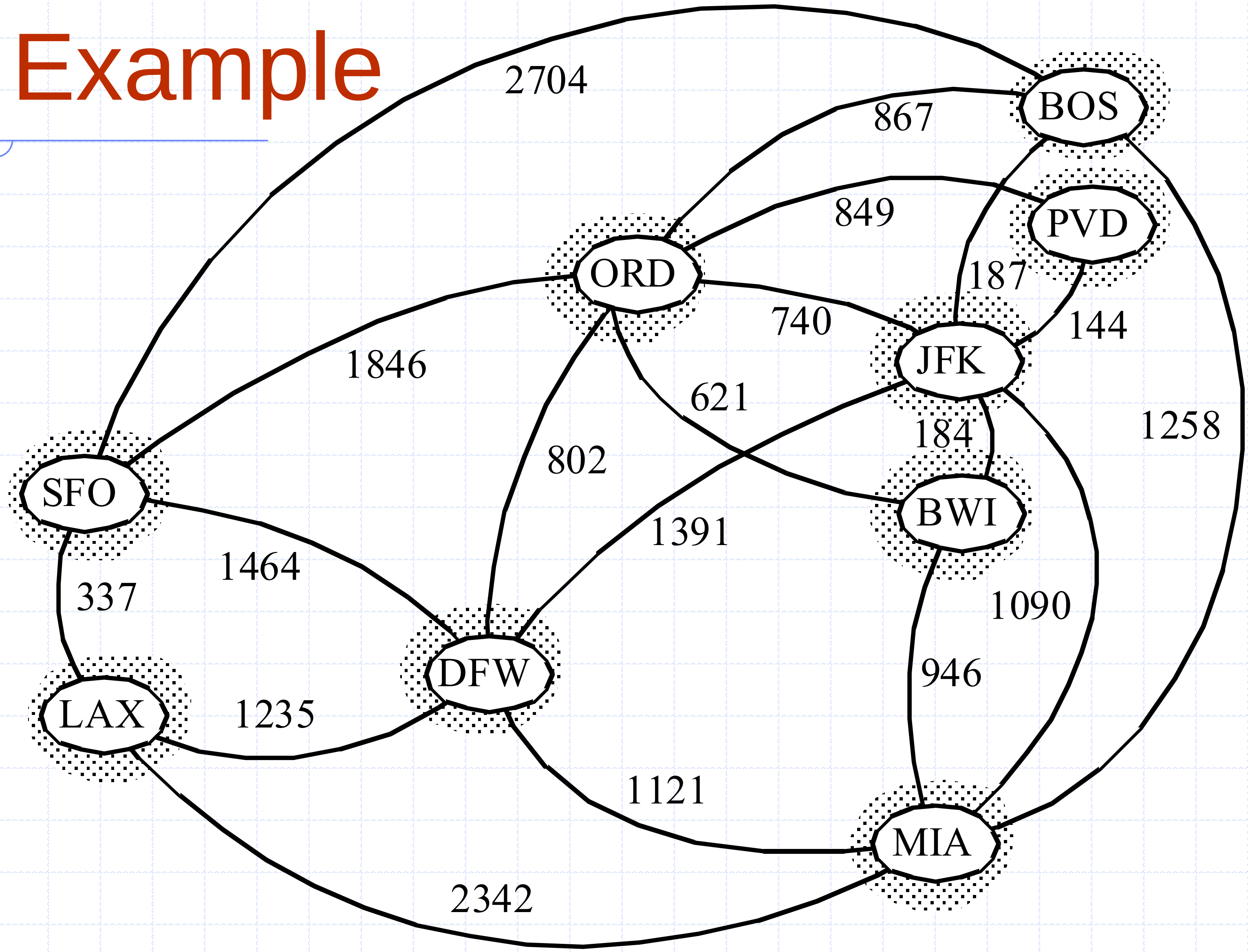
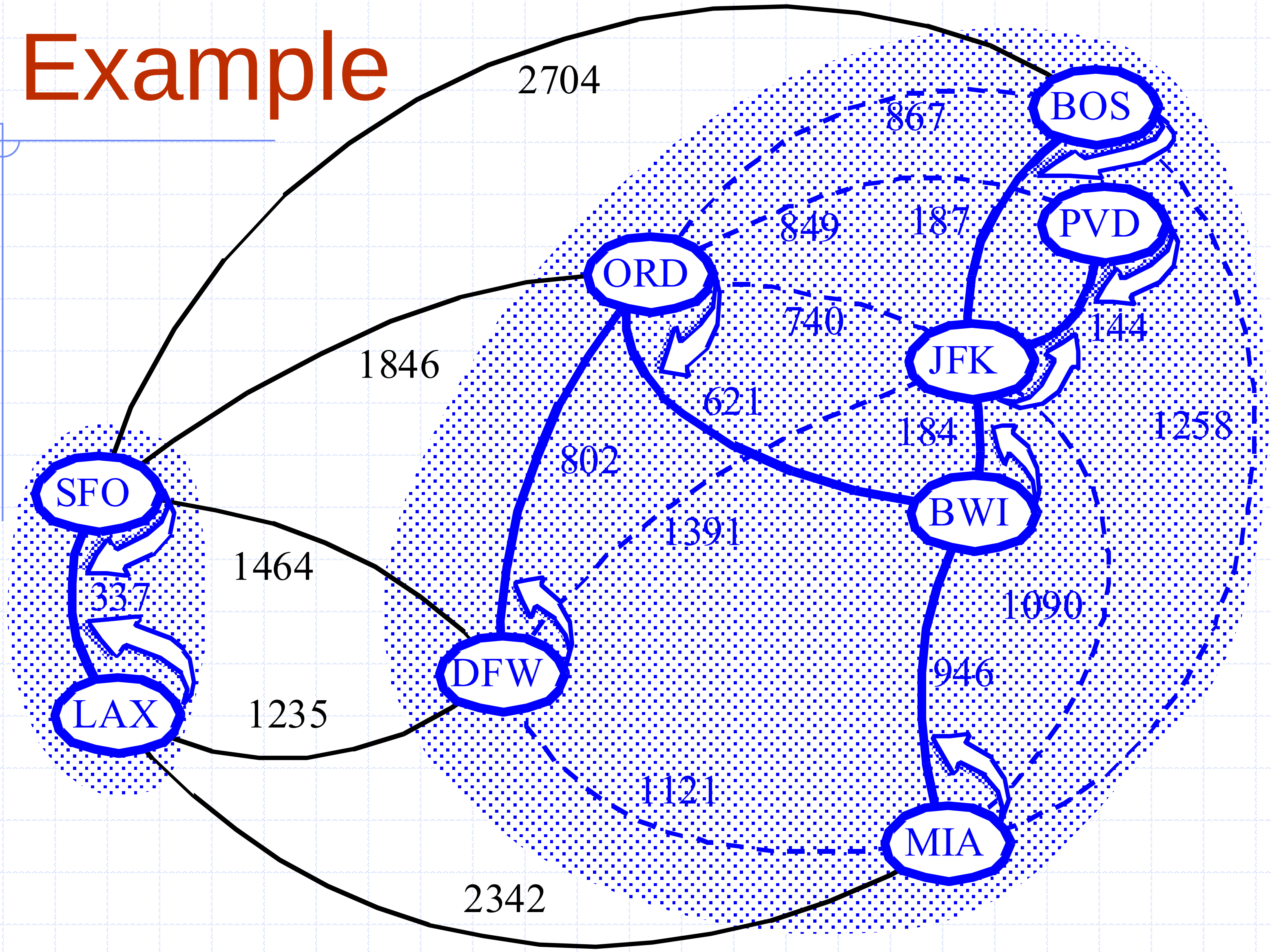
Baruvka's Algorithm (1926)

Template Method Solution <u>without</u> Making a Clone of G

Baruvka Example

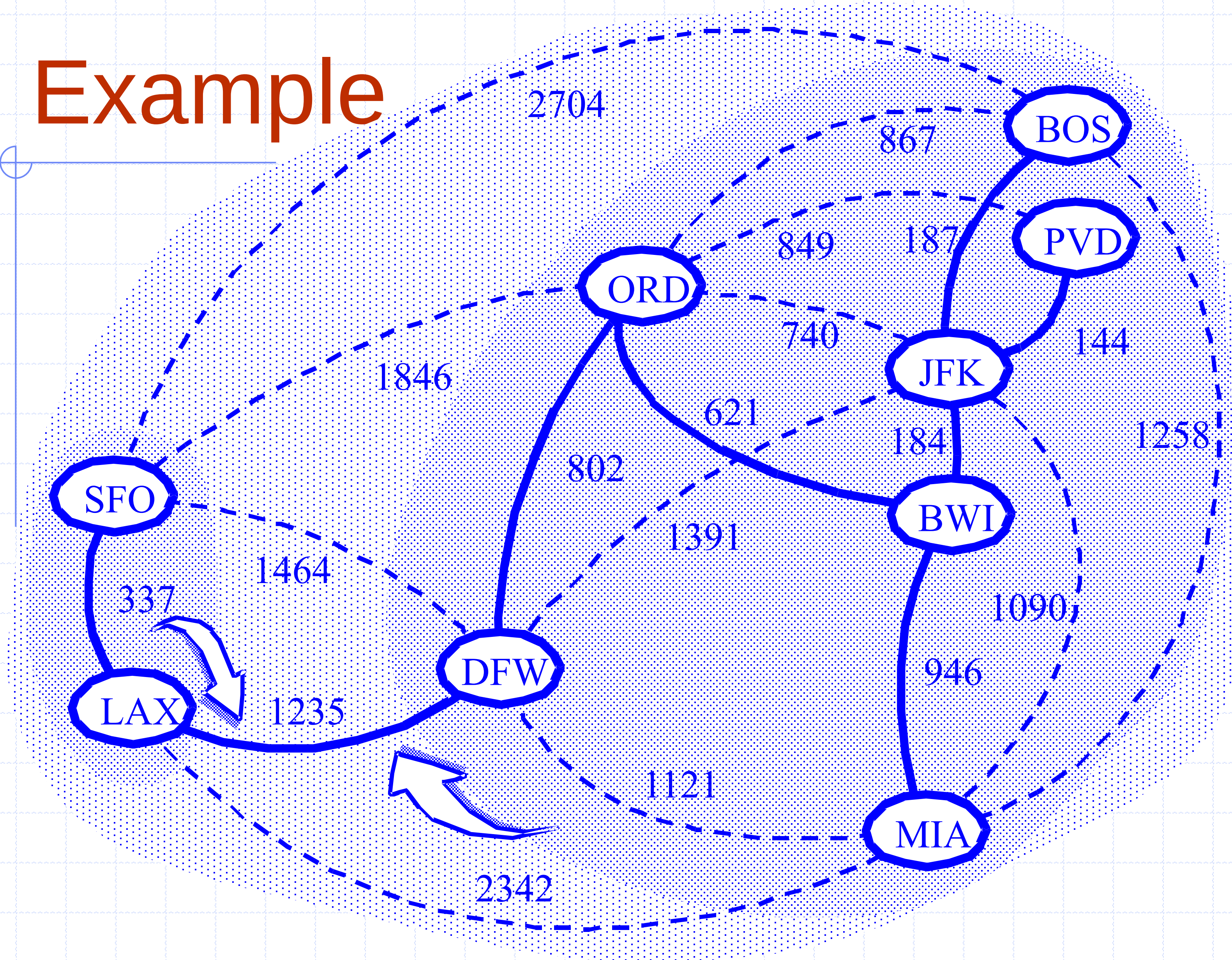


Example



2704
BOS
867
849
187
PVD
ORD
740
144
1846
JFK
621
1258
184
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example


2704
BOS
867
PVD
849
187
ORD
740
144
JFK
1846
621
1258
184
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Baruvka's Algorithm (from Lecture 14)

Algorithm ***BaruvkaMST***(***G***)
  for each e $\in$ ***G.edges()*** **do**
    setMSTLabel(e, ***NOT_IN_MST****)* {no edges in MST}
  numEdges $\leftarrow$ *0*   {numEdges is an instance variable}
  while *numEdges* < ***n***-1 **do**
    count $\leftarrow$ labelVerticesOfEachComponent(G) {BFS}
    insertSmallest-WeightEdges(G, count)
  return ***G***

Template Version of BFS

```
Algorithm BFS(G) {all components}
  Input graph G
  Output labeling of the edges of G as
    discovery edges and cross edges

    initResult(G)
    for all  u ∈ G.vertices() do
        setLabel(u, UNEXPLORED)
        postInitVertex(u)
    for all  e ∈ G.edges() do
        setLabel(e, UNEXPLORED)
        postInitEdge(e)
    for all  v ∈ G.vertices() do
        if  isNextComponent(G, v)
            preComponentVisit(G, v)
            BFScomponent(G, v)
            postComponentVisit(G, v)

    return result(G)

Algorithm isNextComponent(G, v)
    return getLabel(v) = UNEXPLORED
```

```
Algorithm BFScomponent(G, s) {1 component}
    setLabel(s, VISITED)
    Q ← new empty Queue
    Q.enqueue(s)
    startBFScomponent(G, s)
    while ¬Q.isEmpty()  do
        v ← Q.dequeue()
        preVertexVisit(G, v)
        for all  e ∈ G.incidentEdges(v)  do
            preEdgeVisit(G, v, e, w)
            if  getLabel(e) = UNEXPLORED
                w ← opposite(v, e)
                unexploredEdgeVisit(G, v, e, w)
                if getLabel(w) = UNEXPLORED
                    preDiscEdgeVisit(G, v, e, w)
                    setLabel(e, DISCOVERY)
                    setLabel(w, VISITED)
                    Q.enqueue(w)
                    postDiscEdgeVisit(G, v, e, w)
                else
                    setLabel(e, CROSS)
                    crossEdgeVisit(G, v, e, w)
        postVertexVisit(G, v)

    finishBFScomponent(G, s)
```

Label Vertices of Each Component (subclass methods of BFS template)

This algorithm runs in O(n+m) time. Why?

Algorithm **labelVerticesOfEachComponent(G)**
    return BFS(G)   {label vertices of G with component numbers}

Algorithm ***initResult***(***G***)
    count ← 0   {initialize count to zero}

Algorithm ***preEdgeVisit***(***G, v, e, w***)
    if getMSTLabel(e) = NOT_IN_MST **then**   {is e in the MST}
        setLabel(***e, SKIP***)   {so BFS only visits edges in MST}

Algorithm ***postComponentVisit***(***G, v***)
    count ← count + 1   {add one to count}

Algorithm ***preVertexVisit***(***G, v***)
    setComponentNum(***v,*** count)

Algorithm ***result***(***G***)
    return count   {return count which is the number of components}

Insert Minimum-Weight Edges Going Out from each Component

What is the running time?

```
Algorithm insertSmallest-WeightEdges(G, count)
    minEdges ← new array of size count  {could use hashtable based Dictionary}
    for i ← 0 to count - 1 do
        minEdges[i] ← ∅

    BFS(G)      {search for smallest edges connecting different components}
    for i ← 0 to count - 1 do
        e ← minEdges[i]
        if getMSTLabel(e) = NOT_IN_MST then       {is e already in MST}
            setMSTLabel(e, IN_MST)     {insert e into MST}
            numEdges ← numEdges + 1  {increase number of edges in MST}
```

{could be done by traversing G.edges() in a loop instead of during a BFS}

```
Algorithm preEdgeVisit(G, v , e , w)   {called during BFS(G) above}
    cv ← getComponentNum(v)
    cw ← getComponentNum(w)
    if cv ≠ cw then       {does e connect two different components of MST}
        if minEdges[cv] = ∅  then
            minEdges[cv] ← e       {insert new minimum}
        else
            min ← weight(minEdges[cv])    {current min weight for component cv}
            if min > weight(e)  then
                minEdges[cv] ← e         {insert new minimum}
```

An Issue Not Handled by the above algorithm

- Edges with the same weight
- Therefore,
  - Could insert more than n-1 edges
  - Or could create one or more cycles
- How could we fix this?

Insert Minimum-Weight Edges then remove cycles

What is the running time?

```
Algorithm insertSmallest-WeightEdges(G, count)
    minEdges ← new array of size count
    for i ← 0 to count - 1 do
        minEdges[i] ← ∅

    BFS(G)      {search for smallest edges connecting different components}
    for i ← 0 to count - 1 do
        e ← minEdges[i]
        if getMSTLabel(e) = NOT_IN_MST then      {is e already in MST}
            setMSTLabel(e, IN_MST)     {insert e into MST}
            numEdges ← numEdges + 1  {increase number of edges in MST}
    removeCycles(G)    {remove the edge with highest weight in each cycle}

Algorithm preEdgeVisit(G, v , e , w)   {called during BFS(G) above}
    cv ← getComponentNum(v)
    cw ← getComponentNum(w)
    if cv ¬= cw then       {does e connect two different components of MST}
        currentMinE ← minEdges[cv] {min edge for component cv}
        if currentMinE = ∅  then
            minEdges[cv] ← e       {insert new minimum}
        else
            min ← weight(currentMinE)    {current min weight for component cv}
            if min > weight(e)  then
                minEdges[cv] ← e        {insert new minimum}
```

DFS is better for Cycle Finding (Improved version)

```
Algorithm DFS(G) {top level}
    Input graph G
    Output the edges of G are labeled
           as discovery and back edges

  initResult(G)
  for all u ∈ G.vertices() do
      setLabel(u, UNEXPLORED)
      postInitVertex(u)
  for all e ∈ G.edges() do
      setLabel(e, UNEXPLORED)
      postInitEdge(e)
  for all v ∈ G.vertices() do
      if getLabel(v) = UNEXPLORED
          preComponentVisit(G, v)
          DFS(G, v)
          postComponentVisit(G, v)

  result(G)
```

```
Algorithm DFS(G, v)
  setLabel(v, VISITED)
  startVertexVisit(G, v)
  for all e ∈ G.incidentEdges(v)
      w ← opposite(v,e)
      preEdgeVisit(G, v, e, w)
      if getLabel(e) = UNEXPLORED
          if getLabel(w) = UNEXPLORED
              setLabel(e, DISCOVERY)
              preDiscoveryTraversal(G, v, e, w)
              DFS(G, w)
              postDiscoveryTraversal(G, v, e, w)
          else
              setLabel(e, BACK)
              backEdgeVisit(G, v, e, w)
      postEdgeVisit(G, v, e, w)

  finishVertexVisit(G, v)
```

Overriding template methods in subclass to Remove Cycles

```
Algorithm removeCycles(G)
        repeat
            DFS(G)
            if  cycleFound   then numEdges ← numEdges - 1  {decreased edges in MST}
        until ¬ cycleFound

Algorithm initResult(G)
   cycleFound ←  false
Algorithm postInitEdge(e)
   if  getMSTLabel(e) = NOT_IN_MST  then    setLabel(e, SKIP)
Algorithm startVertexVisit(G, v)
    if  ¬ cycleFound   then            S.push(v)
Algorithm finishVertexVisit(G, v)
    if  ¬ cycleFound   then            S.pop()
Algorithm preDiscoveryTraversal(G, v, e, w)
    if  ¬ cycleFound   then            S.push(e)
Algorithm postDiscoveryTraversal(G, v, e, w)
    if  ¬ cycleFound   then            S.pop()
Algorithm backEdgeTraversal(G, v, e, w)
    if  ¬ cycleFound   then
            max ← weight(e)
            maxE ← e
            u ← S.pop() {remove v from S}
            while u ¬ = w
                e ← S.pop()       {next edge}
                 if  weight(e) > max   then
                        max ← weight(e)
                        maxE ← e
                 u ← S.pop()       {next vertex}
            setMSTLabel(maxE, NOT_IN_MST)  {remove max weight edge}
            cycleFound ←  true   {cycleFound is a subclass field, initially set to false}
```

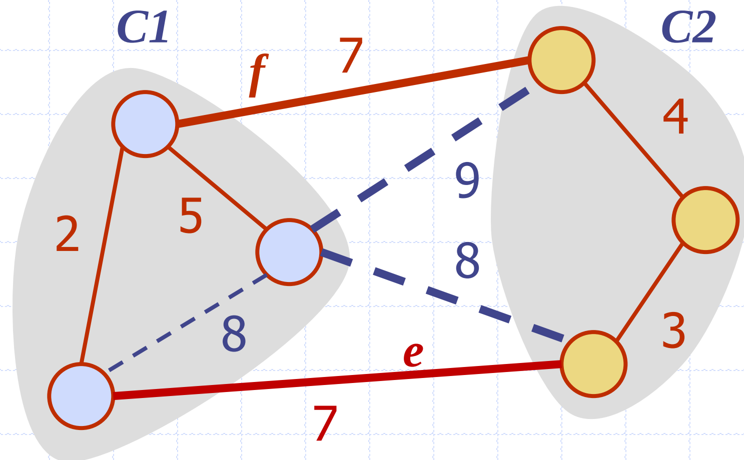
Another Possible Approach

Note that the problem occurs when:

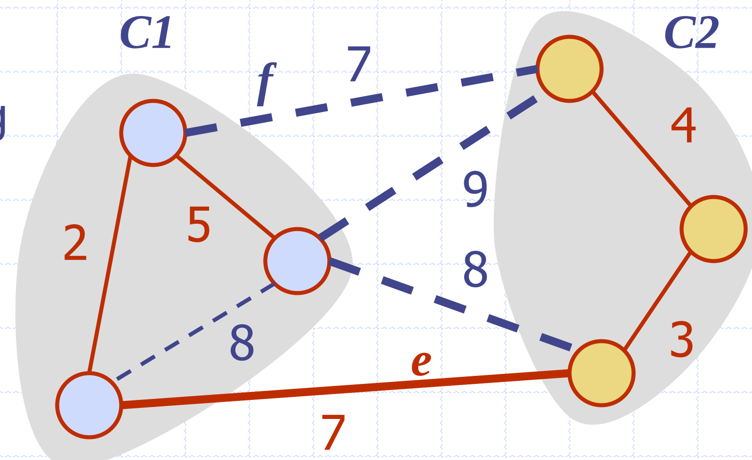
- There are two minimum, equal size edges connecting two components
- And each component chooses a different minimum edge
  - Say C1 chooses edge *f* and C2 chooses edge *e*
- Including *e* and *f* would create a cycle

Solution:

- Force both components C1 and C2 to choose the same edge connecting them
- Either edge *e* or *f* in the diagram



Choose either *f* or *e* for components C1 and C2



Baruvka's Algorithm (1926)

My Latest Version (version 6 at least, April & Aug. 2020)

Baruvka's Algorithm (another refinement, i.e., sorting edges)

Algorithm ***BaruvkaMST***(***G***)

  for each e $\in$ ***G.edges()*** **do**

    setMSTLabel(e, ***NOT_IN_MST****)* {no edges in MST}

  sortedEdges $\leftarrow$ MergeSort(G.edges()) {output is a list}

  components $\leftarrow$ *G.numVertices*()

  while *components* > 1 **do**

    components $\leftarrow$ labelVerticesOfEachComponent(G)

    if *components* > 1 **then**

      insertSmallestEdges(G, *components*, sortedEdges)

  return ***G***

Label Vertices of Each Component (subclass methods of BFS template)

Algorithm labelVerticesOfEachComponent(G)
    return BFS(G)   {label vertices of G with component numbers}

Algorithm ***initResult***(***G***)
    count ← 0   {initialize count to zero}

Algorithm ***preEdgeVisit***(***G, v, e, w***)
    if getMSTLabel(e) **=** NOT_IN_MST **then**   {is e in the MST}
        setLabel(***e, SKIP***)   {skips edges in not in MST}

Algorithm ***postComponentVisit***(***G, v***)
    count ← count + 1   {add one to count}

Algorithm ***preVertexVisit***(***G, v***)
    setComponentNum(***v,*** count)

Algorithm ***result***(***G***)
    return count   {return count which is the number of components}

Three refinements Apr., June, & Aug. 2020

```
Algorithm insertSmallestEdges
            (G, components, sortedEdges)
    minEdges ← new array of size components
    for i ← 0 to components - 1 do
        minEdges[i] ← ∅
    i ← 0
    p ← sortedEdges.first()
    while  i < components  do
       (c, isMax) ← insertEdge(G, minEdges,p)
       q ← p
       if  ! sortedEdges.isLast(p)  then
            p ← sortedEdges.after(p) {next edge}
 {Was edge added to MST or is it a max edge in
    a cycle? If yes, then no longer needed}
        if  c > 0 ∨ isMax then
             sortedEdges.remove(q)
        i ← i + c

{no need to look at edges after each component
    has added its smallest incident edge; loop
    terminates when smallest edge for each
    component has been added to MST}
```

```
Algorithm insertEdge(G, minEdges, p)
{Because edges are sorted, we don't have
    to be concerned with cycles because
    both components pick the same edge!!!}
     c ← 0
     e ← p.element()
     (v, w) ← G.endVertices(e)
     cv ← getComponentNum(v)
     cw ← getComponentNum(w)
 {does e connect two different components}
     if cv ≠ cw then
          {insert edge e if smallest}
        if minEdges[cv] = ∅  then
             minEdges[cv] ← e
             setMSTLabel(e, IN_MST)
             c ← c + 1
        if minEdges[cw] = ∅  then
             minEdges[cw] ← e
             setMSTLabel(e, IN_MST)
             c ← c + 1
    return (c, cv=cw)
       {cv=cw means e is max edge in a cycle}
```

Baruvka's Algorithm (another refinement, i.e., sorting edges)

```
Algorithm BaruvkaMST(G)
for each e ∈ G.edges() do
    setMSTLabel(e, NOT_IN_MST)  {no edges in MST}
sortedEdges ← MergeSort(G.edges()) {output is a list}
components ← G.numVertices()
while components > 1 do
    components ← labelVerticesOfEachComponent(G)
    if components > 1 then
        insertSmallestEdges(G, components, sortedEdges)
return G
```

Analysis of top level BaruvkaMST(G)

- Lines 1-2 runs in O(m) to initialize edges
- Line 3 runs in O(m log m) to sort edges i.e., O(m log n)
- Line 5 runs O(log n) times through the loop since number of components is divided at least in half each time through the loop
- Line 6 is O((m + n) log n) to run BFS log n times
- Line 8 is O(m log n) to search for smallest edges to insert into MST
- Thus this algorithm runs in

  O((m + n) log n + m log m)
- Why can we conclude that it's actual Big-O running time is O(m log n)?
  - That is, how did we eliminate the n log n and m log m terms?