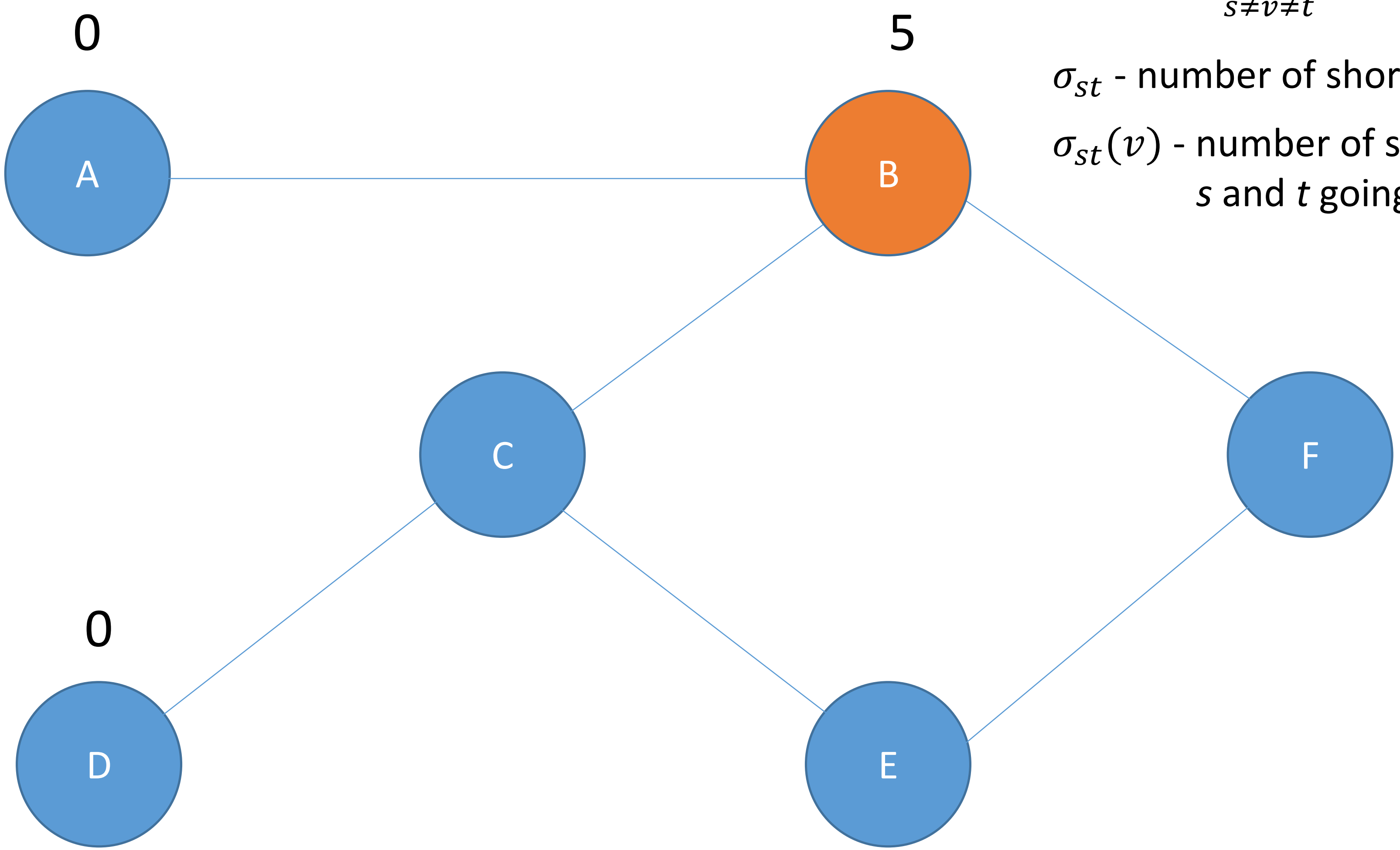
BC(B) = ?

Computation (naive)

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

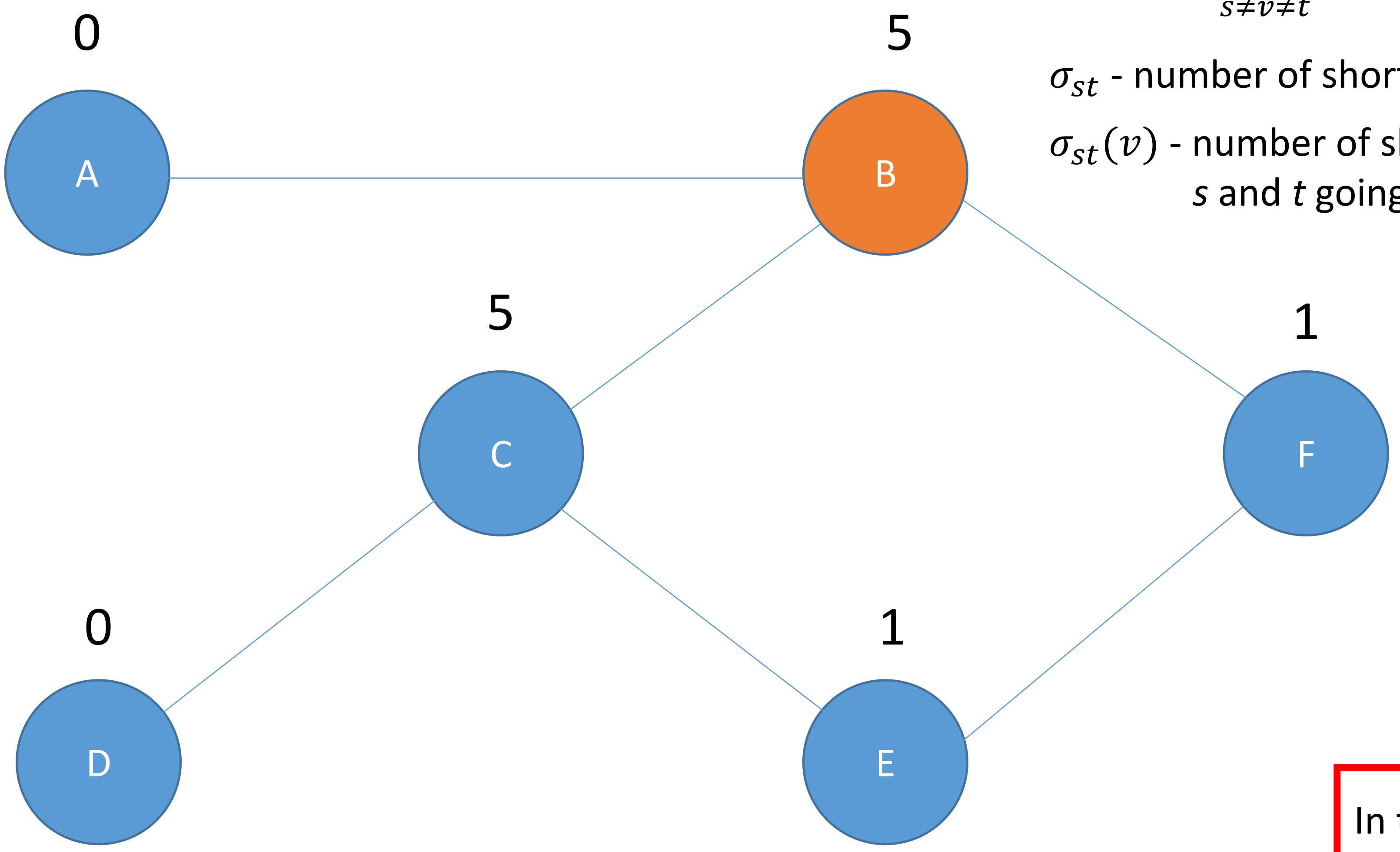
$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*



AC: A-B-C => 1/1
AD: A-B-C-D => 1/1
AE: A-B-C-E
A-B-F-E => 2/2
AF: A-B-F => 1/1
CD: X
CE: X
CF: C-B-F => 1/2
DE: X
DF: D-C-B-F => 1/2

BC(B) = 1 + 1 + 2/2 + 1/1 + 1/2 + 1/2 = 5

Computation (naive)



$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} - number of shortest paths between *s* and *t*

$\sigma_{st}(v)$ - number of shortest paths between *s* and *t* going through *v*

In traffic – G(V,E) is weighted and oriented

Computation (Brandes)

[1] U. Brandes: A Faster Algorithm for Betweenness centrality

[2] U. Brandes: On Variants of Shortest-Path Betweenness Centrality and their Generic Computation

- For each vertex s do:
  - Single-source shortest paths problem (like flooding in pipe network)
    - Use Dijsktras algorithm (weighted directed graphs)
    - During the process save visited vertices on stack
    - When processed, accumulate these values:
      - Pred[v] – because more shortest paths can go from s to v, retain the list of v predecessors
      - σ[v] – number of shortest paths from s to v
      - δ[v] – dependency – amount of betweenness that s will contribute to vertex v
  - Accumulation
    - A vertex *w* is popped from *S* until empty, starting with the furthest one from *s* and ending with *s* itself
    - Foreach predecessor of w calculates dependency with equation:
      - $\delta[v] = \delta[v] + (\sigma[v]/\sigma[w]) * (1 + \delta[w])$
    - Betweenness value of vertex w is increased by dependency of w (for first vertex in stack is 0)

Naive vs. Brandes computation

- Naive computation
  - shortest path computation for all pairs of nodes
  - $O(n^3)$ time complexity and $O(n^2)$ memory

- Brandes
  - Uses Dijkstra algorithm for single source shortest path problem and accumulation of vertex dependencies
  - $O(nm + n^2 \log n)$ time complexity and $O(n + m)$ memory

Thank you for your attention