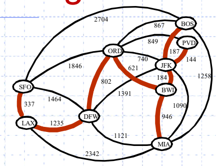
Lecture 17b: Minimum Spanning Trees

Infinite Correlation



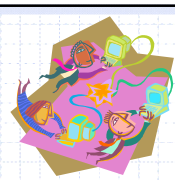
1

Wholeness Statement

A minimum spanning tree is a spanning tree subgraph with minimum total edge weight. Efficient greedy algorithms have been developed to compute MST both with and without special data structures. Pure creative intelligence is the source of all creative algorithms. Regular practice of TM improves our ability to make use of our own innate creative potential.

2

Outline and Reading



- Minimum Spanning Trees
  - Definitions
  - Cycle Property
  - Partition Property
- The Prim-Jarnik Algorithm (1957, 1930)
- Kruskal's Algorithm (1956)
- Baruvka's Algorithm (1926)

3

Minimum Spanning Tree

Spanning subgraph

- Subgraph of a graph G containing all the vertices of G

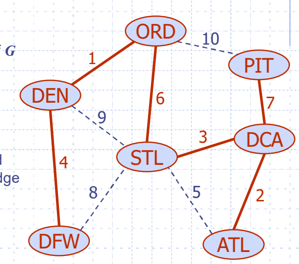
Spanning tree

- Spanning subgraph that is itself a (free) tree

Minimum spanning tree (MST)

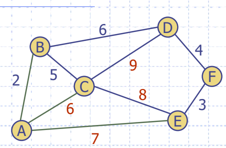
- Spanning tree of a weighted graph with minimum total edge weight

- Applications
  - Communications networks
  - Transportation networks



4

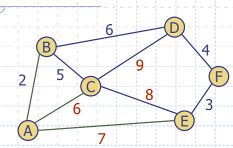
Cycle Property

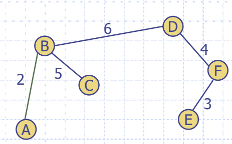


If the weight of an edge ***e*** of a cycle C is larger than the weights of other edges of C, then this edge cannot belong to a MST.

5

Cycle Property





If the weight of an edge ***e*** of a cycle C is larger than the weights of other edges of C, then this edge cannot belong to a MST.

6

Cycle Property

- **Cycle Property**: For any cycle C in a graph, if the weight of an edge ***e*** of C is larger than the weights of other edges of C, then this edge cannot belong to a MST.
- **Proof**: Assume the contrary, i.e. that e belongs to an MST ***T1***; then deleting ***e*** will break ***T1*** into two subtrees with the two ends of ***e*** in different subtrees. The remainder of C reconnects the subtrees, hence there is an edge ***f*** of C with ends in different subtrees, i.e., it reconnects the subtrees into a tree ***T2*** with weight less than that of ***T1***, because the weight of ***f*** is less than the weight of ***e***; thus ***T1*** cannot be a MST.

7

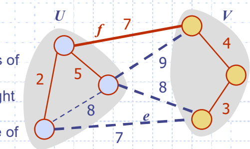
Partition Property- A Crucial Fact

Partition Property:

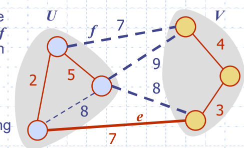
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:

- Let T be an MST of G
- If T does not contain e, consider the cycle C formed by e with T and let f be an edge of C across the partition
- By the cycle property, $weight(f) \le weight(e)$
- Thus, $weight(f) = weight(e)$
- We obtain another MST by replacing f with e



Replacing f with e yields another MST



8

Generic MST Algorithm

Algorithm ***GenericMST(G)***
    T ← a graph with all vertices in G, but no edges
    while ***T does not form a spanning tree*** **do**
        (*u, v*) ← a safe edge of ***G***
        T ← **(*u, v*)** U ***T***
    return ***T***

A *safe edge* is one that when added to T forms a subgraph of a MST

9

Main Point

1. A minimum spanning tree algorithm gradually grows a (sub-solution) tree by adding a "safe edge" that connects a vertex in the tree to a vertex not yet in the tree.
   Science Of Consciousness: The nature of life is to grow and progress to the state of enlightenment, fulfillment.

10

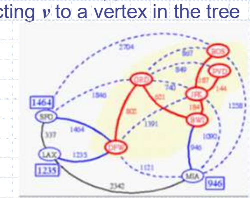
Prim(1957)-Jarnik(1930) Algorithm

AKA Dijkstra-Prim Algorithm

11

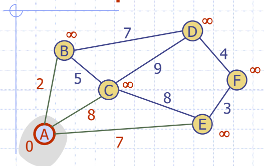
Prim-Jarnik's Algorithm

- Similar to Dijkstra's shortest path algorithm (for a connected graph)
- We pick an arbitrary vertex s and we grow the MST as a tree of vertices, starting from s
- We store with each vertex v a label $d(v)$ = the smallest weight of an edge connecting v to a vertex in the tree
- At each step:
  - We add to the tree, the vertex u outside the tree with the smallest distance label
  - We update the labels of the vertices adjacent to u



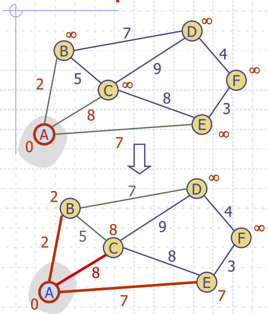
12

Example



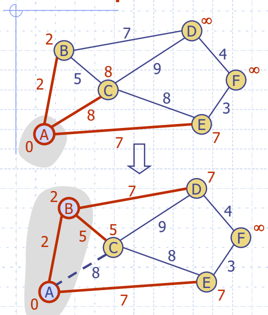
13

Example



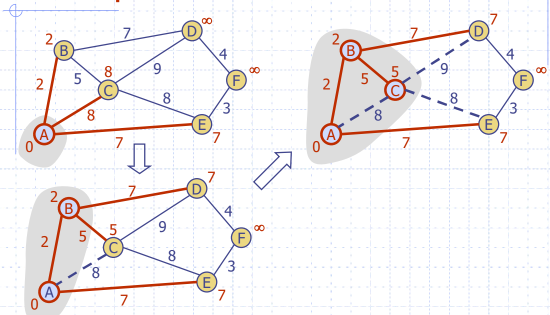
14

Example



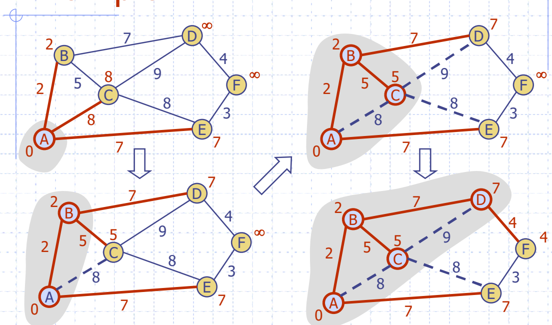
15

Example



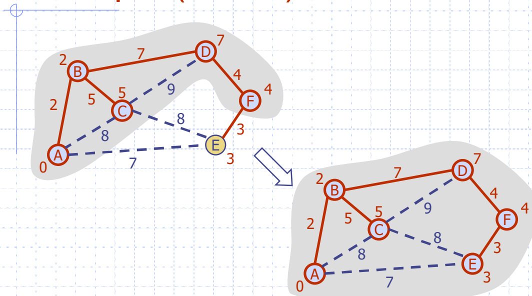
16

Example



17

Example (contd.)



18

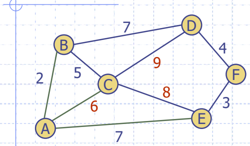
Prim-Jarnik's Algorithm (cont.)

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - *insert*(k, e) returns a locator
  - *replaceKey*(l, k) changes the key of an item
- We store three labels with each vertex:
  - Distance
  - Parent edge in MST
  - Locator in priority queue
- Correction in red inspired by Bereket Chalew (May 2014)

```
Algorithm PrimJarnikMST(G)
  PQ ← new heap (or priority queue)
  s ← G.aVertex ()
  for all  v ∈ G.vertices() do
    if  v = s  then
        setDistance(v, 0)
    else
        setDistance(v, ∞)
     setParent(v, ∅)
     p ← PQ.insertItem(getDistance(v), v)
     setLocator(v, p)
  while  ¬ PQ.isEmpty() do
     u ← PQ.remove()
     setLocator(u, ∅)  {u is now in MST}
     for all  e ∈ G.incidentEdges(u) do
        z ← G.opposite(u, e)
        r ← weight(e)
        p ← getLocator (z)
        if  p ≠ ∅   {z not yet in MST}
              ∧ r < getDistance(z) then
           setDistance(z, r)
           setParent(z, e)
           PQ.replaceKey(p, r)
```


19

Cycle Property



By the Cycle Property, AC, CD, and CE cannot be in a MST. What about AE and BD?

Let's run the algorithm to verify this.

20

Analysis

- Graph operations
  - Method incidentEdges is called once for each vertex
  - Recall that $\Sigma_v \deg(v) = 2m$
- Label operations
  - We set/get the distance, parent and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex w in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Prim-Jarnik's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
- The running time is $O(m \log n)$ since the graph is connected

21

Main Point

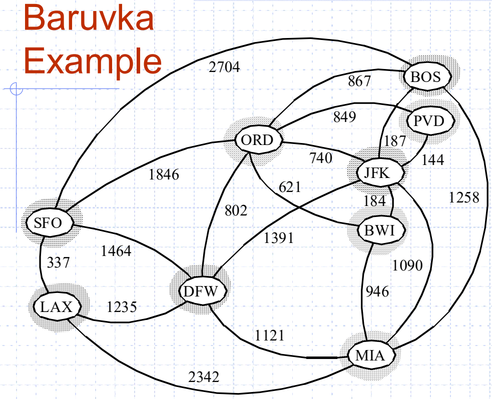
2. A defining feature of the Minimum Spanning Tree (and shortest path) greedy algorithms is that once a vertex becomes in-tree (or "inside the cloud"), the resulting subtree is optimal and nothing can change this state.
   Science of Consciousness: A defining feature of enlightenment is that once this state is reached, one's consciousness is optimal and nothing can change this state.

22

Baruvka's Algorithm (1926)

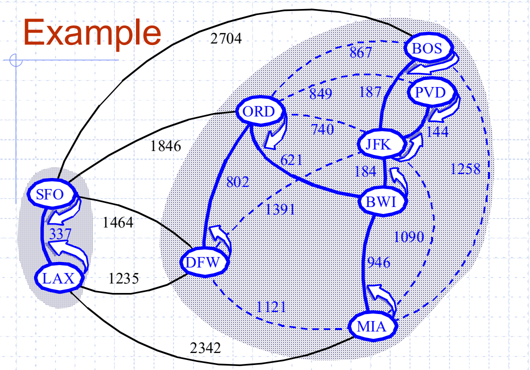
23

Baruvka Example



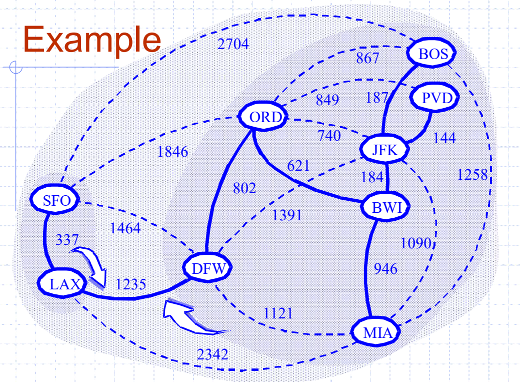
24

Example



25

Example



26

Baruvka's Algorithm

- Like Kruskal's Algorithm, Baruvka's algorithm grows many "clouds" at once.

Algorithm ***BaruvkaMST(G)***
  $T \leftarrow V$ {just the vertices of G, no edges, n connected components}
  while T has fewer than n-1 edges **do** {T is not yet an MST}
    for each connected component C in T **do**
      Find edge e with smallest-weight edge from C to
        another component in T.
      if e is not already in T **then**
        Add edge e to T
  return T

- Each iteration of the while-loop halves the number of connected components in T.

27

Baruvka's Algorithm (more details)

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

28

Required functionality

- Does not use a priority queue or heap!
- Does not use union-find data structures needed by Kruskal!
- Maintains a forest T subject to edge insertion
  - Can be supported in O(1) time using labels on edges in MST
- Step 1: Mark vertices with number of the component to which they belong
  - Traverse forest T to identify connected components
    - O(1) time to label each vertex
    - Requires extra instance variable for each vertex
  - Takes O(n) time using a DFS or a BFS each time through the while-loop to label vertices of each component
- Step 2: Find a smallest-weight edge in E incident on each cluster/component C (insert into MST)
  - Scan adjacency lists of each vertex in each C to find minimum
  - Takes O(m) each time through the for-loop

29

Analysis of Baruvka's Algorithm

- While-loop: each iteration (at worst) halves the number of connected components in T
  - Thus is executed log n times
- Identifying connected components (in for-loop)
  - Vertices are labelled with component name
  - DFS or BFS of T runs in O(n) time
- Find smallest edge incident on each component C
  - Scan adjacency lists of vertices in G
  - O(m) time
- The running time is O(m log n).

30

After labeling each vertex with component number (exercise)

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

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

    result( )
```

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

HW: Insert into T the smallest-weight edge going out from each component

Lower Bound on MST Computation

- There are **randomized algorithms** that compute MST's in **expected linear time**
- Linear time seems to be the lower bound
- Unknown whether there is a deterministic algorithm that runs in linear time (open question)

Main Point

3. The "greedy" algorithms used by MST and shortest path only work for problems where localized attention can produce a globally optimal solution.
   Science of Consciousness: An enlightened person maintains unbounded awareness along with localized awareness. The behavior of such a person is globally optimal for any problem.

Running time

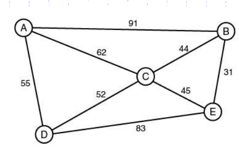
- In the generic MST algorithm, the running time can be computed by observing that a (potentially) exhaustive search of edges (m) is made in each iteration (n-1), leading to a running time of O(mn).
- (Prim-Jarnik): This running time is not optimal unless, as with Dijkstra's shortest path algorithm, the exhaustive search of edges can be replaced by a fast replaceKey operation on a priority queue. The running time using this approach is O(m log n).
- The beauty of Baruvka is that it runs in O(m log n) time without any special data structures and was the earliest MST algorithm (1926)

Intractable Problems

- In this course, we have seen Big Oh values ranging from O(1), O(n), O(nlog n), O(n^2), up to O(n^3).
- Algorithms with these big Oh values can be used to find solutions to most practical problems.
- However, some algorithms have big Oh values that are so large that they can be used only for relatively small values of N. For example O(2^n), O(n!), etc. are impractical. Problems that have these running times are said to be ***intractable***.
- We will see two Intractable Problems next.

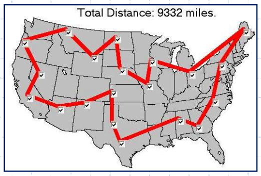
Hamiltonian Cycles

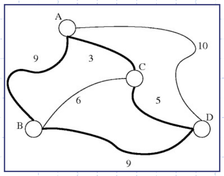
- A Hamiltonian cycle in a graph G is a simple cycle that contains every vertex of G. A graph is a Hamiltonian graph if it contains a Hamiltonian cycle.
- Exercise:



Another famous Example: Traveling Salesman Problem

- *Traveling Salesman Problem* (TSP): Given a complete graph G with cost function c: E → N and a positive integer k, is there a Hamiltonian cycle C in G so that the sum of the costs of the edges in C is at most k? Note that a solution is always a subset of E.





Handling Intractable Problems

- Much of the research in the field of algorithms is dedicated to finding ways to handle intractable problems
- Some approaches that are currently used include
  - Approximation algorithms – devise faster algorithms to solve these problems with outputs that are not optimal but in some cases may be good enough
  - Probabilistic algorithms. Probabilistic algorithms can often execute very efficiently and output results that are correct with high probability.
  - Using intractable problems as an advantage. Modern day cryptosystems use intractable problems to prevent hackers from accessing protected resources.

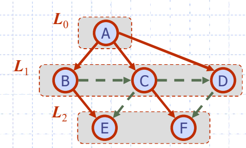
Exercise

Templates

BFS Levels

When actually implemented, the levels are normally merged into a single sequence/queue

How could we keep track of the level of a vertex?



BFS

BFS Algorithm (revised)

- The BFS algorithm using a single sequence L

```
Algorithm BFS(G) {top level}
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      BFS(G, v)
```

```
Algorithm BFS(G, s)
  L ← new empty sequence
  L.insertLast(s)
  setLabel(s, VISITED)
  while ¬L.isEmpty()
    v ← L.removeAtRank(0)
    for all e ∈ G.incidentEdges(v)
      if getLabel(e) = UNEXPLORED
        w ← opposite(v,e)
        if getLabel(w) = UNEXPLORED
          setLabel(e, DISCOVERY)
          setLabel(w, VISITED)
          L.insertLast(w)
        else
          setLabel(e, CROSS)
```


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)
    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)
      BFS(G, v)
      postComponentVisit(G, v)

  return result(G)

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

```
Algorithm BFS(G, s)
    setLabel(s, VISITED)
    L.insertLast(s)
    startBFS(G, s)
    while ¬L.isEmpty() do
      v ← L.removeAtRank(0)
      preVertexVisit(G, v)
      for all e ∈ G.incidentEdges(v) do
        if getLabel(e) = UNEXPLORED
          w ← opposite(v, e)
          preEdgeVisit(G, v, e, w)
          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)
```

BFS Algorithm (revised)

- **The BFS algorithm if we need to know the level**

```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      BFS(G, v)\
```

```
Algorithm BFS(G, s)
  L ← new empty sequence
  L.insertLast(s)
  setLevel(s, 0)  {use only if level # is needed}
  setLabel(s, VISITED)
  while ¬L.isEmpty()
      v ← L.remove (L.first())
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            L.insertLast(w)
            setLevel(w, getLevel(v)+1)
          else
            setLabel(e, CROSS)
```

Shows how we could keep track of the level?

43

43

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Finding the minimum spanning tree can be done by an exhaustive search of all possible spanning trees, then choosing the one with minimum weight.

2. To devise a greedy strategy, we identify a set of candidate choices, determine a selection procedure, and consider whether there is a feasibility problem. Then we have to prove that the strategy works.

Minimum Spanning Trees 44

44

3. **Transcendental Consciousness** is the home of all the laws of nature, the source of all algorithms.
4. **Impulses within Transcendental Consciousness**: The natural laws within this unbounded field are the algorithms of nature governing all the activities of the universe.
5. **Wholeness moving within itself:** In Unity Consciousness, we perceive the spanning tree of natural law and appreciate the unity of all creation.

Minimum Spanning Trees 45

45