CSE 5311 Notes 8: Minimum Spanning Trees

(Last updated 2/18/17 4:10 PM)

CLRS, Chapter 23

CONCEPTS

Given a weighted, connected, undirected graph, find a minimum (total) weight free tree connecting the vertices. (AKA bottleneck shortest path tree)

Cut Property: Suppose S and T partition V such that

1. $S \cap T = \emptyset$
2. $S \cup T = V$
3. $|S| > 0$ and $|T| > 0$

then there is some MST that includes a minimum weight edge {s, t} with $s \in S$ and $t \in T$.

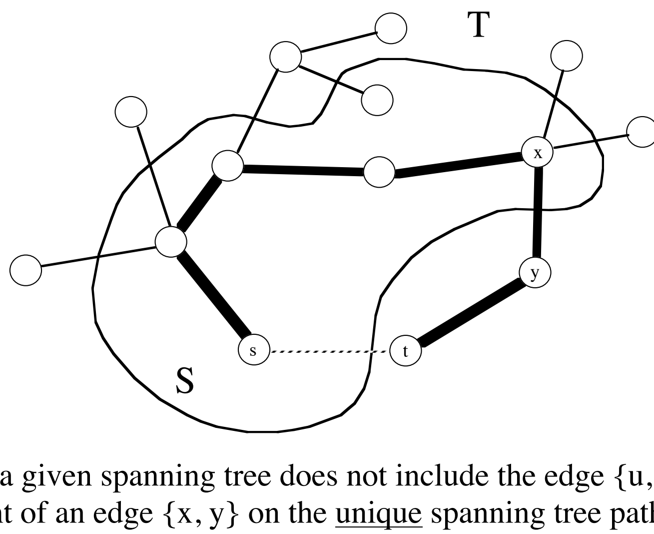
Proof:

Suppose there is a partition with a minimum weight edge {s, t}.

A spanning tree without {s, t} must still have a path between s and t.

Since $s \in S$ and $t \in T$, there must be at least one edge {x, y} on this path with $x \in S$ and $y \in T$.

By removing {x, y} and including {s, t}, a spanning tree whose total weight is no larger is obtained. •••



Cycle Property: Suppose a given spanning tree does not include the edge {u, v}. If the weight of {u, v} is no larger than the weight of an edge {x, y} on the unique spanning tree path between u and v, then replacing {x, y} with {u, v} yields a spanning tree whose weight does not exceed that of the original spanning tree.

Proof: Including {u, v} into the spanning tree introduces a cycle, but removing {x, y} will remove the cycle to yield a modified tree whose weight is no larger.

Does not directly suggest an algorithm, but all algorithms avoid including an edge that violates.

Prove or give counterexample:

The MST path beween two vertices is a shortest path.

True or False?

Choosing the |V| -1 *edges with smallest weights gives a MST.*

Fill in the blank:

Multiple MSTs occur only if ____________________.

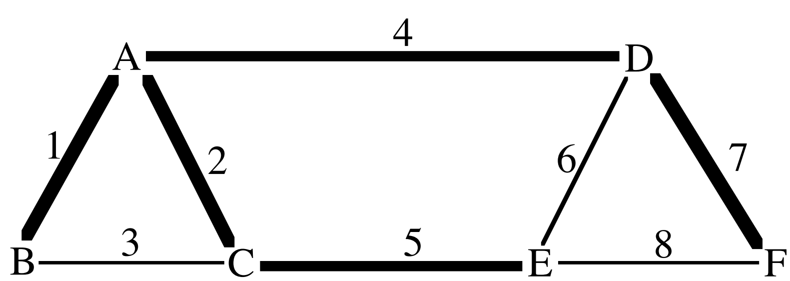
MODIFIED REACHABILITY CONDITION

Towards an algorithm:

1. Assume unique edge weights (easily forced by lexicographically breaking ties).
2. Consider all (cycle-free) paths between some pair of vertices.
3. Consider the maximum weight edge on each path.
4. The edge that is the minimum of the "maximums" *must* be included in the MST.

Any edge that is never a "must" is not in the MST.

Example:



Original Adjacency Matrix

	A	B	C	D	E	F
A	∞	1	2	4	∞	∞
B	1	∞	3	∞	∞	∞
C	2	3	∞	∞	5	∞
D	4	∞	∞	∞	6	7
E	∞	∞	5	6	∞	8
F	∞	∞	∞	7	8	∞

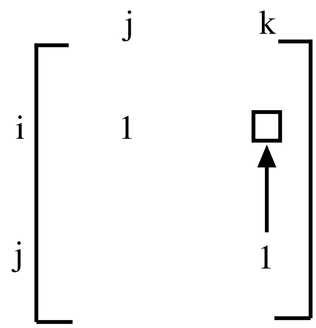
Unchanged Entries ($\boxed{}$) are MST

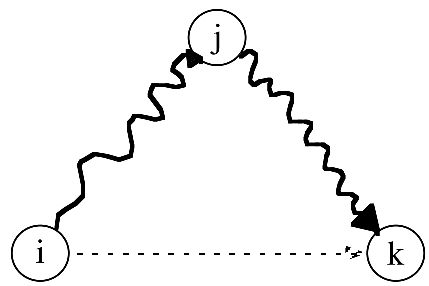
	A	B	C	D	E	F
A	1	$\boxed{1}$	$\boxed{2}$	$\boxed{4}$	5	7
B	$\boxed{1}$	1	2	4	5	7
C	$\boxed{2}$	2	2	4	$\boxed{5}$	7
D	$\boxed{4}$	4	4	4	5	$\boxed{7}$
E	5	5	$\boxed{5}$	5	5	7
F	7	7	7	$\boxed{7}$	7	7

Implementation - Based on Warshall's Algorithm

1. Directed reachability - existence of path:

```
for (j=0; j<V; j++)
  for (i=0; i<V; i++)
    if (A[i][j])
      for (k=0; k<V; k++)
        if (A[j][k])
          A[i][k]=1;
```

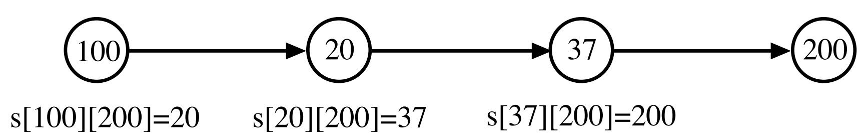




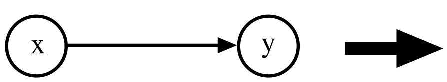
Correctness proof is by math. induction. (CSE 2320 Notes 16.C)

Successor Matrix (CLRS uses predecessor)

7-11 directions:



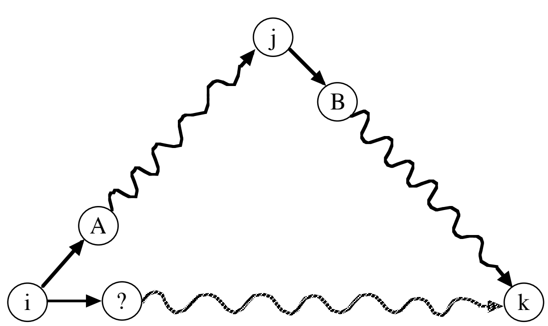
Initialize:



s[x][y]=y

(-1 otherwise)

Warshall Matrix Update:



s[i][j] = A   s[j][k] = B   s[i][k] = ?

```
for (j=0; j<V; j++)
  for (i=0; i<V; i++)
    if (s[i][j] != (-1))
      for (k=0; k<V; k++)
        if (s[i][k] == (-1) && s[j][k] != (-1))
          s[i][k] = s[i][j];
```

2. All-pairs shortest paths - Floyd-Warshall

```
for (j=0; j<V; j++)
  for (i=0; i<V; i++)
    if (dist[i][j] < 999)
      for (k=0; k<V; k++)
      {
        newDist = dist[i][j] + dist[j][k];
        if (newDist < dist[i][k])
        {
          dist[i][k] = newDist;
          s[i][k] = s[i][j];
        }
      }
```

3. Minimum spanning tree: http://ranger.uta.edu/~weems/NOTES5311/MSTWarshall.c

```
// MST based on Warshall's algorithm (Maggs & Plotkin,
//    Information Processing Letters 26, 25 Jan 1988, 291-293)
// 3/6/03 BPW

// Modified 7/15/04 to make more robust, especially edges with same weight

#include <stdio.h>

#define maxSize (20)

struct edge {
  int weight,smallLabel,largeLabel;
};
typedef struct edge edgeType;

edgeType min(edgeType x,edgeType y)
{
// Returns smaller-weighted edge, using lexicographic tie-breaker
if (x.weight<y.weight)
  return x;
if (x.weight>y.weight)
  return y;

if (x.smallLabel<y.smallLabel)
  return x;
if (x.smallLabel>y.smallLabel)
  return y;

if (x.largeLabel<y.largeLabel)
  return x;
return y;
}
```

```
edgeType max(edgeType x,edgeType y)
{
// Returns larger-weighted edge, using lexicographic tie-breaker
if (x.weight>y.weight)
  return x;
if (x.weight<y.weight)
  return y;

if (x.smallLabel>y.smallLabel)
  return x;
if (x.smallLabel<y.smallLabel)
  return y;

if (x.largeLabel>y.largeLabel)
  return x;
return y;
}

main()
{
int numVertices,numEdges, i, j, k;
edgeType matrix[maxSize][maxSize];
int count;

printf("enter # of vertices and edges: ");
fflush(stdout);
scanf("%d %d",&numVertices,&numEdges);
printf("enter undirected edges u v weight\n");
for (i=0;i<numVertices;i++)
  for (j=0;j<numVertices;j++)
  {
    matrix[i][j].weight=999;
    if (i<=j)
    {
      matrix[i][j].smallLabel=i;
      matrix[i][j].largeLabel=j;
    }
    else
    {
      matrix[i][j].smallLabel=j;
      matrix[i][j].largeLabel=i;
    }
  }
for (k=0;k<numEdges;k++)
{
  scanf("%d %d",&i,&j);
  scanf("%d",&matrix[i][j].weight);
  matrix[j][i].weight=matrix[i][j].weight;
}

printf("input matrix\n");
for (i=0;i<numVertices;i++)
{
  for (j=0;j<numVertices;j++)
    printf("%3d ",matrix[i][j].weight);
  printf("\n");
}
```

```
// MST by Warshall
for (k=0;k<numVertices;k++)
  for (i=0;i<numVertices;i++)
    for (j=0;j<numVertices;j++)
      matrix[i][j]=min(matrix[i][j],max(matrix[i][k],matrix[k][j]));
```

```
printf("output matrix\n");
for (i=0;i<numVertices;i++)
{
  for (j=0;j<numVertices;j++)
    printf("%3d(%3d,%3d) ",matrix[i][j].weight,matrix[i][j].smallLabel,
      matrix[i][j].largeLabel);
  printf("\n");
}

count=0;
for (i=0;i<numVertices;i++)
  for (j=i+1;j<numVertices;j++)
    if (matrix[i][j].weight<999 && i==matrix[i][j].smallLabel &&
        j==matrix[i][j].largeLabel)
    {
      count++;
      printf("%d %d %d\n",i,j,matrix[i][j].weight);
    }

if (count<numVertices-1)
  printf("Result is a spanning forest\n");
else if (count>=numVertices)
  printf("Error? . . . \n");
}
```

PRIM'S ALGORITHM

Outline:

1. Vertex set S: tree that grows to MST.

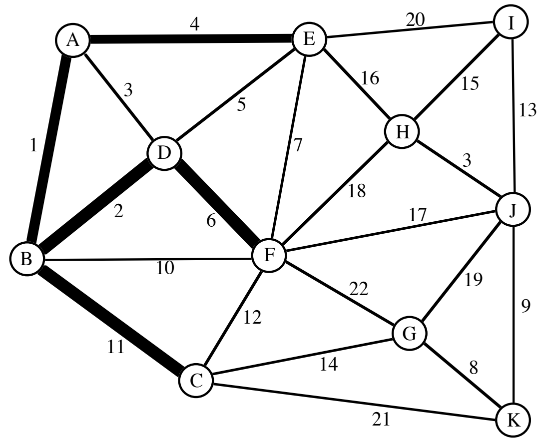
2. T = V - S: vertices not yet in tree.

3. Initialize S with arbitrary vertex.

4. Each step moves one vertex from T to S: the one with the minimum weight edge to an S vertex.

Different data structures lead to various performance characteristics.

Which edge does Prim's algorithm select next?

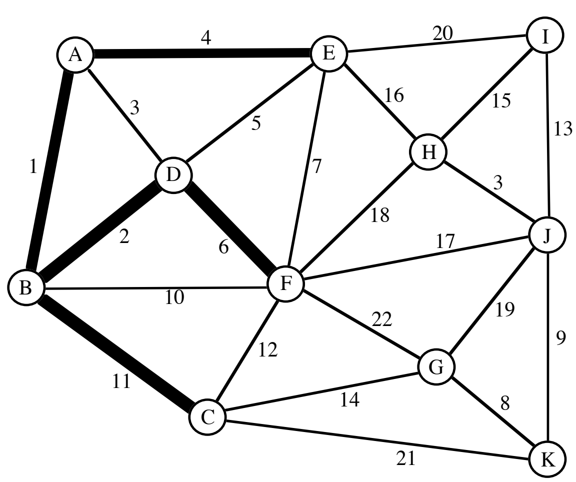


1. Maintains T-table that provides the closest vertex in S for each vertex in T.

Scans the list of the last vertex moved from T to S.

Place any vertex $x \in V$ in S.
$T = V - \{x\}$
for each $t \in T$
    Initialize T-table entry with weight of $\{t, x\}$ (or ∞ if non-existent) and x as best-S-neighbor.
while $T \neq \emptyset$
    Scan T-table entries for the minimum weight edge {t, best-S-neighbor[t]}
        over all $t \in T$ and all $s \in S$.
    Include edge {t, best-S-neighbor[t]} in MST.
    $T = T - \{t\}$
    $S = S \cup \{t\}$
    for each vertex x in adjacency list of t
        if $x \in T$ and weight of {x, t} is smaller than T-weight[x]
            T-weight[x] = weight of {x, t}
            best-S-neighbor[x] = t

What are the T-table contents before and after the next MST vertex is selected?



Analysis:

Initializing the T-table takes $\Theta(V)$.
Scans of T-table entries contribute $\Theta(V^2)$.
Traversals of adjacency lists contribute $\Theta(E)$.
$\Theta(V^2 + E)$ overall worst-case.

2. Replace T-table by a heap.

The time for updating for best-S-neighbor increases, but the time for selection of the next vertex to move from T to S improves.

```
Place any vertex x ∈ V in S.
T = V – {x}
for each t ∈ T
    Load T-heap entry with weight (as the priority) of {t, x} (or ∞ if non-existent) and x as
        best-S-neighbor
BUILD-MIN-HEAP(T-heap)
while T ≠ ∅
    Use HEAP-EXTRACT-MIN to obtain T-heap entry with the minimum weight edge over all t ∈ T
        and all s ∈ S.
    Include edge {t, best-S-neighbor[t]} in MST.
    T = T – {t}
    S = S ∪ {t}
    for each vertex x in adjacency list of t
        if x ∈ T and weight of {x, t} is smaller than T-weight[x]
            T-weight[x] = weight of {x, t}
            best-S-neighbor[x] = t
            MIN-HEAP-DECREASE-KEY(T-heap)
```

Analysis for binary heap:

Initializing the T-heap takes $\Theta(V)$.
Total cost for HEAP-EXTRACT-MINS is $\Theta(V \log V)$.
Traversals of adjacency lists and MIN-HEAP-DECREASE-KEYS contribute $\Theta(E \log V)$.
$\Theta(E \log V)$ overall worst-case, since $E > V$.

Analysis (amortized) for Fibonacci heap:

Initializing the T-heap takes $\Theta(V)$.
Total cost for HEAP-EXTRACT-MINS is $\Theta(V \log V)$.
Traversals of adjacency lists and MIN-HEAP-DECREASE-KEYS contribute $\Theta(E)$.
$\Theta(E + V \log V)$ overall worst-case, since $E > V$.

Which version is the fastest?

		Sparse $\left(E = O(V)\right)$	Dense $\left(E = \Omega\left(V^2\right)\right)$
table	$\Theta\left(V^2 + E\right)$	$\Theta\left(V^2\right)$	$\Theta\left(V^2\right)$
binary heap	$\Theta(E \log V)$	$\Theta(V \log V)$	$\Theta\left(V^2 \log V\right)$
Fibonacci heap	$\Theta(E + V \log V)$	$\Theta(V \log V)$	$\Theta\left(V^2\right)$

Analysis also applies to Dijkstra's shortest path.

KRUSKAL'S ALGORITHM

(Discussed in Notes 7 as an application of UNION-FIND trees.)

```
http://ranger.uta.edu/~weems/NOTES5311/kruskal.c
. . .

main()
{
. . .

qsort(edgeTab,numEdges,sizeof(edgeType),weightAscending);
for (i=0;i<numEdges;i++)
{
  root1=find(edgeTab[i].tail);
  root2=find(edgeTab[i].head);
  if (root1==root2)
    printf("%d %d %d discarded\n",edgeTab[i].tail,edgeTab[i].head,
      edgeTab[i].weight);
  else
  {
    printf("%d %d %d included\n",edgeTab[i].tail,edgeTab[i].head,
      edgeTab[i].weight);
    MSTweight+=edgeTab[i].weight;
    makeEquivalent(root1,root2);
  }
}
if (numTrees!=1)
  printf("MST does not exist\n");
printf("Sum of weights of spanning edges %d\n",MSTweight);
}
```

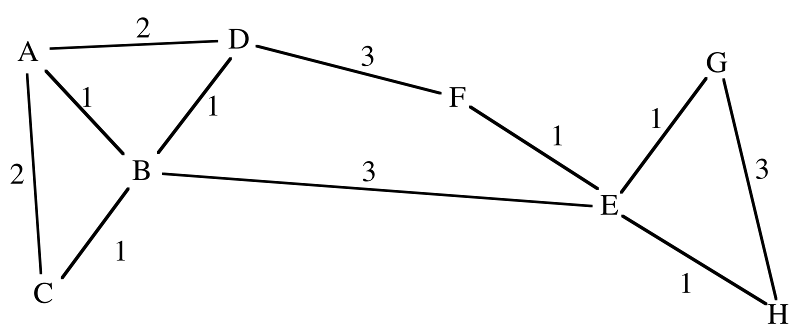
Incremental sorting (e.g. QUICKSORT or HEAPSORT) may be used.

Can adapt to determine if MST is unique:

```
http://ranger.uta.edu/~weems/NOTES5311/kruskalDup.c
. . .

main()
{

. . .
numTrees=numVertices;
qsort(edgeTab,numEdges,sizeof(edgeType),weightAscending);
i=0;
while (i<numEdges)
{
  for (k=i;
       k<numEdges && edgeTab[k].weight==edgeTab[i].weight;
       k++)
    ;
  for (j=i;j<k;j++)
  {
    root1=find(edgeTab[j].tail);
    root2=find(edgeTab[j].head);
    if (root1==root2)
    {
      printf("%d %d %d discarded\n",edgeTab[j].tail,edgeTab[j].head,
        edgeTab[j].weight);
      edgeTab[j].weight=(-1);
    }
  }
  for (j=i;j<k;j++)
    if (edgeTab[j].weight!=(-1))
    {
      root1=find(edgeTab[j].tail);
      root2=find(edgeTab[j].head);
      if (root1==root2)
        printf("%d %d %d alternate\n",edgeTab[j].tail,edgeTab[j].head,
          edgeTab[j].weight);
      else
      {
        printf("%d %d %d included\n",edgeTab[j].tail,edgeTab[j].head,
          edgeTab[j].weight);
        makeEquivalent(root1,root2);
      }
    }
  i=k;
}
if (numTrees!=1)
  printf("MST does not exist\n");
}
```



BORUVKA'S ALGORITHM

Similar to Kruskal:

1. Initially, each vertex is a component.

2. Each component has a "best edge" to some other component.

3. Boruvka step:

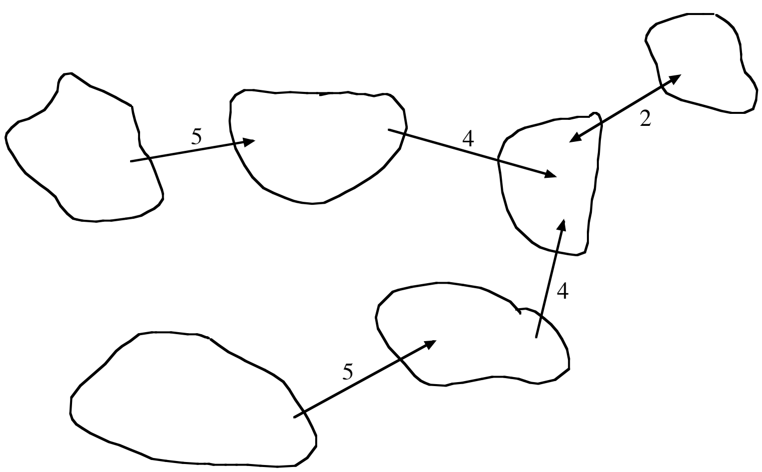
    For each best edge from a component x to a component y:

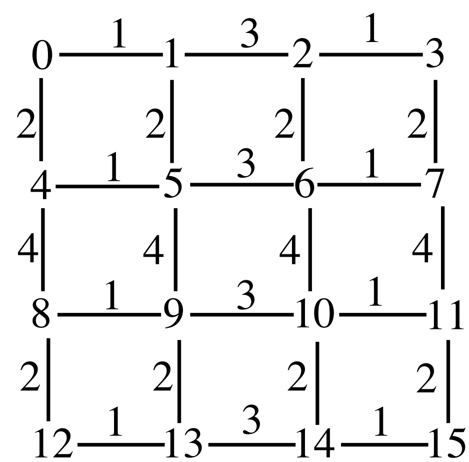
        a = FIND(x)
        b = FIND(y)
        if a ≠ b
            UNION(a,b)

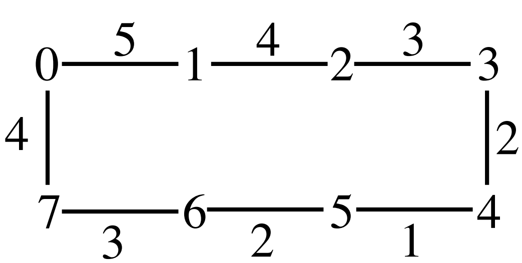
Worst-case: number of components decreases by at least half in each phase.

Gives O(E log V) time.

In some cases a cluster of several components may collapse:







http://ranger.uta.edu/~weems/NOTES5311/boruvka.c

```
. . .
main()
{
. . .
numTrees=numVertices;  // Each vertex is initially in its own subtree
usefulEdges=numEdges;  // An edge is useful if the two vertices are separate
while (numTrees>1 && usefulEdges>0)
{
  for (i=0;i<numVertices;i++)
    bestEdgeNum[i]=(-1);
  usefulEdges=0;
  for (i=0;i<numEdges;i++)
  {
    root1=find(edgeTab[i].tail);
    root2=find(edgeTab[i].head);
    if (root1==root2)
      printf("%d %d %d useless\n",edgeTab[i].tail,edgeTab[i].head,
        edgeTab[i].weight);
    else
    {
      usefulEdges++;
      if (bestEdgeNum[root1]==(-1)
      || edgeTab[bestEdgeNum[root1]].weight>edgeTab[i].weight)
        bestEdgeNum[root1]=i;  // Have a new best edge from this component

      if (bestEdgeNum[root2]==(-1)
      || edgeTab[bestEdgeNum[root2]].weight>edgeTab[i].weight)
        bestEdgeNum[root2]=i;  // Have a new best edge from this component
    }
  }
  for (i=0;i<numVertices;i++)
    if (bestEdgeNum[i]!=(-1))
    {
      root1=find(edgeTab[bestEdgeNum[i]].tail);
      root2=find(edgeTab[bestEdgeNum[i]].head);
      if (root1==root2)
        continue;  // This round has already connected these components.
      MSTweight+=edgeTab[bestEdgeNum[i]].weight;
      printf("%d %d %d included in MST\n",
        edgeTab[bestEdgeNum[i]].tail,edgeTab[bestEdgeNum[i]].head,
        edgeTab[bestEdgeNum[i]].weight);
      makeEquivalent(root1,root2);
    }
  printf("numTrees is %d\n",numTrees);
}
if (numTrees!=1)
  printf("MST does not exist\n");
printf("Sum of weights of spanning edges %d\n",MSTweight);
}
```

ASIDE - COUNTING SPANNING TREES

https://en.wikipedia.org/wiki/Kirchhoff's_theorem