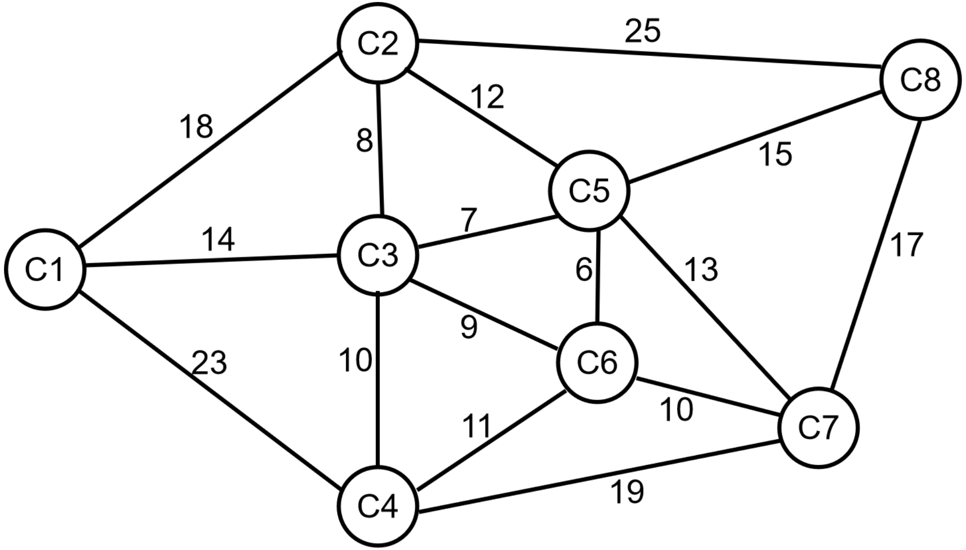
Graph

- **Graph - shortest paths**
  - directed graph G=(V,E)
  - *Dijkstra* algorithm - single-pair shortest path
  - *Dijkstra* algorithm **- all-pairs shortest paths**

	C1	C2	C3	C4	C5	C6	C7	C8
C1	0	18	14	23	21	23	33	36
C2	18	0	8	18	12	17	25	25
C3	14	8	0	10	7	9	19	22
C4	23	18	10	0	17	11	19	32
C5	21	12	7	17	0	6	13	15
C6	23	17	9	11	6	0	10	21
C7	33	25	19	19	13	10	0	17
C8	36	25	22	32	15	21	17	0

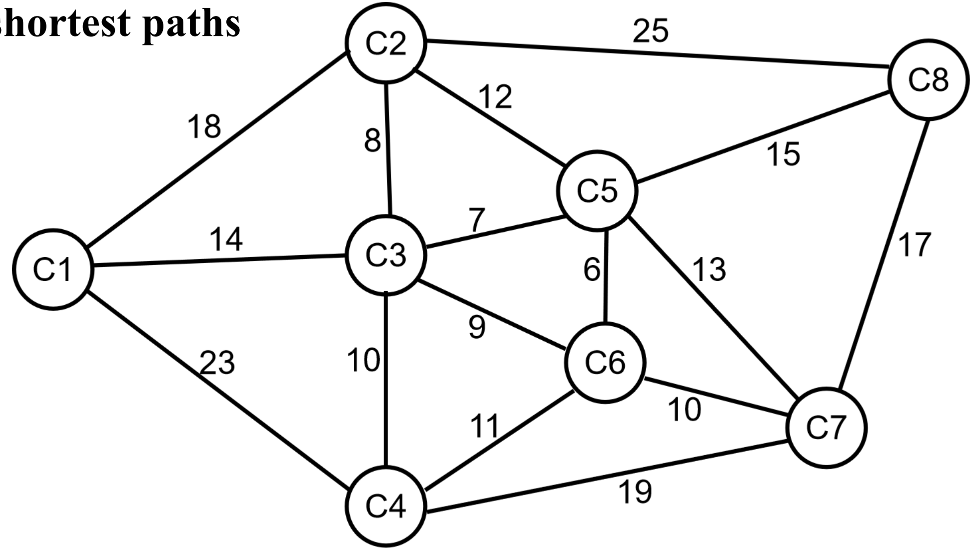
solution directly based on the *Dijkstra* algorithm
FOR i IN {C1, ..., C8}
    Dijkstra(i) until all other vertices are visited



	C1	C2	C3	C4	C5	C6	C7	C8
C1	0	18	14	23	21	23	33	36
C2	18	0	8	18	12	17	25	25
C3	14	8	0	10	7	9	19	22
C4	23	18	10	0	17	11	19	32
C5	21	12	7	17	0	6	13	15
C6	23	17	9	11	6	0	10	21
C7	33	25	19	19	13	10	0	17
C8	36	25	22	32	15	21	17	0

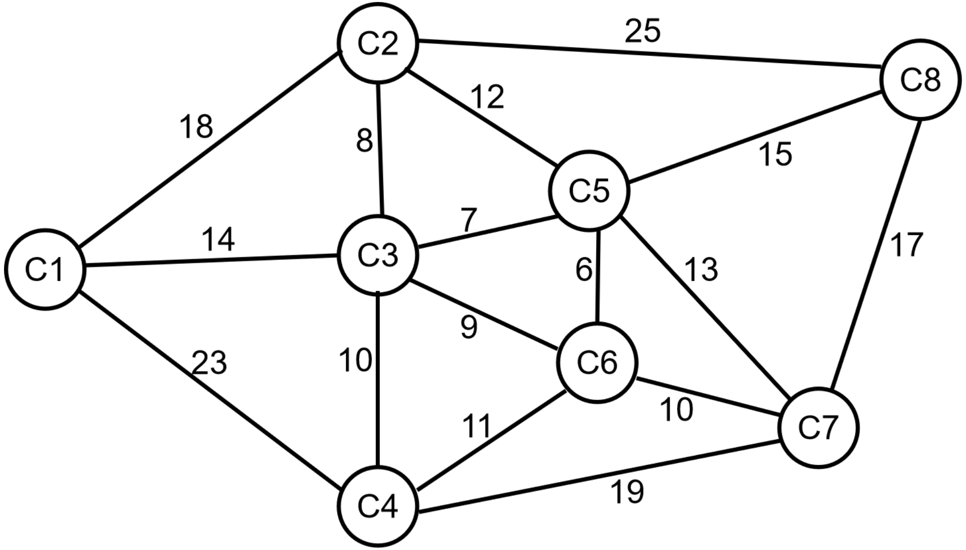
Graph

- **Graph - shortest paths**
  - directed graph G=(V,E)
  - *Dijkstra* algorithm - single-pair shortest path
  - *Dijkstra* algorithm - all-pairs shortest paths
  - *Floyd* algorithm - **all-pairs shortest paths**
  - *dynamic programming*



Graph

- **Graph - Floyd algorithm** - all-pairs shortest paths
  - k-path : intermediate vertices (except the two ends) all have indices less than k
  - best (k+1)-path from s to t is either case 1) or case 2)
    - 1) the best k-path from s to k followed by the best k-path from k to t
    - 2) the best k-path from s to t

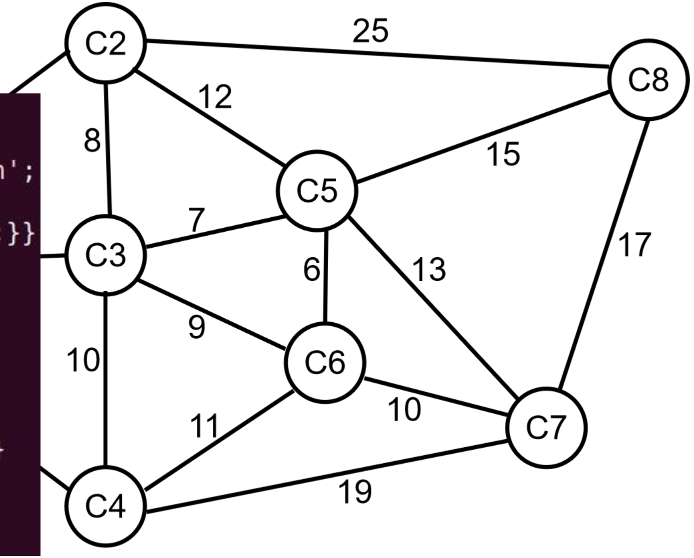


Floyd algorithm
INIT: Set the 0-path (direct edges among vertices)
LOOP: k FROM 1 TO n
    Compute the k-path according to the (k-1)-path

Graph

- **Graph - Floyd algorithm** - all-pairs shortest paths
  - k-path : intermediate vertices (except the two ends) all have indices less than k
  - best (k+1)-path from s to t is either case 1) or case 2)
    - 1) the best k-path from s to k followed by the best k-path from k to t
    - 2) the best k-path from s to t

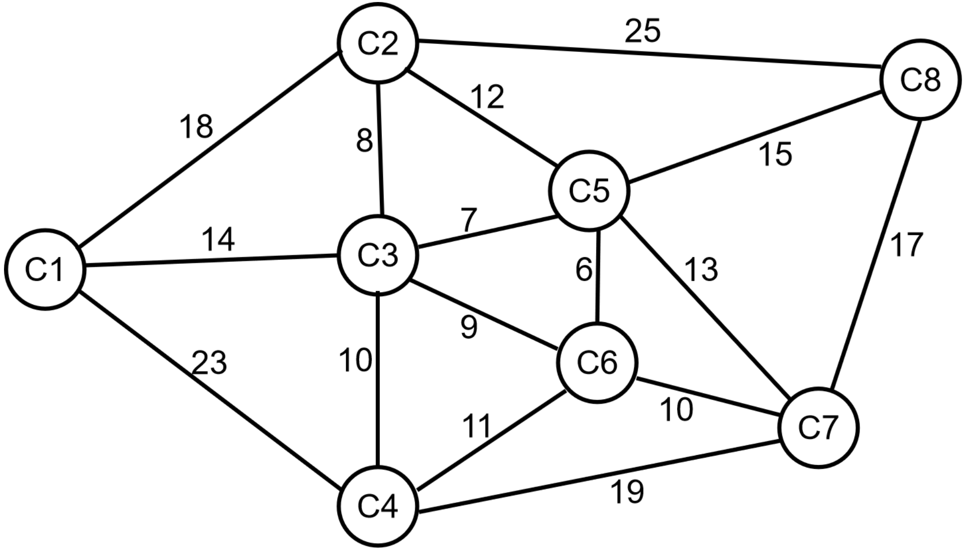
```
// Floyd algorithm - all-pairs shortest paths
void showFloyd(int *d[],int n){
        std::cout<<'\t';for(int j=0;j<n;j++) std::cout<<j<<'\t'; std::cout<<'\n';
        for(int i=0;i<n;i++){std::cout<<i<<'\t';
                for(int j=0;j<n;j++) std::cout<<d[i][j]<<'\t'; std::cout<<'\n';}}
void Floyd(LGraph* g,int *d[]){int n=g->num(),i,j,k;
        for(i=0;i<n;i++) for(j=0;j<n;j++)
                if(g->wgt(i,j)<0) d[i][j]=D_INF; else d[i][j]=g->wgt(i,j);
        for(i=0;i<n;i++) d[i][i]=0;
        for(k=0;k<n;k++){std::cout<<k<<"-path => \n";showFloyd(d,n);
                for(i=0;i<n;i++) for(j=0;j<n;j++)
                        if(d[i][j]>(d[i][k]+d[k][j])) d[i][j]=d[i][k]+d[k][j];}
        std::cout<<k<<"-path => \n";showFloyd(d,n);
}
// END Floyd algorithm - all-pairs shortest paths
```



Graph

- **Graph - Floyd algorithm** - all-pairs shortest paths

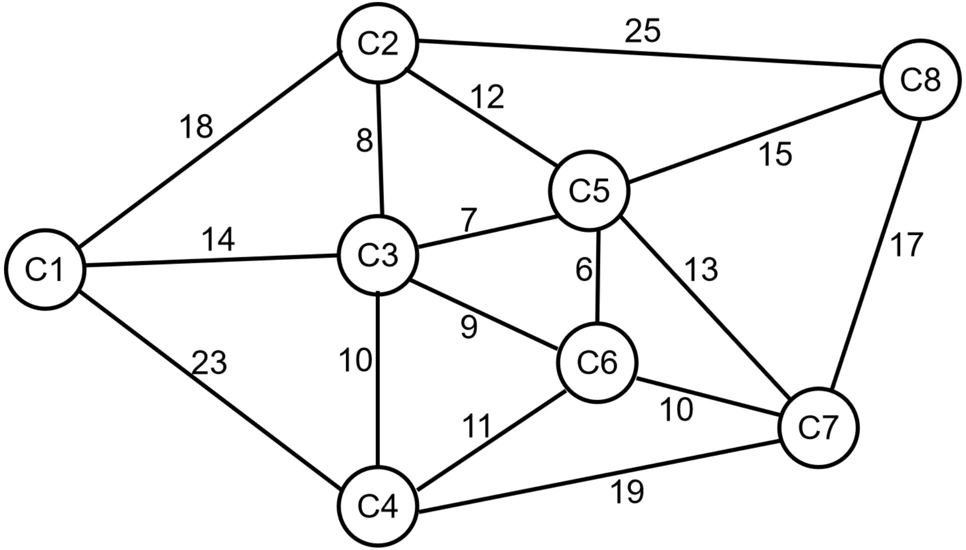
```
0-path =>
        0       1       2       3       4       5       6       7
0       0       18      14      23      10000   10000   10000   10000
1       18      0       8       10000   12      10000   10000   25
2       14      8       0       10      7       9       10000   10000
3       23      10000   10      0       10000   11      19      10000
4       10000   12      7       10000   0       6       13      15
5       10000   10000   9       11      6       0       10      10000
6       10000   10000   10000   19      13      10      0       17
7       10000   25      10000   10000   15      10000   17      0
1-path =>
        0       1       2       3       4       5       6       7
0       0       18      14      23      10000   10000   10000   10000
1       18      0       8       41      12      10000   10000   25
2       14      8       0       10      7       9       10000   10000
3       23      41      10      0       10000   11      19      10000
4       10000   12      7       10000   0       6       13      15
5       10000   10000   9       11      6       0       10      10000
6       10000   10000   10000   19      13      10      0       17
7       10000   25      10000   10000   15      10000   17      0
2-path =>
        0       1       2       3       4       5       6       7
0       0       18      14      23      30      10000   10000   43
1       18      0       8       41      12      10000   10000   25
2       14      8       0       10      7       9       10000   33
3       23      41      10      0       53      11      19      66
4       30      12      7       53      0       6       13      15
5       10000   10000   9       11      6       0       10      10000
6       10000   10000   10000   19      13      10      0       17
7       43      25      33      66      15      10000   17      0
```



Graph

- **Graph - Floyd algorithm** - all-pairs shortest paths

```
3-path =>
        0       1       2       3       4       5       6       7
0       0       18      14      23      21      23      10000   43
1       18      0       8       18      12      17      10000   25
2       14      8       0       10      7       9       10000   33
3       23      18      10      0       17      11      19      43
4       21      12      7       17      0       6       13      15
5       23      17      9       11      6       0       10      42
6       10000   10000   10000   19      13      10      0       17
7       43      25      33      43      15      42      17      0
4-path =>
        0       1       2       3       4       5       6       7
0       0       18      14      23      21      23      42      43
1       18      0       8       18      12      17      37      25
2       14      8       0       10      7       9       29      33
3       23      18      10      0       17      11      19      43
4       21      12      7       17      0       6       13      15
5       23      17      9       11      6       0       10      42
6       42      37      29      19      13      10      0       17
7       43      25      33      43      15      42      17      0
5-path =>
        0       1       2       3       4       5       6       7
0       0       18      14      23      21      23      34      36
1       18      0       8       18      12      17      25      25
2       14      8       0       10      7       9       20      22
3       23      18      10      0       17      11      19      32
4       21      12      7       17      0       6       13      15
5       23      17      9       11      6       0       10      21
6       34      25      20      19      13      10      0       17
7       36      25      22      32      15      21      17      0
```



Graph

- **Graph - Floyd algorithm** - all-pairs shortest paths

6-path =>

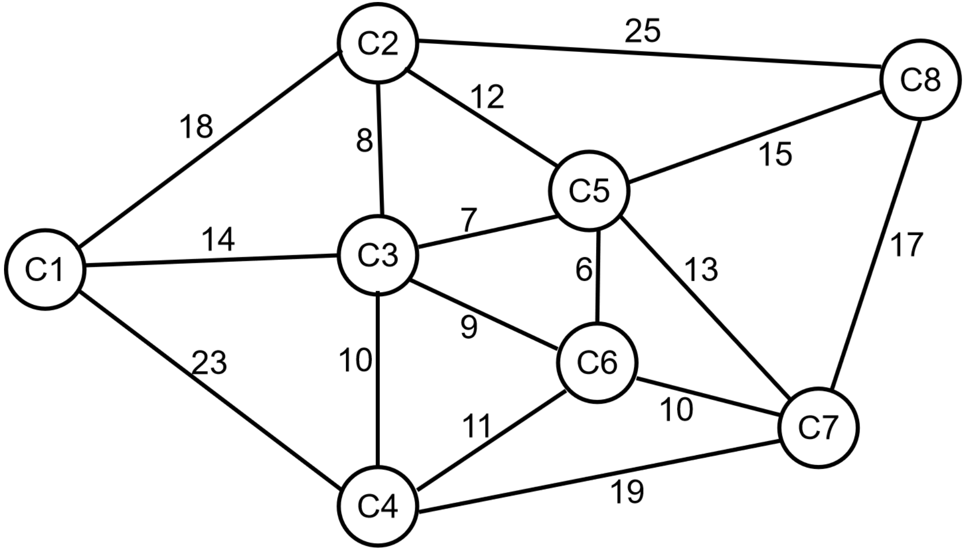
	0	1	2	3	4	5	6	7
0	0	18	14	23	21	23	33	36
1	18	0	8	18	12	17	25	25
2	14	8	0	10	7	9	19	22
3	23	18	10	0	17	11	19	32
4	21	12	7	17	0	6	13	15
5	23	17	9	11	6	0	10	21
6	33	25	19	19	13	10	0	17
7	36	25	22	32	15	21	17	0

7-path =>

	0	1	2	3	4	5	6	7
0	0	18	14	23	21	23	33	36
1	18	0	8	18	12	17	25	25
2	14	8	0	10	7	9	19	22
3	23	18	10	0	17	11	19	32
4	21	12	7	17	0	6	13	15
5	23	17	9	11	6	0	10	21
6	33	25	19	19	13	10	0	17
7	36	25	22	32	15	21	17	0

8-path =>

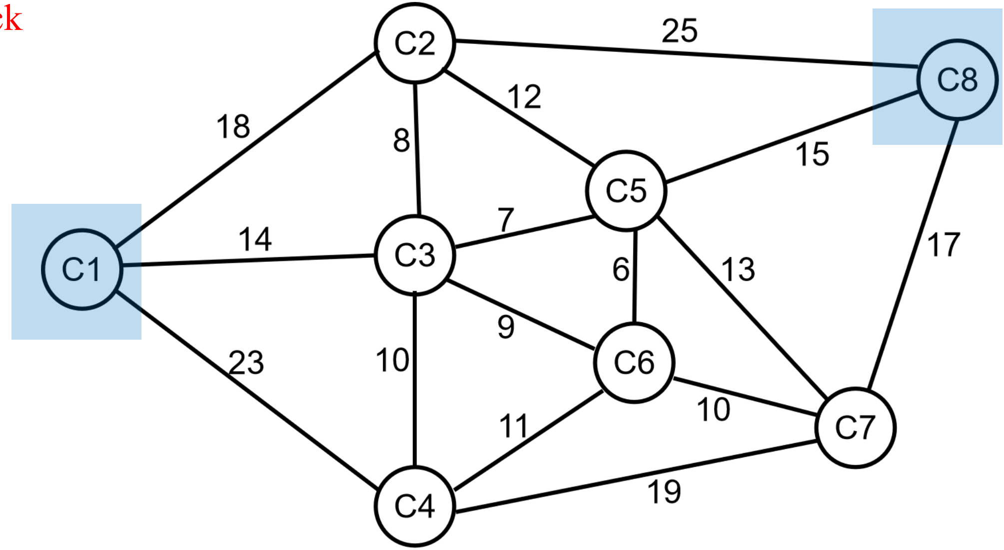
	0	1	2	3	4	5	6	7
0	0	18	14	23	21	23	33	36
1	18	0	8	18	12	17	25	25
2	14	8	0	10	7	9	19	22
3	23	18	10	0	17	11	19	32
4	21	12	7	17	0	6	13	15
5	23	17	9	11	6	0	10	21
6	33	25	19	19	13	10	0	17
7	36	25	22	32	15	21	17	0



Graph

- **Graph - Floyd algorithm** - all-pairs shortest paths
  - k-path : intermediate vertices (except the two ends) all have indices less than k
  - **D.P.** (k+1)-path^{best}[s to t] <= k-path^{best}[s to k]+k-path^{best}[k to t] or k-path^{best}[s to t]
  - retrieve the min-path
    - no preceding vertex to track
    - how ?

	C1	C2	C3	C4	C5	C6	C7	C8
C1	0	18	14	23	21	23	33	36
C2	18	0	8	18	12	17	25	25
C3	14	8	0	10	7	9	19	22
C4	23	18	10	0	17	11	19	32
C5	21	12	7	17	0	6	13	15
C6	23	17	9	11	6	0	10	21
C7	33	25	19	19	13	10	0	17
C8	36	25	22	32	15	21	17	0

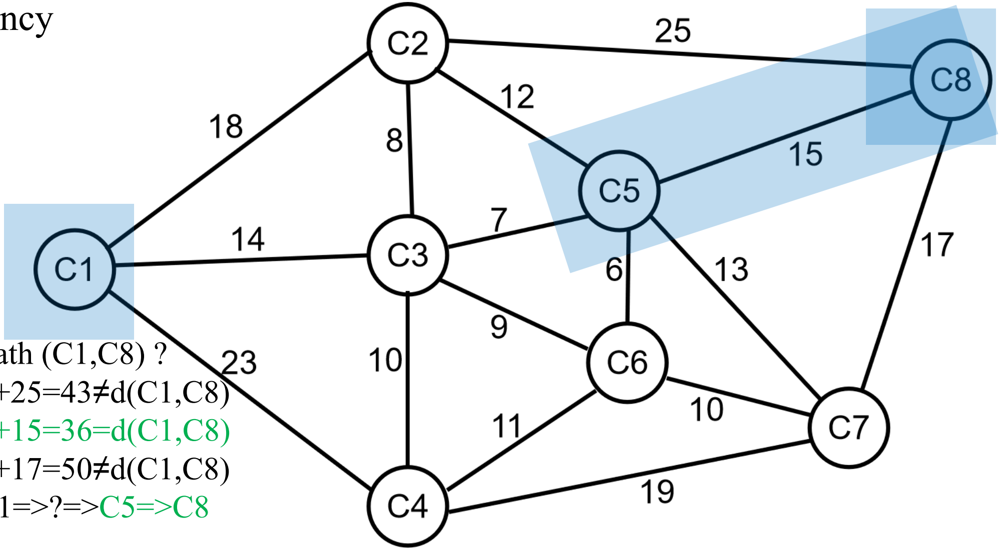


Graph

- **Graph - Floyd algorithm** - all-pairs shortest paths
  - k-path : intermediate vertices (except the two ends) all have indices less than k
  - **D.P.** (k+1)-path^{best}[s to t] <= k-path^{best}[s to k]+k-path^{best}[k to t] or k-path^{best}[s to t]
  - retrieve the min-path
    - check the distance consistency

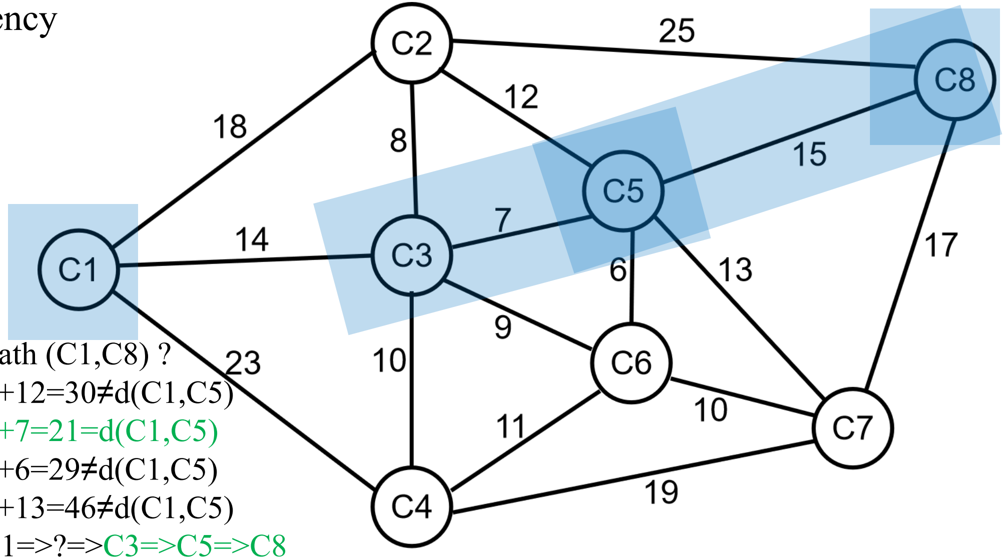
	C1	C2	C3	C4	C5	C6	C7	C8
C1	0	18	14	23	21	23	33	36
C2	18	0	8	18	12	17	25	25
C3	14	8	0	10	7	9	19	22
C4	23	18	10	0	17	11	19	32
C5	21	12	7	17	0	6	13	15
C6	23	17	9	11	6	0	10	21
C7	33	25	19	19	13	10	0	17
C8	36	25	22	32	15	21	17	0

e.g. d(C1,C8)=36, min-path (C1,C8) ?
d(C1,C2)+w(C2,C8)=18+25=43≠d(C1,C8)
d(C1,C5)+w(C5,C8)=21+15=36=d(C1,C8)
d(C1,C7)+w(C7,C8)=33+17=50≠d(C1,C8)
thus min-path(C1,C8): C1=>?=>C5=>C8



Graph

- **Graph - Floyd algorithm** - all-pairs shortest paths
  - k-path : intermediate vertices (except the two ends) all have indices less than k
  - **D.P.** (k+1)-path^{best}[s to t] <= k-path^{best}[s to k]+k-path^{best}[k to t] or k-path^{best}[s to t]
  - retrieve the min-path
    - check the distance consistency

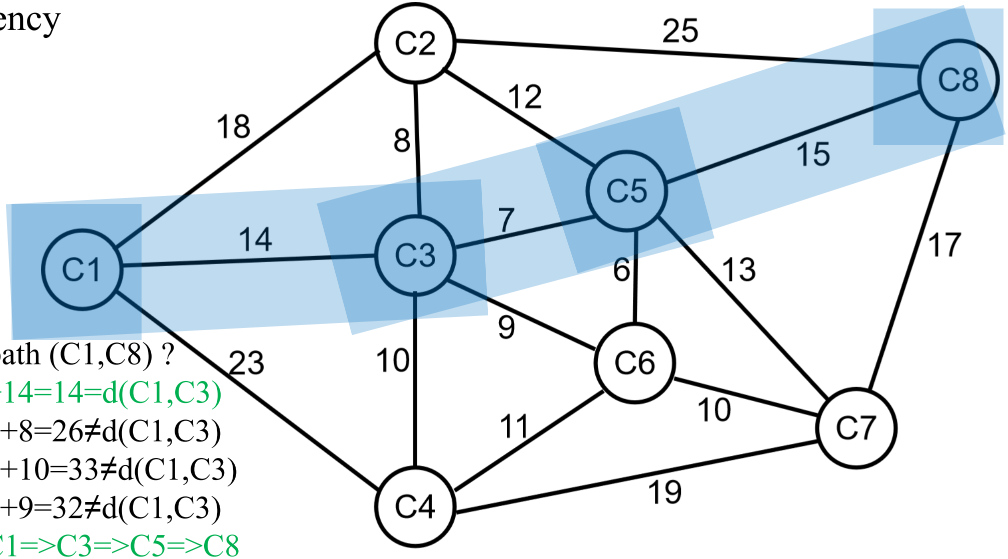


	C1	C2	C3	C4	C5	C6	C7	C8
C1	0	18	14	23	21	23	33	36
C2	18	0	8	18	12	17	25	25
C3	14	8	0	10	7	9	19	22
C4	23	18	10	0	17	11	19	32
C5	21	12	7	17	0	6	13	15
C6	23	17	9	11	6	0	10	21
C7	33	25	19	19	13	10	0	17
C8	36	25	22	32	15	21	17	0

e.g. d(C1,C8)=36, min-path (C1,C8) ?
d(C1,C2)+w(C2,C5)=18+12=30≠d(C1,C5)
d(C1,C3)+w(C3,C5)=14+7=21=d(C1,C5)
d(C1,C6)+w(C6,C5)=23+6=29≠d(C1,C5)
d(C1,C7)+w(C7,C5)=33+13=46≠d(C1,C5)
thus min-path(C1,C8): C1=>?=>C3=>C5=>C8

Graph

- **Graph - Floyd algorithm** - all-pairs shortest paths
  - k-path : intermediate vertices (except the two ends) all have indices less than k
  - **D.P.** (k+1)-path^{best}[s to t] <= k-path^{best}[s to k]+k-path^{best}[k to t] or k-path^{best}[s to t]
  - retrieve the min-path
    - check the distance consistency



	C1	C2	C3	C4	C5	C6	C7	C8
C1	0	18	14	23	21	23	33	36
C2	18	0	8	18	12	17	25	25
C3	14	8	0	10	7	9	19	22
C4	23	18	10	0	17	11	19	32
C5	21	12	7	17	0	6	13	15
C6	23	17	9	11	6	0	10	21
C7	33	25	19	19	13	10	0	17
C8	36	25	22	32	15	21	17	0

e.g. d(C1,C8)=36, min-path (C1,C8) ?
d(C1,C1)+w(C1,C3)=0+14=14=d(C1,C3)
d(C1,C2)+w(C2,C3)=18+8=26≠d(C1,C3)
d(C1,C4)+w(C4,C3)=23+10=33≠d(C1,C3)
d(C1,C6)+w(C6,C3)=23+9=32≠d(C1,C3)
thus min-path(C1,C8): C1=>C3=>C5=>C8

Graph

- **Graph - Floyd algorithm** - all-pairs shortest paths
  - k-path : intermediate vertices (except the two ends) all have indices less than k
  - **D.P.** (k+1)-path^{best}[s to t] <= k-path^{best}[s to k]+k-path^{best}[k to t] or k-path^{best}[s to t]
  - retrieve the min-path - check the distance consistency
  - complexity
    - adjacency list based *Dijkstra* with heap based MDFO: O(|V| (|V|+|E|) *log* |V|)
      - suitable to sparse graphs
    - adjacency matrix based *Dijkstra*: O(|V| |V|^2) = O(|V|^3)
    - *Floyd*: O(|V|^3)
      - suitable to dense graphs (in terms of not only efficiency but also implementation simplicity)

```
void Floyd(LGraph* g,int *d[]){int n=g->num(),i,j,k;
        for(i=0;i<n;i++) for(j=0;j<n;j++)
                if(g->wgt(i,j)<0) d[i][j]=D_INF; else d[i][j]=g->wgt(i,j);
        for(i=0;i<n;i++) d[i][i]=0;
        for(k=0;k<n;k++){std::cout<<k<<"-path => \n";showFloyd(d,n);
                for(i=0;i<n;i++) for(j=0;j<n;j++)
                        if(d[i][j]>(d[i][k]+d[k][j])) d[i][j]=d[i][k]+d[k][j];}
        std::cout<<k<<"-path => \n";showFloyd(d,n);
}
```

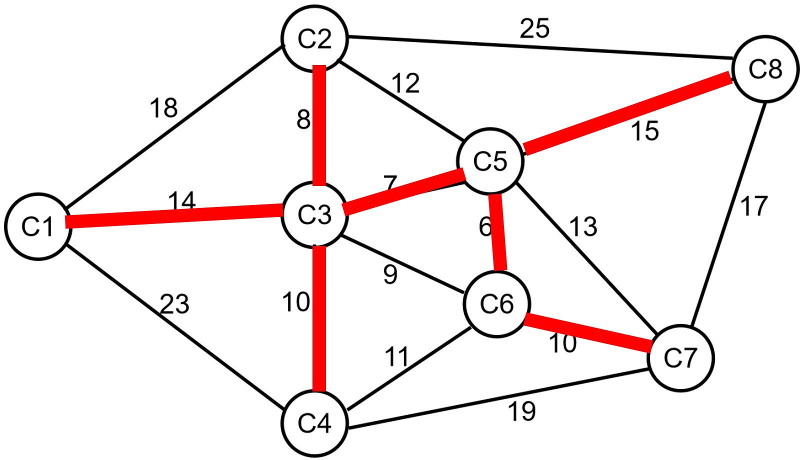
THANK YOU

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
    - minimum-cost means having the minimum sum of all edge weights of the sub-graph
    - such sub-graph must be a *tree* (so why called MS“T”)
      - a cycle must have a redundant edge that can be removed without violating sub-graph connectivity
  -

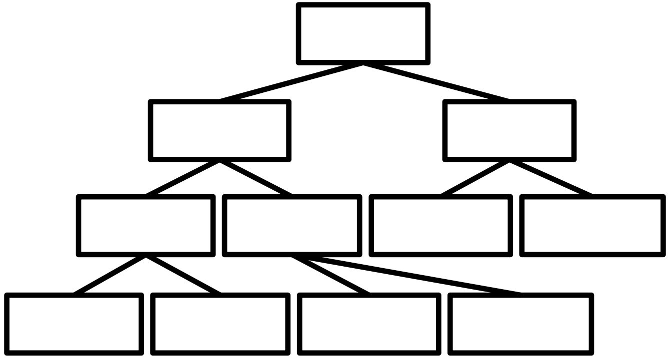
MST = {C1C3,C2C3,C3C4,C3C5,C5C6,C5C8,C6C7}
minimum-cost |MST| = 14+8+10+7+6+15+10 = 70

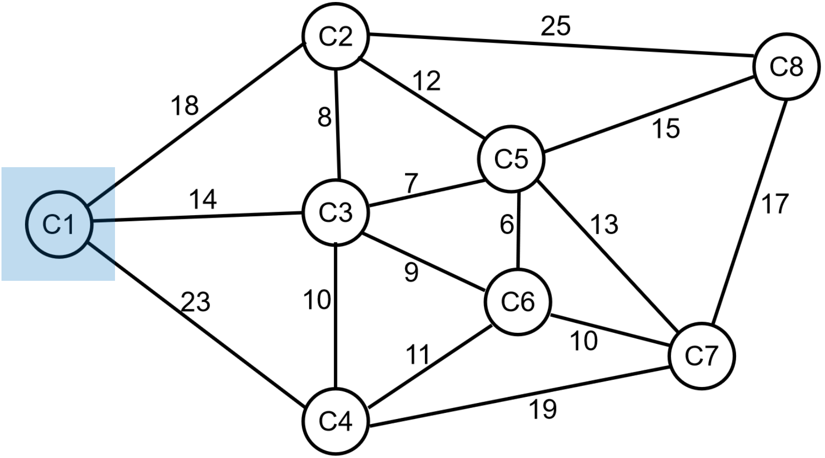


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1}
E = {}

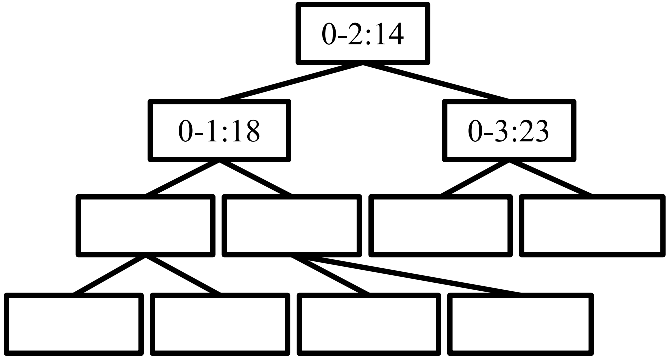


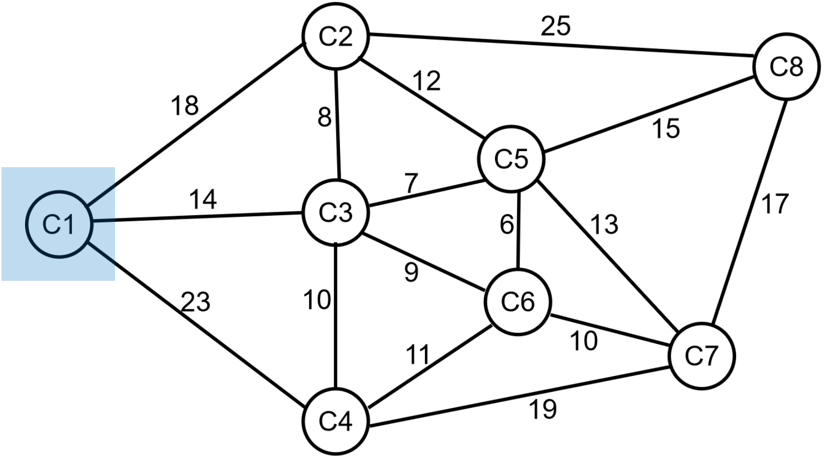


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1}
E = {}

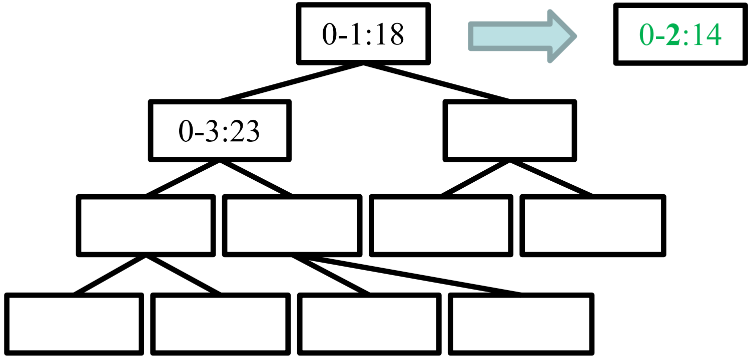


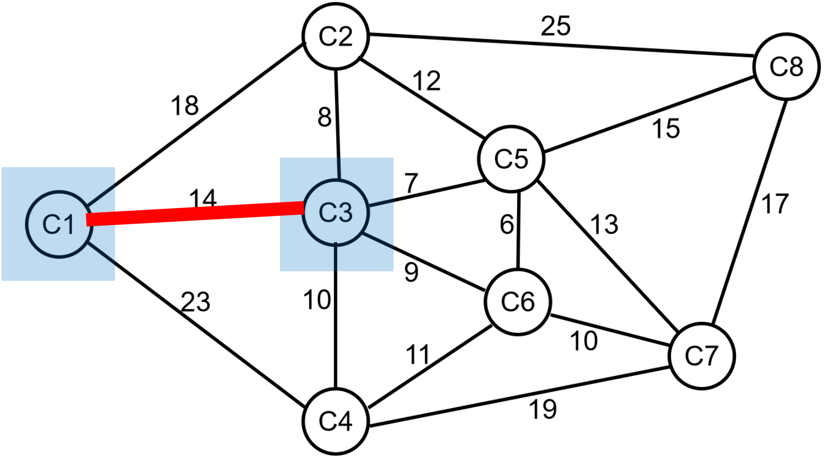


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3}
E = {C1C3}

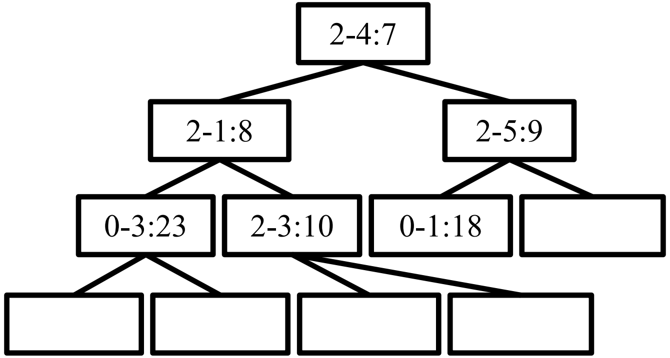


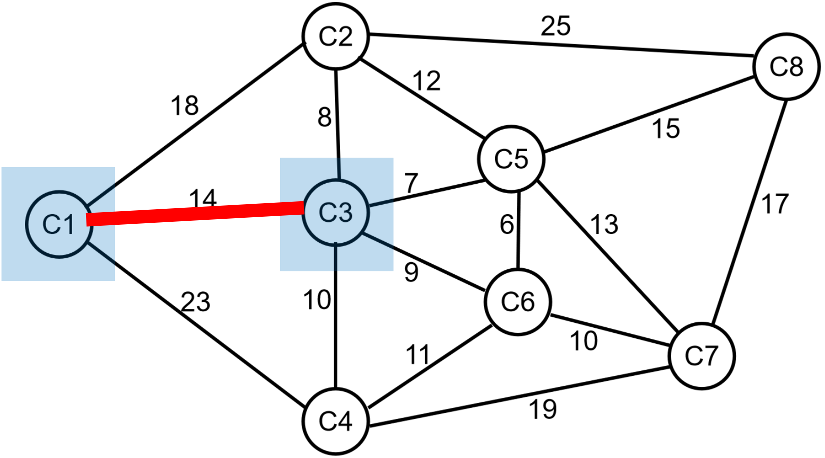


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3}
E = {C1C3}

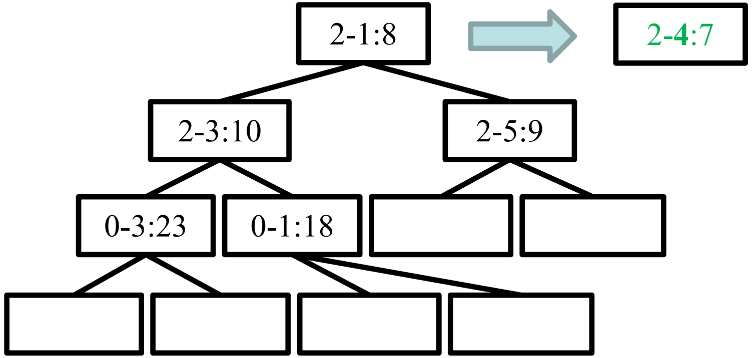


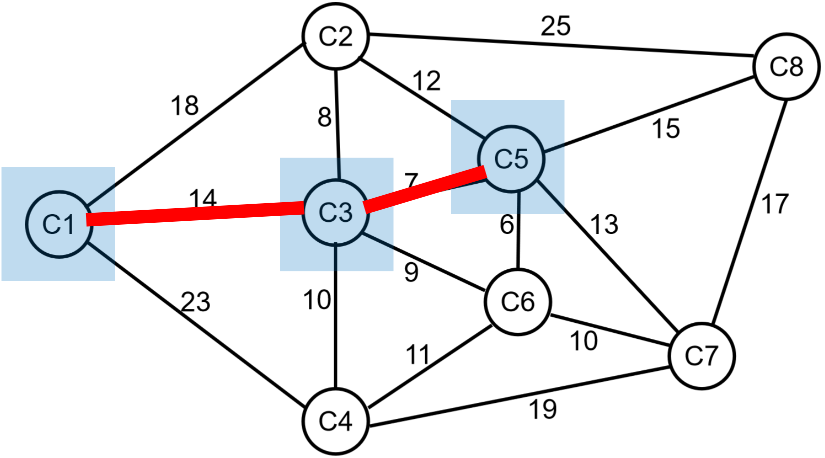


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5}
E = {C1C3,C3C5}

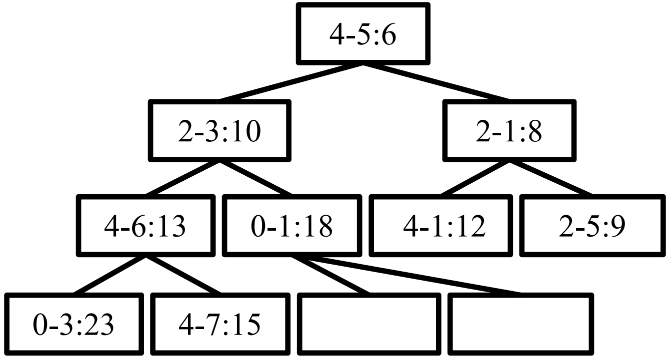


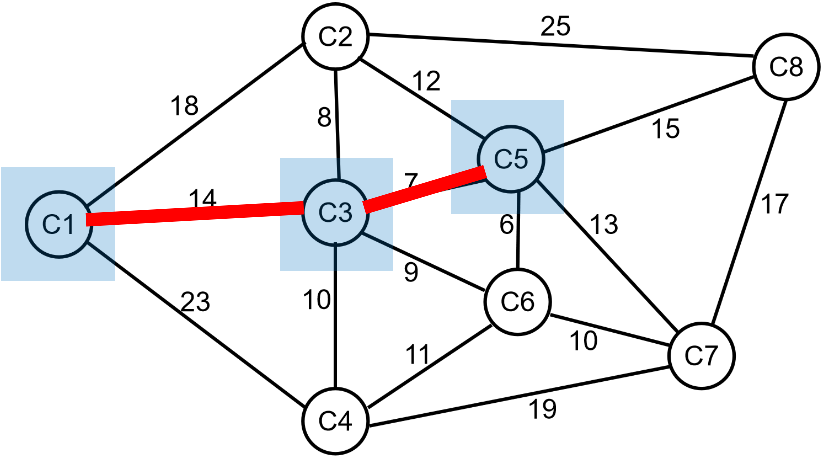


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5}
E = {C1C3,C3C5}

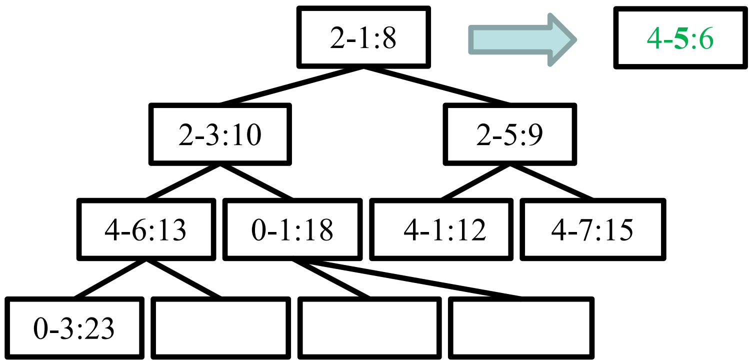


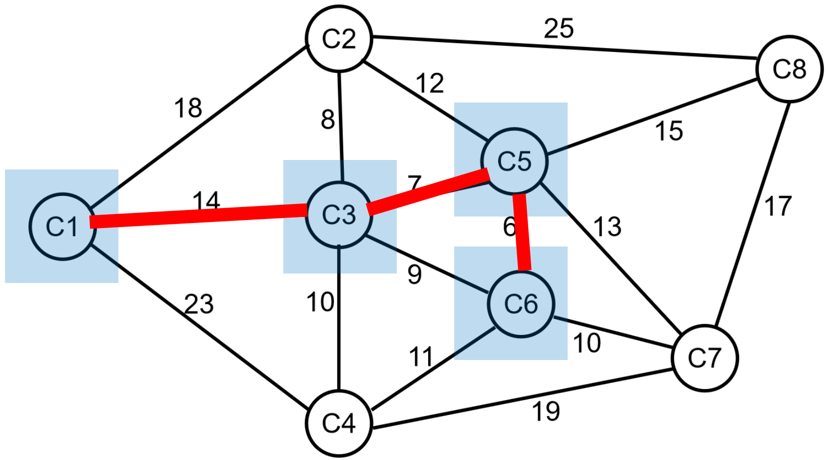


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5, C6}
E = {C1C3,C3C5,C5C6}

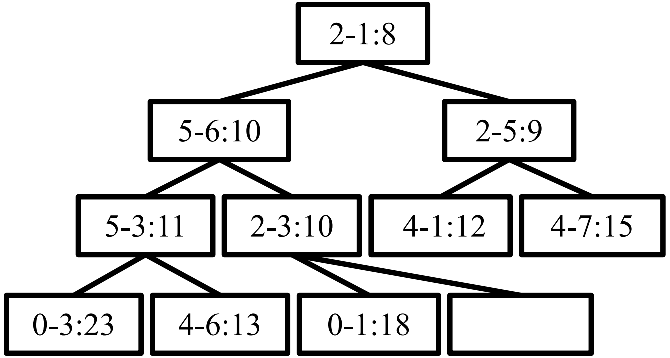


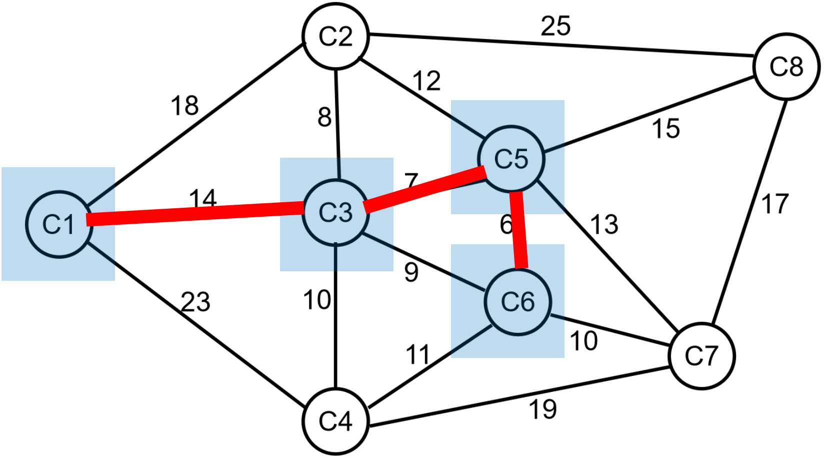


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5, C6}
E = {C1C3,C3C5,C5C6}

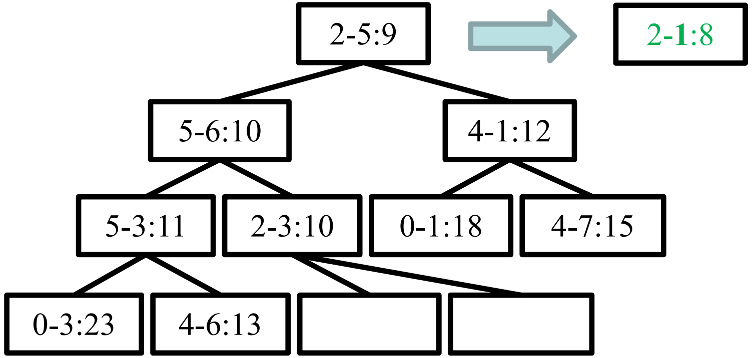


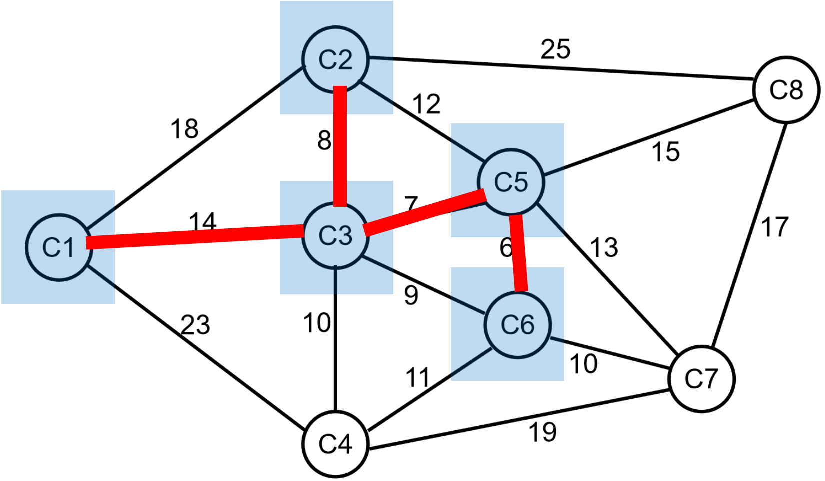


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5, C6, C2}
E = {C1C3,C3C5,C5C6,C3C2}

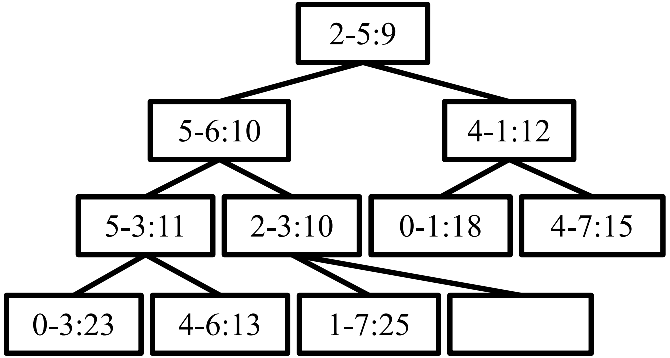


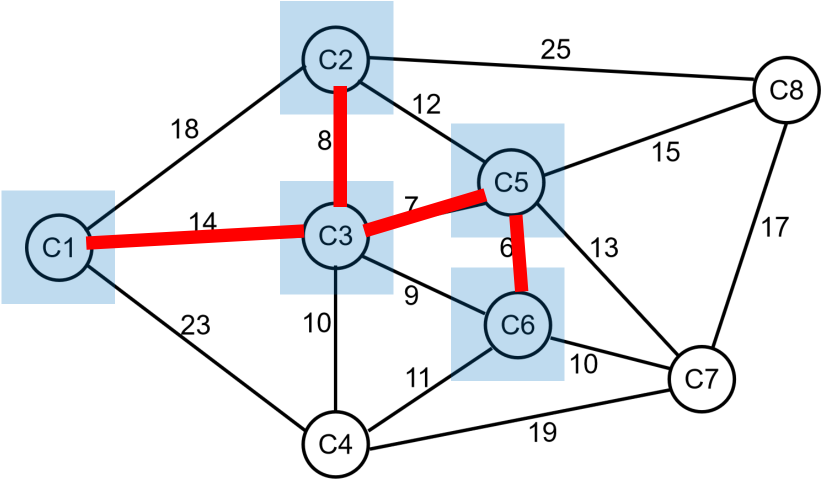


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5, C6, C2}
E = {C1C3,C3C5,C5C6,C3C2}

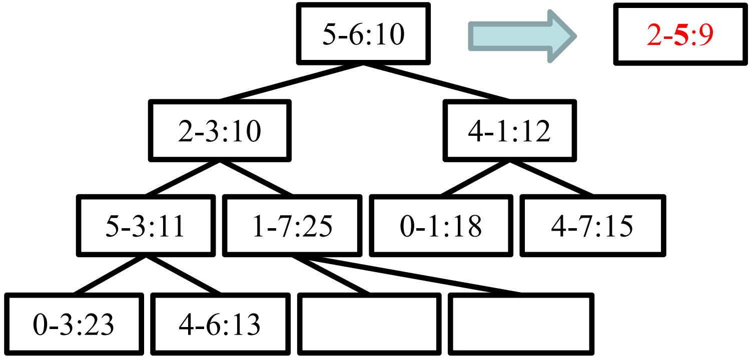


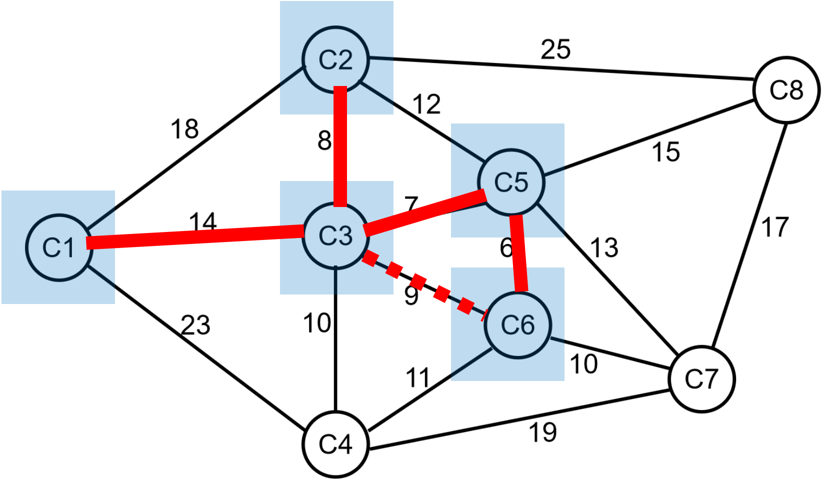


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5, C6, C2}
E = {C1C3,C3C5,C5C6,C3C2}

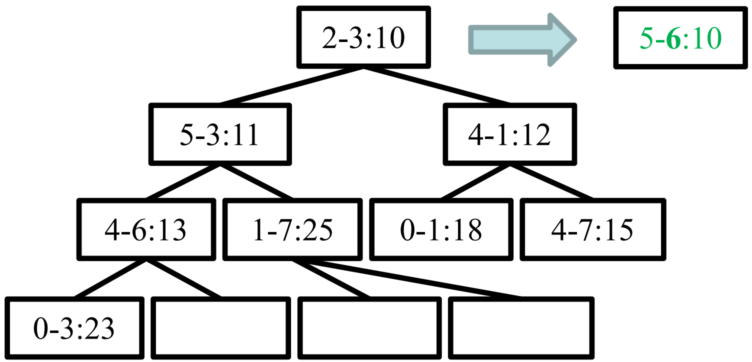


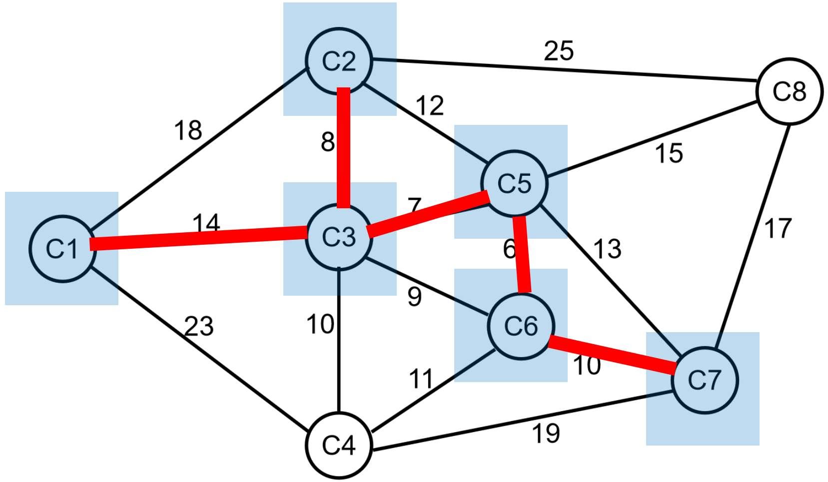


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5, C6, C2, C7}
E = {C1C3,C3C5,C5C6,C3C2,C6C7}

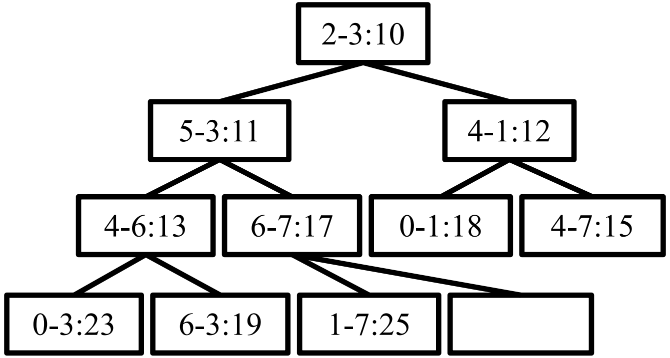


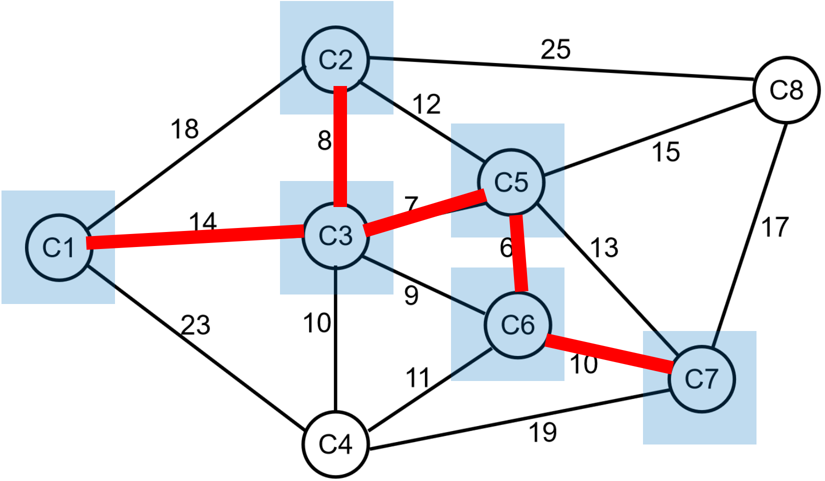


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5, C6, C2, C7}
E = {C1C3,C3C5,C5C6,C3C2,C6C7}

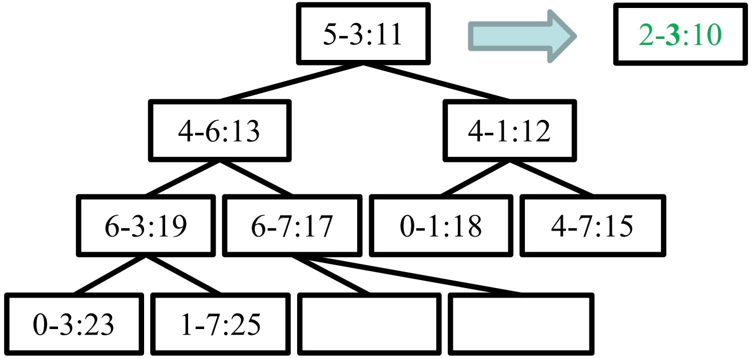


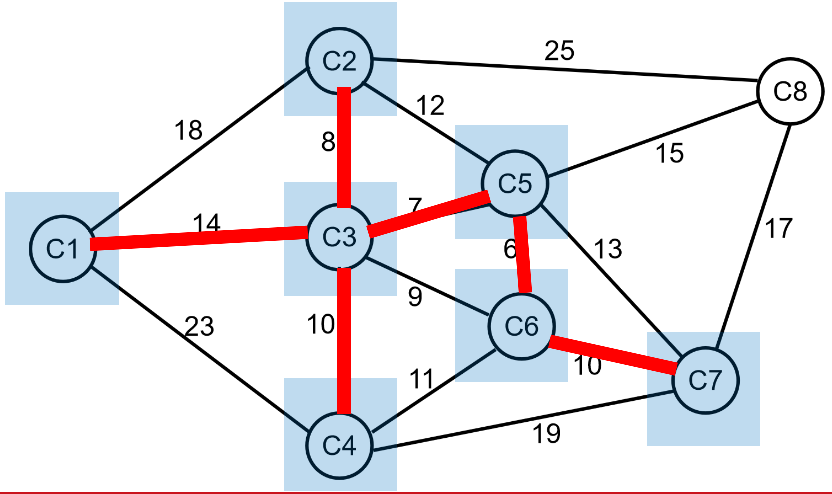


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5, C6, C2, C7, C4}
E = {C1C3,C3C5,C5C6,C3C2,C6C7,C3C4}

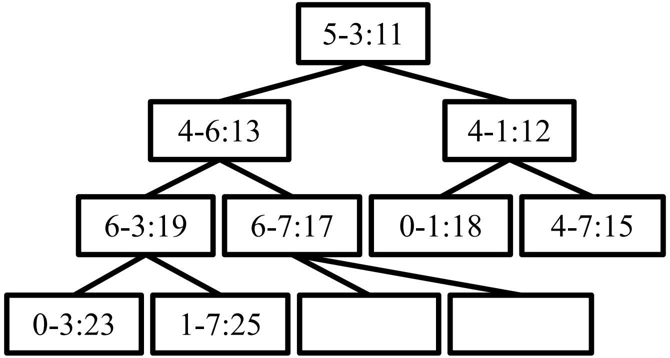


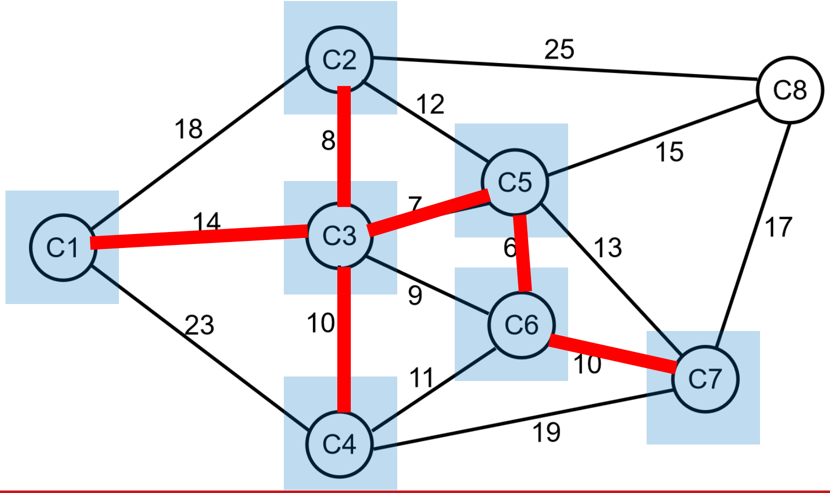


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5, C6, C2, C7, C4}
E = {C1C3,C3C5,C5C6,C3C2,C6C7,C3C4}

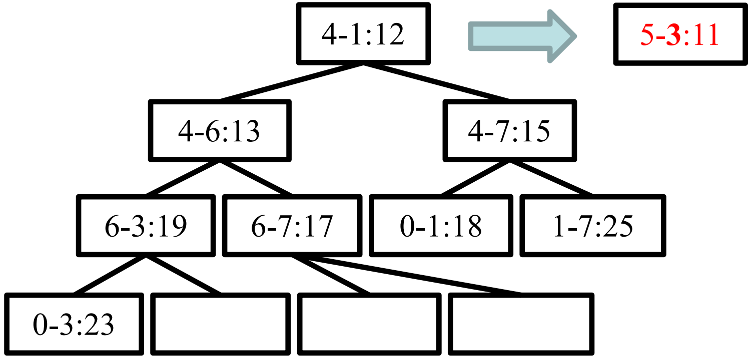


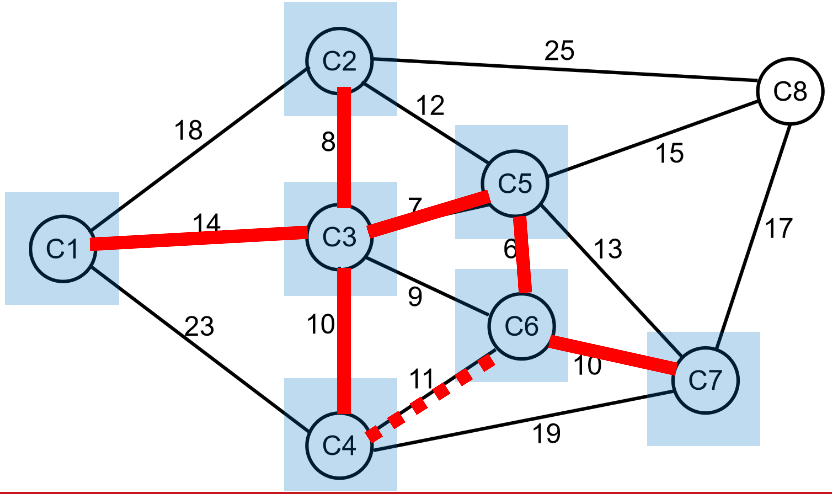


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5, C6, C2, C7, C4}
E = {C1C3,C3C5,C5C6,C3C2,C6C7,C3C4}

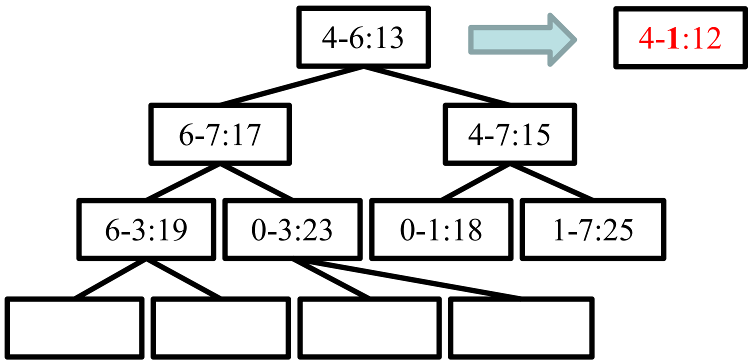


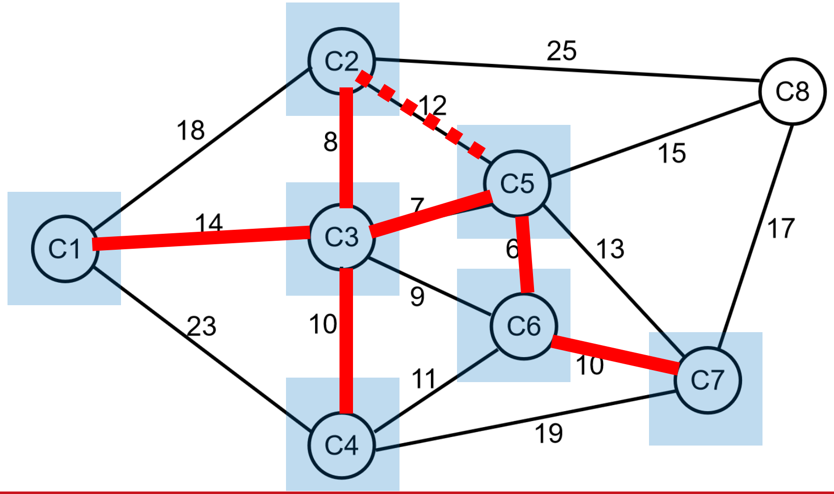


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5, C6, C2, C7, C4}
E = {C1C3,C3C5,C5C6,C3C2,C6C7,C3C4}

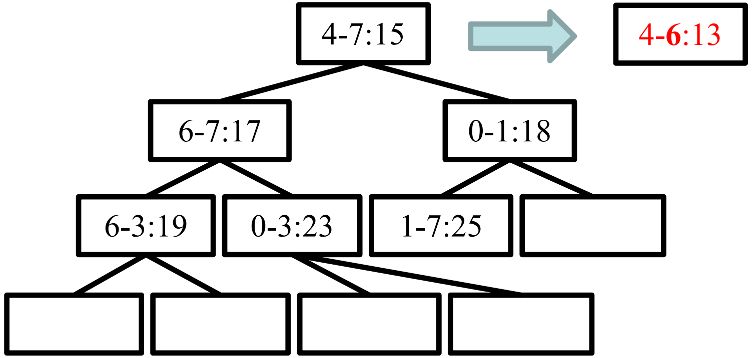


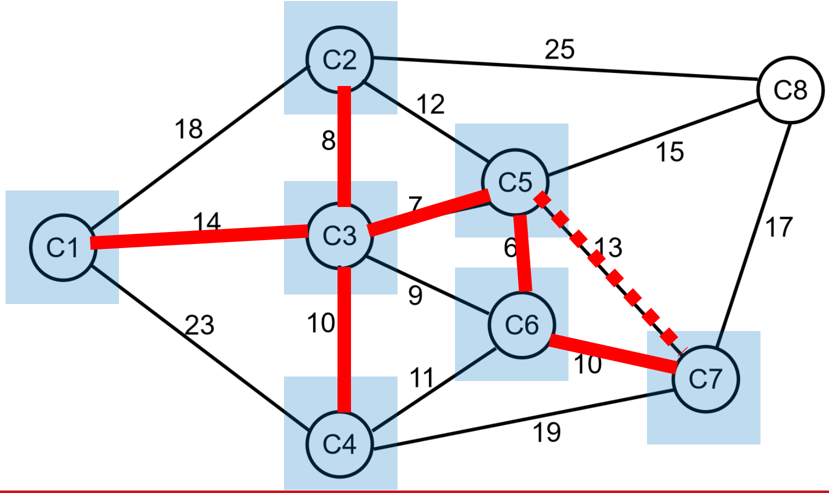


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5, C6, C2, C7, C4}
E = {C1C3,C3C5,C5C6,C3C2,C6C7,C3C4}

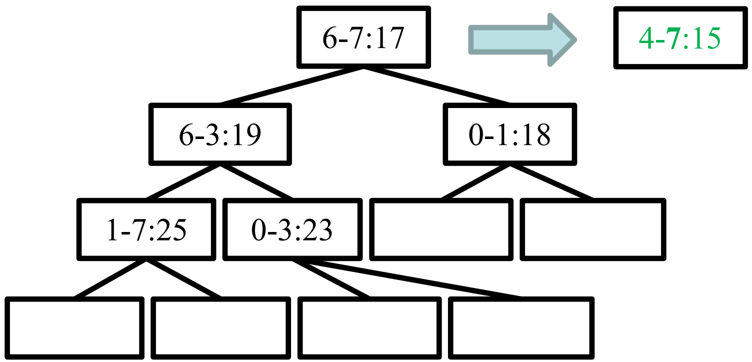


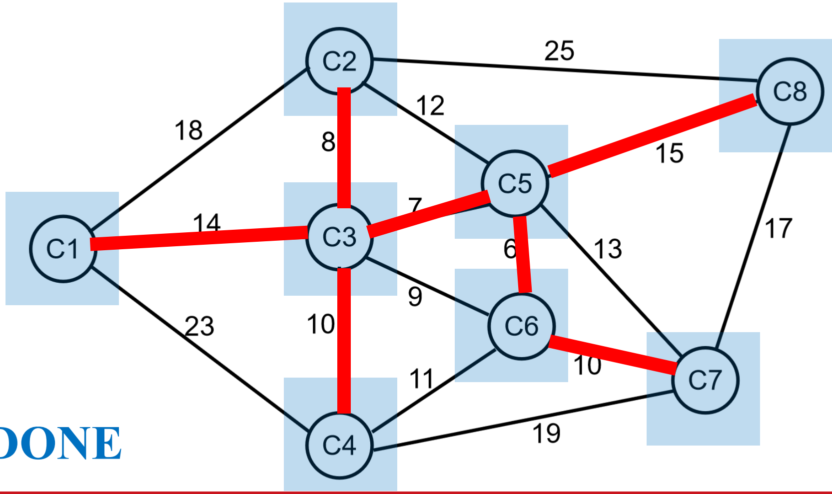


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm
    - incrementally incorporate the closest vertex until all vertices are incorporated

V = {C1, C3, C5, C6, C2, C7, C4, C8}
E = {C1C3,C3C5,C5C6,C3C2,C6C7,C3C4,C5C8}



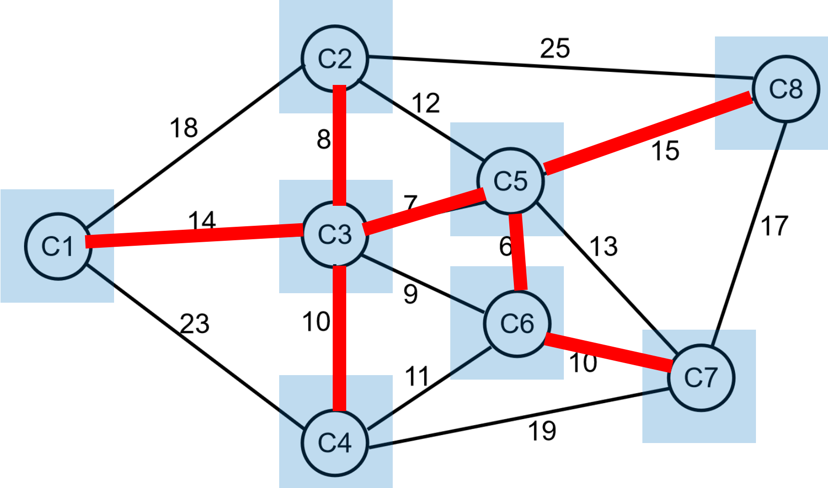


DONE

Graph

- **Minimum-cost spanning tree (MST)**
  - *Prim* algorithm - incrementally incorporate the closest vertex

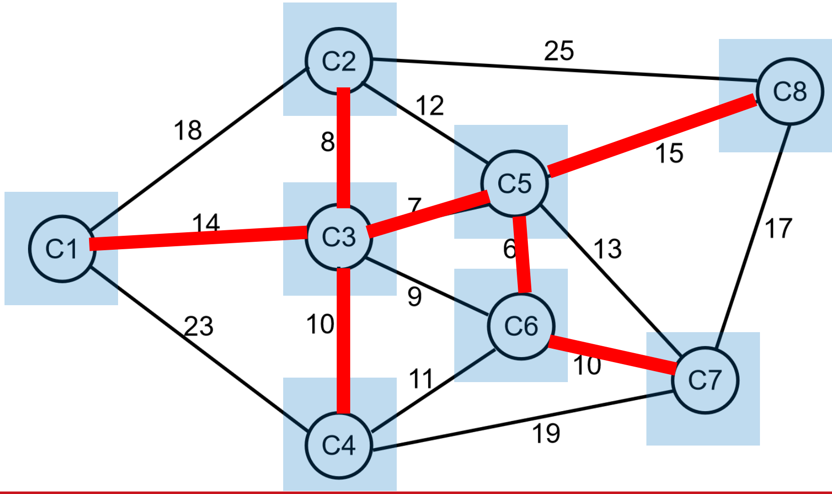
```
Prim MST from 0 =>
add out-MST edges from 0 : =>0-1:18=>0-2:14=>0-3:23
remove min-edge 0-2:14=>2 incorporated into MST
add out-MST edges from 2 : =>2-1:8=>2-3:10=>2-4:7=>2-5:9
remove min-edge 2-4:7=>4 incorporated into MST
add out-MST edges from 4 : =>4-1:12=>4-5:6=>4-6:13=>4-7:15
remove min-edge 4-5:6=>5 incorporated into MST
add out-MST edges from 5 : =>5-3:11=>5-6:10
remove min-edge 2-1:8=>1 incorporated into MST
add out-MST edges from 1 : =>1-7:25
remove min-edge 2-5:9=>5 already in MST
remove min-edge 5-6:10=>6 incorporated into MST
add out-MST edges from 6 : =>6-3:19=>6-7:17
remove min-edge 2-3:10=>3 incorporated into MST
add out-MST edges from 3 :
remove min-edge 5-3:11=>3 already in MST
remove min-edge 4-1:12=>1 already in MST
remove min-edge 4-6:13=>6 already in MST
remove min-edge 4-7:15=>7 incorporated into MST
Prim MST from 0 =>| 0-2:14 2-4:7 4-5:6 2-1:8 5-6:10 2-3:10 4-7:15
```



Graph

- **Minimum-cost spanning tree (MST)**
  - *Prim* algorithm - incrementally incorporate the closest vertex

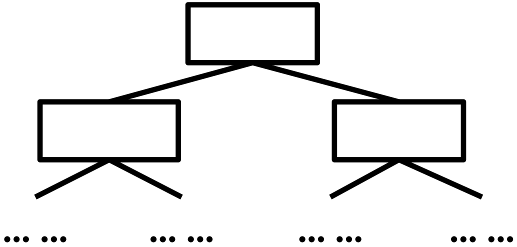
```
Prim MST from 7 =>
add out-MST edges from 7 : =>7-1:25=>7-4:15=>7-6:17
remove min-edge 7-4:15=>4 incorporated into MST
add out-MST edges from 4 : =>4-1:12=>4-2:7=>4-5:6=>4-6:13
remove min-edge 4-5:6=>5 incorporated into MST
add out-MST edges from 5 : =>5-2:9=>5-3:11=>5-6:10
remove min-edge 4-2:7=>2 incorporated into MST
add out-MST edges from 2 : =>2-0:14=>2-1:8=>2-3:10
remove min-edge 2-1:8=>1 incorporated into MST
add out-MST edges from 1 : =>1-0:18
remove min-edge 5-2:9=>2 already in MST
remove min-edge 2-3:10=>3 incorporated into MST
add out-MST edges from 3 : =>3-0:23=>3-6:19
remove min-edge 5-6:10=>6 incorporated into MST
add out-MST edges from 6 :
remove min-edge 5-3:11=>3 already in MST
remove min-edge 4-1:12=>1 already in MST
remove min-edge 4-6:13=>6 already in MST
remove min-edge 2-0:14=>0 incorporated into MST
Prim MST from 7 =>| 7-4:15 4-5:6 4-2:7 2-1:8 2-3:10 5-6:10 2-0:14
```

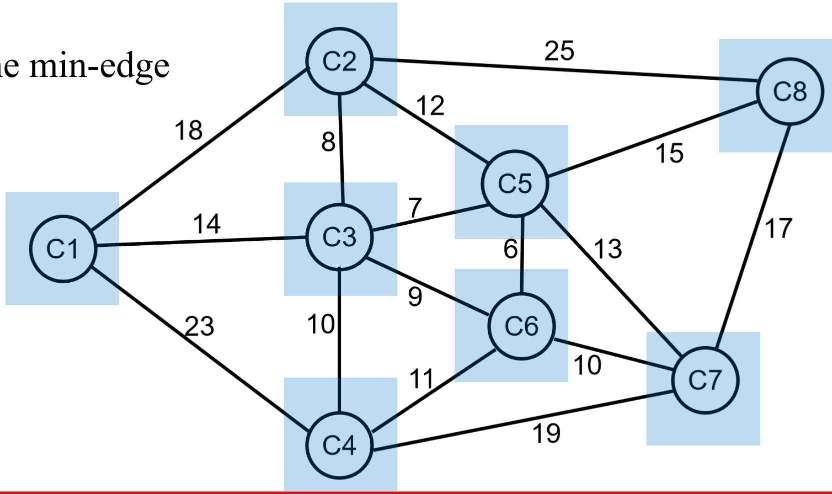


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm - incrementally incorporate the closest vertex
  - *Kruskal* algorithm
    - organize graph edges via heap
    - incrementally merge sub-graphs via the min-edge

{C1} {C2} {C3} {C4} {C5} {C6} {C7} {C8}
E = {}

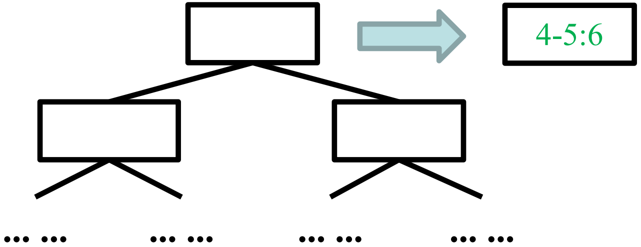


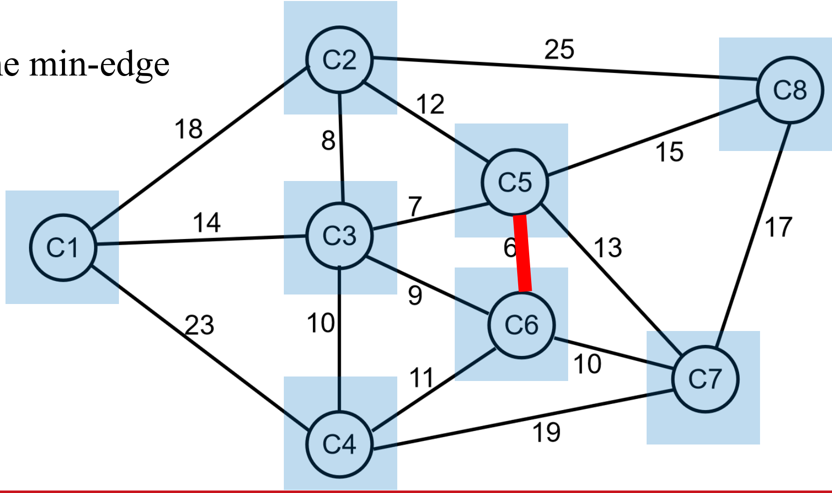


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm - incrementally incorporate the closest vertex
  - *Kruskal* algorithm
    - organize graph edges via heap
    - incrementally merge sub-graphs via the min-edge

{C1} {C2} {C3} {C4} {C5, C6} {C7} {C8}
E = {C5C6}

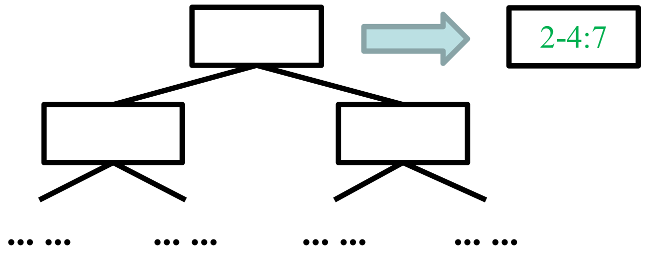


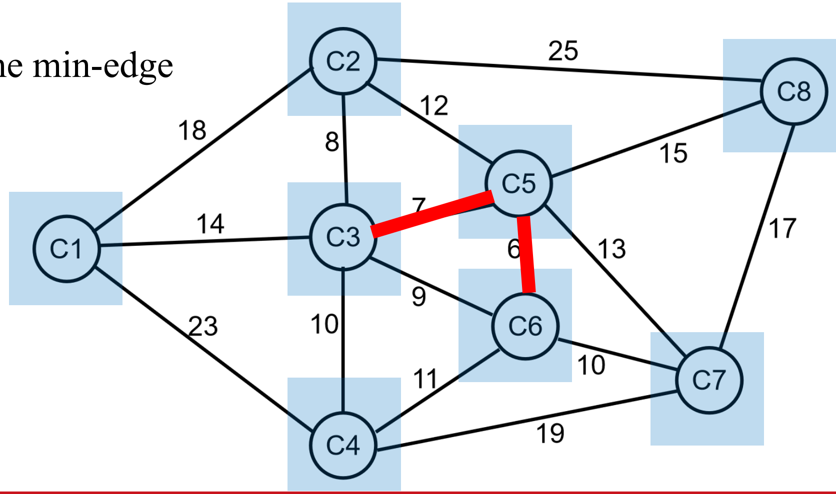


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm - incrementally incorporate the closest vertex
  - *Kruskal* algorithm
    - organize graph edges via heap
    - incrementally merge sub-graphs via the min-edge

{C1} {C2} {C3, C5, C6} {C4} {C7} {C8}
E = {C5C6,C3C5}

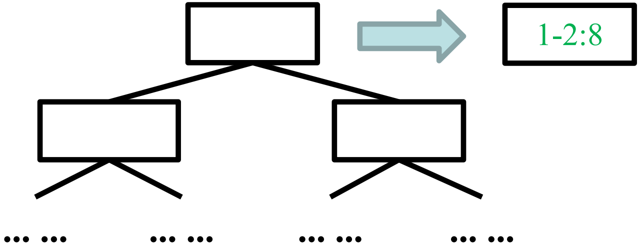


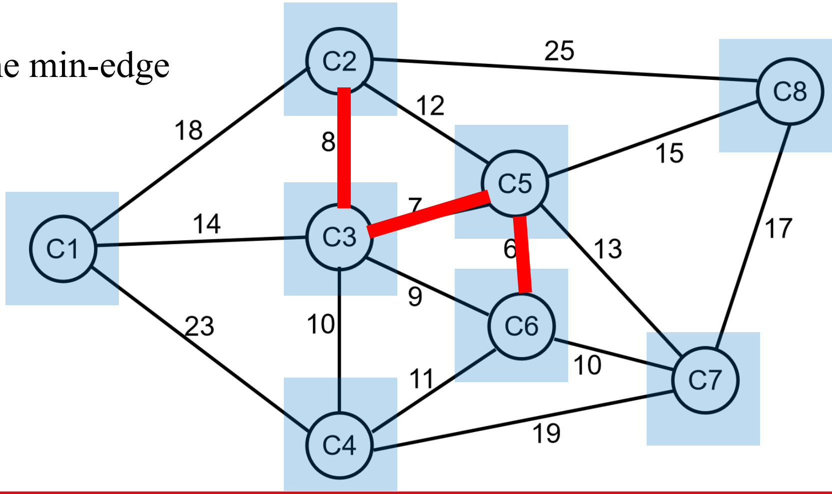


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm - incrementally incorporate the closest vertex
  - *Kruskal* algorithm
    - organize graph edges via heap
    - incrementally merge sub-graphs via the min-edge

{C1} {C2, C3, C5, C6} {C4} {C7} {C8}
E = {C5C6,C3C5,C2C3}

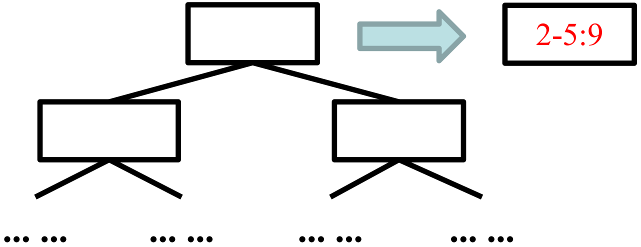


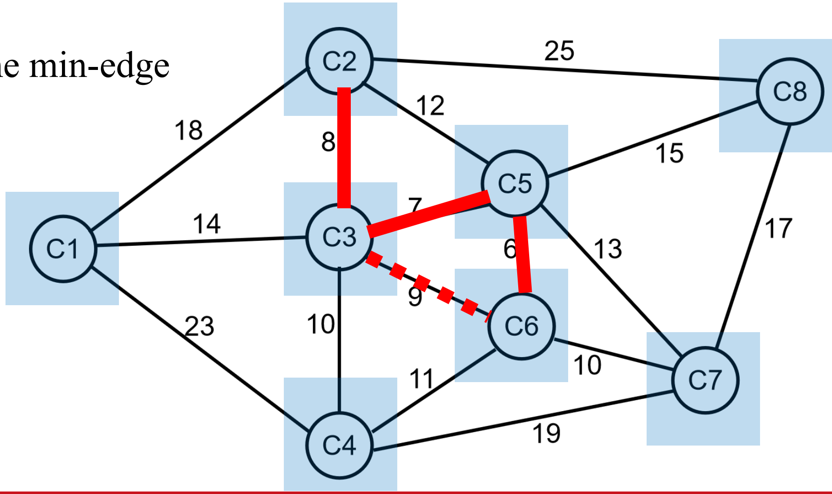


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm - incrementally incorporate the closest vertex
  - *Kruskal* algorithm
    - organize graph edges via heap
    - incrementally merge sub-graphs via the min-edge

{C1} {C2, C3, C5, C6} {C4} {C7} {C8}
E = {C5C6,C3C5,C2C3}

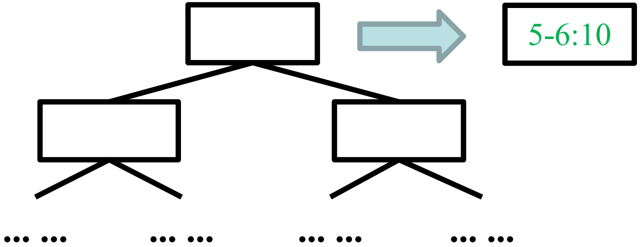


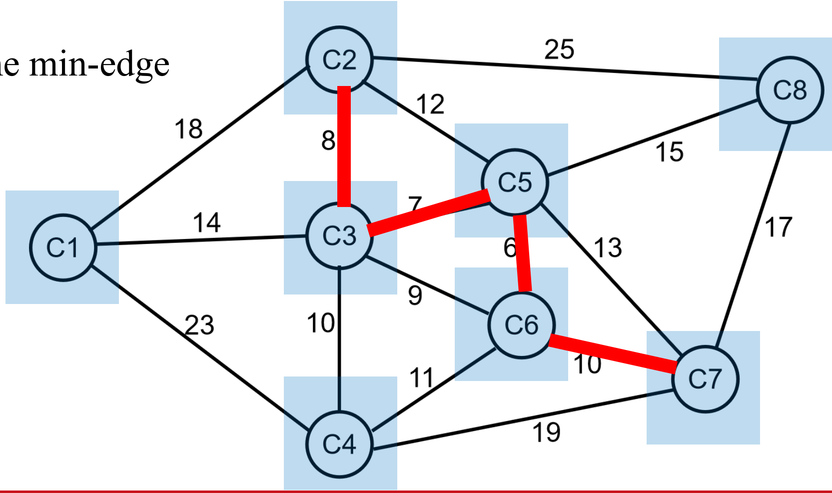


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm - incrementally incorporate the closest vertex
  - *Kruskal* algorithm
    - organize graph edges via heap
    - incrementally merge sub-graphs via the min-edge

{C1} {C2, C3, C5, C6, C7} {C4} {C8}
E = {C5C6,C3C5,C2C3,C6C7}

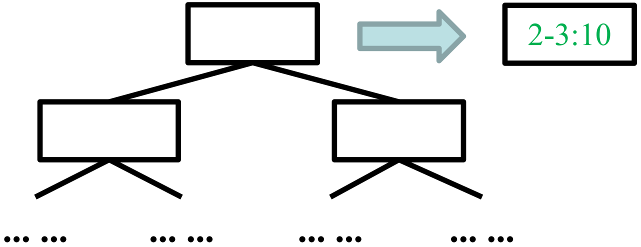


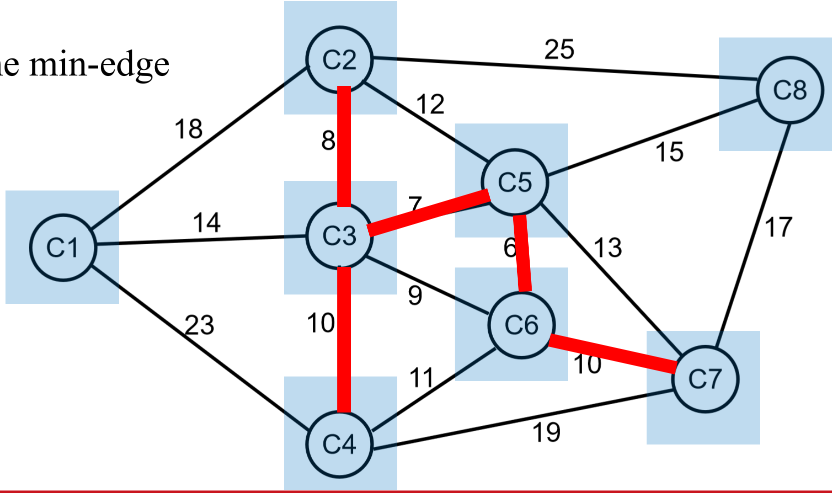


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm - incrementally incorporate the closest vertex
  - *Kruskal* algorithm
    - organize graph edges via heap
    - incrementally merge sub-graphs via the min-edge

{C1} {C2, C3, C4, C5, C6, C7} {C8}
E = {C5C6,C3C5,C2C3,C6C7,C3C4}

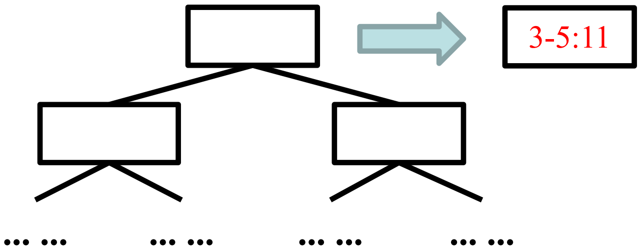


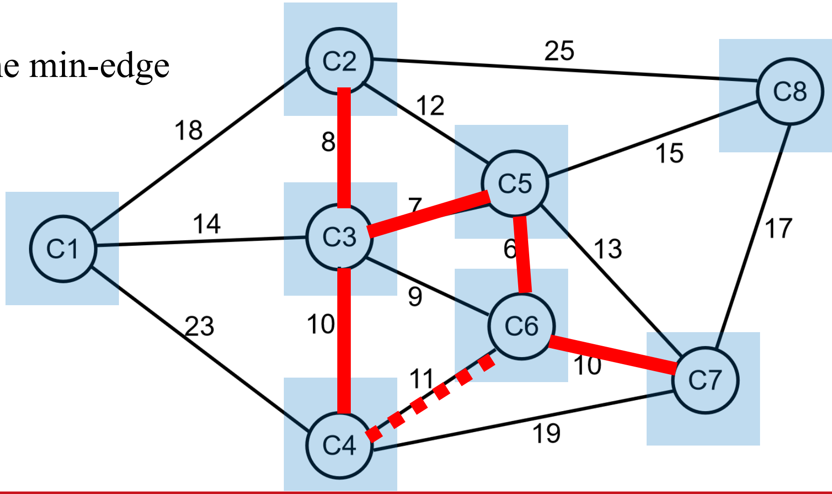


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm - incrementally incorporate the closest vertex
  - *Kruskal* algorithm
    - organize graph edges via heap
    - incrementally merge sub-graphs via the min-edge

{C1} {C2, C3, C4, C5, C6, C7} {C8}
E = {C5C6,C3C5,C2C3,C6C7,C3C4}



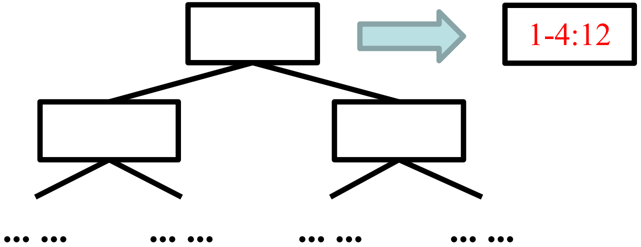


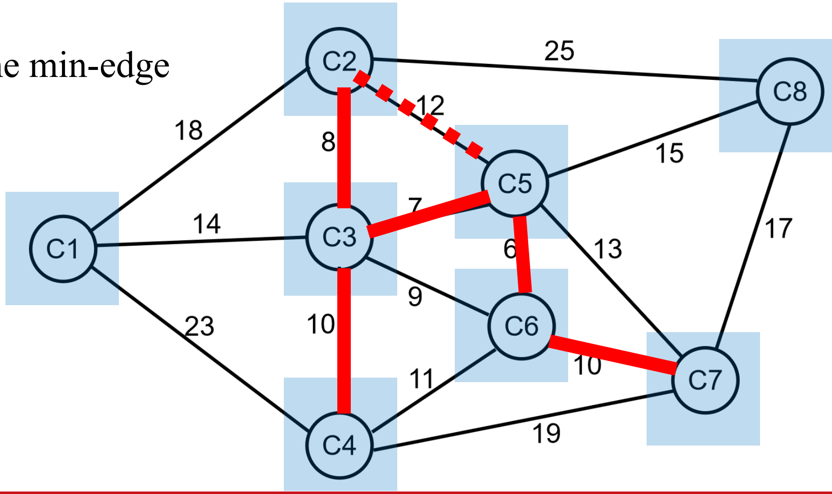
Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm - incrementally incorporate the closest vertex
  - *Kruskal* algorithm
    - organize graph edges via heap
    - incrementally merge sub-graphs via the min-edge

{C1} {C2, C3, C4, C5, C6, C7} {C8}

E = {C5C6,C3C5,C2C3,C6C7,C3C4}

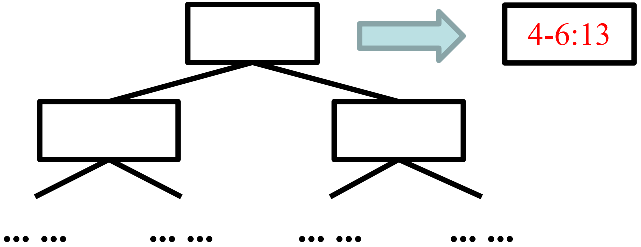


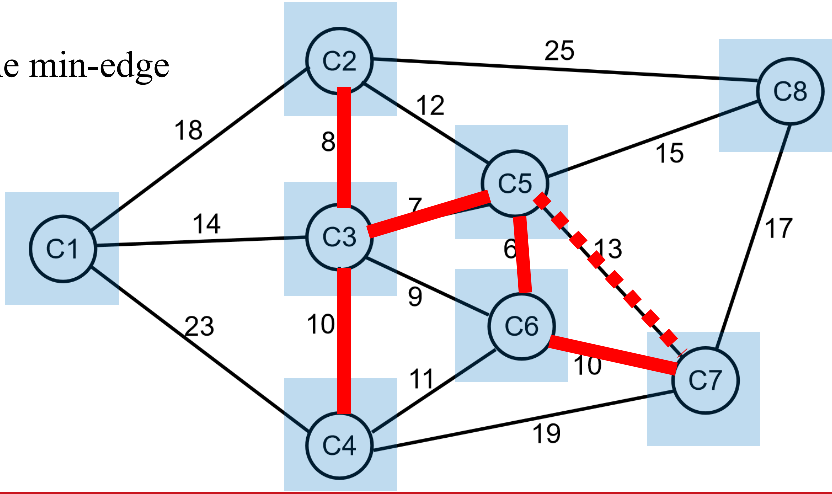


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm - incrementally incorporate the closest vertex
  - *Kruskal* algorithm
    - organize graph edges via heap
    - incrementally merge sub-graphs via the min-edge

{C1} {C2, C3, C4, C5, C6, C7} {C8}
E = {C5C6,C3C5,C2C3,C6C7,C3C4}

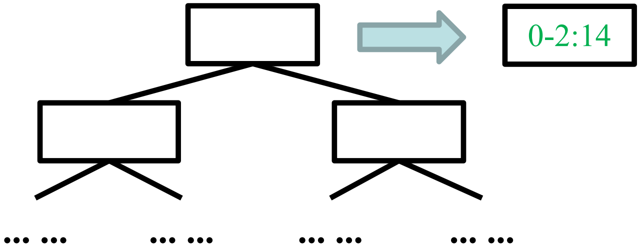


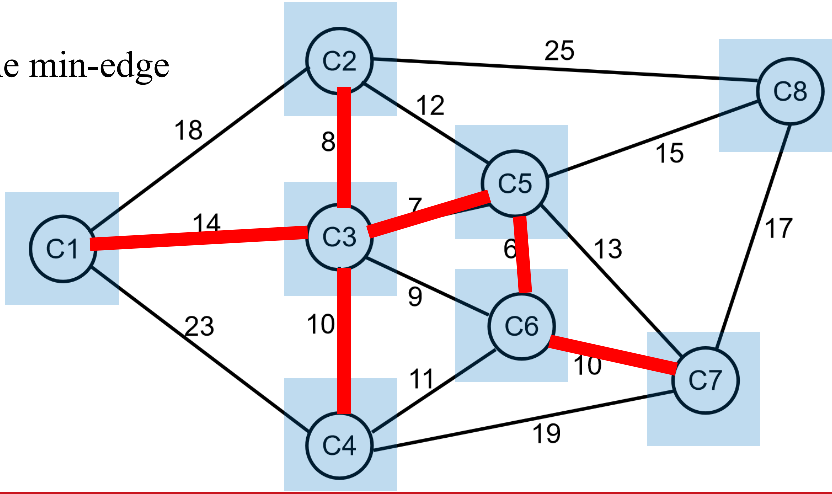


Graph

- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm - incrementally incorporate the closest vertex
  - *Kruskal* algorithm
    - organize graph edges via heap
    - incrementally merge sub-graphs via the min-edge

{C1, C2, C3, C4, C5, C6, C7} {C8}
E = {C5C6,C3C5,C2C3,C6C7,C3C4,C1C3}



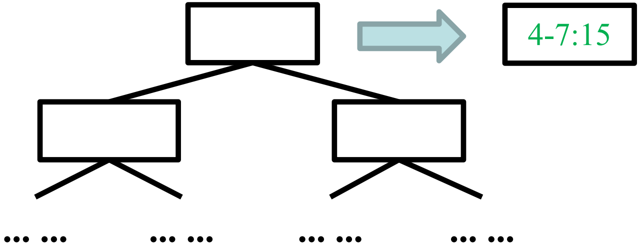


Graph

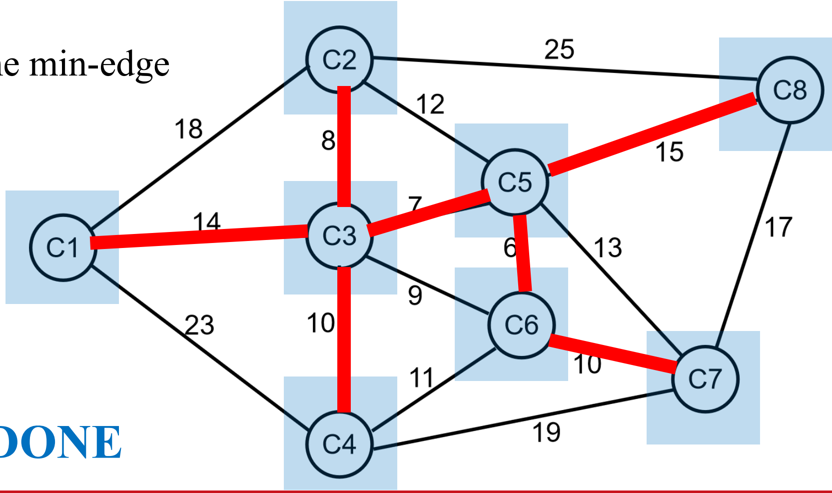
- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm - incrementally incorporate the closest vertex
  - *Kruskal* algorithm
    - organize graph edges via heap
    - incrementally merge sub-graphs via the min-edge

{C1, C2, C3, C4, C5, C6, C7, C8}

E = {C5C6,C3C5,C2C3,C6C7,C3C4,C1C3,C5C8}

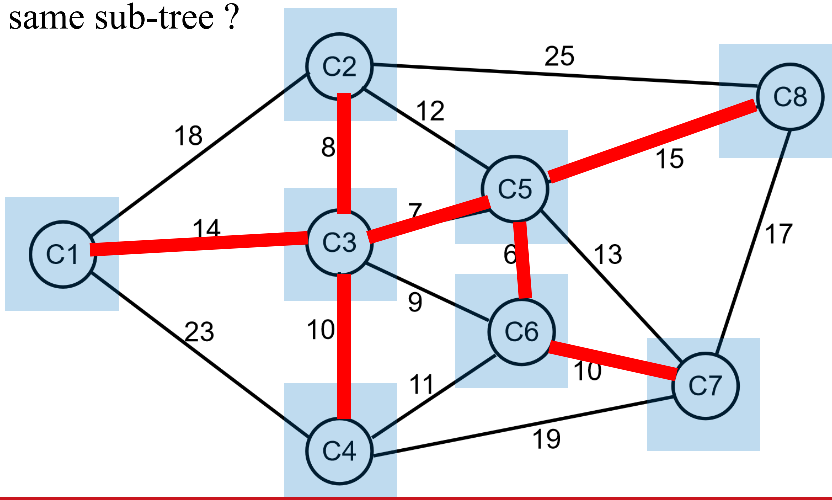


DONE



Graph

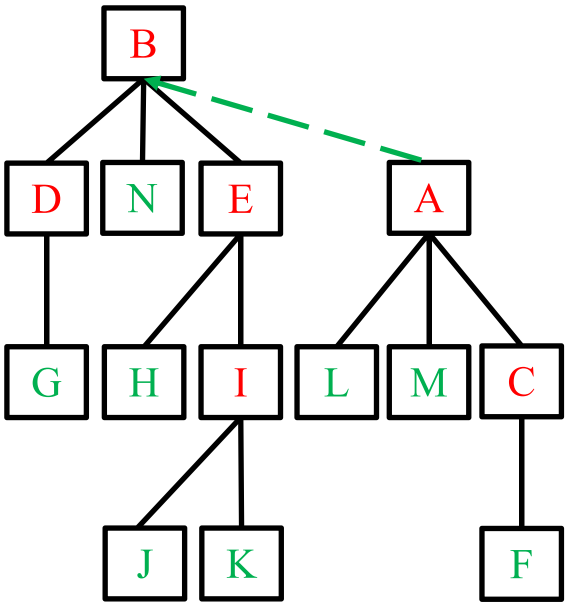
- **Minimum-cost spanning tree (MST)**
  - connected & undirected graph G=(V,E)
  - *connected sub-graph* that has the minimum cost
  - *Prim* algorithm - incrementally incorporate the closest vertex
  - *Kruskal* algorithm - incrementally merge sub-graphs (sub-trees) via the min-edge
    - how to check if two vertices are in the same sub-tree ?



Graph

REVIEW

- **General tree**
  - *parent pointer* implementation



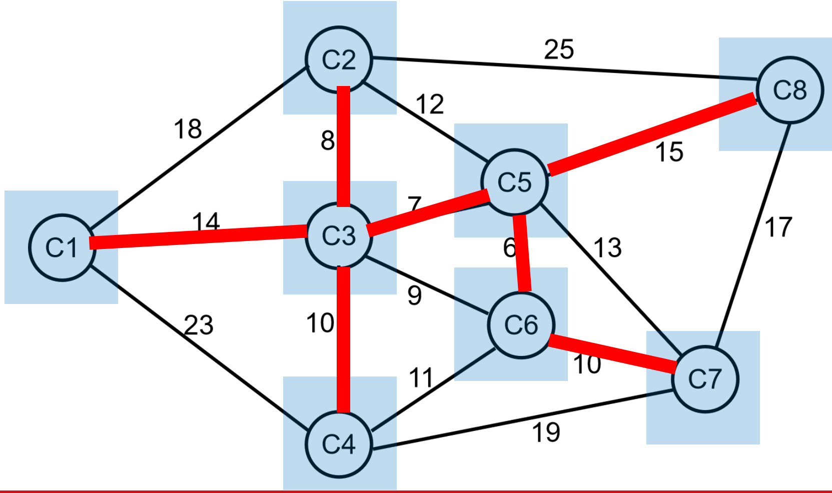

```
template <class T> class PPNode{ // parent pointer GT node abstract class
private:T e;int n; // node's element; number of nodes of the node's tree
        PPNode *p; // node's parent
public: PPNode(){p=NULL;n=1;} ~PPNode(){} PPNode(const T& ei){e=ei;p=NULL;n=1;}
        const T& getE() const{return e;} void setE(const T& ei){e=ei;}
        int getN() const{return n;} void setN(int ni){n=ni;}
        inline PPNode* getP() const{return p;} void setP(PPNode* g){p=g;}
        static PPNode* find(PPNode* g){while(g->getP()!=NULL) g=g->getP();return g;}
        static void ppunion(PPNode* a,PPNode* b){a=find(a);b=find(b);if(a==b) return;
                if(a->getN()<=b->getN()){a->setP(b);b->setN(b->getN()+a->getN());}
                else{b->setP(a);a->setN(a->getN()+b->getN());}}
};
// ostream overloading, so that 'cout<<{PPNode<T> object}' can have meaning
template <typename T> std::ostream& operator<<(std::ostream& out,PPNode<T>* b){
        out<<b->getE()<<':'<<b->getN(); return out;}
```

```
PPNode<char> *p1,*p2;PPNode<char>* pp[26]={new PPNode<char>('A'),
new PPNode<char>('B'), new PPNode<char>('C'), new PPNode<char>('D'), new PPNode<char>('E'), new PPNode<char>('F'),
new PPNode<char>('G'), new PPNode<char>('H'), new PPNode<char>('I'), new PPNode<char>('J'), new PPNode<char>('K'),
new PPNode<char>('L'), new PPNode<char>('M'), new PPNode<char>('N'), new PPNode<char>('O'), new PPNode<char>('P'),
new PPNode<char>('Q'), new PPNode<char>('R'), new PPNode<char>('S'), new PPNode<char>('T'), new PPNode<char>('U'),
new PPNode<char>('V'), new PPNode<char>('W'), new PPNode<char>('X'), new PPNode<char>('Y'), new PPNode<char>('Z')};
while(true){int i=rand()%26,j=rand()%26;p1=PPNode<char>::find(pp[i]);p2=PPNode<char>::find(pp[j]);
        cout<<"merge "<<pp[i]<<" & "<<pp[j]<<" : "<<pp[i]<<" => "<<p1<<" | "<<pp[j]<<" => "<<p2<<" : ";
        if(p1==p2){cout<<p1<<"=="<<p2<<" unnecessary to merge!\n";}
        else{PPNode<char>::ppunion(p1,p2);
                if(p1->getN()<p2->getN()){cout<<" merge into "<<p2<<endl;if(26==p2->getN()) break;}
                else{cout<<" merge into "<<p1<<endl;if(26==p1->getN()) break;}}}
```

Graph

- **Minimum-cost spanning tree (MST)**
  - *Kruskal* algorithm - incrementally merge sub-graphs (sub-trees) via the min-edge

```
Kruskal MST =>
remove min-edge 4-5:6=>vertices 4 & 5 connected (merged)
remove min-edge 2-4:7=>vertices 2 & 4 connected (merged)
remove min-edge 1-2:8=>vertices 1 & 2 connected (merged)
remove min-edge 2-5:9=>vertices 2 & 5 in the same sub-tree
remove min-edge 2-3:10=>vertices 2 & 3 connected (merged)
remove min-edge 5-6:10=>vertices 5 & 6 connected (merged)
remove min-edge 3-5:11=>vertices 3 & 5 in the same sub-tree
remove min-edge 1-4:12=>vertices 1 & 4 in the same sub-tree
remove min-edge 4-6:13=>vertices 4 & 6 in the same sub-tree
remove min-edge 0-2:14=>vertices 0 & 2 connected (merged)
remove min-edge 4-7:15=>vertices 4 & 7 connected (merged)
Kruskal MST =>| 4-5:6 2-4:7 1-2:8 2-3:10 5-6:10 0-2:14 4-7:15
```



THANK YOU

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY