Baruvka's Algorithm (1926)

Template Method Solution

Template Version of BFS

```
Algorithm BFS(G) {top level}
    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)
        initVertices(u)
    for all e ∈ G.edges() do
        setLabel(e, UNEXPLORED)
        initEdges(e)
    for all v ∈ G.vertices() do
        if getLabel(v) = UNEXPLORED
            preComponentVisit(G, v)
            BFS(G, v)
            postComponentVisit(G, v)

    result(G)
```

```
Algorithm BFS(G, s)
    startBFS(G, s)
    setLabel(s, VISITED)
    L.insertLast(s)
    while ¬L.isEmpty() do
        v ← L.remove(L.first())
        preVertexVisit(G, v)
        for all e ∈ G.incidentEdges(v) do
            preEdgeVisit(G, v, e)
            if getLabel(e) = UNEXPLORED
                w ← opposite(v, e)
                if getLabel(w) = UNEXPLORED
                    preDiscEdgeVisit(G, v, e, w)
                    setLabel(e, DISCOVERY)
                    setLabel(w, VISITED)
                    L.insertLast(w)
                    postDiscEdgeVisit(G, v, e, w)
                else
                    setLabel(e, CROSS)
                    crossEdgeVisit(G, v, e, w)
        postVertexVisit(G, v)

    finishBFS(G, s)
```


Baruvka's Algorithm (from Lecture 14)

```
Algorithm BaruvkaMST(G)
  for each e ∈G.edges() do
    setMSTLabel(e, NOT_IN_MST) {no edges in MST}
  numEdges ← 0   {numEdges is an instance variable}
  while numEdges < n-1 do
    labelVerticesOfEachComponent(G) {BFS}
    insertSmallest-WeightEdgeOutOfComponents(G)
  return G
```


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

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

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

Algorithm initEdges(e)
    if getMSTLabel(e) = NOT_IN_MST then   {is e in the MST}
        setLabel(e, VISITED)   {so BFS only visits edges in MST}

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

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

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


Insert Minimum-Weight Edges Going Out from each Component

What is the running time?

```
Algorithm insertSmallest-WeightEdgeOutOfComponents(G)
    minEdges ← new hash table based Dictionary
    BFS(G)   {search for smallest edges connecting different components}
    for all e in minEdges.elements() do
        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)  {called during BFS(G) above}
    cv ← getComponentNum(v)
    cw ← getComponentNum(G.opposite(v, e))
    if cv ≠ cw then   {does e connect two different components of MST}
        currentMinE ← minEdges.findElement(cv) {min edge for component cv}
        if currentMinE = No_Such_Key then
            minEdges.insertItem(cv, e)   {insert new minimum}
        else
            min ← weight(currentMinE)   {current min weight for component cv}
            if min > weight(e) then
                minEdges.removeElement(cv)   {remove old minimum}
                minEdges.insertItem(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?

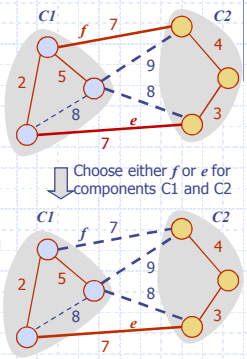
Another Possible Approach

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



Insert Minimum-Weight Edges then remove cycles

```
Algorithm insertSmallest-WeightEdgeOutOfComponents(G)
    minEdges ← new hash table based Dictionary
    BFS(G)    {search for smallest edges of G connecting different components}
    for all e in minEdges.elements() do
        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)    {called during BFS(G) above}
    cv ← getComponentNum(v)
    cw ← getComponentNum(G.opposite(v, e))
    if cv ≠ cw then    {does e connect two different components of MST}
        currentMinE ← minEdges.findElement(cv) {min edge for component cv}
        if currentMinE = No_Such_Key then
            minEdges.insertItem(cv, e)    {insert new minimum}
        else
            min ← weight(currentMinE)    {current min weight for component cv}
            if min > weight(e) then
                minEdges.removeElement(cv)    {remove old minimum}
                minEdges.insertItem(cv, e)    {insert new minimum}
```


DFS is better for Cycle Finding

```
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)
        initVertices(u)
    for all e ∈ G.edges() do
        setLabel(e, UNEXPLORED)
        initEdges(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)
                backEdgeTraversal(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 initEdges(e)
    if getMSTLabel(e) = NOT_IN_MST then setLabel(e, NOT_IN_MST)
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}
```


Insert Min-Weight Edges Going Out from Components (better)

```
Algorithm insertSmallest-WeightEdgeOutOfComponents(G)
    minEdges ← new hash table based Dictionary
    BFS(G)    {search for smallest edges of G connecting different components}

    for all e in minEdges.elements() do {remove edges that could cause a cycle}
        (v, w) ← G.endVertices(e)
        cv ← getComponentNum(v)
        cw ← getComponentNum(w)
        minEdgeCV ← minEdges.findElement(cv) {min edge for component cv}
        minEdgeCW ← minEdges.findElement(cw) {min edge for component cw}
        if minEdgeCV ≠ minEdgeCW then {if both chose different edges}
            if weight(minEdgeCV) = weight(minEdgeCW) then {if are same weight?}
                if minEdgeCV = e then {if e is min edge out of cv}
                    minEdges.removeElement(cw)    {remove old minimum for cw}
                    minEdges.insertItem(cw, e)    {cw and cv now both select e}
                else    {e is min edge out of cw}
                    minEdges.removeElement(cv)    {remove old minimum for cv}
                    minEdges.insertItem(cv, e)    {cw and cv now both select e}

    for all e in minEdges.elements() do {insert min edges into MST}
        if getMSTLabel(e) = NOT_IN_MST then    {if e is not already in MST}
            setMSTLabel(e, IN_MST)    {insert e into MST by setting label}
            numEdges ← numEdges + 1 {increase number of edges in MST}

Algorithm preEdgeVisit(G, v, e)  {same as above; done during BFS(G)}
```