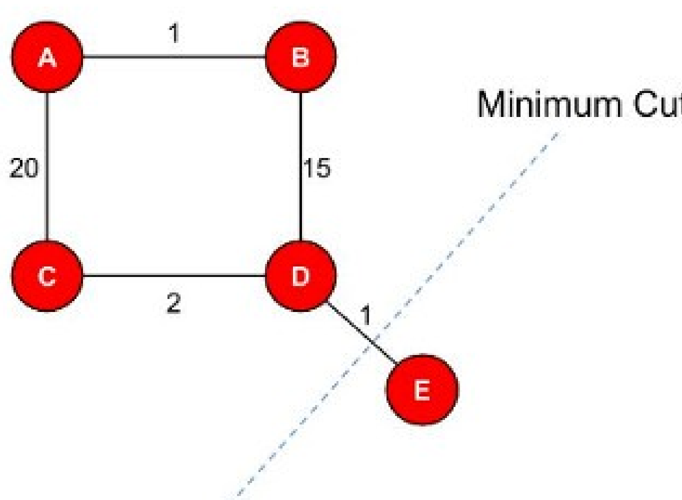
Distributed Karger-Stein Algorithm

D. Flatow, D. Penner

The Problem: Global Minimum Cut

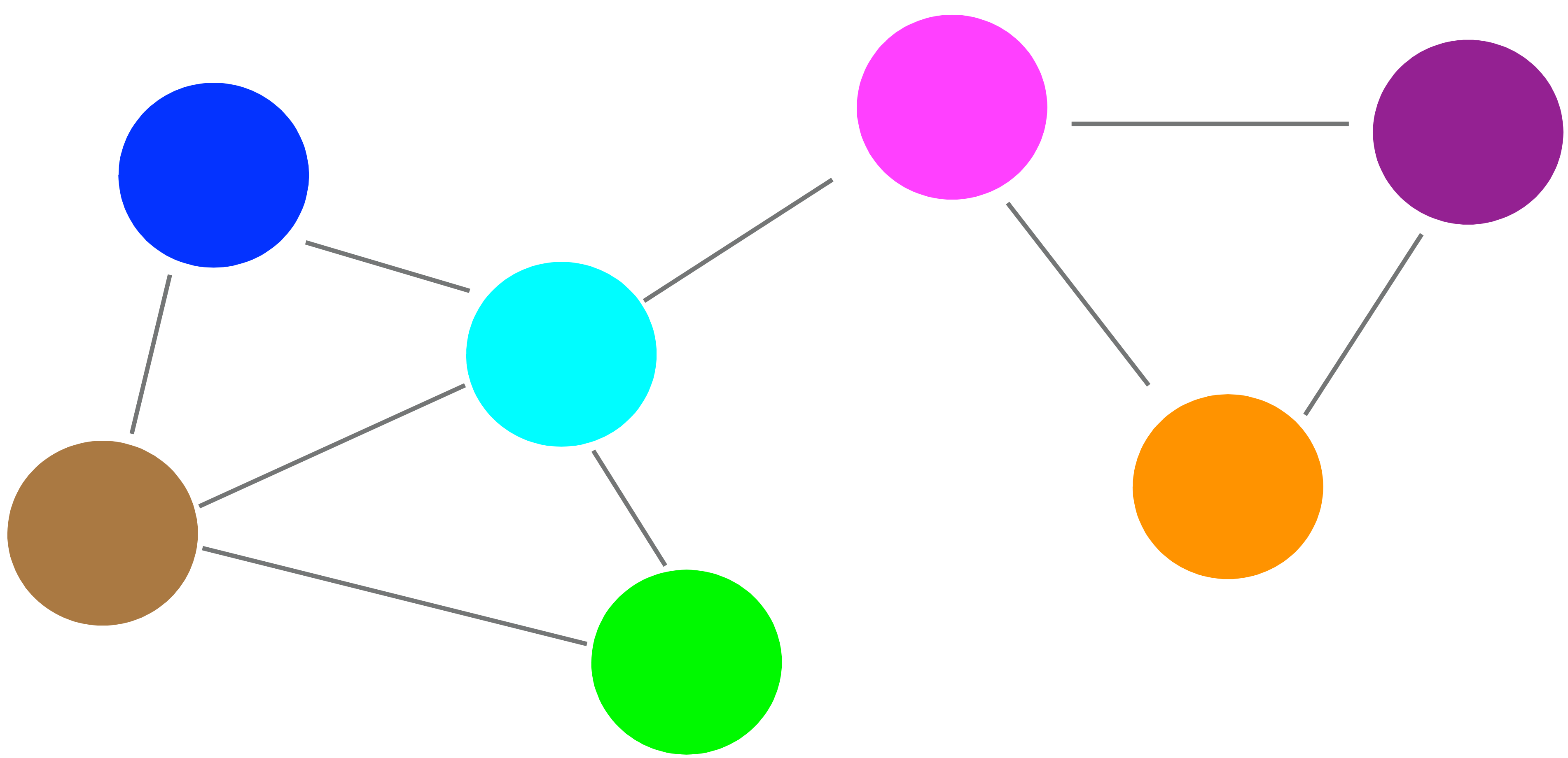
- Undirected, connected graph G = (V, E)
- Non-negative weights w: E -> R
- A cut is a subset S of V
- The cutsize of S is the sum of weights of edges leaving S
- What is the cut S of minimal cutsi ?



The Basic Algorithm: Karger’s Randomized Algorithm

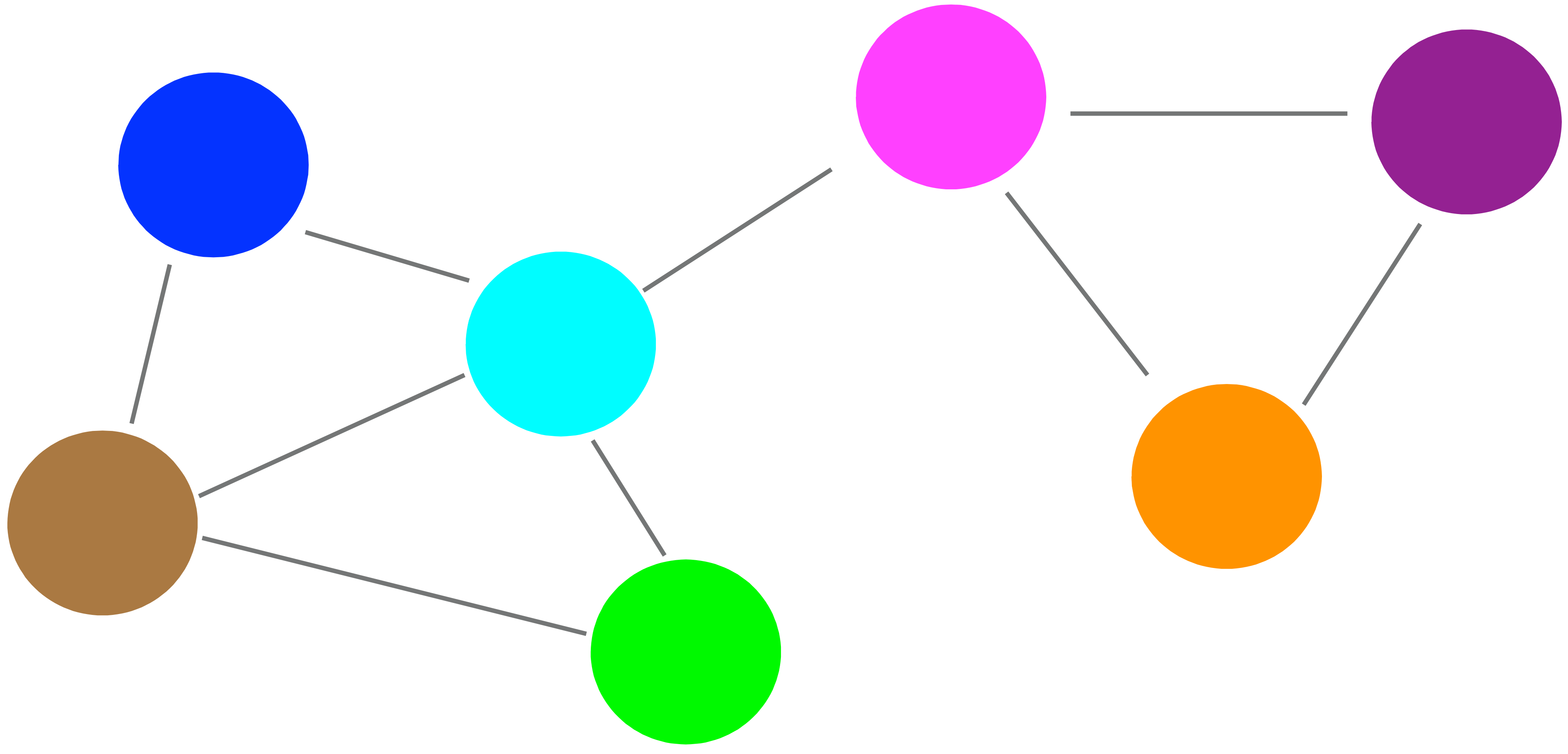
- The idea: randomly contract edges until you have a cut.
- Probability that the result is minimal is $\Omega(1/n^2)$, so we perform $\Omega(n^2 \lg n)$ trials to get a high probability of success.
- Concretely, what do we mean by ‘contract edges’?
- Give each vertex v a ‘group’ label g(v), and when contracting (u,v), change g(u) to g(v), and remove (u,v) from E.
- Sequential complexity: $n^2 \lg n$ trials, O(m) work per trial:
- $O(n^2 m \lg n)$

Distributing Karger



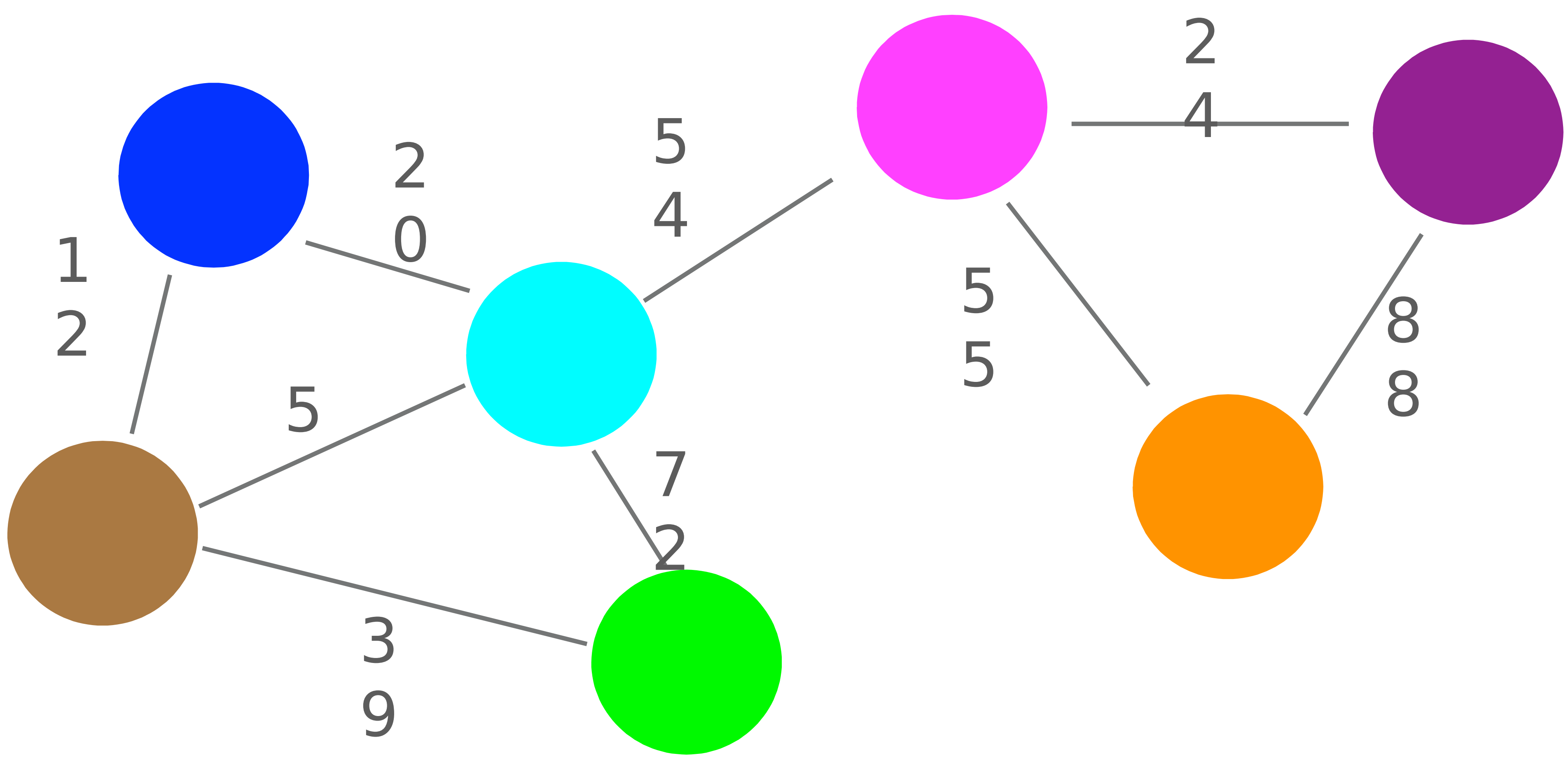
Distributing Karger

```
g = g.mapEdges(e => e.attr.copy(rank=genRandRank()))
val max_edge = g.triplets.reduce((a, b) => max(a.rank, b.rank))
val broadcast_max_edge = sc.broadcast(max_edge)
g = g.mapVertices((_, currentGroup) => groupMapper(currentGroup, broadcast_max_edge))
     .subgraph(epred = triplet => triplet.srcAttr != triplet.dstAttr)
```



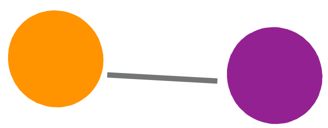
Distributing Karger

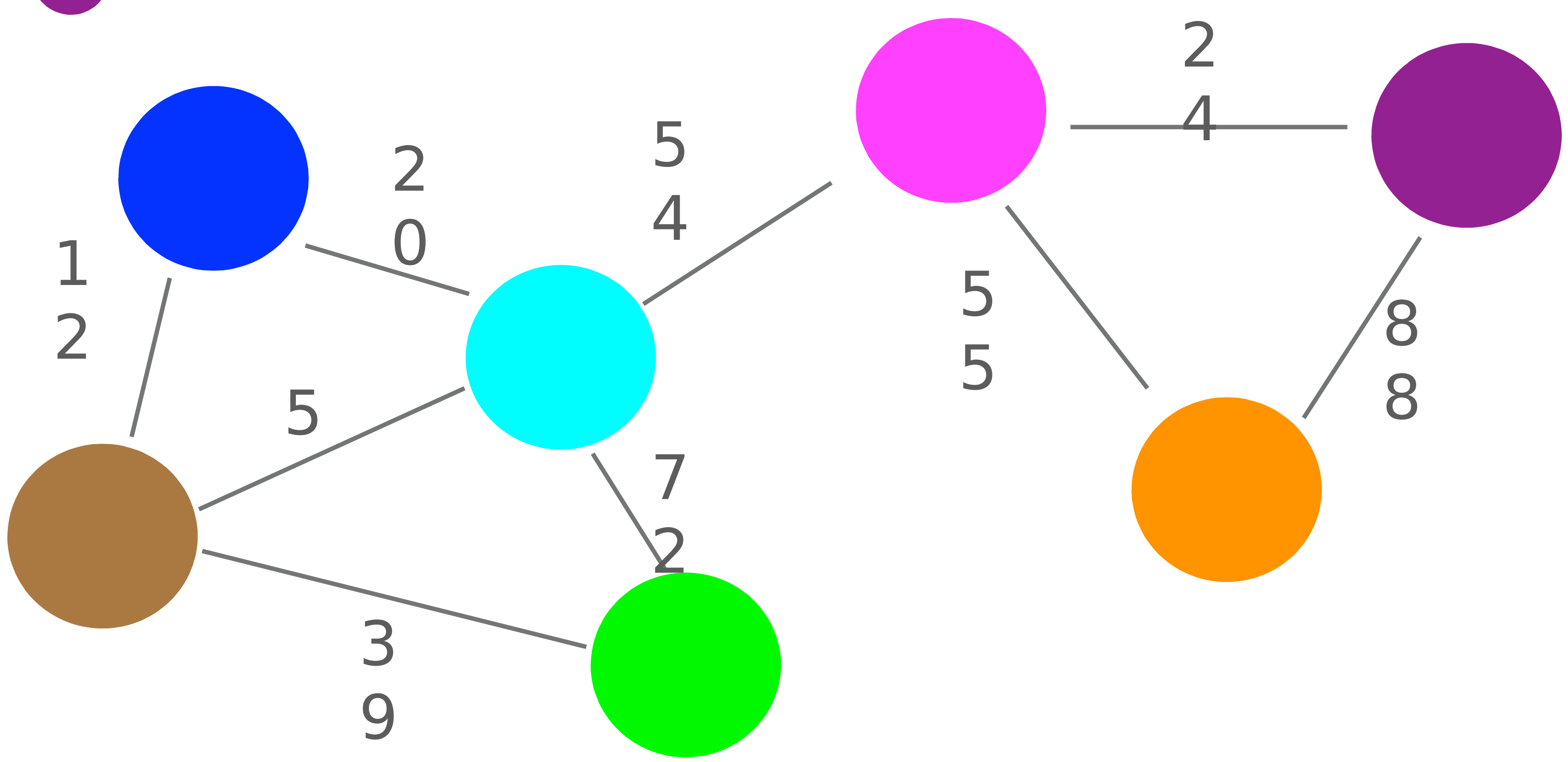
```
g = g.mapEdges(e => e.attr.copy(rank=genRandRank()))
val max_edge = g.triplets.reduce((a, b) => max(a.rank, b.rank))
val broadcast_max_edge = sc.broadcast(max_edge)
g = g.mapVertices((_, currentGroup) => groupMapper(currentGroup, broadcast_max_edge))
     .subgraph(epred = triplet => triplet.srcAttr != triplet.dstAttr)
```



Distributing Karger

```
g = g.mapEdges(e => e.attr.copy(rank=genRandRank()))
val max_edge = g.triplets.reduce((a, b) => max(a.rank, b.rank))
val broadcast_max_edge = sc.broadcast(max_edge)
g = g.mapVertices((_, currentGroup) => groupMapper(currentGroup, broadcast_max_edge))
      .subgraph(epred = triplet => triplet.srcAttr != triplet.dstAttr)
```

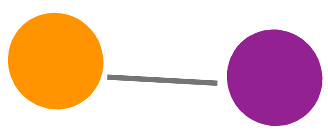
`max_edge` = 

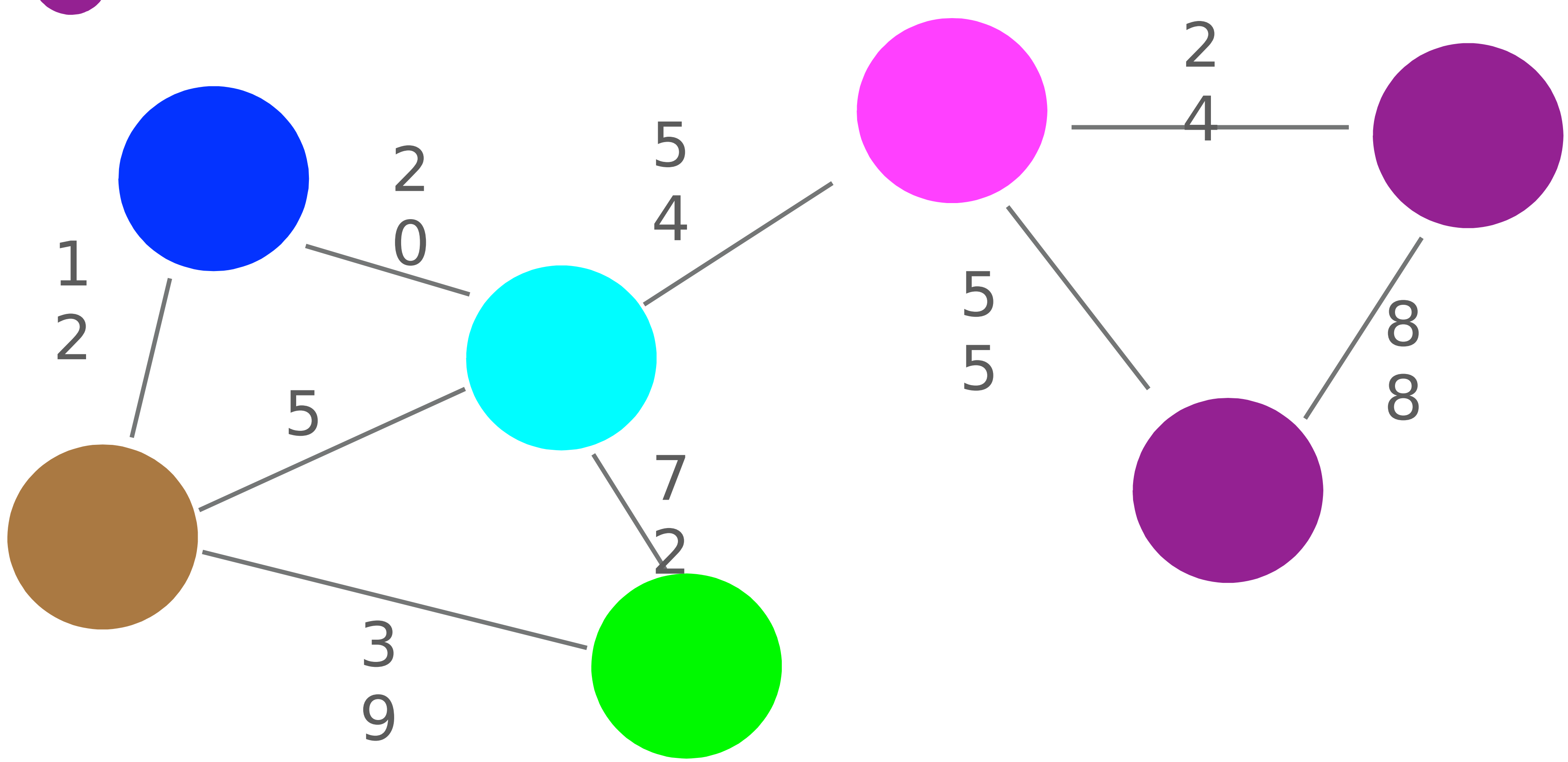


Distributing Karger

```
g = g.mapEdges(e => e.attr.copy(rank=genRandRank()))
val max_edge = g.triplets.reduce((a, b) => max(a.rank, b.rank))
val broadcast_max_edge = sc.broadcast(max_edge)
g = g.mapVertices((_, currentGroup) => groupMapper(currentGroup, broadcast_max_edge))
      .subgraph(epred = triplet => triplet.srcAttr != triplet.dstAttr)
```

max_edge =

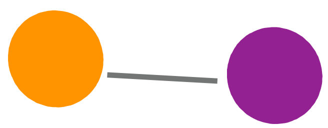


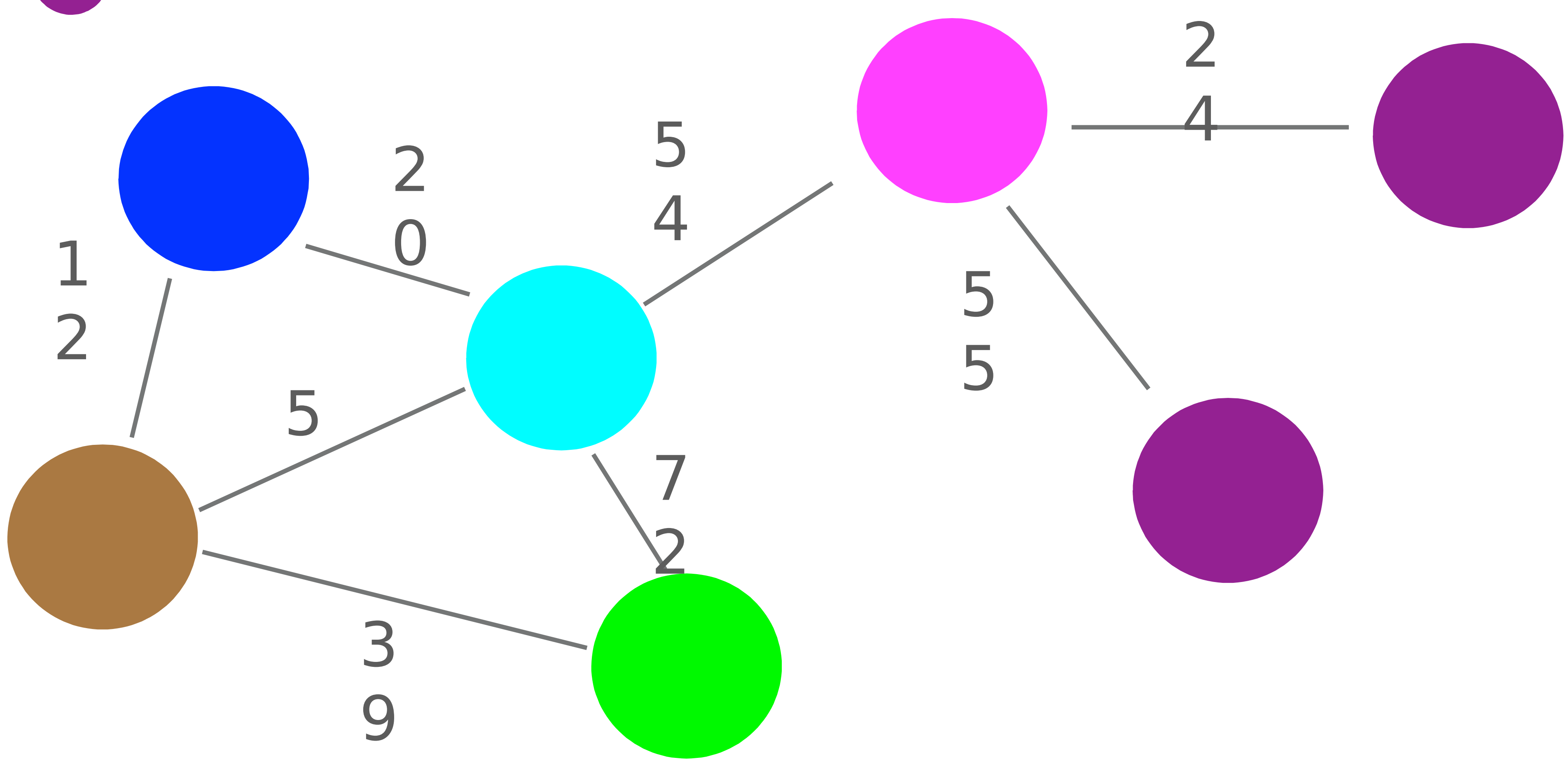


Distributing Karger

```
g = g.mapEdges(e => e.attr.copy(rank=genRandRank()))
val max_edge = g.triplets.reduce((a, b) => max(a.rank, b.rank))
val broadcast_max_edge = sc.broadcast(max_edge)
g = g.mapVertices((_, currentGroup) => groupMapper(currentGroup, broadcast_max_edge))
     .subgraph(epred = triplet => triplet.srcAttr != triplet.dstAttr)
```

`max_edge` =

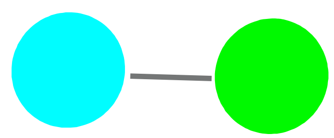


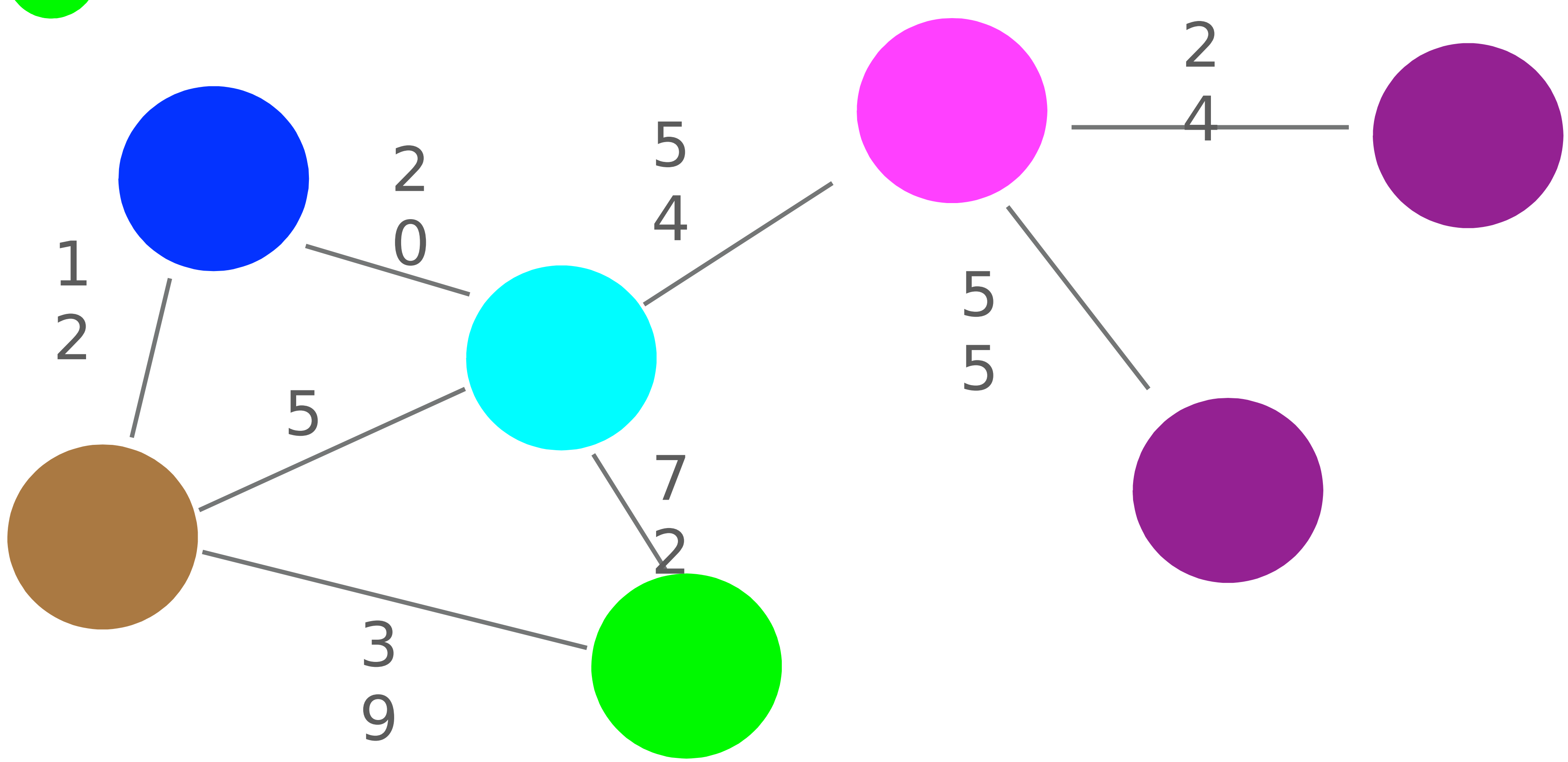


Distributing Karger

repeat

```
g = g.mapEdges(e => e.attr.copy(rank=genRandRank()))
val max_edge = g.triplets.reduce((a, b) => max(a.rank, b.rank))
val broadcast_max_edge = sc.broadcast(max_edge)
g = g.mapVertices((_, currentGroup) => groupMapper(currentGroup, broadcast_max_edge))
      .subgraph(epred = triplet => triplet.srcAttr != triplet.dstAttr)
```

`max_edge` = 

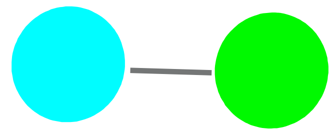


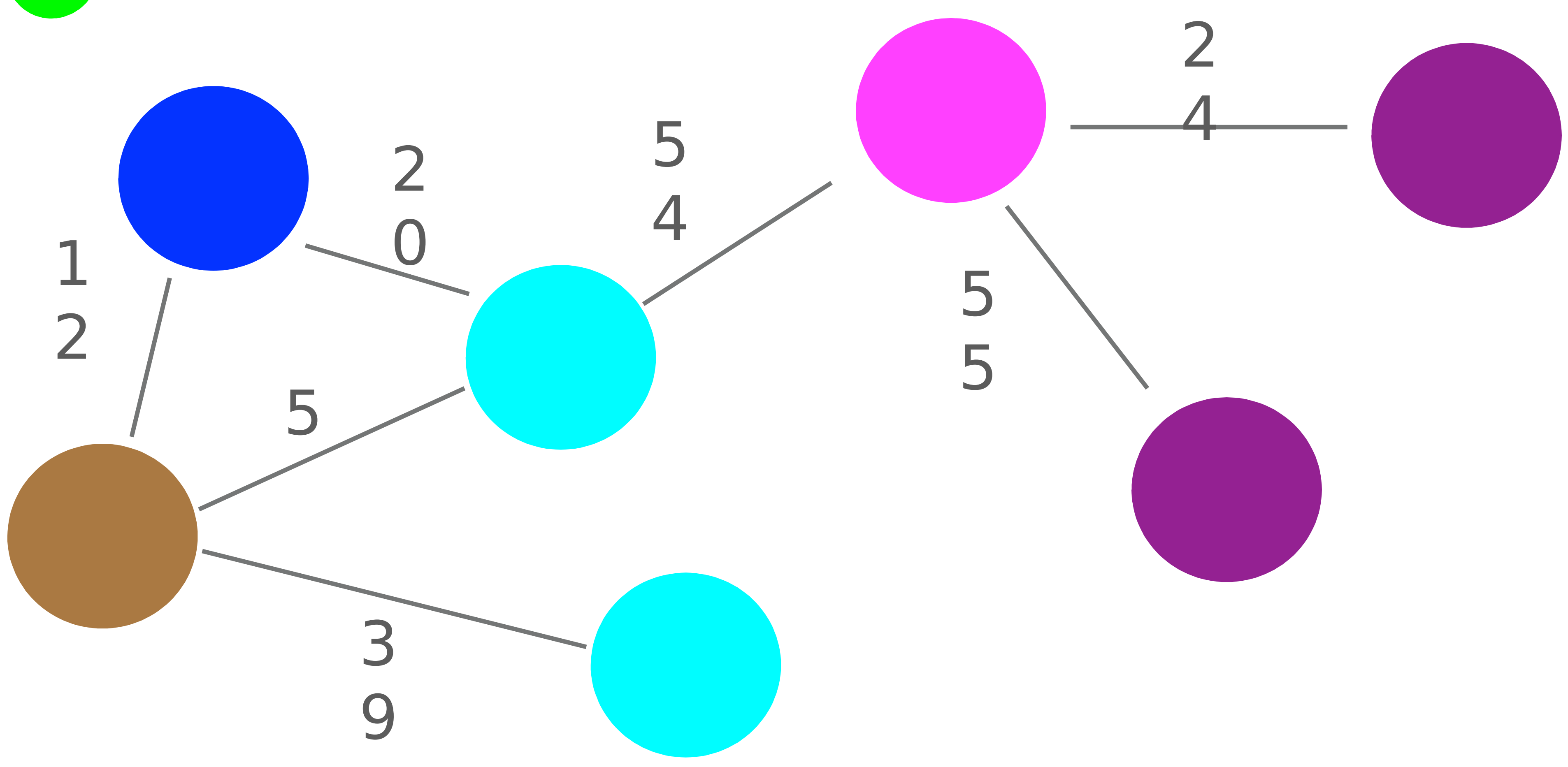
Distributing Karger

repeat

```
g = g.mapEdges(e => e.attr.copy(rank=genRandRank()))
val max_edge = g.triplets.reduce((a, b) => max(a.rank, b.rank))
val broadcast_max_edge = sc.broadcast(max_edge)
g = g.mapVertices((_, currentGroup) => groupMapper(currentGroup, broadcast_max_edge))
      .subgraph(epred = triplet => triplet.srcAttr != triplet.dstAttr)
```

`max_edge` =



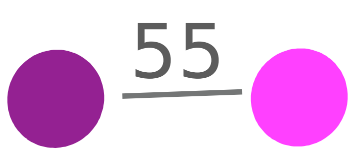


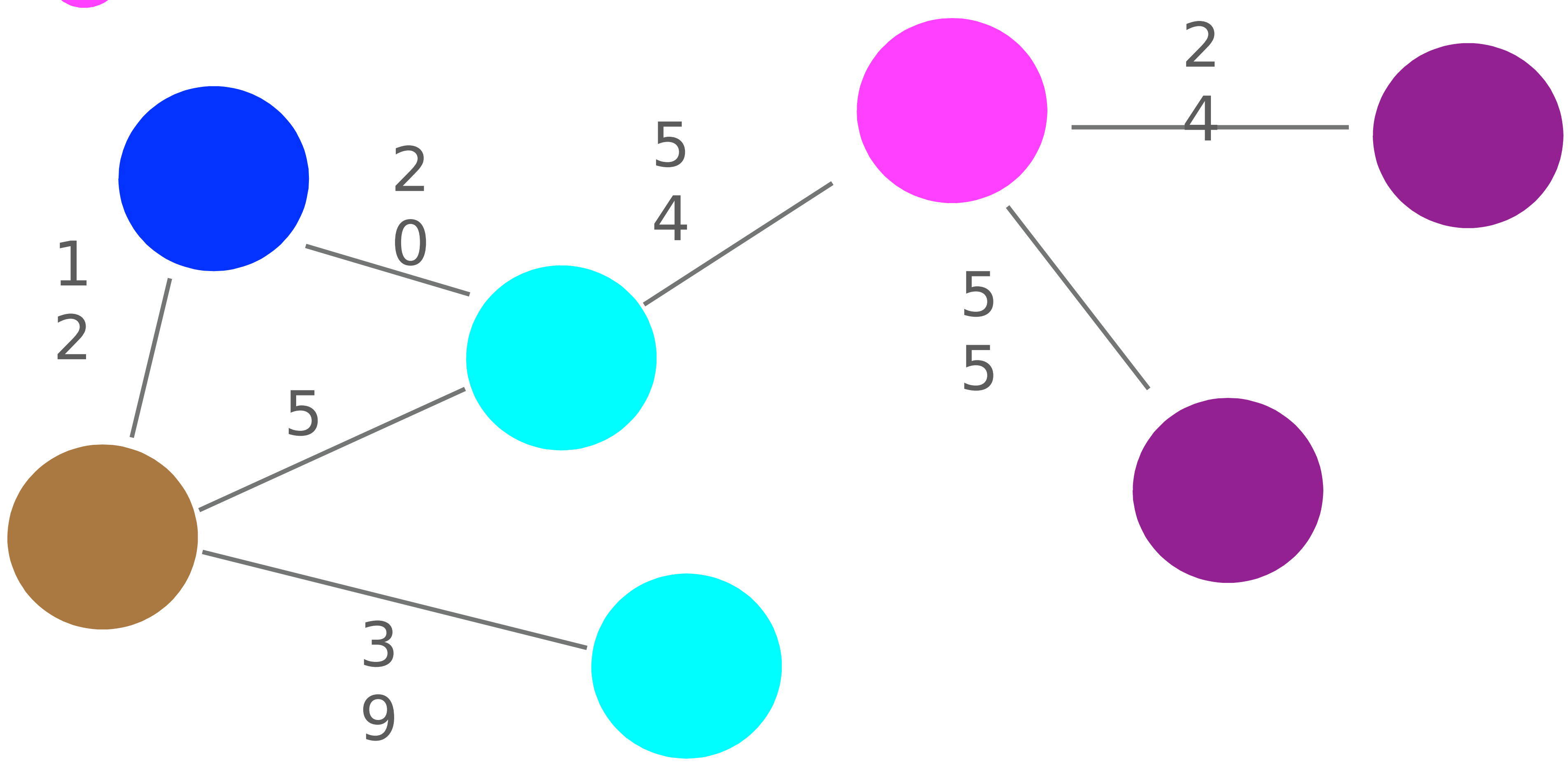
Distributing Karger

repeat

```
g = g.mapEdges(e => e.attr.copy(rank=genRandRank()))
val max_edge = g.triplets.reduce((a, b) => max(a.rank, b.rank))
val broadcast_max_edge = sc.broadcast(max_edge)
g = g.mapVertices((_, currentGroup) => groupMapper(currentGroup, broadcast_max_edge))
     .subgraph(epred = triplet => triplet.srcAttr != triplet.dstAttr)
```

`max_edge` =



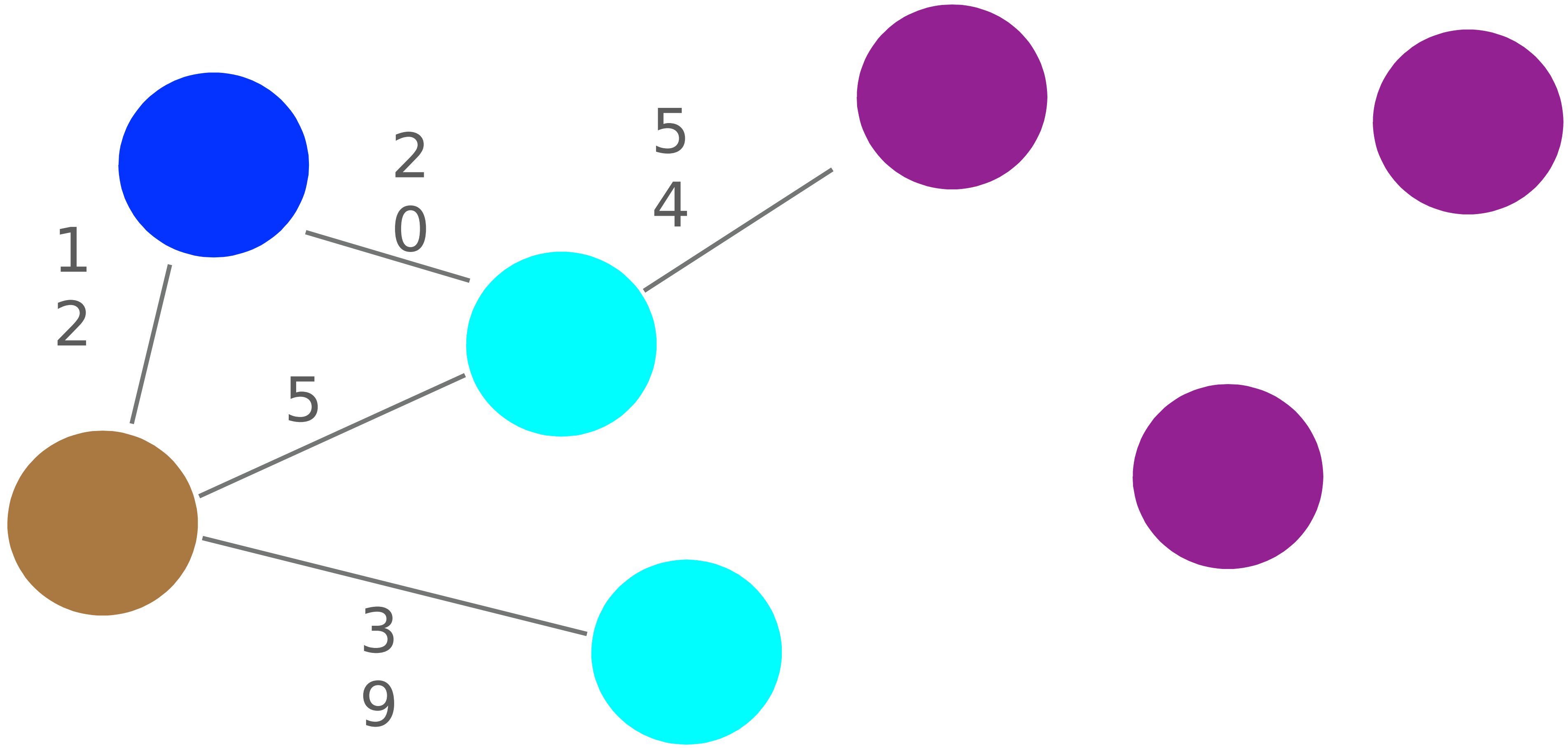


Distributing Karger

repeat

```
g = g.mapEdges(e => e.attr.copy(rank=genRandRank()))
val max_edge = g.triplets.reduce((a, b) => max(a.rank, b.rank))
val broadcast_max_edge = sc.broadcast(max_edge)
g = g.mapVertices((_, currentGroup) => groupMapper(currentGroup, broadcast_max_edge))
      .subgraph(epred = triplet => triplet.srcAttr != triplet.dstAttr)
```

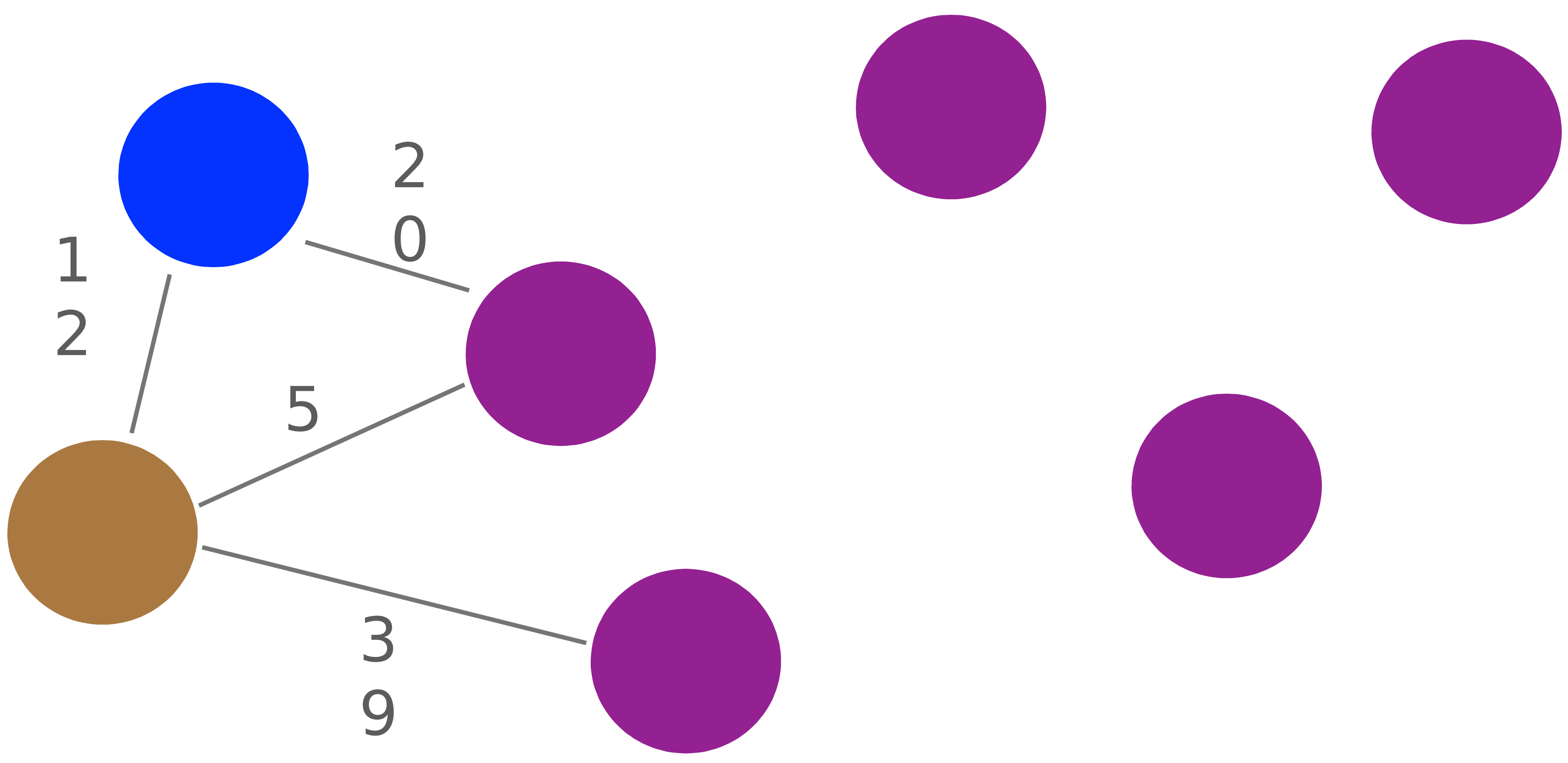
`max_edge` =



Distributing Karger

repeat

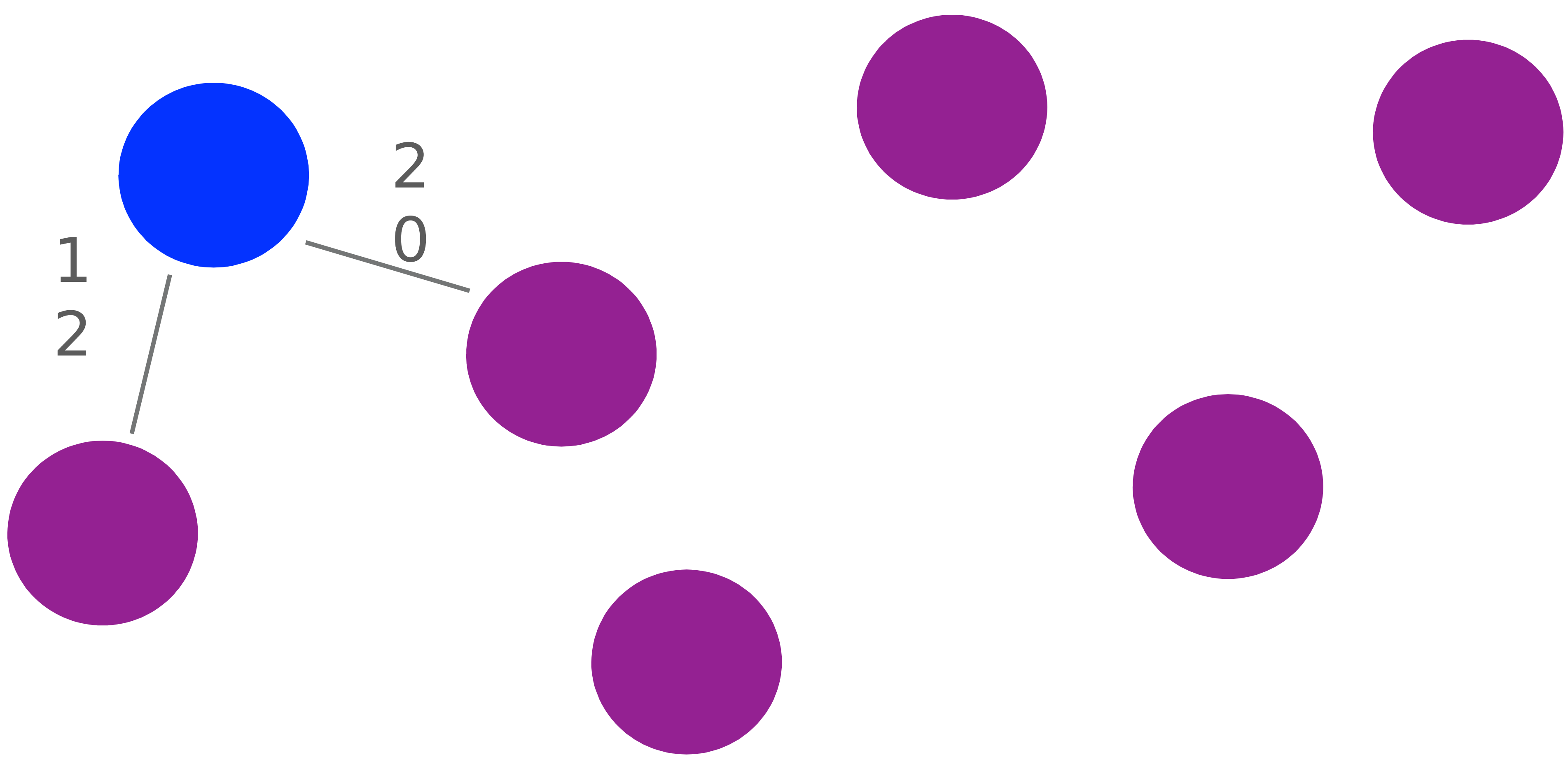
```
g = g.mapEdges(e => e.attr.copy(rank=genRandRank()))
val max_edge = g.triplets.reduce((a, b) => max(a.rank, b.rank))
val broadcast_max_edge = sc.broadcast(max_edge)
g = g.mapVertices((_, currentGroup) => groupMapper(currentGroup, broadcast_max_edge))
     .subgraph(epred = triplet => triplet.srcAttr != triplet.dstAttr)
```



Distributing Karger

repeat

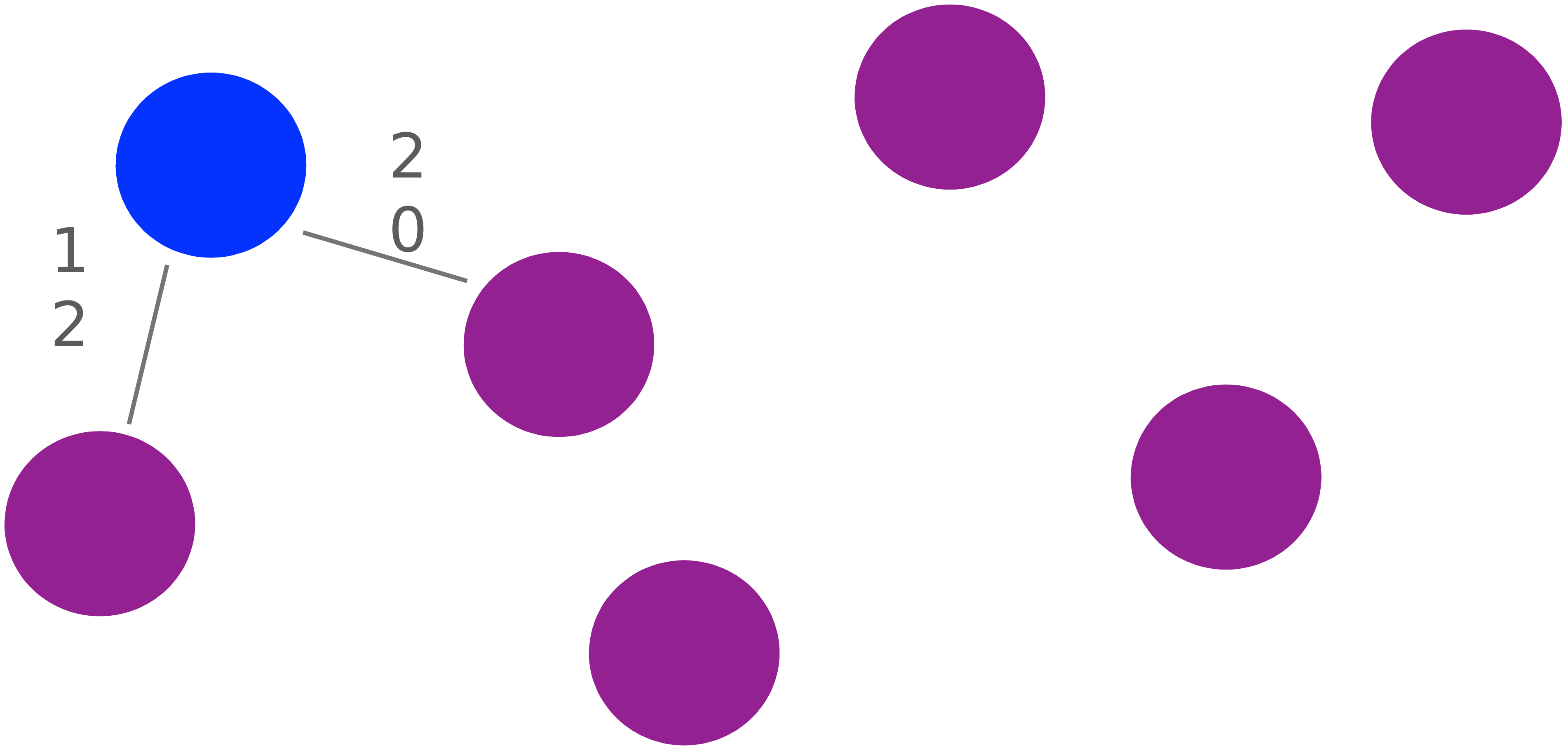
```
g = g.mapEdges(e => e.attr.copy(rank=genRandRank()))
val max_edge = g.triplets.reduce((a, b) => max(a.rank, b.rank))
val broadcast_max_edge = sc.broadcast(max_edge)
g = g.mapVertices((_, currentGroup) => groupMapper(currentGroup, broadcast_max_edge))
      .subgraph(epred = triplet => triplet.srcAttr != triplet.dstAttr)
```



Distributing Karger

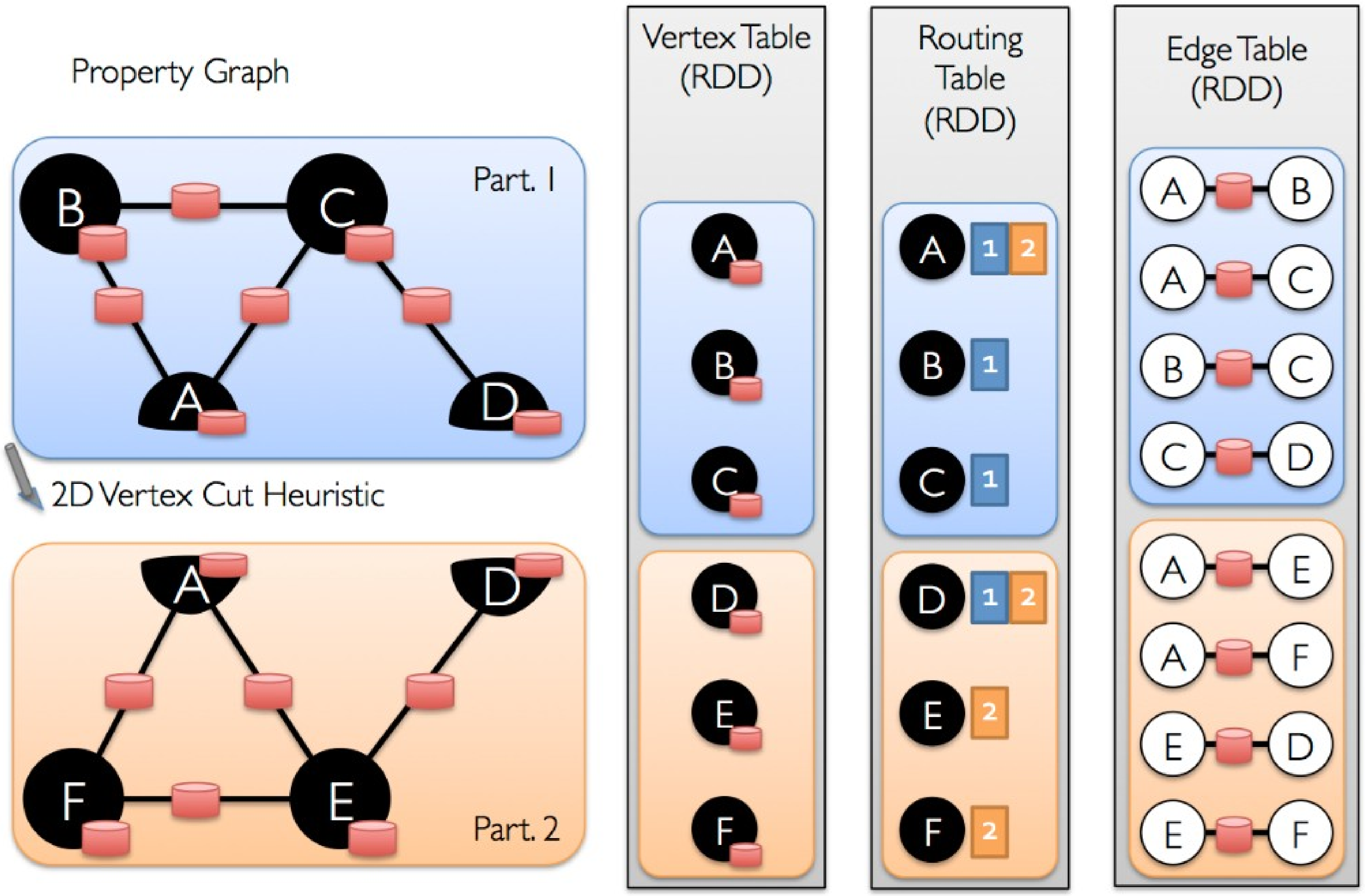
```
g.edges.reduce(_ + _)
```

min_cut_value = 12 + 20



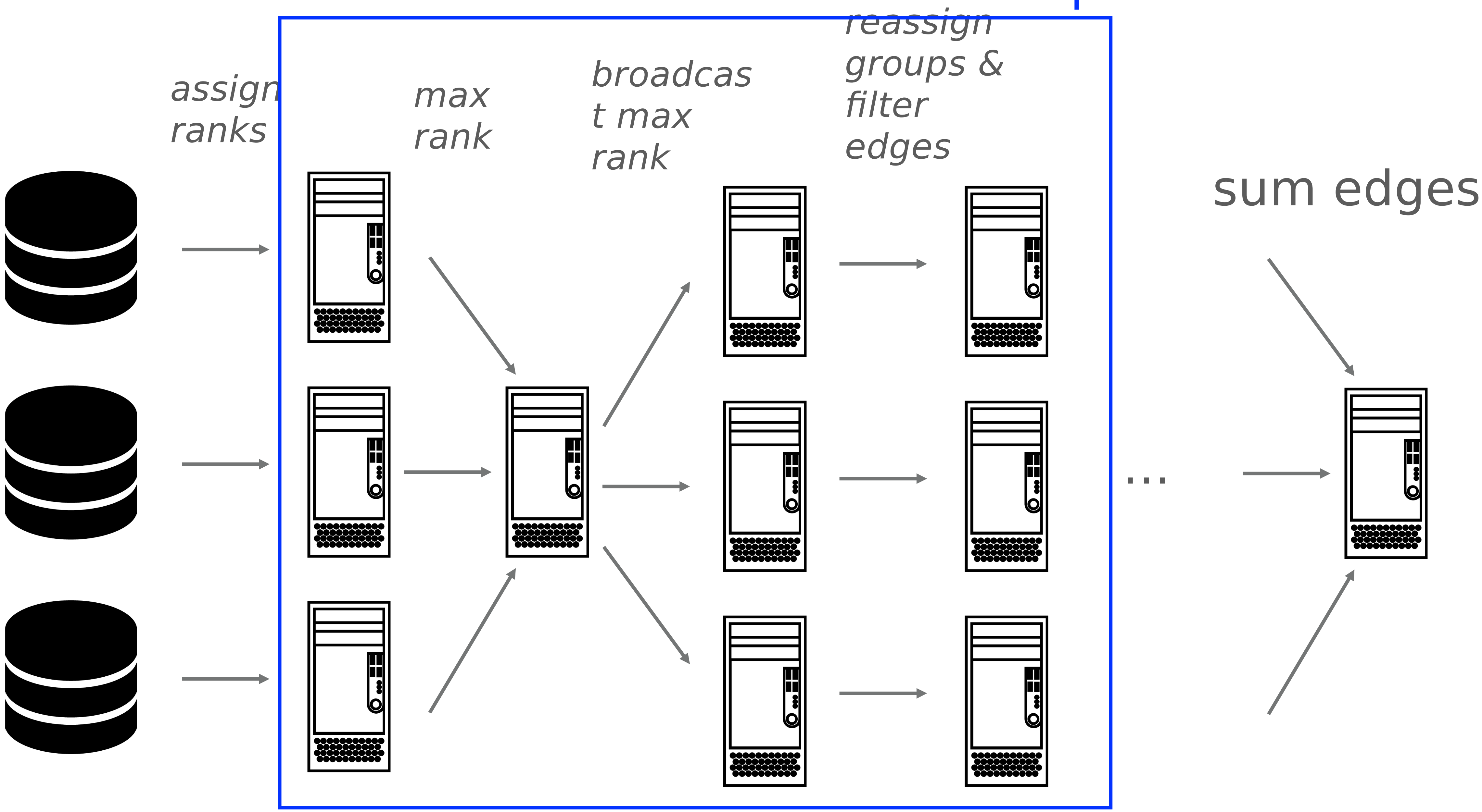
Distributing Karger

- Data Representation: Vertex Cut



Distributing Karger

- one iteration:



Distributed Karger: One edge contraction

- Communication:
  - computing max rank & broadcast.
  - Shuffle size: O(k)
- Depth on each machine:
  - computing max rank: O(log(m/k))
  - assign ranks: O(1)
  - reassign groups: O(1)
  - filter edges: O(1)

Distributed Karger: n-2 edge contractions

- Communication:

  shuffle size: O(nk)

- Depth on each machine:

  O(n log(m/k))

Distributed Karger: Total communication & Depth

- Communication:

  Total shuffle size: $O(n^2 * \log(n) * n * k)$

- Depth:

  Total depth = depth of one trial = $O(n \log(m/k))$