Introduction to Algorithms

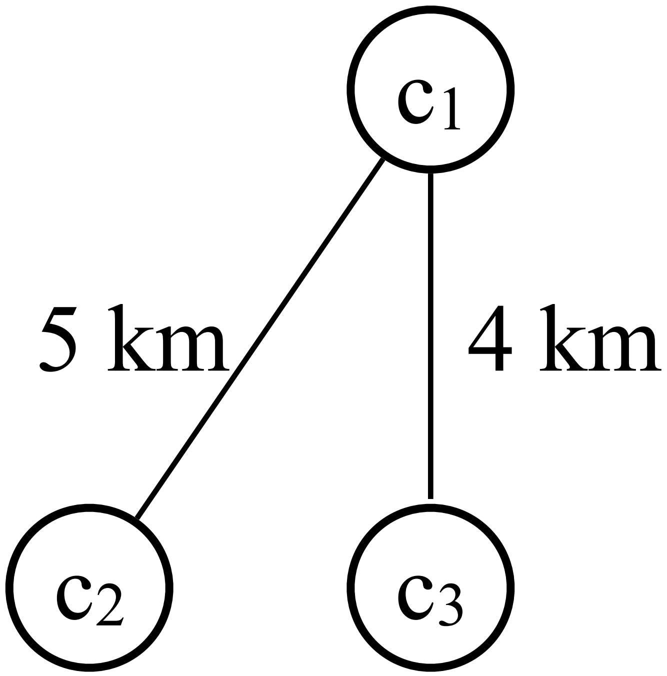
Meng-Tsung Tsai

11/26/2019

Minimum Spanning Trees

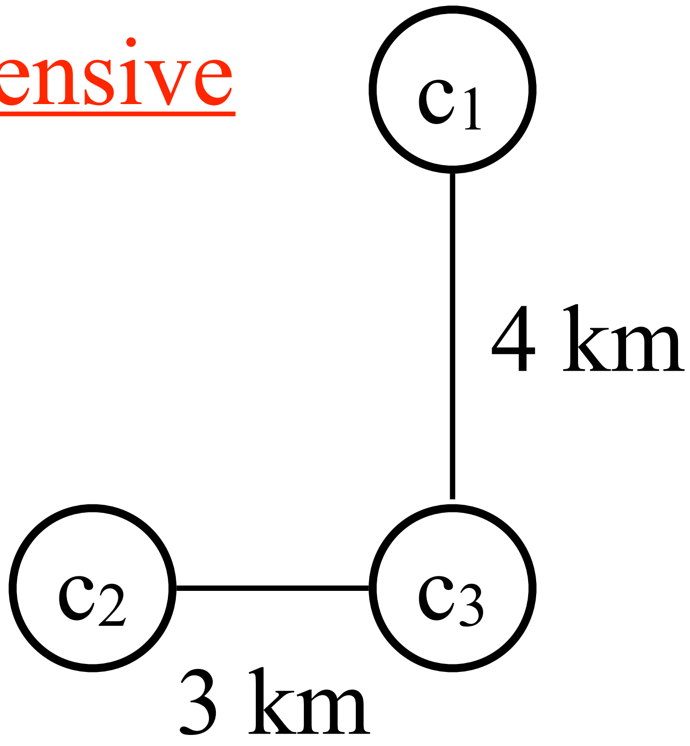
Motivation

We plan to connect n cities by railway routes under the limited budget. The total cost is proportional to the total length of the built railway routes.



cost: 9 units

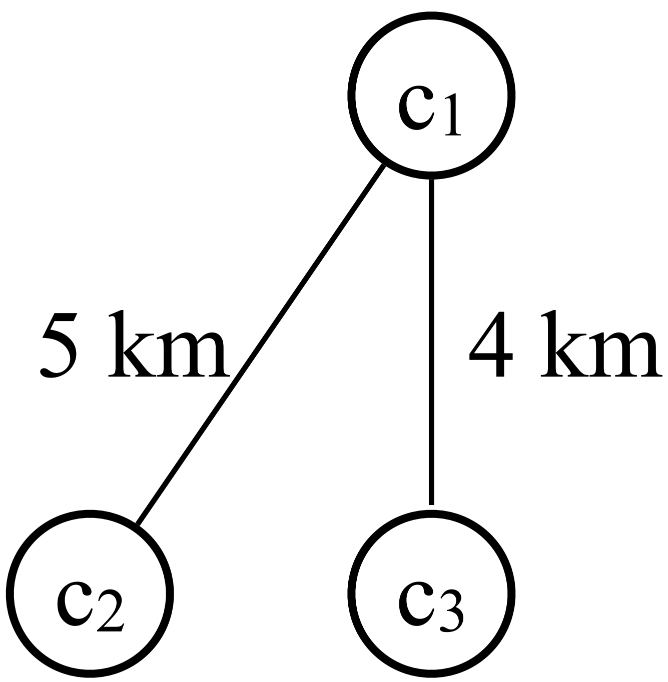
less expensive



cost: 7 units

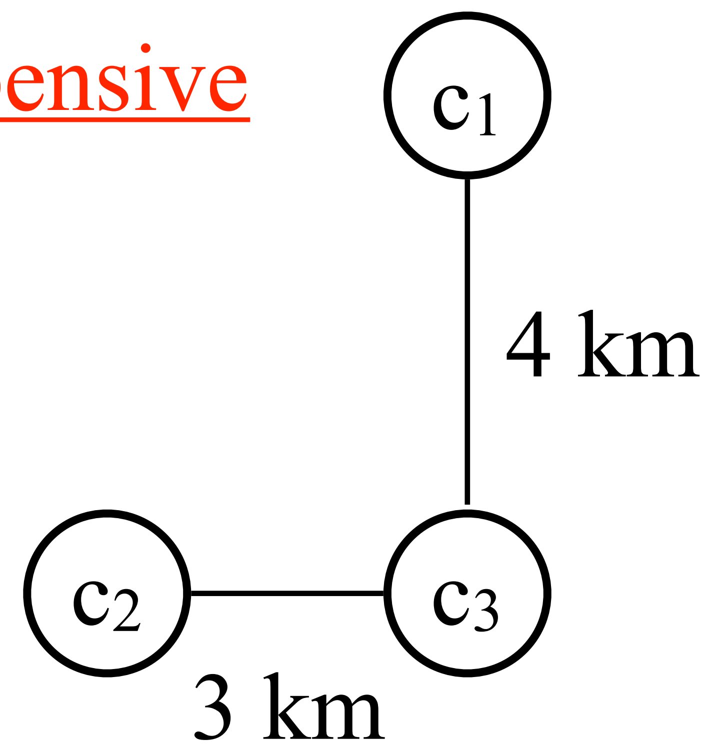
Motivation

We plan to connect n cities by railway routes under the limited budget. The total cost is proportional to the total length of the built railway routes.



cost: 9 units

less expensive



cost: 7 units

If n is small, it is okay to enumerate all possibilites and pick the least expensive way. For large n, exhaustive searching is too slow.

Describing the problem in the language of graph theory (first attempt)

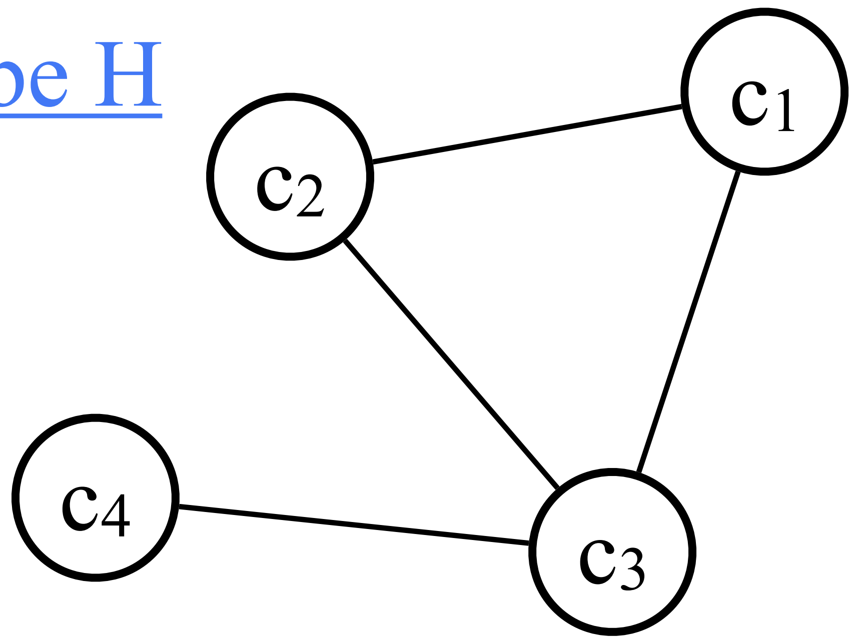
Given a graph G in which each edge $e \in G$ has a weight $w(e) > 0$, find a subgraph H of G so that all nodes in G are connected and $w(H) = \sum_{e \in H} w(e)$ is minimized.

H has no cycle.

Describing the problem in the language of graph theory (first attempt)

Given a graph G in which each edge $e \in G$ has a weight $w(e) > 0$, find a subgraph H of G so that all nodes in G are connected and $w(H) = \sum_{e \in H} w(e)$ is minimized.

cannot be H



Removing an edge from a cycle in H cannot disconnect H, but reduces $w(H)$ if $w(e) > 0$ for all e.

H has no cycle.

Describing the problem in the language of graph theory (first attempt)

Given a graph G in which each edge $e \in G$ has a weight $w(e) > 0$, find a subgraph H of G so that all nodes in G are connected and $w(H) = \sum_{e \in H} w(e)$ is minimized.

(1) H must be a tree if $w(e) > 0$ for all e.
(2) + H spans all the nodes in G. $\Rightarrow$ H is called a spanning tree.
(3) + w(H) is minimized. $\Rightarrow$ H is called the minimum spanning tree.

Describing the problem in the language of graph theory

Given a graph G in which each edge $e \in G$ has a weight $w(e) \in R$, find an acyclic subgraph H of G so that all nodes in G are connected and $w(H) = \sum_{e \in H} w(e)$ is minimized; that is, the minimum spanning tree of G.

To generalize the problem to cover the case of $w(e) \leq 0$, we need to require H acyclic. This doesn't change anything for the case of $w(e) > 0$, but forces the case of $w(e) \leq 0$ to return a tree.

Describing the problem in the language of graph theory

Given a graph G in which each edge $e \in G$ has a weight $w(e) \in R$, find an acyclic subgraph H of G so that all nodes in G are connected and $w(H) = \sum_{e \in H} w(e)$ is minimized; that is, the minimum spanning tree of G.

To generalize the problem to cover the case of $w(e) \leq 0$, we need to require H acyclic. This doesn't change anything for the case of $w(e) > 0$, but forces the case of $w(e) \leq 0$ to return a tree.

Note that if $w(e)$ can be ≤ 0, the minimum conncected spanning subgraph may be not a tree. However, in this case, we still output the minimum spanning tree.

Why do we need to translate an encountered problem into the language of graph theory?

The language of graph theory is like a dictionary. To find whether there are existing solutions to a problem P, we can translate P into a graph problem (a canonical representation) and google it.

A Generic Algorithm Based on the Cut Property

A generic algorithm

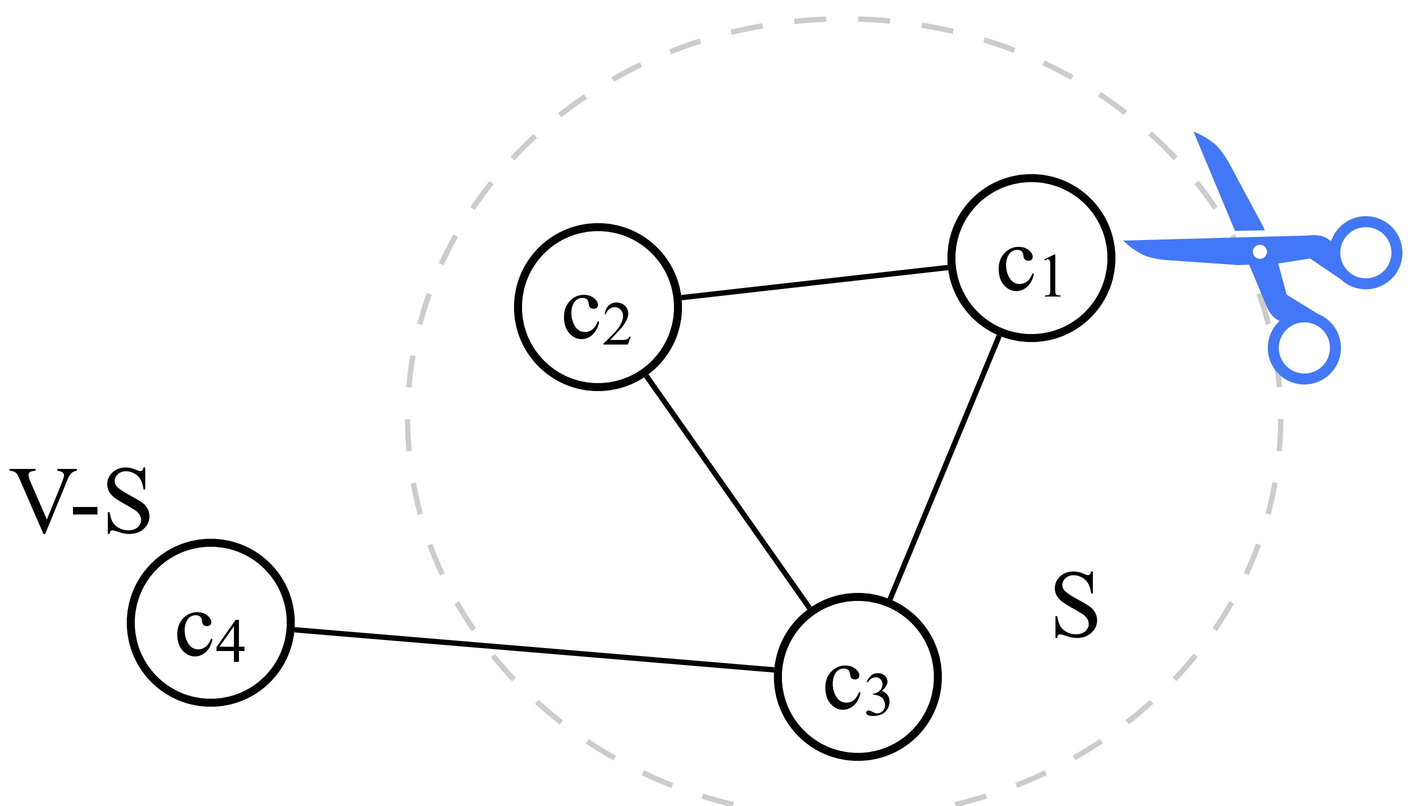
Generic-MST(G, w){ // G is a connected undirected graph
    A ← ∅;
    while(A does not form a spanning tree){
        find "an" edge (u, v) safe for A; // we say an edge e is safe for an edge set A if A ∪ {e} is a subset of some MST
        A ← A ∪ {(u, v)};
    }
}

The edge found at each iteration varies among algorithms. How many iterations are in the while-loop?

The cut property

<u>Claim</u>. Let A be a subset of the edges in some MST of G. Let (S, V-S) be any cut that respects A, and let (u, v) be a light edge crossing the cut. Then, (u, v) is safe for A.

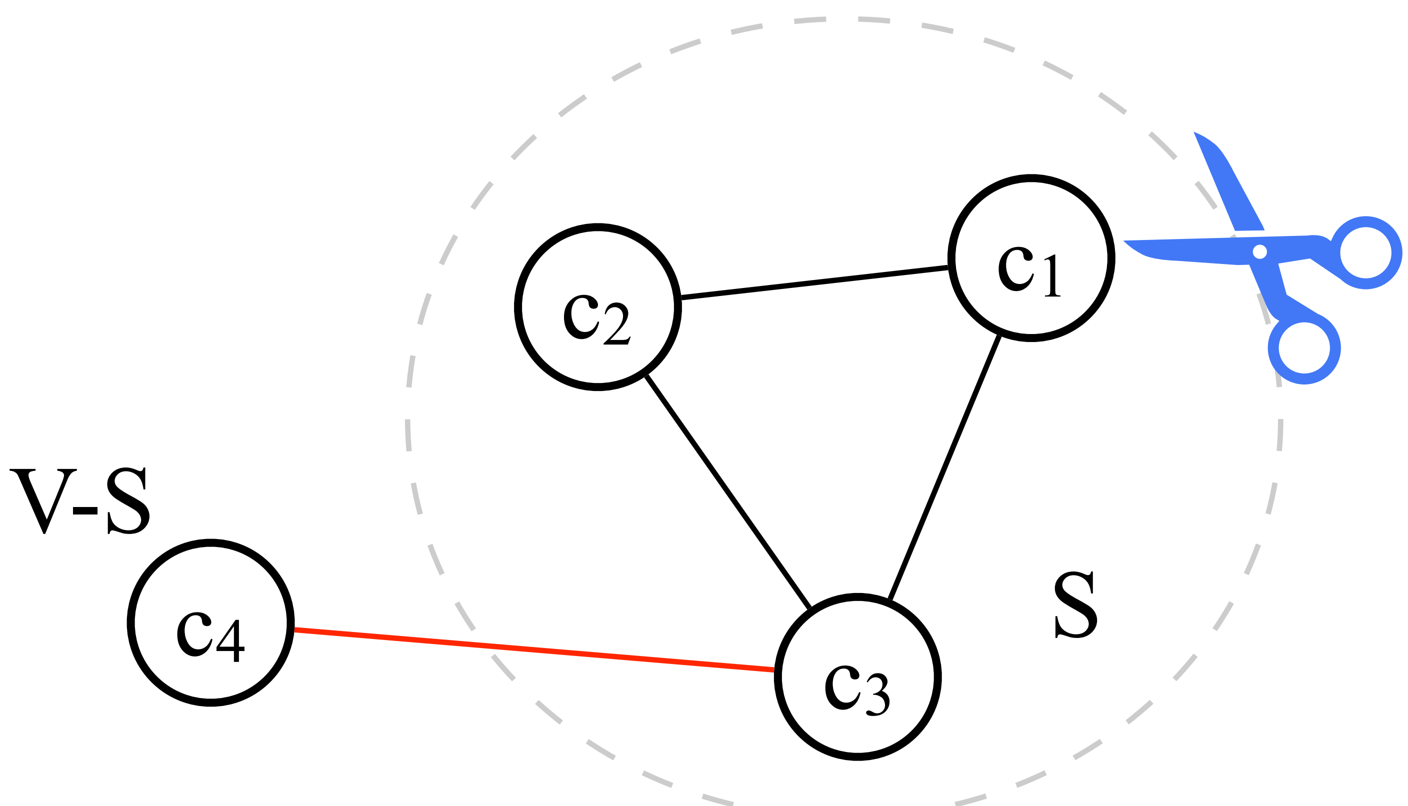
A cut is a partition of nodes into two subsets S and V-S.



The cut property

Claim. Let A be a subset of the edge set of some MST of G. Let (S, V-S) be any cut that respects A, and let (u, v) be a light edge crossing the cut. Then, (u, v) is safe for A.

An edge (u, v) that crosses cut (S, V-S) means exactly one of u, v is in S.

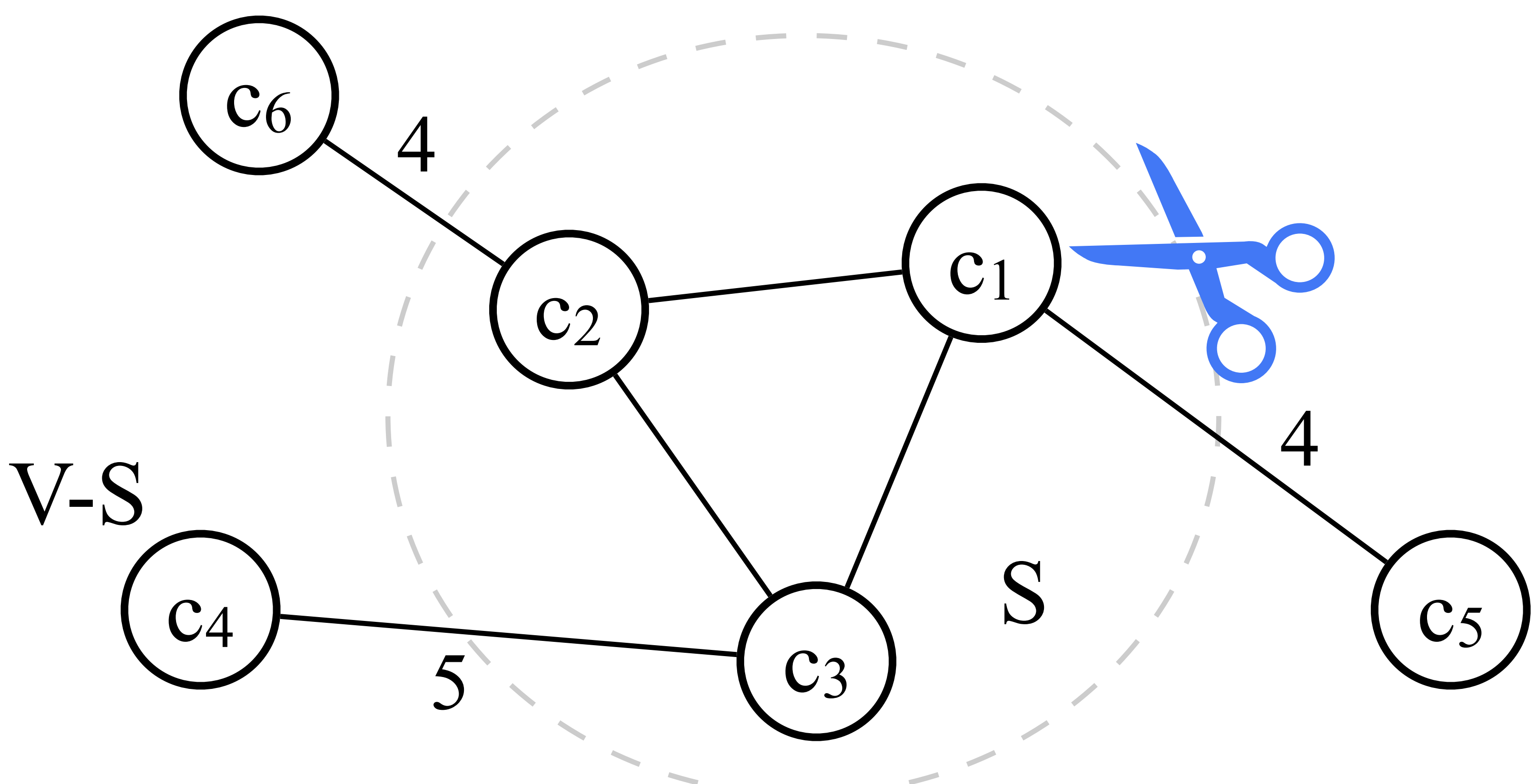


(c_3, c_4) is an edge crossing (S, V-S).

The cut property

<u>Claim</u>. Let A be a subset of the edge set of some MST of G. Let (S, V-S) be any cut that respects A, and let (u, v) be a light edge crossing the cut. Then, (u, v) is safe for A.

A light edge of a cut is one of the edges that cross the cut and have the minimum weight.

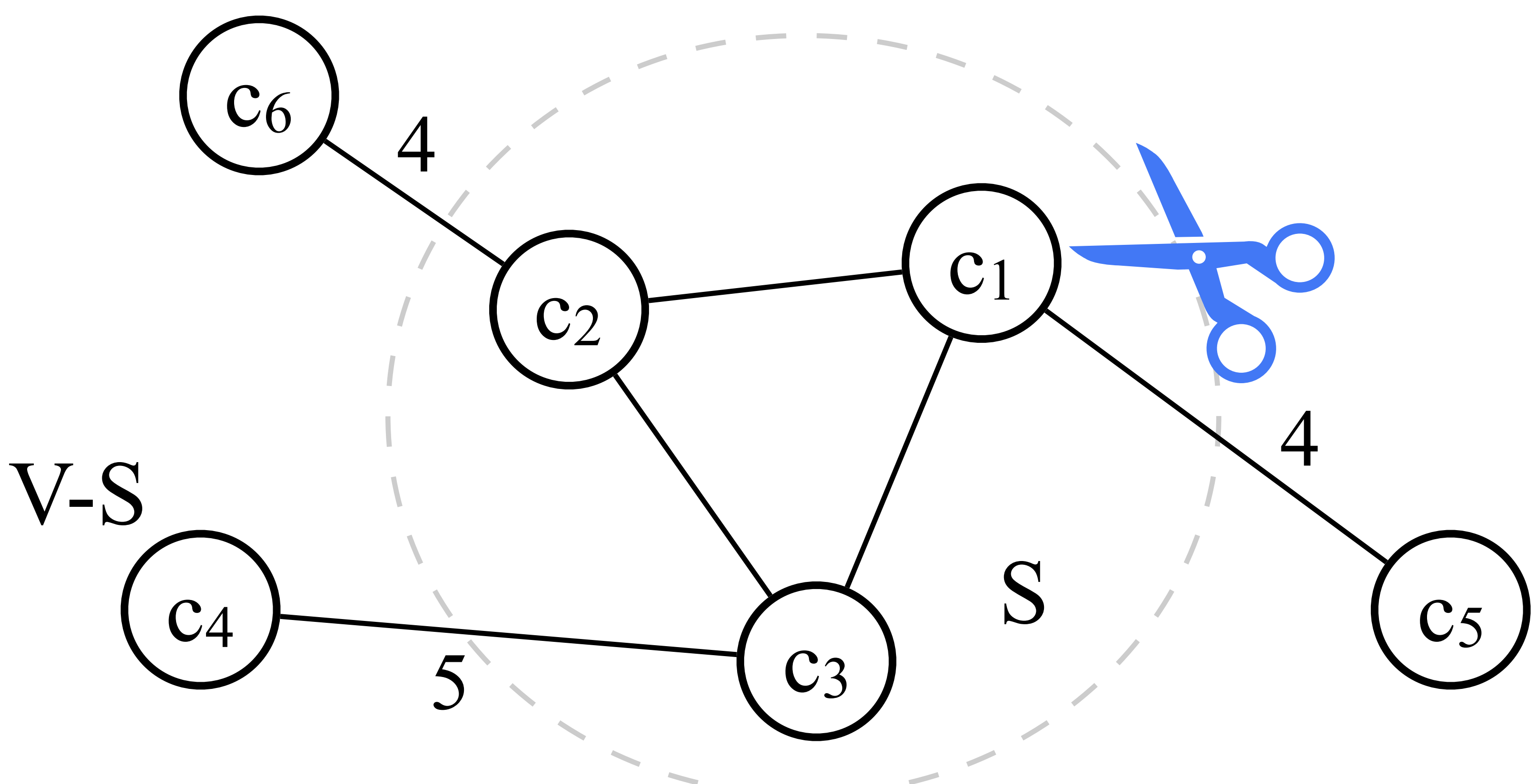


(c_1, c_5) and (c_2, c_6) are the light edges that cross the cut (S, V-S).

The cut property

Claim. Let A be a subset of the edge set of some MST of G. Let (S, V-S) be any cut that respects A, and let (u, v) be a light edge crossing the cut. Then, (u, v) is safe for A.

We say a cut respects an edge set A if A has no edge that crosses the cut.

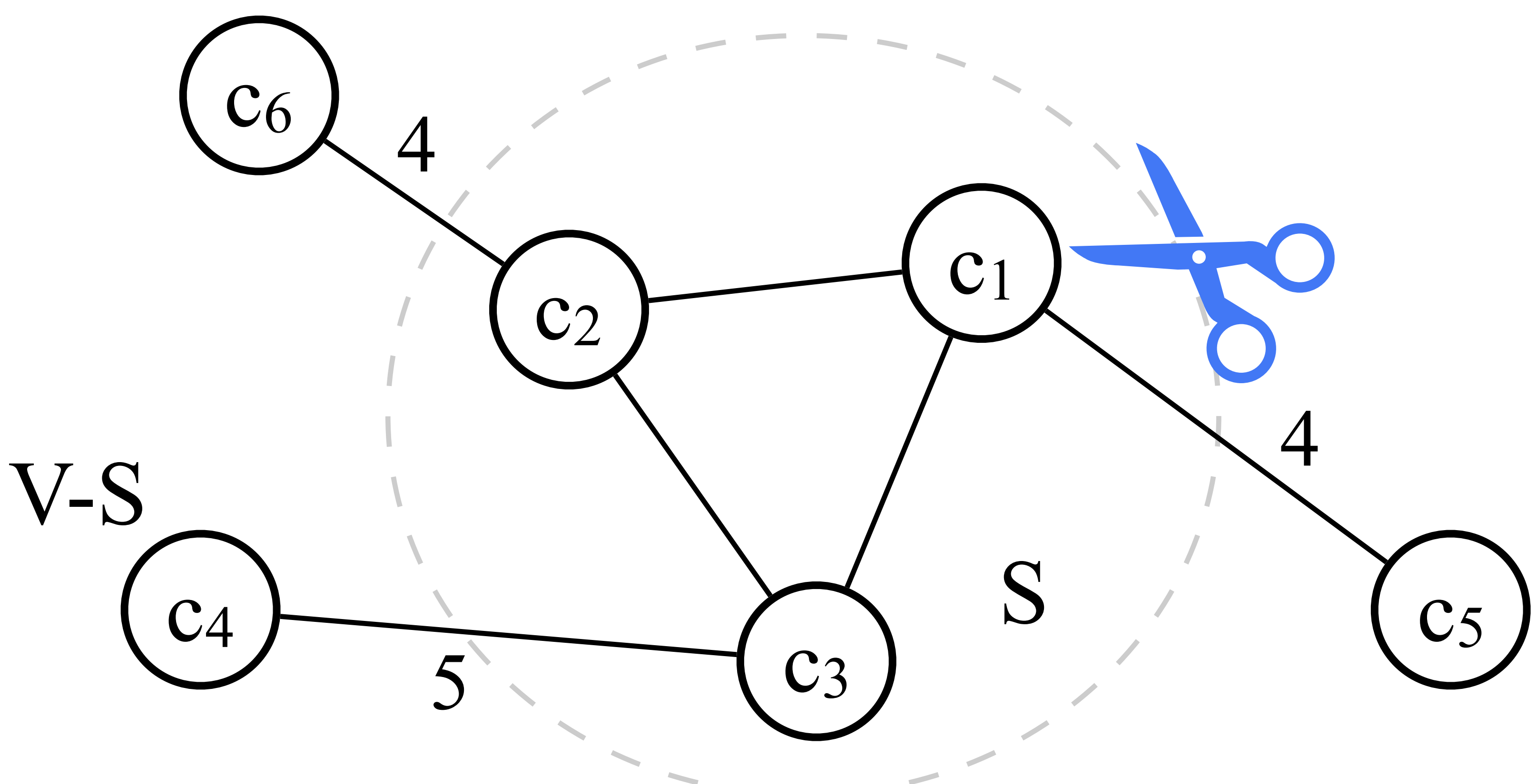


Let $A = \{(c_1, c_2), (c_2, c_3), (c_1, c_3)\}$. The cut (S, V-S) respects A.

The cut property

<u>Claim</u>. Let A be a subset of the edge set of some MST of G. Let (S, V-S) be any cut that respects A, and let (u, v) be a light edge crossing the cut. Then, (u, v) is safe for A.

We say a cut respects an edge set A if A has no edge that crosses the cut.

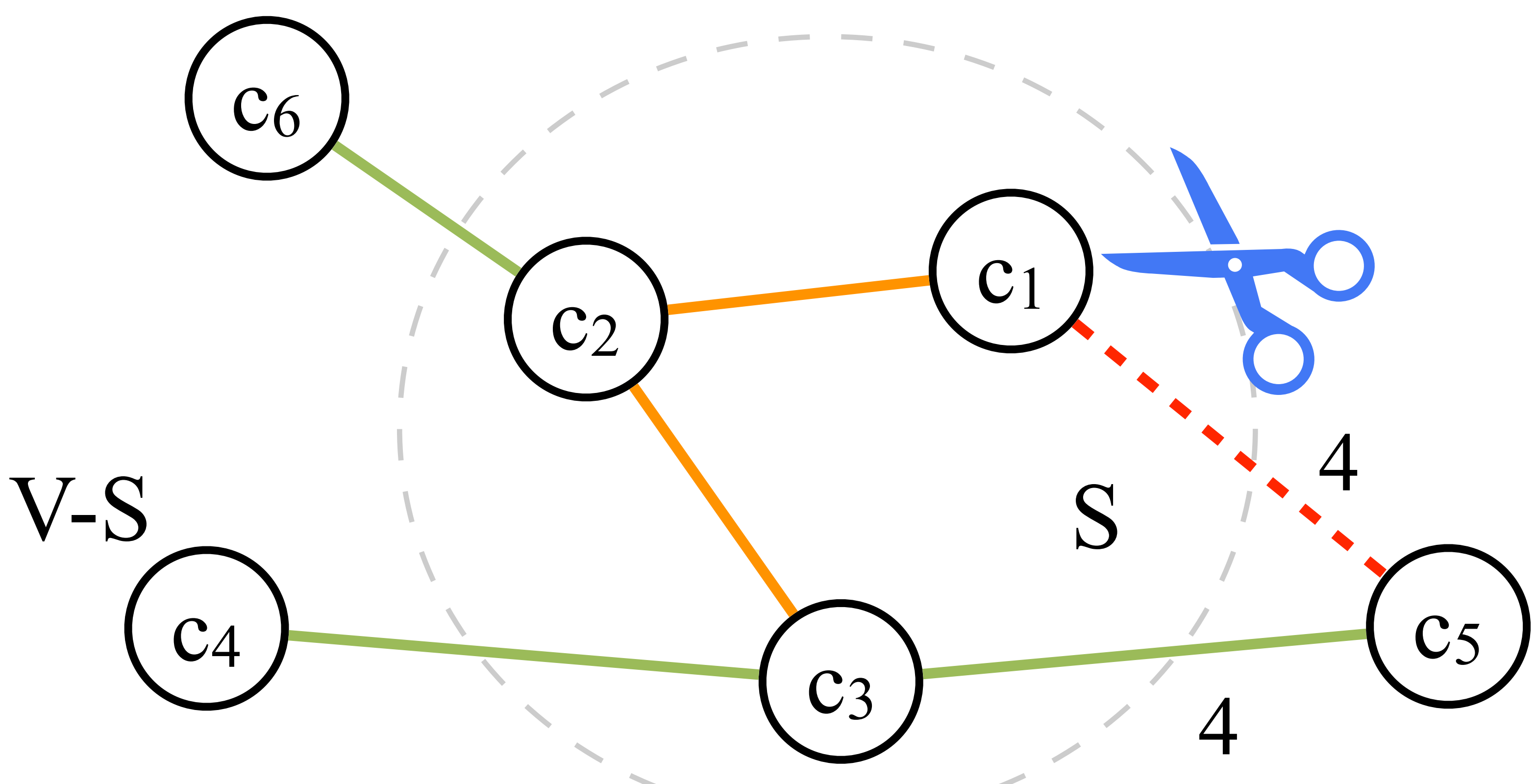


Let $A = \{(c_2, c_6)\}$. The cut (S, V-S) does not respect A.

The cut property

Claim. Let A be a subset of the edge set of some MST of G. Let (S, V-S) be any cut that respects A, and let (u, v) be a light edge crossing the cut. Then, (u, v) is safe for A.

Let orange edges be A, and let the union of green and orange edges be a MST T that contains A.

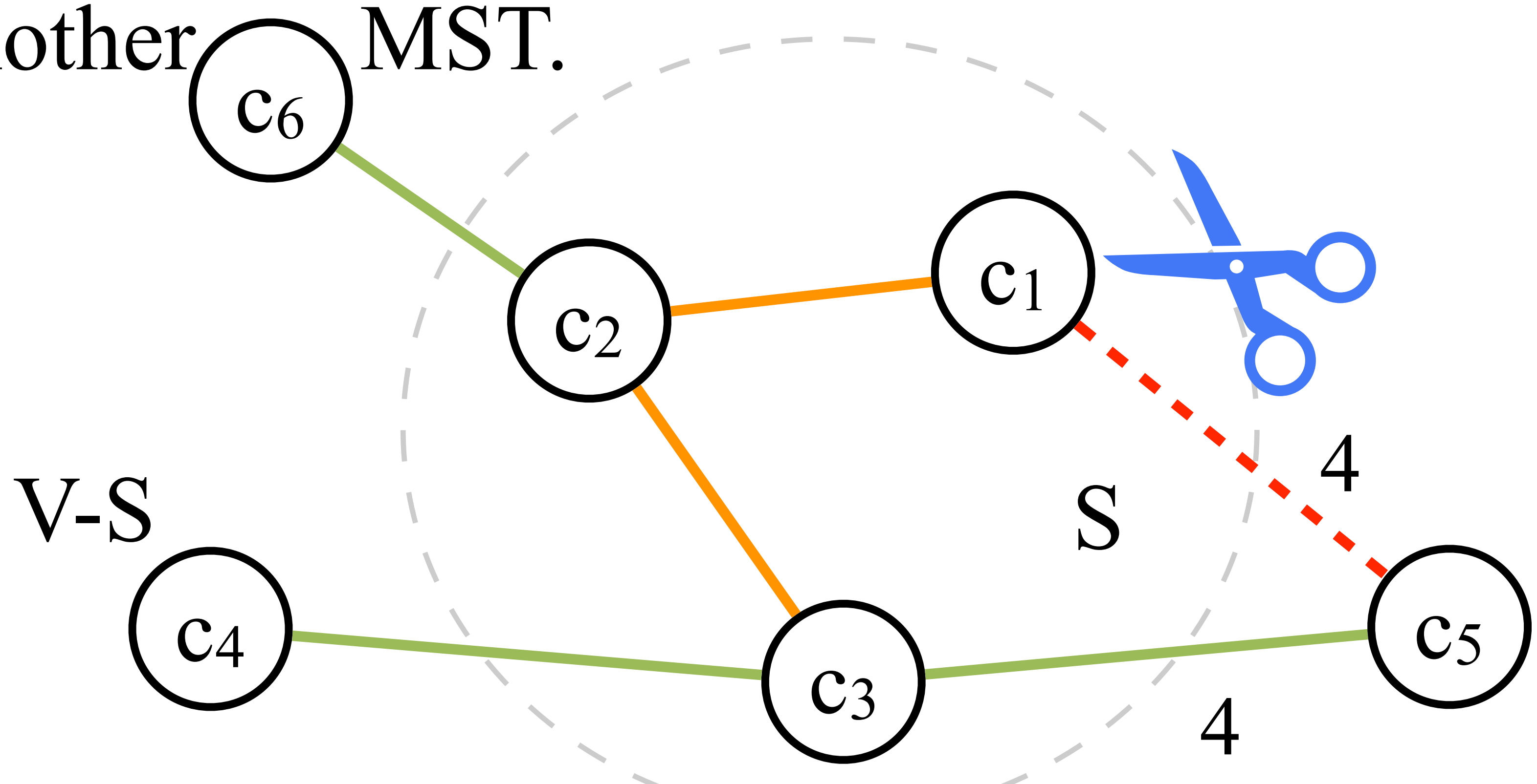


$T \cup \{(c_1, c_5)\}$ has a cycle C crossing the cut ≥ 2 times.

The cut property

<u>Claim</u>. Let A be a subset of the edge set of some MST of G. Let (S, V-S) be any cut that respects A, and let (u, v) be a light edge crossing the cut. Then, (u, v) is safe for A.

In this example, (c_3, c_5) and (c_1, c_5) are the two edges crossing the cut. One can replace (c_3, c_5) with (c_1, c_5) and get annother MST.



The light edge of the cut that respects A is safe for A.

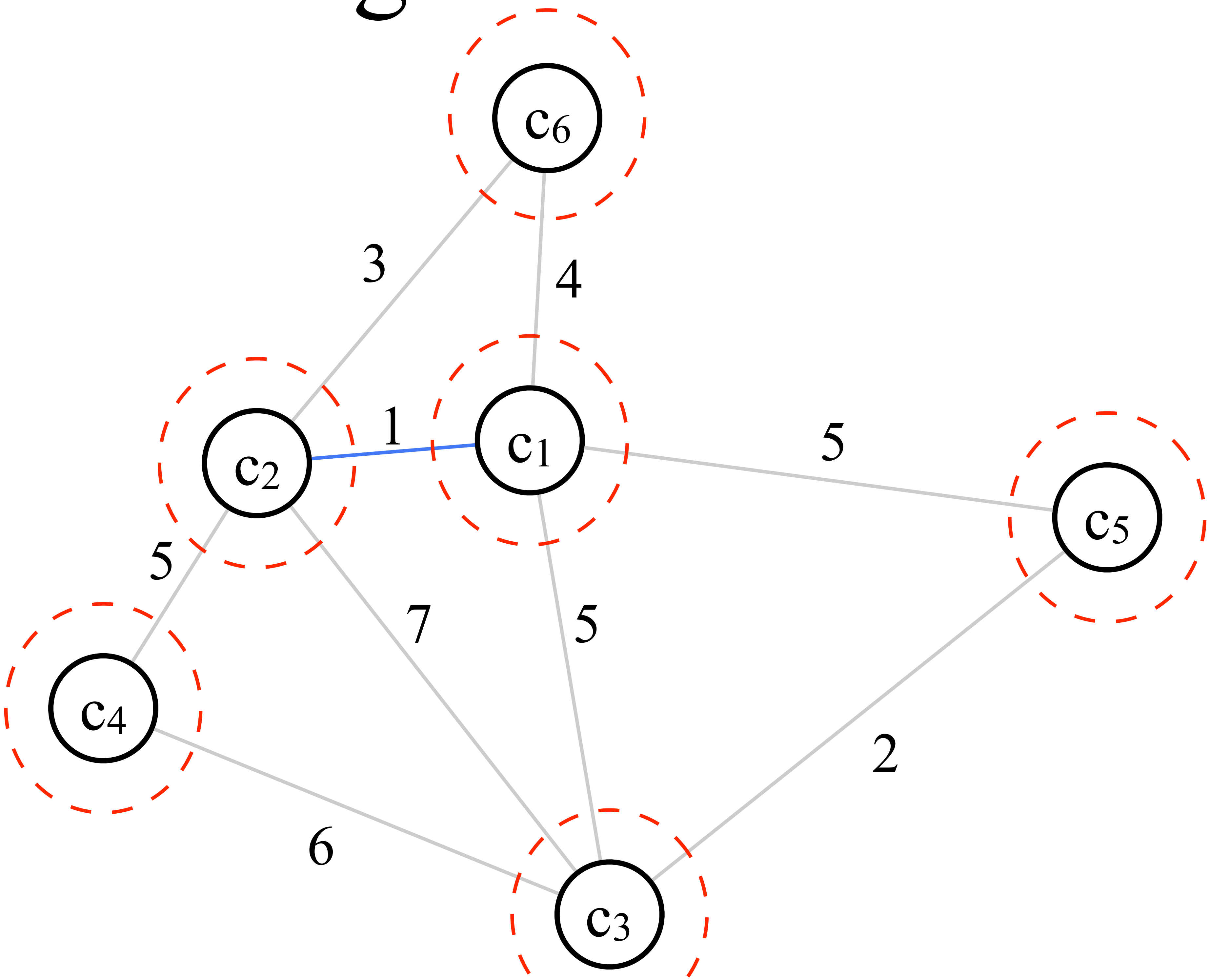
Kruskal's and Prim's Algorithms

Comparison

Algorithms	Kruskal's with disjoint sets	Prim's with binary heap	Prim's with Fibonacci heap
Running Time	O(m log n)	O(m log n)	O(m + n log n)
Difficulty to Implement	Easy	Okay	Hard

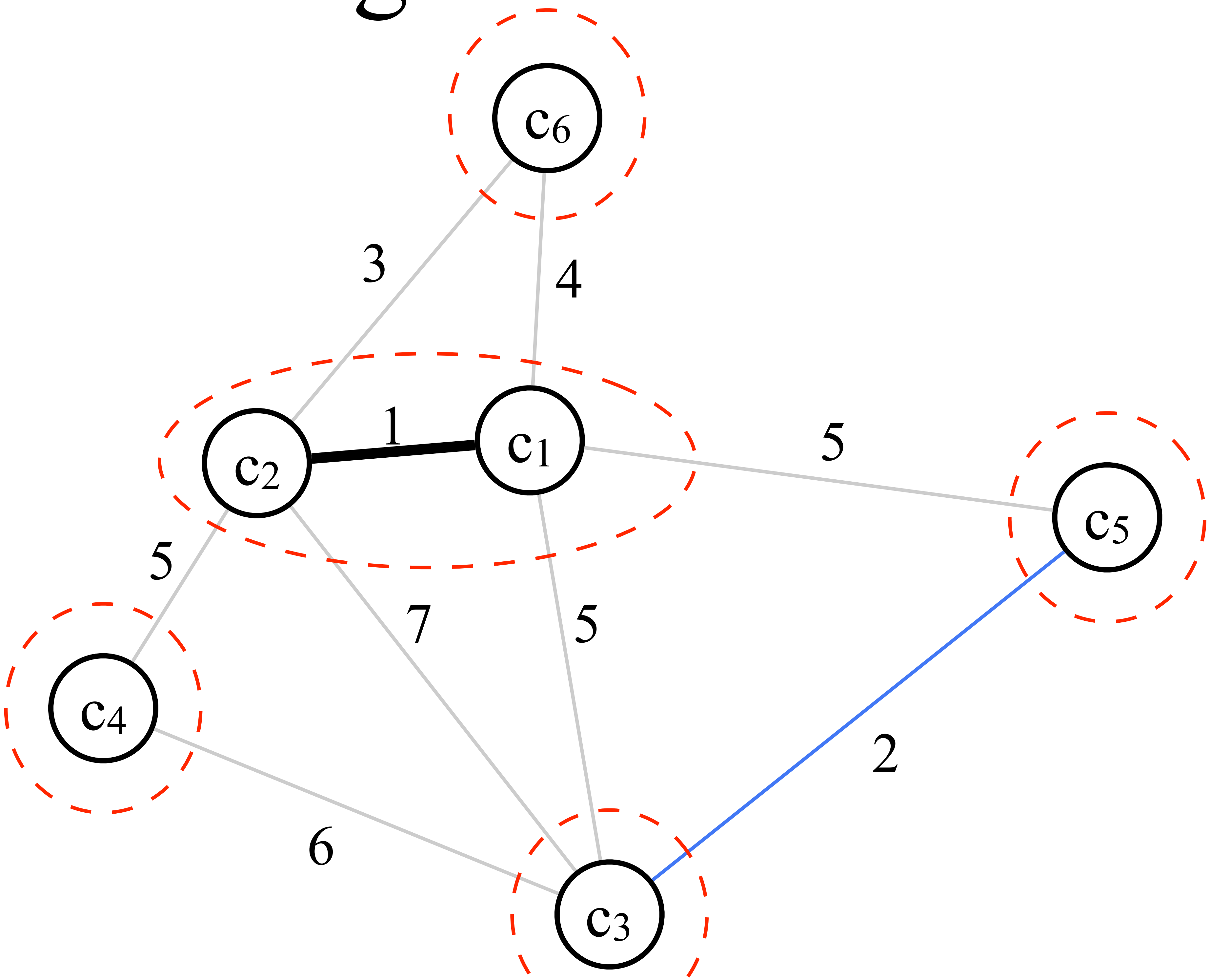
If $n \ll m$ (dense graphs), then Prim's algorithm with Fibonacci heap has the best time complexity.

Kruskal's algorithm



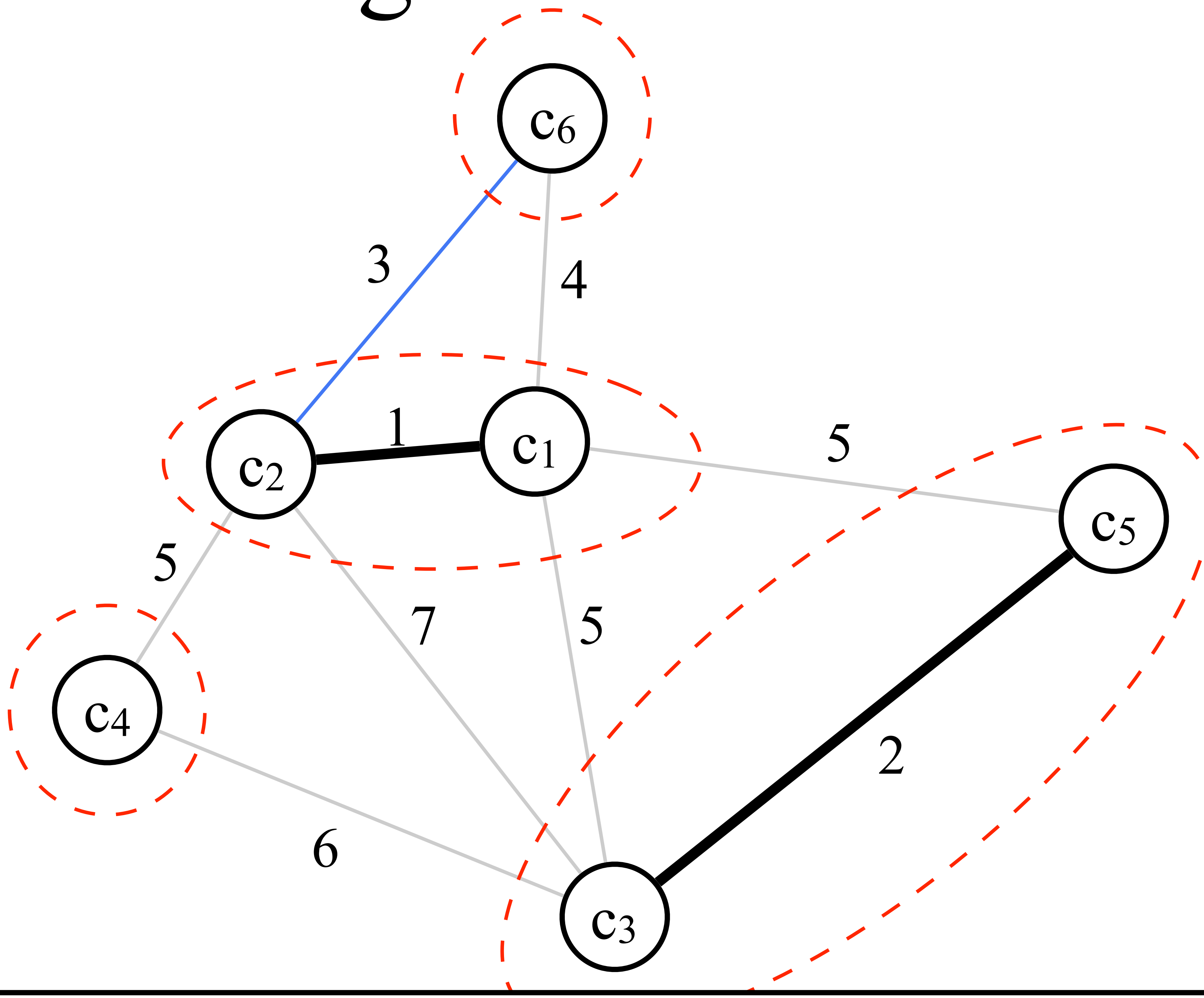
Initially, let each node be a connected component. At each step, pick the lightest edge e among all pairs of connected components. Add e into A.

Kruskal's algorithm



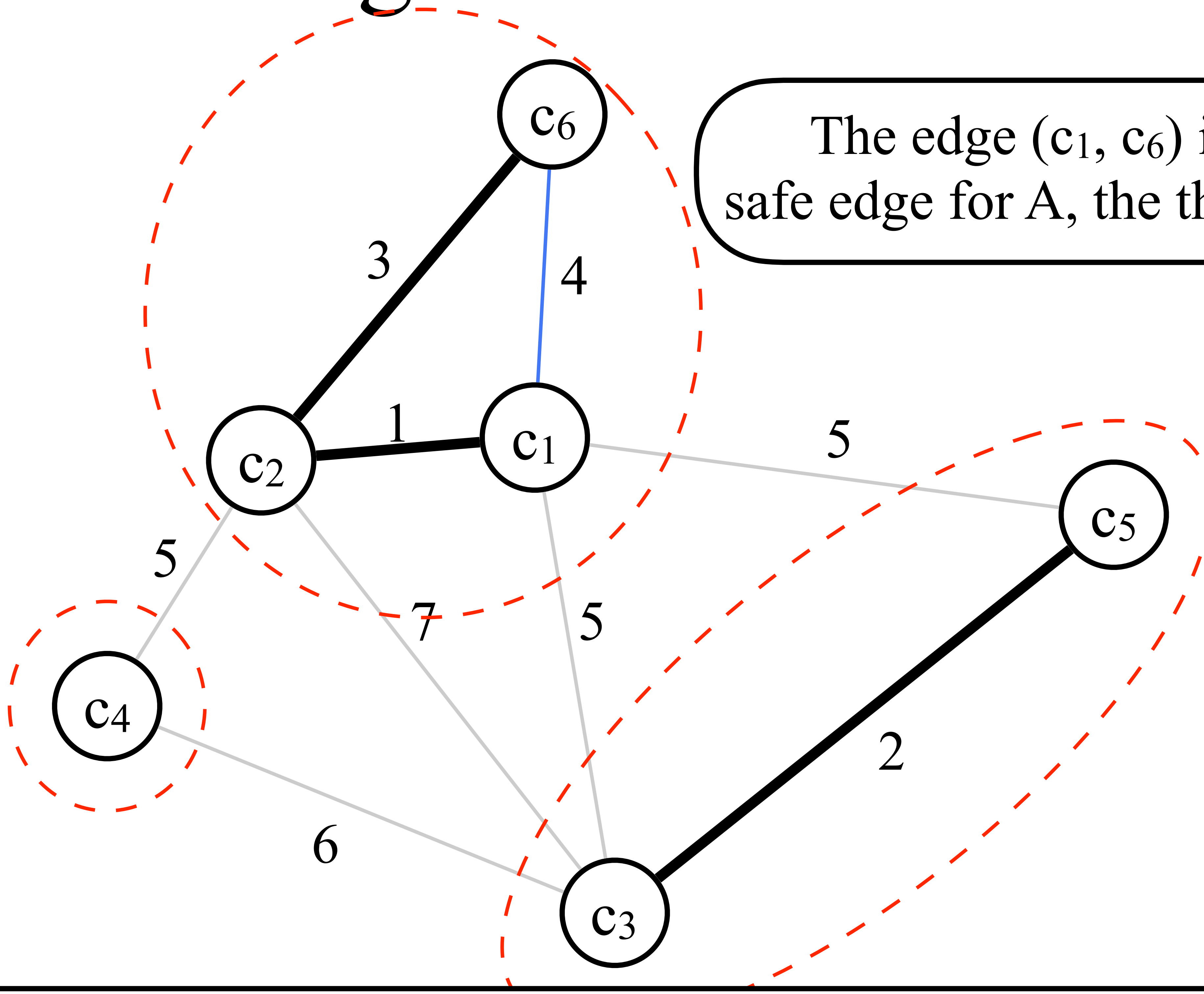
Initially, let each node be a connected component. At each step, pick the lightest edge e among all pairs of connected components. Add e into A.

Kruskal's algorithm



Initially, let each node be a connected component. At each step, pick the lightest edge e among all pairs of connected components. Add e into A.

Kruskal's algorithm



The edge (c_1, c_6) is not a safe edge for A, the thick edges.

Initially, let each node be a connected component. At each step, pick the lightest edge e among all pairs of connected components. Add e into A.

Kruskal's algorithm

```
MST-Kruskal(G, w){
    A ← ∅;
    foreach node v in G {
        Make-set(v); // each node forms a singleton set
    }
    sort edges in G in the non-decreasing order by weight w
    for i = 1 to m {
        let (u, v) ← e_i; // the edge of the i-th smallest weight
        if(Find-set(u) ≠ Find-set(v)){ // (u, v) is a light edge
        crossing cut (T_u, V-T_u) where T_u is the tree containing u
            A ← A ∪ {(u, v)};
            Union(u, v);
        }
    }
}
```

Kruskal's algorithm

```
MST-Kruskal(G, w){
  A ← ∅;
  foreach node v in G {
    Make-set(v); // each node forms a singleton set
  }
  sort edges in G in the non-decreasing order by weight w
  for i = 1 to m {
    let (u, v) ← e_i; // the edge of the i-th smallest weight
    if(Find-set(u) ≠ Find-set(v)){ // (u, v) is a light edge
    crossing cut (T_u, V-T_u) where T_u is the tree containing u
      A ← A ∪ {(u, v)};
      Union(u, v);
    }
  }
}
```

Clearly, (u, v) is a crossing edge of cut (T_u, V-T_u). If it is not a light edge, say e_s is the light edge, then why not add e_s into A upon e_s is visited? →←

Kruskal's algorithm

```
MST-Kruskal(G, w){ // G is an connected undirected graph
  A ← ∅;
  foreach node v in G {
    Make-set(v); // each node forms a singleton set
  }
  sort edges in G in the non-decreasing order by weight w
  for i = 1 to m {
    let (u, v) ← e_i; // the edge of the i-th smallest weight
    if(Find-set(u) ≠ Find-set(v)){ // (u, v) is a light edge
    crossing cut (T_u, V-T_u) where T_u is the tree containing u
      A ← A ∪ {(u, v)};
      Union(u, v);
    }
  }
}
```

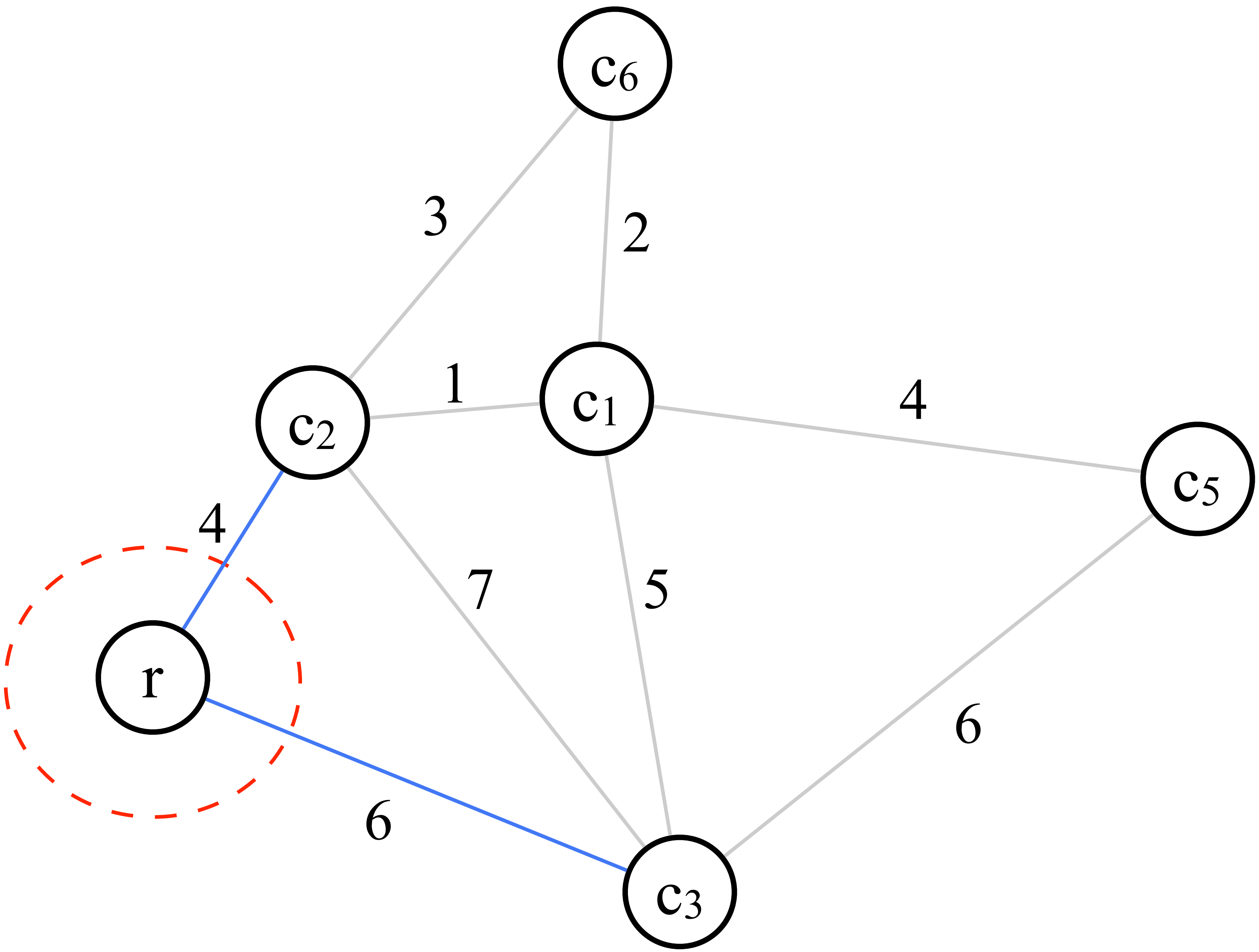
$O(m+n) = O(m)$ disjoint-set operations need $O(m\alpha(n))$ time.
Sorting needs $O(m \log m) = O(m \log(n))$ time.
Since $\alpha(n) = o(\log(n))$, K's algorithm runs in $O(m \log(n))$ time.

Kruskal's algorithm

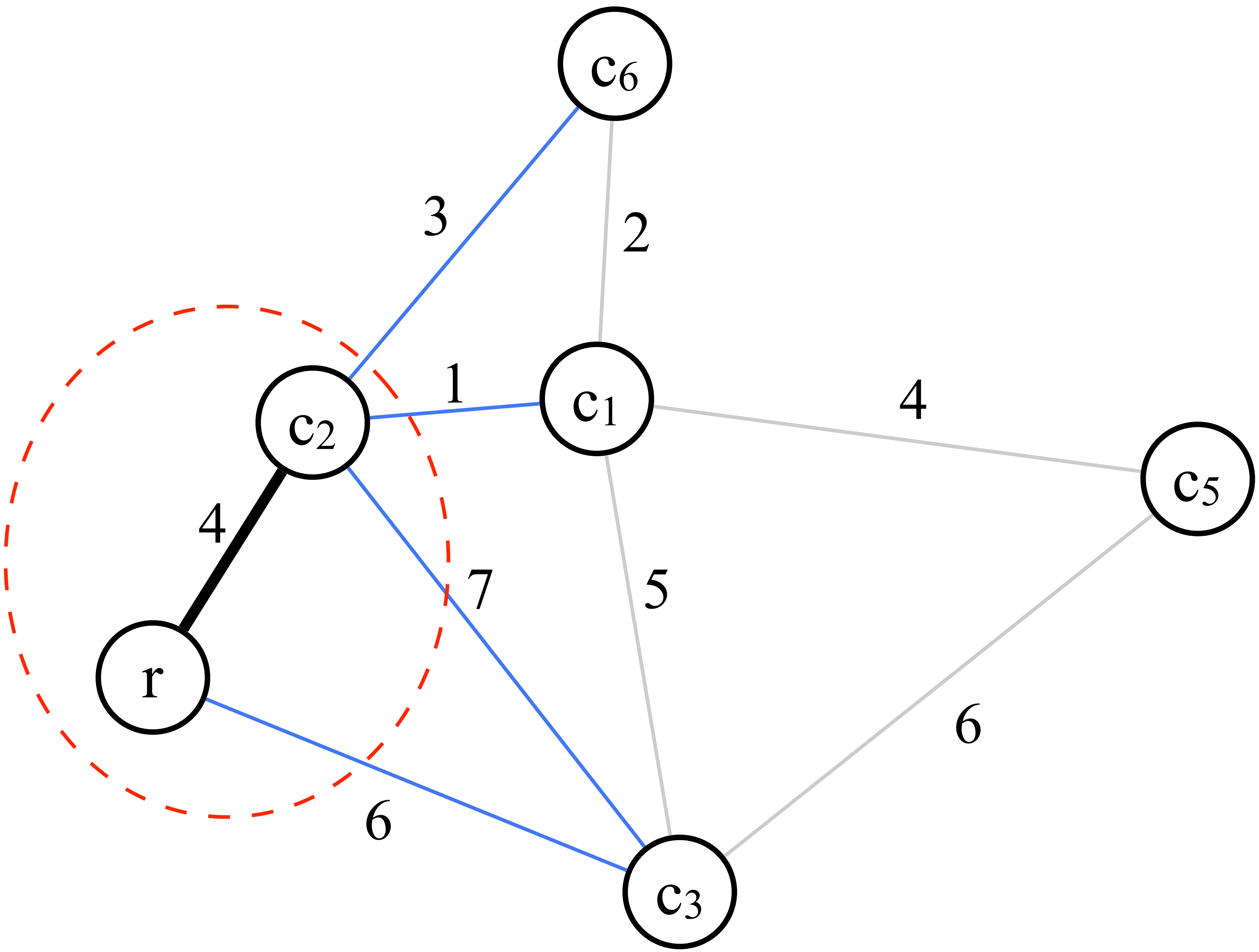
```
MST-Kruskal(G, w){ // G is an connected undirected graph
   A ← ∅;
   foreach node v in G {
      Make-set(v); // each node forms a singleton set
   }
   sort edges in G in the non-decreasing order by weight w
   for i = 1 to m {
      let (u, v) ← e_i; // the edge of the i-th smallest weight
   }
   if(Find-set(u) ≠ Find-set(v)){ // (u, v) is a light edge
crossing cut (T_u, V-T_u) where T_u is the tree containing u
      A ← A ∪ {(u, v)};
      Union(u, v);
   }
}
```

K's algorithm is an implementation of the greedy algorithm for the graphic matroid.

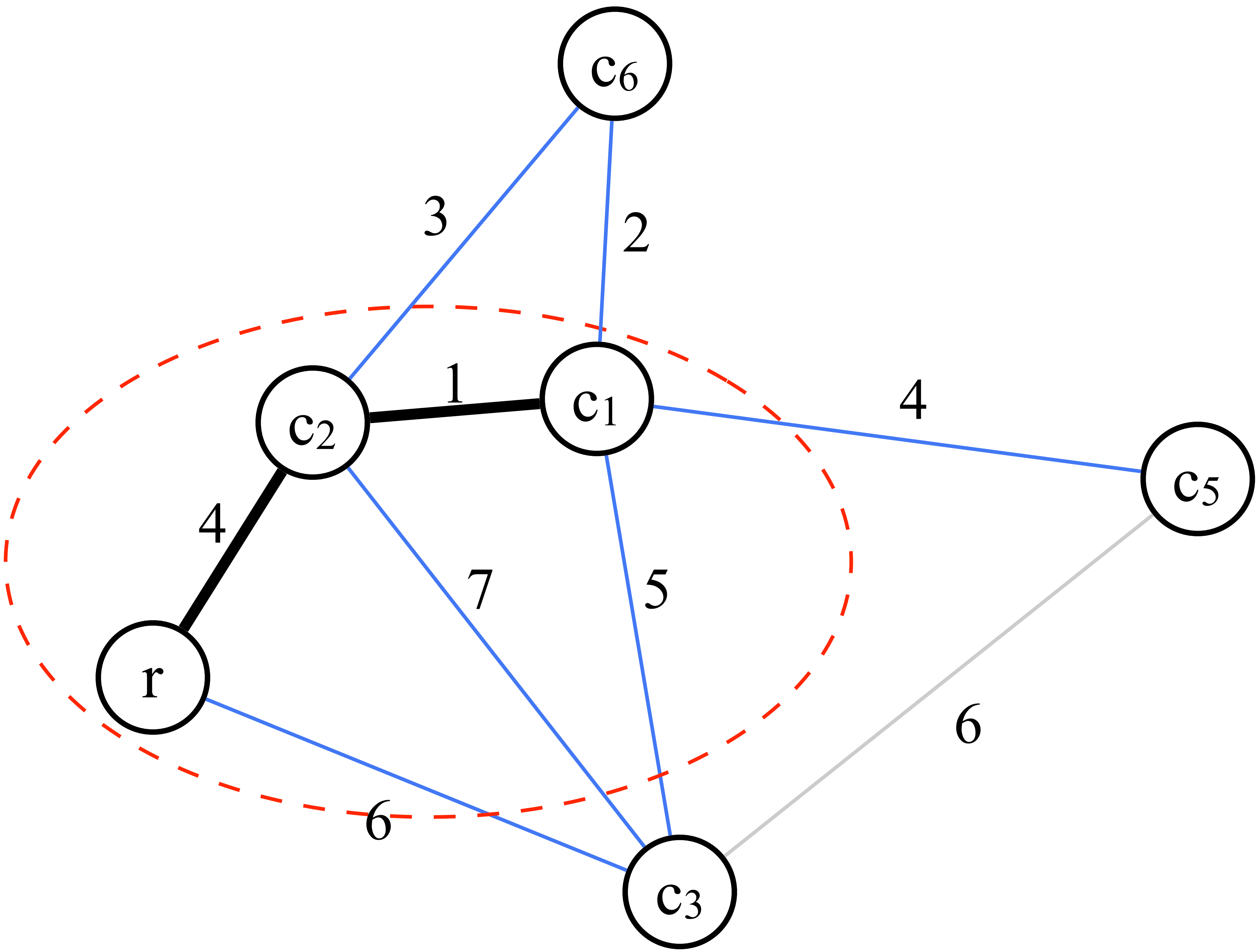
Prim's algorithm



Prim's algorithm



Prim's algorithm



Prim's algorithm

```
MST-Prim(G, w, r){ // r is an arbitrary node in G, G is connected
    foreach node u in G{
        u.dis = ∞; // The distance between u and Tr where Tr is the tree
containing r. Initially, Tr = {r}.
        u.parent = NIL
    }
    r.dis = 0;
    Q ← Make-heap(V); // V is the node set of G using the "dis"
attributes as keys
    while(Q ≠ ∅){
        u = Extract-min(Q); // the closest node to Tr but not in Tr
        foreach (node v in Adj[u] and v in Q){
            if(w(u, v) < v.dis){
                Decrease-key(v, w(u, v)); // update the distance
                v.parent = u;
            }
        }
    }
```

Prim's algorithm

```
MST-Prim(G, w, r){ // r is an arbitrary node in G, G is connected
    foreach node u in G{
        u.dis = ∞; // The distance between u and T_r where T_r is the tree containing r. Initially, T_r = {r}.
        u.parent = NIL
    }
    r.dis = 0;
    Q ← Make-heap(V); // V is the node set of G using the "dis" attributes as keys
    while(Q ≠ ∅){
        u = Extract-min(Q); // the closest node to T_r but not in T_r
        foreach (node v in Adj[u] and v in Q){
            if(w(u, v) < v.dis){
                Decrease-key(v, w(u, v)); // update the distance
                v.parent = u;
```

We need to update the distance between each node to T_r if we newly add a node to T_r.

Prim's algorithm

MST-Prim(G, w, r){ // r is an arbitrary node in G, G is connected

foreach node u in G{

u.dis = ∞; // The distance between u and T_r where T_r is the tree containing r. Initially, $T_r = \{r\}$.

u.parent = NIL

}

r.dis = 0;

Q $\leftarrow$ Make-heap(V); // V is the node set of G using the "dis" attributes as keys

while(Q $\neq \varnothing$){

u = Extract-min(Q); // the closest node to T_r but not in T_r

foreach (node v in Adj[u] and v

if(w(u, v) < v.dis){

Decrease-key(v, w(u, v));

v.parent = u;

}

}

}

O(n) Extract-min operations and O(m) Decrease-key operations, the running time is thus O((m+n)log(n)) = O(mlog(n)) for binary heaps and O(nlog(n)+m) for Fibonacci heaps.

Prim's algorithm

```
MST-Prim(G, w, r){ // r is an arbitrary node in G, G is connected
    foreach node u in G{
        u.dis = ∞; // The distance between u and T_r where T_r is the tree containing r. Initially, T_r = {r}.
        u.parent = NIL
    }
    r.dis = 0;
    Q ← Make-heap(V); // V is the node set of G using the "dis" attributes as keys
    while(Q ≠ ∅){
        u = Extract-min(Q); // the closest node to T_r but not in T_r
        foreach (node v in Adj[u] and v
            if(w(u, v) < v.dis){
                Decrease-key(v, w(u, v));
                v.parent = u;
            }
    }
}
```

We need a direct address table to keep track with the position of each node in the heap because Decrease-key requires us to specify a pointer to that node. For example, hash[v_7] = &Q[3], hash[v_{11}] = &Q[20].

Exercise

Input: an undirected graph $G = (V, E)$ and a weight function $w: E \rightarrow \mathbf{R}$. Each $w(e)$ for e in E is either 1 or 2.

Output: the MST of G.

Is this problem solvable in $O(n+m)$ time?

A Parallel Algorithm for Minimum Spanning Trees

Boruvka's algorithm (aka Sollin's algorithm)

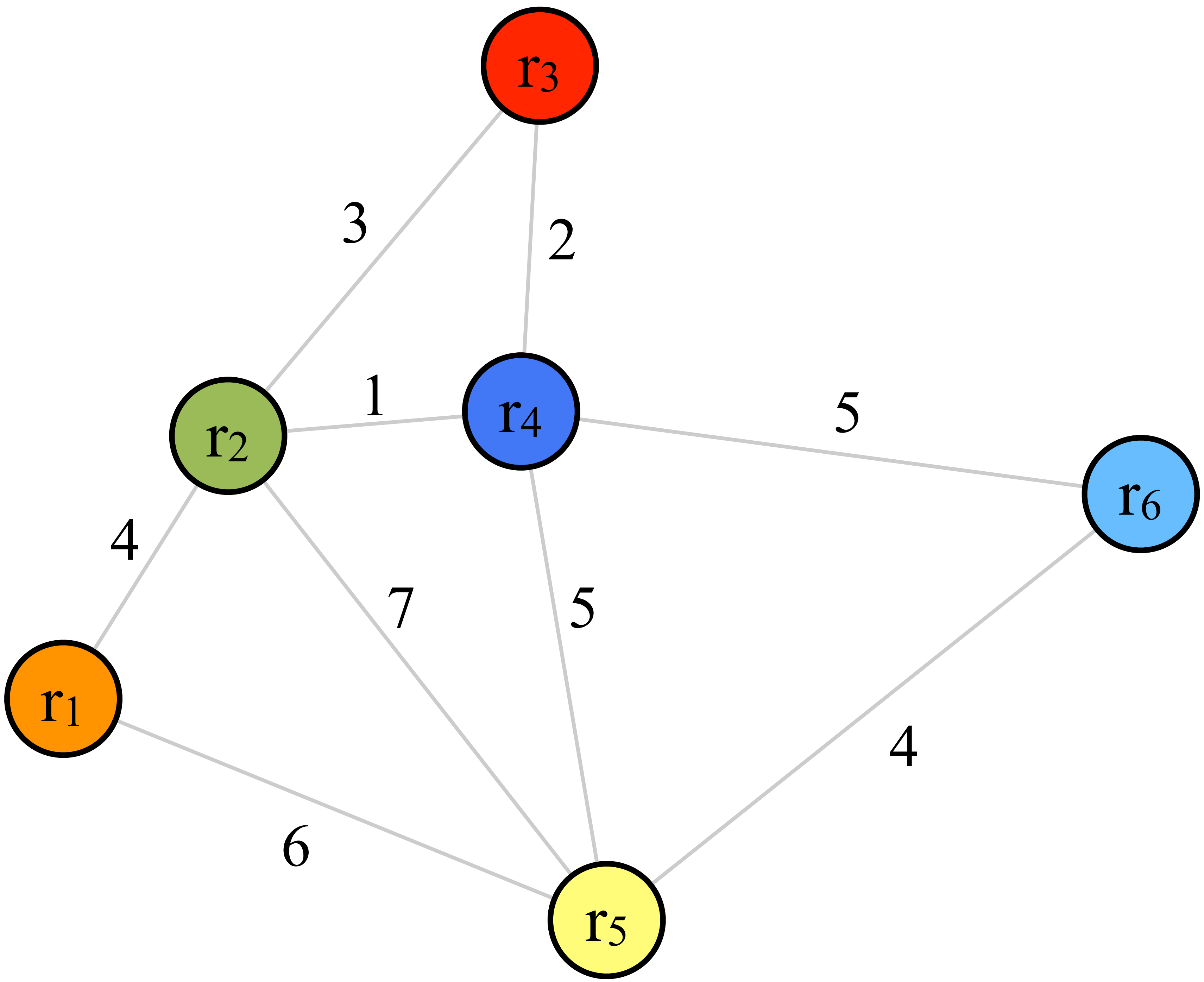
MST-Boruvka(G, w){
forks n threads;

Each thread T_i will pick a unique v_i as its root and invoke MST-Prim(G, w, v_i);

If two threads T_i and T_j ($i < j$) work on the same connected component, T_i returns and T_j takes over T_i's work.
}

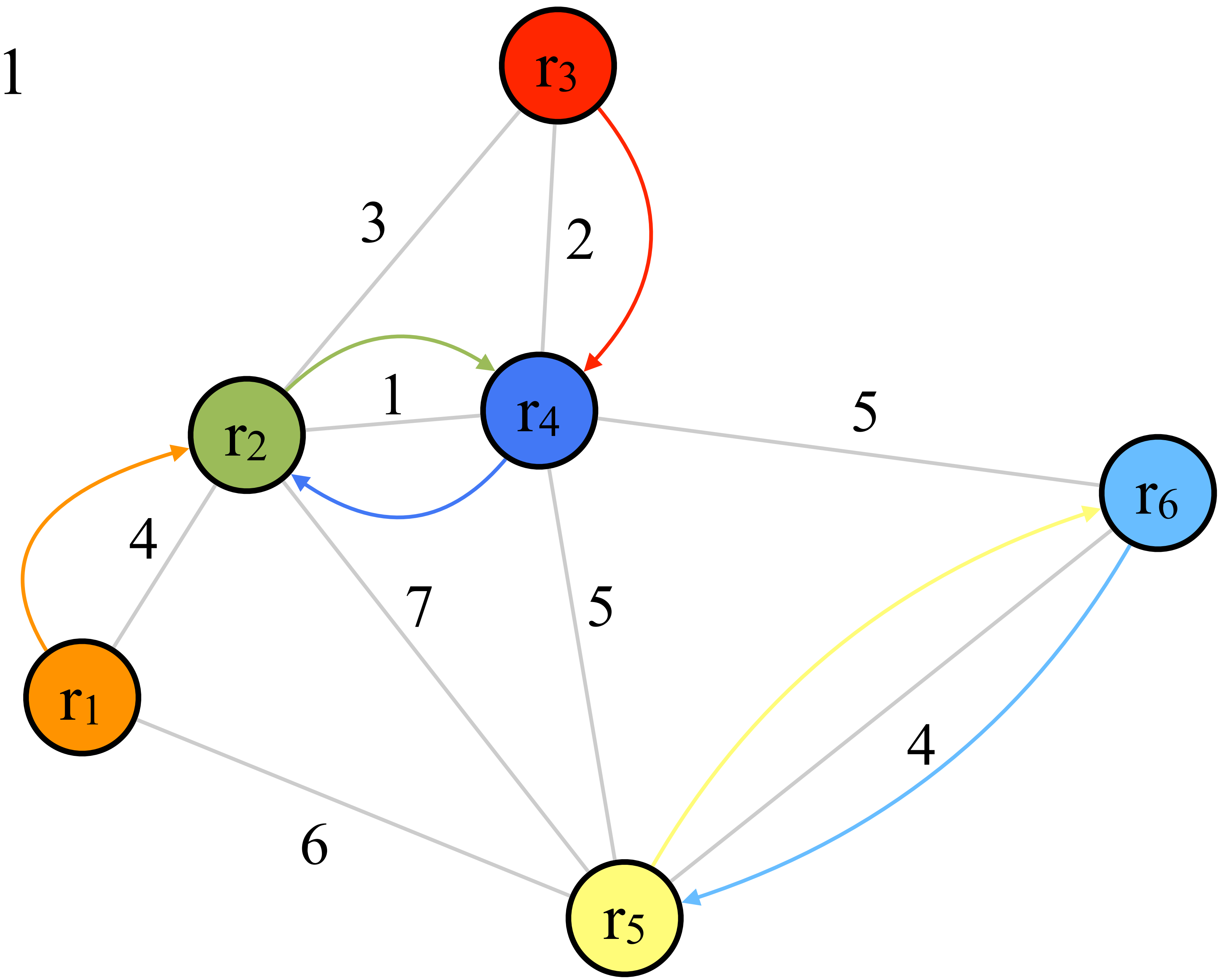
Prim's algorithm picks a node as the root r and iteratively enlarge the connected component that contains r.
Boruvka's algorithm picks all nodes as the root.

Boruvka's algorithm (aka Sollin's algorithm)



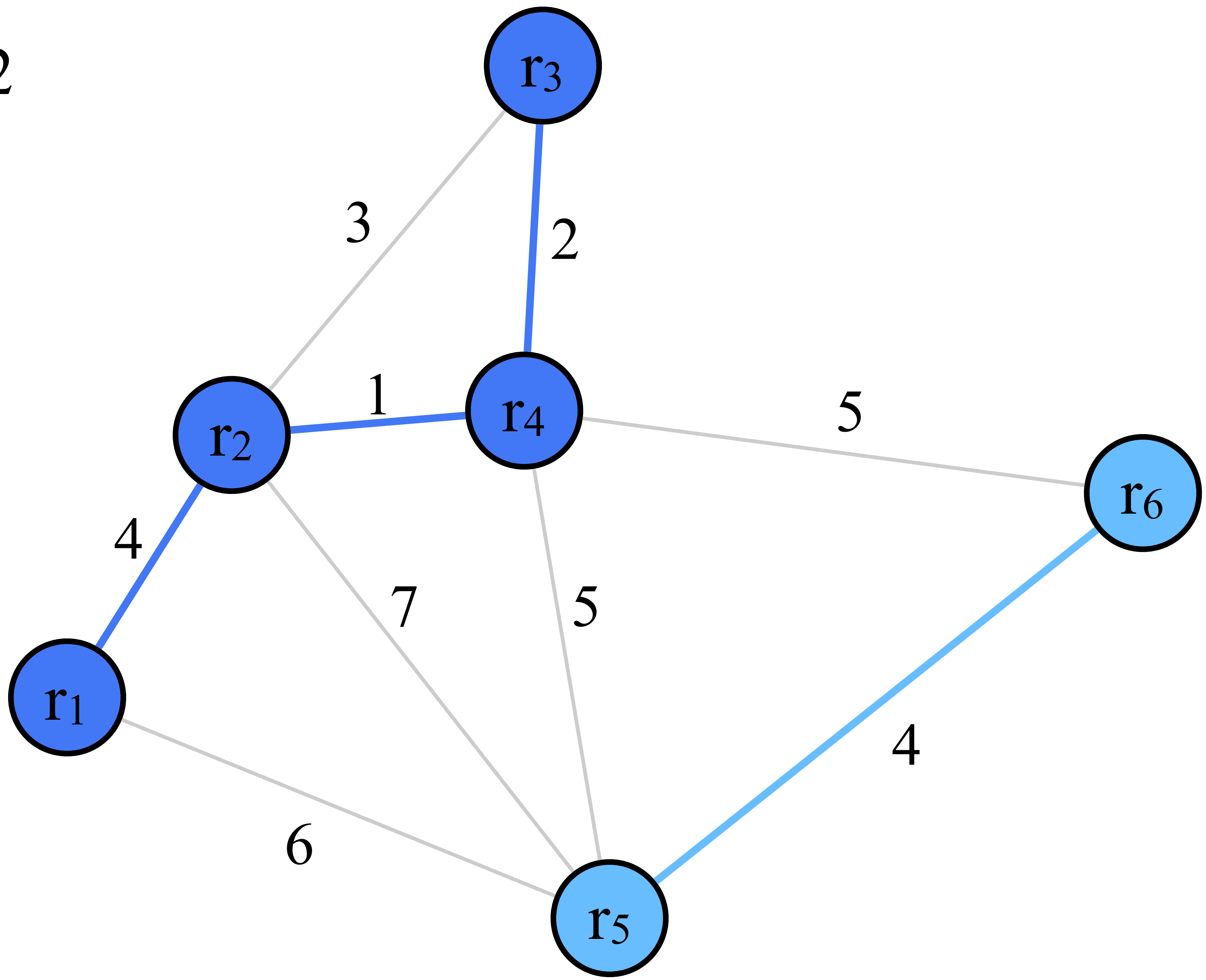
Boruvka's algorithm (aka Sollin's algorithm)

Round 1



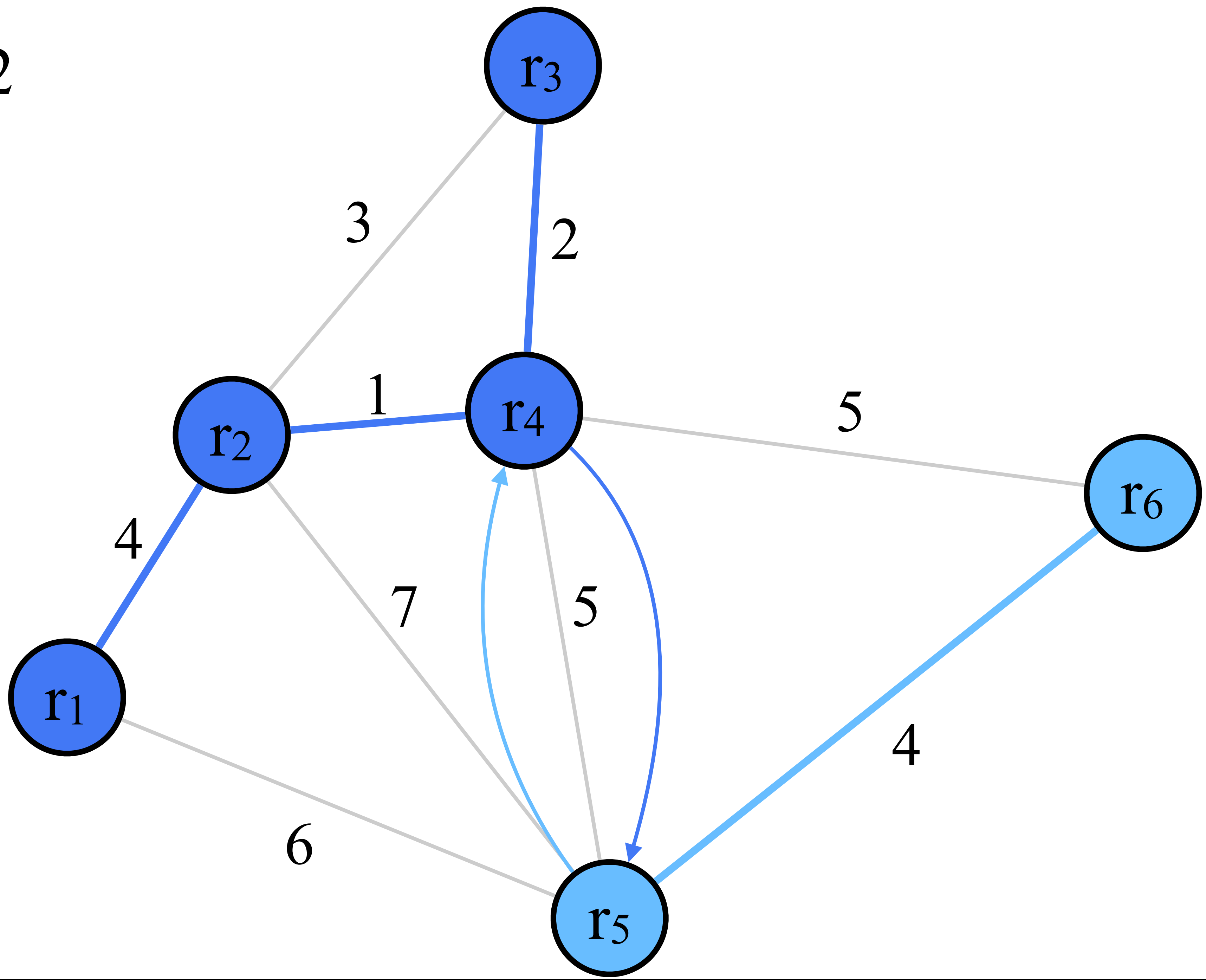
Boruvka's algorithm (aka Sollin's algorithm)

Round 2



Boruvka's algorithm (aka Sollin's algorithm)

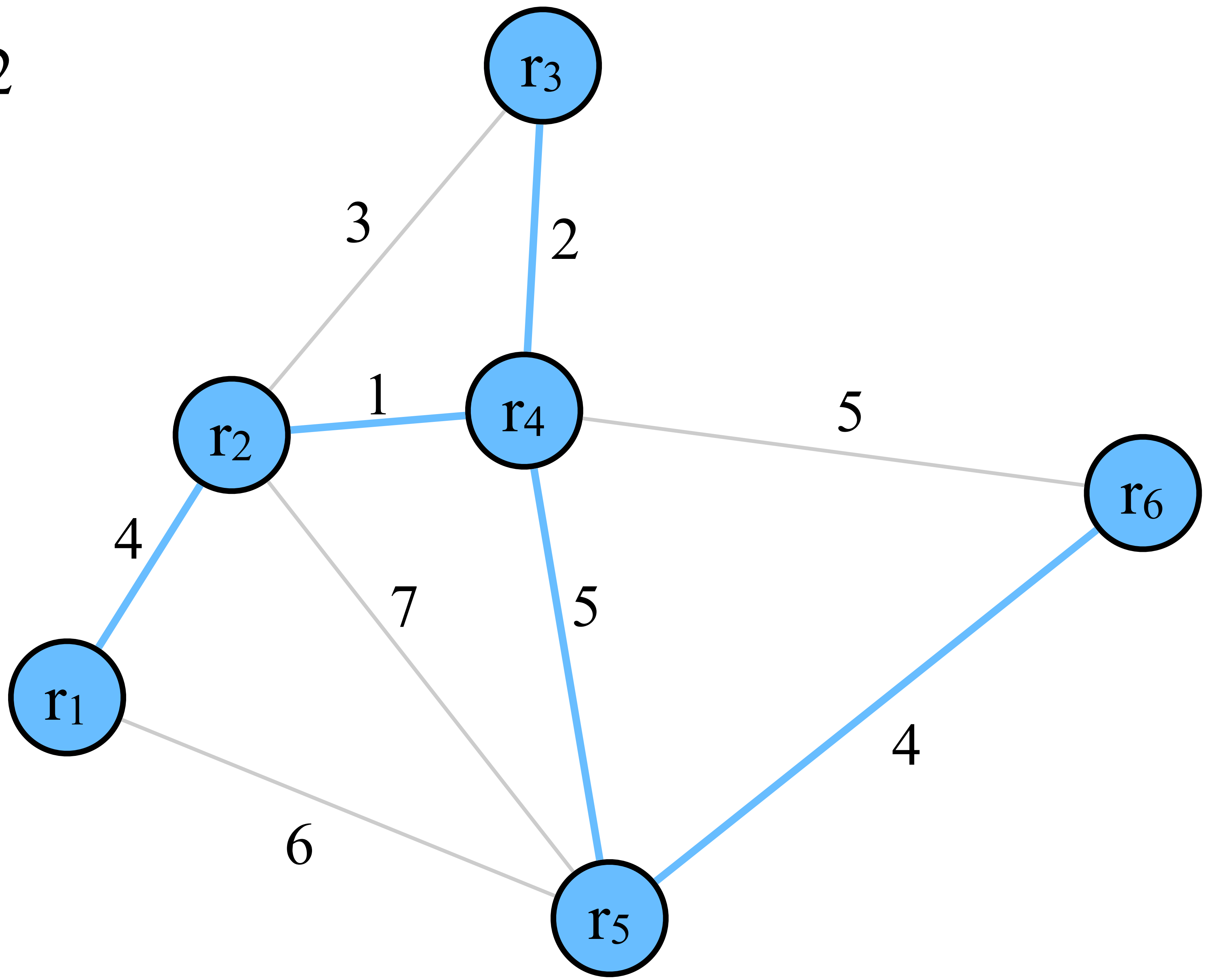
Round 2



If two edges have the same weight, tie-break by indices.

Boruvka's algorithm (aka Sollin's algorithm)

Round 2



Running time (1/2)

Claim. There are at most O(log n) rounds in Boruvka's algorithm, if there are n processors.

Proof. In a round that k threads still survive, each thread will pick a directed edge to extend. We need at least k/2 (undirected) edges to cover all such directed edges. After adding these k/2 (undirected) edges to A. # of connected component in A reduces to a half. Thus, O(log n) rounds suffice.

Running time (2/2)

Claim. Selecting the min-weight edge from k edges can be done in O(log k) time, if there are k processors.

Proof. Fork k threads, each of which handles an edge. Then, we pair the threads. For each pair, if the edge handling by a thread has weight larger than that of the other thread, then the thread returns. We repeat this procedure until only 1 thread survives.

Boruvka's algorithm runs in $O(\log^2 n)$ time
if there are $O(n^3)$ processors.

Simulating the Parallel Algorithm with 1 Thread

Boruvka's algorithm

```
MST-Boruvka(G, w){
    While(V(G) > 1){
      (1) Pick the lightest edge from each node in G.
      (2) Contract the picked edges.
      // Components now become supernodes, which are nodes for the
      next run.
    }
}
```

Boruvka's algorithm

```
MST-Boruvka(G, w){
    While(V(G) > 1){
      (1) Pick the lightest edge from each node in G.
      (2) Contract the picked edges.
      // Components now become supernodes, which are nodes for the
      next run.
    }
}
```

Step 1 takes O(m) time.
The total running time is O(m log n).

Yao's algorithm

```
MST-Yao(G, w){
    (0) For each node, partition the edges incident to it into k
    categories, so that the edges in the i-th category have weight no
    greater than that of the edges in the j-th category for every i < j.

    While(V(G) > 1){
        (1) Pick the lightest edge from each node in G.
        // from the first category if some edges in it are valid.
        // Pop invalid edges, which each edge turns invalid only once.
        // Pop operations takes O(m) time in total.
        (2) Contract the picked edges.
        // Components now become supernodes, which are nodes for the
        next run.
    }
}
```

Yao's algorithm

MST-Yao(G, w){

(0) For each node, partition the edges incident to it into k categories, so that the edges in the i-th category have weight no greater than that of the edges in the j-th category for every $i < j$.

While($V(G) > 1$){

(1) Pick the lightest edge from each node in G.

// from the first category if some edges in it are valid.

// Pop invalid edges, which each edge turns invalid only once.

// Pop operations takes $O(m)$ time in total.

(2) Contract the picked edges.

// Components now become supernodes, which are nodes for the next run.

Step 0 takes $O(m \log k)$ time and Step 1 takes $O(m/k)$ time.
The total running time is $O(m \log k + (m/k) \log n)$
$= O(m \log\log n)$ by picking $k = O(\log n)$.