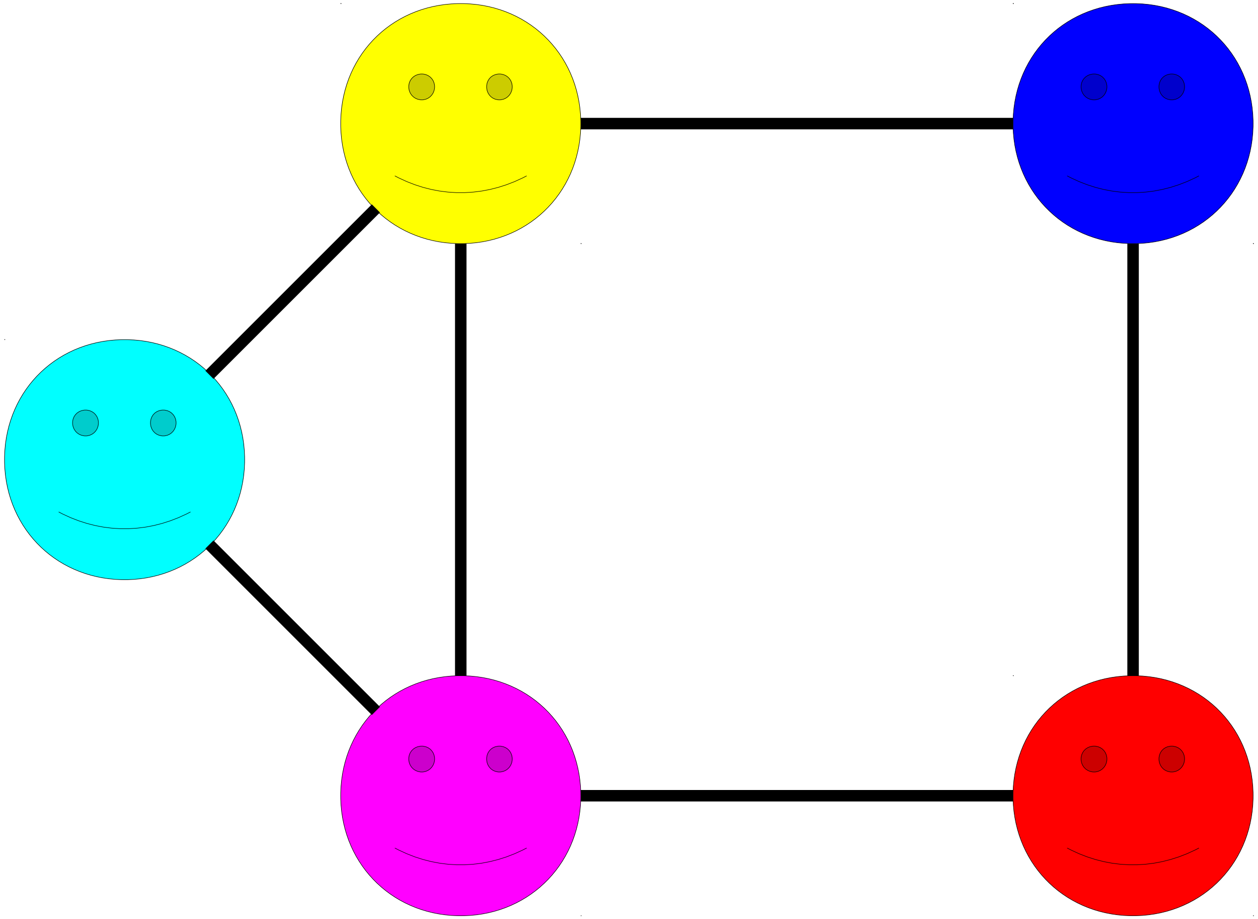
Graph Representations and Algorithms

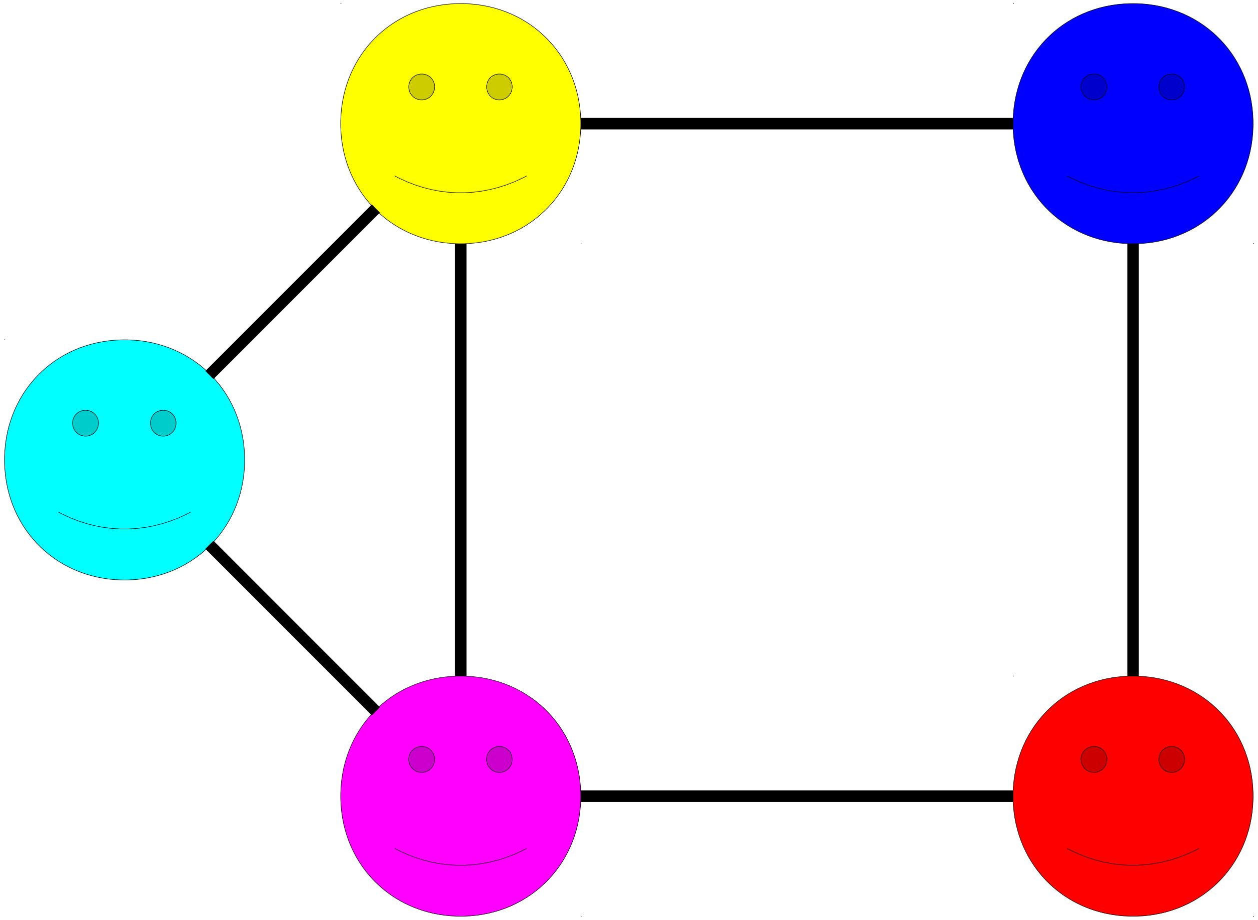
Announcements

- Second midterm is tomorrow, **Thursday, May 31**.
- Exam location by last name:
  - A – F: Go to Hewlett 201.
  - G – Z: Go to Hewlett 200.
- Covers material up through and including Friday's lecture.
- Comprehensive, but primarily focuses on algorithmic efficiency and data structures.

A **graph** is a mathematical structure for representing relationships.

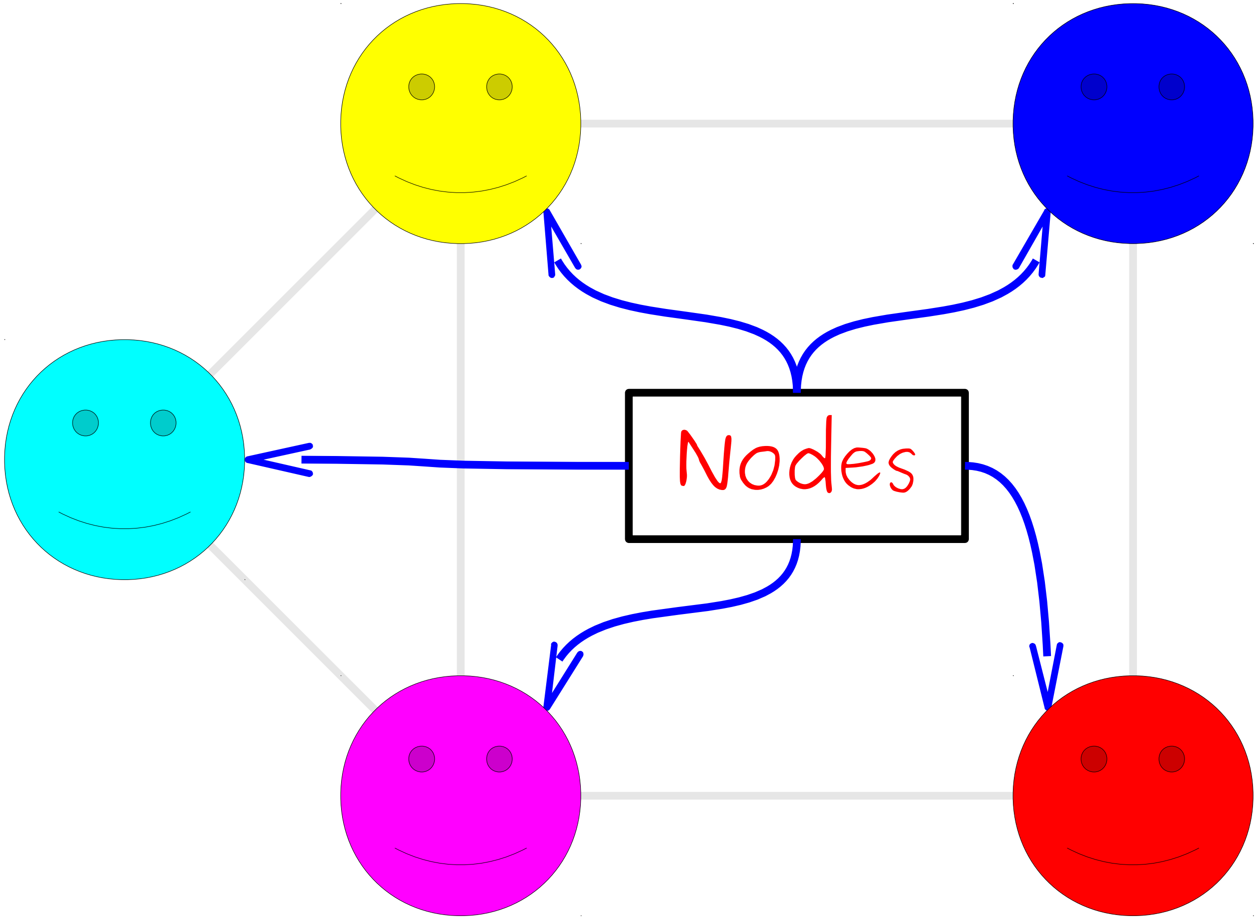


A **graph** is a mathematical structure for representing relationships.



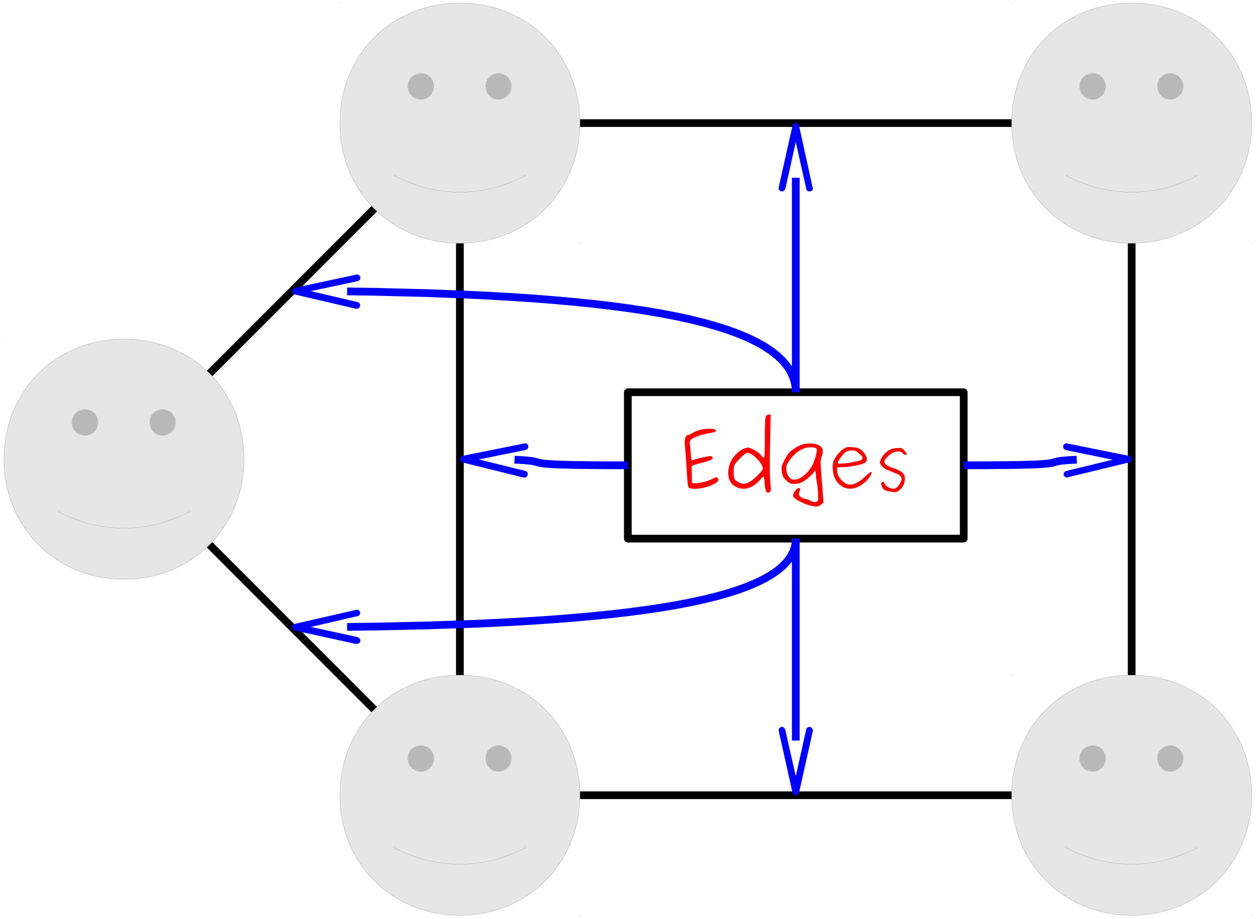
A graph consists of a set of **nodes** connected by **edges**.

A **graph** is a mathematical structure for representing relationships.



A graph consists of a set of **nodes** connected by **edges**.

A **graph** is a mathematical structure for representing relationships.



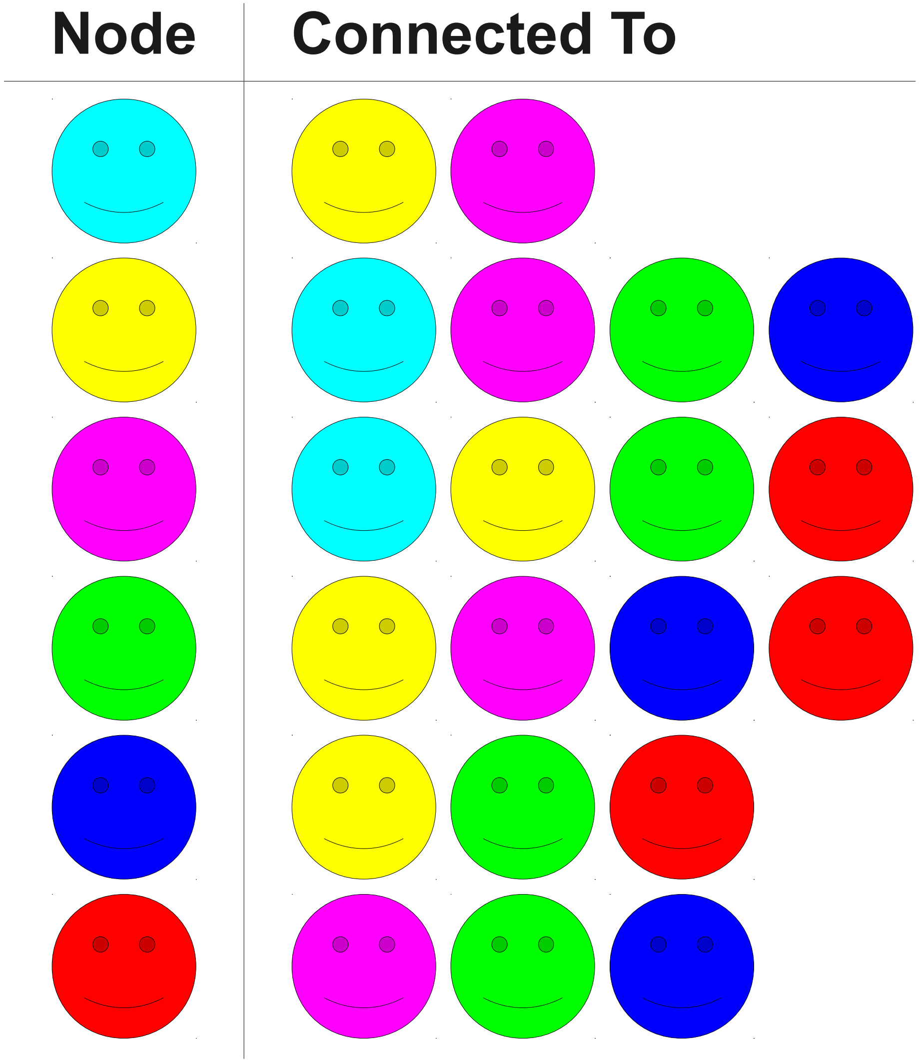
A graph consists of a set of **nodes** connected by **edges**.

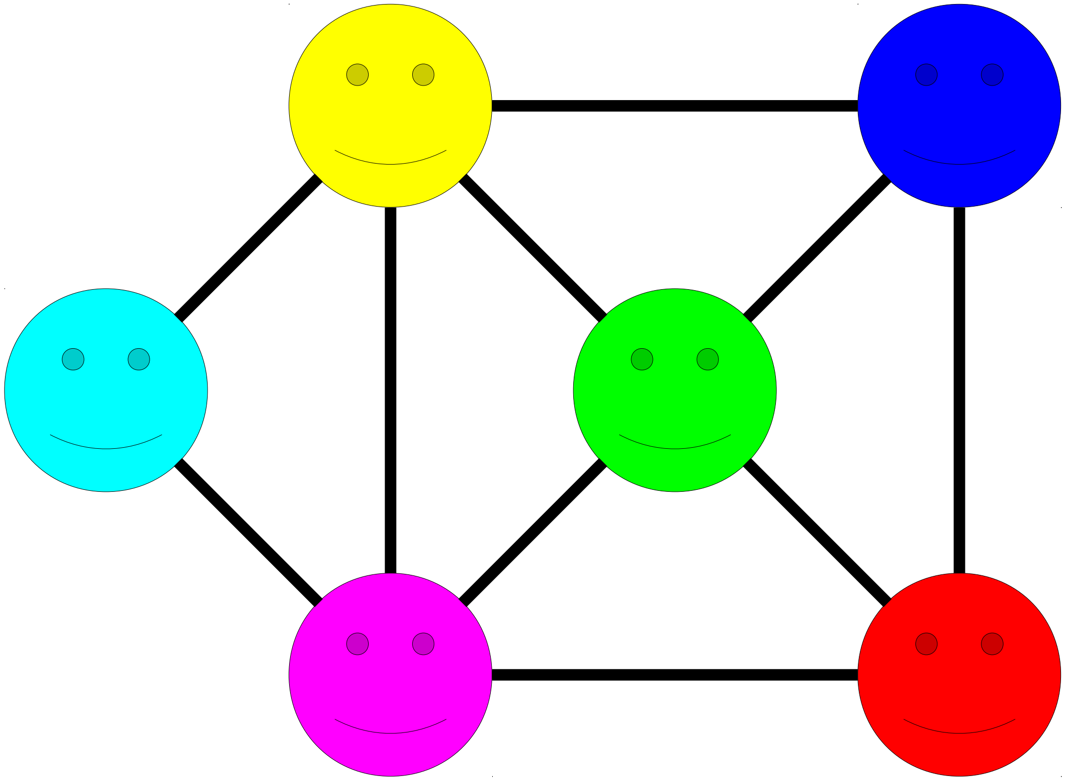
Representing Graphs

We can represent a graph as a map from nodes to the list of nodes each node is connected to.

`Map<Node*, Vector<Node*> >`

`Node*` `Vector<Node*>`



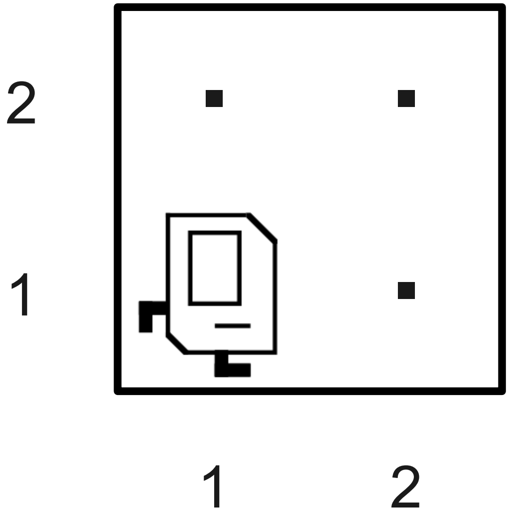


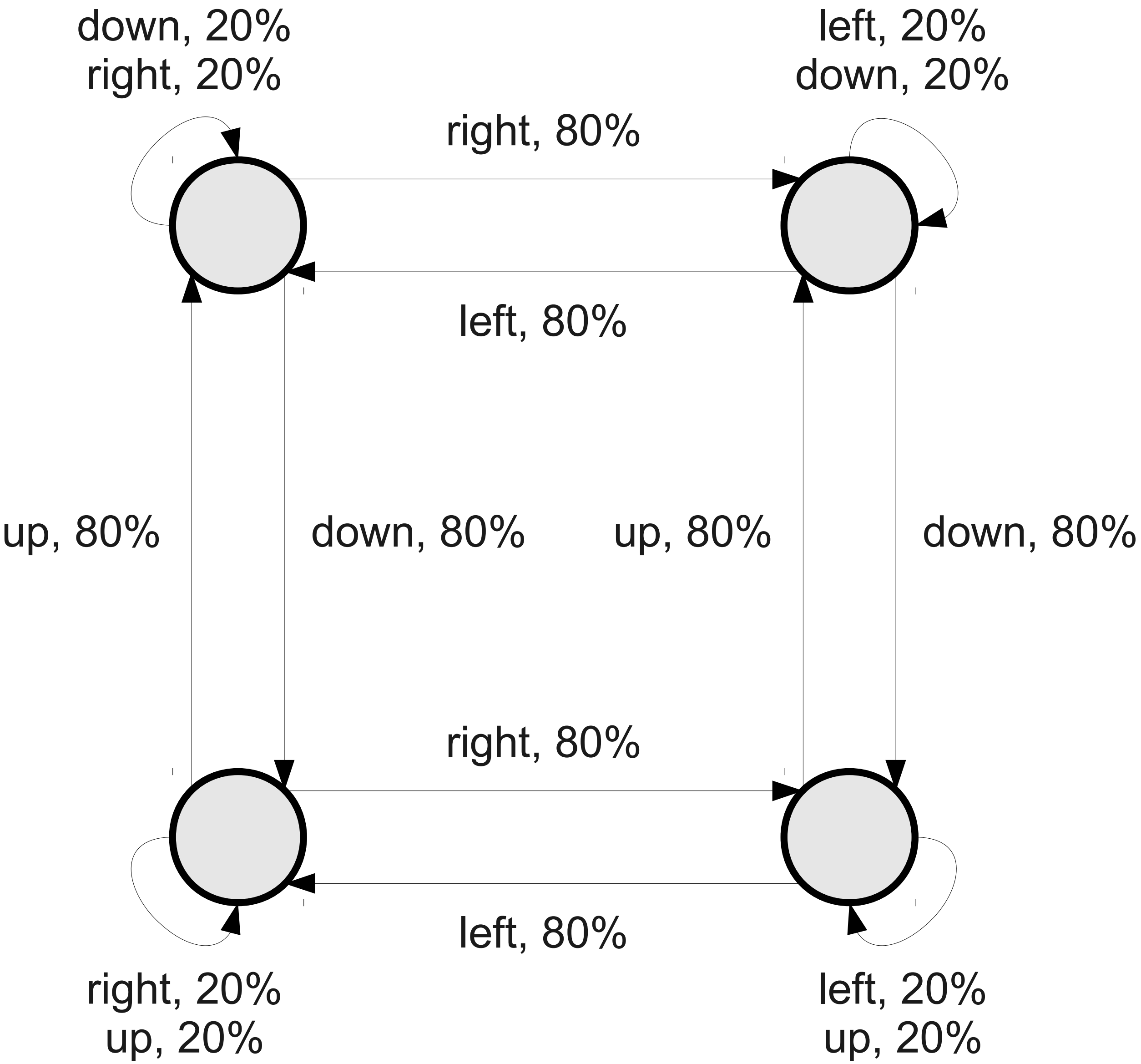
As The Crow Flies

How would we represent this graph?

Karel Goes Ice Skating

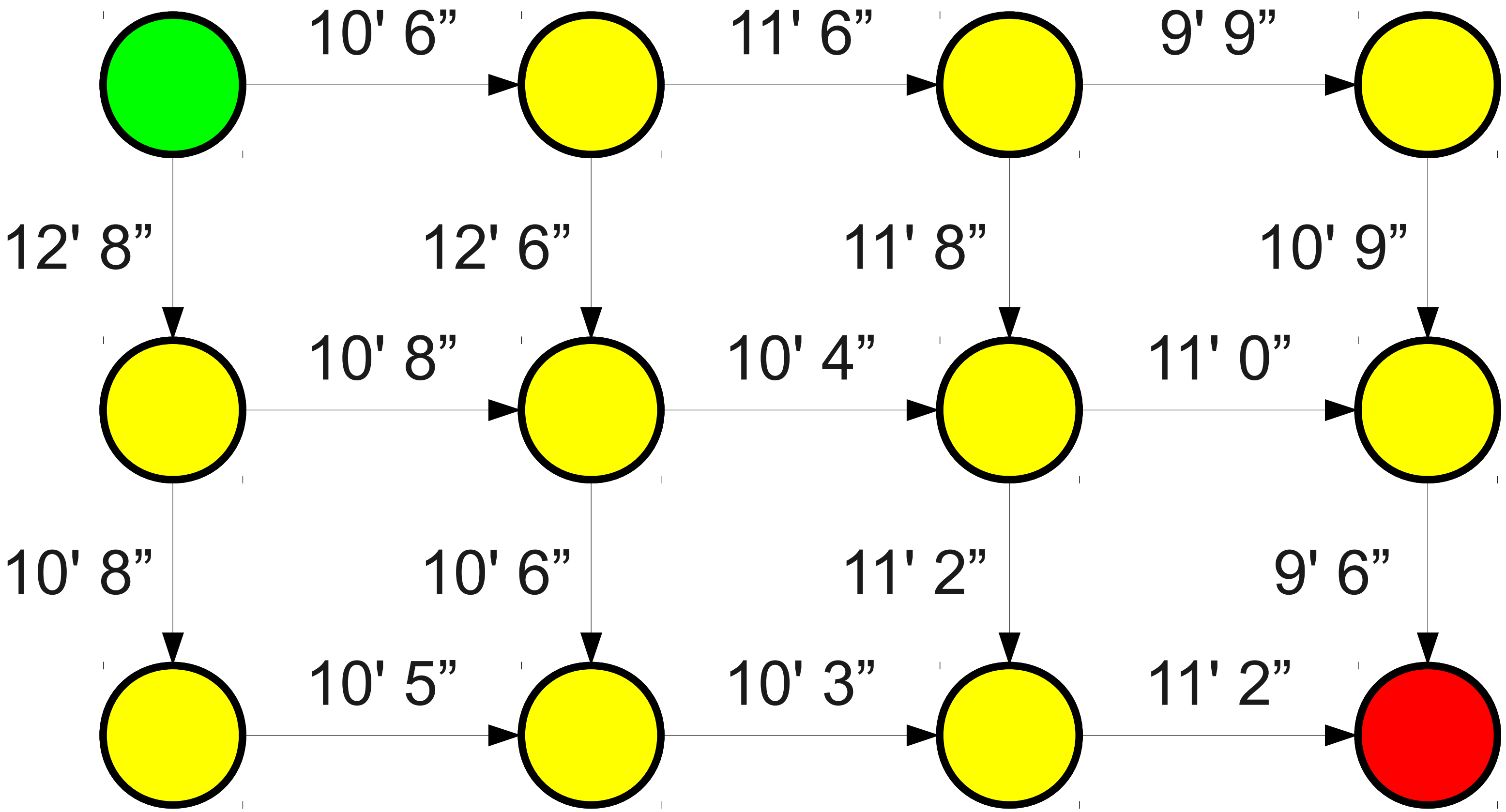
(This graph is called a Markov model)





How would we represent this graph?

Keep on Truckin'



How would we represent this graph?

Representing Graphs

- Our initial approach of encoding a graph as a `Map<`***Node****`, Vector<`***Node****`> >` will not work if the edges have extra information associated with them.
- We will need to adopt a different strategy.

Nodes and Arcs

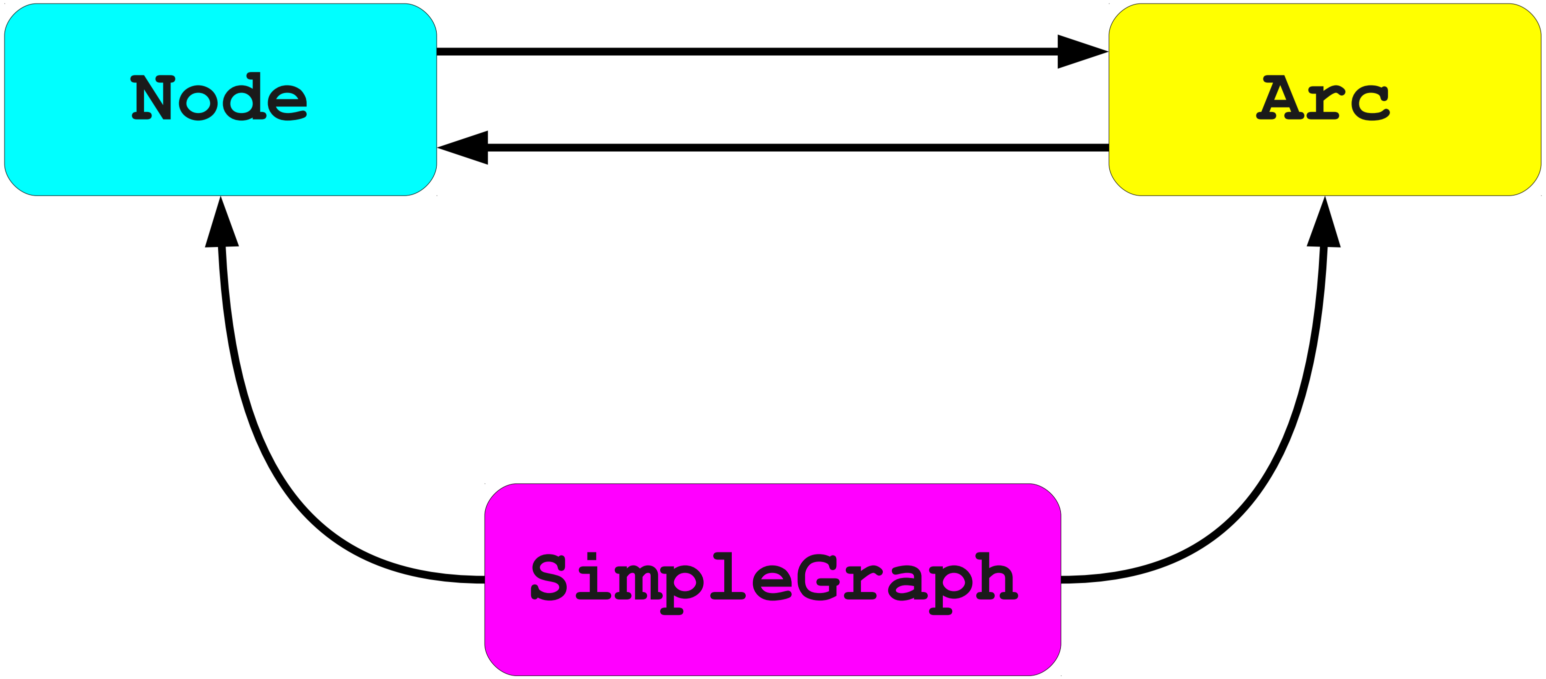
- **Idea One**: Have two separate types, one for nodes and one for arcs.
- Each node stores the set of arcs leaving that node, plus any extra information.
- Each arc stores the nodes it connects, plus any extra information.

```
struct Node {
    string name;
    Set<Arc*> arcs;
    /* … other data … */
};

struct Arc {
    Node* start;
    Node* finish;
    /* … other data … */
};

struct SimpleGraph {
    Set<Node*> nodes;
    Set<Arc*> edges;
};
```

Node depends on Arc …

… and Arc depends on Node!

A Dependency Graph



```
struct Node;
struct Arc;

struct Node {
    string name;
    Set<Arc*> arcs;
    /* … other data … */
};

struct Arc {
    Node* start;
    Node* finish;
    /* … other data … */
};

struct SimpleGraph {
    Set<Node*> nodes;
    Set<Arc*> edges;
};
```

These are called **forward declarations** and tell C++ to expect struct definitions later. They're similar to function prototypes.

Analyzing our Approach

- Advantages:
  - Allows arbitrary values to be stored in each node.
  - Allows arbitrary values to be stored in each edge.
- Disadvantages:
  - No encapsulation; can create arcs without adding them into nodes; can remove nodes without removing corresponding arcs, etc.
  - No memory management: Need to explicitly free all nodes we've created.

A `Graph` Class

- We can use this strategy as the basis for building an encapsulated `Graph` class.
- Similar to the previous approach:
  - Stores nodes and edges separately.
  - Nodes store pointers to edges and vice-versa.
- Fewer drawbacks:
  - Automatically frees all memory for you.
  - Ensures that arcs and nodes are linked properly.

Using `Graph`

- The `Graph` class we provide you is a template; You must provide the node and arc types.
- For example:

```
Graph<Node, Arc> g1;
Graph<Node, LengthyArc> g2;
Graph<FlowchartNode, FlowchartArc> g3;
```

- Requirements:
  - The node type must have a `string` called `name` and a `Set` of arc pointers called `arcs`.
  - The arc type must have two pointers to nodes named `start` and `finish`.

Graph Types for Distances

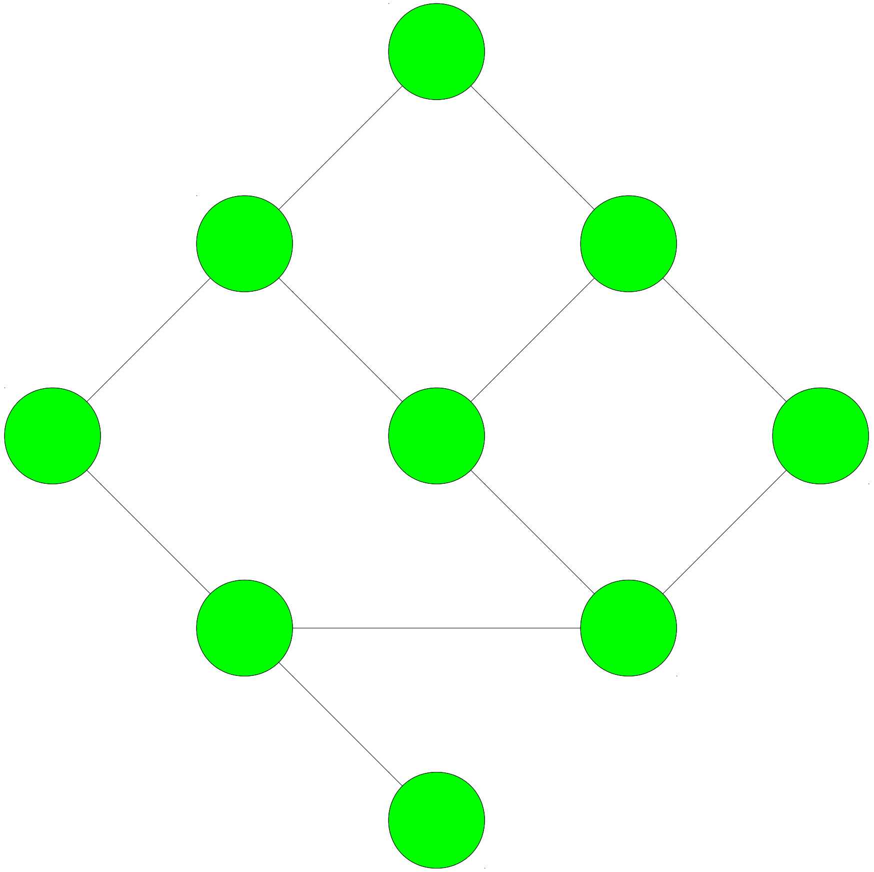
```
struct USCity;
struct USArc;

struct USCity {
    string name;
    Set<USArc*> arcs;
};

struct USArc {
    double distance;
    USCity* start;
    USCity* finish;
};
```

Graph Types for Robots

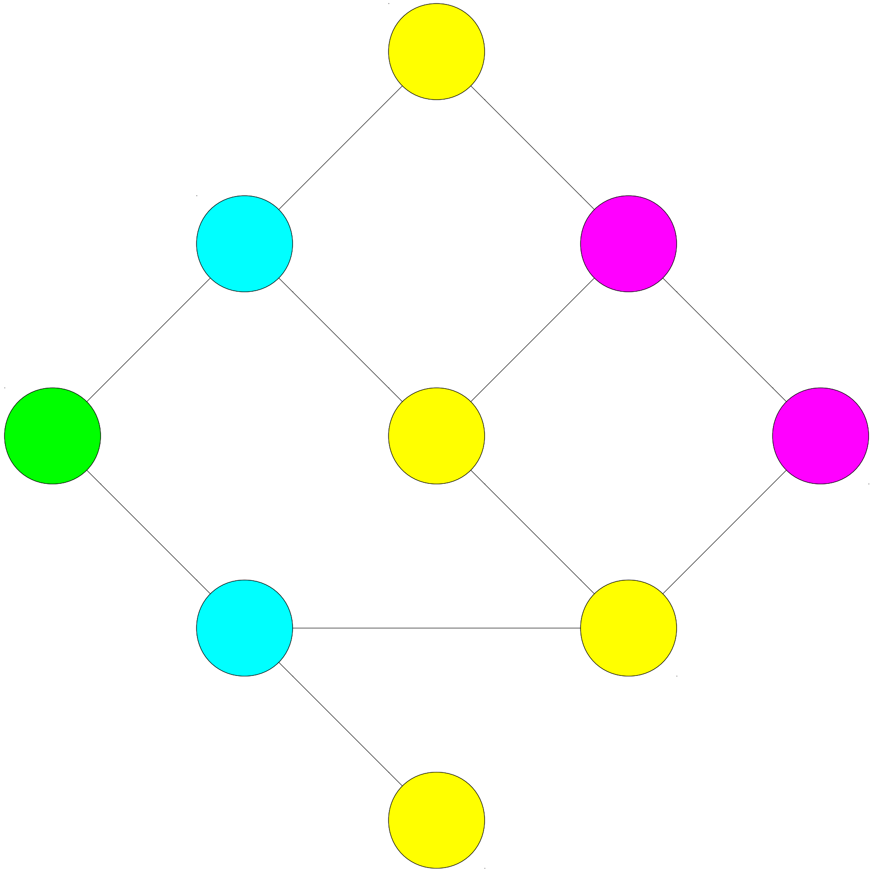
```
struct RobotLocation;
struct Transition;

struct RobotLocation {
    string name;
    Set<Transition*> arcs;
};

struct Transition {
    double probability;
    string event;
    RobotLocation* start;
    RobotLocation* finish;
};
```

Graph Algorithms

Depth-First Search



Breadth-First Search

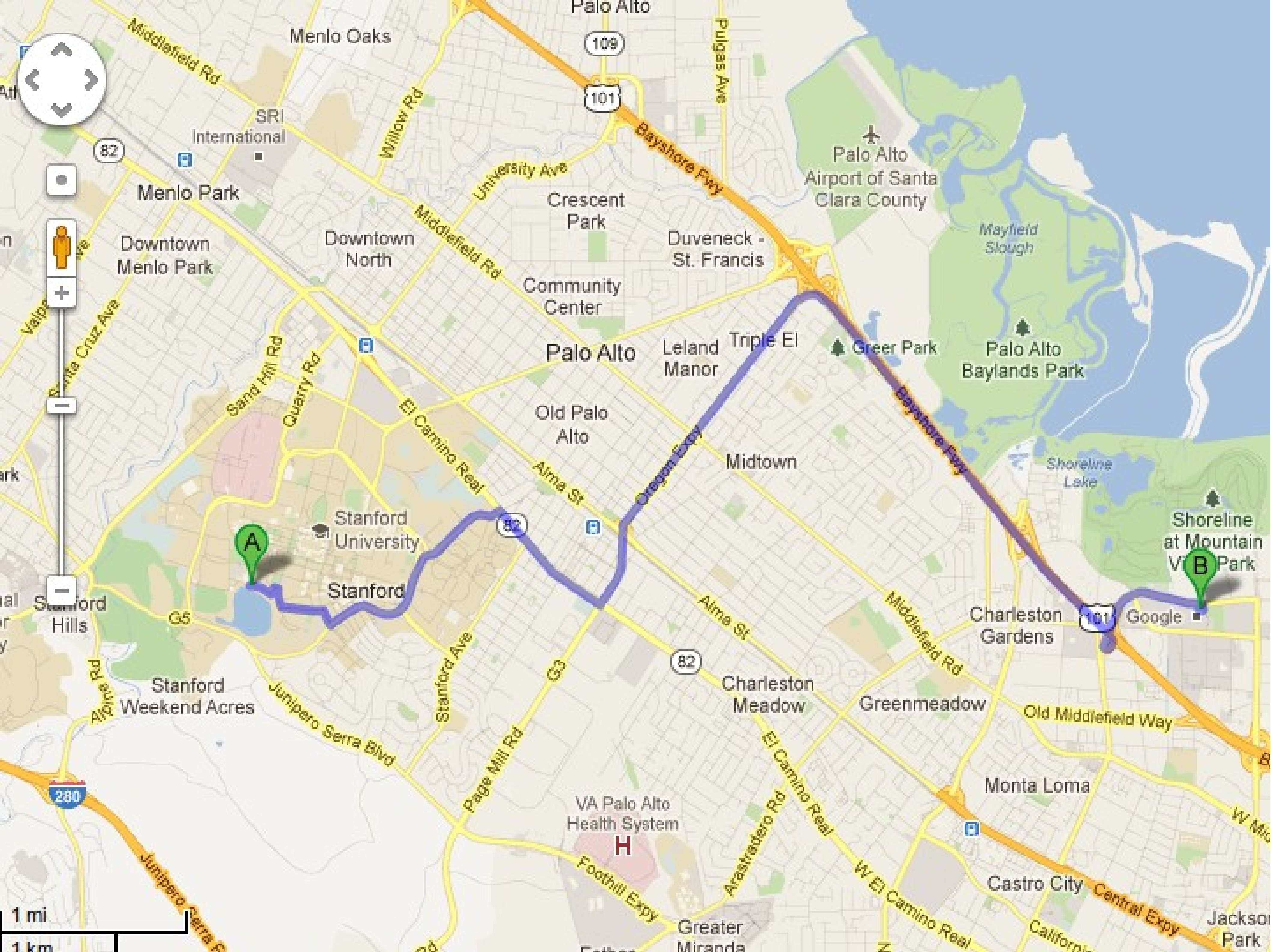


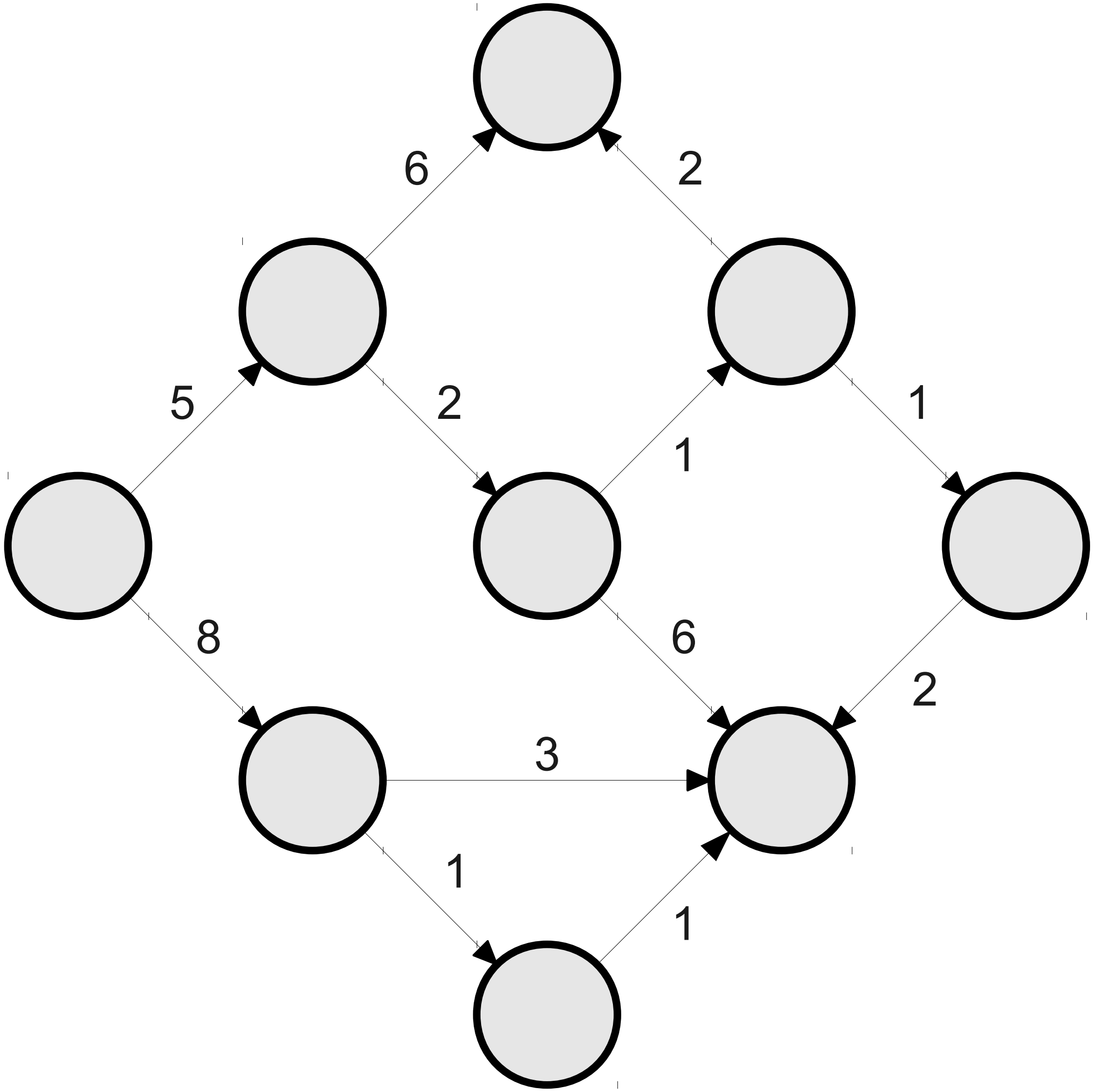
BFS and DFS

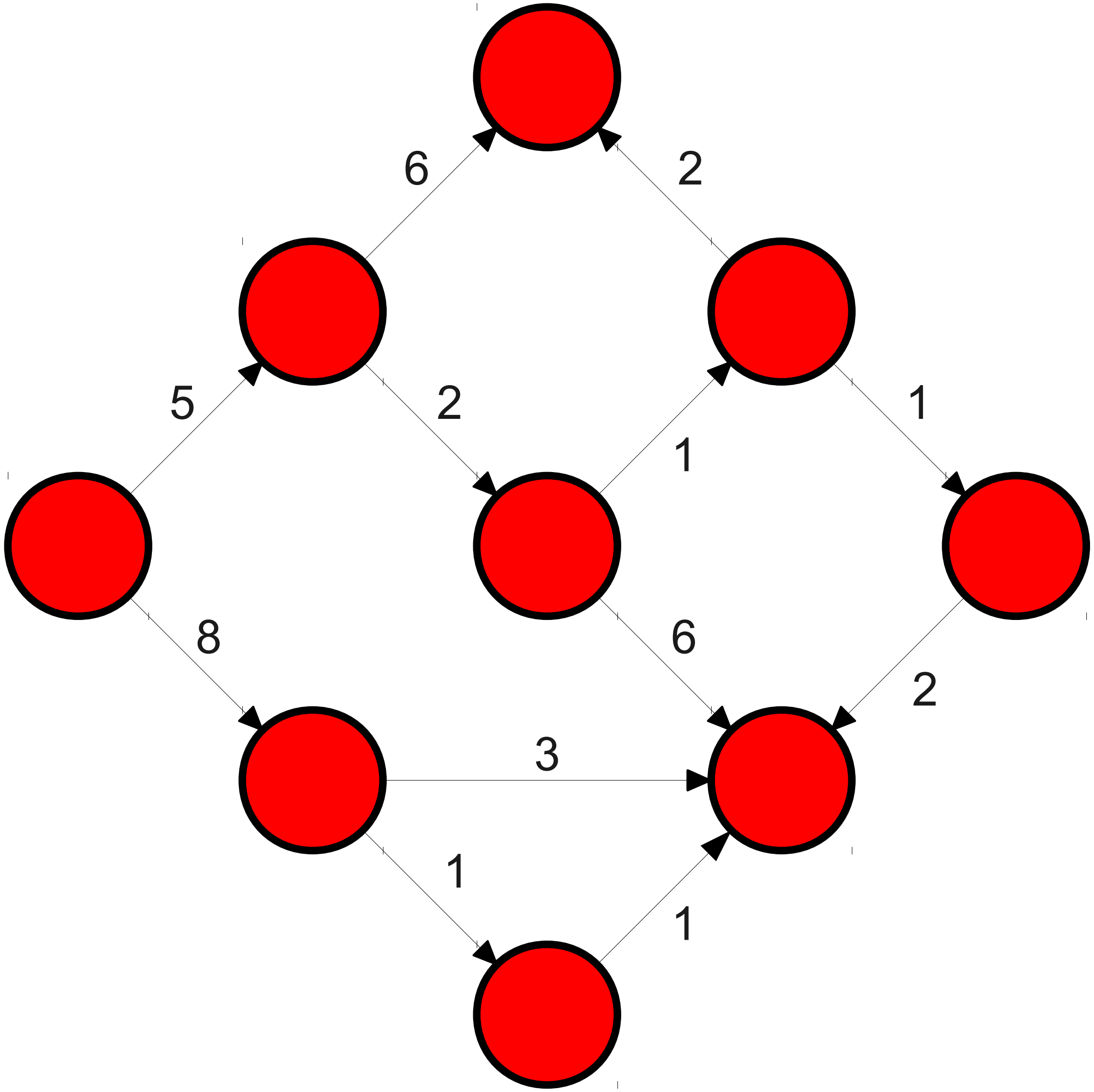
- **Depth-first search** is good for determining whether or not there exists a path from s to t.
  - Uses a stack.
- **Breadth-first search** is good for determining the shortest path from s to t.
  - Uses a queue.
- What happens if the edges now have different lengths?

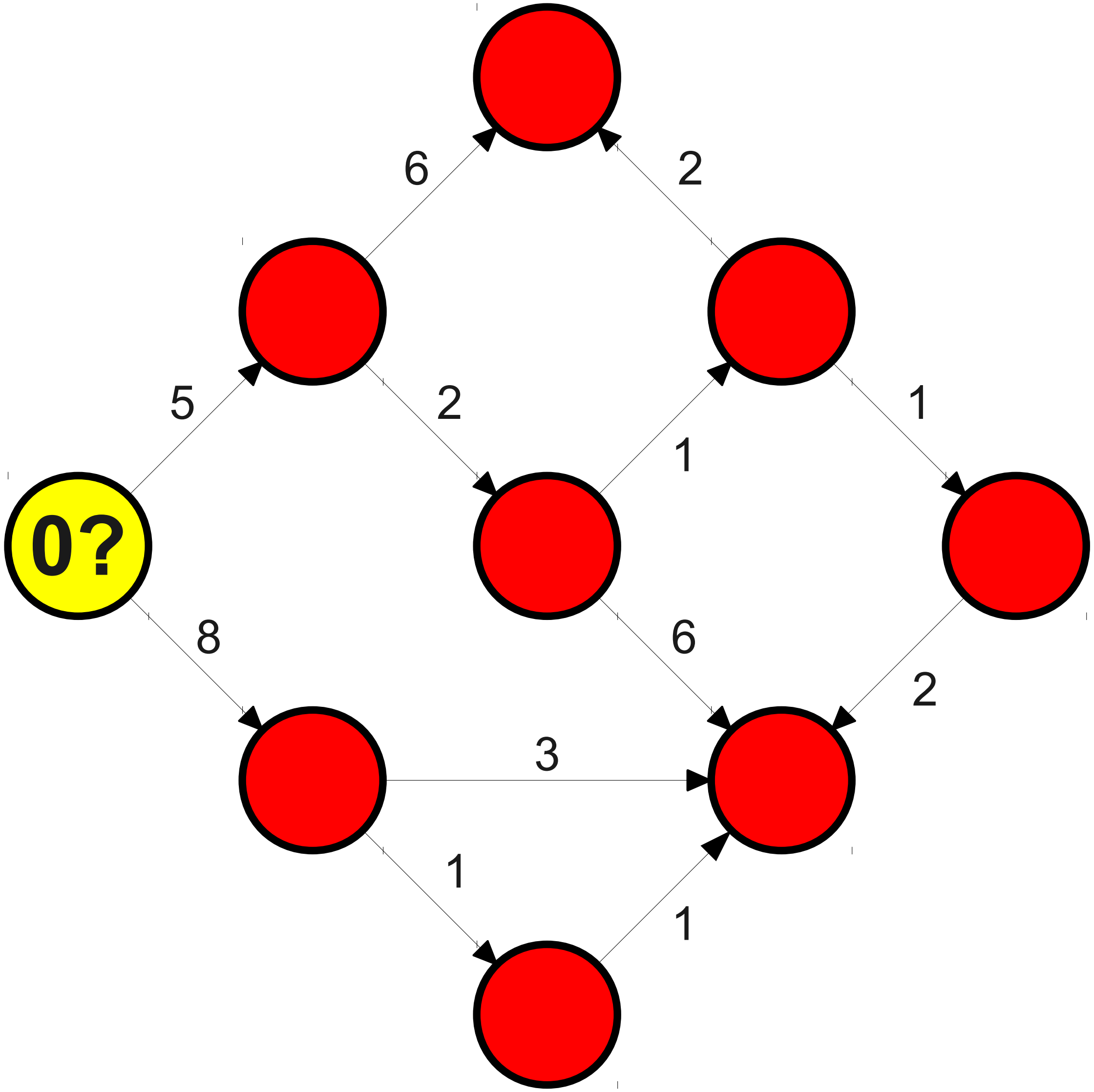
Shortest Paths

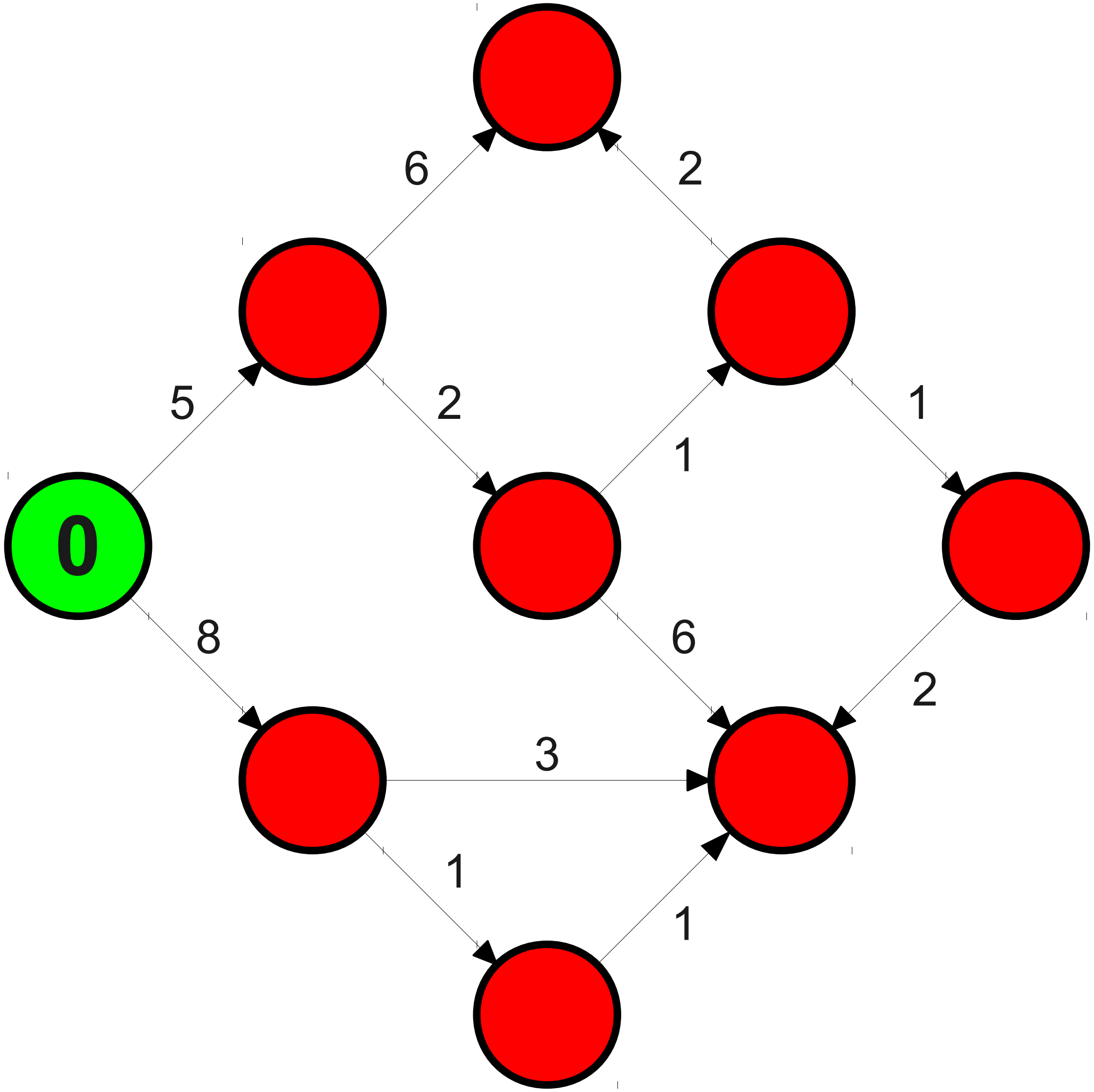
- You are given a directed graph where each edge has a nonnegative weight.
- Given a starting node s, find the shortest path (in terms of total weight) from s to each other node t.

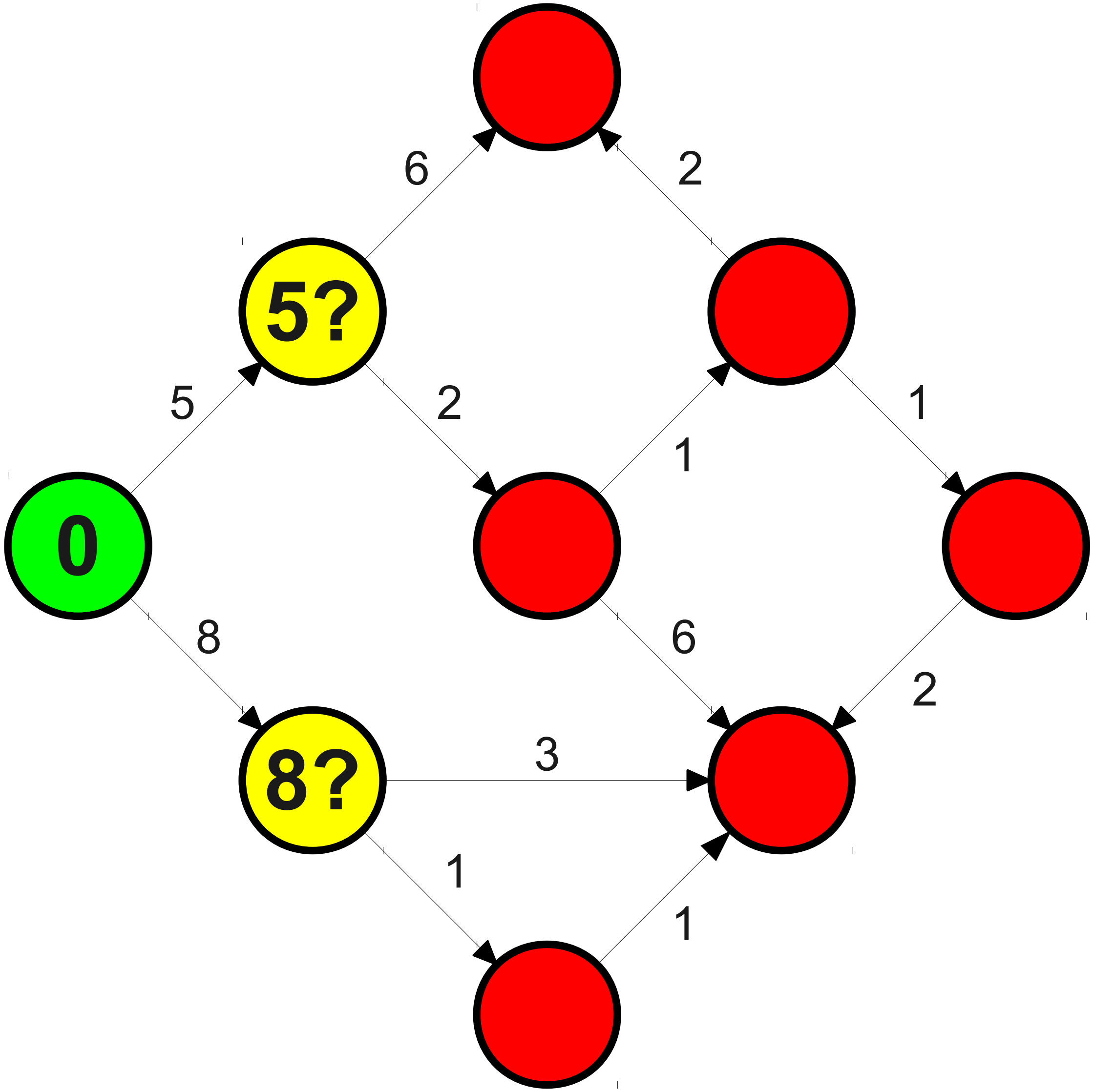

Palo Alto
Menlo Oaks
Middlefield Rd
109
101
Pulgas Ave
SRI International
82
Willow Rd
Bayshore Fwy
Palo Alto Airport of Santa Clara County
University Ave
Menlo Park
Crescent Park
Mayfield Slough
Downtown North
Middlefield Rd
Duveneck - St. Francis
Downtown Menlo Park
Community Center
Santa Cruz Ave
Sand Hill Rd
Quarry Rd
Palo Alto
Leland Manor
Triple El
Greer Park
Palo Alto Baylands Park
El Camino Real
Old Palo Alto
Oregon Expy
Bayshore Fwy
Midtown
Shoreline Lake
Alma St
Stanford University
82
A
Shoreline at Mountain View Park
B
Stanford
Stanford Hills
G5
Charleston Gardens
101
Google
Alma St
Middlefield Rd
Stanford Ave
G3
82
Alpine Rd
Stanford Weekend Acres
Charleston Meadow
Greenmeadow
Old Middlefield Way
Junipero Serra Blvd
Page Mill Rd
El Camino Real
Monta Loma
280
VA Palo Alto Health System
H
Arastradero Rd
Foothill Expy
W El Camino Real
Castro City
Central Expy
Junipero Serra
Greater Miranda
1 mi
1 km

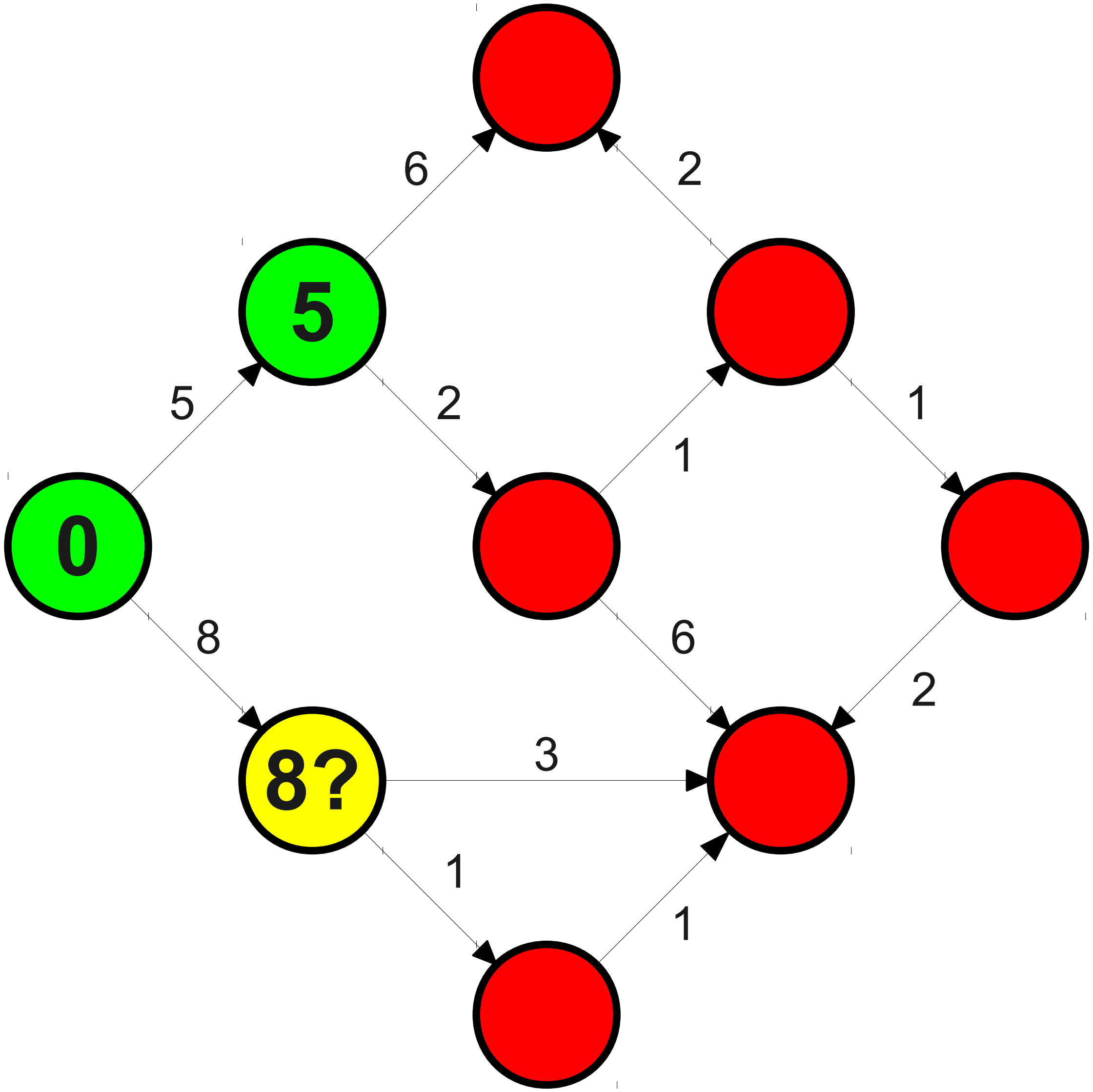

6
2
5
2
1
1
8
6
2
3
1
1

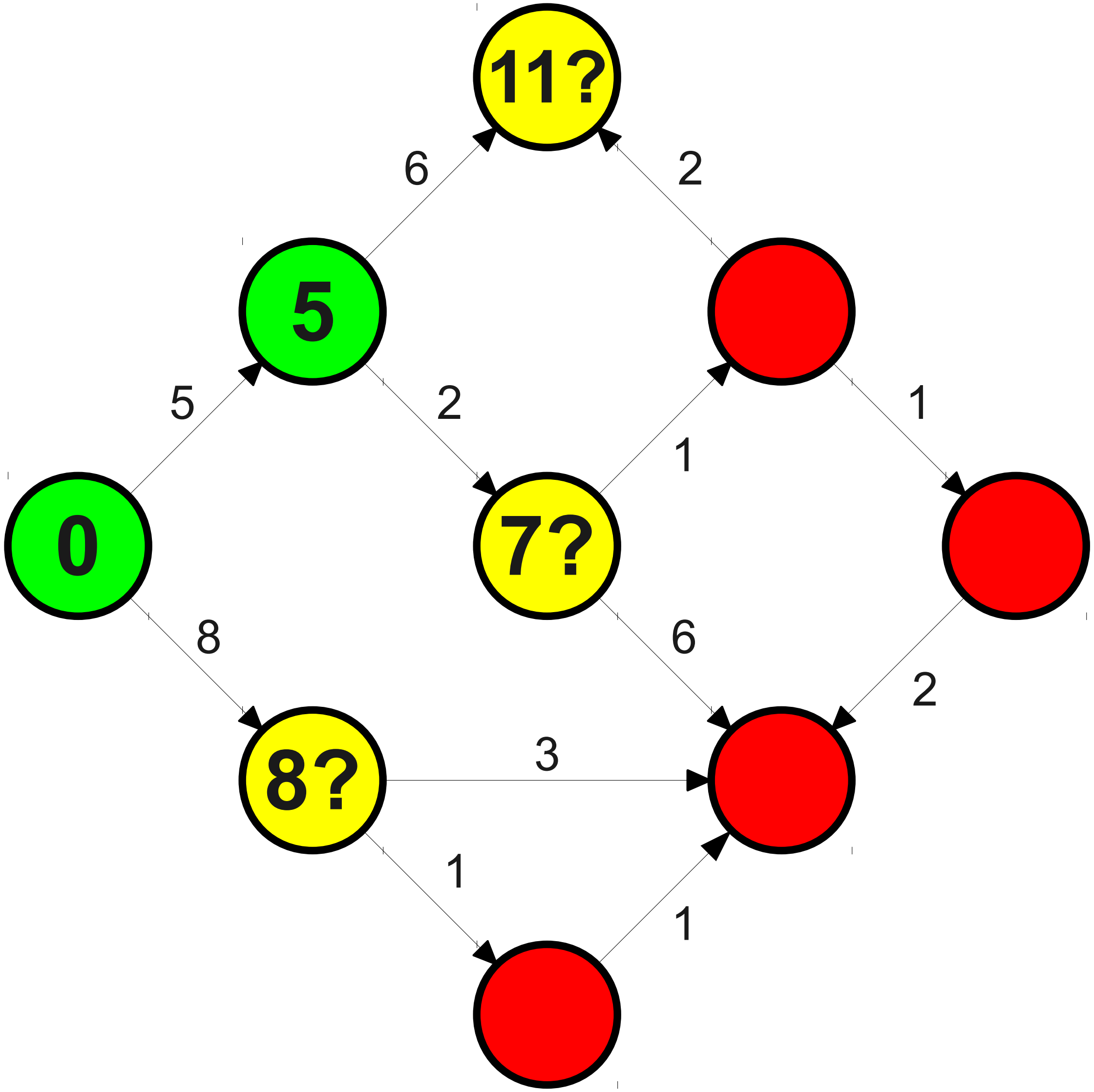

6
2
5
2
1
1
8
6
2
3
1
1

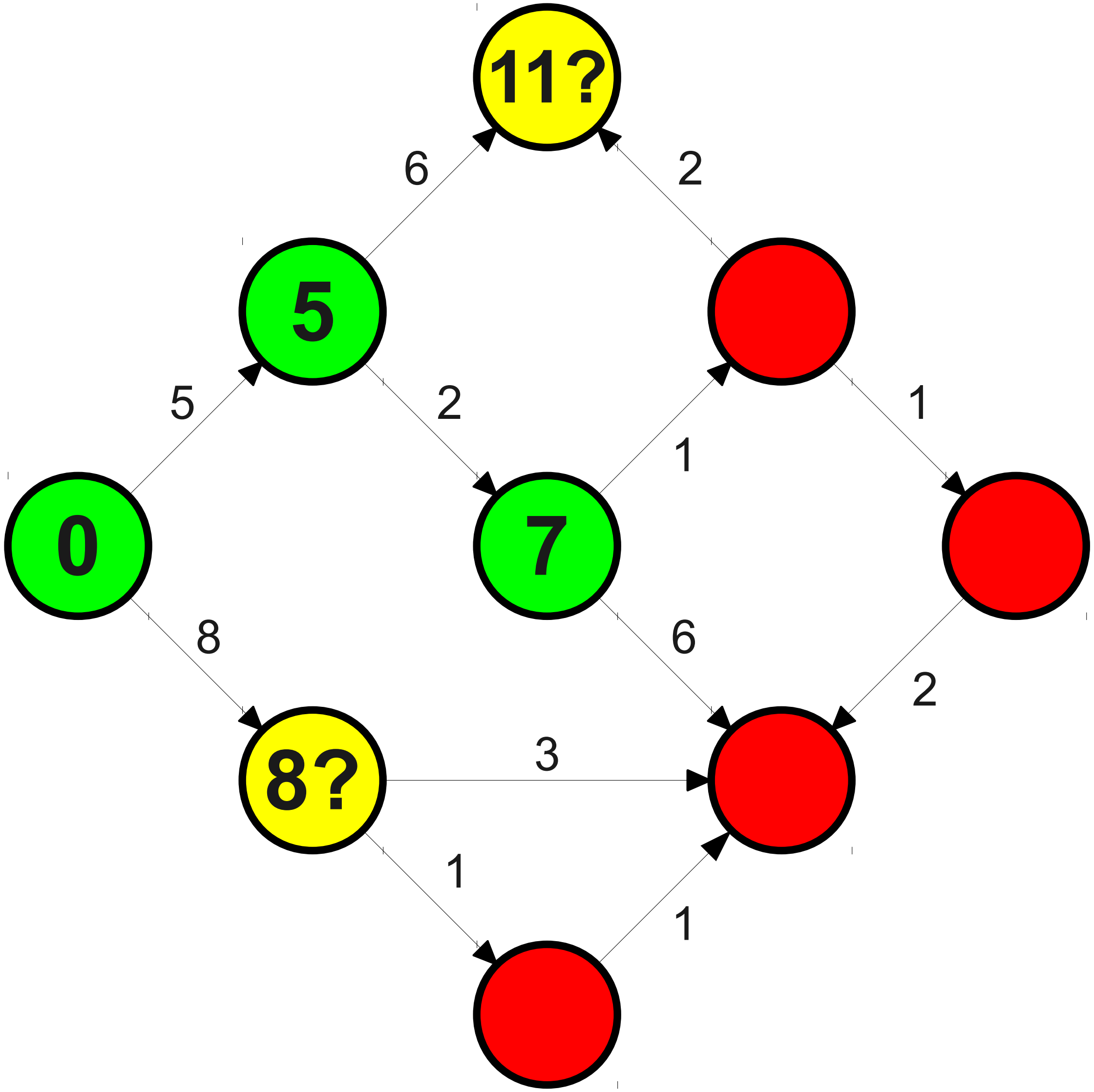

0?
6
2
5
2
1
1
8
6
2
3
1
1

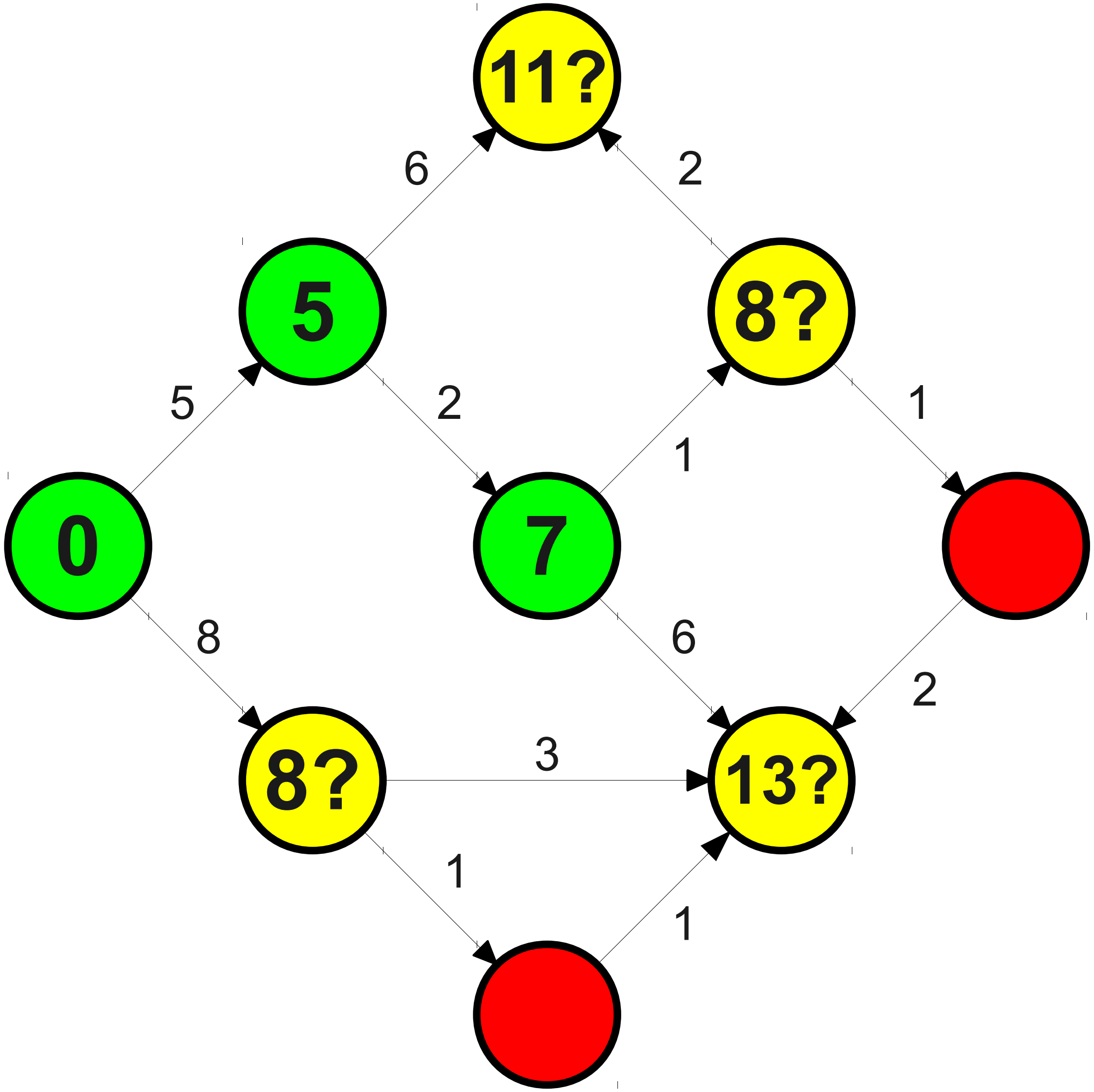

0
5
8
6
2
2
1
1
6
2
3
1
1

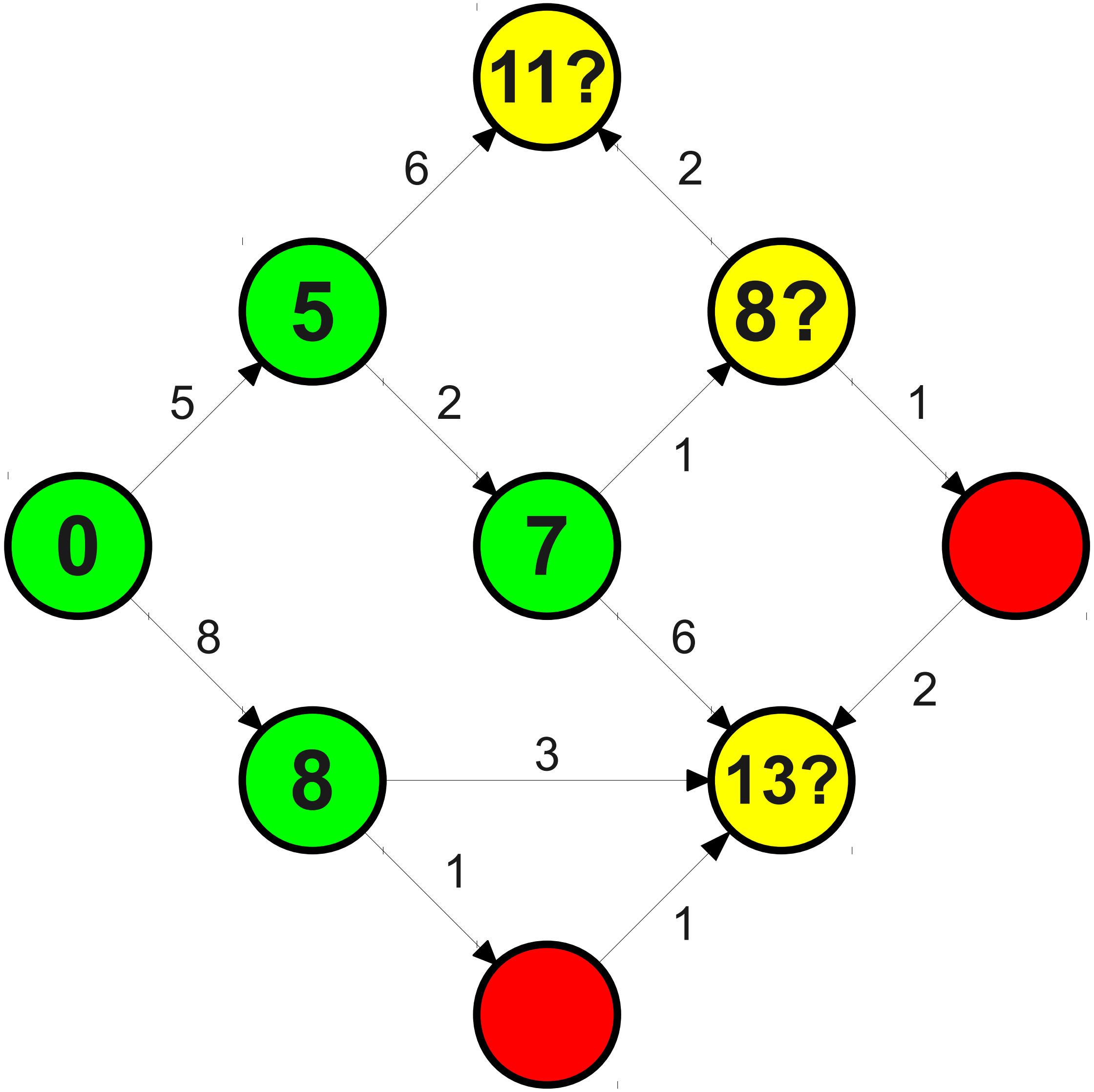

6
2
5?
5
2
1
1
0
8
6
2
8?
3
1
1


6
2
5
2
1
1
5
0
6
8
2
8?
3
1
1

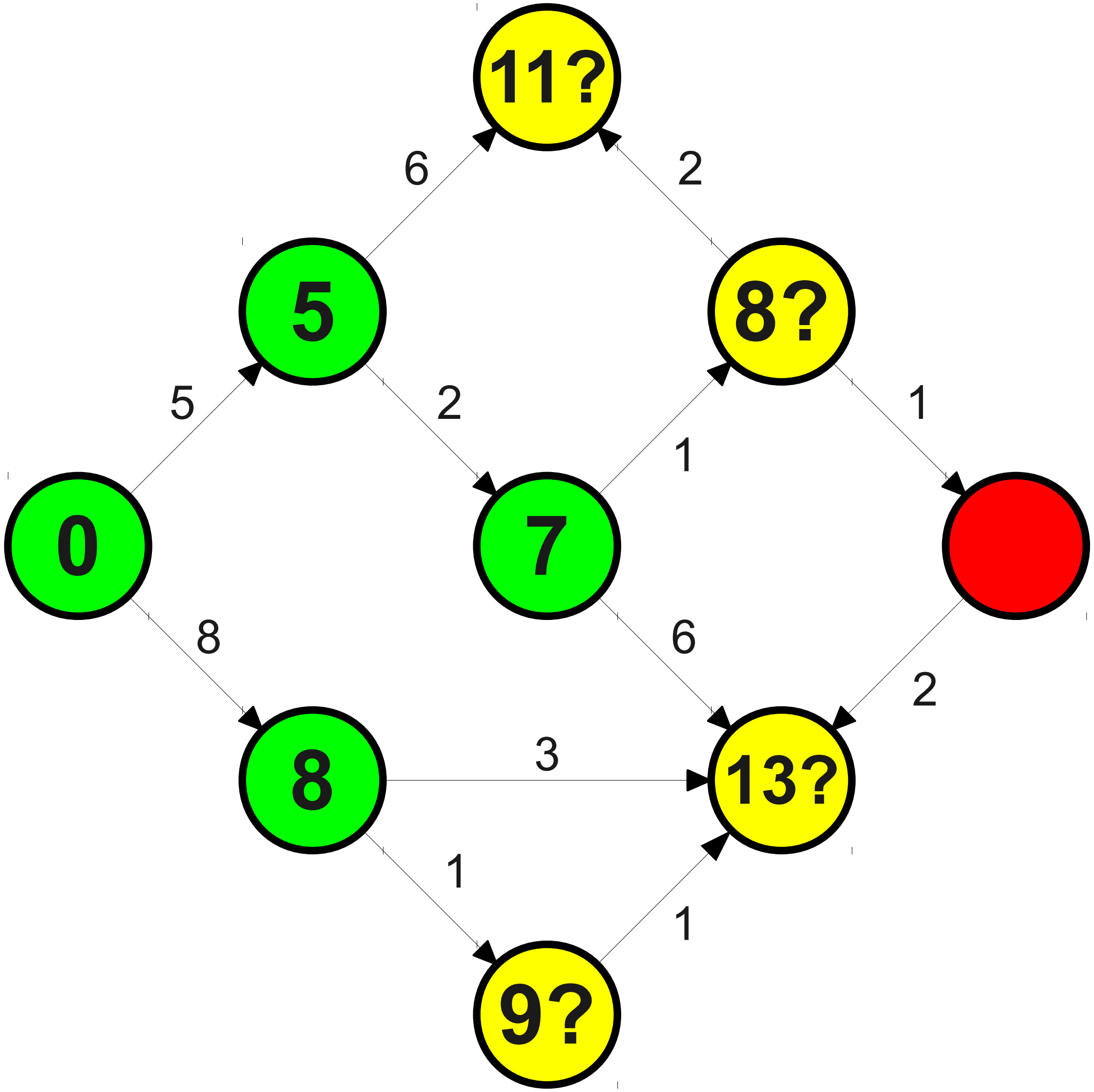

11?
6
2
5
5
2
1
1
0
7?
8
6
2
8?
3
1
1

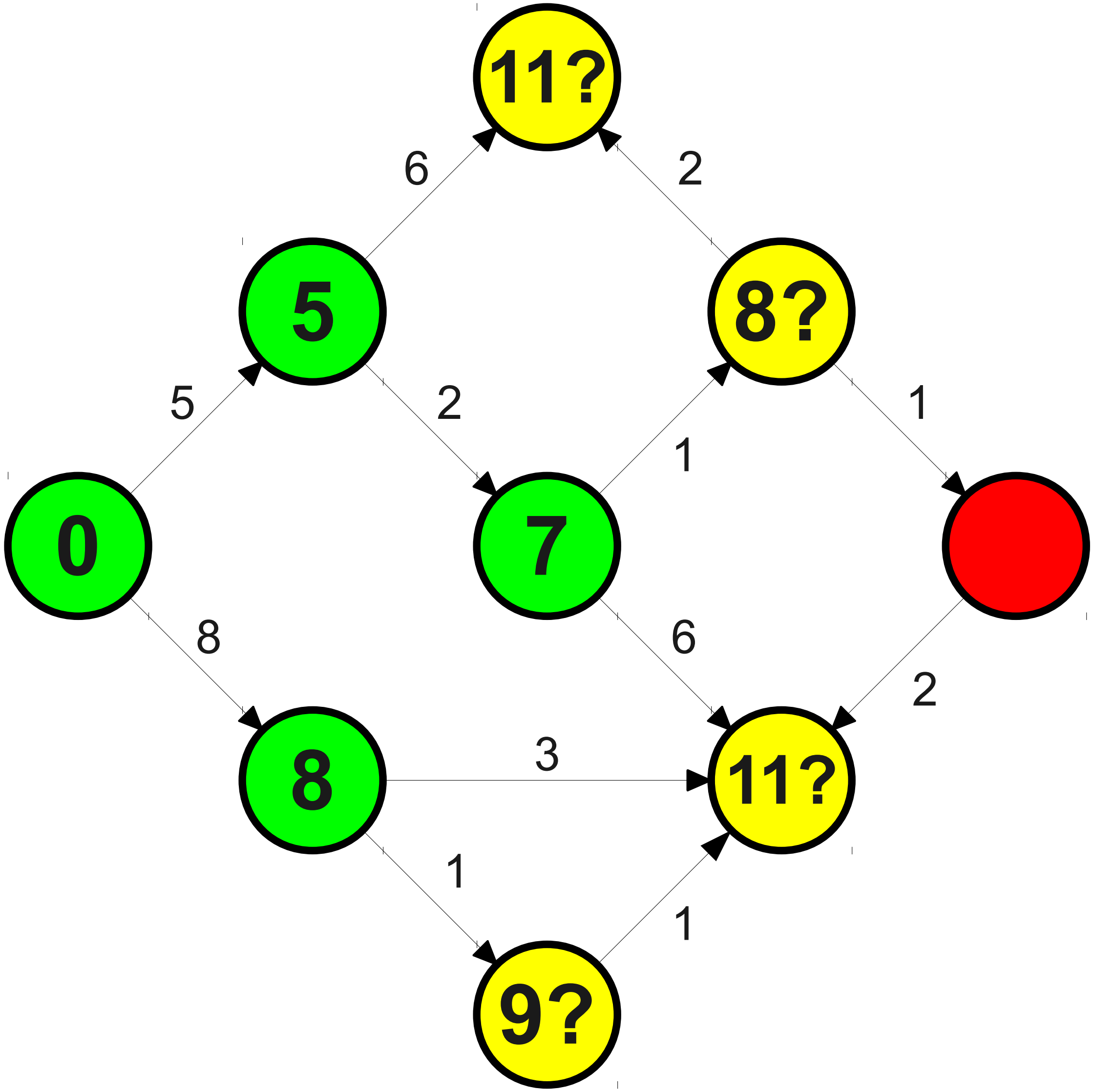

11?
6
2
5
5
2
1
1
0
7
8
6
2
8?
3
1
1

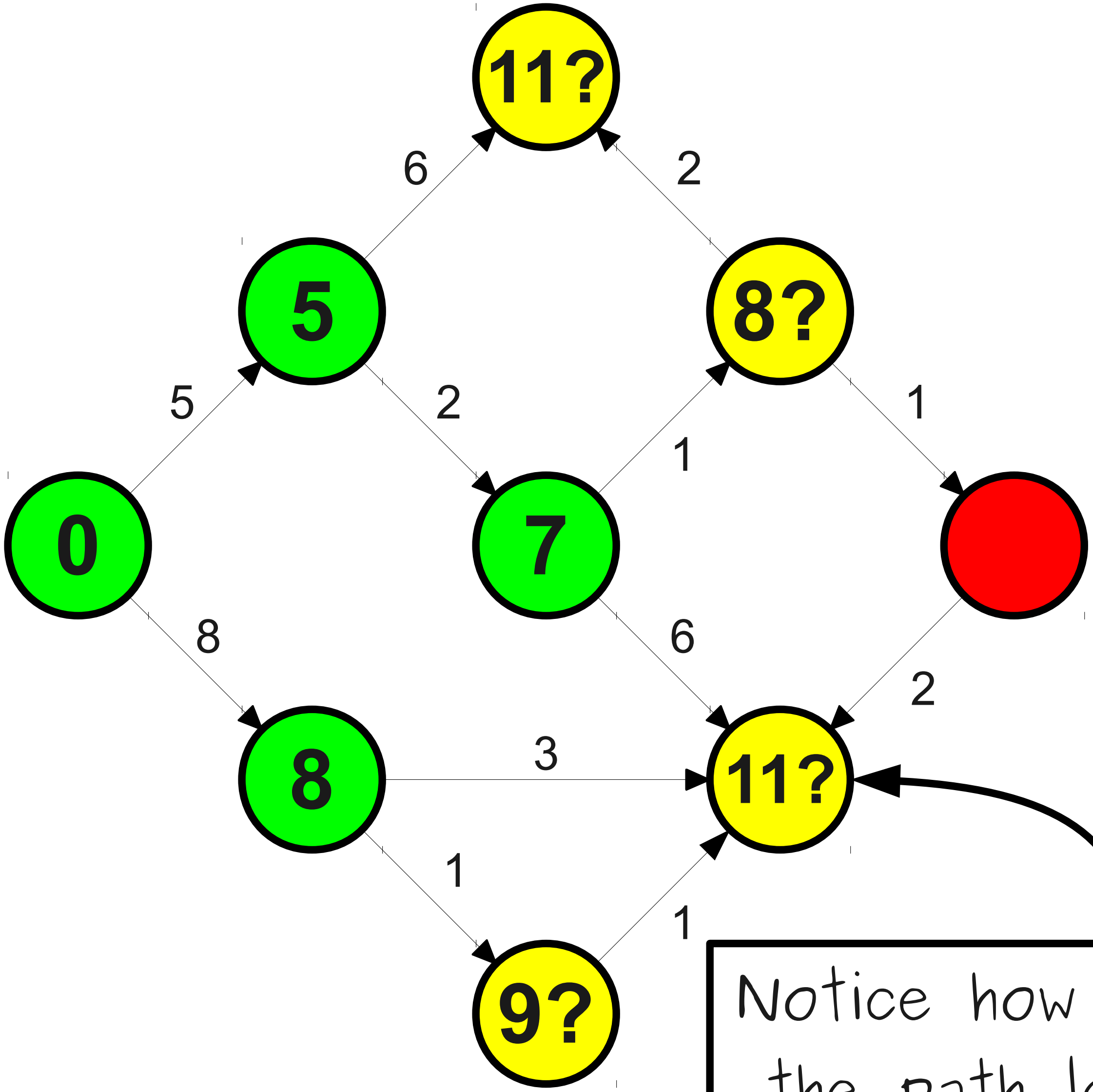

11?
6
2
5
8?
5
2
1
1
0
7
8
6
2
8?
3
13?
1
1



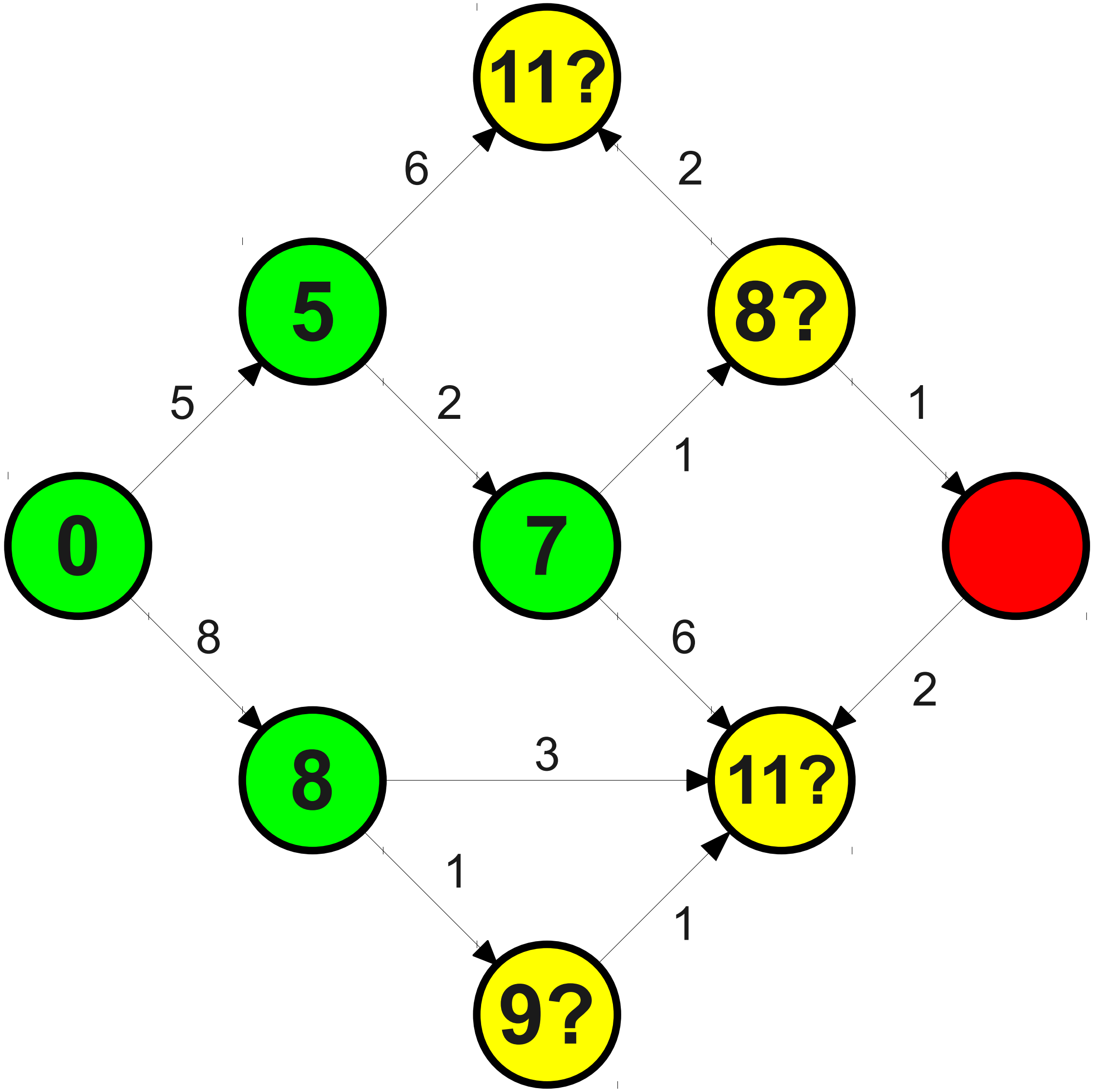
11?
6
2
5
8?
5
2
1
1
0
7
8
6
2
8
3
13?
1
1

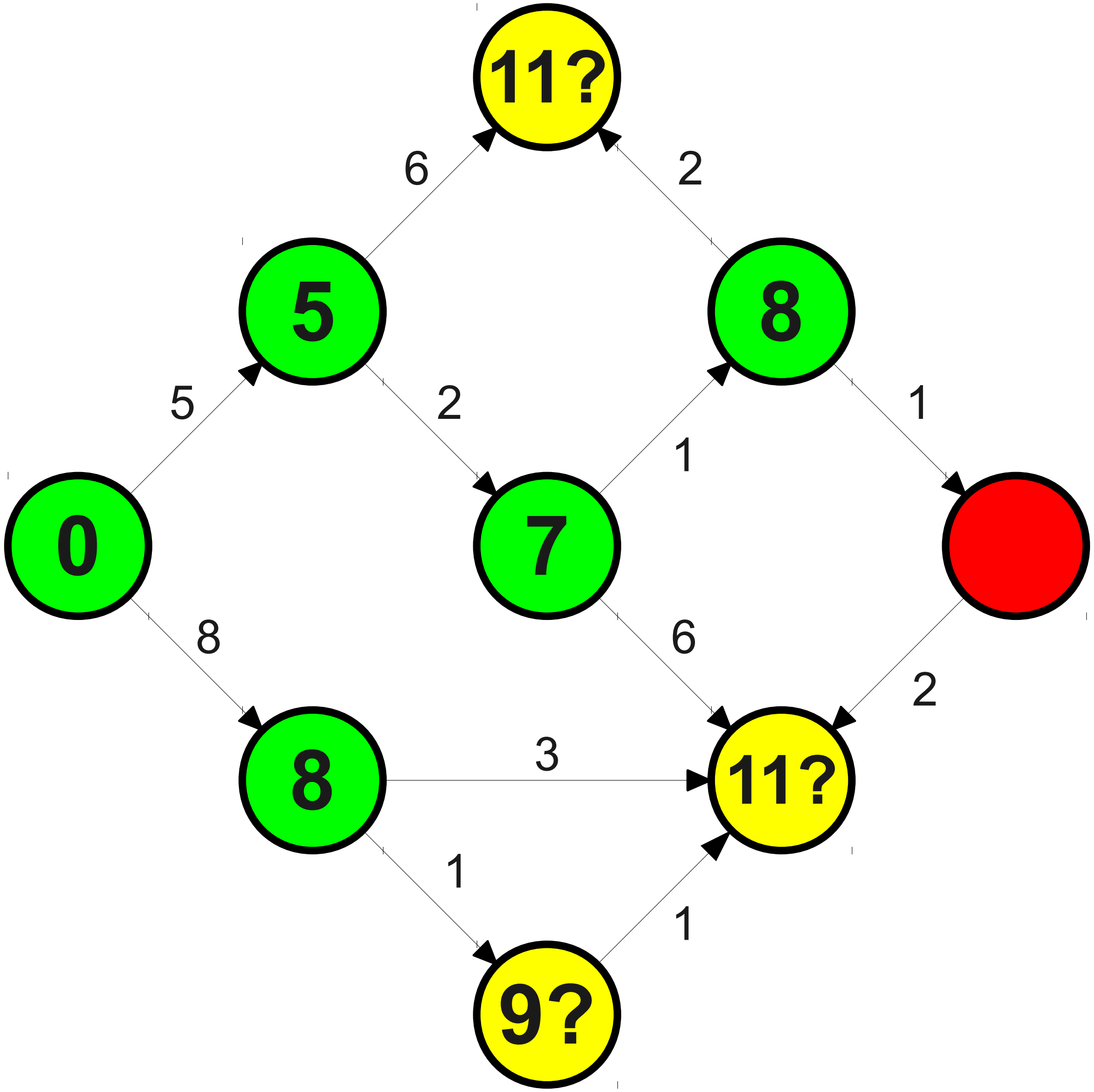

11?
5
8?
0
7
8
13?
9?
6
2
5
2
1
1
8
6
2
3
1
1

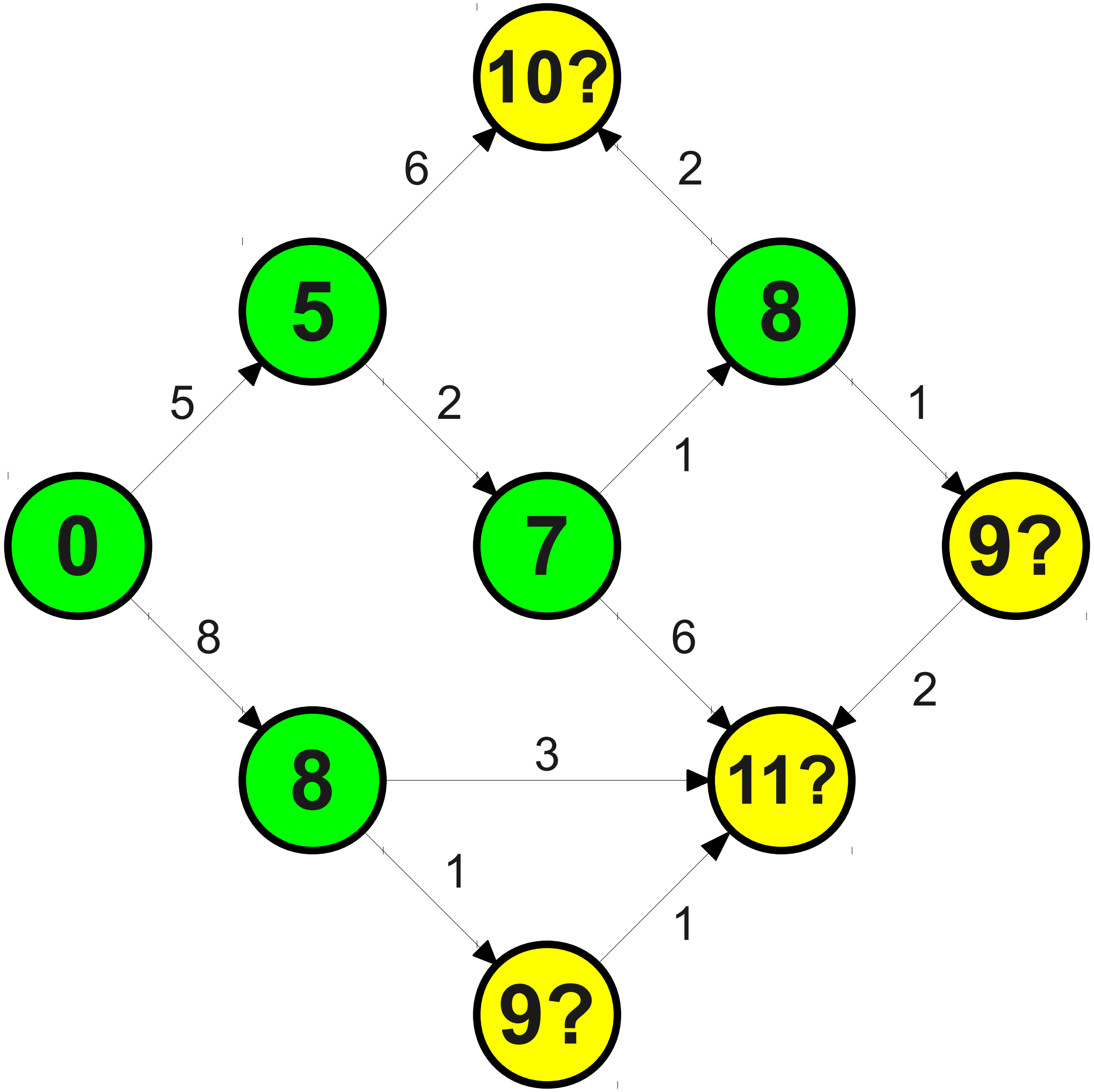

11?
6
2
5
8?
5
2
1
1
0
7
8
6
2
8
3
11?
1
1
9?

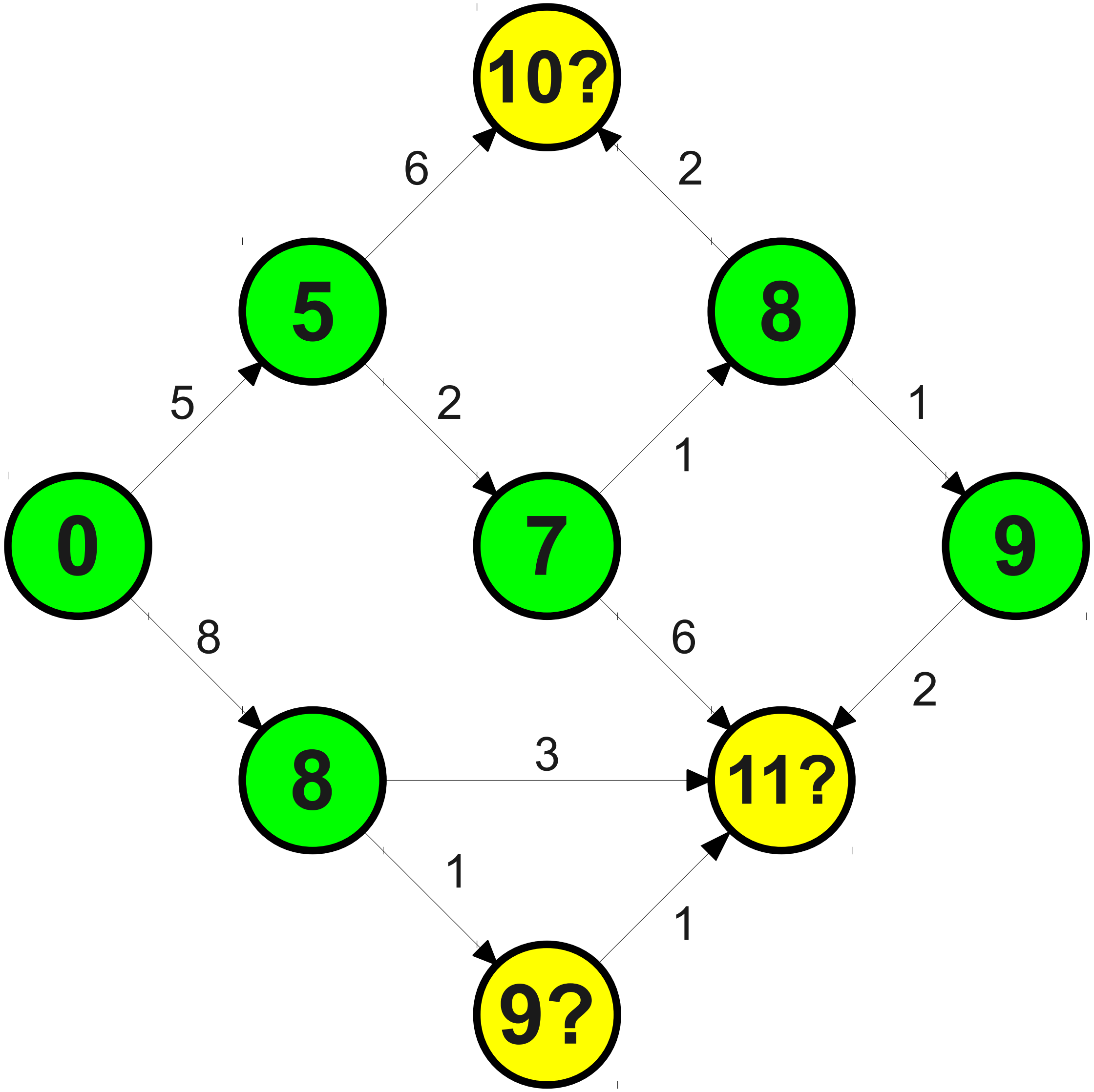


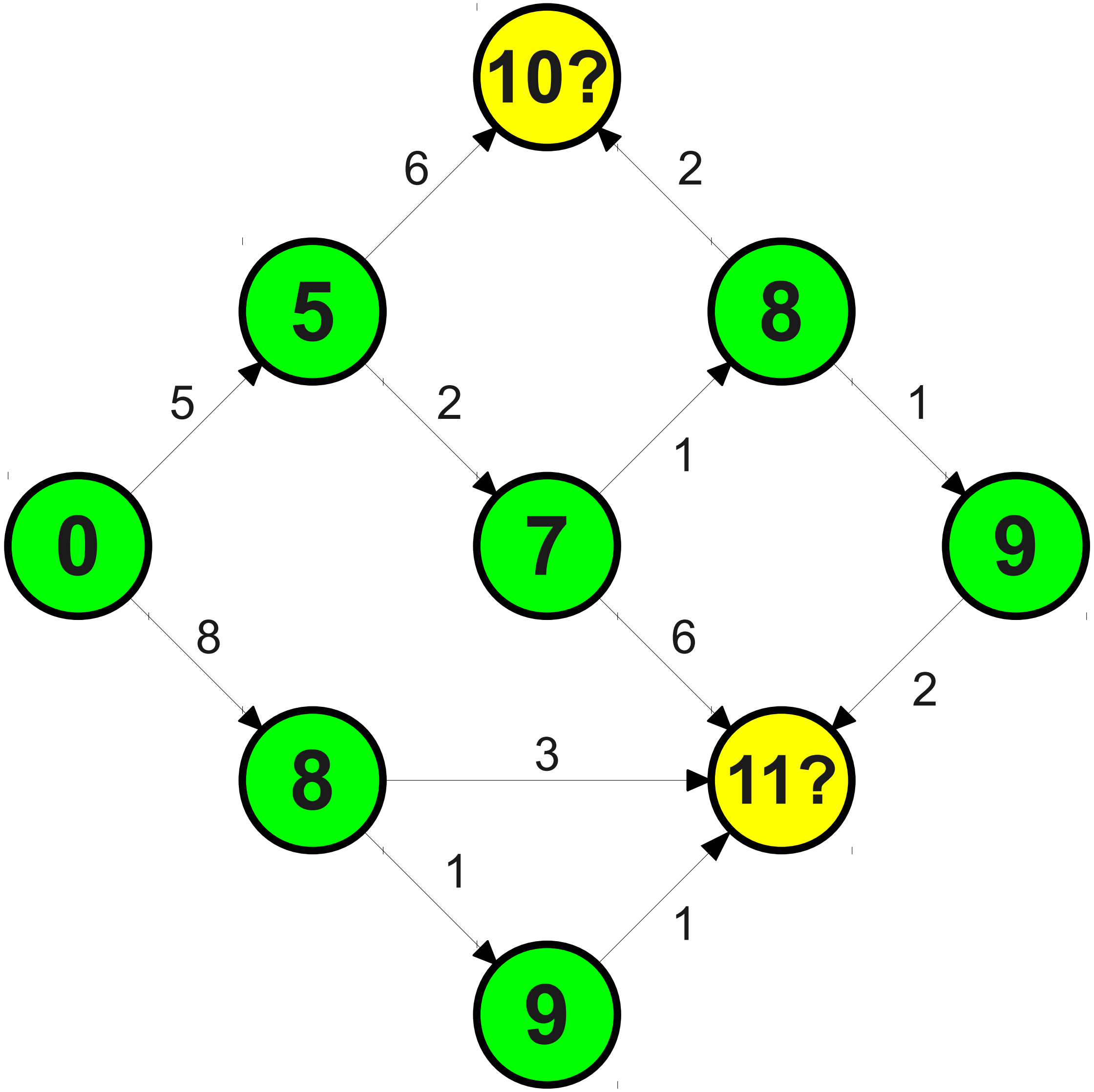
Notice how our guess of the path length to this node just changed.

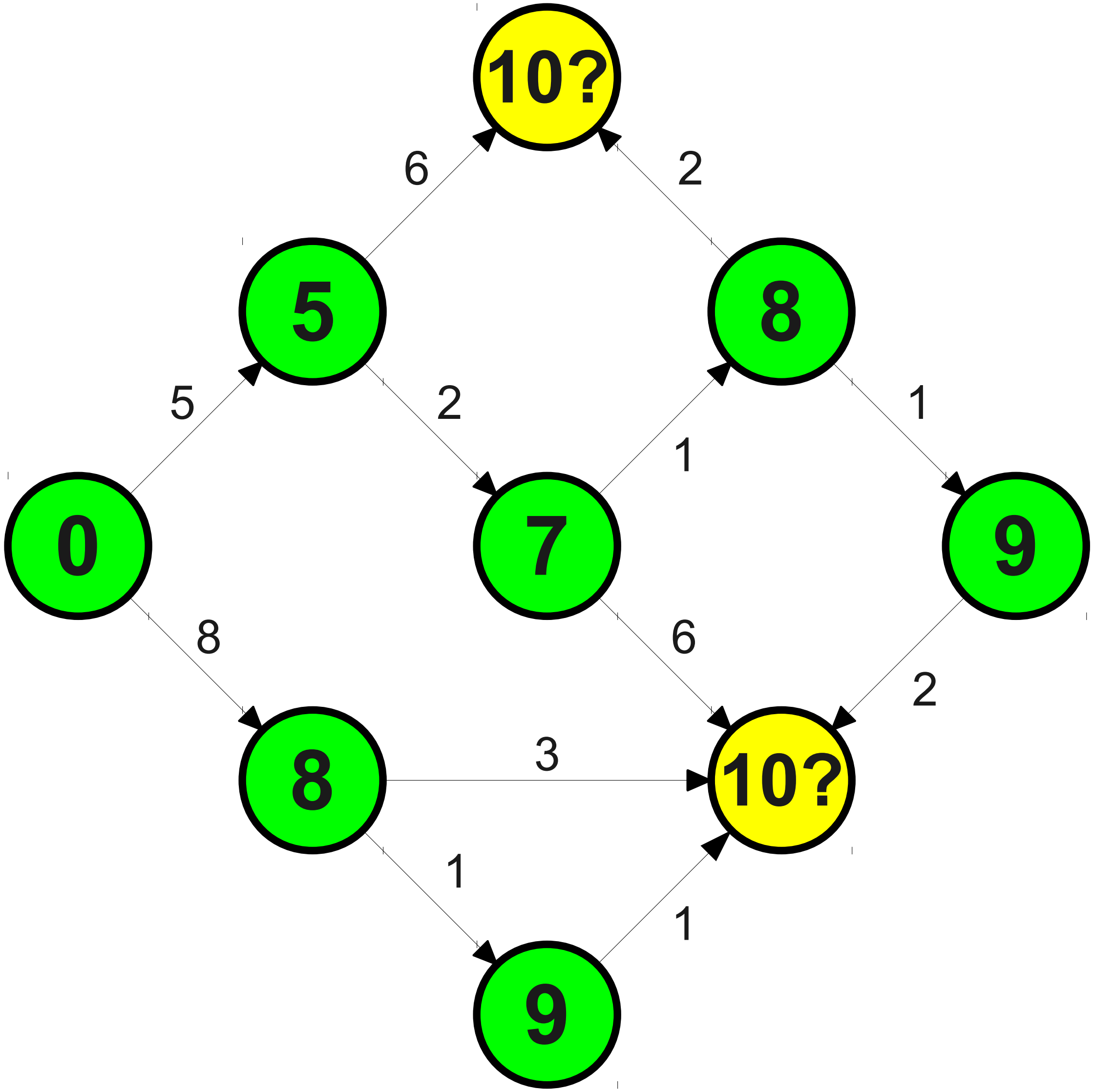

11?
6
2
5
8?
5
2
1
1
0
7
8
6
2
8
3
11?
1
1
9?

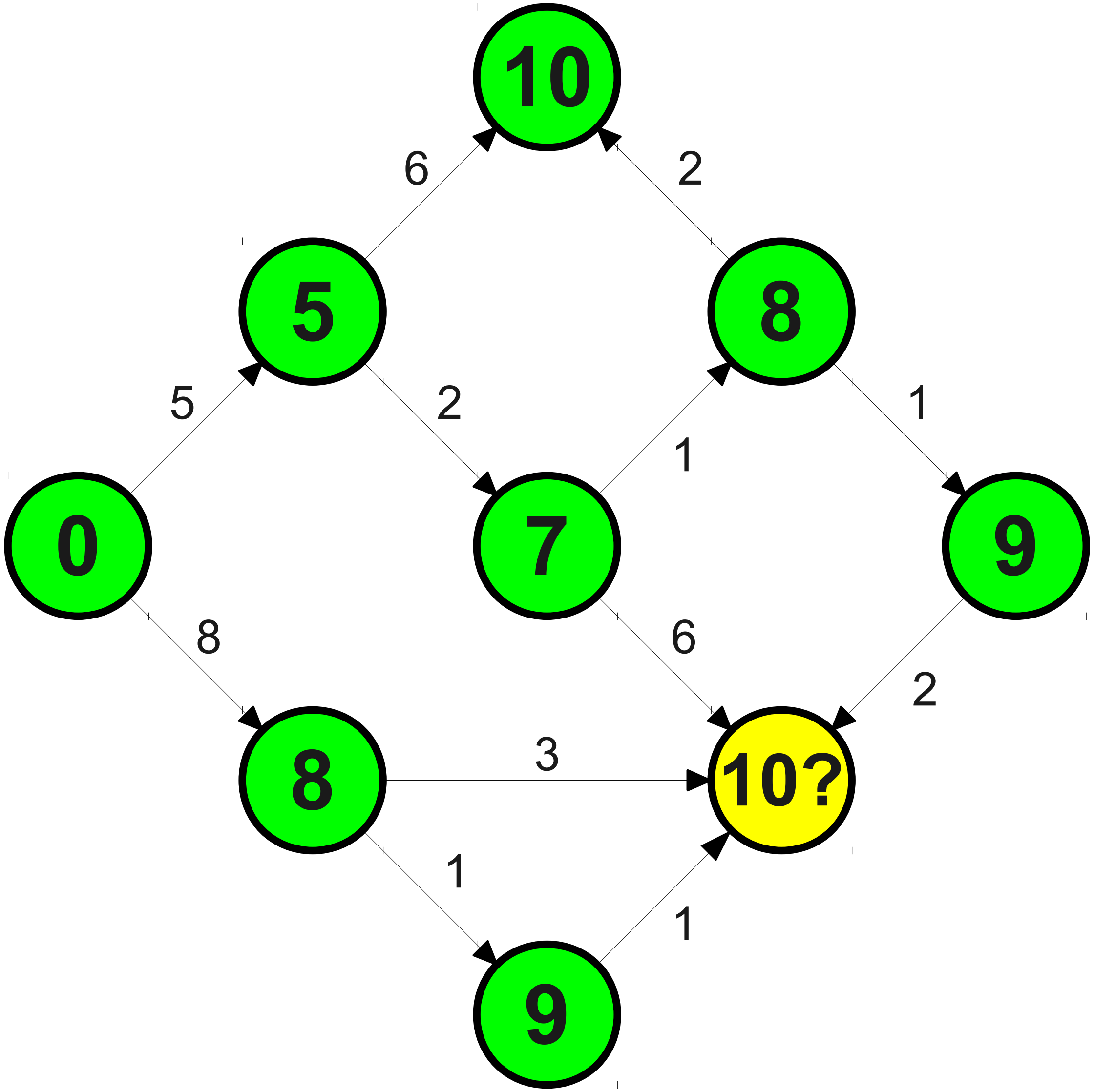

11?
6
2
5
8
5
2
1
1
0
7
8
6
2
8
3
11?
1
1
9?

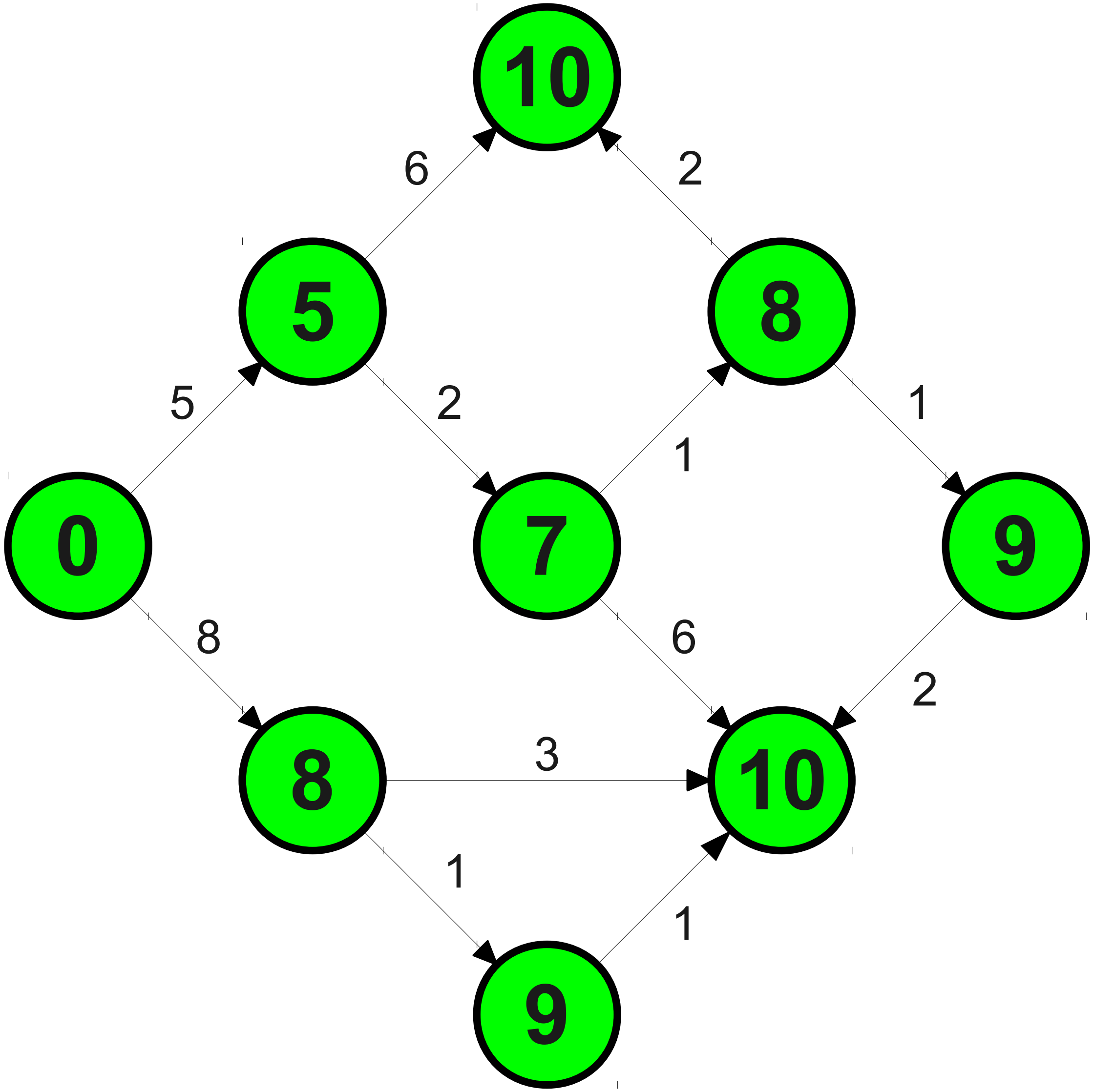

10?
6
2
5
8
5
2
1
1
0
7
9?
8
6
2
8
3
11?
1
1
9?


10?
6
2
5
8
5
2
1
1
0
7
9
8
6
2
8
3
11?
1
1
9?


10?
6
2
5
8
5
2
1
1
0
7
9
8
6
2
8
3
11?
1
1
9


10?
6
2
5
8
5
2
1
1
0
7
9
8
6
2
8
3
10?
1
1
9


10
6
2
5
8
5
2
1
1
0
7
9
8
6
2
8
3
10?
1
9
1

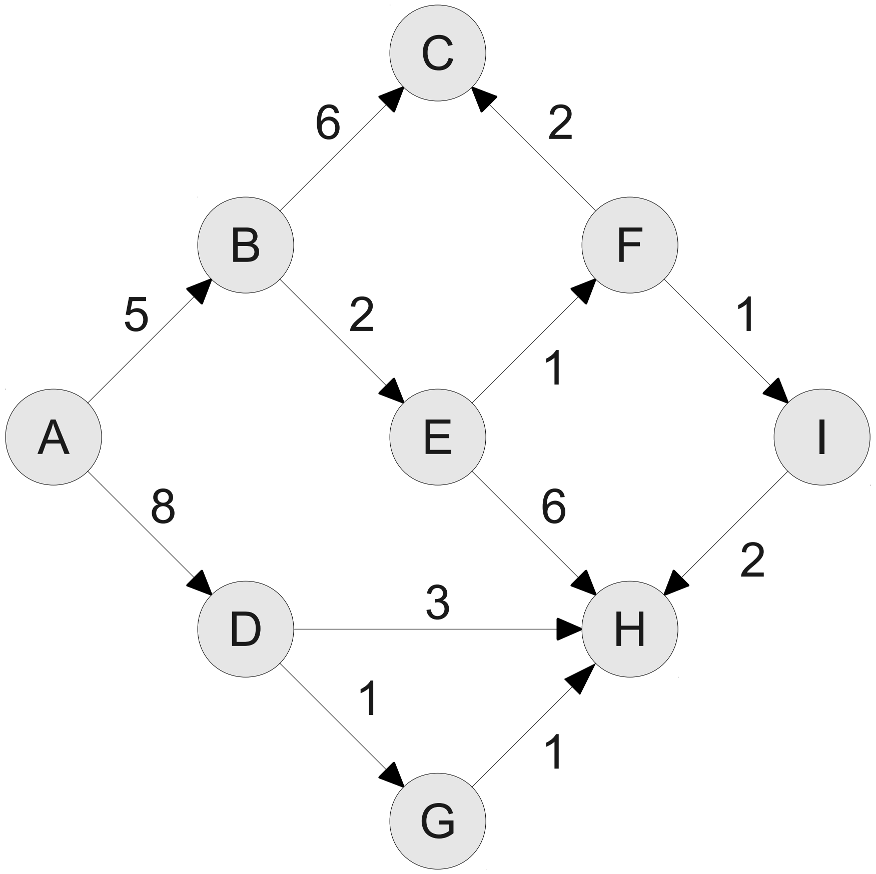

10
6
2
5
8
5
2
1
1
0
7
9
8
6
2
8
3
10
1
1
9

One Possible Approach

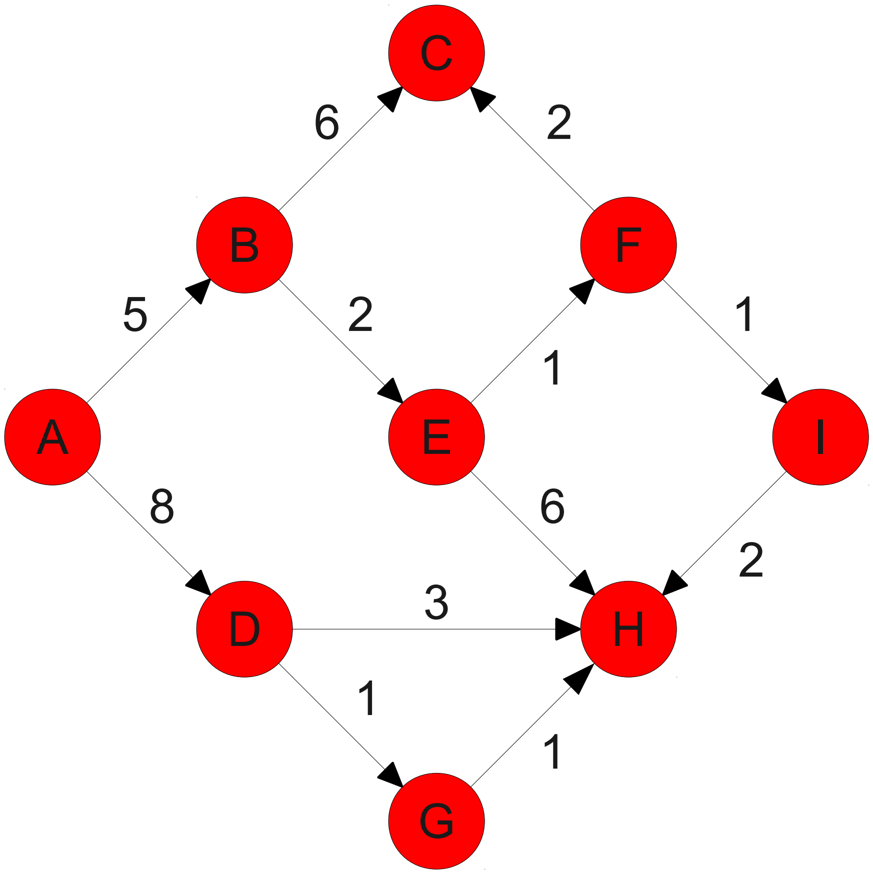
- Split nodes into three groups:
  - 🟢 13 Green nodes, where we know the length of the shortest path,
  - 🟡 42? Yellow nodes, where we have a guess of the length of the shortest path, and
  - 🔴 Red nodes, where we have no idea what the path length is.
- Repeatedly remove the lowest-cost yellow node, make it green, and update all connected nodes.

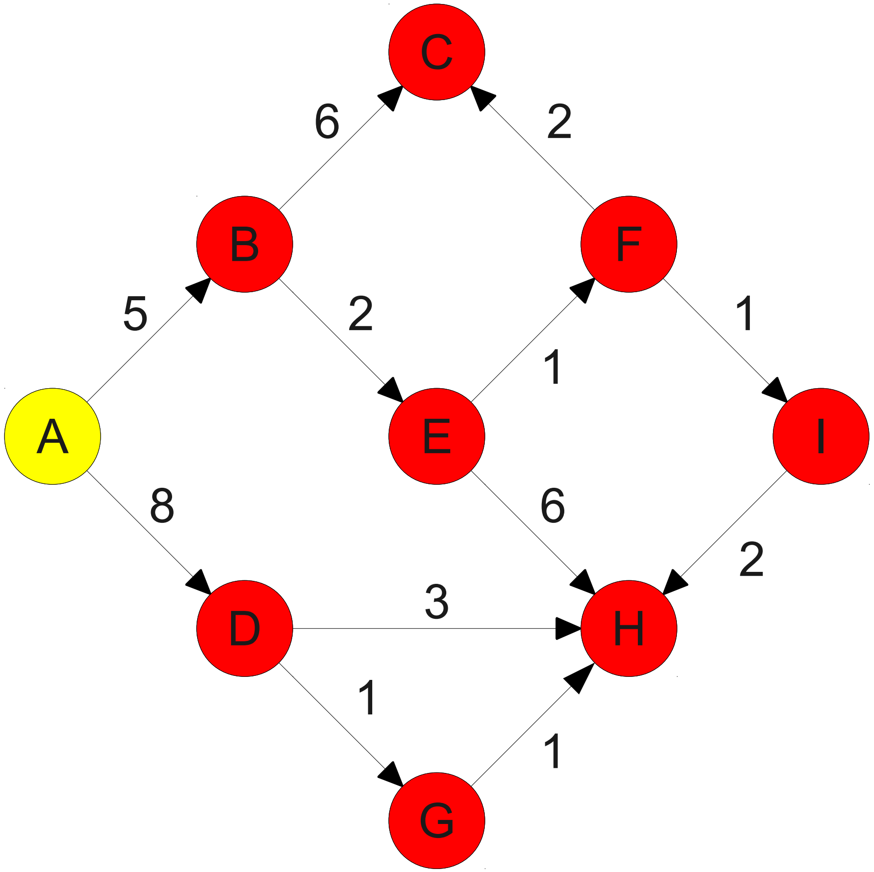
Dijkstra's Algorithm

- This algorithm for finding shortest paths is called **Dijkstra's algorithm**.
- One of the fastest algorithms for finding the shortest path from *s* to all other nodes in the graph.
- There are many ways to implement this algorithm.

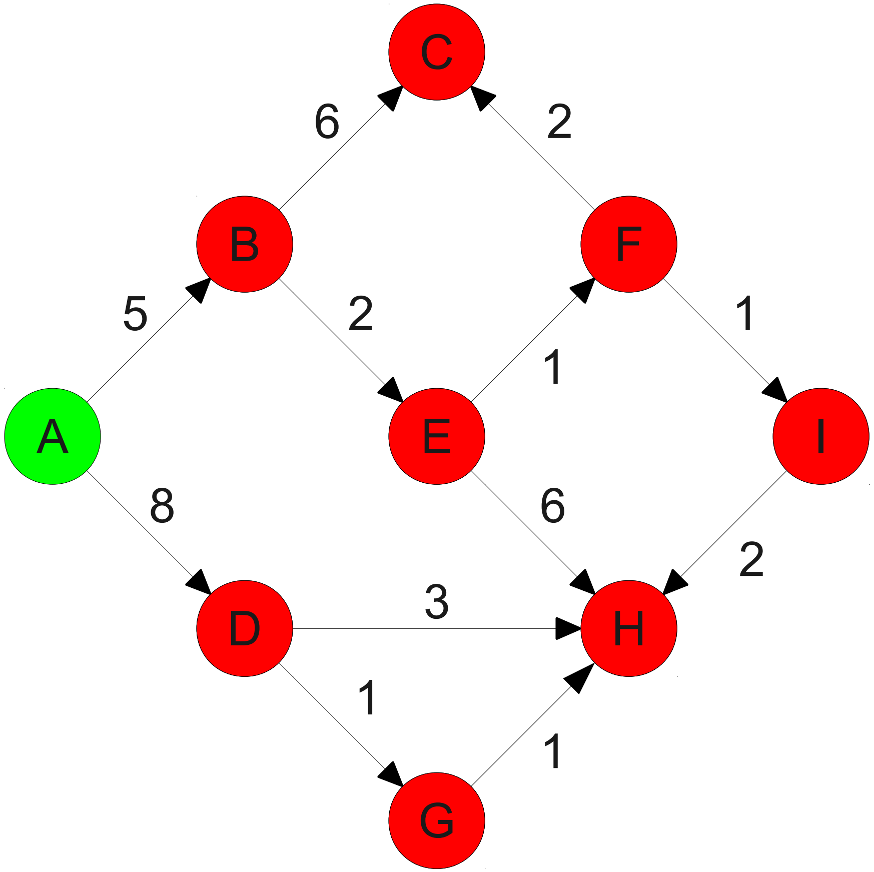


C
6
2
B
F
5
2
1
1
A
E
I
8
6
2
D
3
H
1
1
G

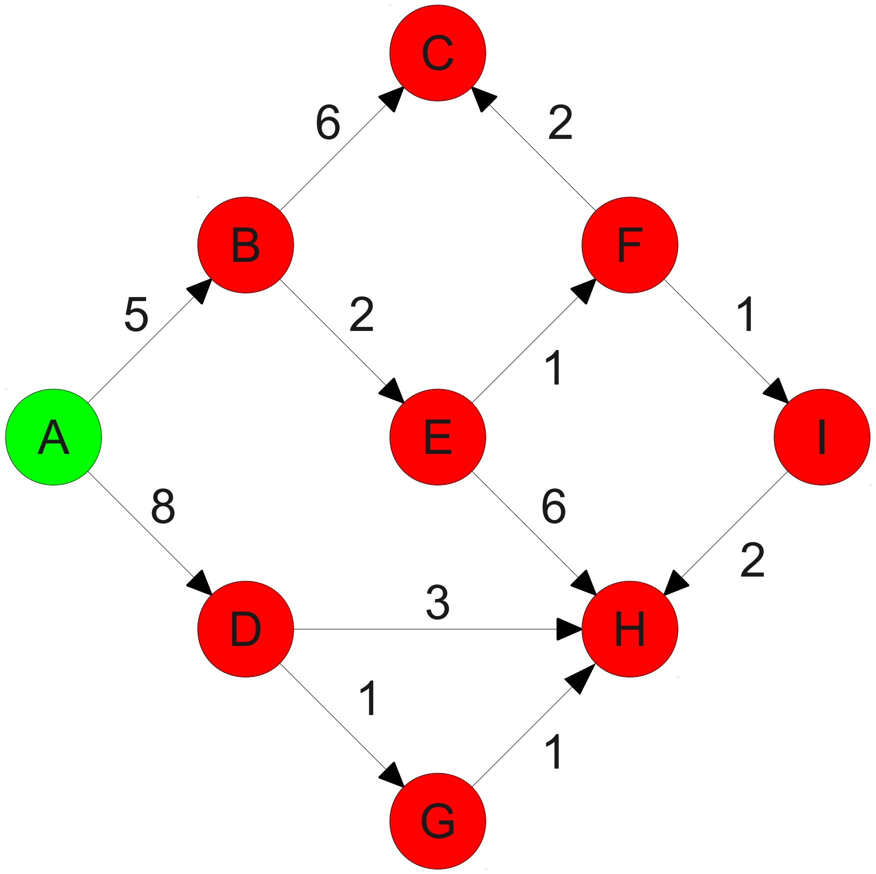

C
6
2
B
F
5
2
1
1
A
E
I
8
6
2
D
3
H
1
1
G



(0) A

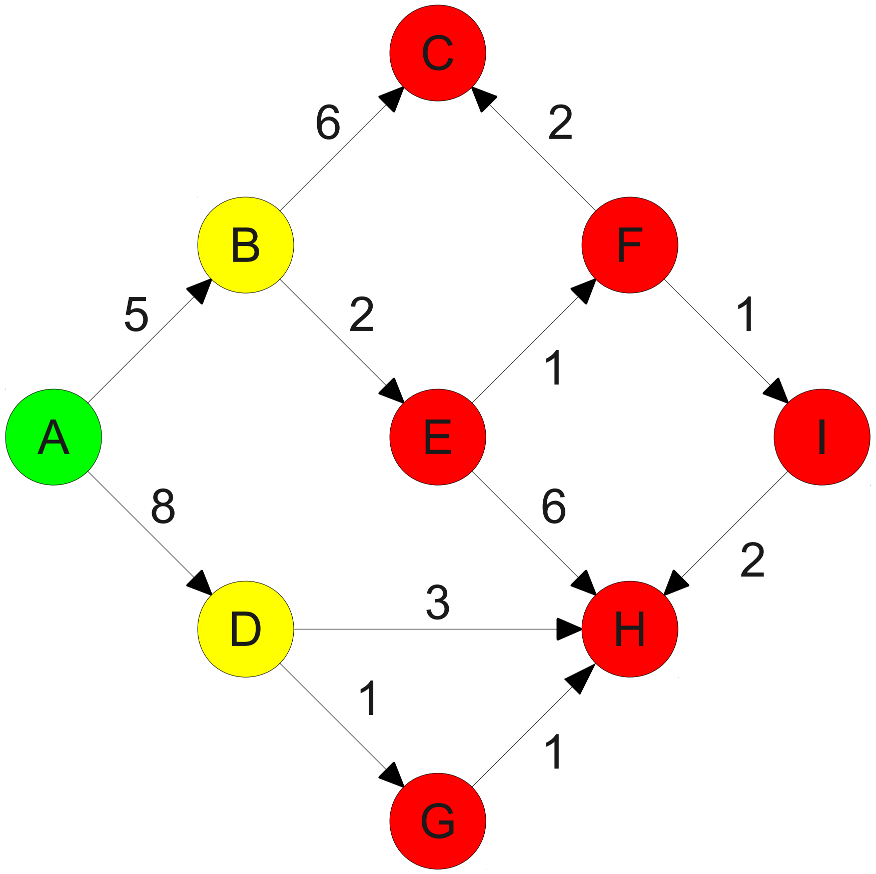


(0) A



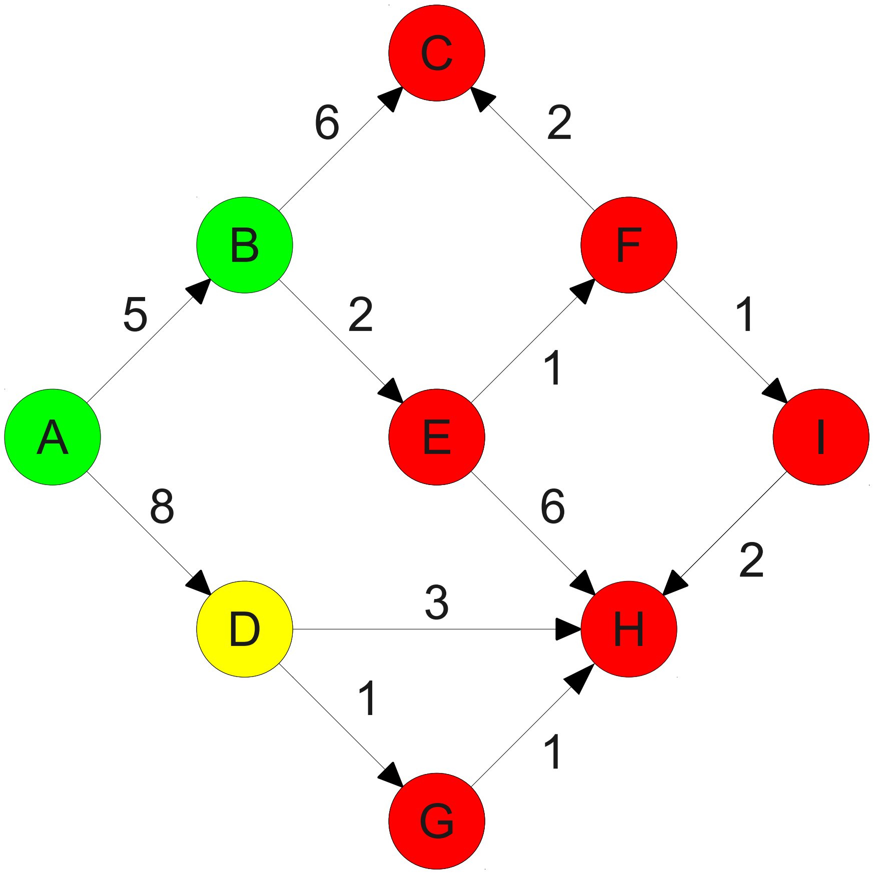
(0) A

(0) A



(5) A→B

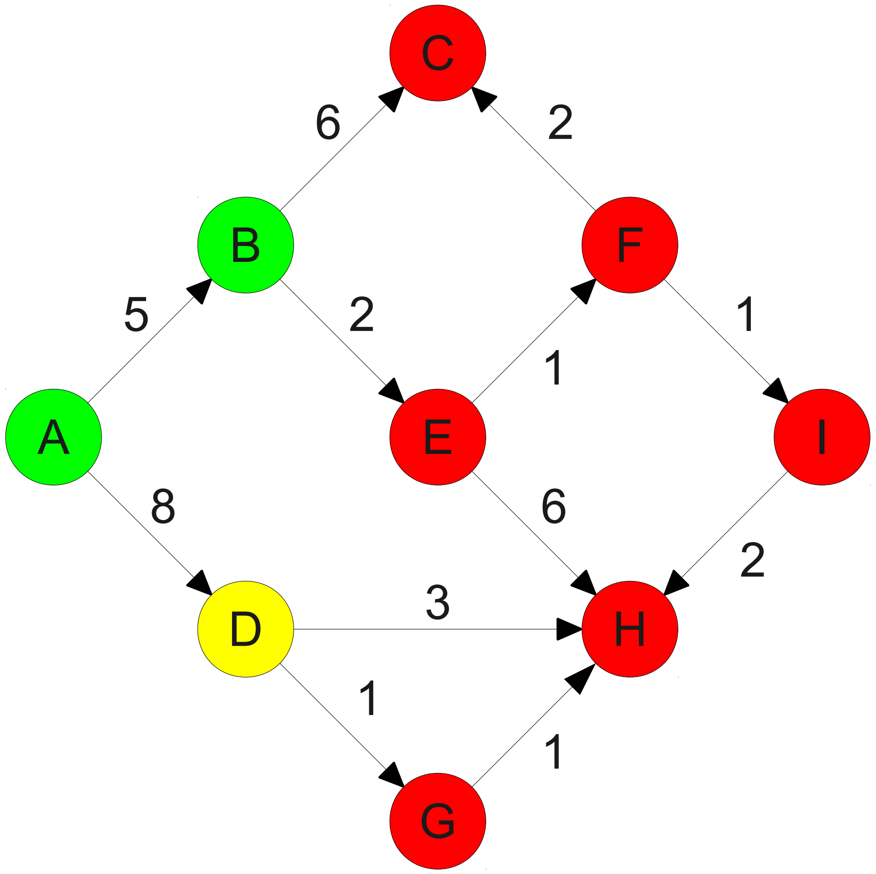
(8) A→D



(0) A

(5) A→B

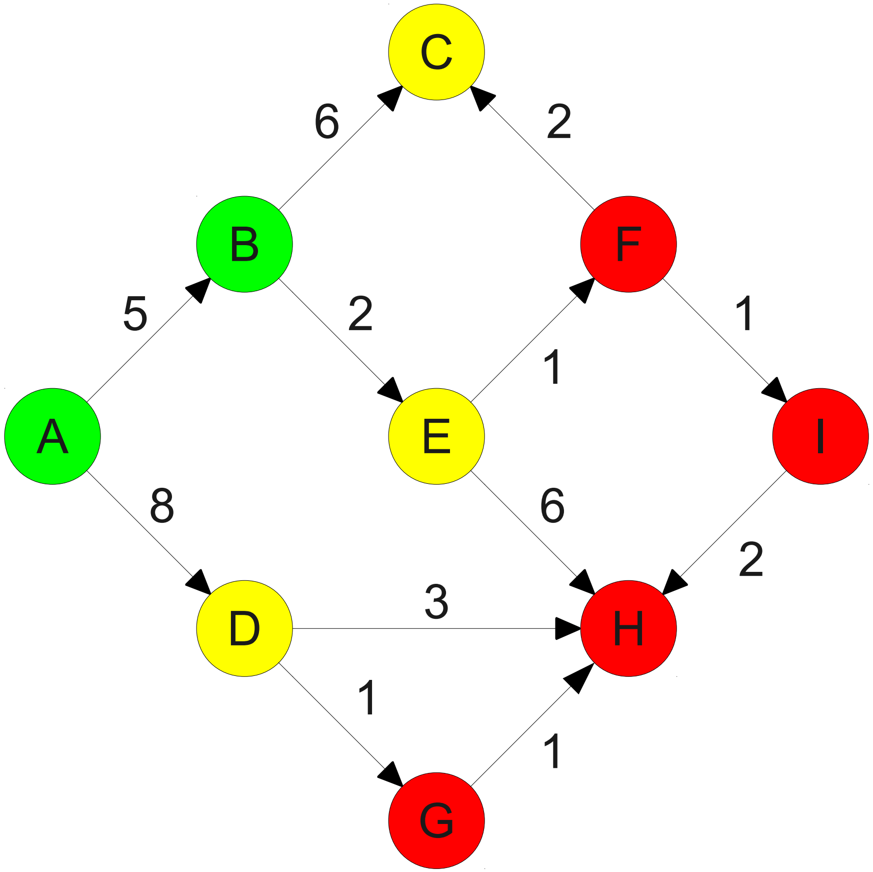
(8) A→D



(0) A

(5) A→B

(8) A→D



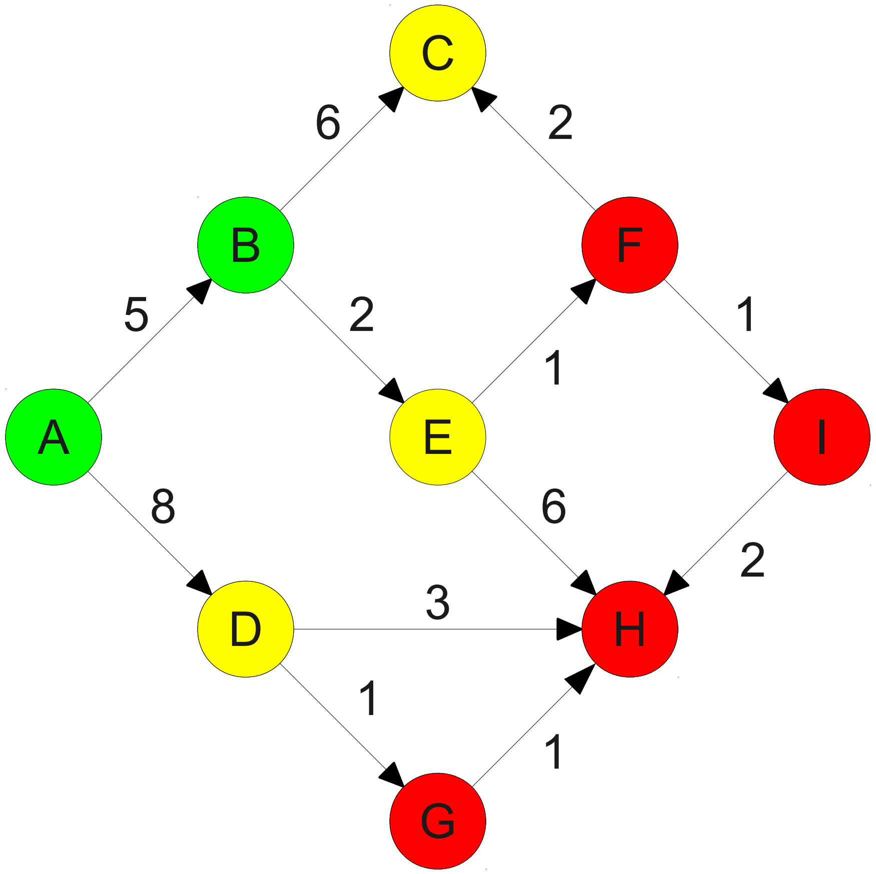
(0) A

(5) A→B

(8) A→D

(11) A→B→C

(7) A→B→E



(0) A

(5) A→B

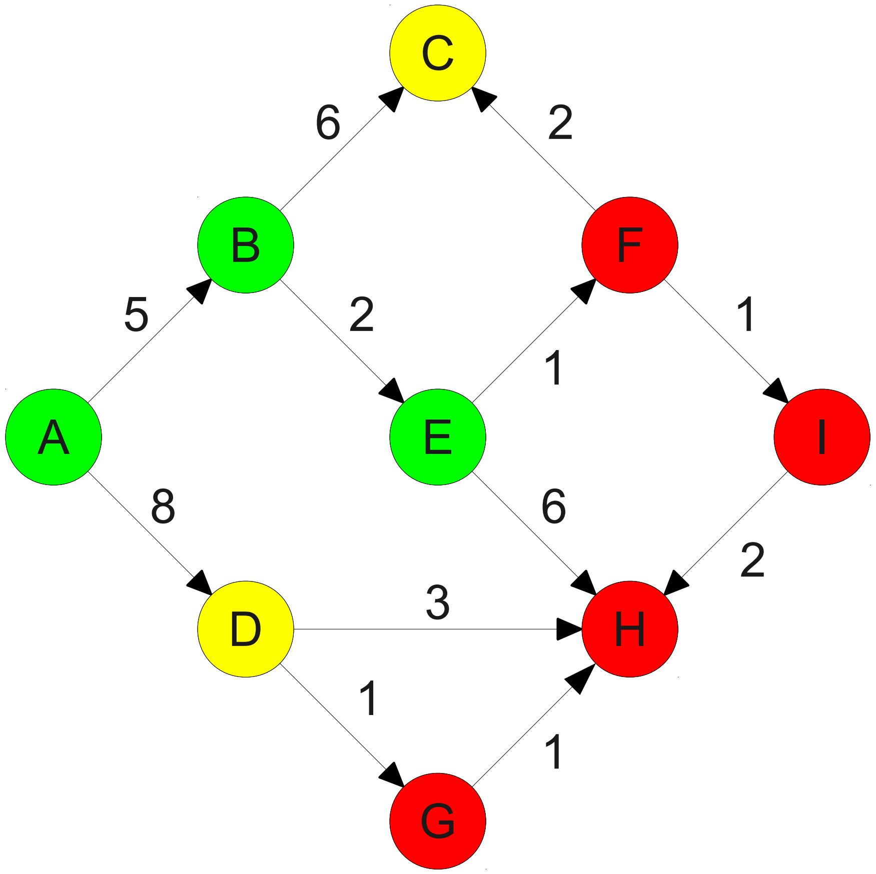
(7) A→B→E

(8) A→D

(11) A→B→C

(0) A

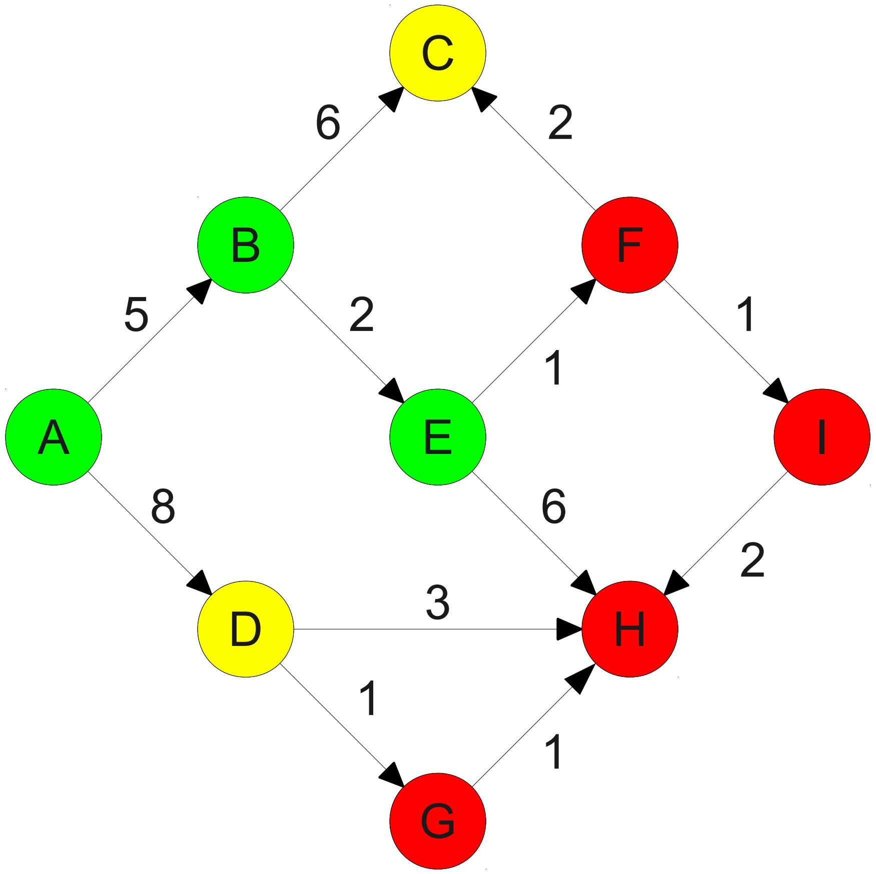
(5) A→B



(7) A→B→E

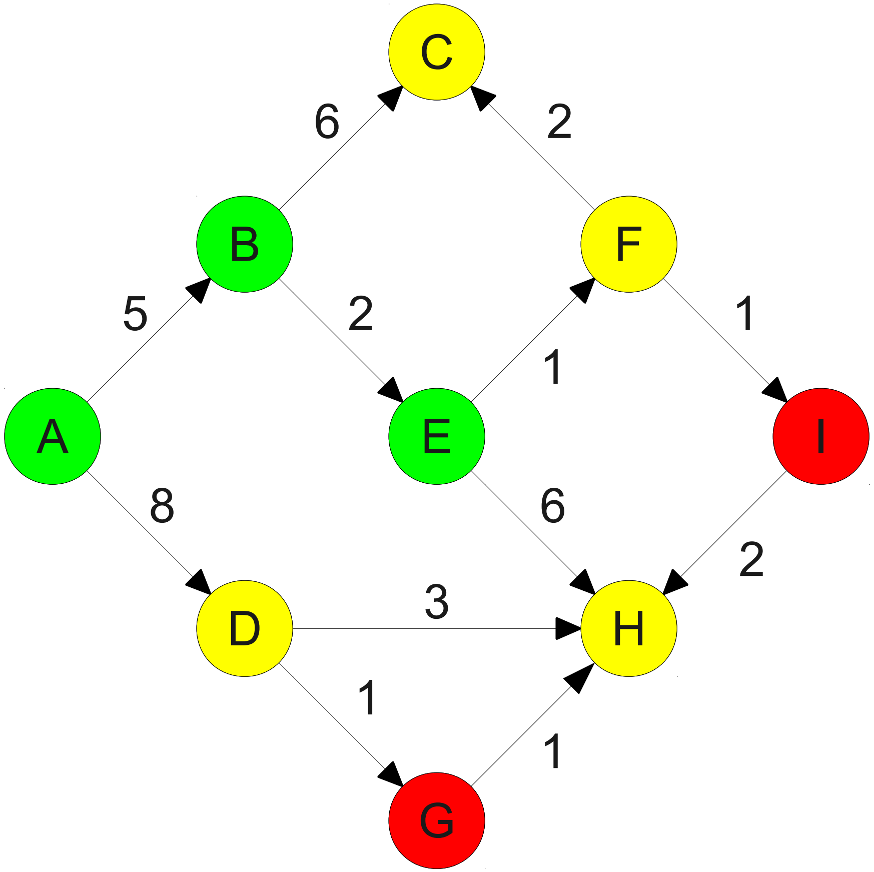
(8) A→D

(11) A→B→C



(0) A
(5) A→B
(7) A→B→E

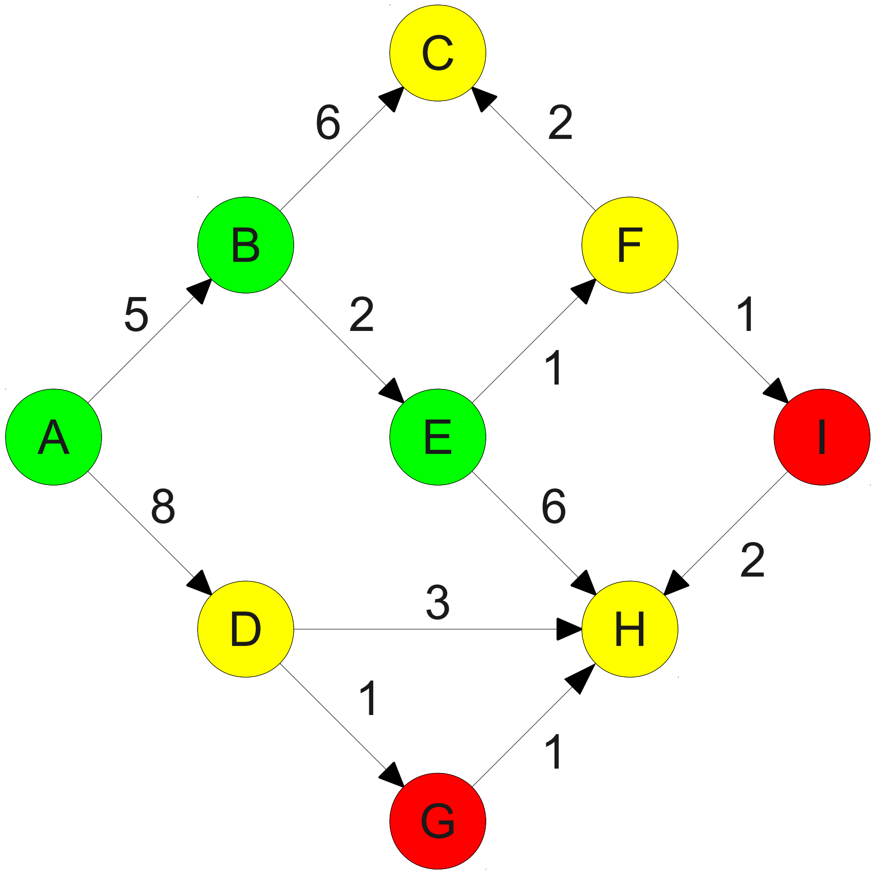
(8) A→D
(11) A→B→C



(0) A
(5) A→B
(7) A→B→E

(8) A→D
(11) A→B→C

(8) A→B→E→F
(13) A→B→E→H



(0) A

(5) A→B

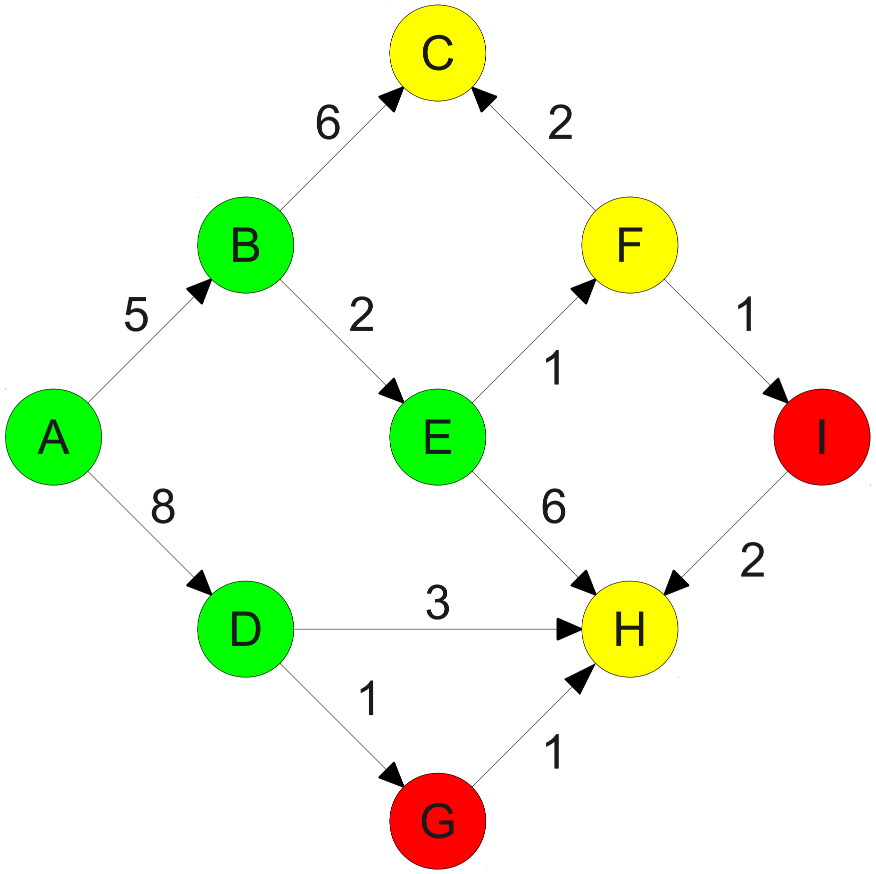
(7) A→B→E

(8) A→D

(8) A→B→E→F

(11) A→B→C

(13) A→B→E→H



(0) A

(5) A→B

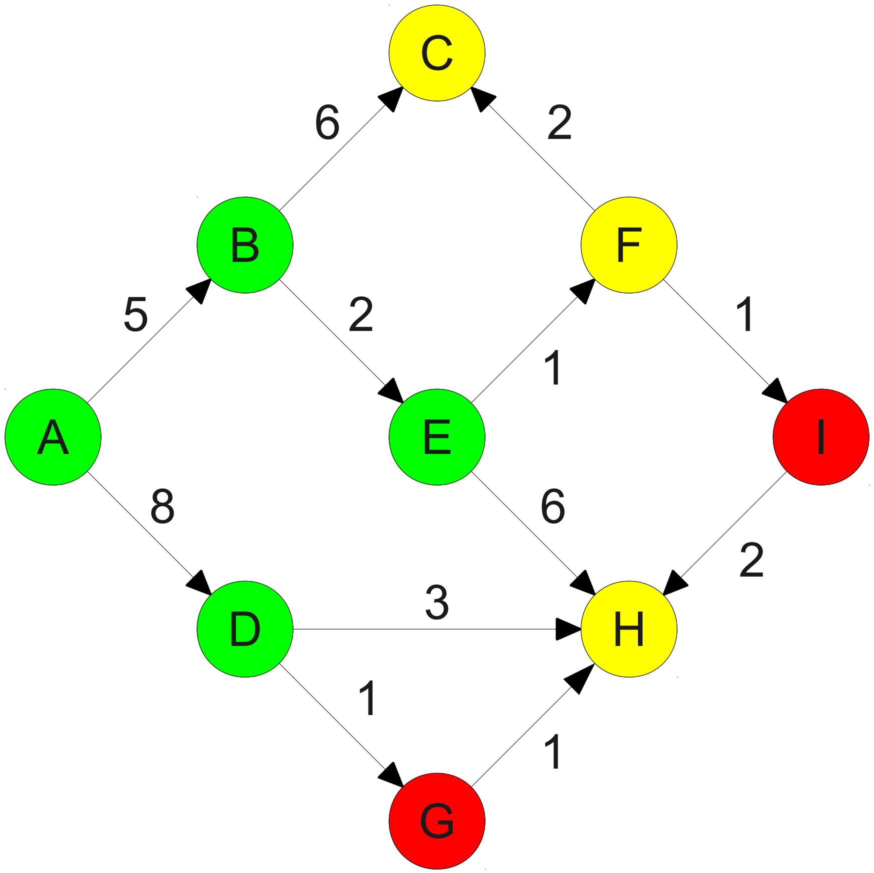
(7) A→B→E

(8) A→D

(8) A→B→E→F

(11) A→B→C

(13) A→B→E→H



(0) A

(5) A→B

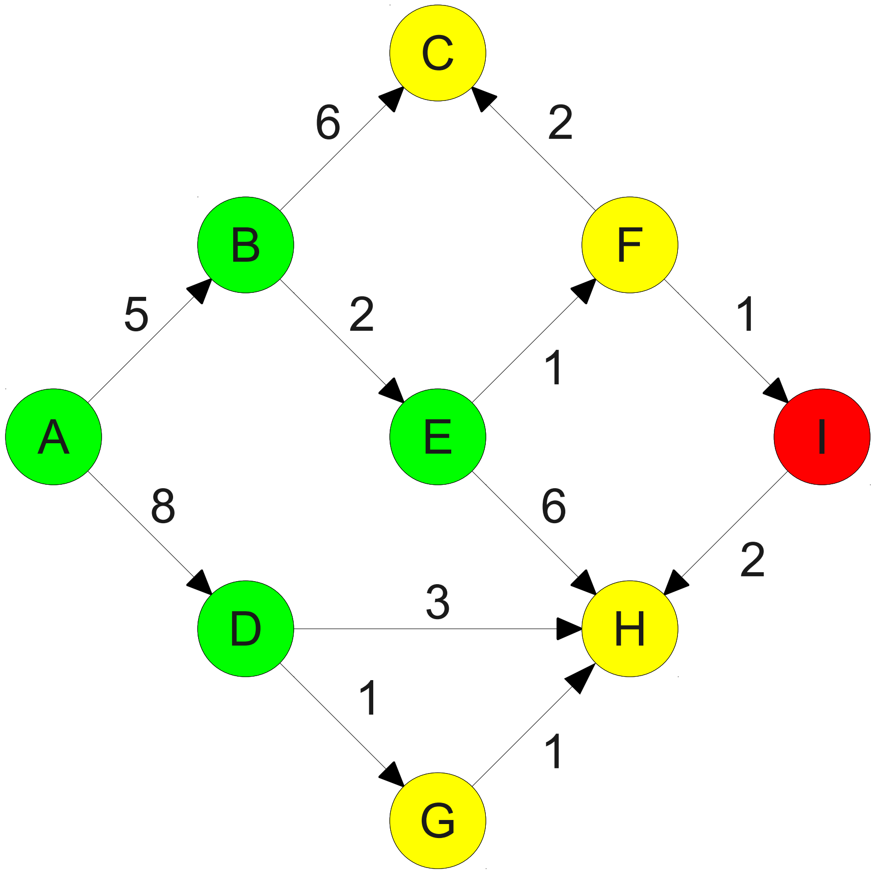
(7) A→B→E

(8) A→D

(8) A→B→E→F

(11) A→B→C

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

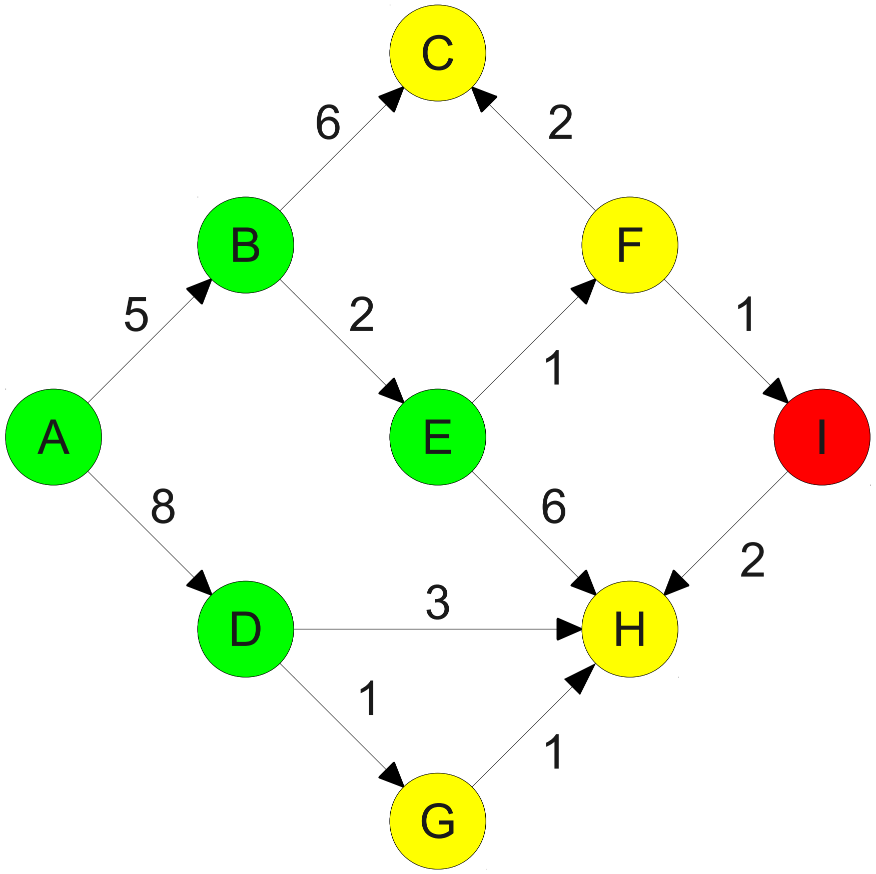
(8) A→B→E→F

(11) A→B→C

(13) A→B→E→H

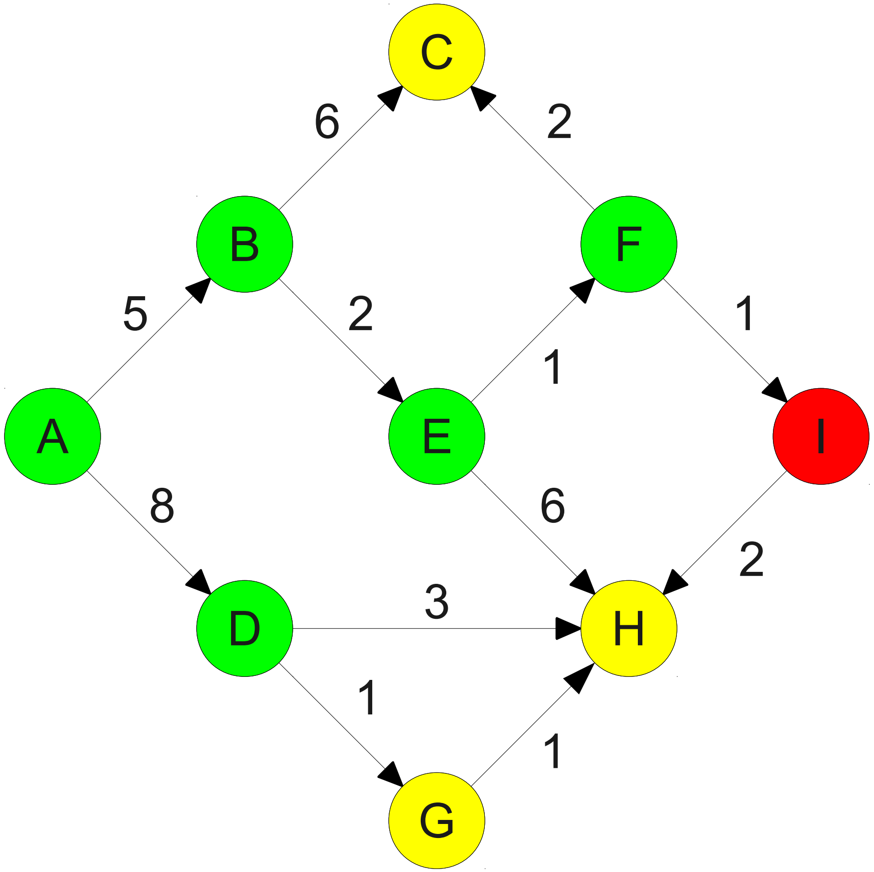
(9) A→D→G

(11) A→D→H



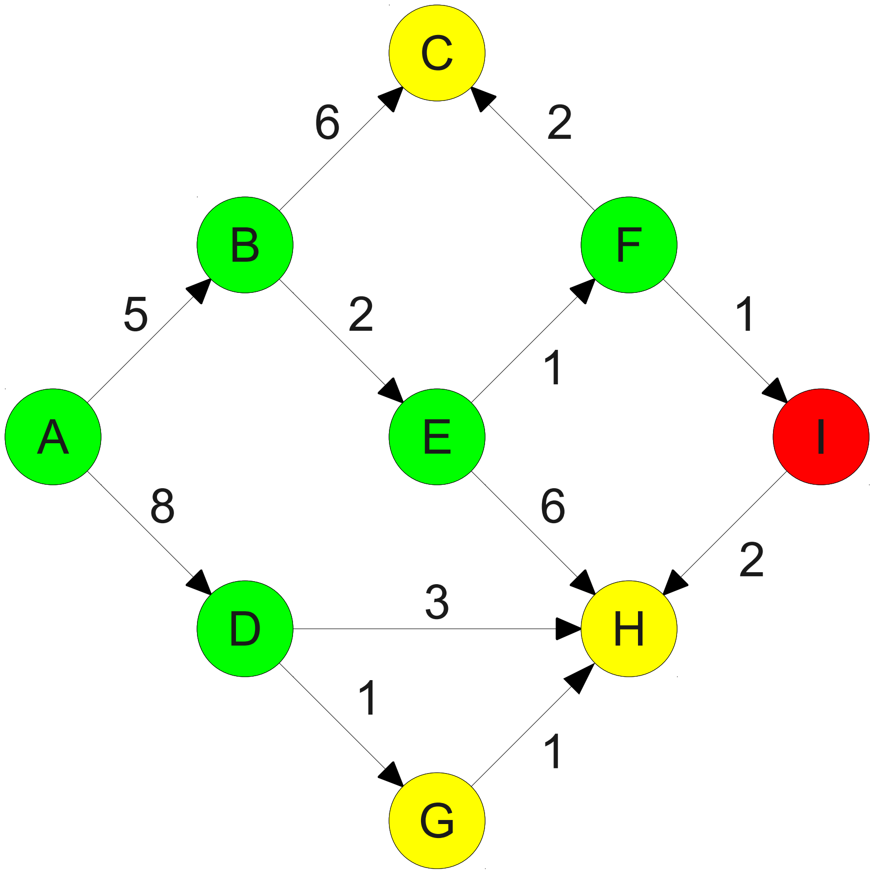
(0) A
(5) A→B
(7) A→B→E
(8) A→D

(8) A→B→E→F
(9) A→D→G
(11) A→B→C
(11) A→D→H
(13) A→B→E→H



(0) A
(5) A→B
(7) A→B→E
(8) A→D

(8) A→B→E→F
(9) A→D→G
(11) A→B→C
(11) A→D→H
(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

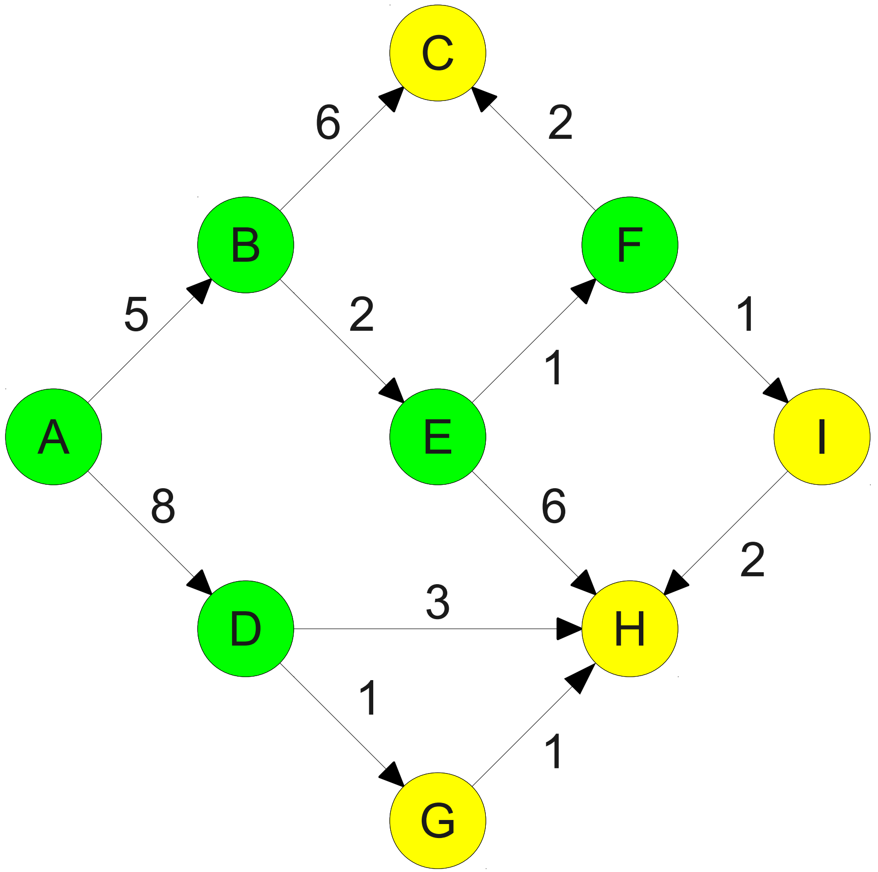
(8) A→B→E→F

(9) A→D→G

(11) A→B→C

(11) A→D→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

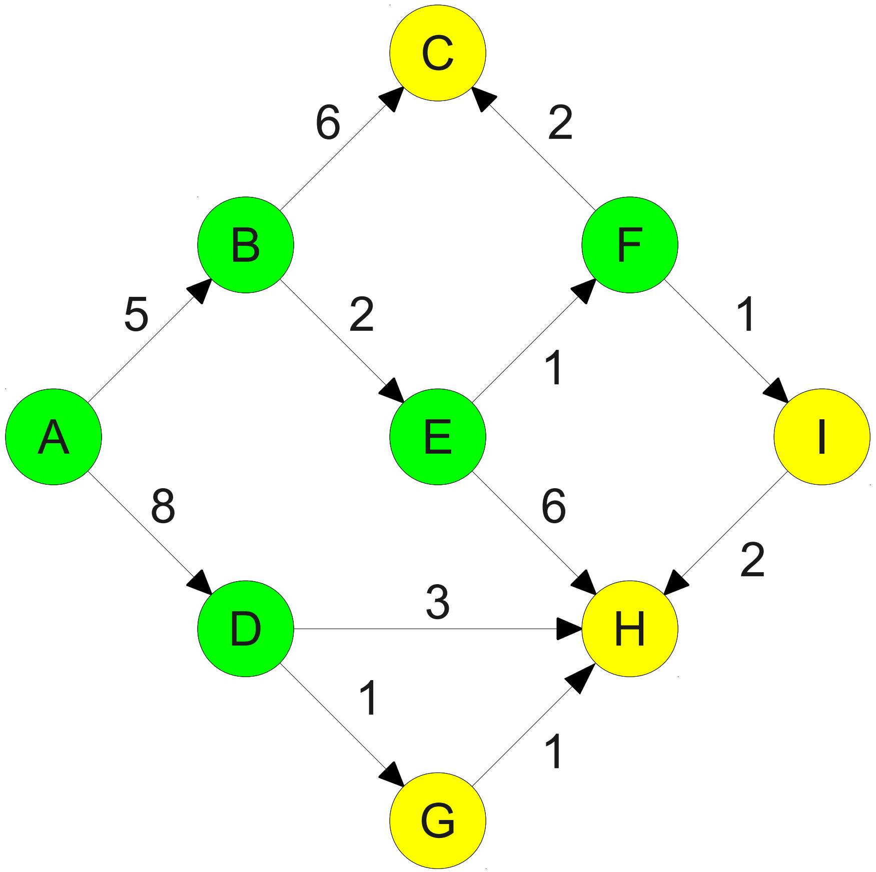
(11) A→B→C

(11) A→D→H

(13) A→B→E→H

(9) A→B→E→F→I

(10) A→B→E→F→C



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

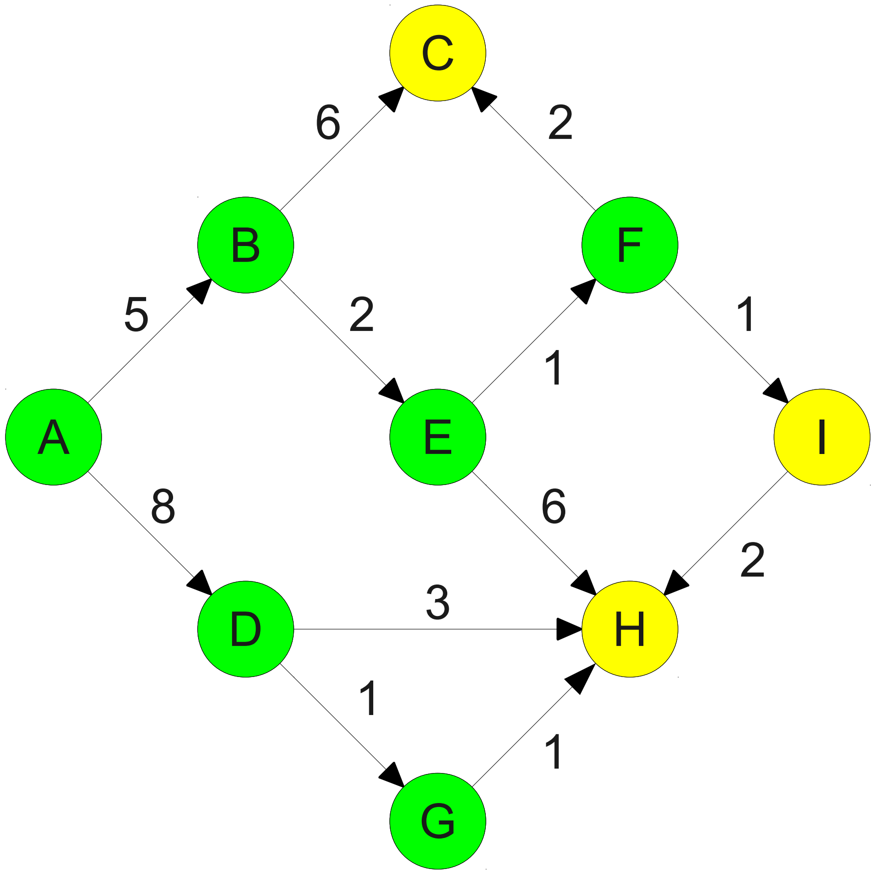
(9) A→B→E→F→I

(10) A→B→E→F→C

(11) A→B→C

(11) A→D→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

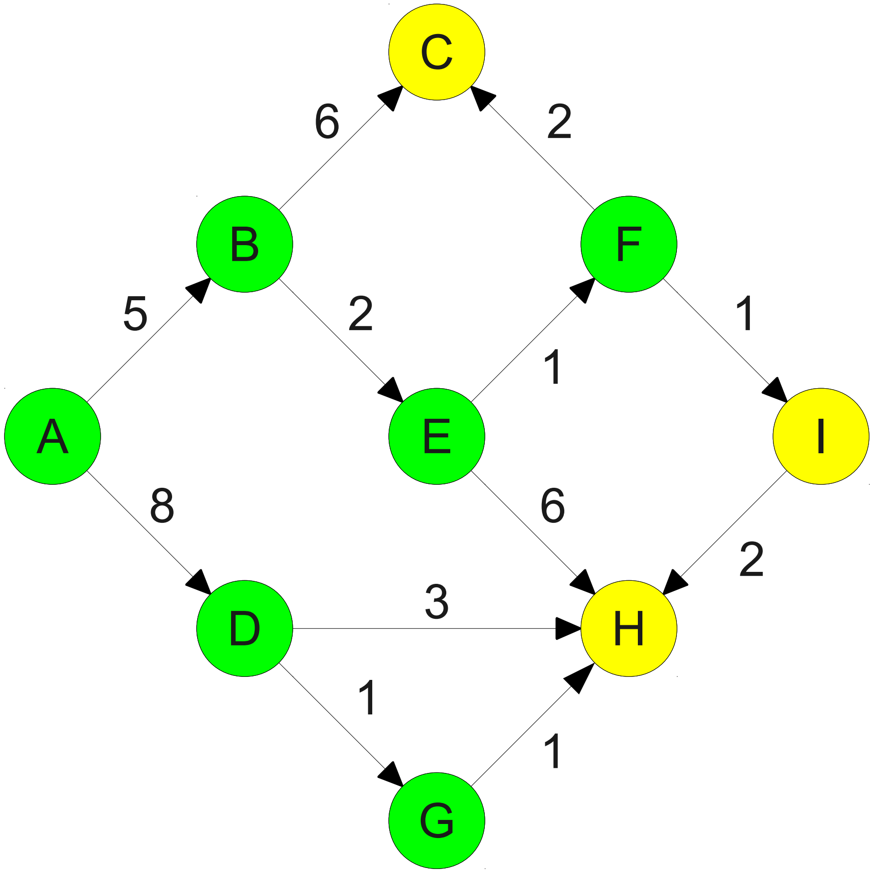
(9) A→B→E→F→I

(10) A→B→E→F→C

(11) A→B→C

(11) A→D→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

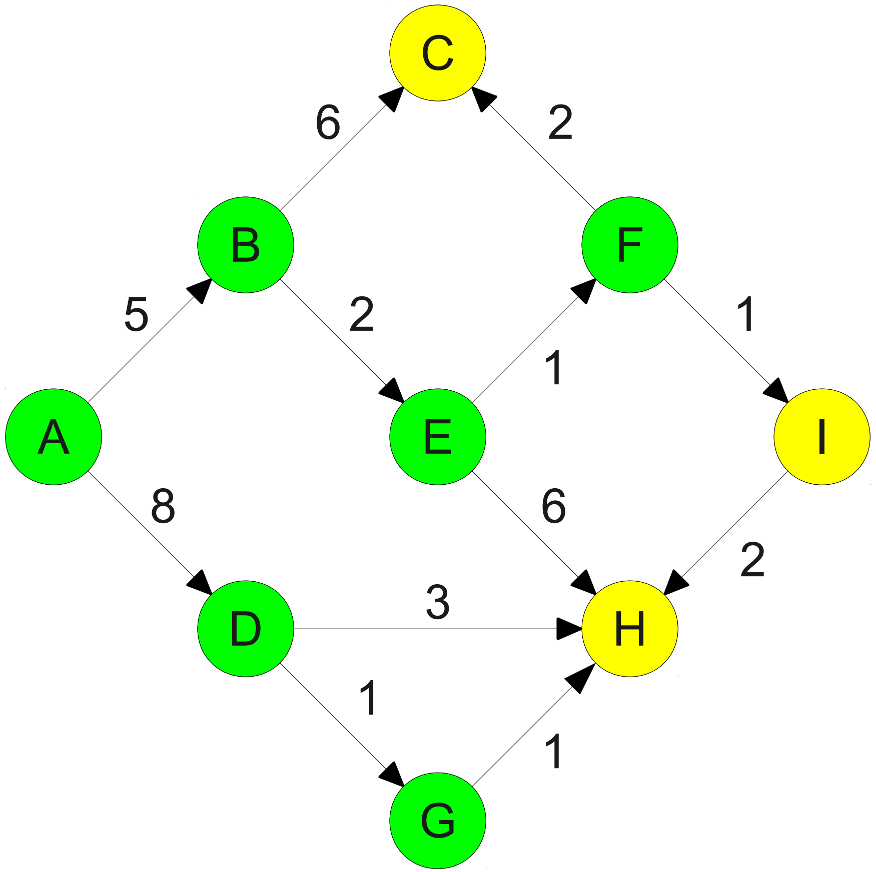
(9) A→B→E→F→I

(10) A→B→E→F→C

(11) A→B→C

(11) A→D→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

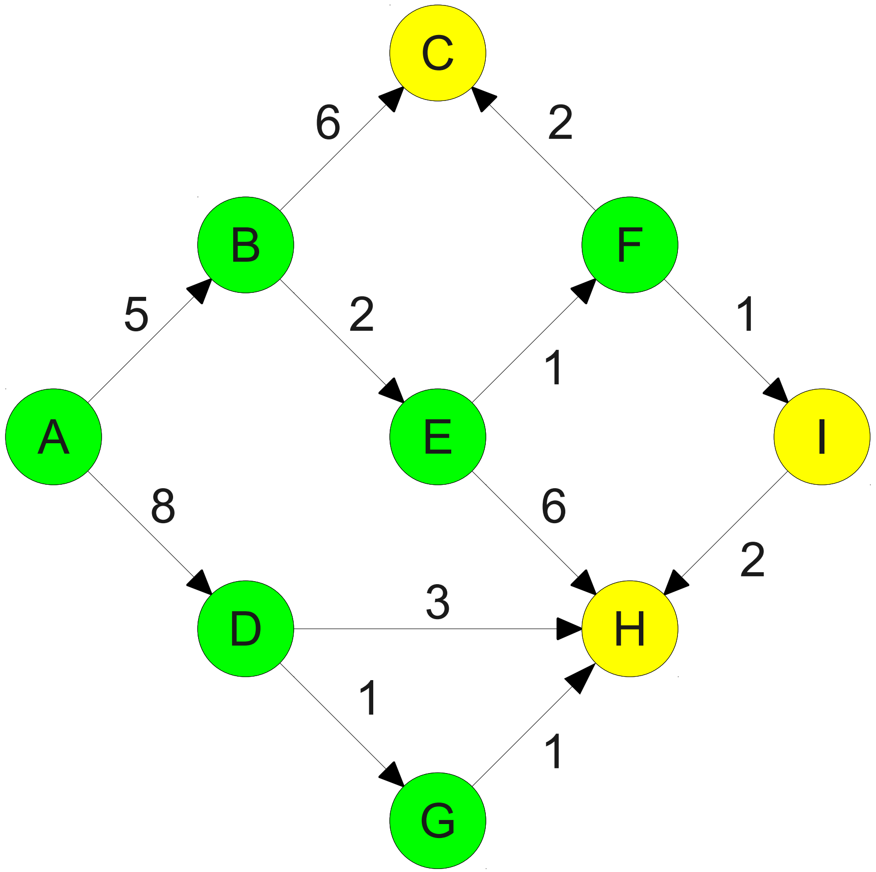
(10) A→B→E→F→C

(11) A→B→C

(11) A→D→H

(13) A→B→E→H

(10) A→D→G→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

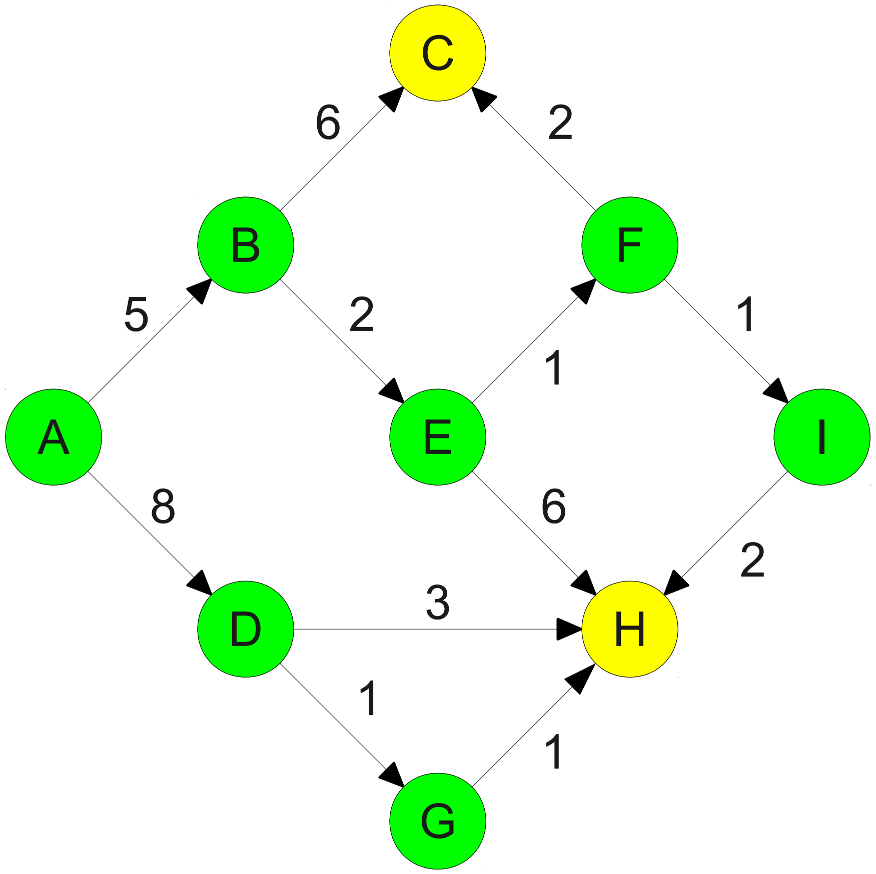
(10) A→B→E→F→C

(10) A→D→G→H

(11) A→B→C

(11) A→D→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

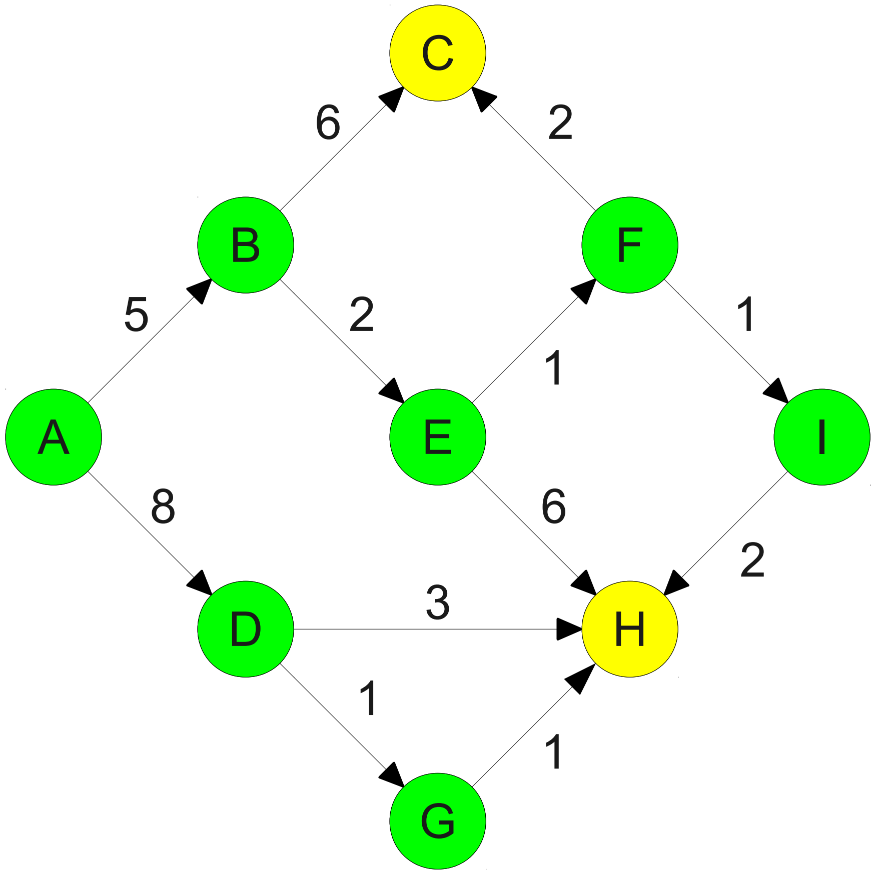
(10) A→B→E→F→C

(10) A→D→G→H

(11) A→B→C

(11) A→D→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

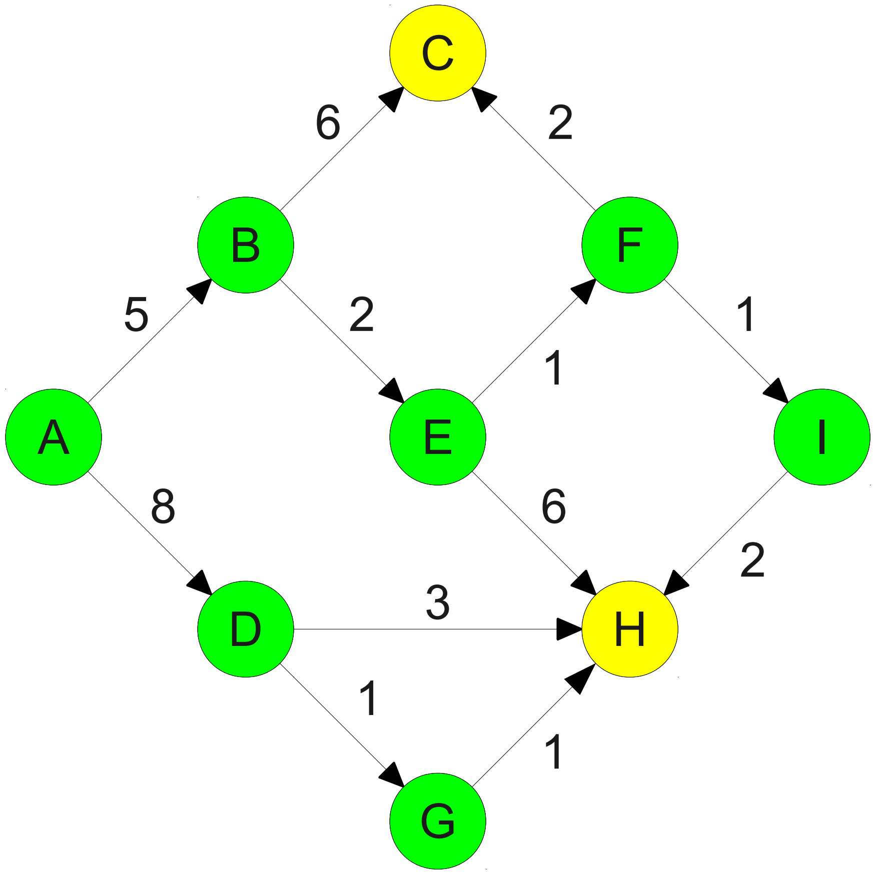
(10) A→B→E→F→C

(10) A→D→G→H

(11) A→B→C

(11) A→D→H

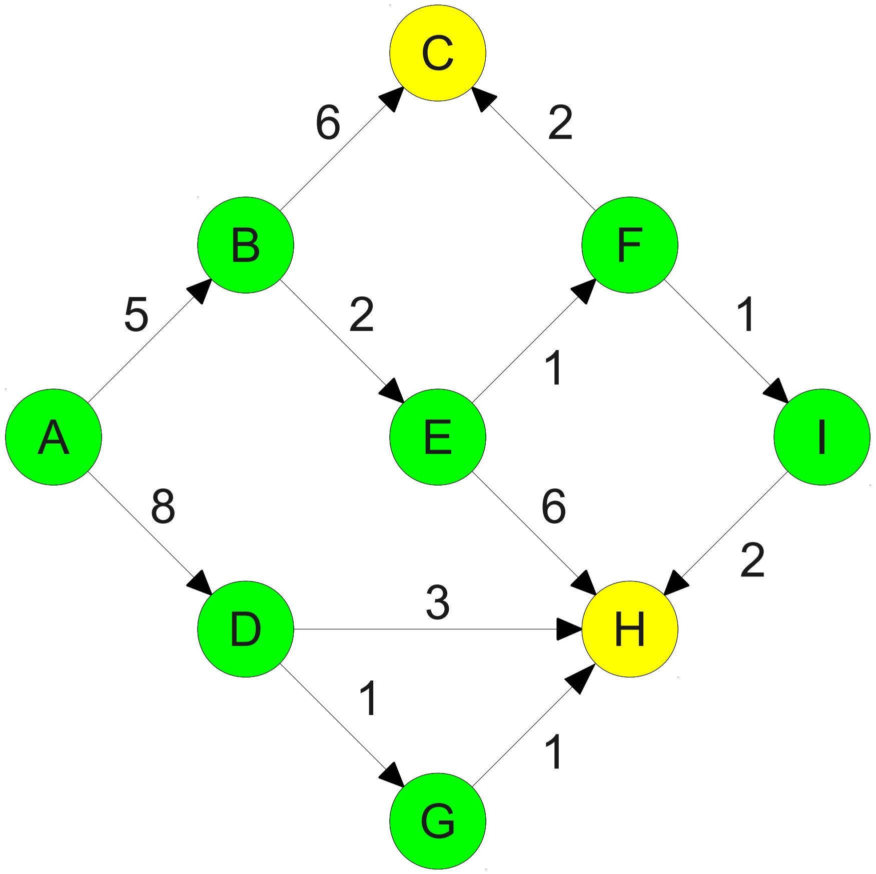
(13) A→B→E→H



(0) A
(5) A→B
(7) A→B→E
(8) A→D
(8) A→B→E→F
(9) A→D→G
(9) A→B→E→F→I

(10) A→B→E→F→C
(10) A→D→G→H
(11) A→B→C
(11) A→D→H
(13) A→B→E→H

(11) A→B→E→F→I→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

(10) A→B→E→F→C

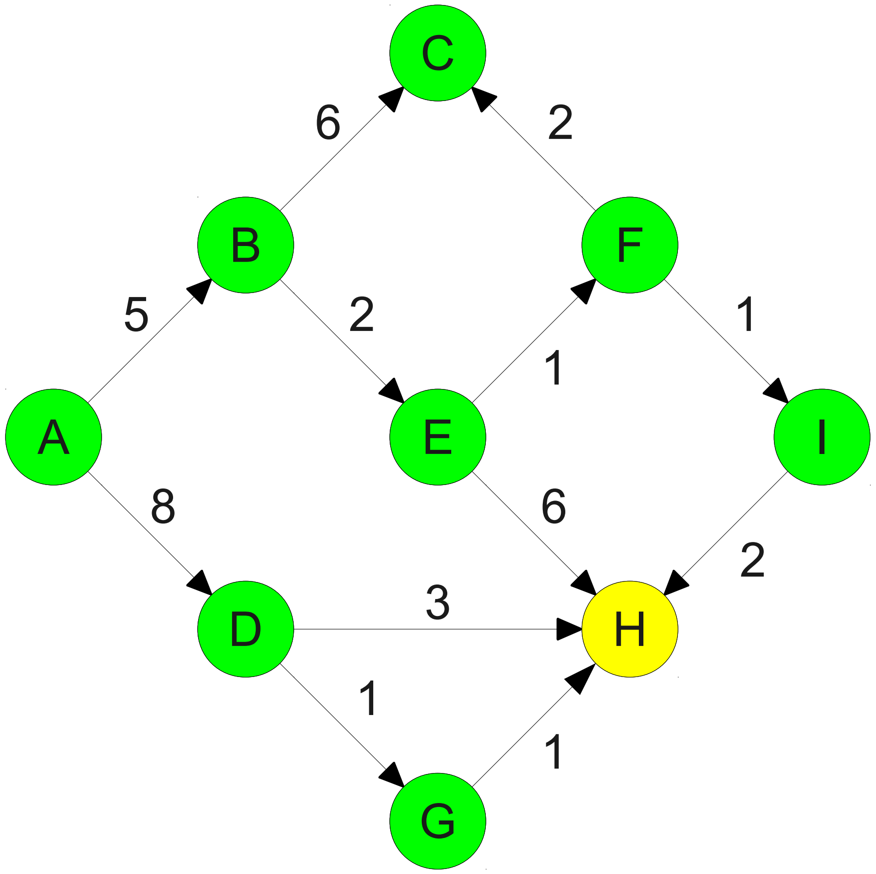
(10) A→D→G→H

(11) A→B→C

(11) A→D→H

(11) A→B→E→F→I→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

(10) A→B→E→F→C

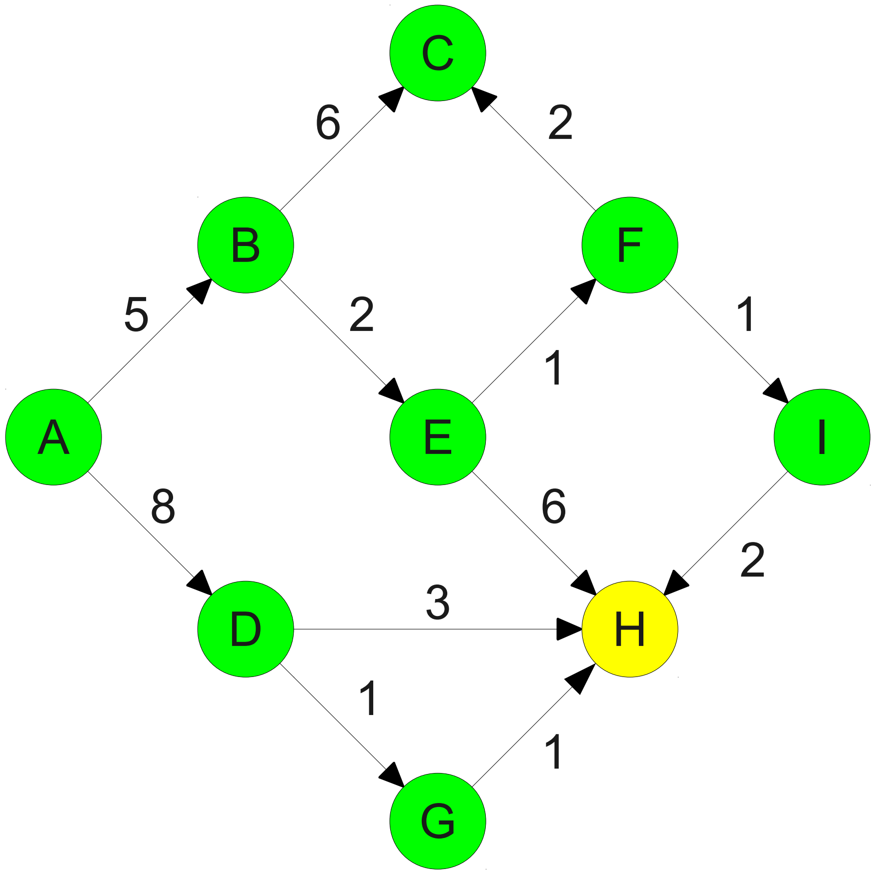
(10) A→D→G→H

(11) A→B→C

(11) A→D→H

(11) A→B→E→F→I→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

(10) A→B→E→F→C

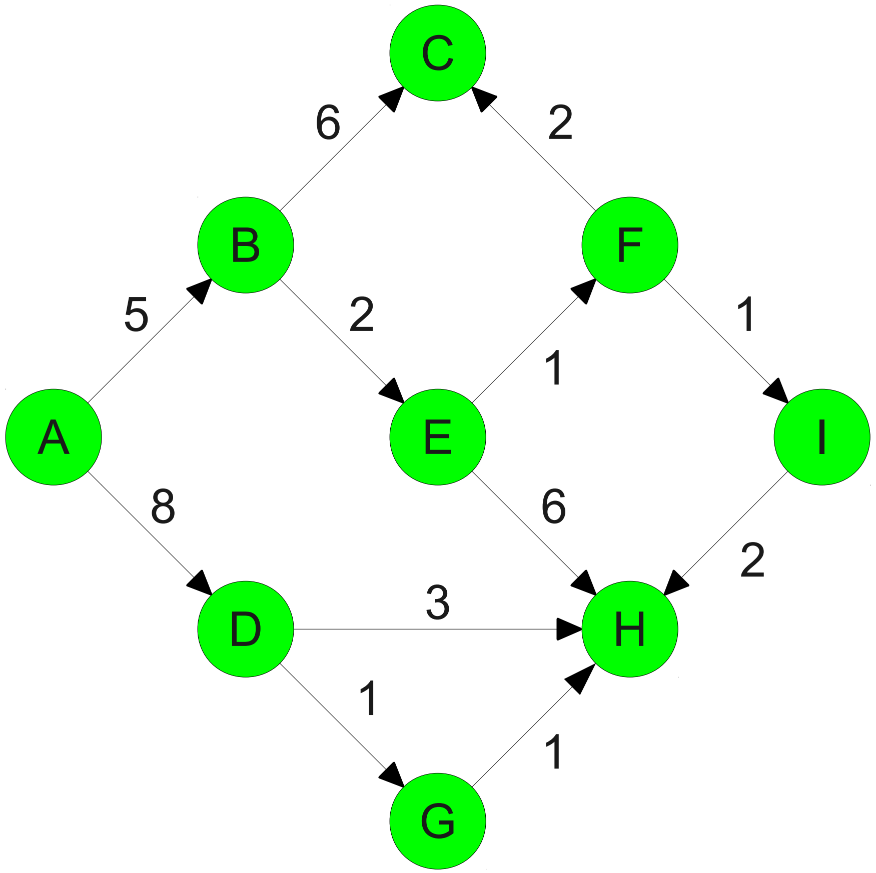
(10) A→D→G→H

(11) A→B→C

(11) A→D→H

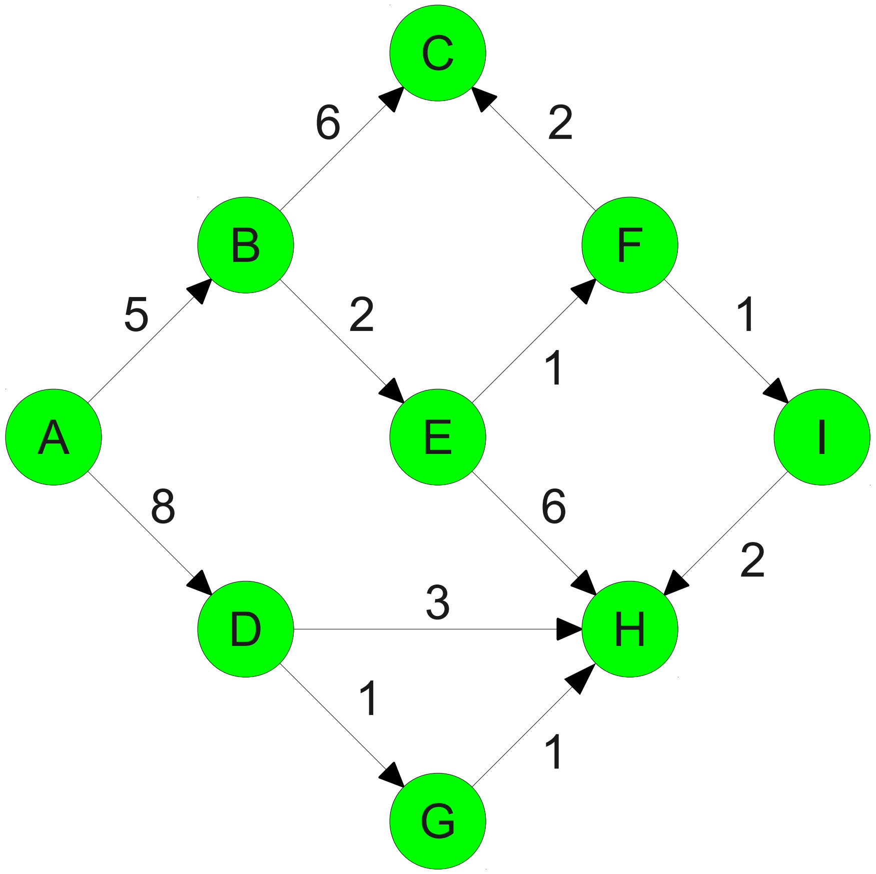
(11) A→B→E→F→I→H

(13) A→B→E→H



(0) A
(5) A→B
(7) A→B→E
(8) A→D
(8) A→B→E→F
(9) A→D→G
(9) A→B→E→F→I
(10) A→B→E→F→C

(10) A→D→G→H
(11) A→B→C
(11) A→D→H
(11) A→B→E→F→I→H
(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

(10) A→B→E→F→C

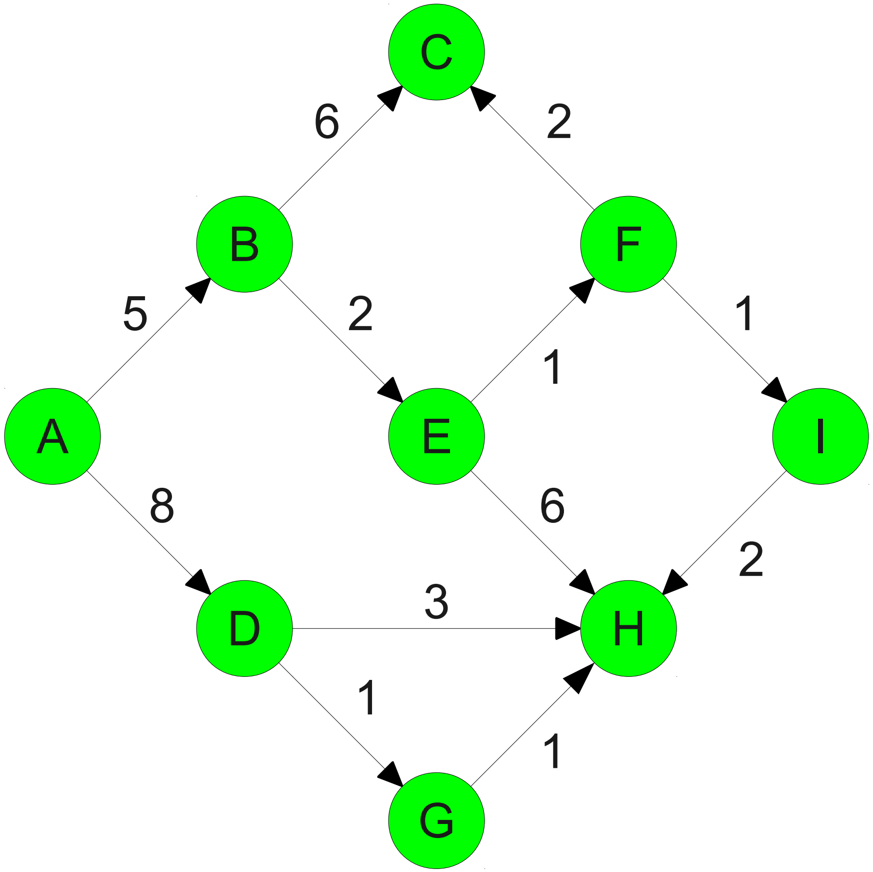
(10) A→D→G→H

(11) A→B→C

(11) A→D→H

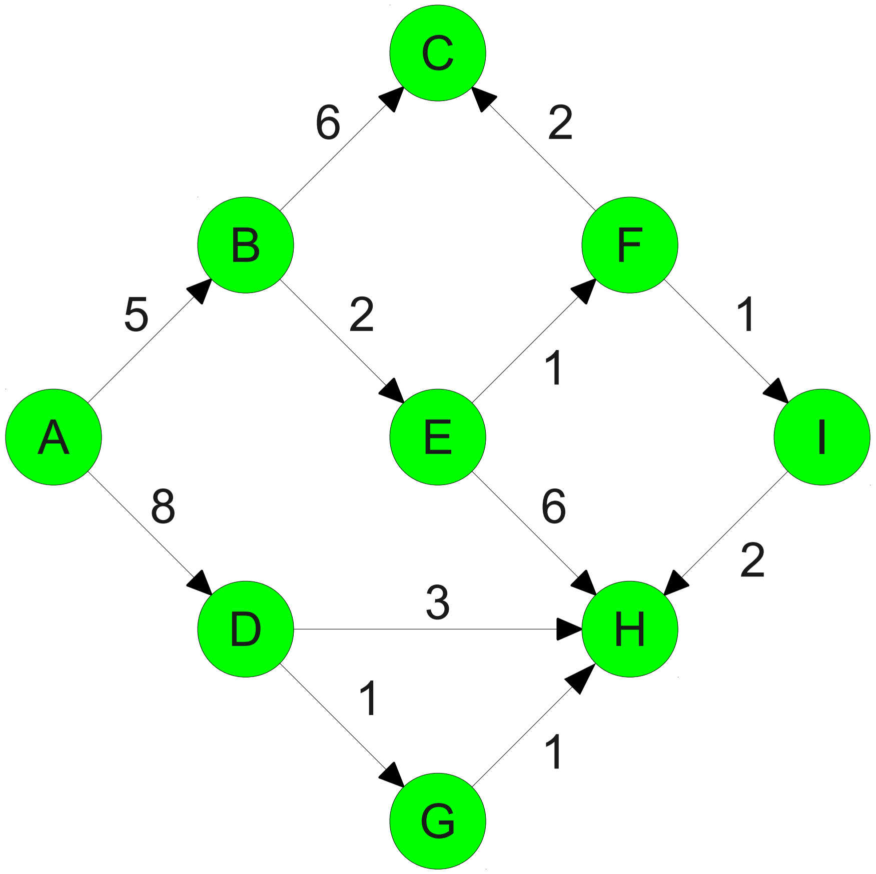
(11) A→B→E→F→I→H

(13) A→B→E→H



(0) A
(5) A→B
(7) A→B→E
(8) A→D
(8) A→B→E→F
(9) A→D→G
(9) A→B→E→F→I
(10) A→B→E→F→C
(10) A→D→G→H

(11) A→D→H
(11) A→B→E→F→I→H
(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

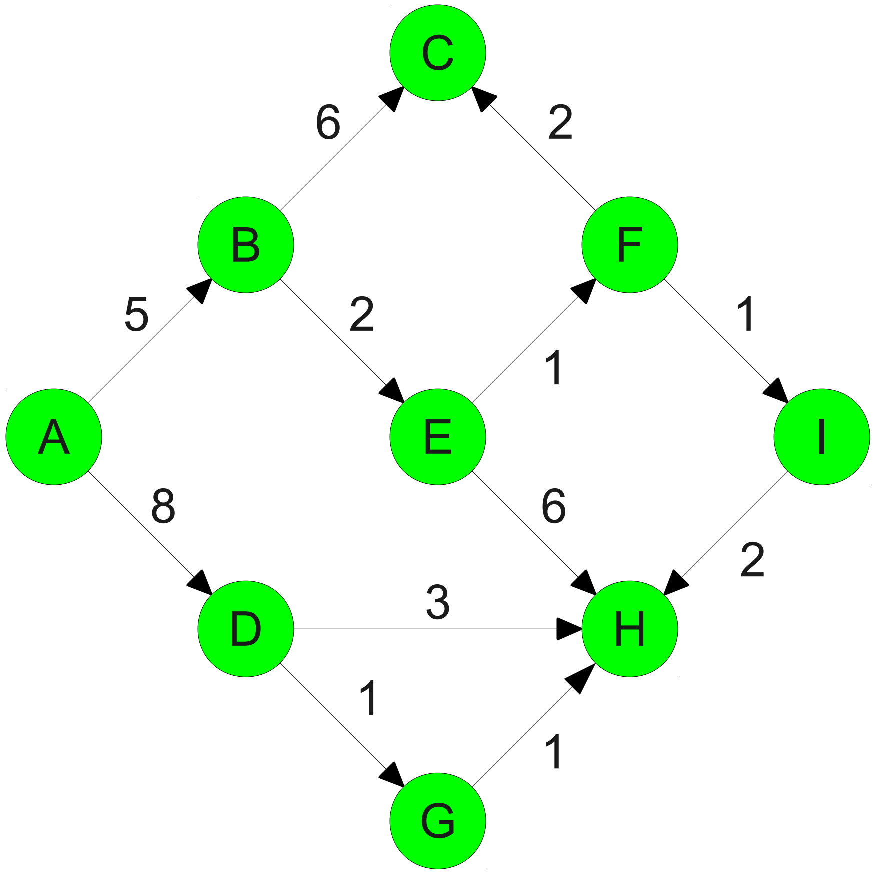
(9) A→B→E→F→I

(10) A→B→E→F→C

(10) A→D→G→H

(11) A→B→E→F→I→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

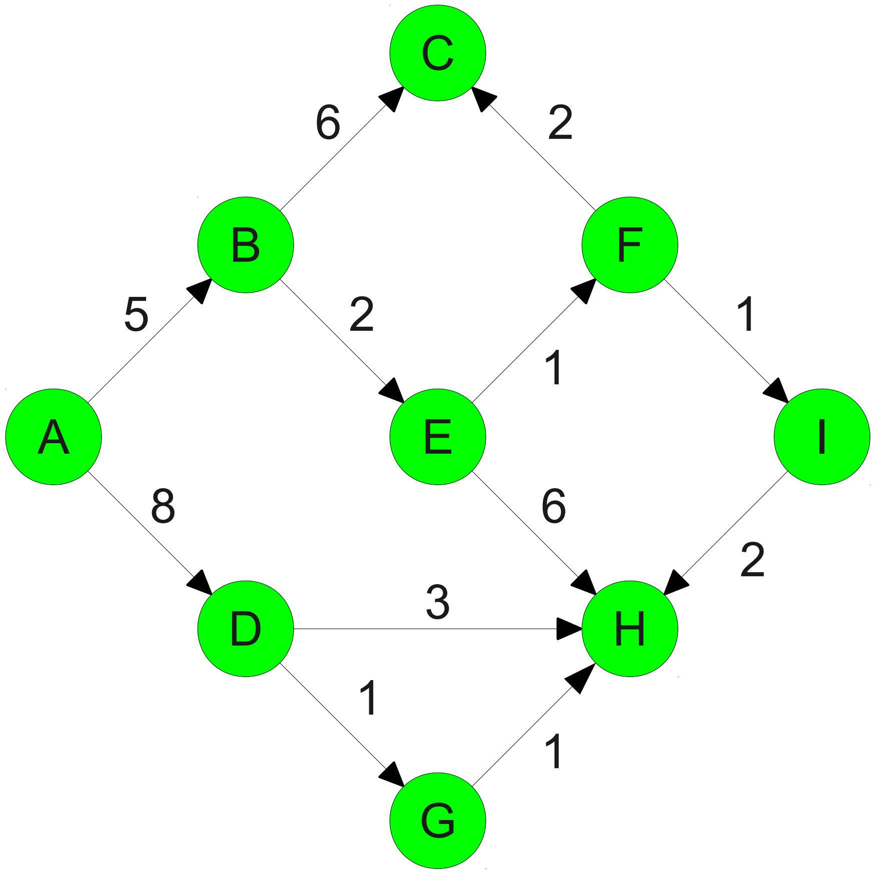
(9) A→D→G

(9) A→B→E→F→I

(10) A→B→E→F→C

(10) A→D→G→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

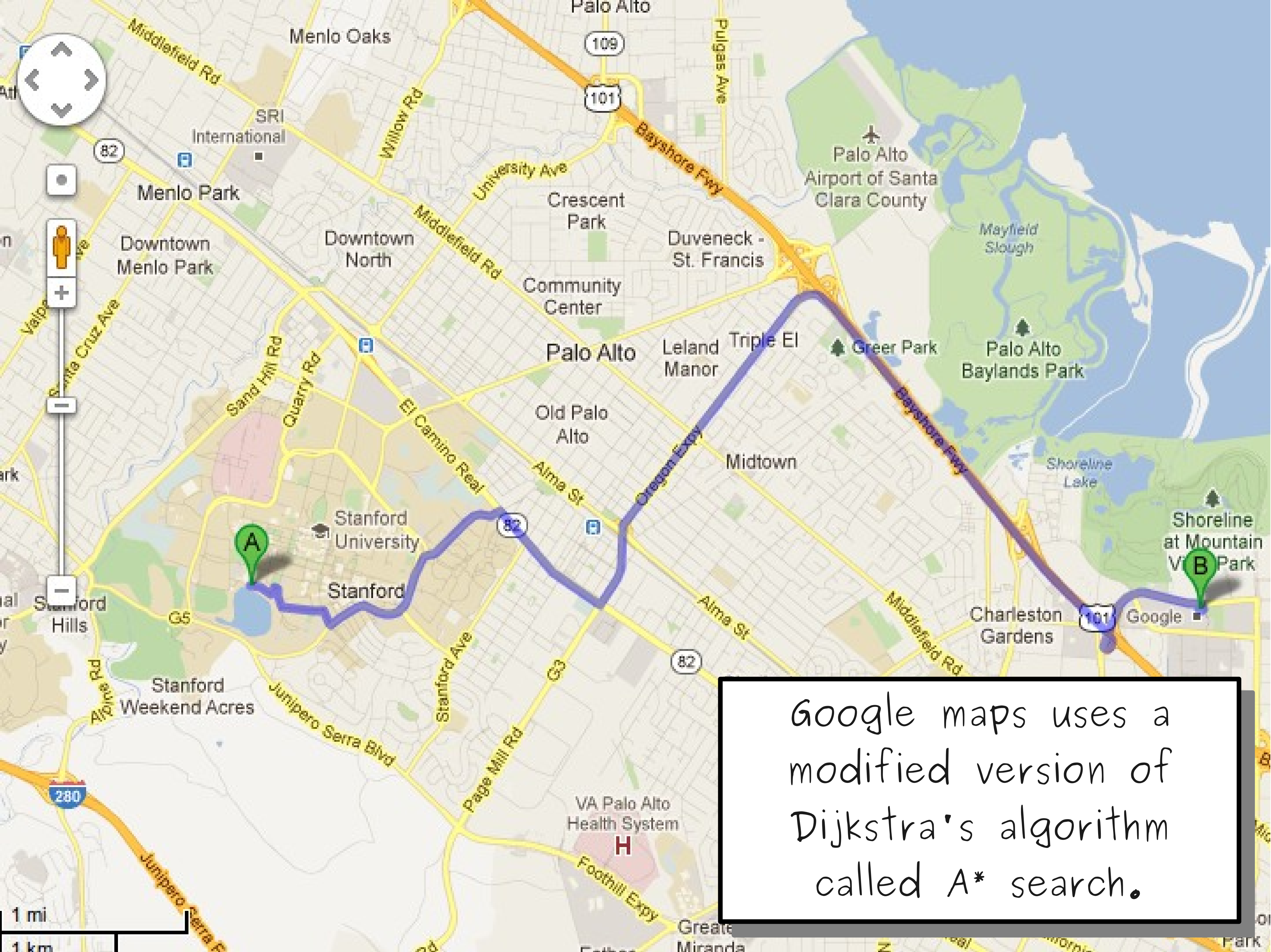
(9) A→B→E→F→I

(10) A→B→E→F→C

(10) A→D→G→H

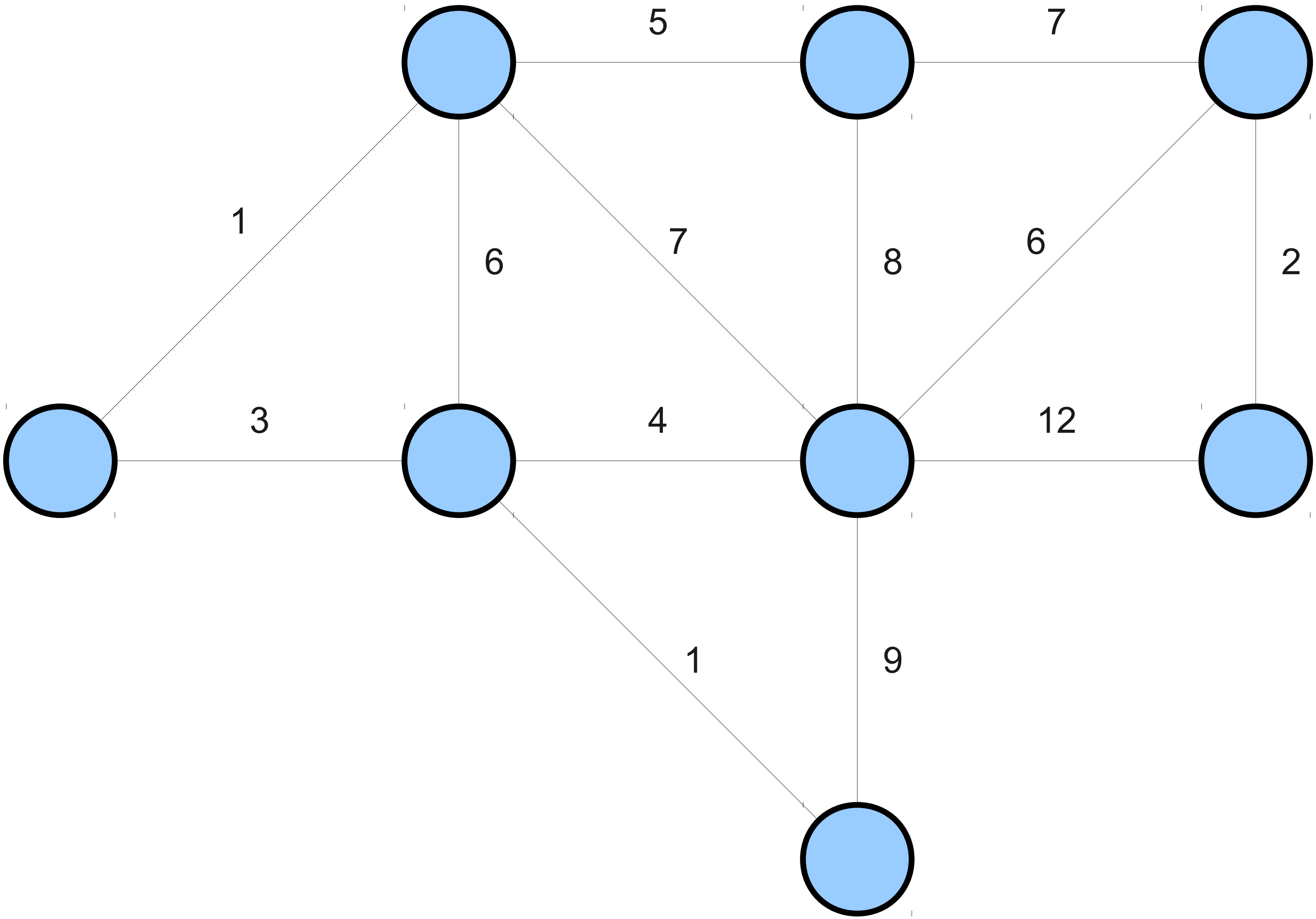
Dijkstra's Algorithm

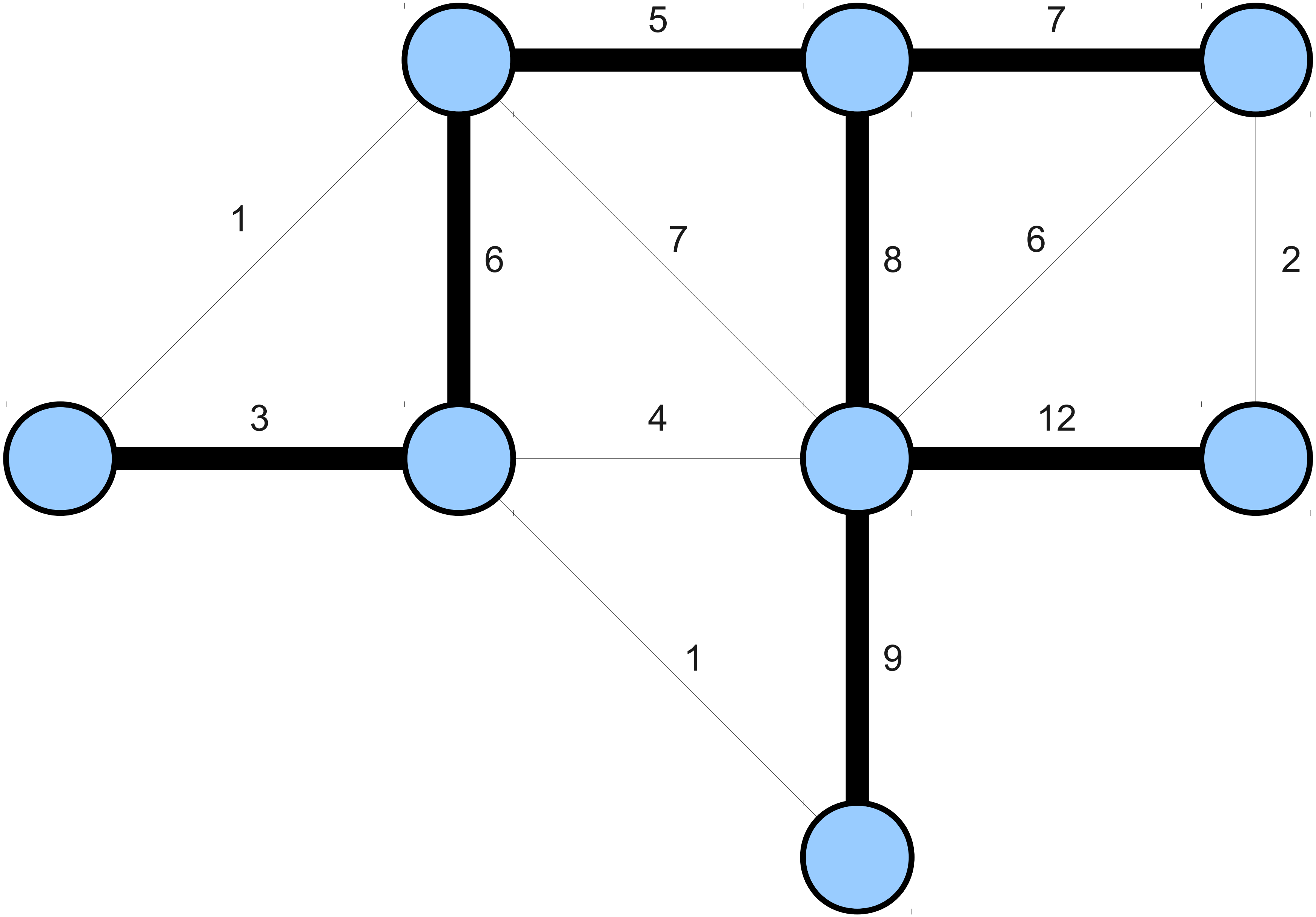
- Maintain a set of “finished nodes.”
- Add the path of just *s* to a priority queue with length 0.
- While the queue is not empty:
  - Dequeue the current path.
  - If the end node has not already been finished:
    - Mark the end node as finished.
    - Add to the priority queue all paths formed by expanding this current path by one step.

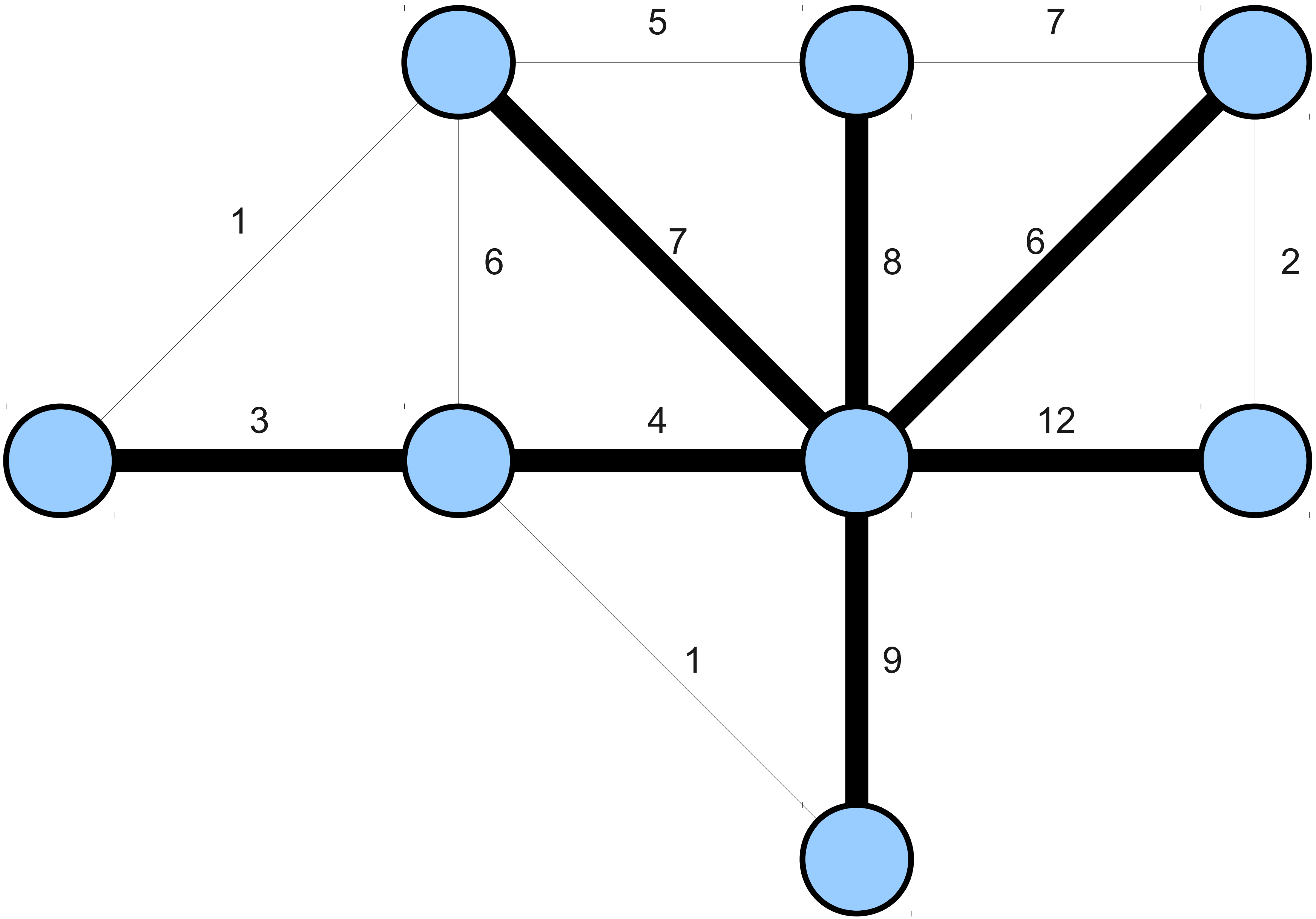


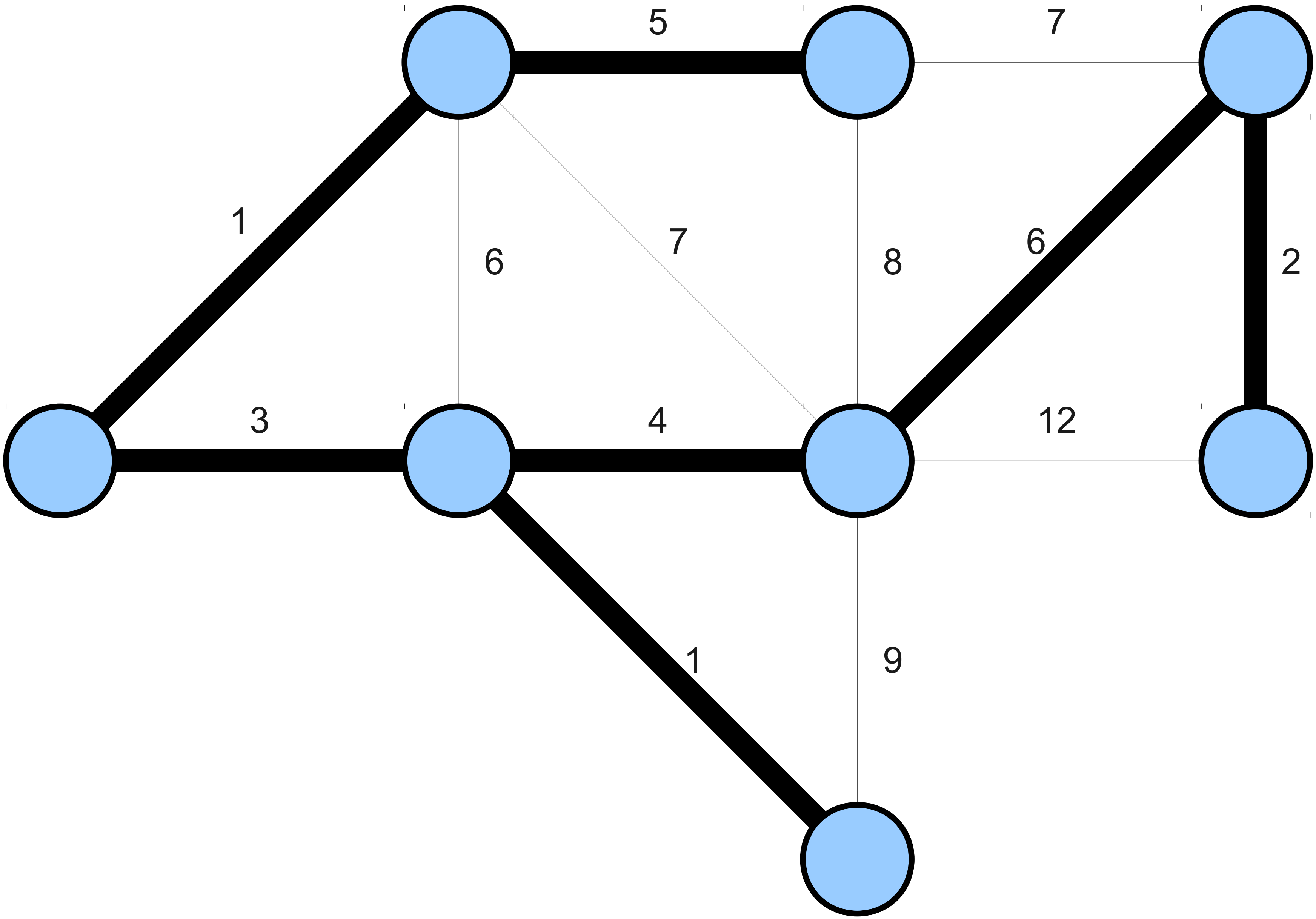
Google maps uses a modified version of Dijkstra's algorithm called A* search.

Minimum Spanning Trees


5
7
1
6
7
8
6
2
3
4
12
1
9


5
7
1
6
7
8
6
2
3
4
12
1
9


5
7
1
6
7
8
6
2
3
4
12
1
9


5
7
1
6
7
8
6
2
3
4
12
1
9

A **spanning tree** in an undirected
graph is a set of edges with
no cycles that connects all nodes.

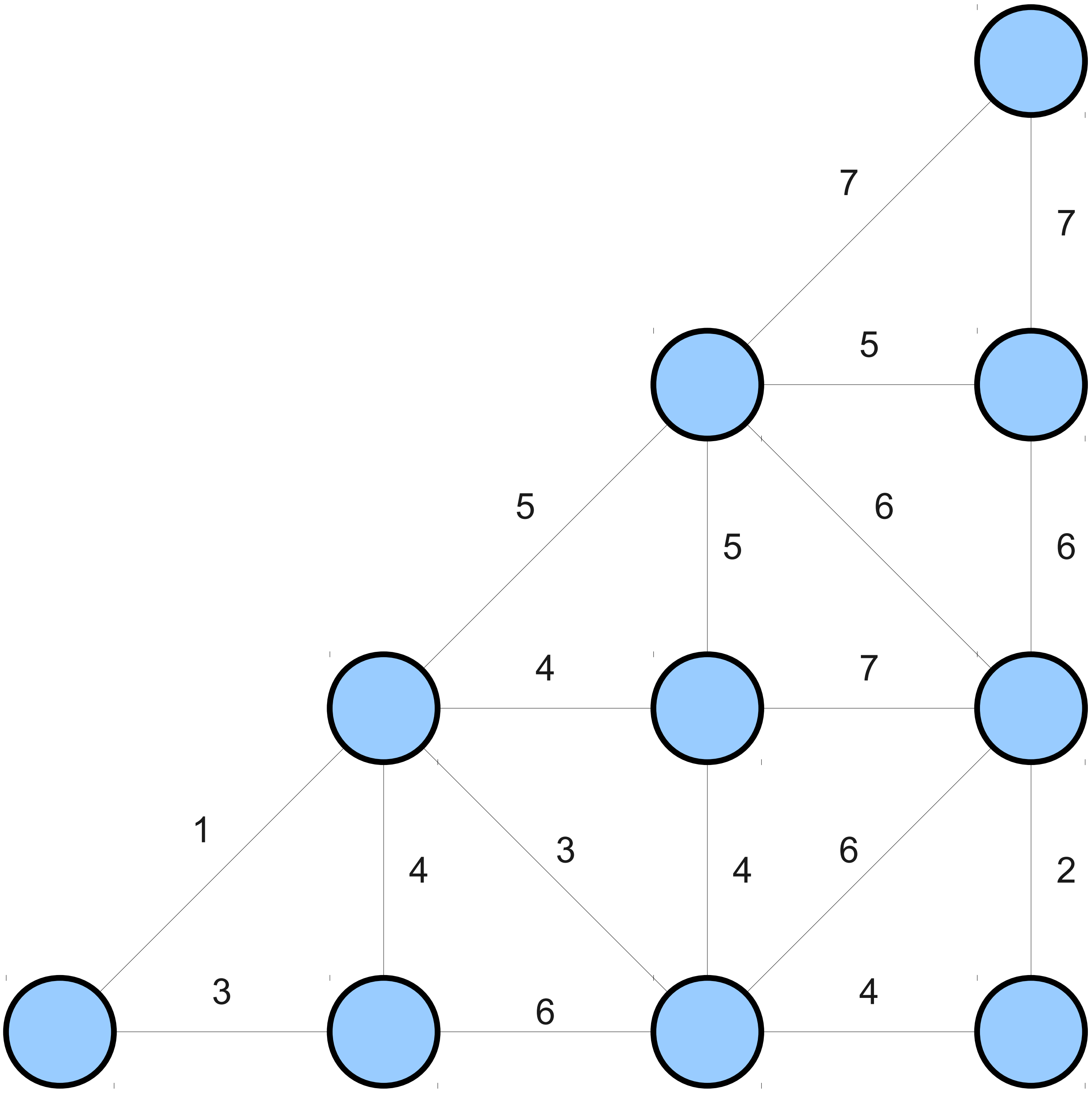
A **minimum spanning tree** (or **MST**) is a spanning tree with the least total cost.

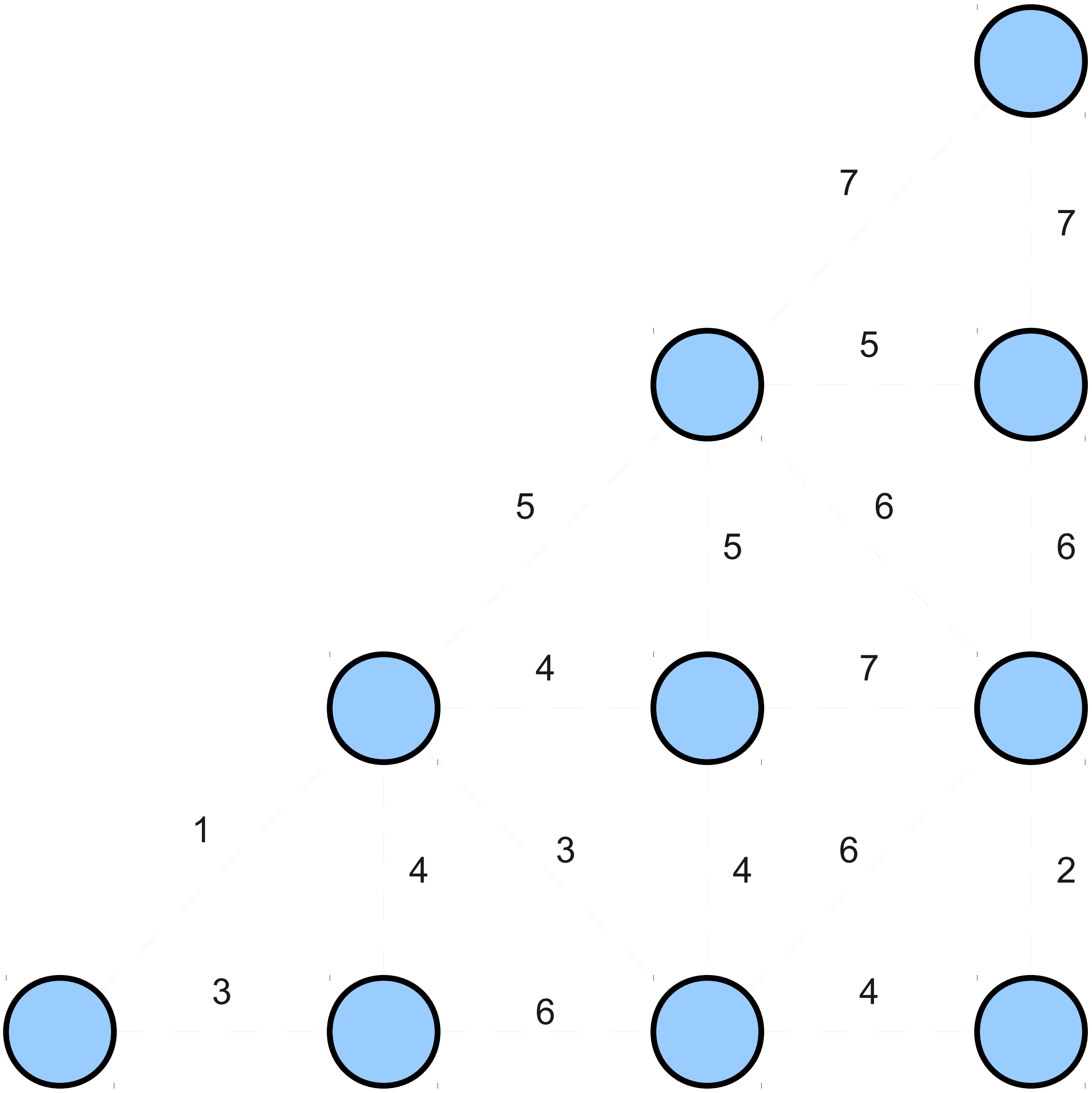
Applications

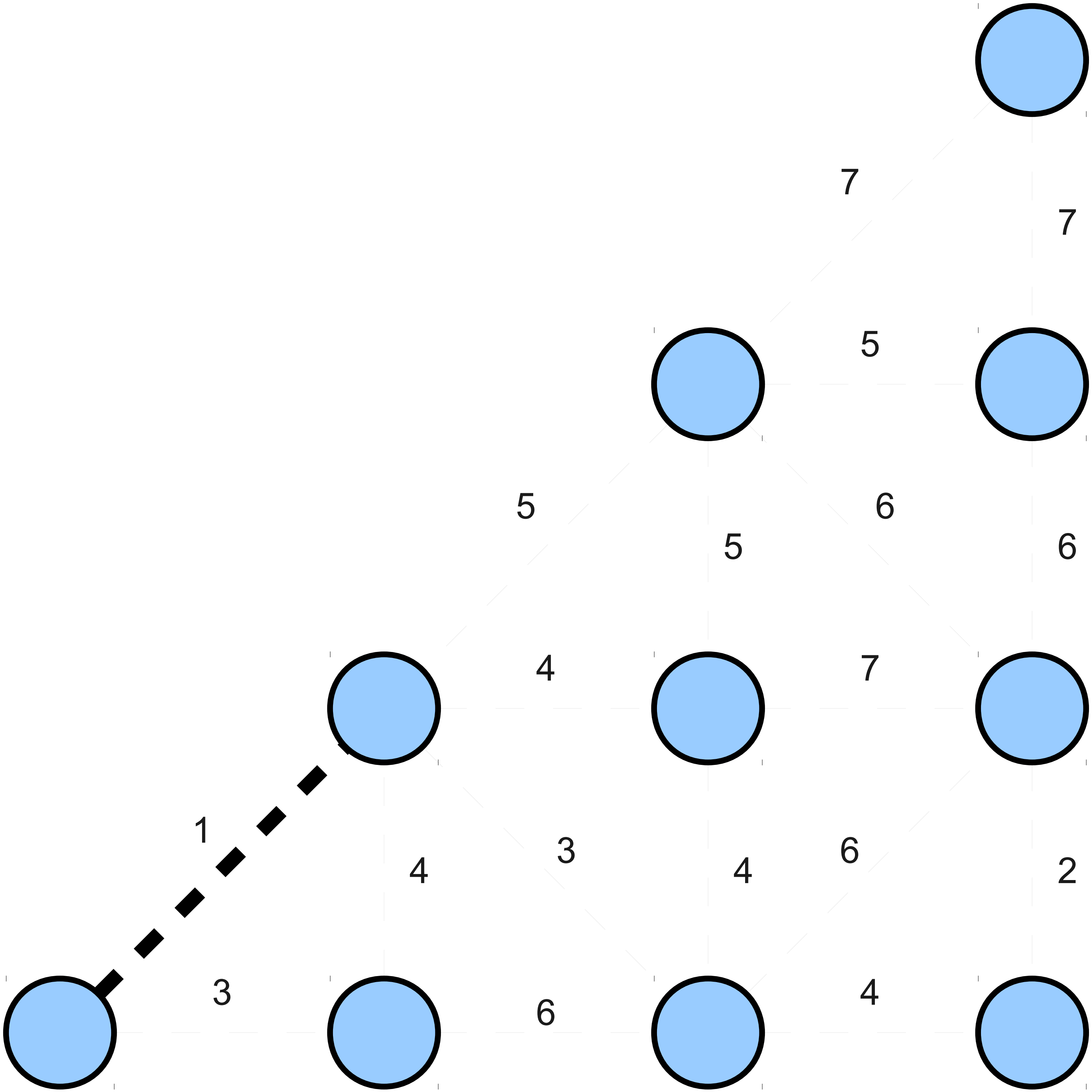
- **Electric Grids**
  - Given a collection of houses, where do you lay wires to connect all houses with the least total cost?
  - This was the initial motivation for studying minimum spanning trees in the early 1920's. (work done by Czech mathematician Otakar Borůvka)
- **Data Clustering**
  - More on that later...

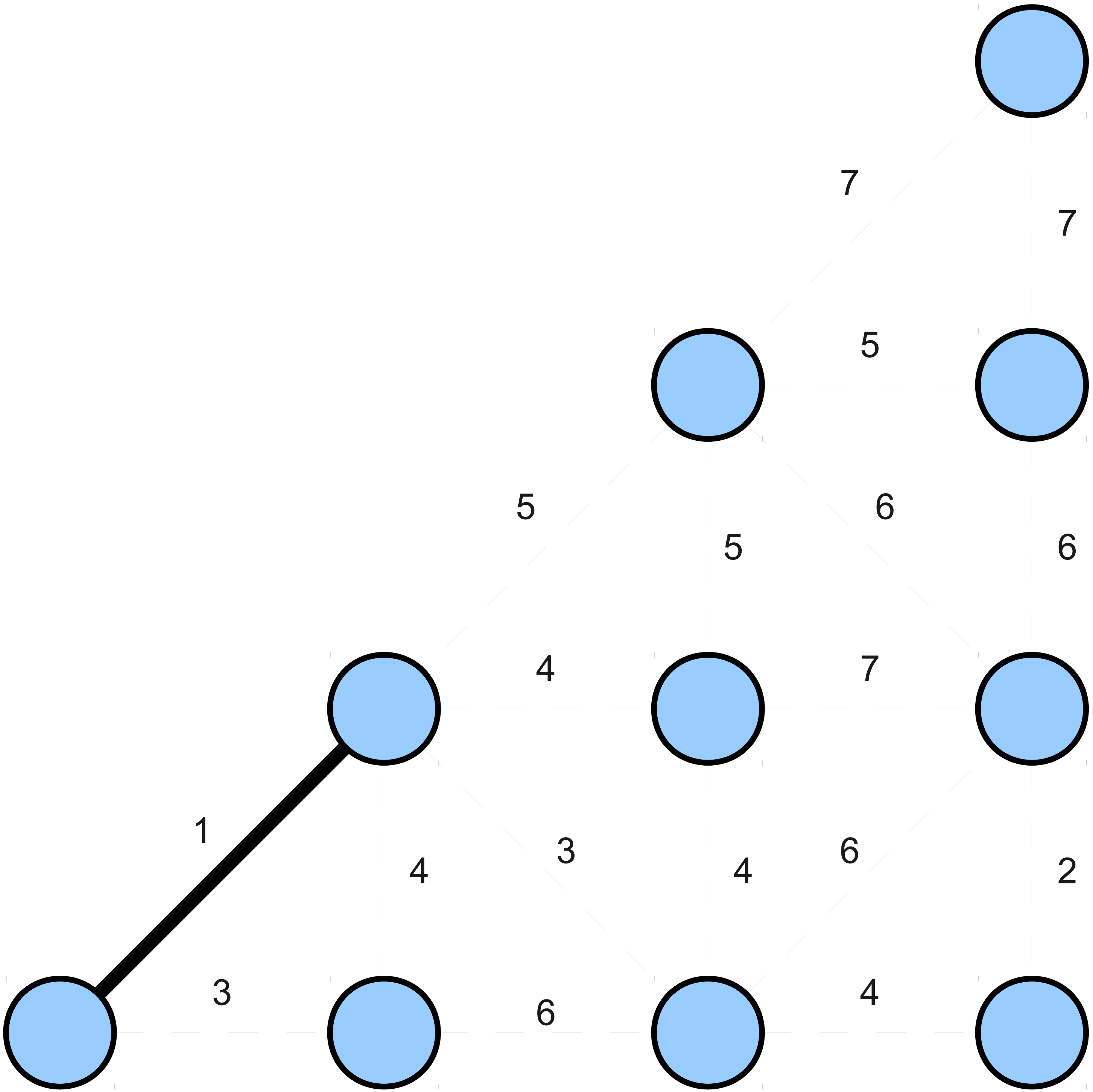
Kruskal's Algorithm

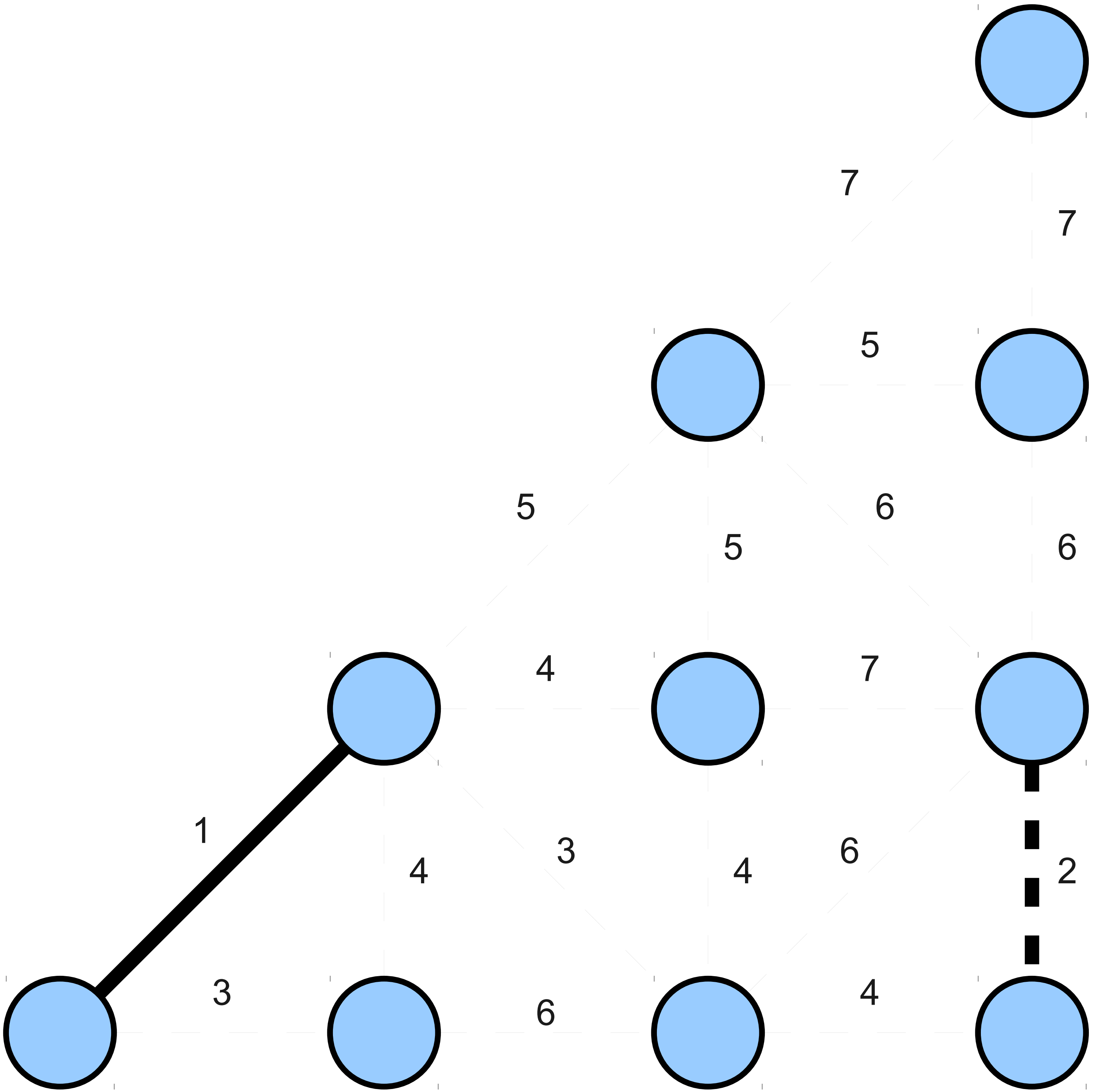
- **Kruskal's algorithm** is an efficient algorithm for finding minimum spanning trees.
- Idea is as follows:
  - Remove all edges from the graph.
  - Sort the edges into ascending order by length.
  - For each edge:
    - If the endpoints of the edge aren't already connected to one another, add in that edge.
    - Otherwise, skip the edge.

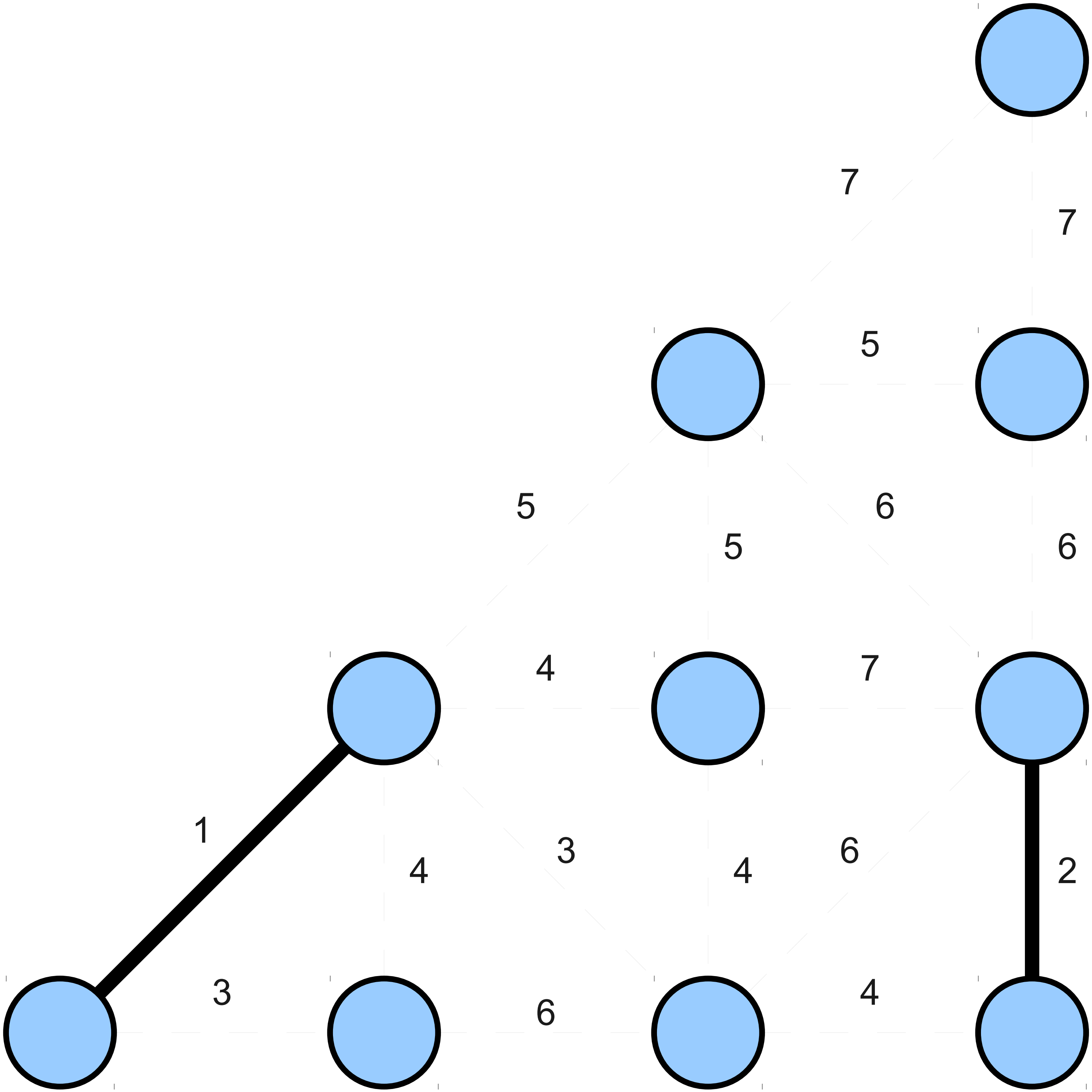

7
7
5
5
6
5
6
4
7
1
3
4
4
6
2
3
6
4

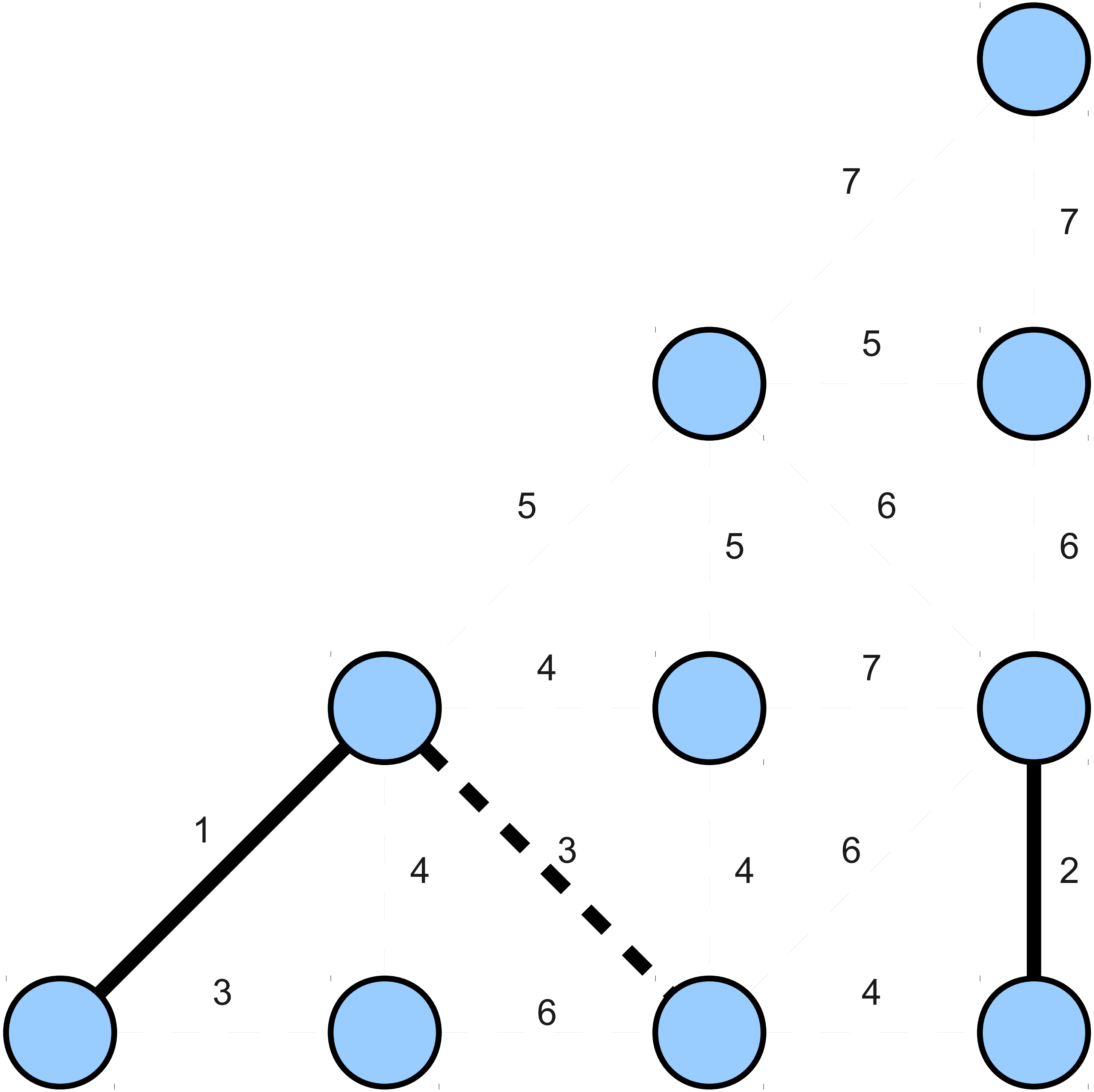

7
7
5
5
5
6
6
4
7
1
4
3
4
6
2
3
6
4

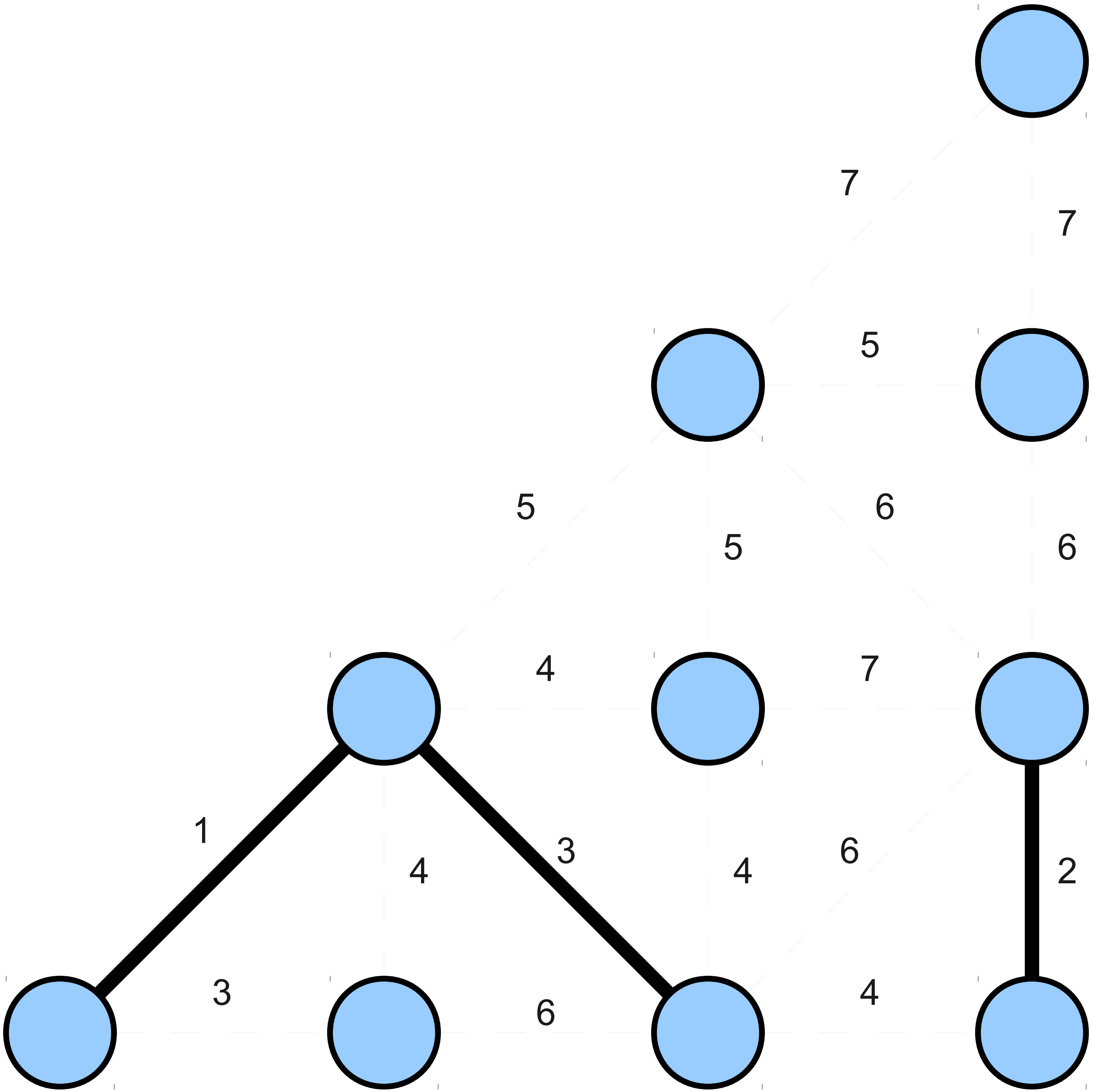

7
7
5
5
5
6
6
4
7
1
4
3
4
6
2
3
6
4

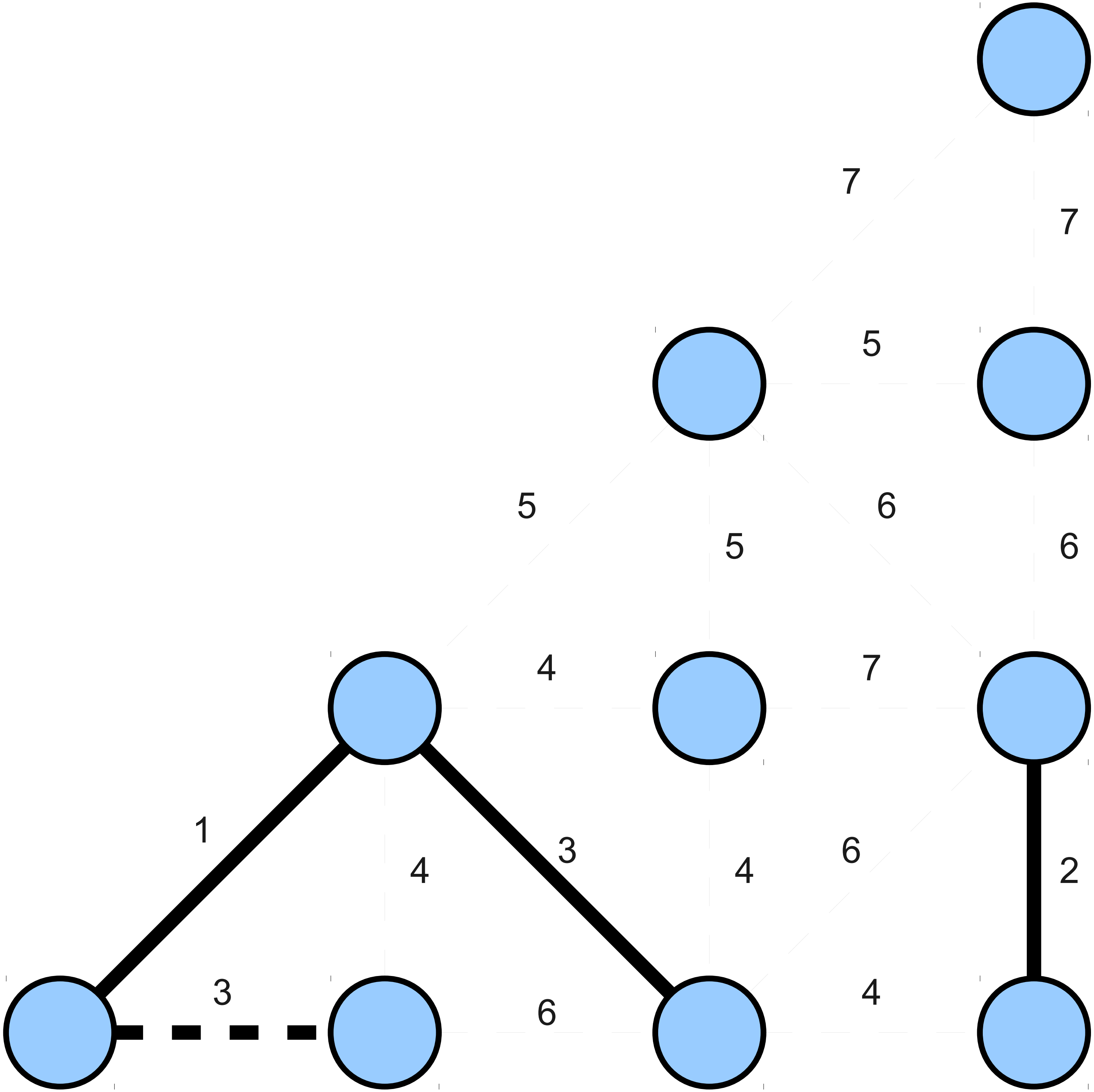

7
7
5
5
5
6
6
4
7
1
4
3
4
6
2
3
6
4

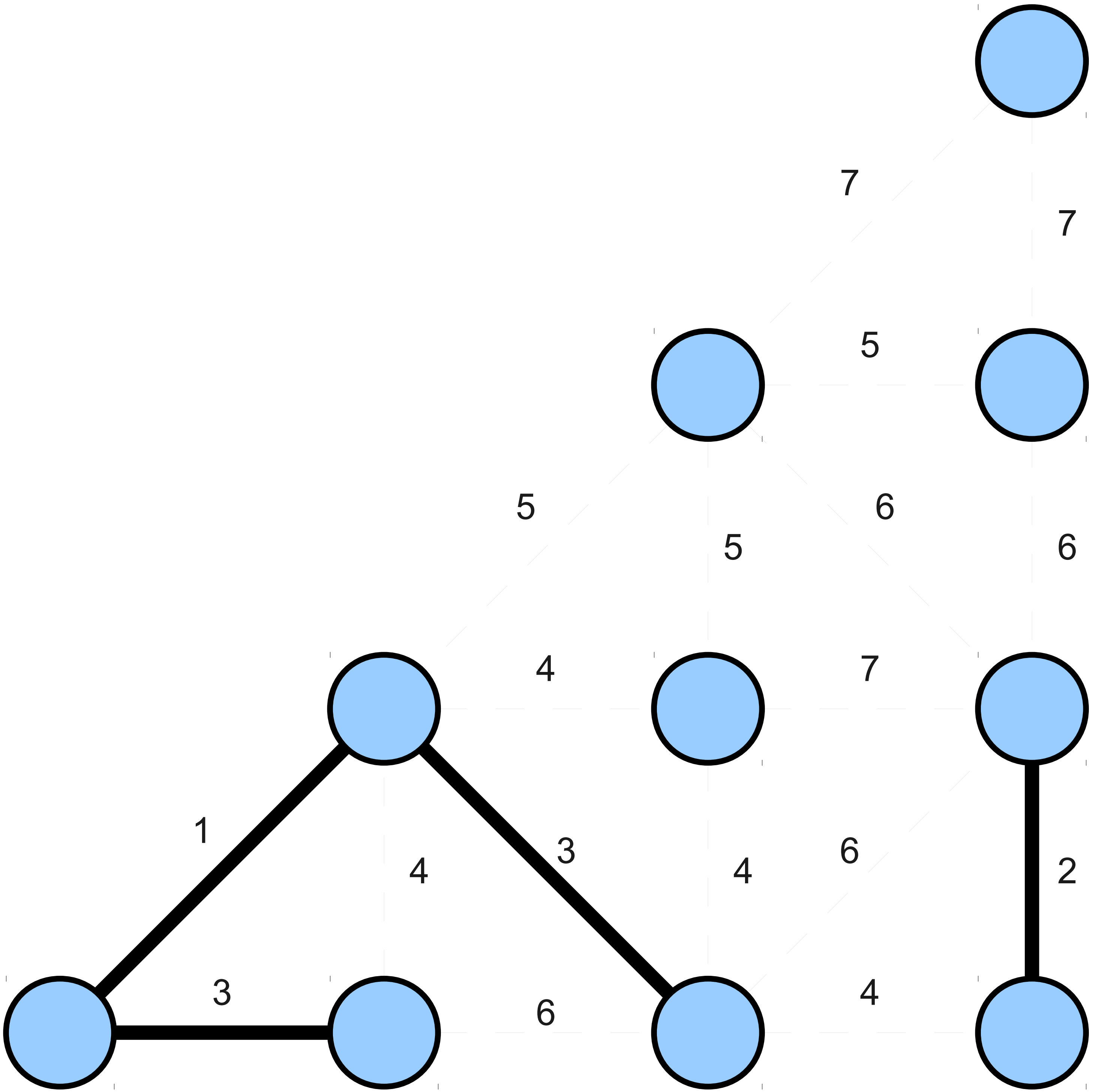

7
7
5
5
5
6
6
4
7
1
4
3
4
6
2
3
6
4

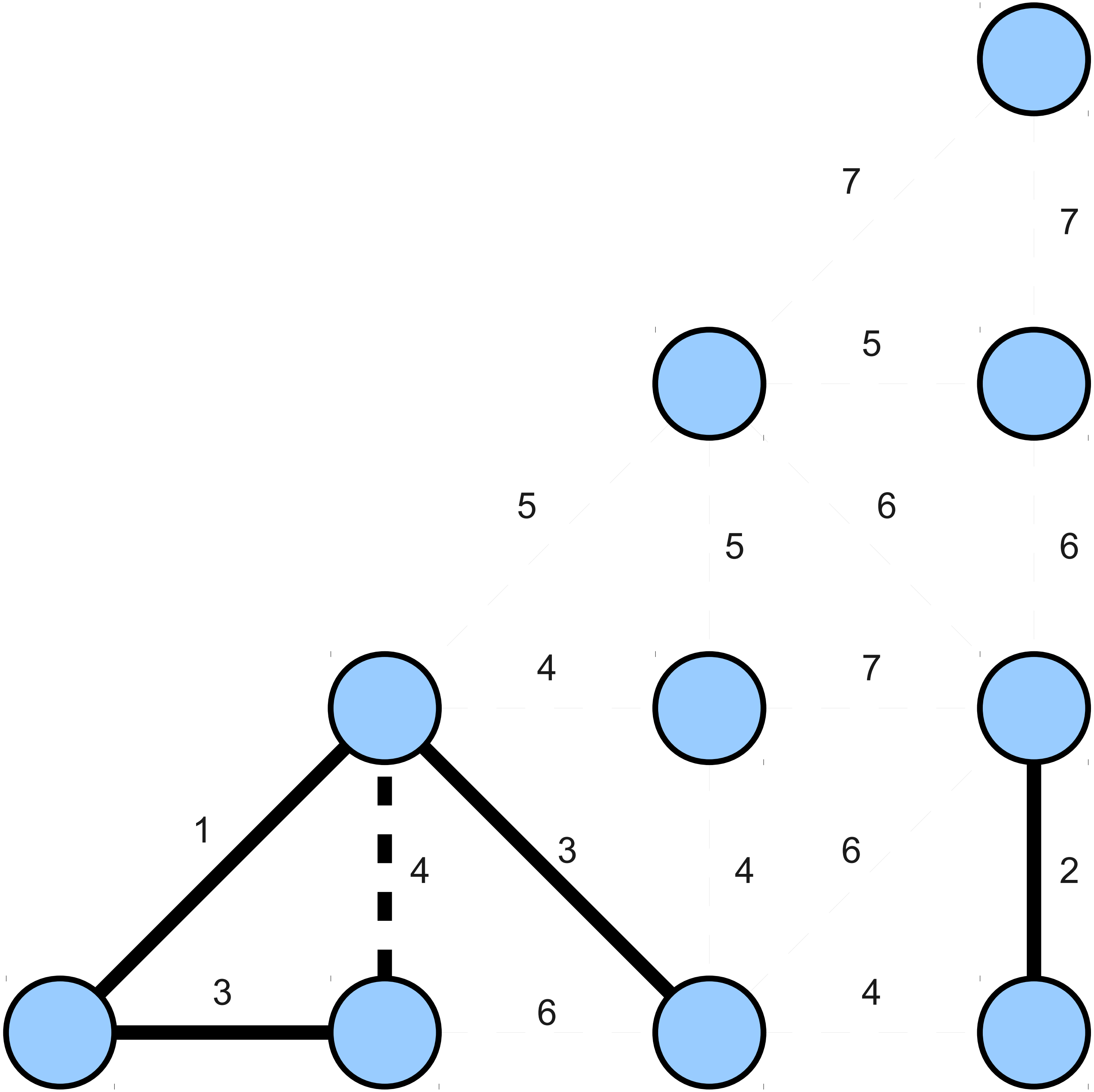

7
7
5
5
5
6
6
4
7
1
4
3
4
6
2
3
6
4

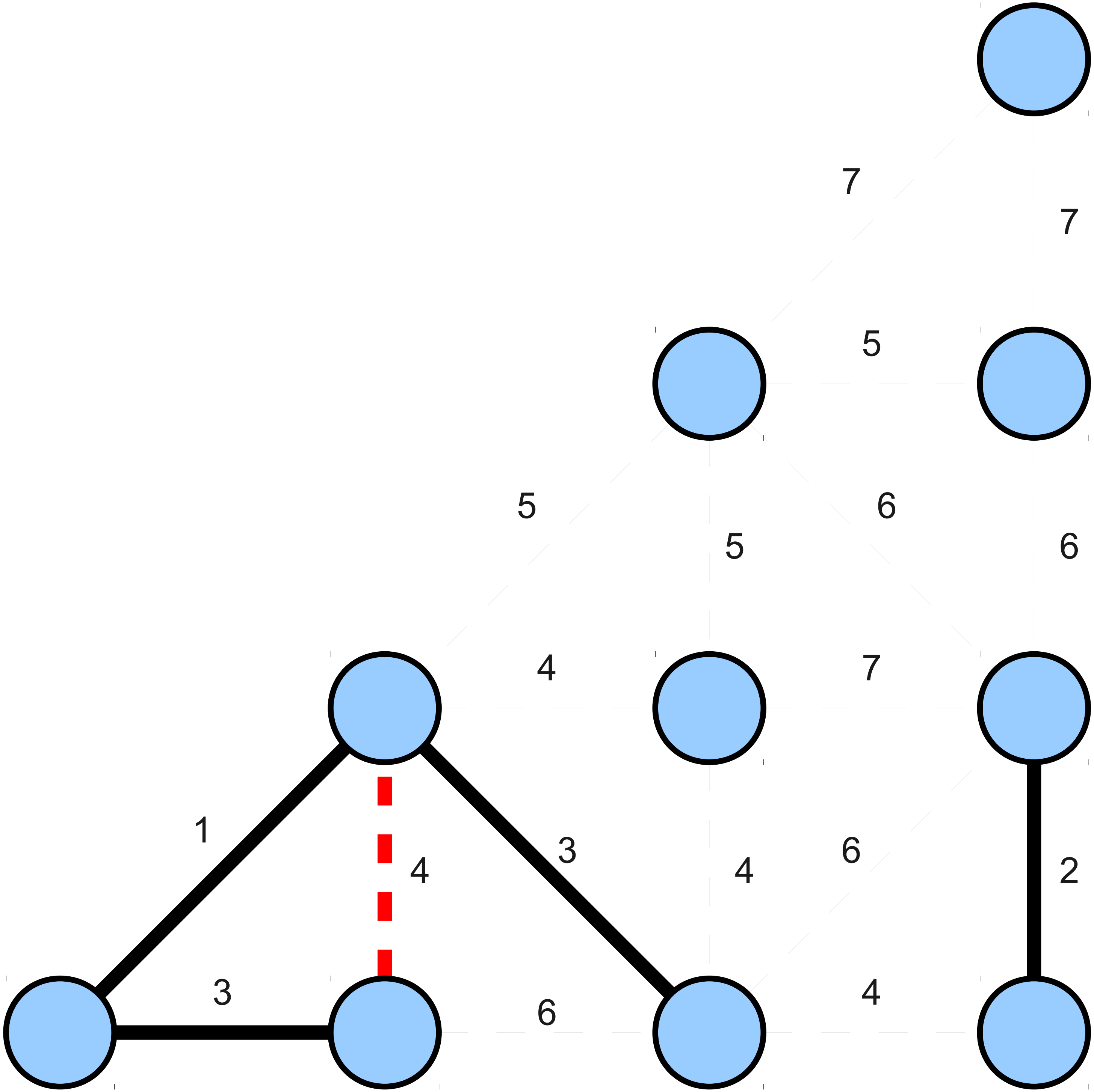

7
7
5
5
5
6
6
4
7
1
4
3
4
6
2
3
6
4

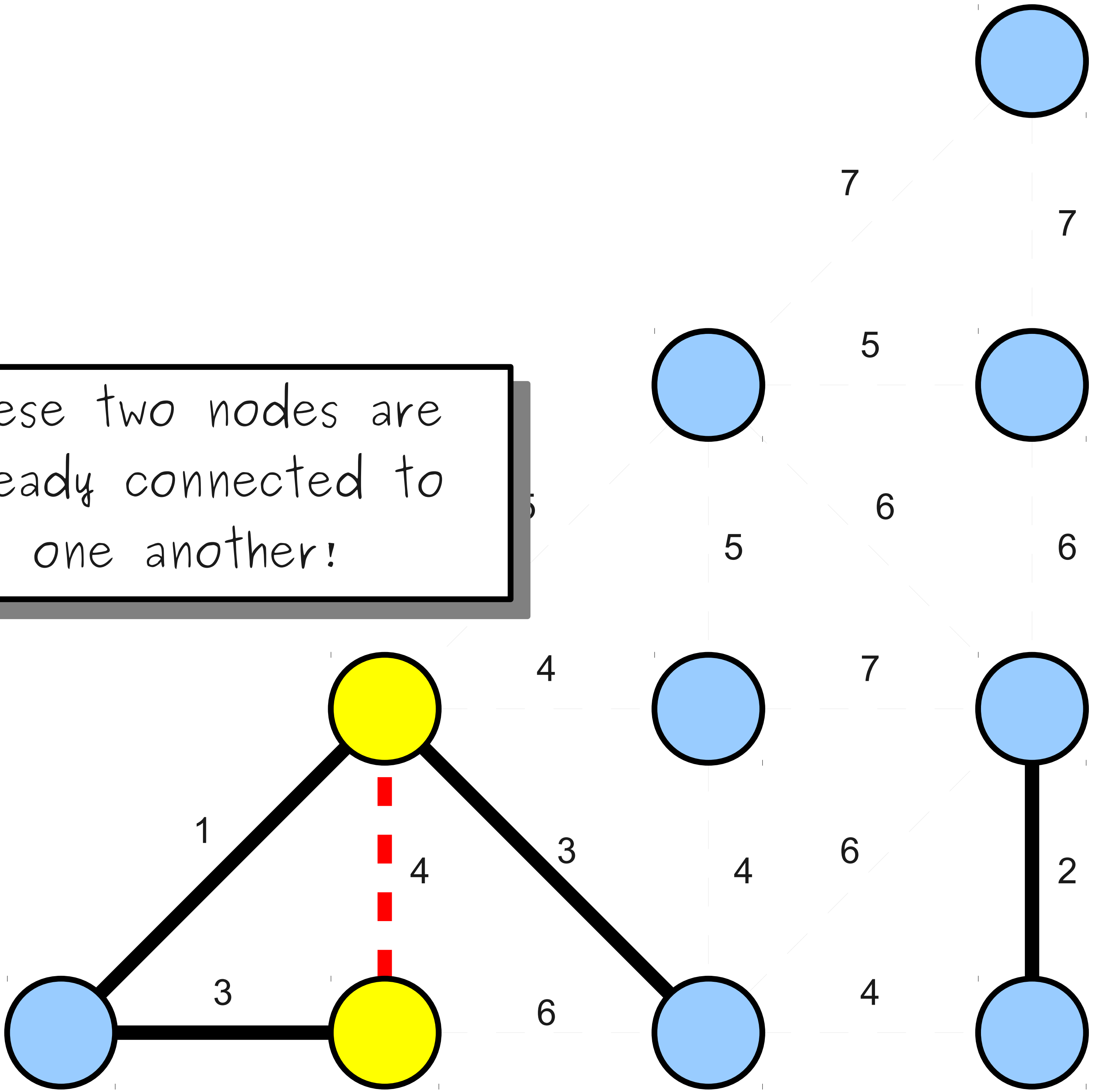

7
7
5
5
5
6
6
4
7
1
3
4
4
6
2
3
6
4


7
7
5
5
5
6
6
4
7
1
3
4
4
6
2
3
6
4

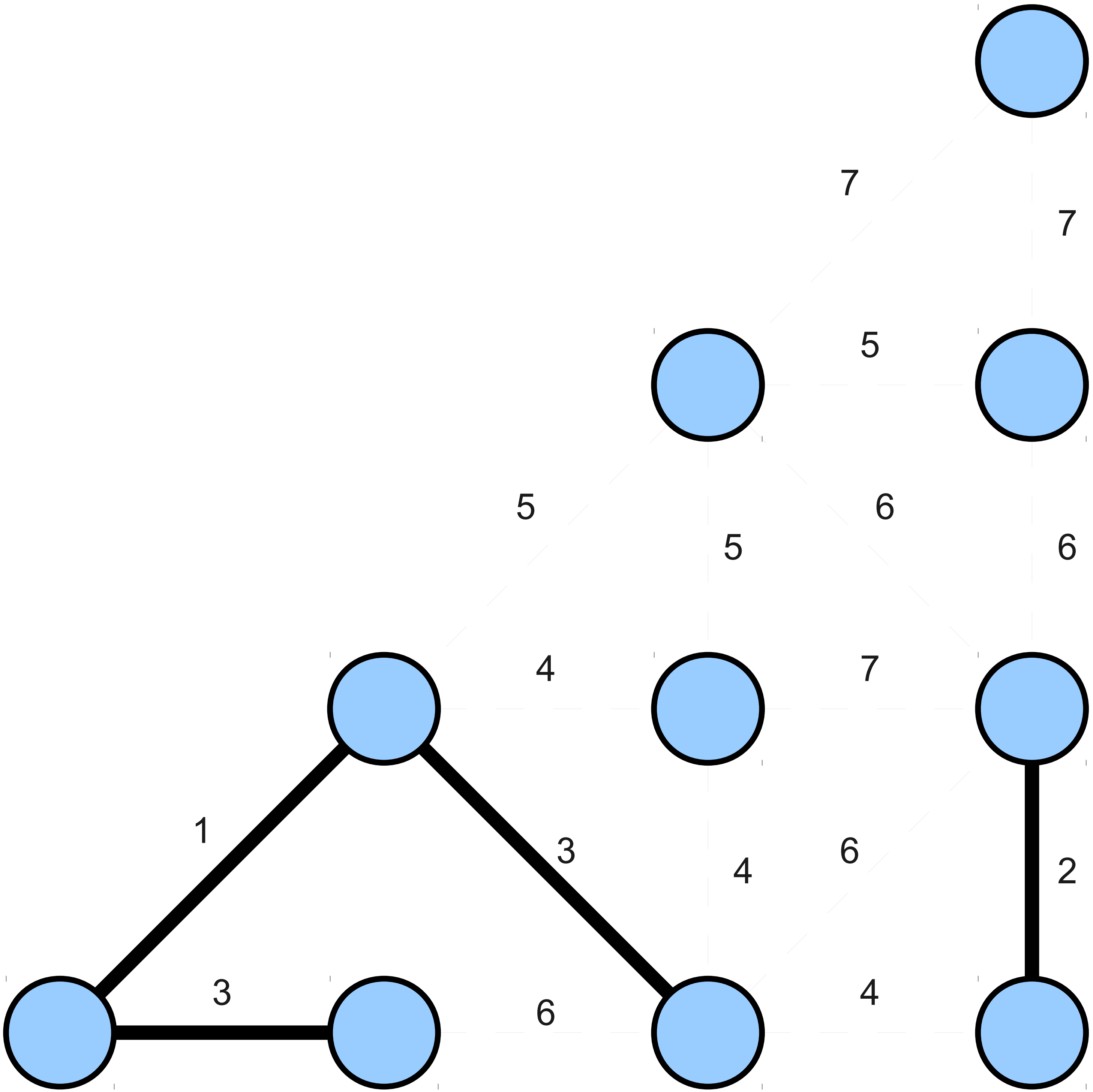

7
7
5
5
5
6
6
4
7
1
4
3
4
6
2
3
6
4

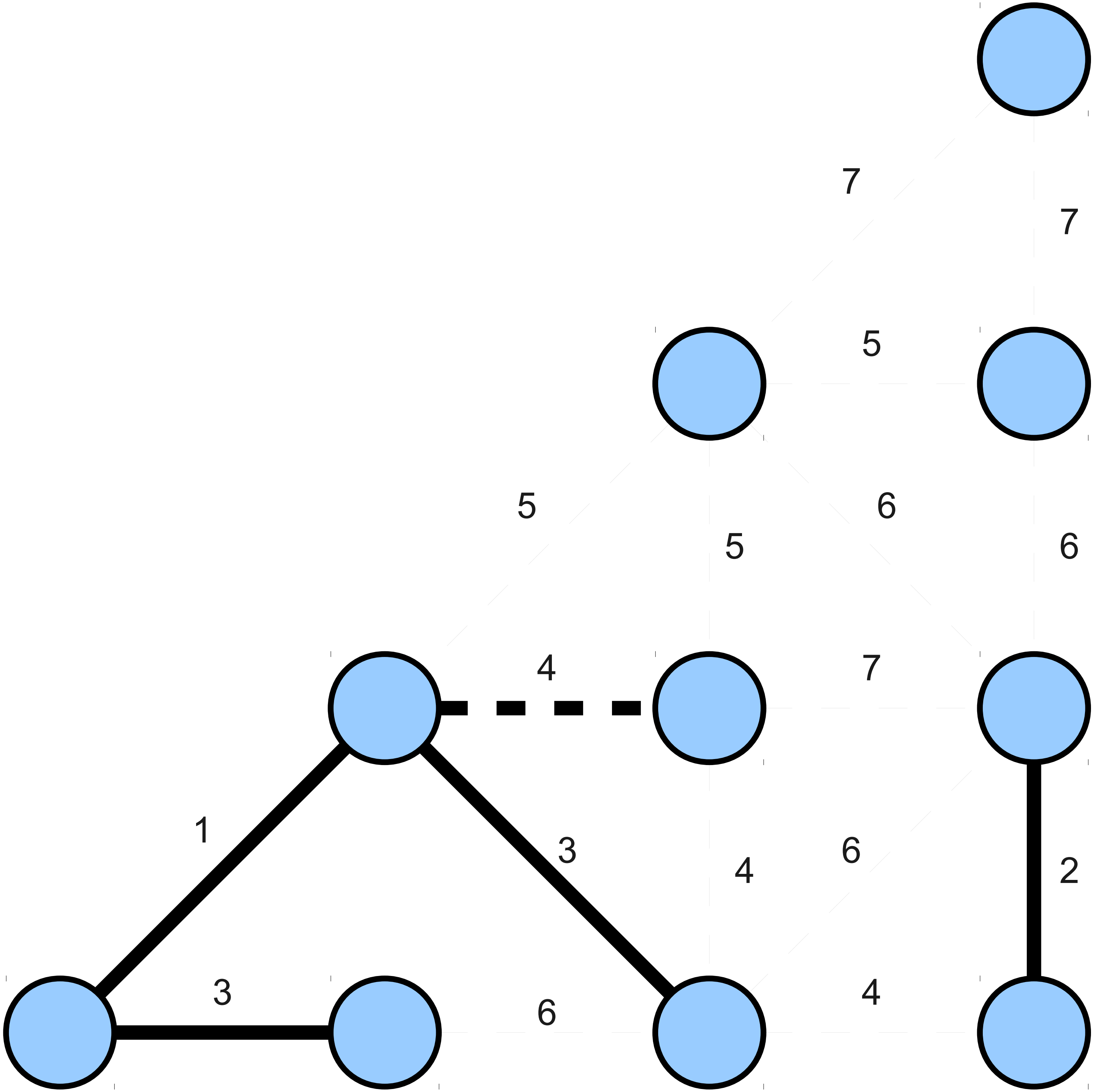

7
7
5
5
5
6
6
4
7
1
4
3
4
6
2
3
6
4

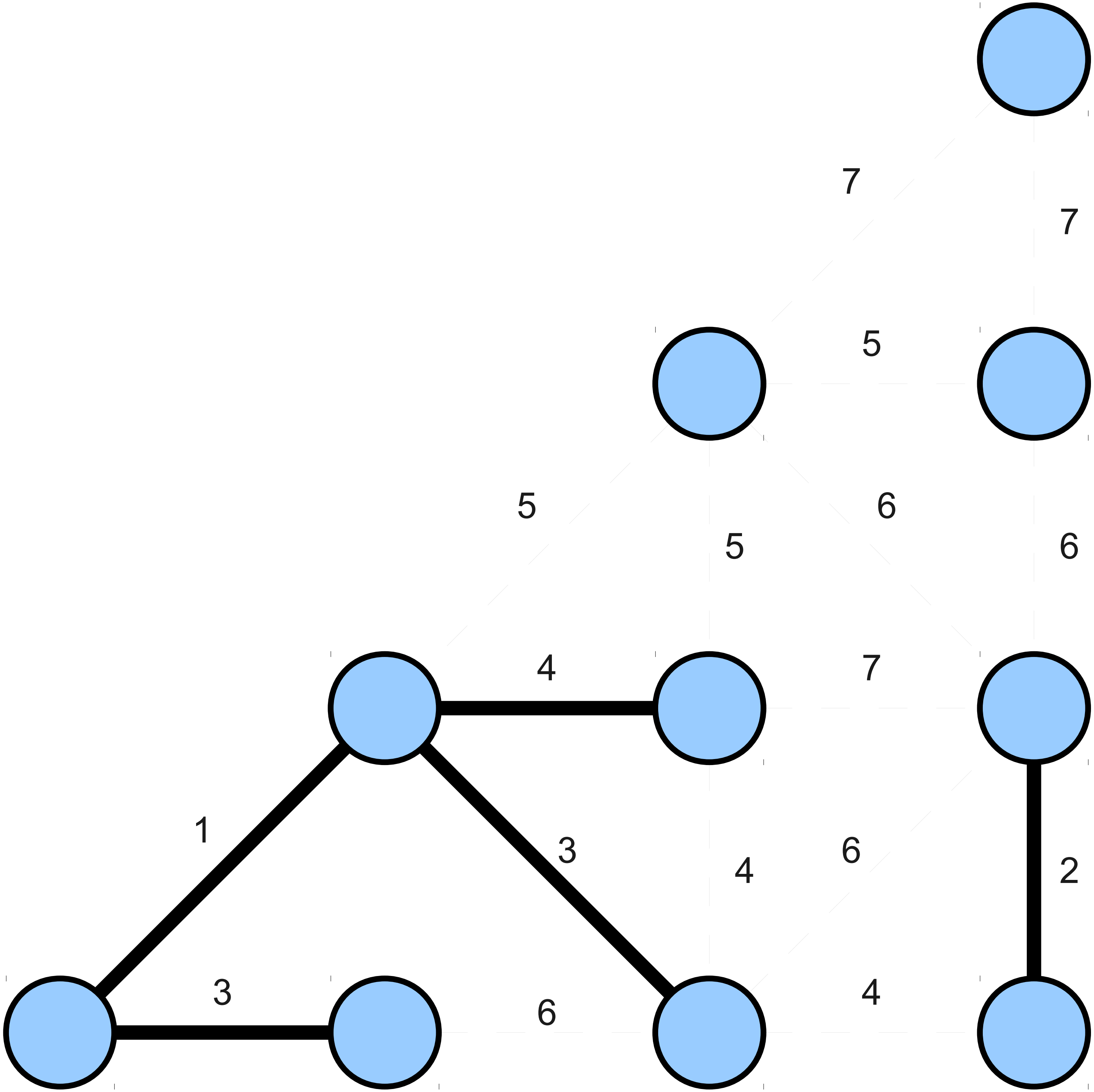

7
7
5
5
5
6
6
4
7
1
4
3
4
6
2
3
6
4

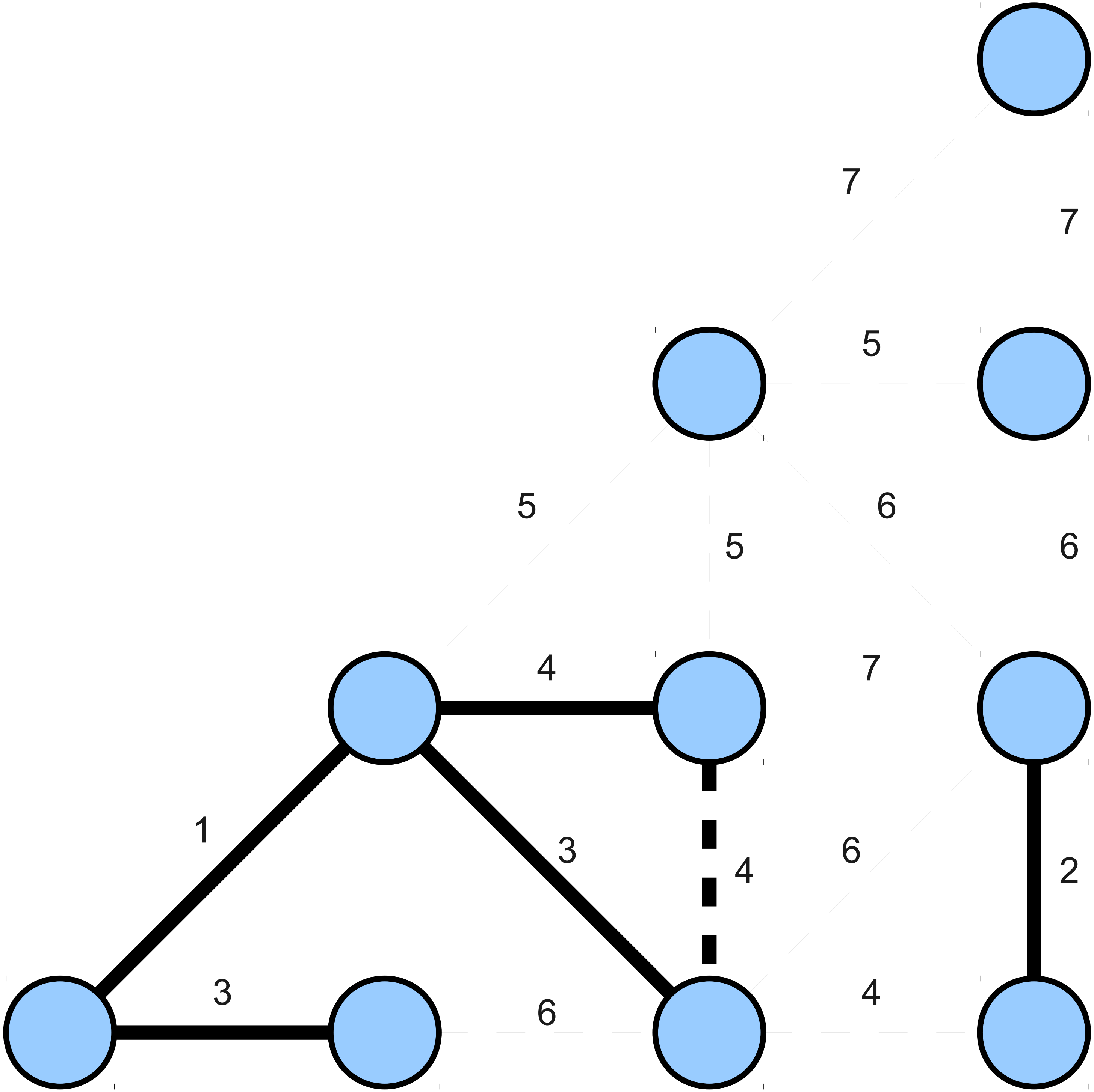


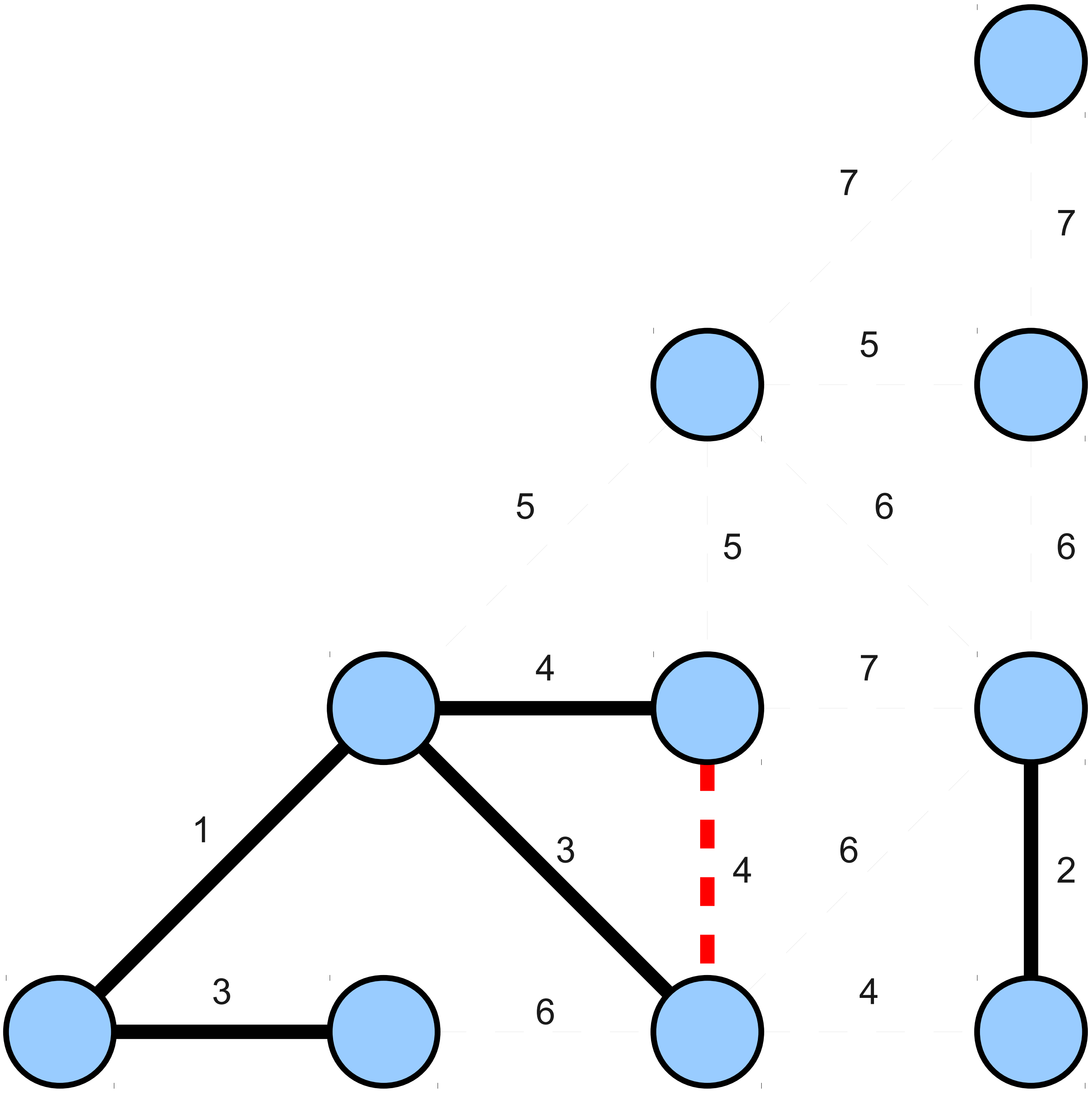
These two nodes are already connected to one another!

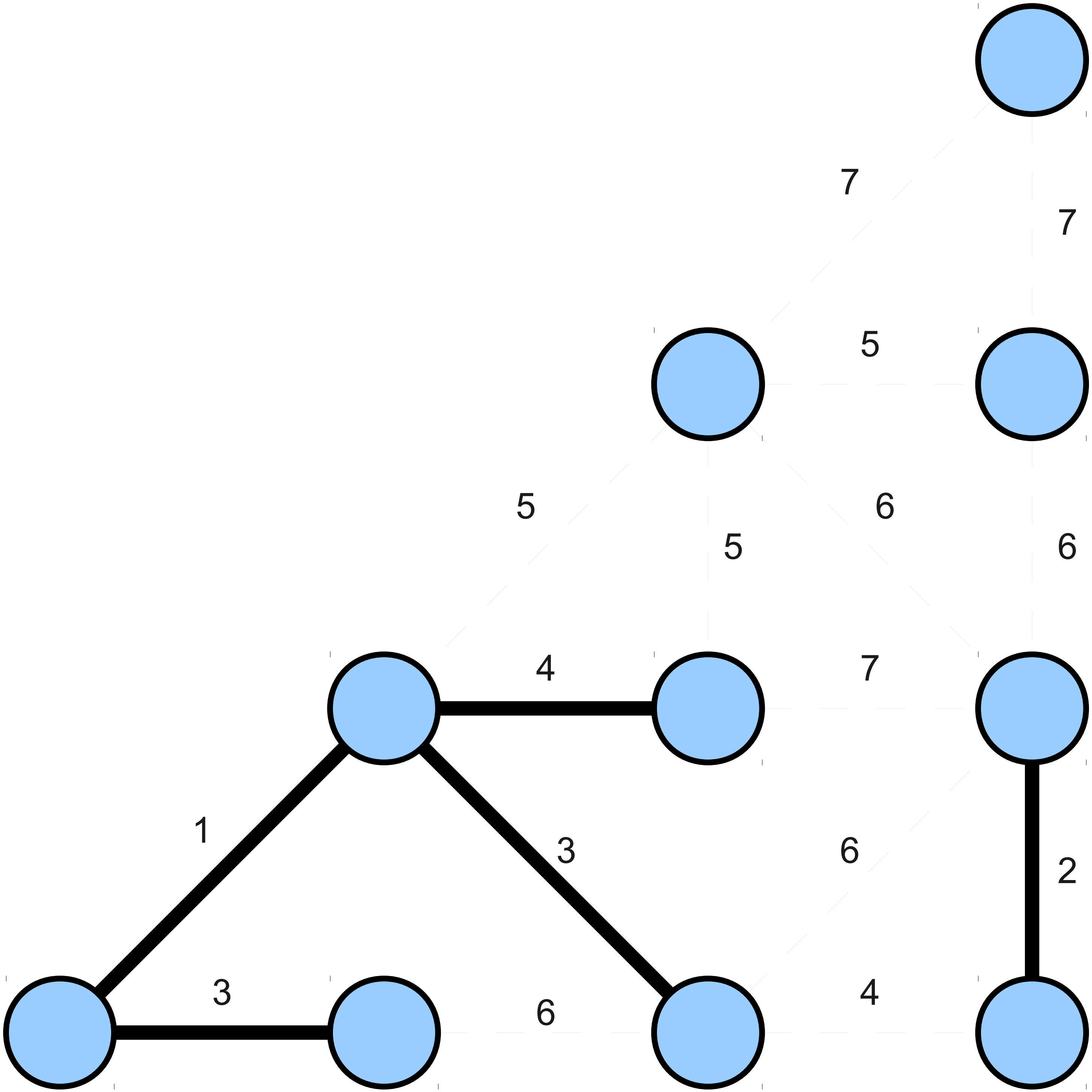

7
7
5
5
5
6
6
4
7
1
3
6
4
2
3
6
4

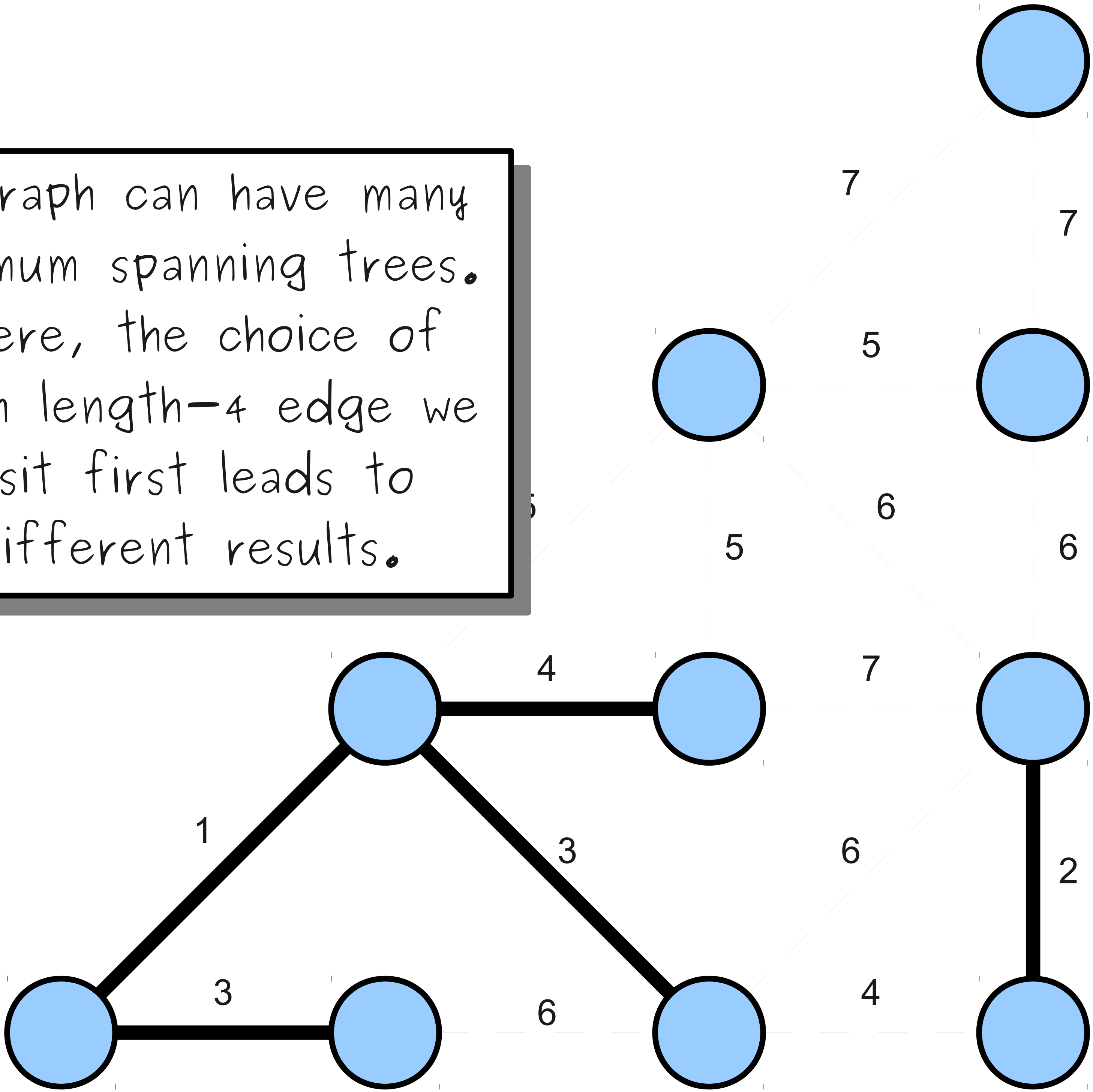

7
7
5
5
5
6
6
4
7
1
3
4
6
2
3
6
4


7
7
5
5
5
6
6
4
7
1
3
6
4
2
3
6
4

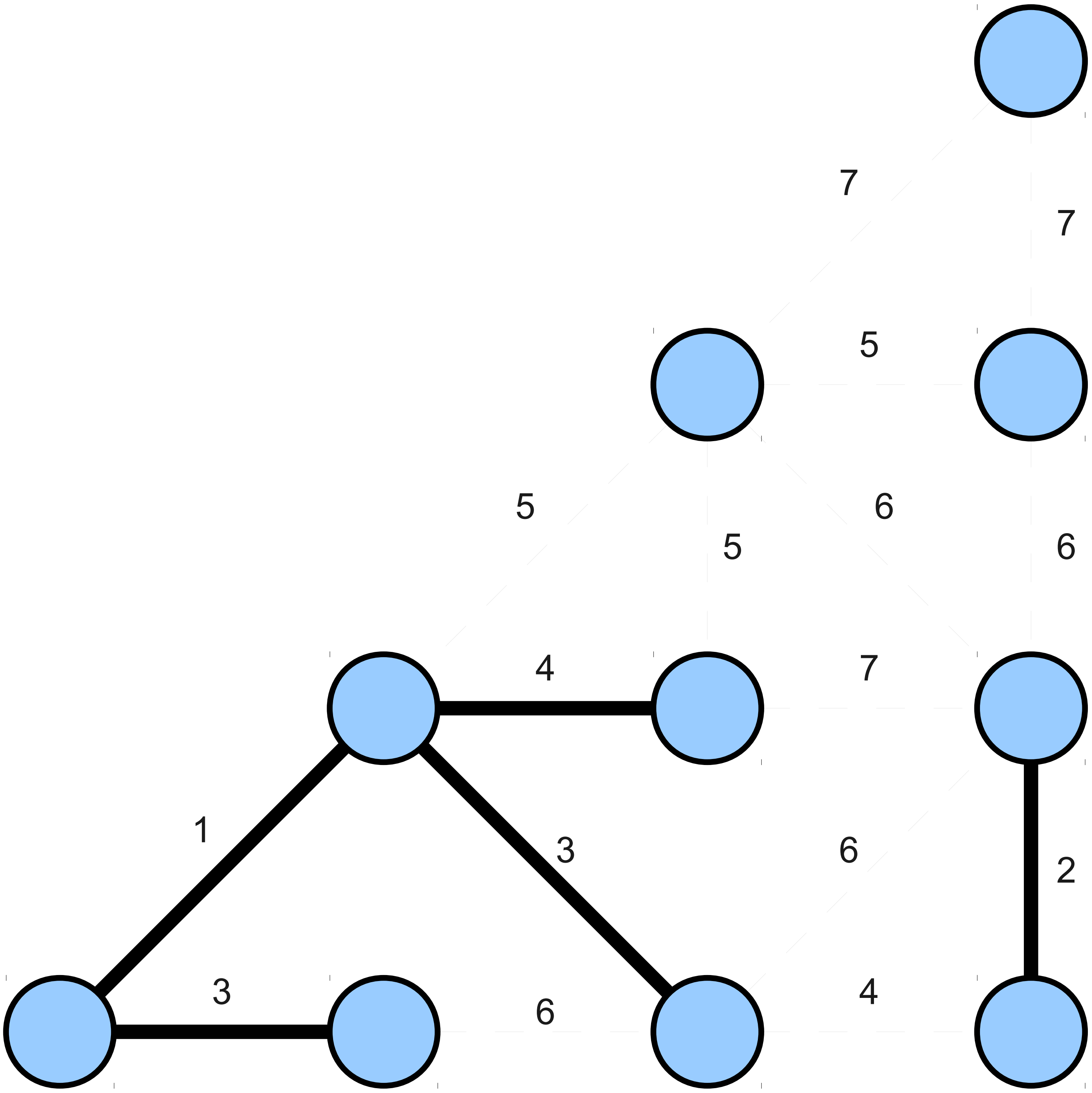

7
7
5
5
5
6
6
4
7
1
3
4
6
2
3
6
4

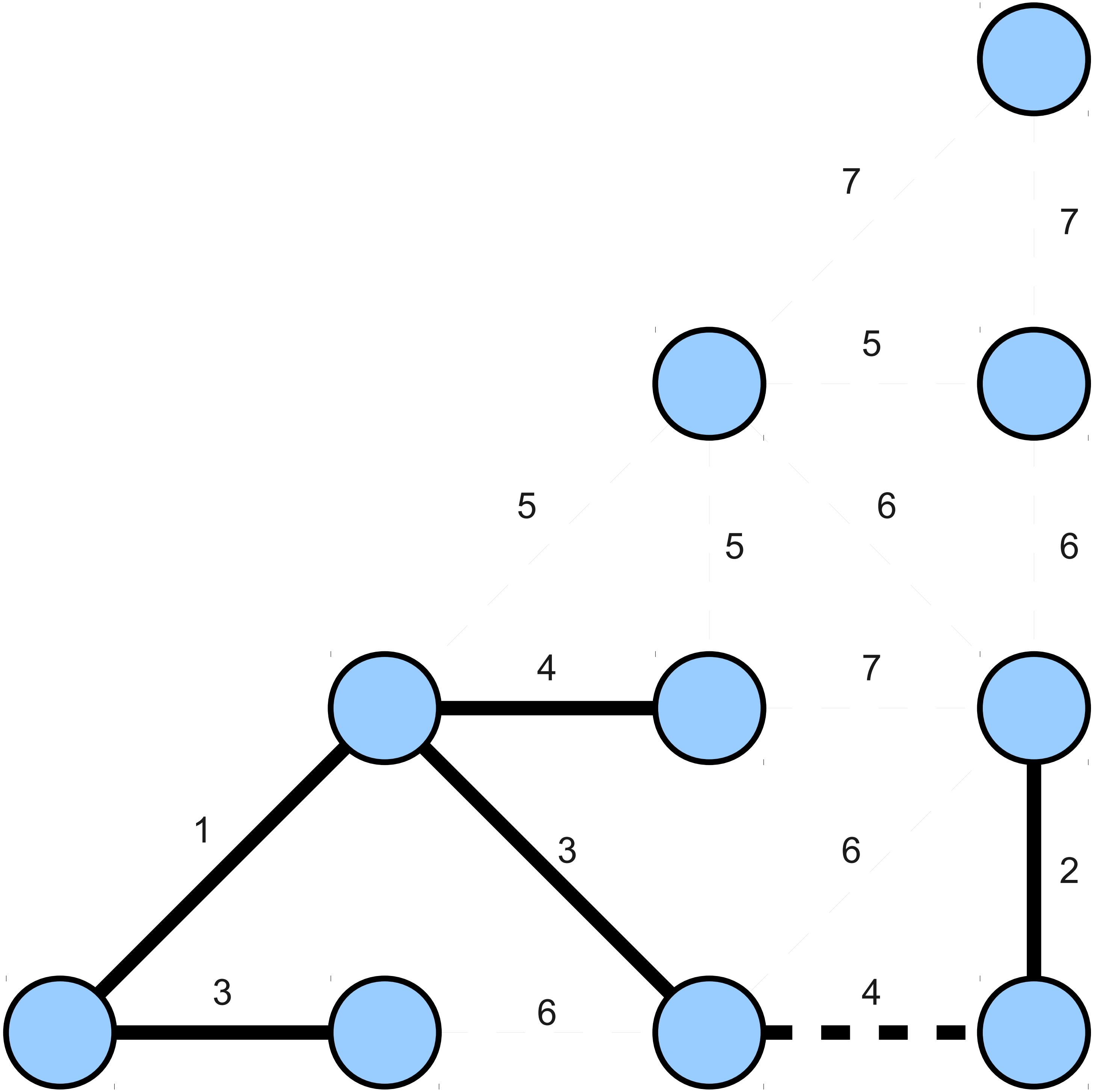

7
7
5
5
5
6
6
4
7
1
3
4
6
2
3
6
4

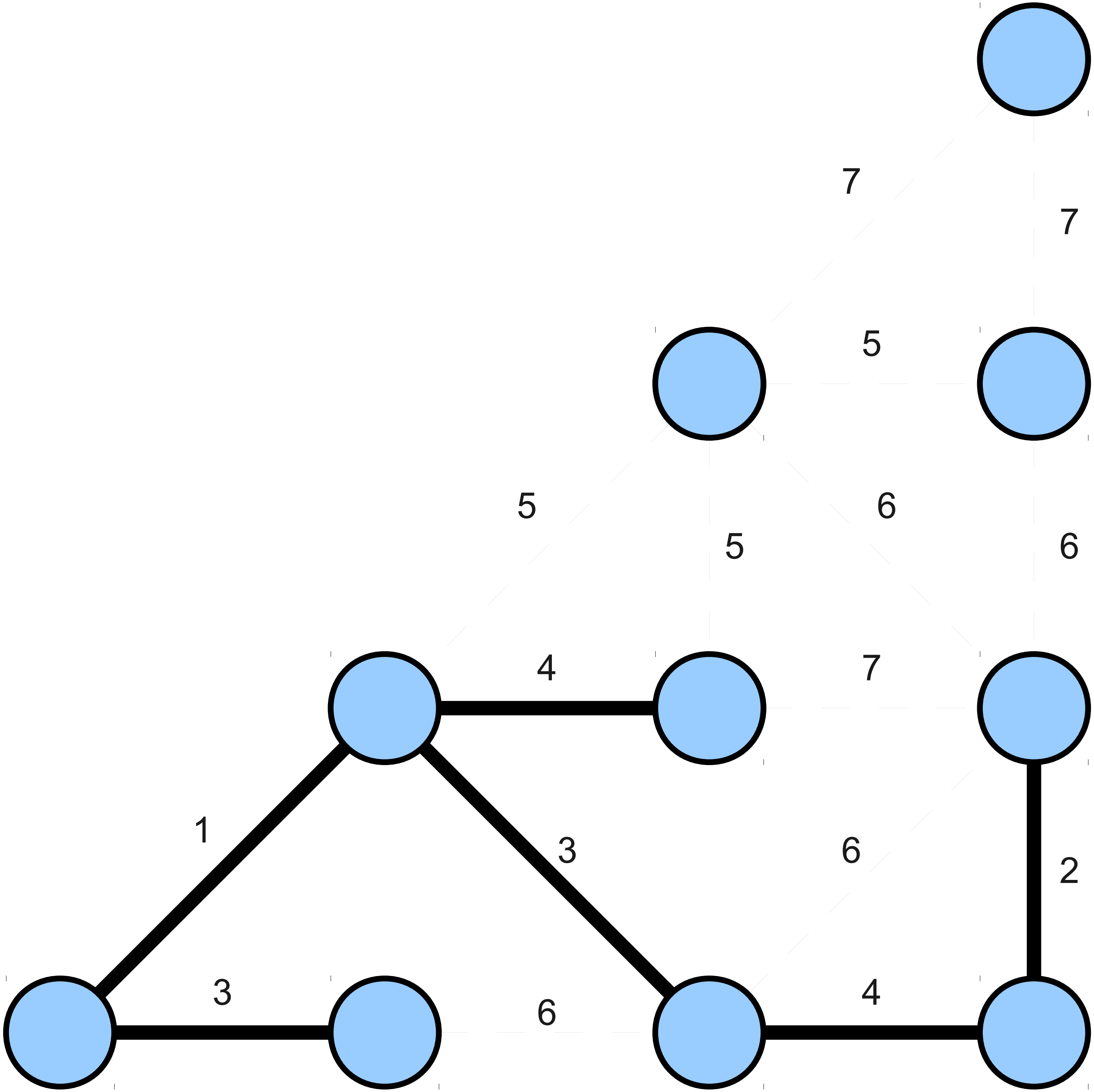

7
7
5
5
5
6
6
4
7
1
3
6
2
3
6
4

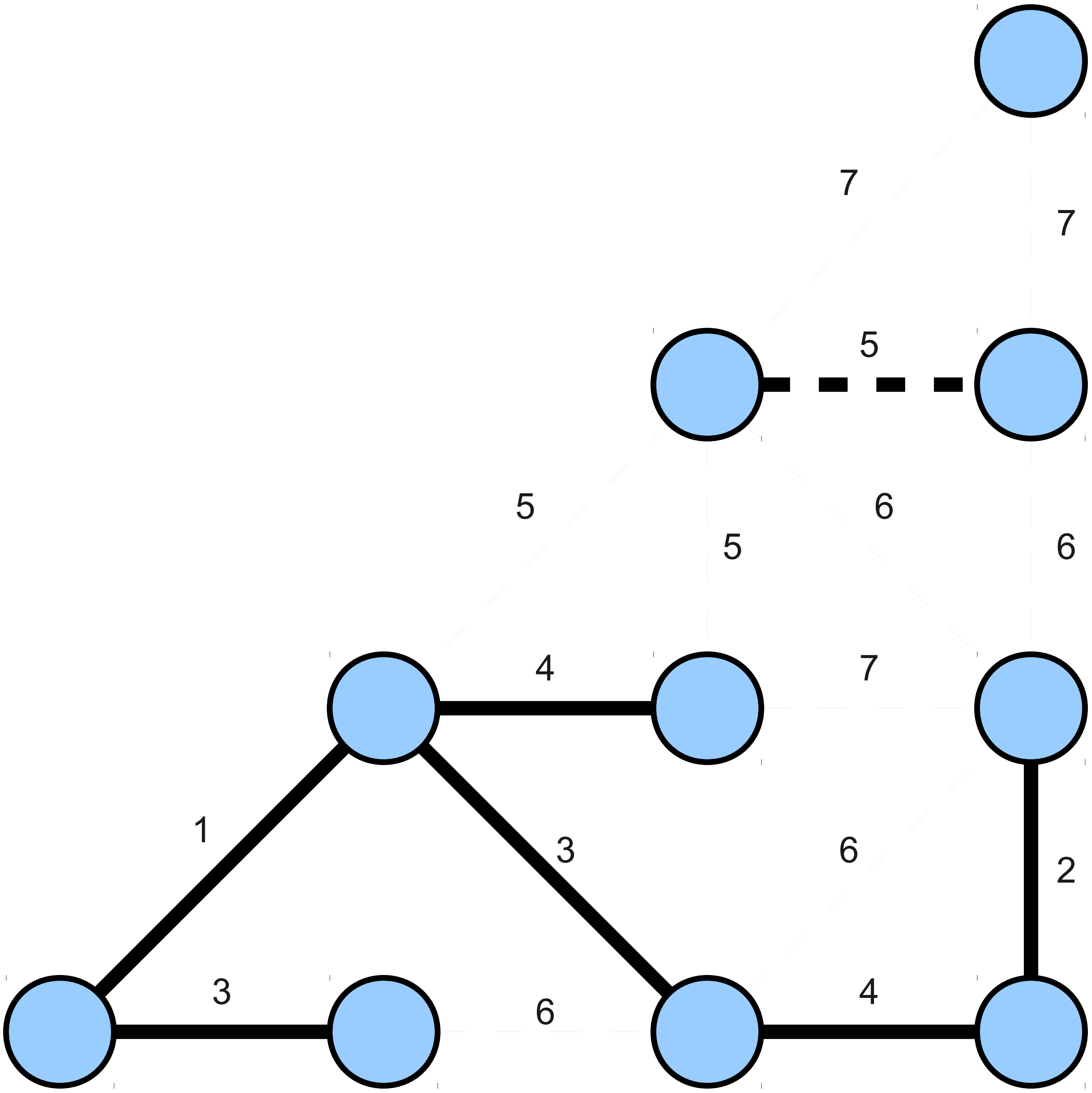


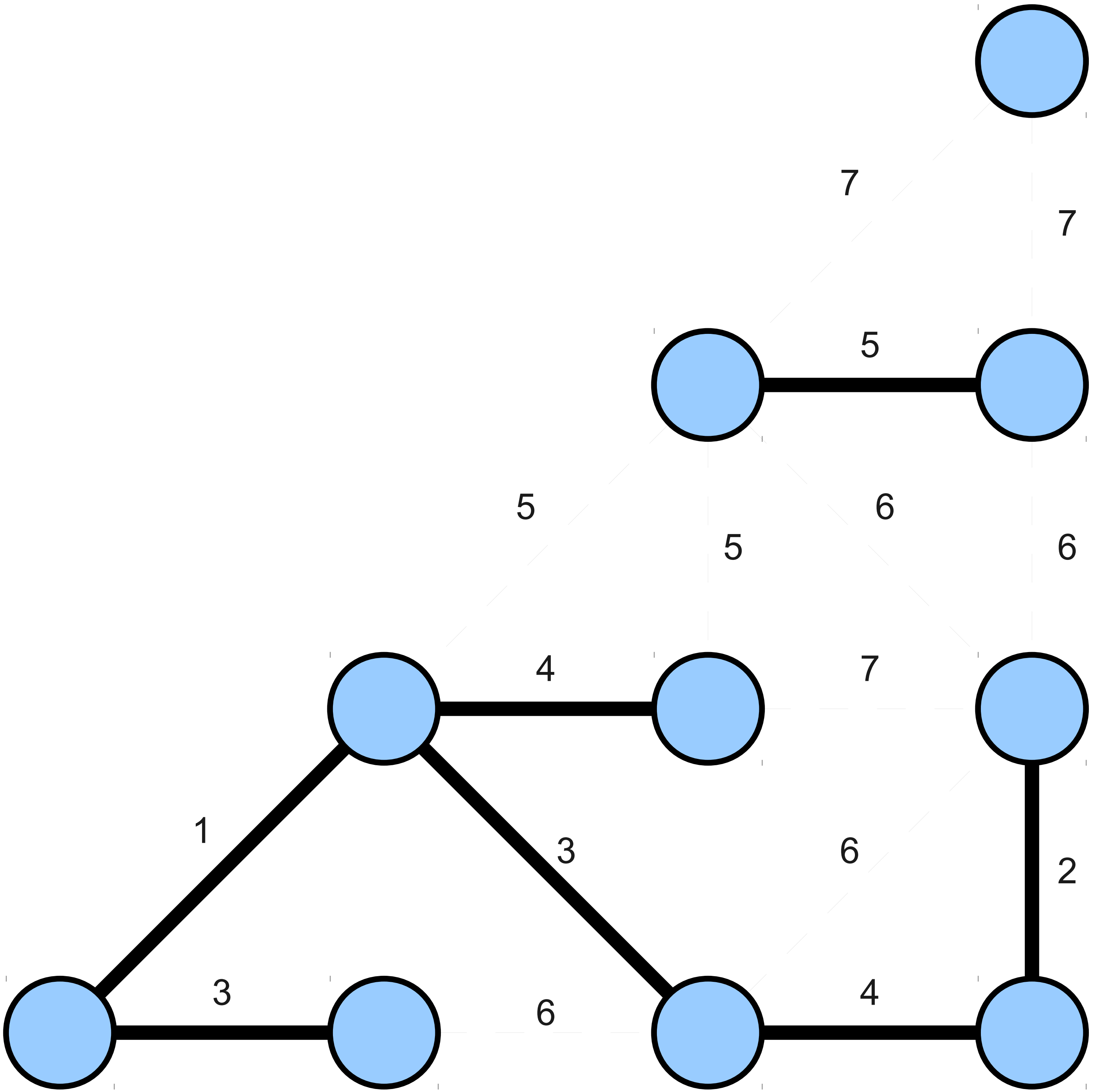
A graph can have many minimum spanning trees. Here, the choice of which length-4 edge we visit first leads to different results.

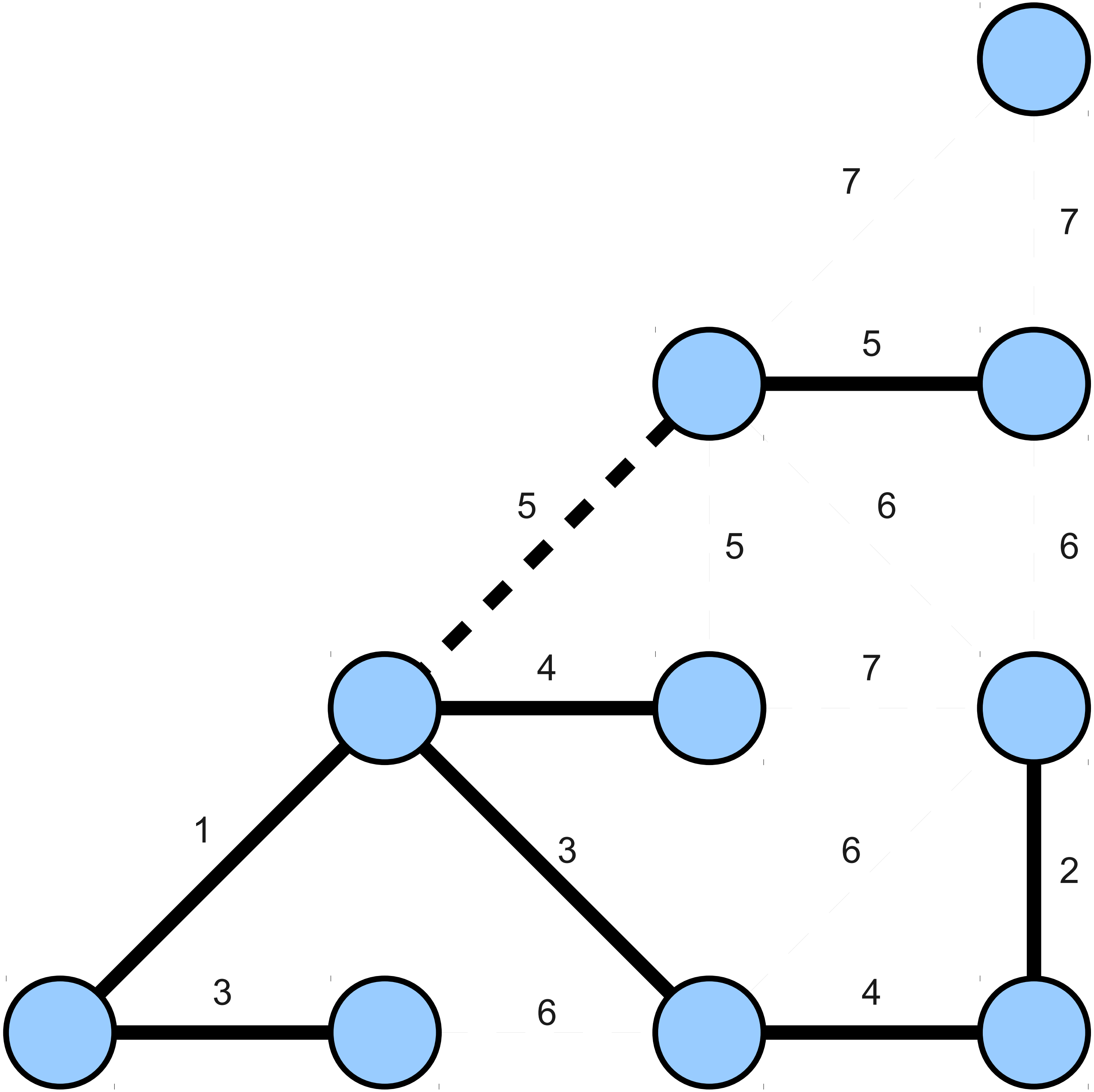

7
7
5
5
5
6
6
4
7
1
3
6
2
3
6
4

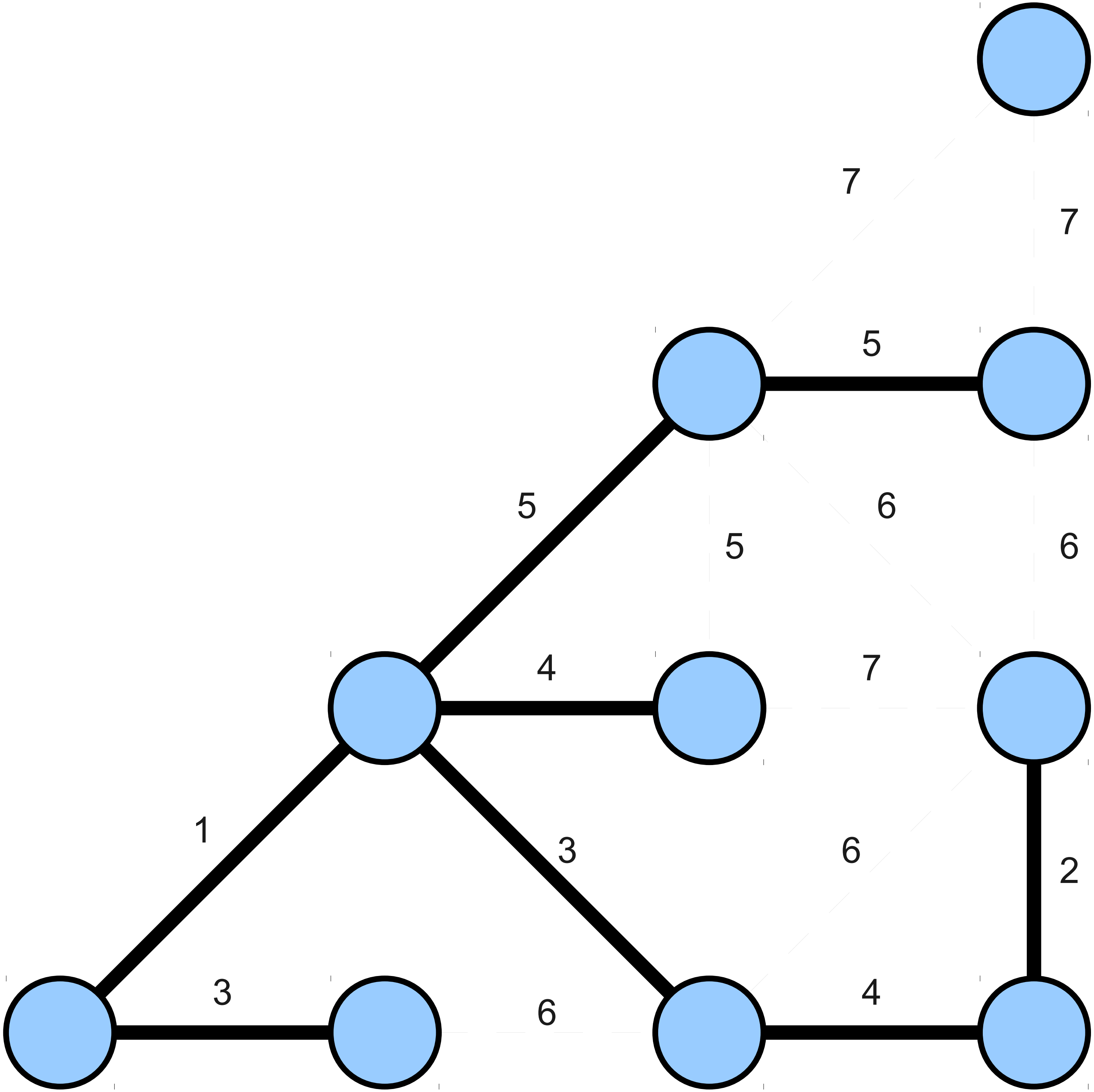

7
7
5
5
5
6
6
4
7
1
3
6
2
3
6
4

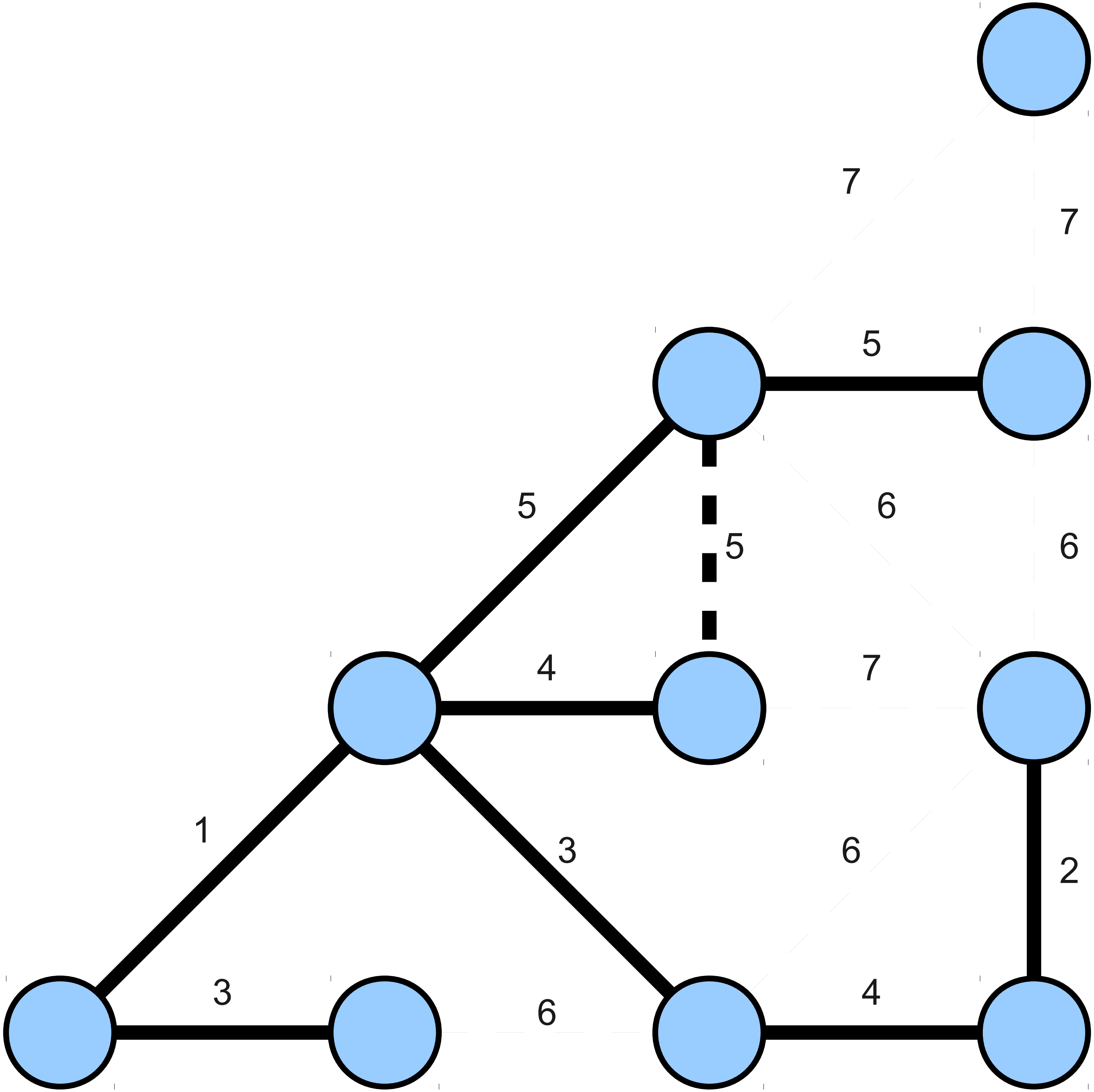

7
7
5
5
5
6
6
4
7
1
3
6
2
3
6
4

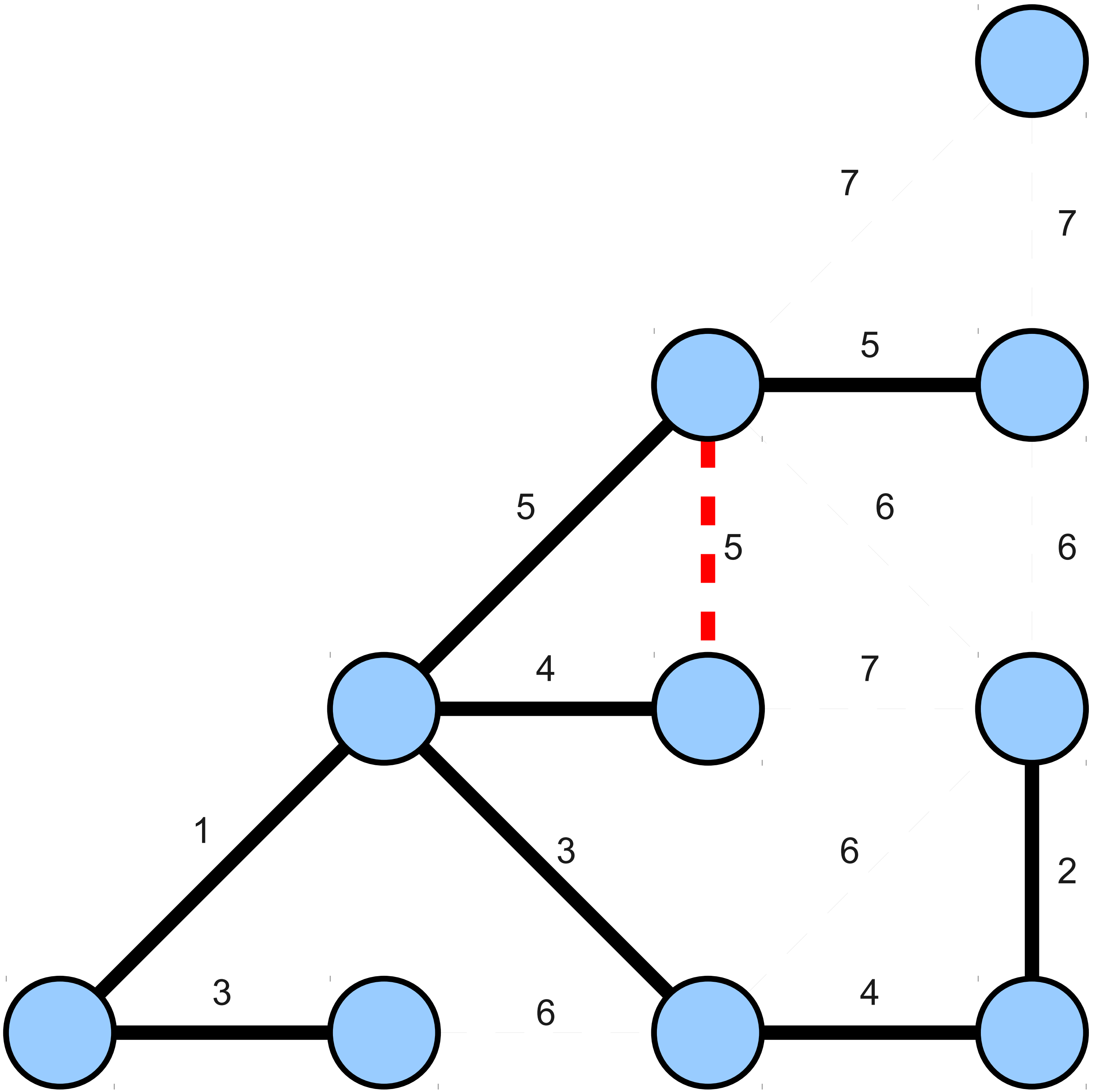

7
7
5
5
5
6
6
4
7
1
3
6
2
3
6
4

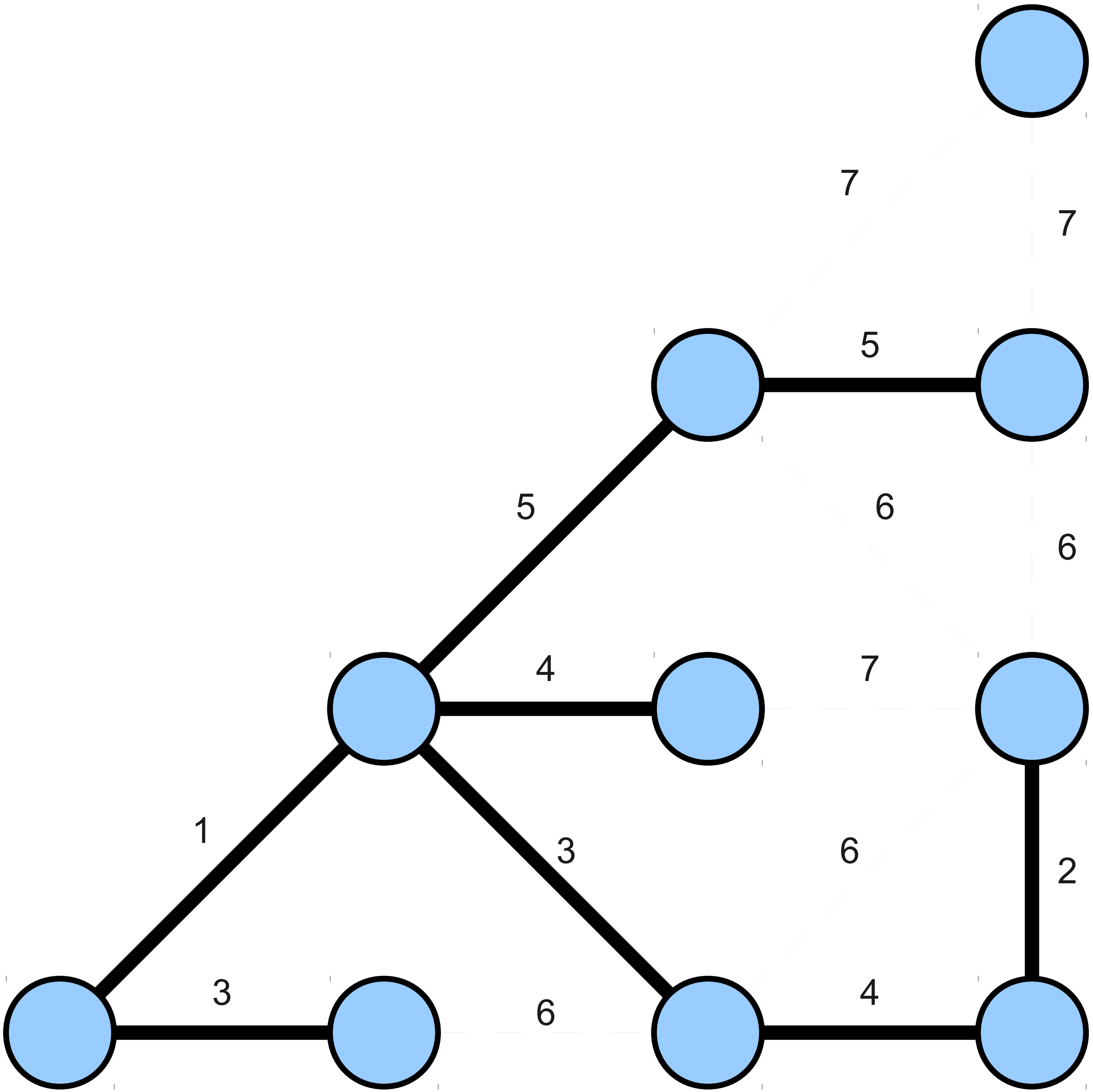

7
7
5
5
5
6
6
4
7
1
3
6
2
3
6
4

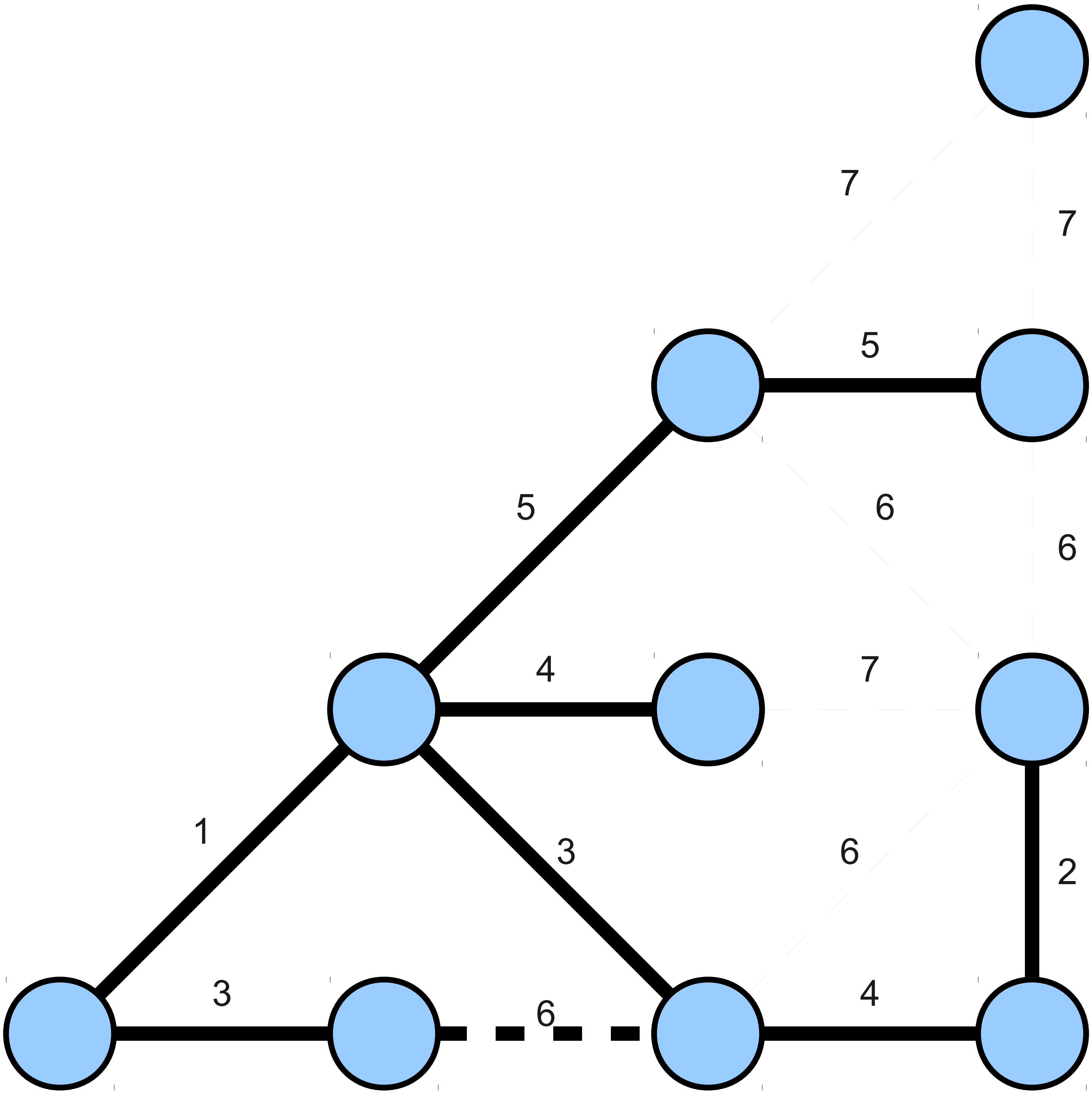

7
7
5
5
5
6
6
4
7
1
3
6
2
3
6
4

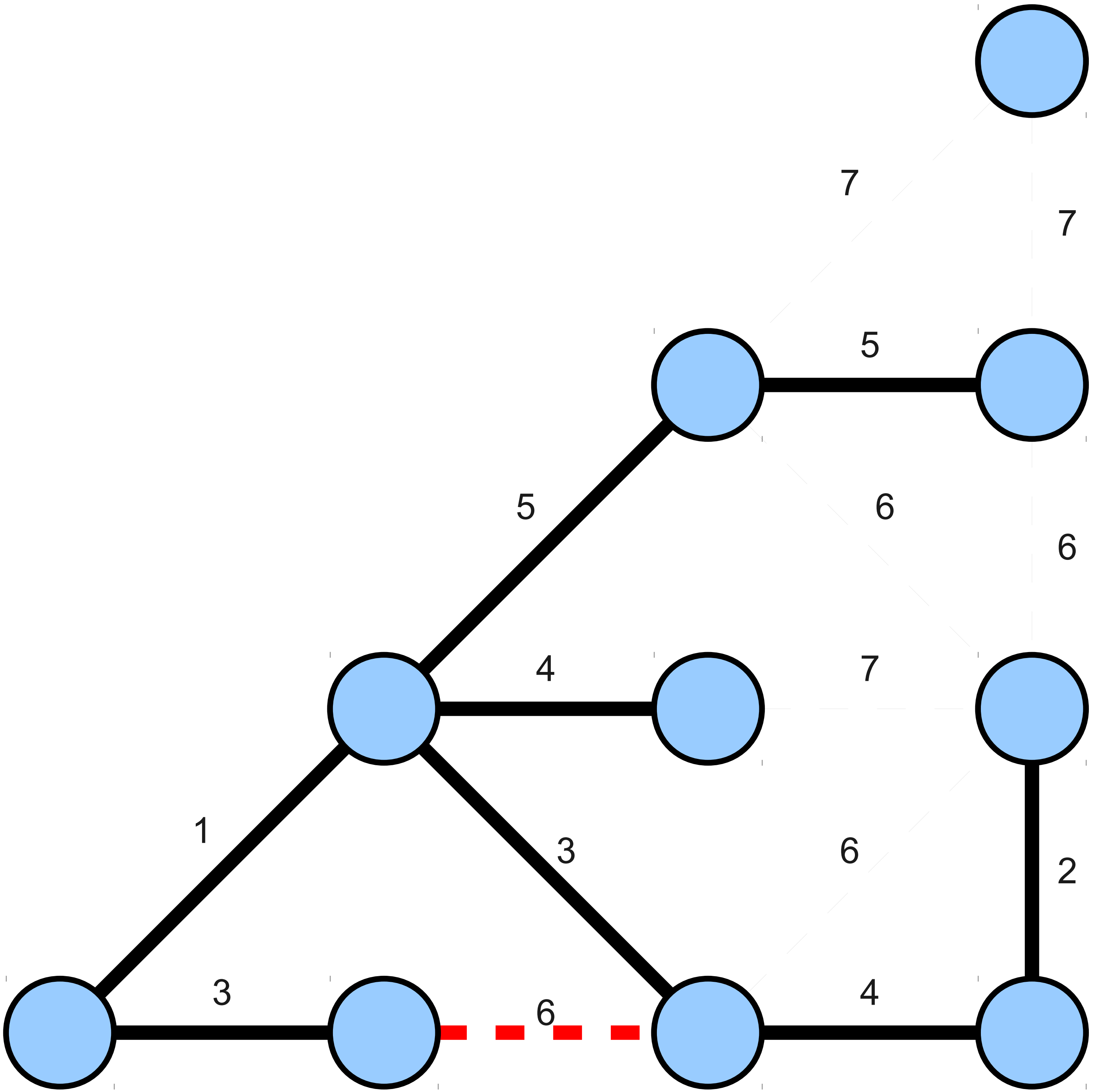

7
7
5
5
6
5
6
4
7
1
3
6
2
3
6
4

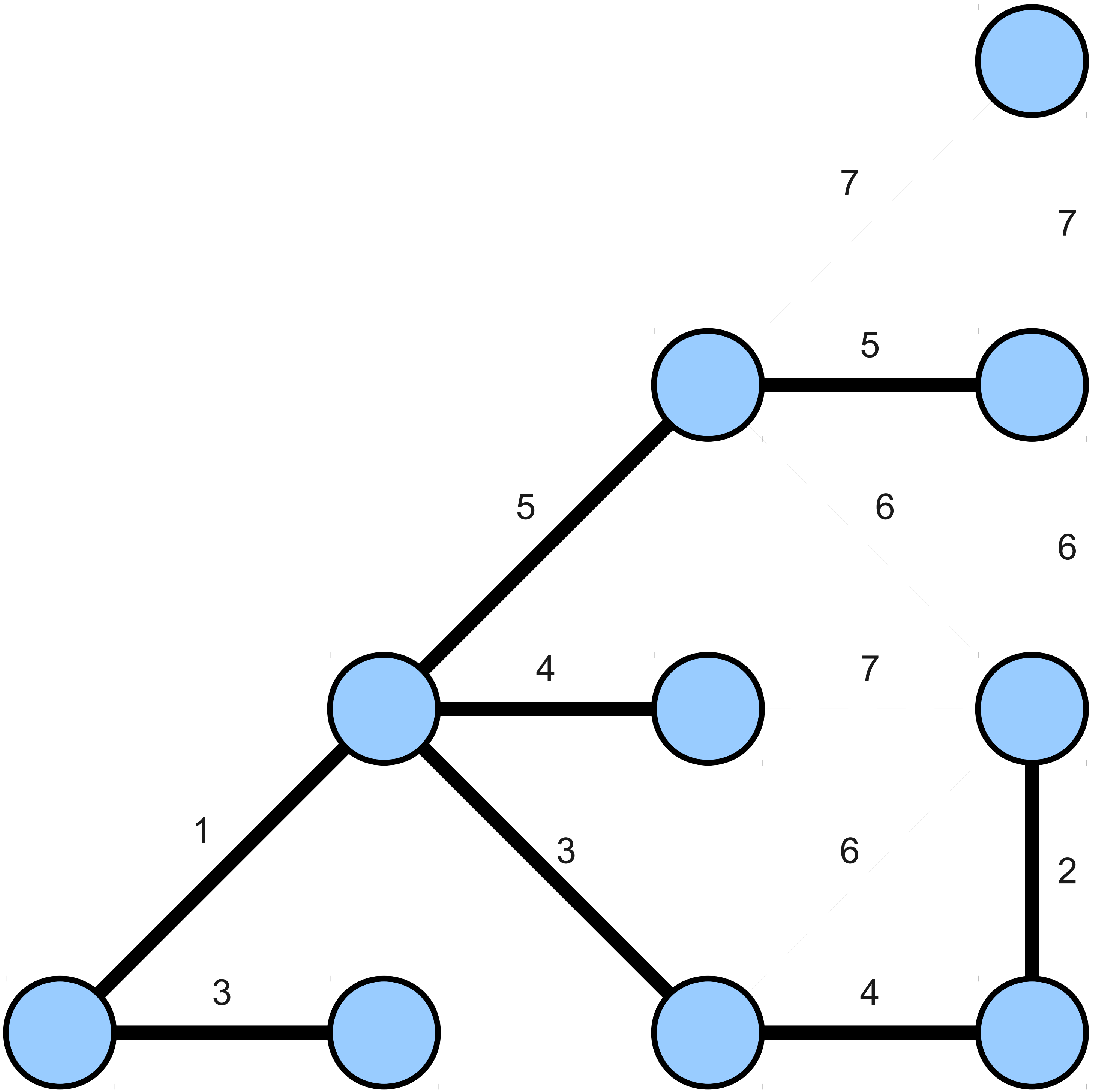

7
7
5
5
5
6
6
4
7
1
3
6
2
3
6
4

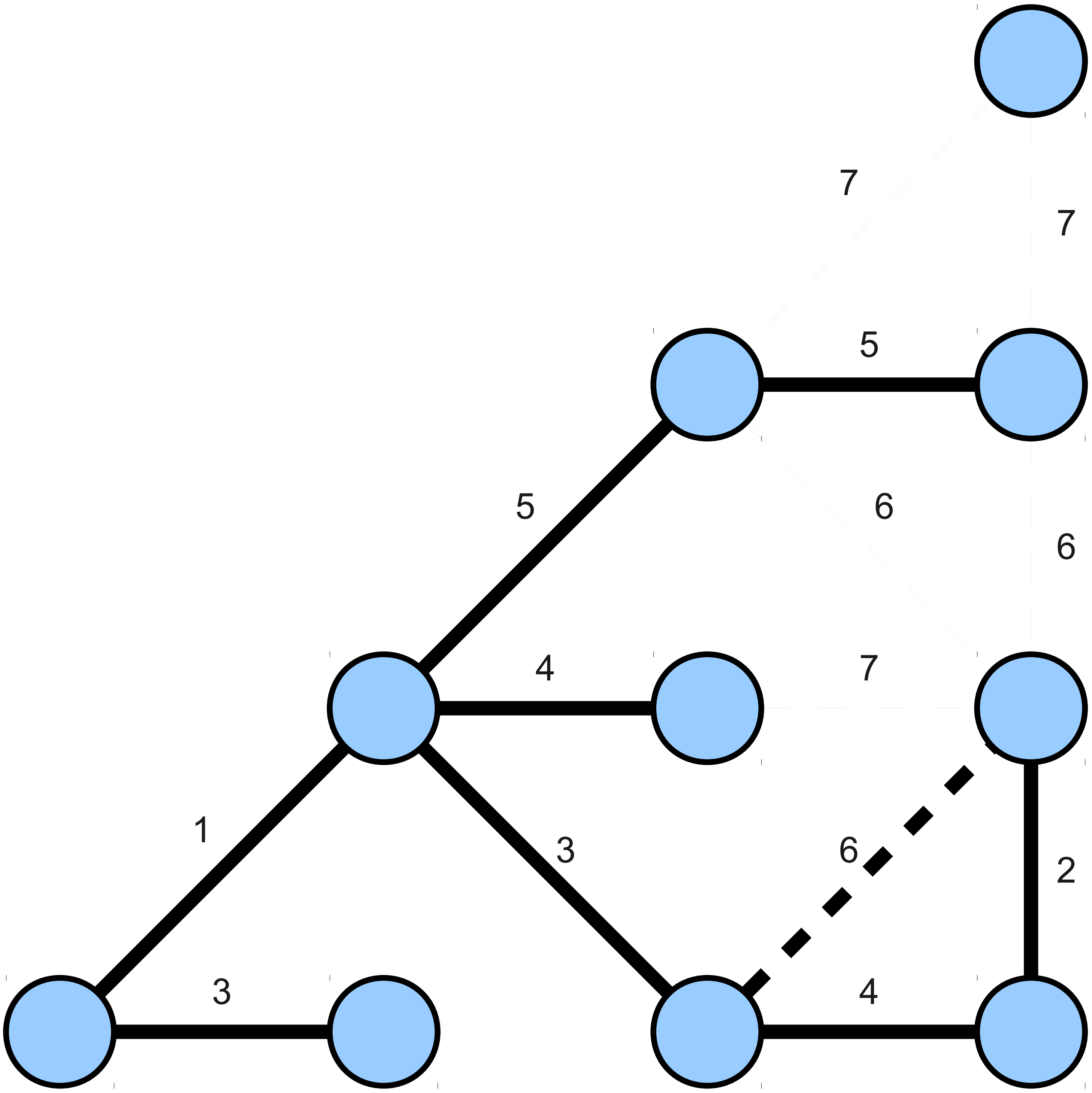

7
7
5
5
6
5
6
4
7
1
3
6
2
3
6
4

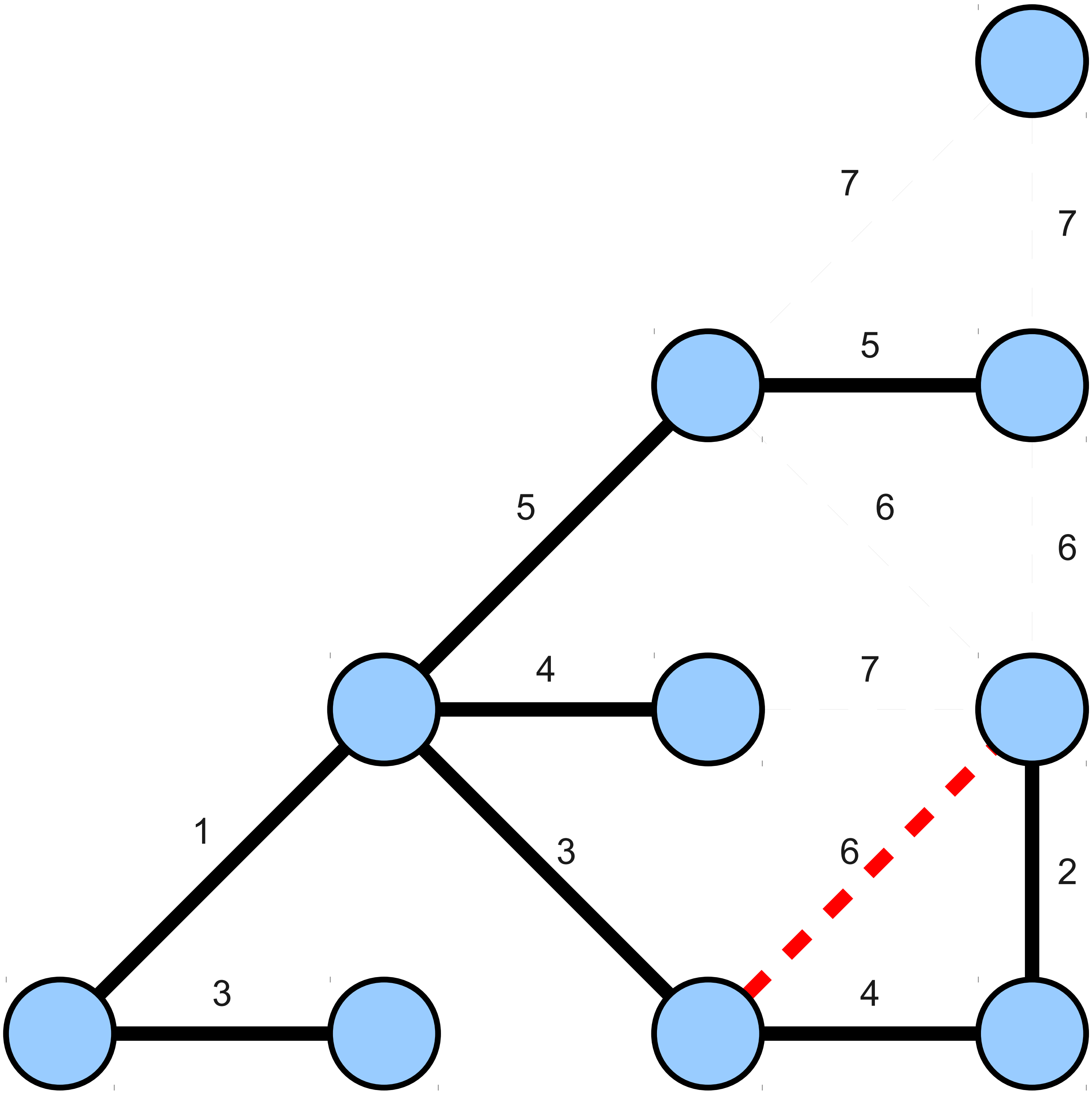

7
7
5
5
6
6
4
7
1
3
6
2
3
6
4

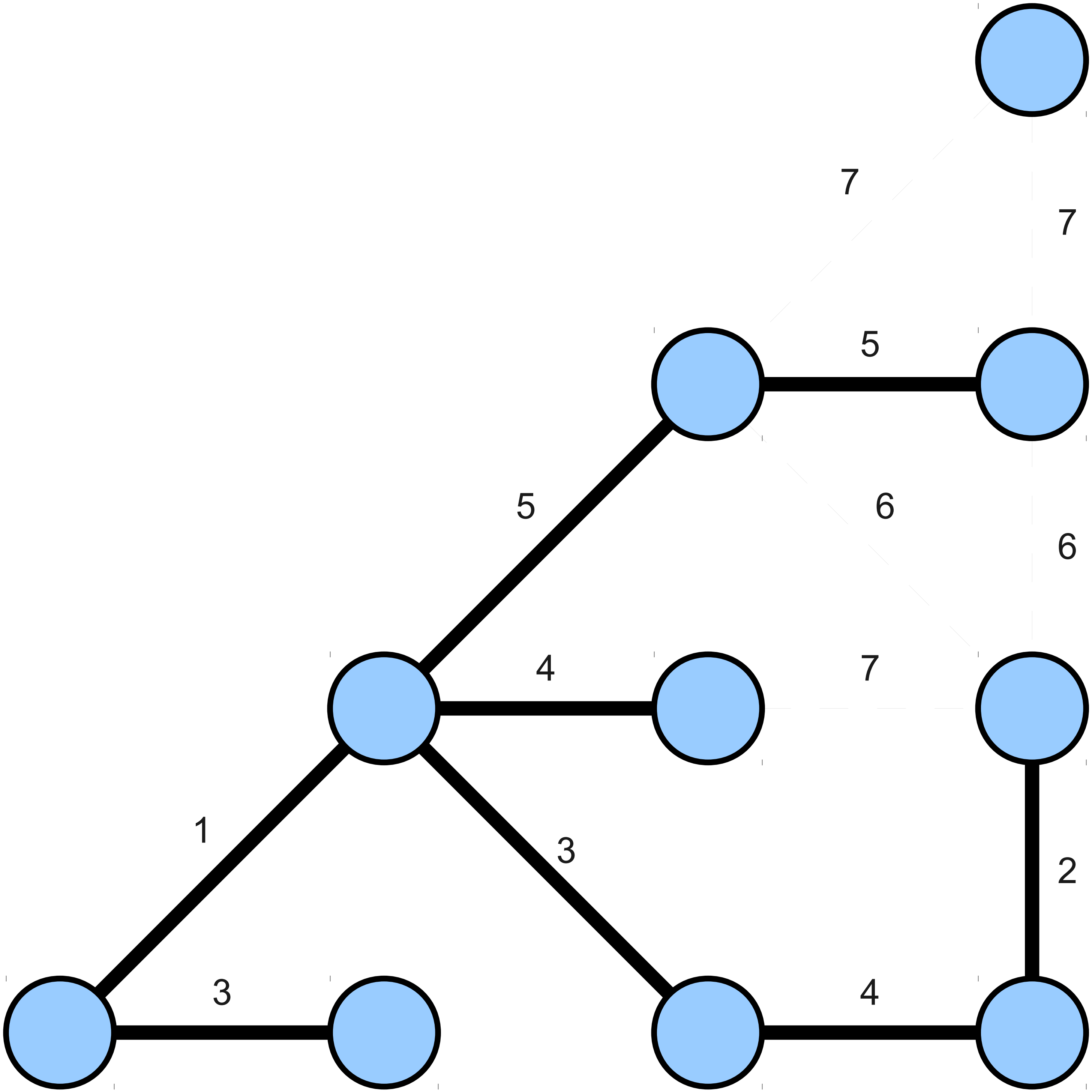

7
7
5
5
6
6
4
7
1
3
6
2
3
6
4

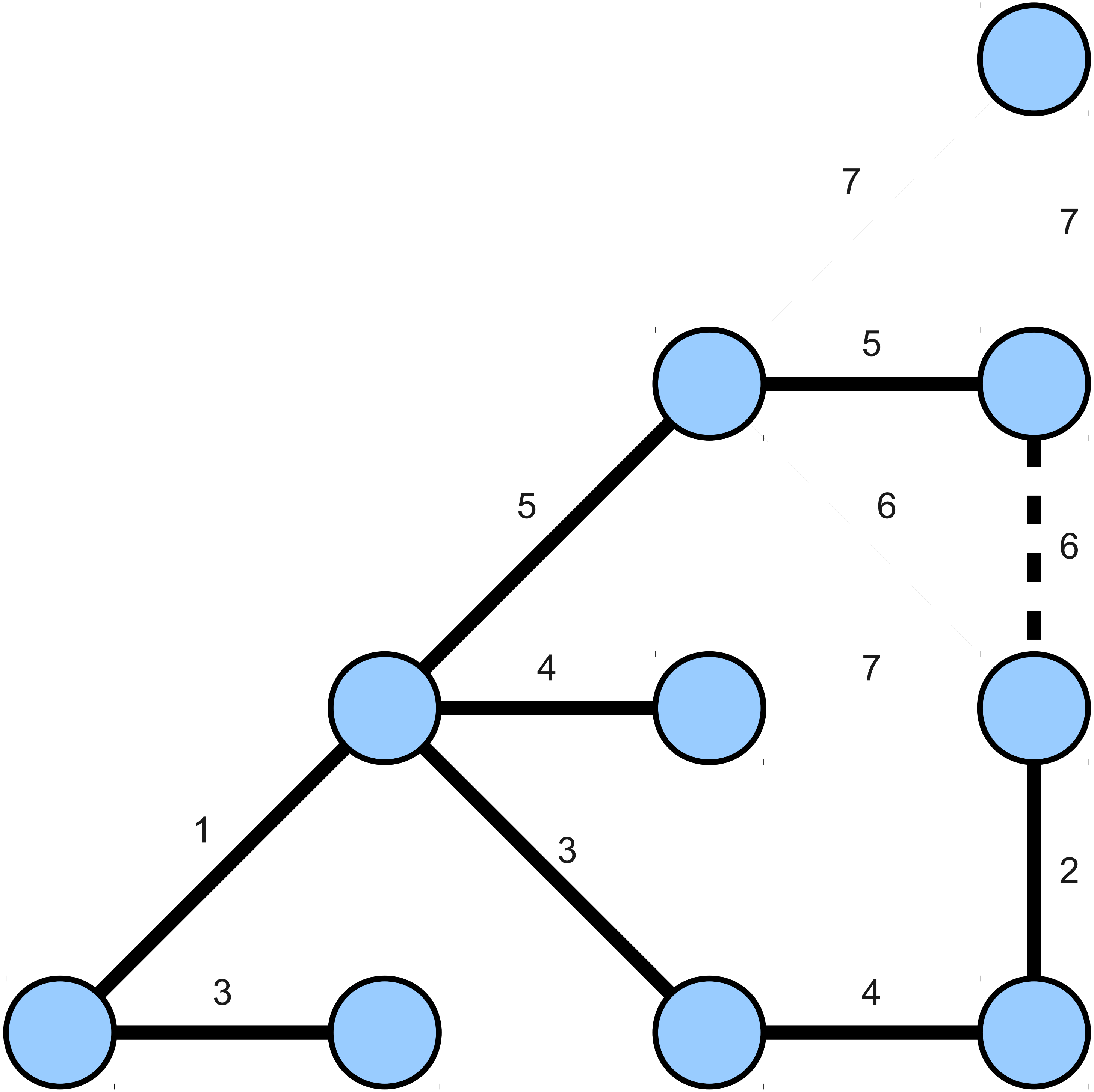

7
7
5
5
6
6
4
7
1
3
6
2
3
6
4

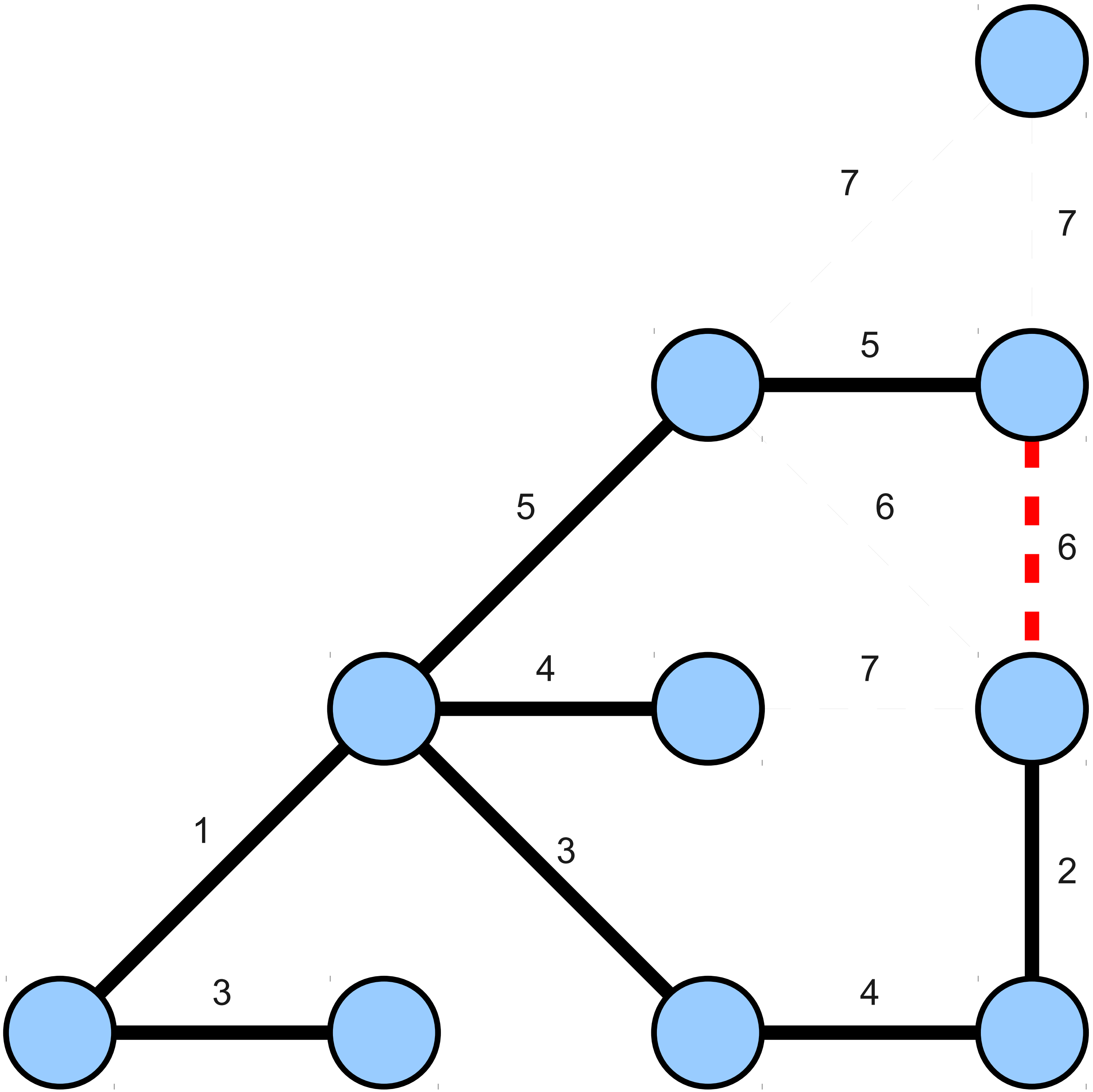

7
7
5
5
6
6
4
7
1
3
6
2
3
4

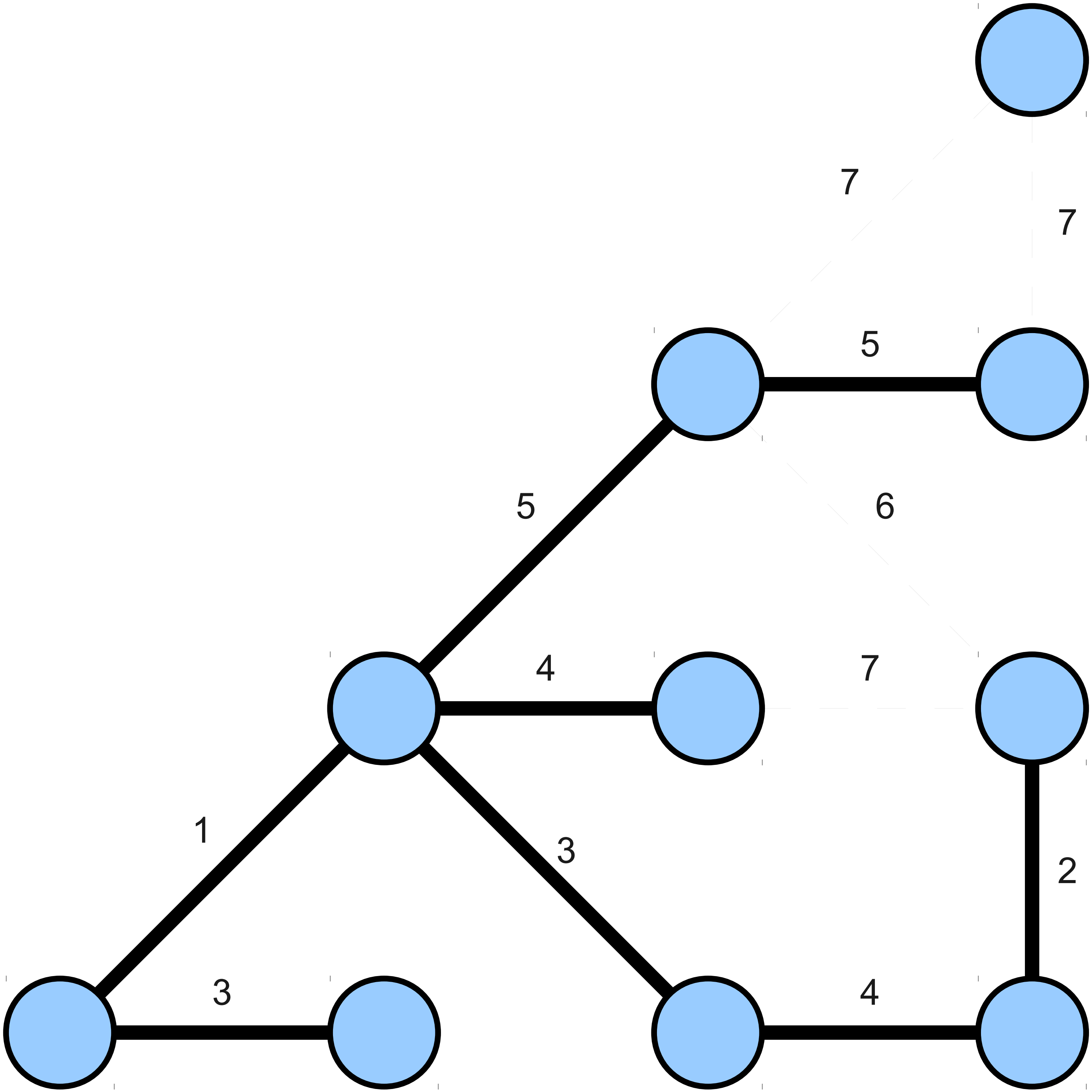

7
7
5
5
6
6
4
7
1
3
6
2
3
4

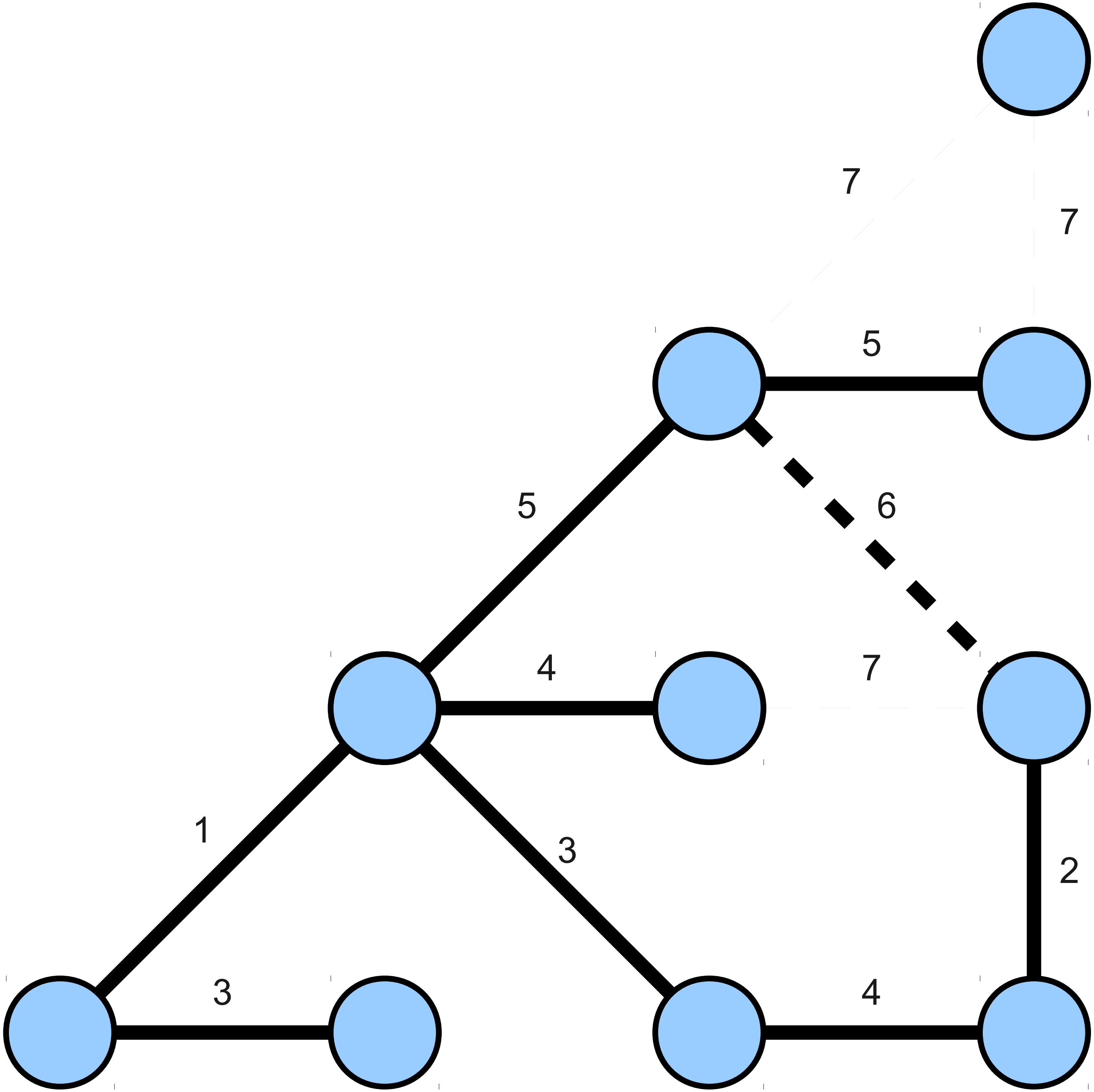

7
7
5
5
6
6
4
7
1
3
6
2
3
4

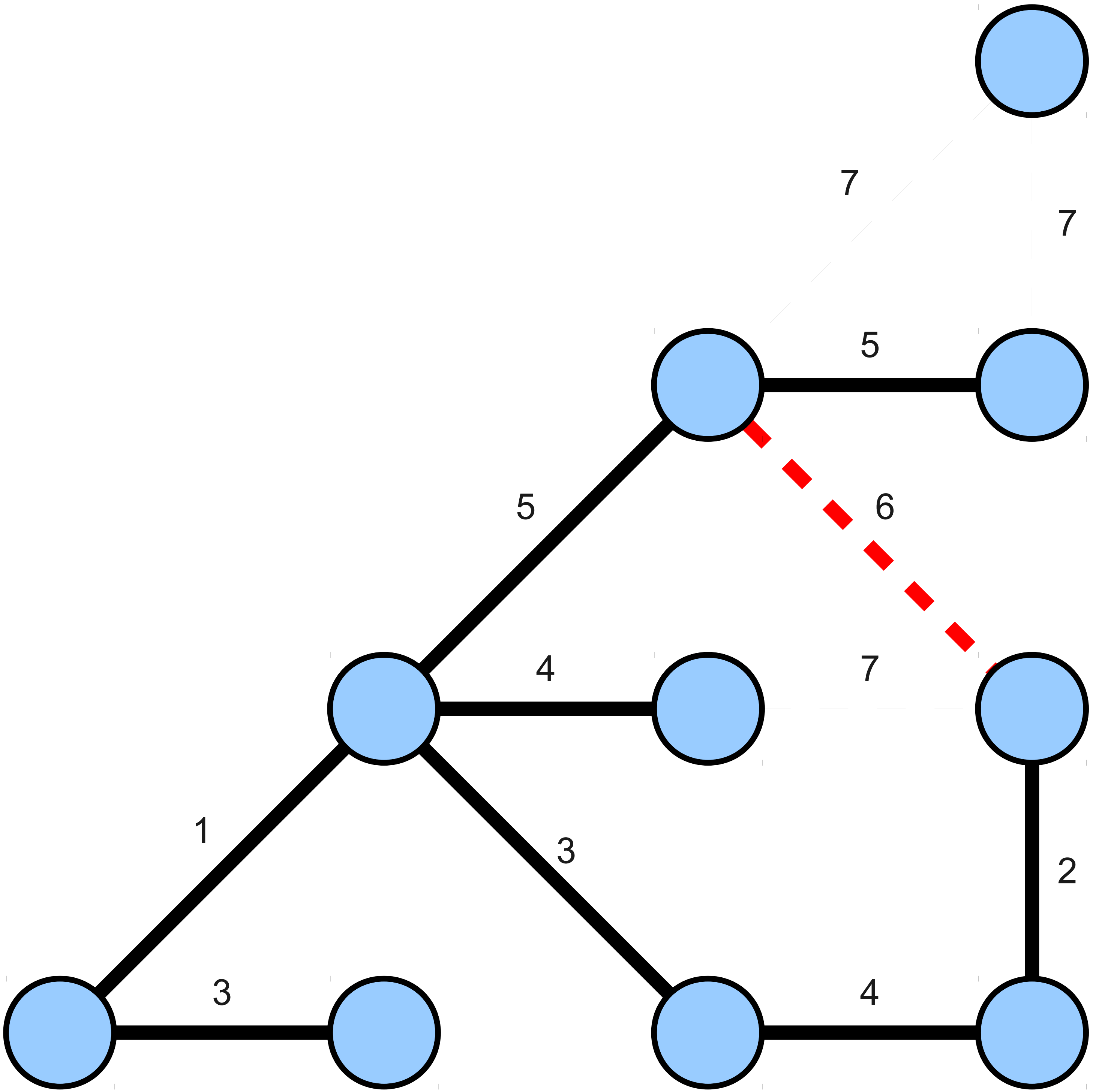

7
7
5
5
6
6
4
7
1
3
2
3
4

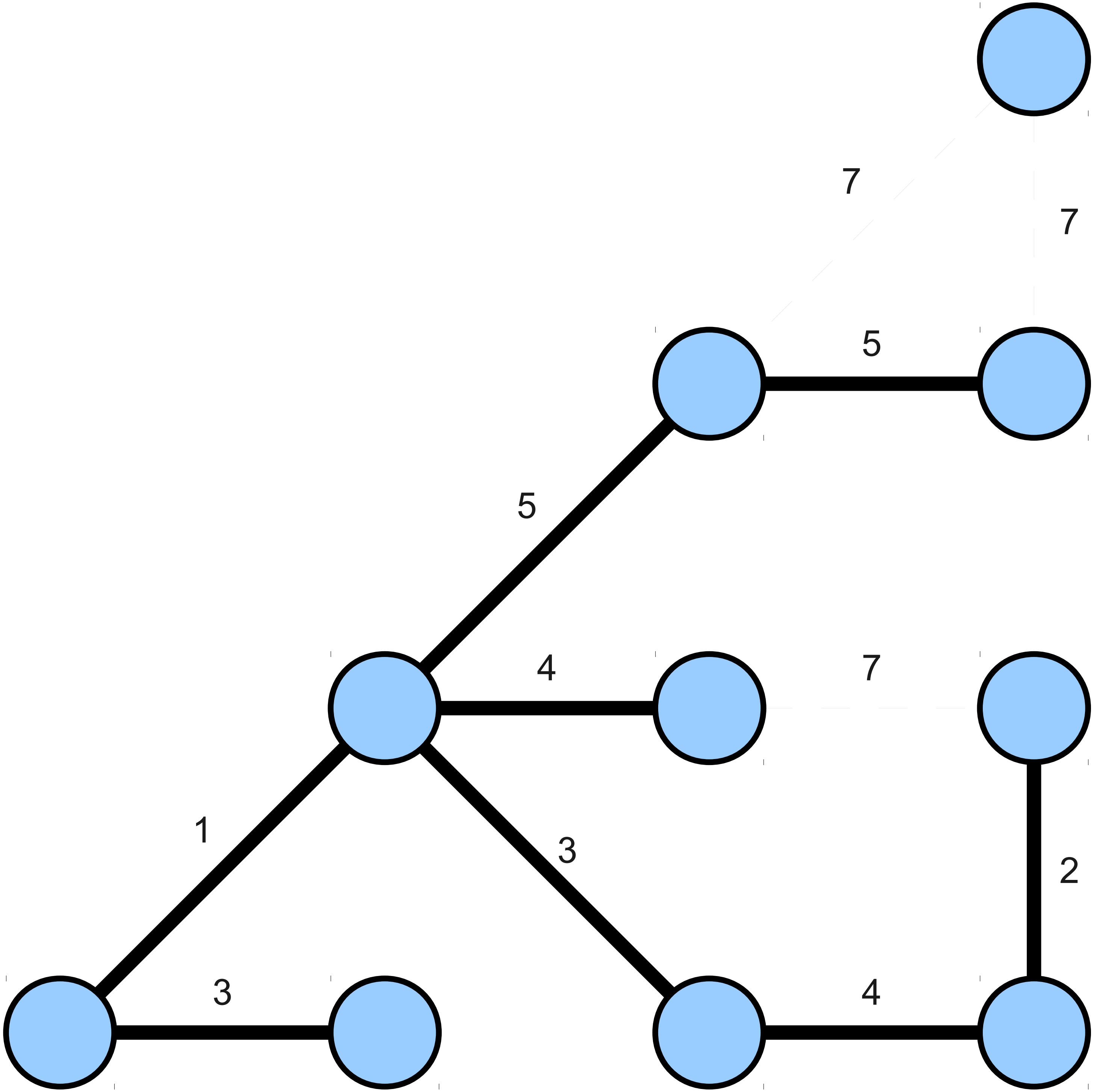

7
7
5
5
6
6
4
7
1
3
2
3
4

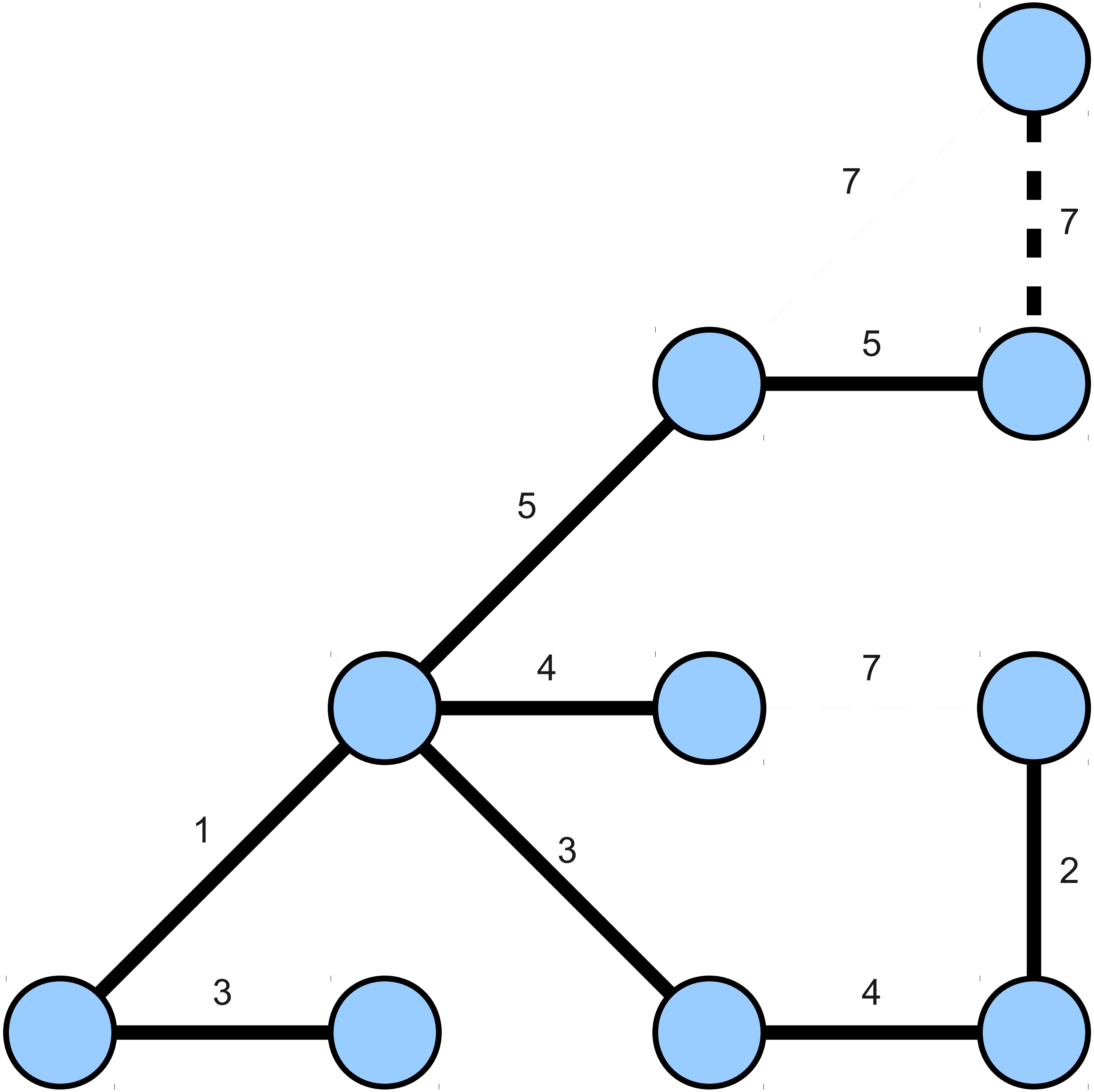

7
7
5
5
6
6
4
7
1
3
2
3
4

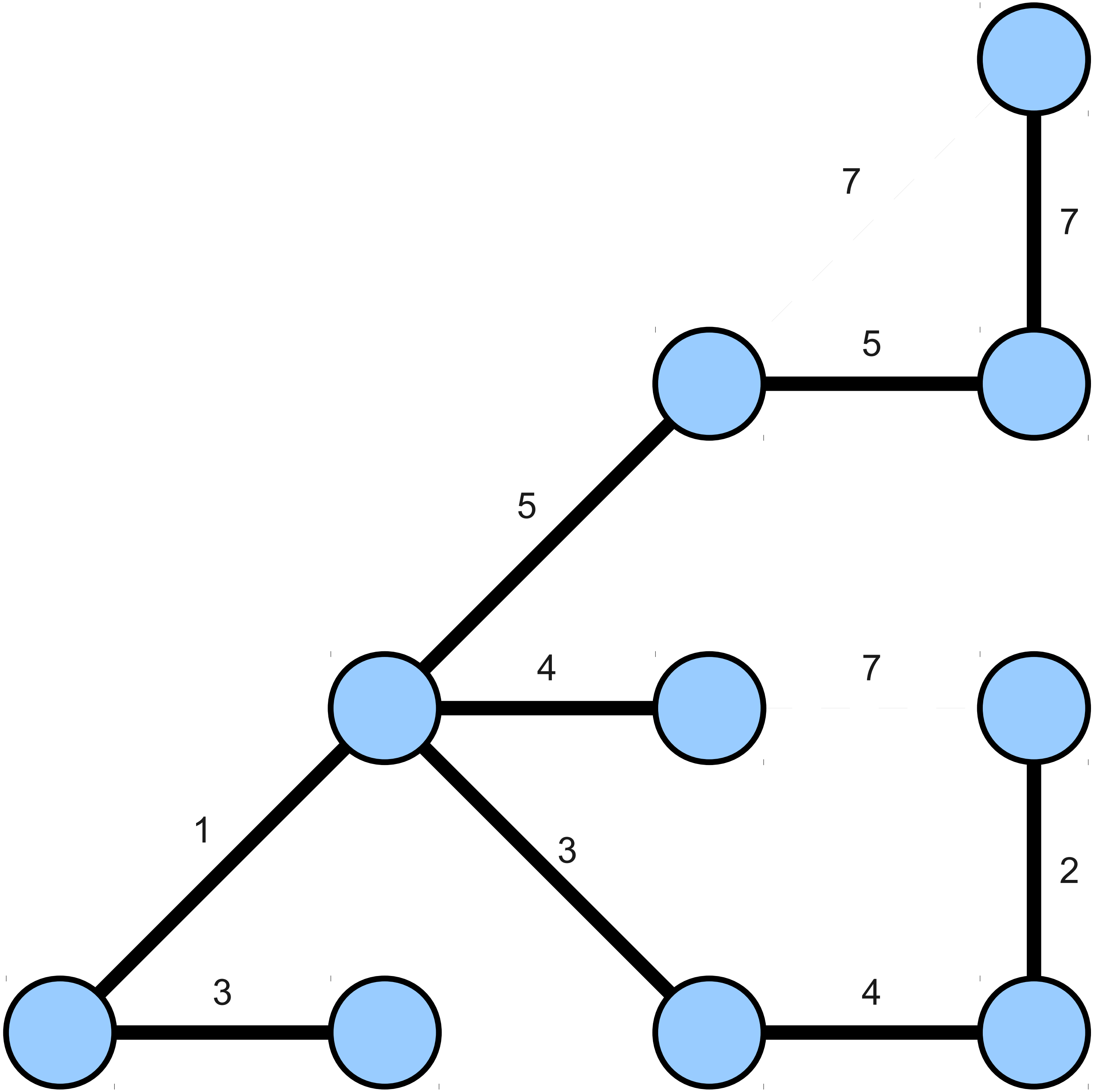

7
7
5
5
6
4
7
1
3
2
3
4

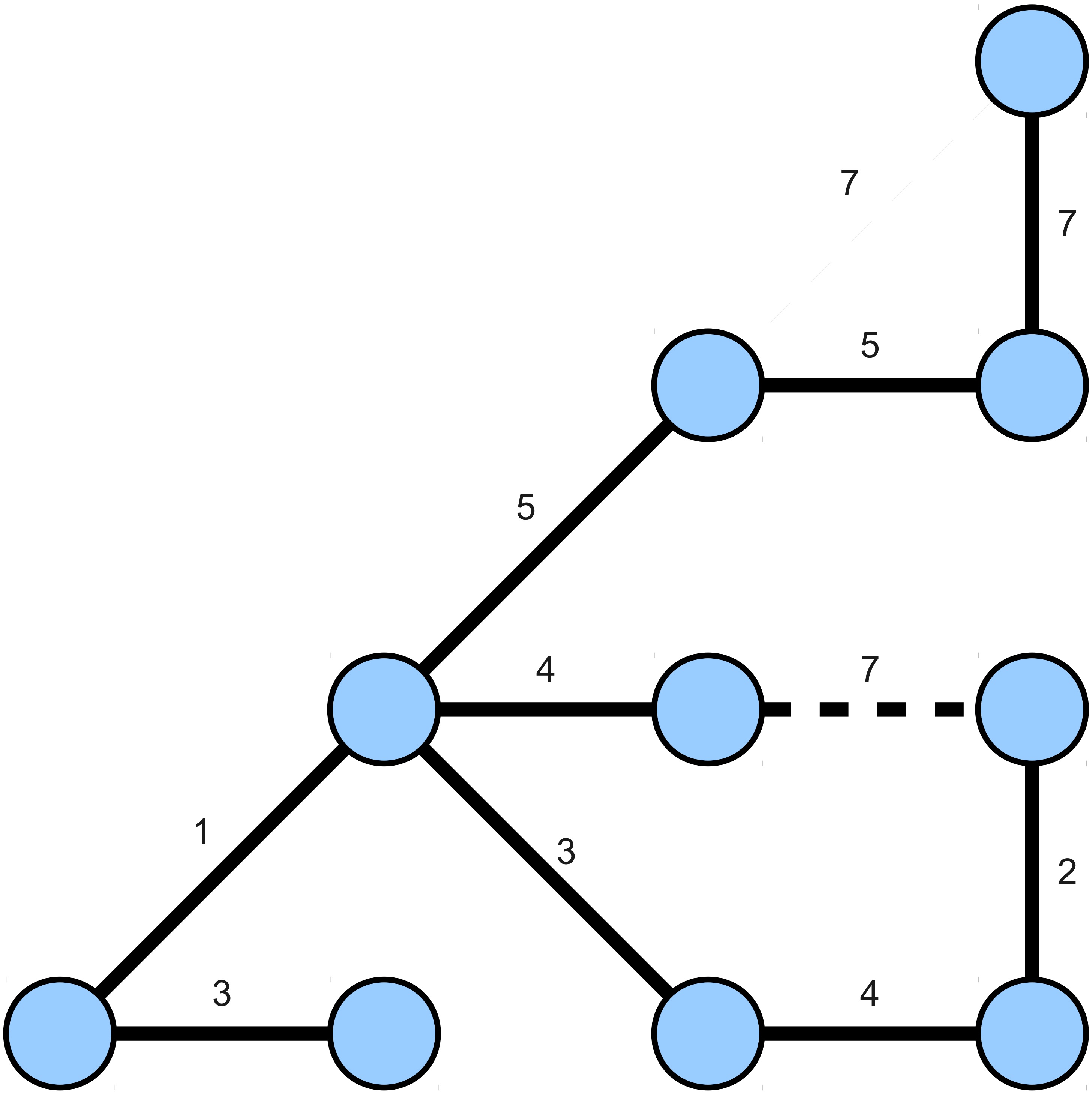

7
7
5
5
6
4
7
1
3
2
3
4

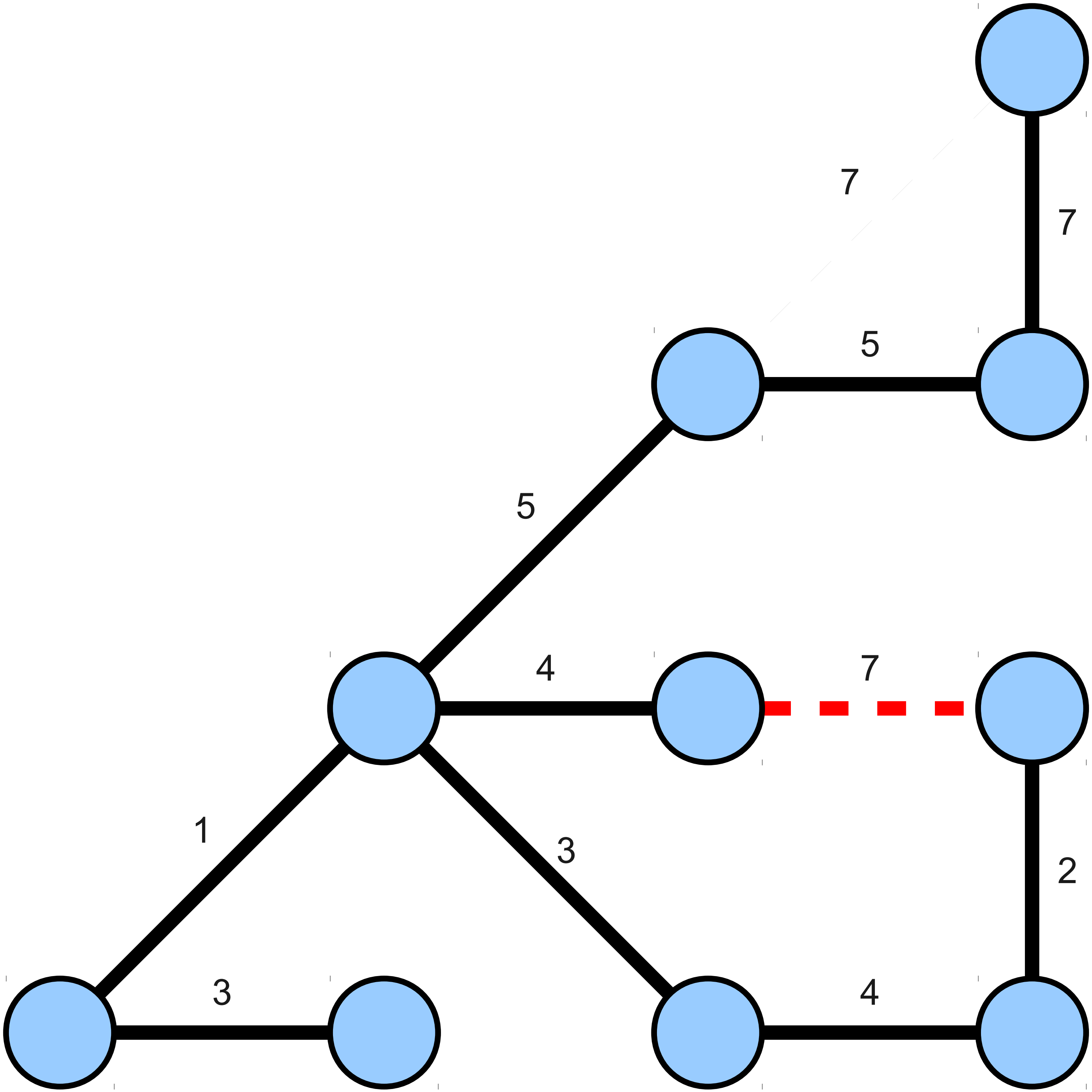

7
7
5
5
6
4
7
1
3
2
3
4

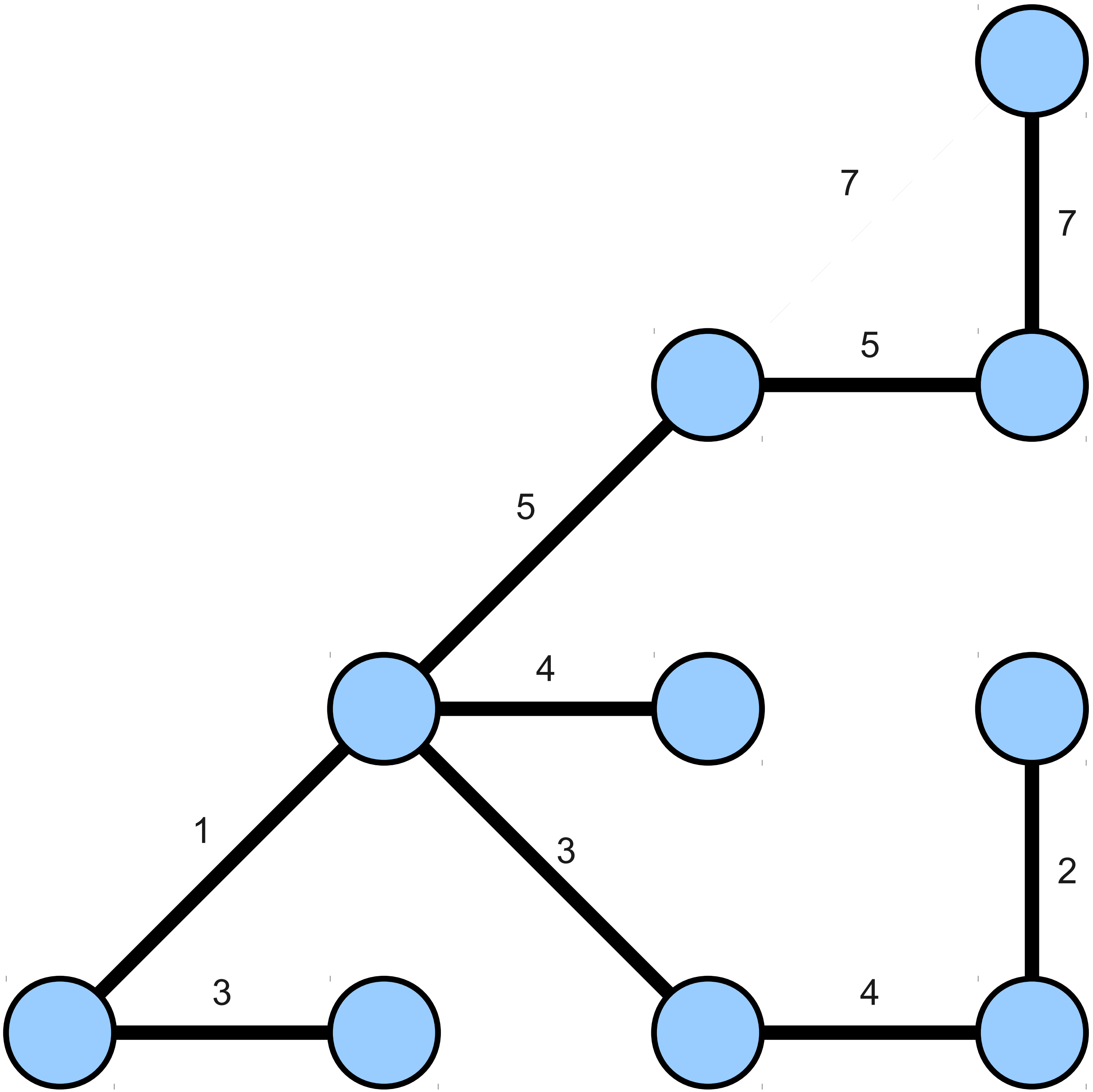

7
7
5
5
4
7
1
3
2
3
4

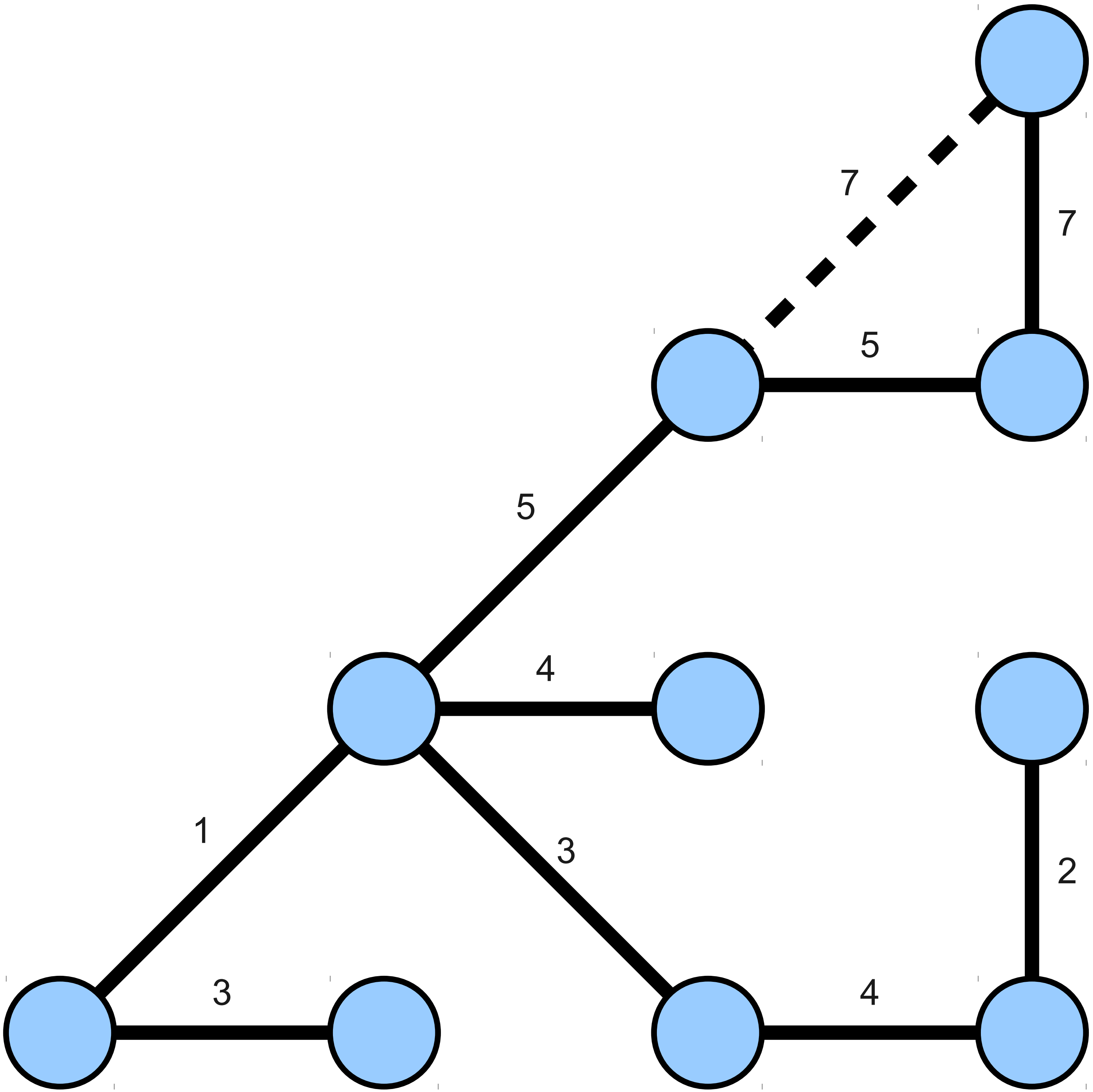

7
7
5
5
4
7
1
3
2
3
4

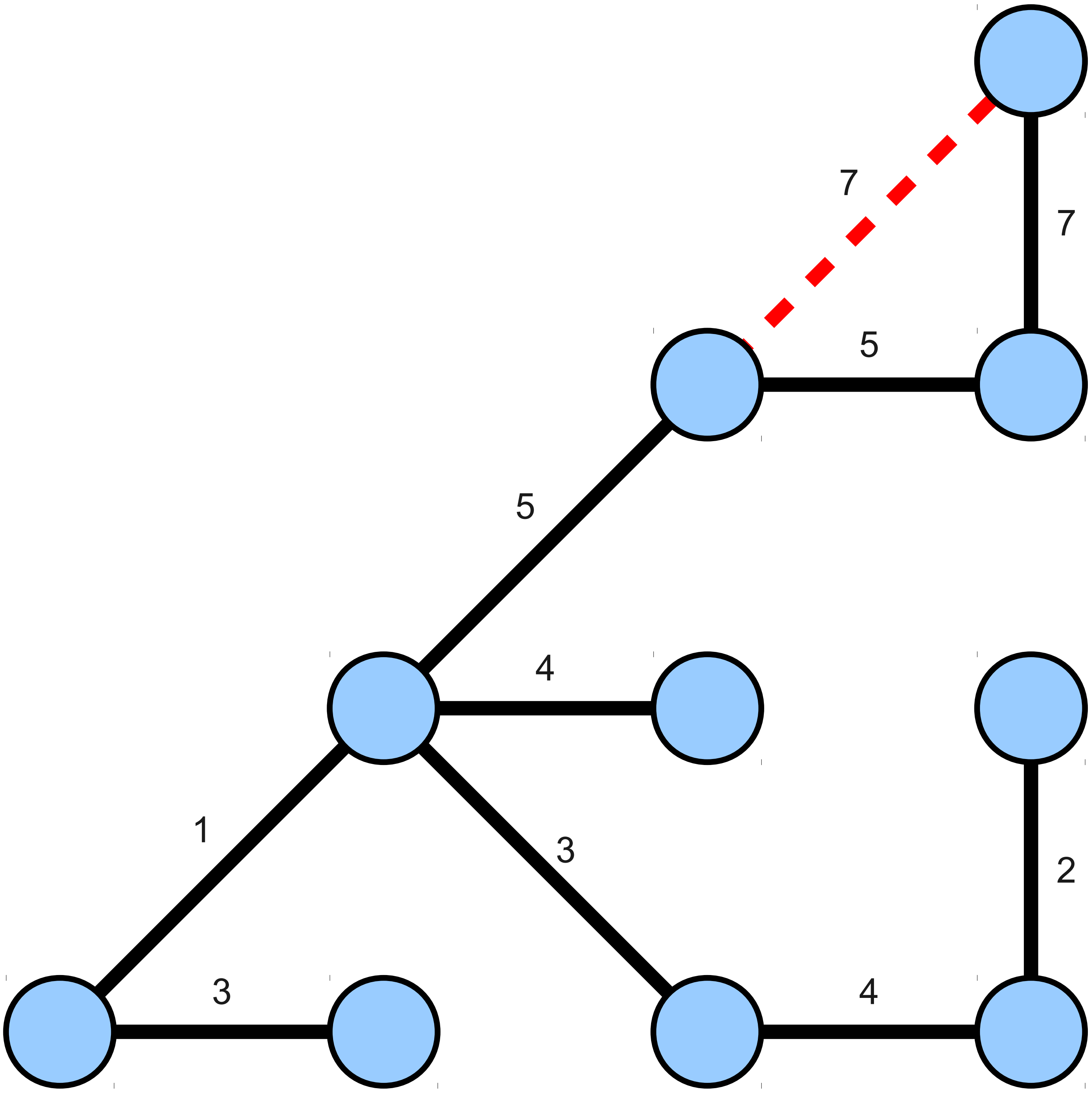

7
7
5
5
4
7
1
3
2
3
4

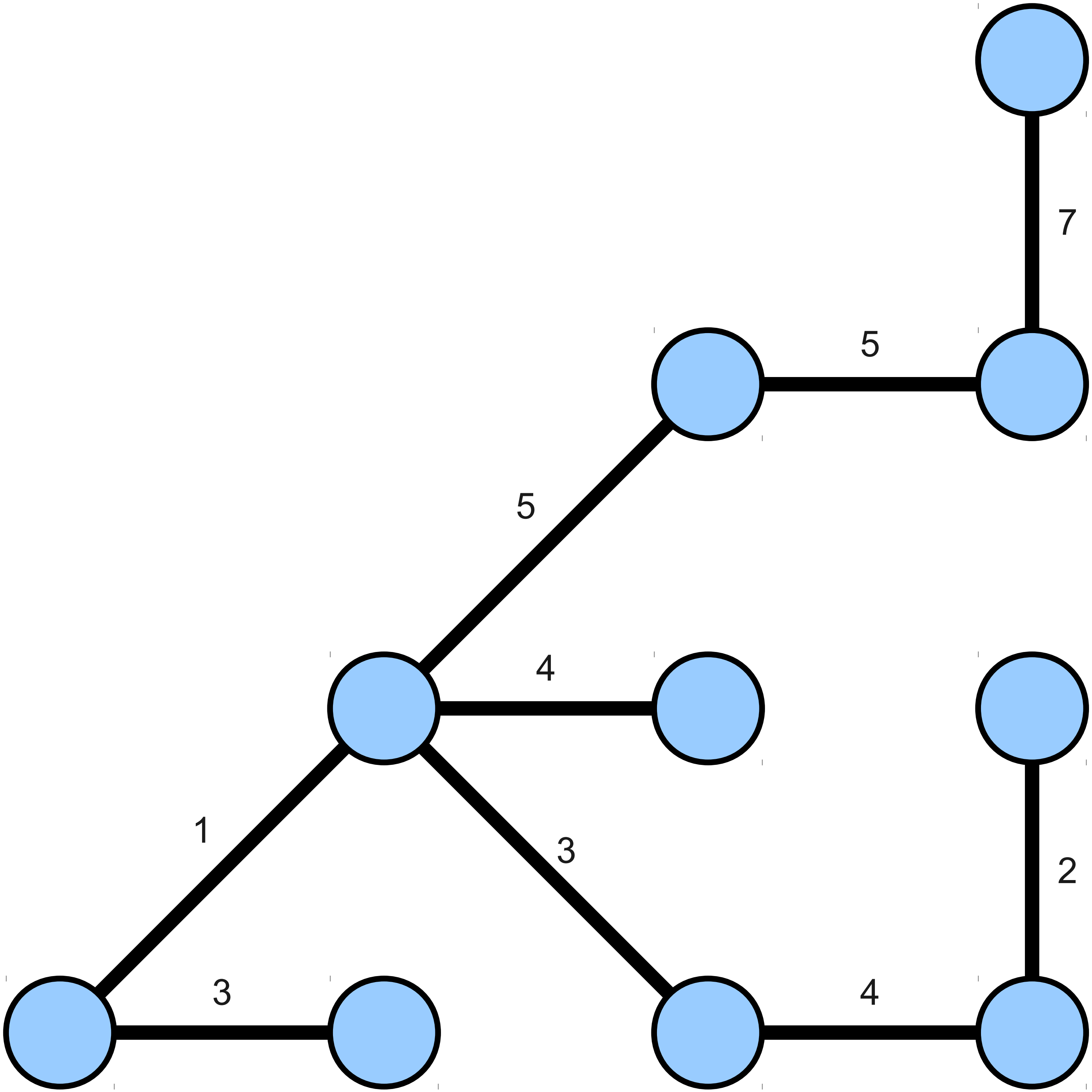

7
7
5
5
4
7
1
3
2
3
4


7
7
5
5
4
7
1
3
2
3
4


7
7
5
5
4
1
3
2
3
4


7
7
5
5
4
1
3
2
3
4

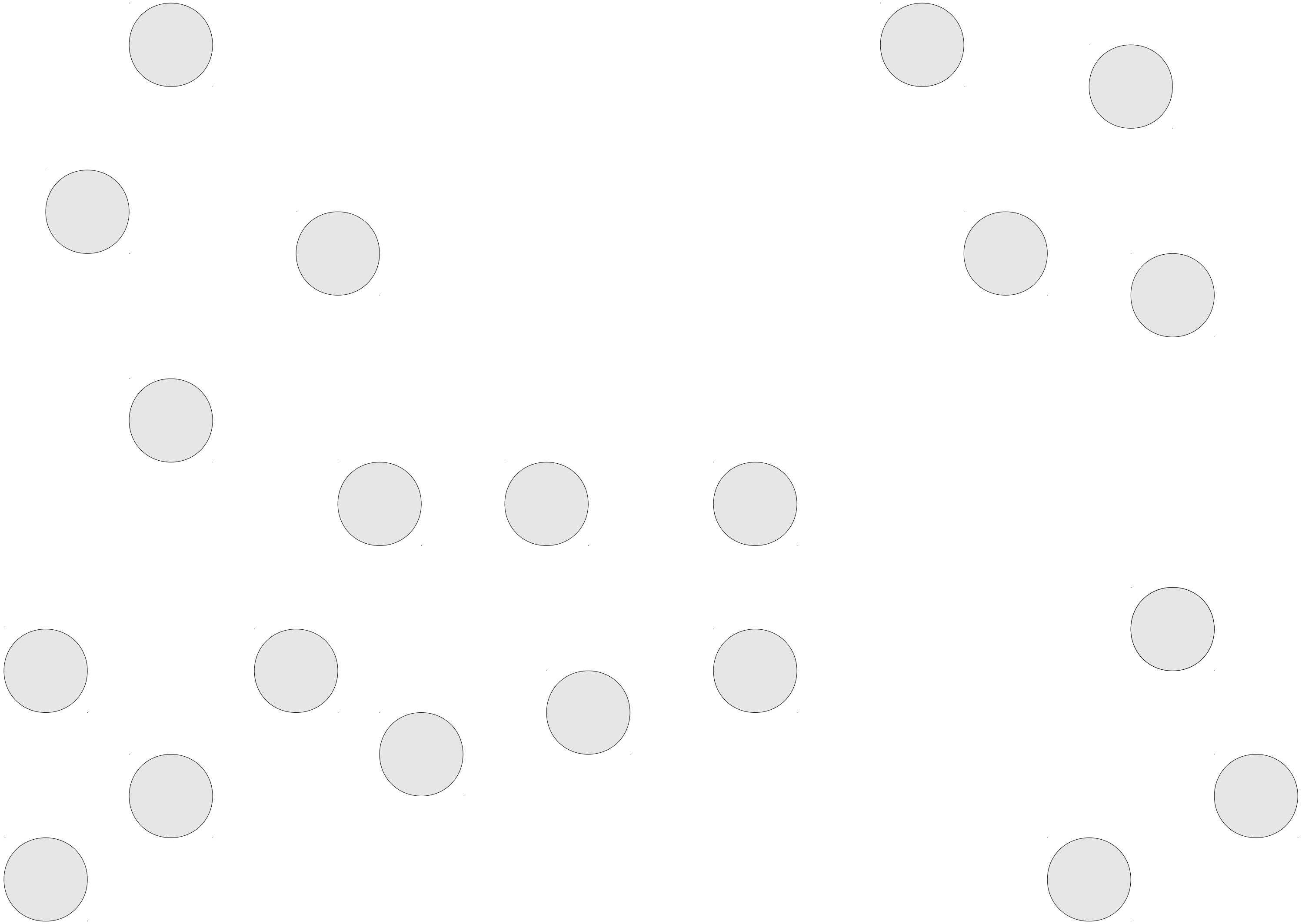

7
7
5
5
4
1
3
2
3
4


7
5
5
4
1
3
2
3
4

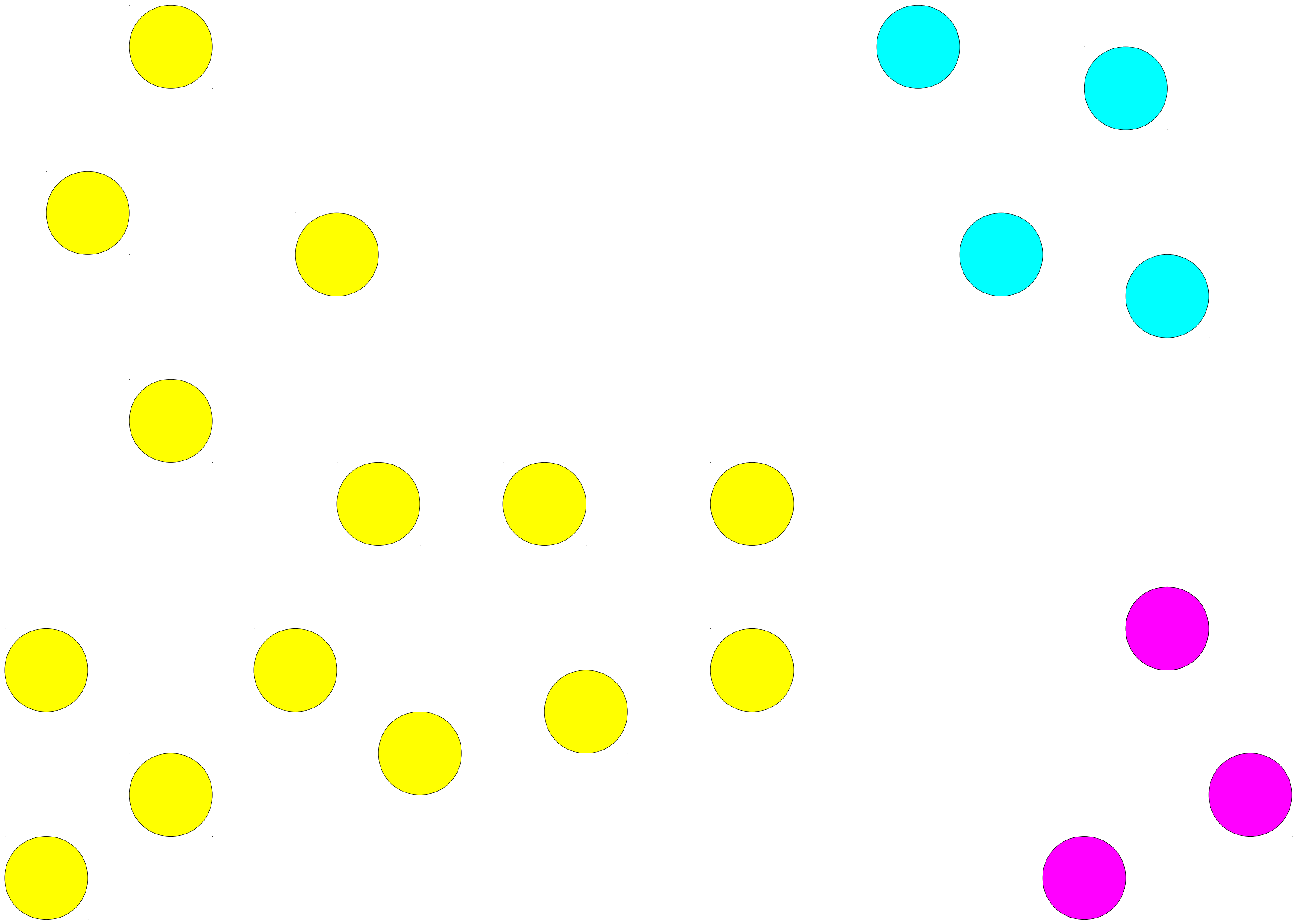
Maintaining Connectivity

- One of the key steps in Kruskal's algorithm is determining whether two nodes are connected to one another.
- There are many ways to do this:
  - Could do a DFS in the partially-constructed graph to see if the two nodes are reachable from one another.
  - Could store a list of all the clusters of nodes that are connected to one another.
- Classiest implementation: use a **union/find data structure**.
  - Check Wikipedia for details; it's surprisingly simple!

Data Clustering



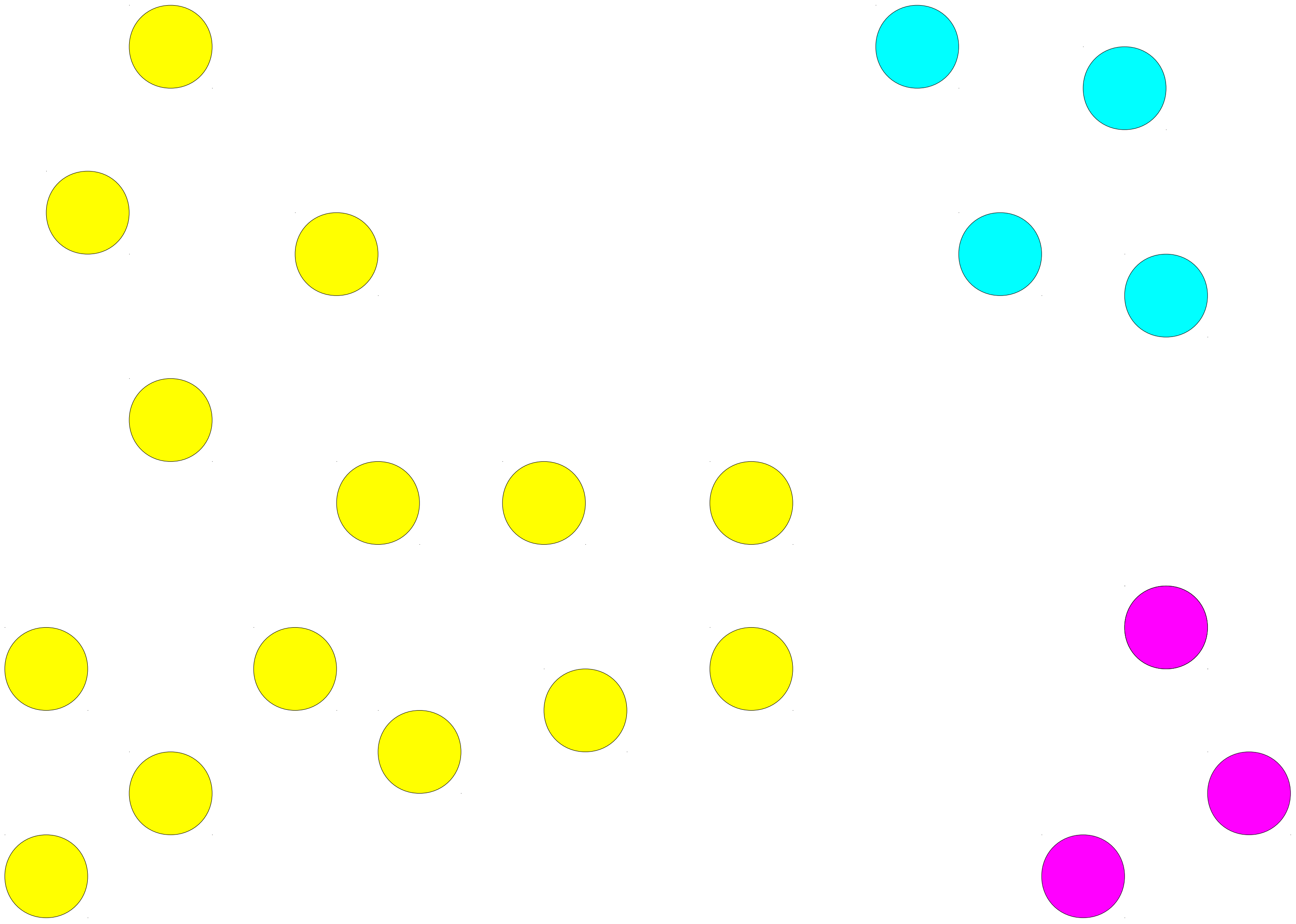
Data Clustering



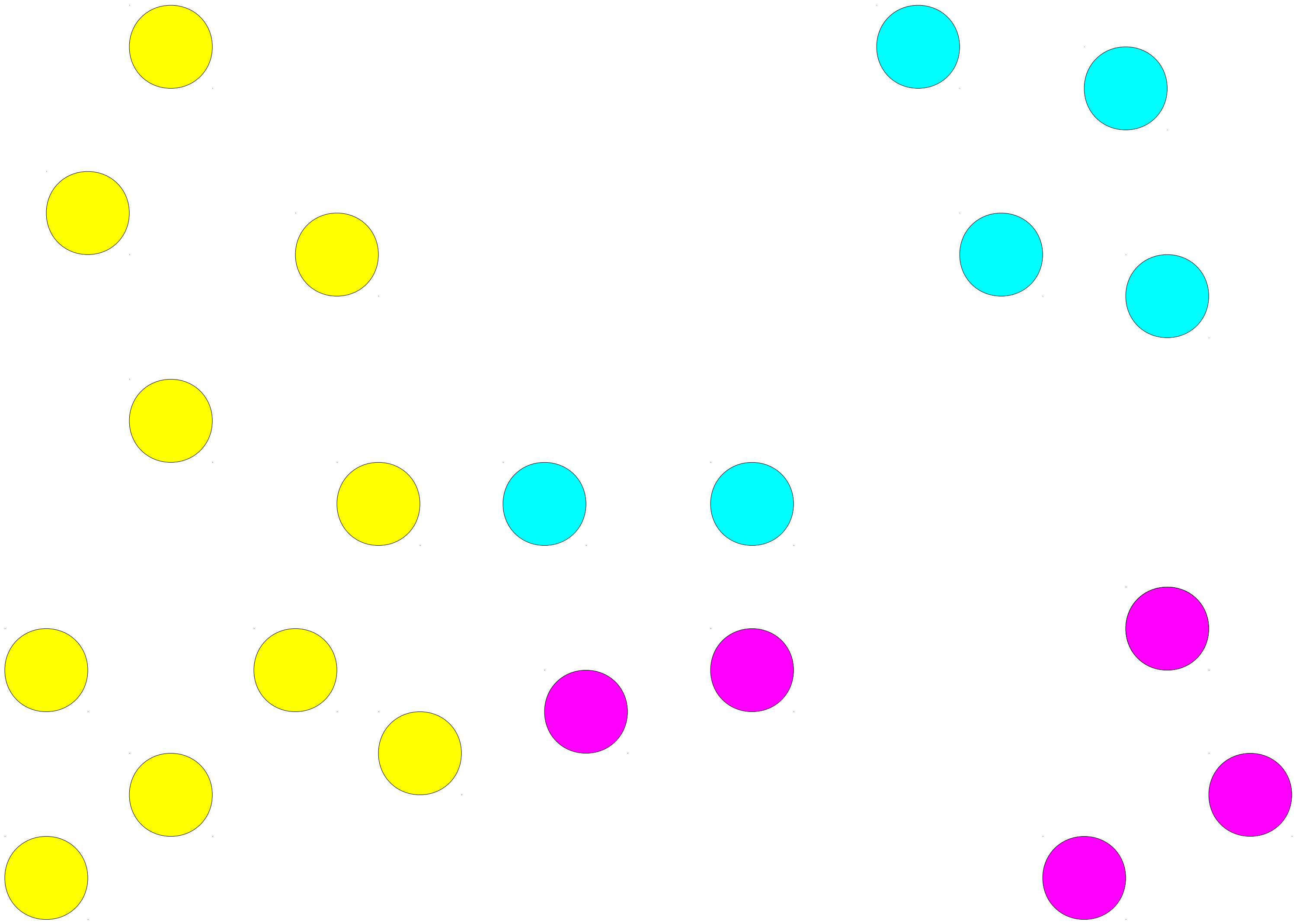
Data Clustering

- Given a set of points, break those points apart into clusters.
- Immensely useful across all disciplines:
    - Cluster individuals by phenotype to try to determine what genes influence which traits.
    - Cluster images by pixel color to identify objects in pictures.
    - Cluster essays by various features to see how students learn to write.

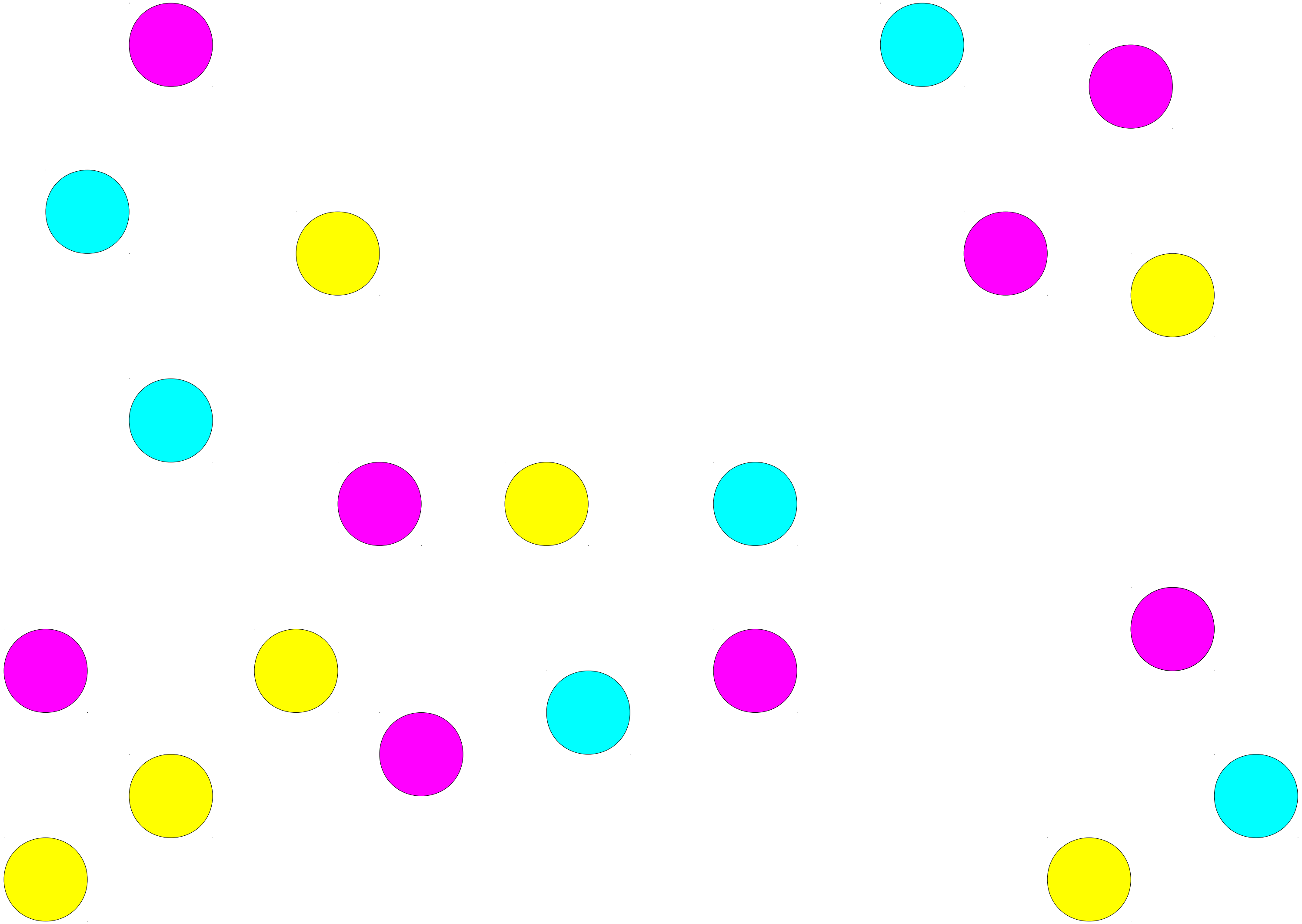
Data Clustering



Data Clustering



Data Clustering

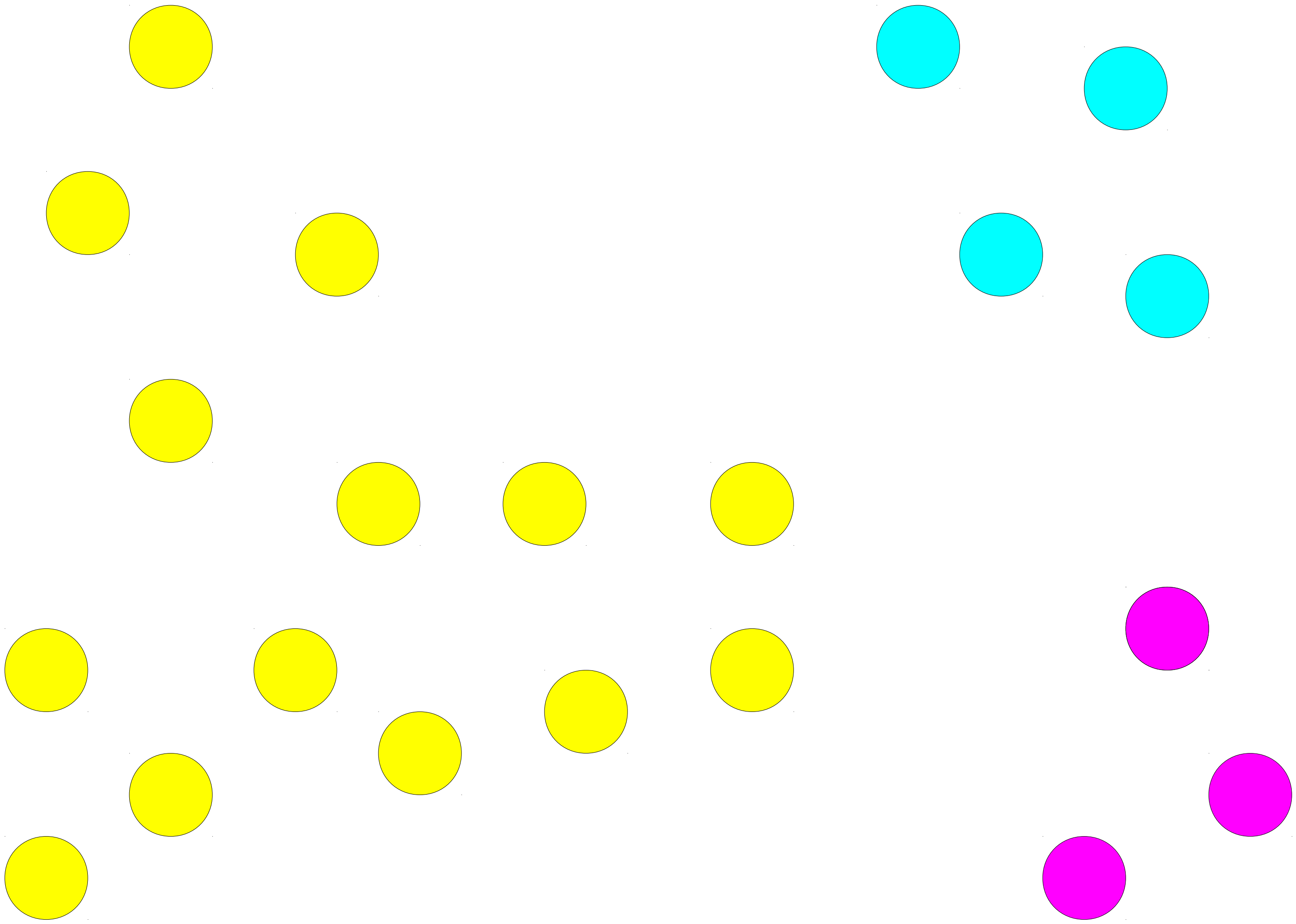


What makes a clustering “good?”

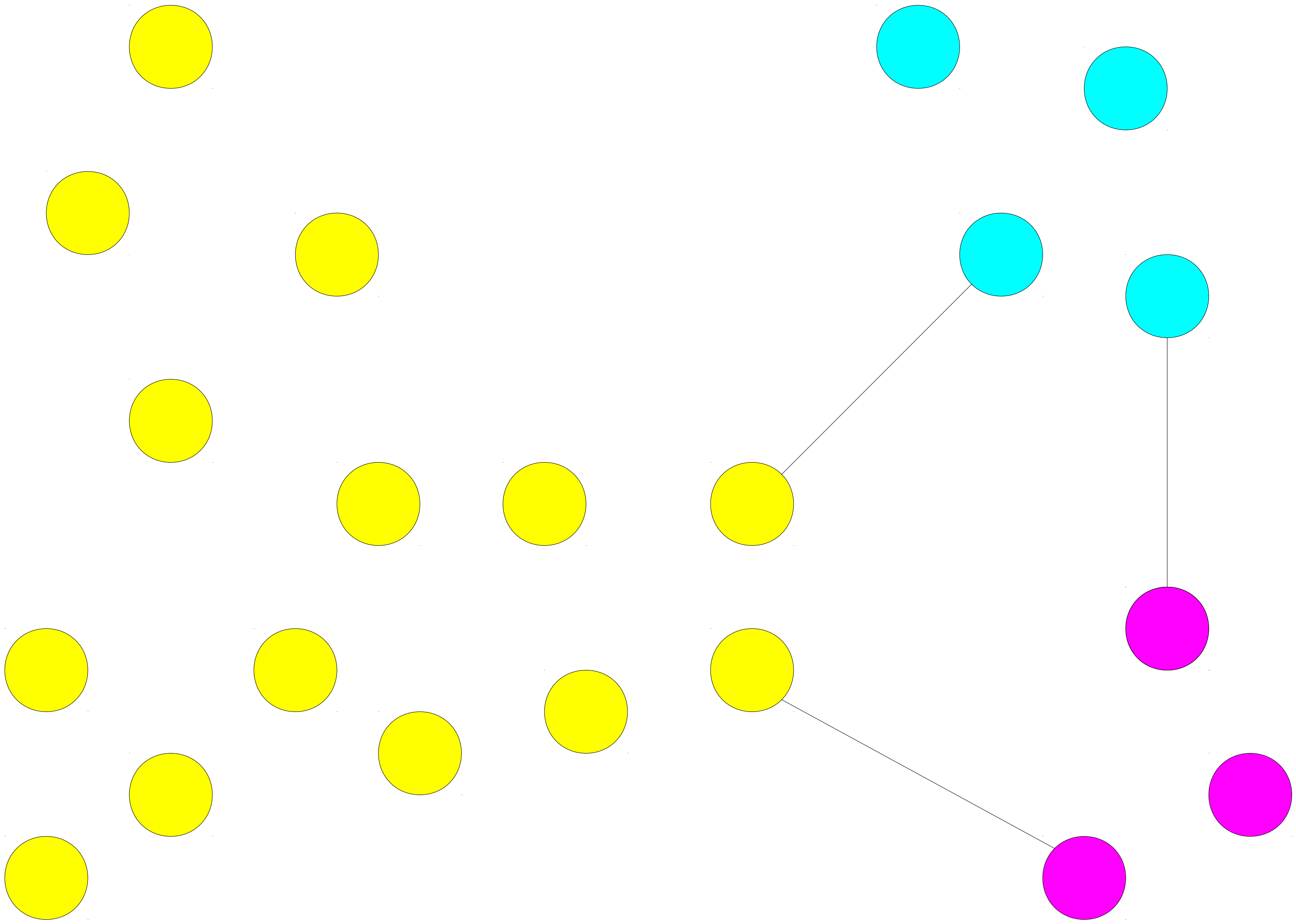
Maximum-Separation Clustering

- **Maximum-separation clustering** tries to find a clustering that maximizes the separation between different clusters.
- Specifically: Maximize the minimum distance between any two points of different clusters.
- Very good on many data sets, though not always ideal.

Maximum-Separation Clustering



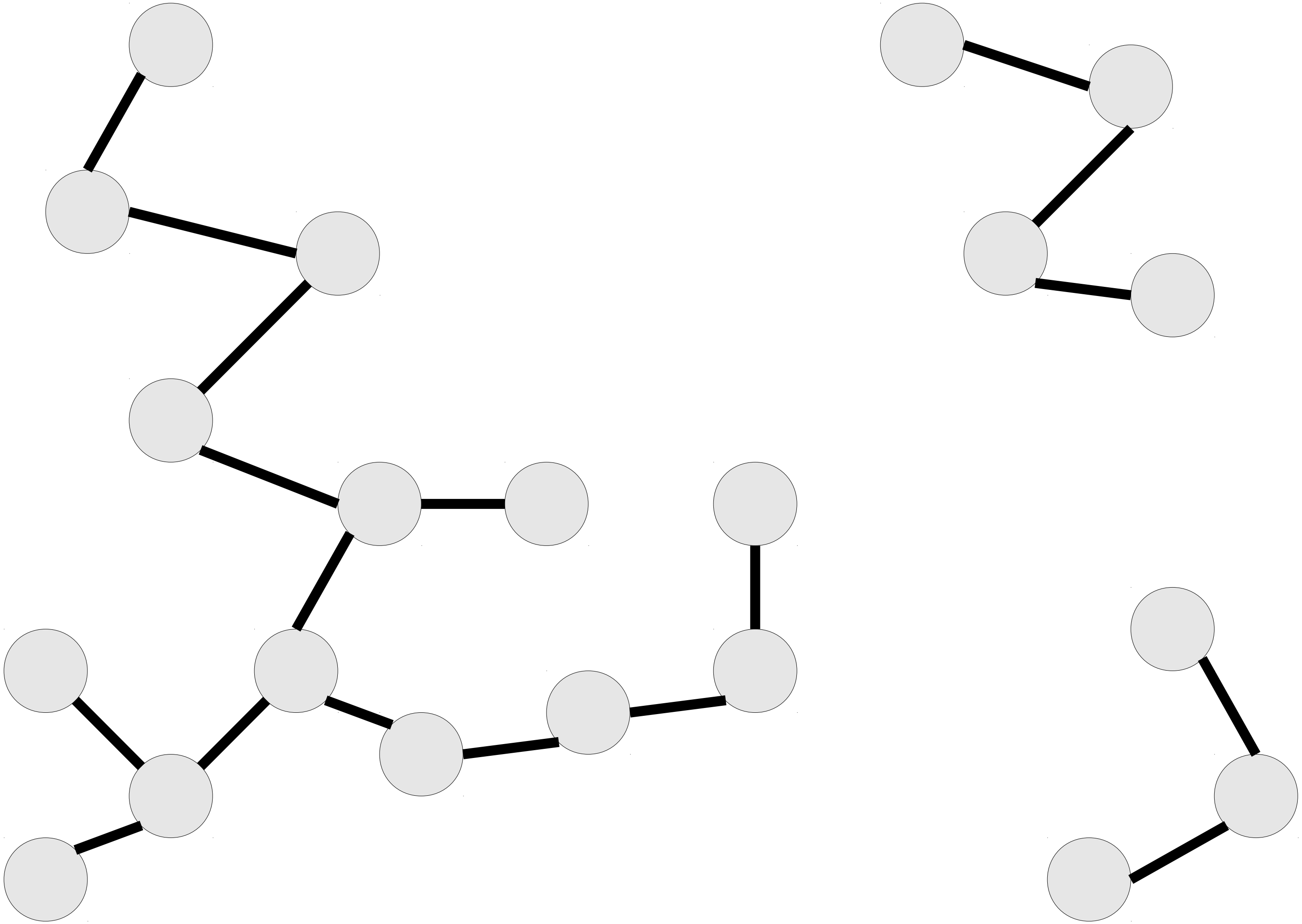
Maximum-Separation Clustering



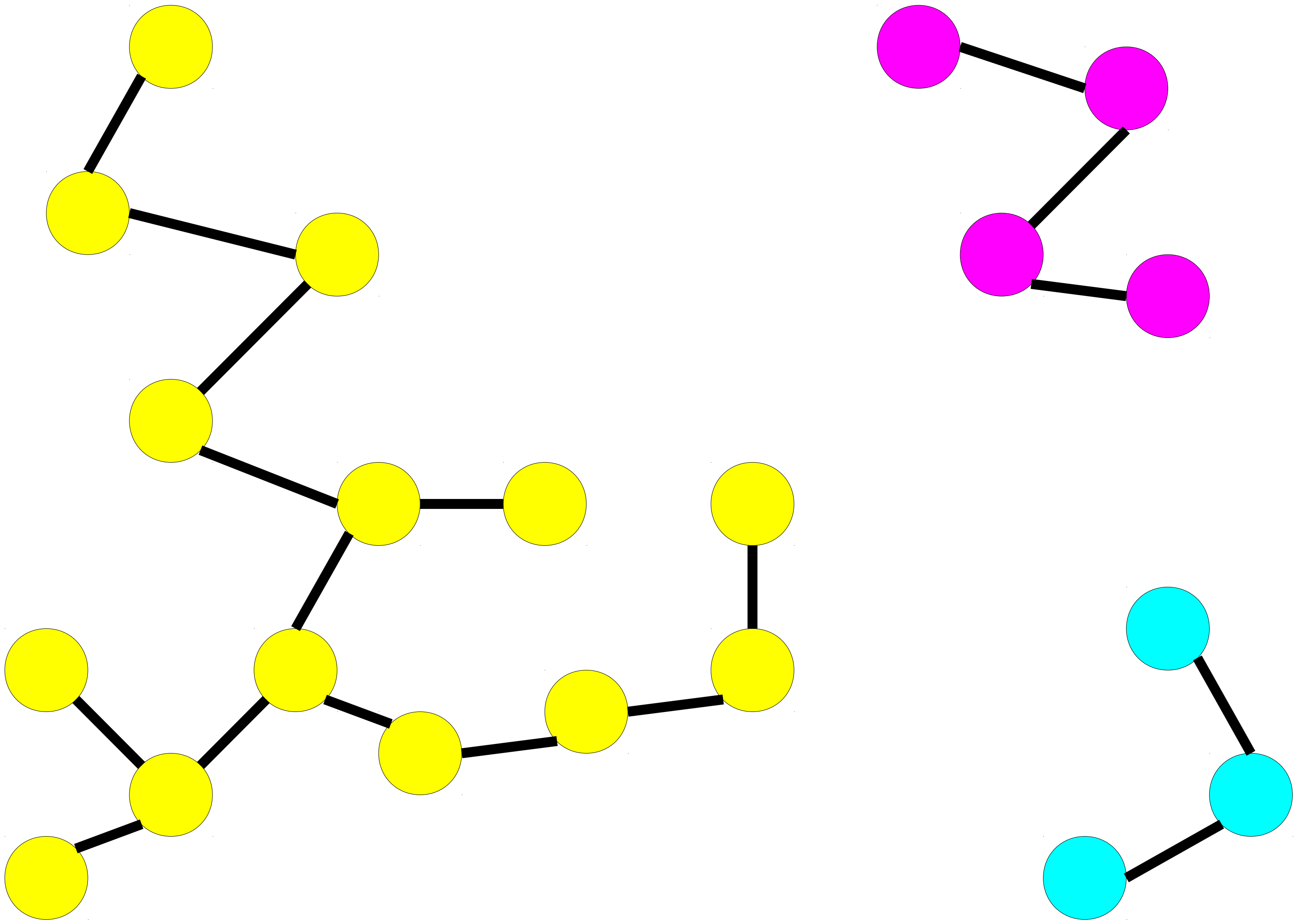
Maximum-Separation Clustering

- It is extremely easy to adopt Kruskal's algorithm to produce a maximum-separation set of clusters.
  - Suppose you want k clusters.
  - Given the data set, add an edge from each node to each other node whose length depends on their similarity.
  - Run Kruskal's algorithm until $n - k$ edges have been added.
  - The pieces of the graph that have been linked together are k maximally-separated clusters.

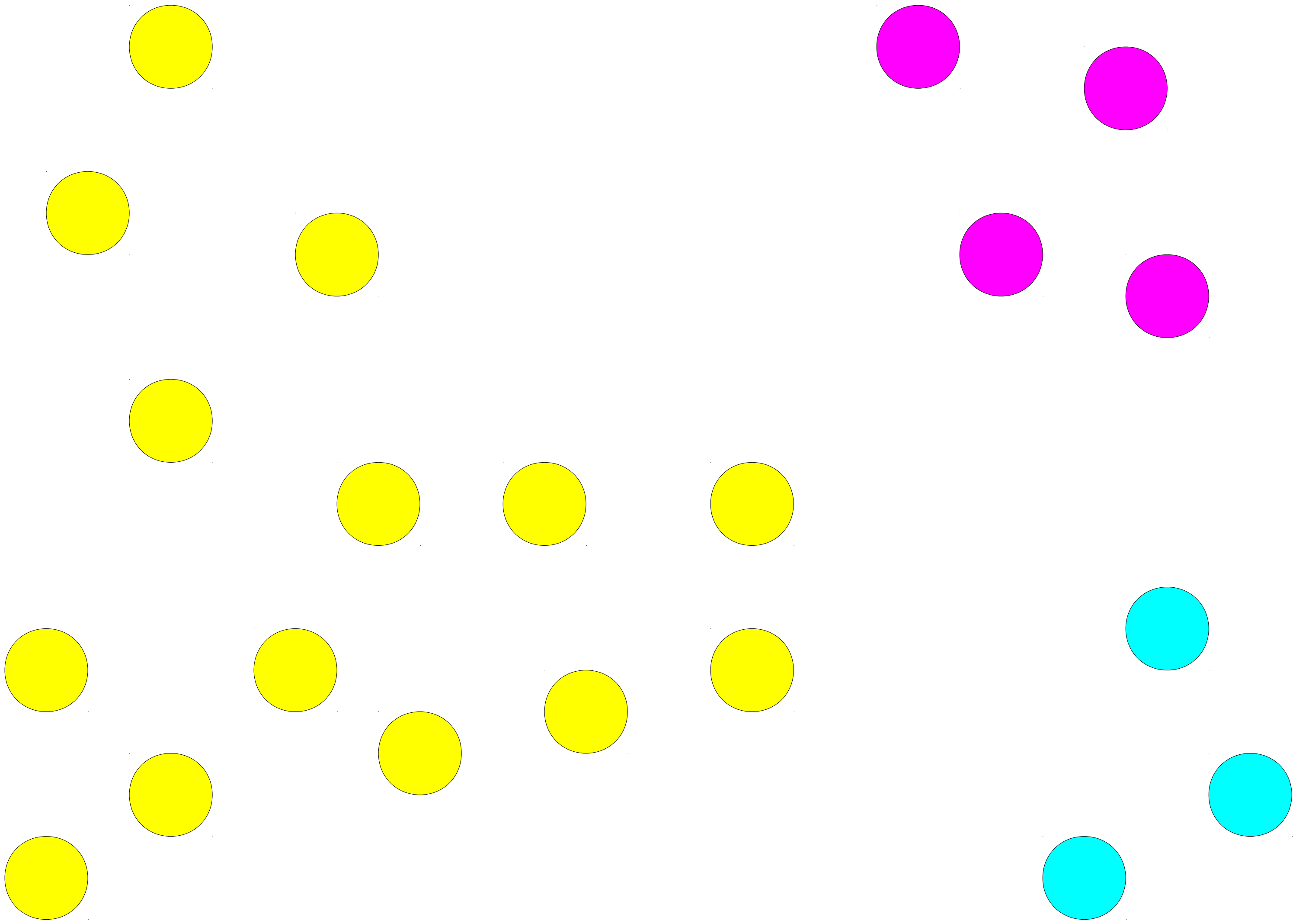
Maximum-Separation Clustering



Maximum-Separation Clustering



Maximum-Separation Clustering



Next Time

- **Fun and Exciting Extra Topics**
  - Machine learning?
  - Advanced graph algorithms?
  - Applications?