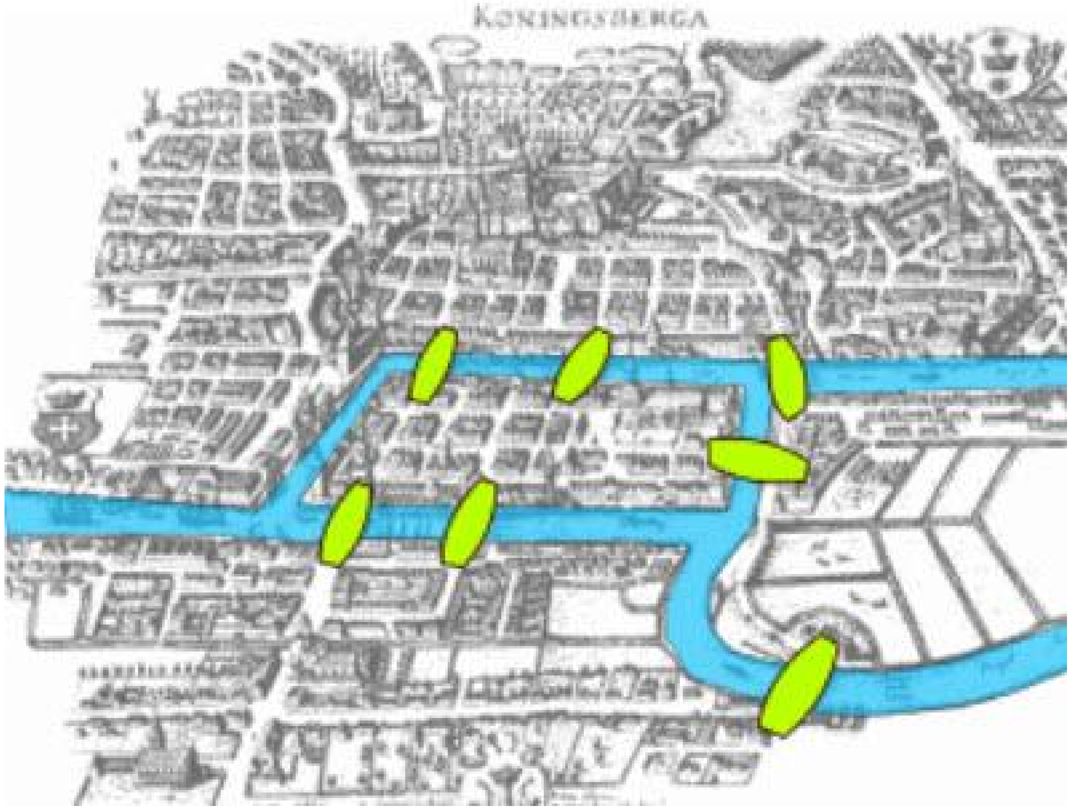
Assignment 5: Graphs

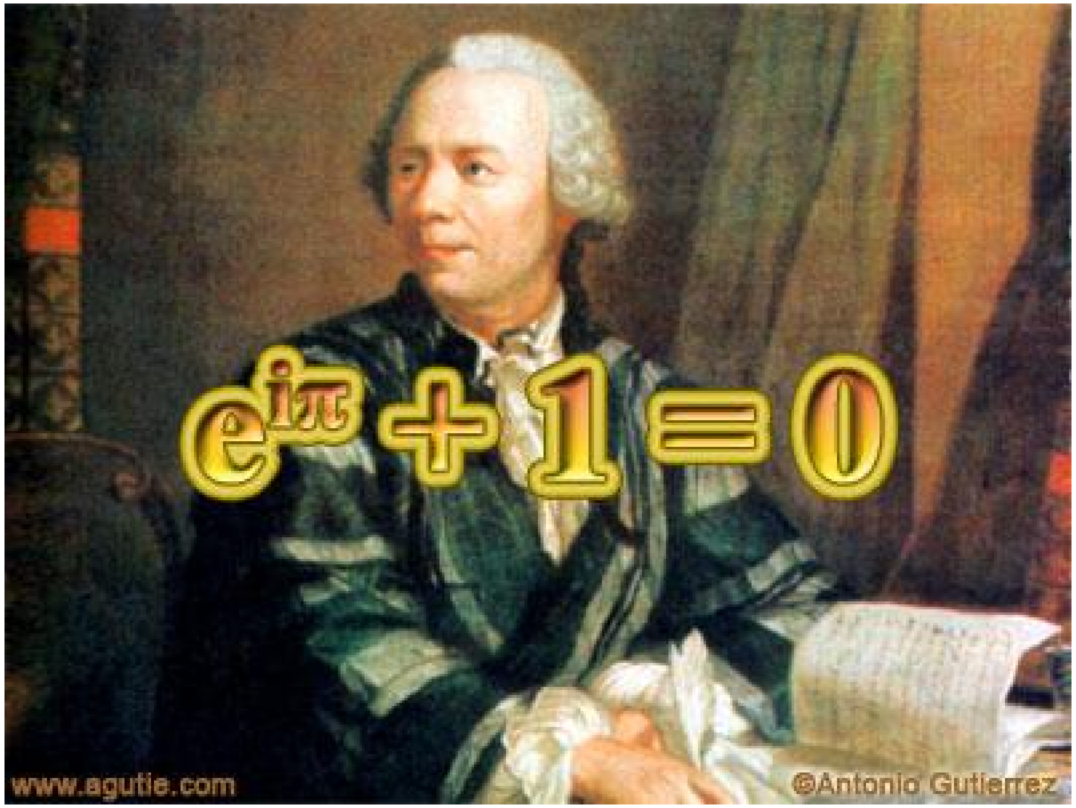
CS2 Recitation 2/5/16

Seven Bridges of Konigsberg



Can you take a walk through the town, visiting each part of the town and **crossing each bridge once and only once?**

Seven Bridges of Konigsberg

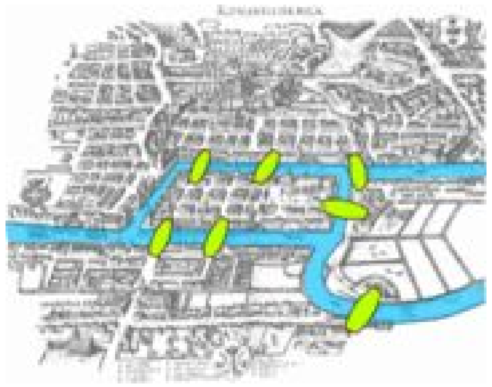


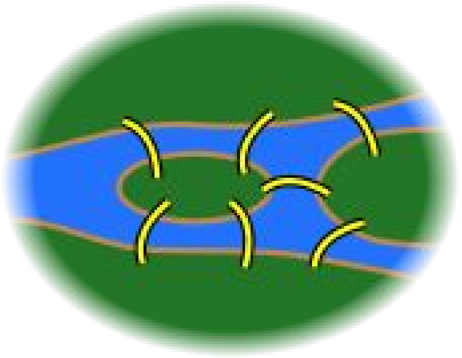
Euler for Rescue

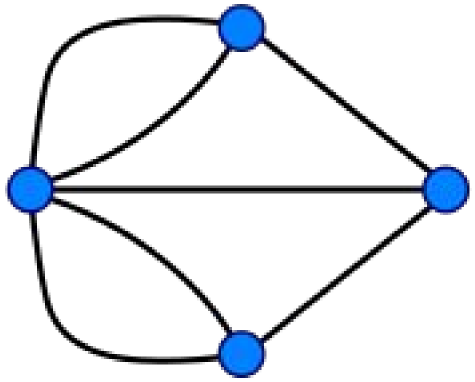
Seven Bridges of Konigsberg

Simplification!

The Start of Graph Theory...



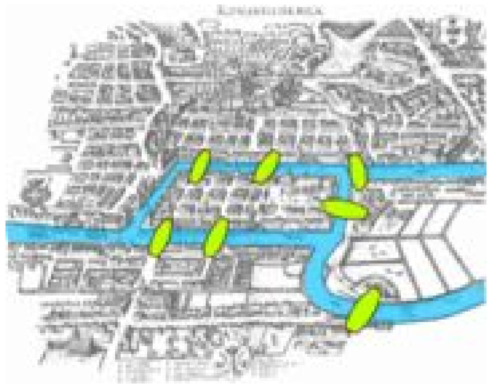
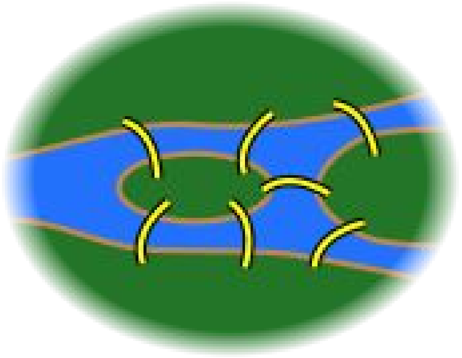
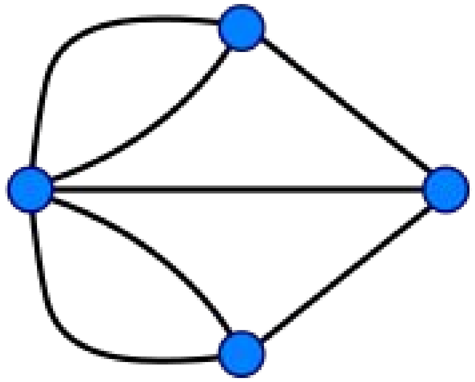




Seven Bridges of Konigsberg

Simplification!

The Start of Graph Theory…

Solution: Zero or two odd nodes.

Graphs

- G = (V, E)
- V = set of vertices
- E = set of edges with weights

STL Containers

- std::vector<T>
  - dynamic array (random-access)
- std::list<T>
  - doubly-linked list (appending on either end)
- std::map<K, V>
  - key-value mapping (like python dictionaries)

STL Vector/List

std::vector<Foo *> vec1;

std::list<Foo *> lst1;

- insert element
  - vec1.push_back(foo);
  - lst1.push_back(foo); lst1.push_front(foo);
- remove element
  - vec1.pop_back();
  - lst1.pop_back(foo); lst1.pop_front(foo);
- vectors can be indexed like arrays:
  - vec1[i]
  - vec1.at(i)

STL Vector/List Iterator

```
std::vector<Foo*> vec1;
...
std::vector<Foo*>::iterator i;
for (i = vec1.begin(); i != vec1.end(); i++) {
    Foo *item = *i;
    item->doSomething();
}
```

STL Map

std::map<int, Foo *> map1;

- insert element
  - map1.insert(std::pair<int, Foo*>(id, foo));
  - map1[id] = foo;
- check if map contains element
  - map1.count(id) == 1
- erase element
  - map1.erase(id);

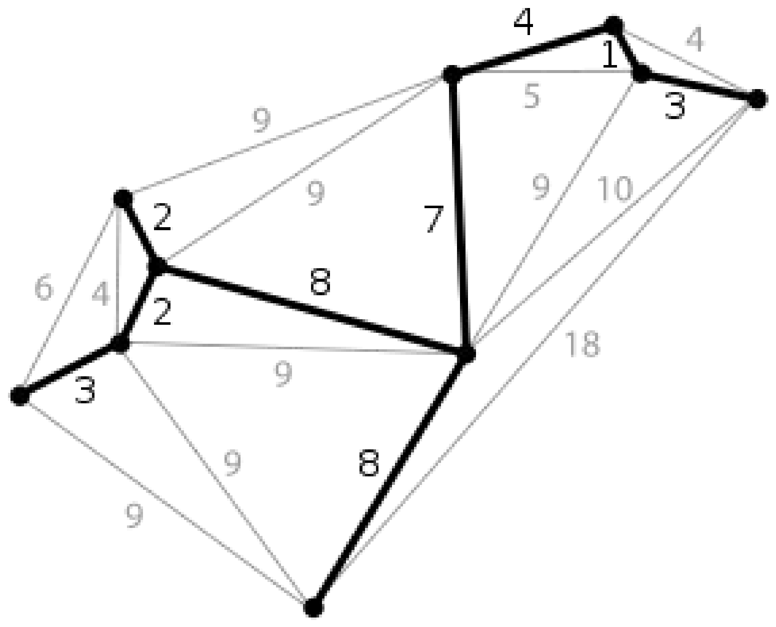
STL Map Iterator

```
std::map<int, Foo *> map1;
...
std::map<int, Foo*>::iterator i;
for (i = map1.begin(); i != map1.end(); i++) {
    // Note: *i is of type std::pair<int, Foo*>
    int item_id = i->first;
    Foo * item = i->second;
    item->doSomething();
}
```

Minimum Spanning Tree (MST)

A tree (connected) with:

- all vertices V(G)
- subset of E(G)
- sum of edge weights is **minimal**



Minimum Spanning Tree (MST)

Conspirators Example:
Agents A,B,C,D,E wants to cover up an affair.

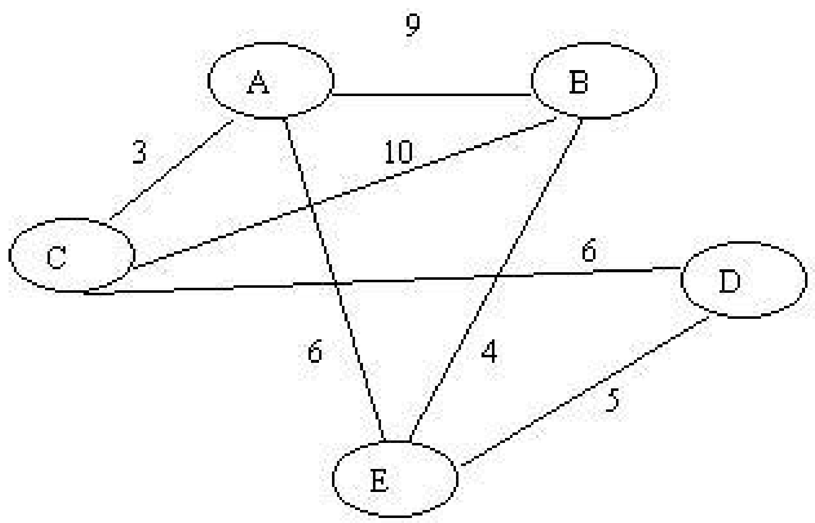
They need communication channel to share information.

But each connection has its risk associated.

What is the best way to connect all of the agents with minimum total risk?

Agent pairs	A B	A C	A E	B C	B E	C D	D E
Risk factors	9	3	6	10	4	6	5

Minimum Spanning Tree (MST)



Agent pairs	A B	A C	A E	B C	B E	C D	D E
Risk factors	9	3	6	10	4	6	5

Prim's Algorithm

Idea: Start from 1 vertex and grow tree

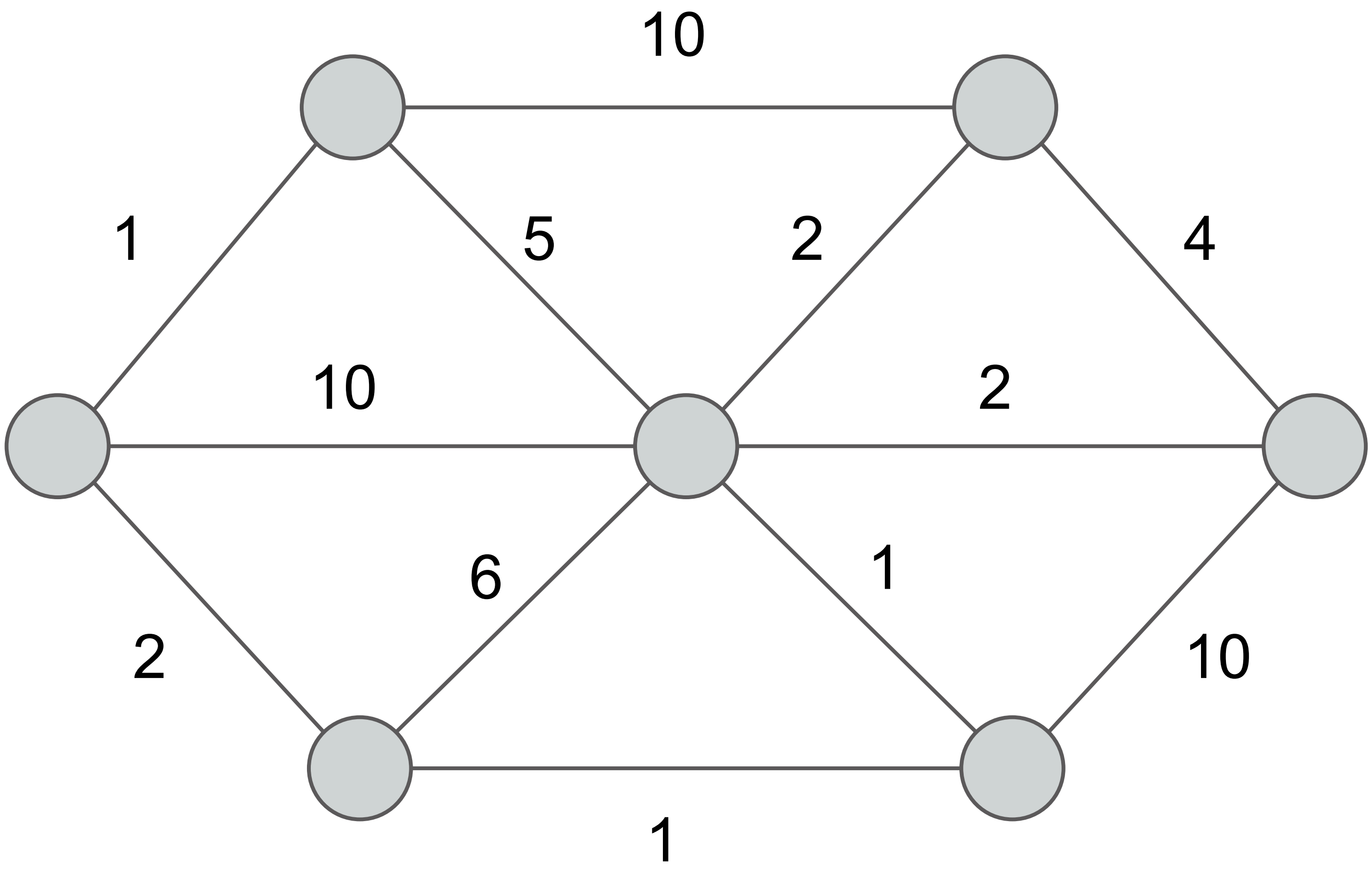
Question: Which vertex do we add next?

- Choose edge with min weight to add new vertex

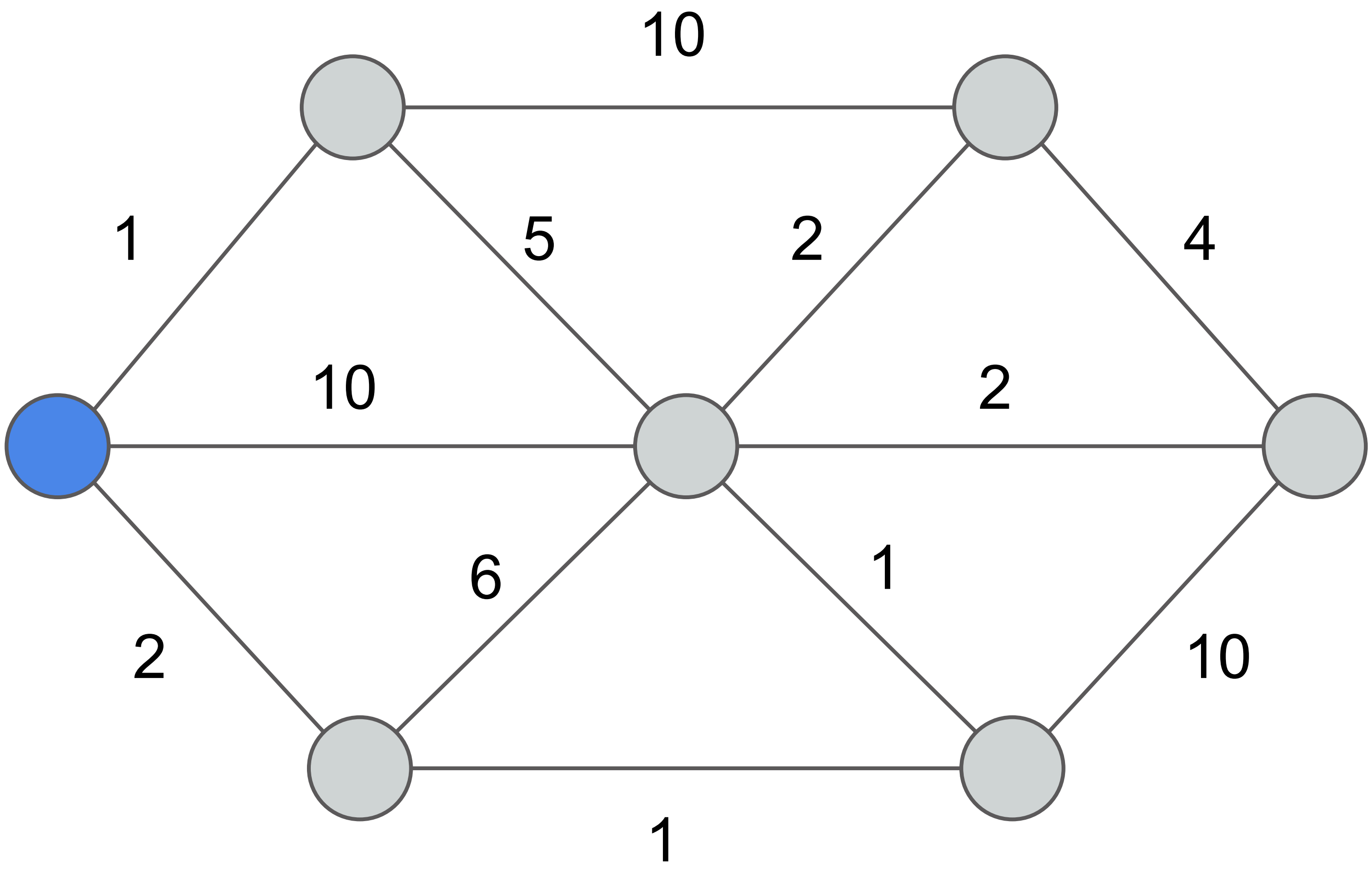
Algorithm:

1. Pick min edge that has 1 vertex in tree and 1 vertex outside
2. Add vertex and edge to tree
3. Repeat (until no more potential edges)

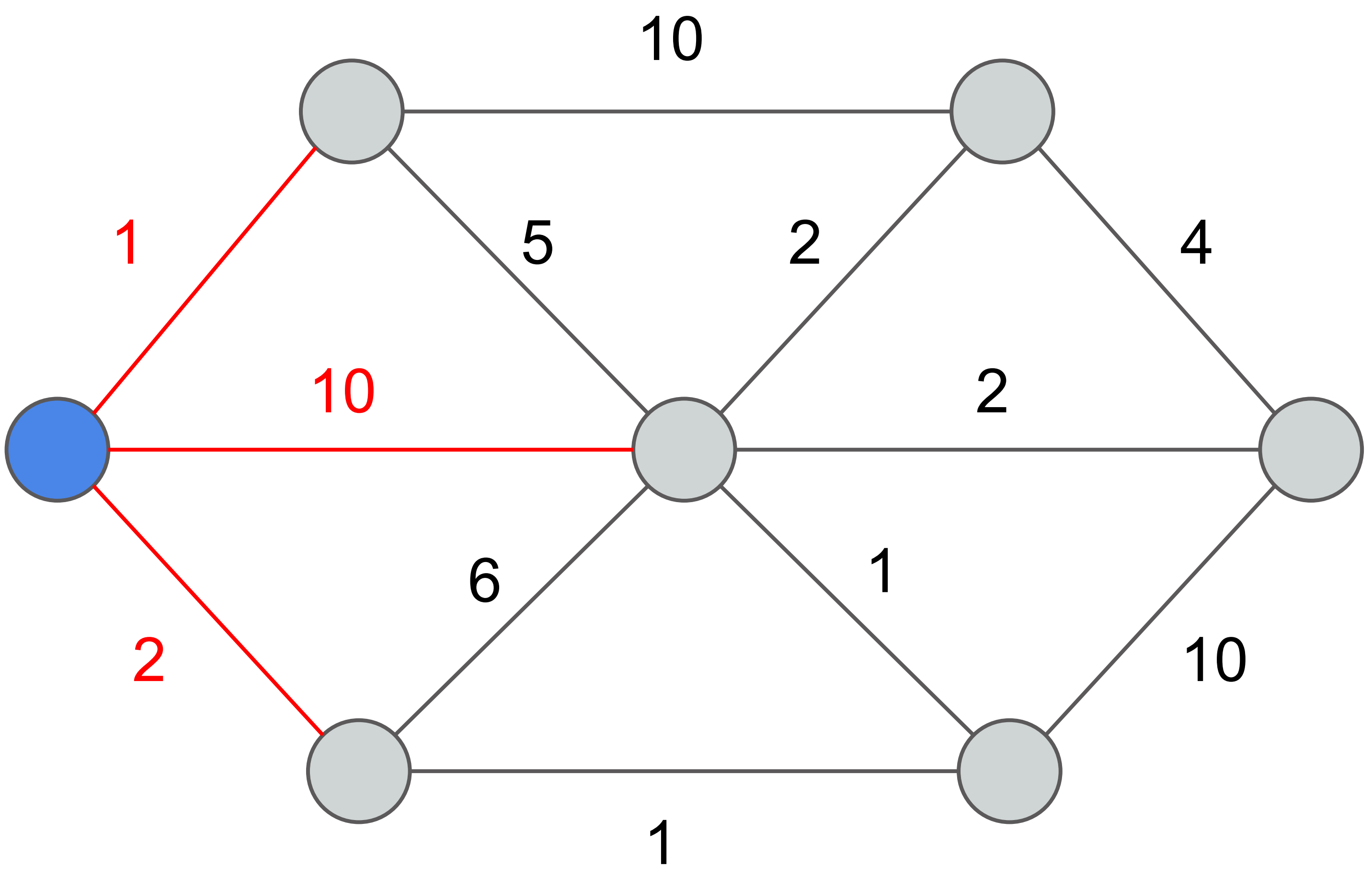
Prim's Algorithm



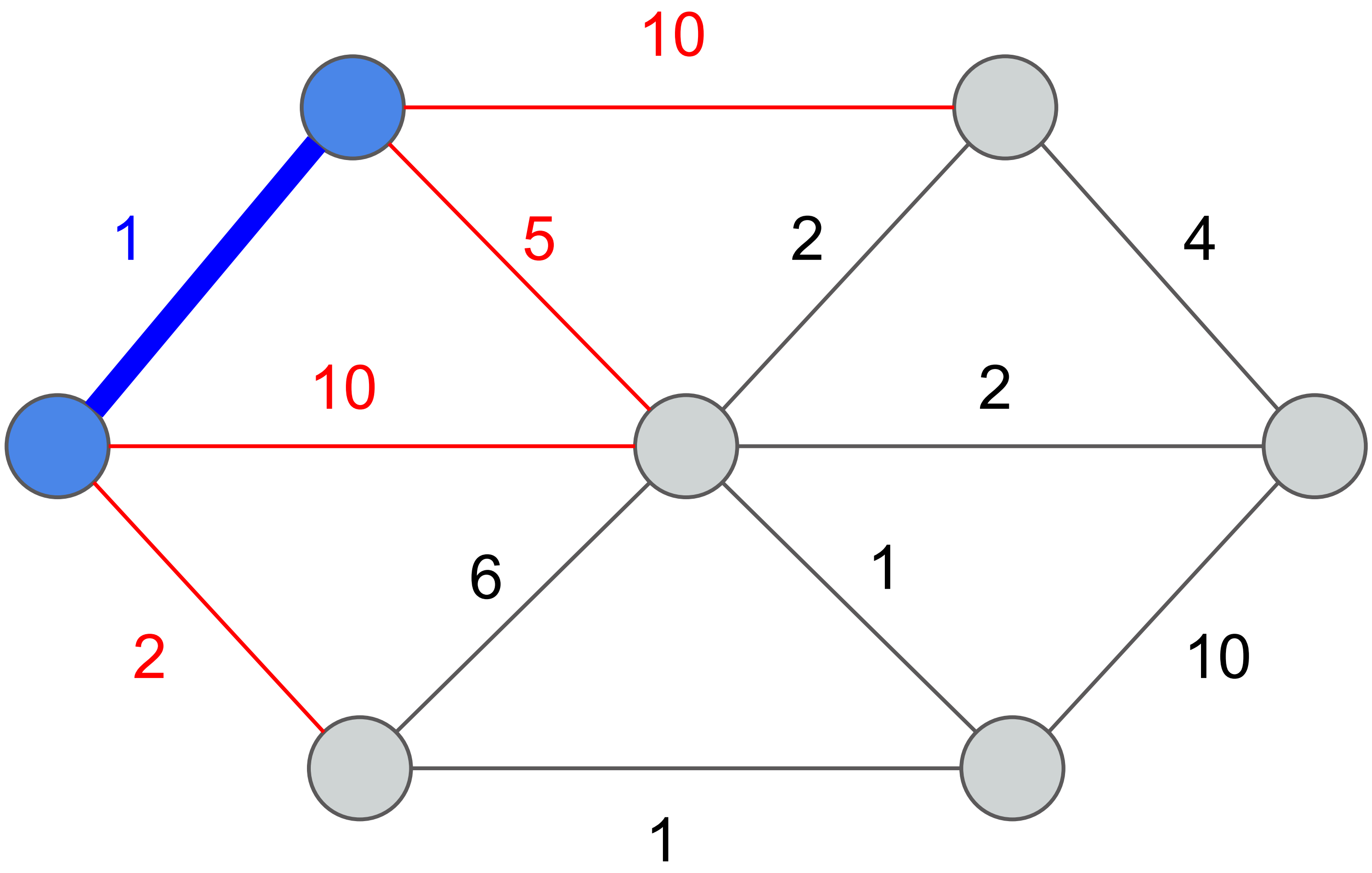
Prim's Algorithm



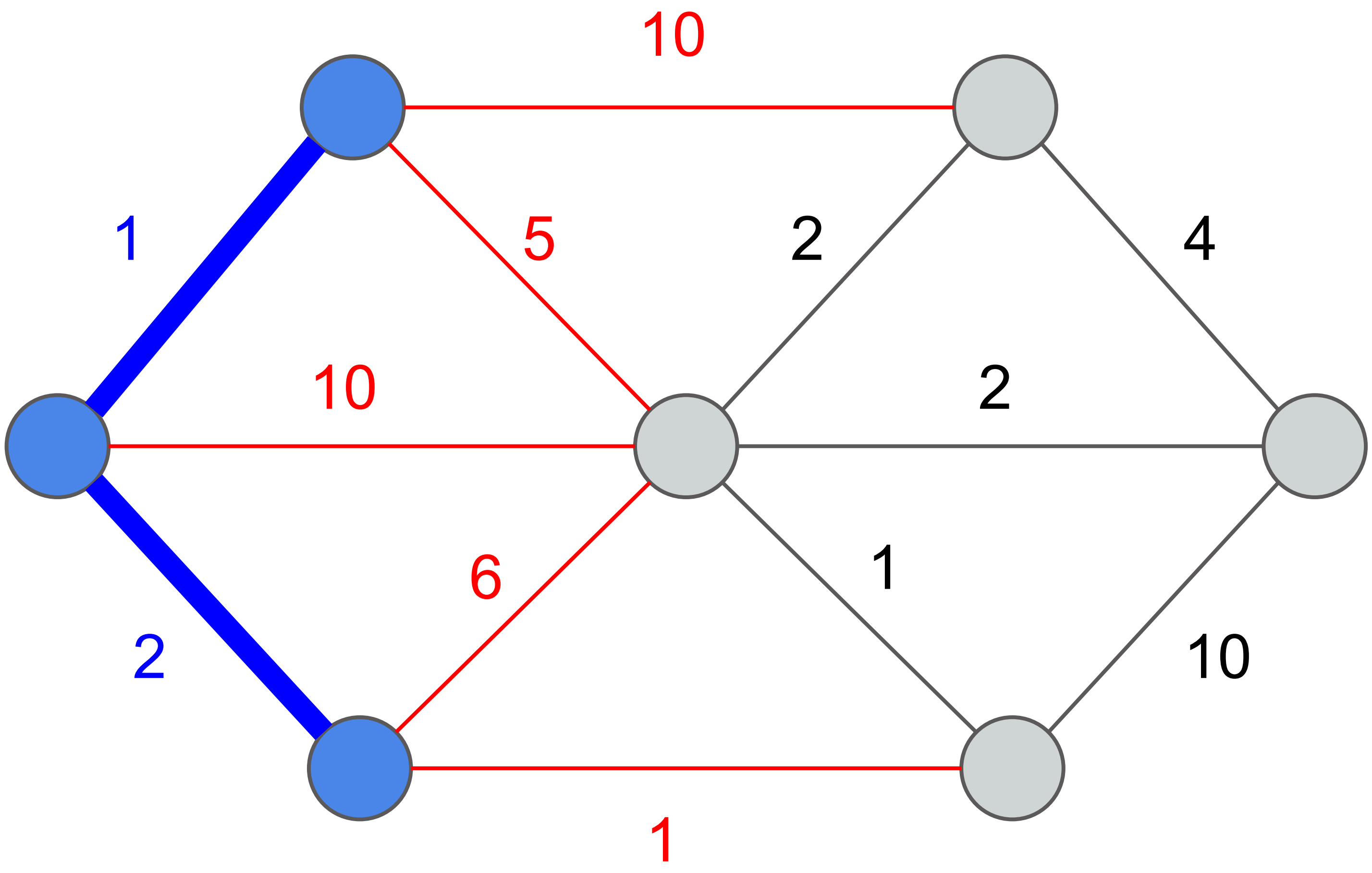
Prim's Algorithm



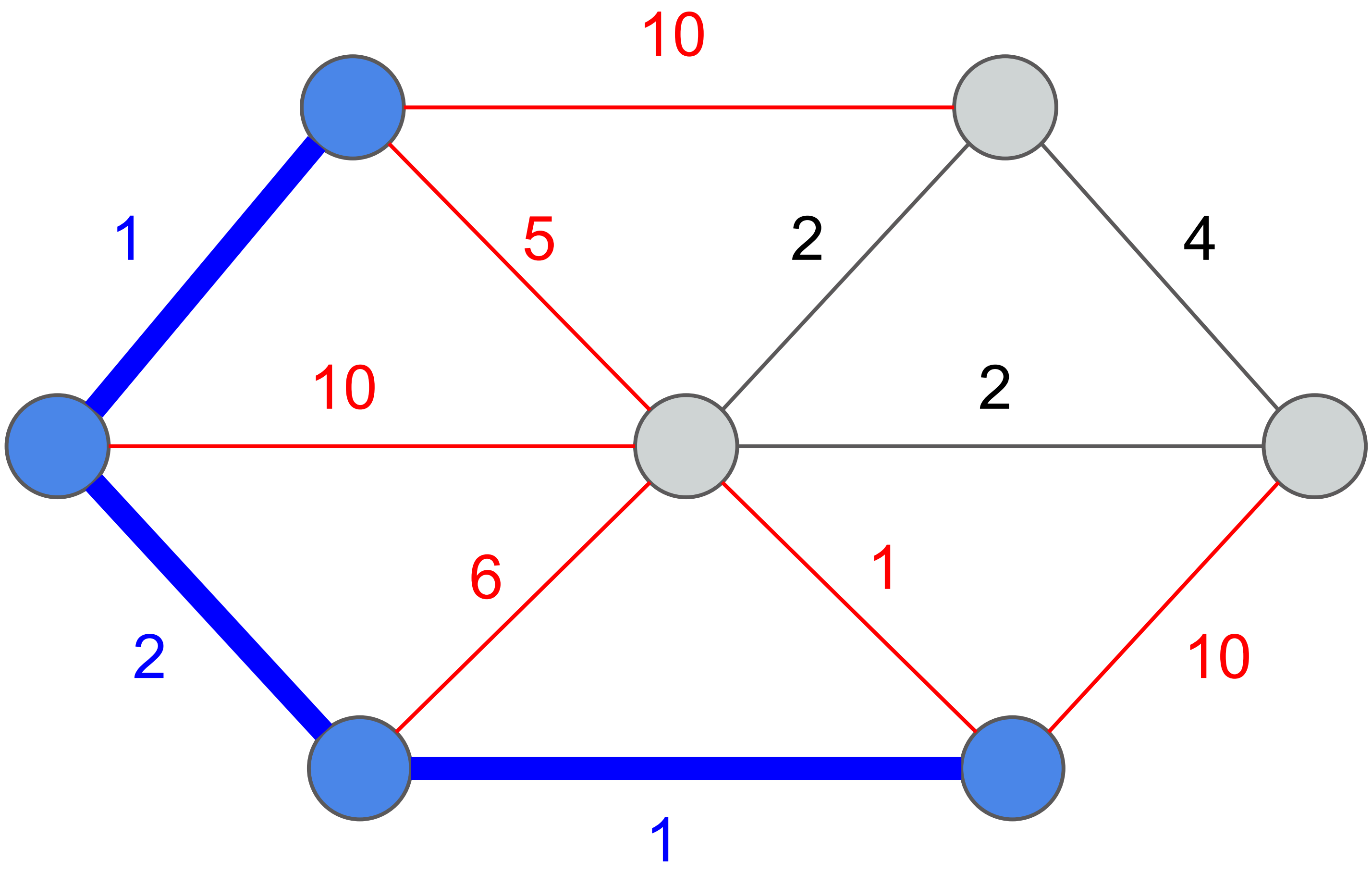
Prim's Algorithm



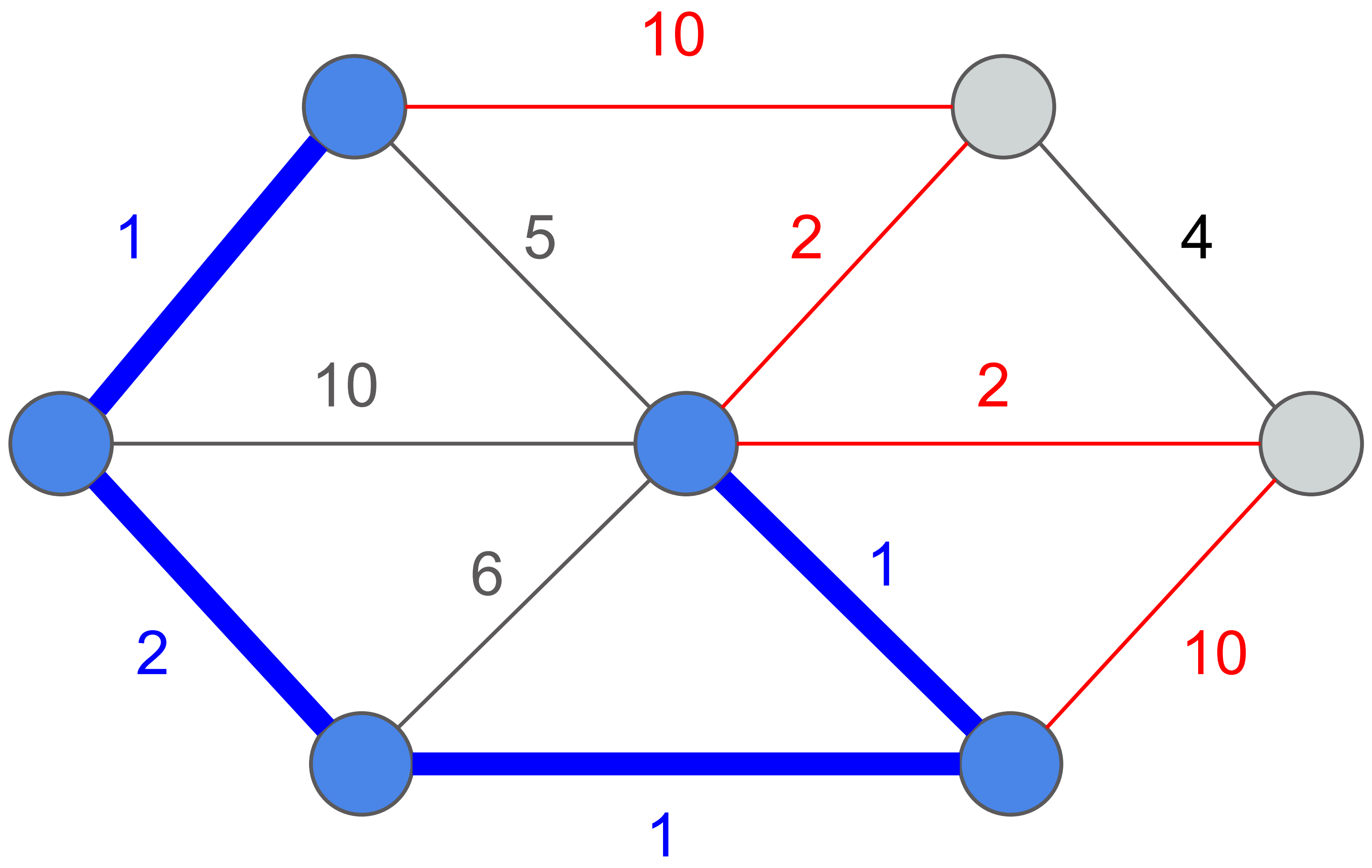
Prim's Algorithm



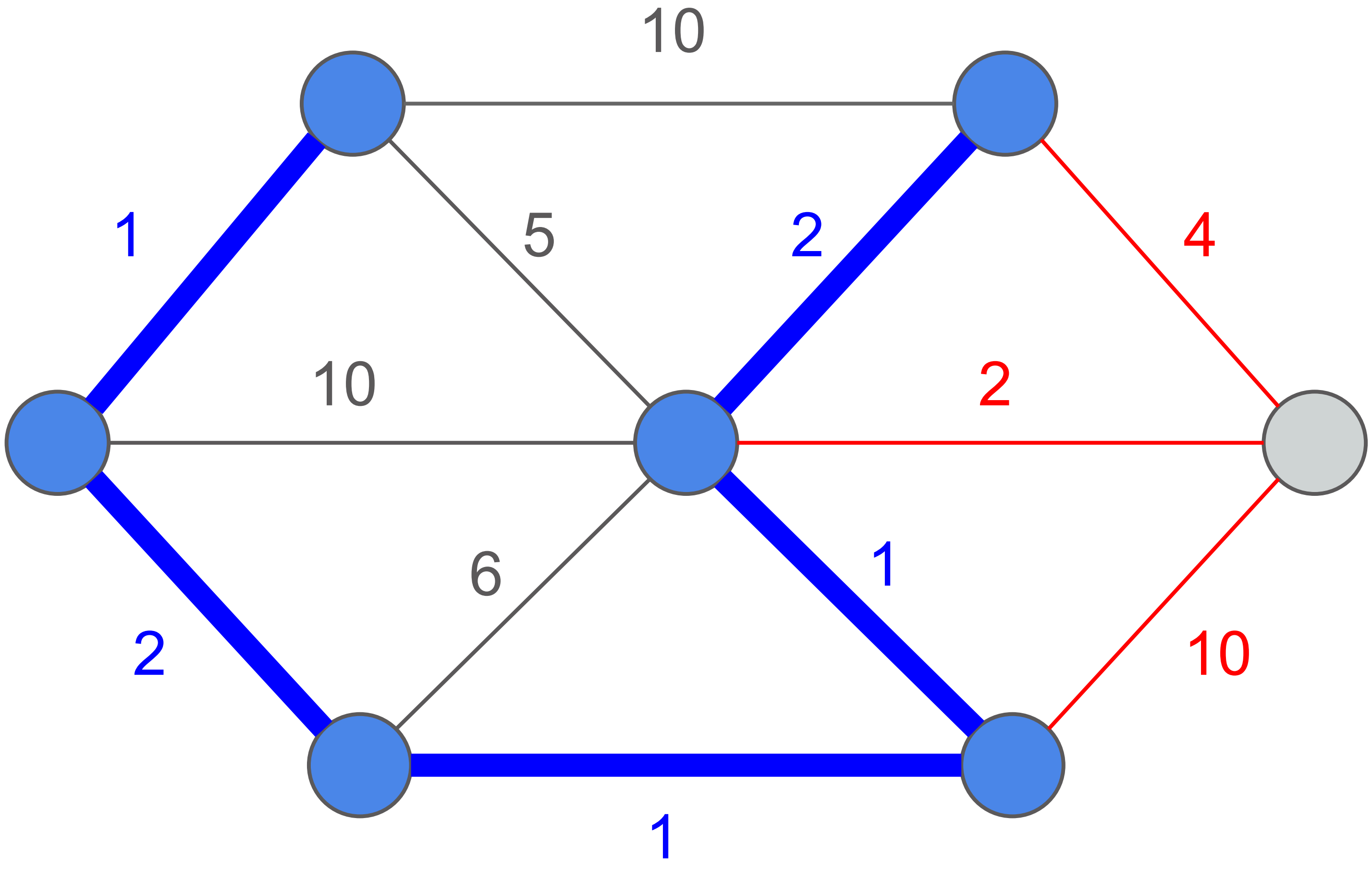
Prim's Algorithm



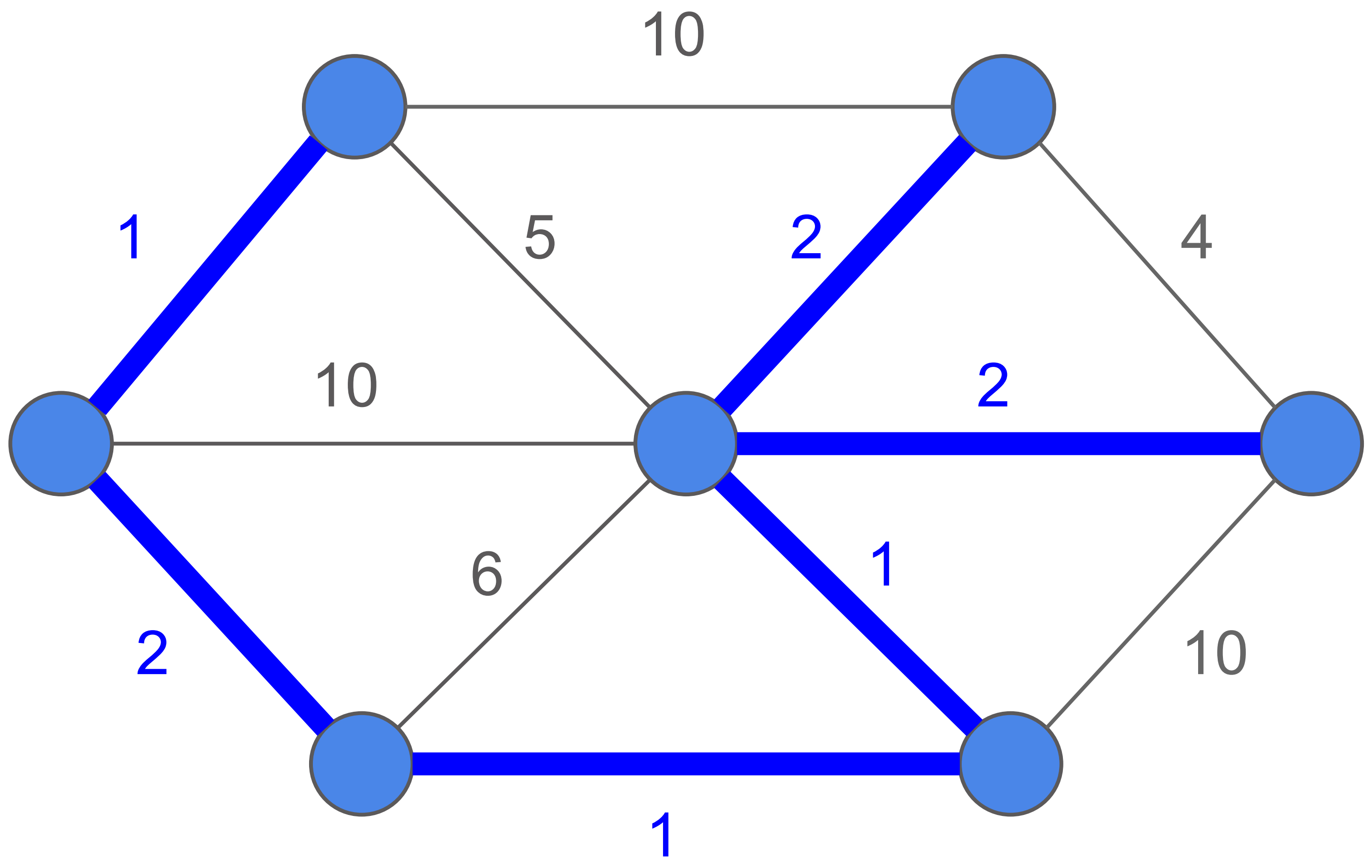
Prim's Algorithm



Prim's Algorithm



Prim's Algorithm

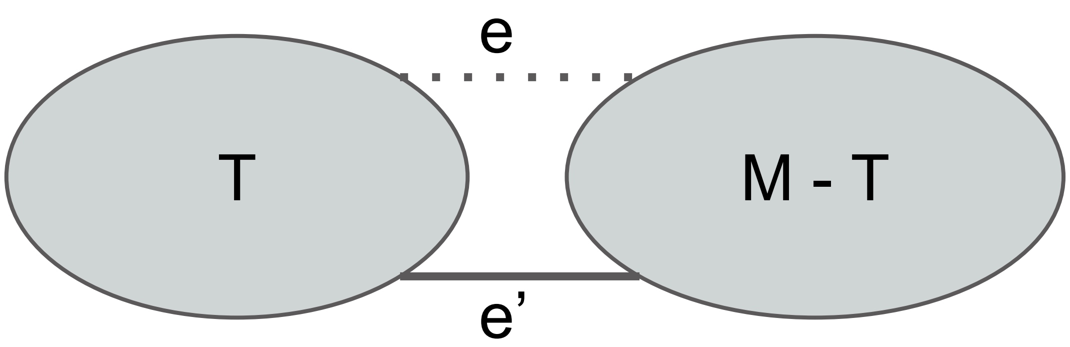


Total cost = 9

Prim's Algorithm: Correctness

Proof by contradiction:

- Let e be the first edge not consistent with actual MST
- T = the tree so far
- M = MST that contains T



- w(e) < w(e’) from Prim’s
- Thus, M is not a MST; contradiction!

Prim's Algorithm: Complexity

- Naive:
  - O(|V|^3)
- keeping track of min weight of each vertex to current tree:
  - O(|V|^2)
- priority queue:
  - binary heap - O(|E| log |V|)
  - fibonacci heap - O(|E| + |V| log |V|)

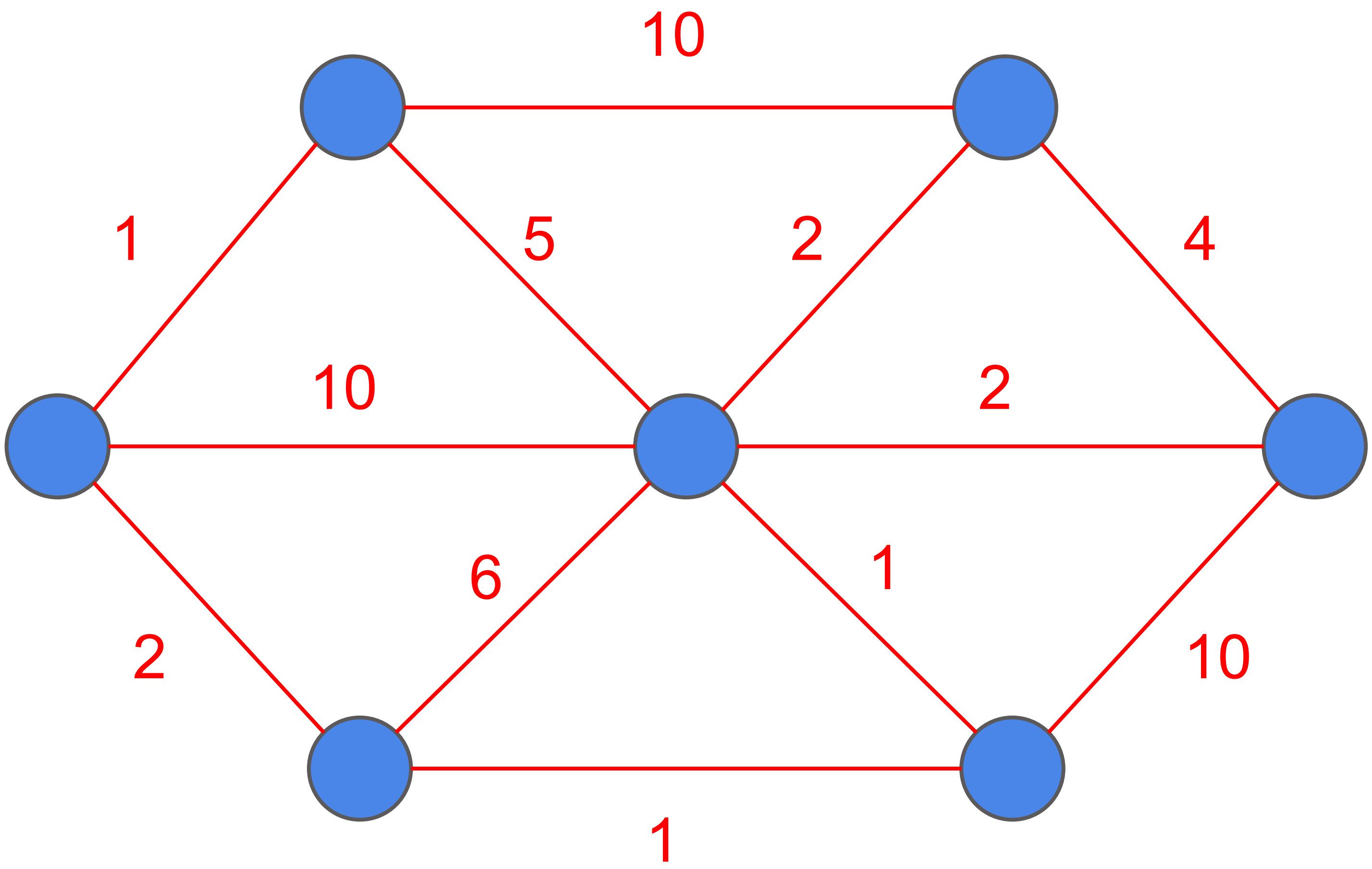
Kruskal's Algorithm

Idea: Start with a 'forest' (set) of one-vertex trees and connect trees until we get an MST

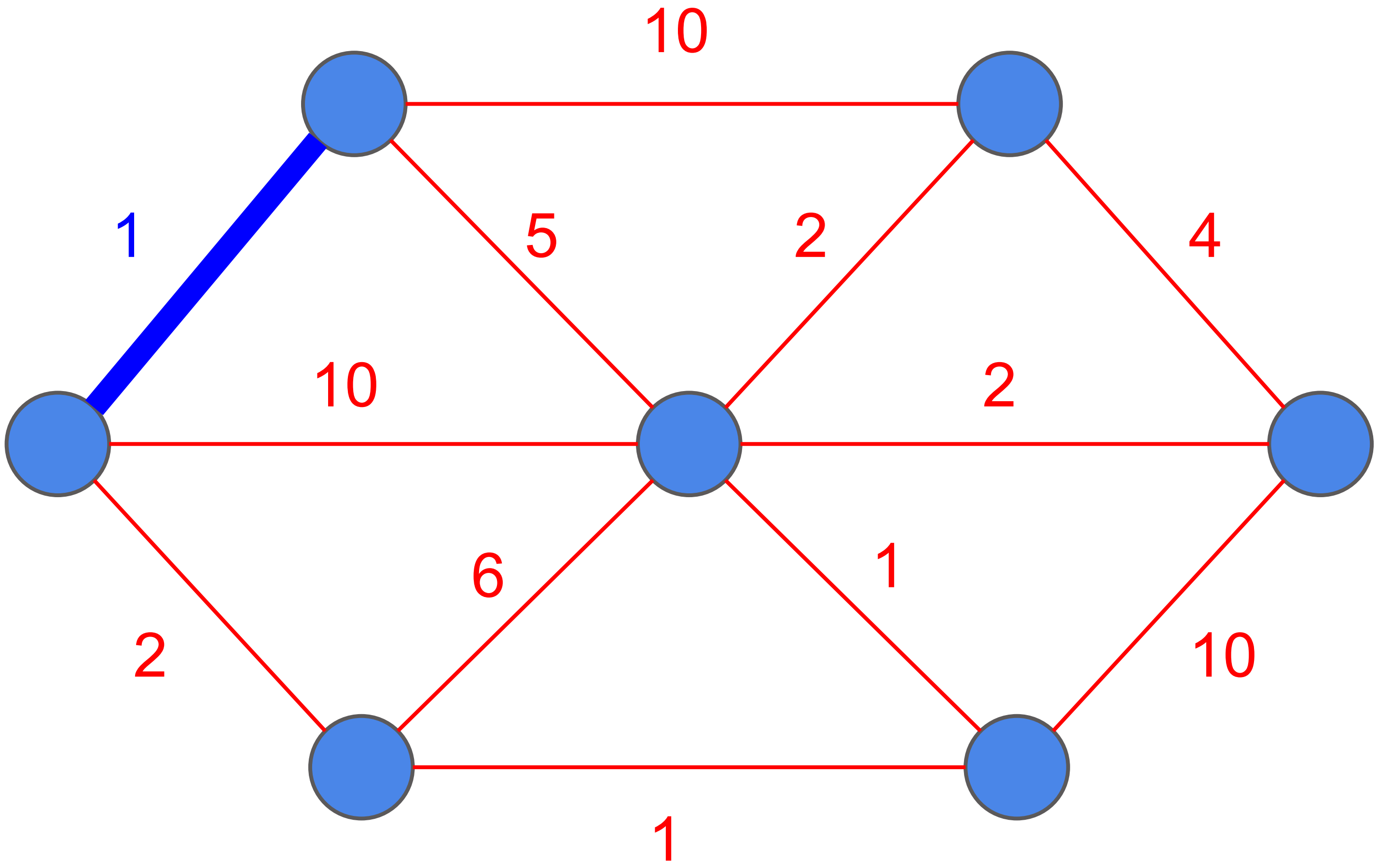
Question: How to connect trees?

1. Choose edge with min weight that connects two different trees.
2. Add edge (number of trees reduced by 1)
3. Repeat (until we have only 1 tree)

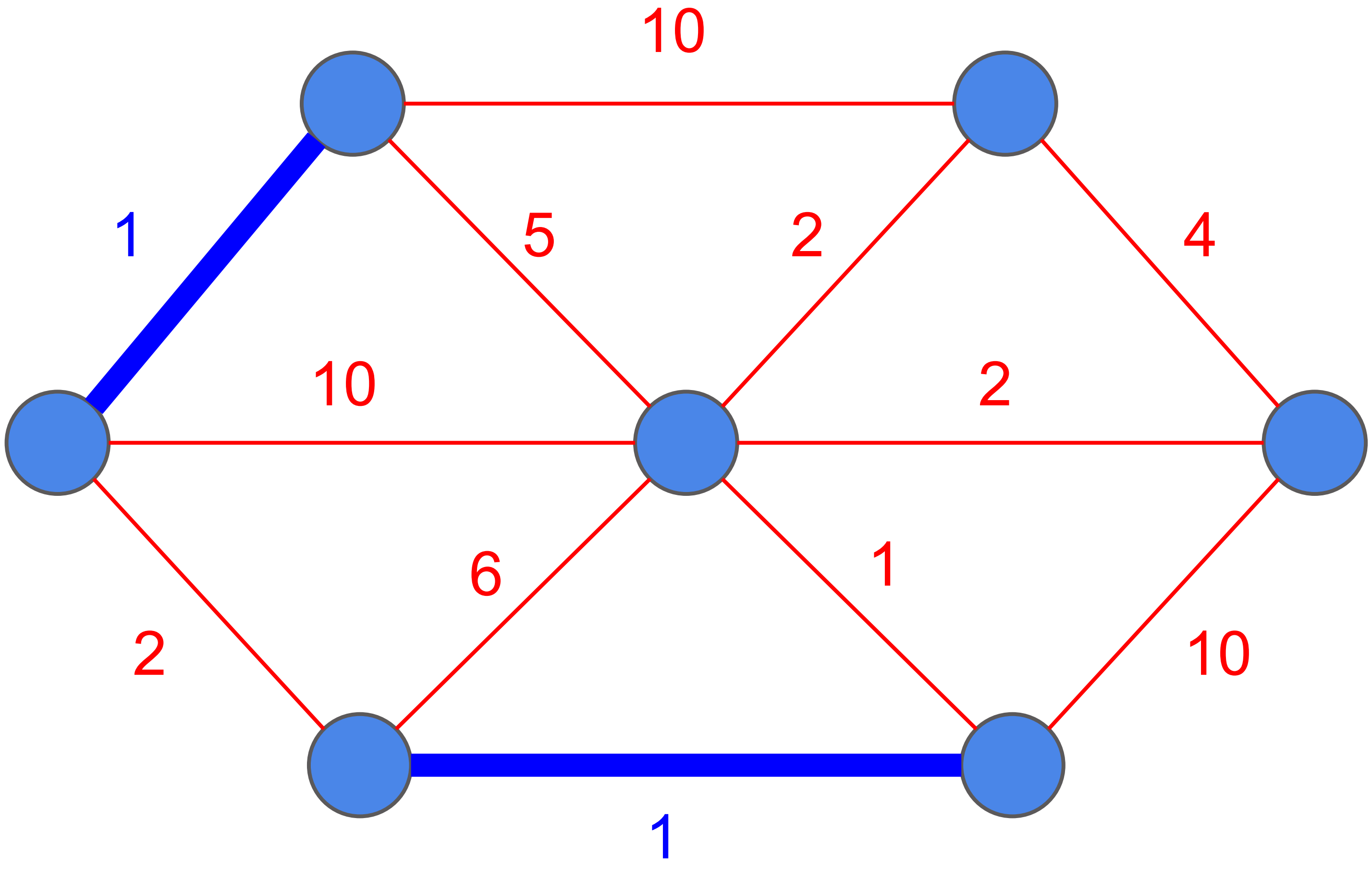
Kruskal's Algorithm



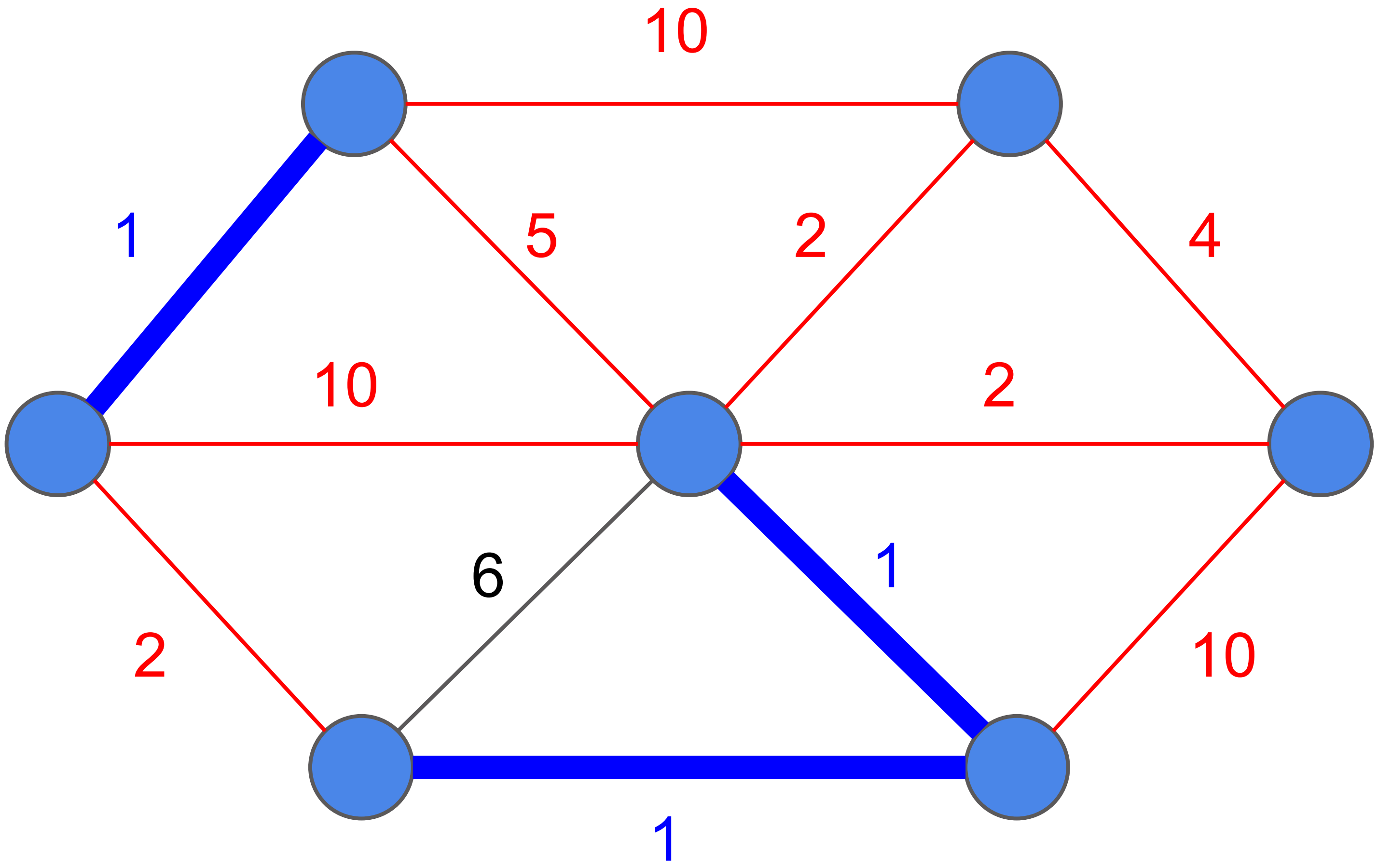
Kruskal's Algorithm



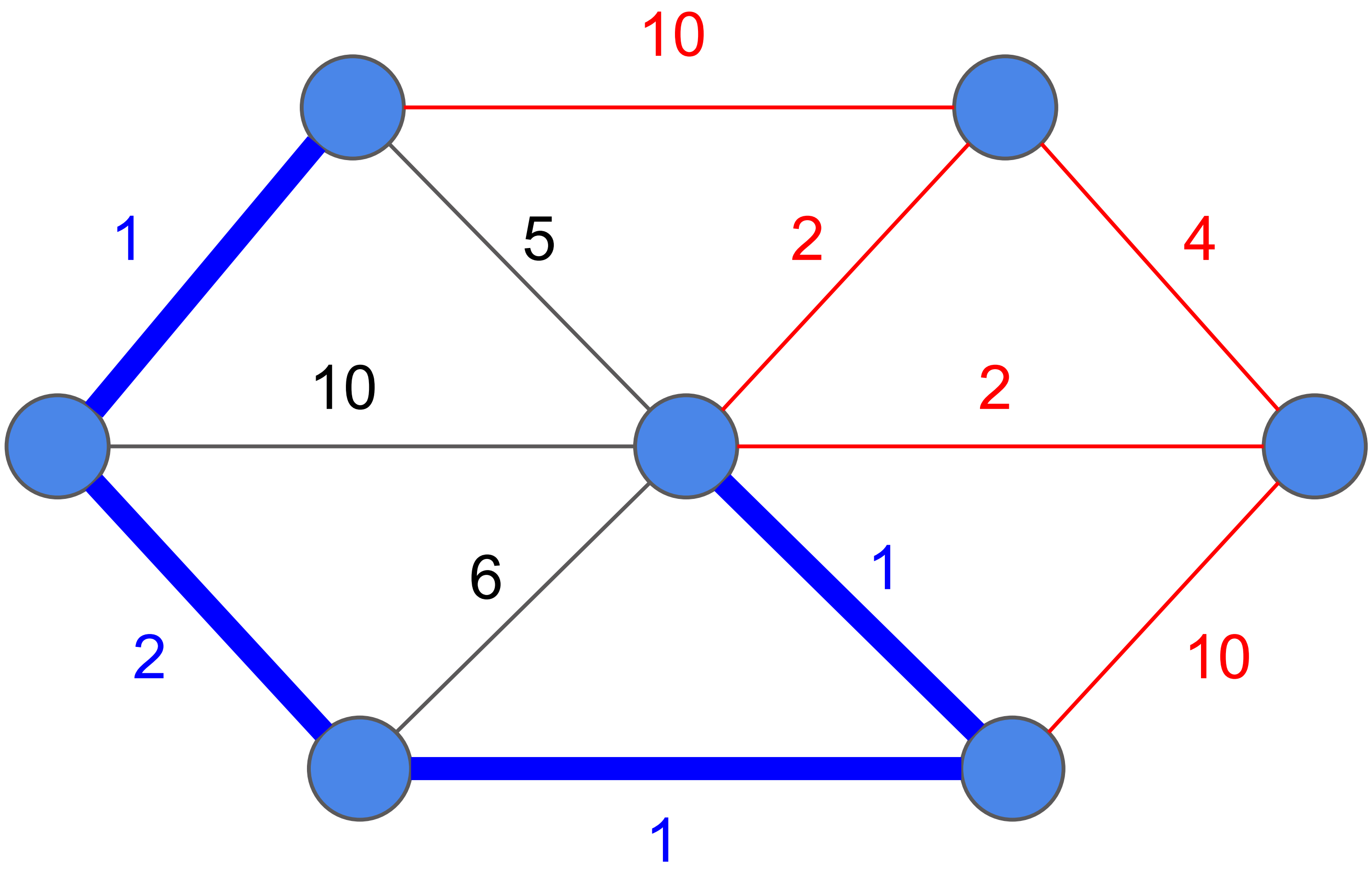
Kruskal's Algorithm



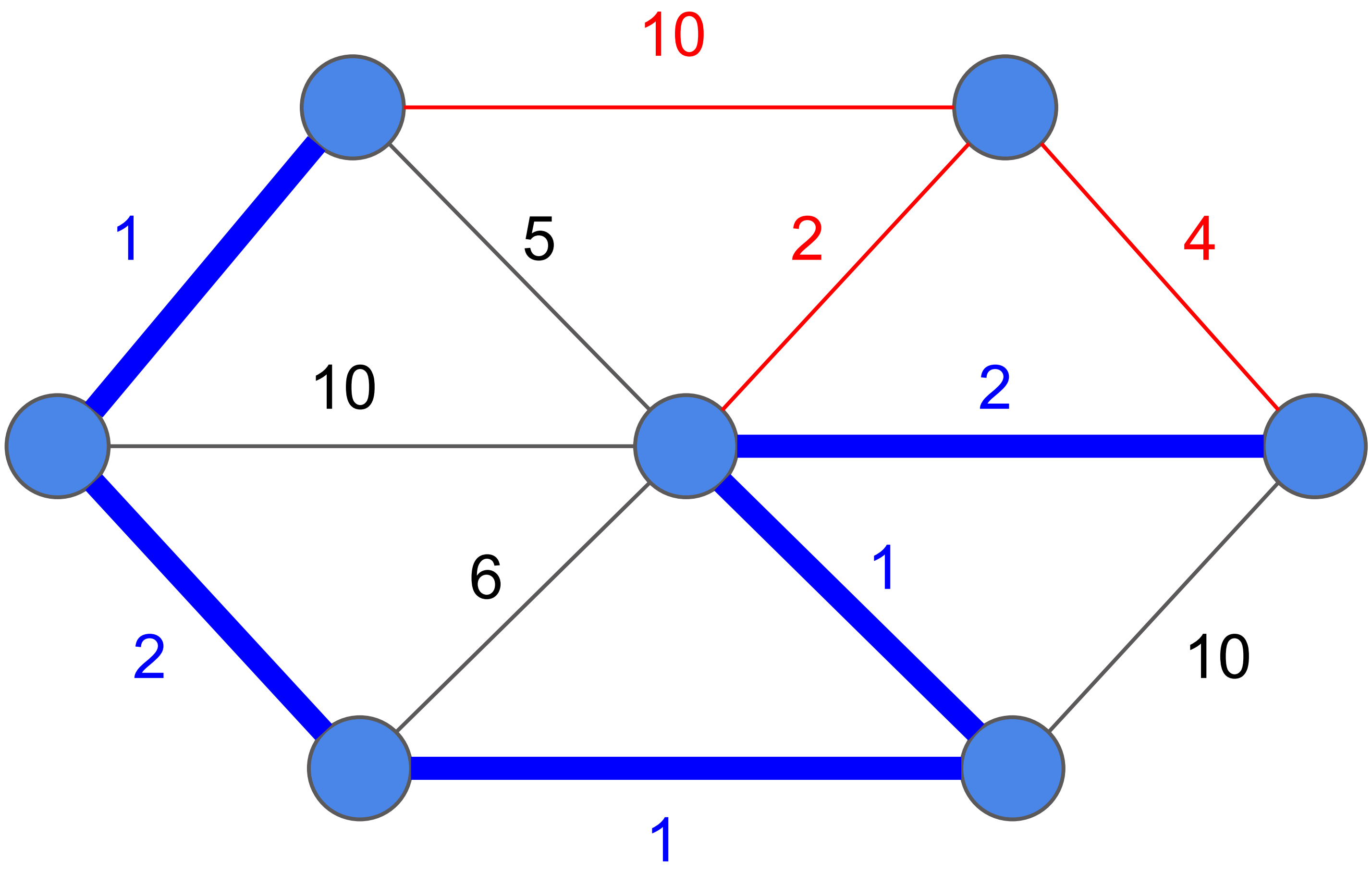
Kruskal's Algorithm



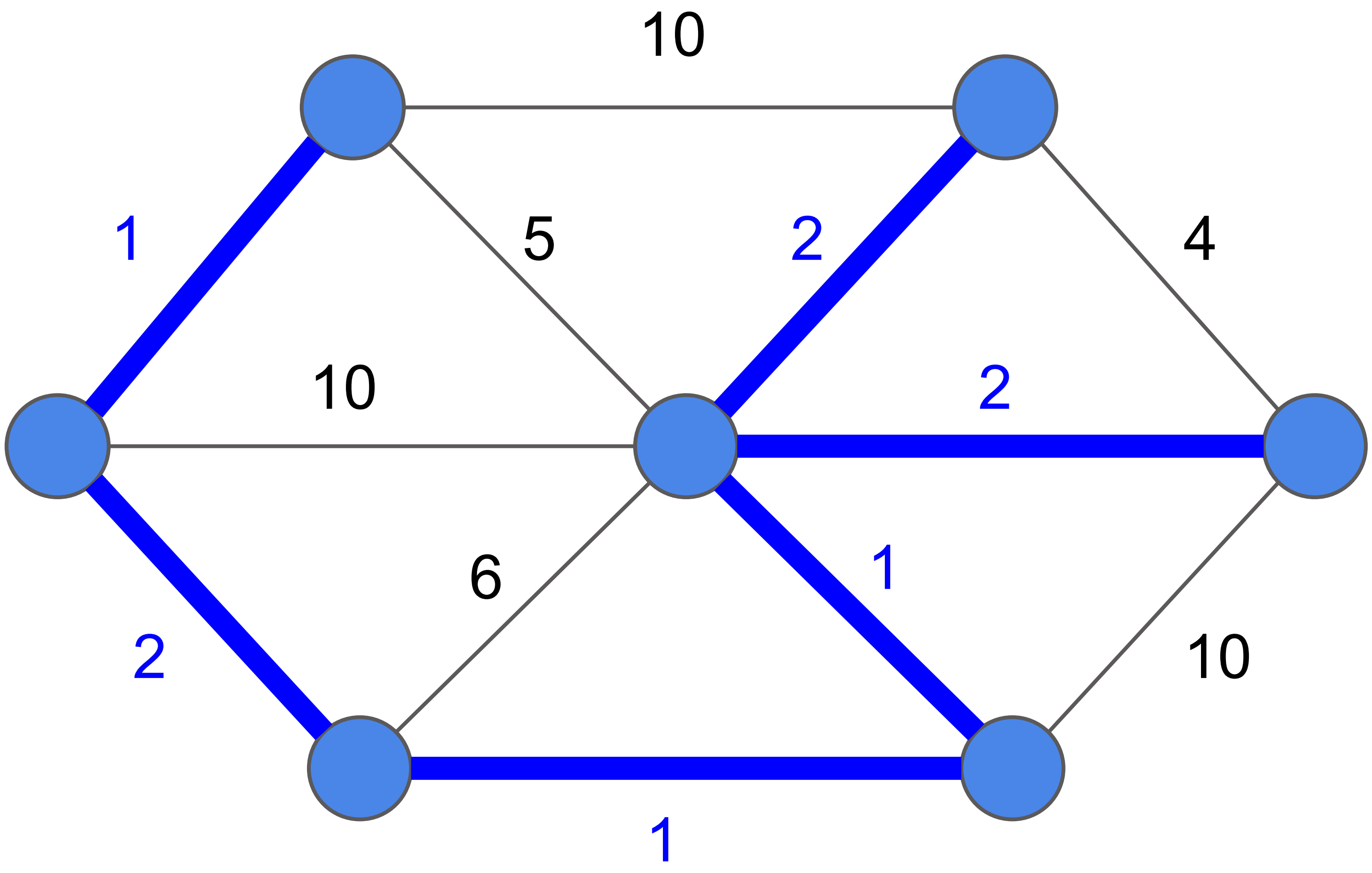
Kruskal's Algorithm



Kruskal's Algorithm



Kruskal's Algorithm



Total cost = 9

Kruskal's Algorithm: Correctness

Essentially same as proof for Prim’s (proof by contradiction)...

Kruskal's Algorithm: Complexity

- Sorting edges by weight:
  - O(|E| log |E|)
- Finding edges that connect different trees and connecting trees (disjoint-set):
  - O(|E| log |V|)
- Total:
  - O(|E| log |E|) = O(|E| log |V|)

Assignment Demo (MST)

MST Implementation Details

- void Starmap::generateMST(...)
  - Your MST algorithm goes in here
- Star class
  - int id - numeric identifier
  - std::vector<Star *> edges - pointers to neighbor stars
  - addMSTEdgeTo(Star * dest) - add MST edge
- std::map<int, Star *> stars
  - key - id of star (get using star->getID())
  - value - pointer to Star object

MST Implementation Details

- Prim's
  - Keep track of stars that are in the tree so far
    - use a map or a set for fast lookup
  - Doesn't quite work if graph is disconnected (the full dataset isn't!)
    - run algorithm on each component
- Kruskal's
  - std::priority_queue for sorting
  - Can store integer in each Star that represents which "tree" it's part of
    - Join trees by changing integers of one subtree
  - Or implement a disjoint-set data structure

Dijkstra's Algorithm - Shortest Path (Single Source)

Idea: Similar to Prim's algorithm; start from source and greedily find shortest paths.

Each vertex has a:

- tentative distance (dist)
  - shortest distance to source found so far
- previous vertex pointer (prev)
  - points to previous vertex in shortest path found so far
  - allows us to reconstruct shortest path

Also, some way to mark “unvisited” vertices

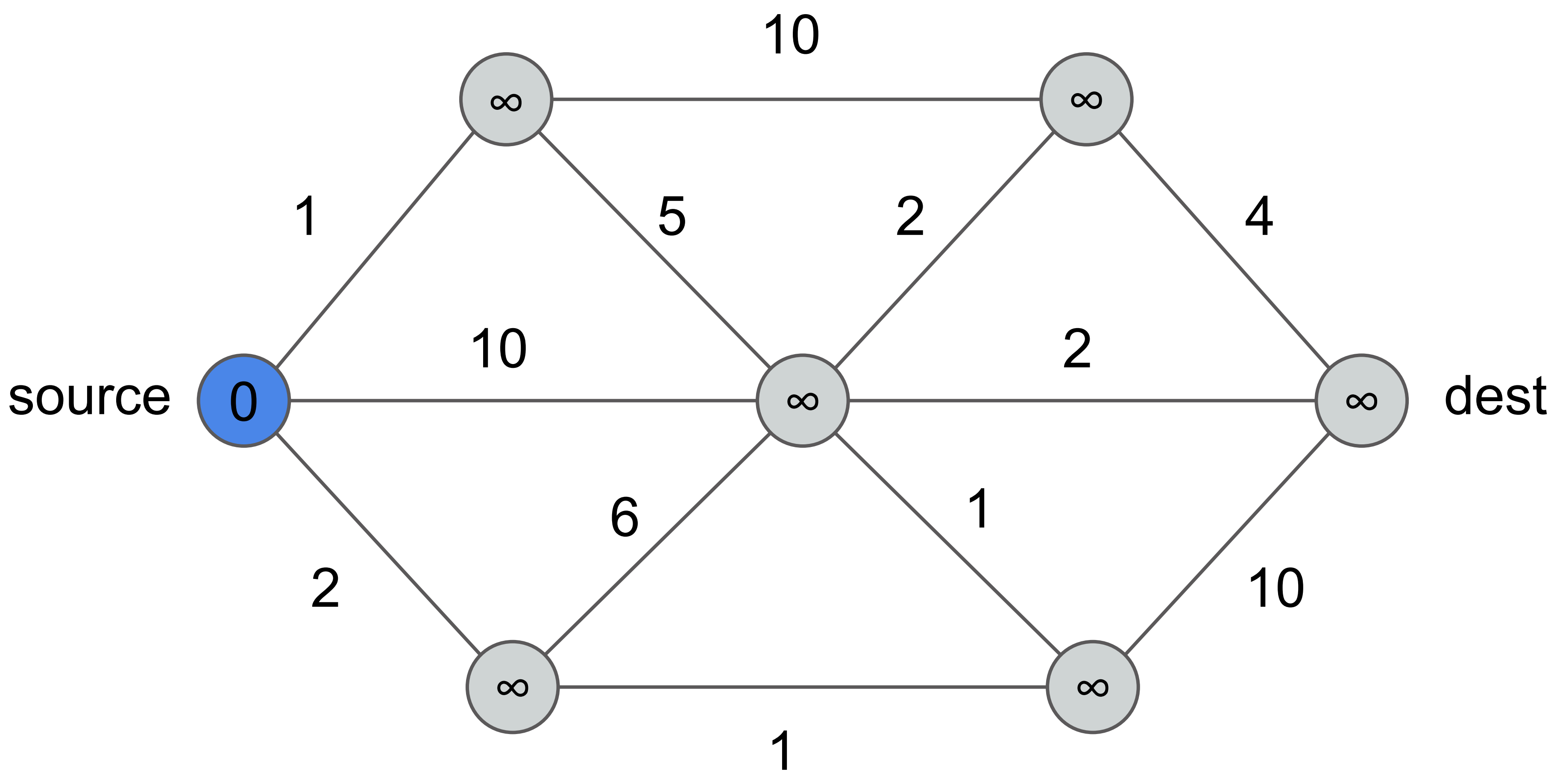
Dijkstra's Algorithm - Shortest Path (Single Source)

```
function Dijkstra(Graph, source):

    dist[source] ← 0                  // Distance from source to source
    prev[source] ← undefined          // Previous node in optimal path initialization

    for each vertex v in Graph:       // Initialization
        if v ≠ source                 // Source has already been initialized
            dist[v] ← infinity        // Unknown distance function from source to v
            prev[v] ← undefined       // Previous node in optimal path from source
        end if
        add v to Q                    // All nodes initially in Q (unvisited nodes)
    end for
```

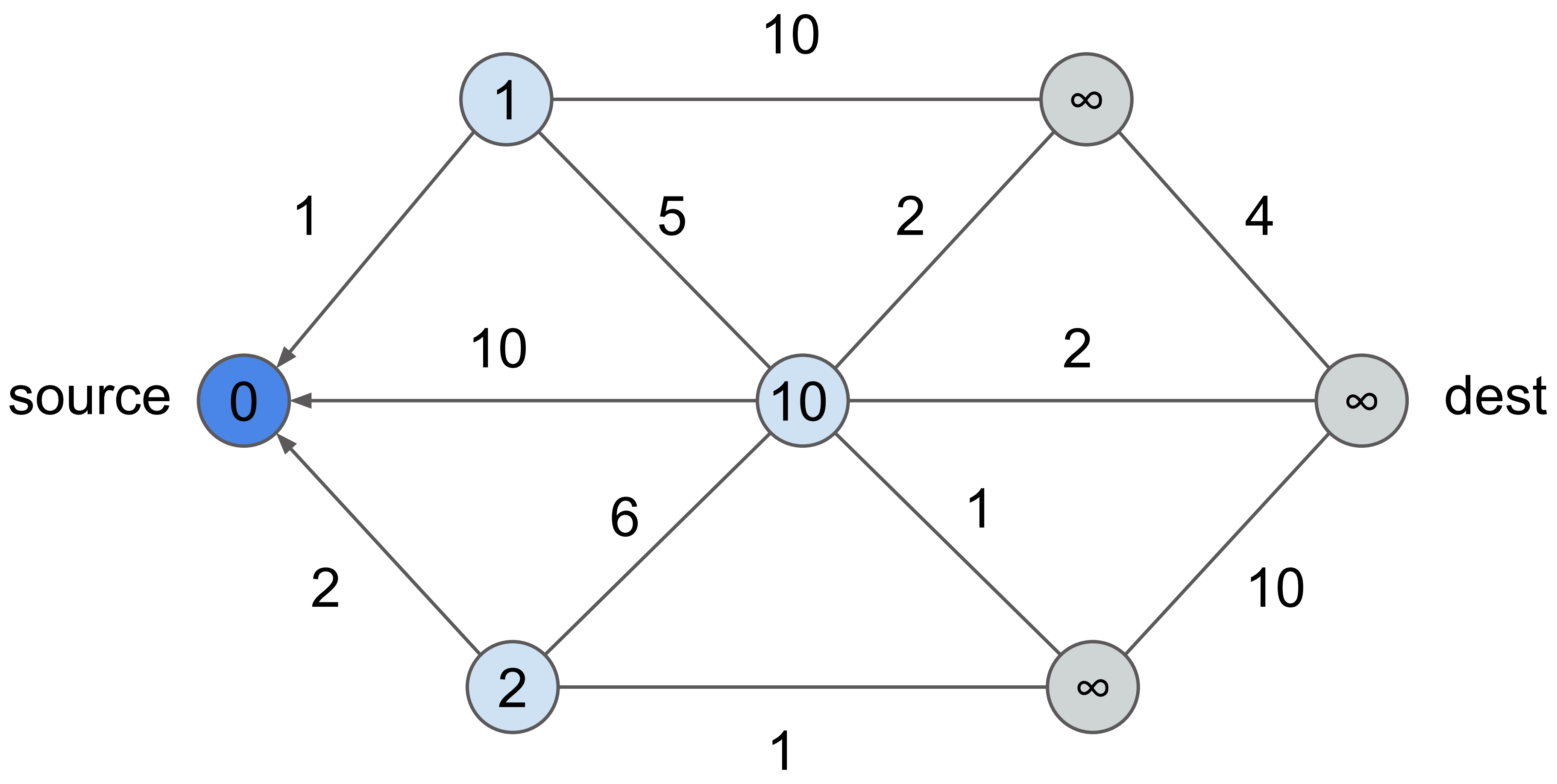
Dijkstra's Algorithm - Shortest Path (Single Source)

```
    while Q is not empty:
        u ← vertex in Q with min dist[u]  // Source node in first case
        remove u from Q

        for each neighbor v of u:           // where v has not yet been removed from Q.
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:               // A shorter path to v has been found
                dist[v] ← alt
                prev[v] ← u
            end if
        end for
    end while

    return dist[], prev[]

end function
```

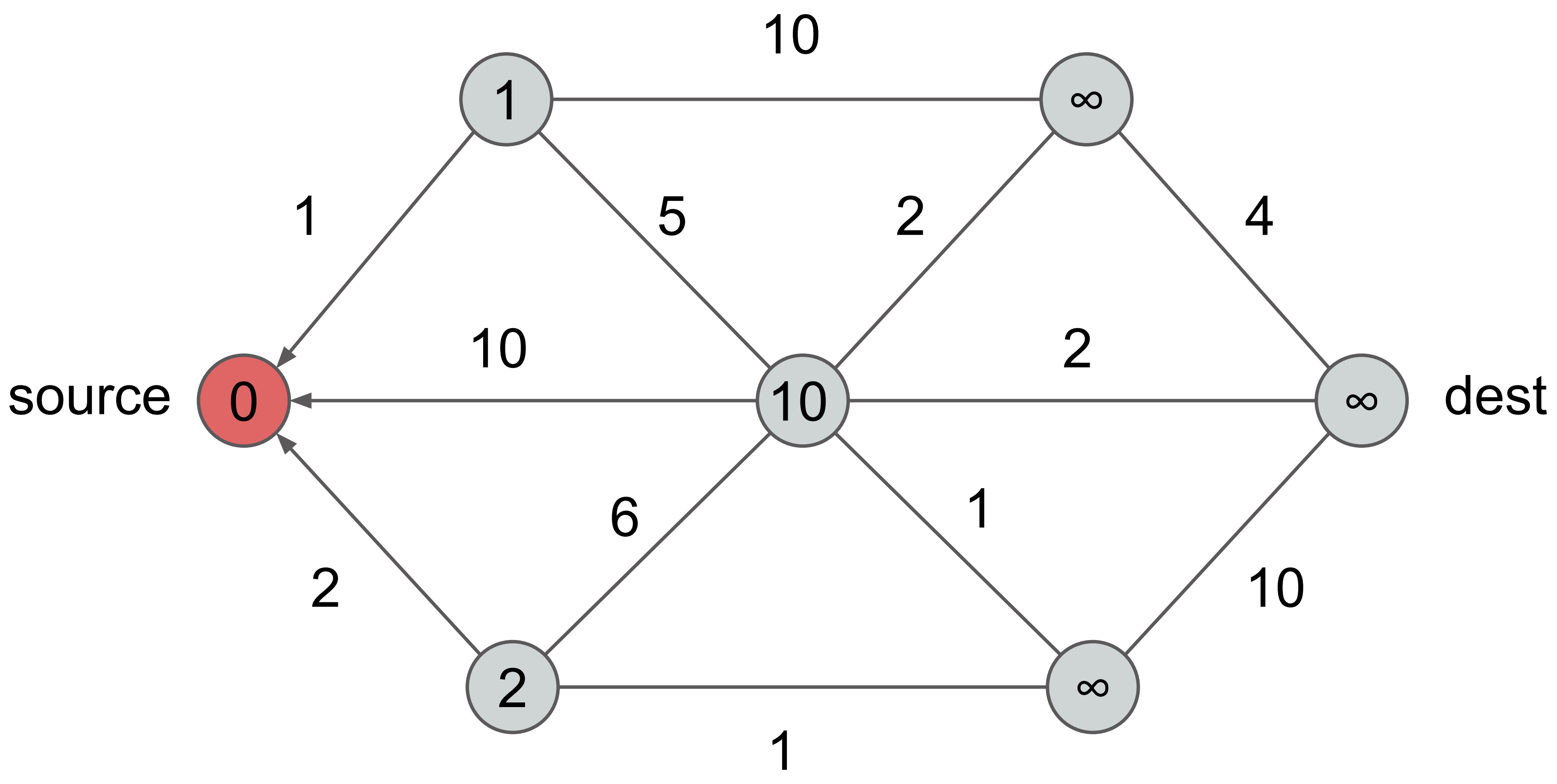
Dijkstra's Algorithm



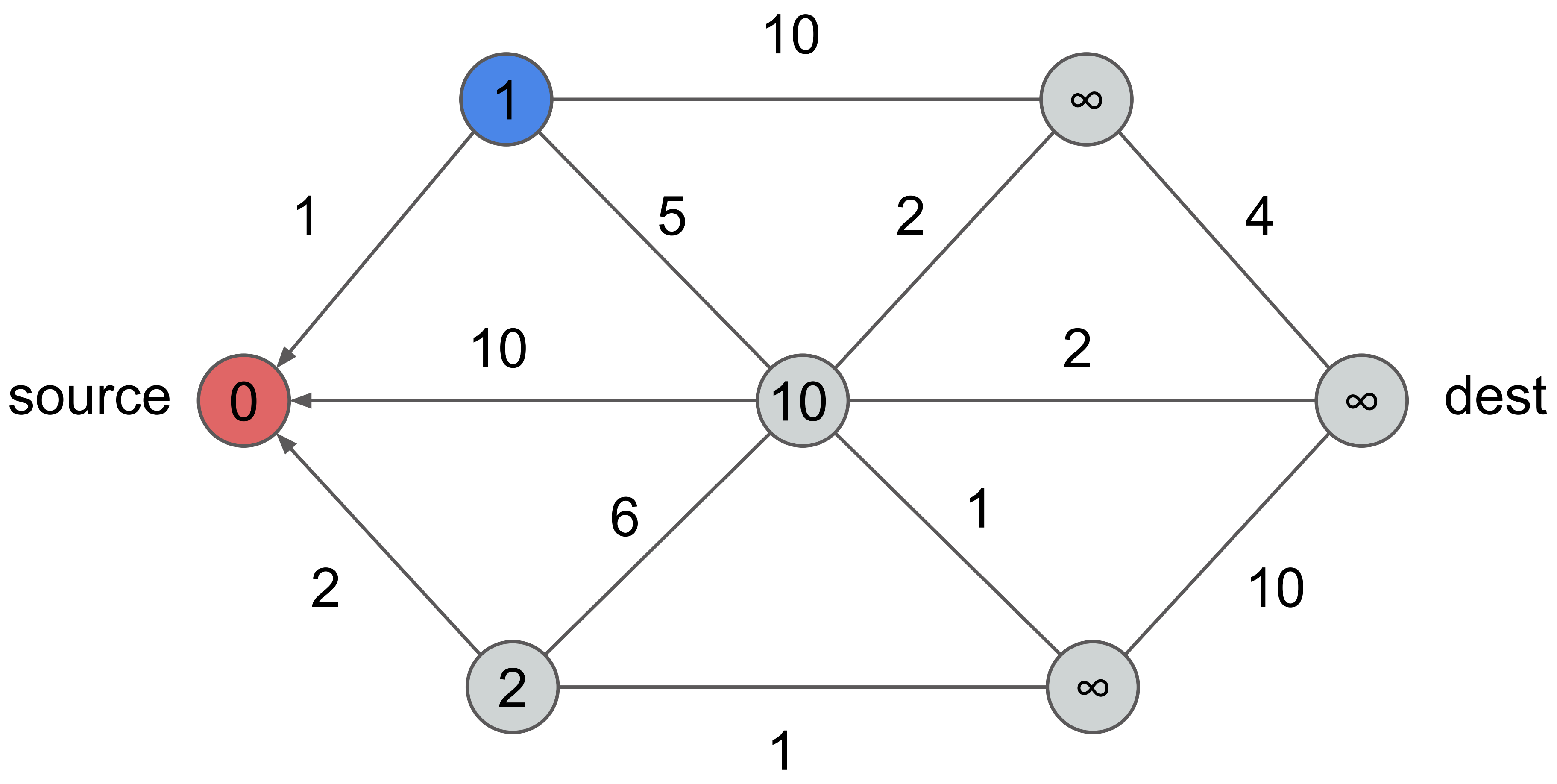
Dijkstra's Algorithm



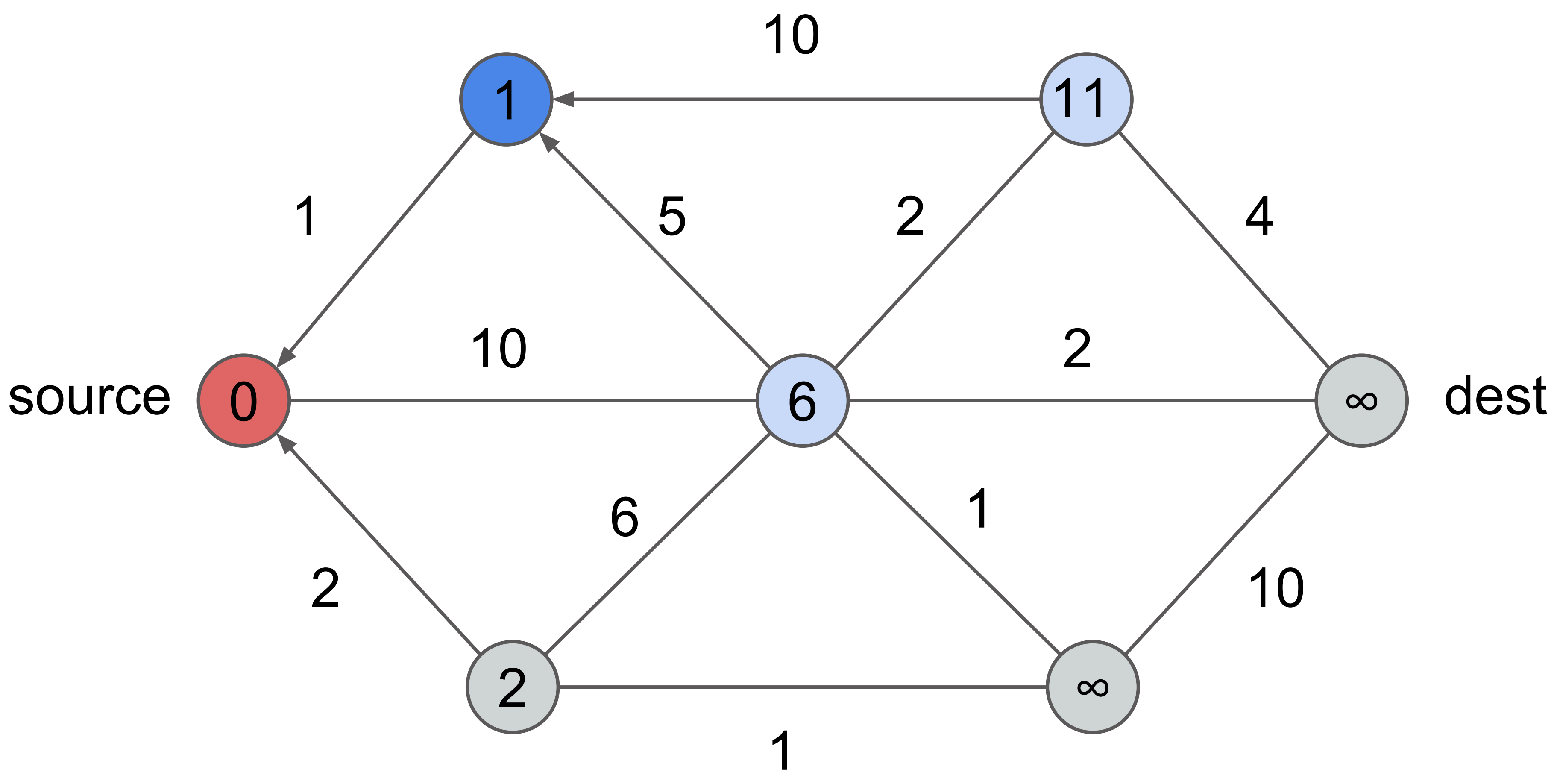
Dijkstra's Algorithm



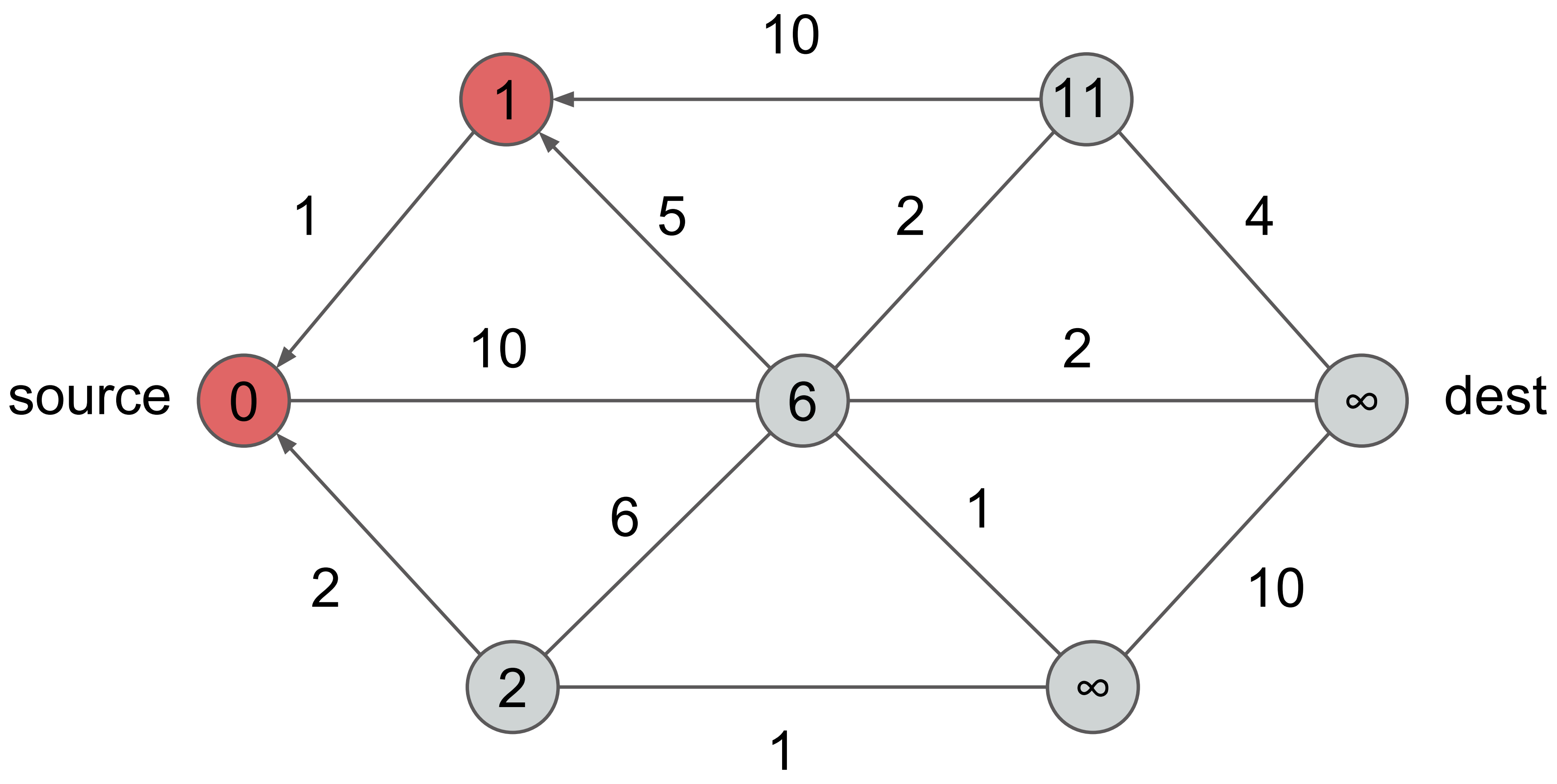
Dijkstra's Algorithm



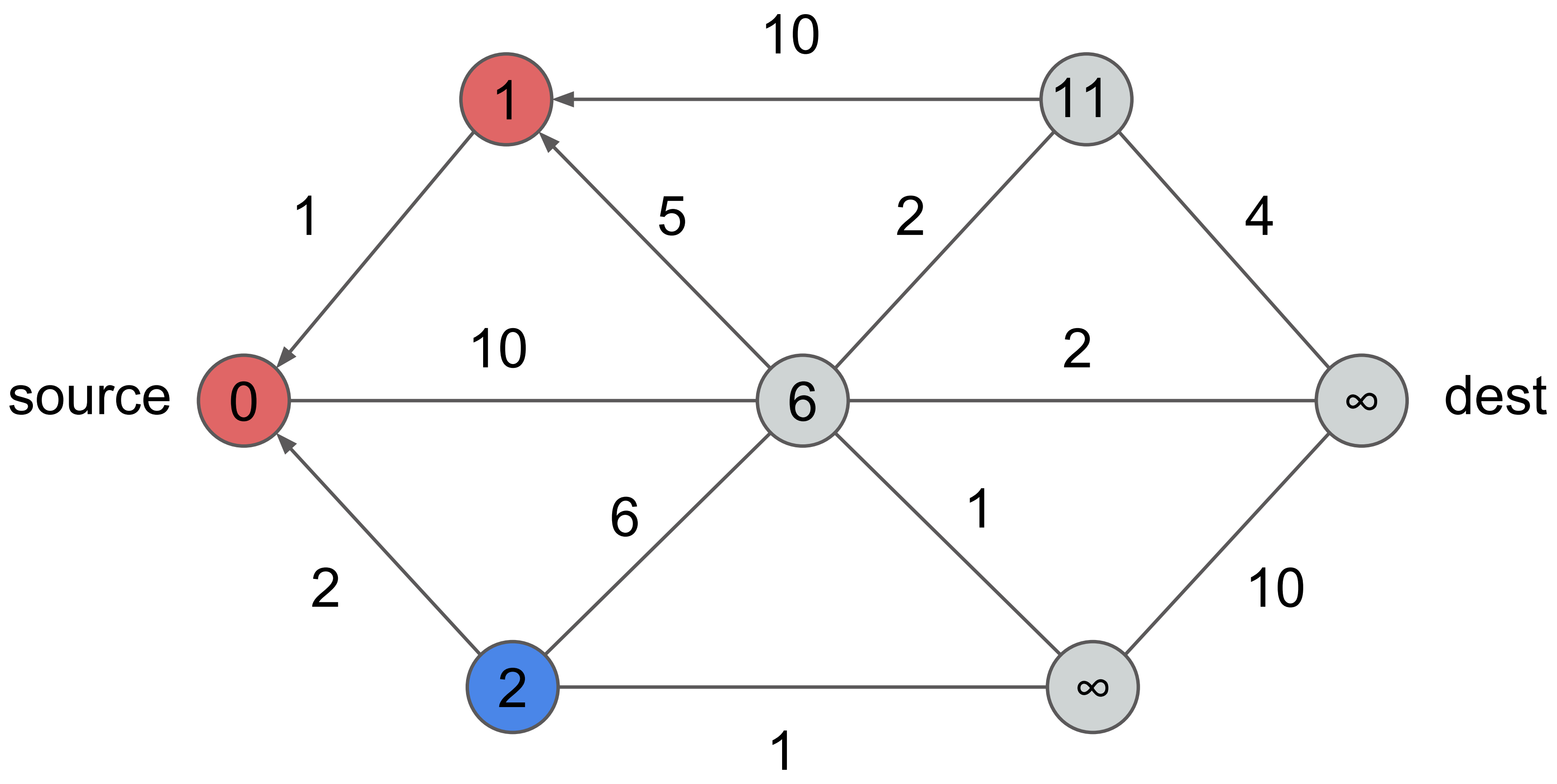
Dijkstra's Algorithm



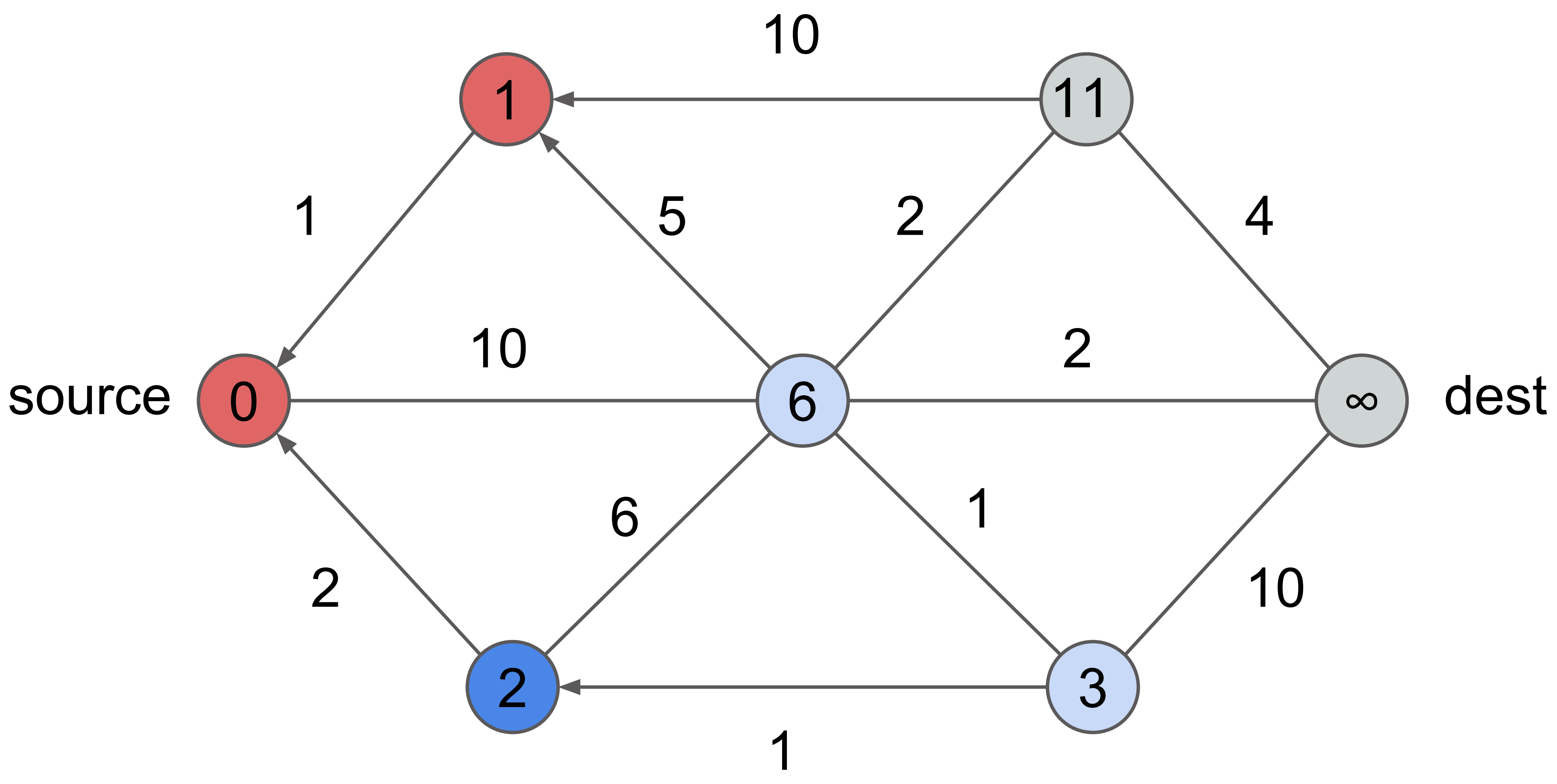
Dijkstra's Algorithm



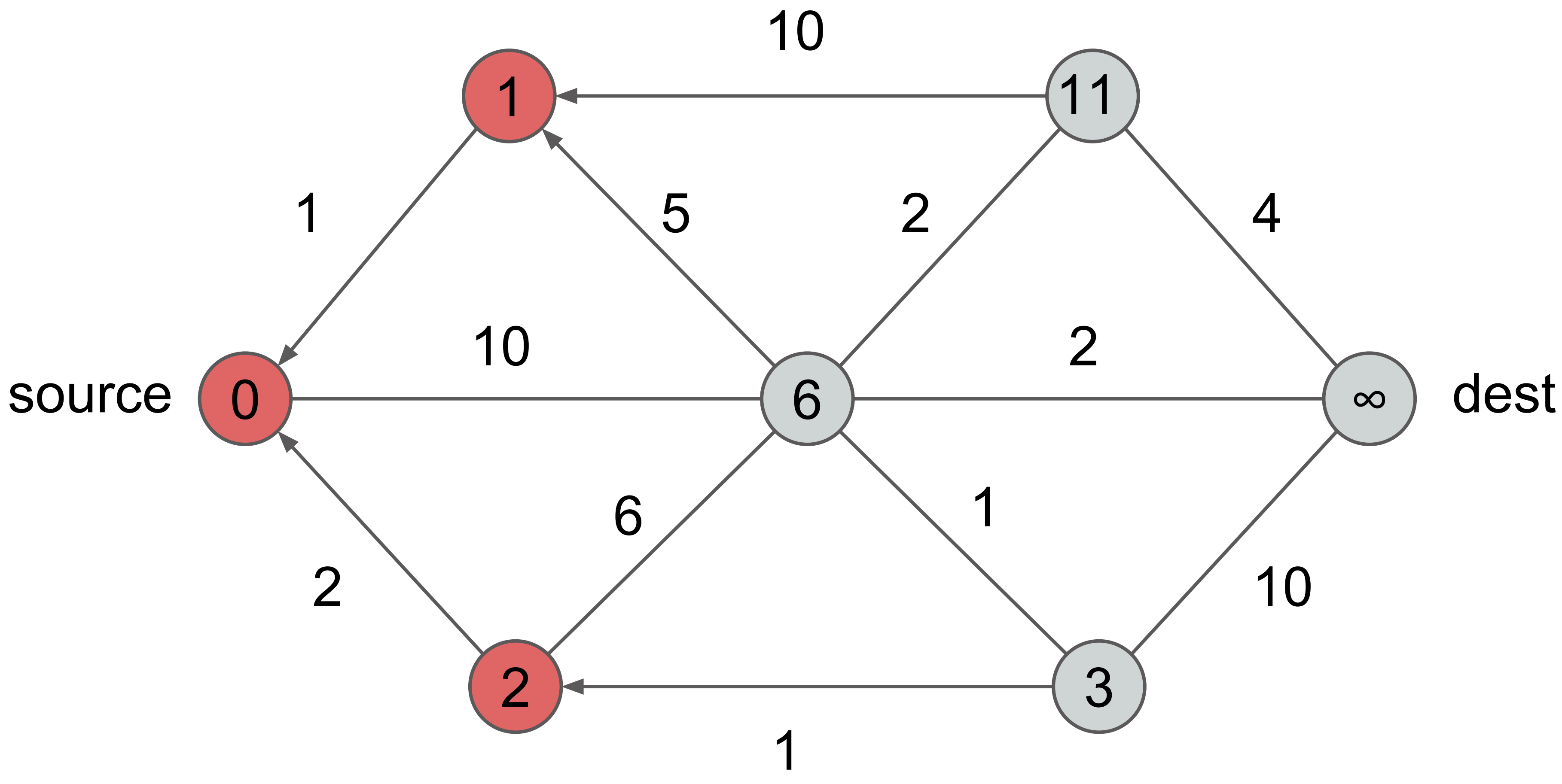
Dijkstra's Algorithm



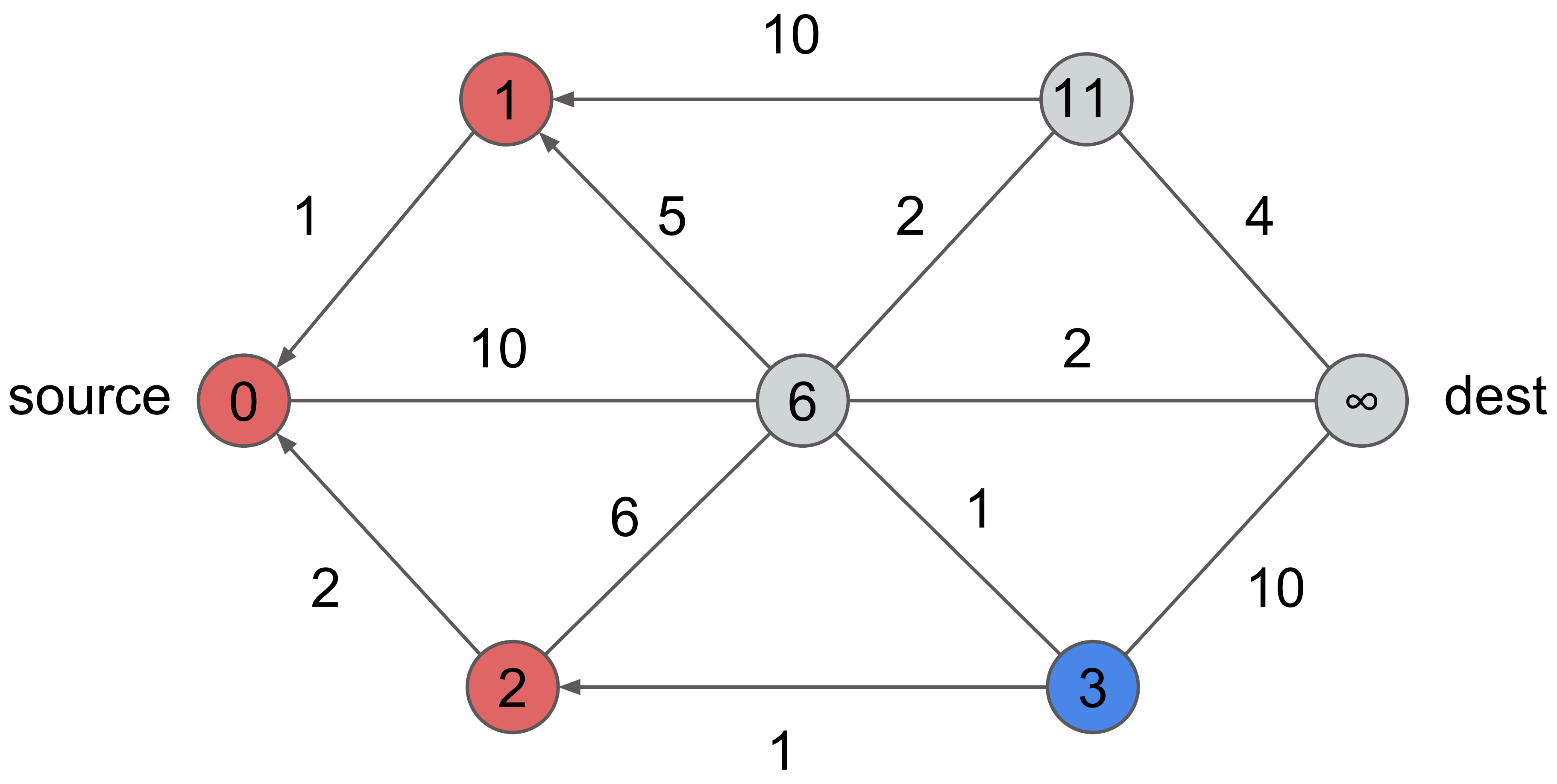
Dijkstra's Algorithm



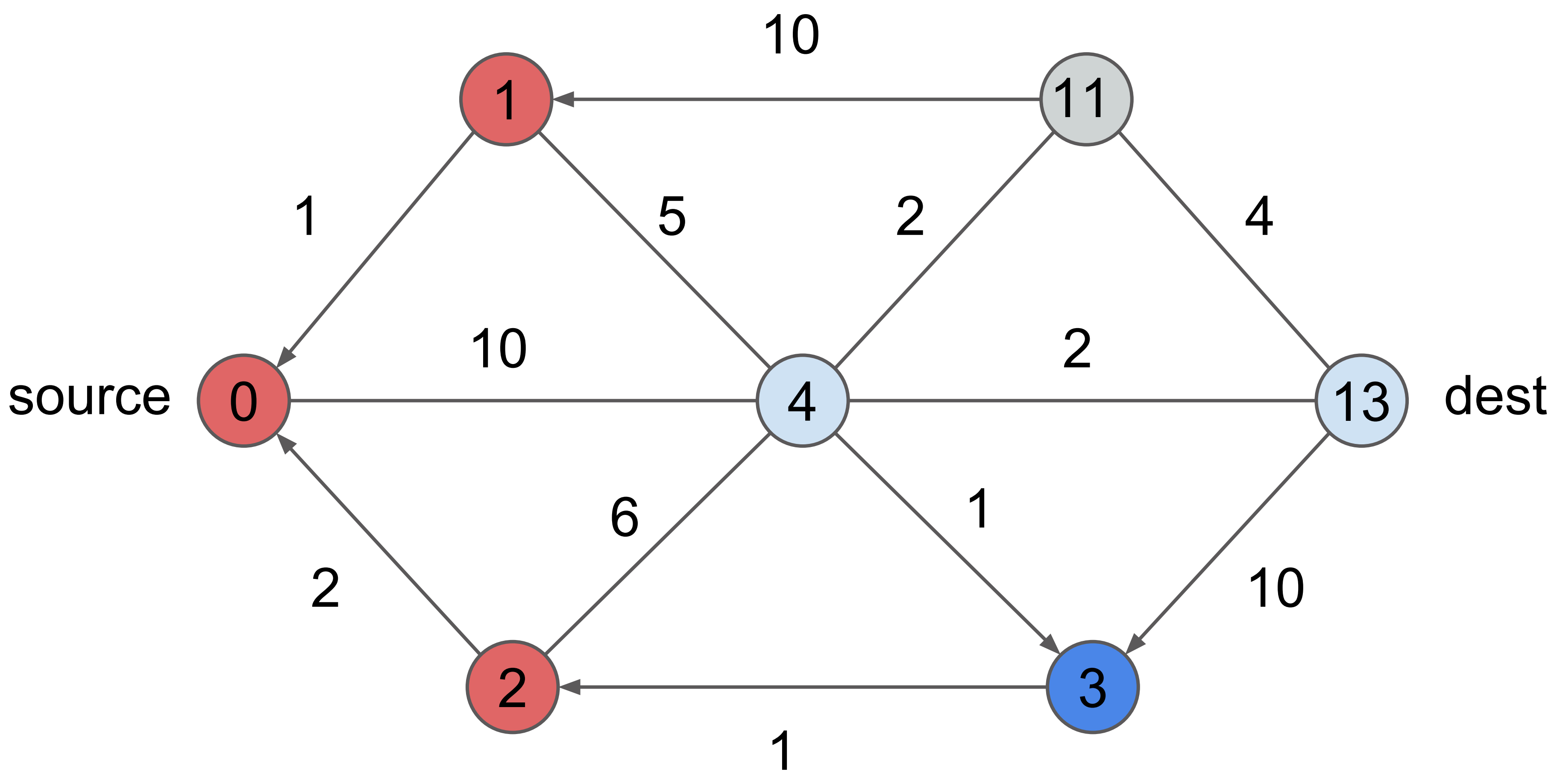
Dijkstra's Algorithm



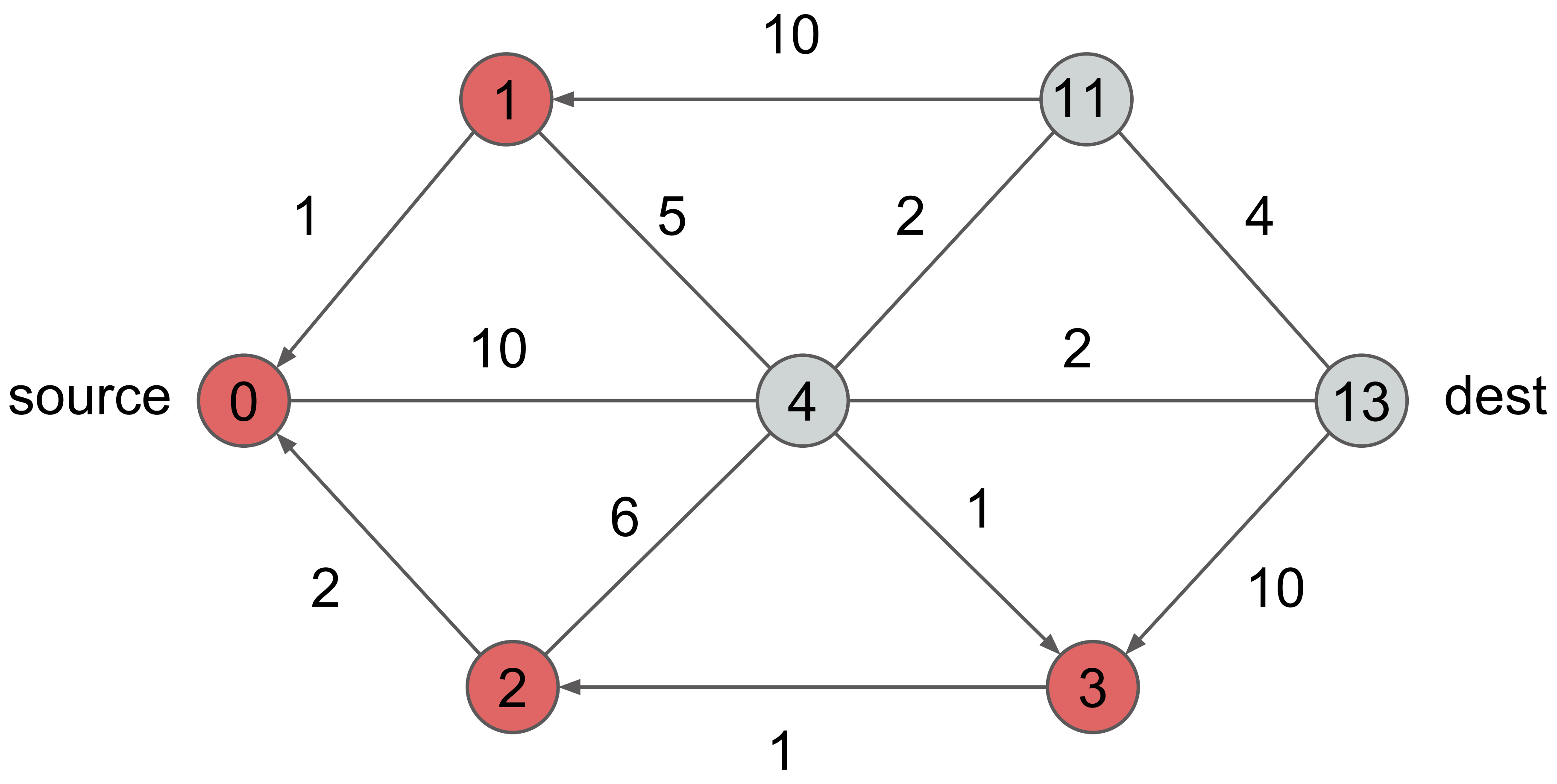
Dijkstra's Algorithm



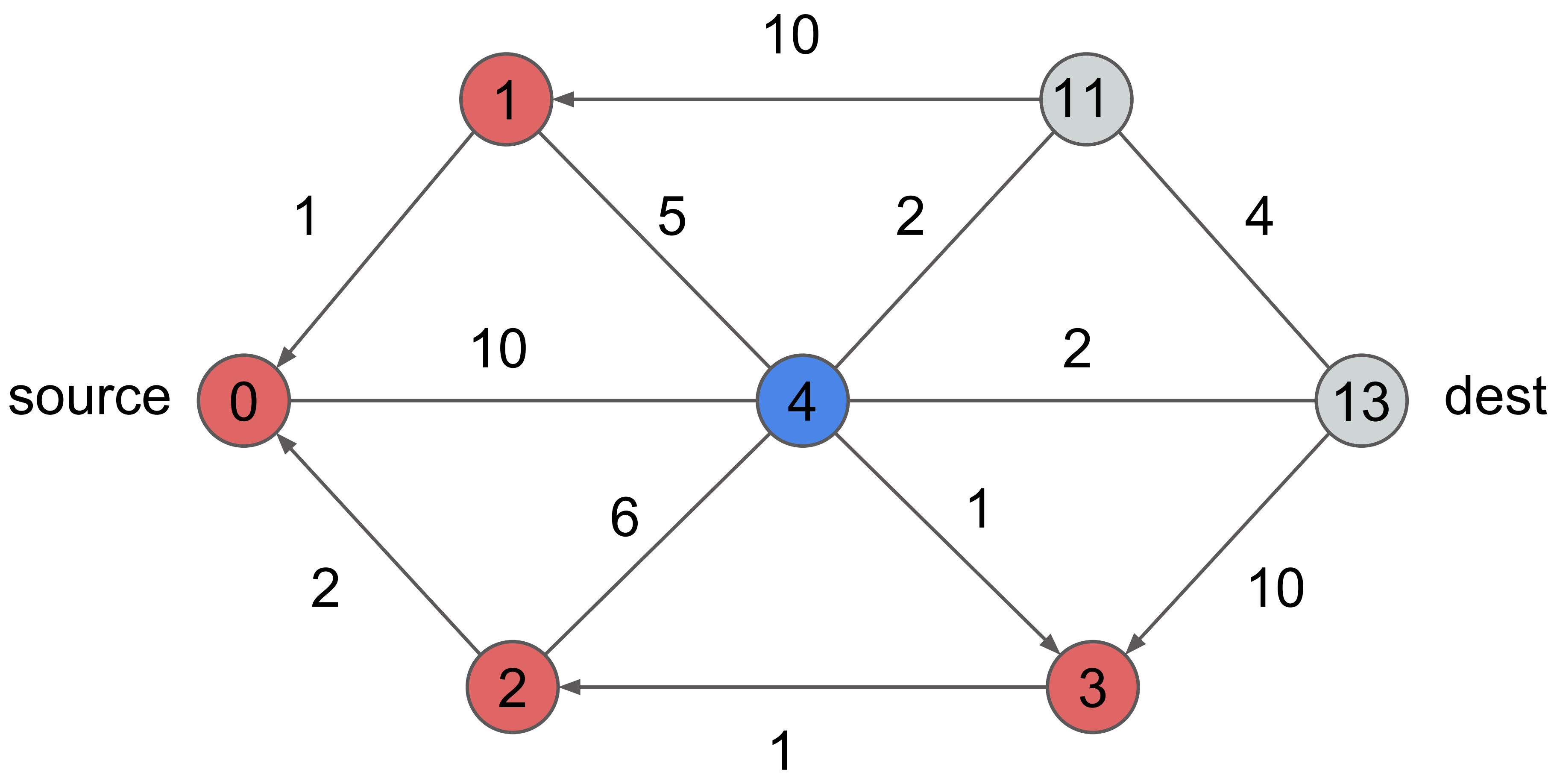
Dijkstra's Algorithm



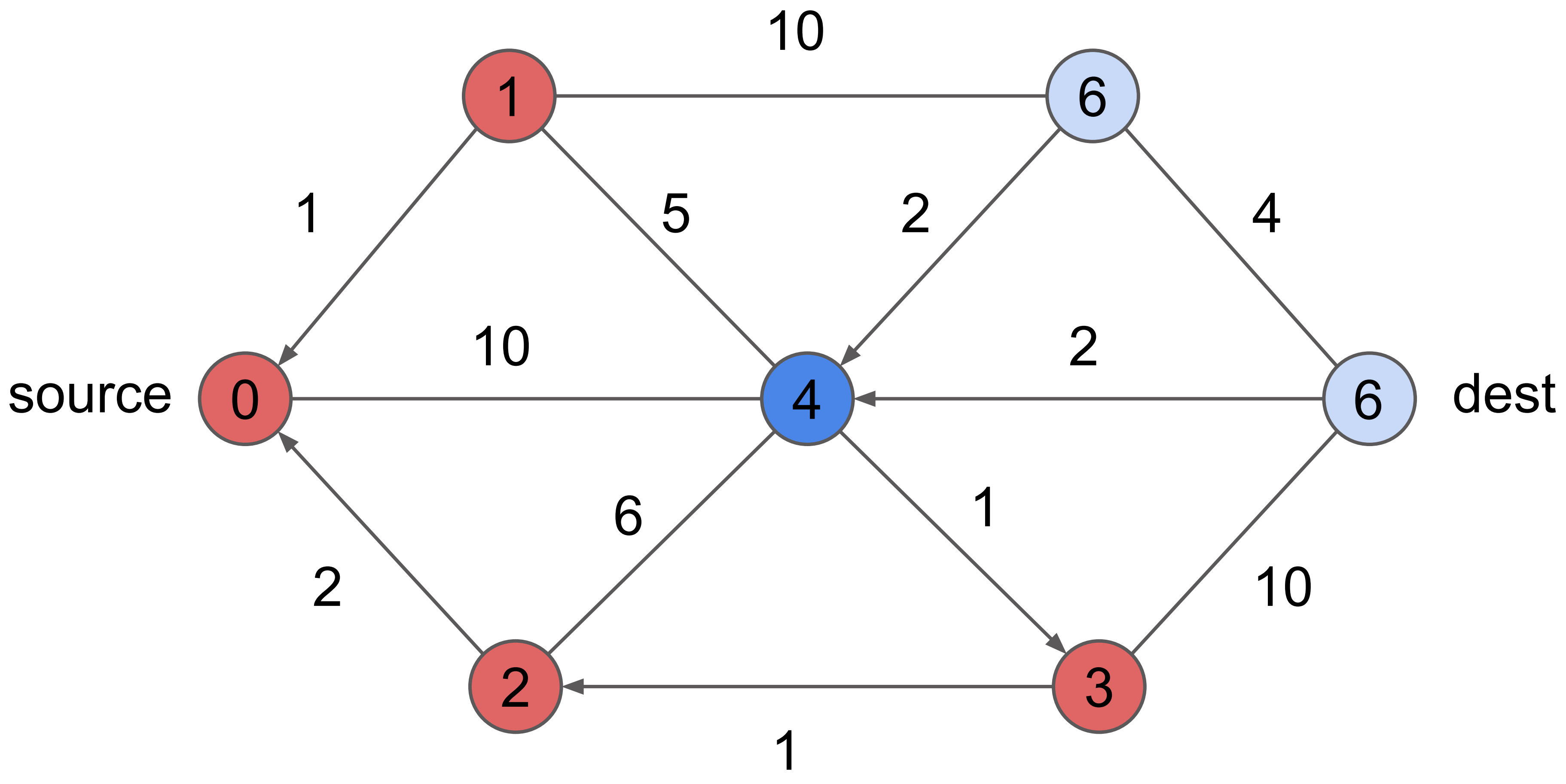
Dijkstra's Algorithm



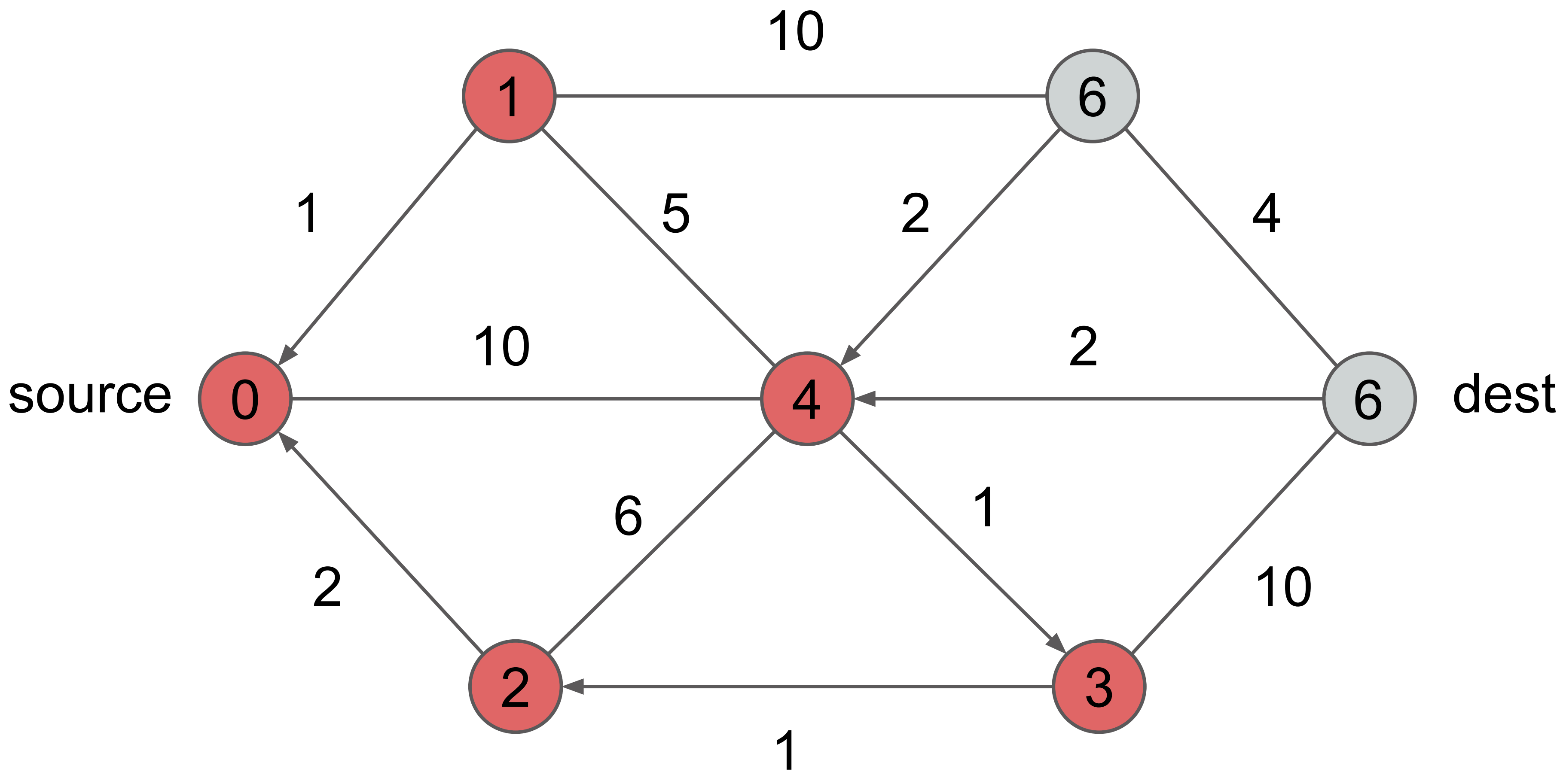
Dijkstra's Algorithm



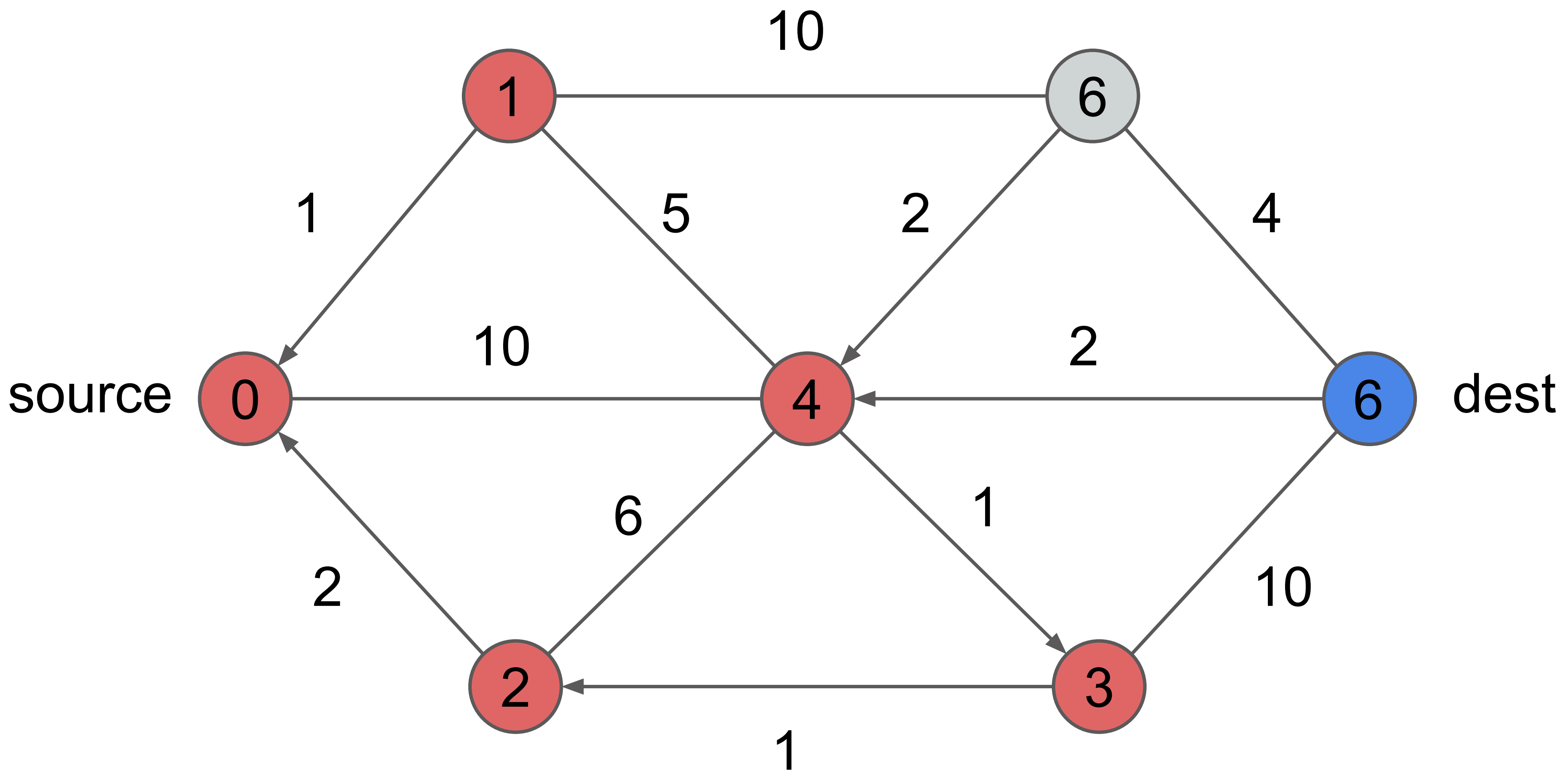
Dijkstra's Algorithm



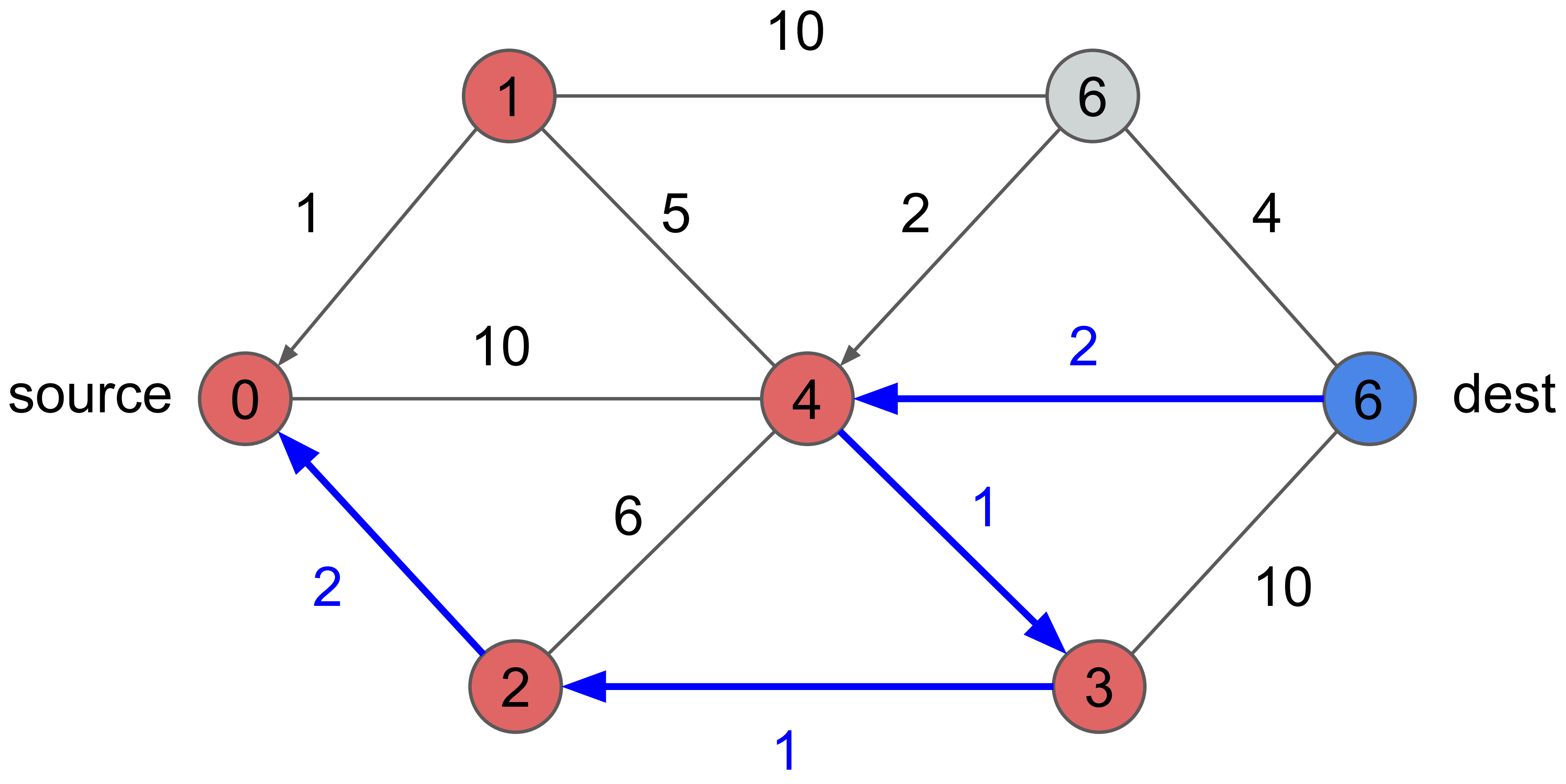
Dijkstra's Algorithm



Dijkstra's Algorithm



Dijkstra's Algorithm



Path length = 6

Assignment Demo (Shortest Path)

Dijkstra's Implementation Details

- std::list<Star*> Starmap::shortestPath(Star * src, Star * dest, CostSpec costs)
  - return list of stars along shortest path (in order)
- tentative distance / previous pointer
  - can add as member variables in Star class
- vector/list of unvisited stars
  - iterate through list to find star with min dist
  - remove stars from list to mark them as visited
  - this is slow - bonus points if you implement a heap to speed this up!

Questions?