Fundamental Graph Algorithms

Part One

Announcements

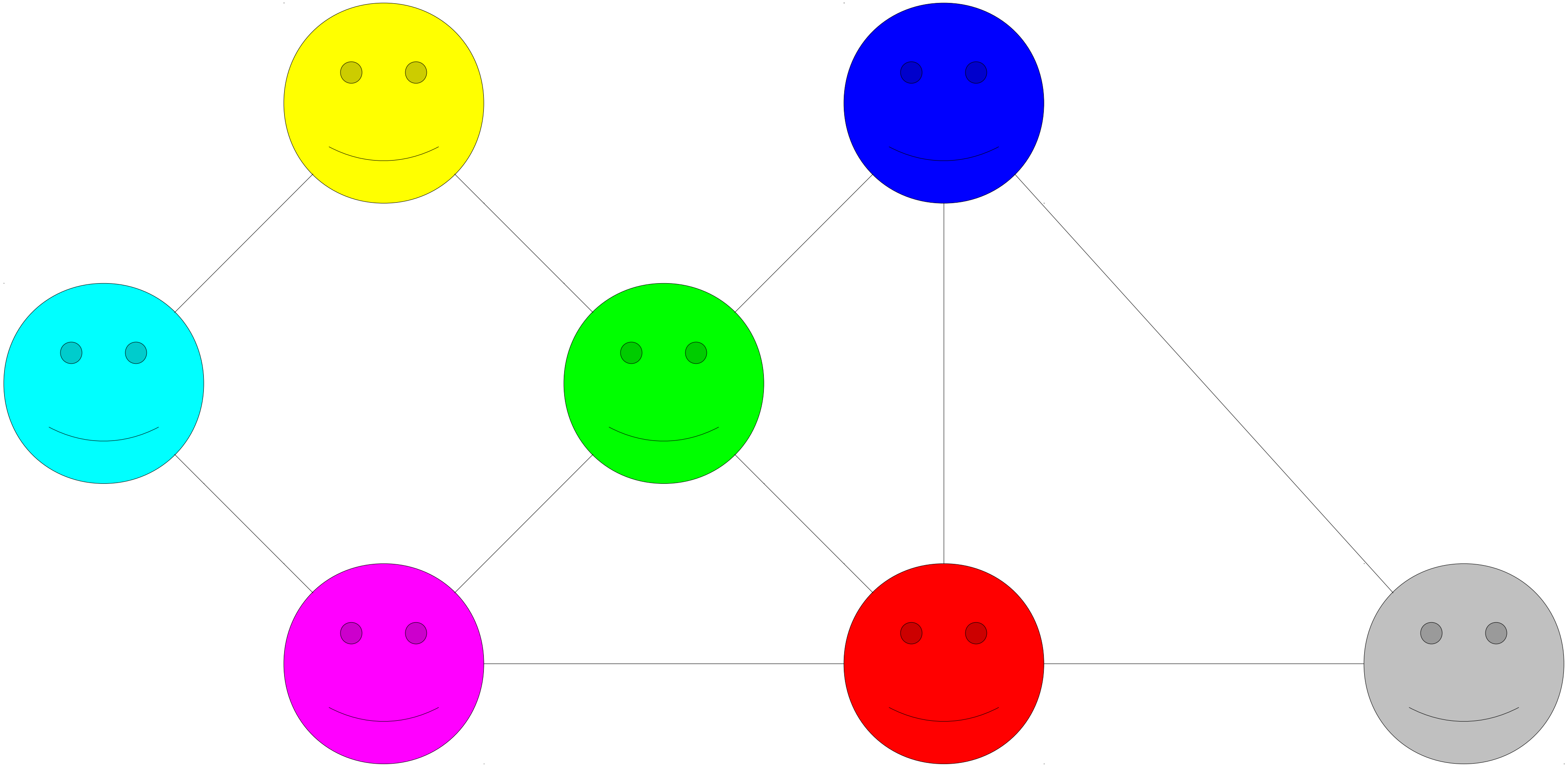
- Problem Set One out, due **Wednesday, July 3**.
  - Play around with O, Ω, and Θ notations!
  - Get your feet wet designing and analyzing algorithms.
  - Explore today's material on graphs.
- Can be completed using just material from the first two lectures.
- We suggest reading through the handout on how to approach the problem sets. There's a lot of useful information there!
- Office hours schedule will be announced tomorrow.

Announcements

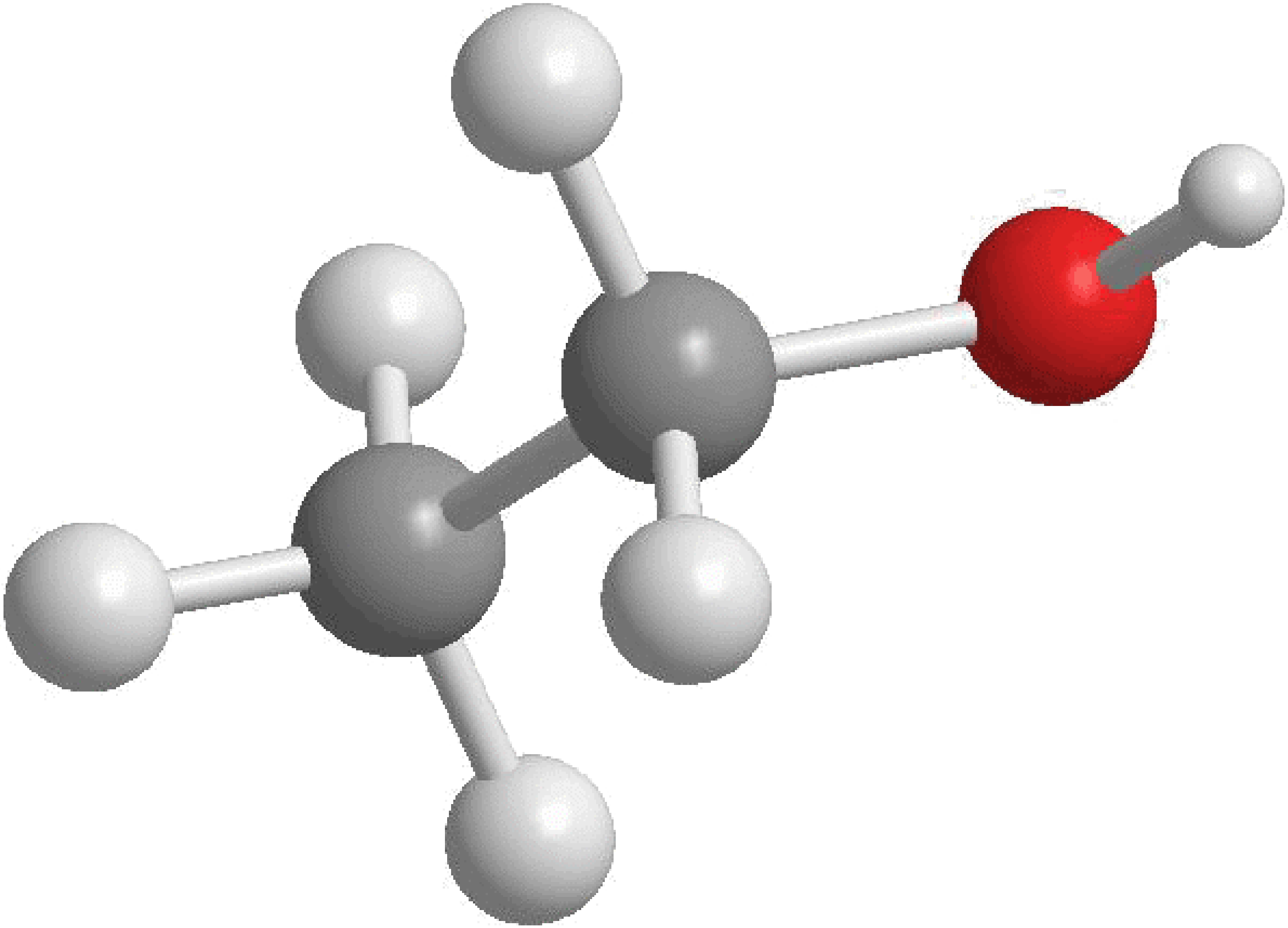
- We will not be writing any code in CS161; we'll focus more on the design and analysis techniques.
- Each week, we will have an optional programming section where you can practice coding up these algorithms.
- Run by TA Andy Nguyen, who coaches Stanford's ACM programming team.
- Meets **Thursdays, 4:15PM – 5:05PM** in **Gates B08**.

Graphs

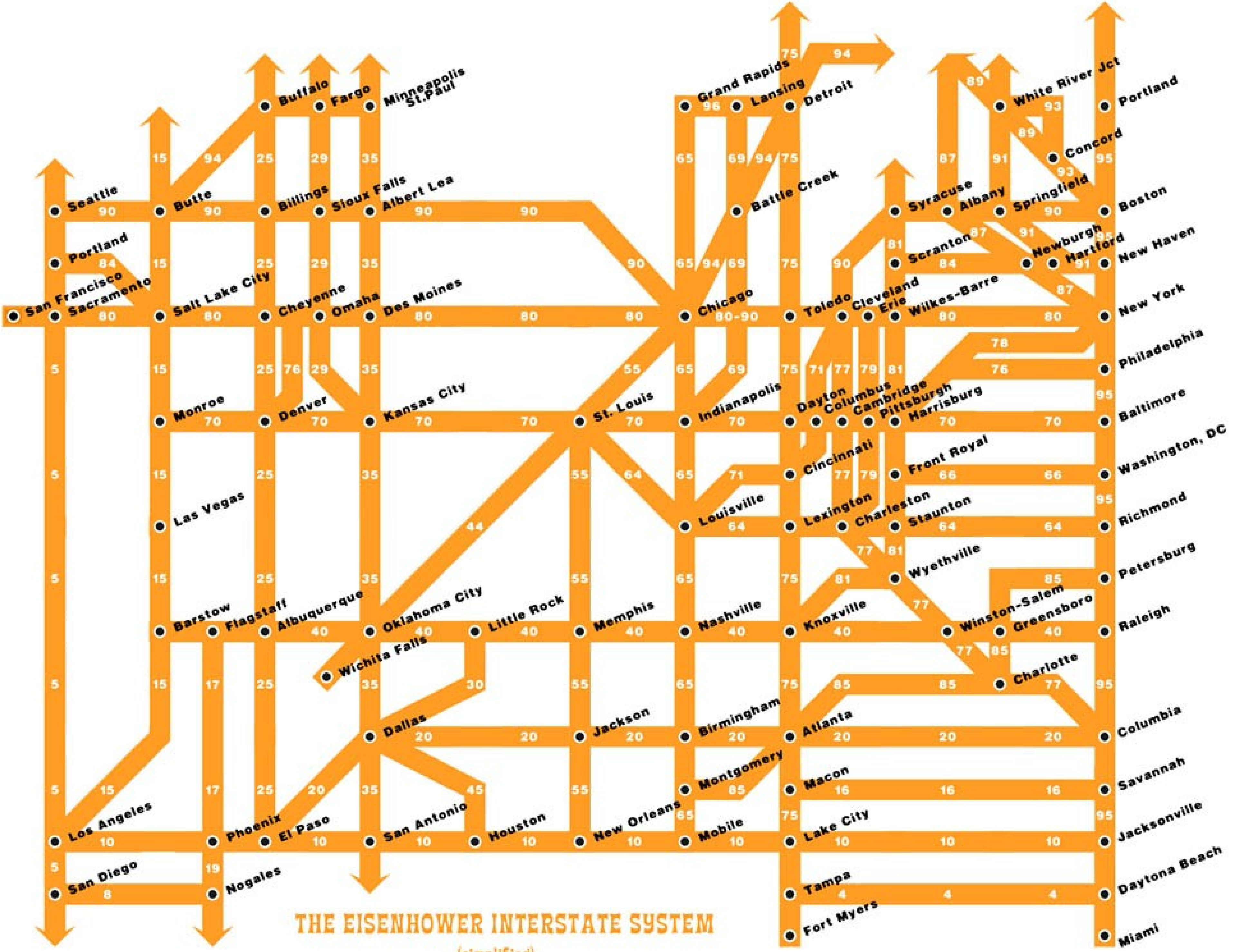
A Social Network



Chemical Bonds



http://4.bp.blogspot.com/-xCtBJ8lKHqA/Tjm0BONWBRI/AAAAAAAAAK4/-mHrbAUOHHg/s1600/Etha

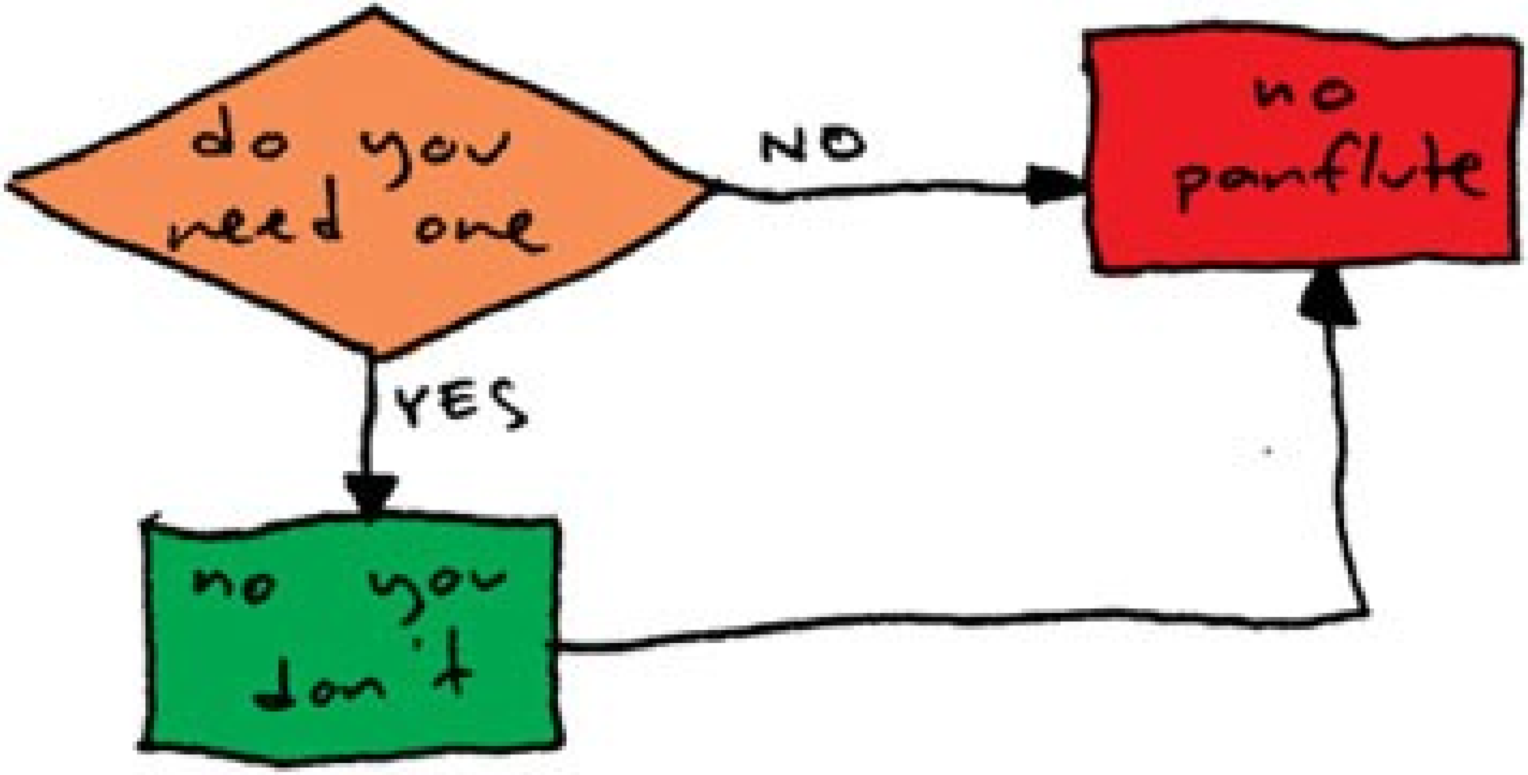


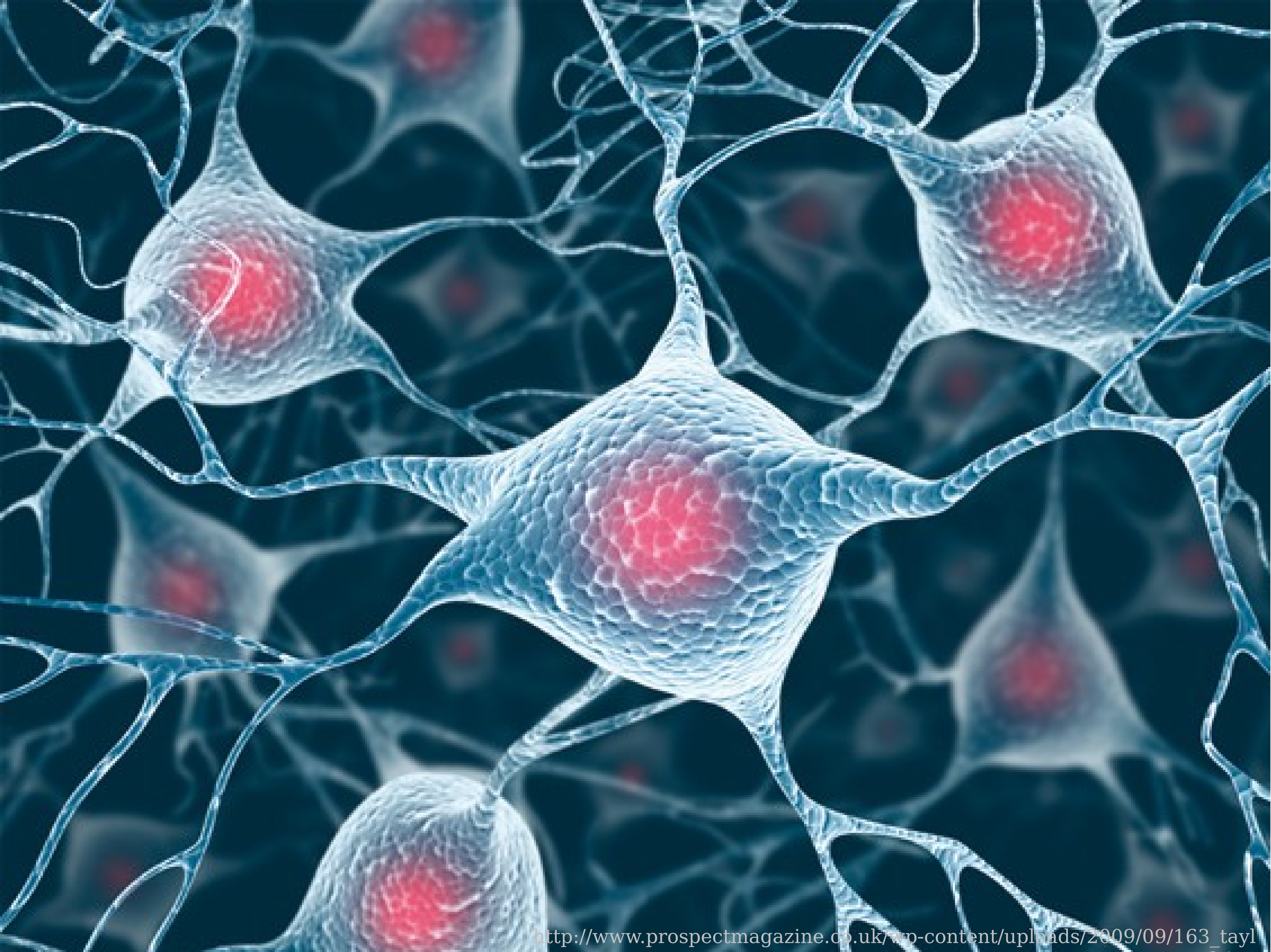
THE EISENHOWER INTERSTATE SYSTEM

(simplified)

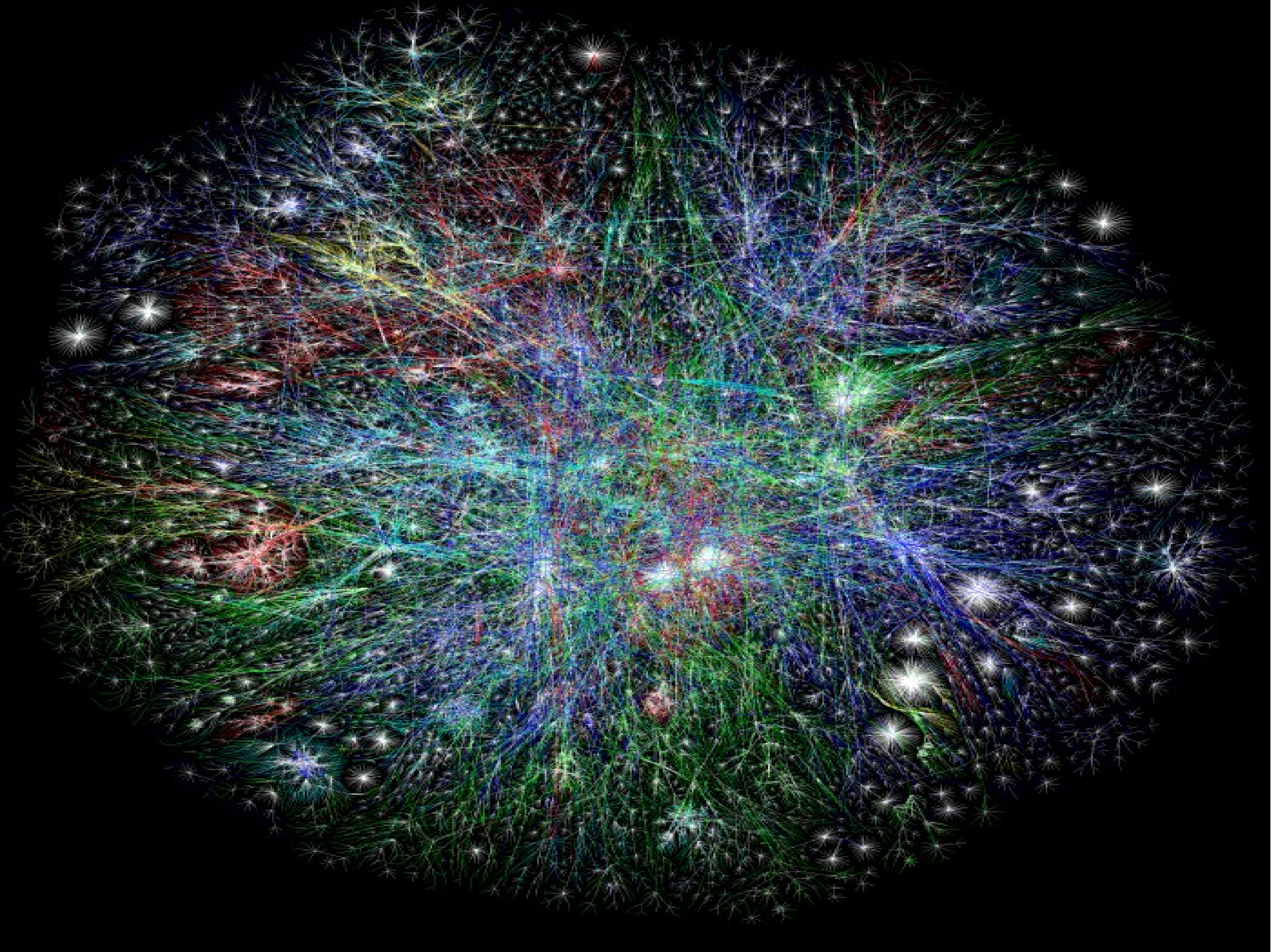
CHRIS YATES 2007

PANFLUTE FLOWCHART





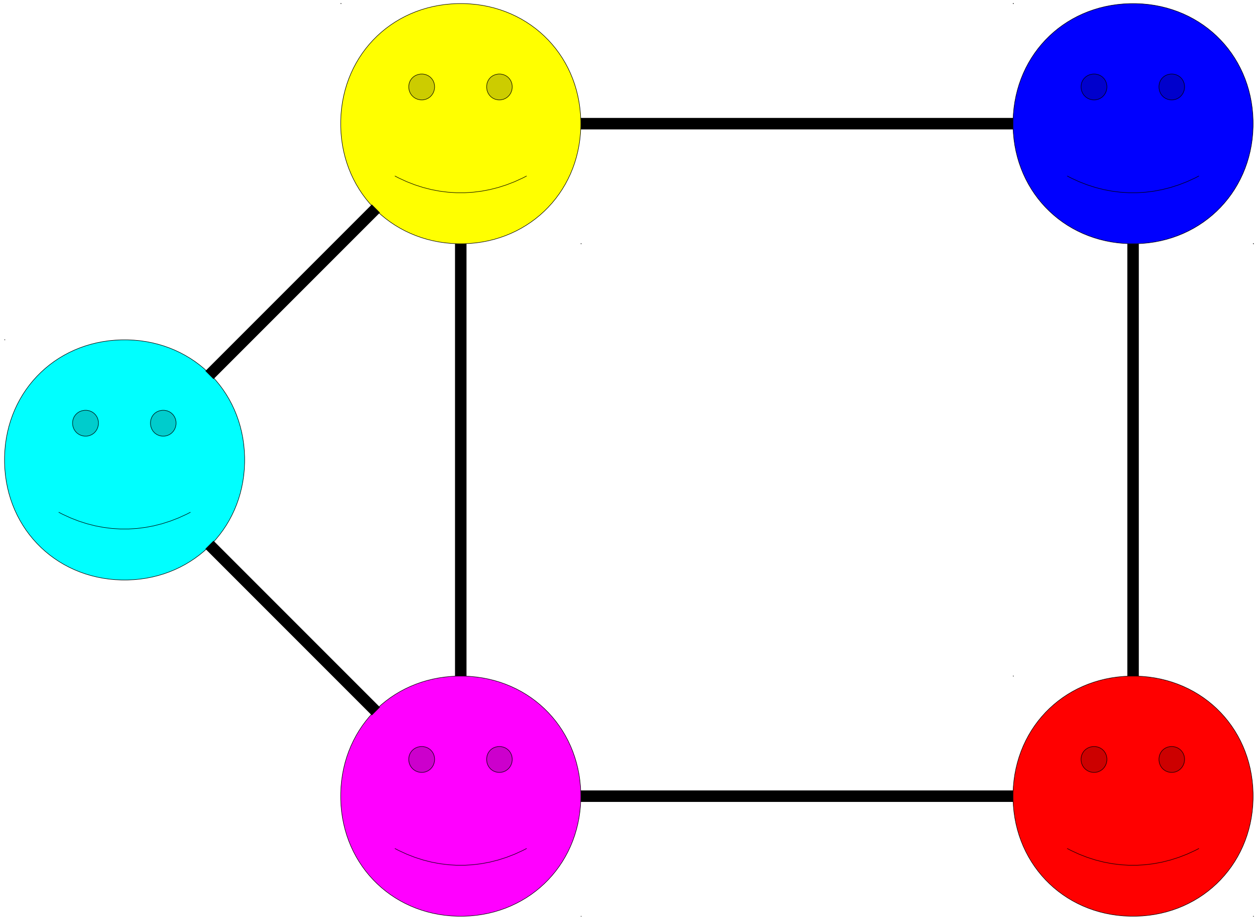
http://www.prospectmagazine.co.uk/wp-content/uploads/2009/09/163_tayl



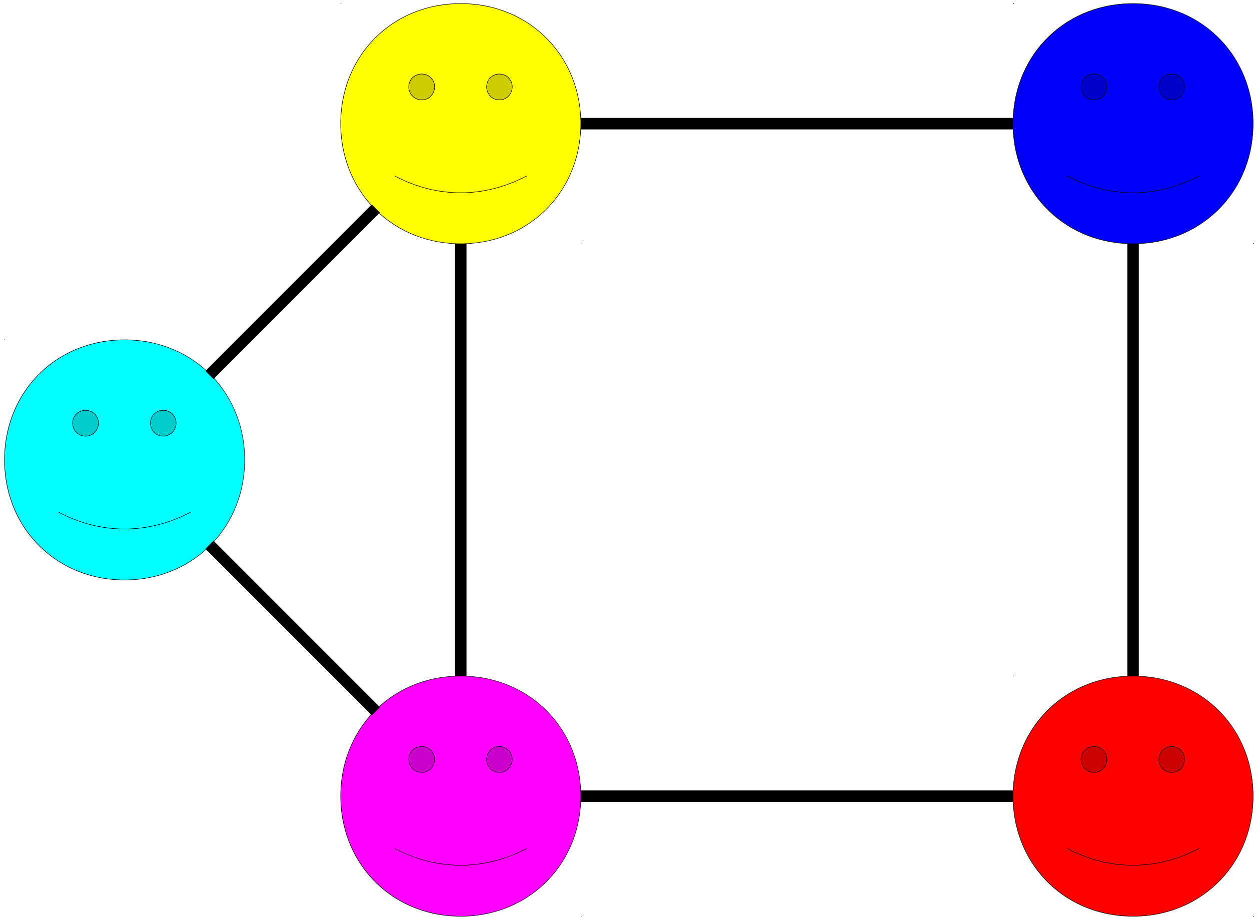
A **graph** is a mathematical structure for representing relationships.

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

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

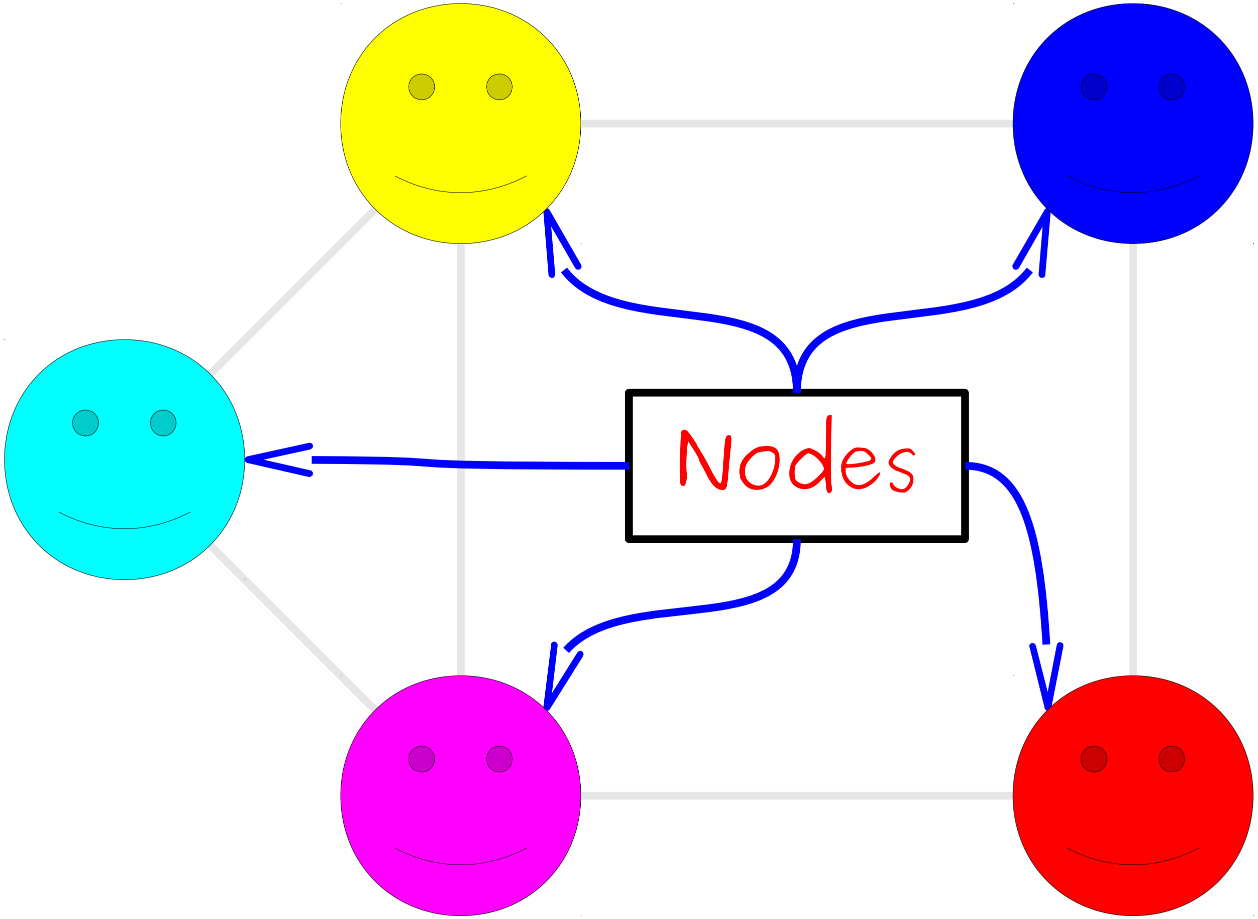


A **graph** is a mathematical structure for representing relationships.



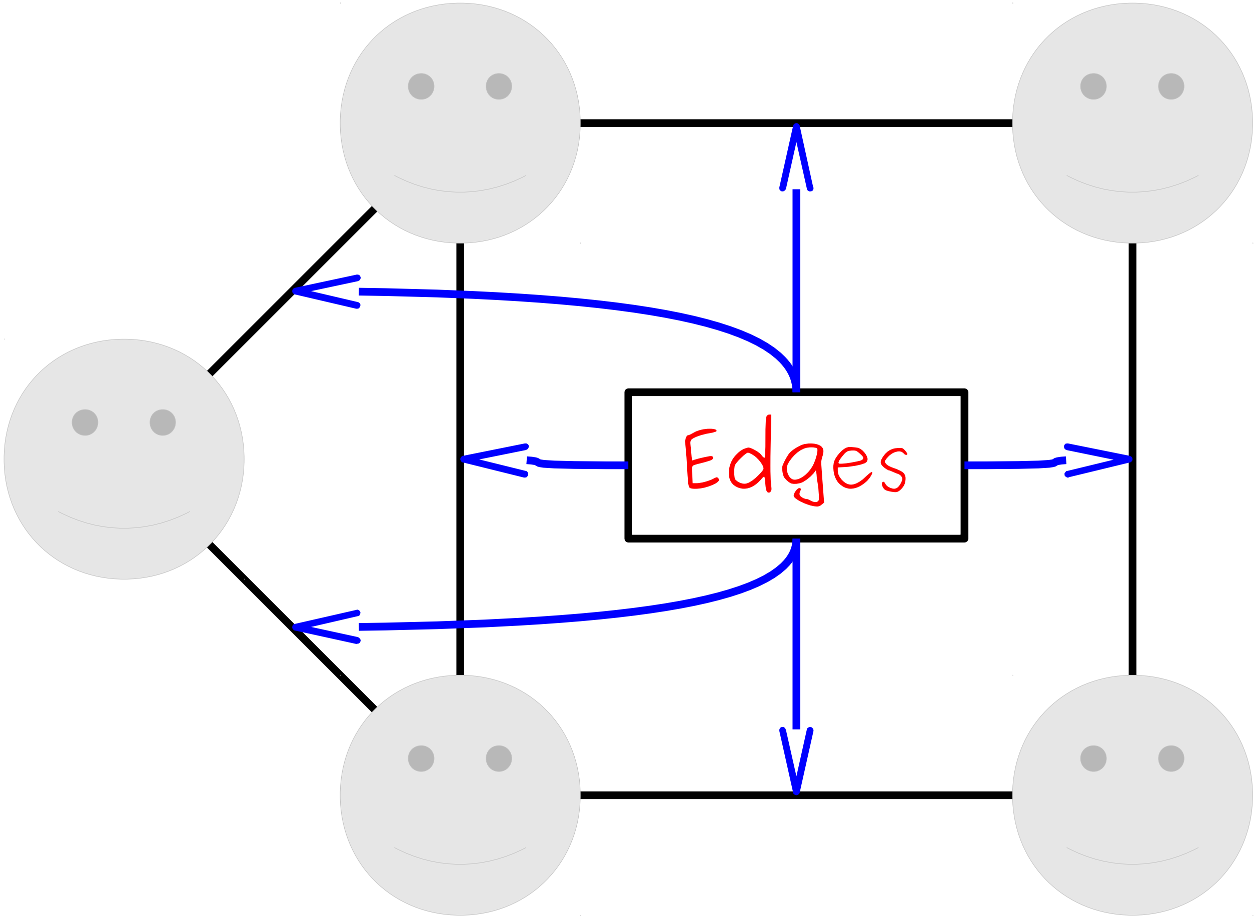
A graph consists of a set of **nodes** connected by **edges**.

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



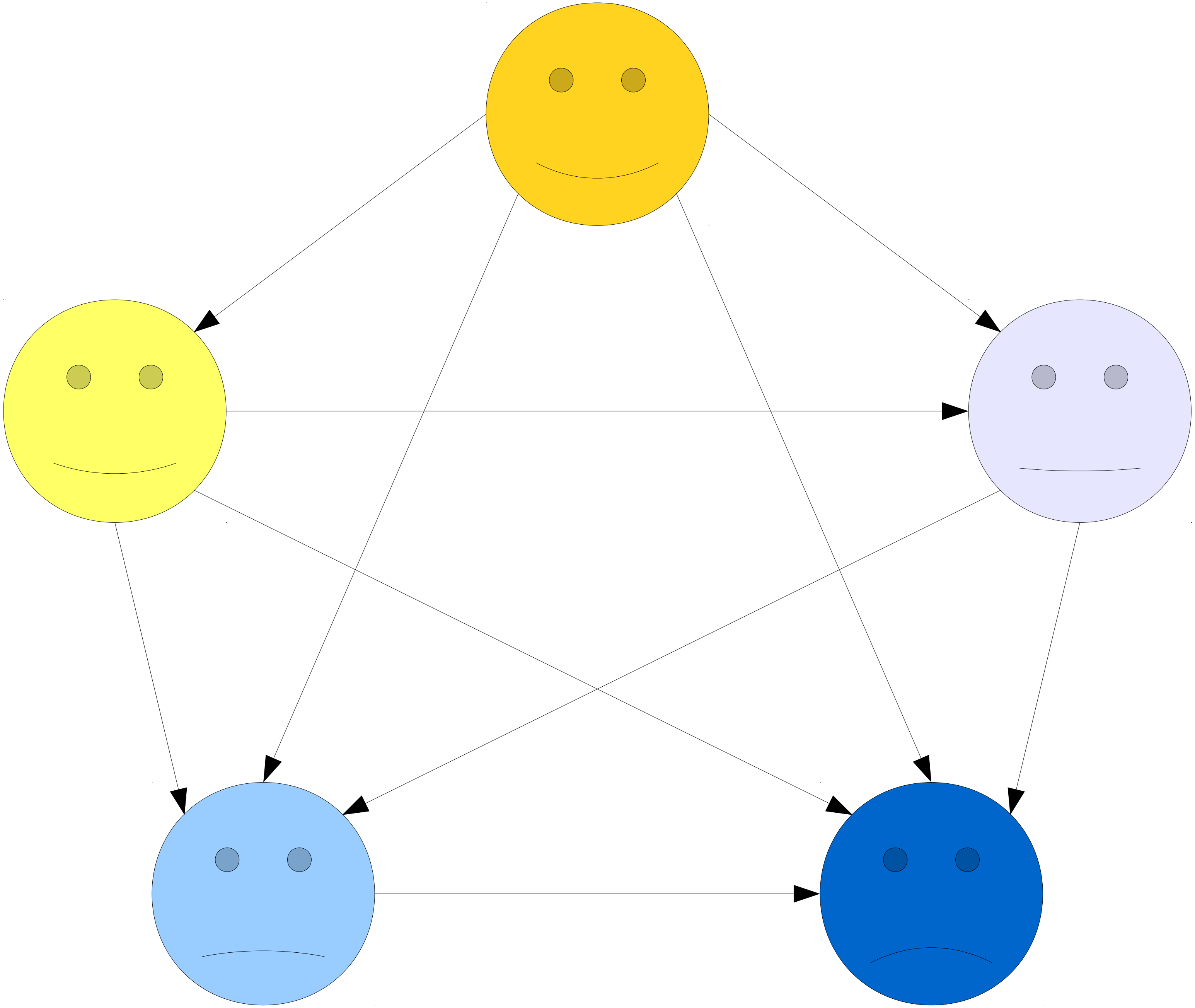
A graph consists of a set of **nodes** connected by **edges**.

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

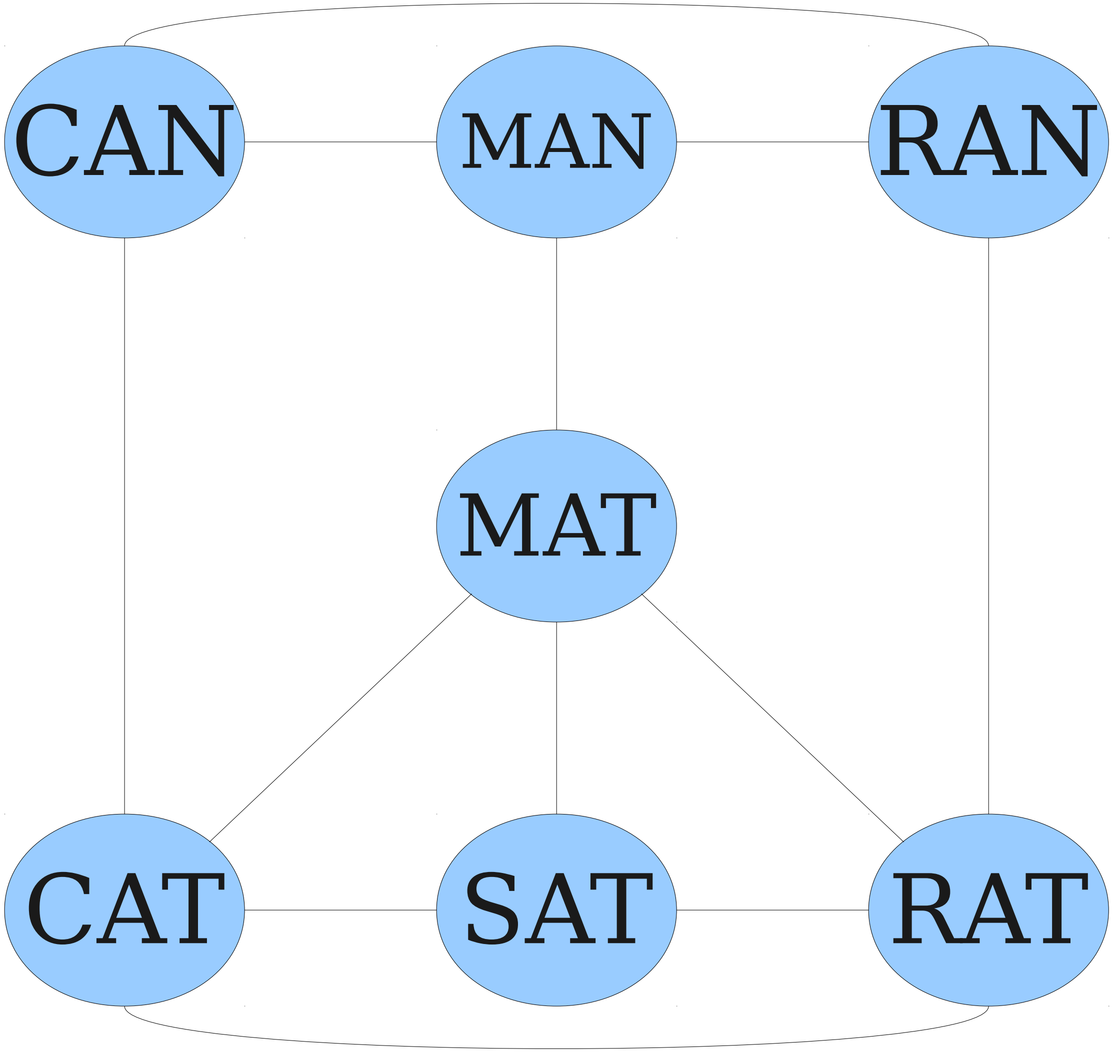


A graph consists of a set of **nodes** connected by **edges**.

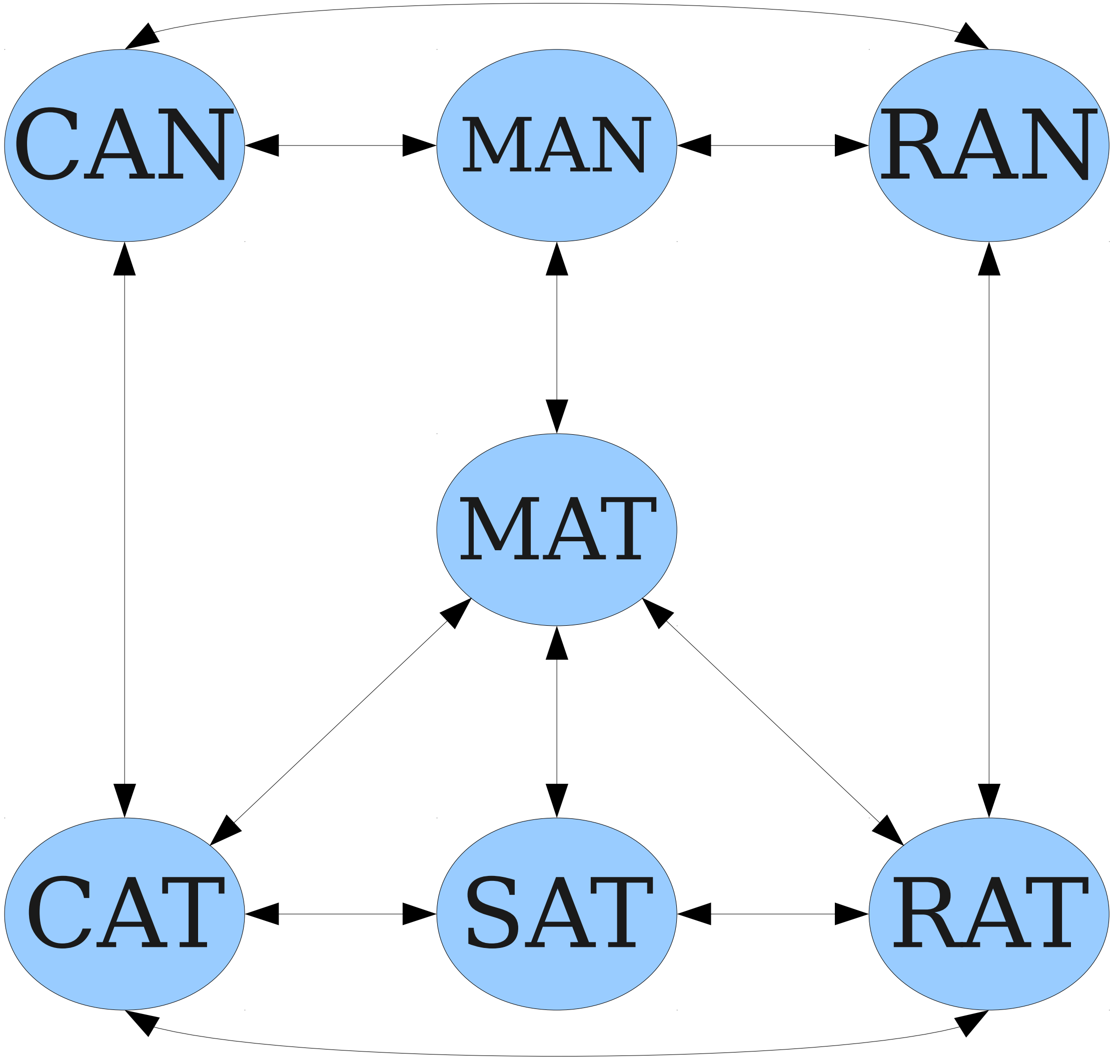
Some graphs are **directed**.



Some graphs are **undirected**.



Some graphs are **undirected**.



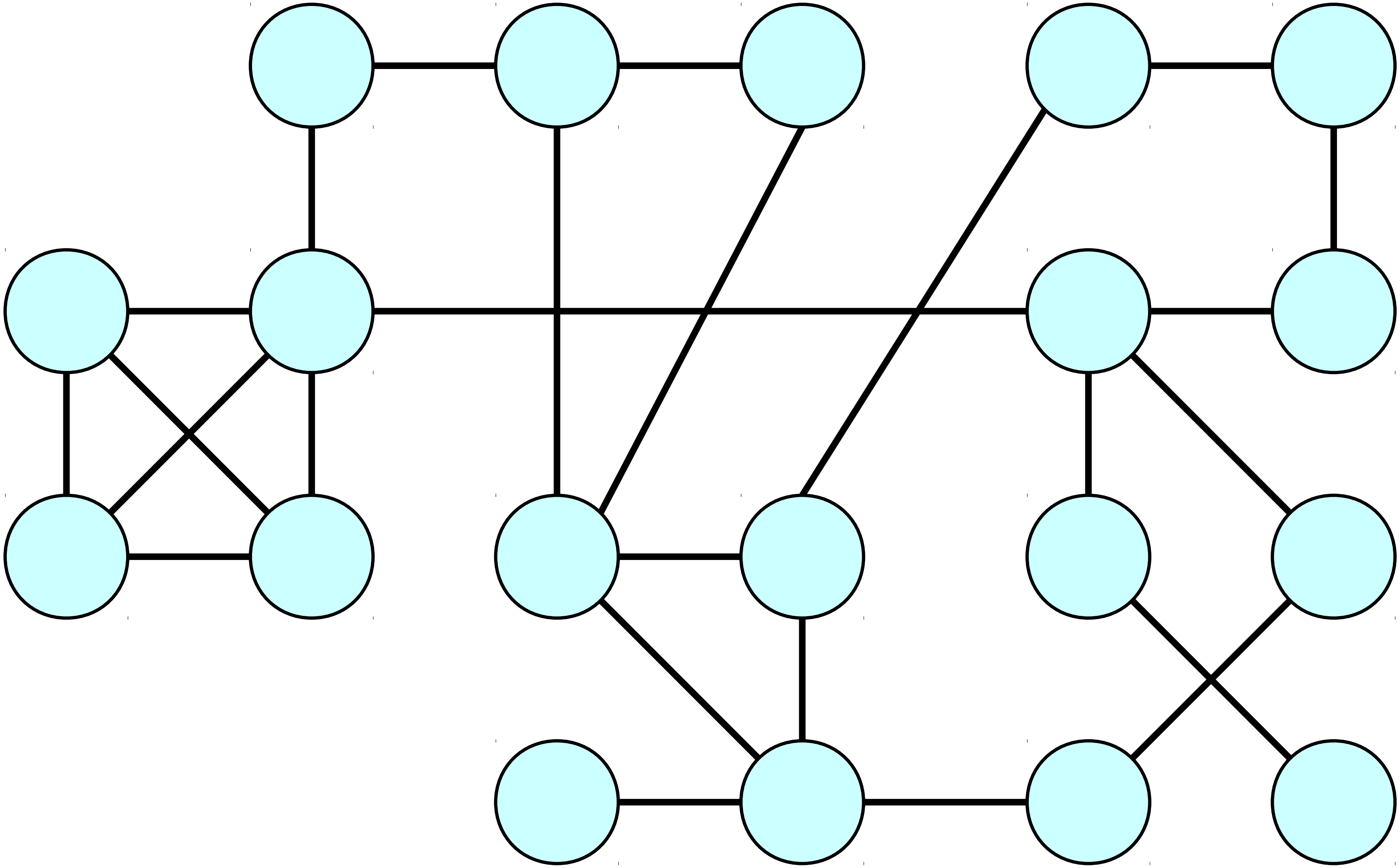
You can think of them as directed graphs with edges both ways.

Formalisms

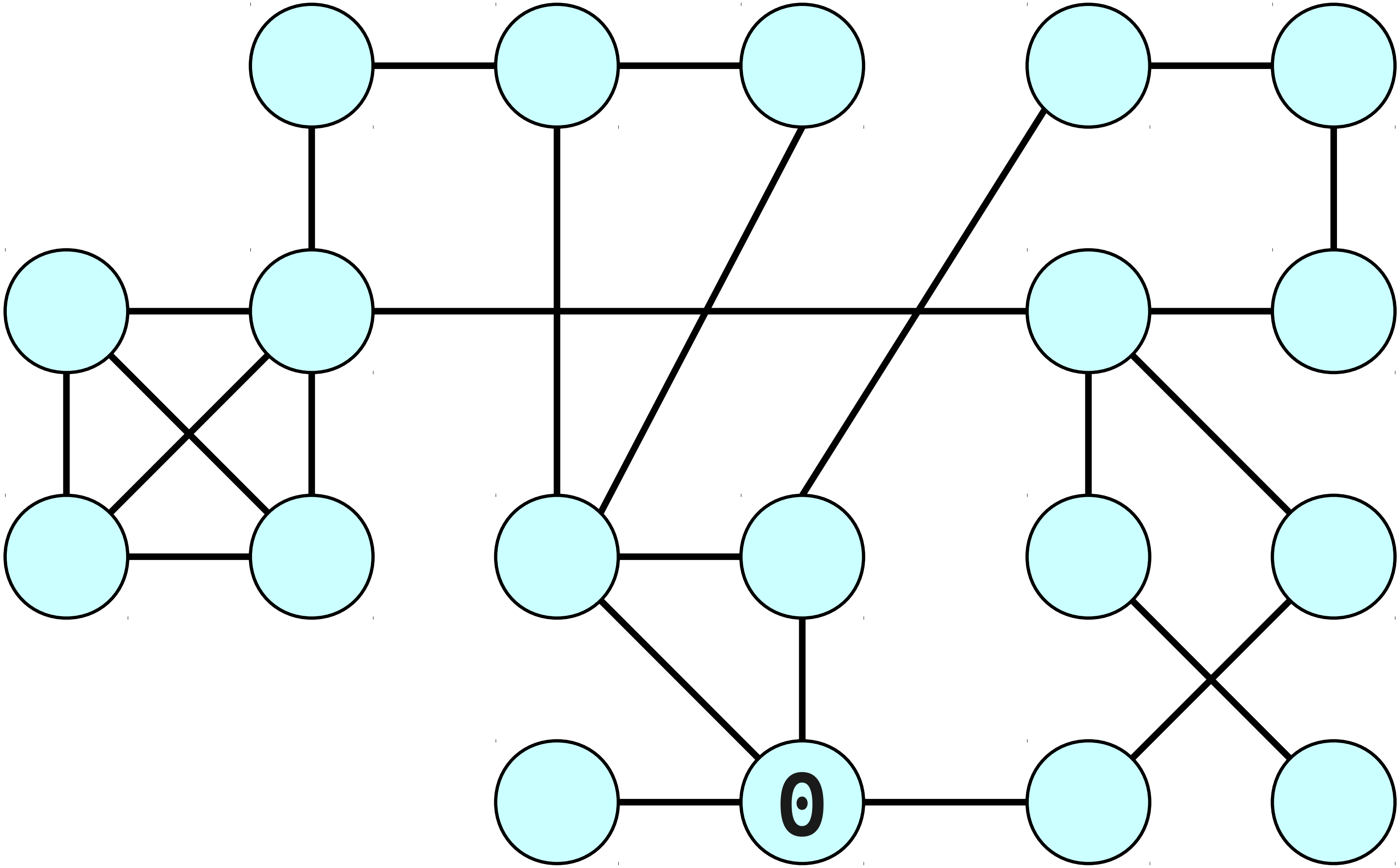
- A **graph** is an ordered pair $G = (V, E)$ where
  - V is a set of the **vertices** (nodes) of the graph.
  - E is a set of the **edges** (arcs) of the graph.
- E can be a set of ordered pairs or unordered pairs.
  - If E consists of ordered pairs, G is **directed**
  - If E consists of unordered pairs, G is **undirected**.
- In an *undirected* graph, the **degree** of node v (denoted **deg(v)**) is the number of edges incident to v.
- In a *directed* graph, the **indegree** of a node v (denoted **deg^{-}(v)**) is the number of edges entering v and the **outdegree** of a node v (denoted (**deg^{+}(v)**) is the number of edges leaving v.

An Application: Six Degrees of Separation

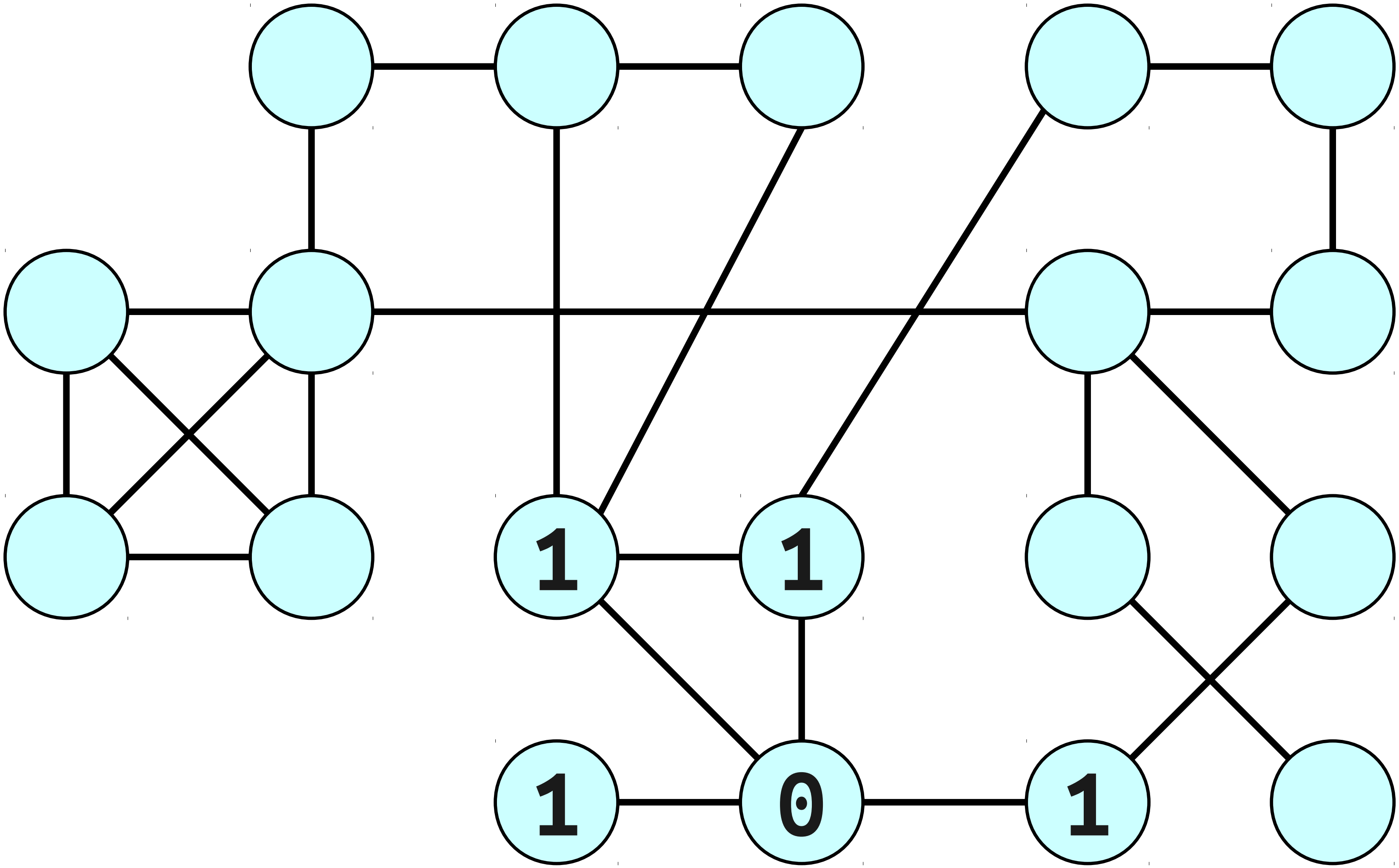
A Social Network



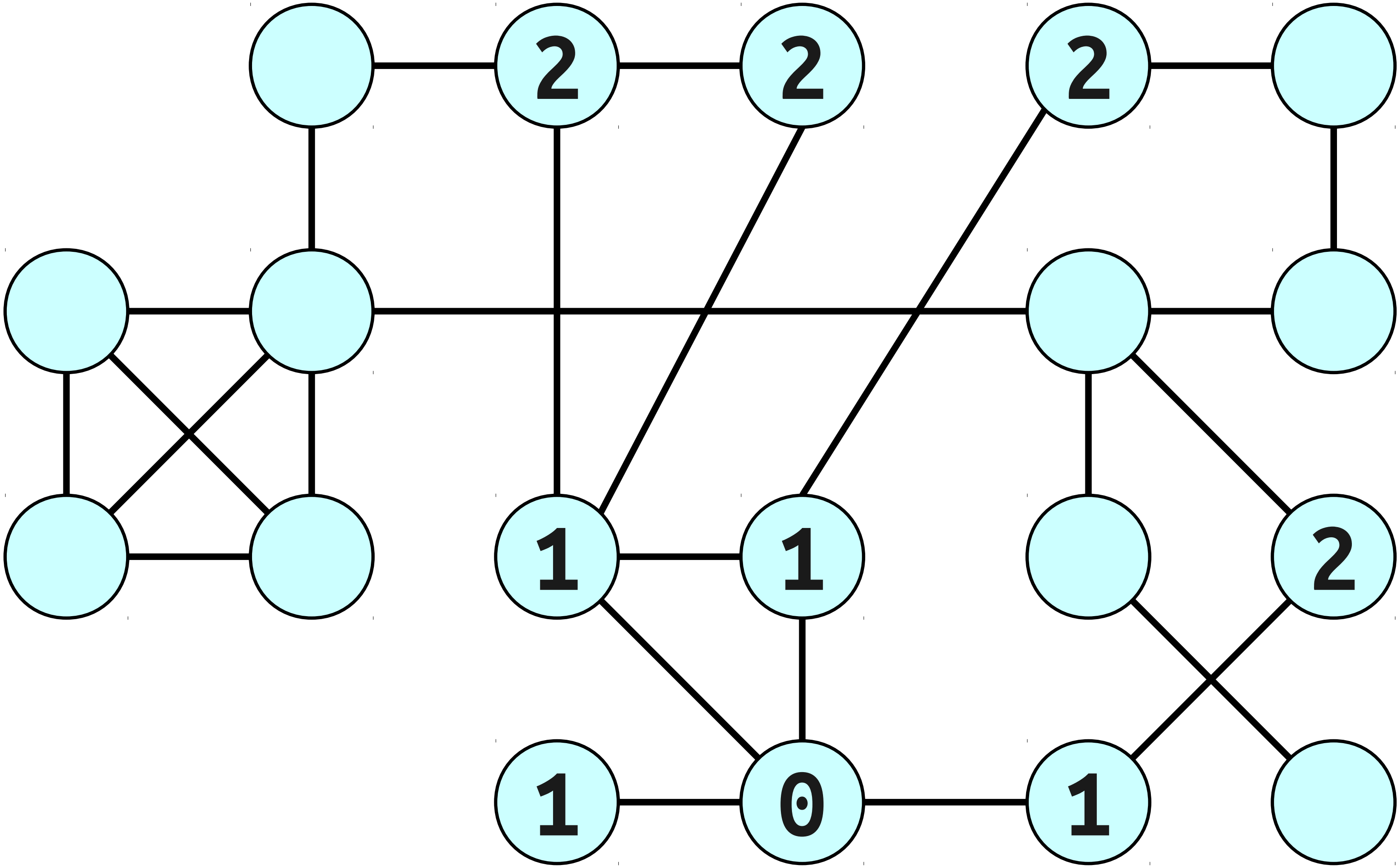
A Social Network



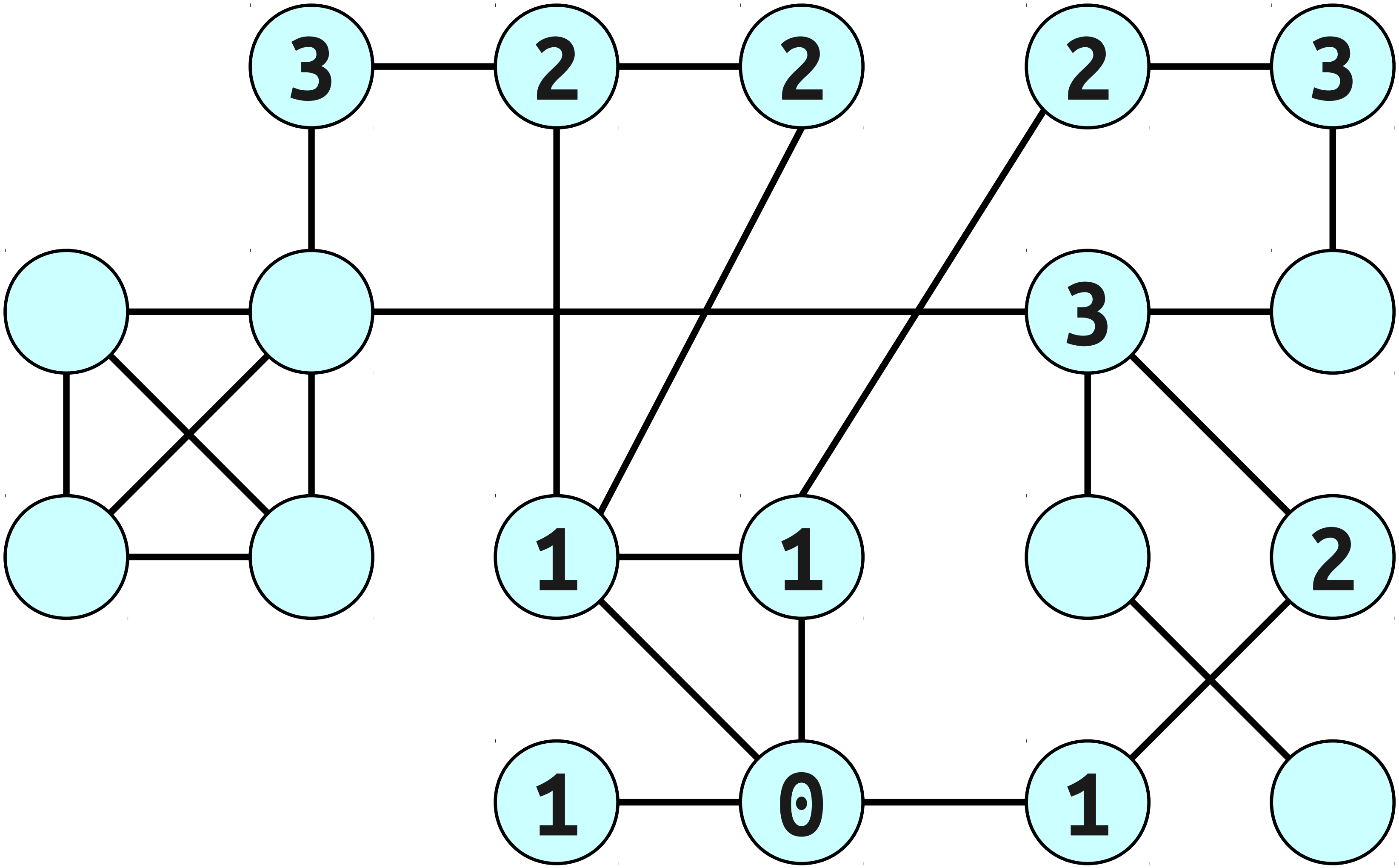
A Social Network



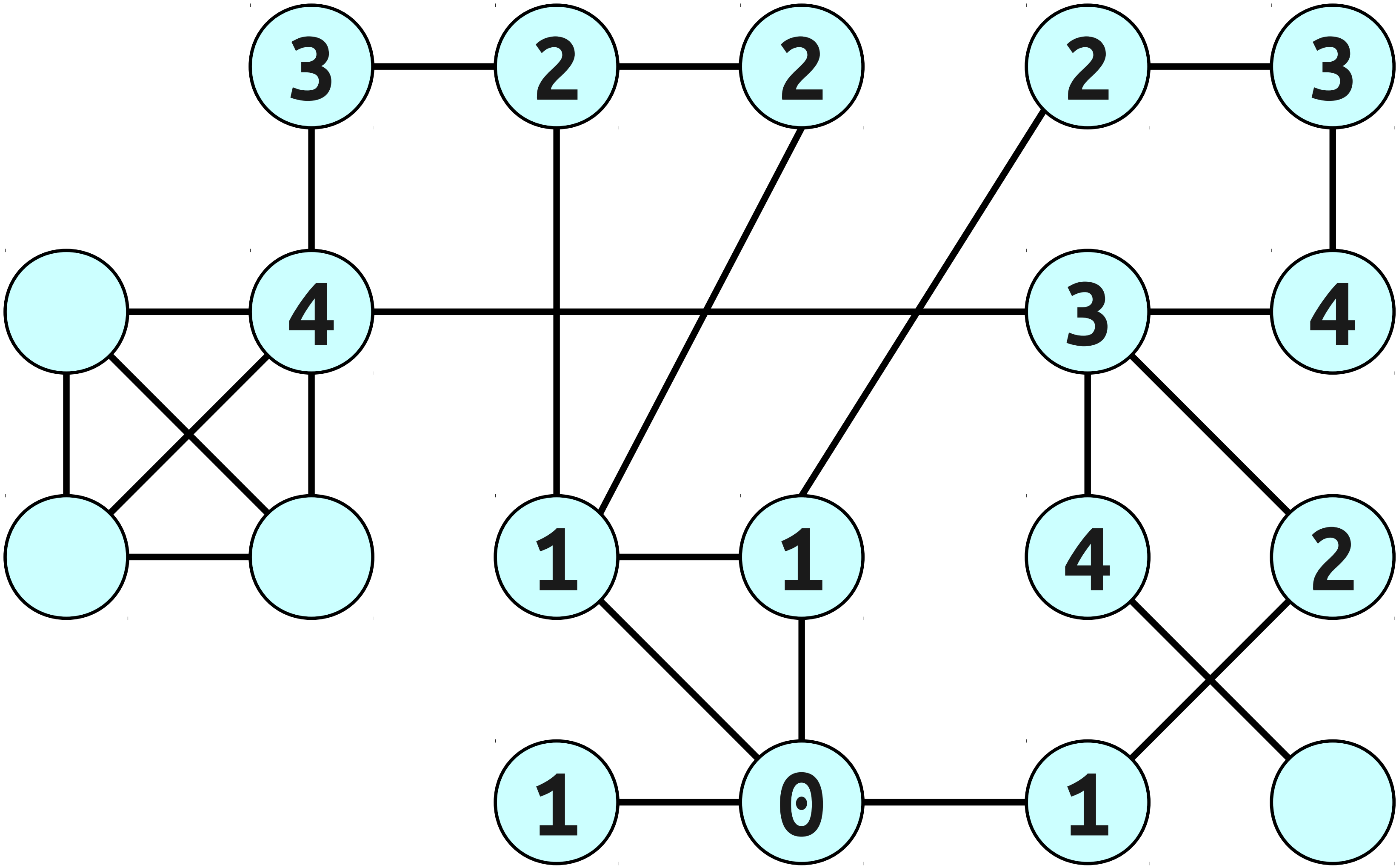
A Social Network



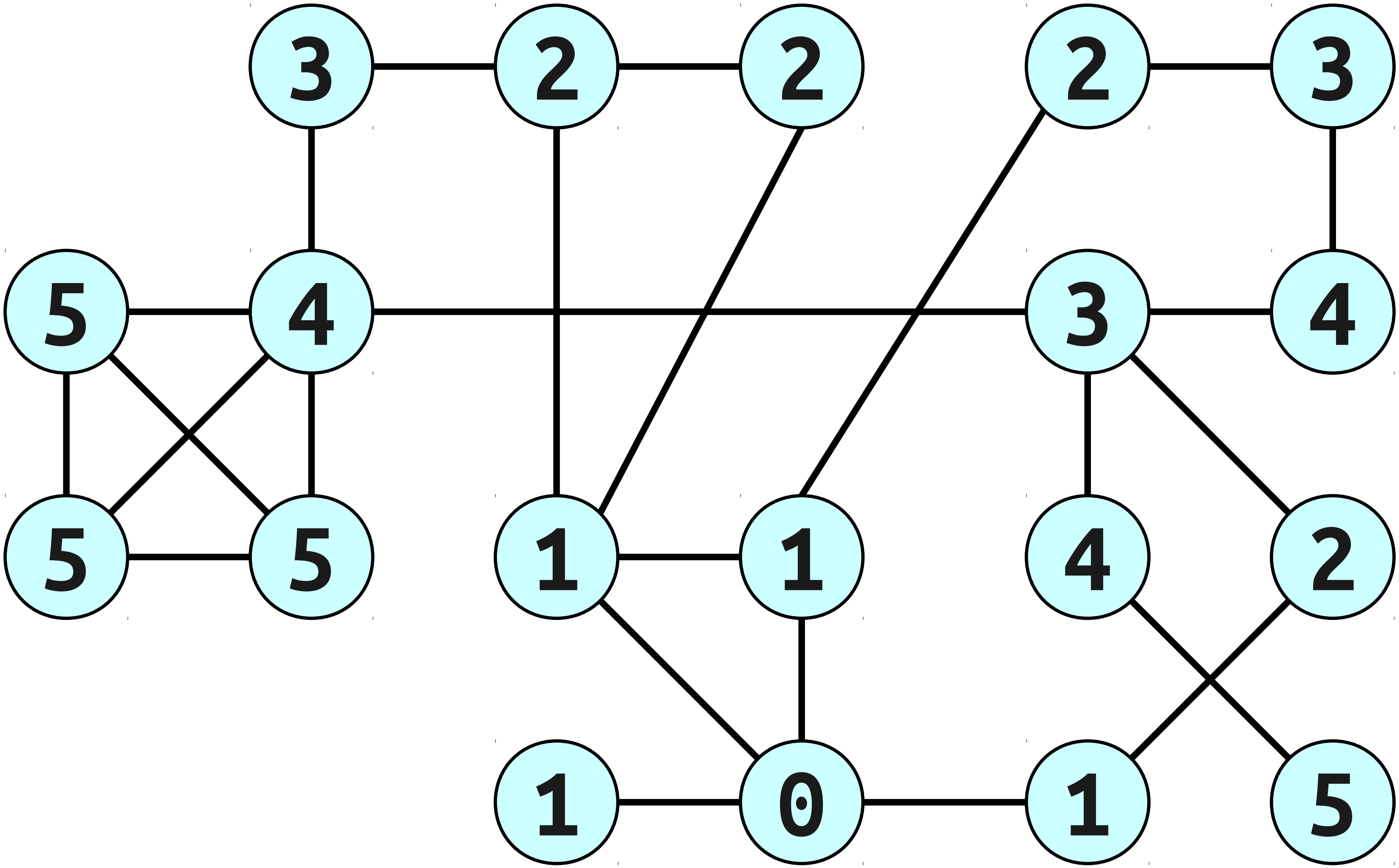
A Social Network



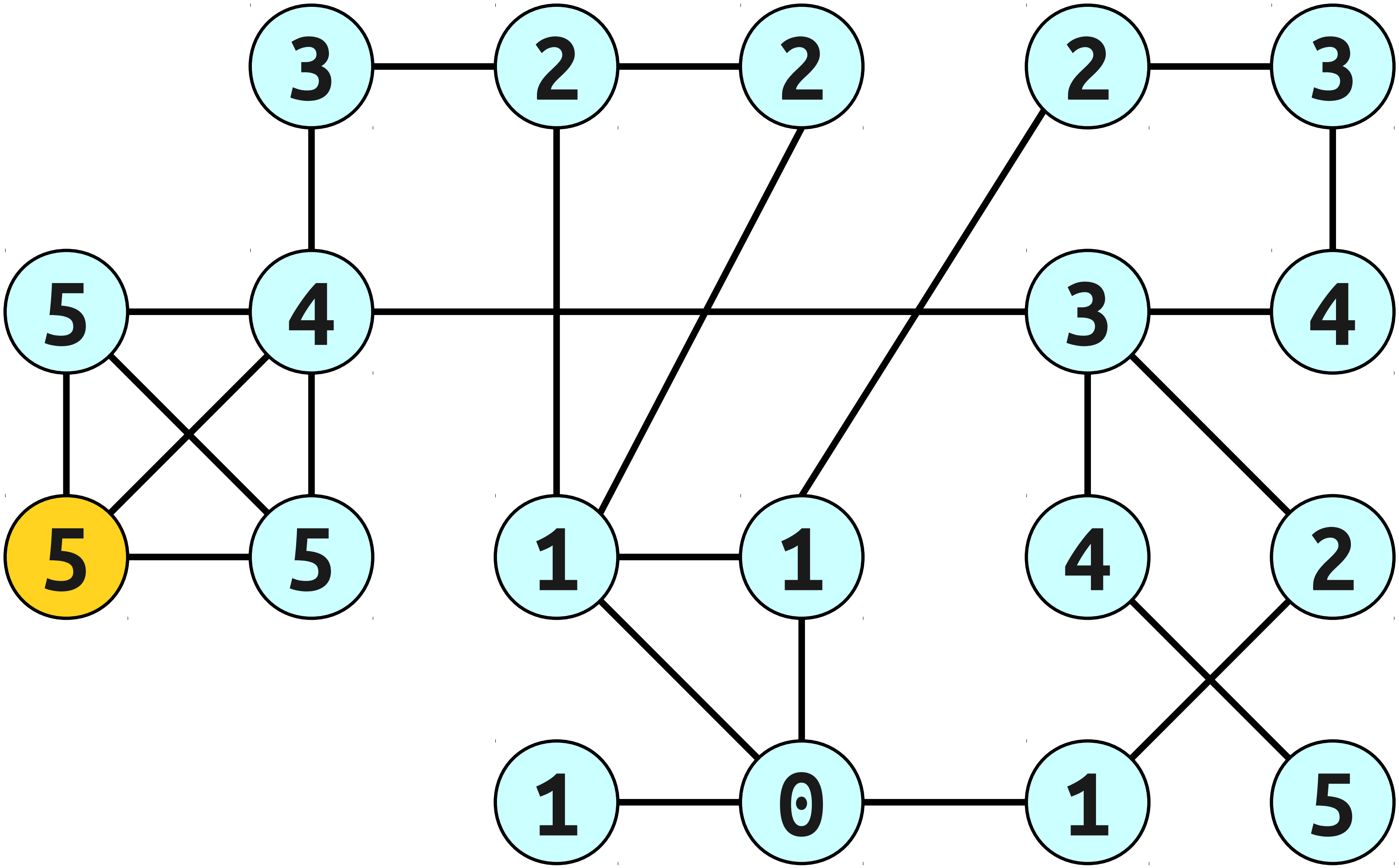
A Social Network



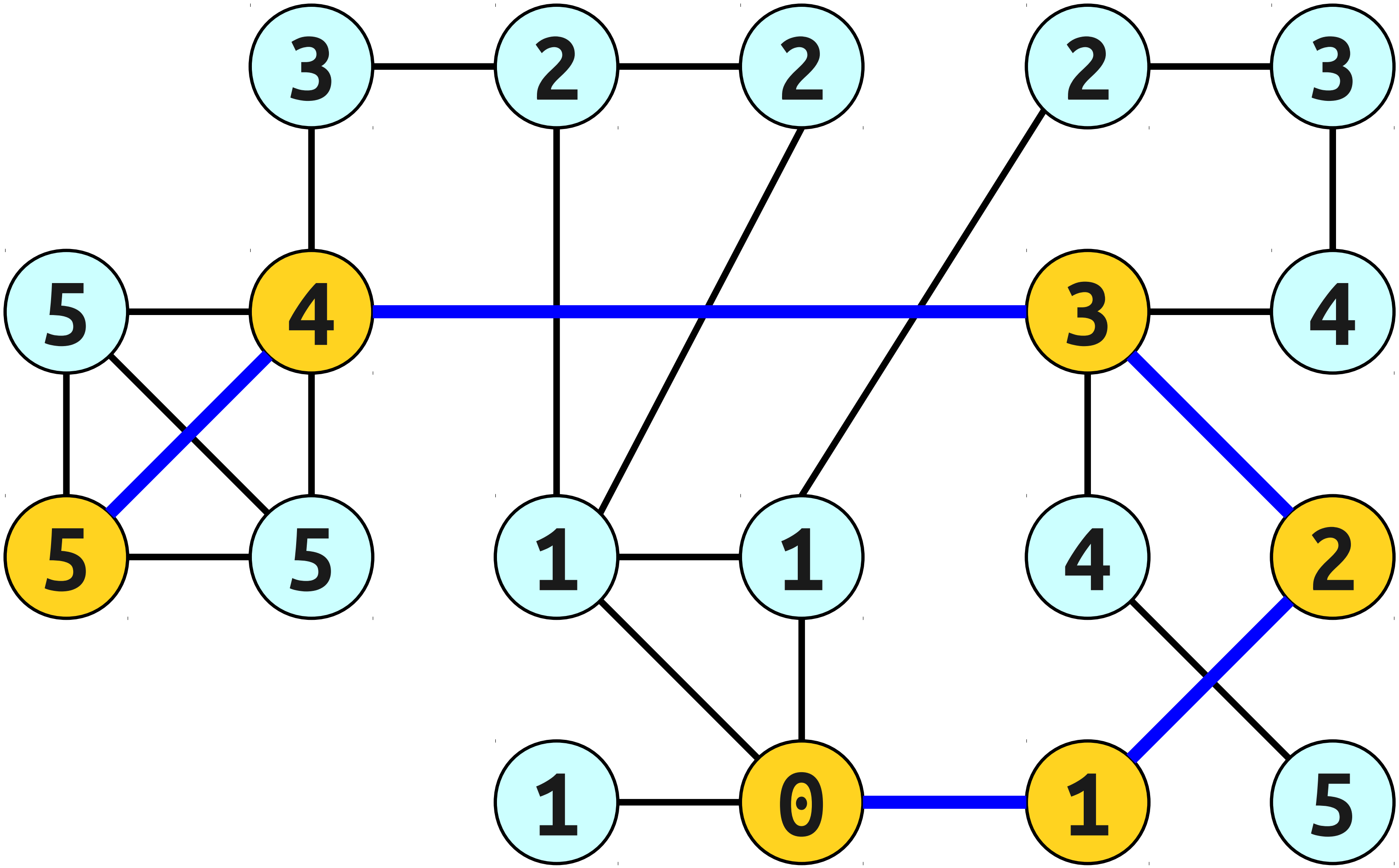
A Social Network



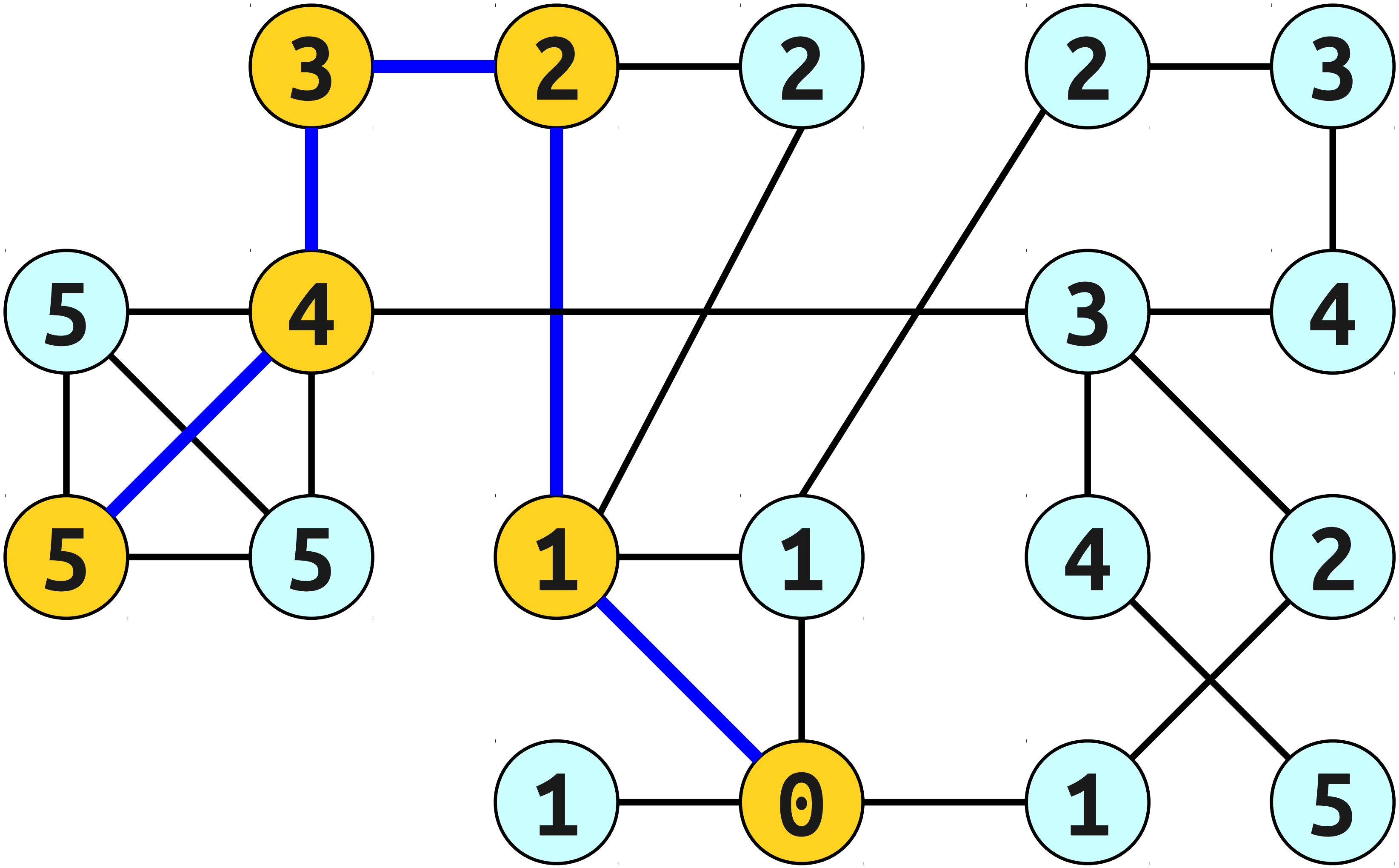
A Social Network



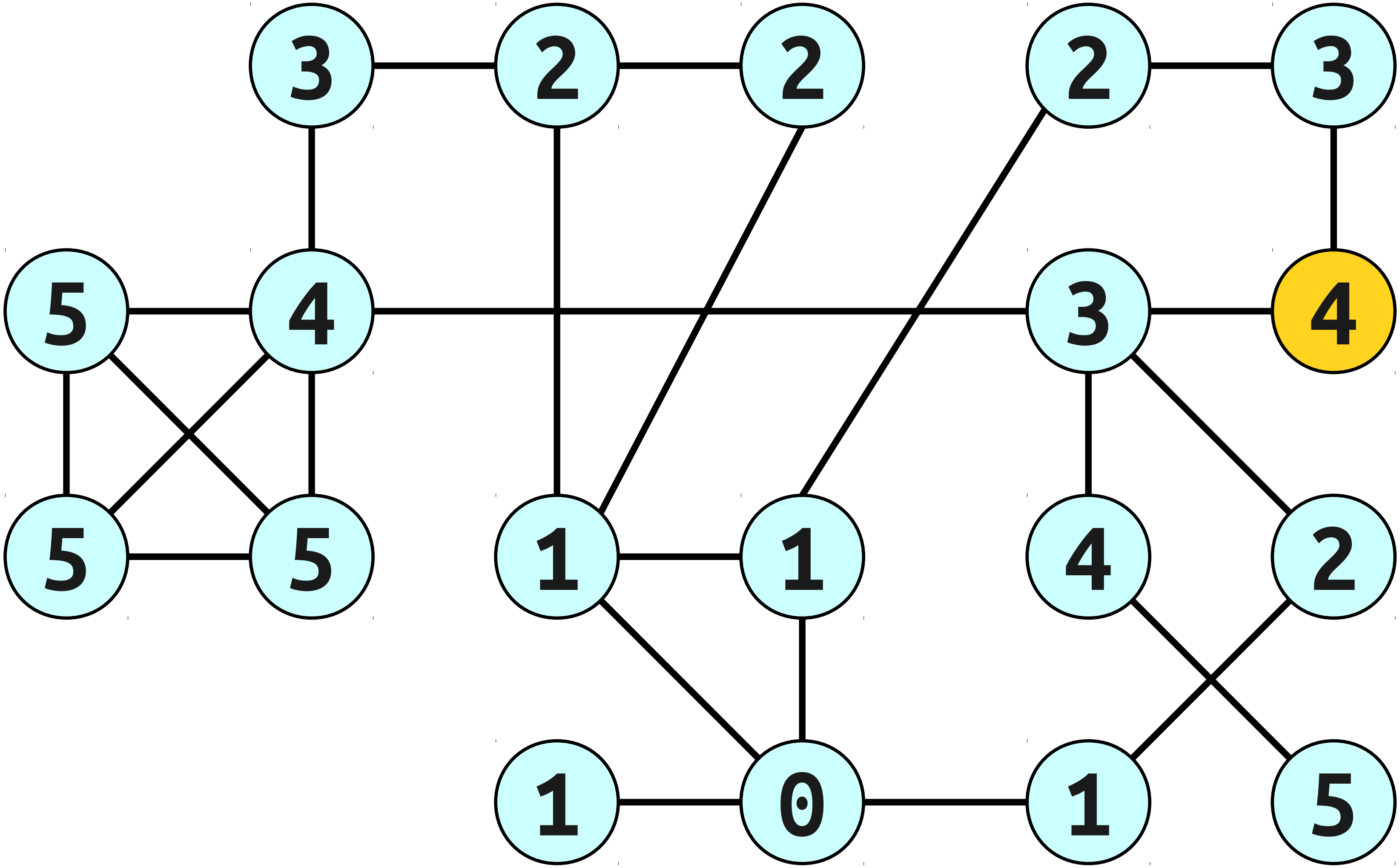
A Social Network



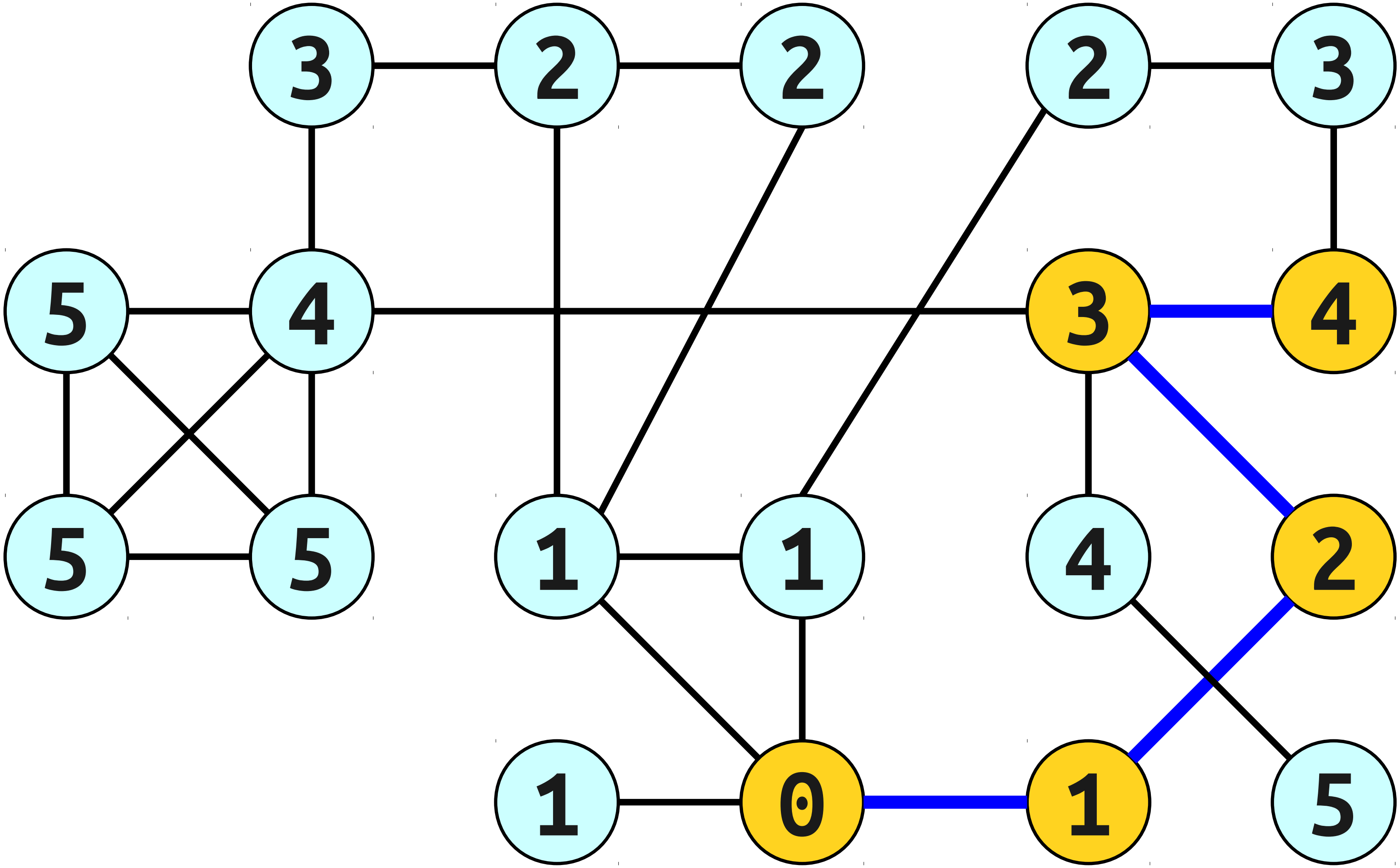
A Social Network



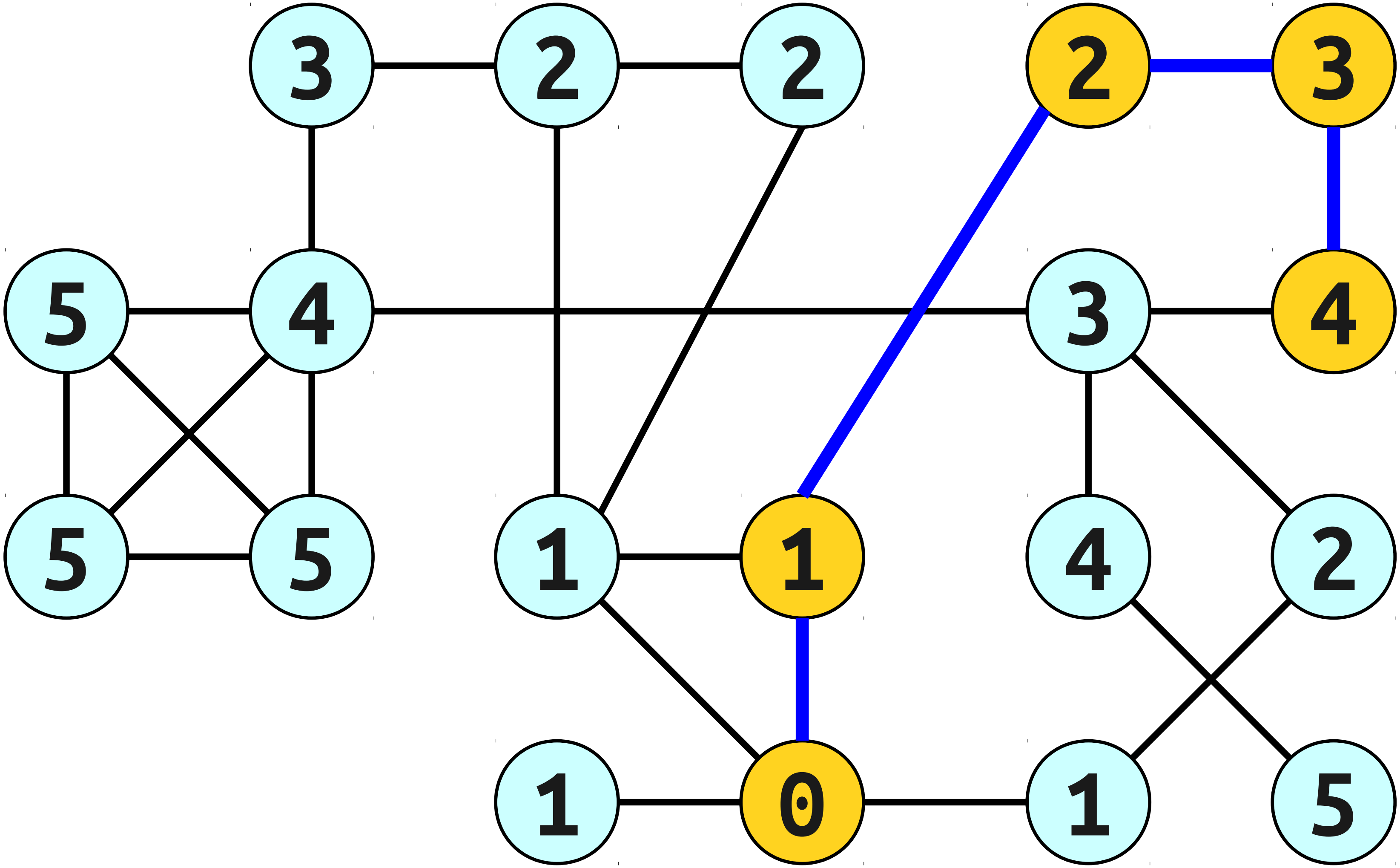
A Social Network



A Social Network



A Social Network



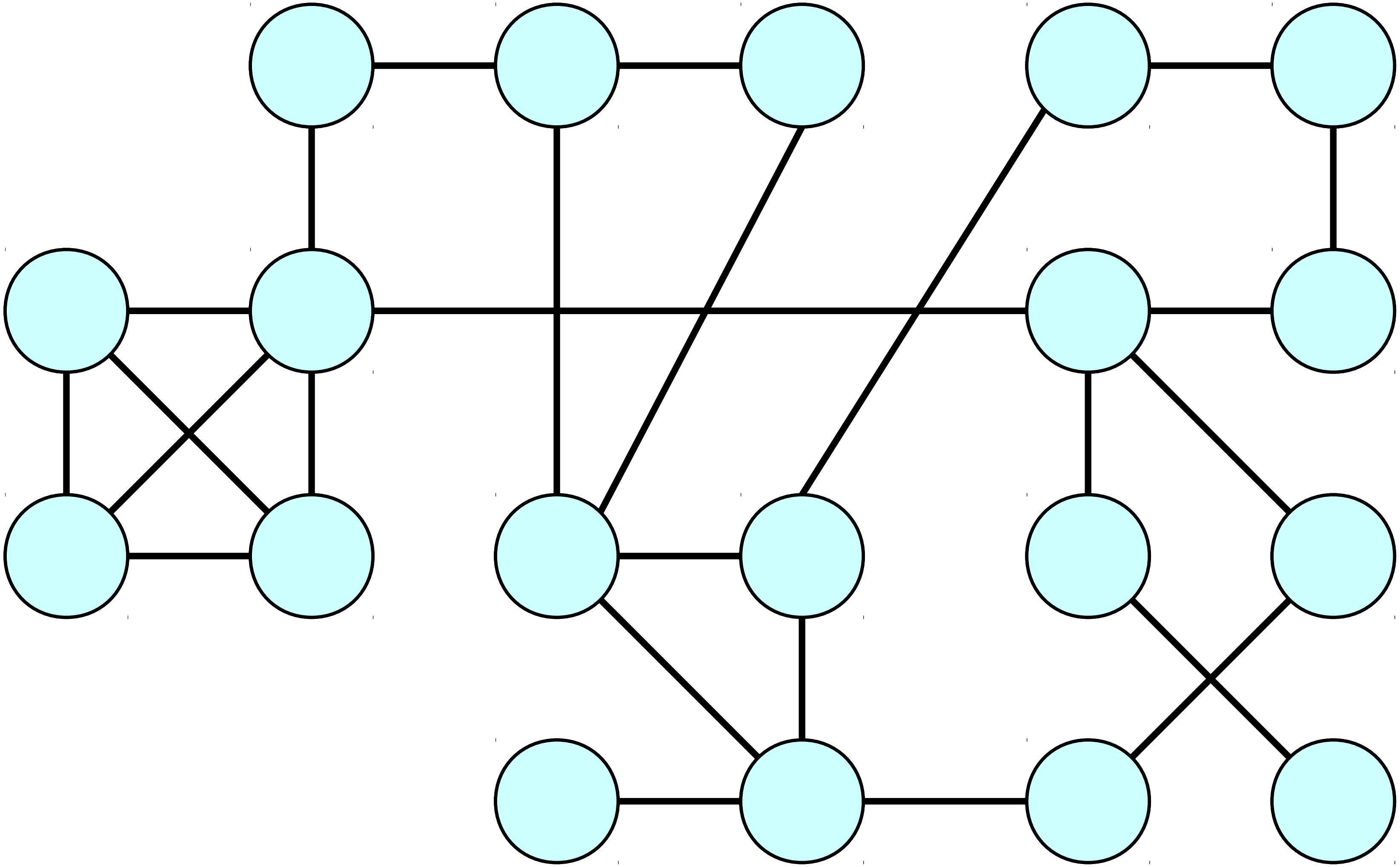
Shortest Paths

- The **length** of a path P (denoted $|P|$) in a graph is the number of edges it contains.
- A **shortest path** between u and v is a path P where $|P| \leq |P'|$ for any path P' from u to v.
- For any nodes u and v, define **d(*u*, *v*)** to be the length of the shortest path from u to v, or ∞ if no such path exists.
- What is d(v, v) for any $v \in V$?

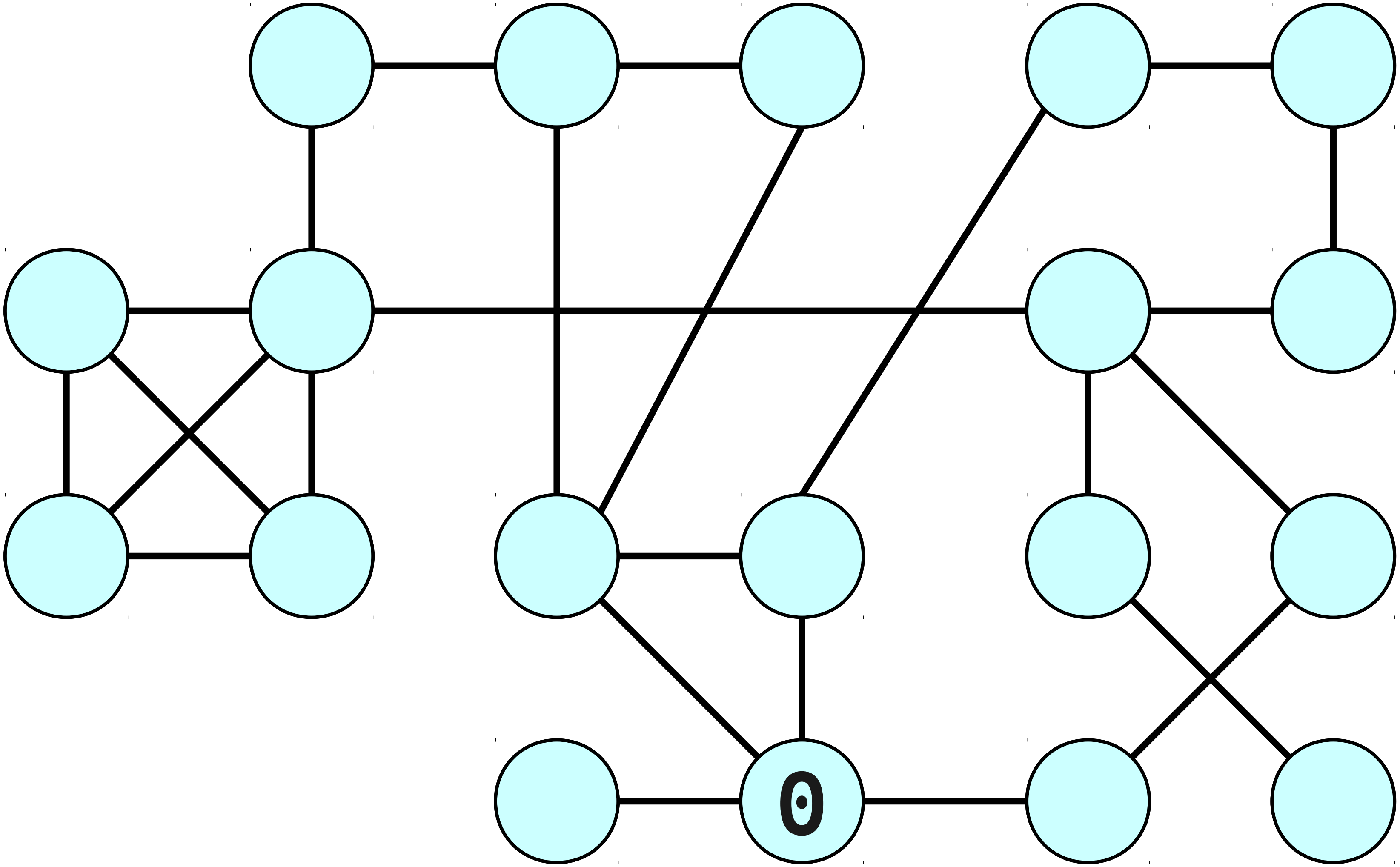
The Shortest Path Problem

- **Input:**
  - A graph $G = (V, E)$, which may be directed or undirected.
  - A start node $s \in V$.
- **Output:**
  - A table dist[v], where dist[v] = d(s, v) for any $v \in V$.

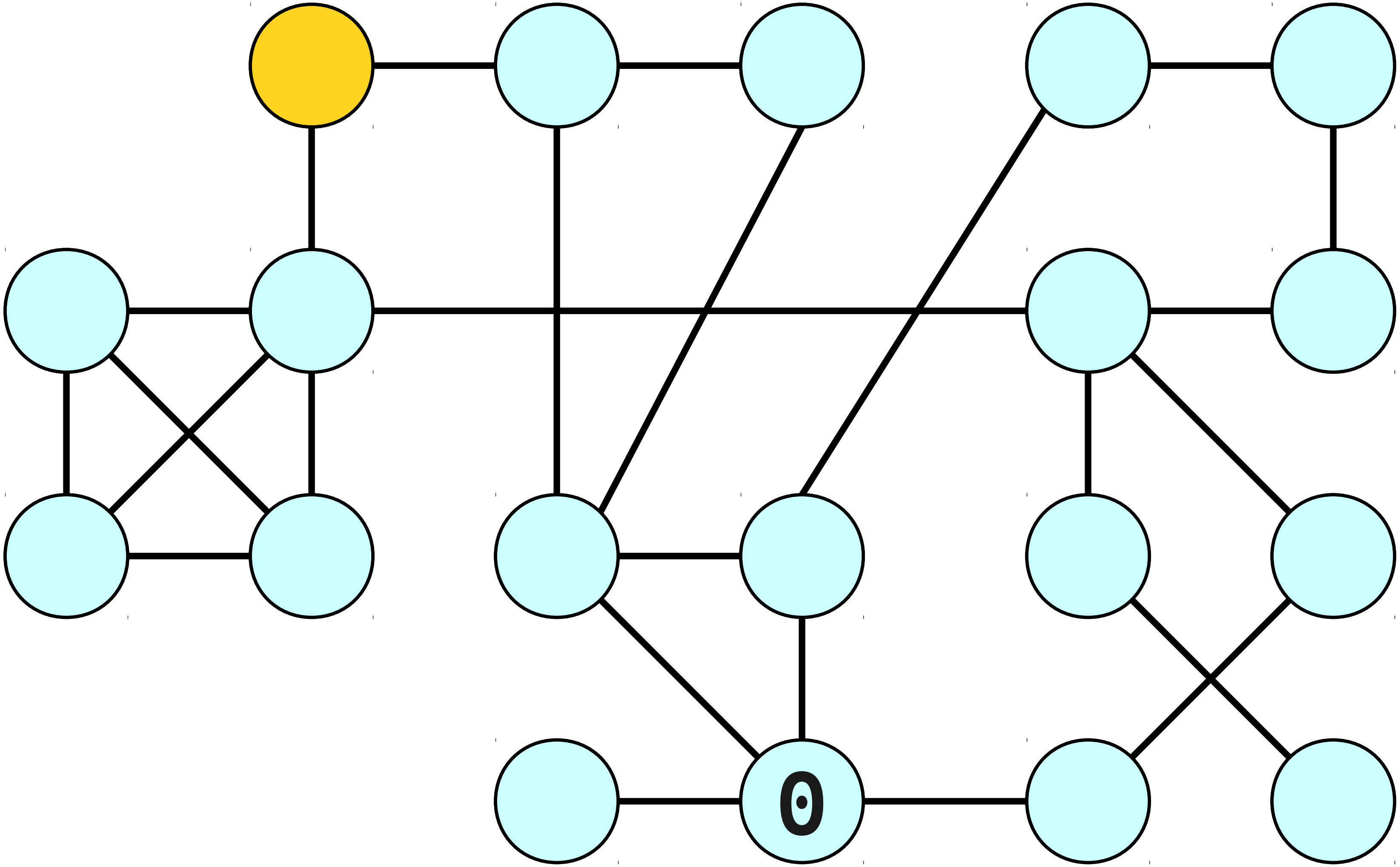
An Inefficient Algorithm



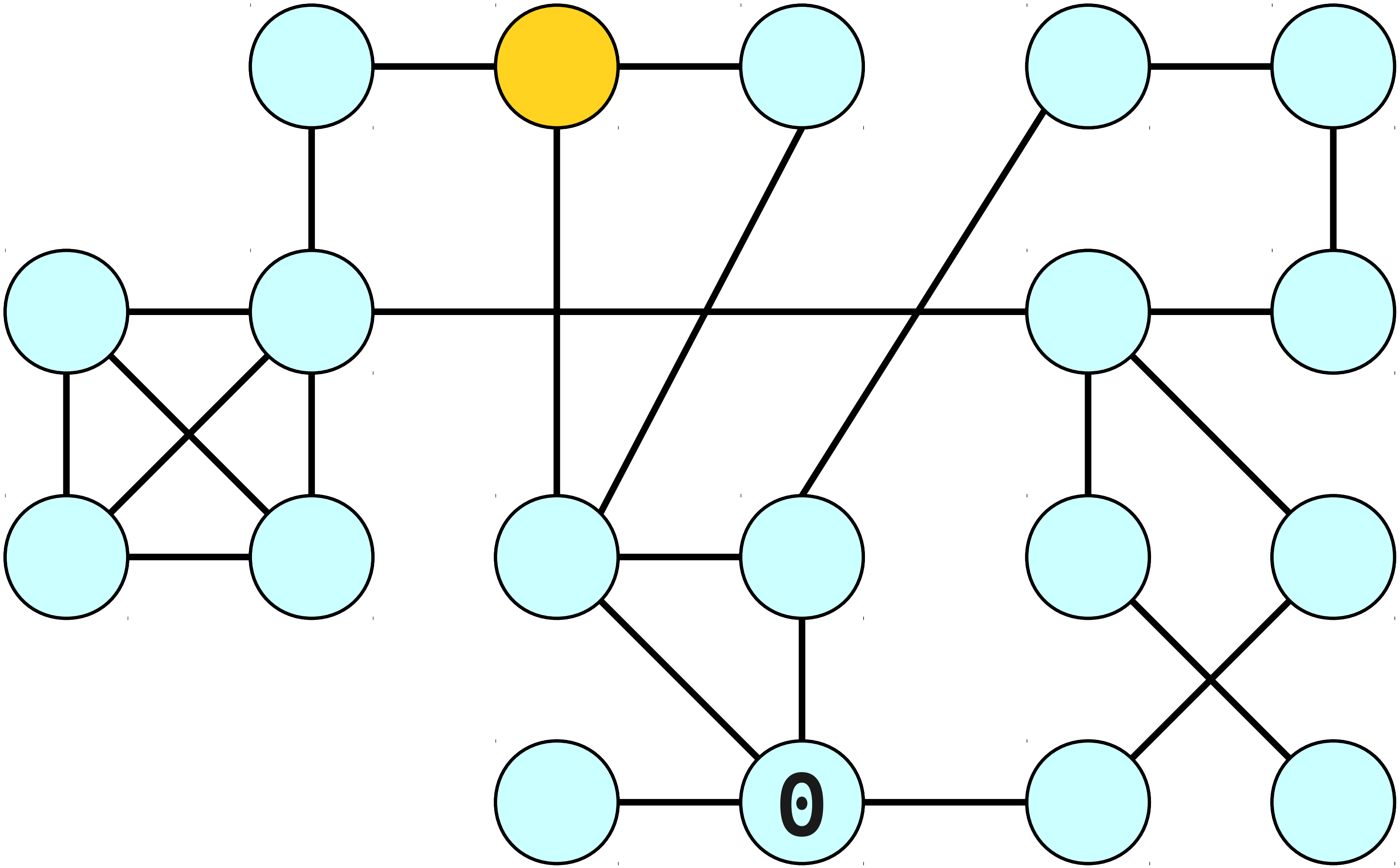
An Inefficient Algorithm



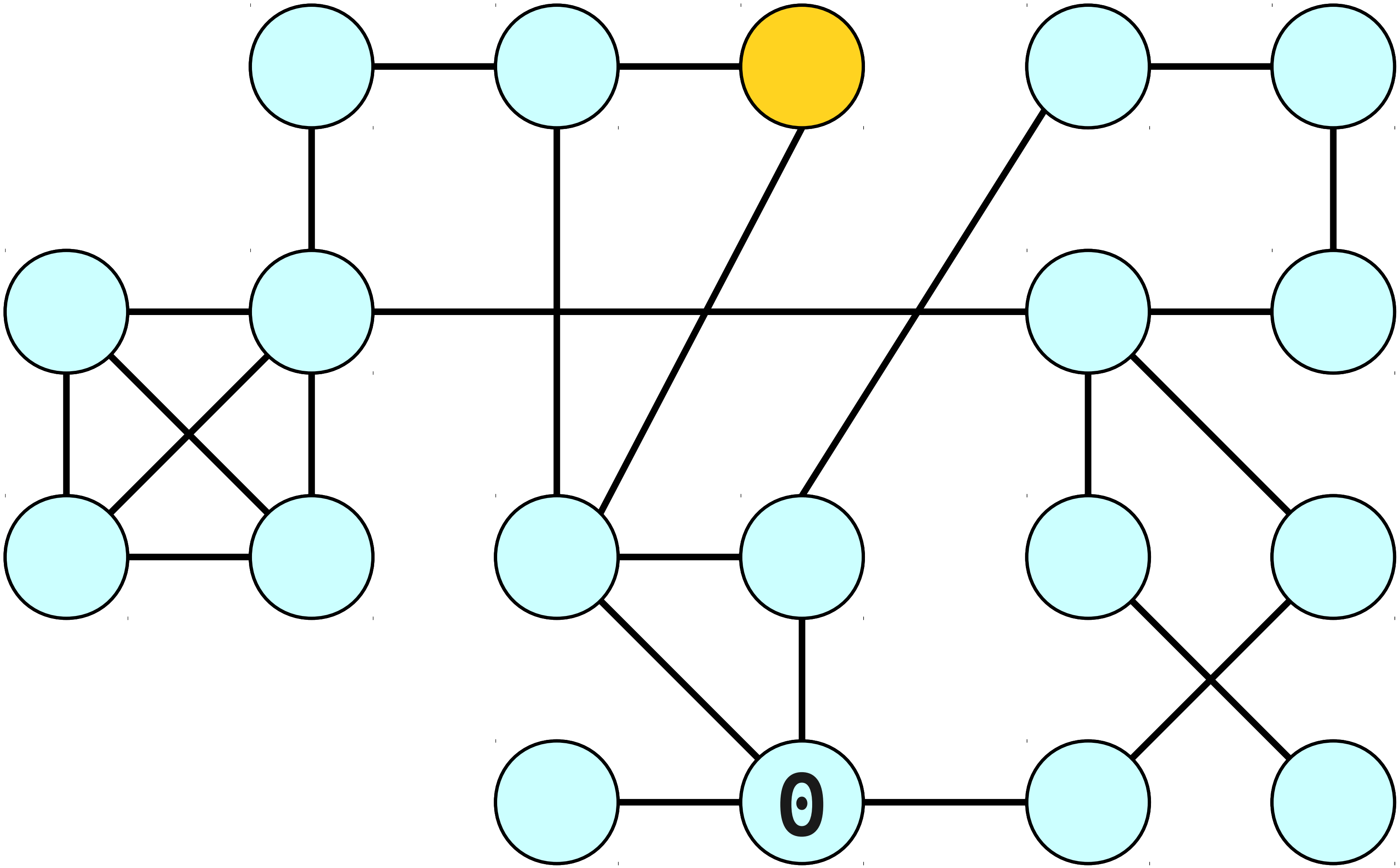
An Inefficient Algorithm



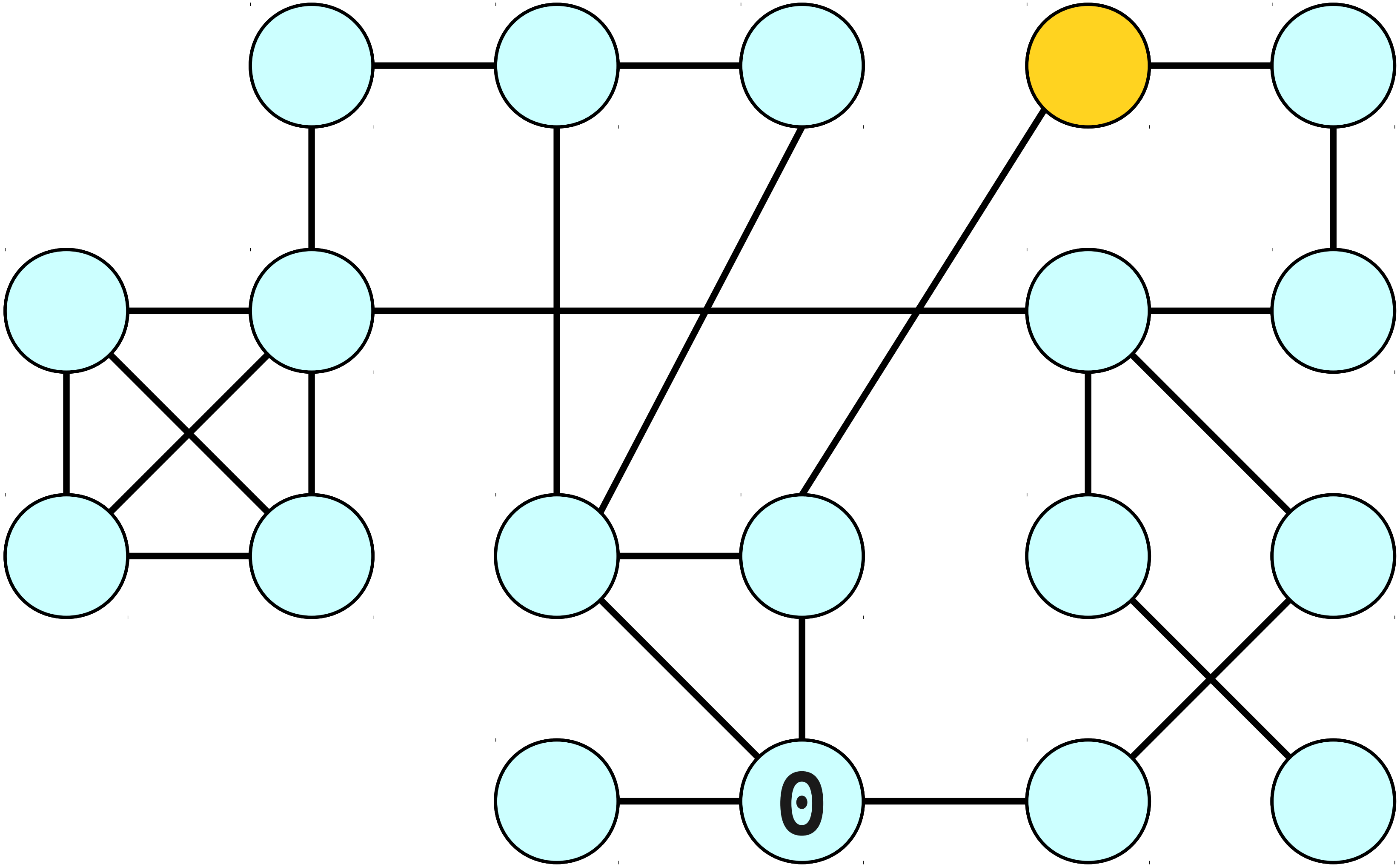
An Inefficient Algorithm



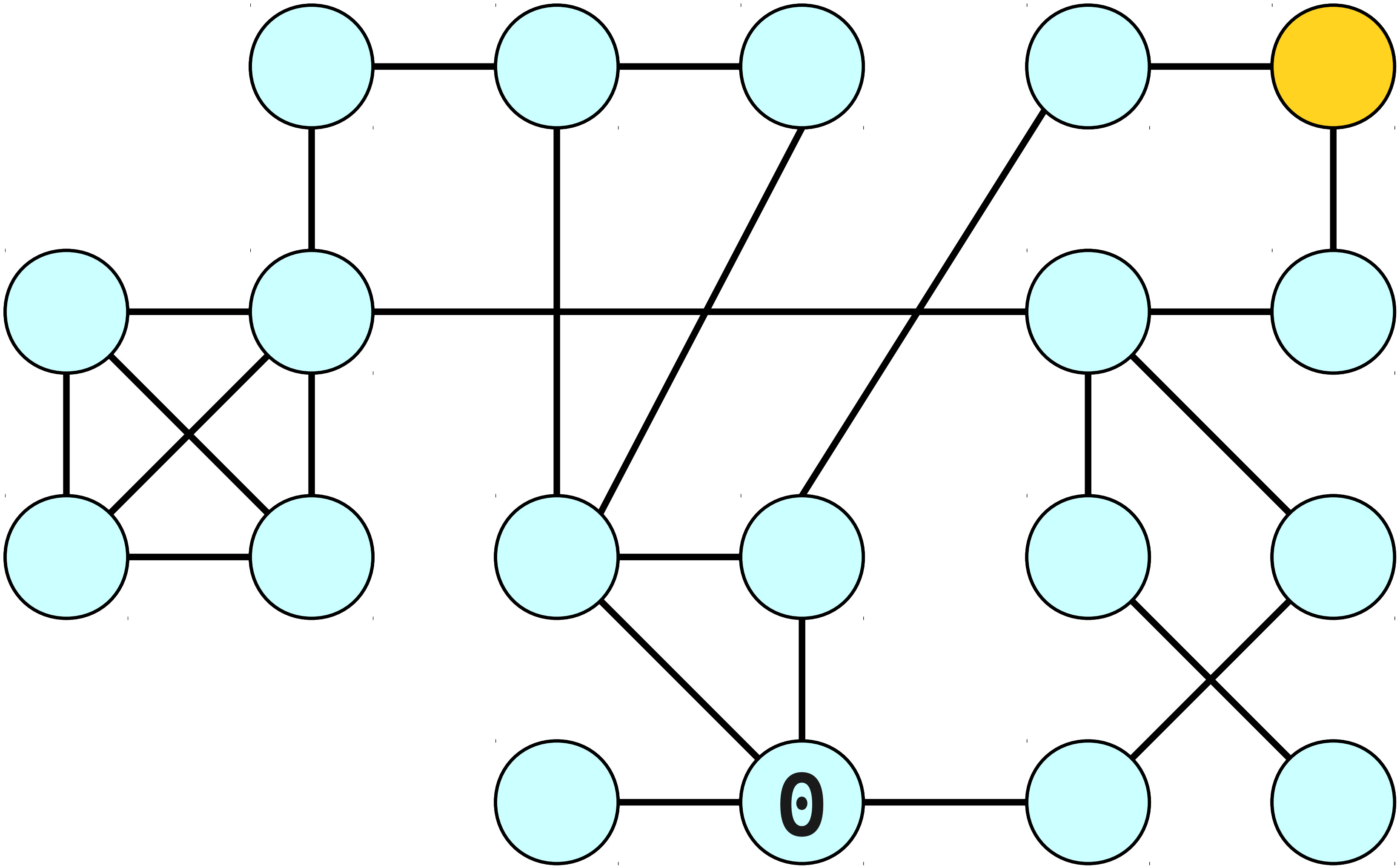
An Inefficient Algorithm



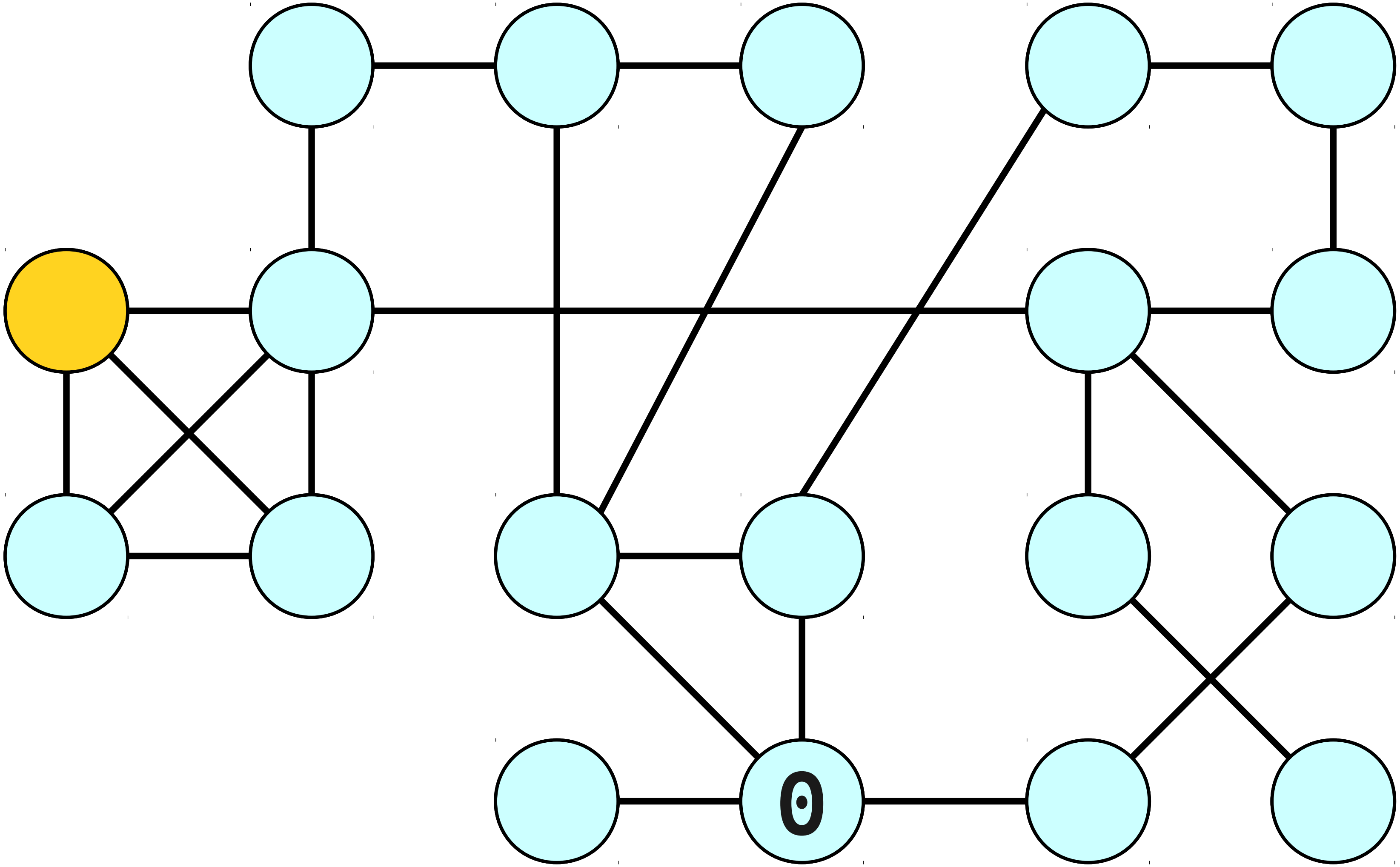
An Inefficient Algorithm



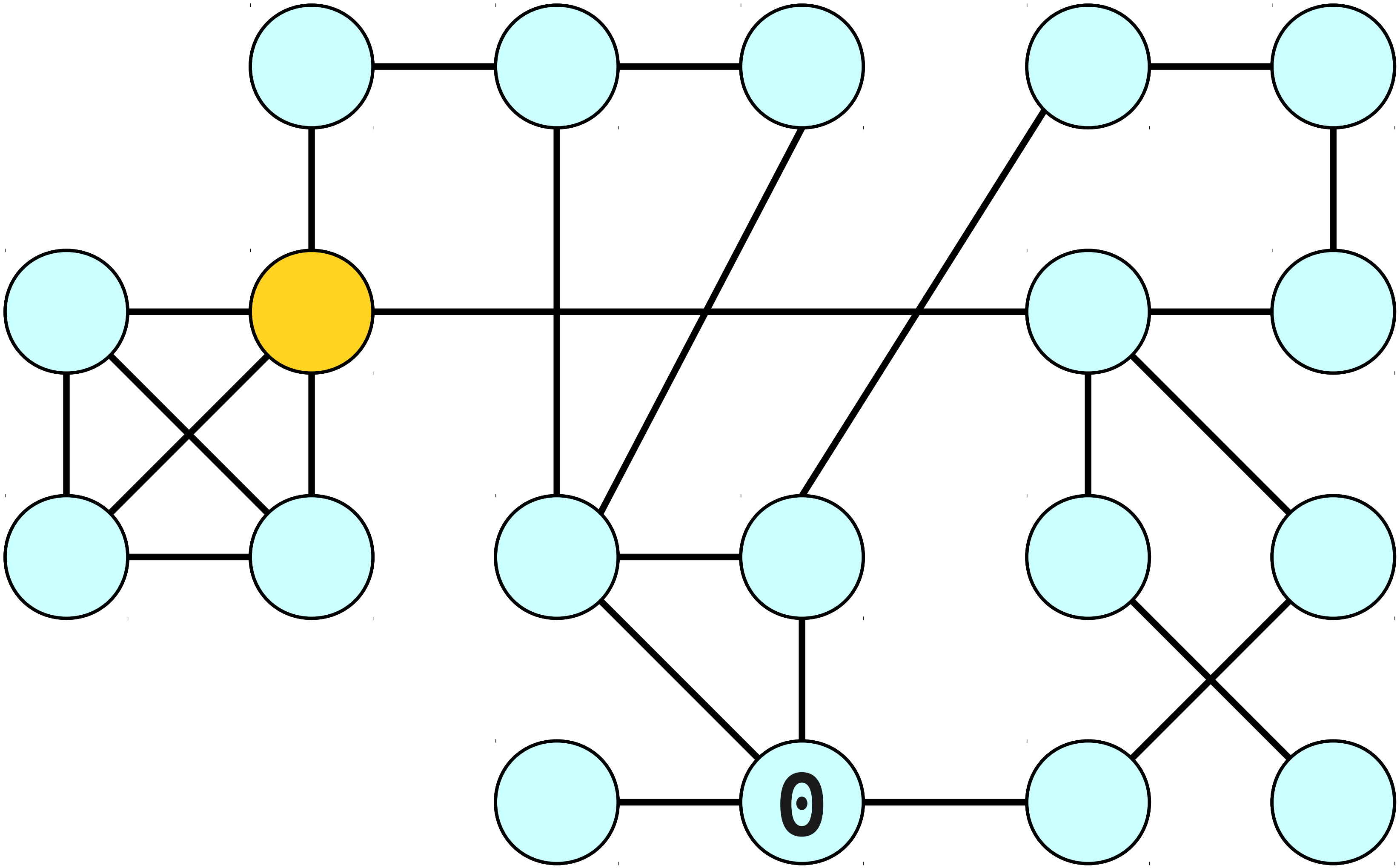
An Inefficient Algorithm



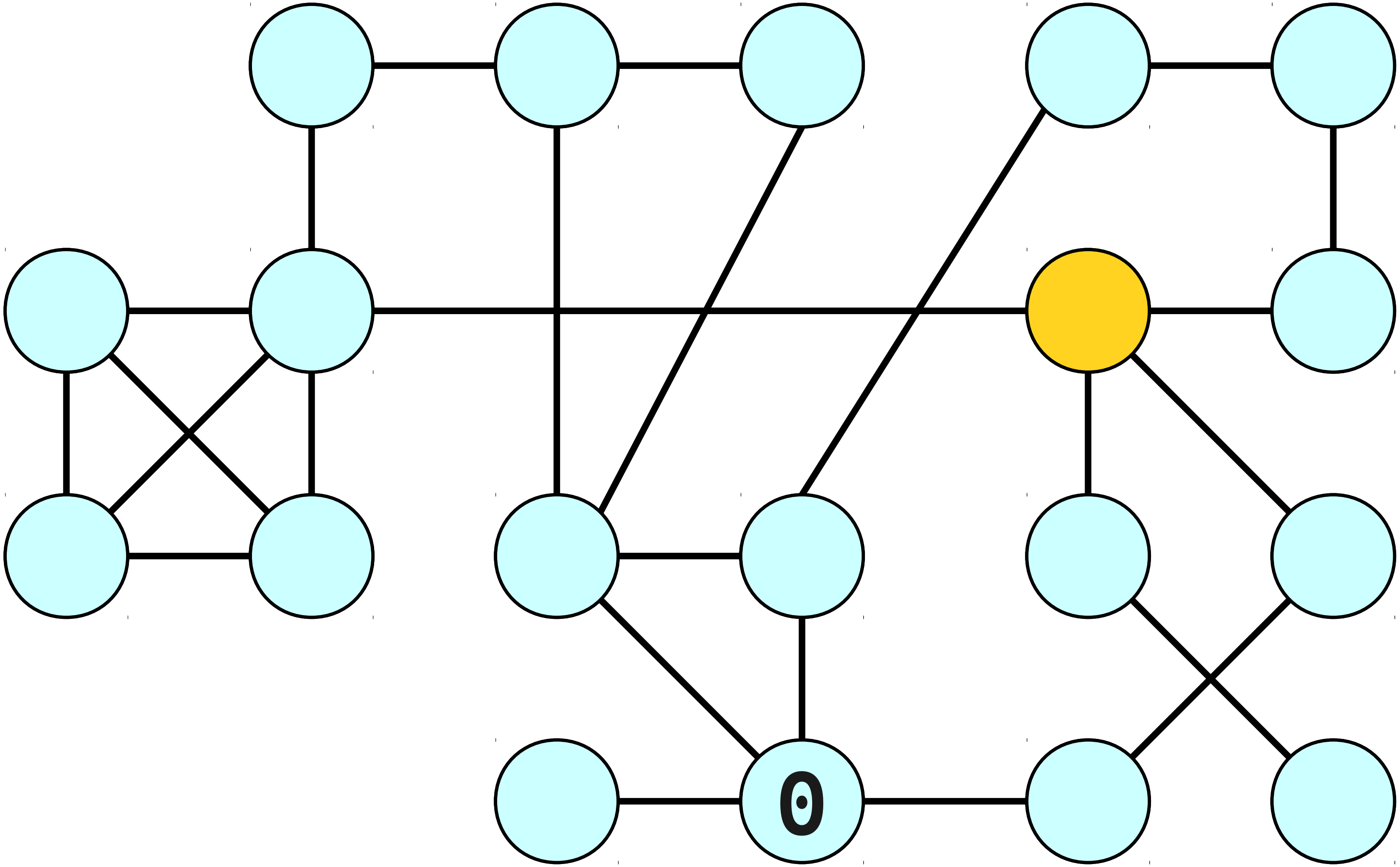
An Inefficient Algorithm



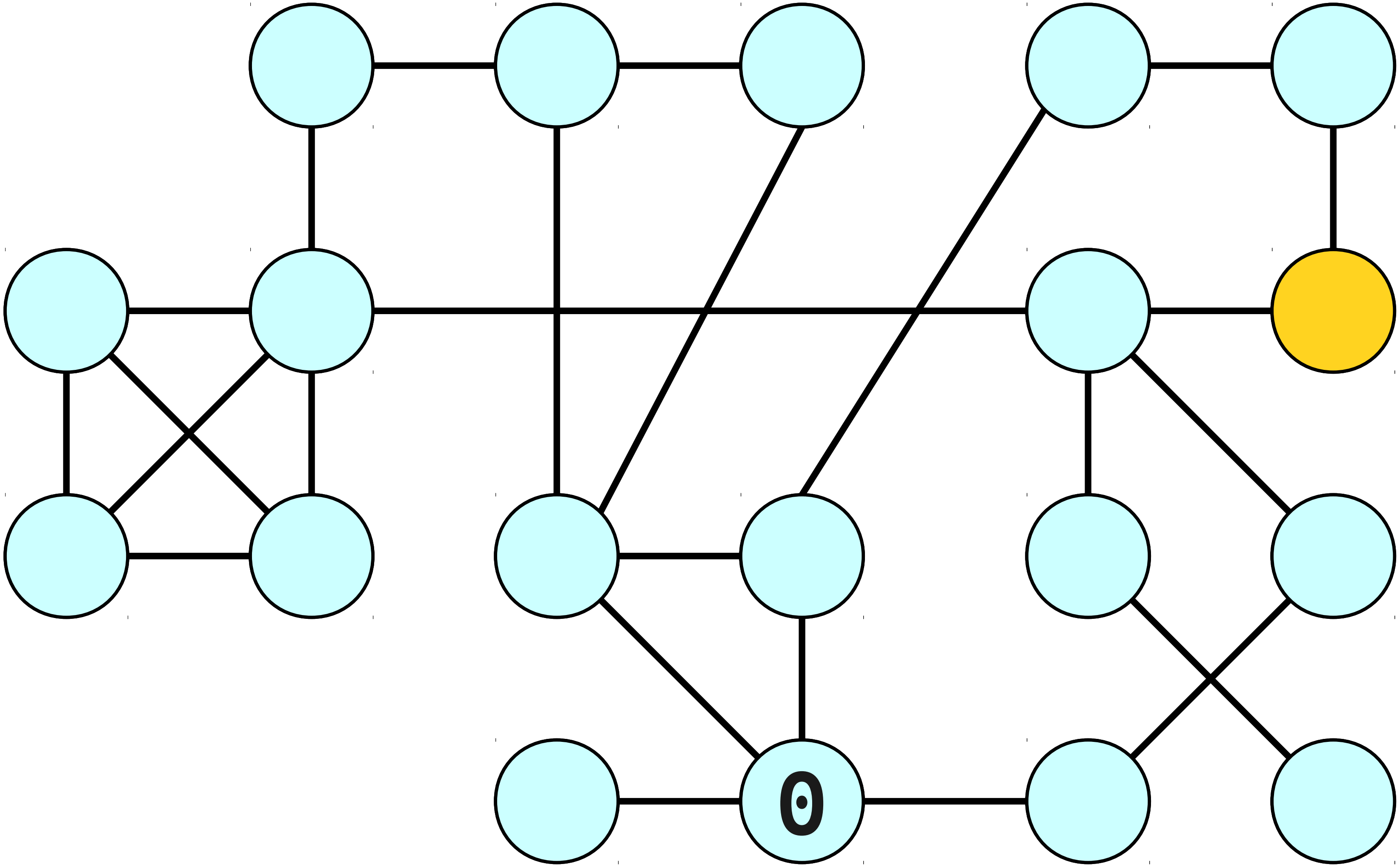
An Inefficient Algorithm



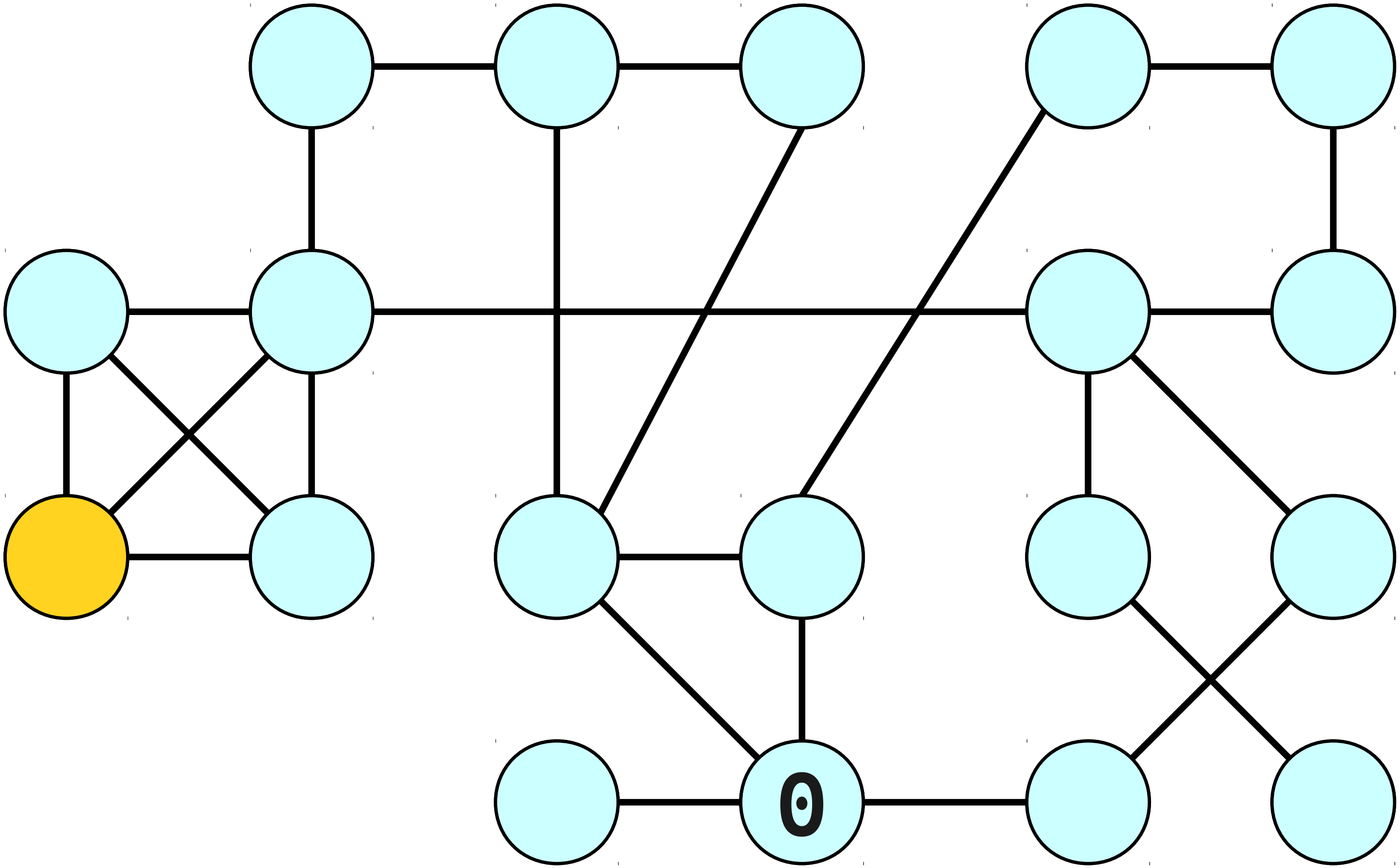
An Inefficient Algorithm



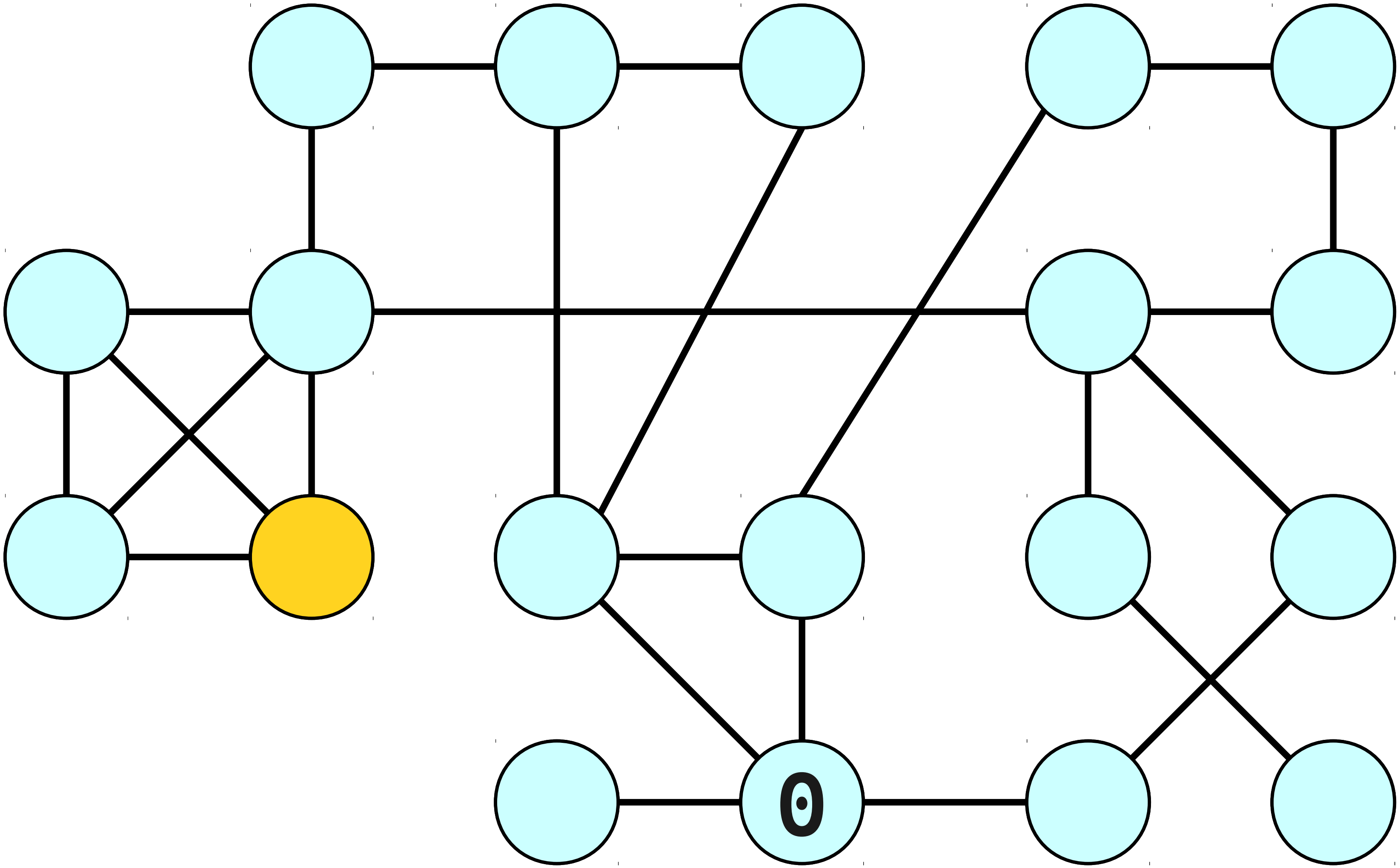
An Inefficient Algorithm



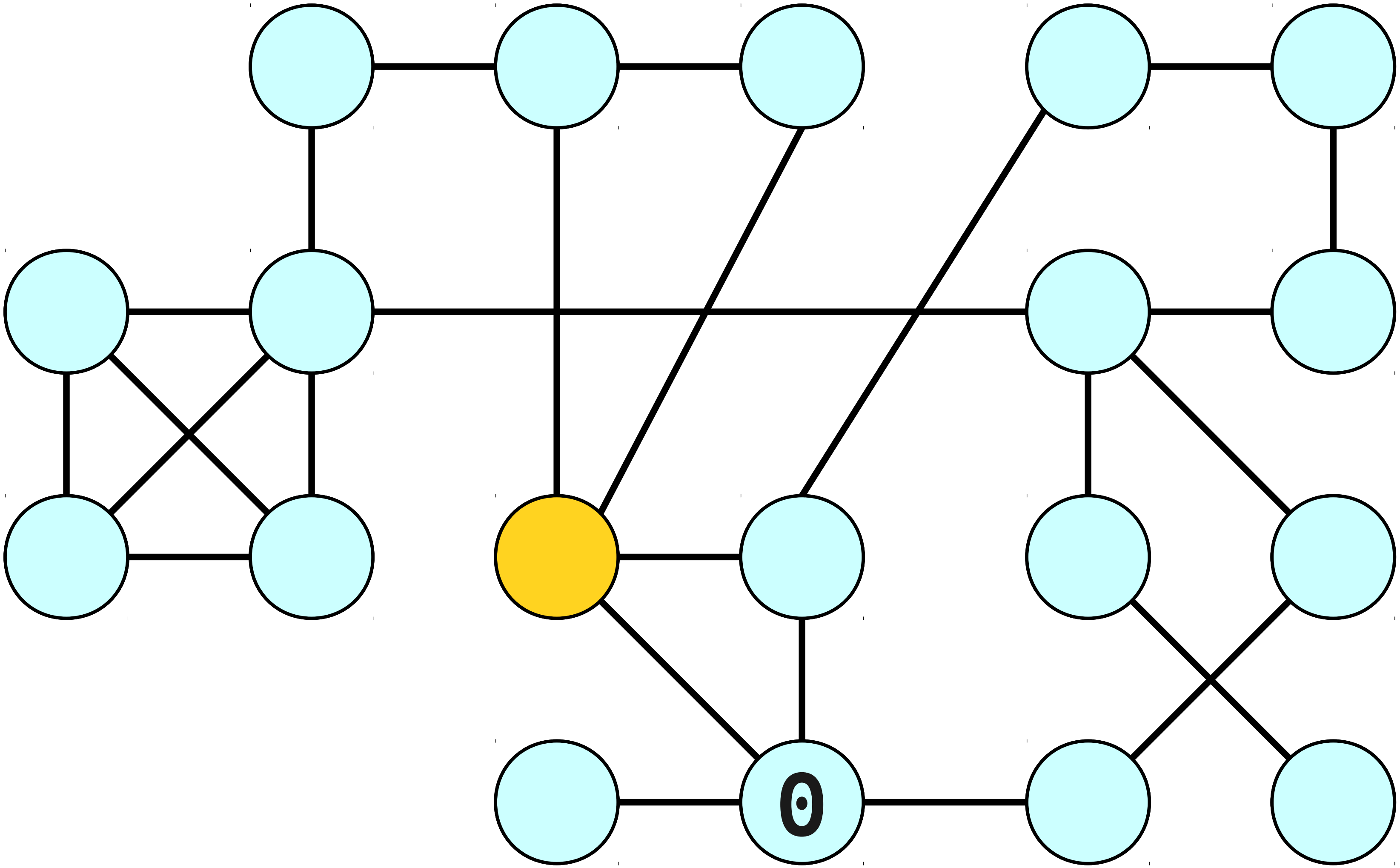
An Inefficient Algorithm



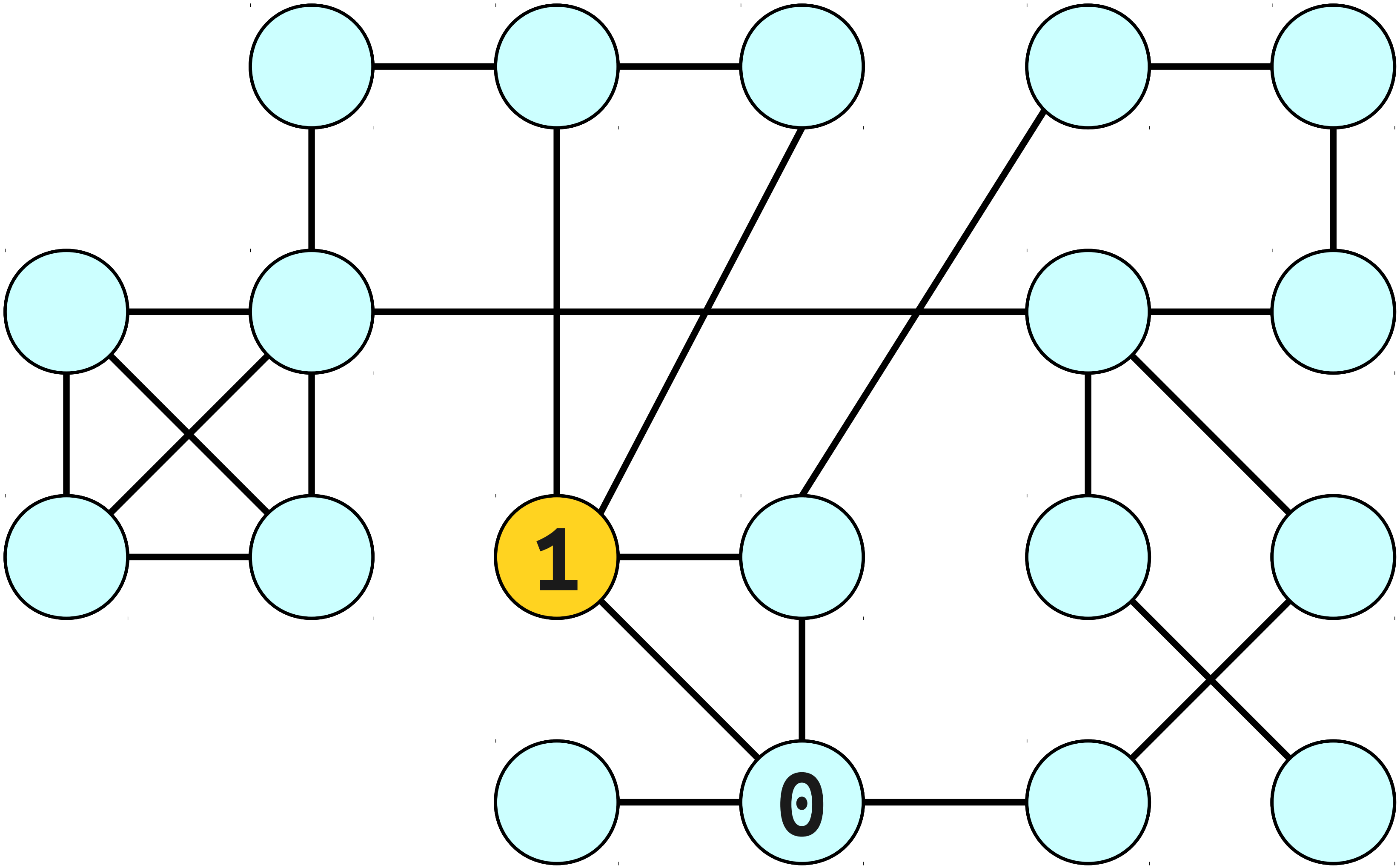
An Inefficient Algorithm



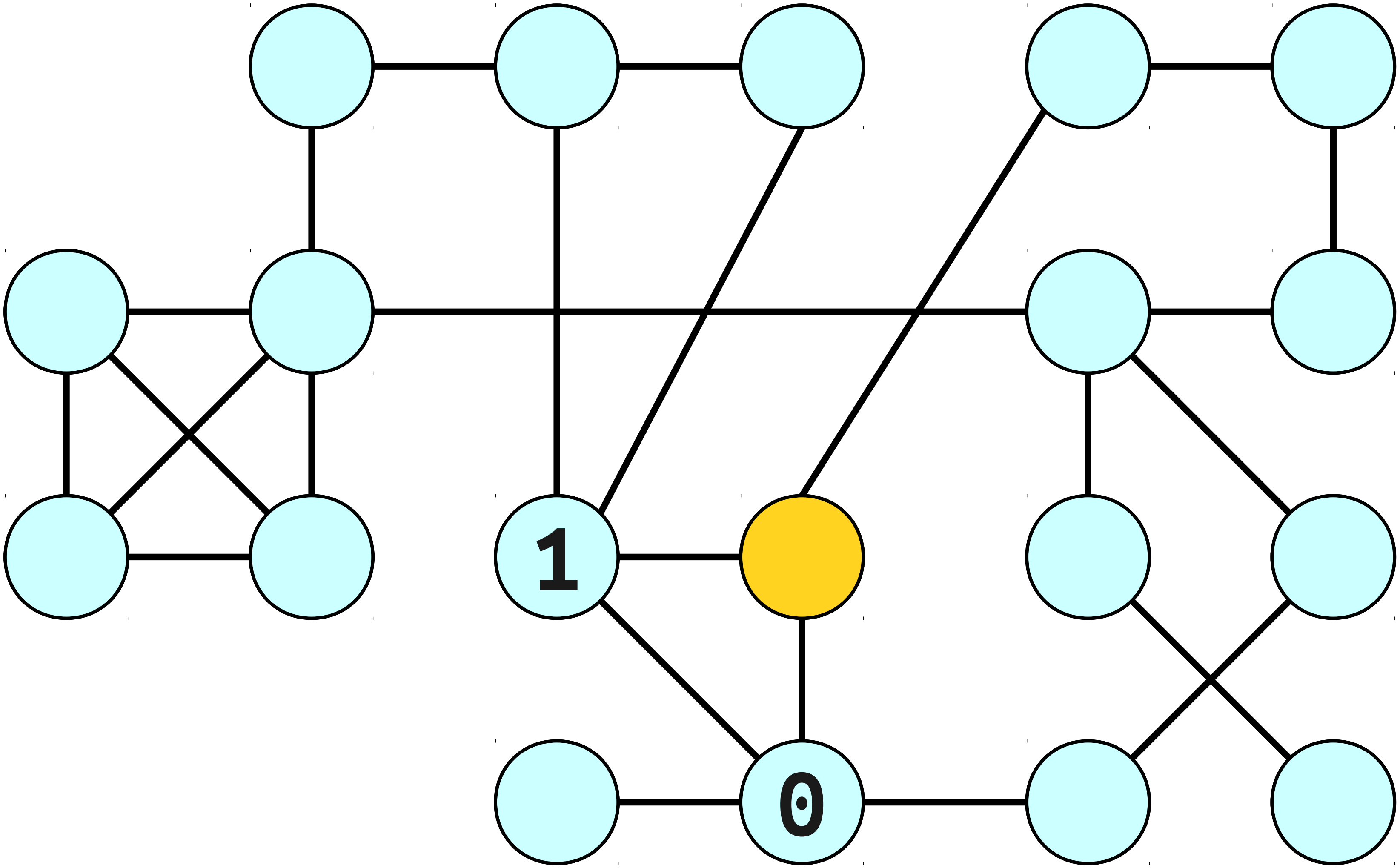
An Inefficient Algorithm



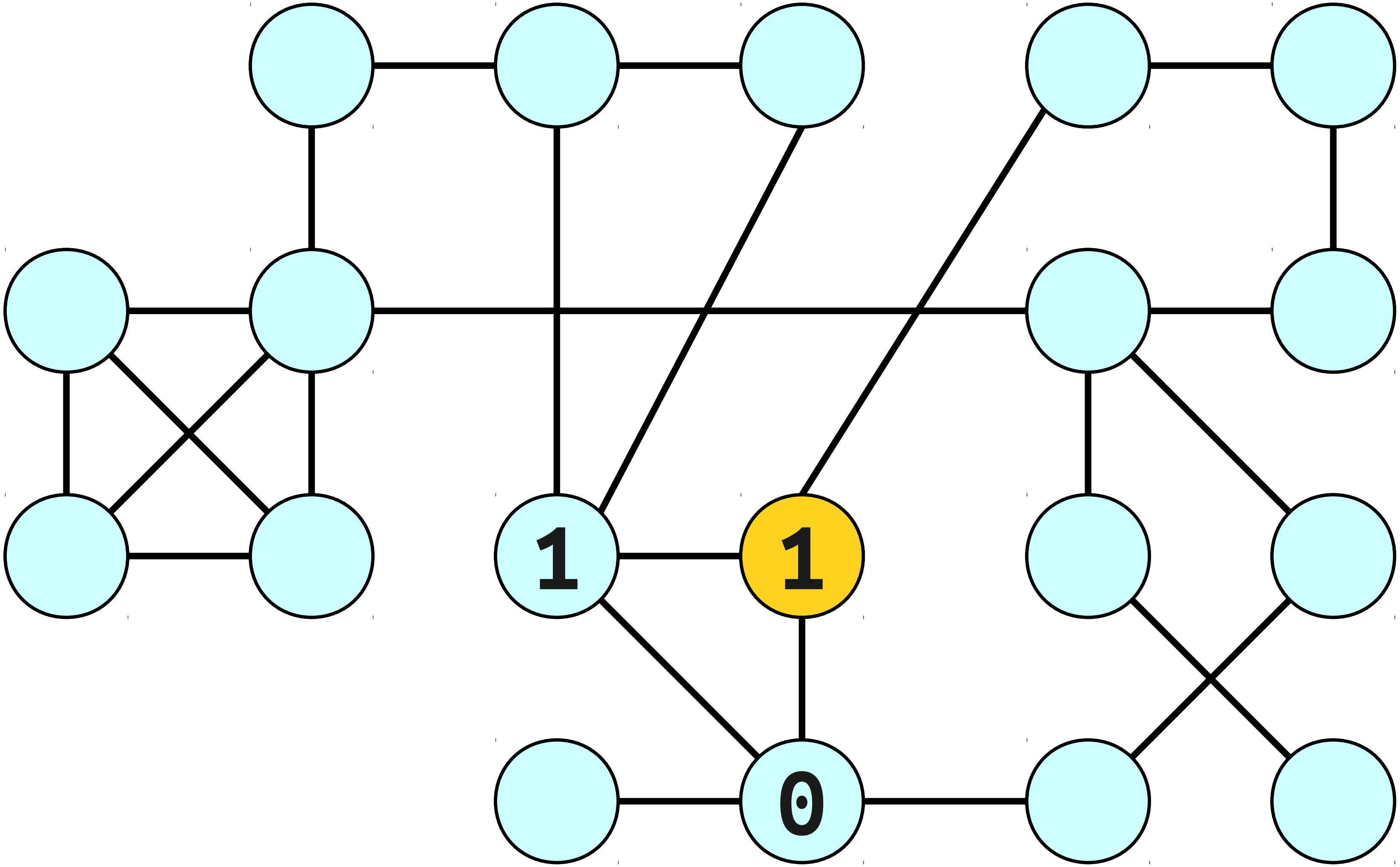
An Inefficient Algorithm



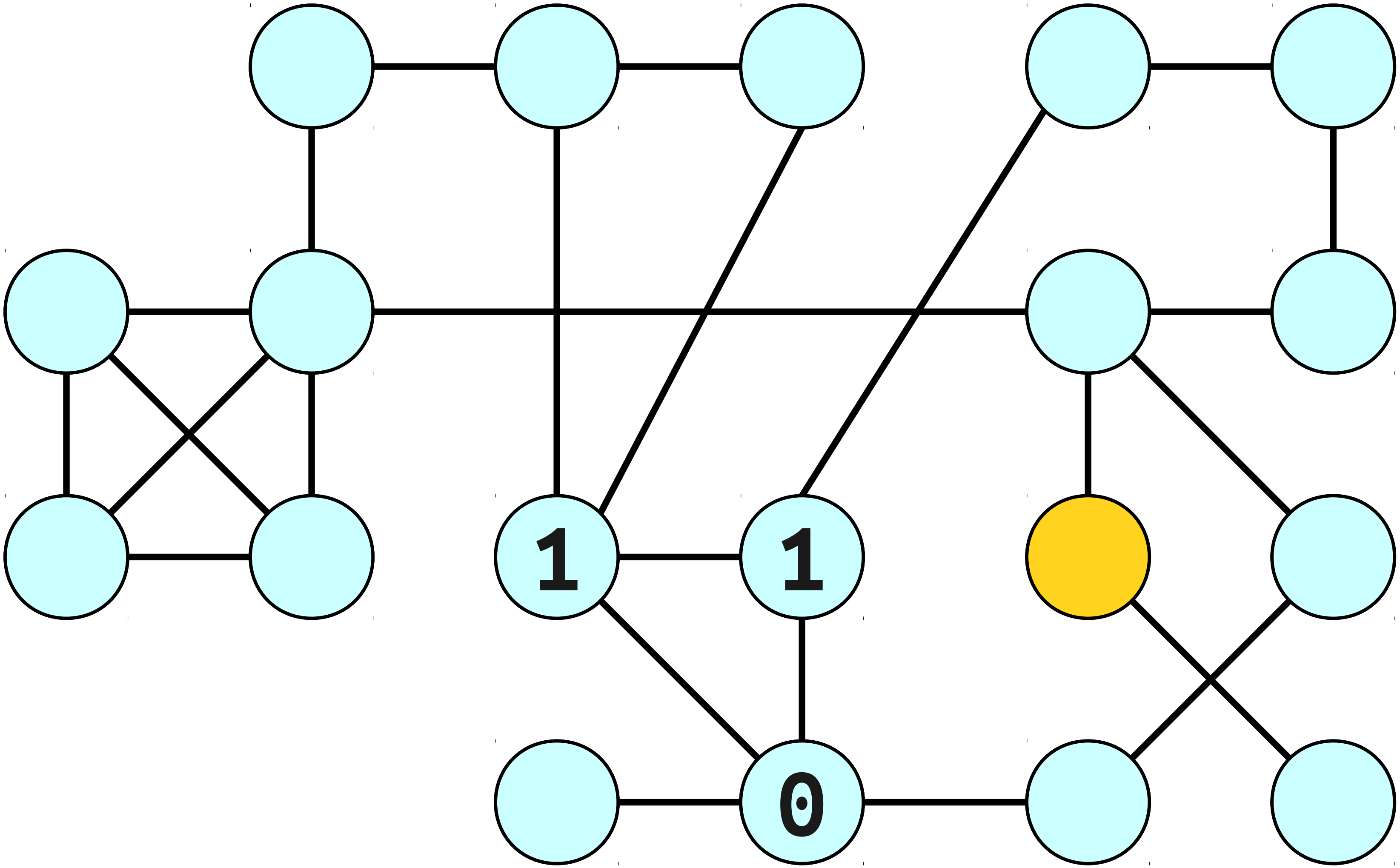
An Inefficient Algorithm



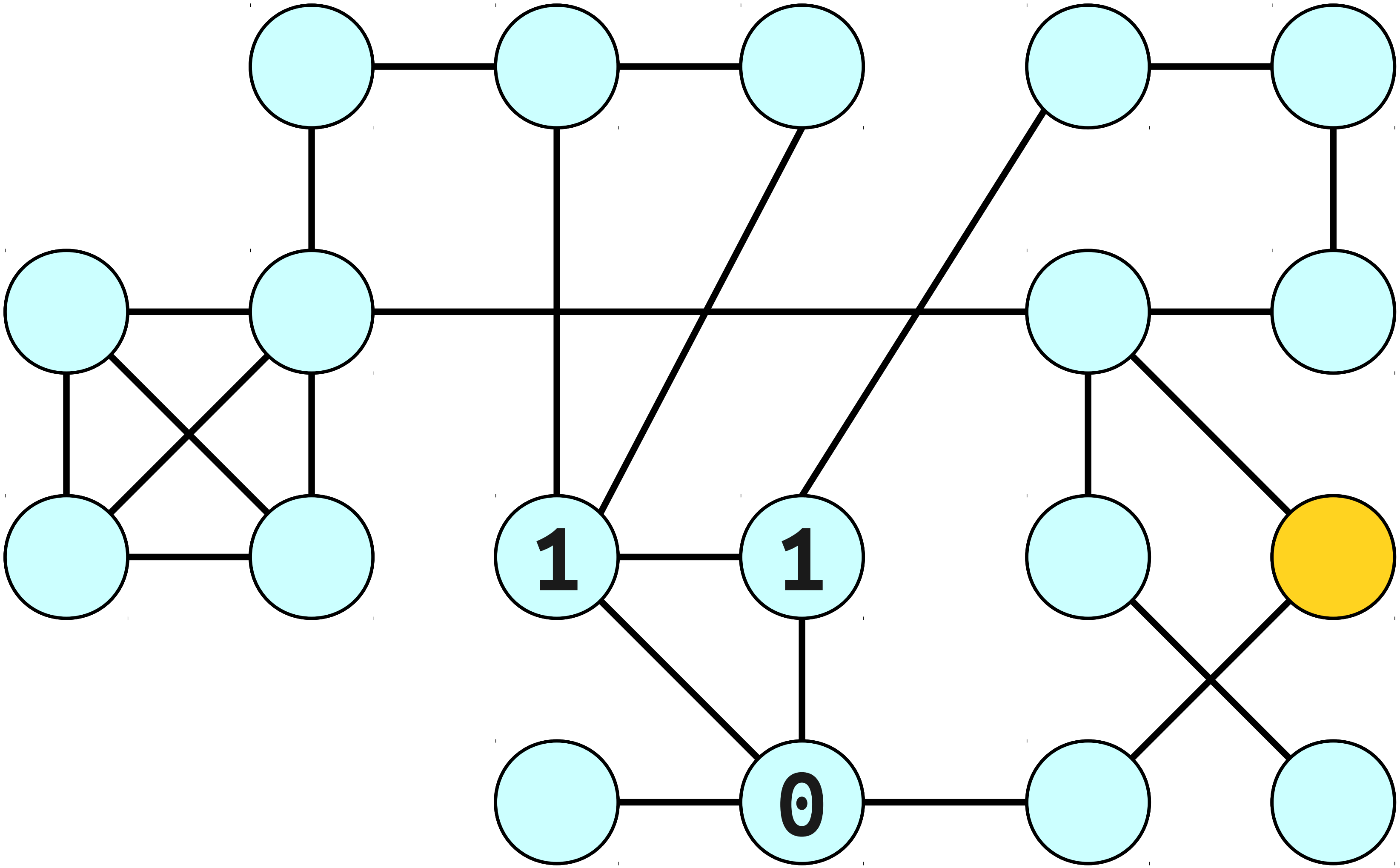
An Inefficient Algorithm



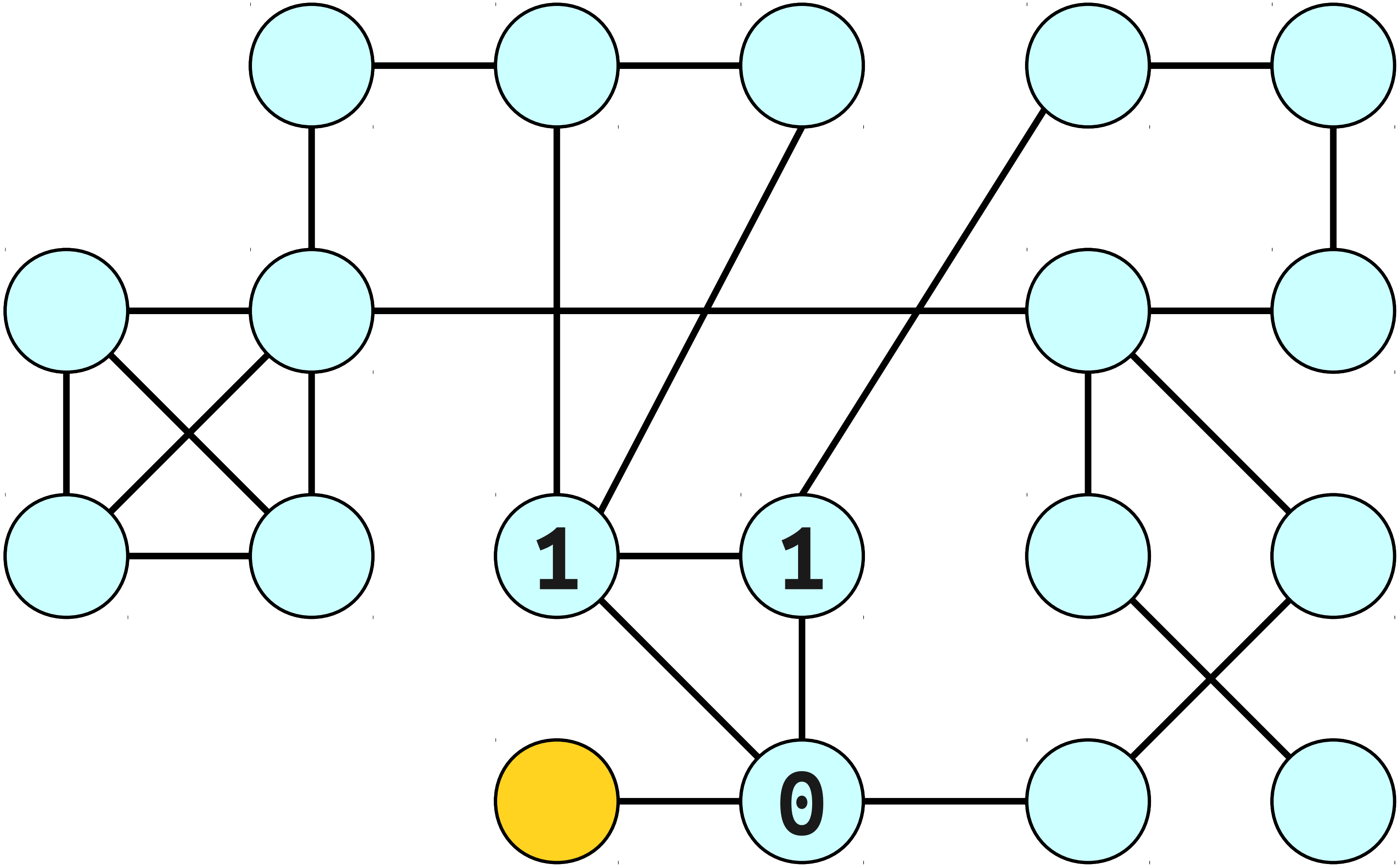
An Inefficient Algorithm



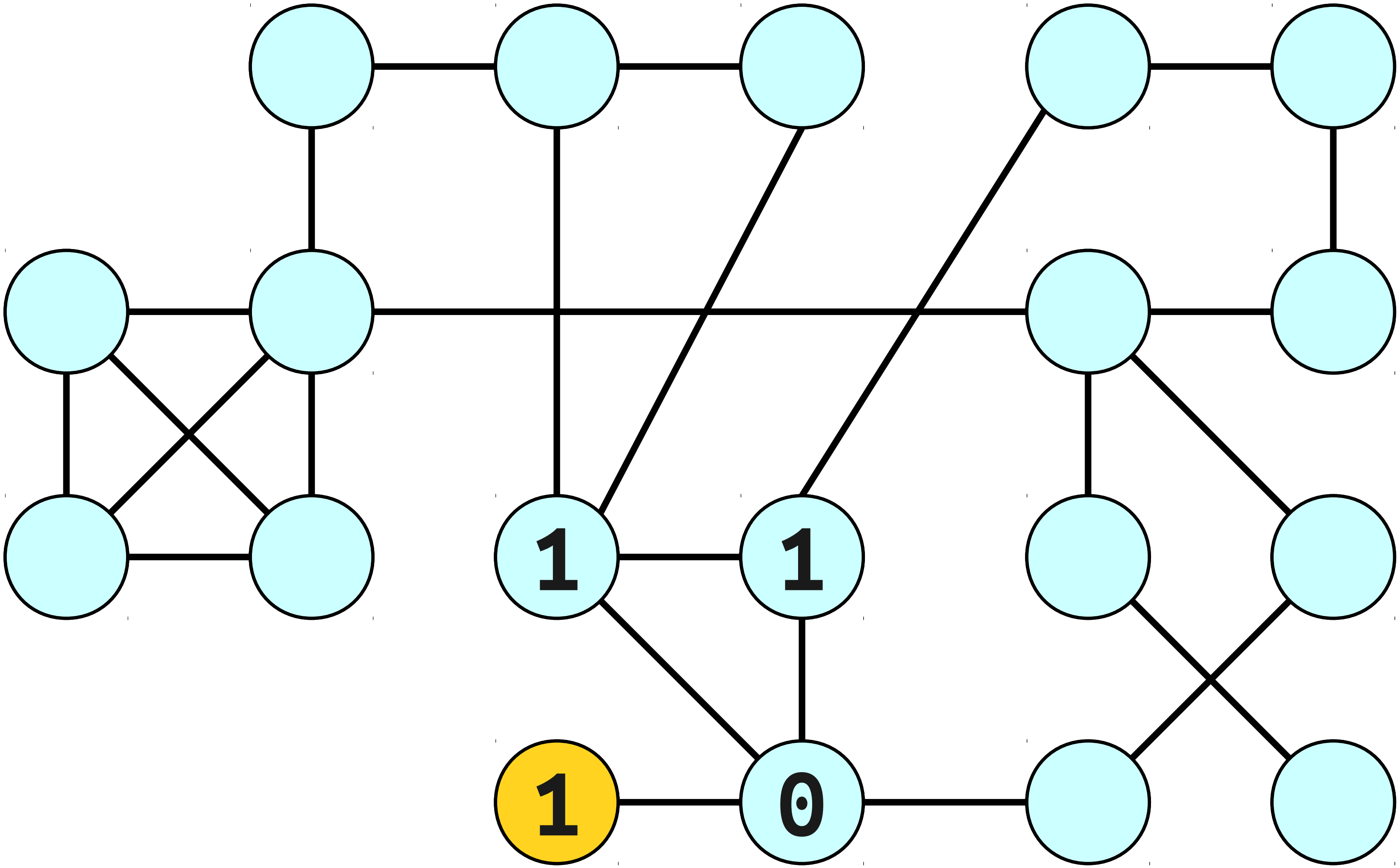
An Inefficient Algorithm



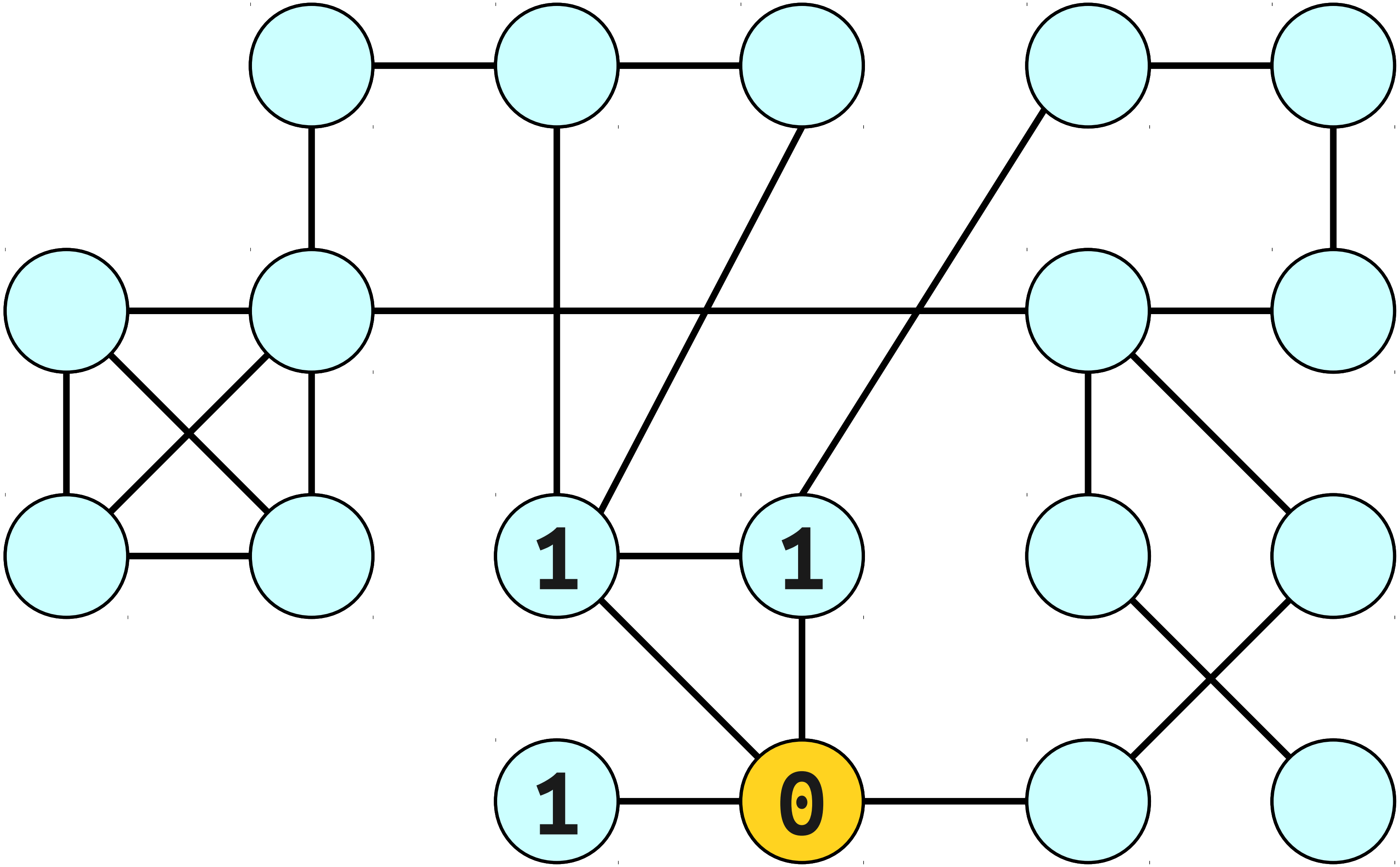
An Inefficient Algorithm



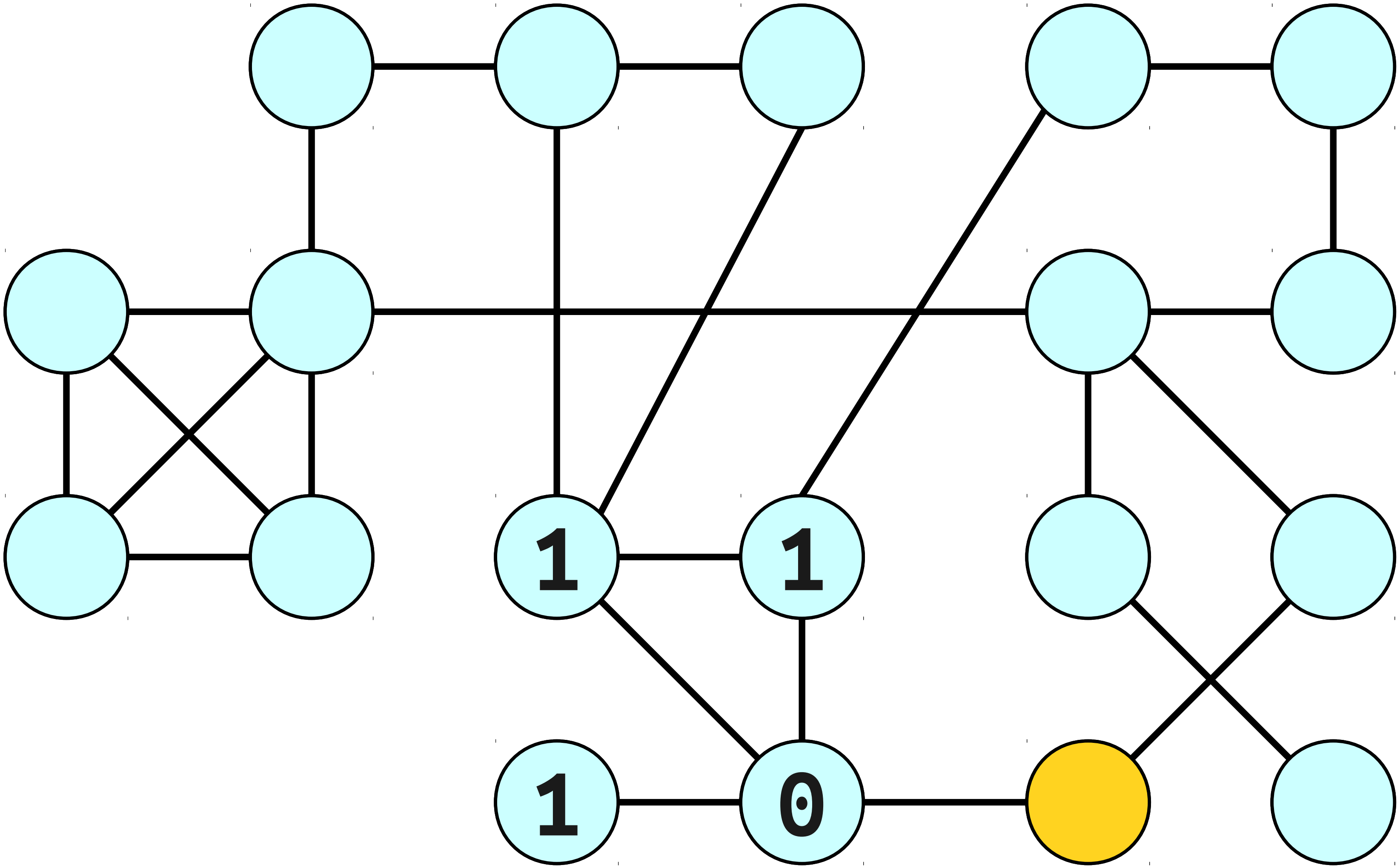
An Inefficient Algorithm



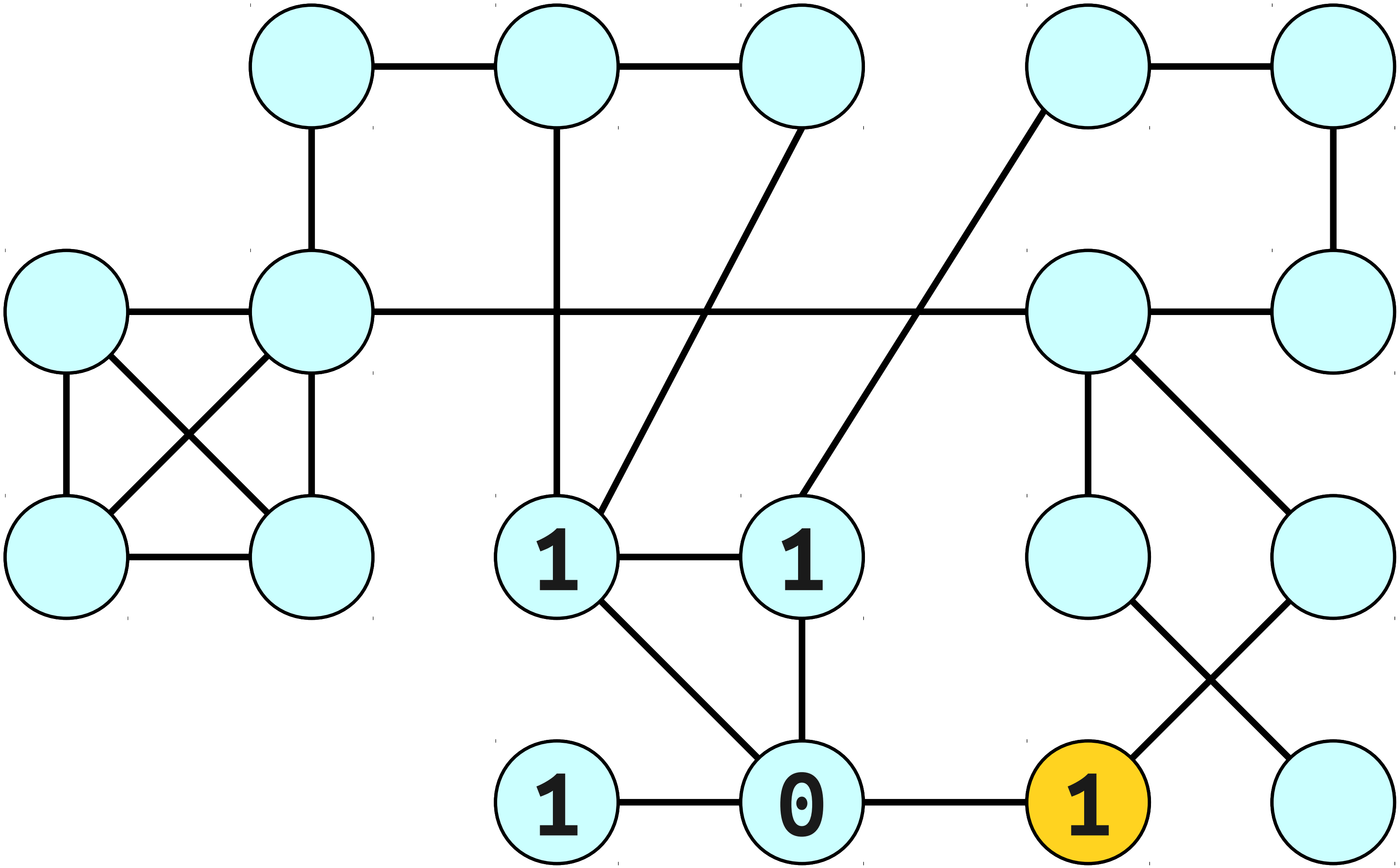
An Inefficient Algorithm



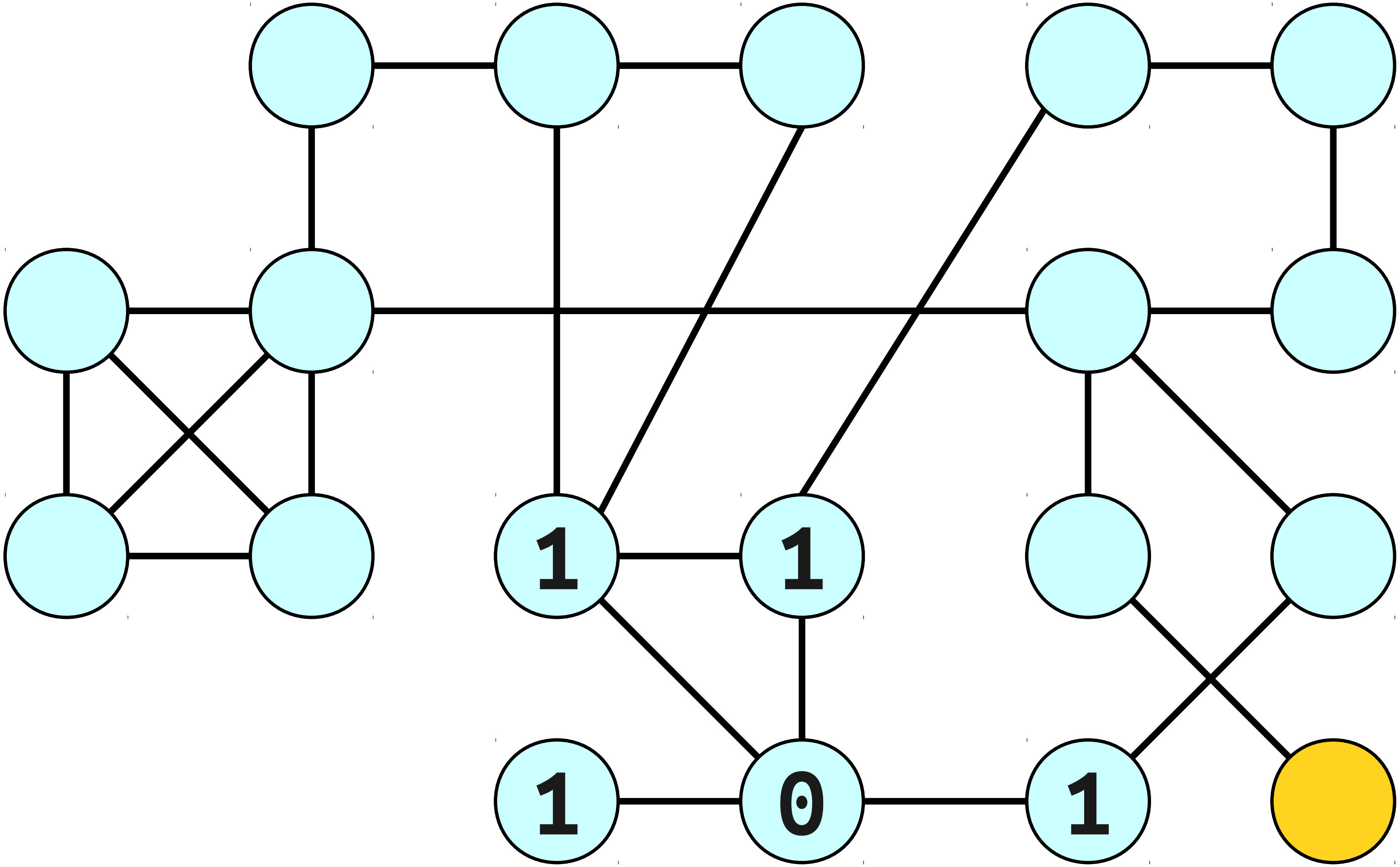
An Inefficient Algorithm



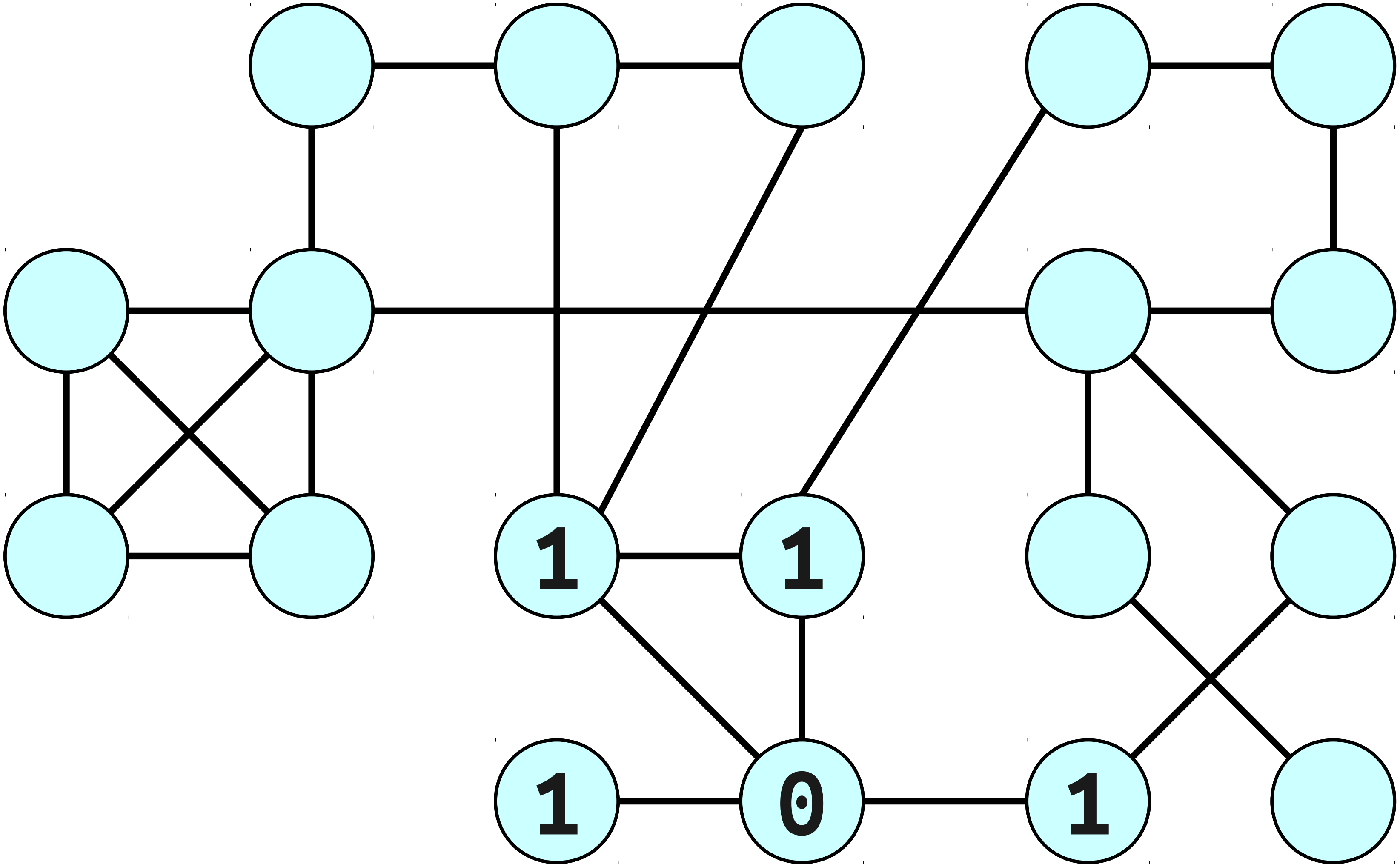
An Inefficient Algorithm



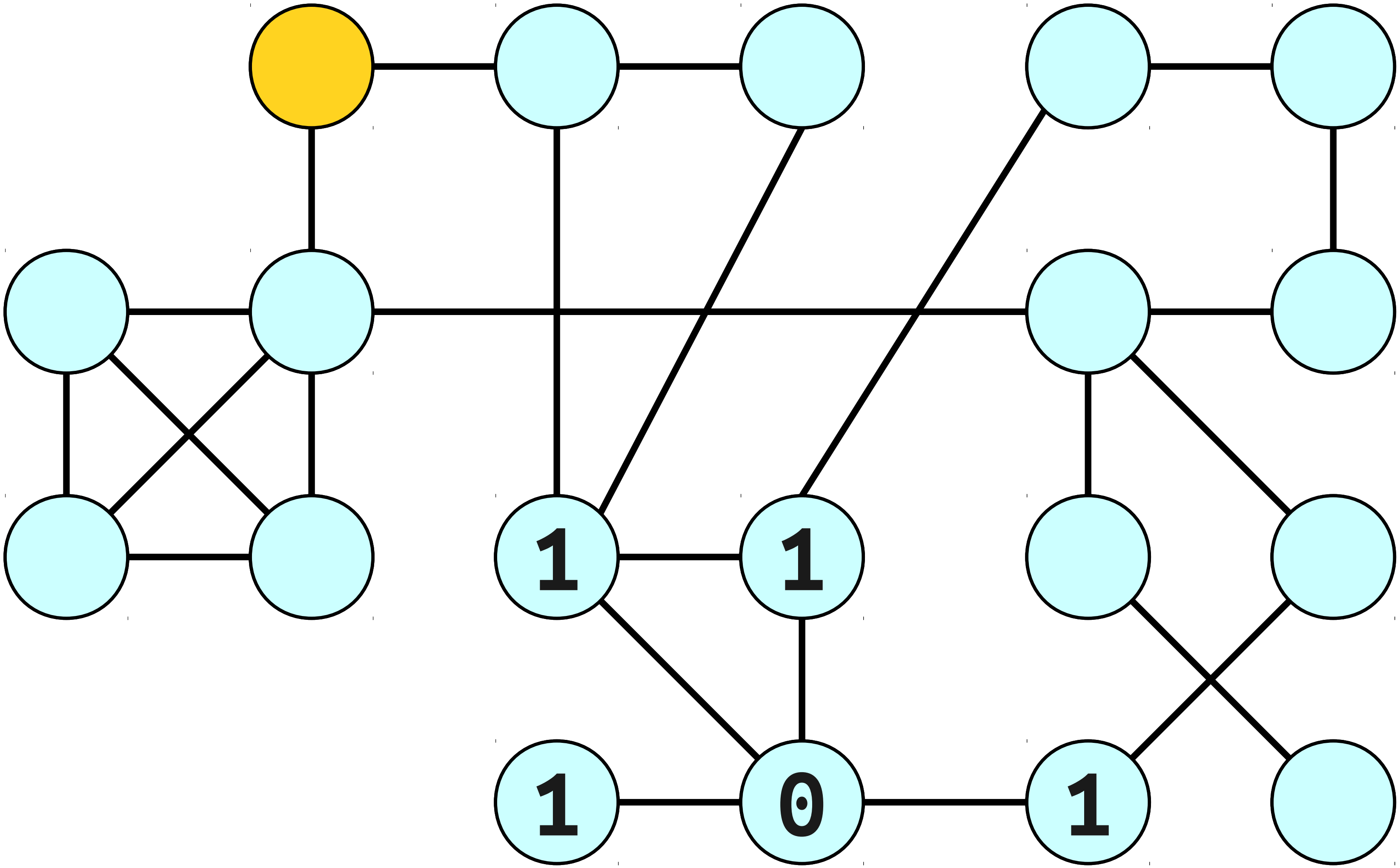
An Inefficient Algorithm



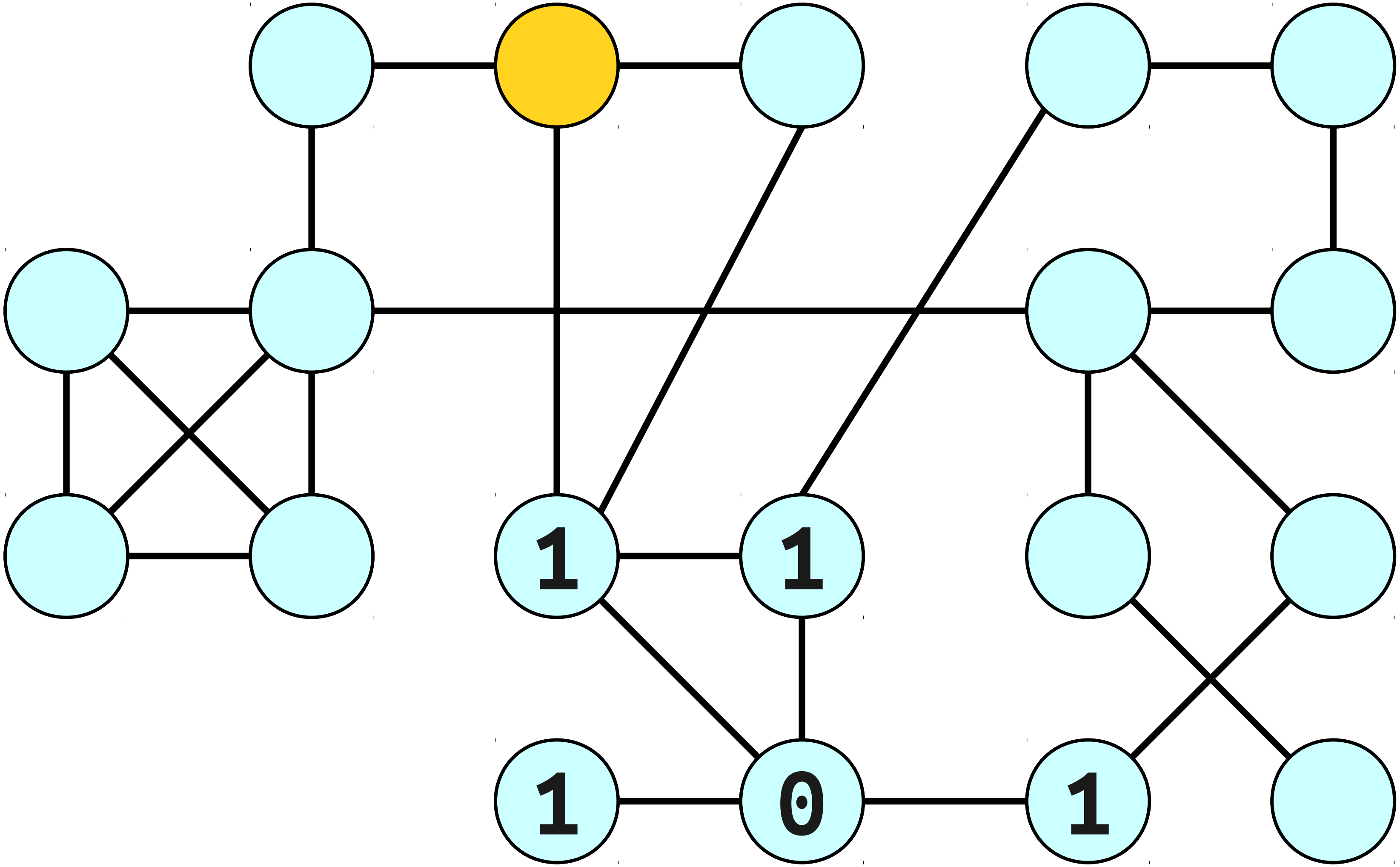
An Inefficient Algorithm



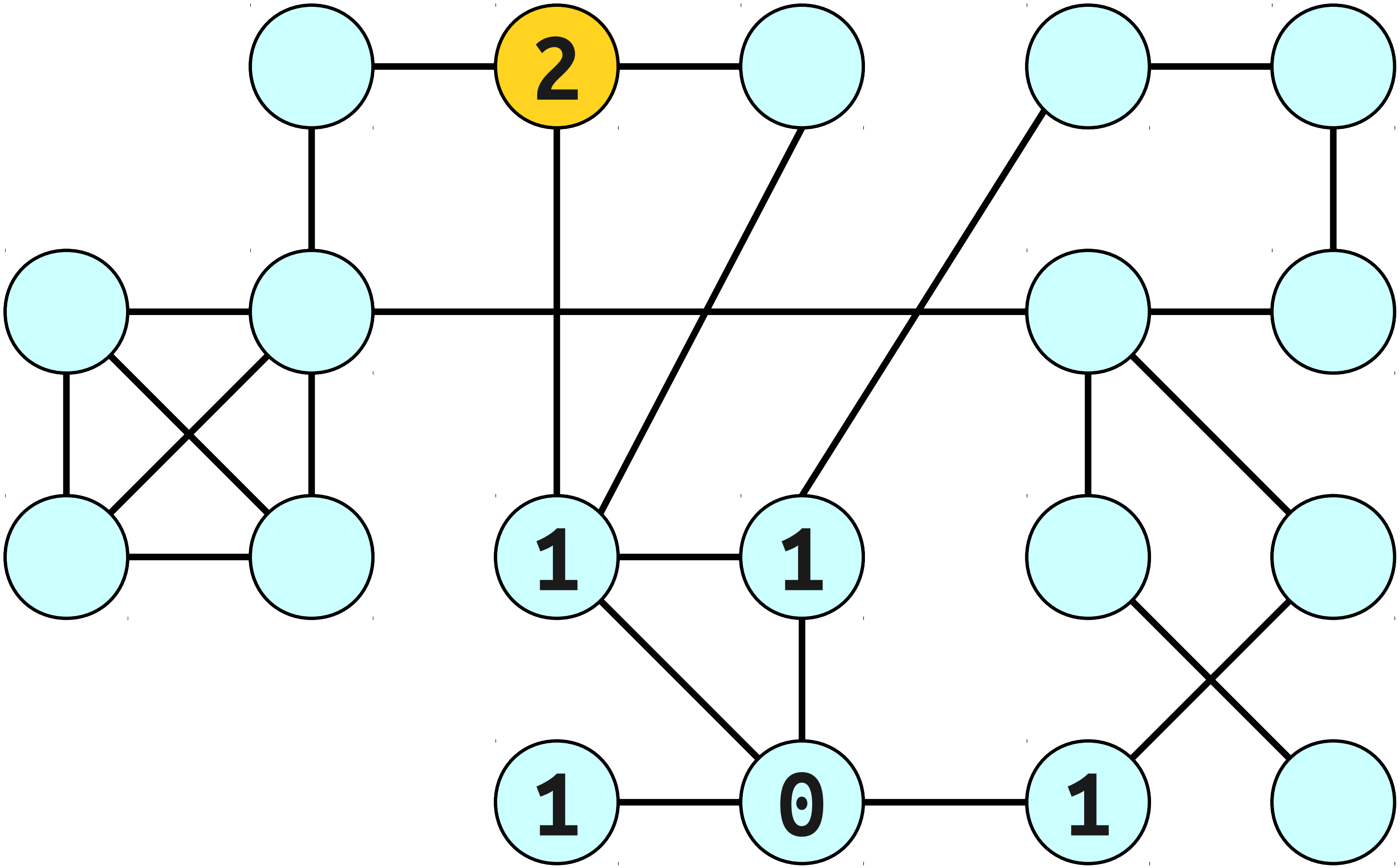
An Inefficient Algorithm



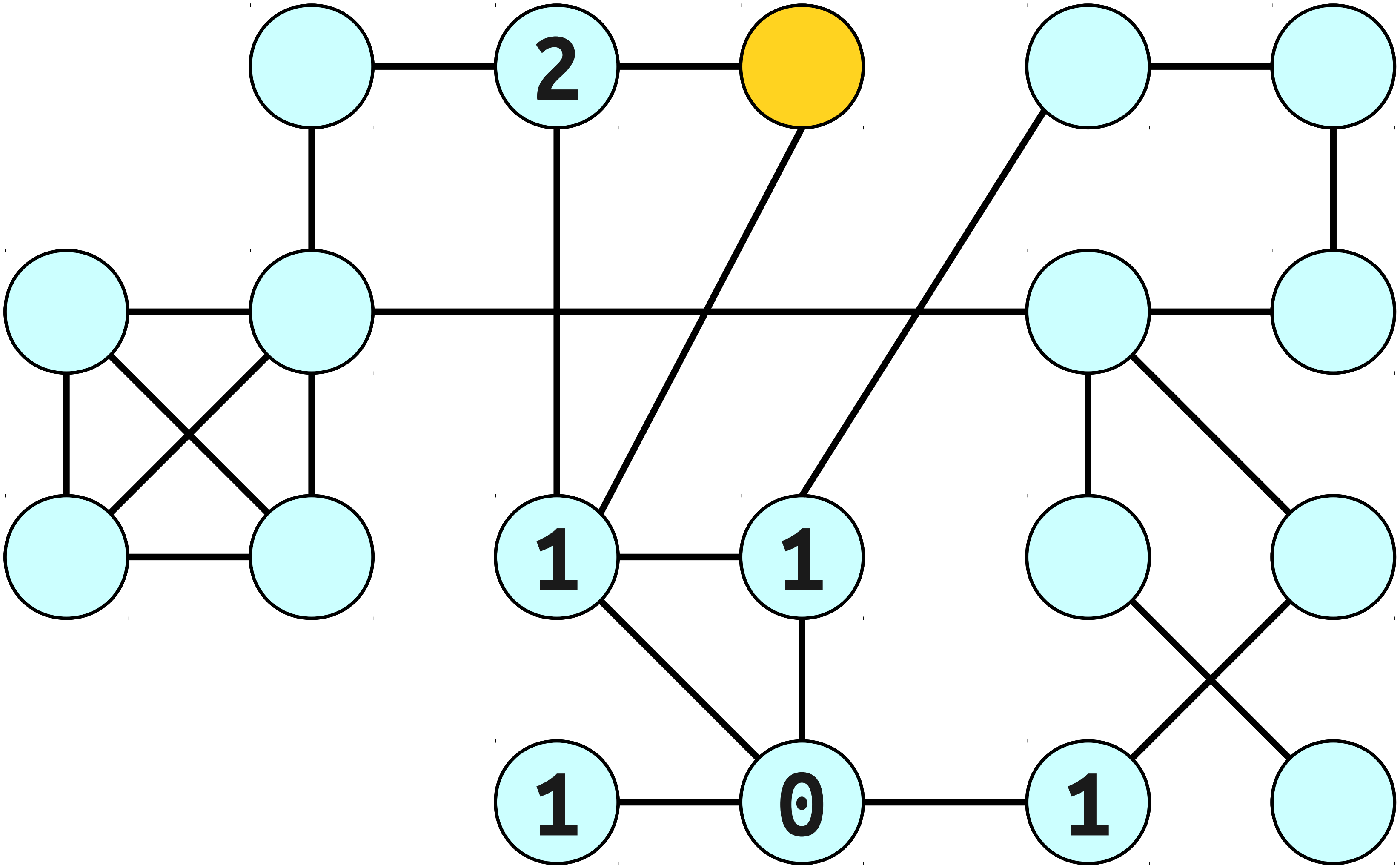
An Inefficient Algorithm



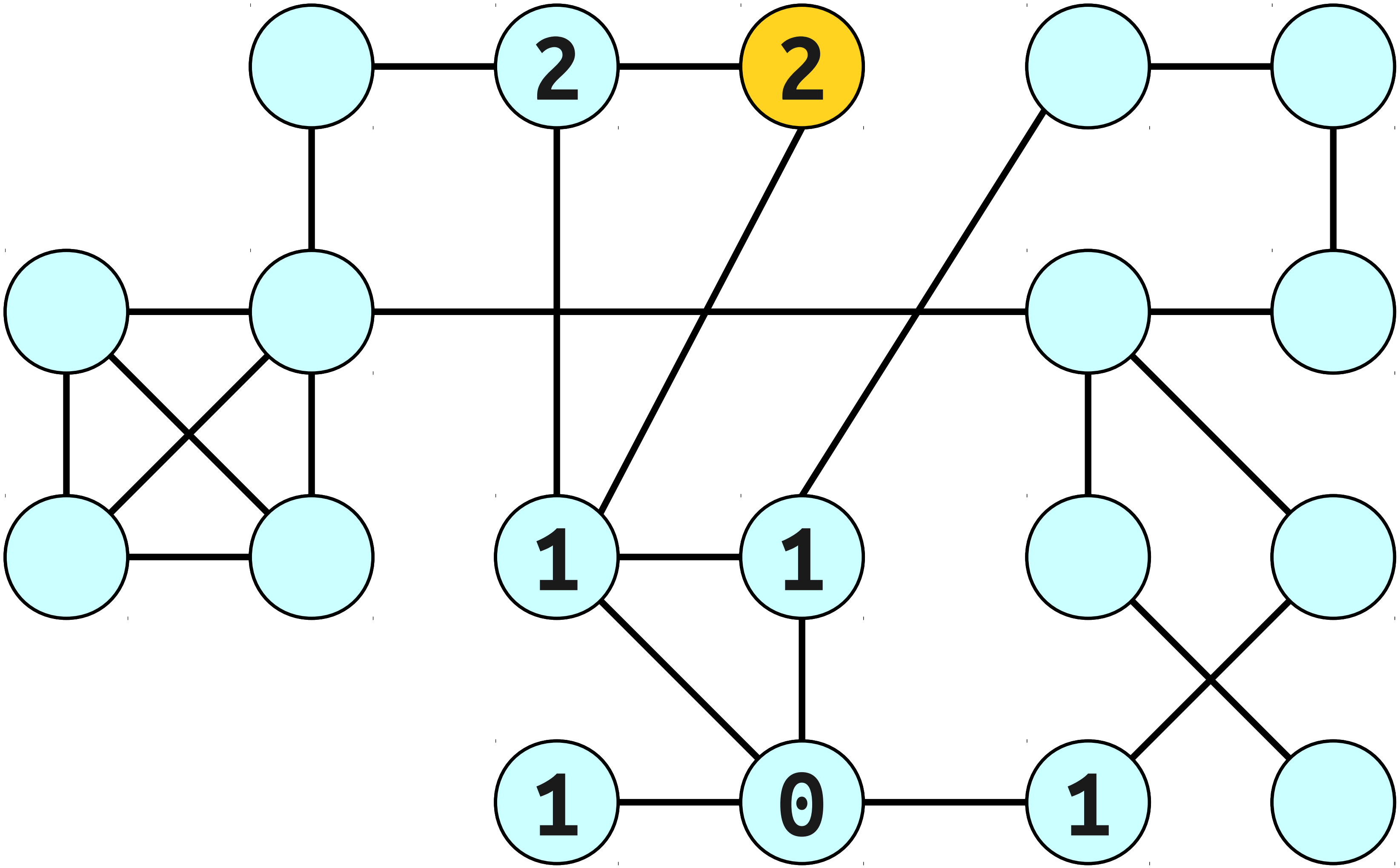
An Inefficient Algorithm



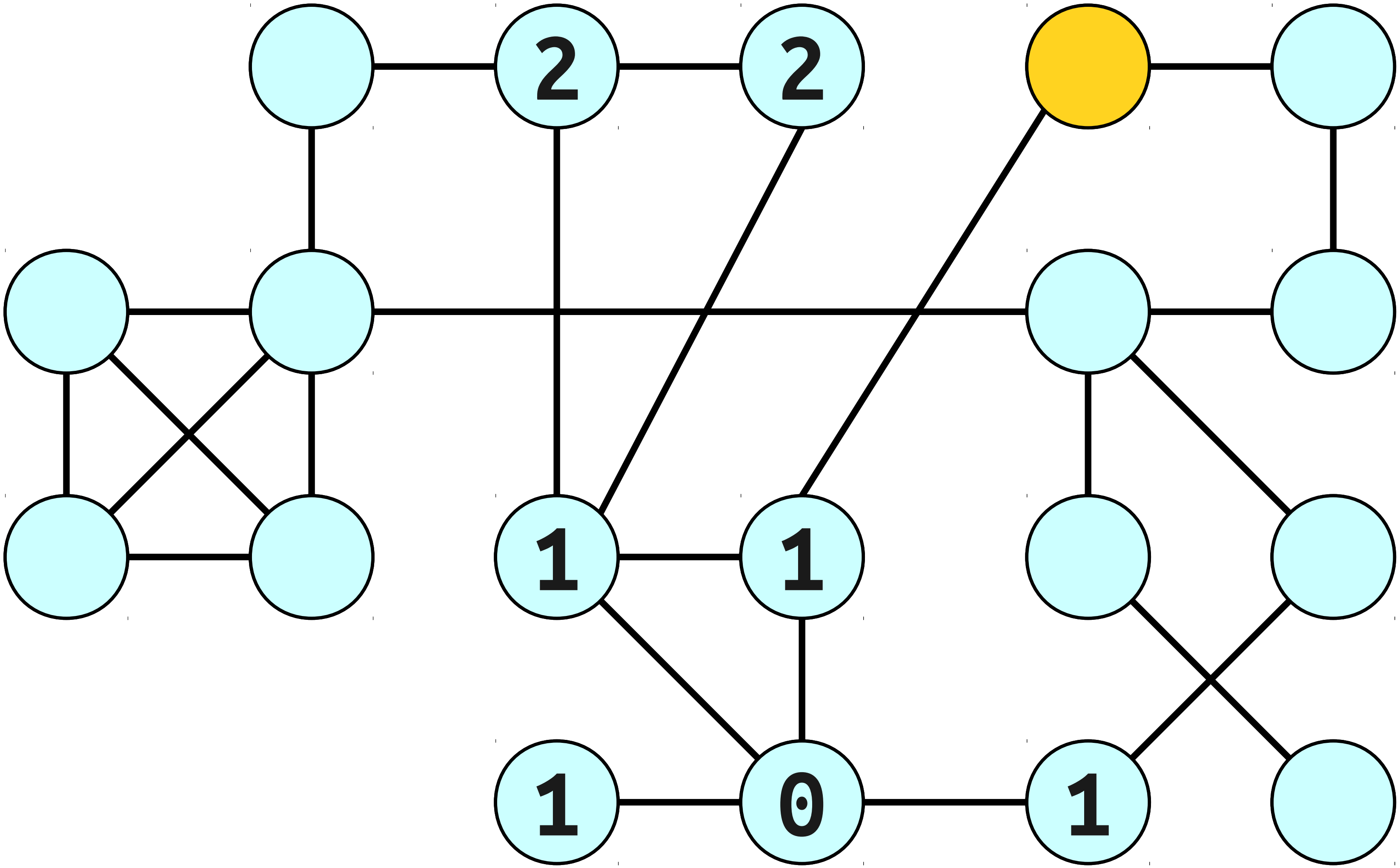
An Inefficient Algorithm



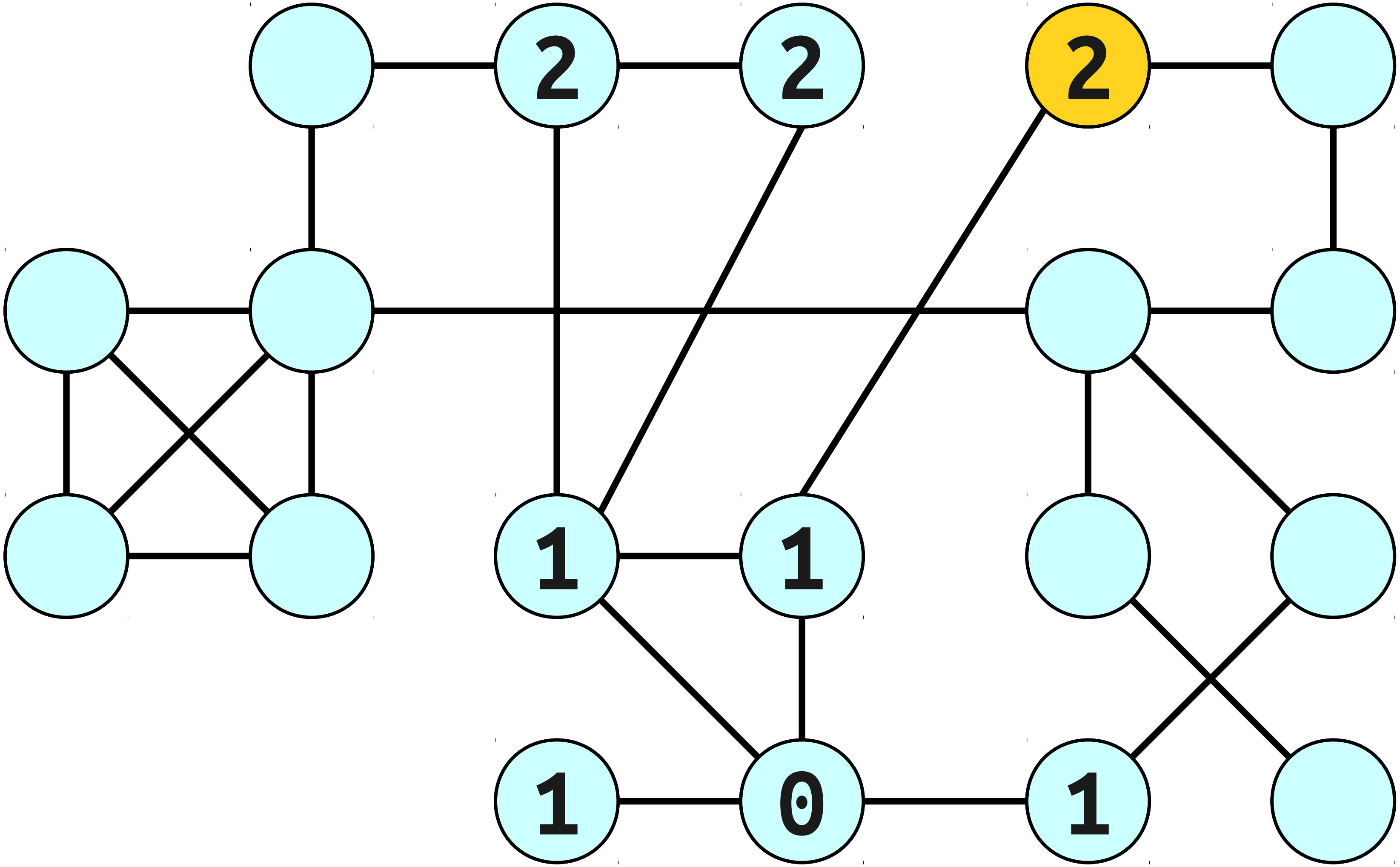
An Inefficient Algorithm



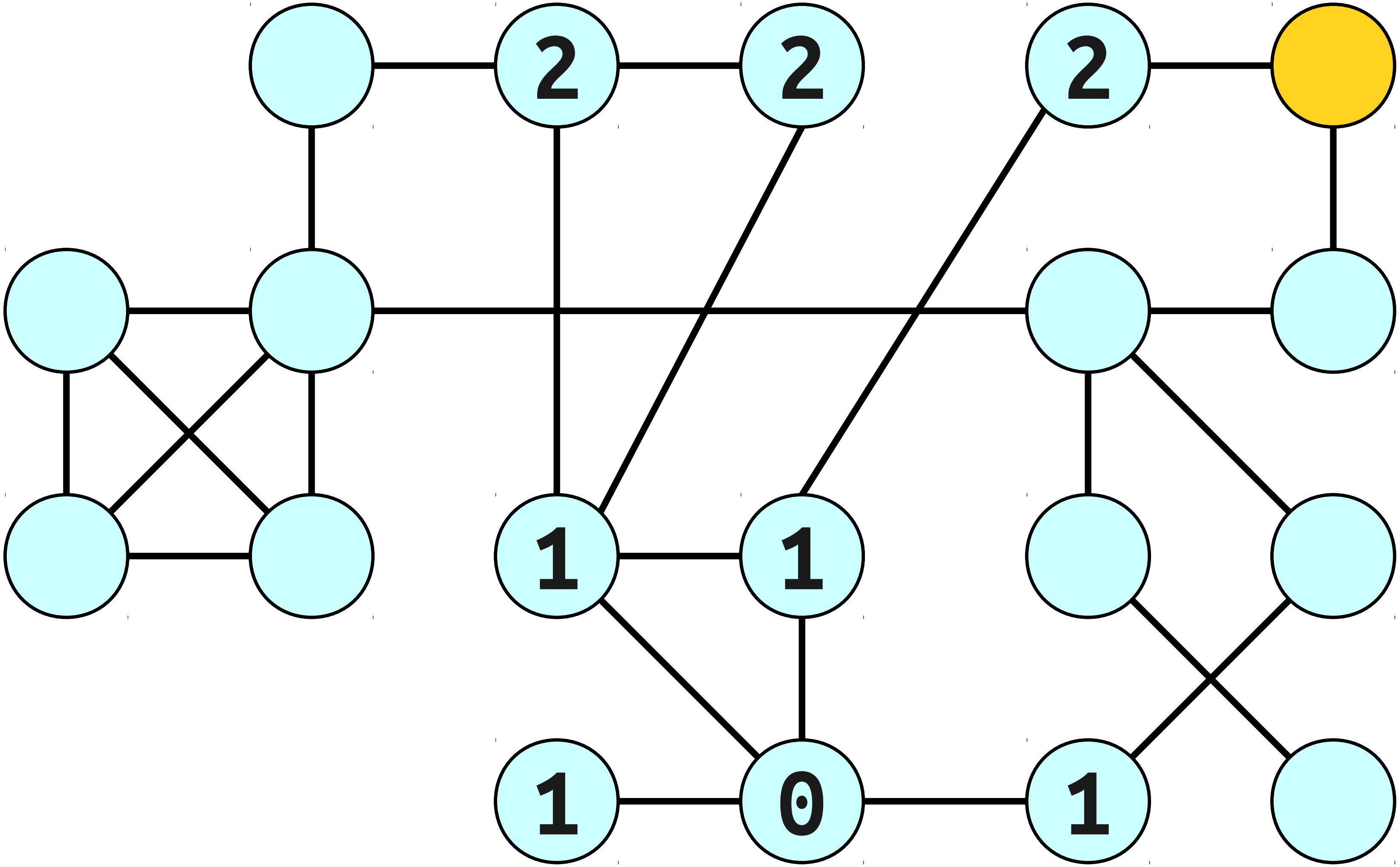
An Inefficient Algorithm



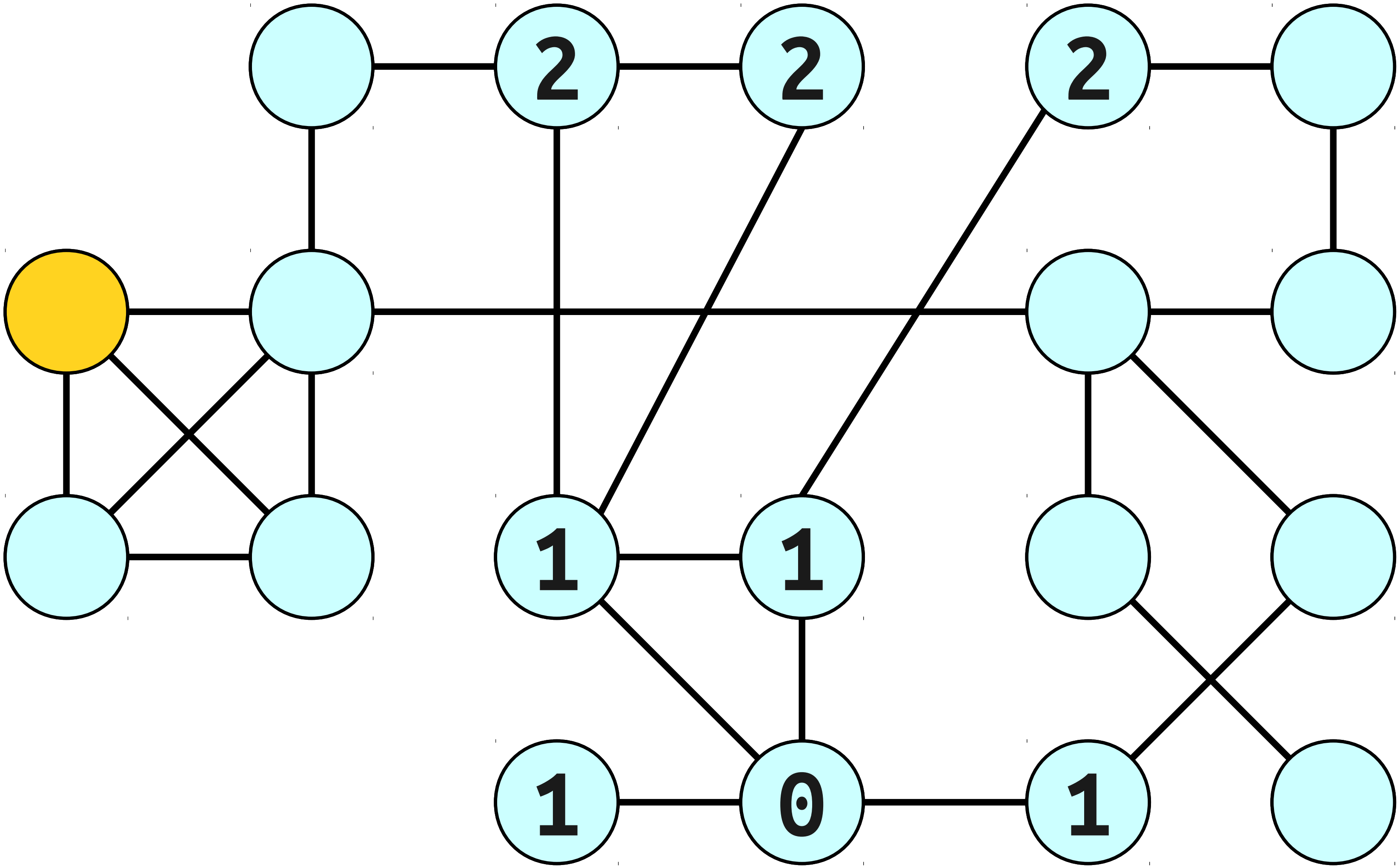
An Inefficient Algorithm



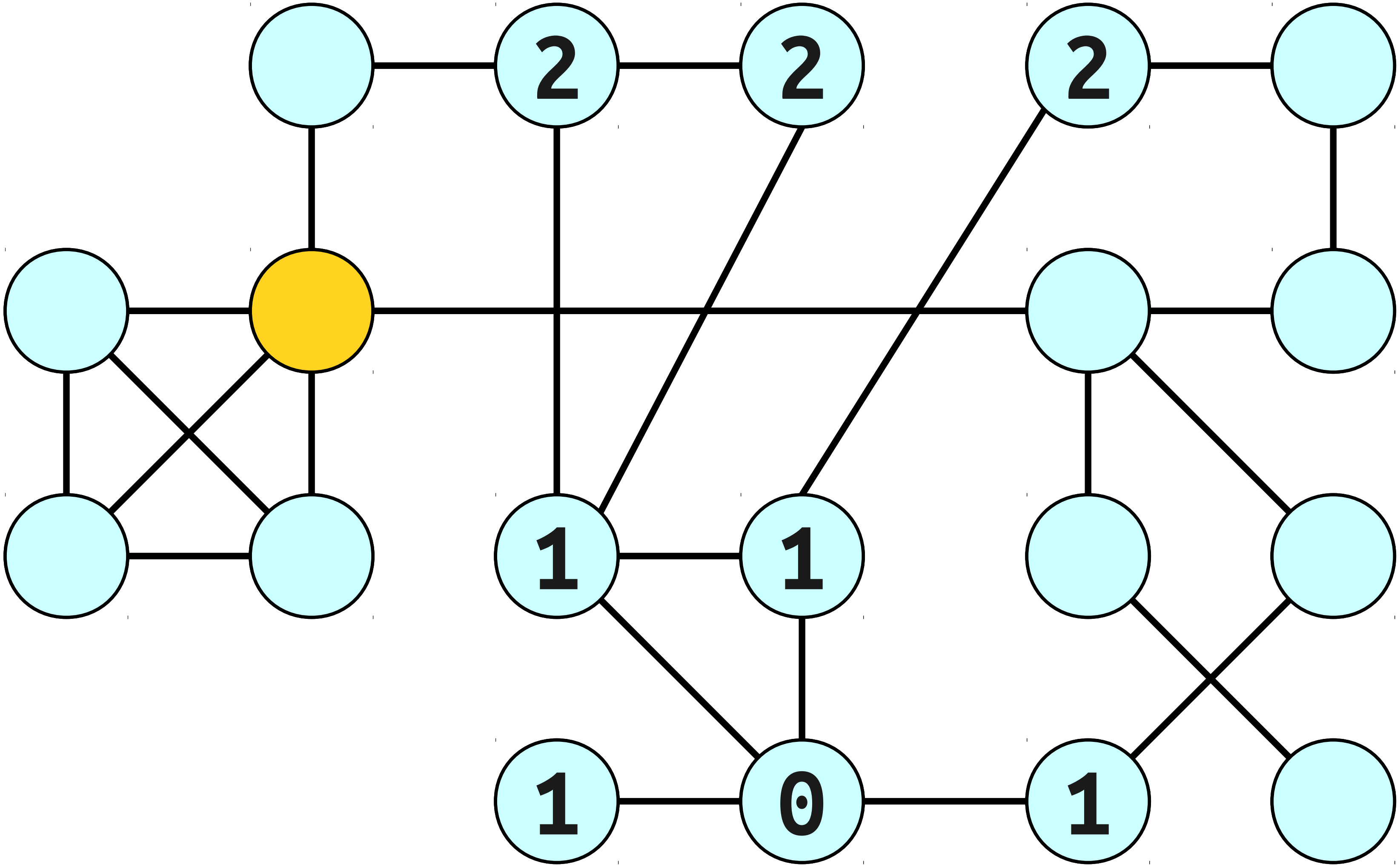
An Inefficient Algorithm



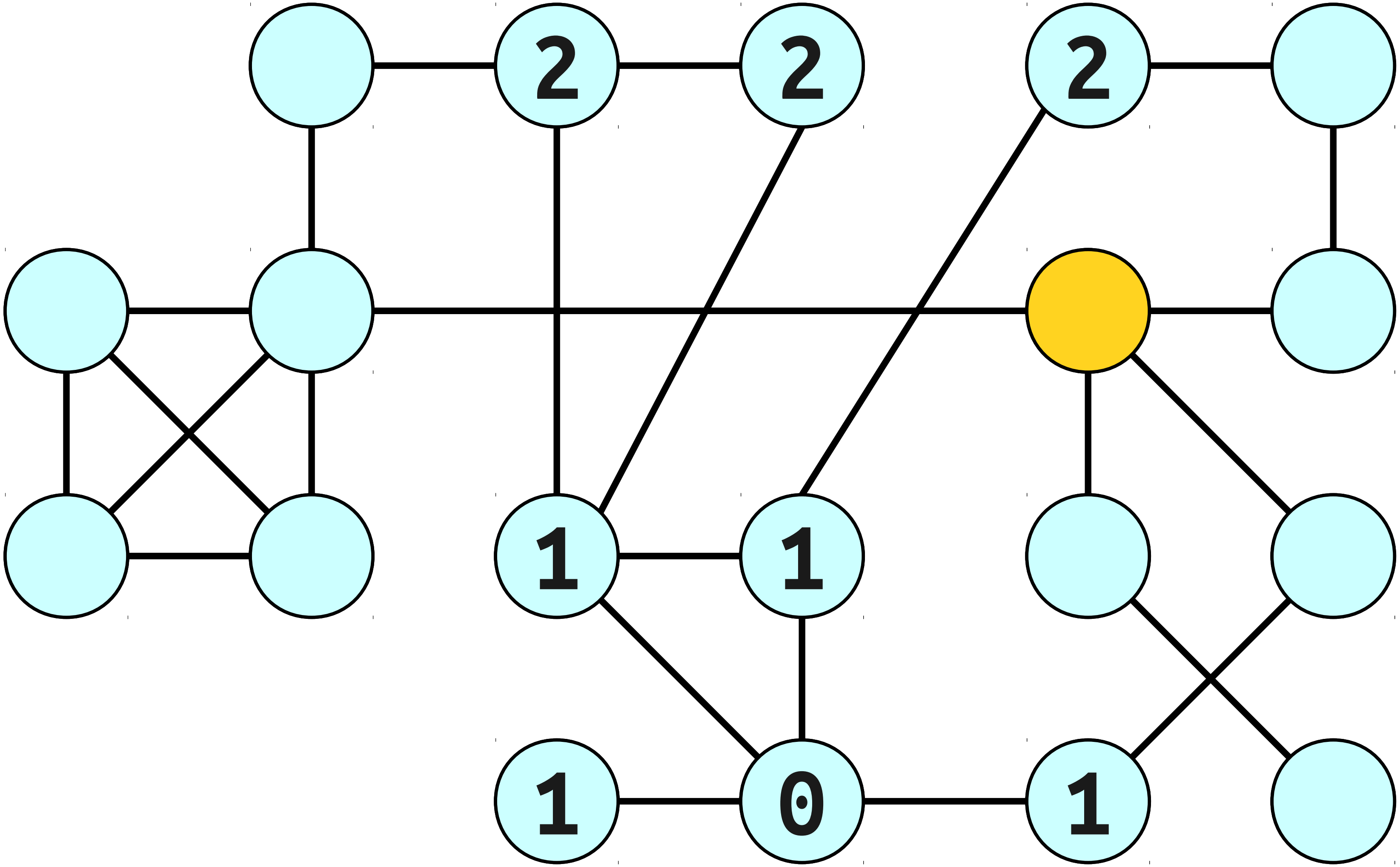
An Inefficient Algorithm



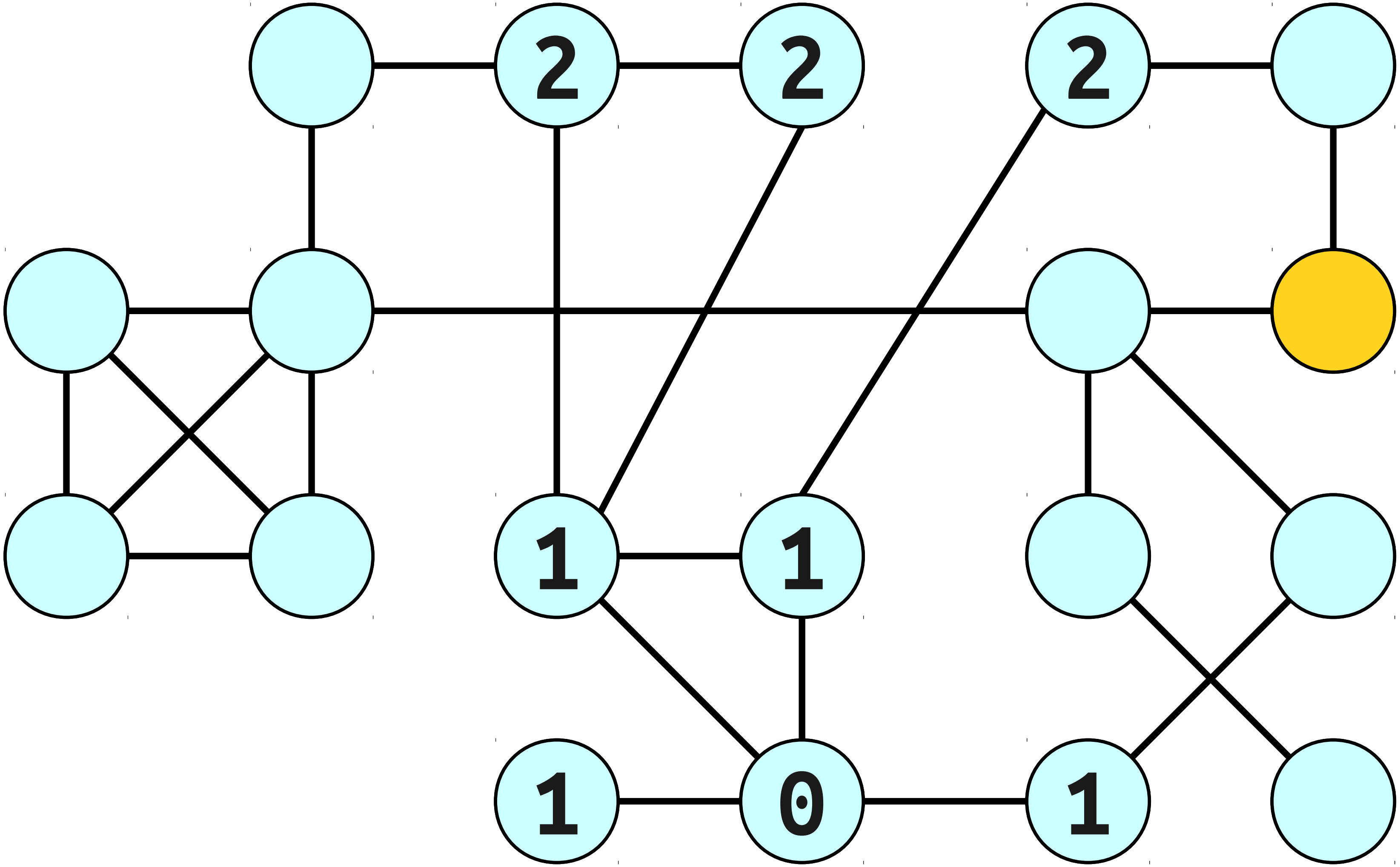
An Inefficient Algorithm



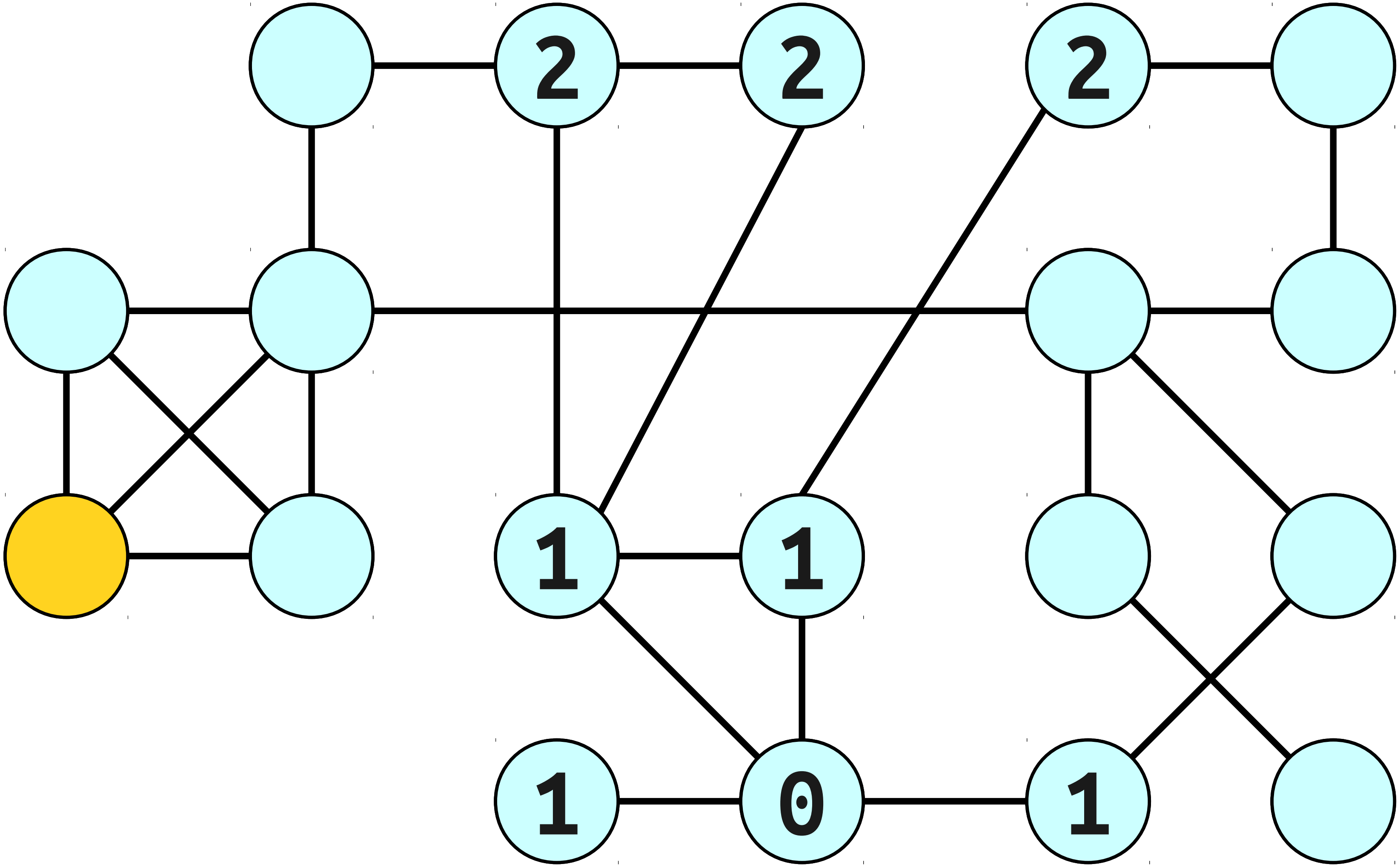
An Inefficient Algorithm



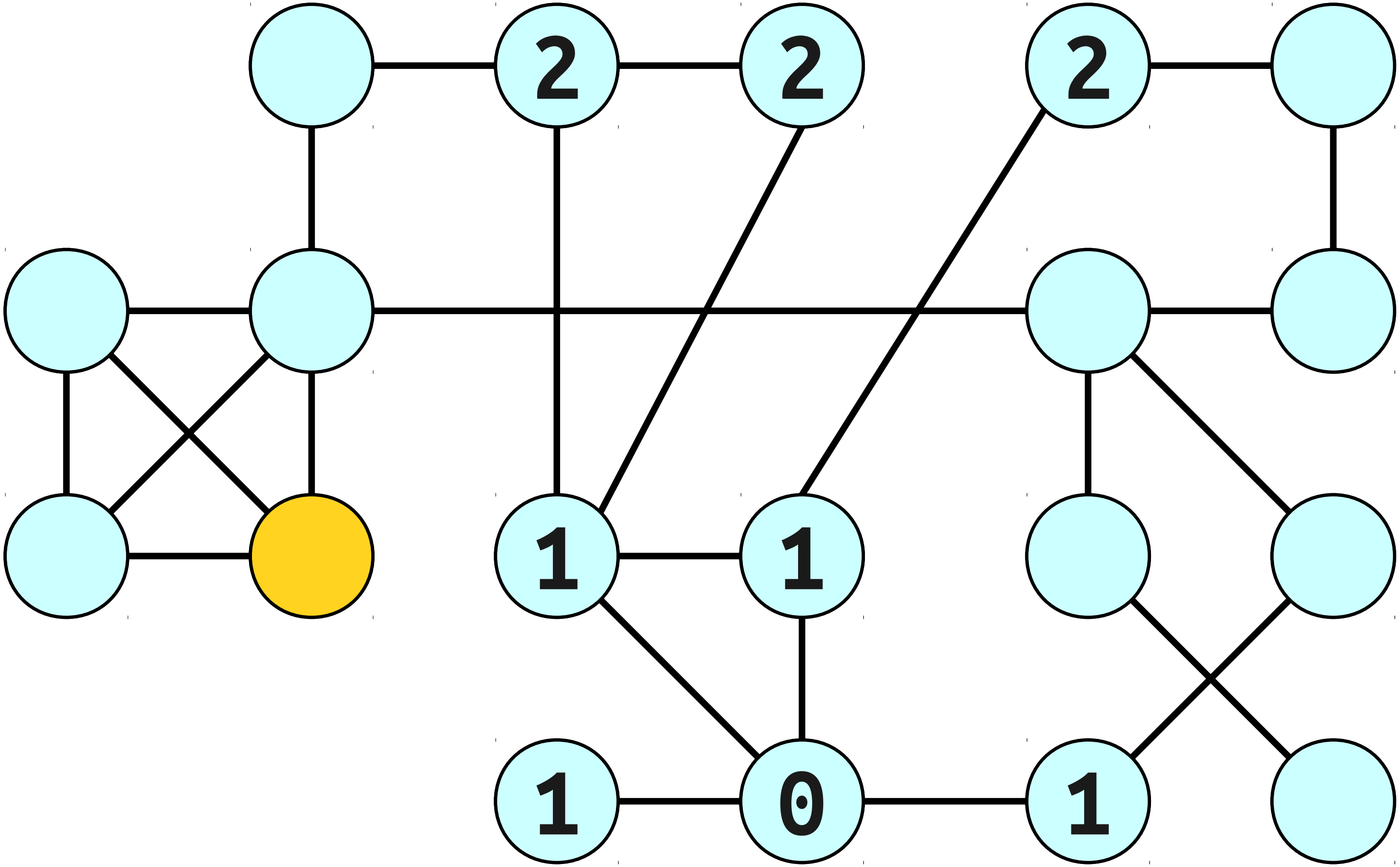
An Inefficient Algorithm



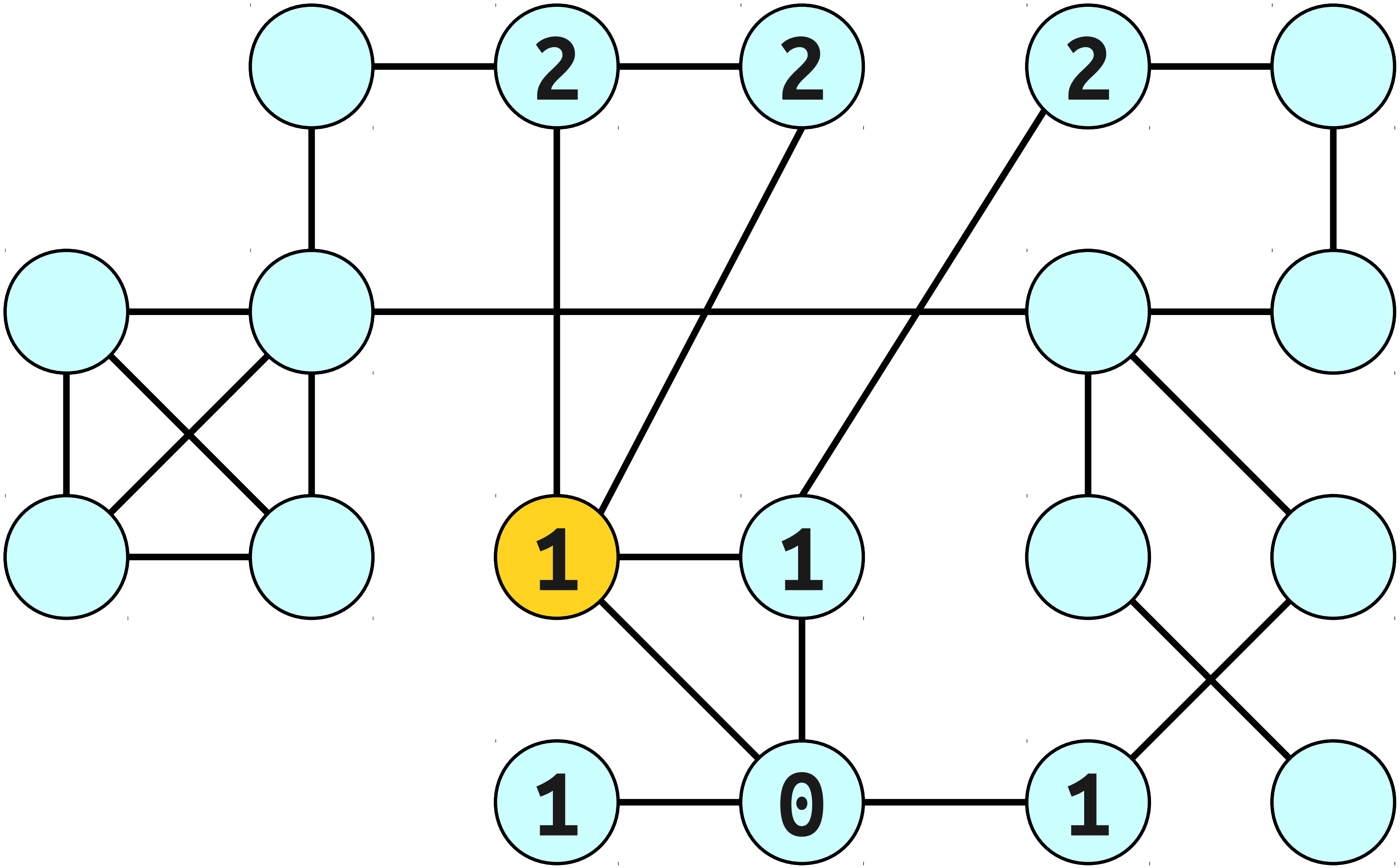
An Inefficient Algorithm



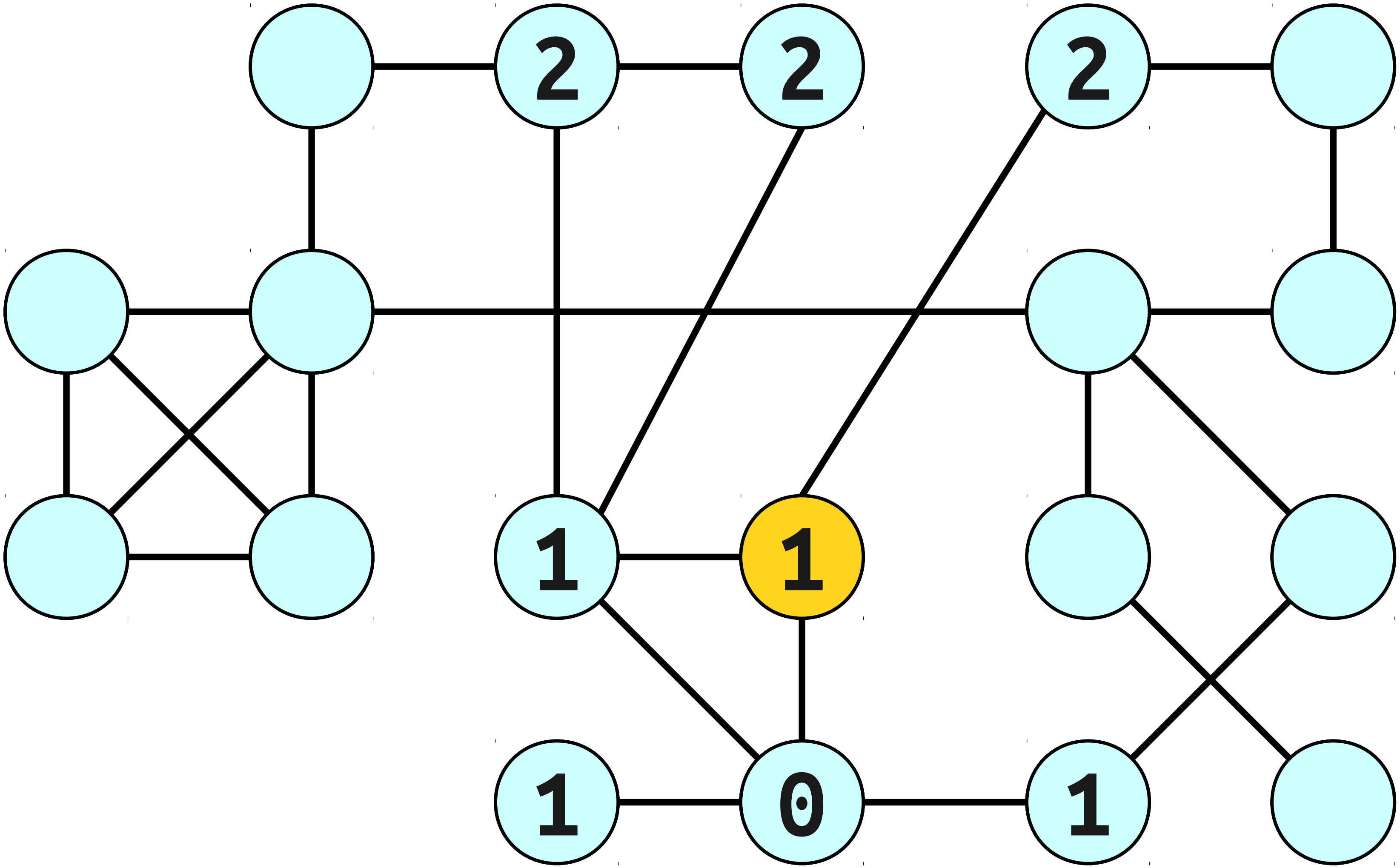
An Inefficient Algorithm



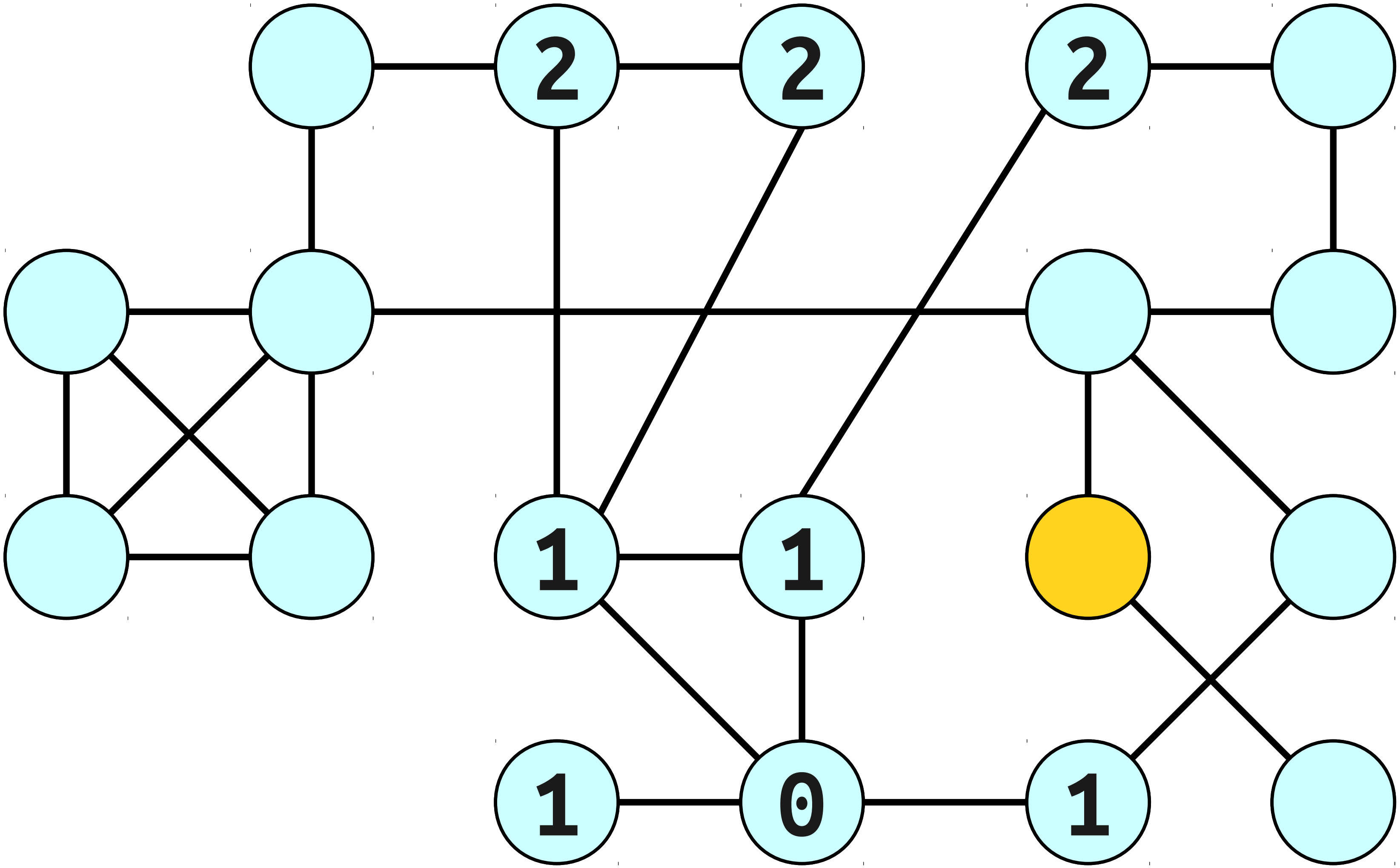
An Inefficient Algorithm



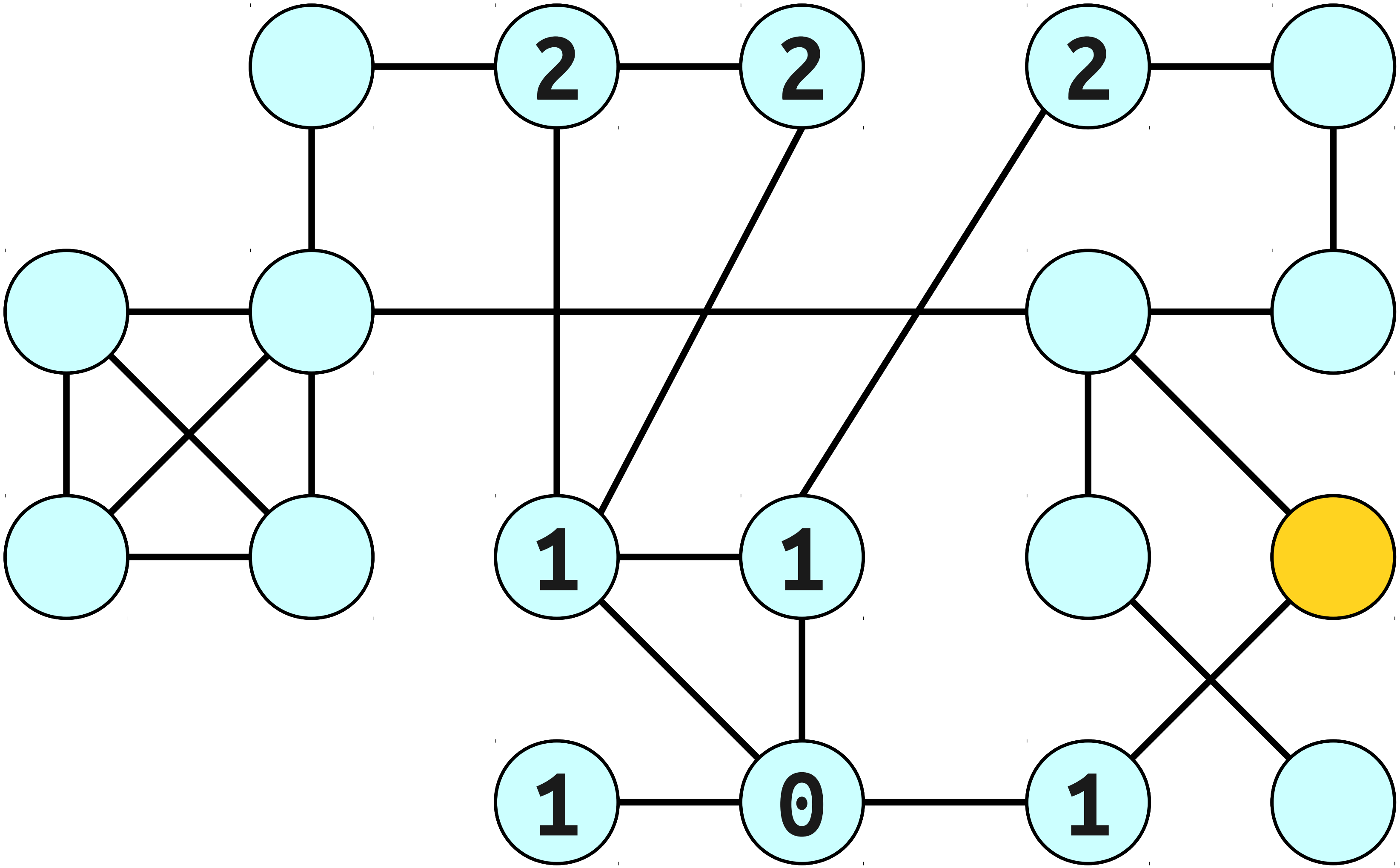
An Inefficient Algorithm



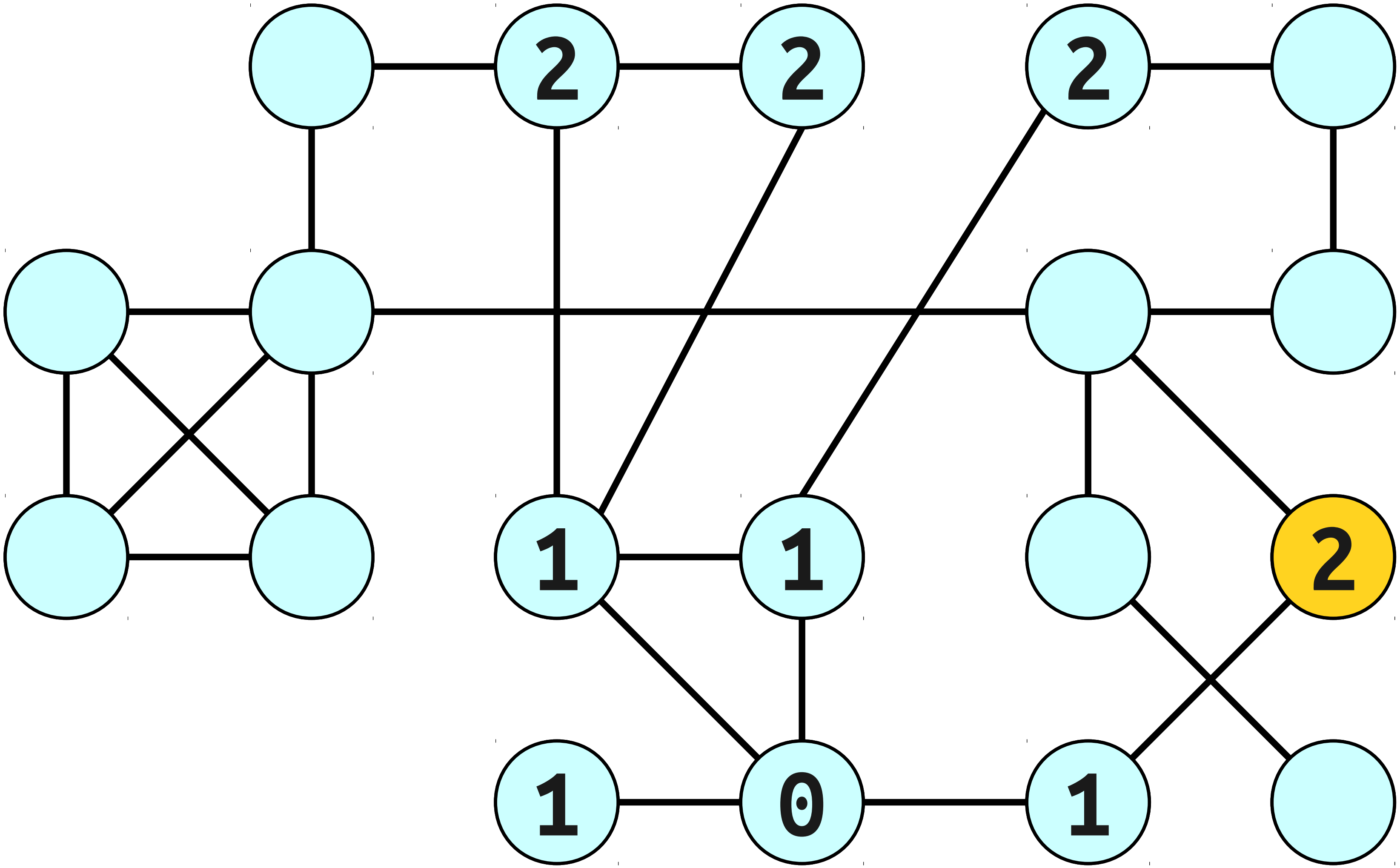
An Inefficient Algorithm



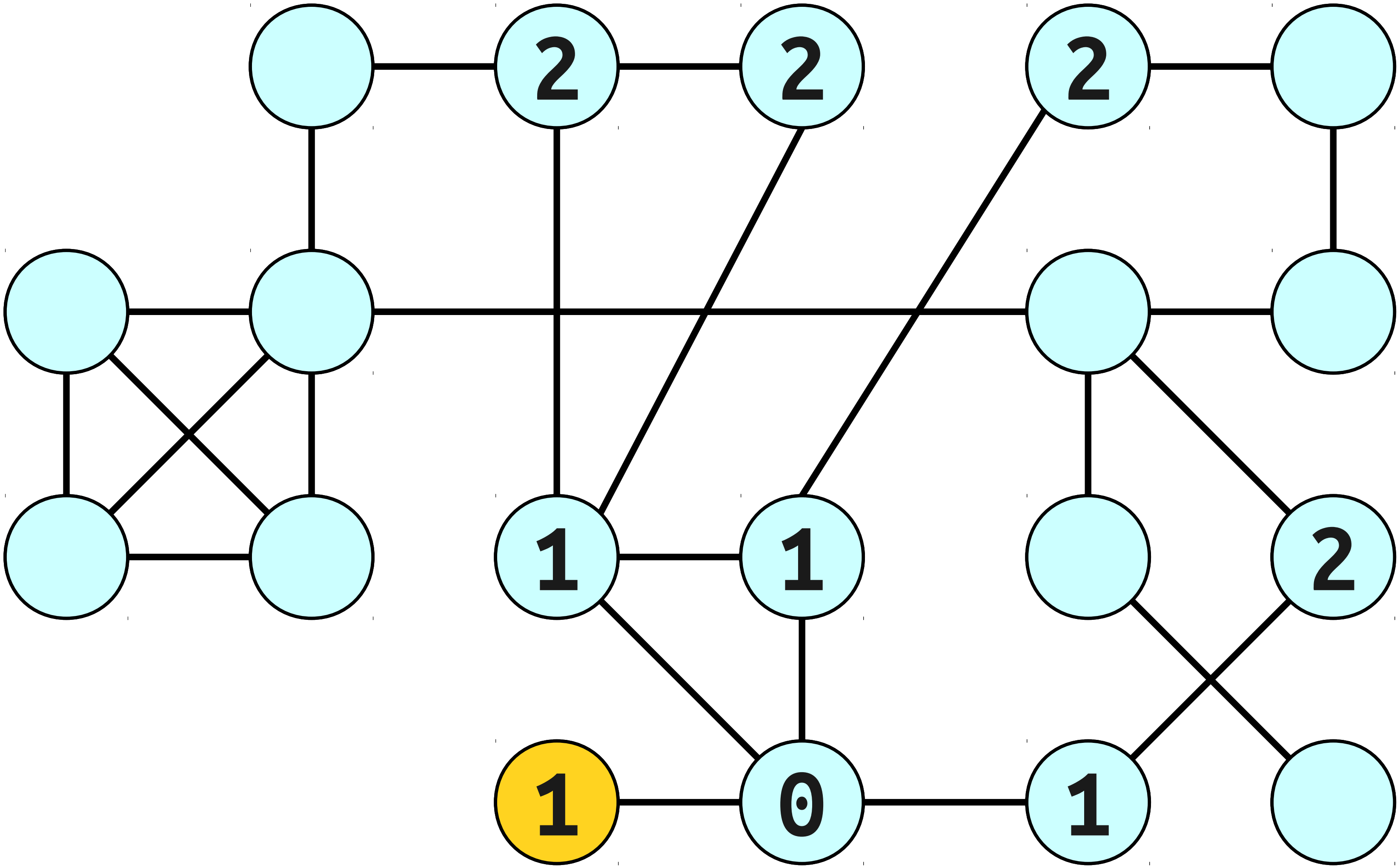
An Inefficient Algorithm



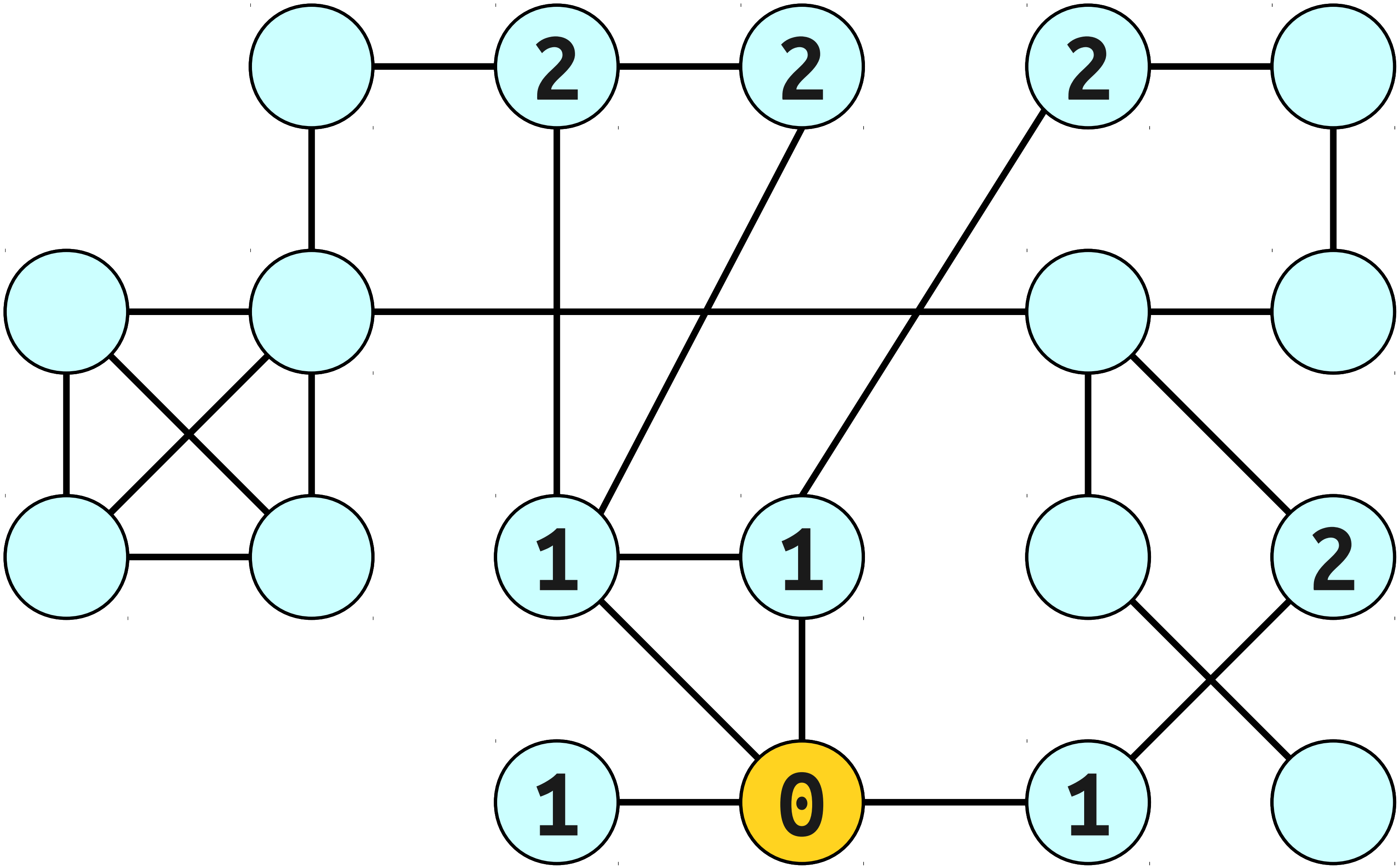
An Inefficient Algorithm



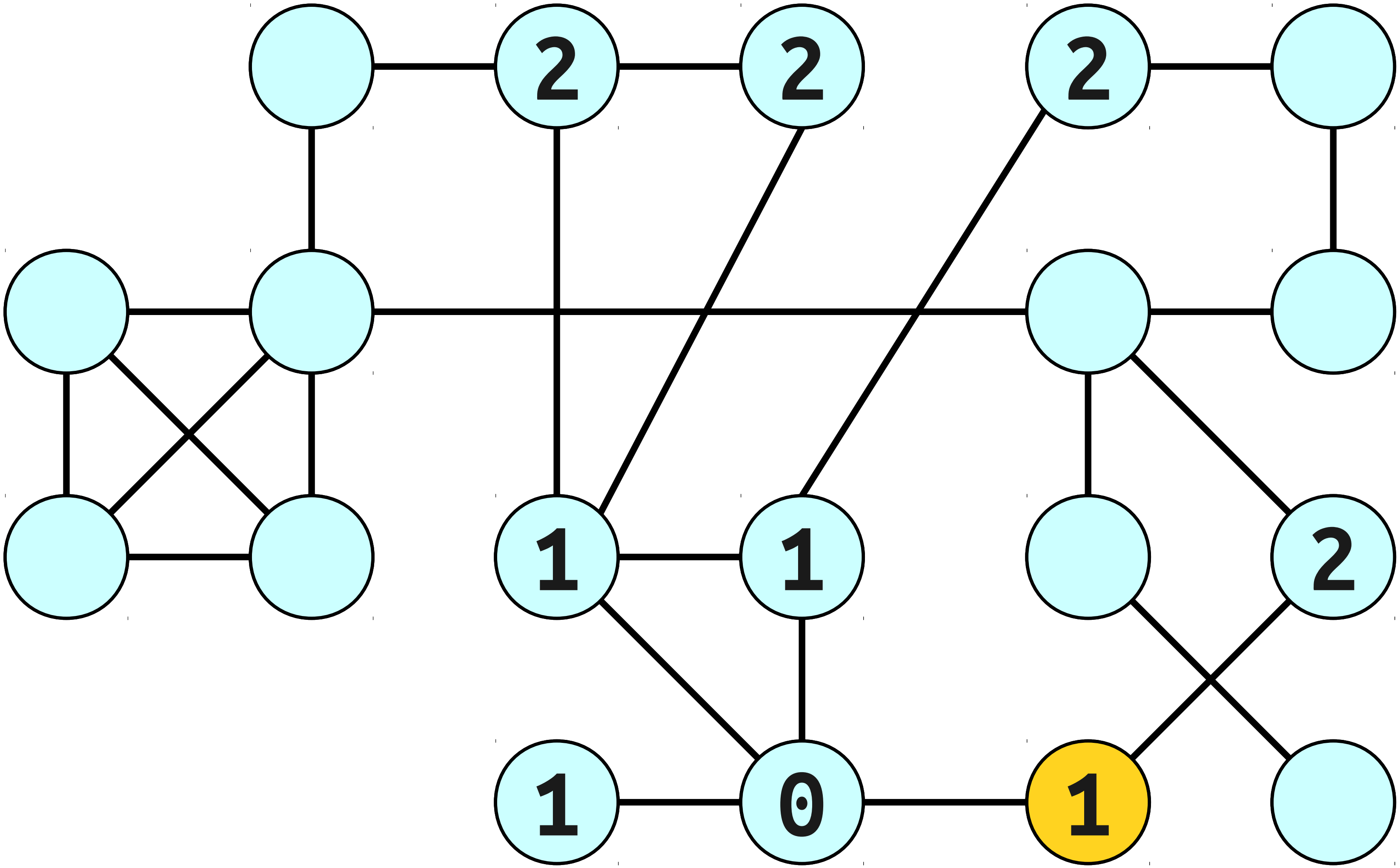
An Inefficient Algorithm



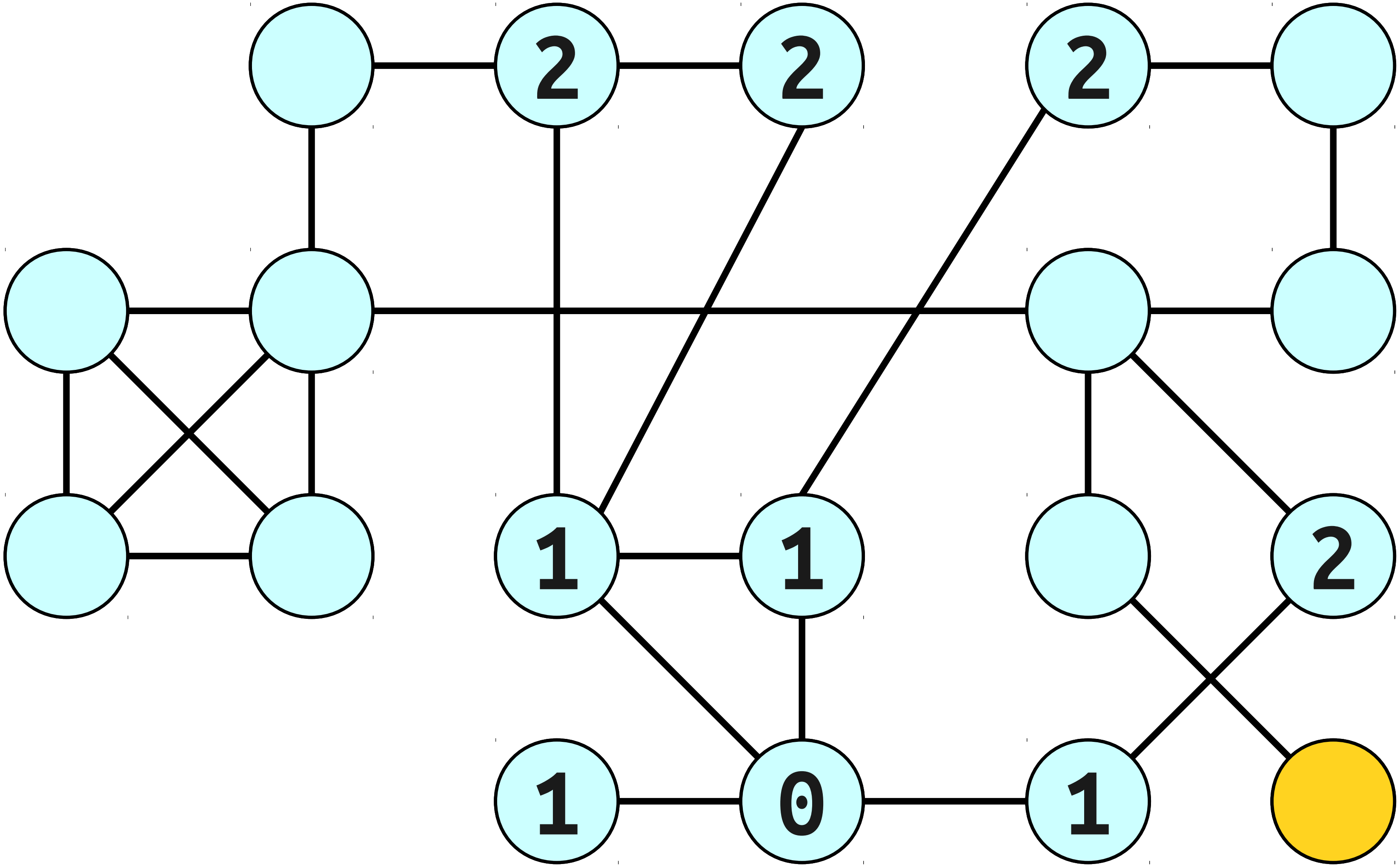
An Inefficient Algorithm



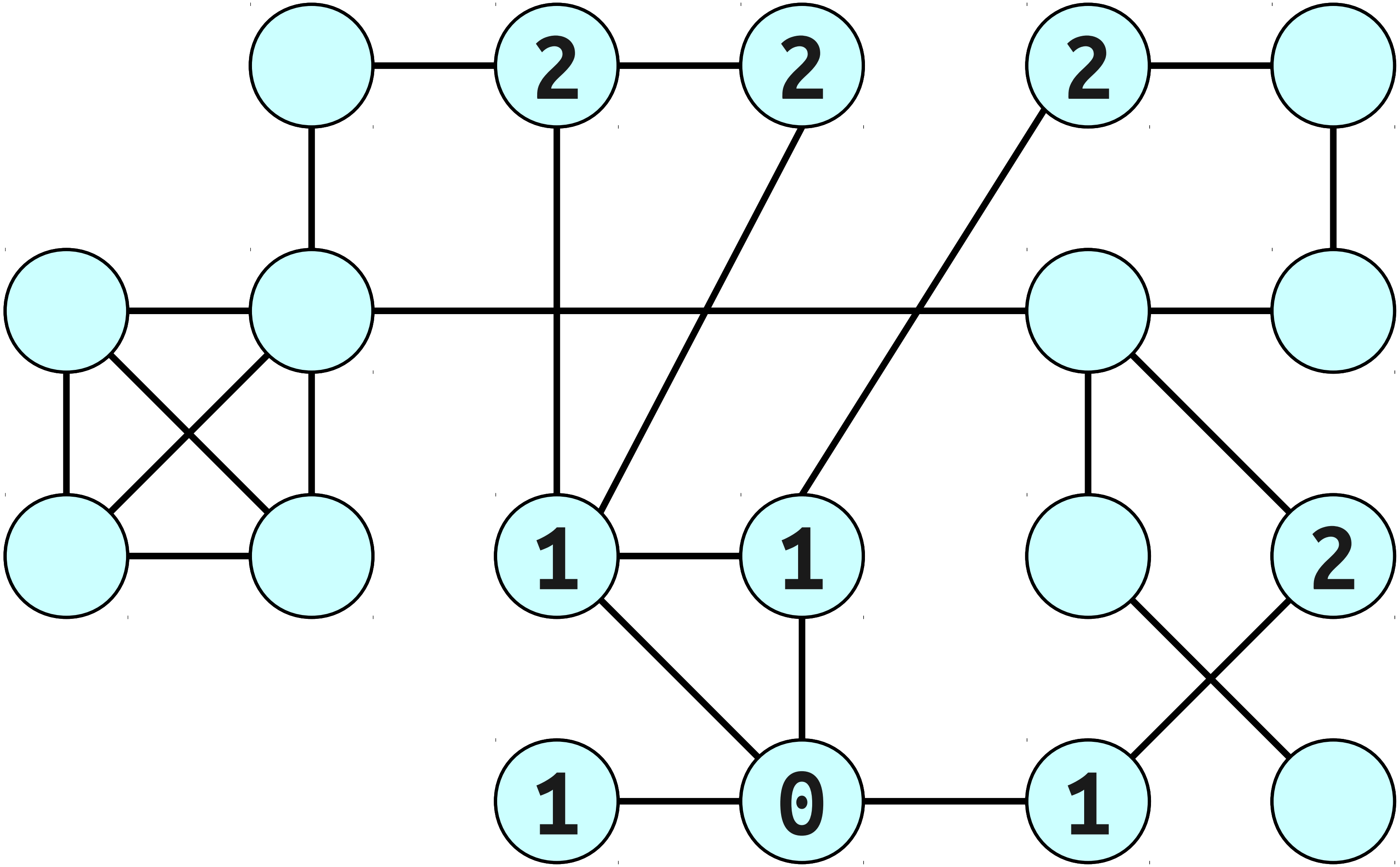
An Inefficient Algorithm



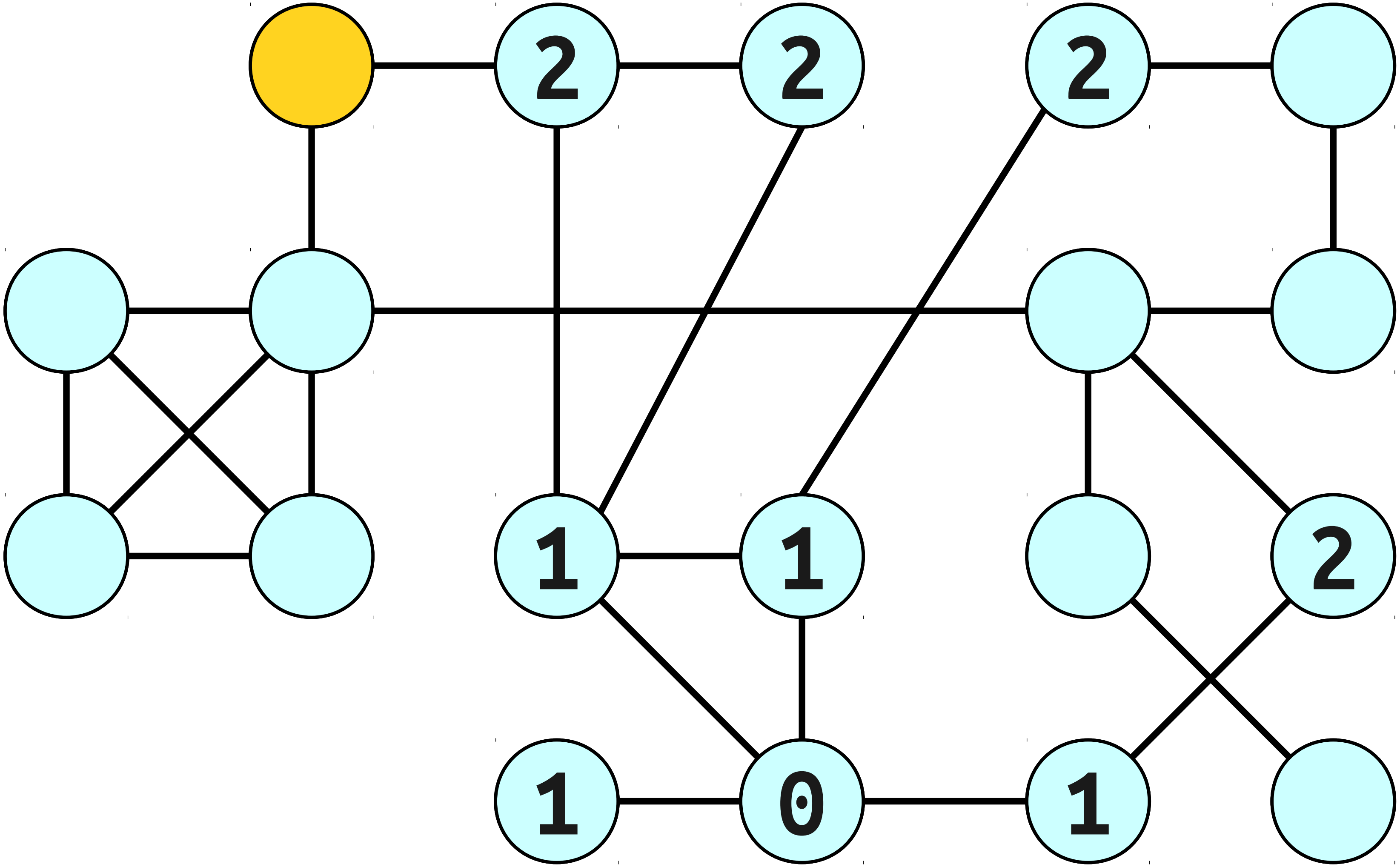
An Inefficient Algorithm



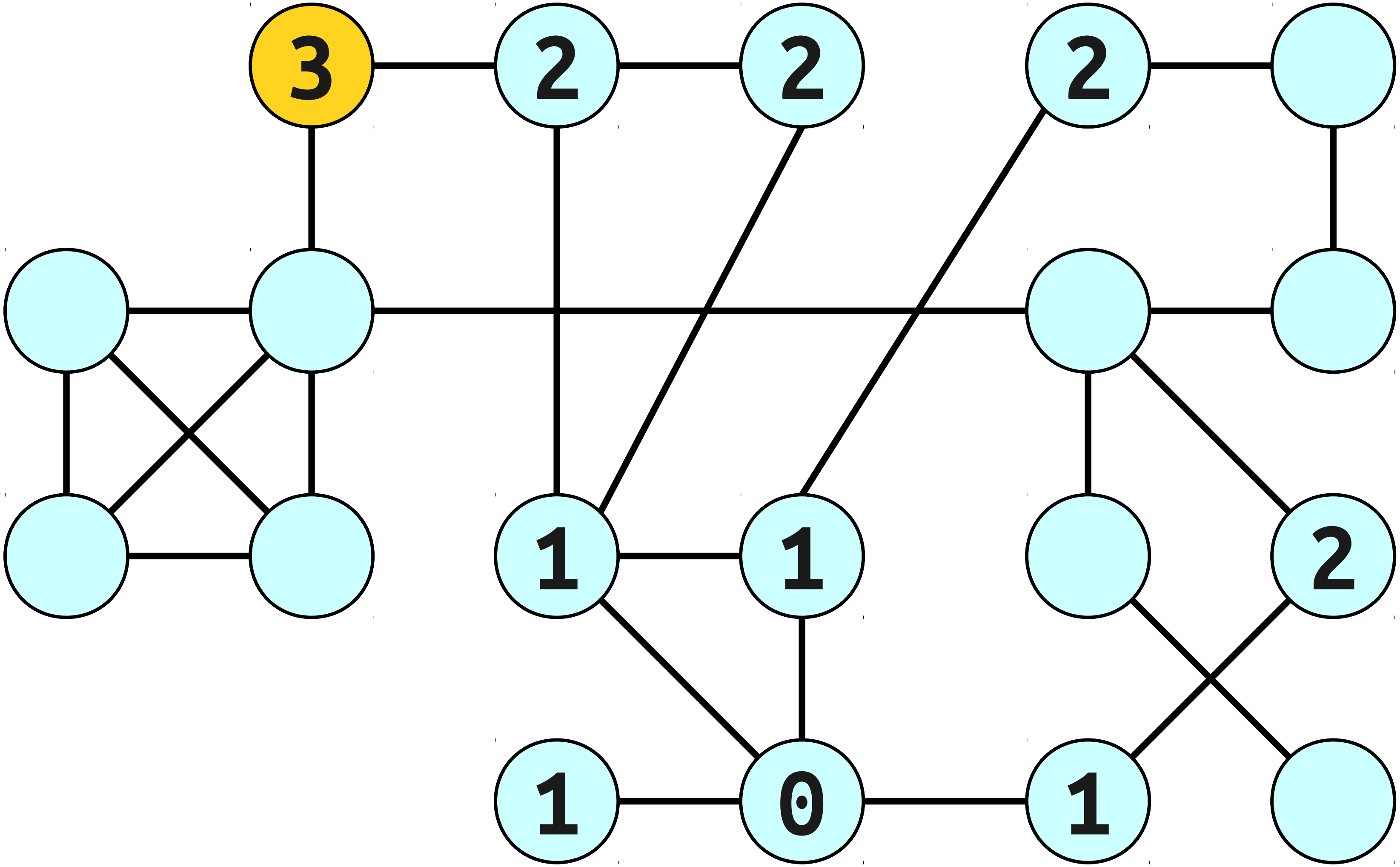
An Inefficient Algorithm



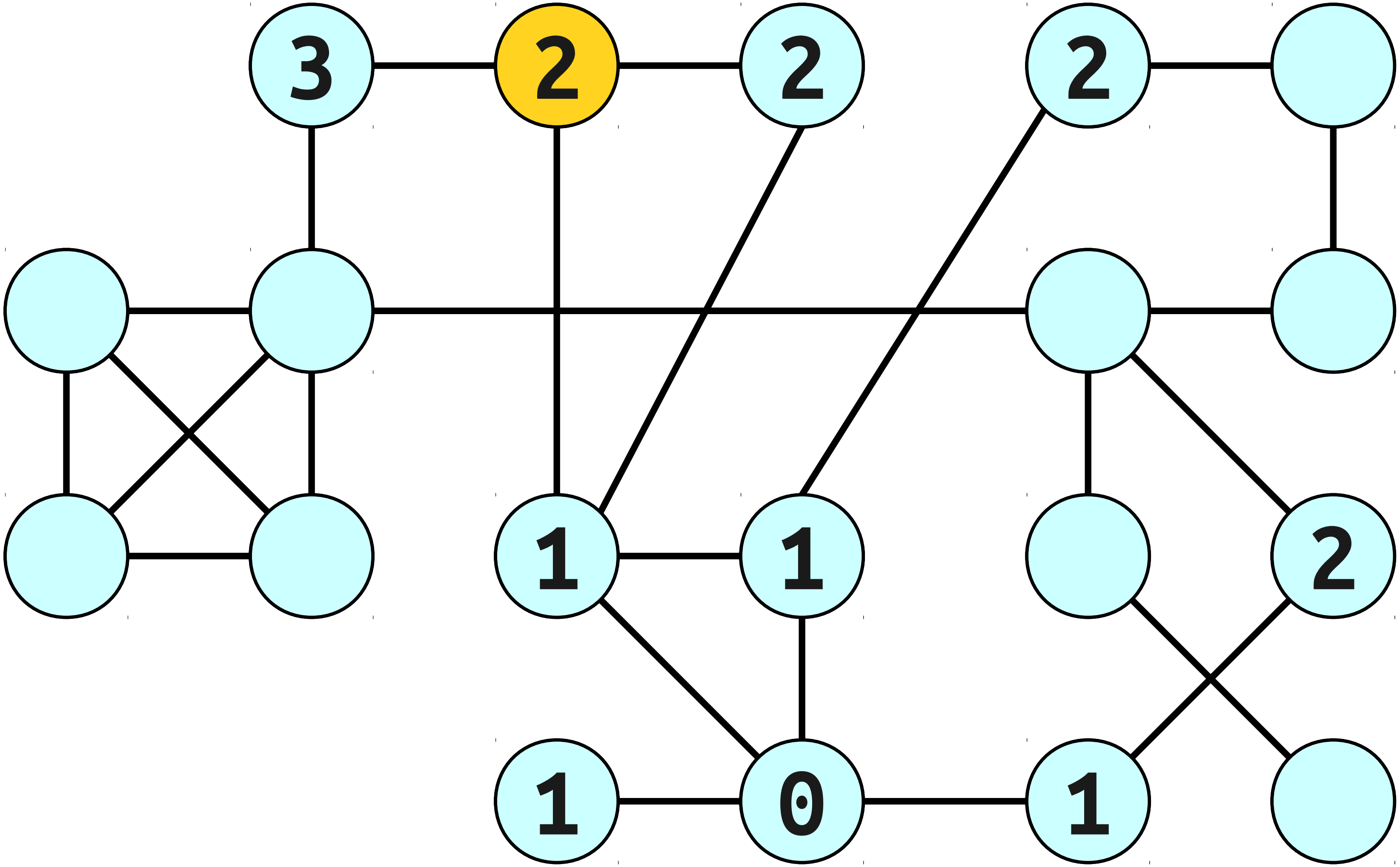
An Inefficient Algorithm



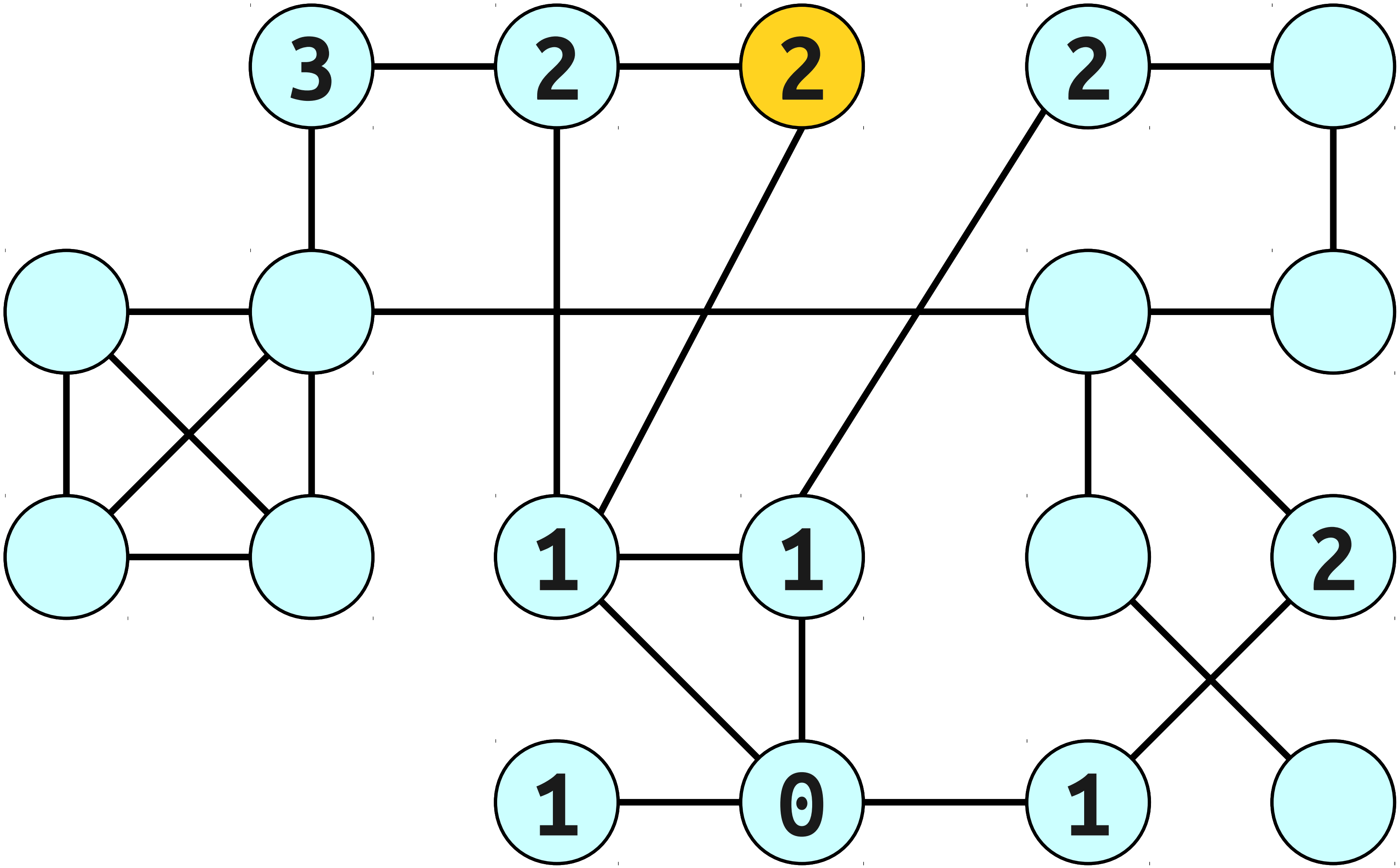
An Inefficient Algorithm



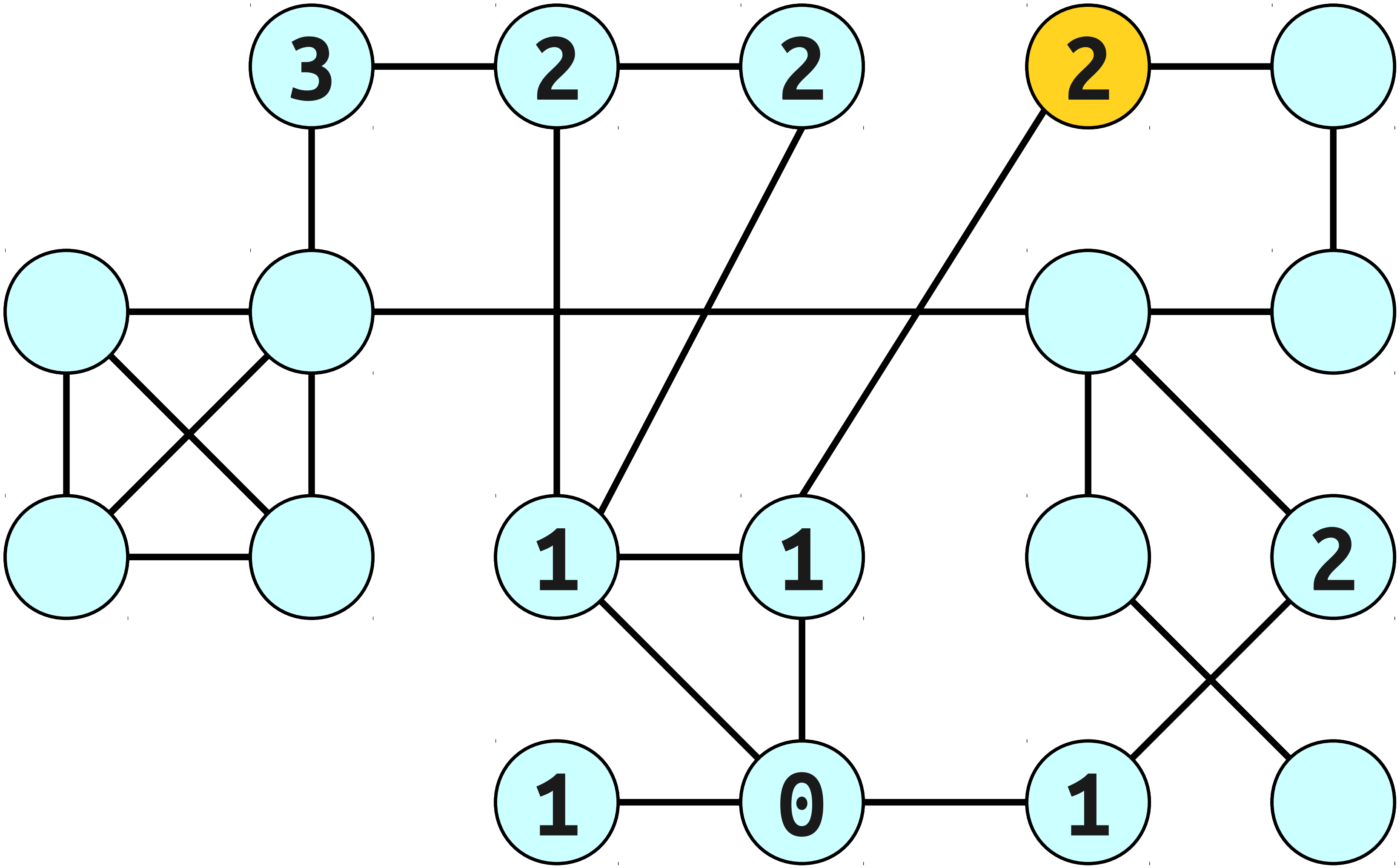
An Inefficient Algorithm



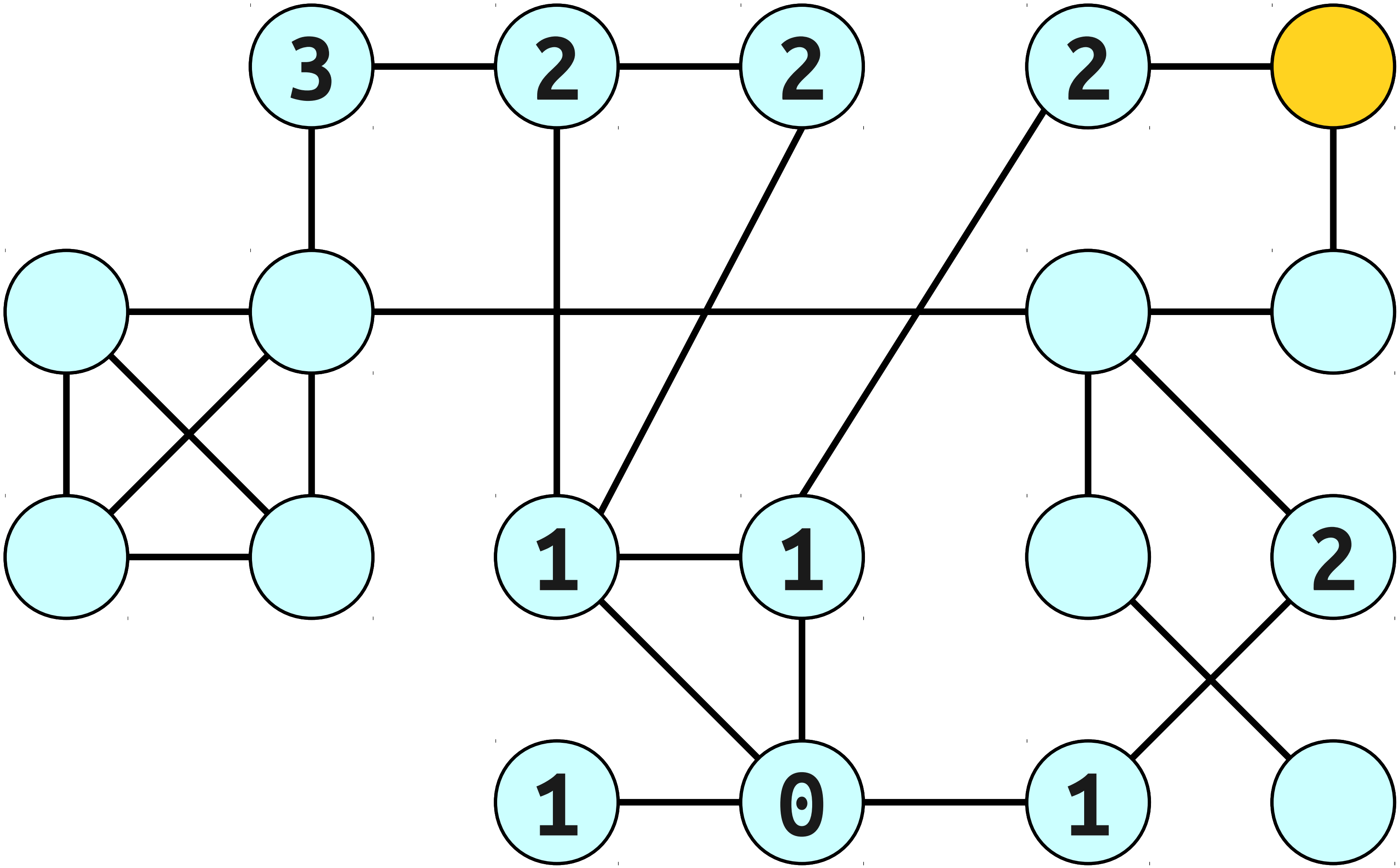
An Inefficient Algorithm



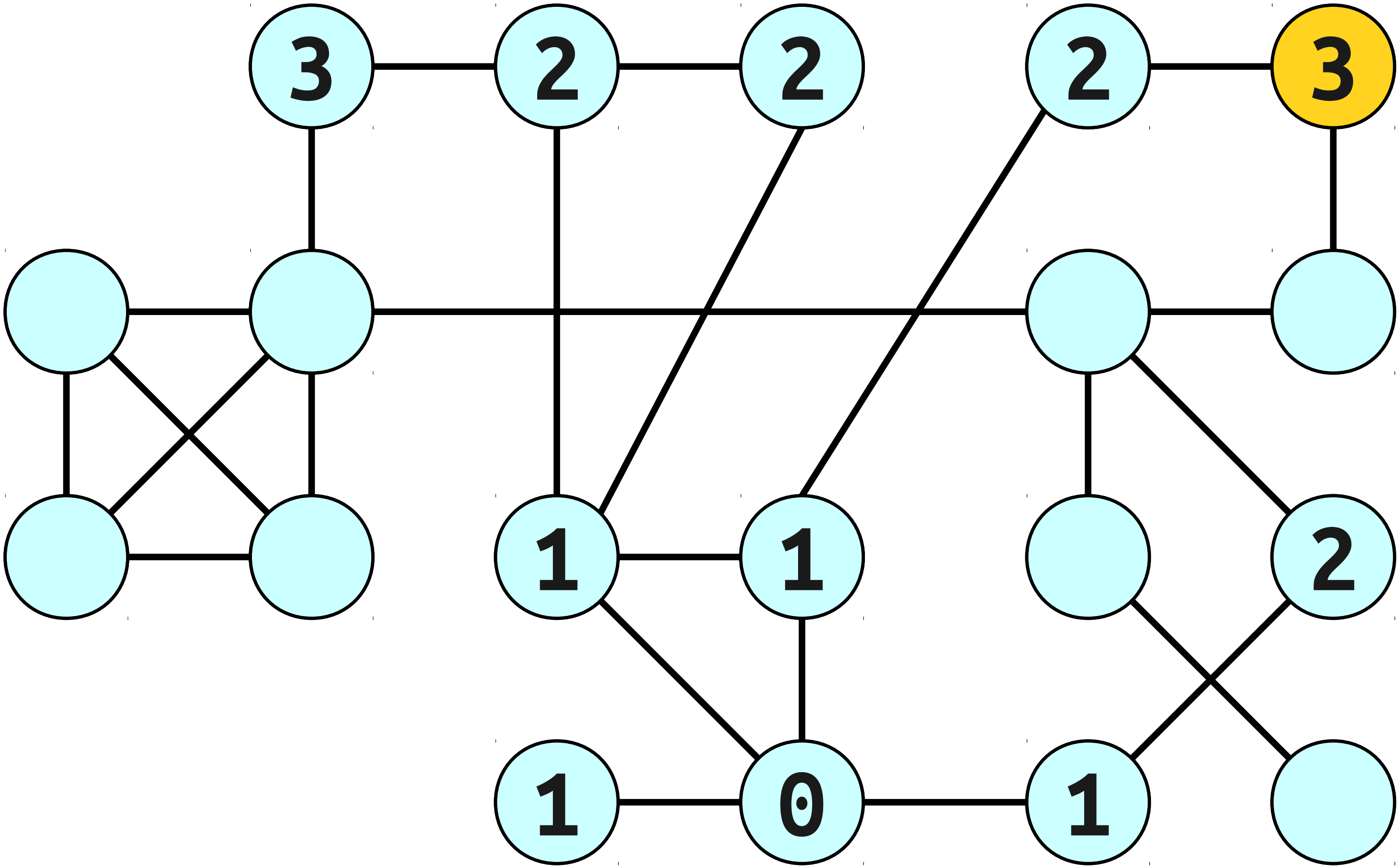
An Inefficient Algorithm



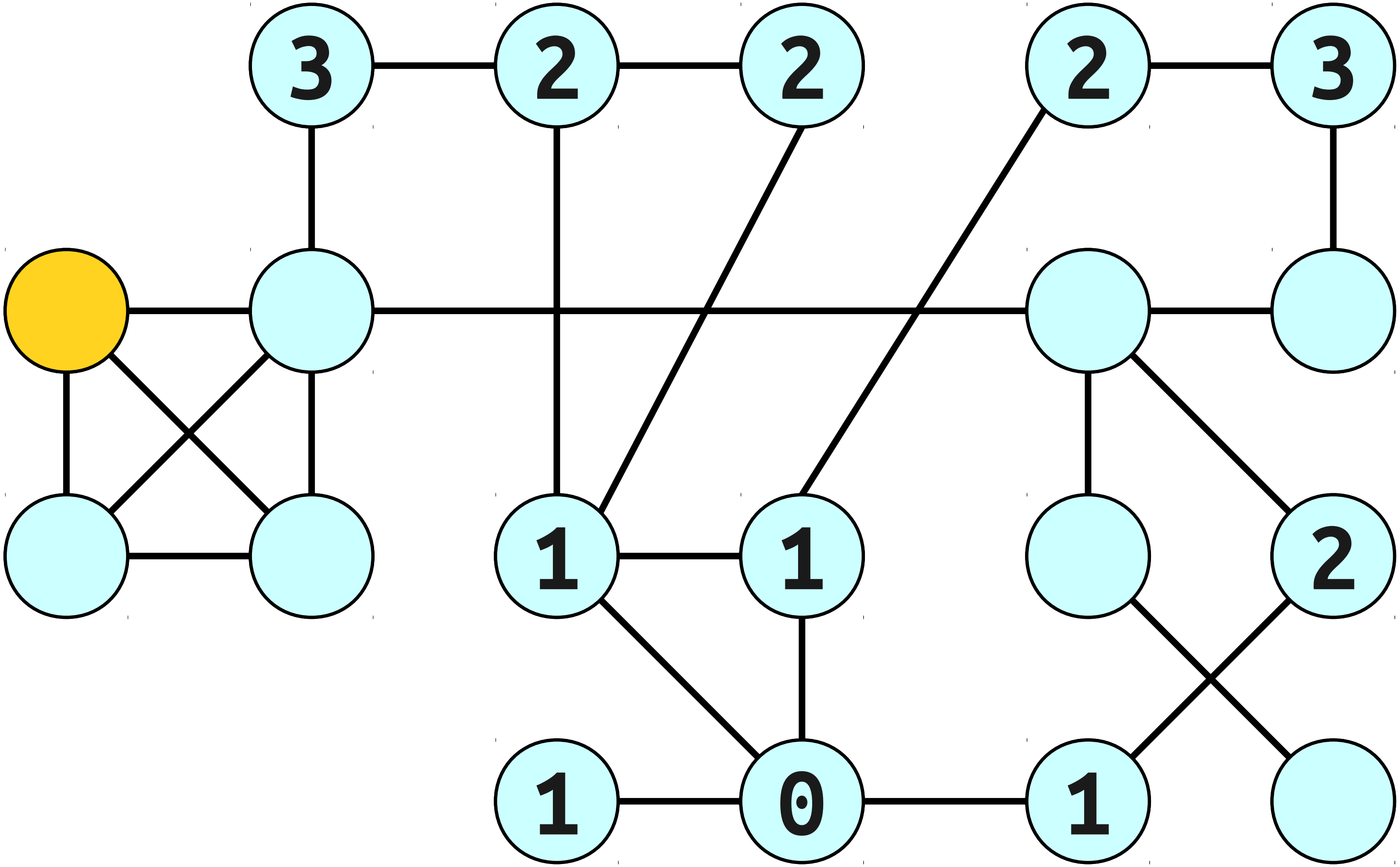
An Inefficient Algorithm



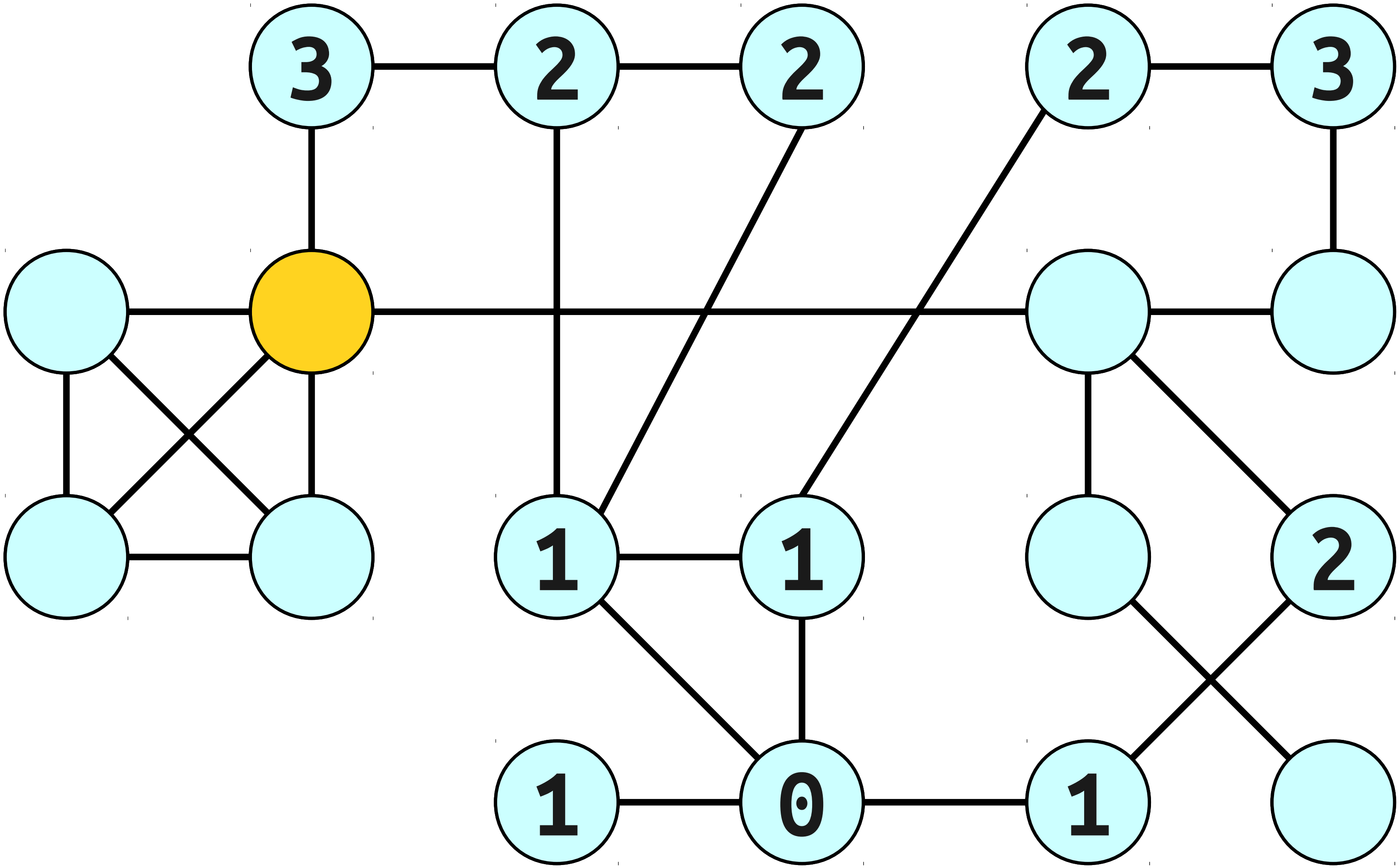
An Inefficient Algorithm



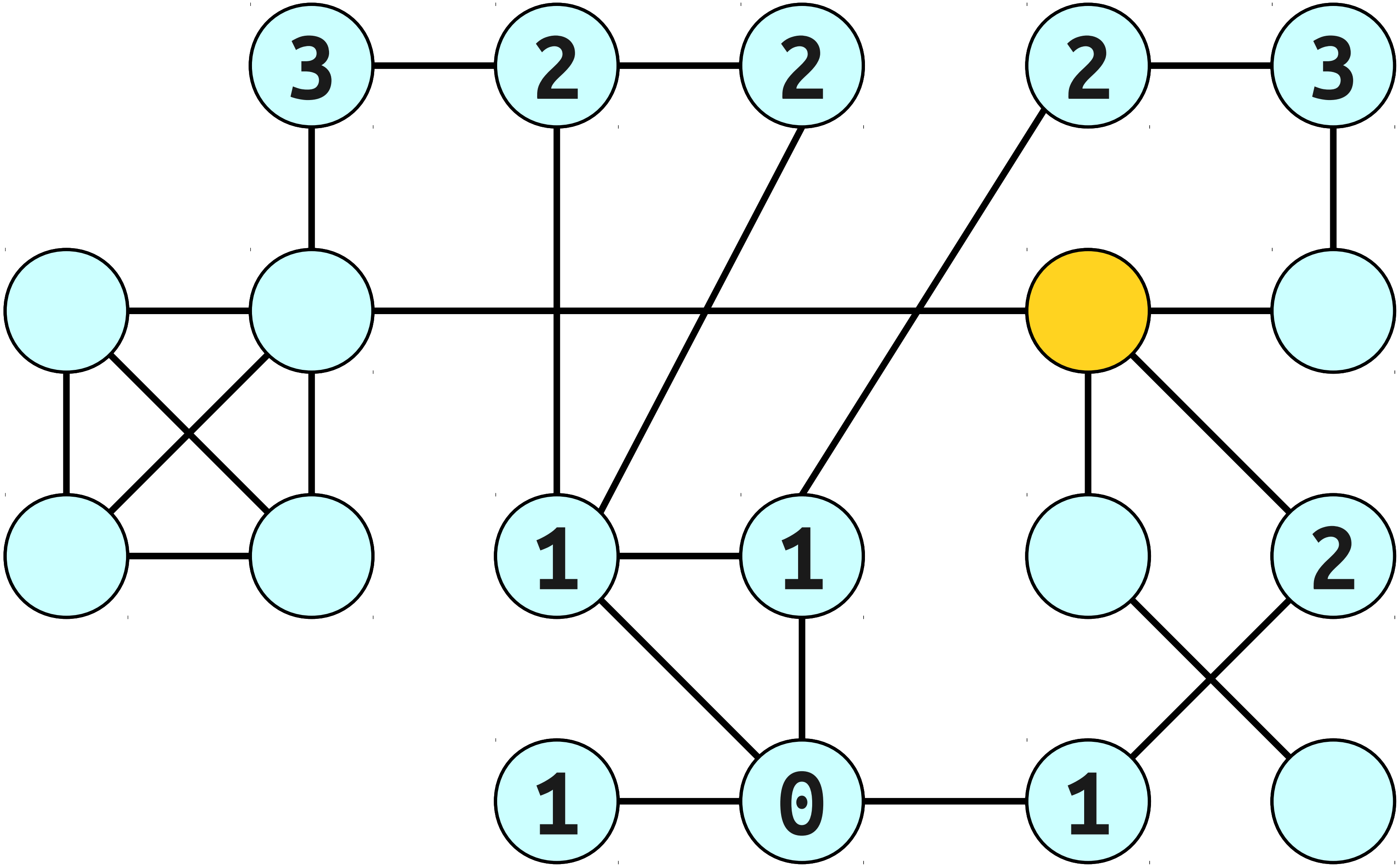
An Inefficient Algorithm



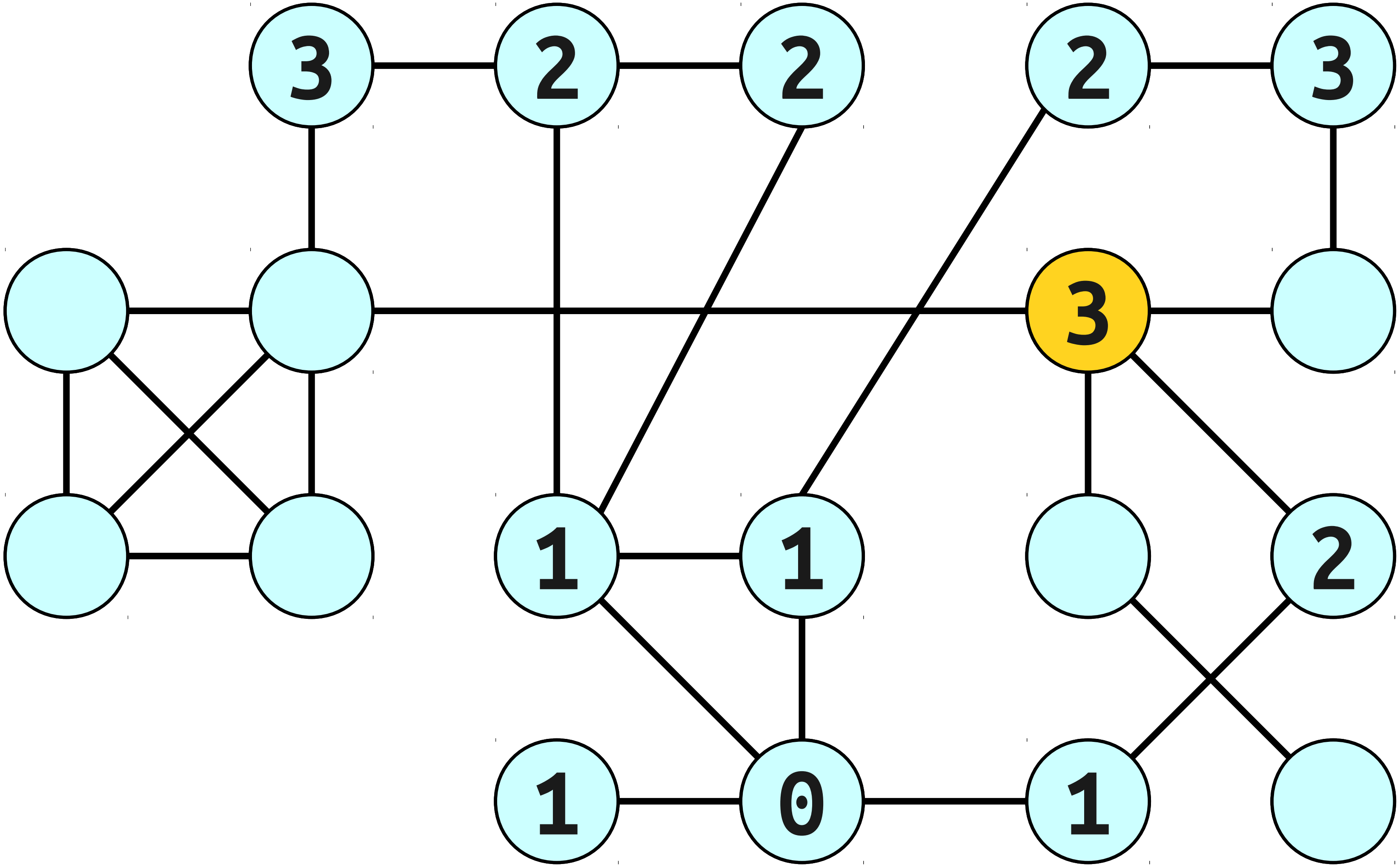
An Inefficient Algorithm



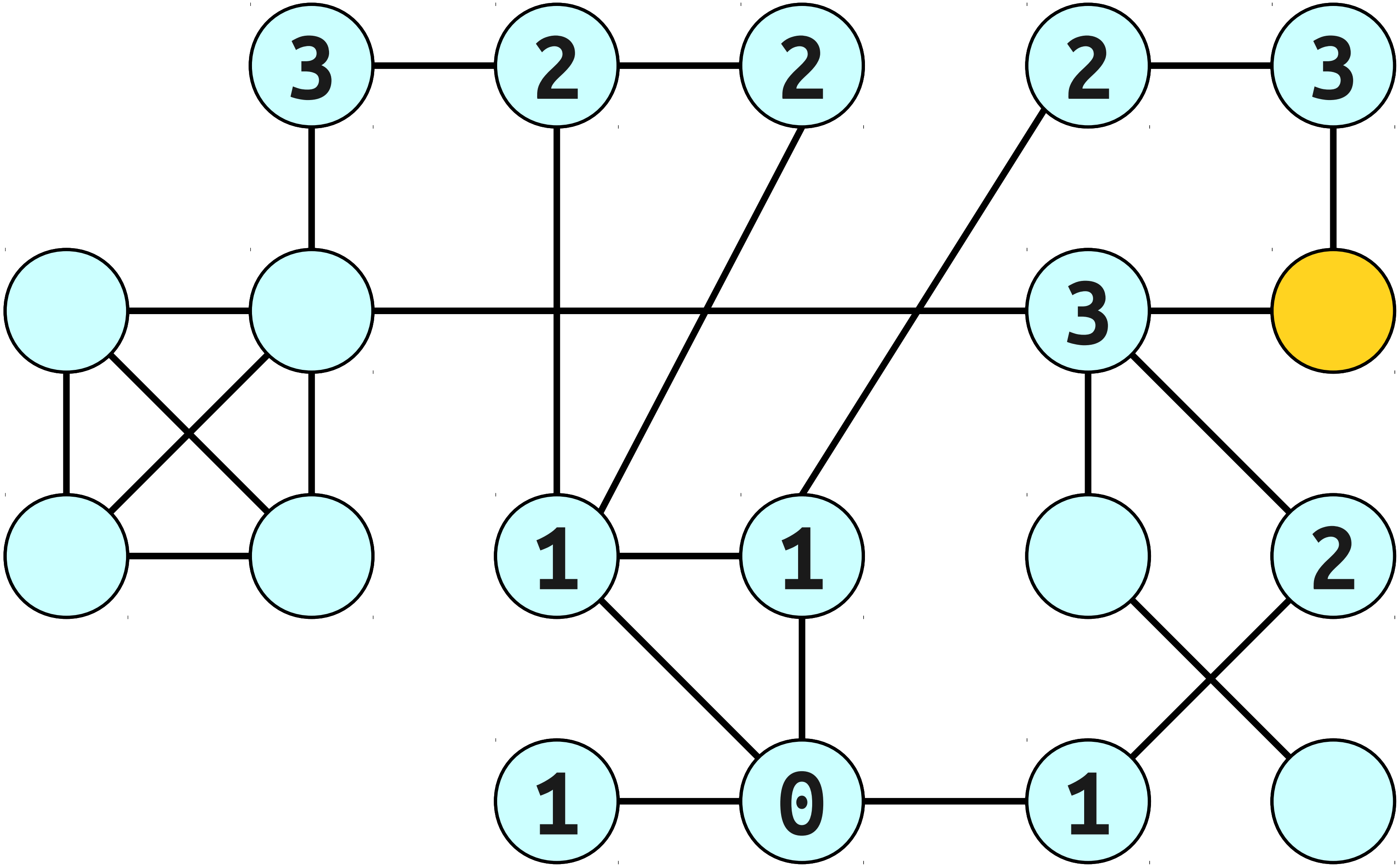
An Inefficient Algorithm



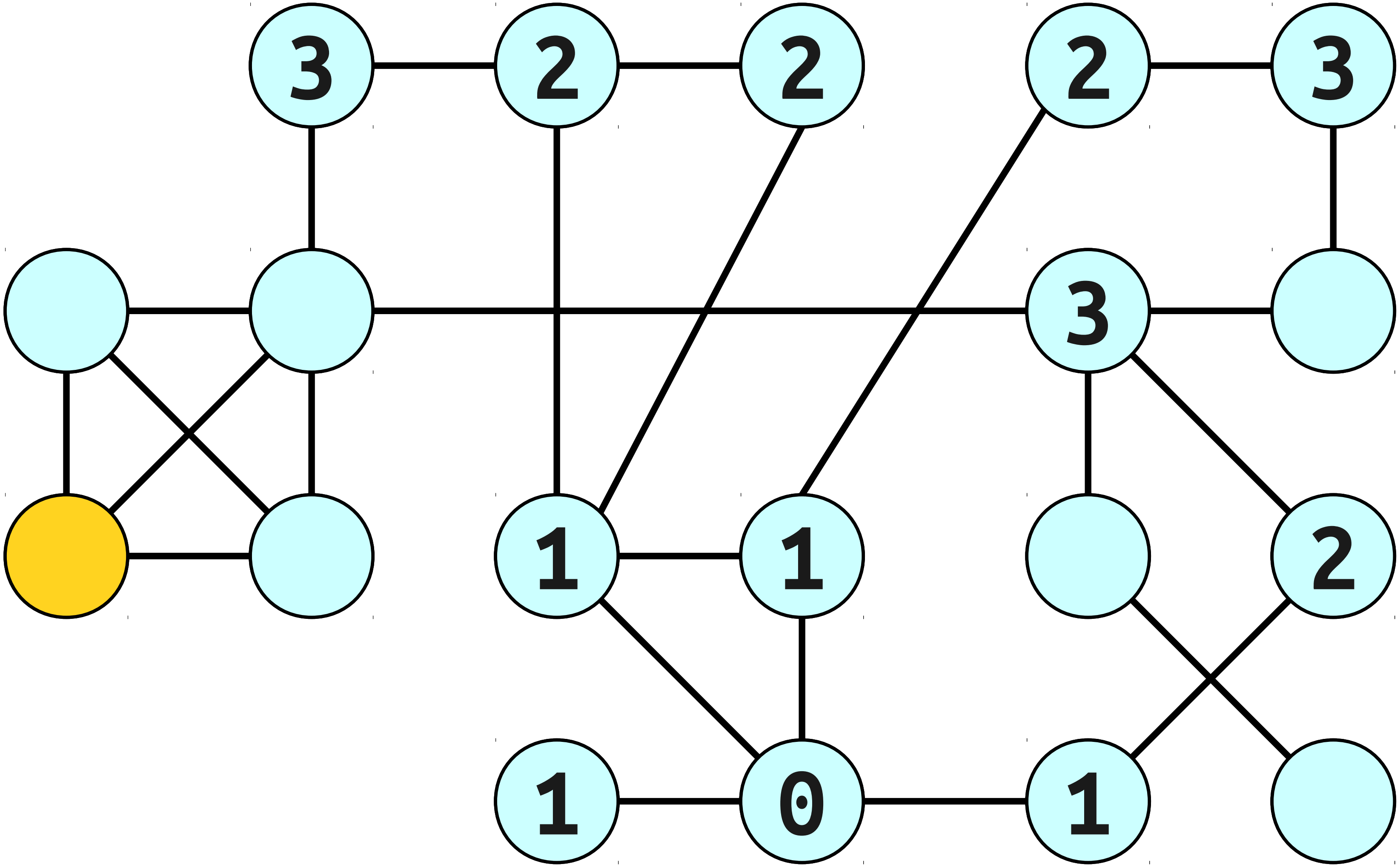
An Inefficient Algorithm



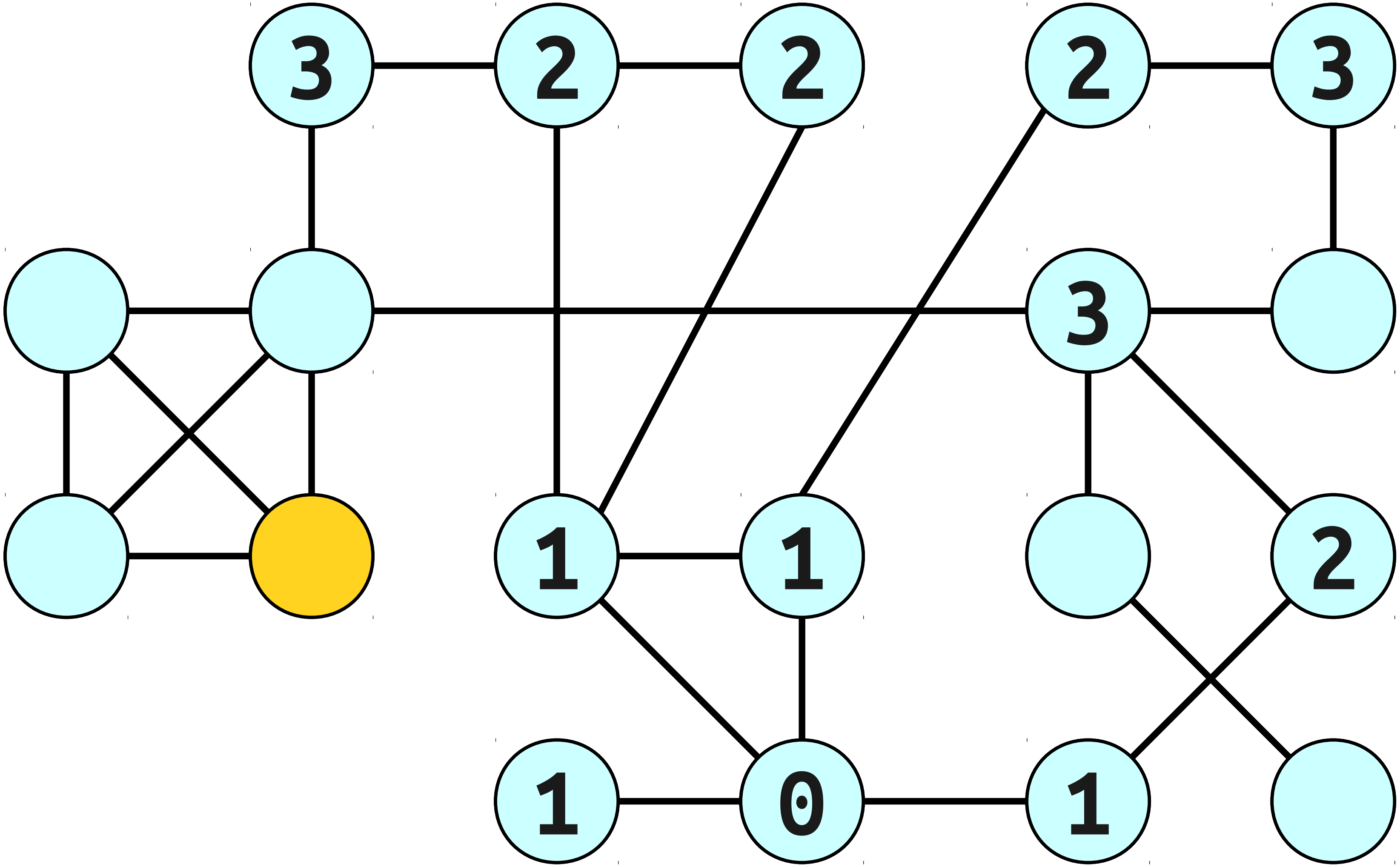
An Inefficient Algorithm



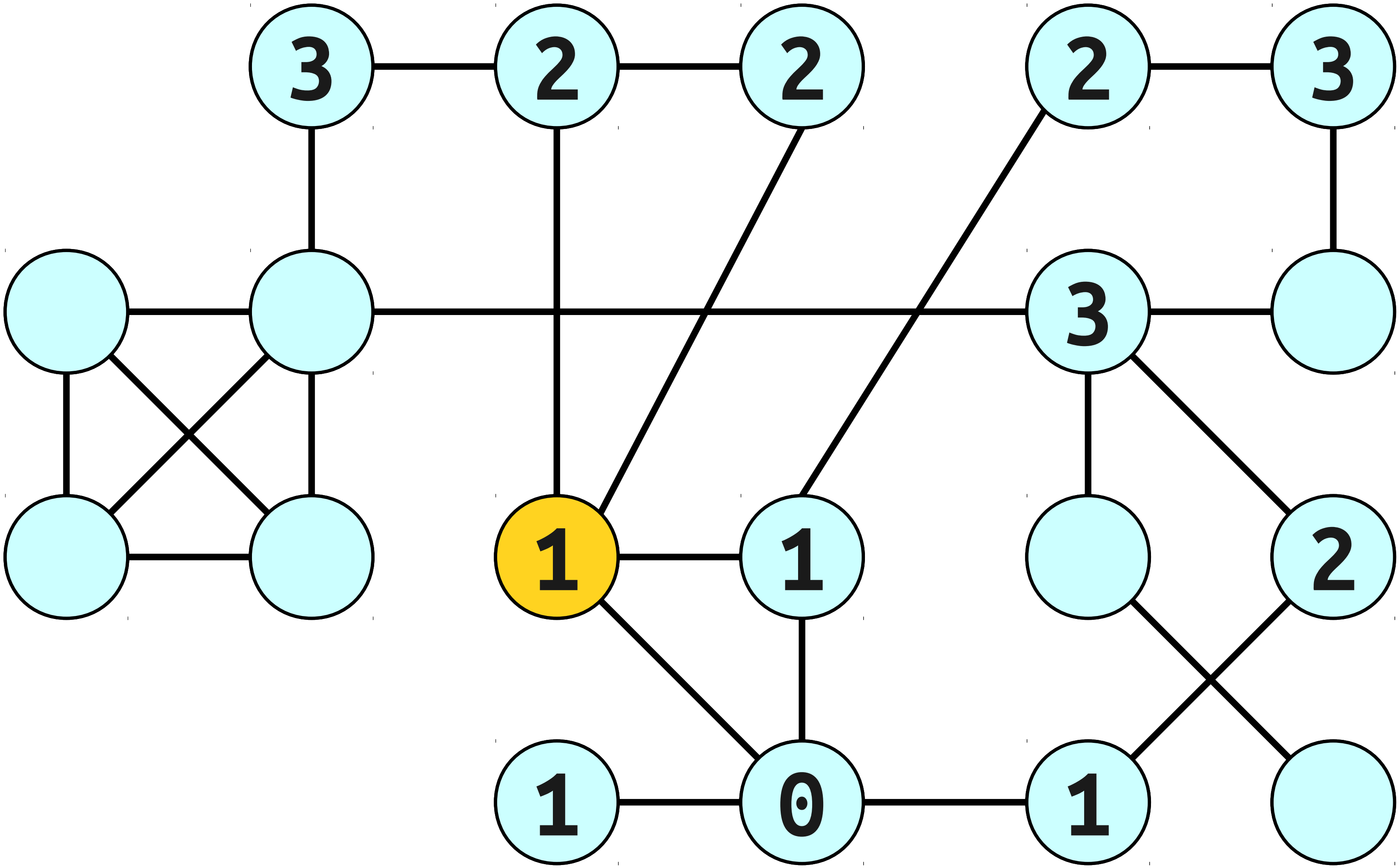
An Inefficient Algorithm



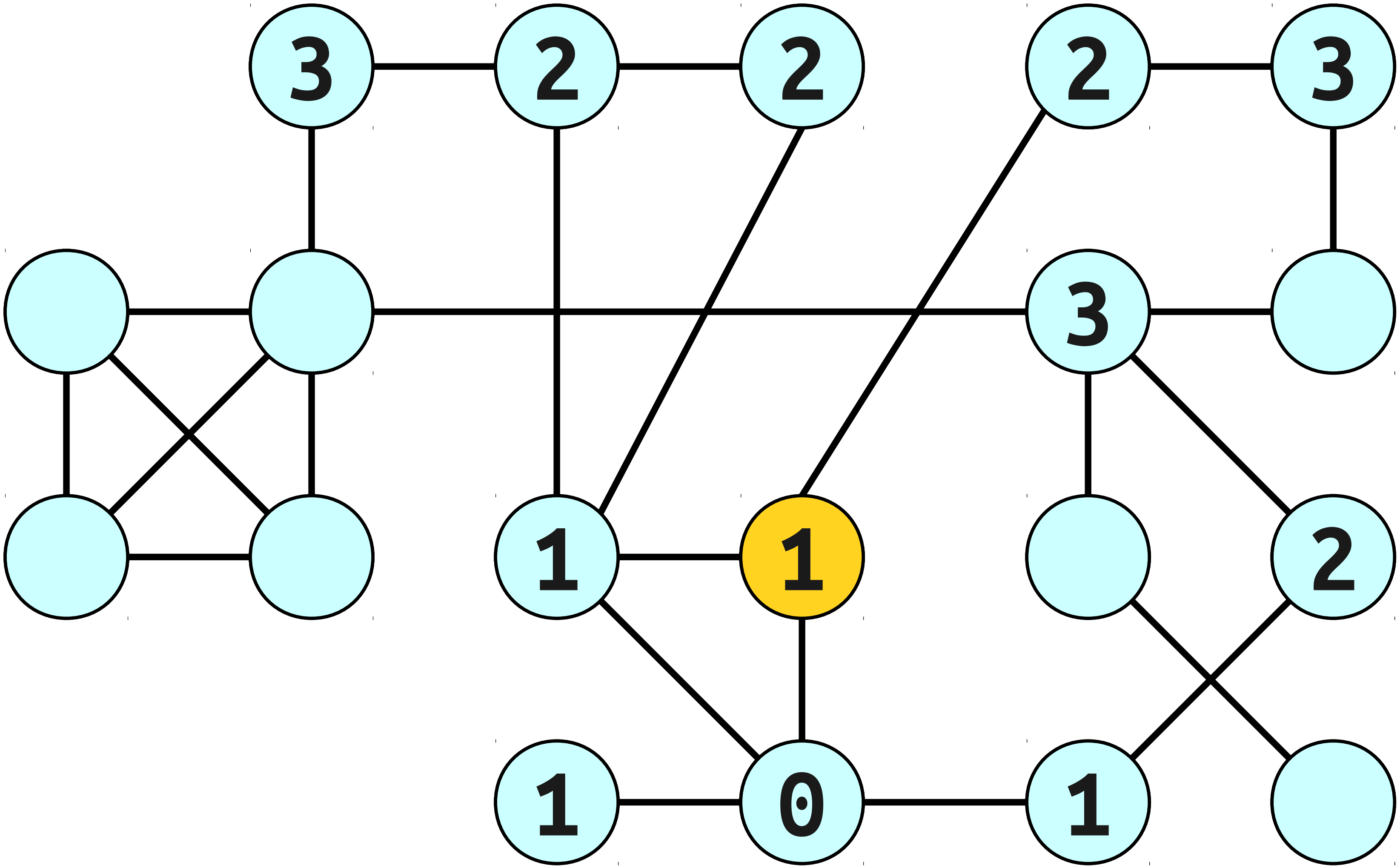
An Inefficient Algorithm



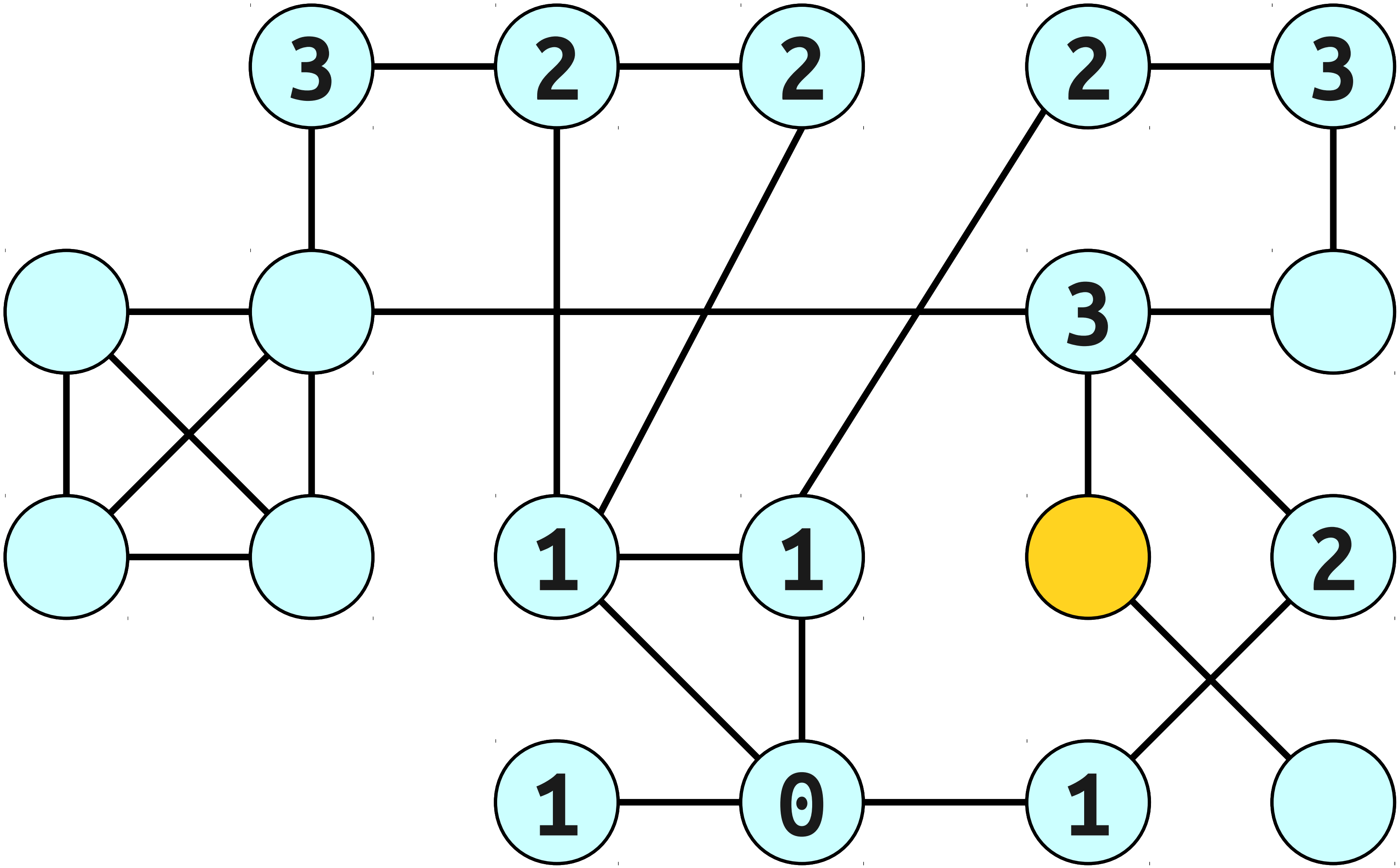
An Inefficient Algorithm



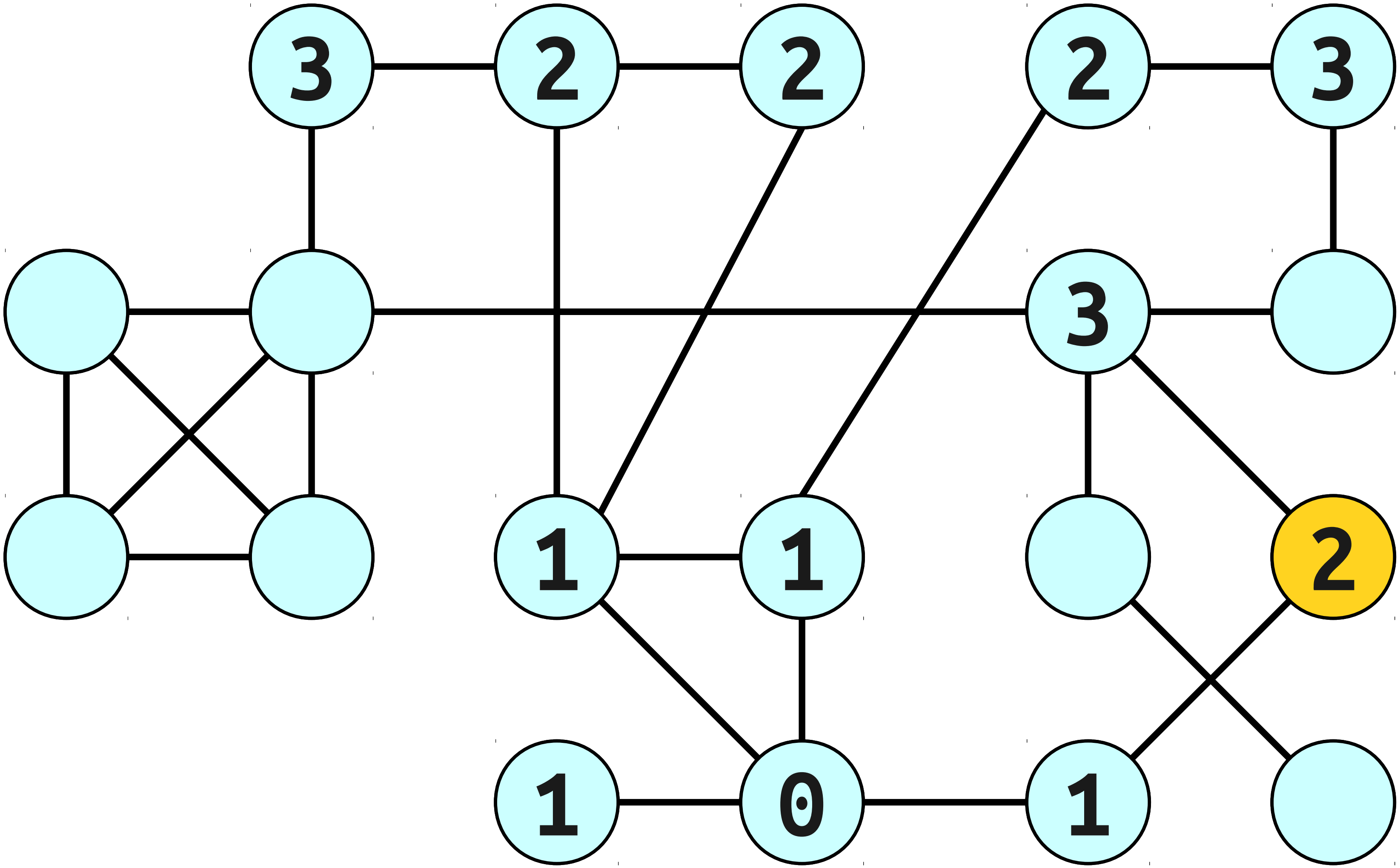
An Inefficient Algorithm



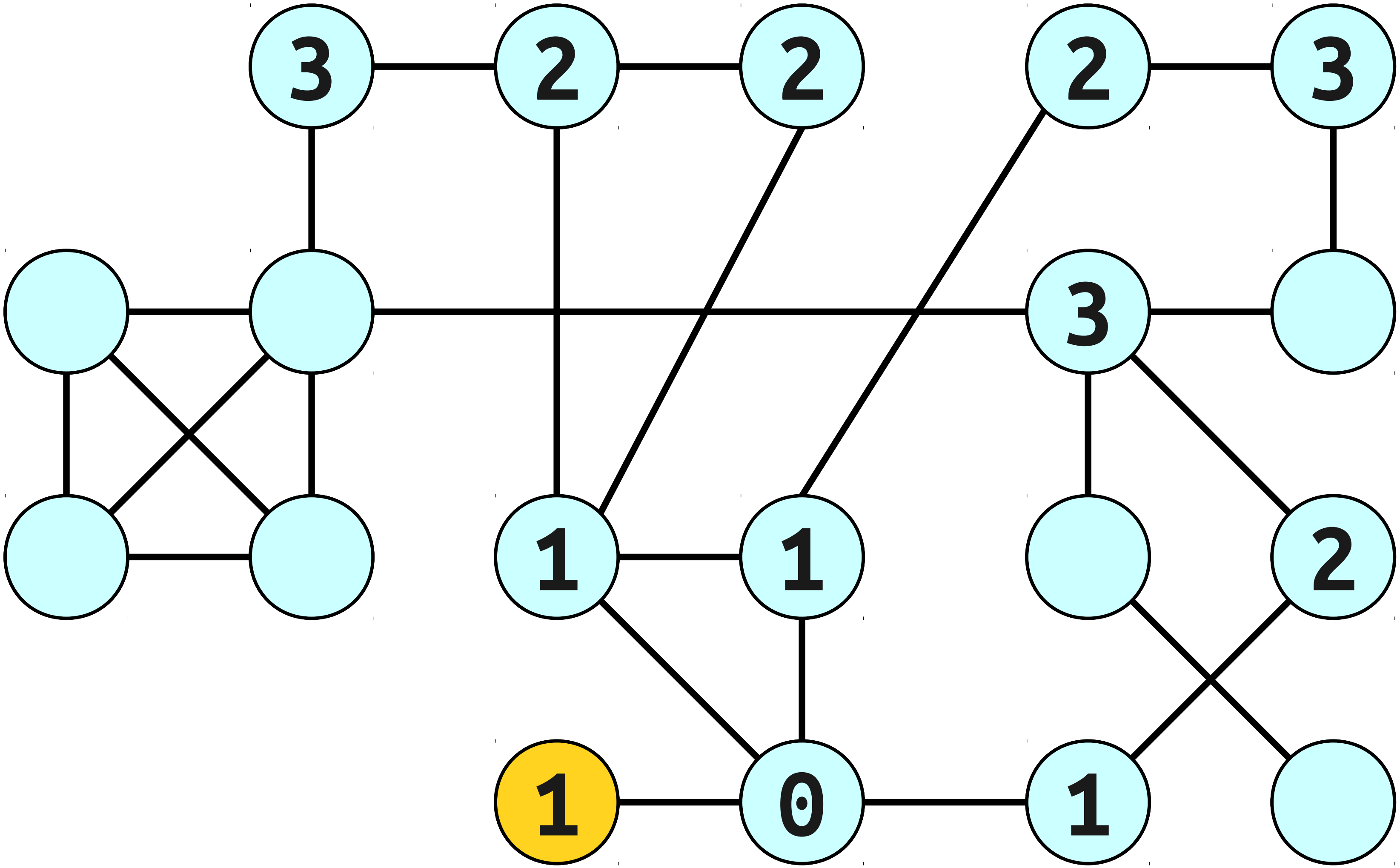
An Inefficient Algorithm



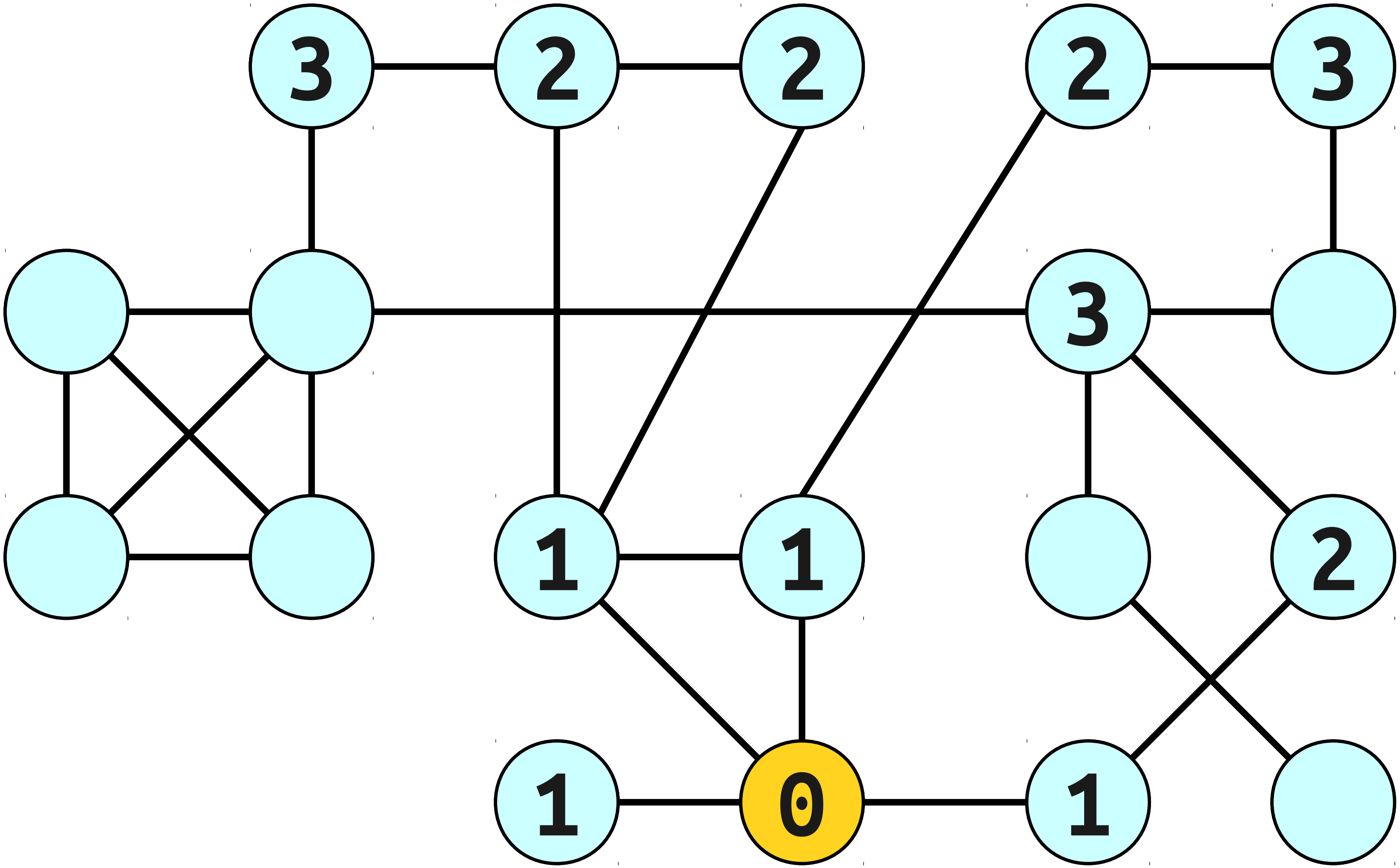
An Inefficient Algorithm



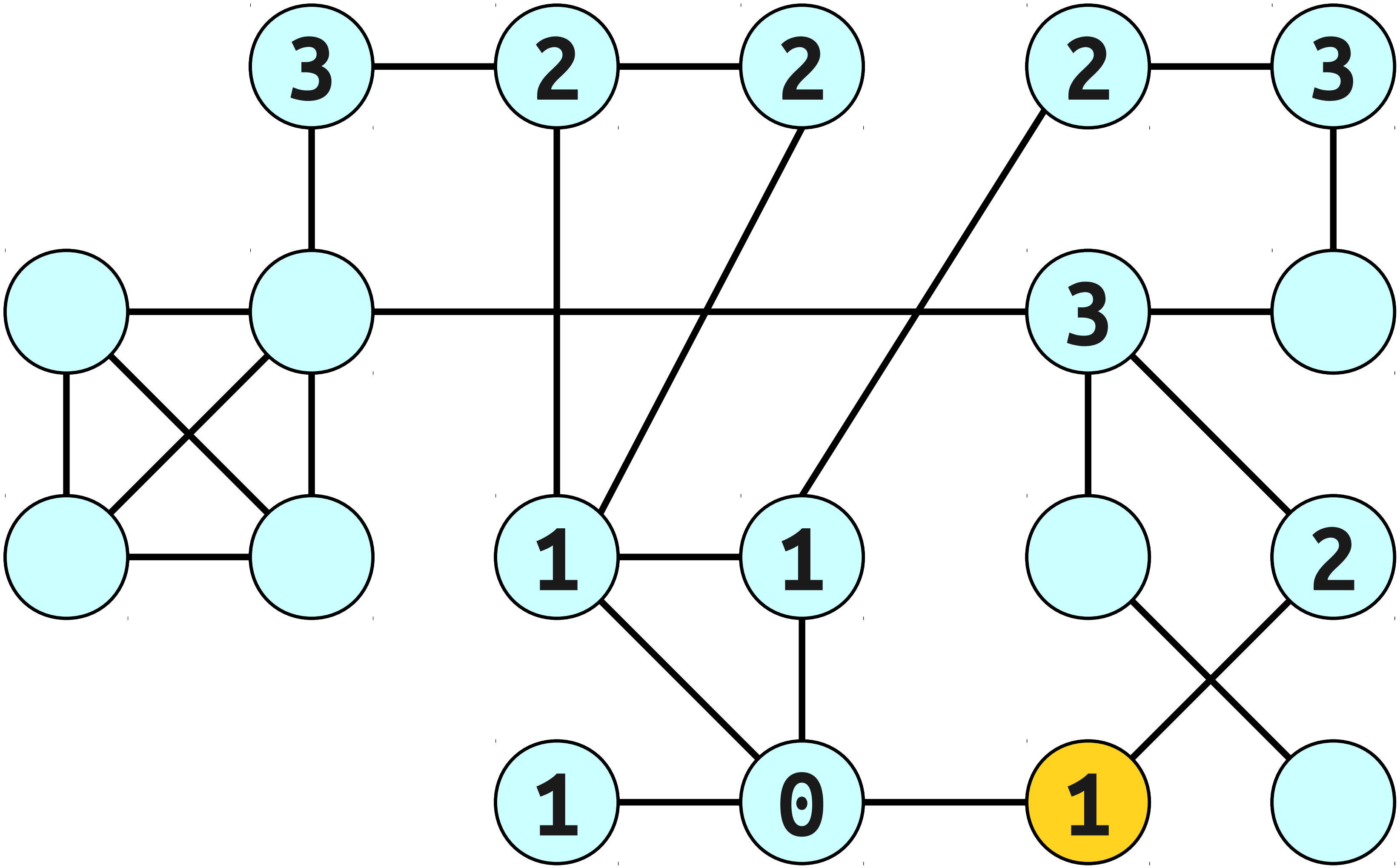
An Inefficient Algorithm



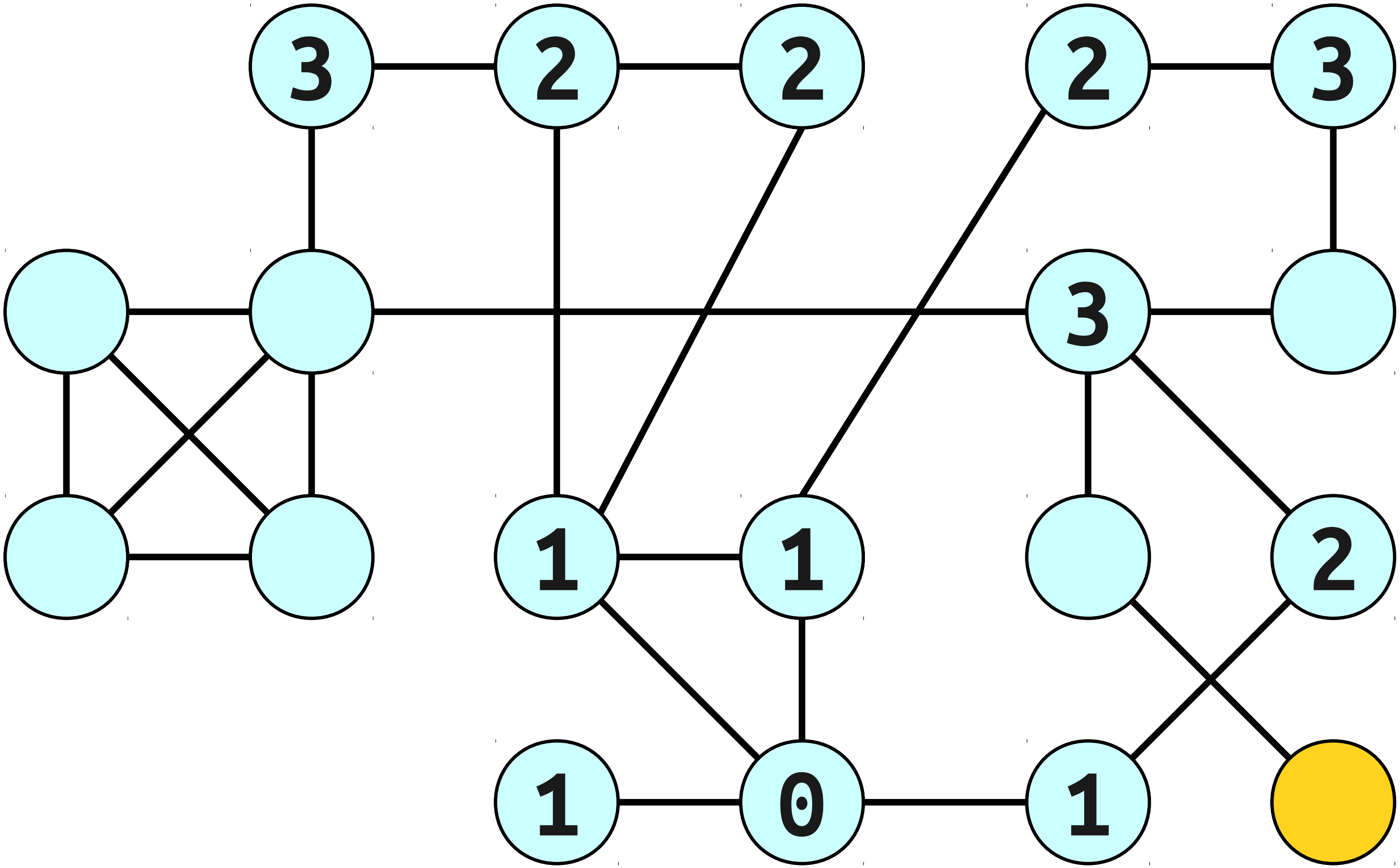
An Inefficient Algorithm



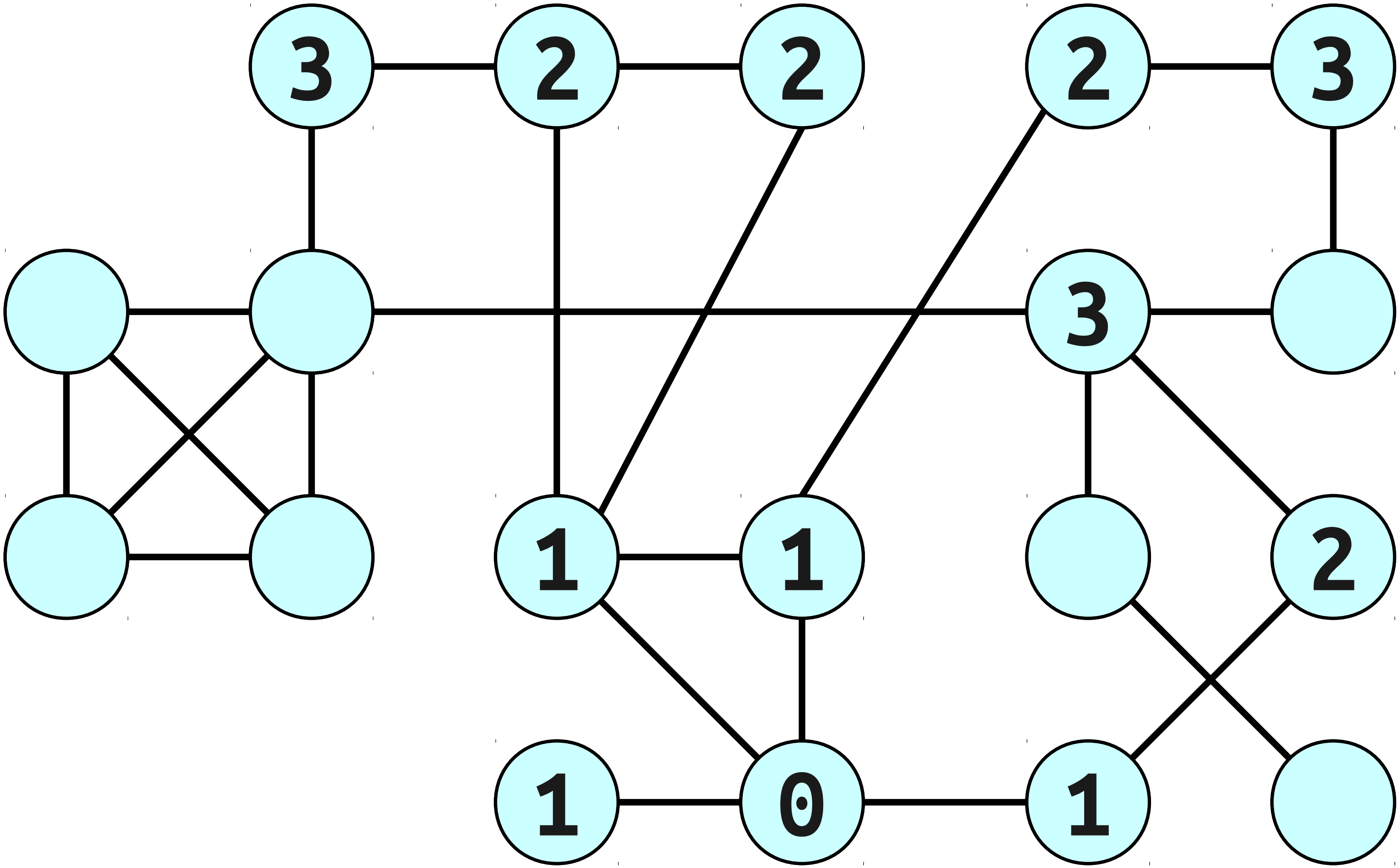
An Inefficient Algorithm



An Inefficient Algorithm

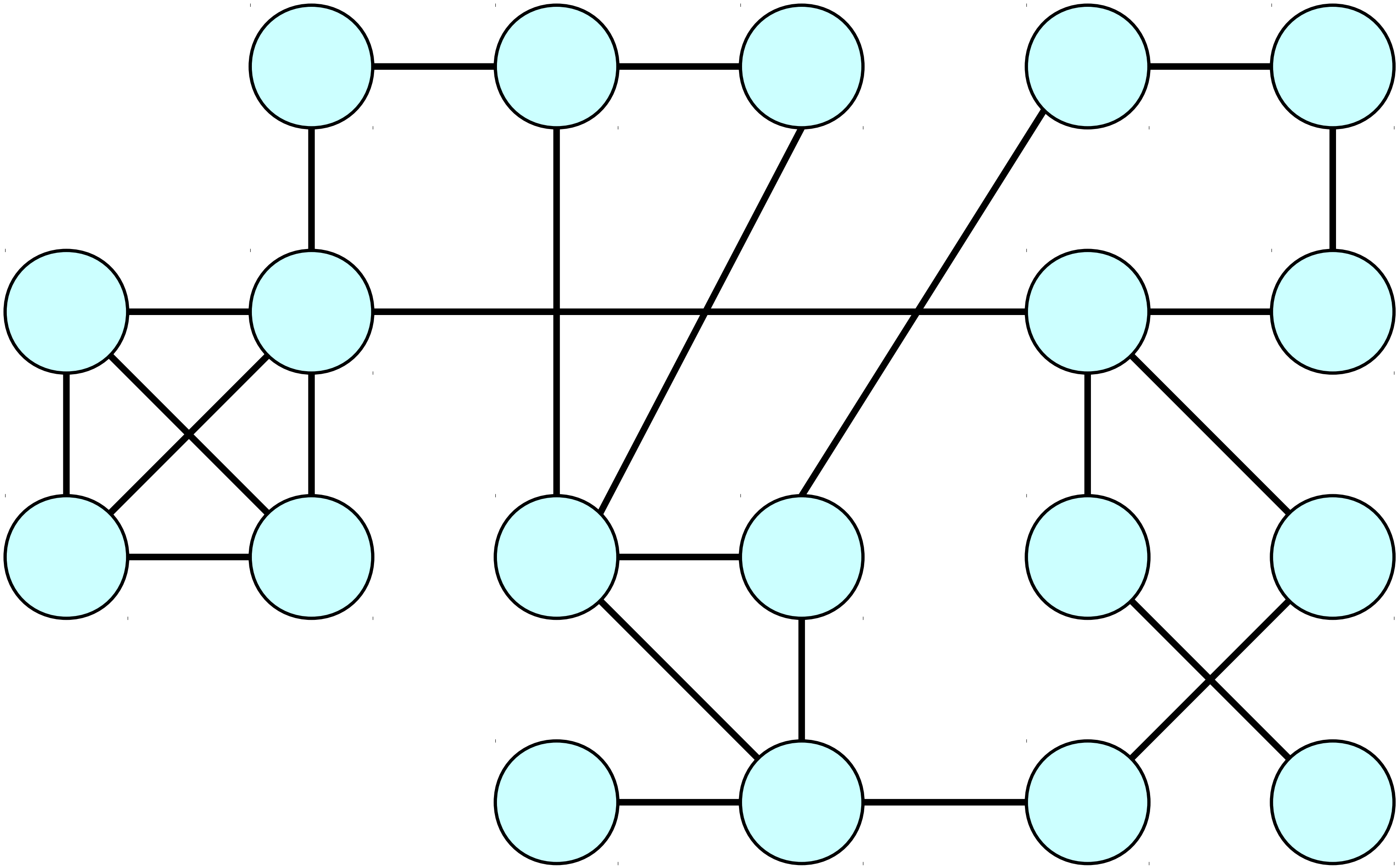


An Inefficient Algorithm

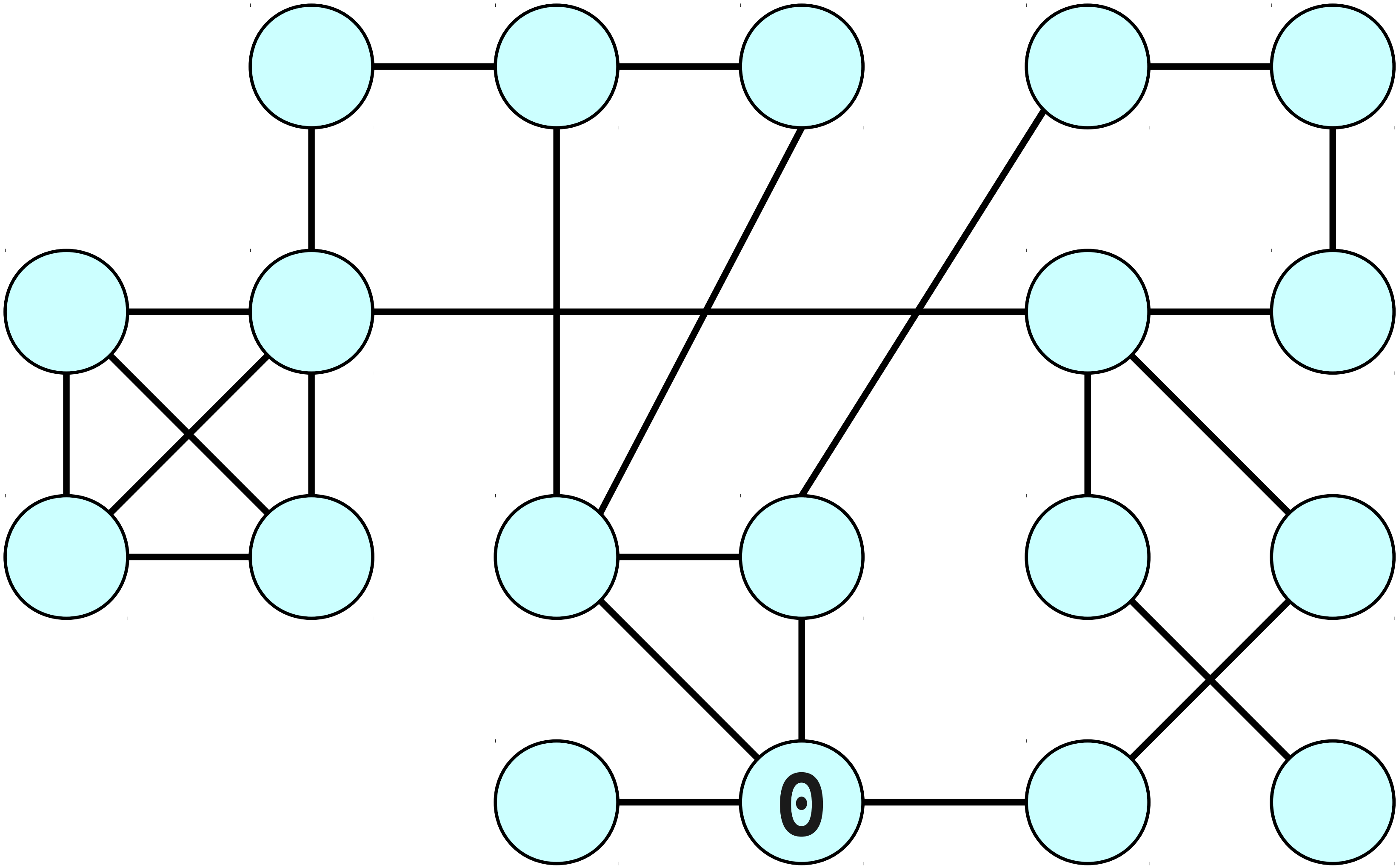


A Better Approach

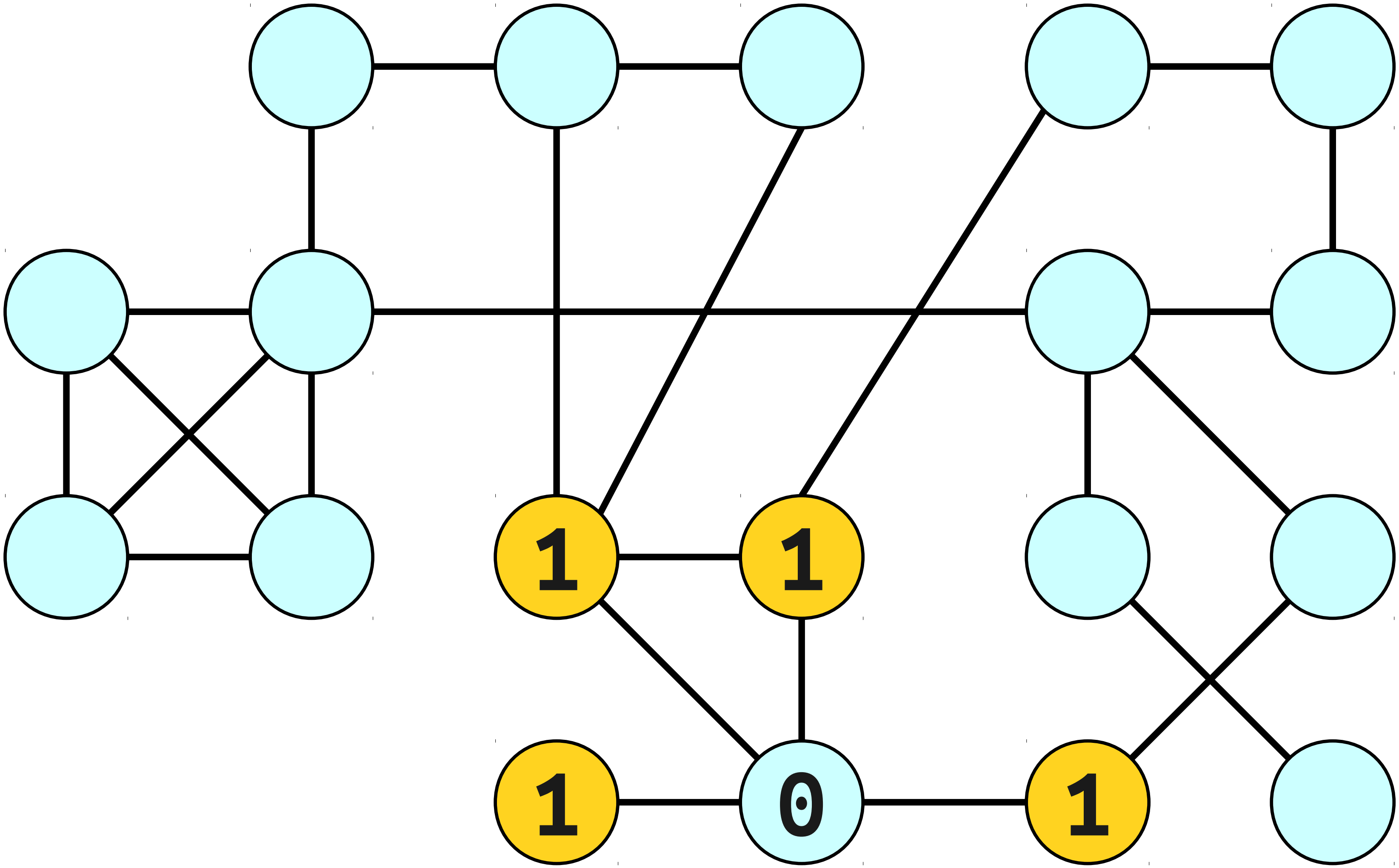
Radiating Outward



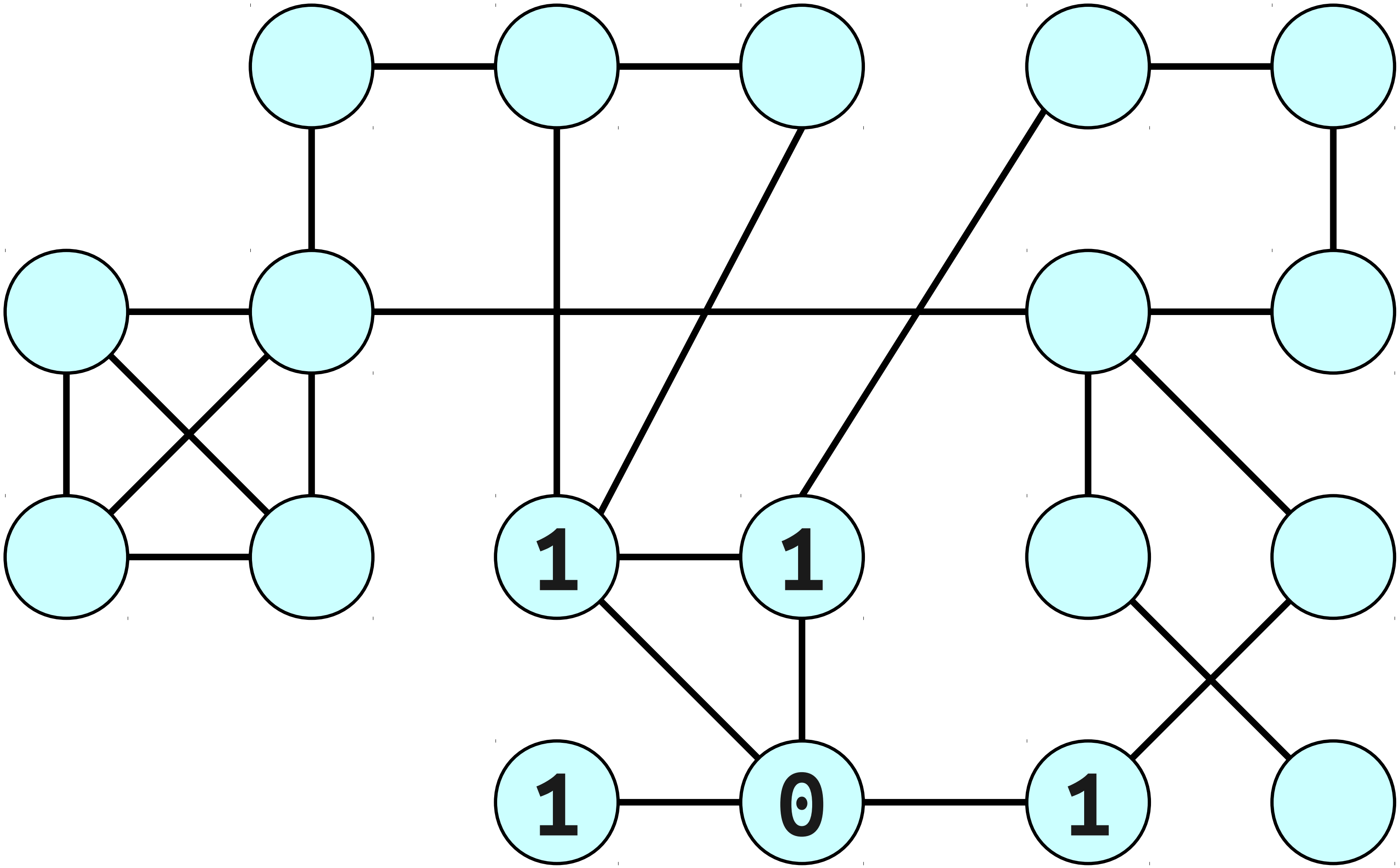
Radiating Outward



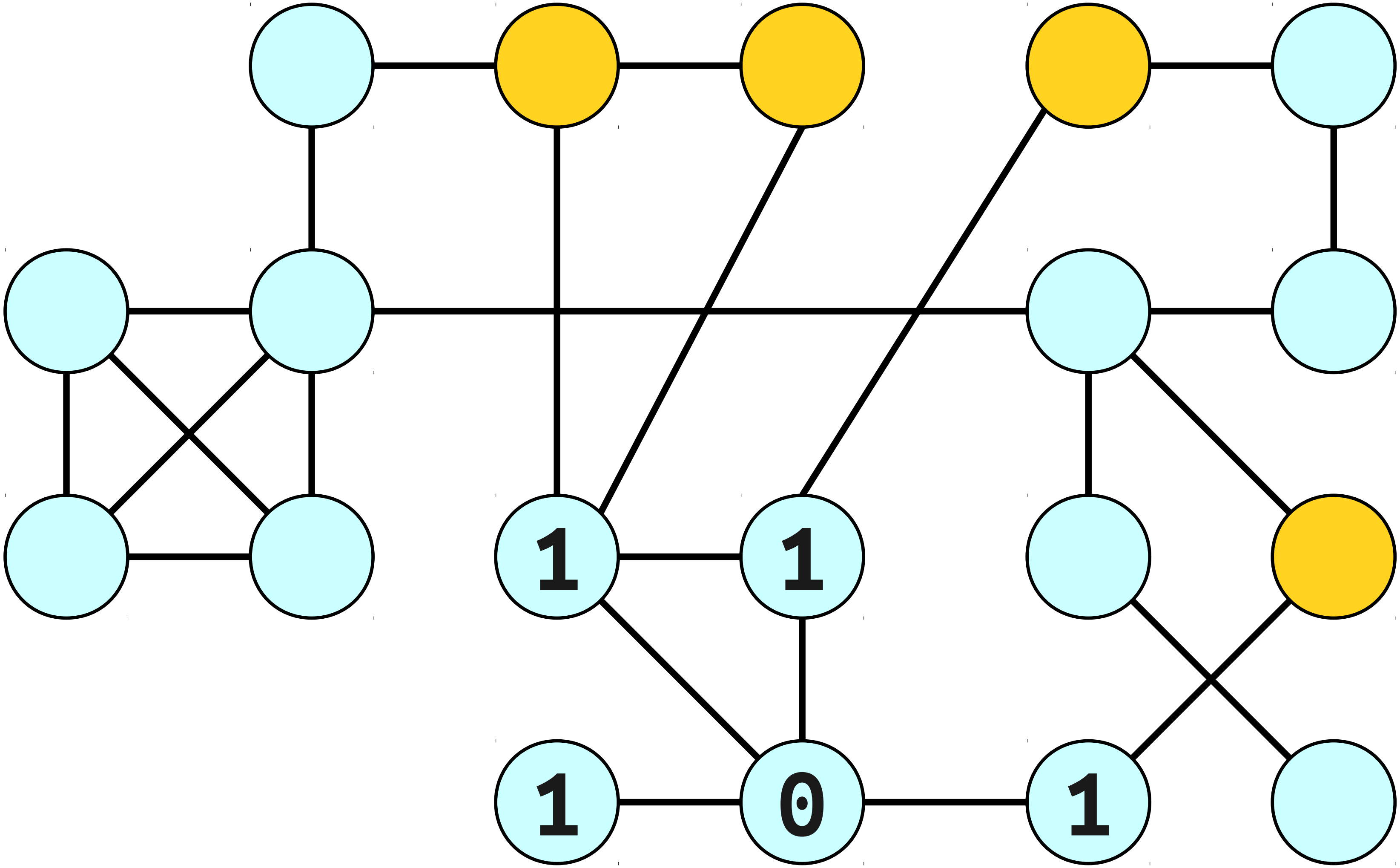
Radiating Outward



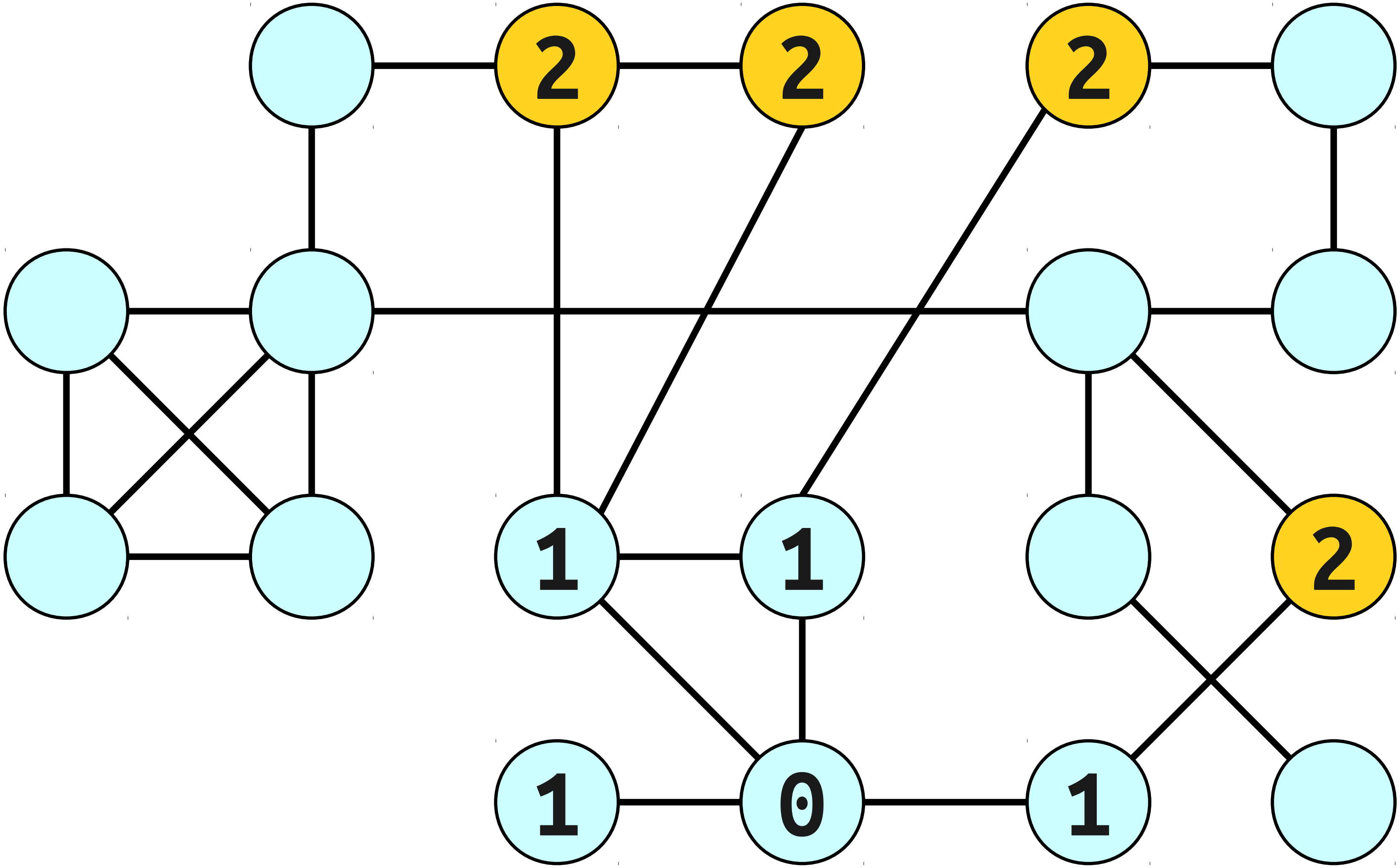
Radiating Outward



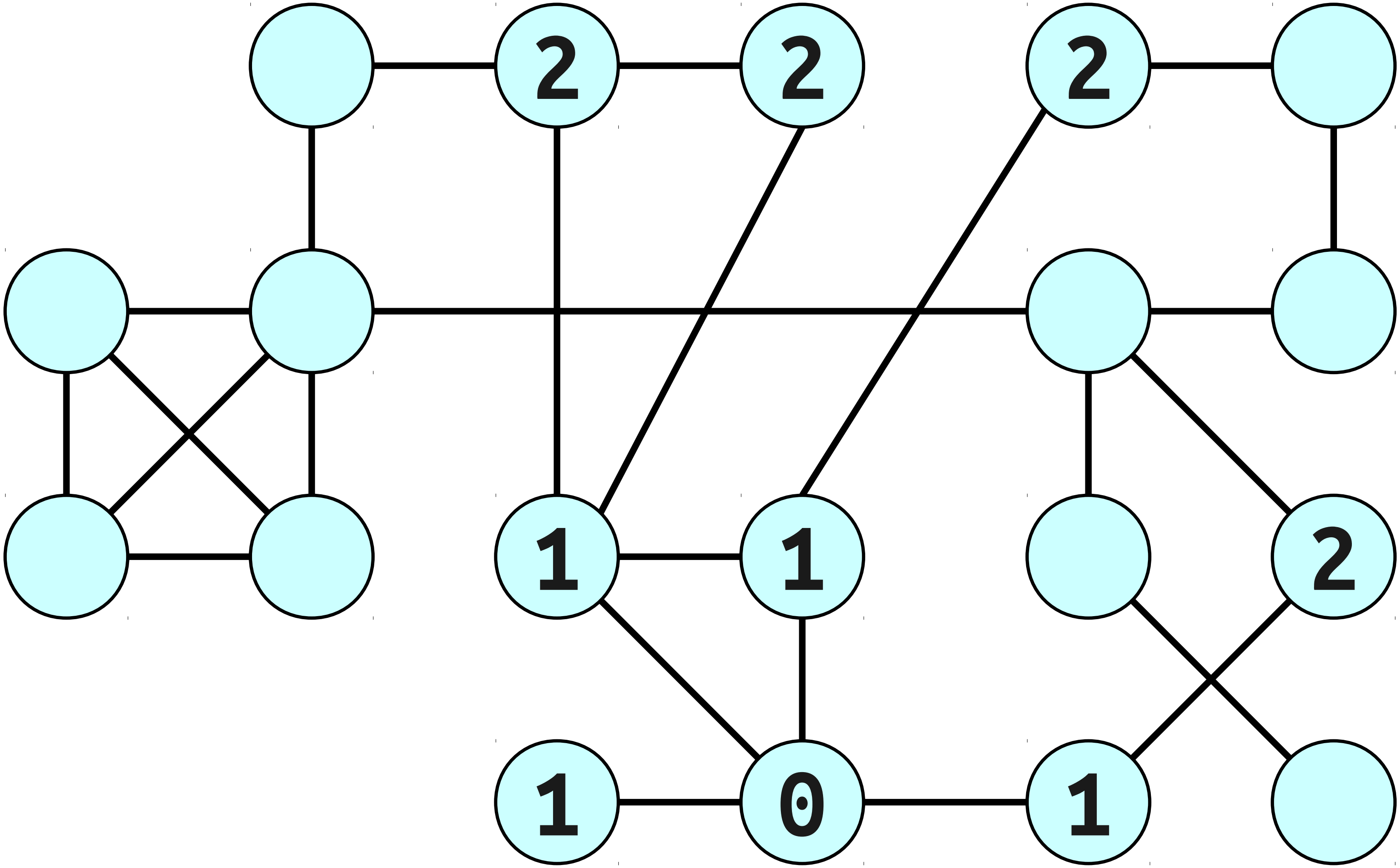
Radiating Outward



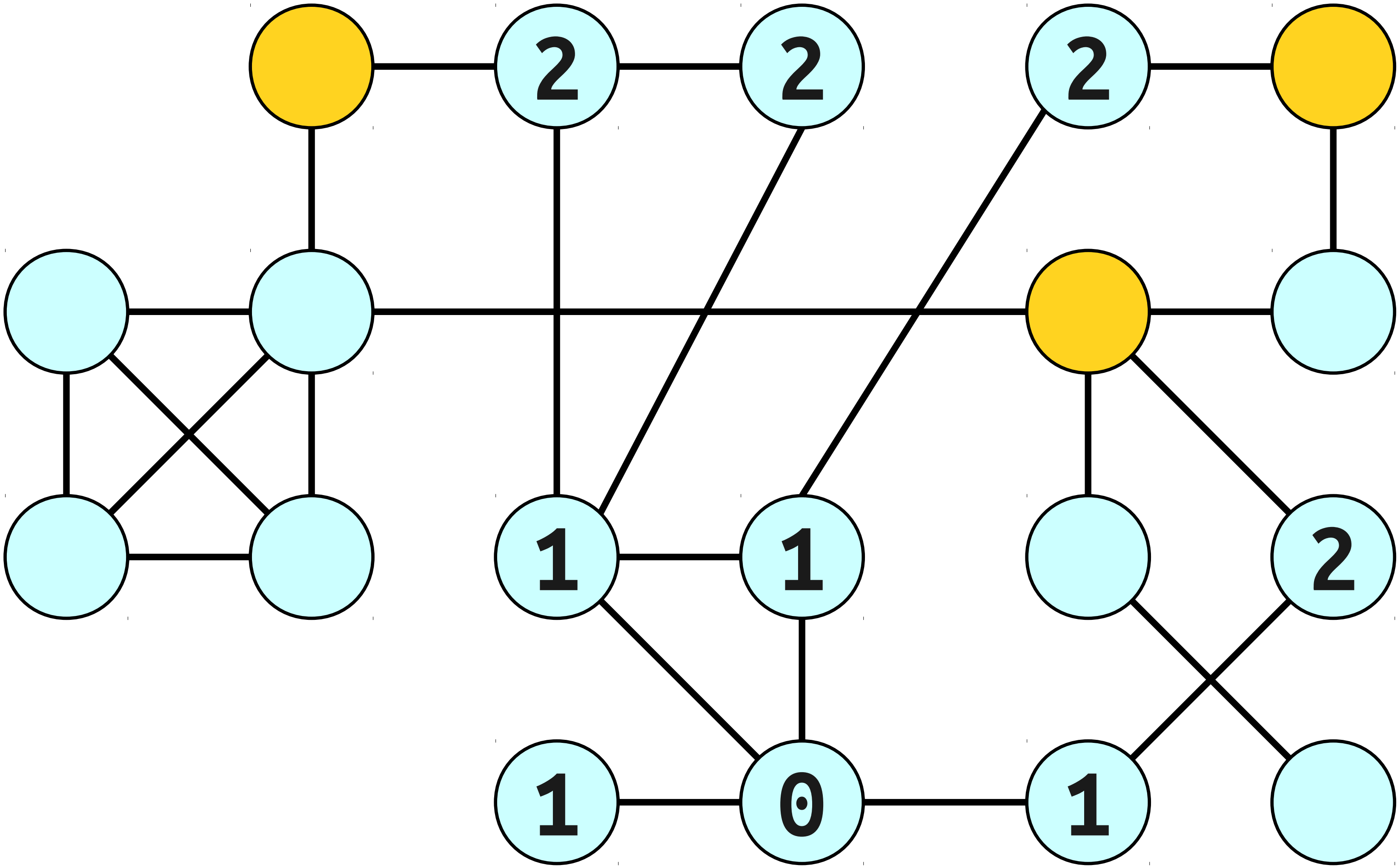
Radiating Outward



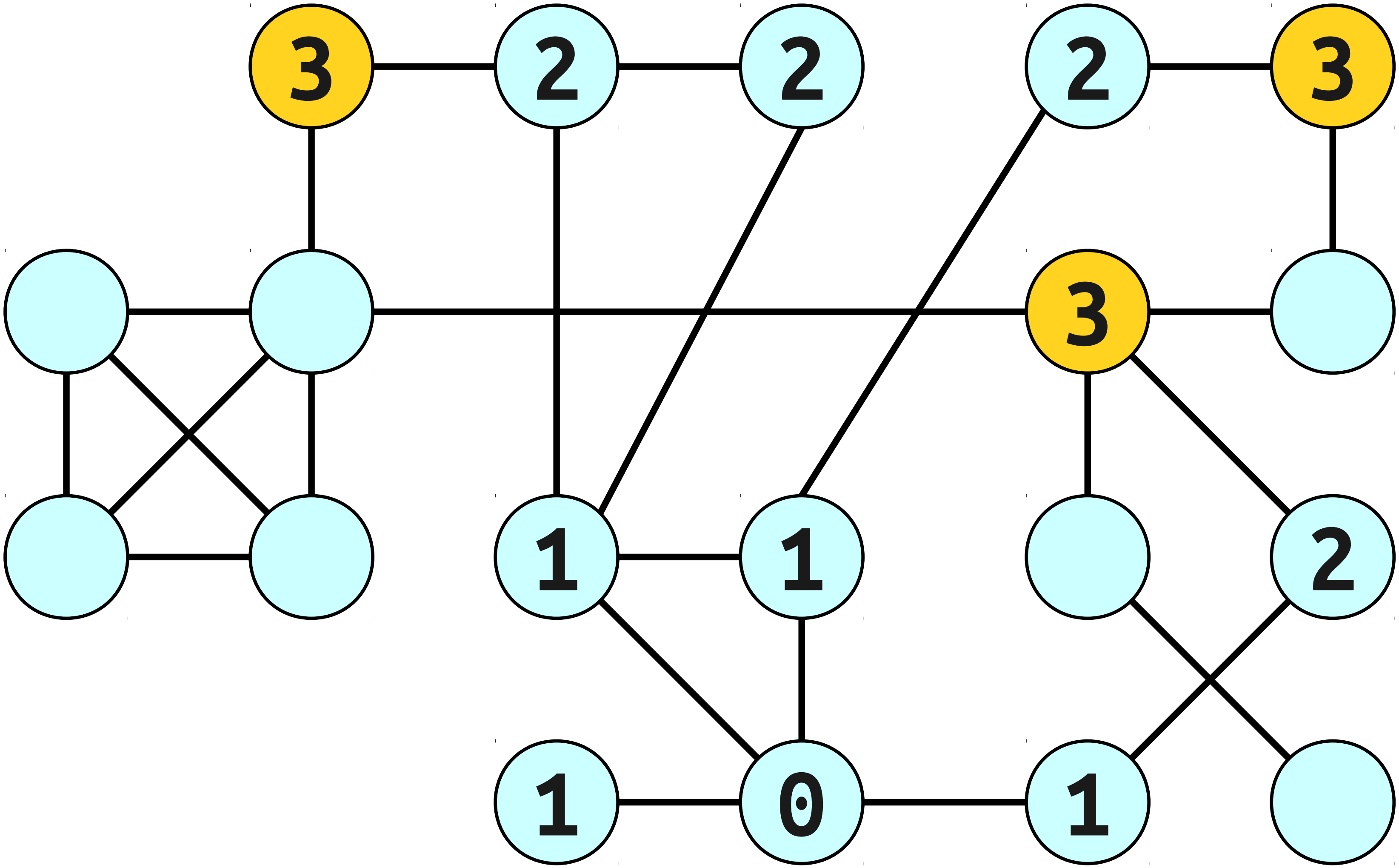
Radiating Outward



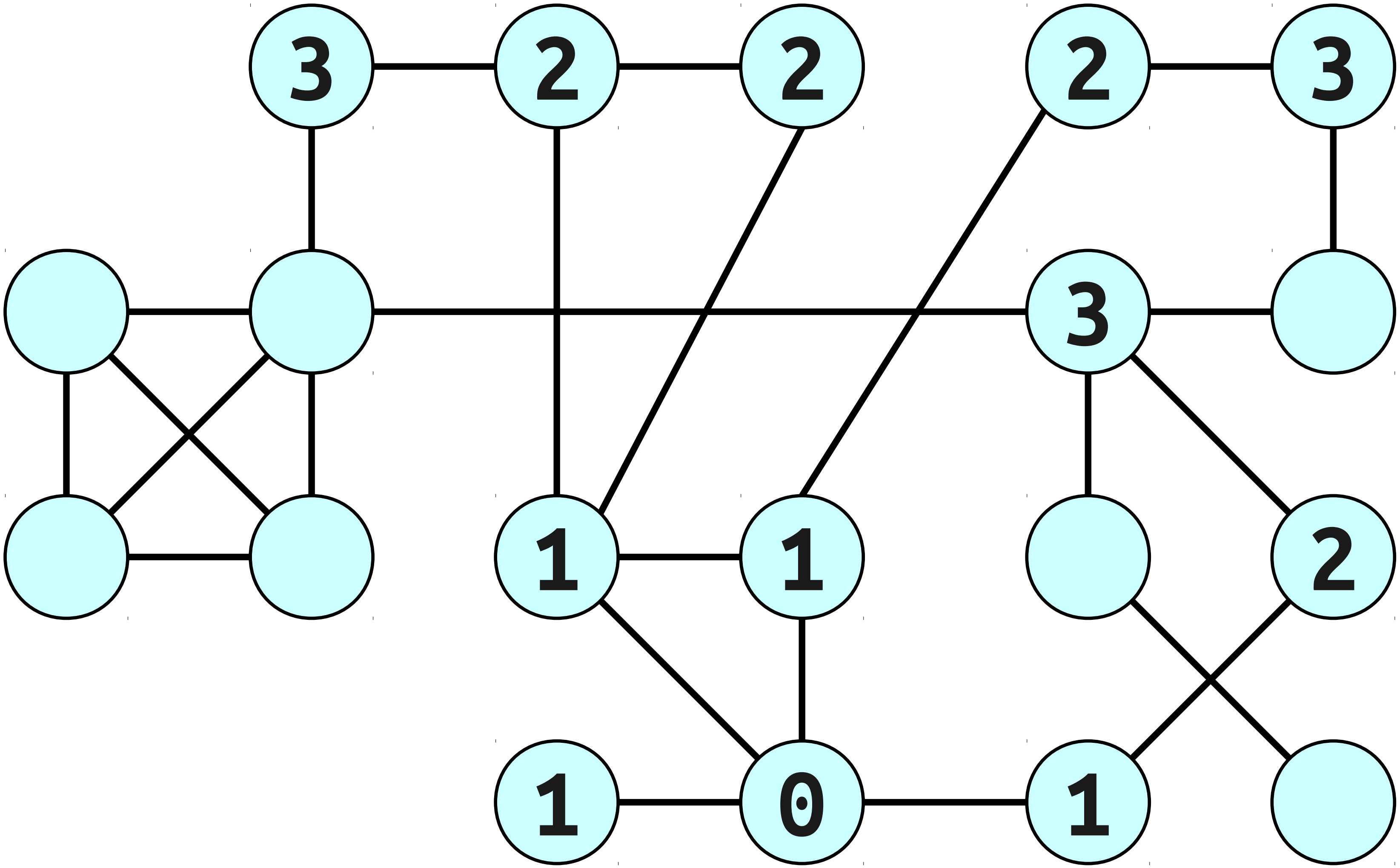
Radiating Outward



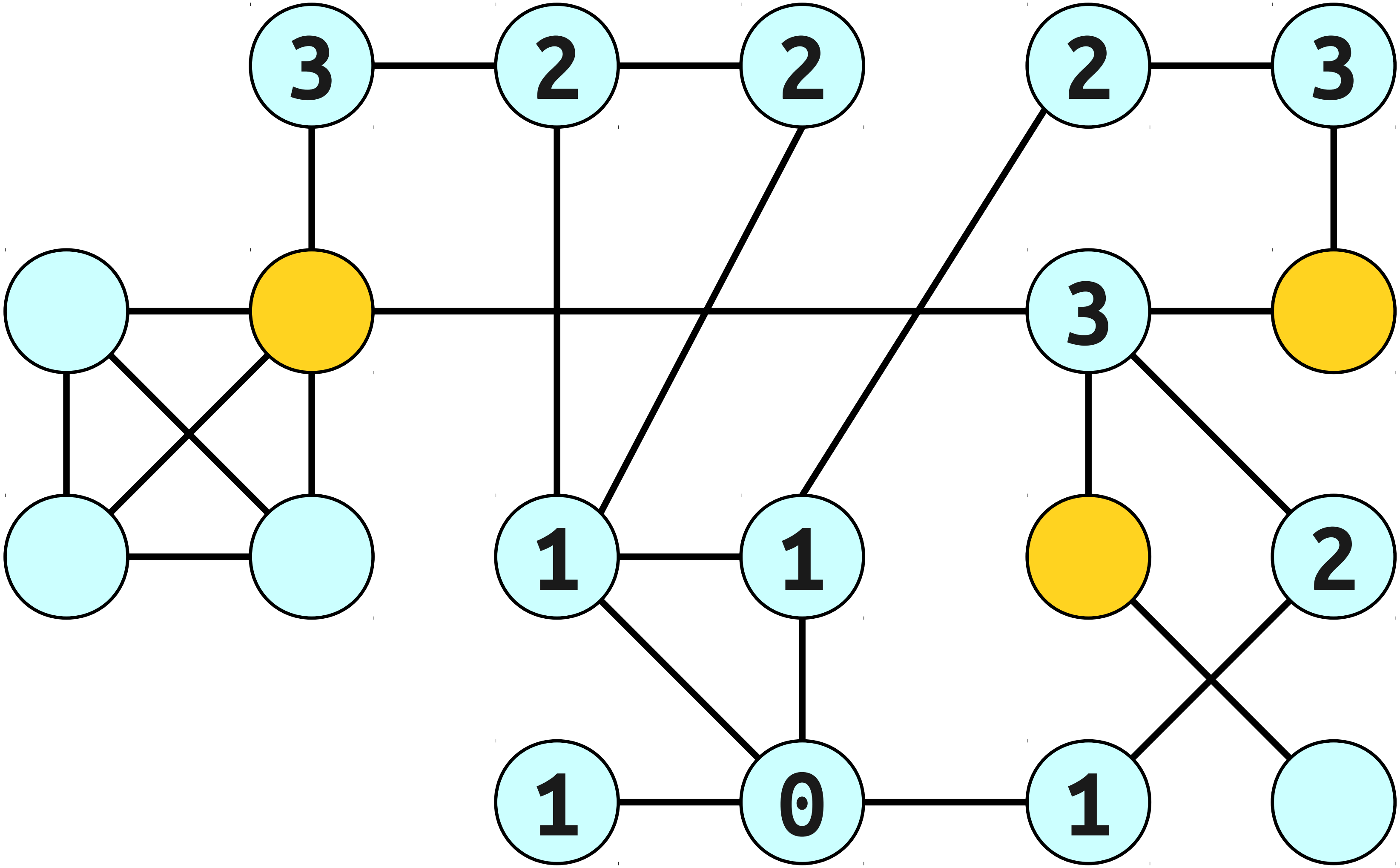
Radiating Outward



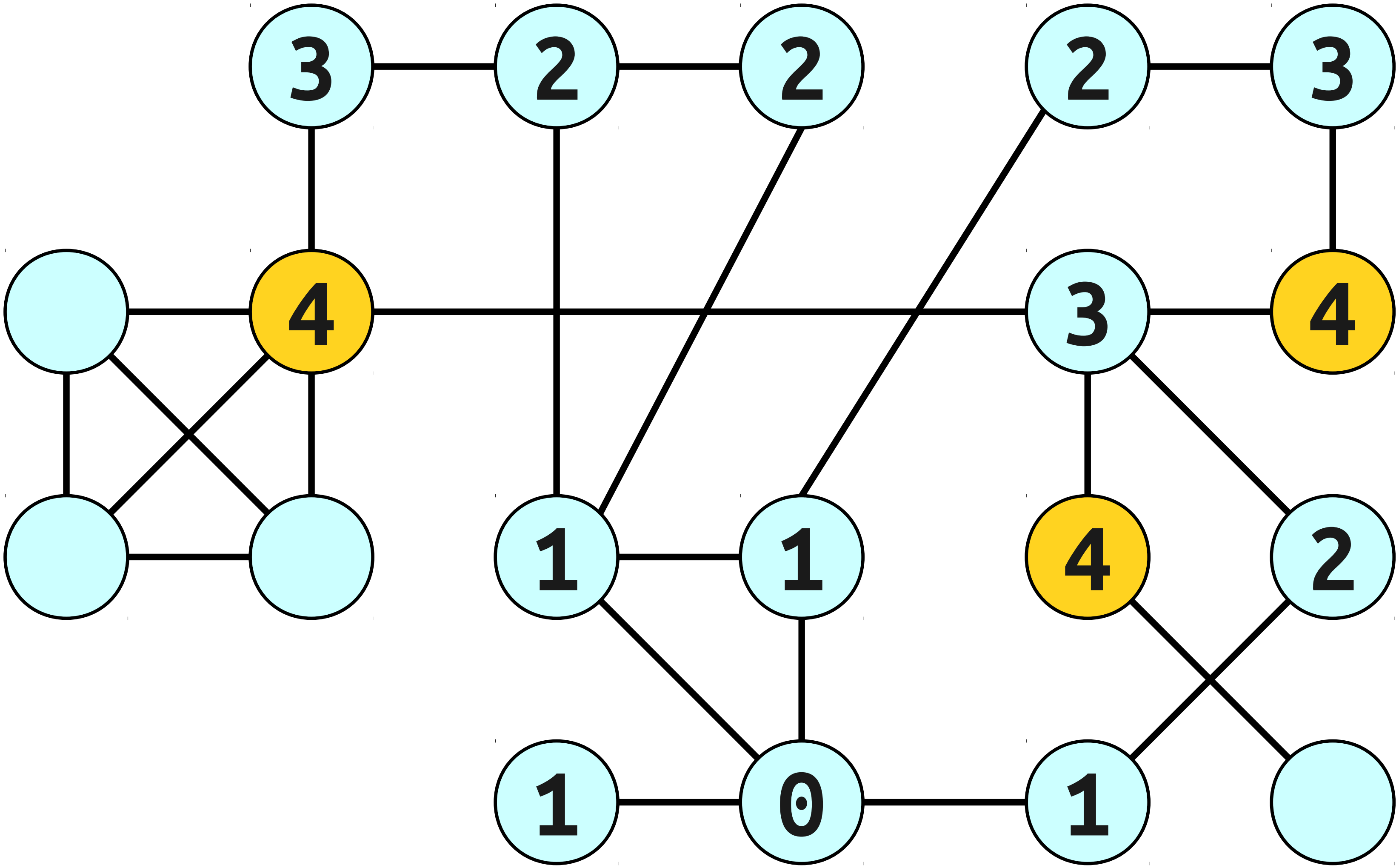
Radiating Outward



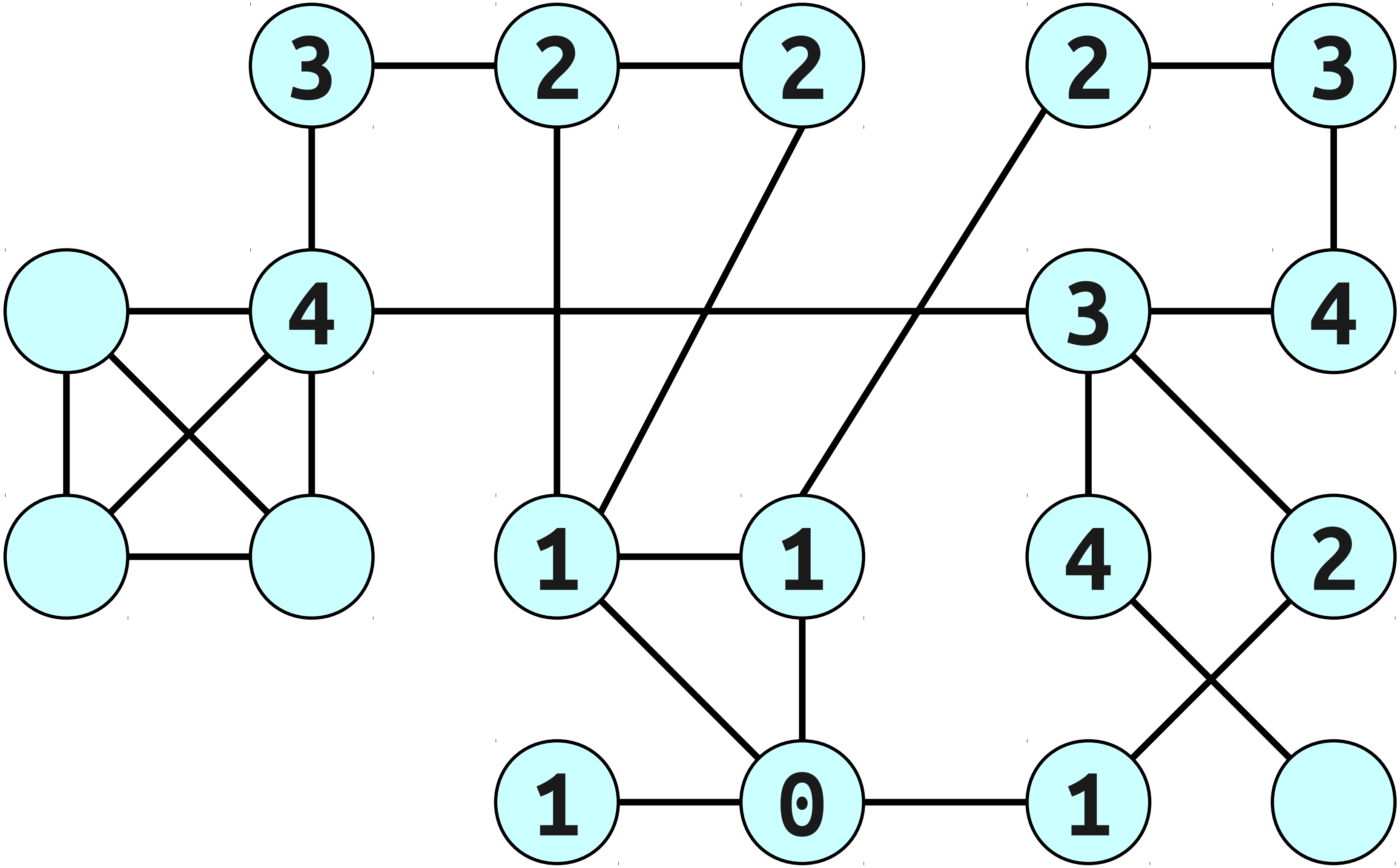
Radiating Outward



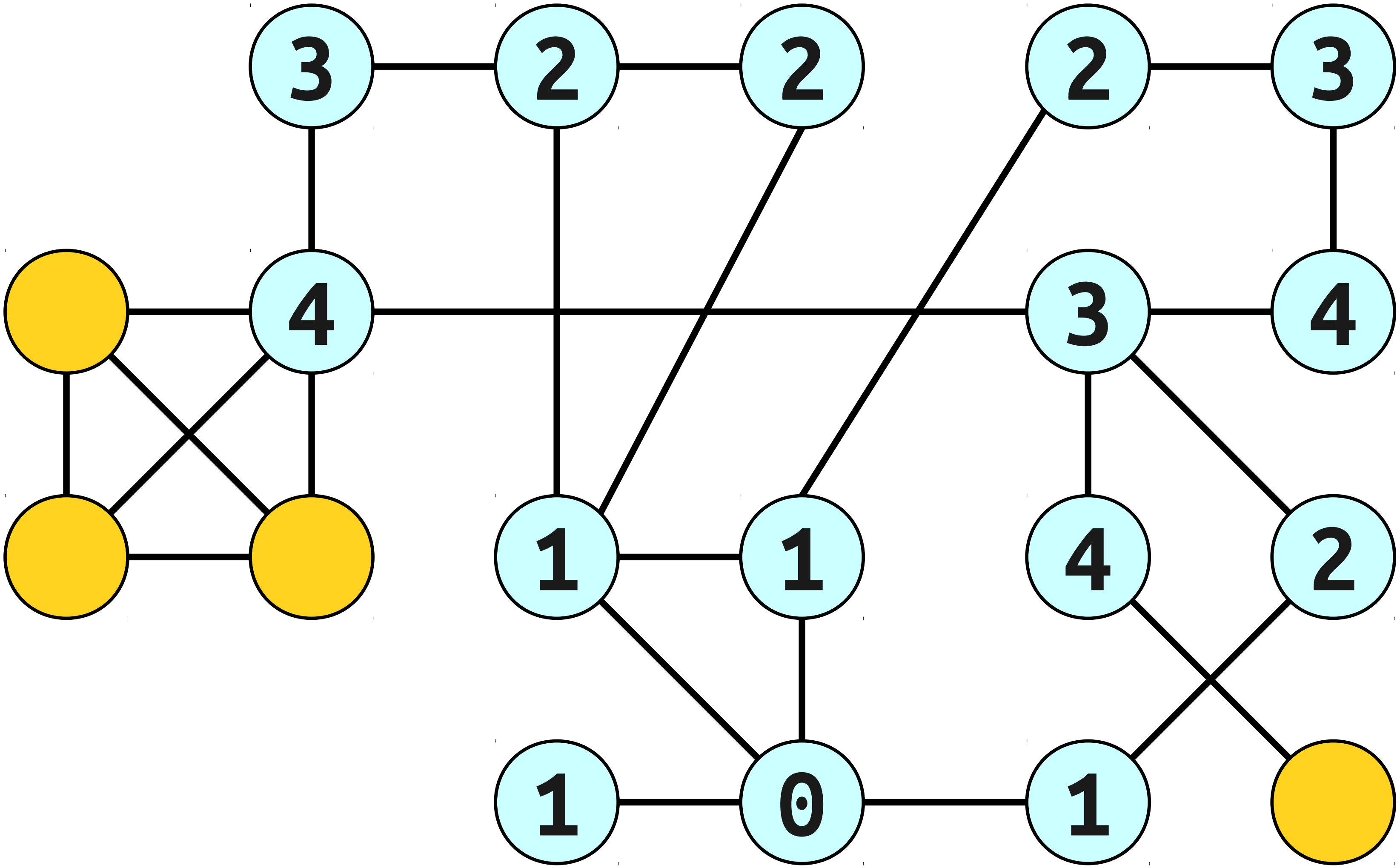
Radiating Outward



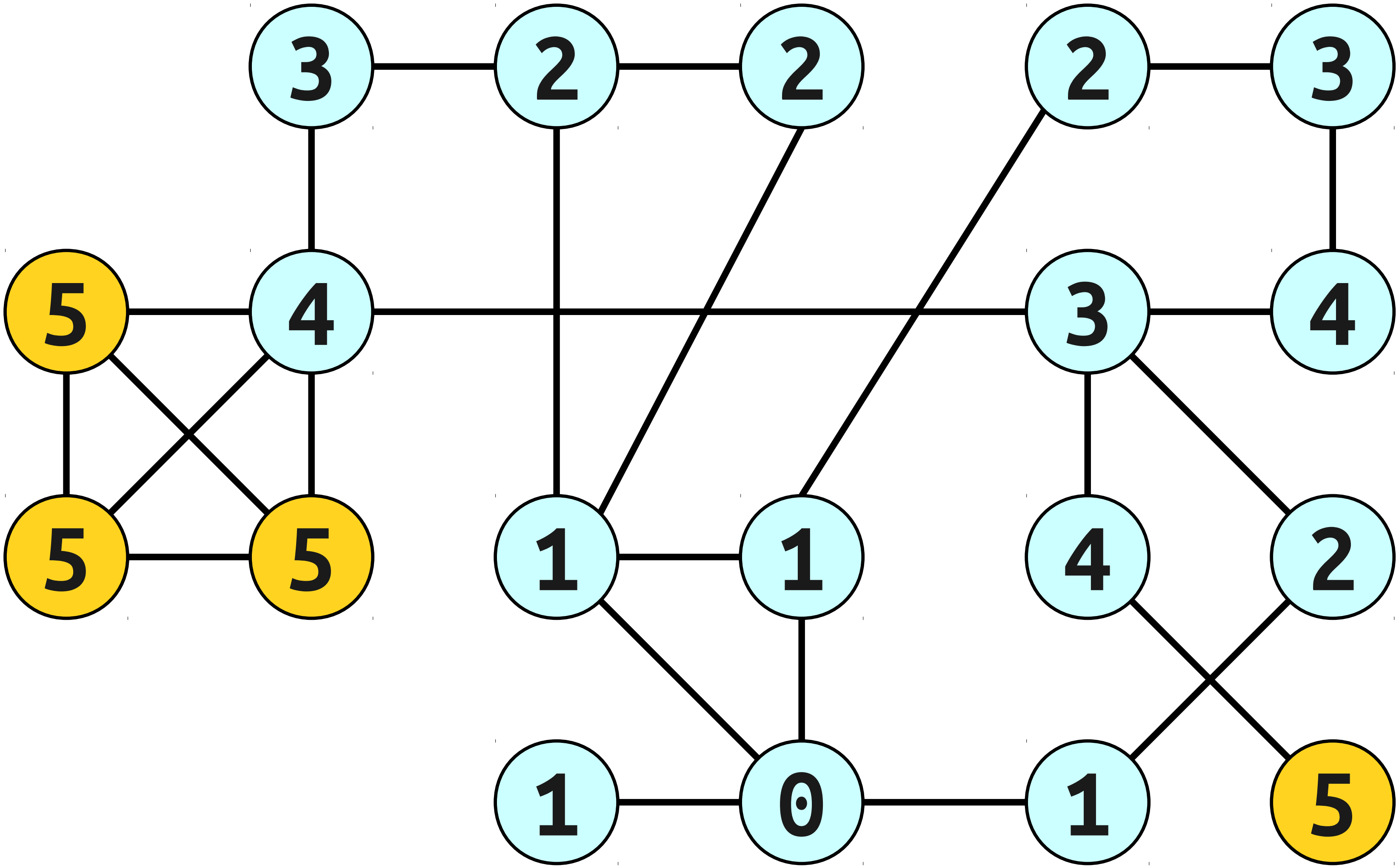
Radiating Outward



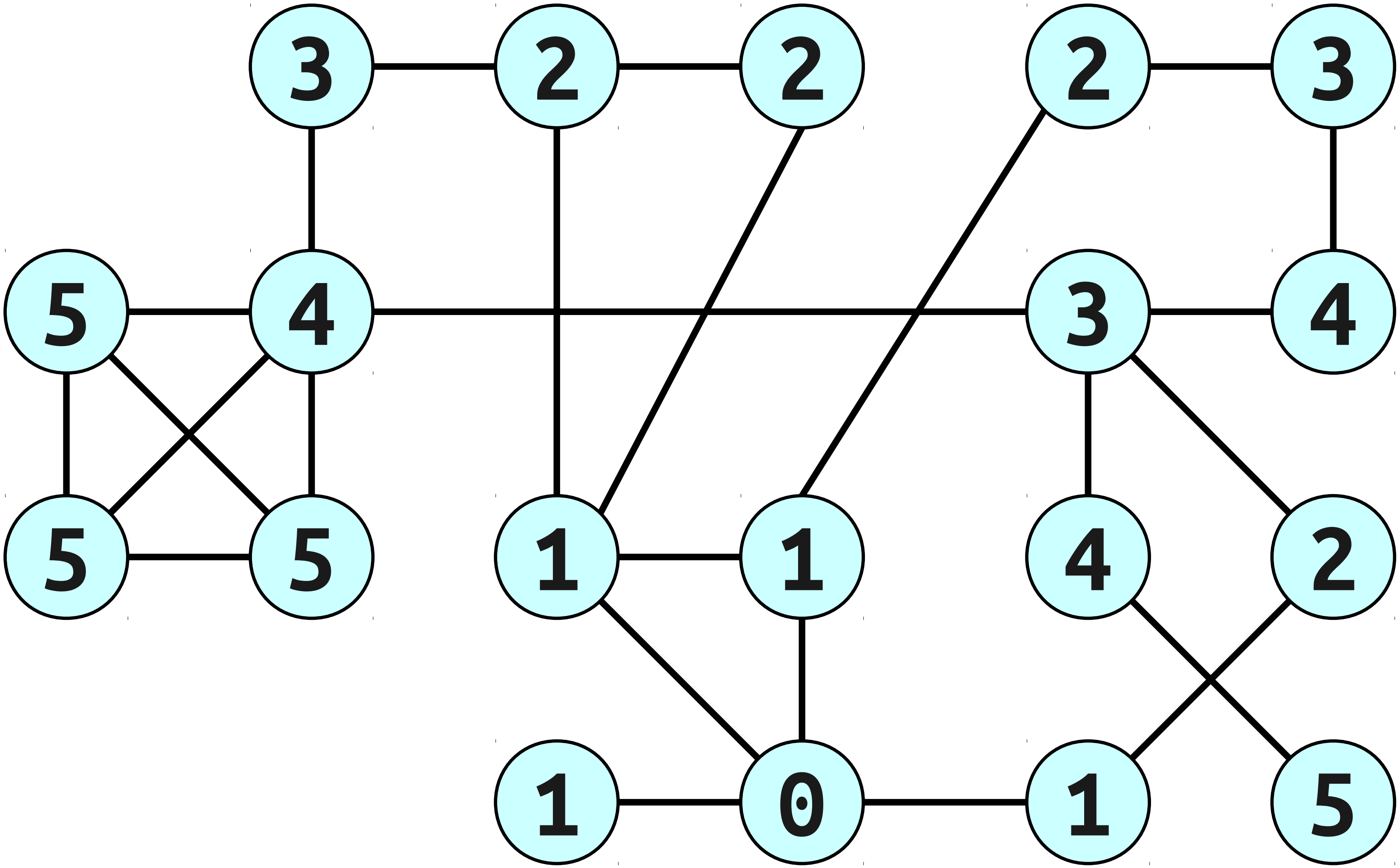
Radiating Outward



Radiating Outward



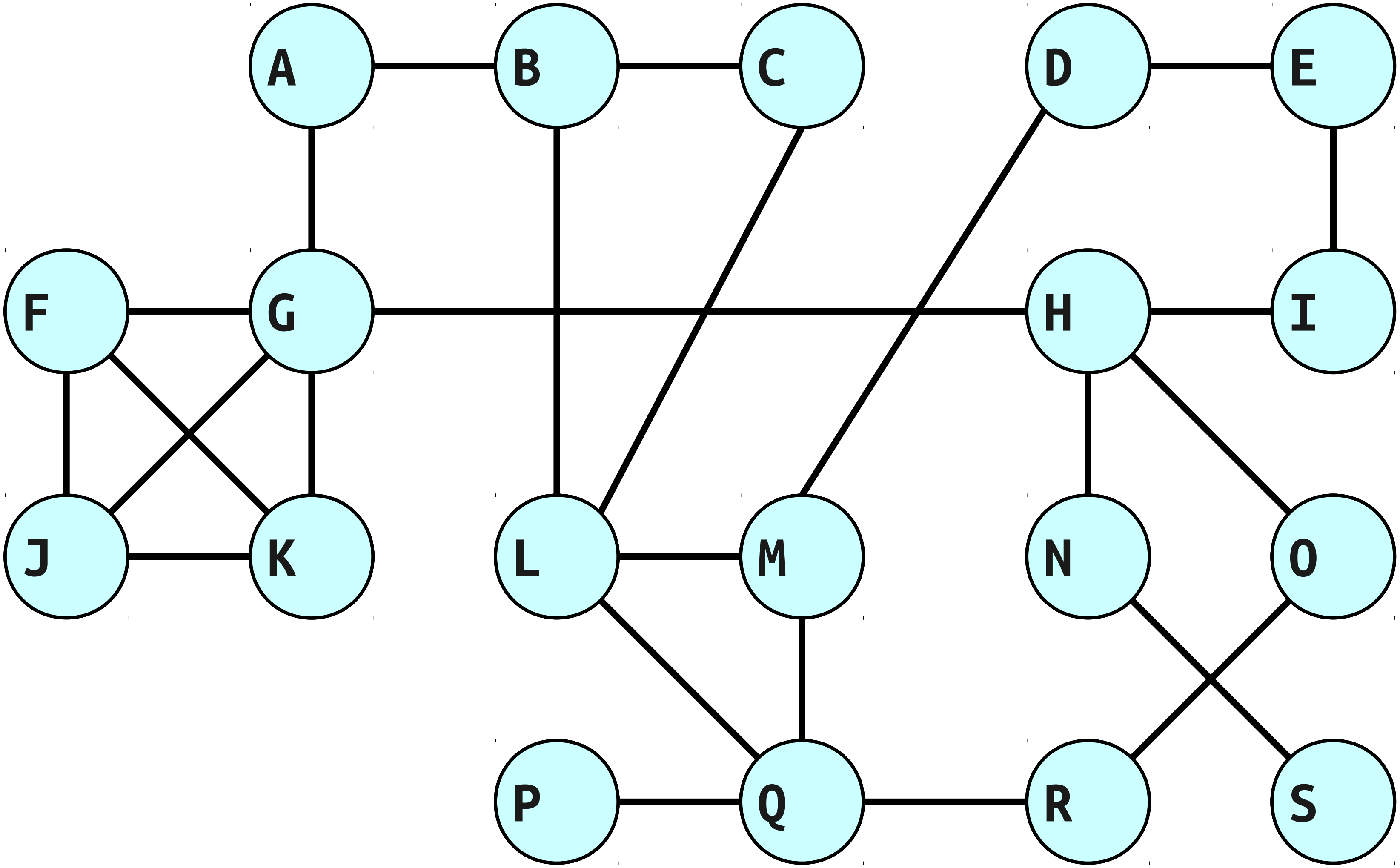
Radiating Outward



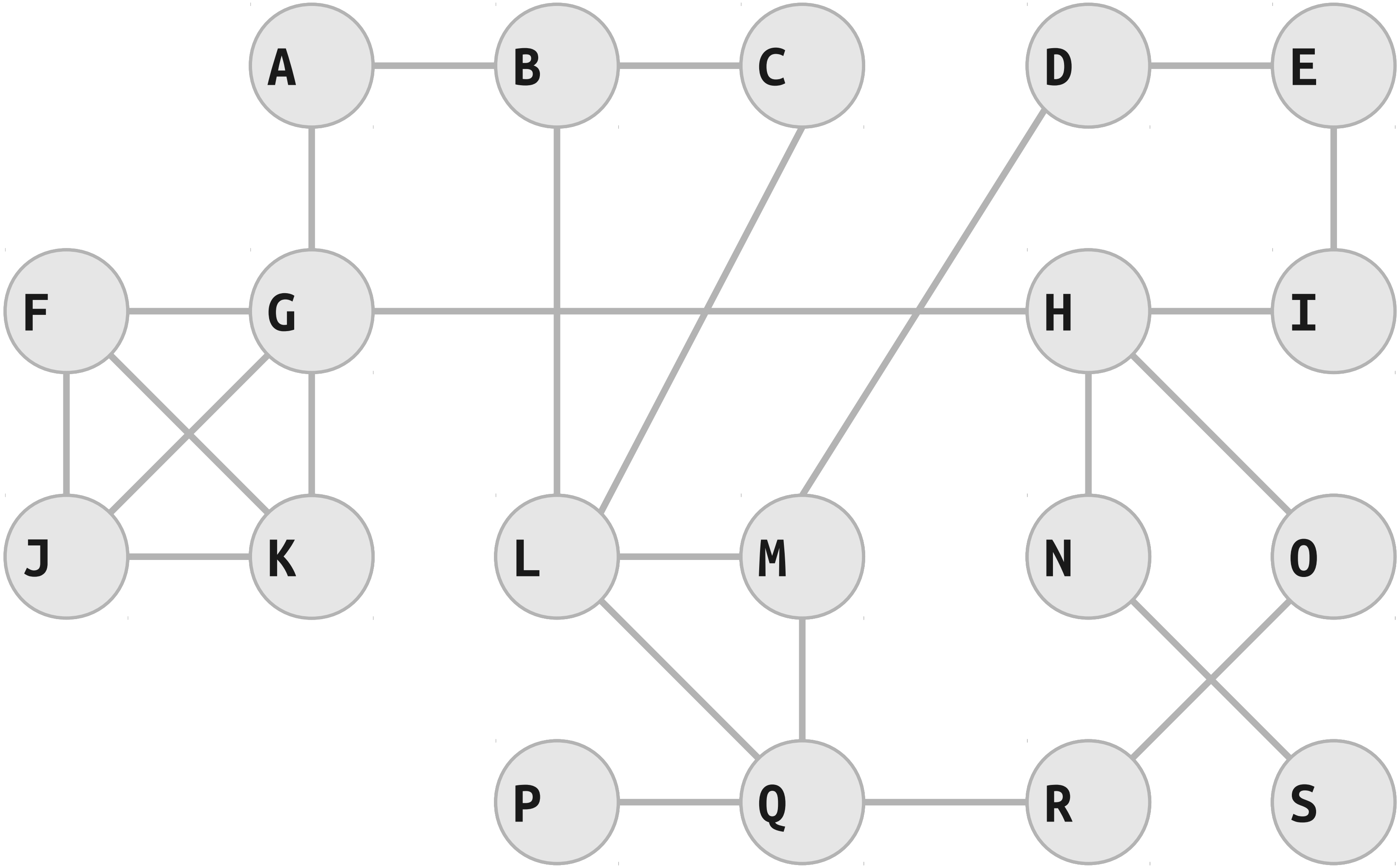
A Secondary Idea

- Proceed outward from the source node *s* in “layers.”
  - The first layer is all nodes of distance 0.
  - The second layer is all nodes of distance 1.
  - The third layer is all nodes of distance 2.
  - etc.
- This gives rise to **breadth-first search**.

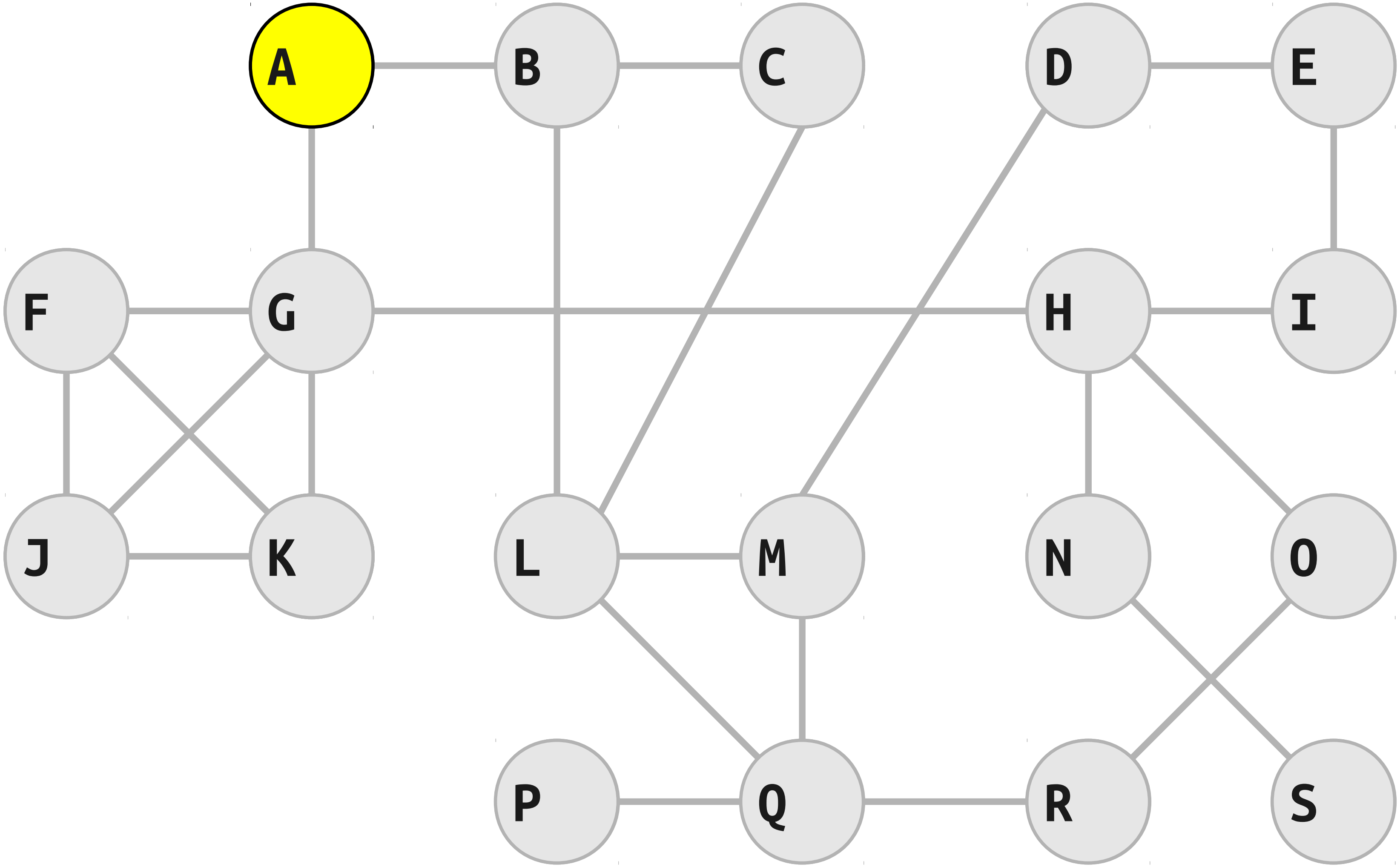
Breadth-First Search



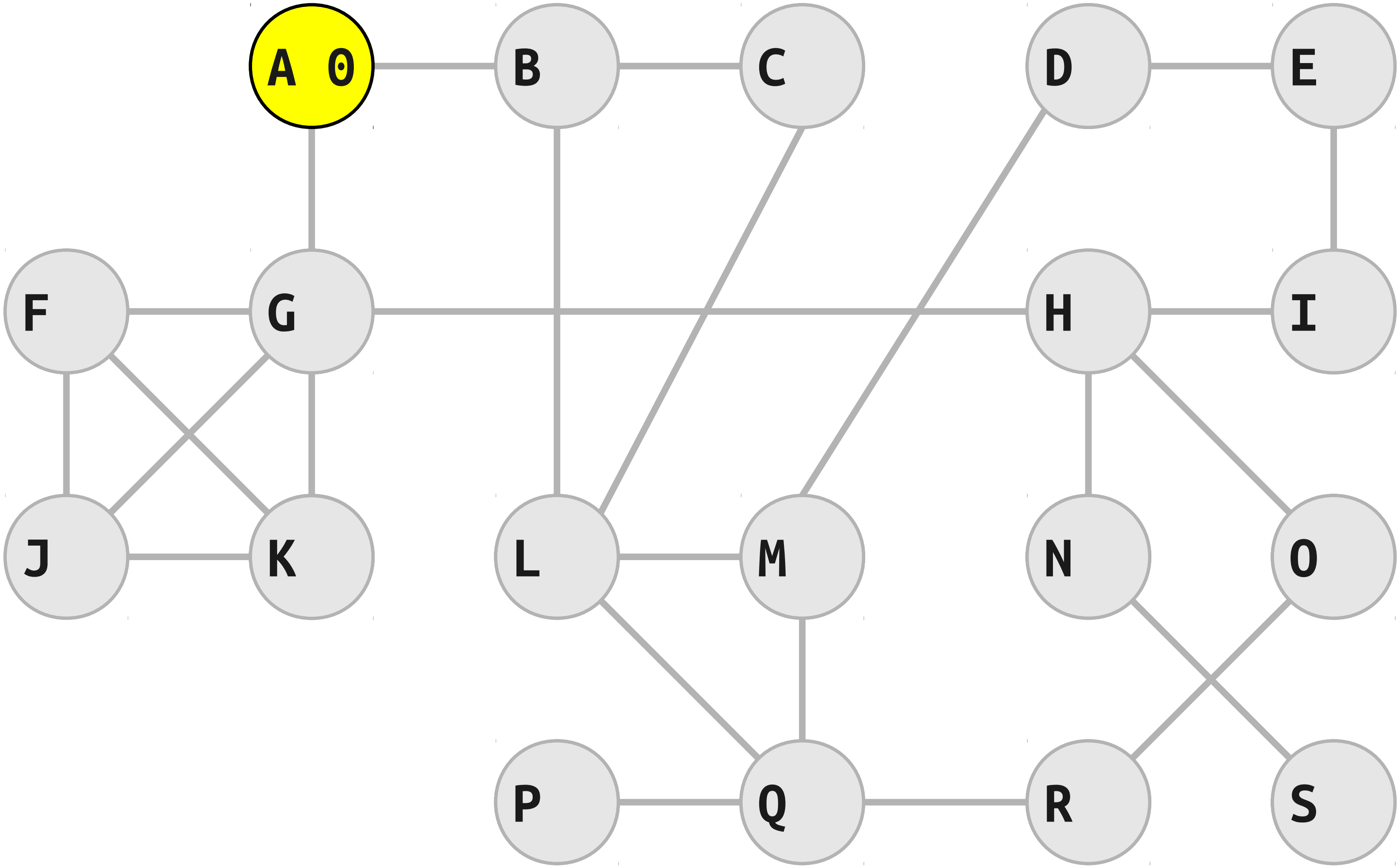
Breadth-First Search



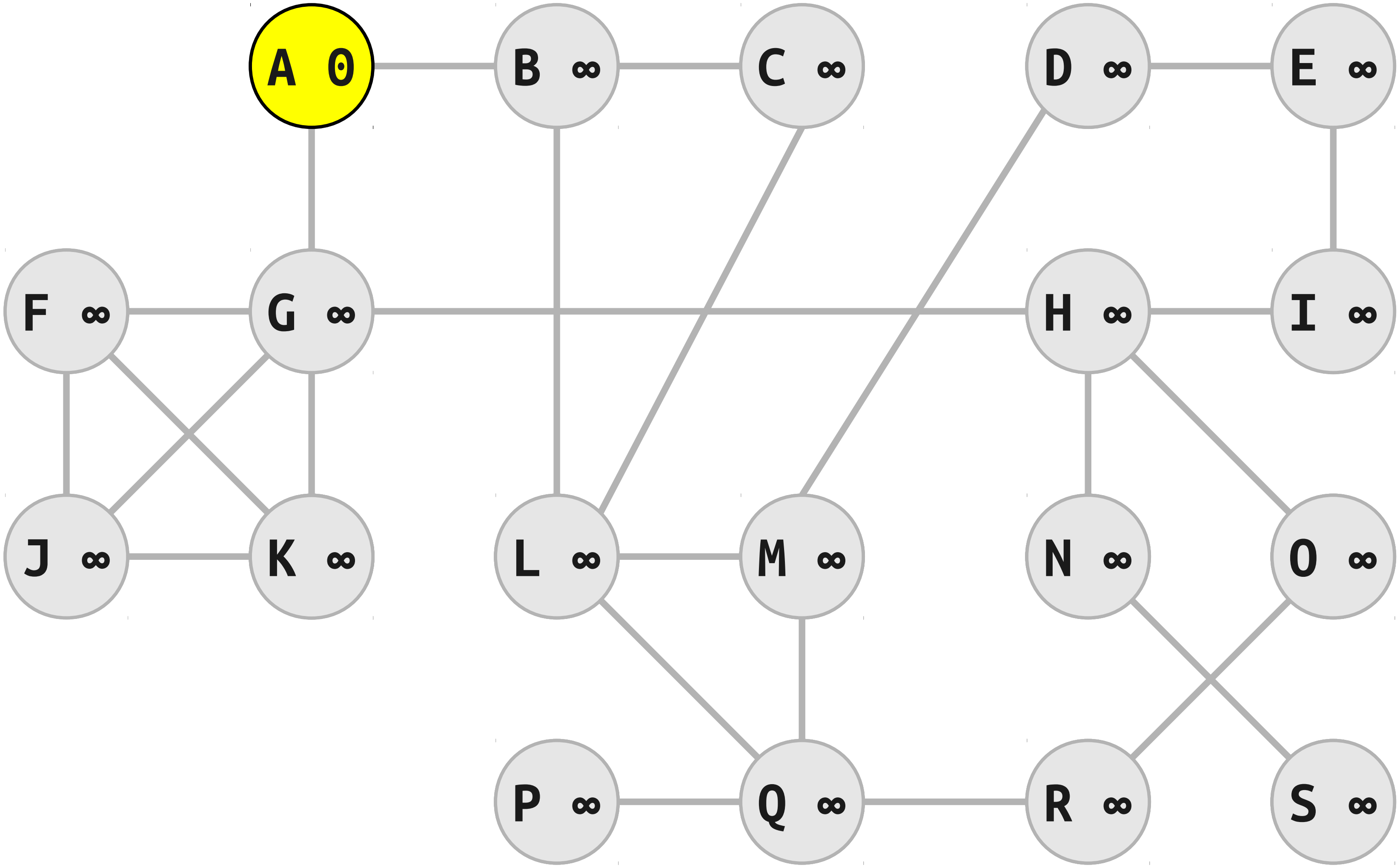
Breadth-First Search



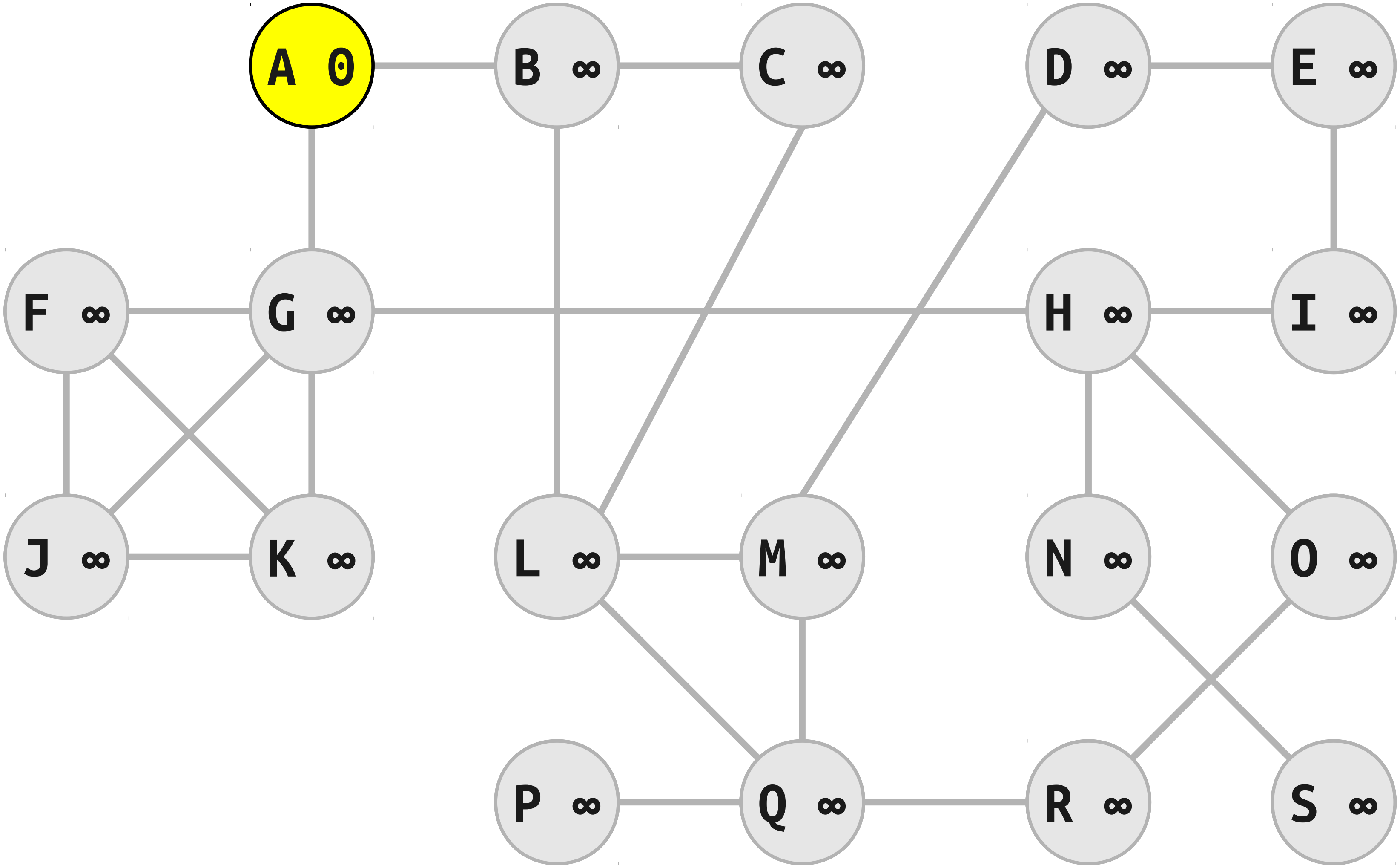
Breadth-First Search



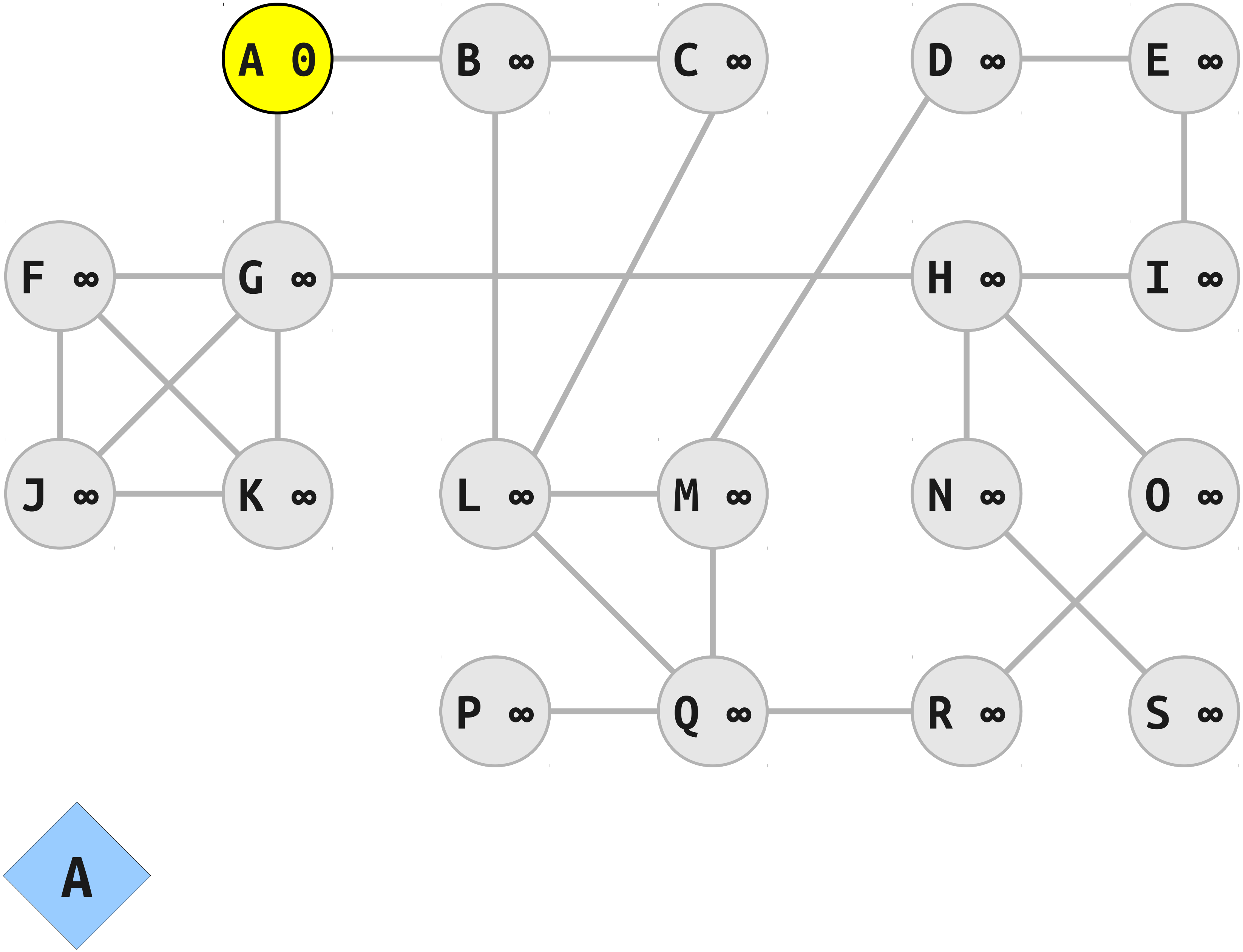
Breadth-First Search



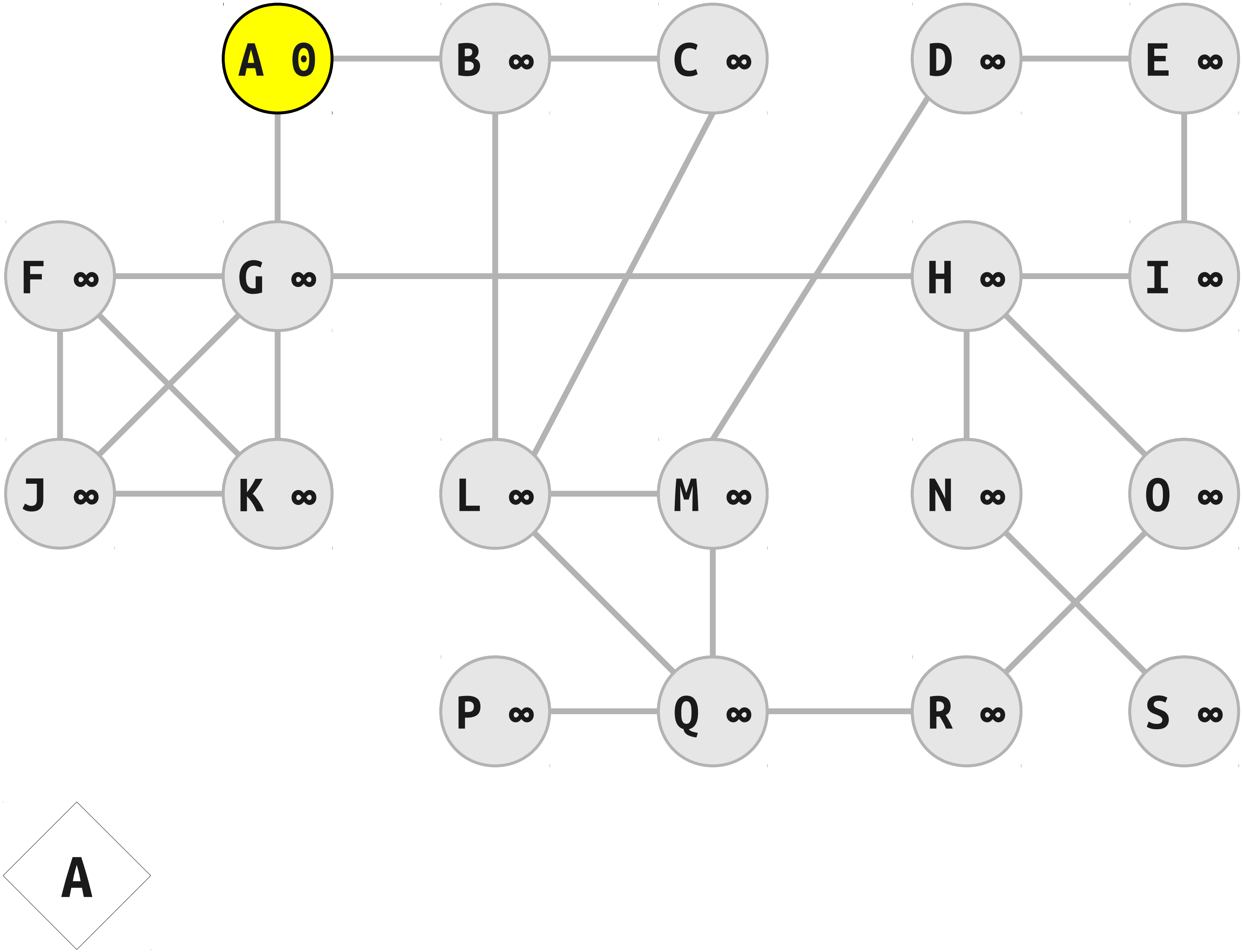
Breadth-First Search



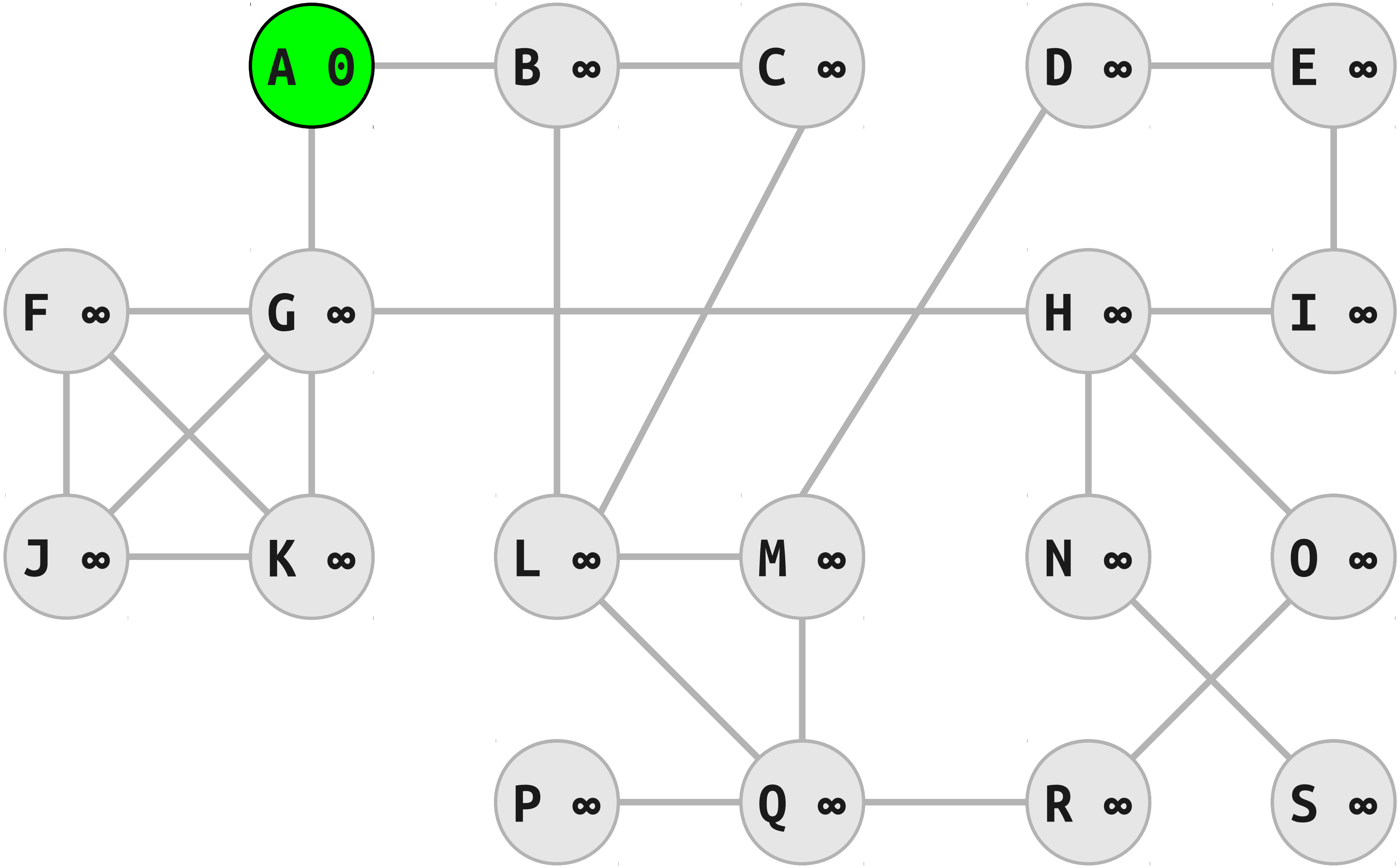
Breadth-First Search



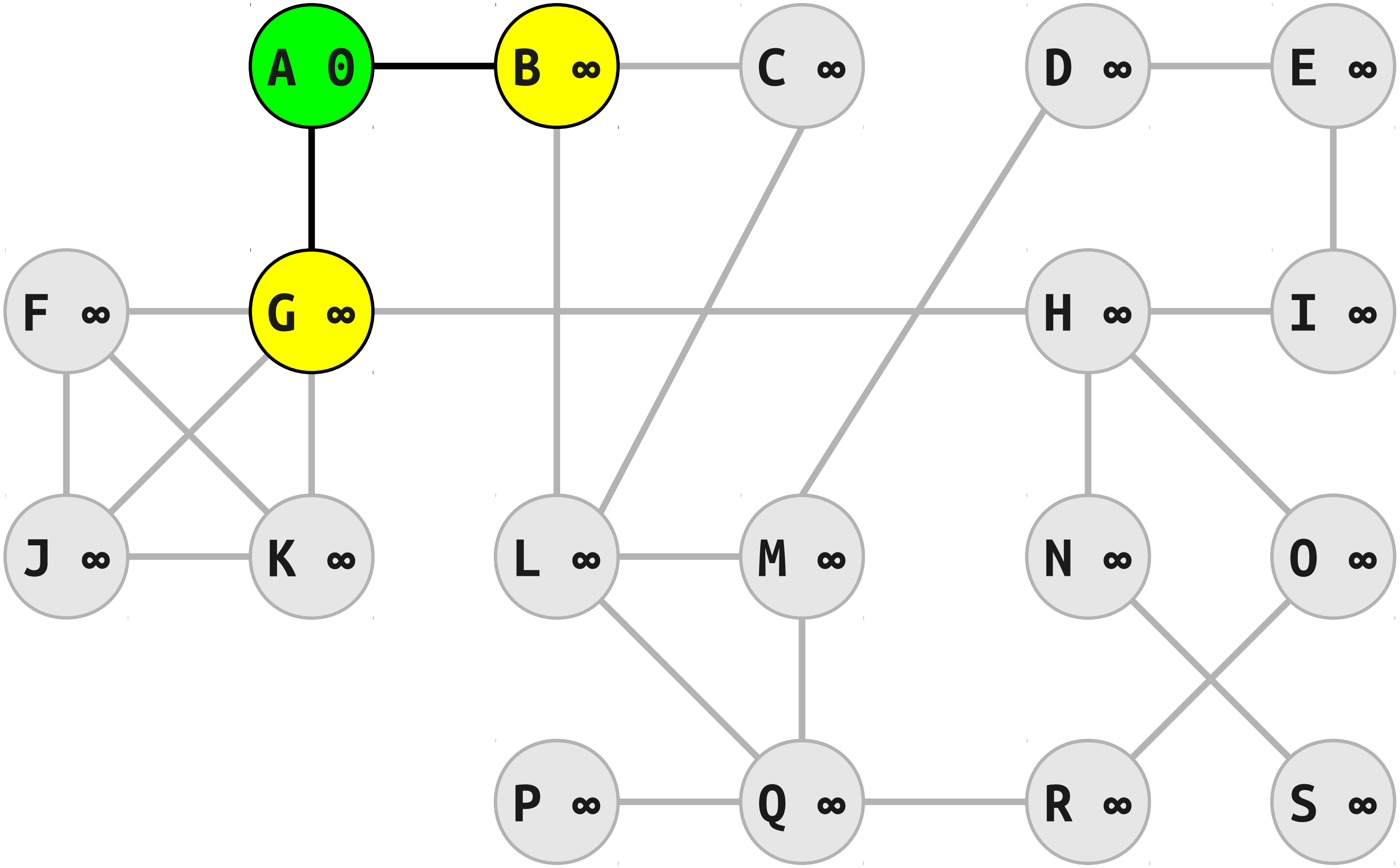
Breadth-First Search



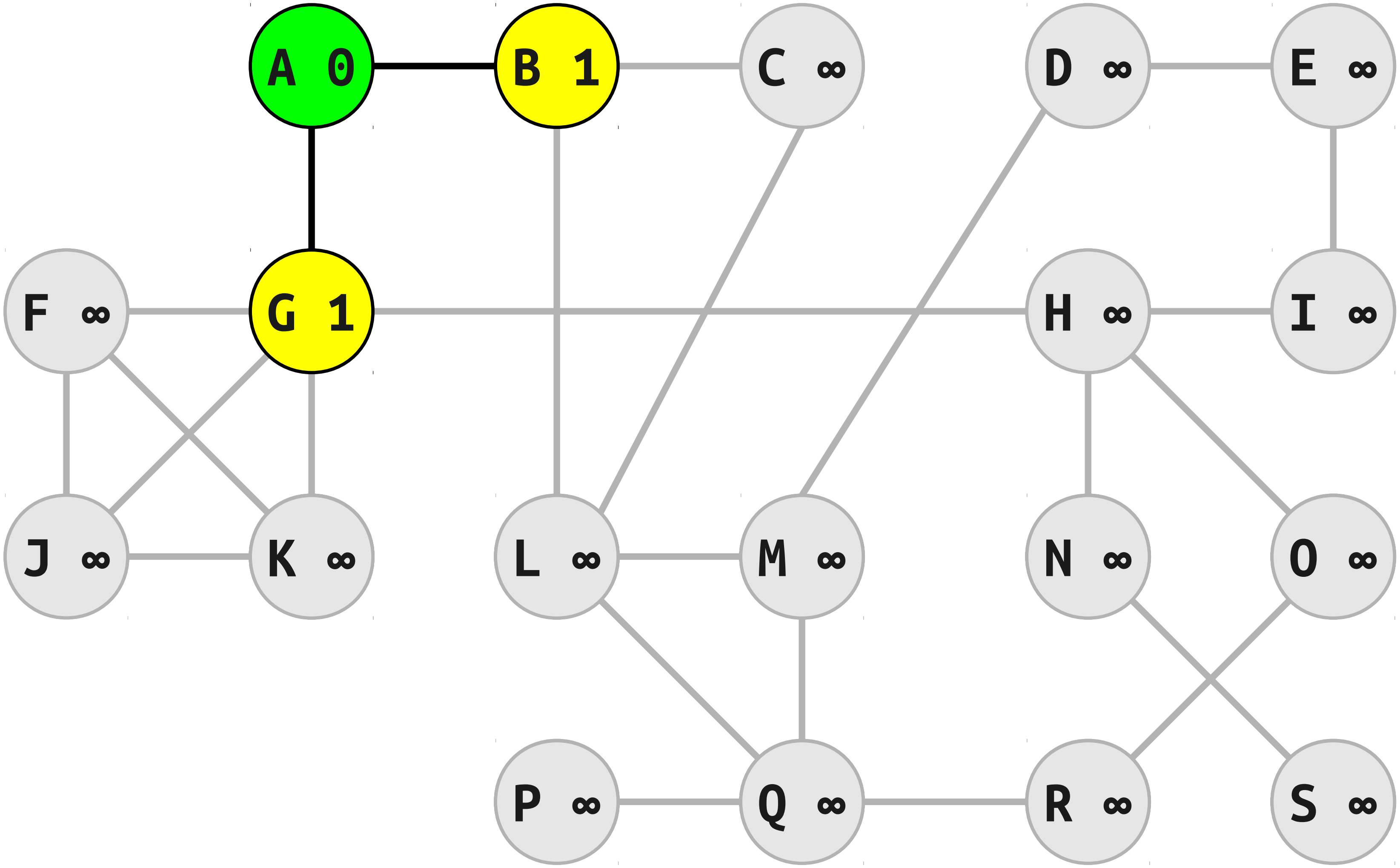
Breadth-First Search



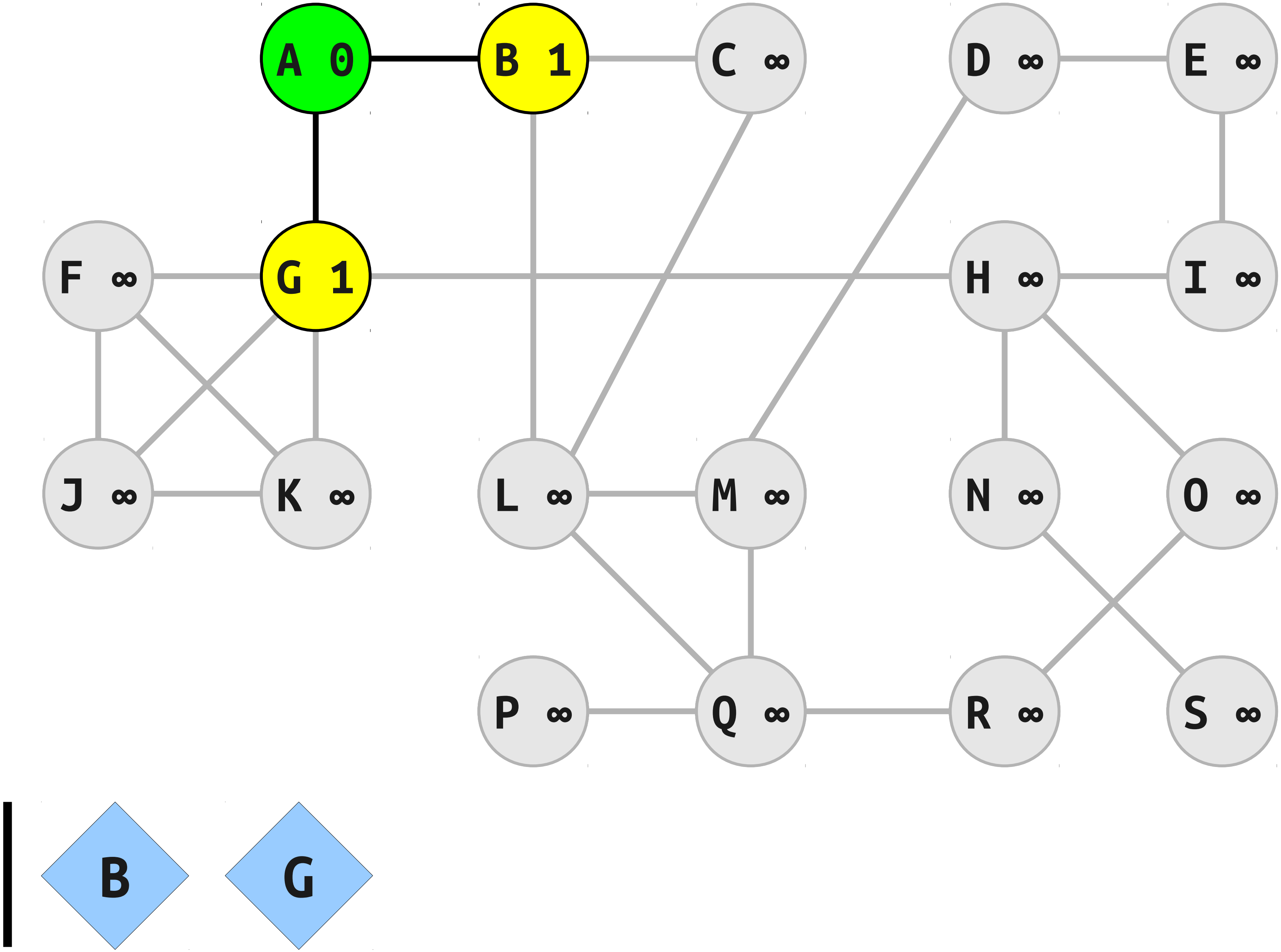
Breadth-First Search



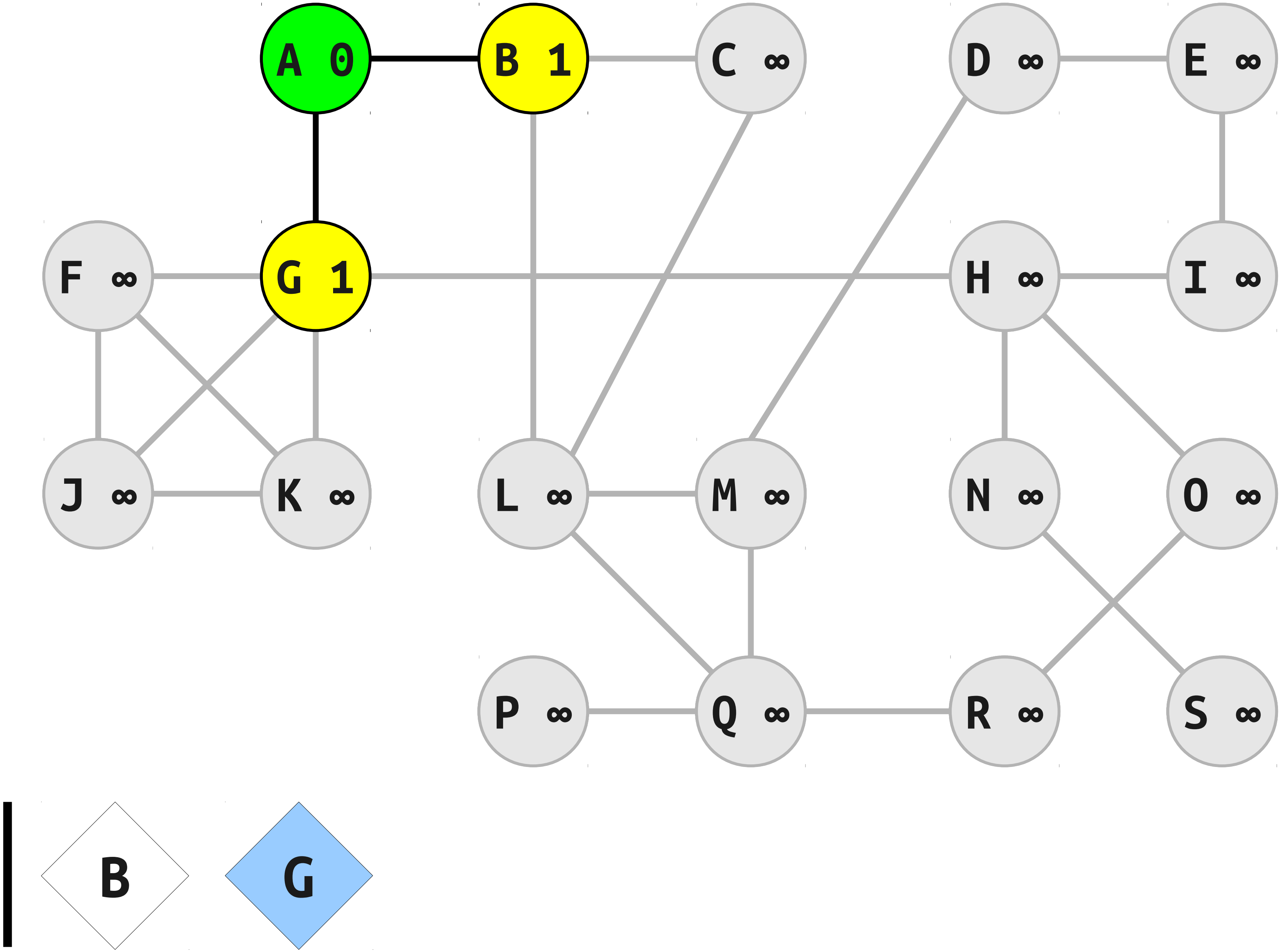
Breadth-First Search



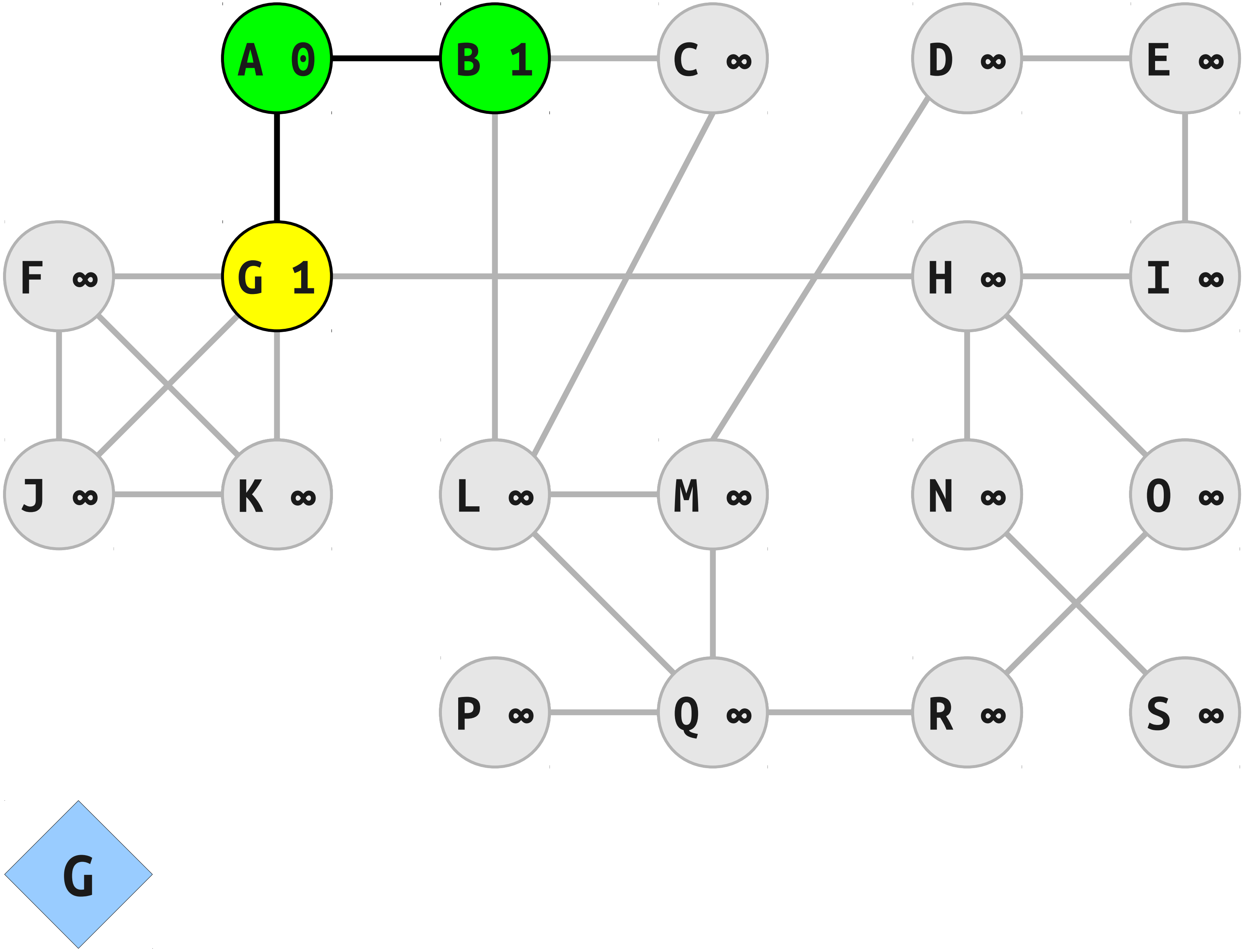
Breadth-First Search



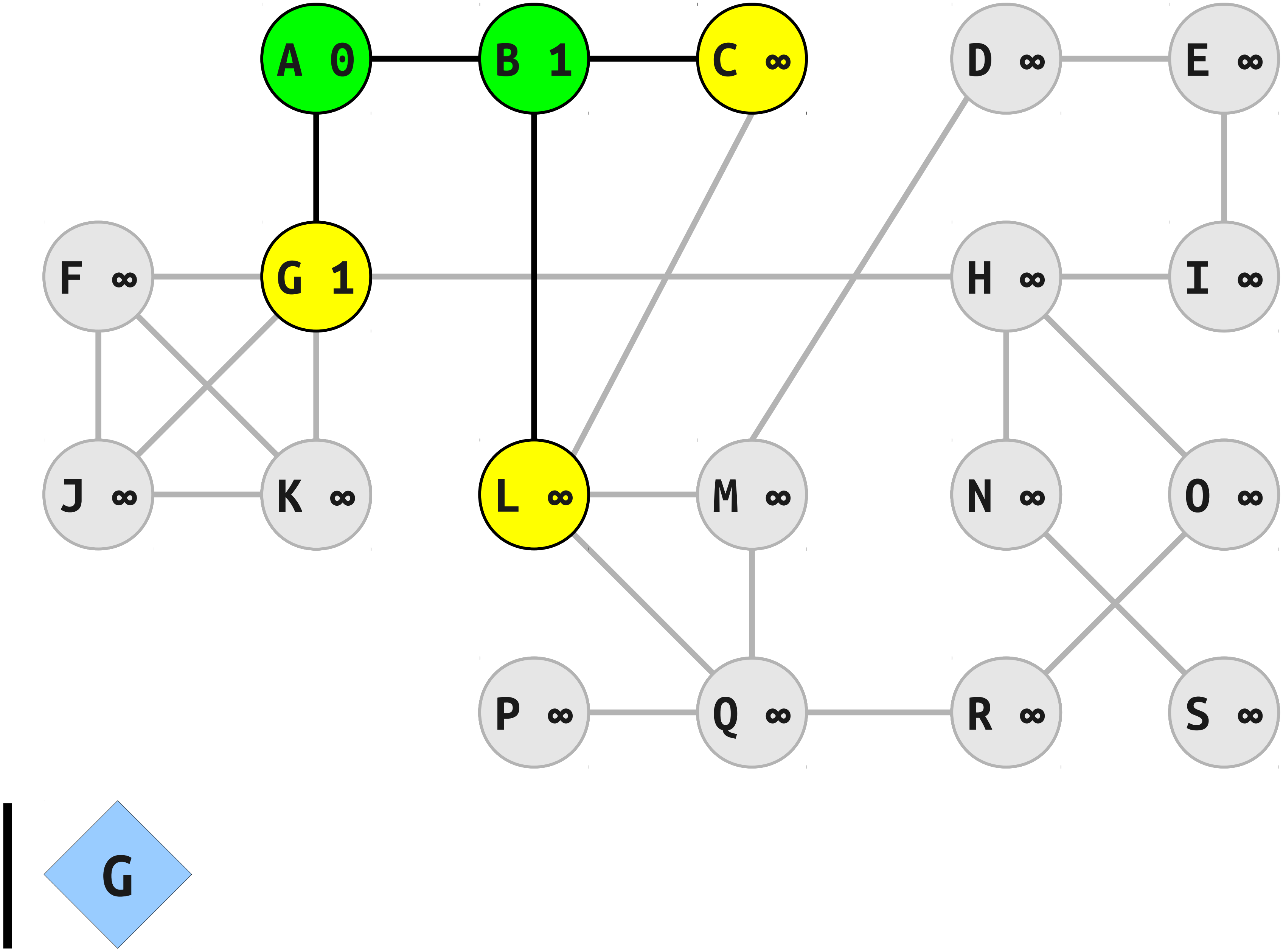
Breadth-First Search



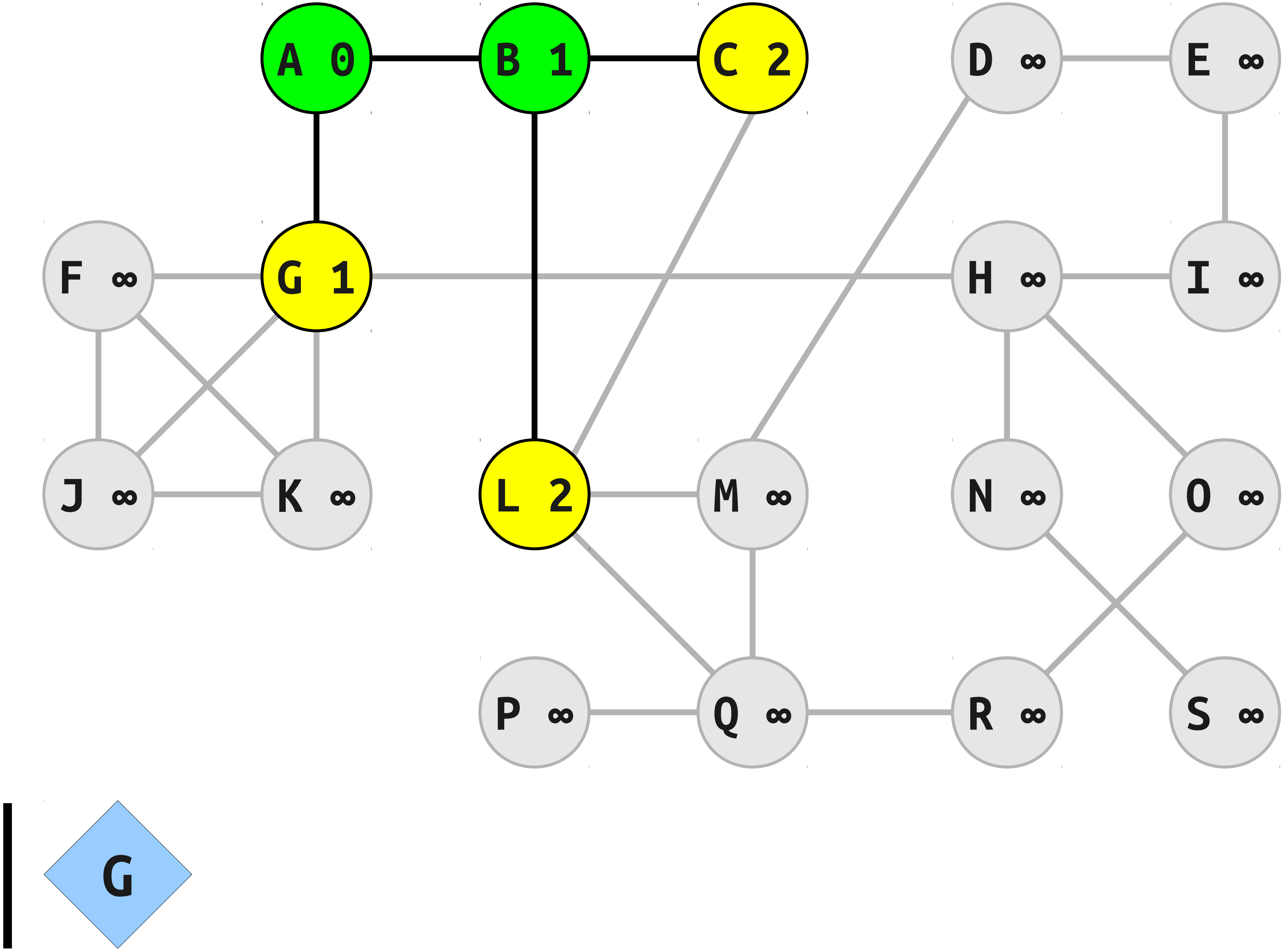
Breadth-First Search



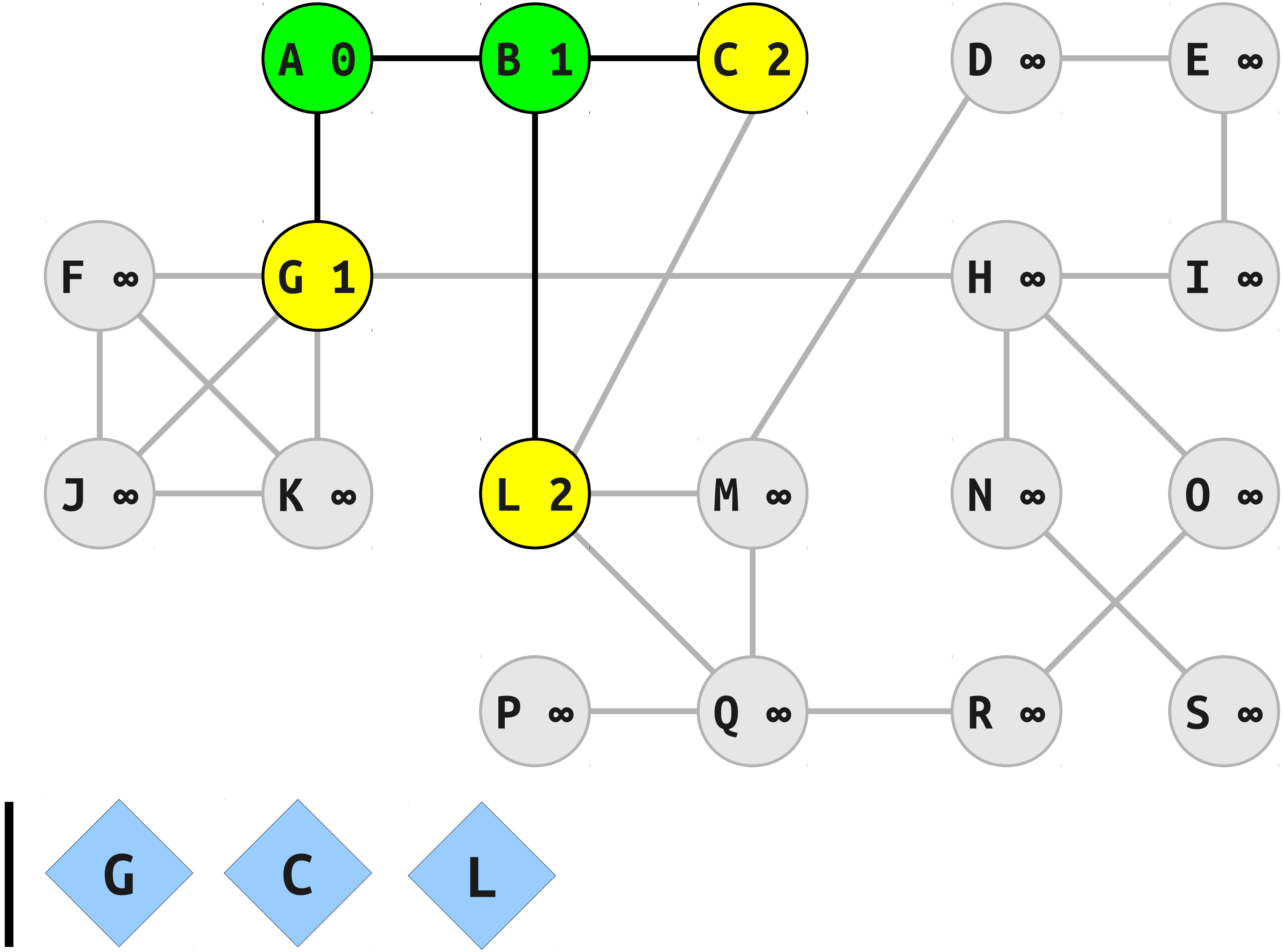
Breadth-First Search



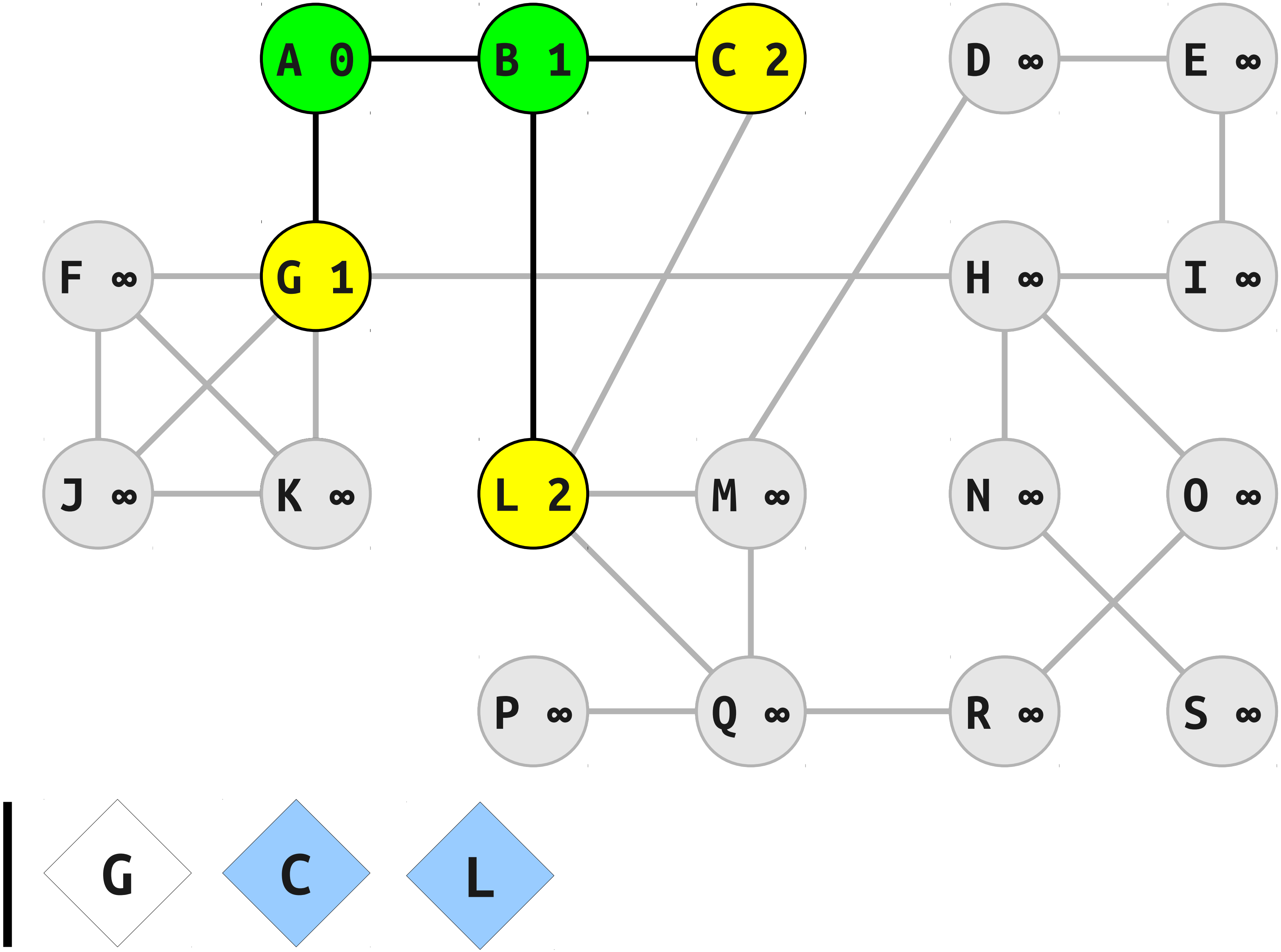
Breadth-First Search



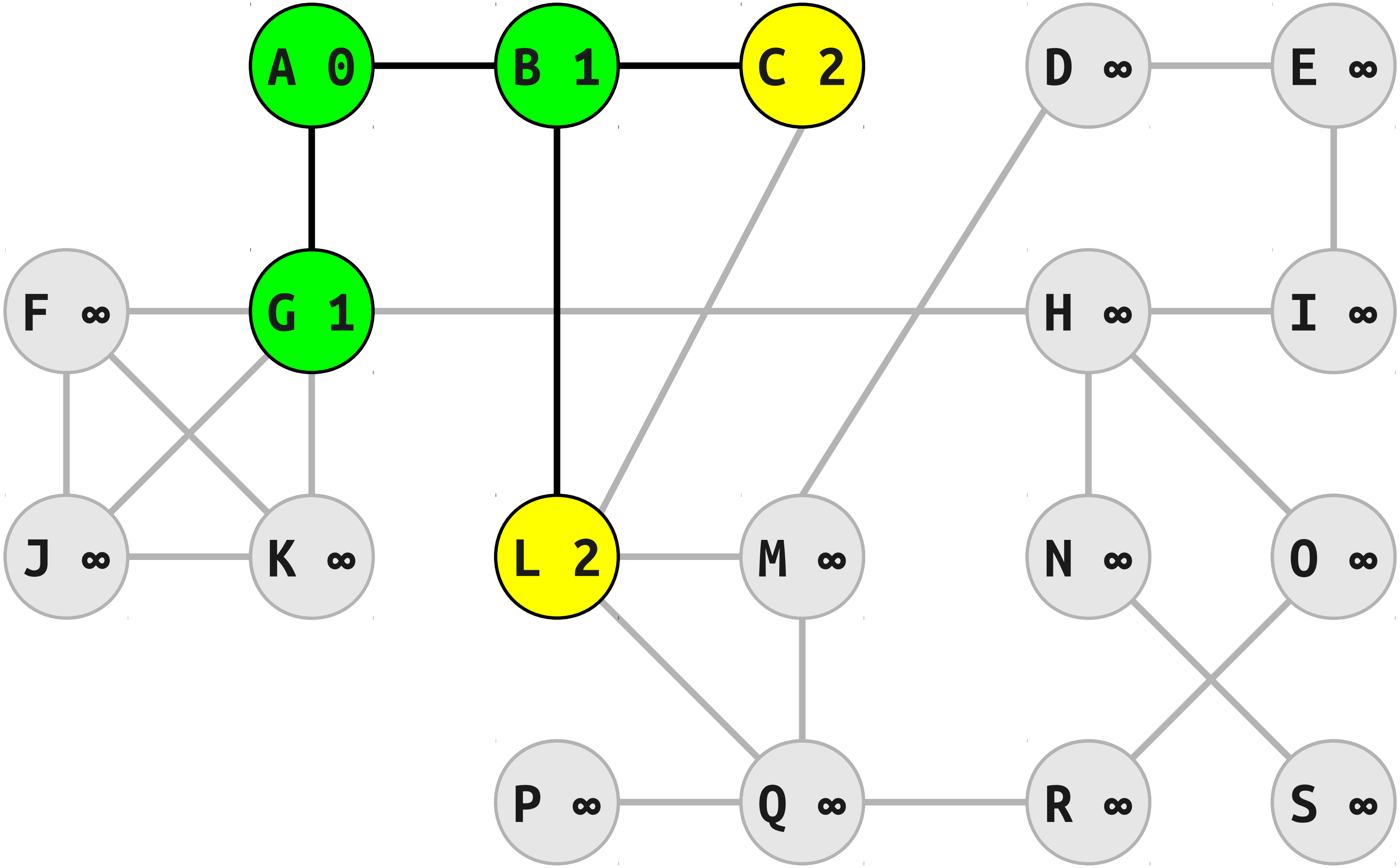
Breadth-First Search

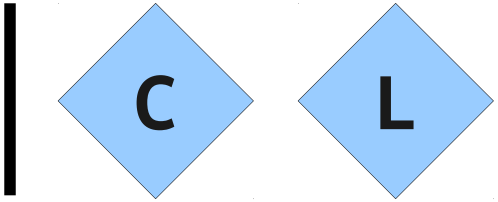


Breadth-First Search

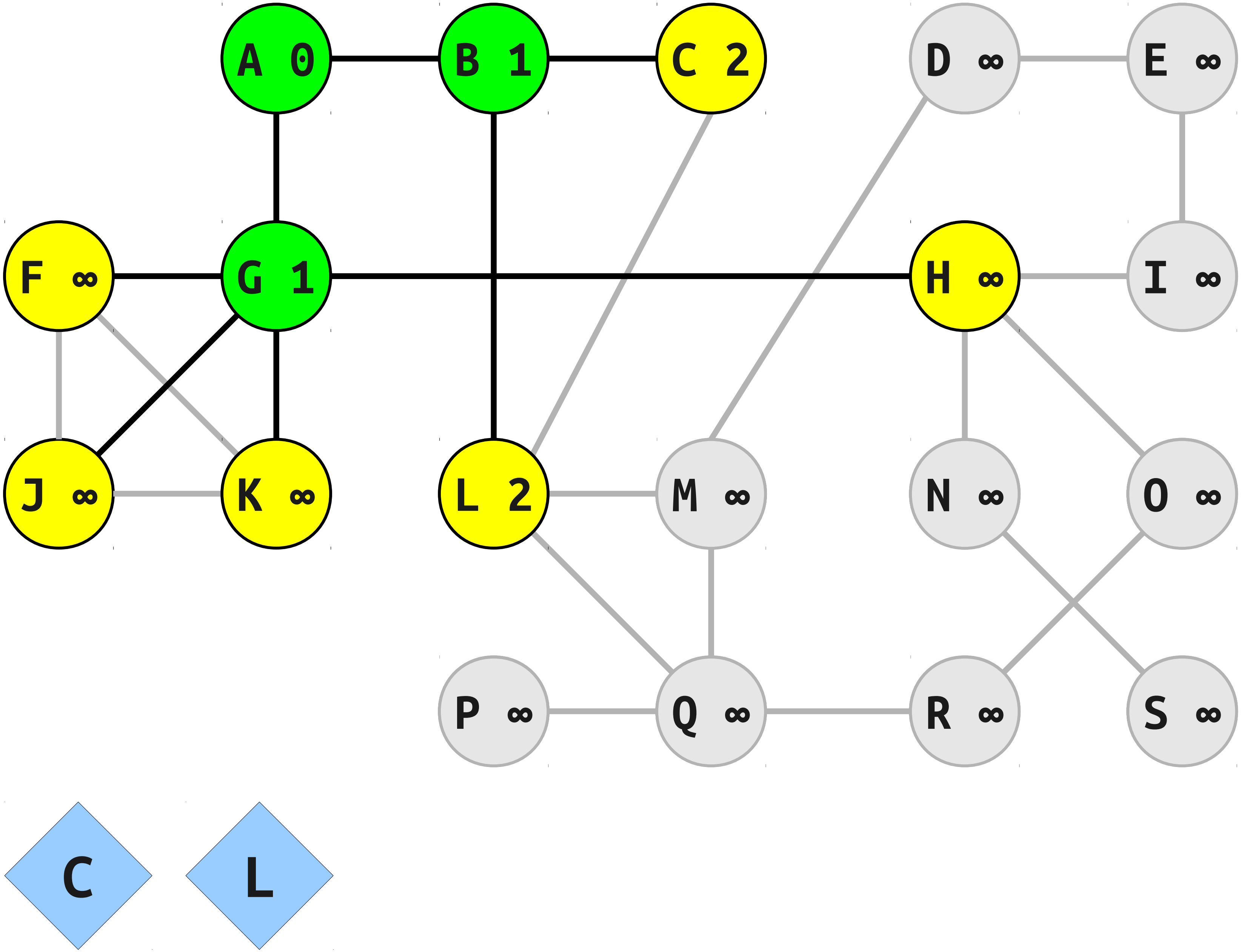


Breadth-First Search

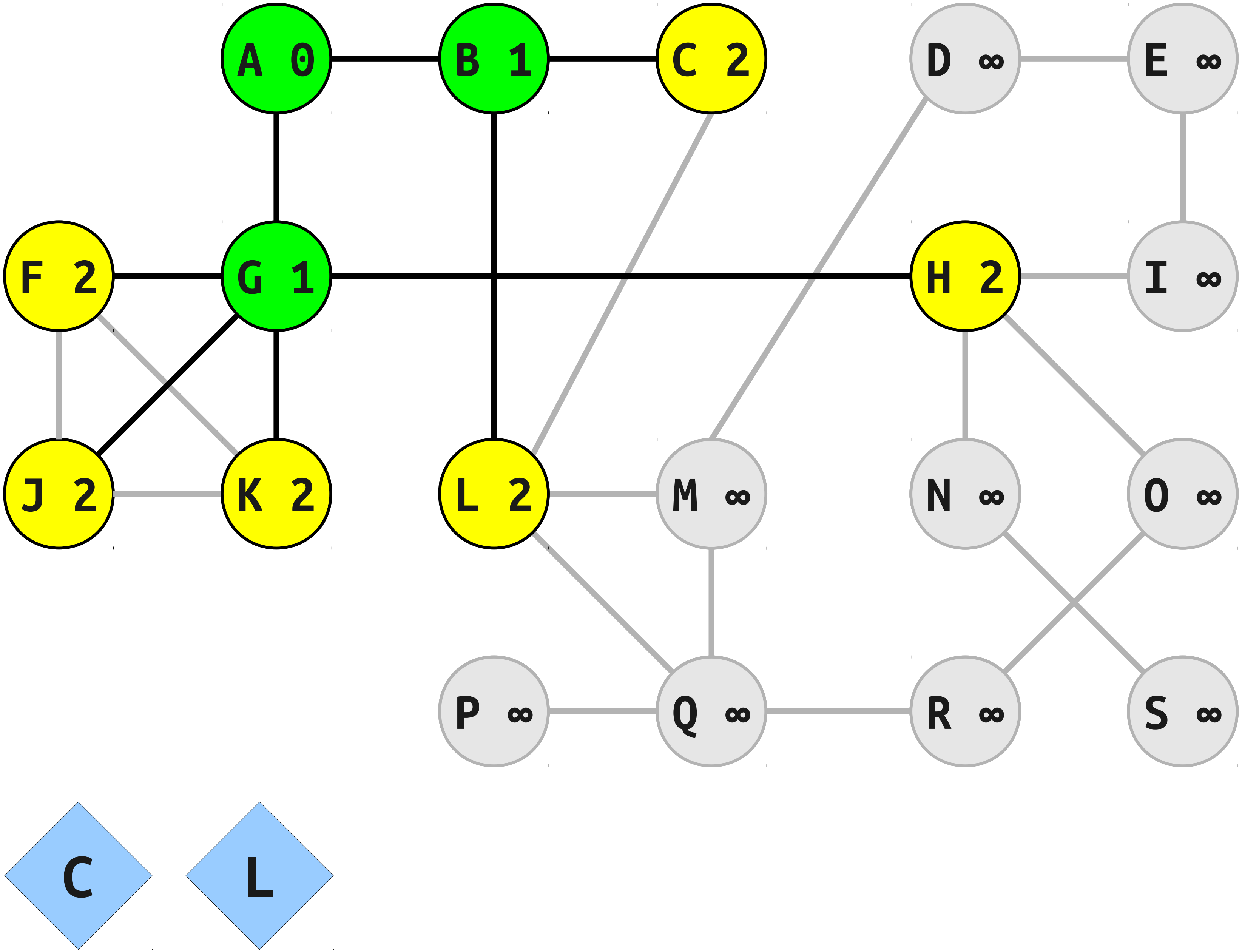




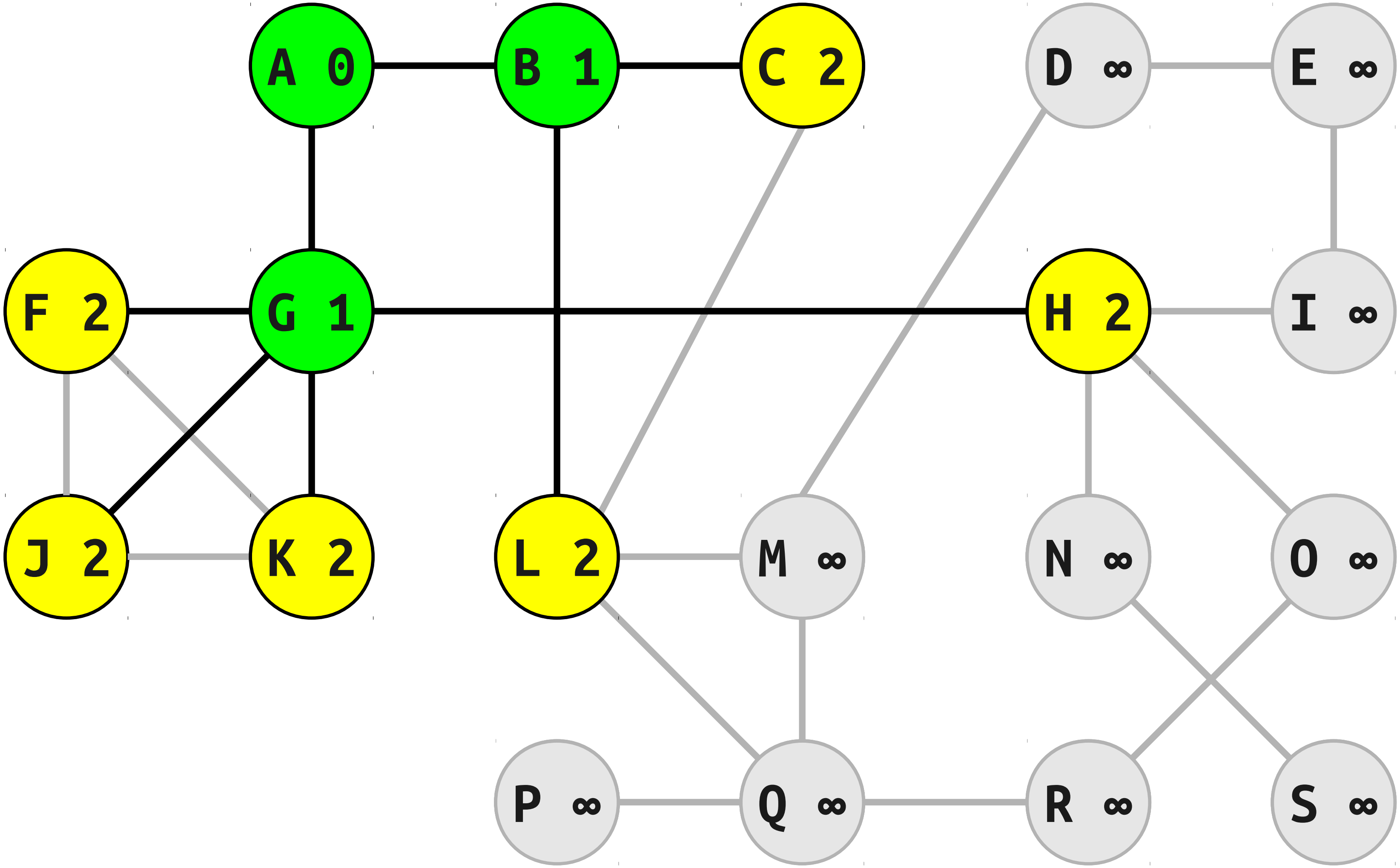
Breadth-First Search

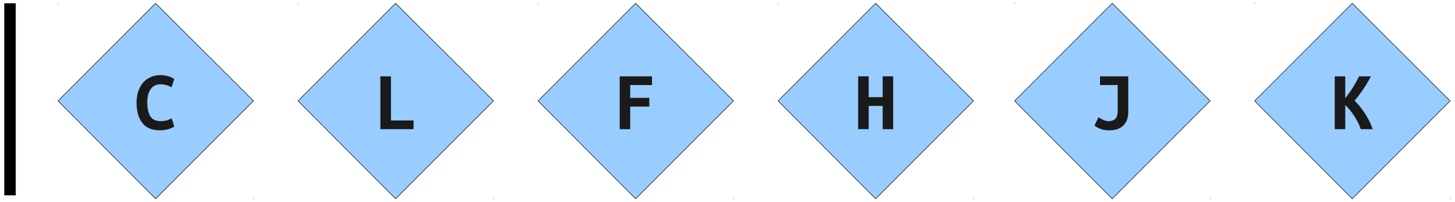


Breadth-First Search

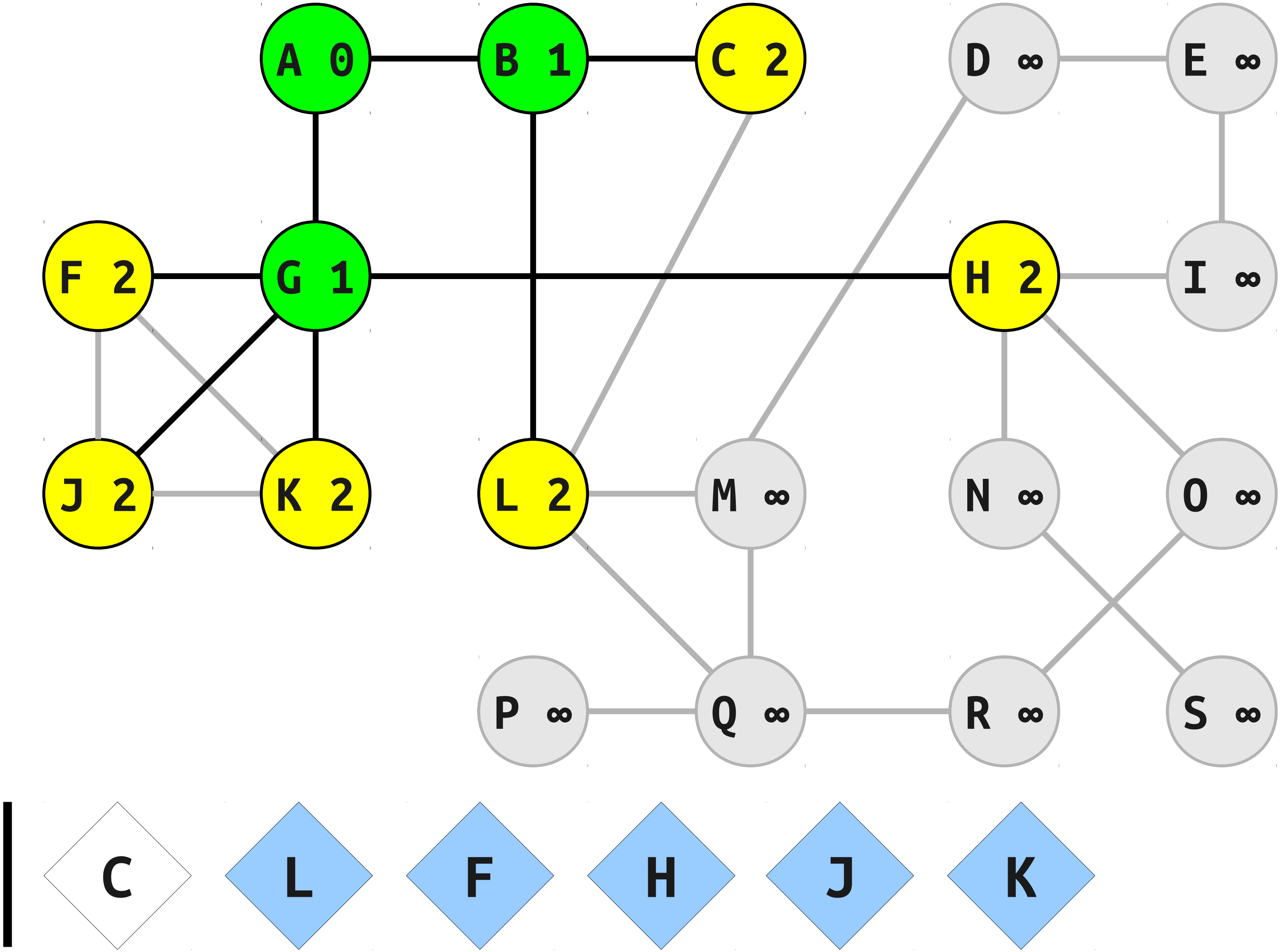


Breadth-First Search

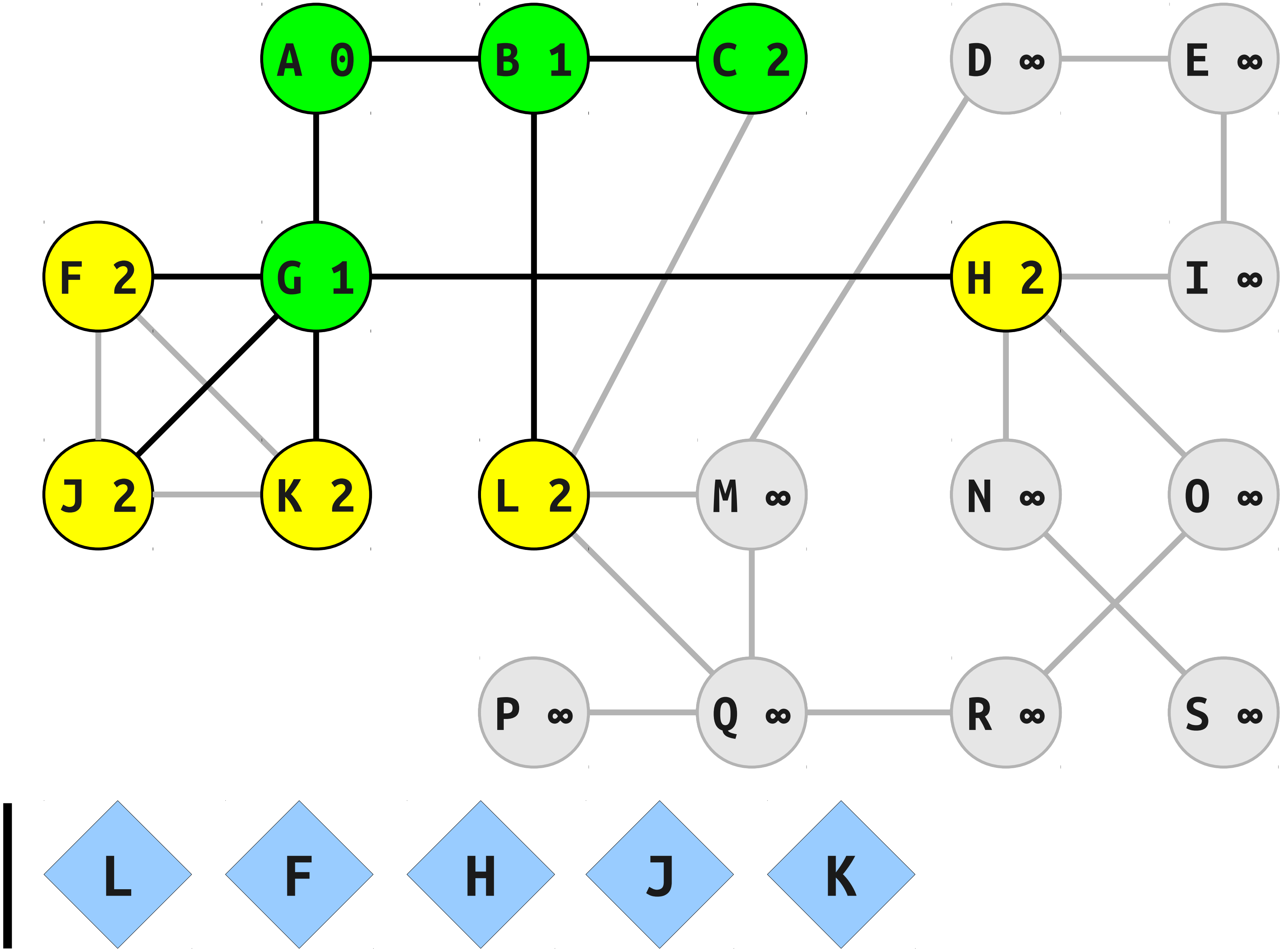




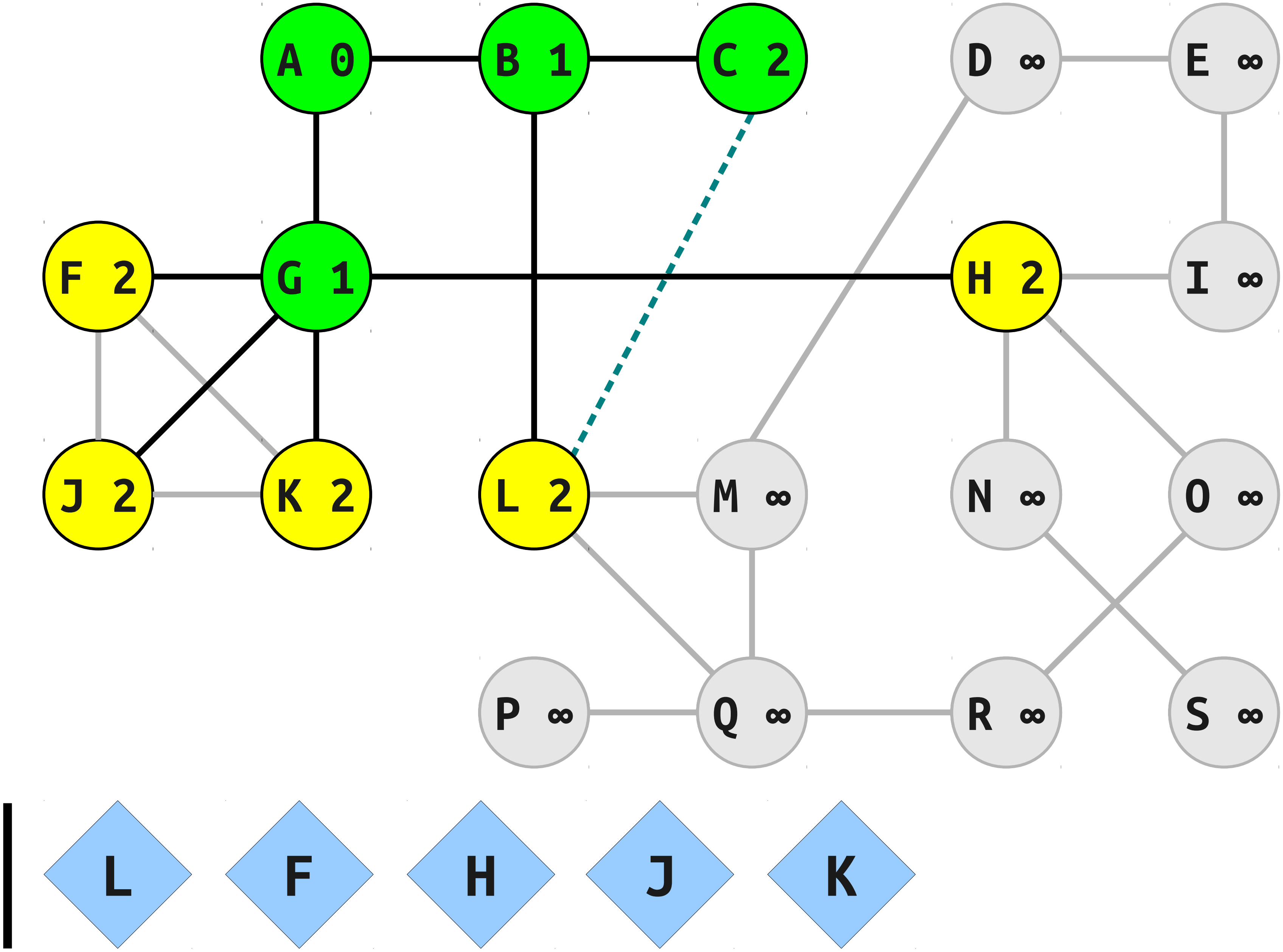
Breadth-First Search



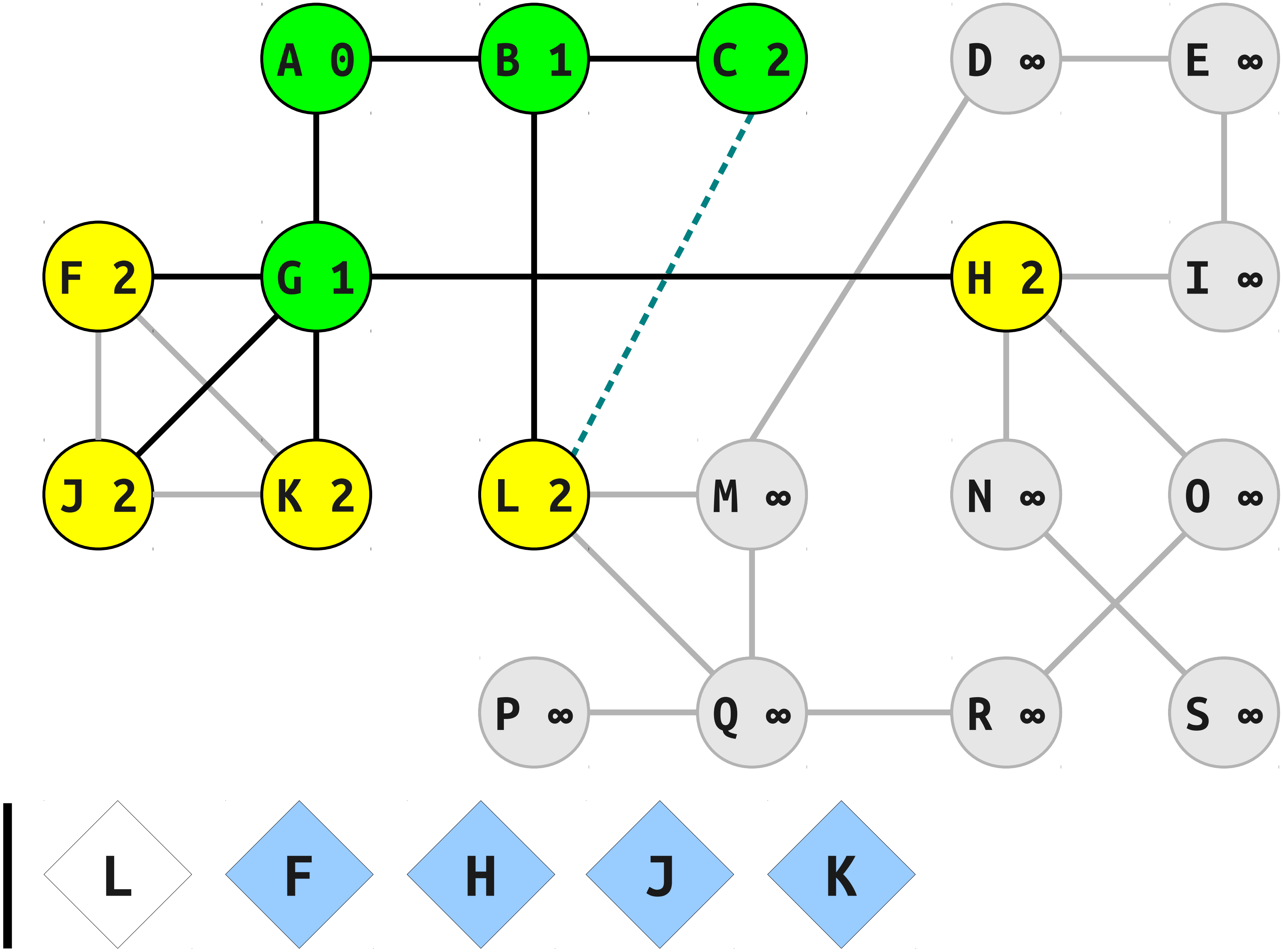
Breadth-First Search



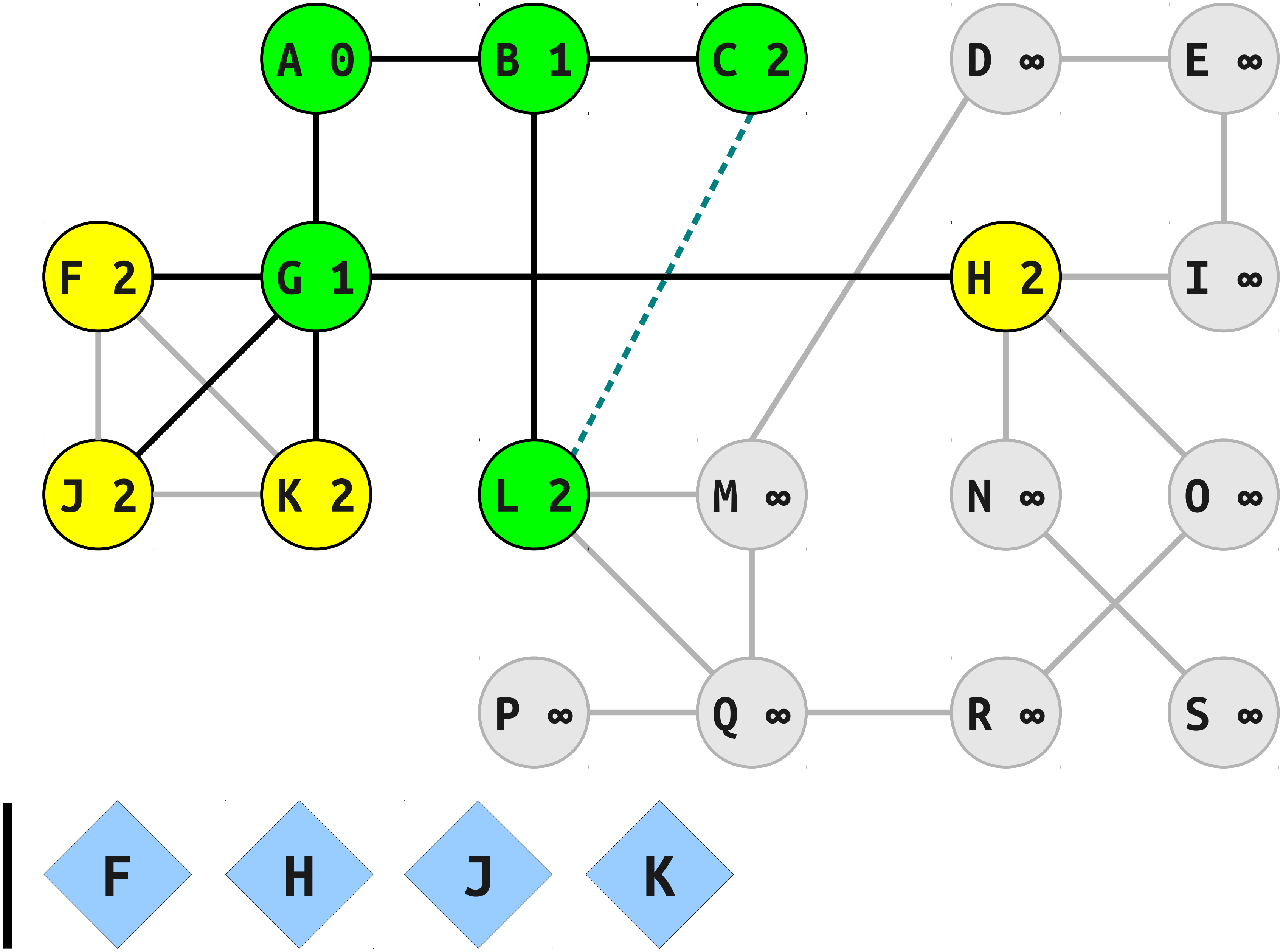
Breadth-First Search



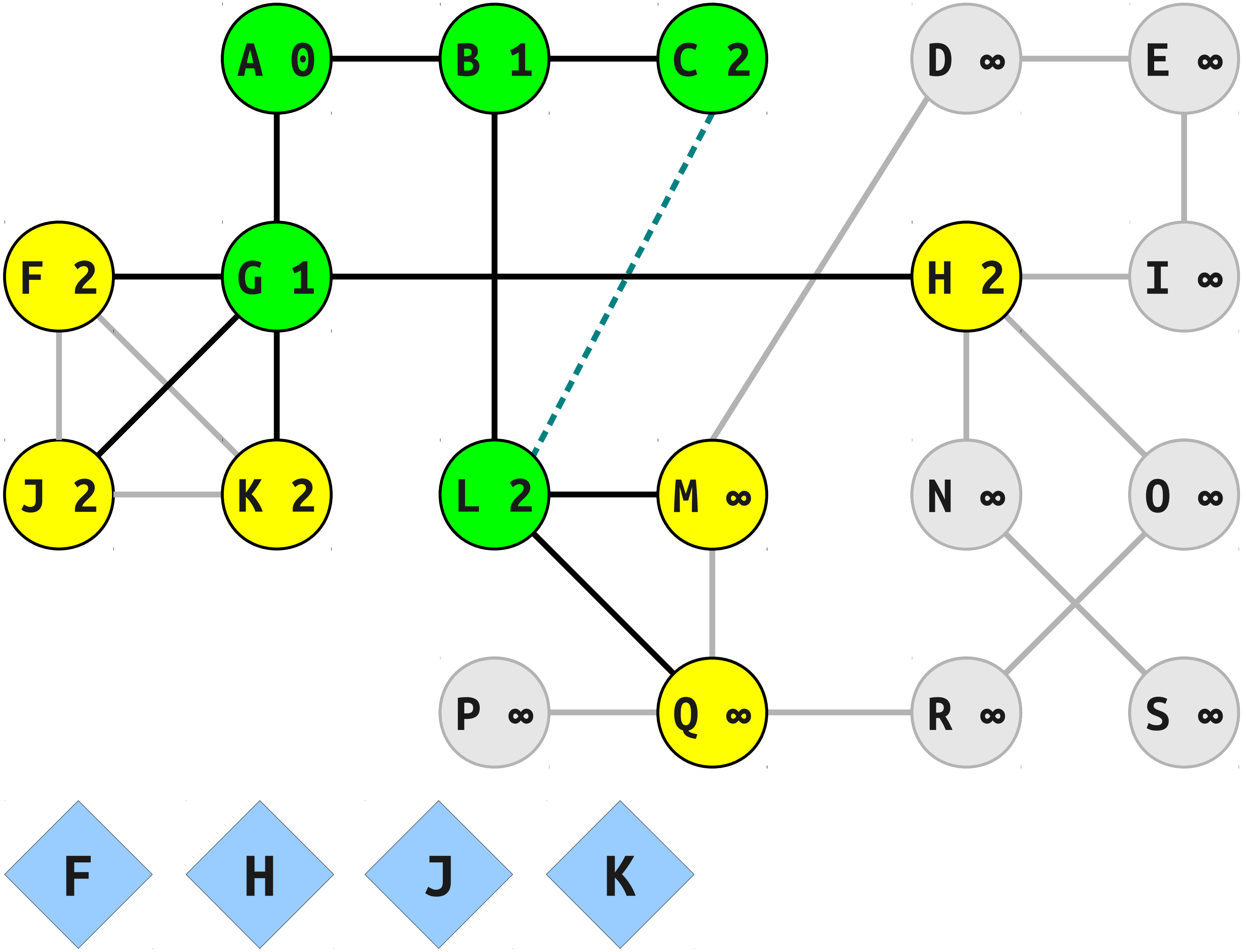
Breadth-First Search



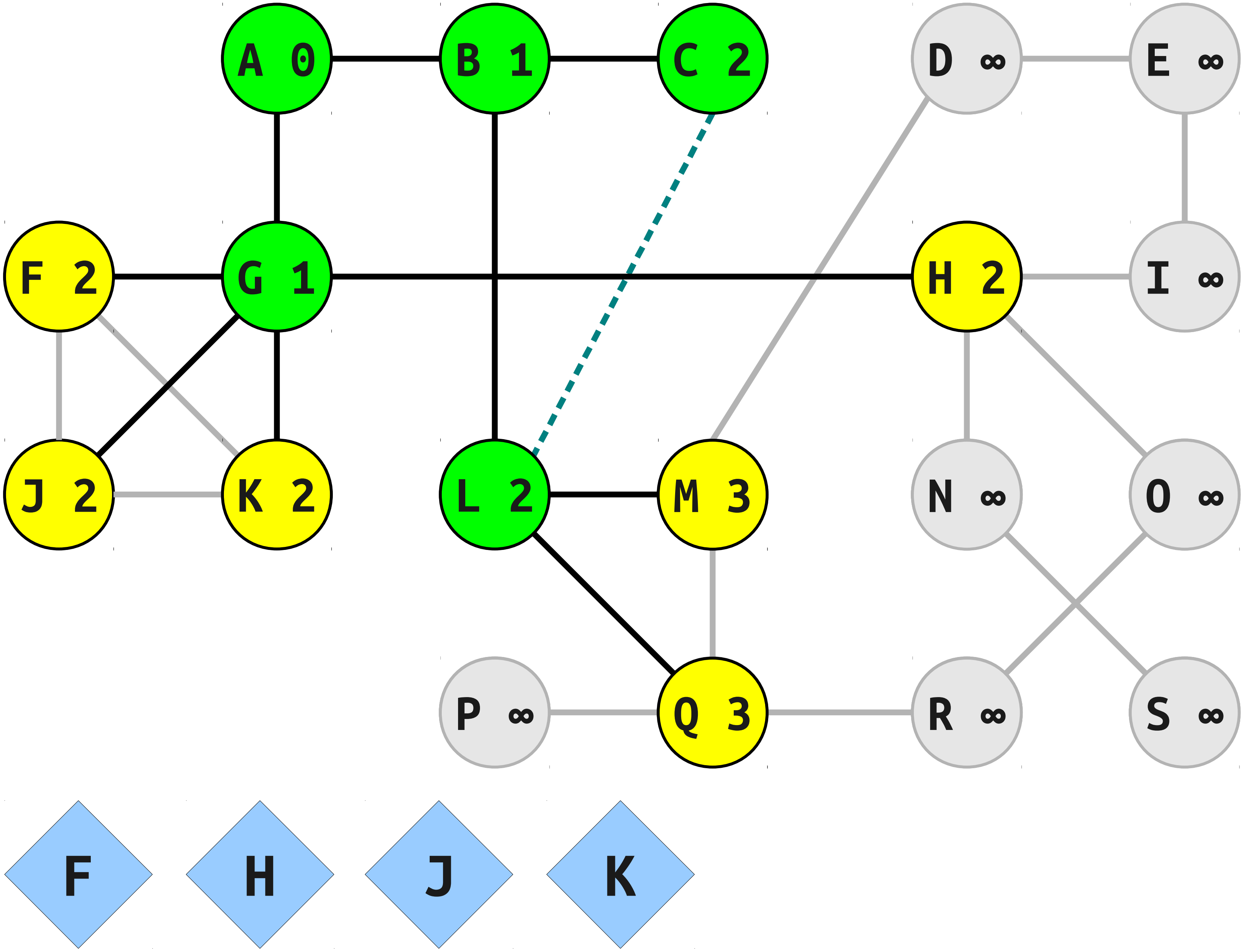
Breadth-First Search



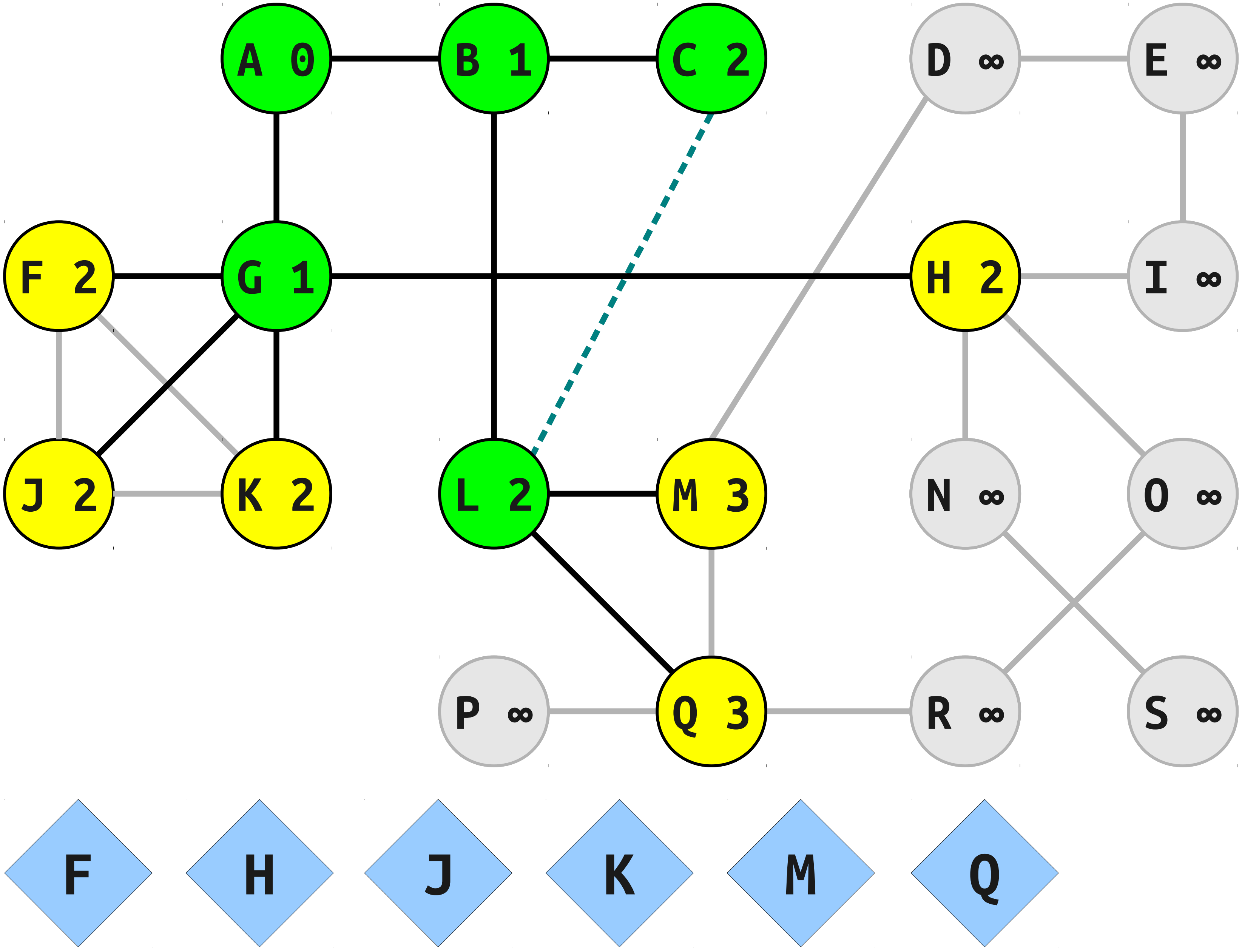
Breadth-First Search



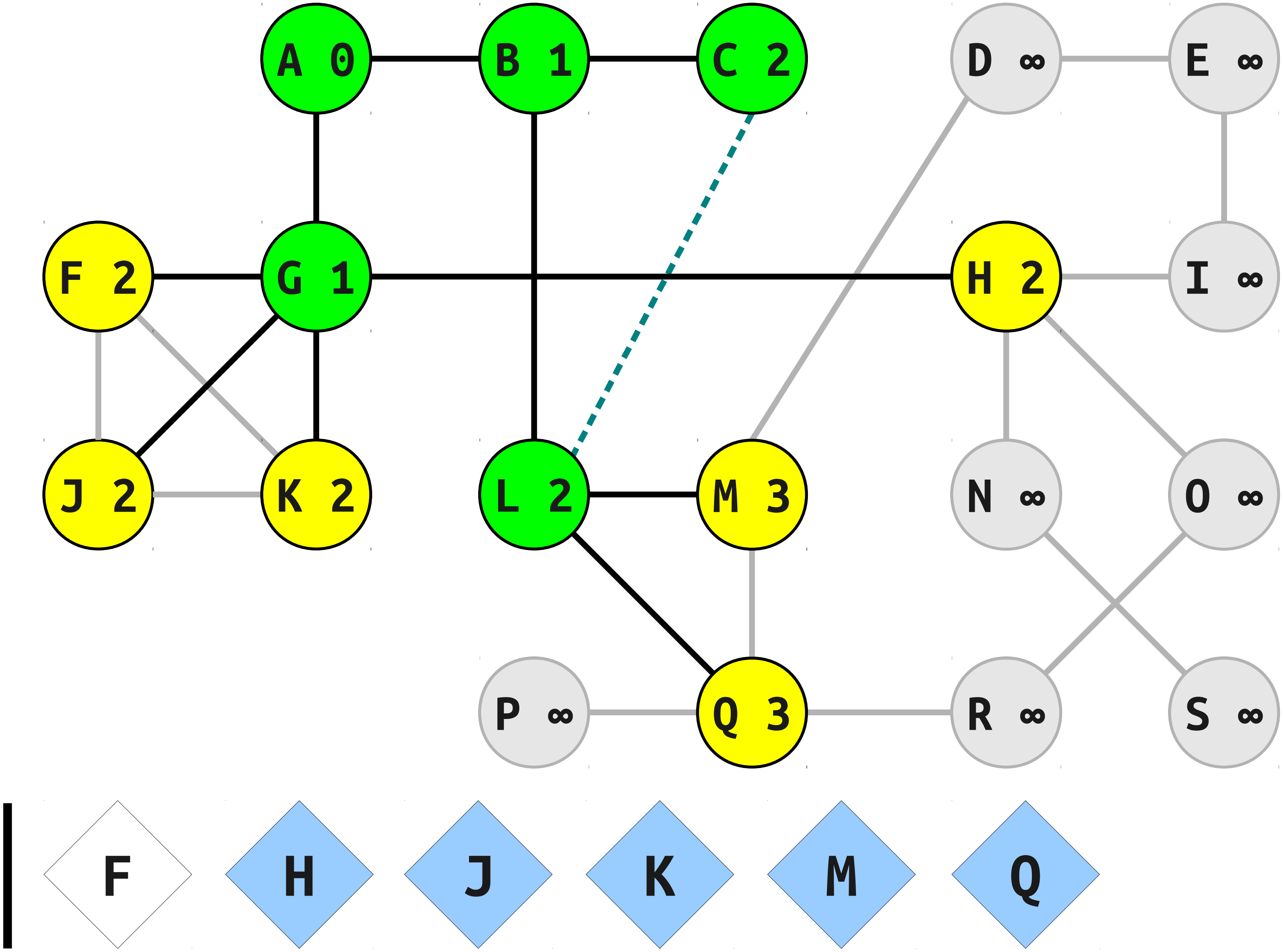
Breadth-First Search



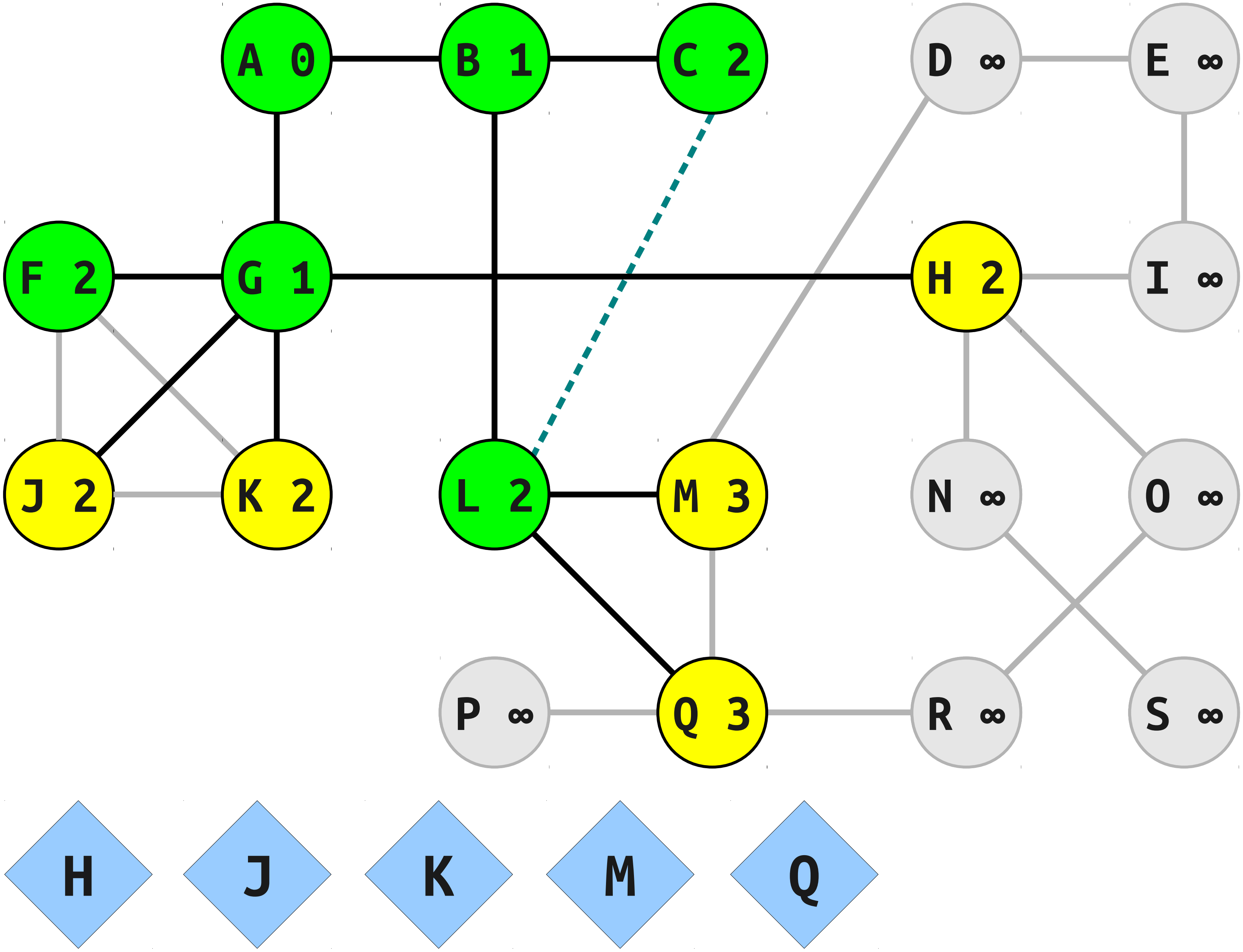
Breadth-First Search



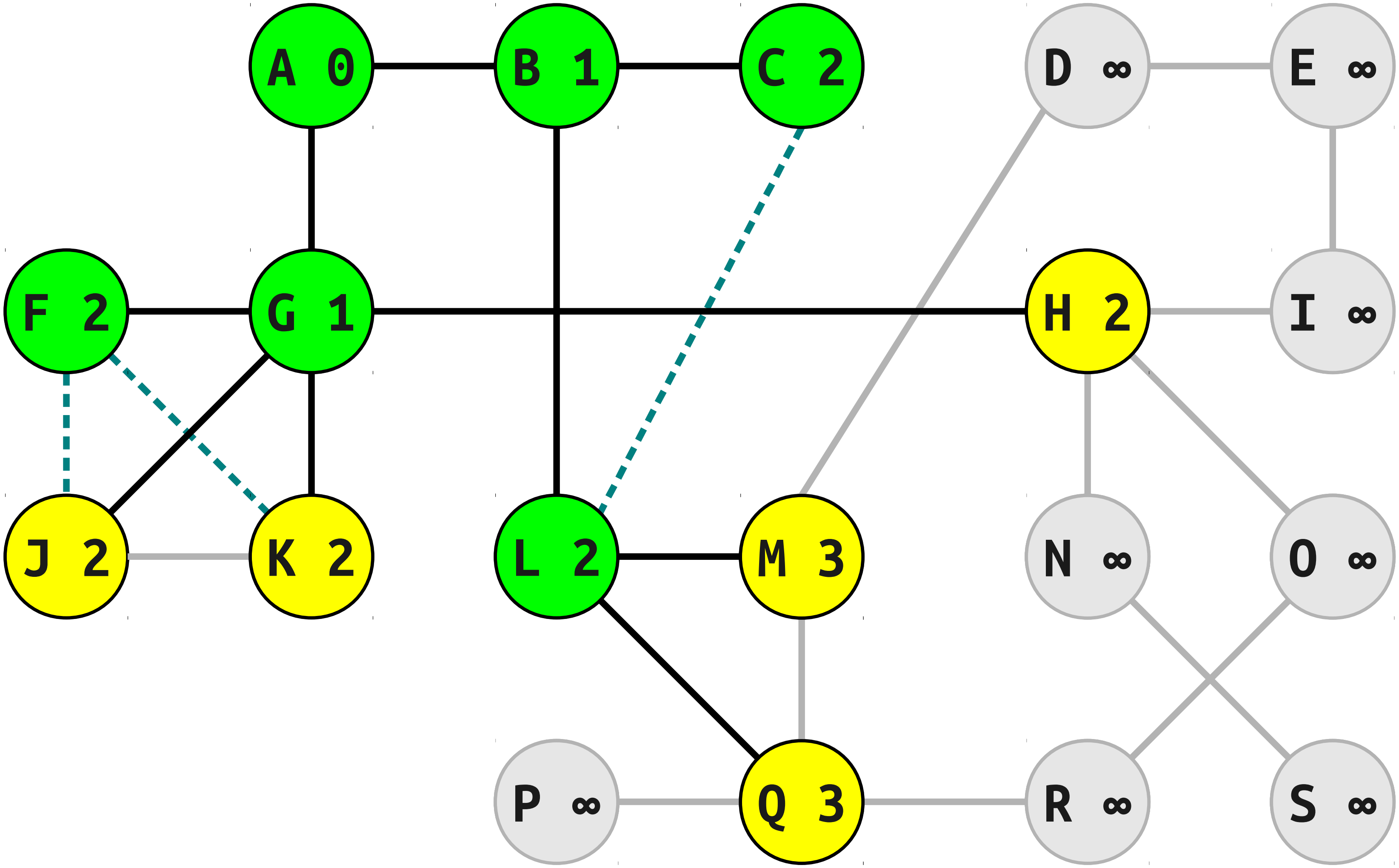
Breadth-First Search

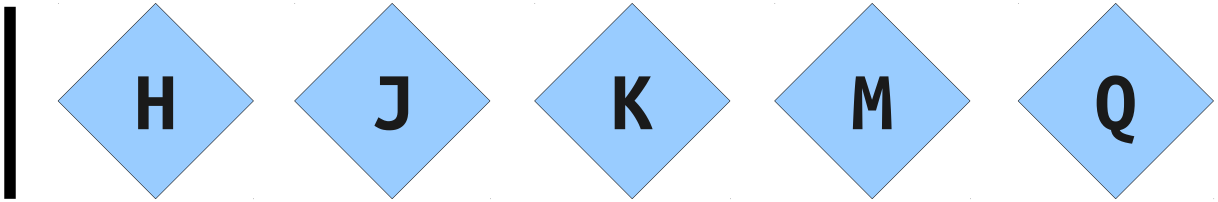


Breadth-First Search

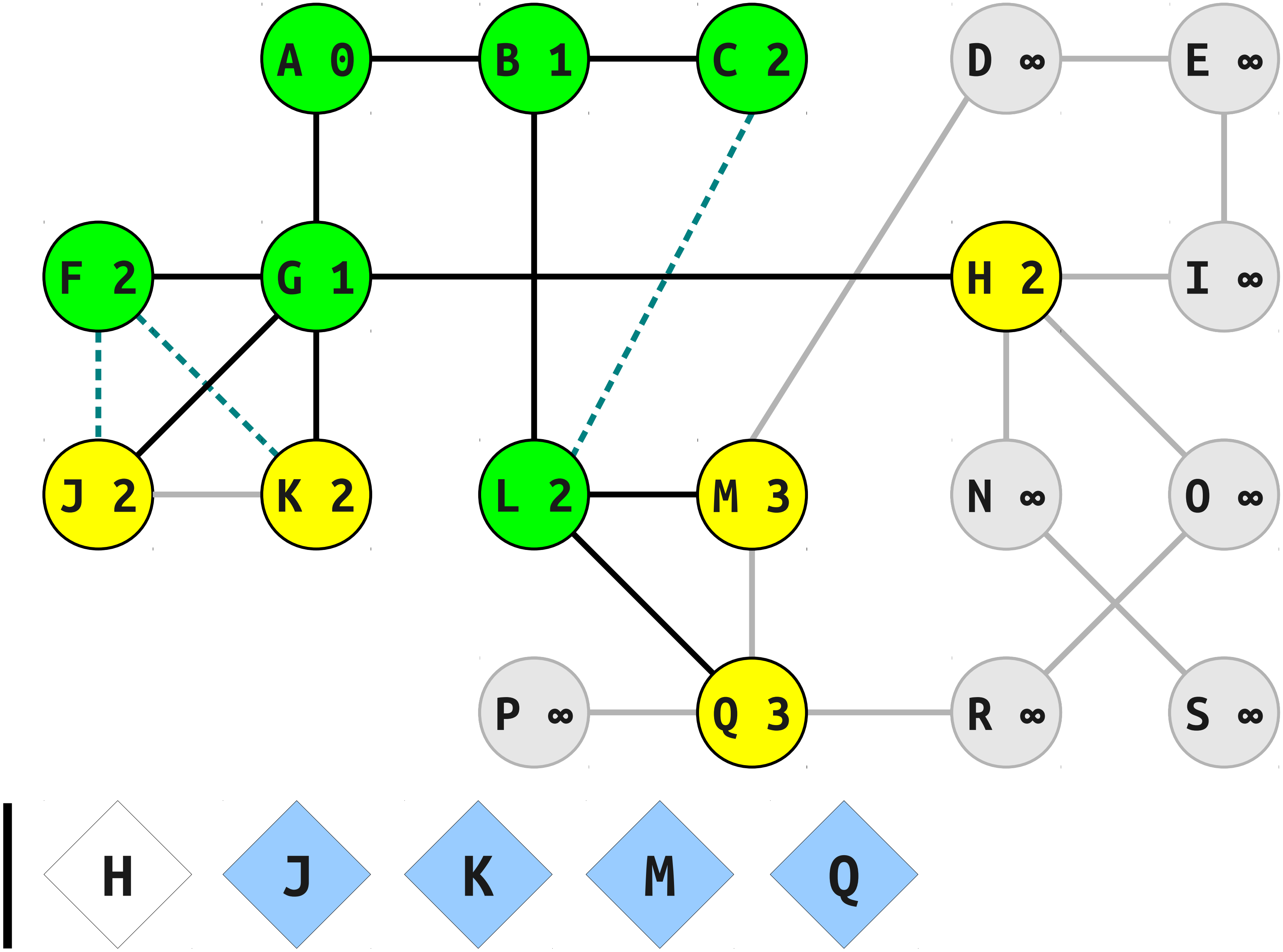


Breadth-First Search

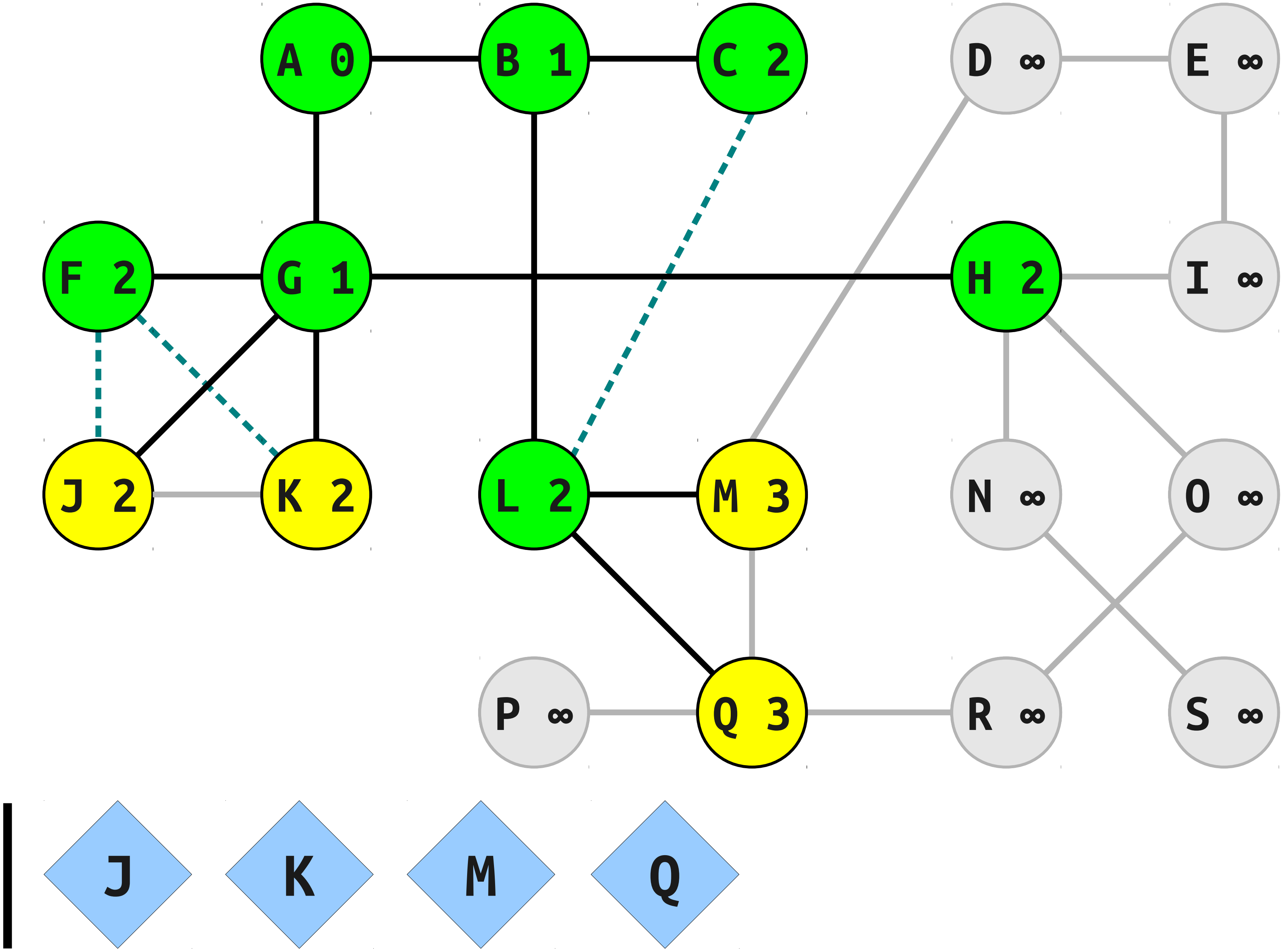




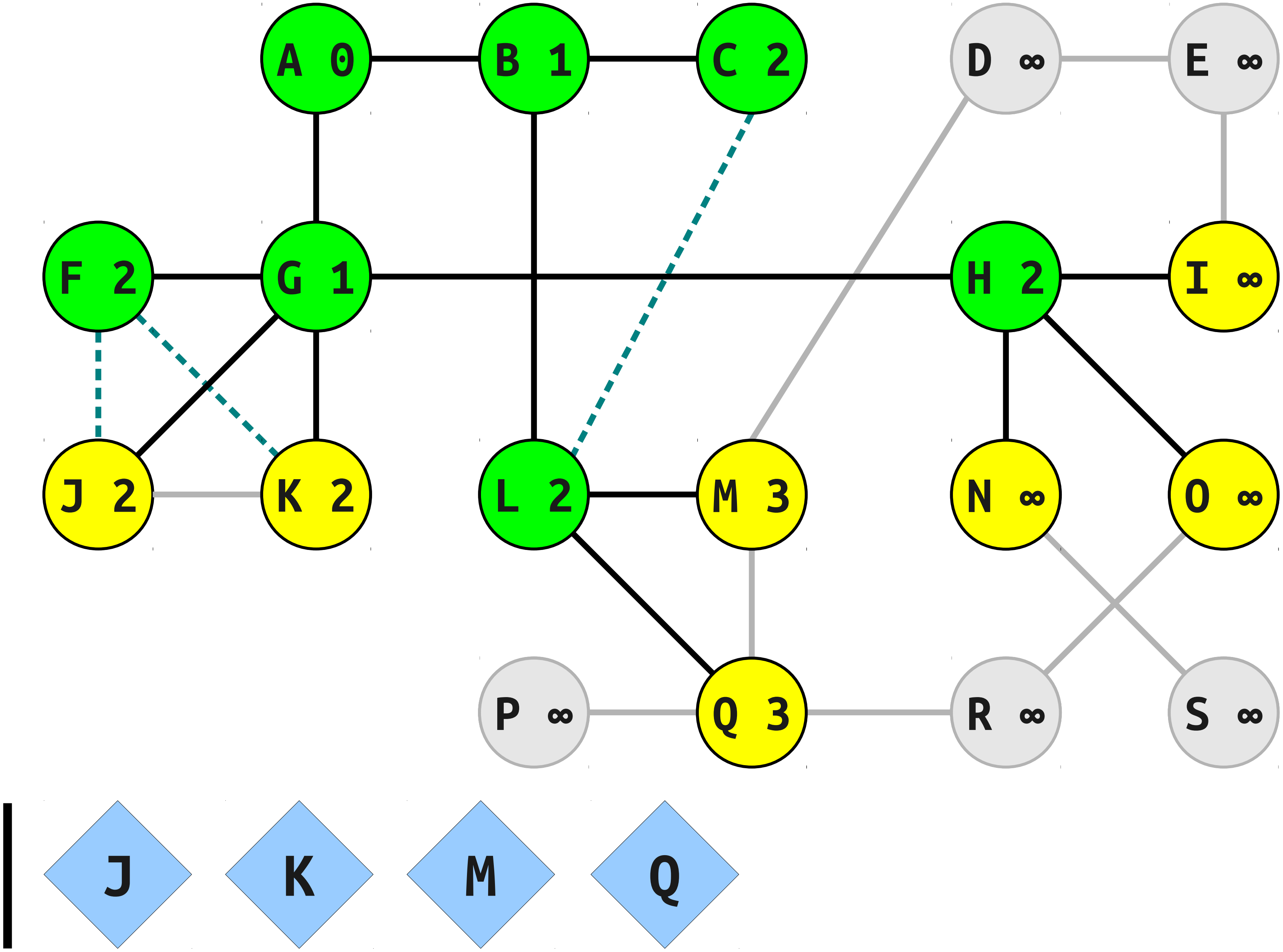
Breadth-First Search



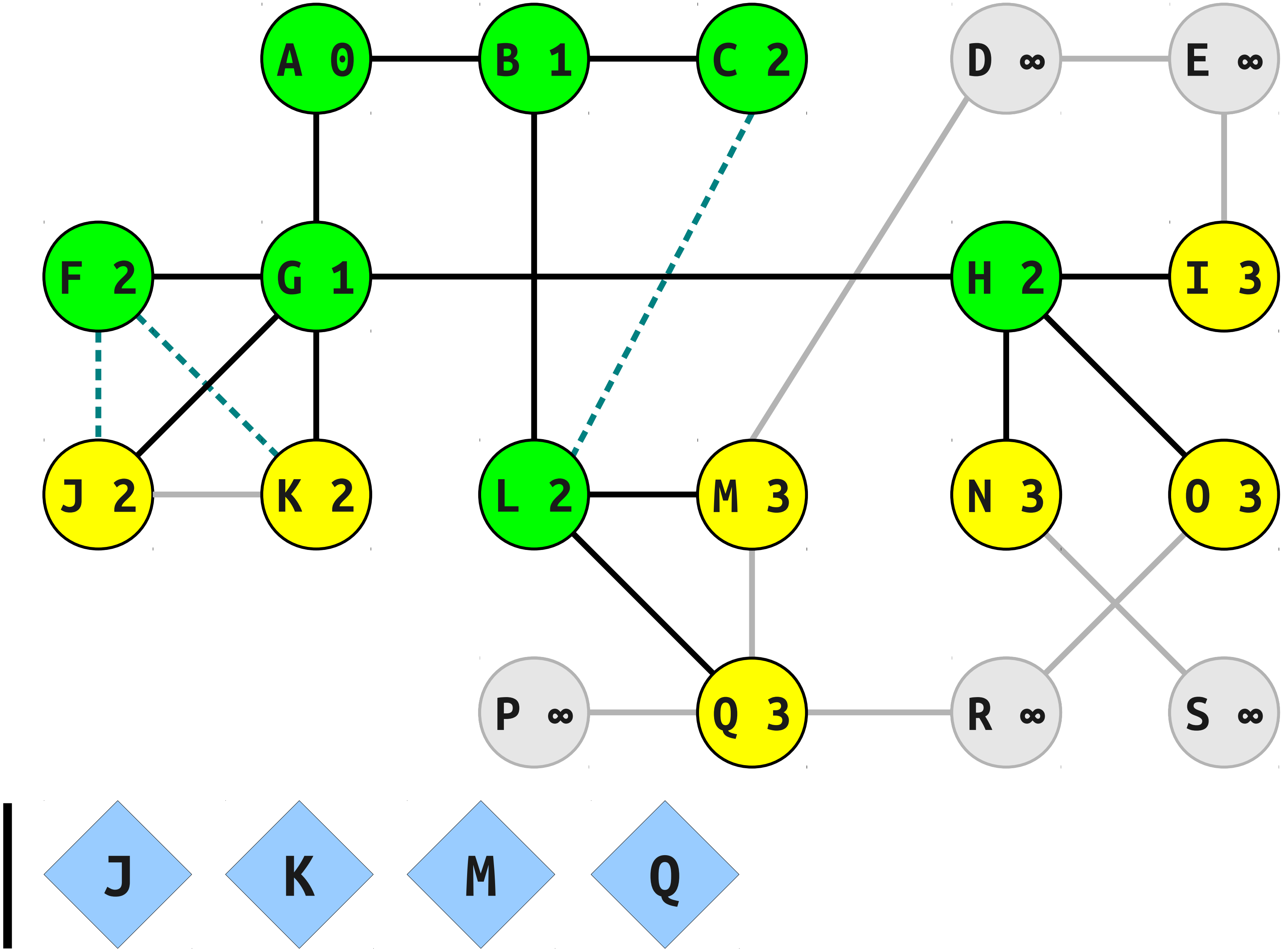
Breadth-First Search



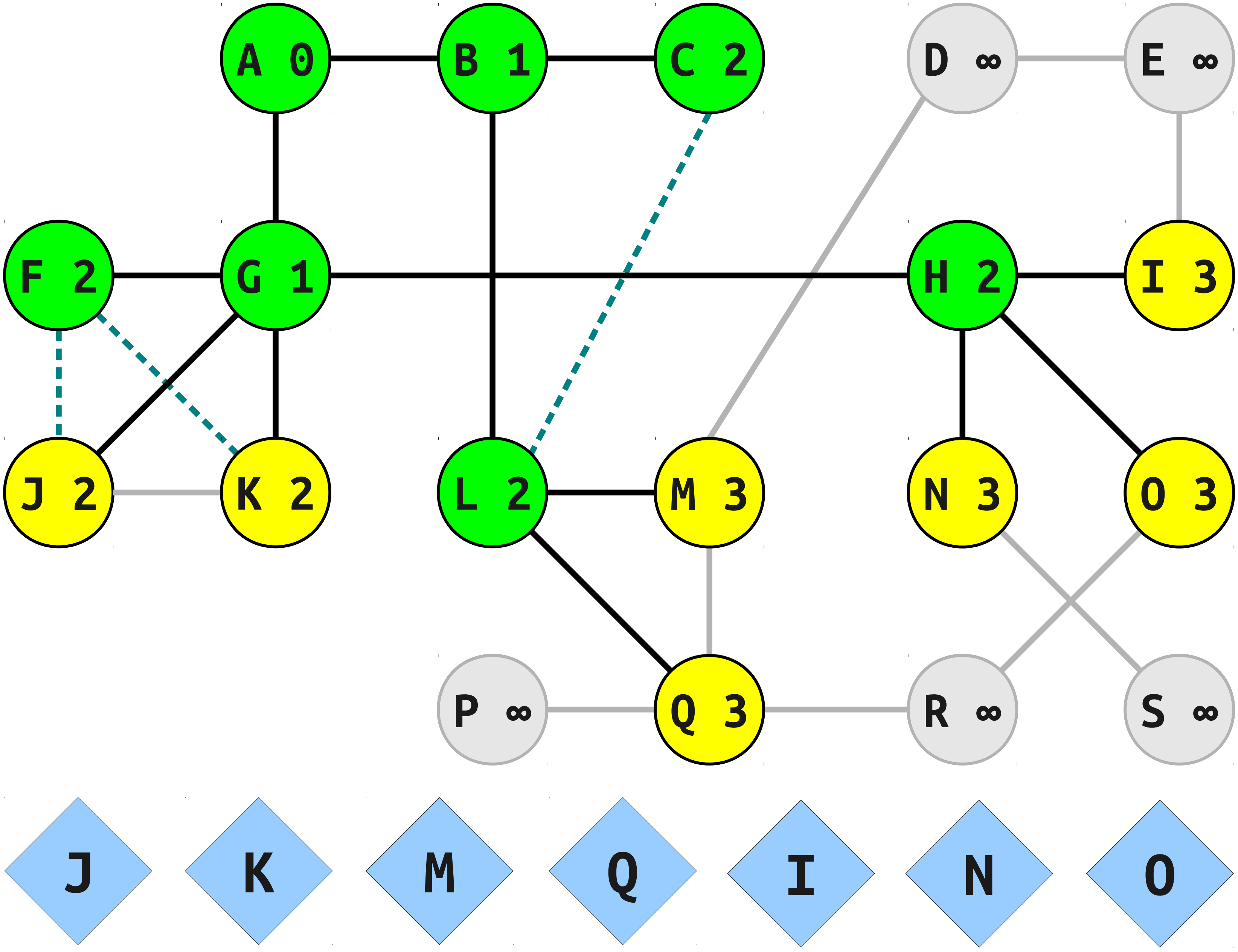
Breadth-First Search



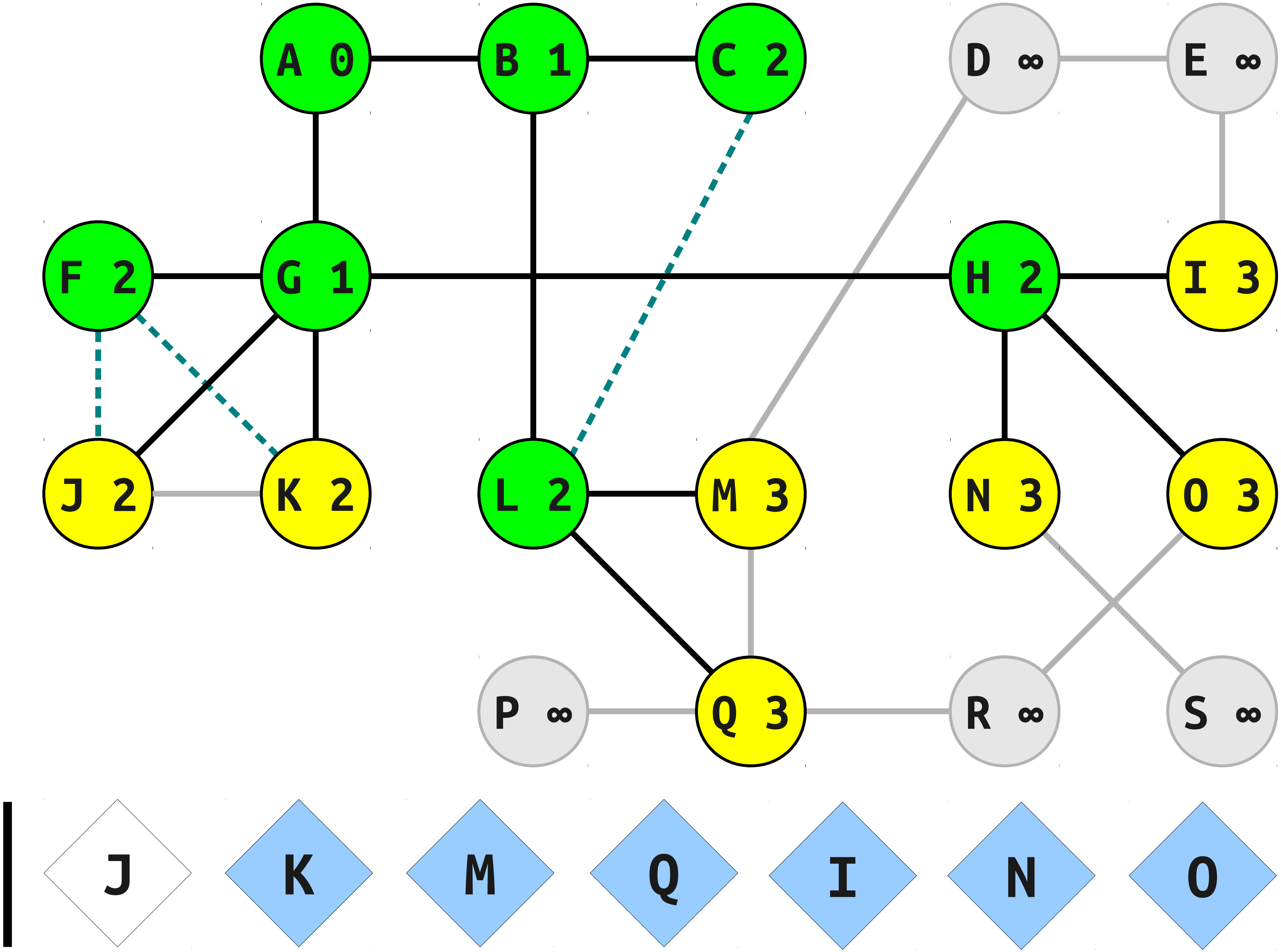
Breadth-First Search



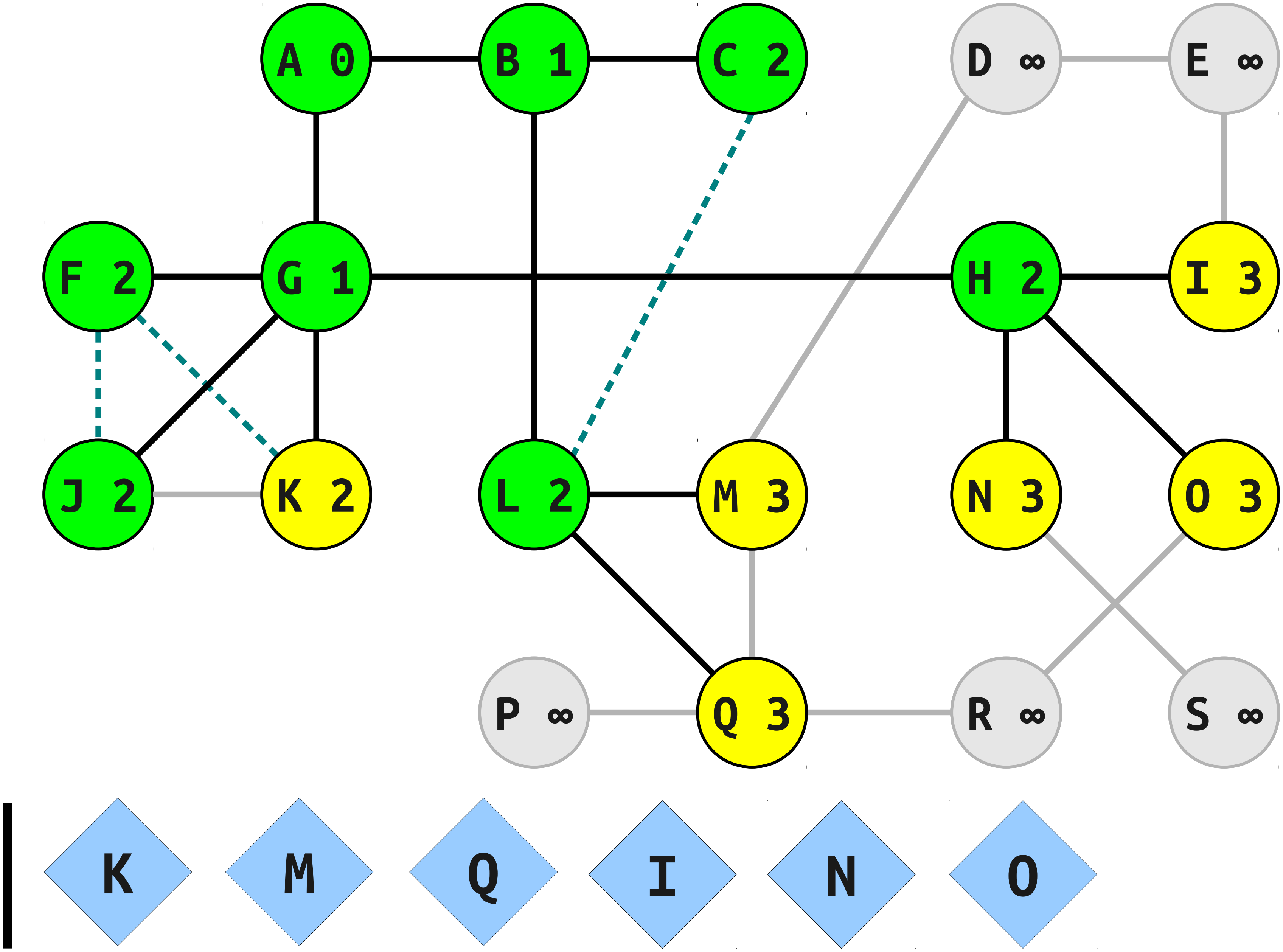
Breadth-First Search



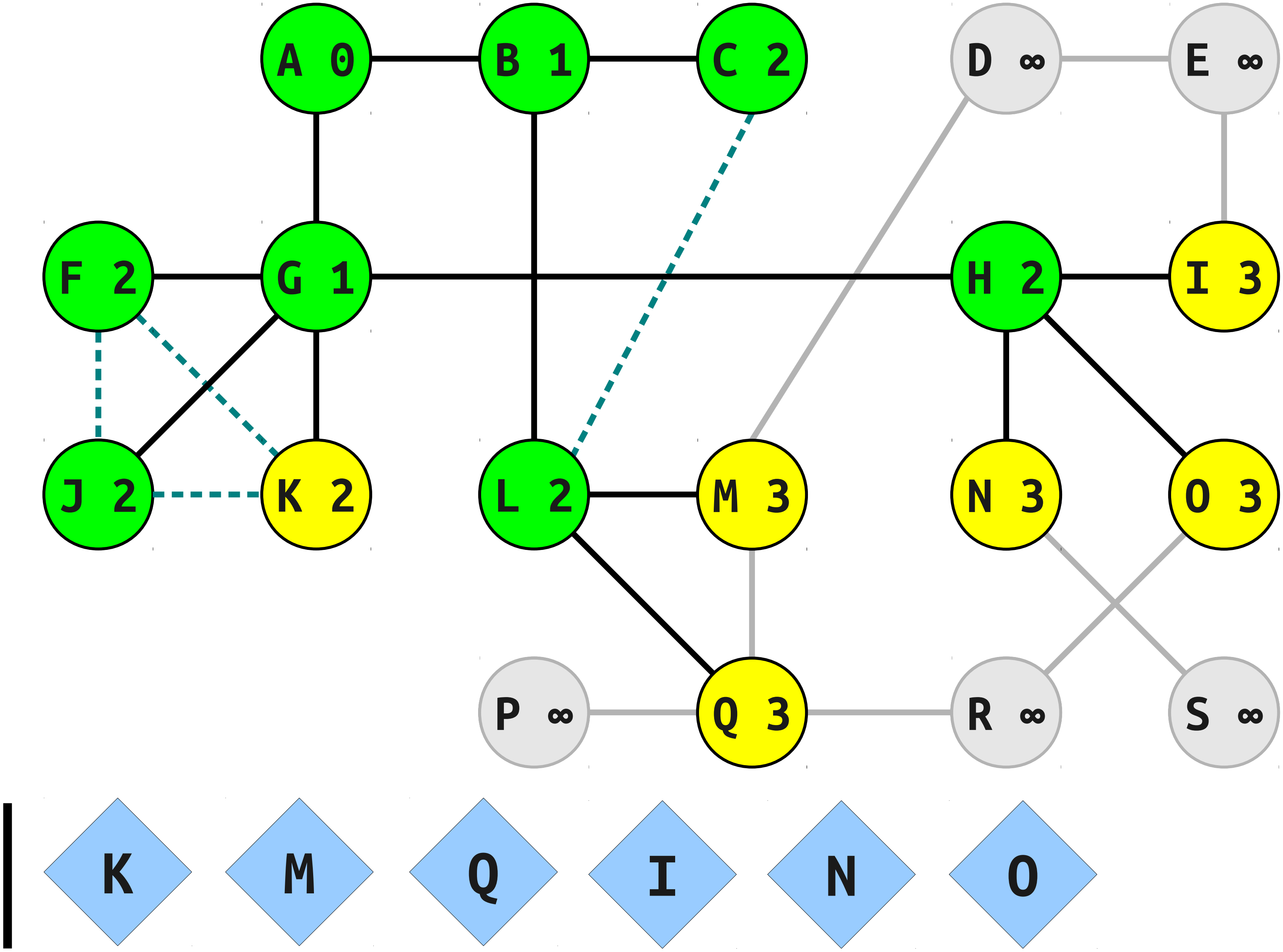
Breadth-First Search



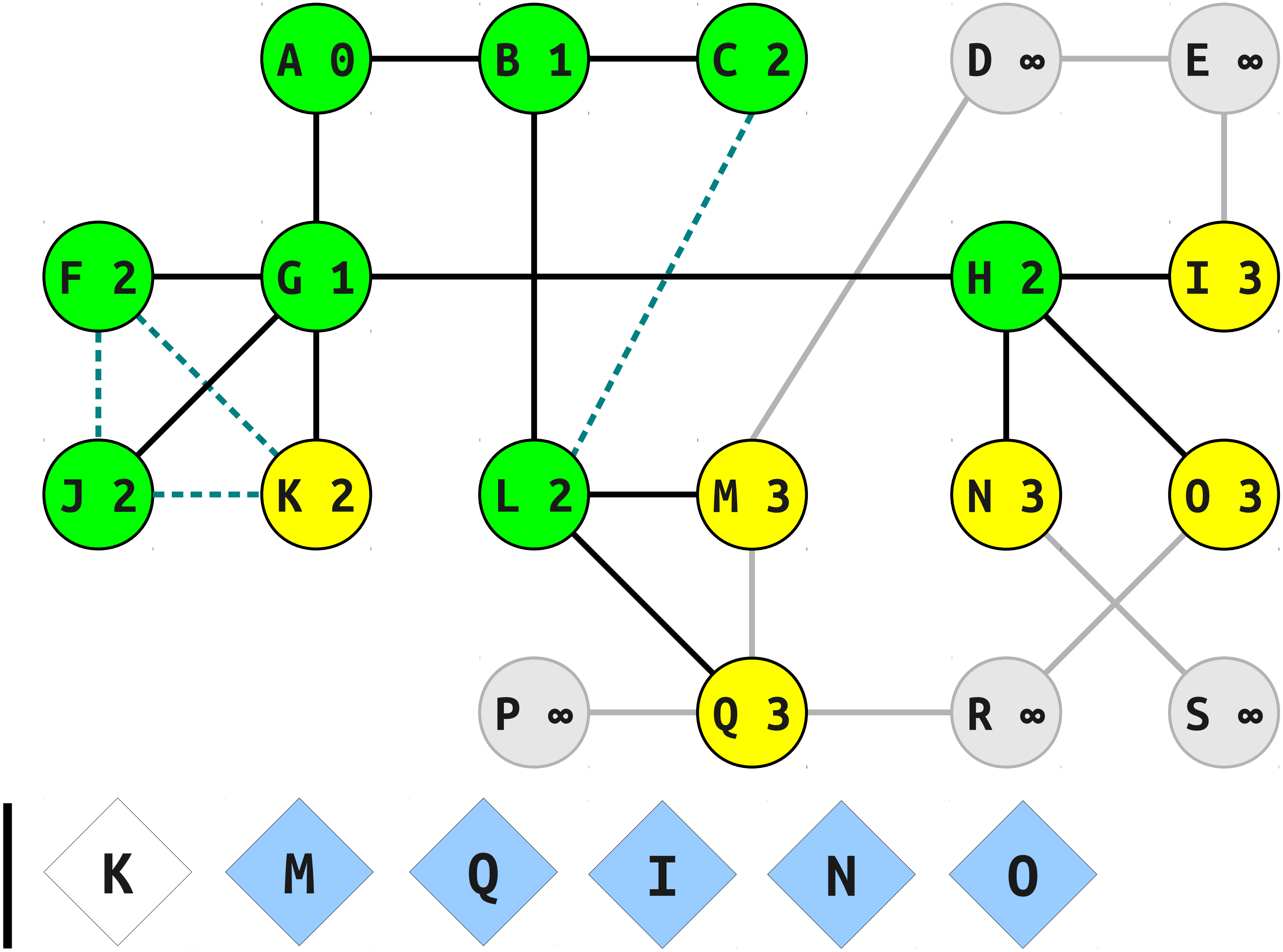
Breadth-First Search



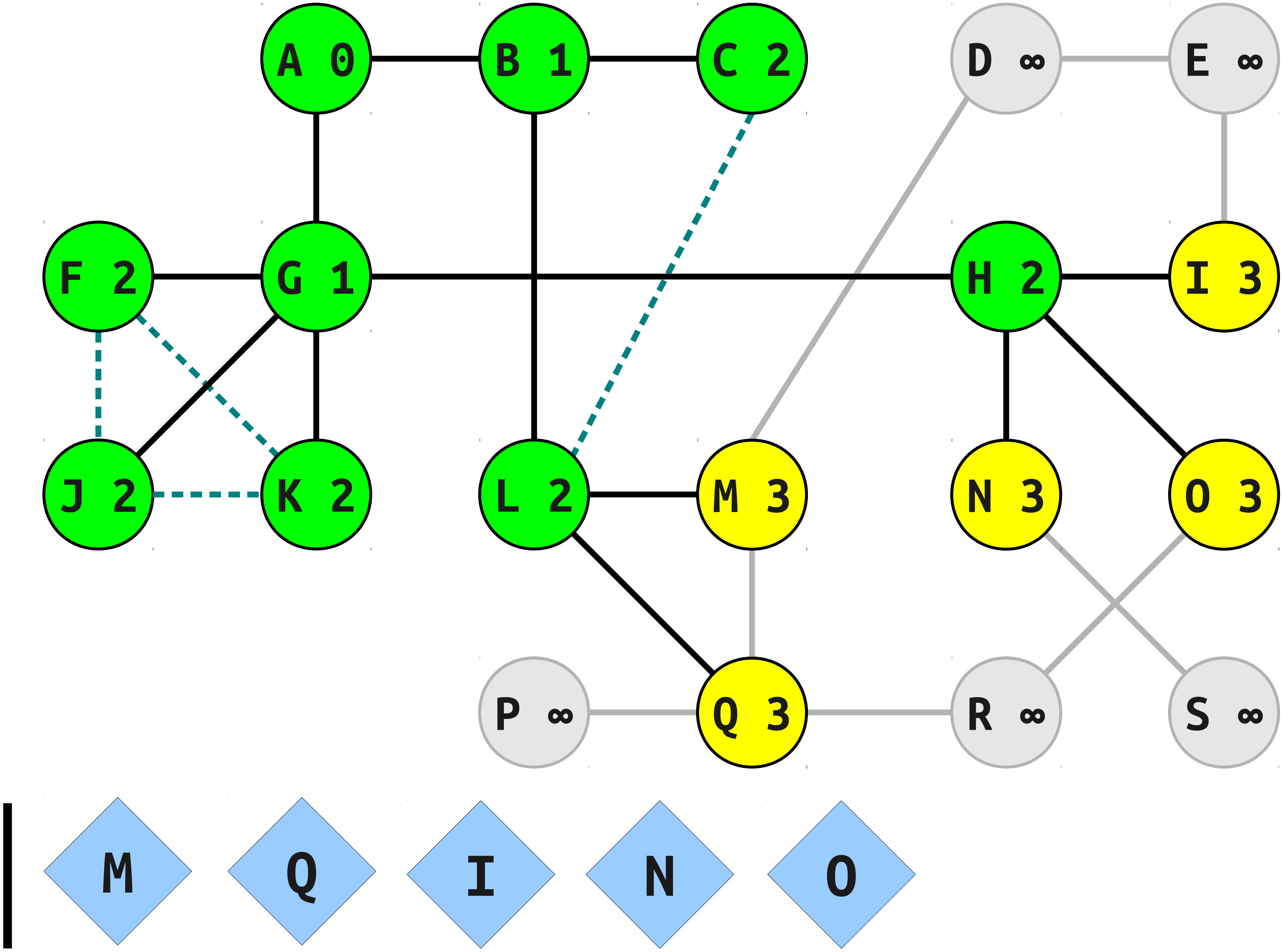
Breadth-First Search



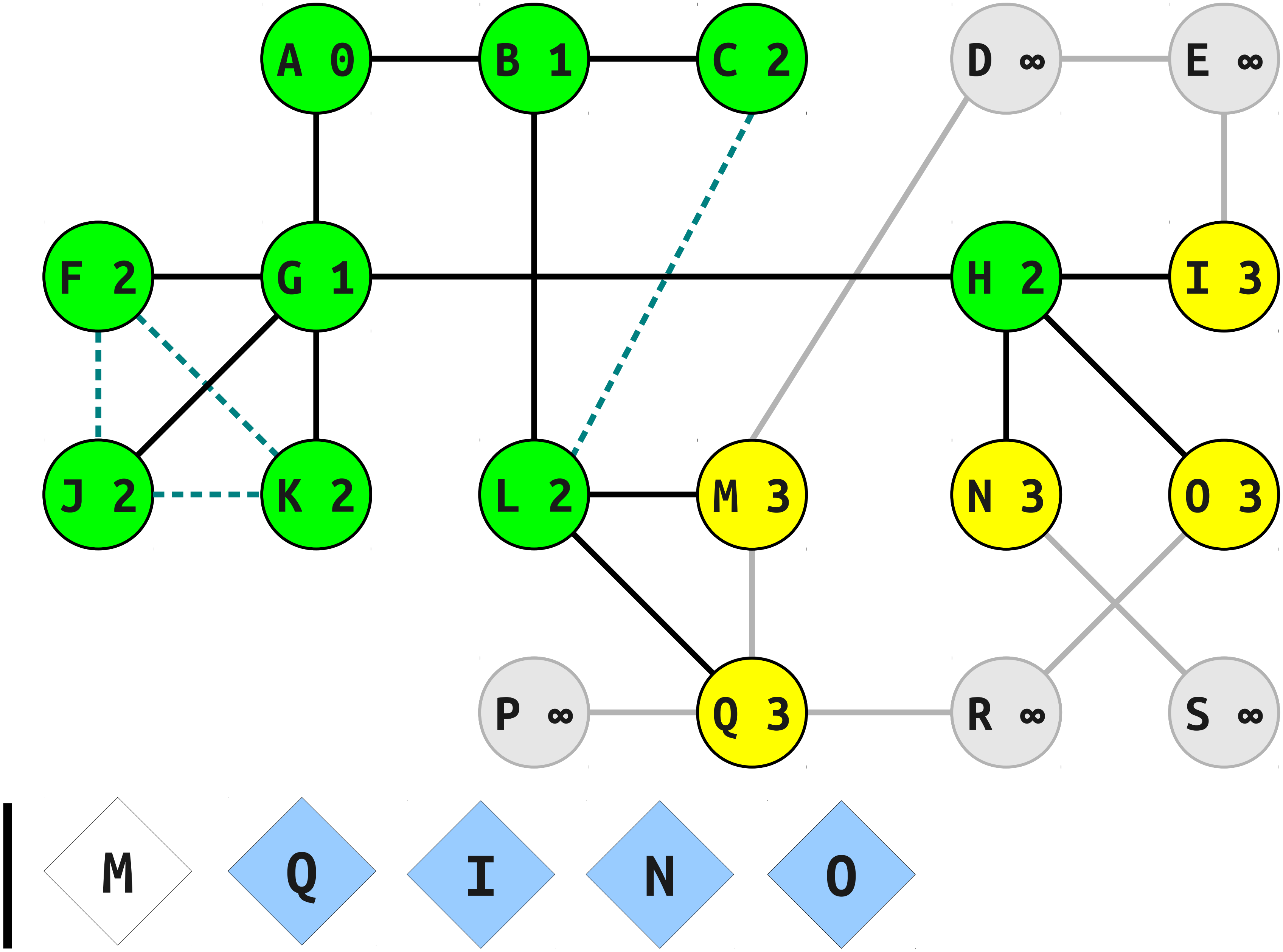
Breadth-First Search



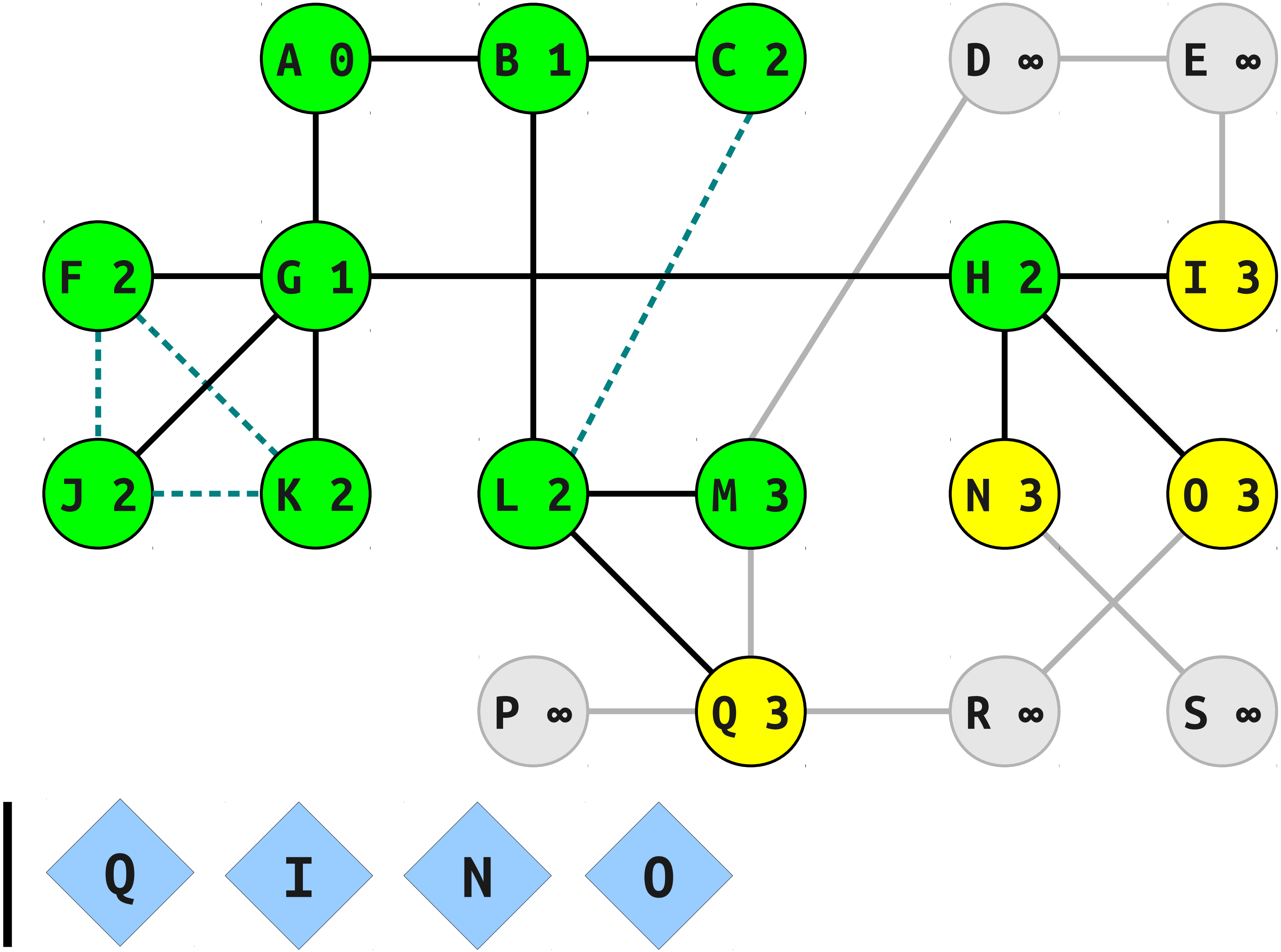
Breadth-First Search



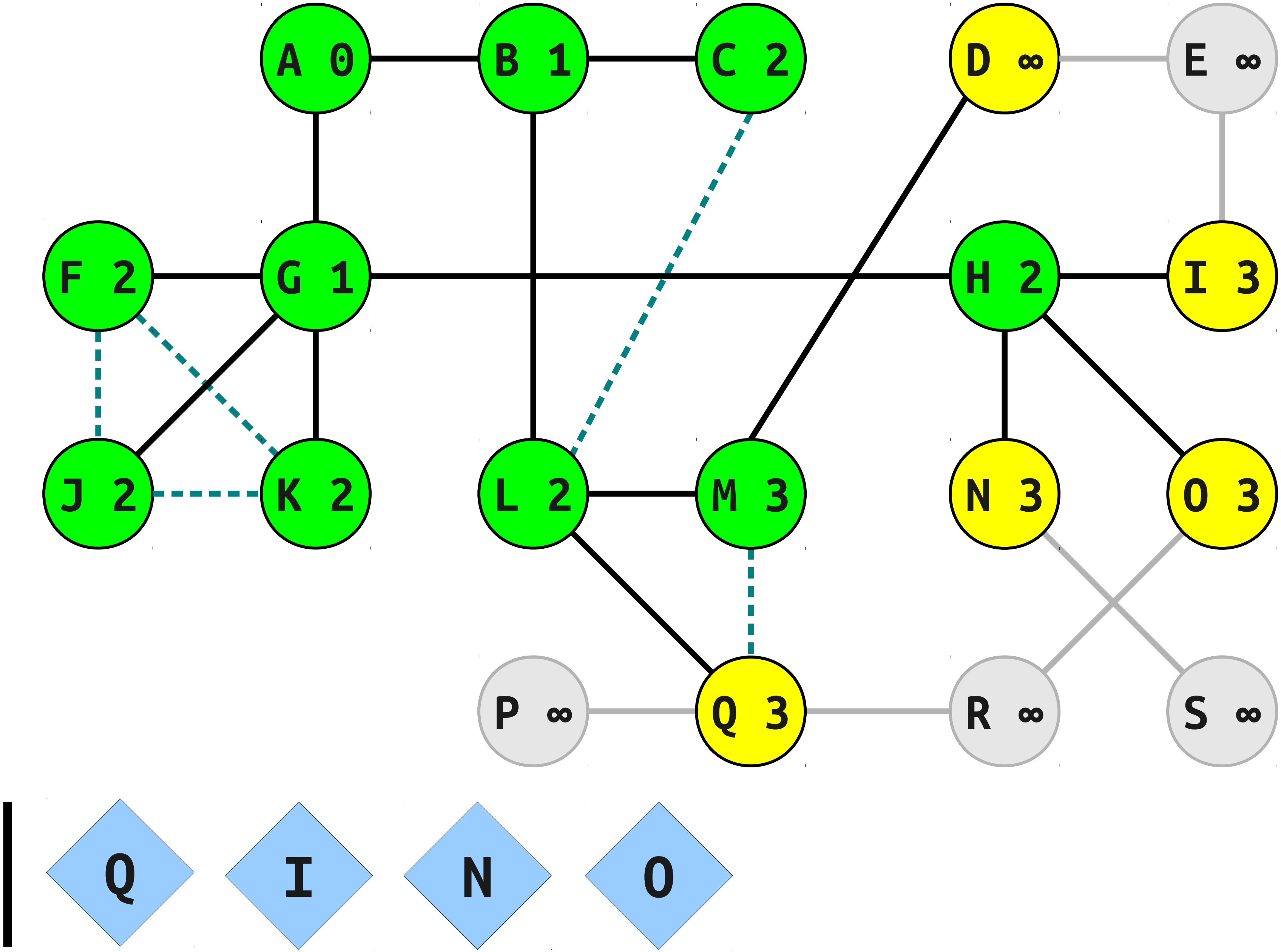
Breadth-First Search



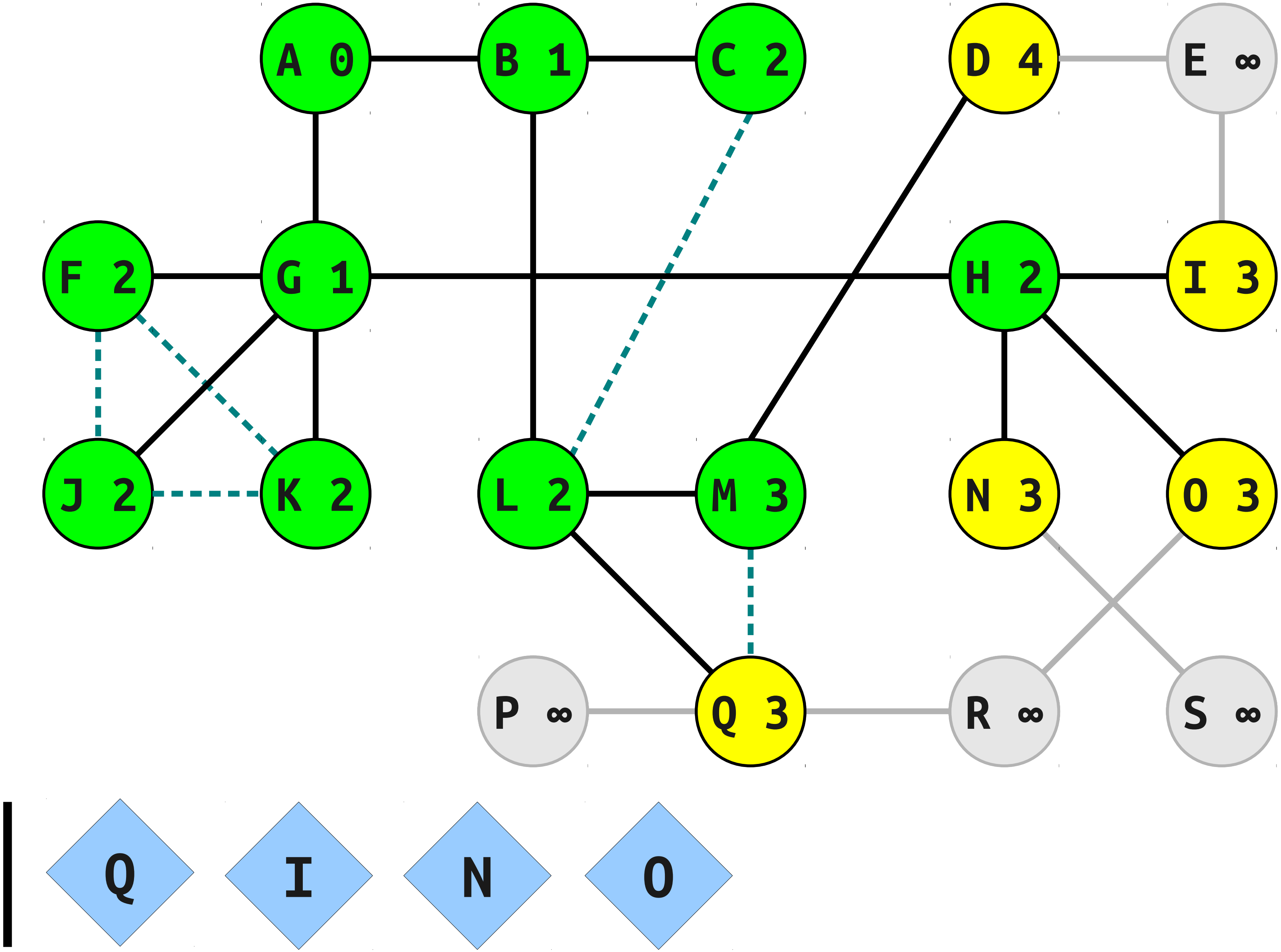
Breadth-First Search



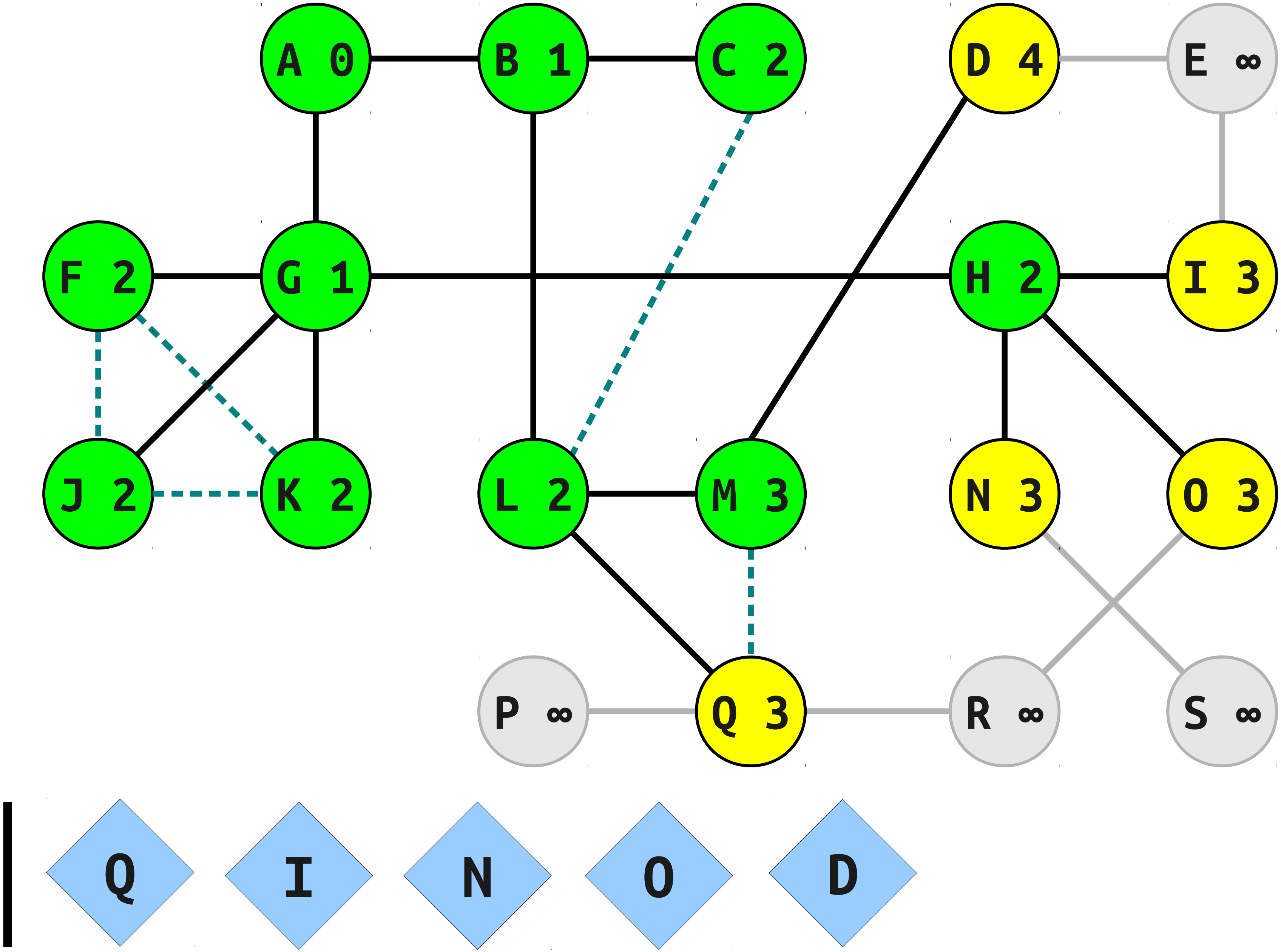
Breadth-First Search



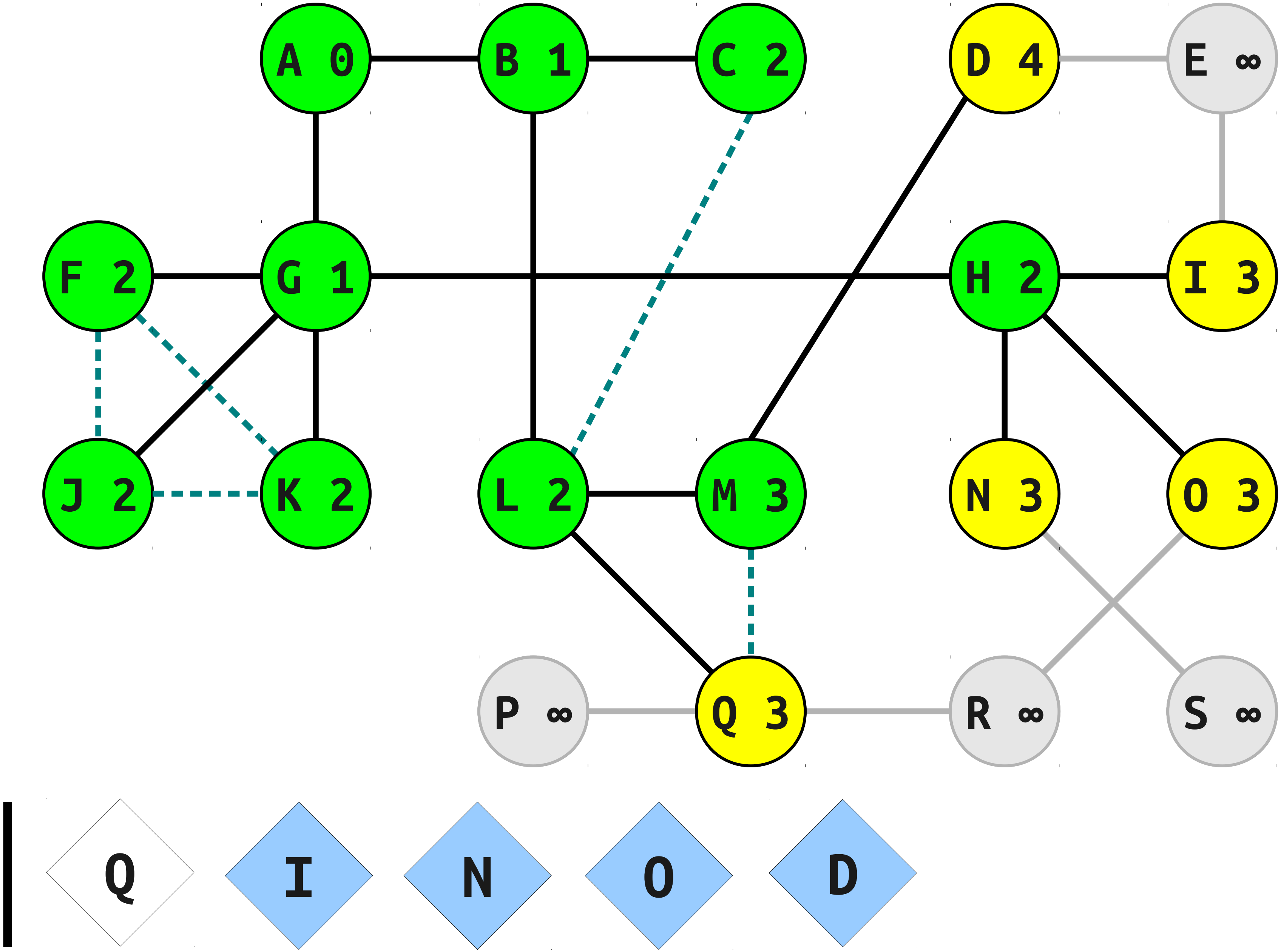
Breadth-First Search



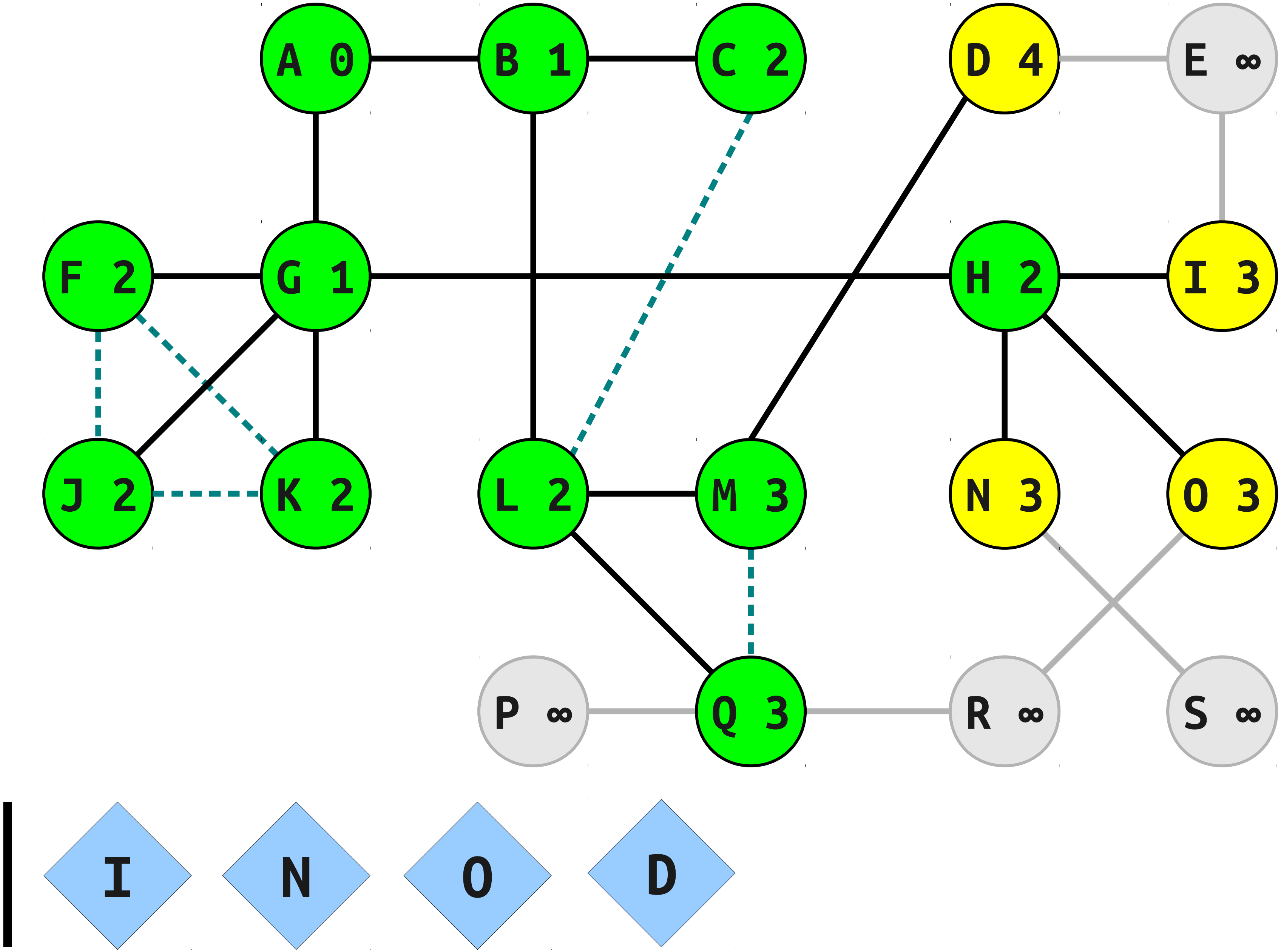
Breadth-First Search



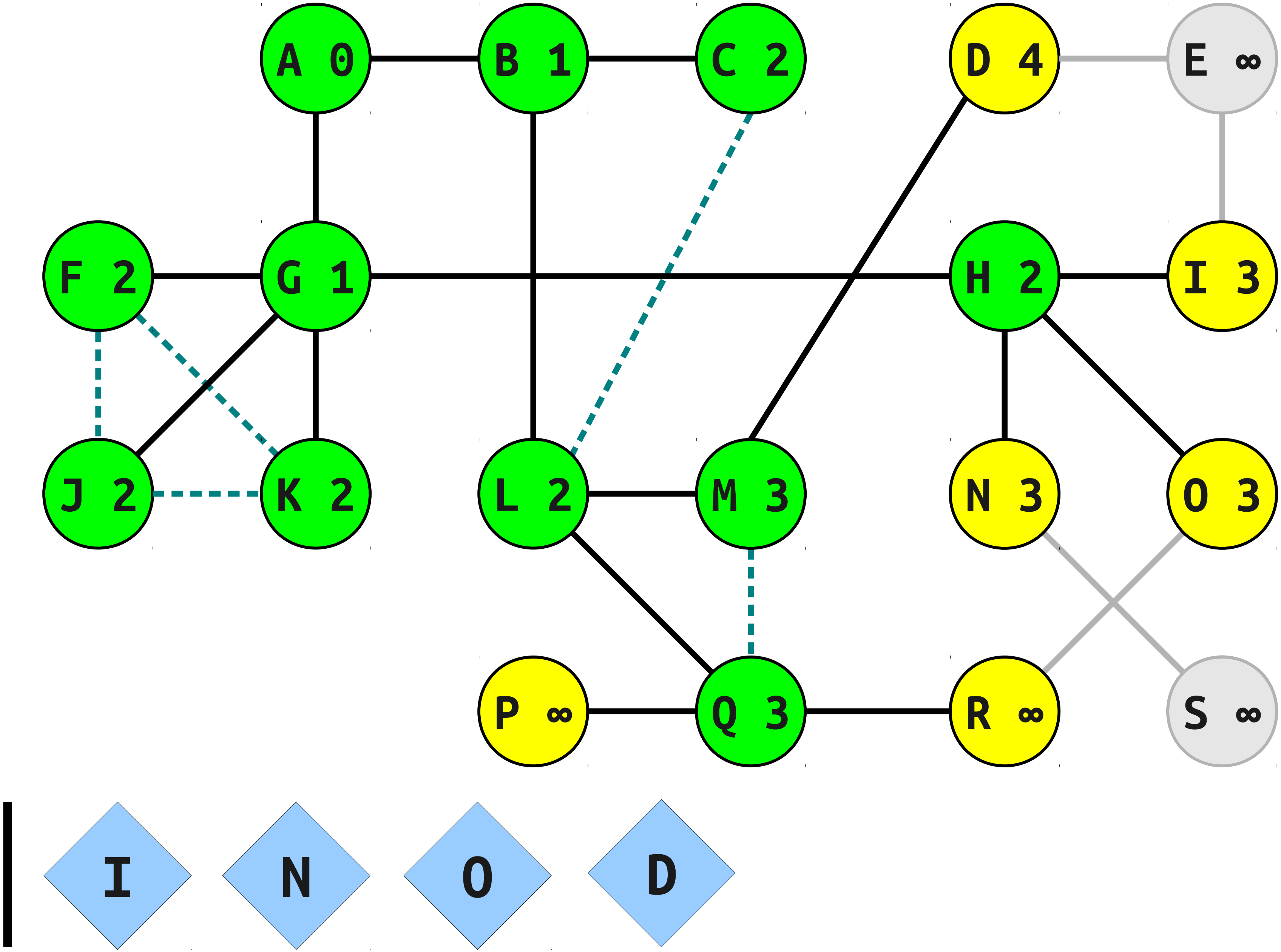
Breadth-First Search



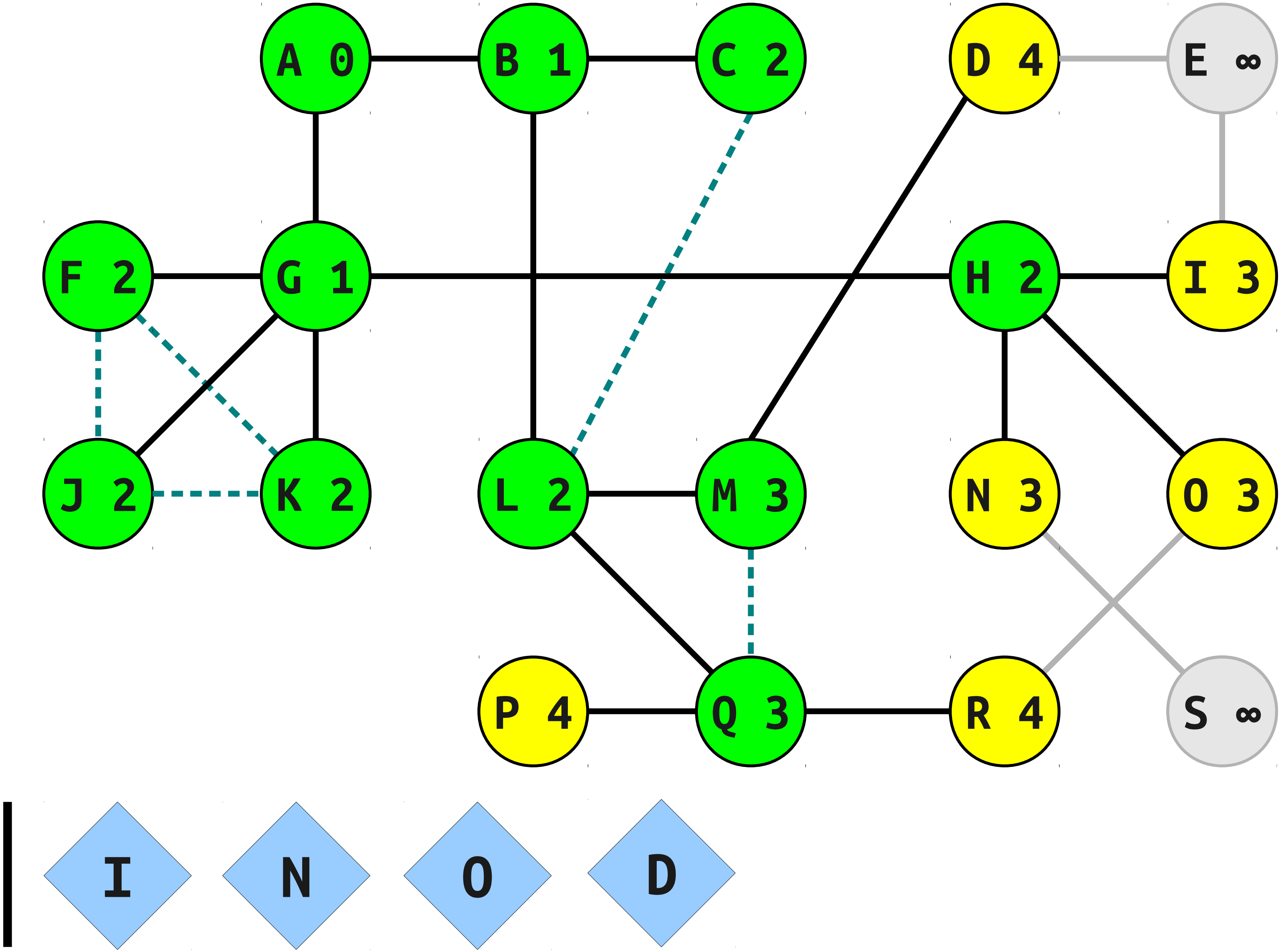
Breadth-First Search



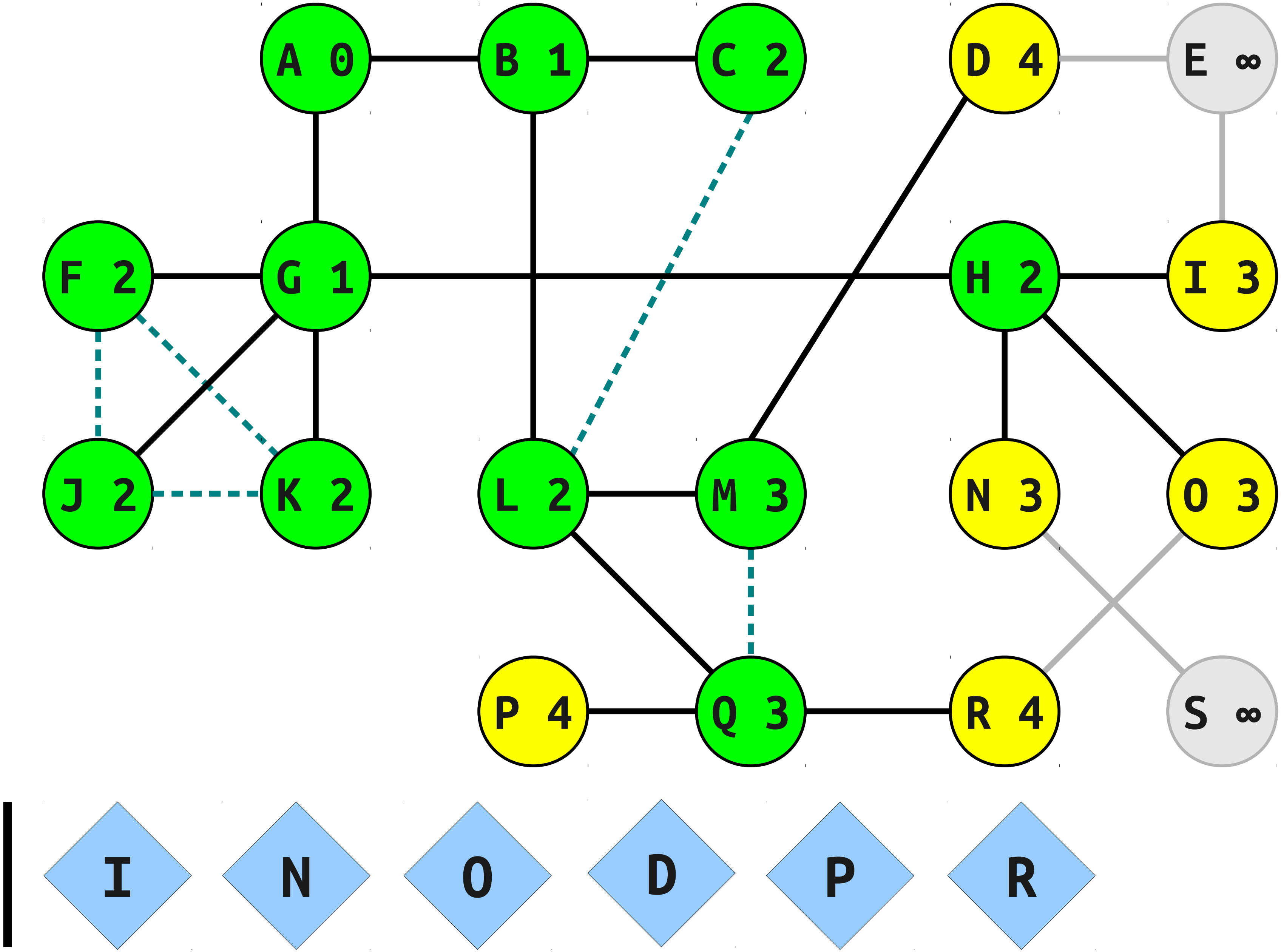
Breadth-First Search



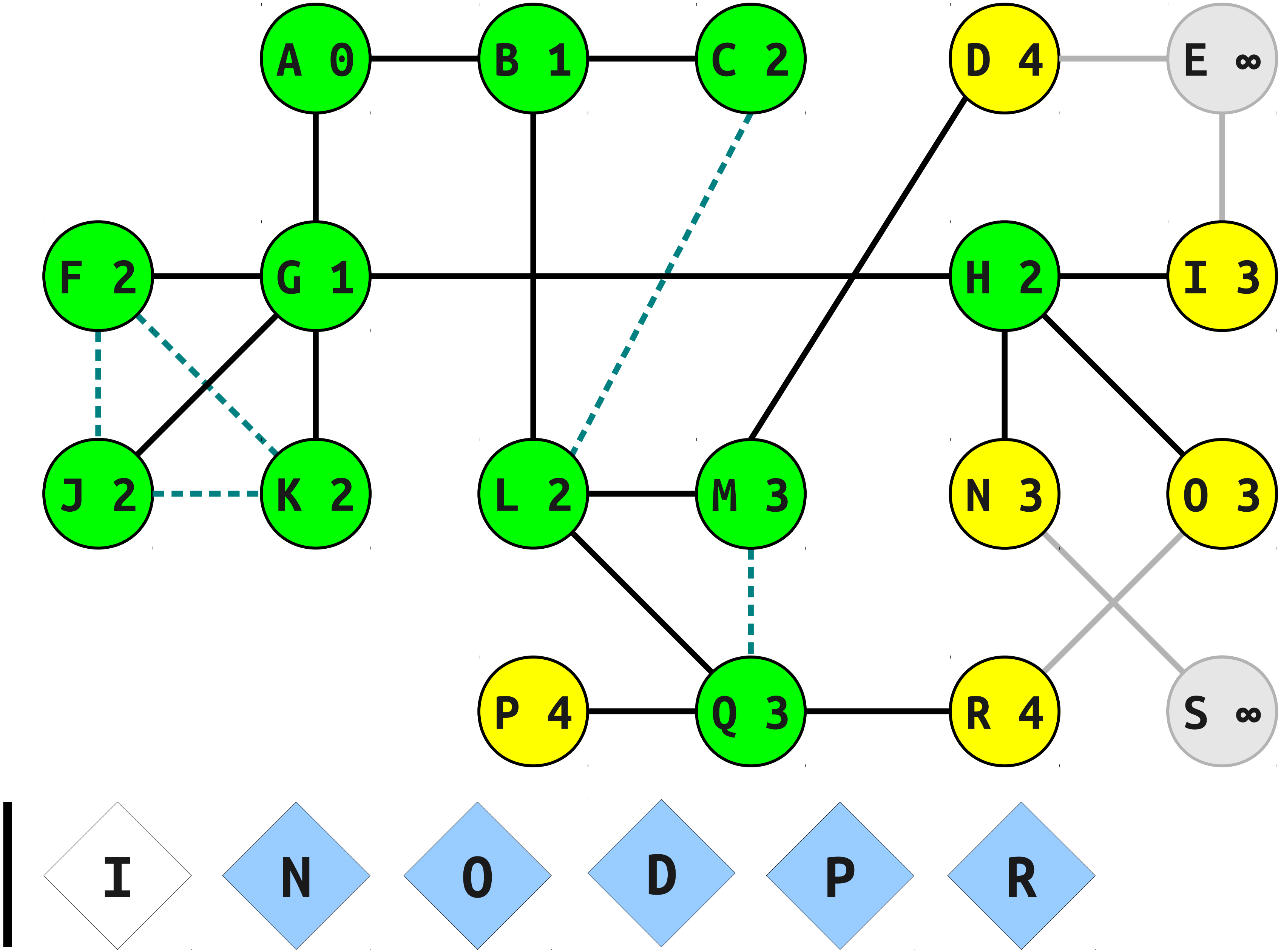
Breadth-First Search



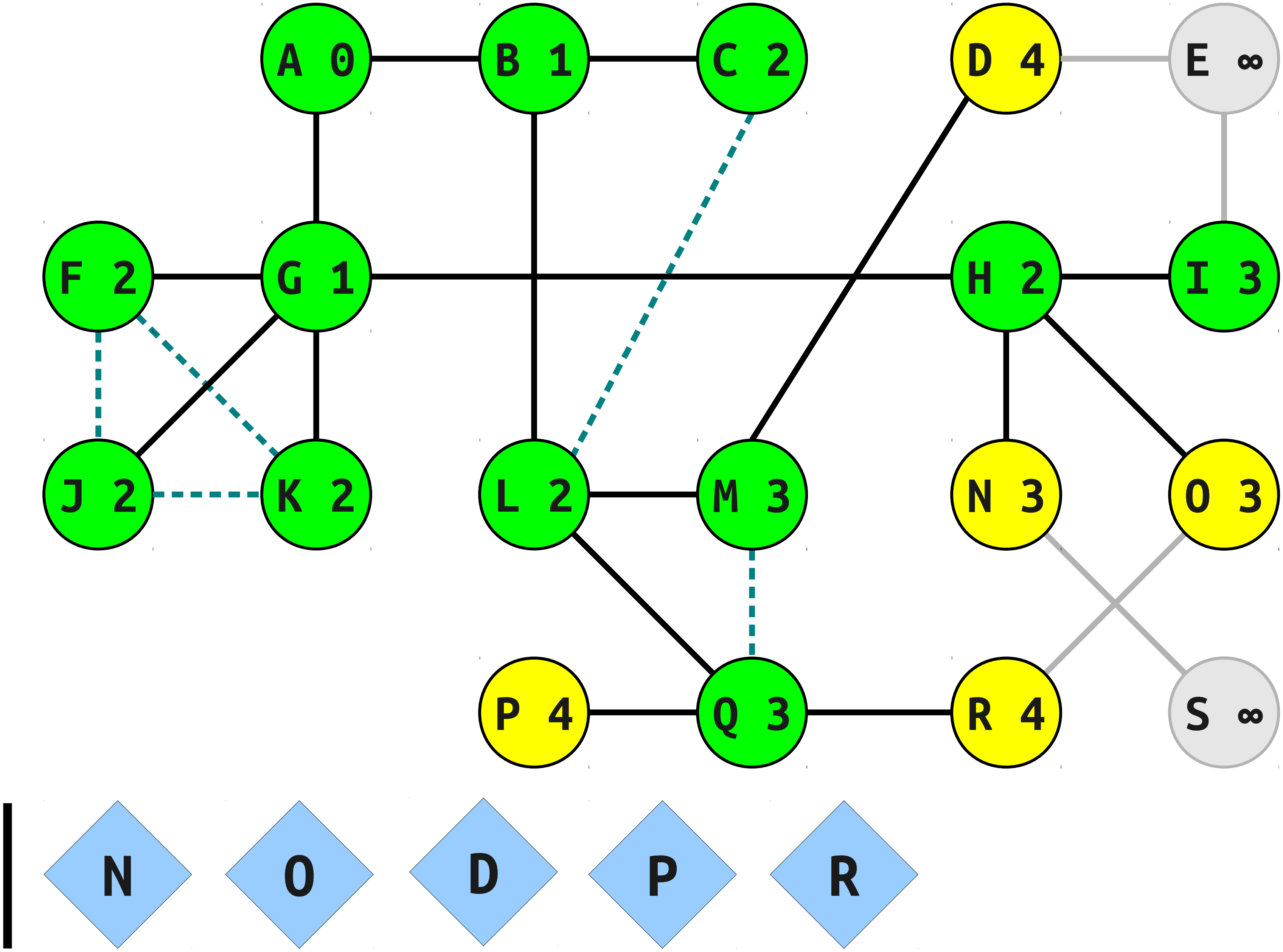
Breadth-First Search



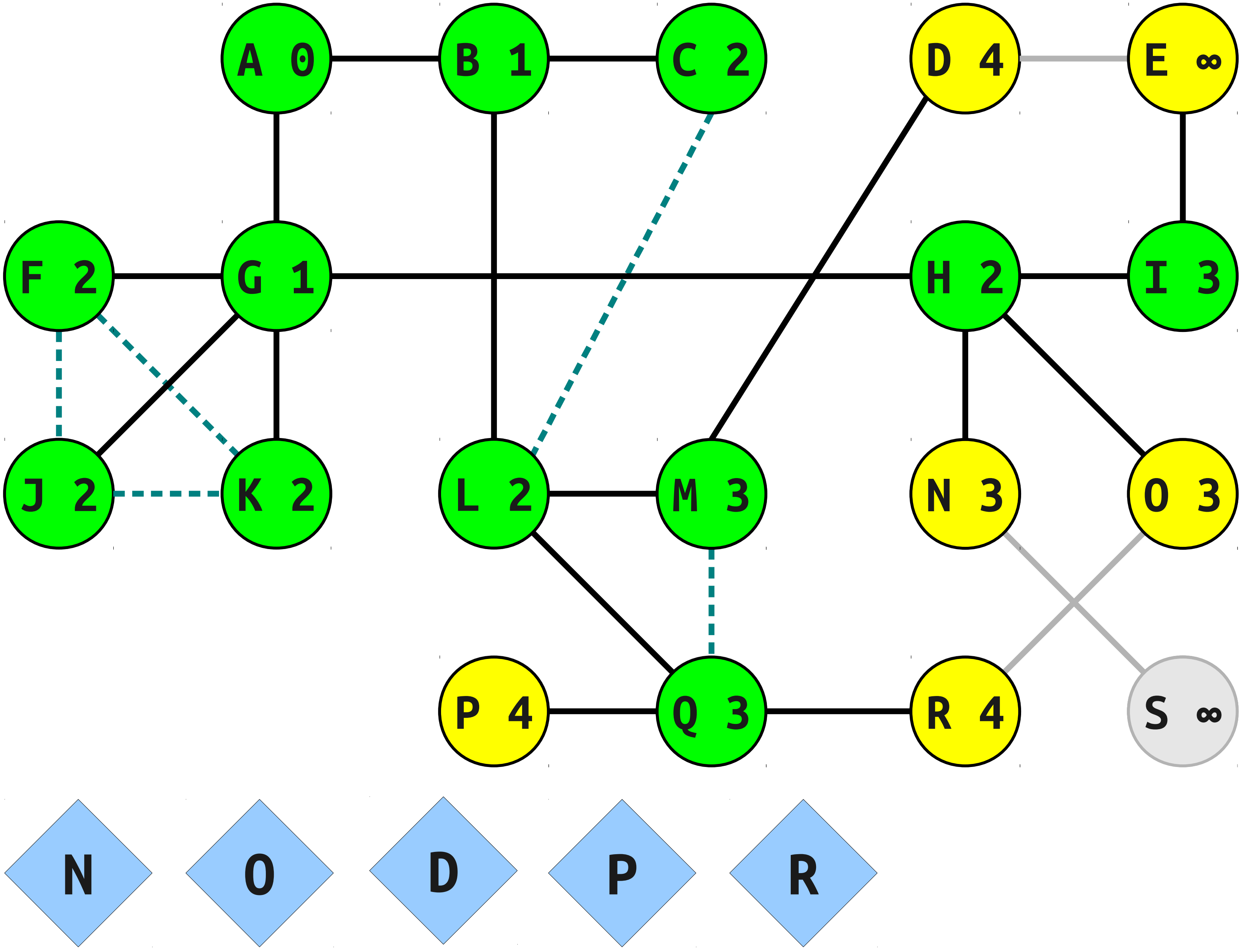
Breadth-First Search



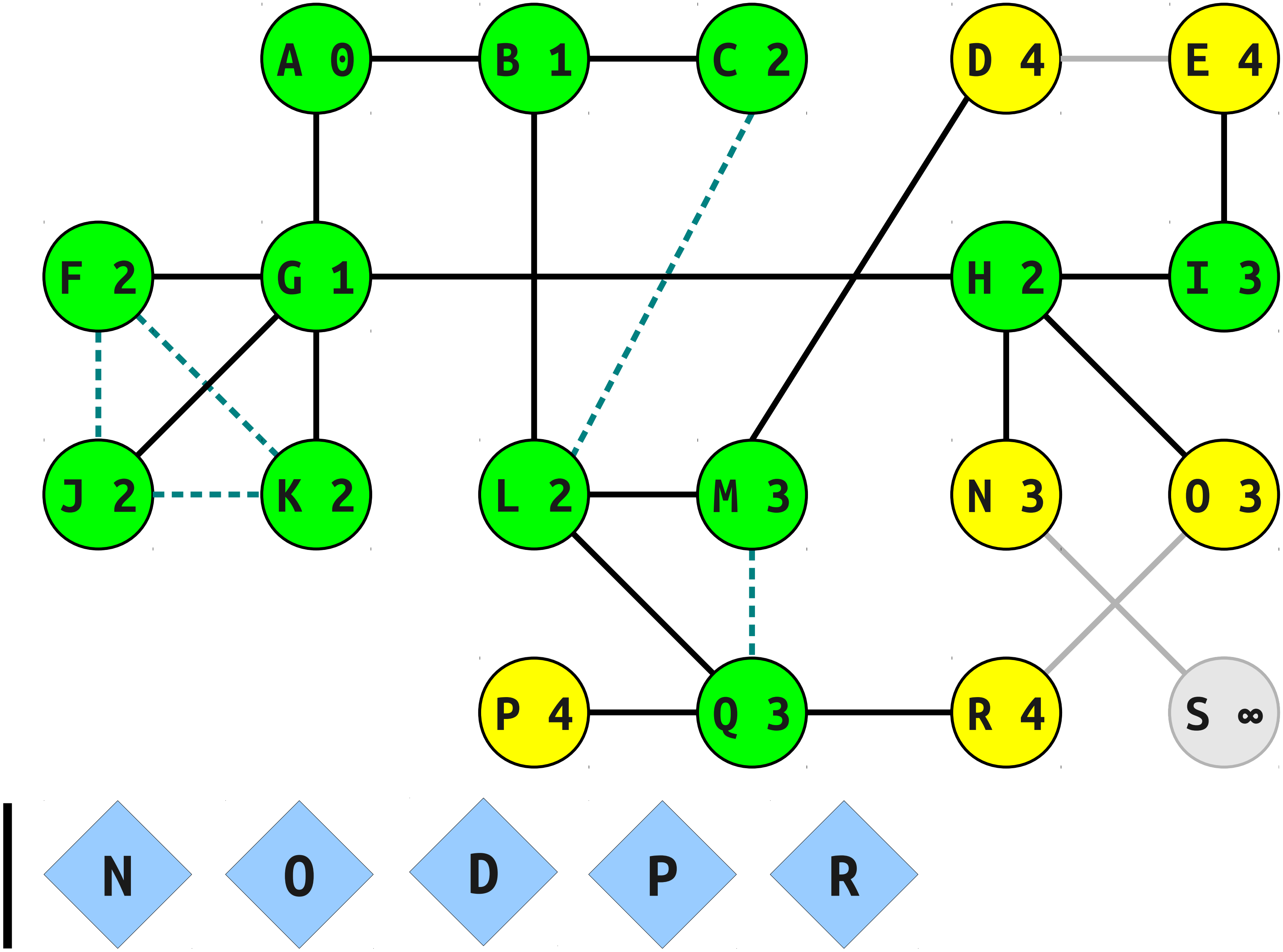
Breadth-First Search



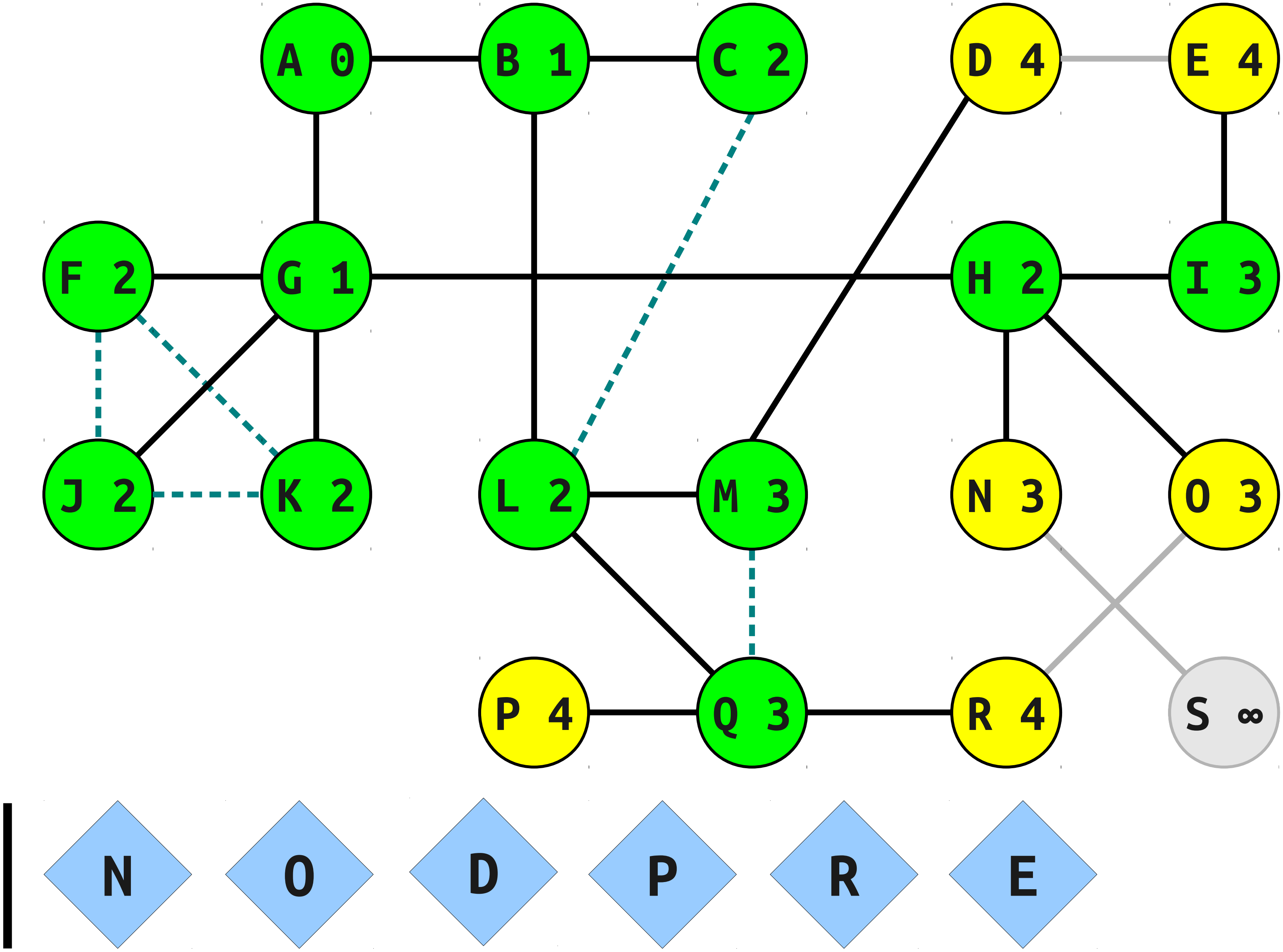
Breadth-First Search



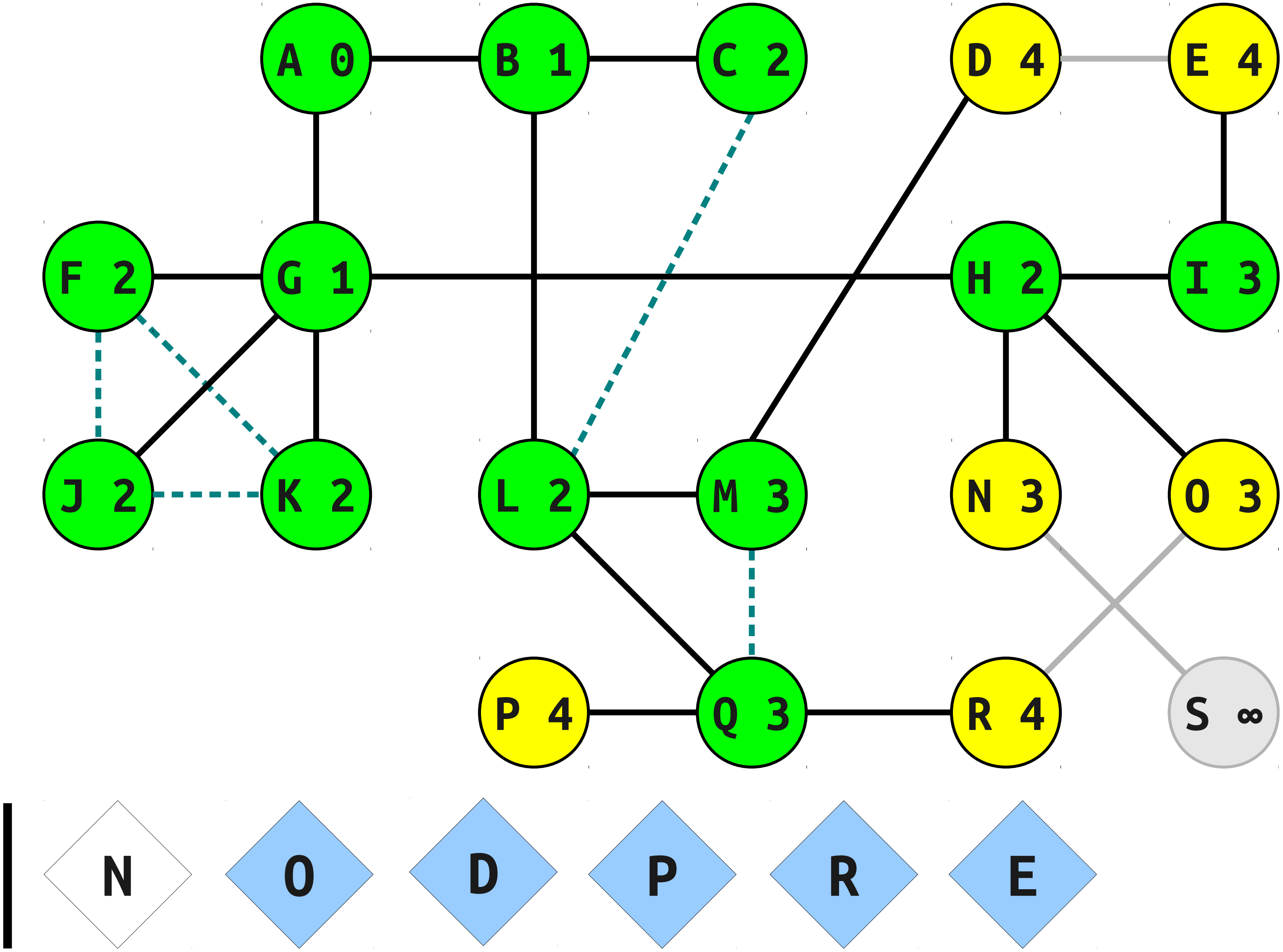
Breadth-First Search



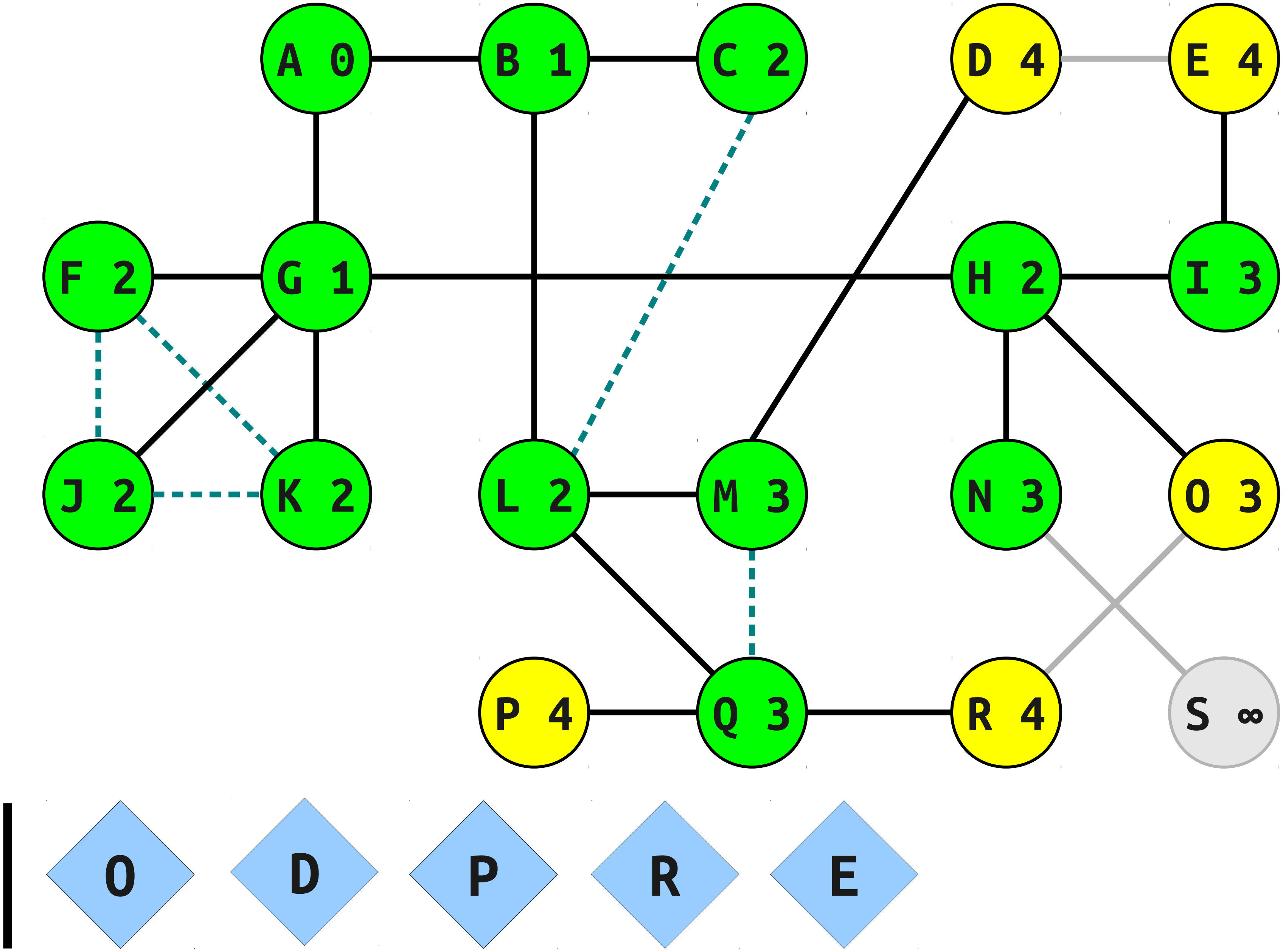
Breadth-First Search



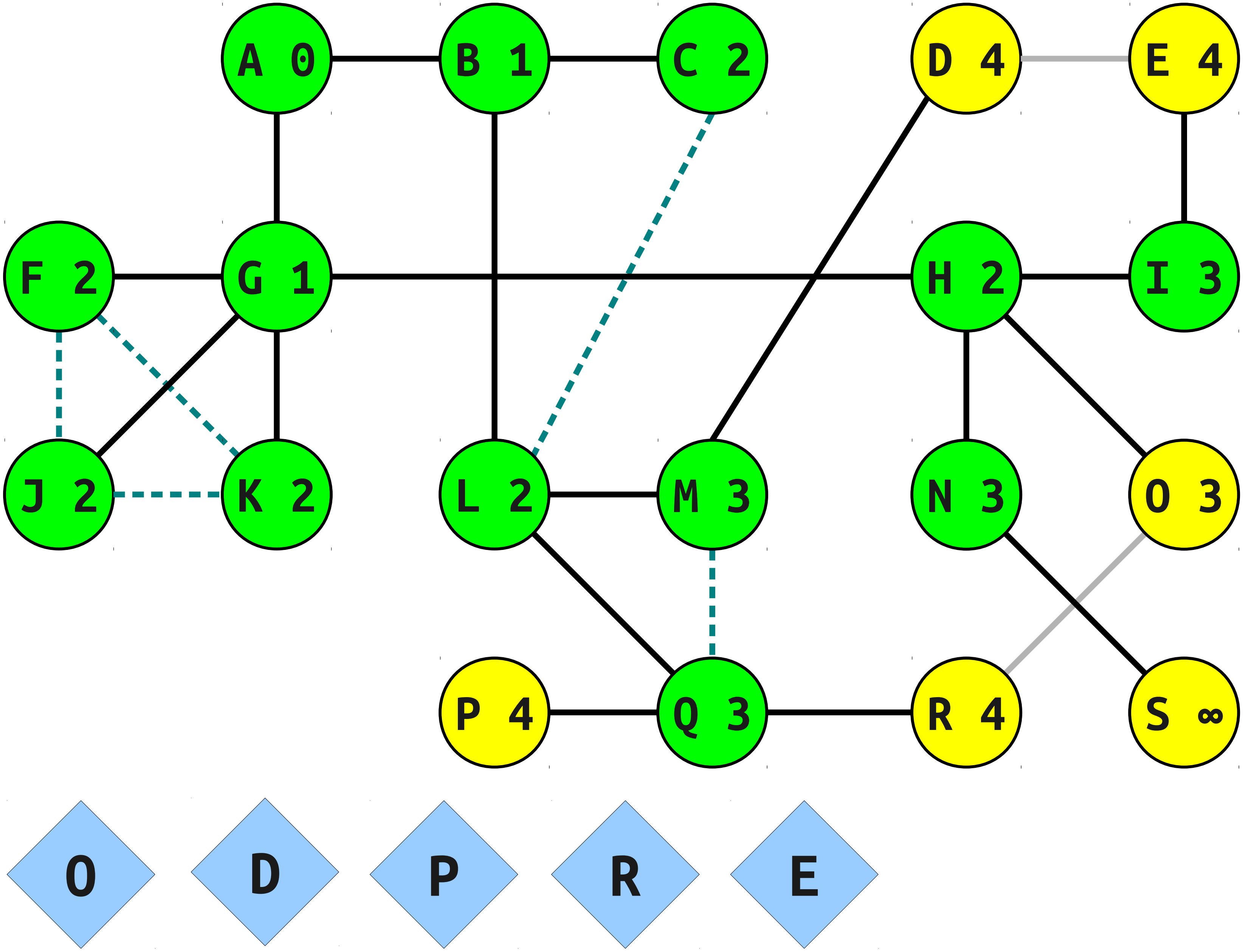
Breadth-First Search



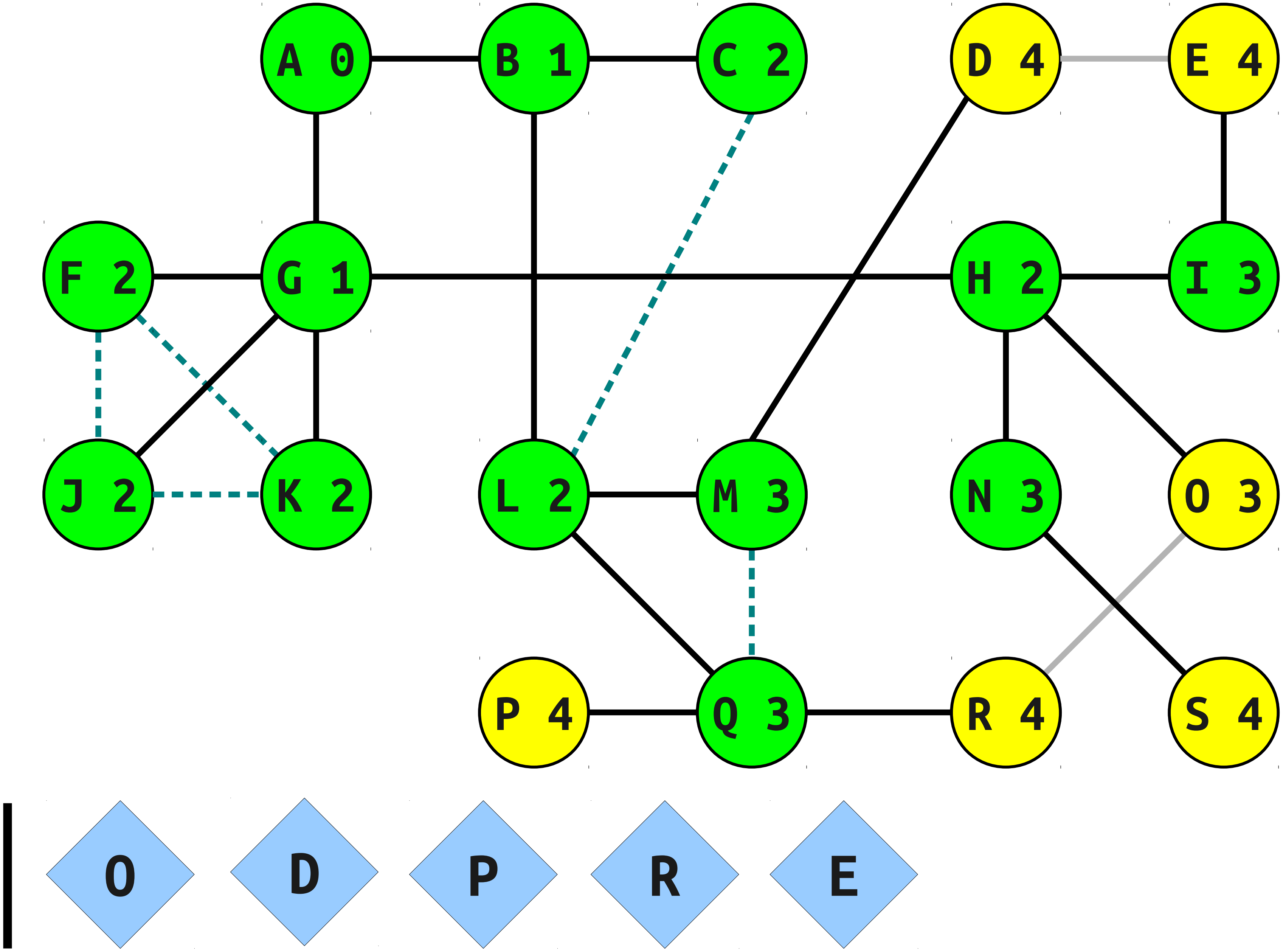
Breadth-First Search



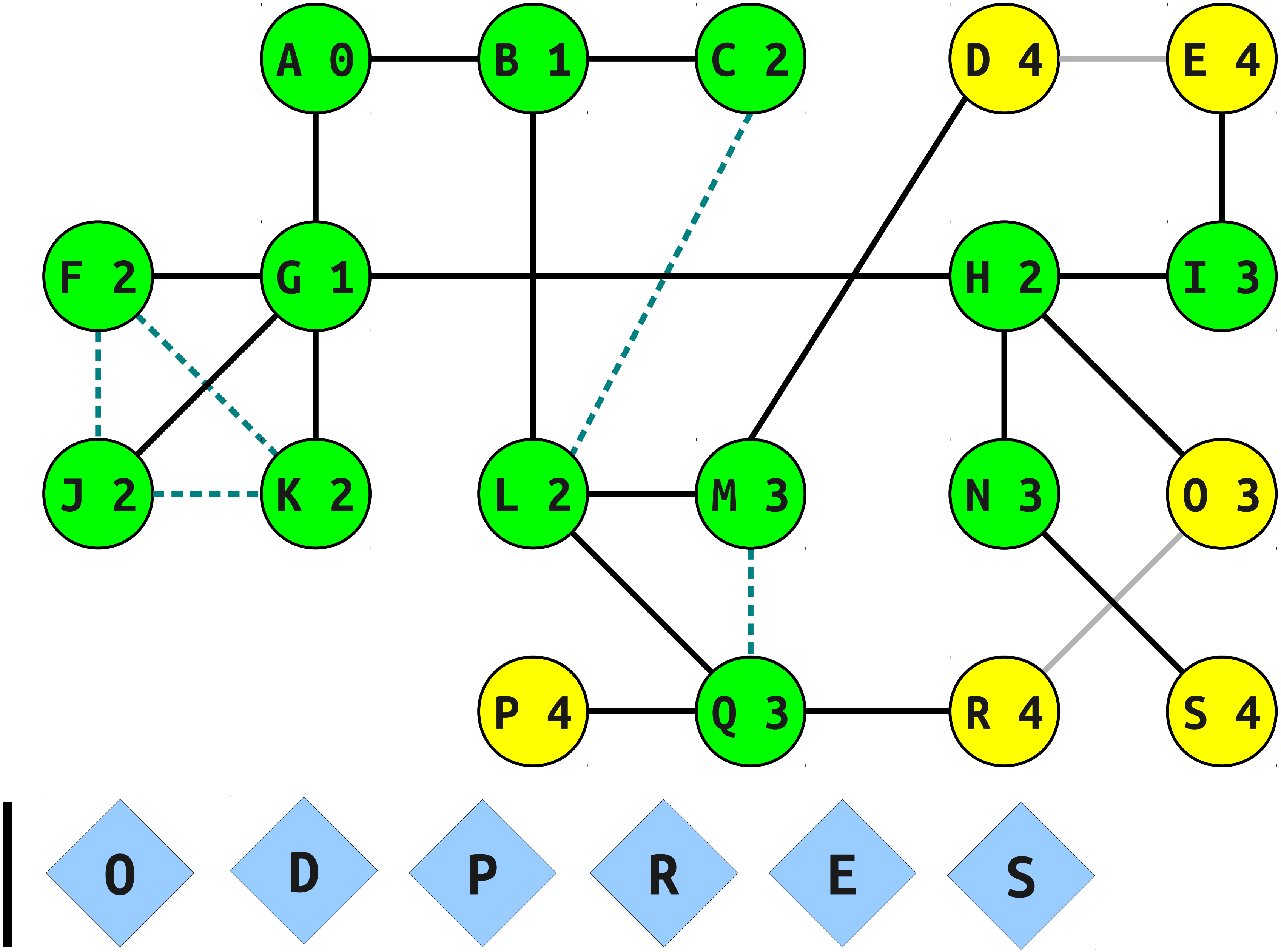
Breadth-First Search



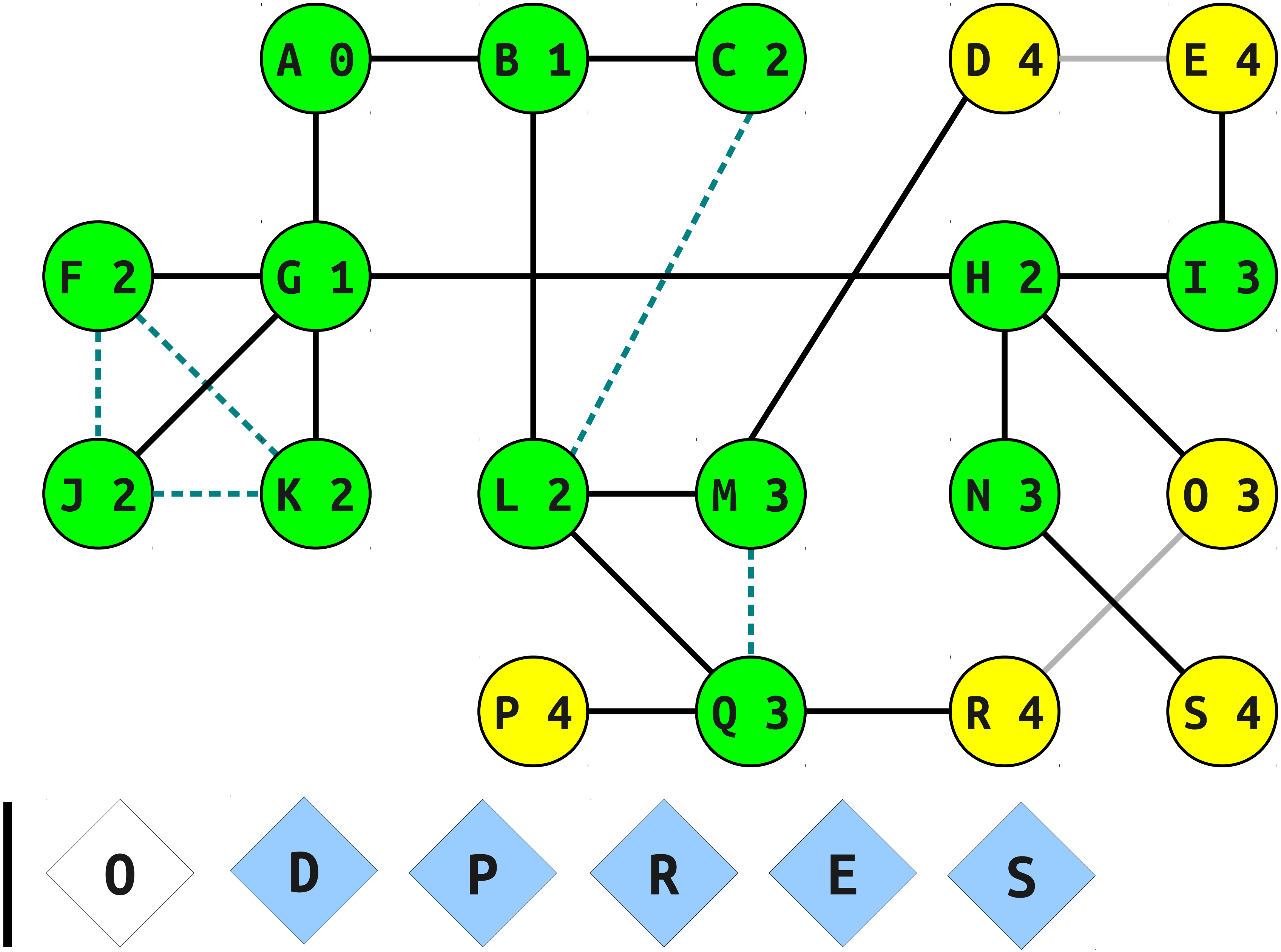
Breadth-First Search



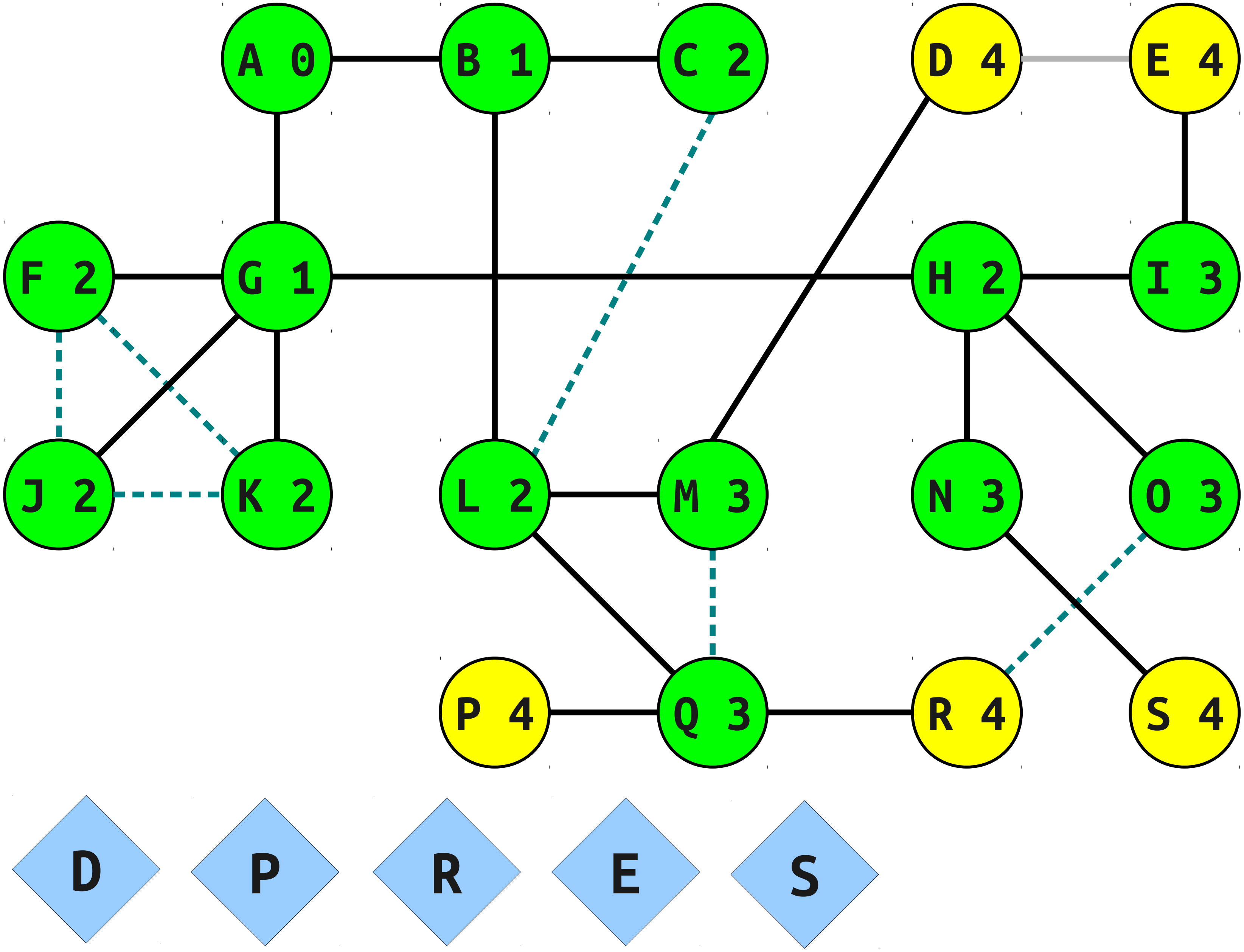
Breadth-First Search



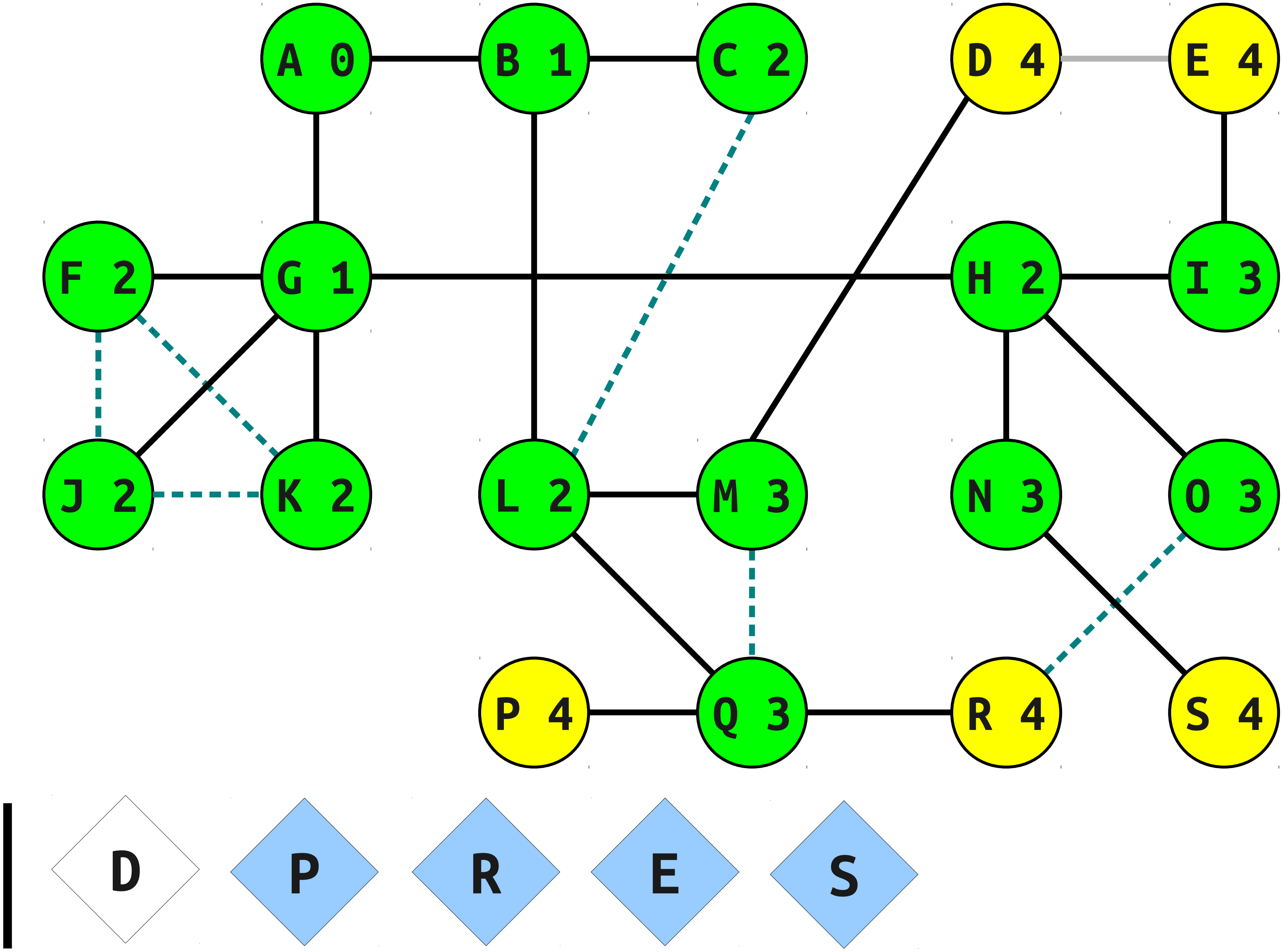
Breadth-First Search



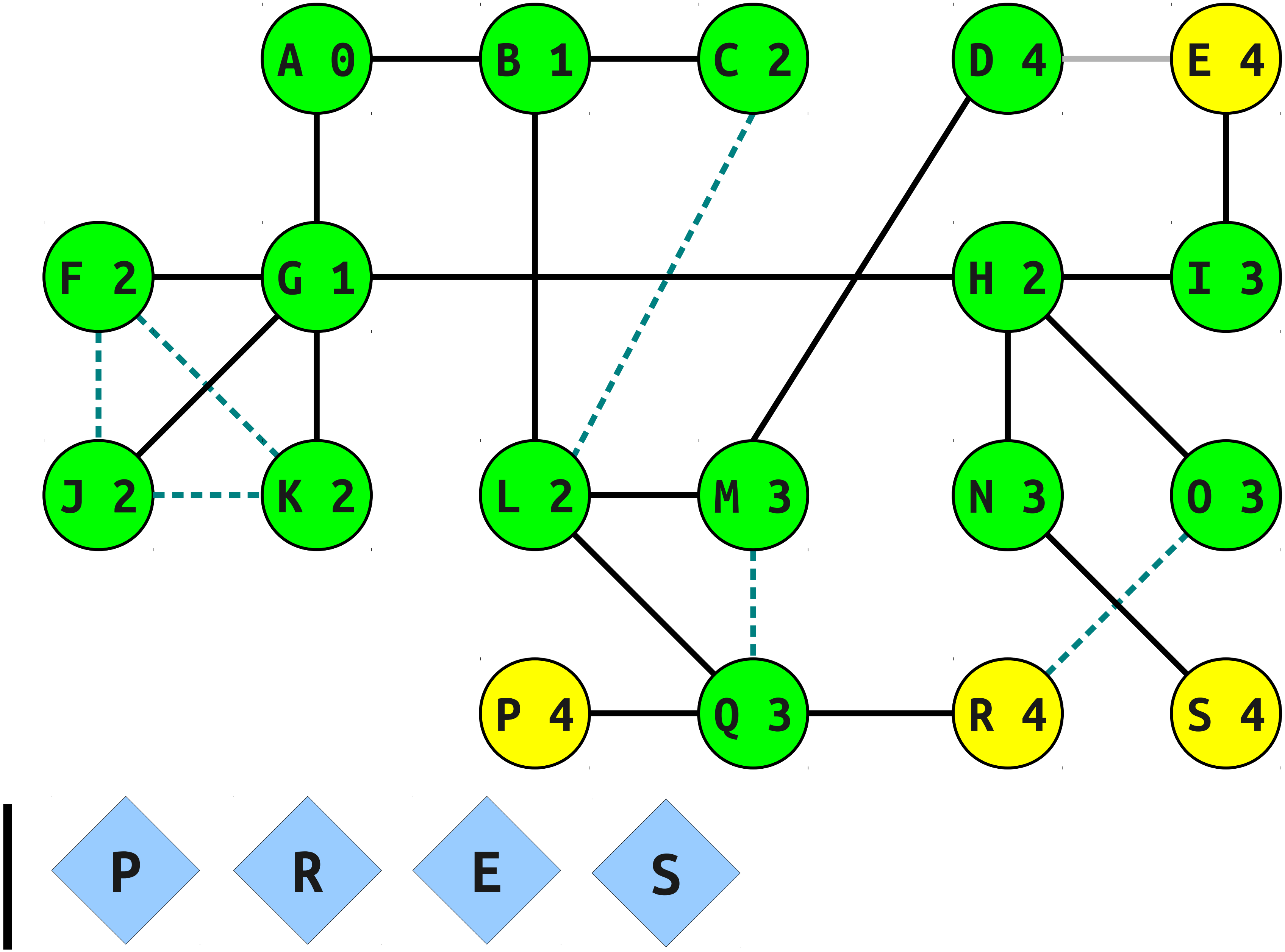
Breadth-First Search



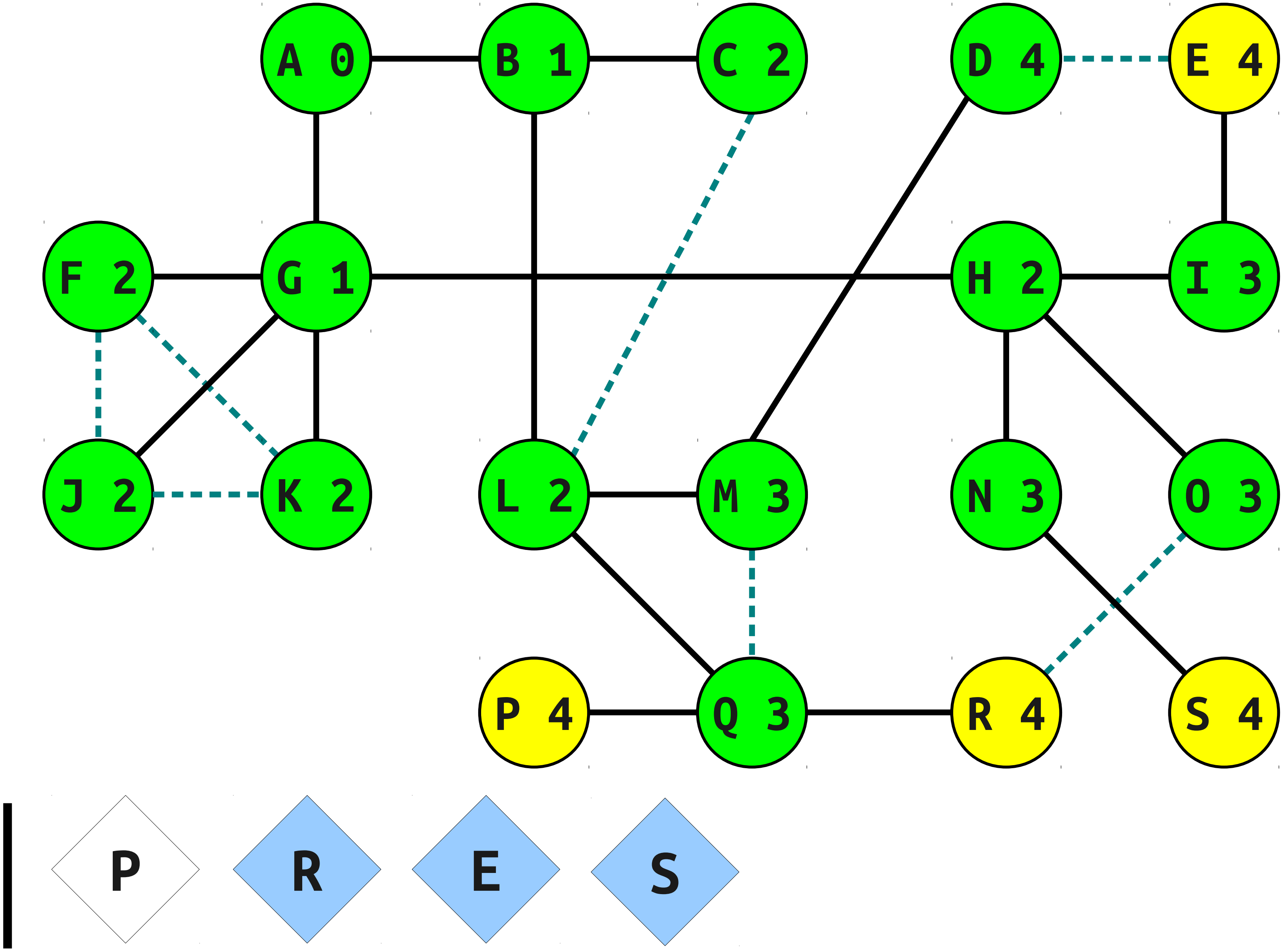
Breadth-First Search



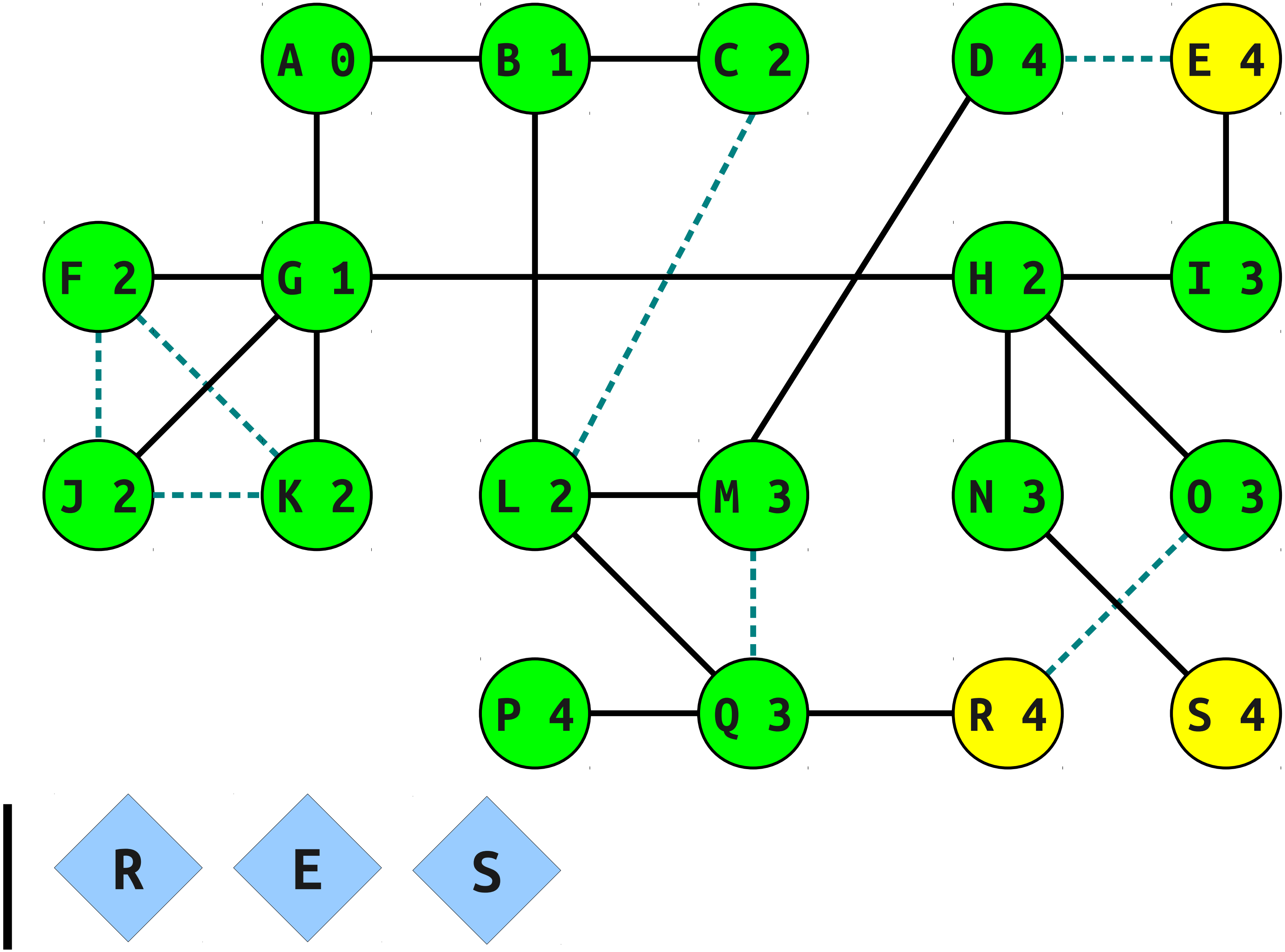
Breadth-First Search



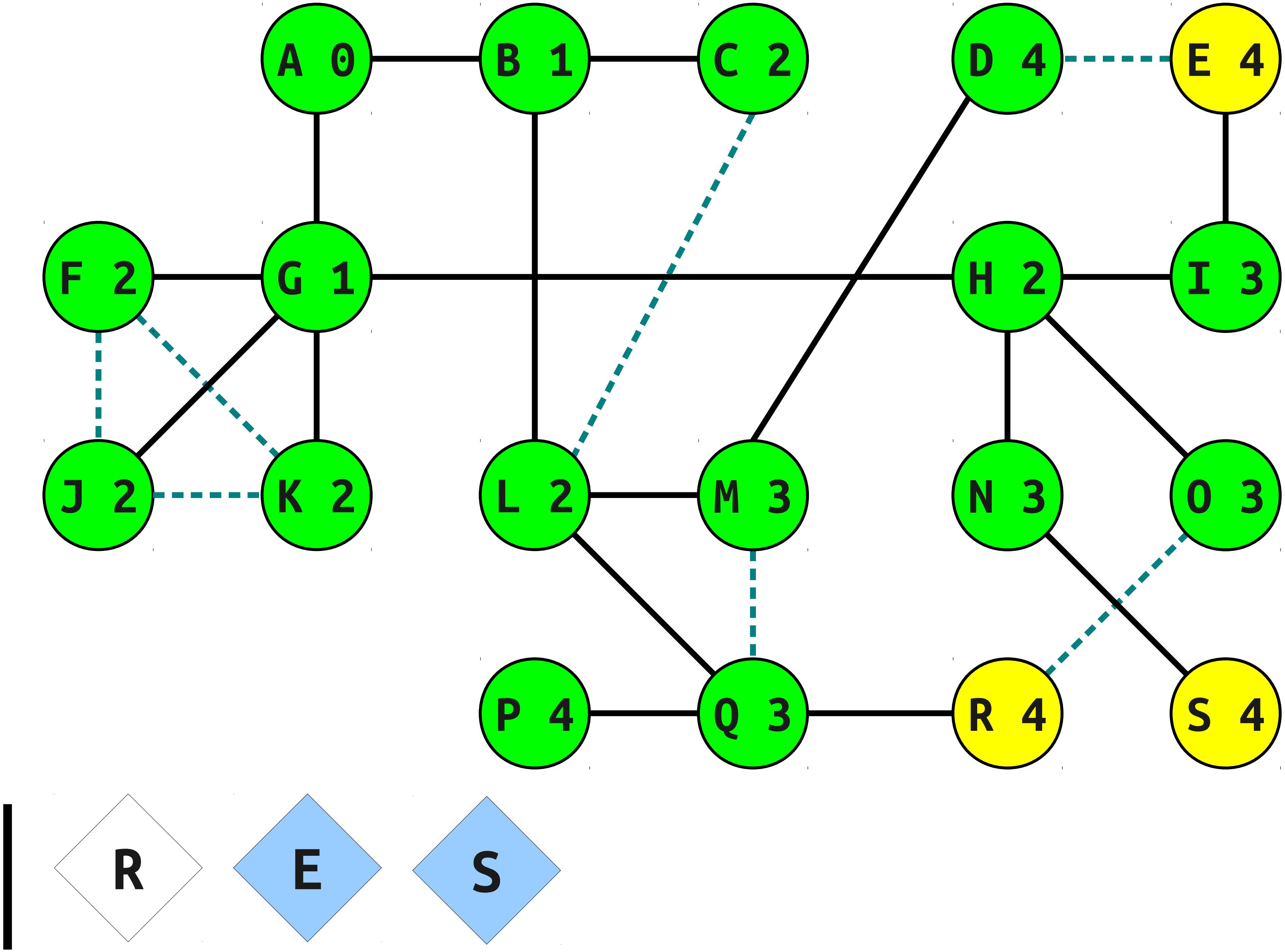
Breadth-First Search



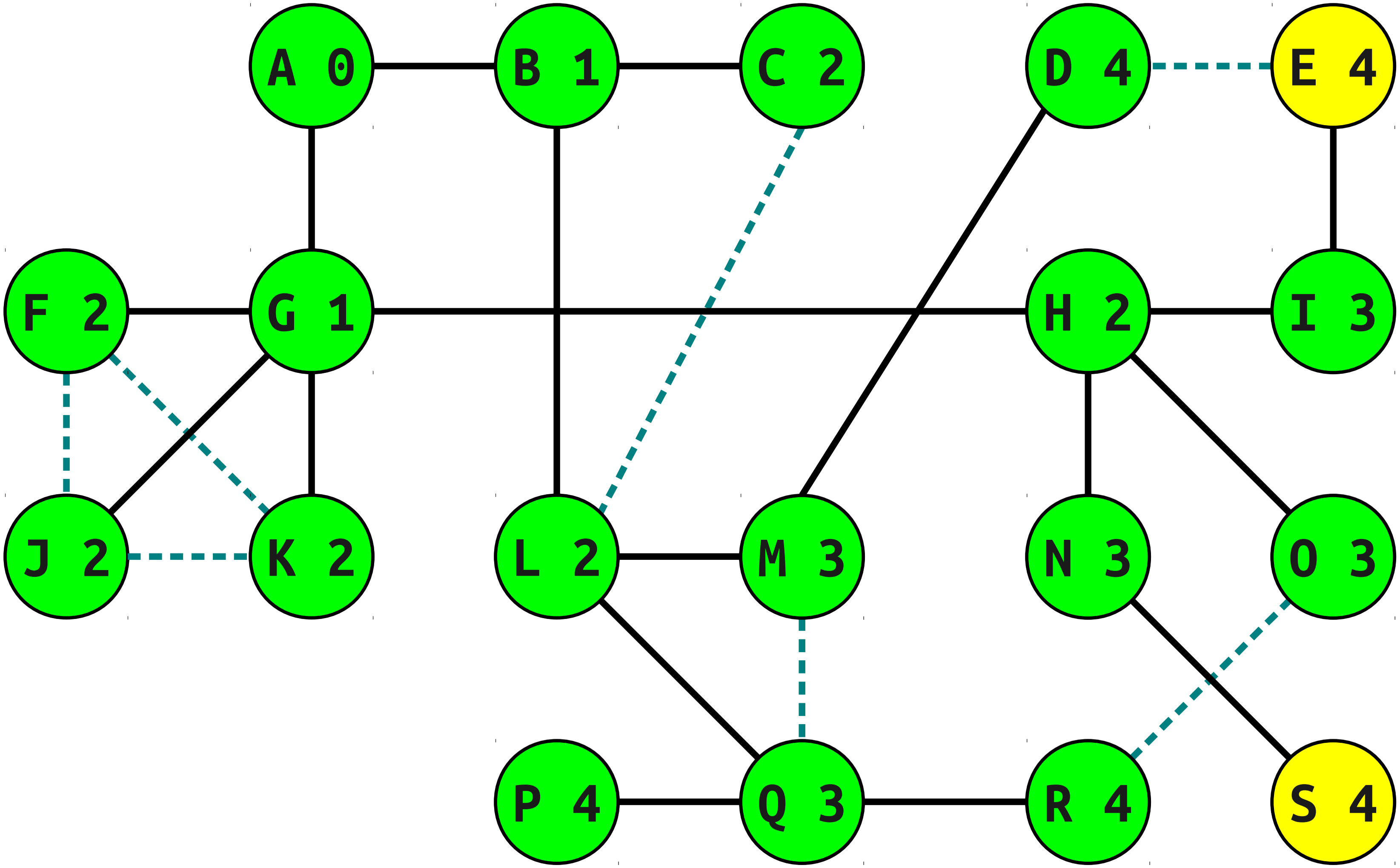
Breadth-First Search

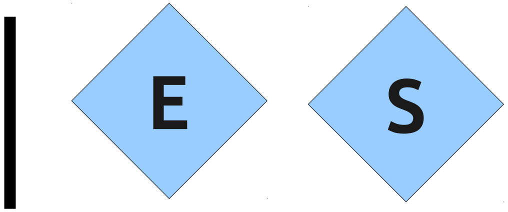


Breadth-First Search

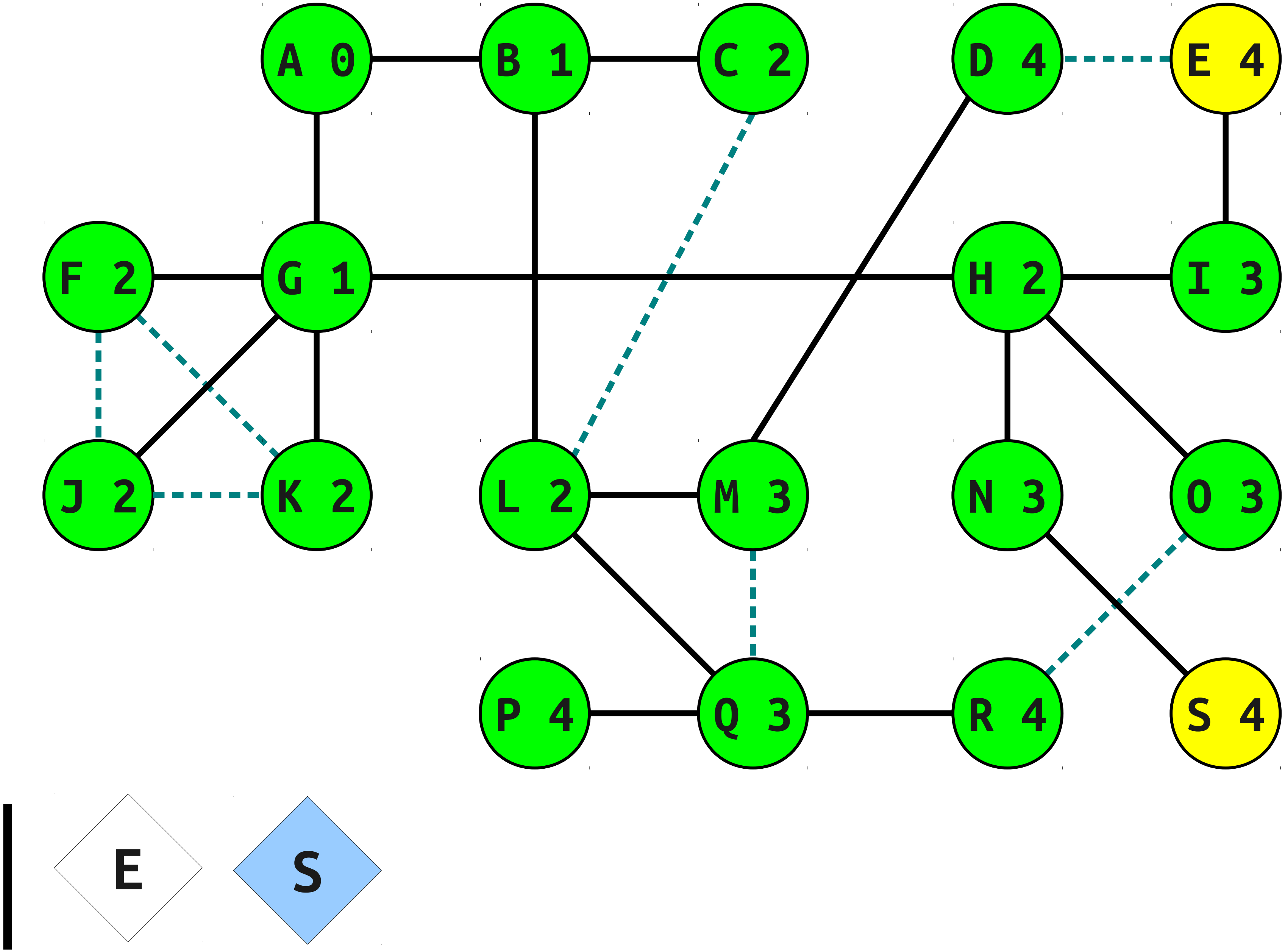


Breadth-First Search

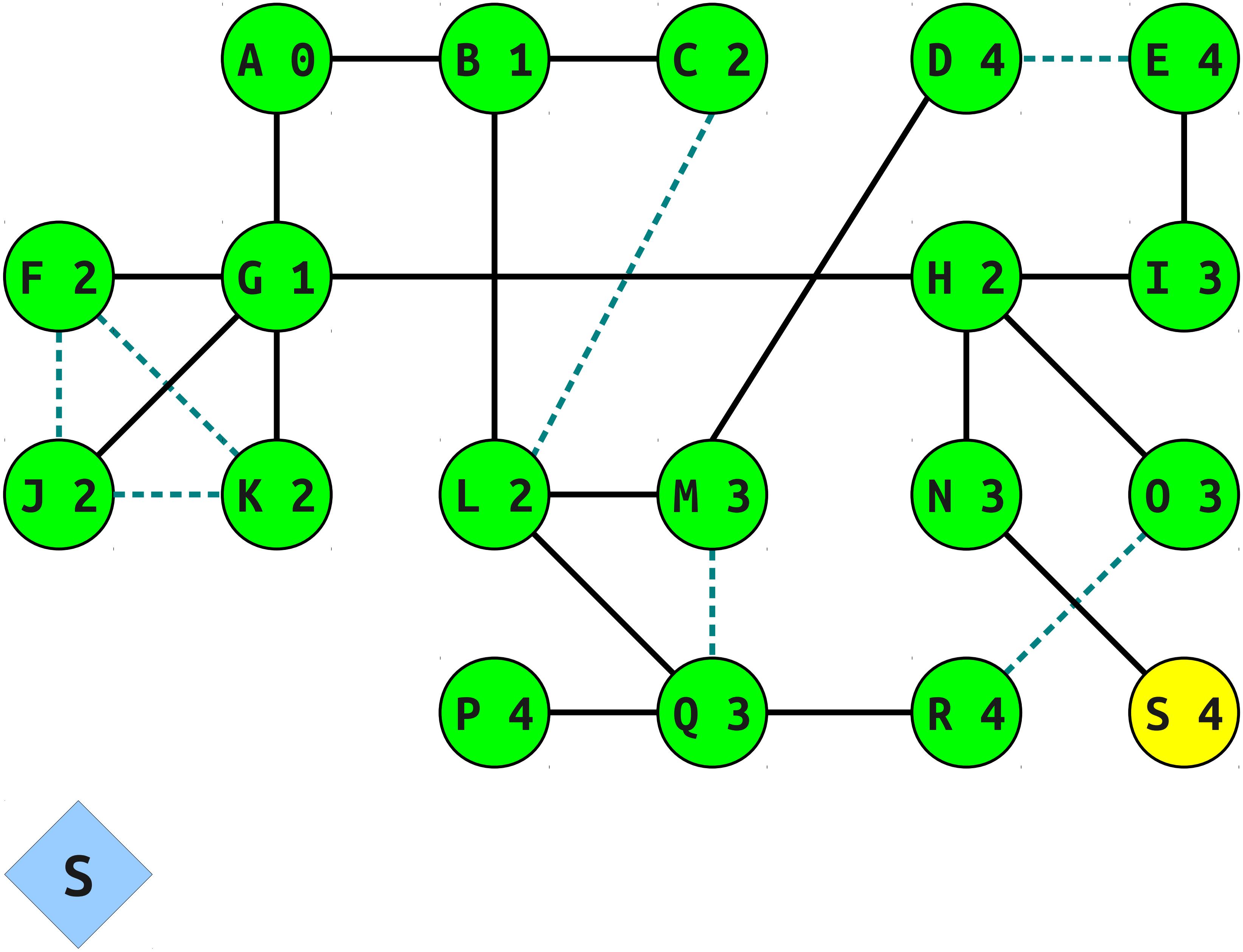




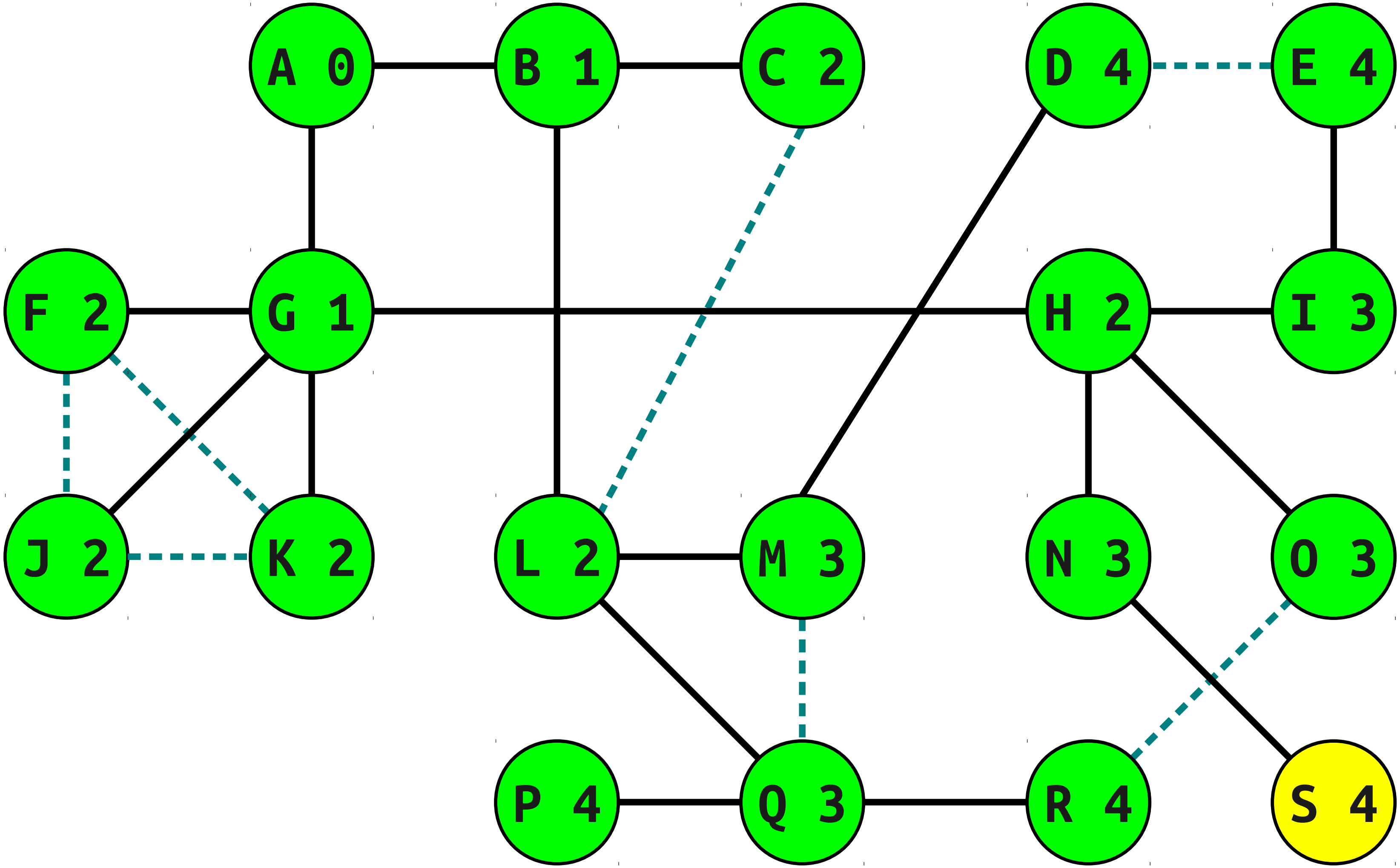
Breadth-First Search

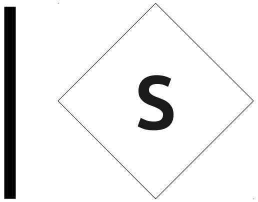


Breadth-First Search

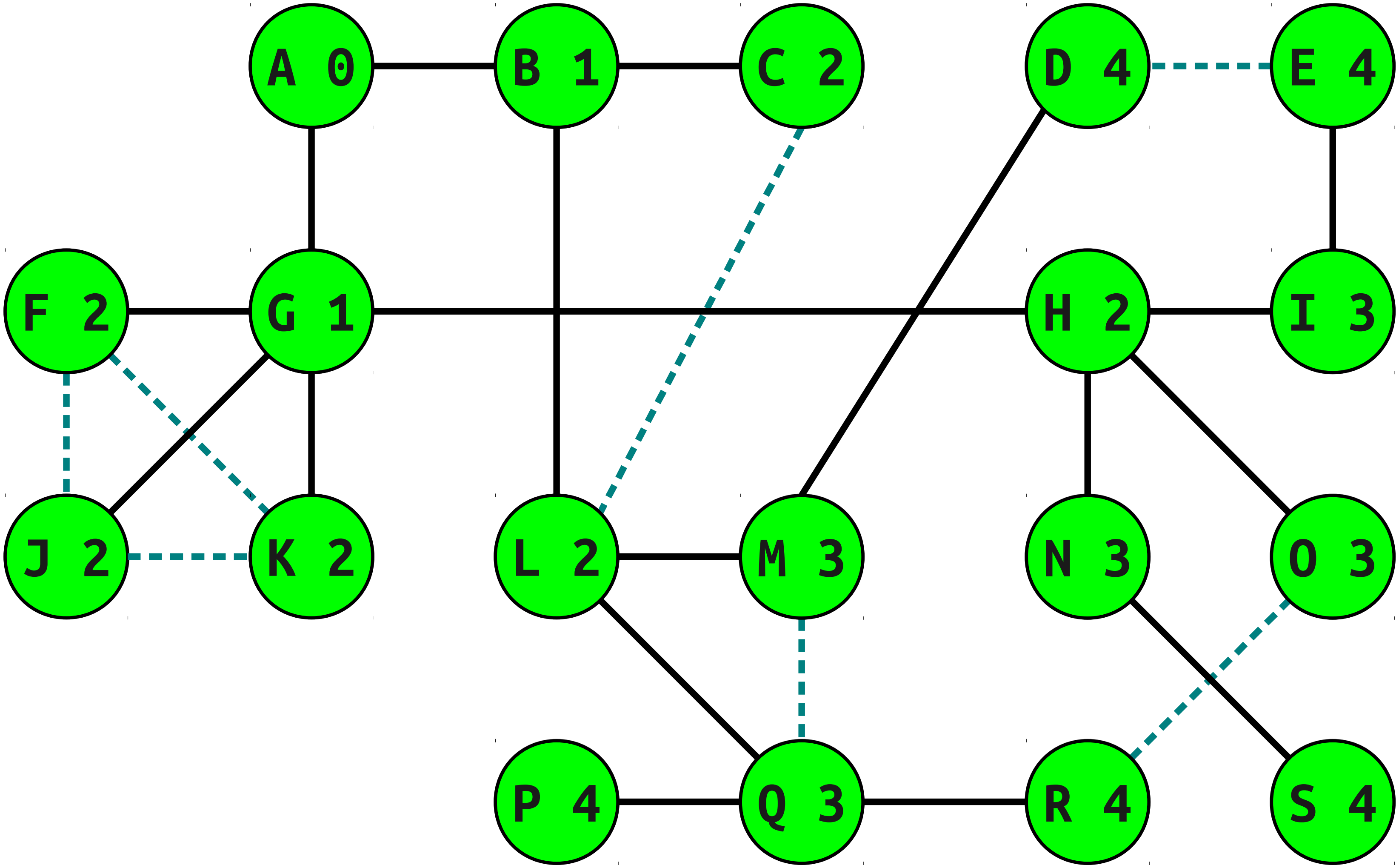


Breadth-First Search

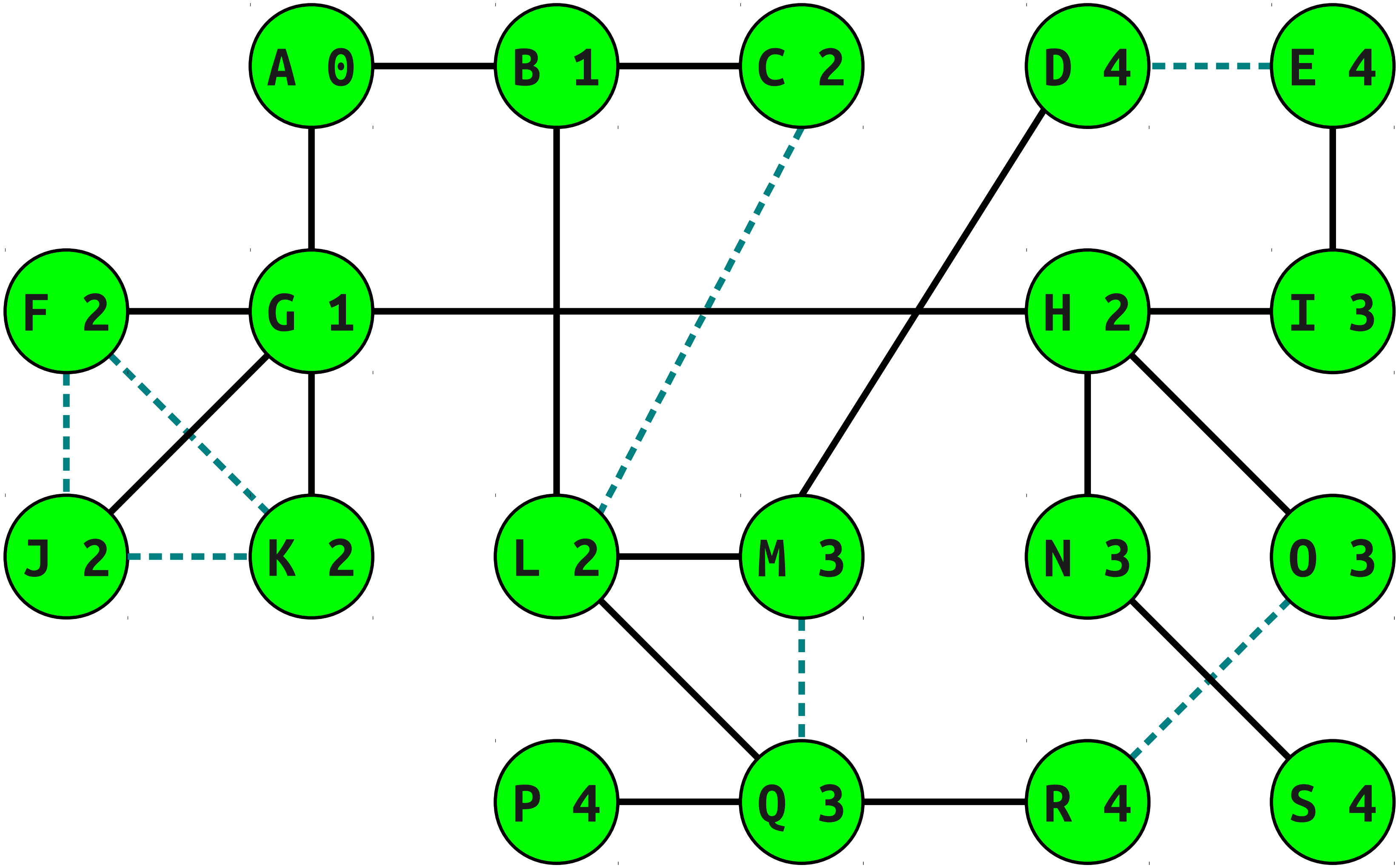




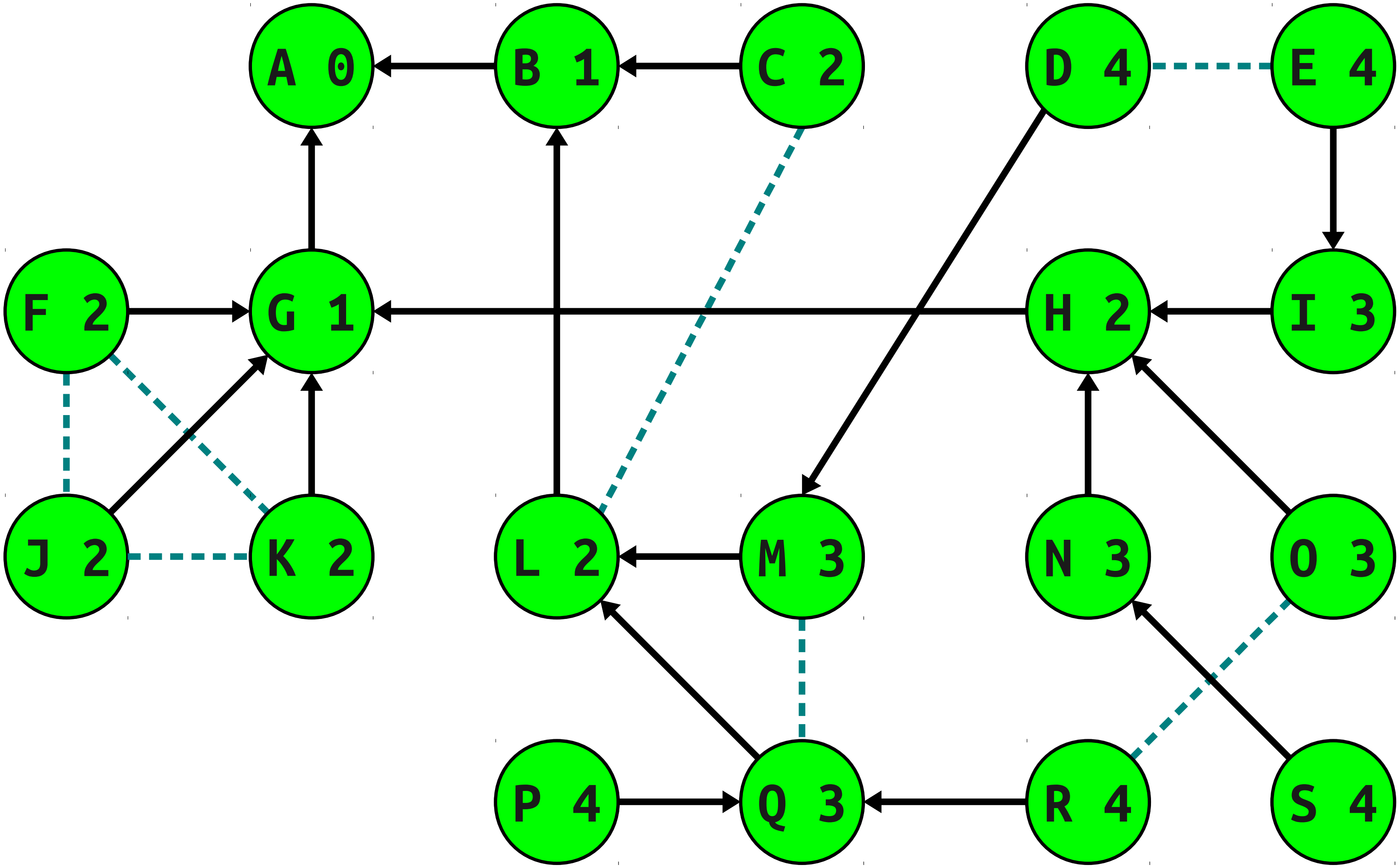
Breadth-First Search



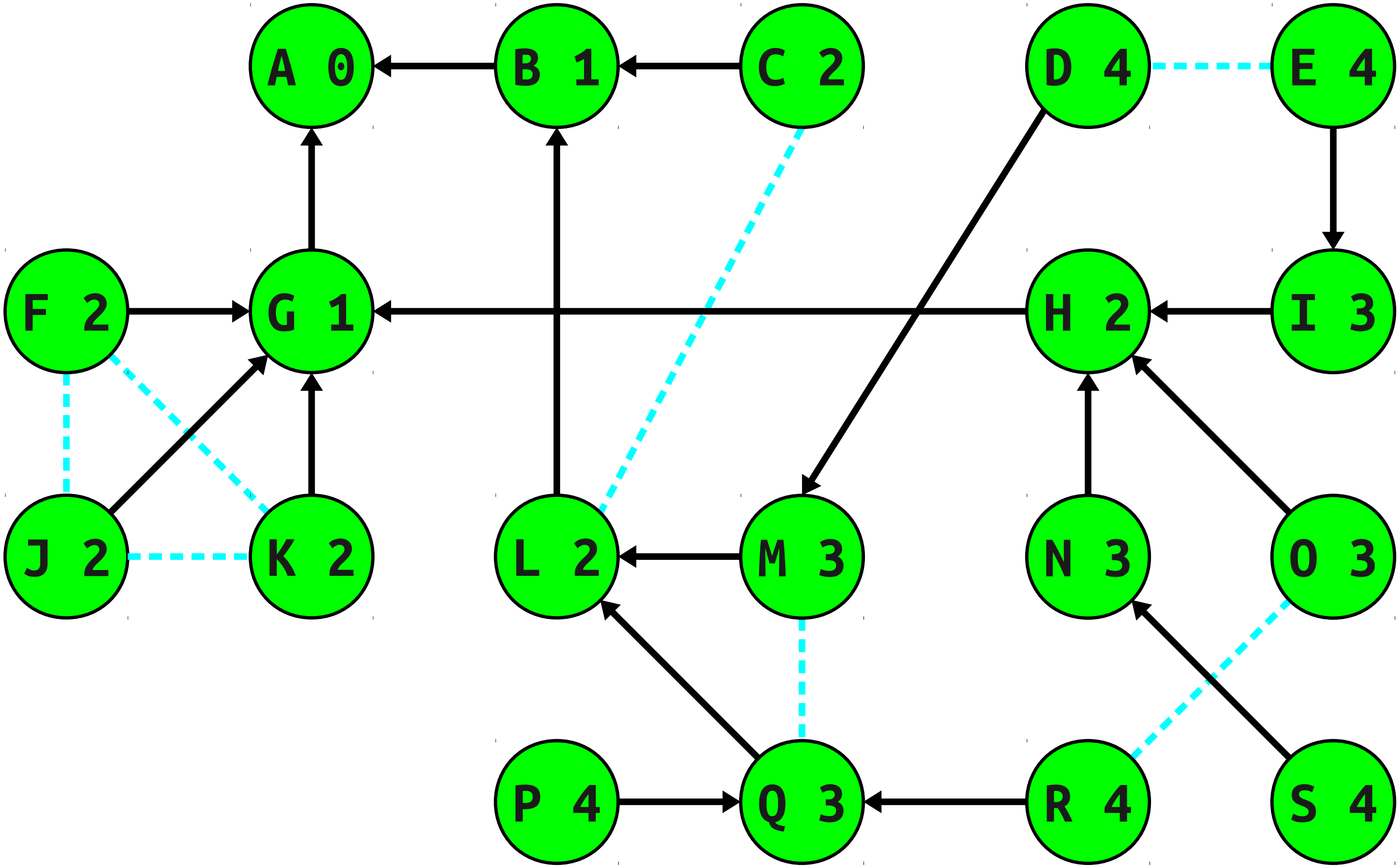
Breadth-First Search



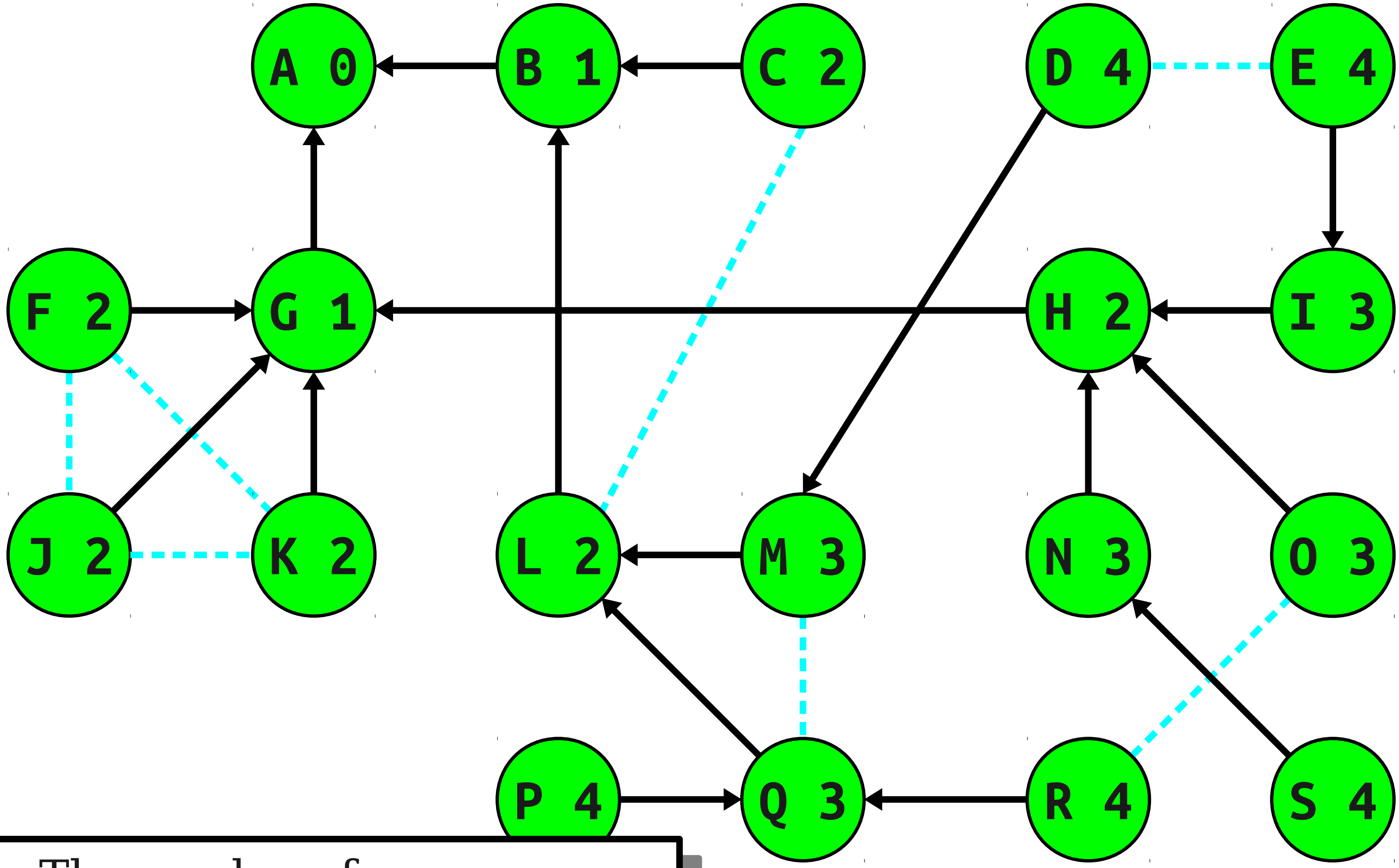
Breadth-First Search



Breadth-First Search



Breadth-First Search



These edges form a **breadth-first search tree**: the path from any *v* to node *A* gives a shortest path from *v* to *A*.

```
procedure breadthFirstSearch(s, G):
   let q be a new queue.
   for each node v in G:
     dist[v] = ∞

   dist[s] = 0
   enqueue(s, q)

   while q is not empty:
      let v = dequeue(q)
      for each neighbor u of v:
         if dist[u] = ∞:
            dist[u] = dist[v] + 1
            enqueue(u, q)
```

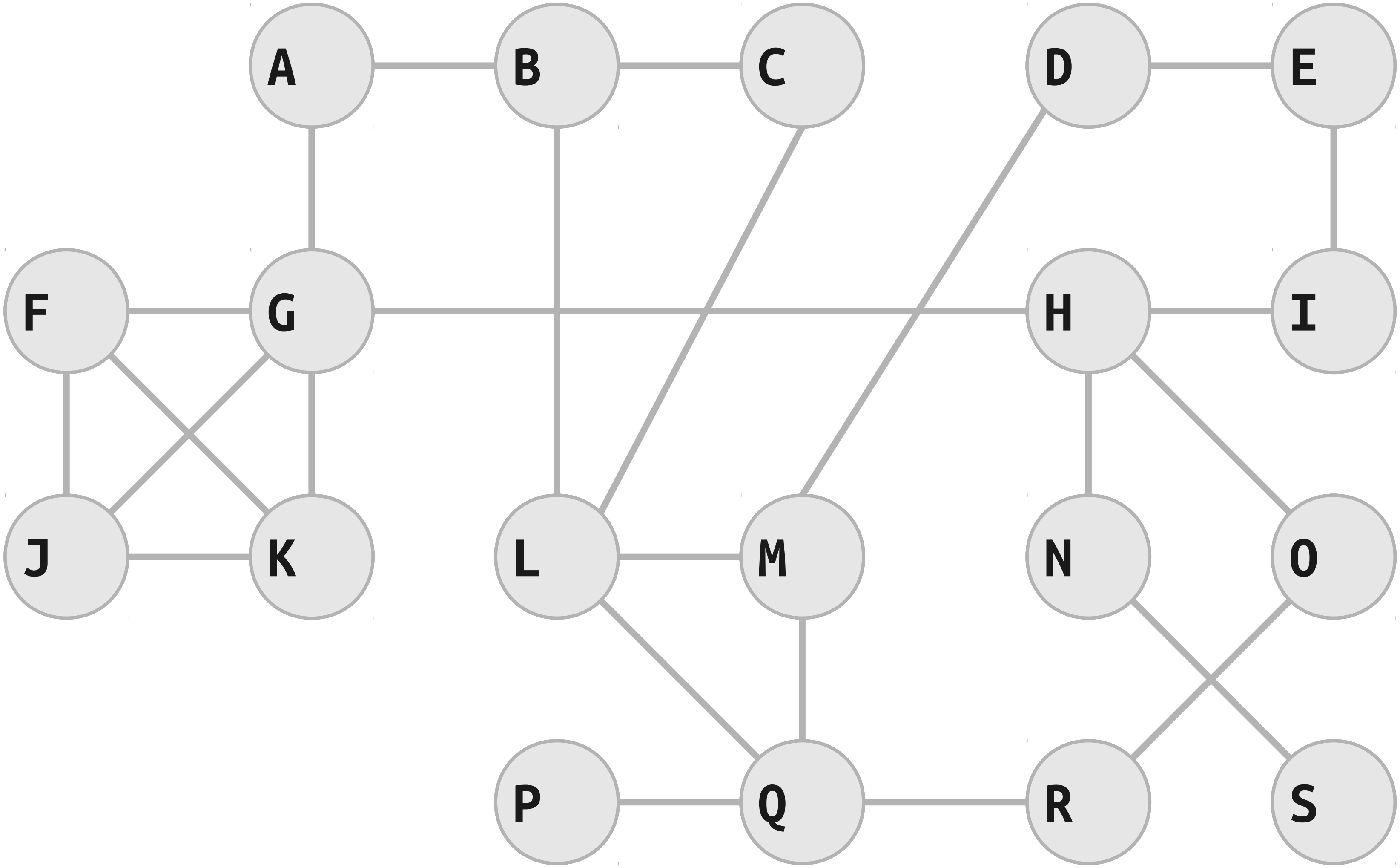
Question 1: How do we prove this always finds the right distances?

Question 2: How *efficiently* does this find the right distances?

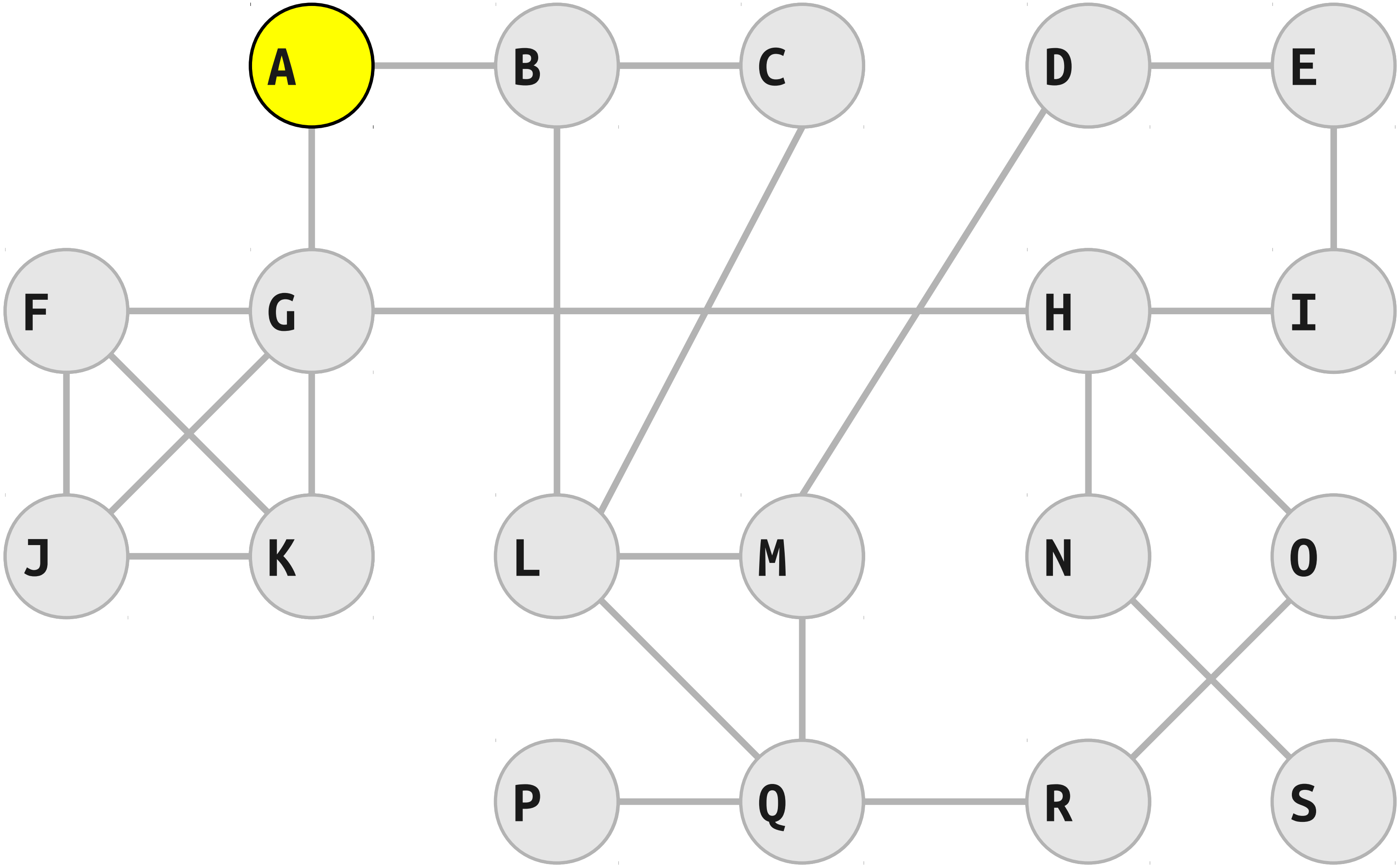
Question 1: How do we prove this always finds the right distances?

Question 2: How *efficiently* does this find the right distances?

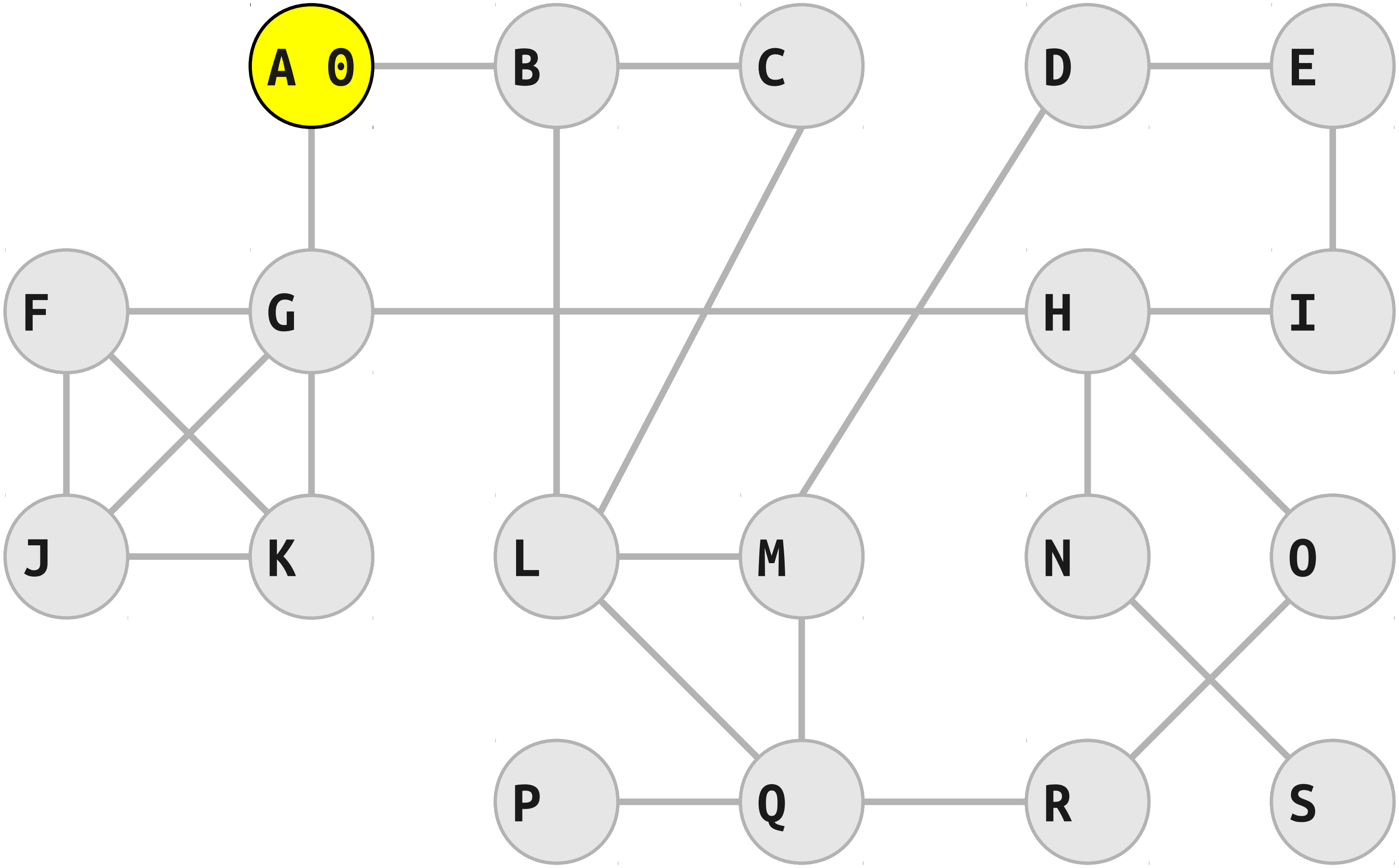
Breadth-First Search



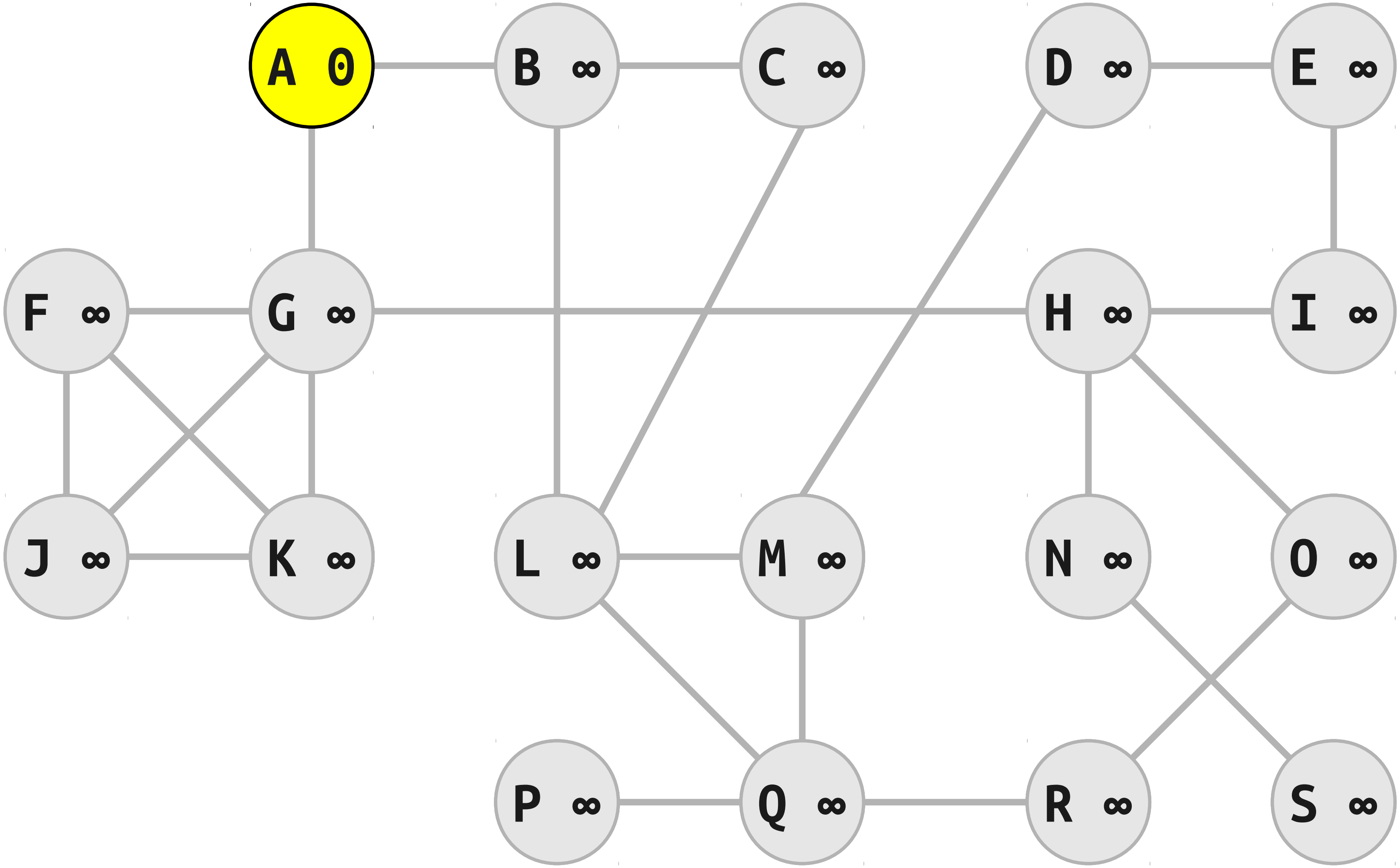
Breadth-First Search



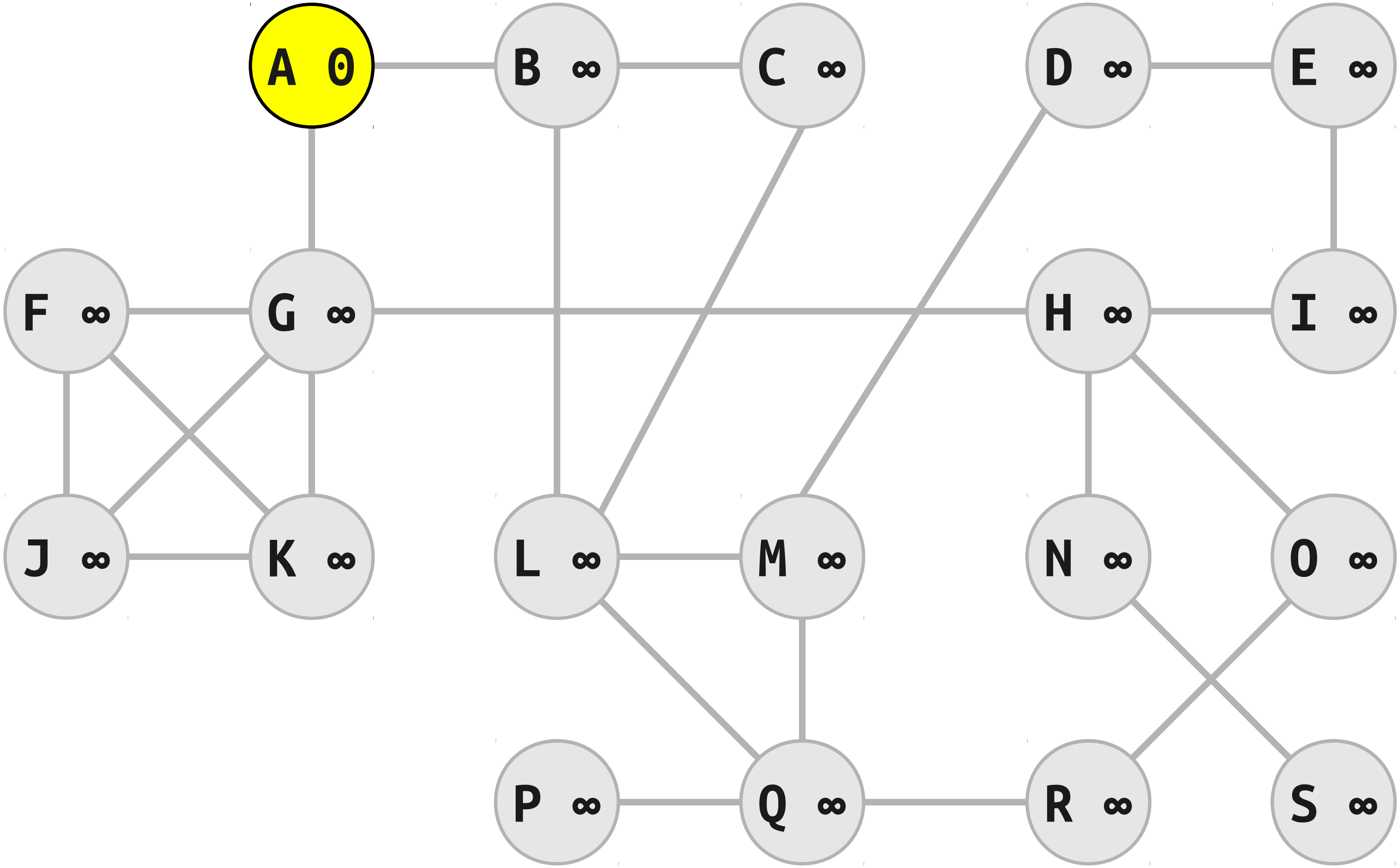
Breadth-First Search



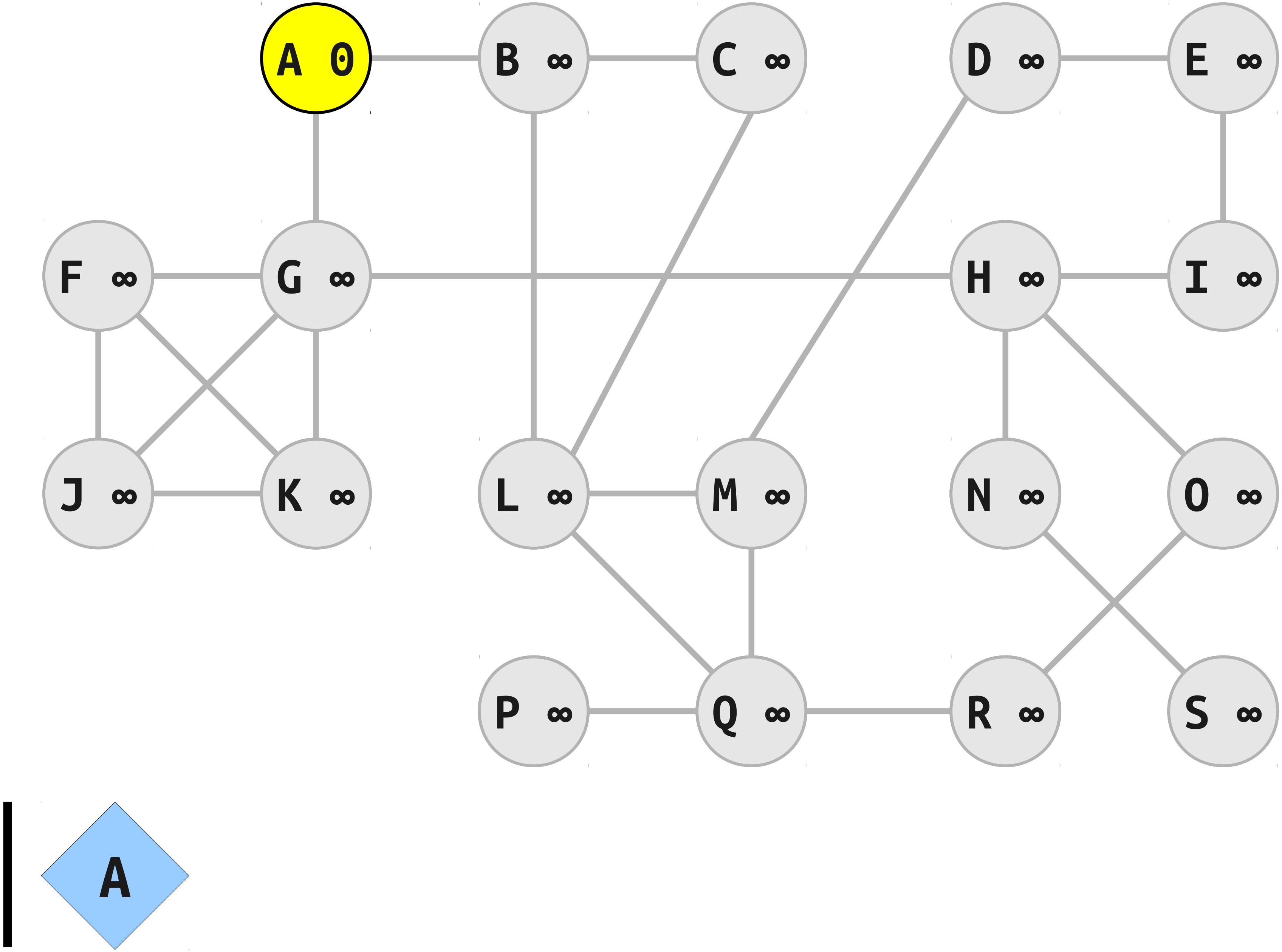
Breadth-First Search



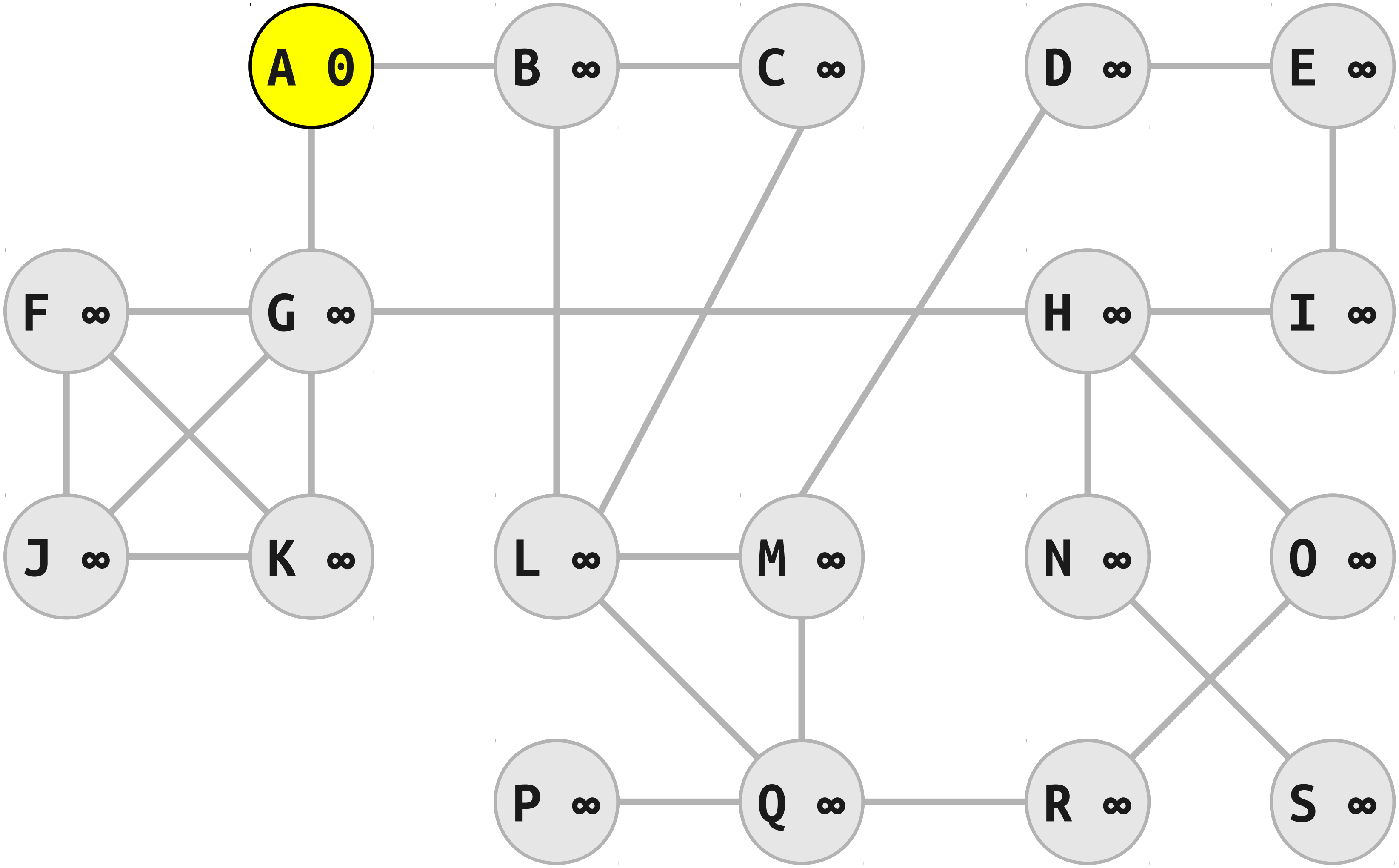
Breadth-First Search

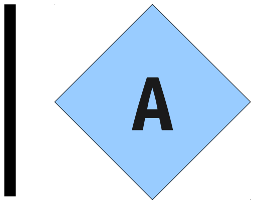


Breadth-First Search



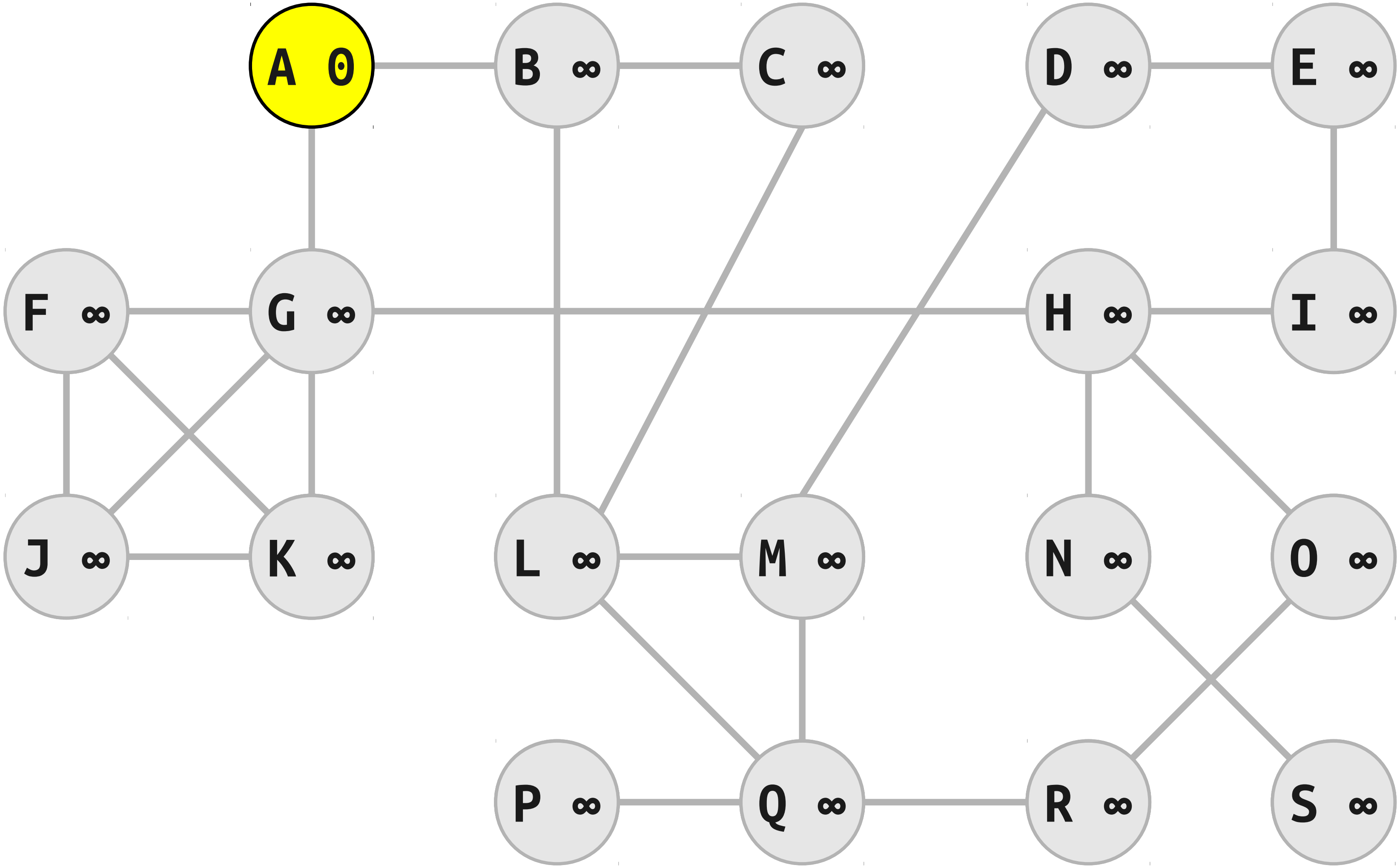
Breadth-First Search

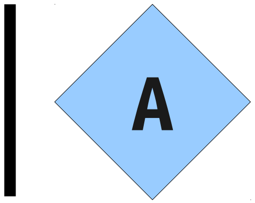




All nodes in the queue are at distance **0** from A.

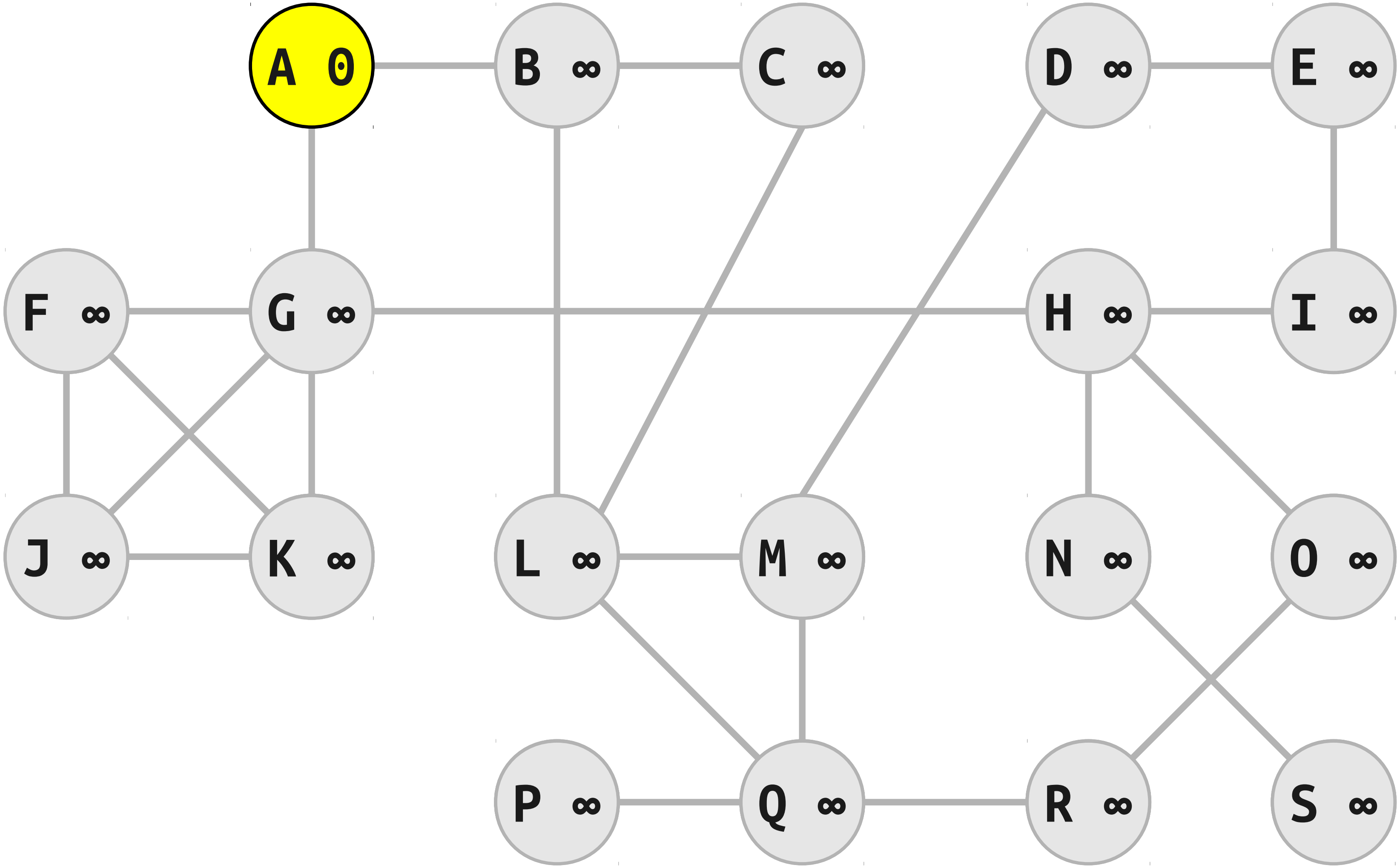
Breadth-First Search

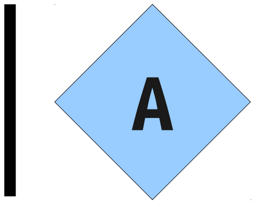




All nodes at distance **0** from *A* are in the queue.

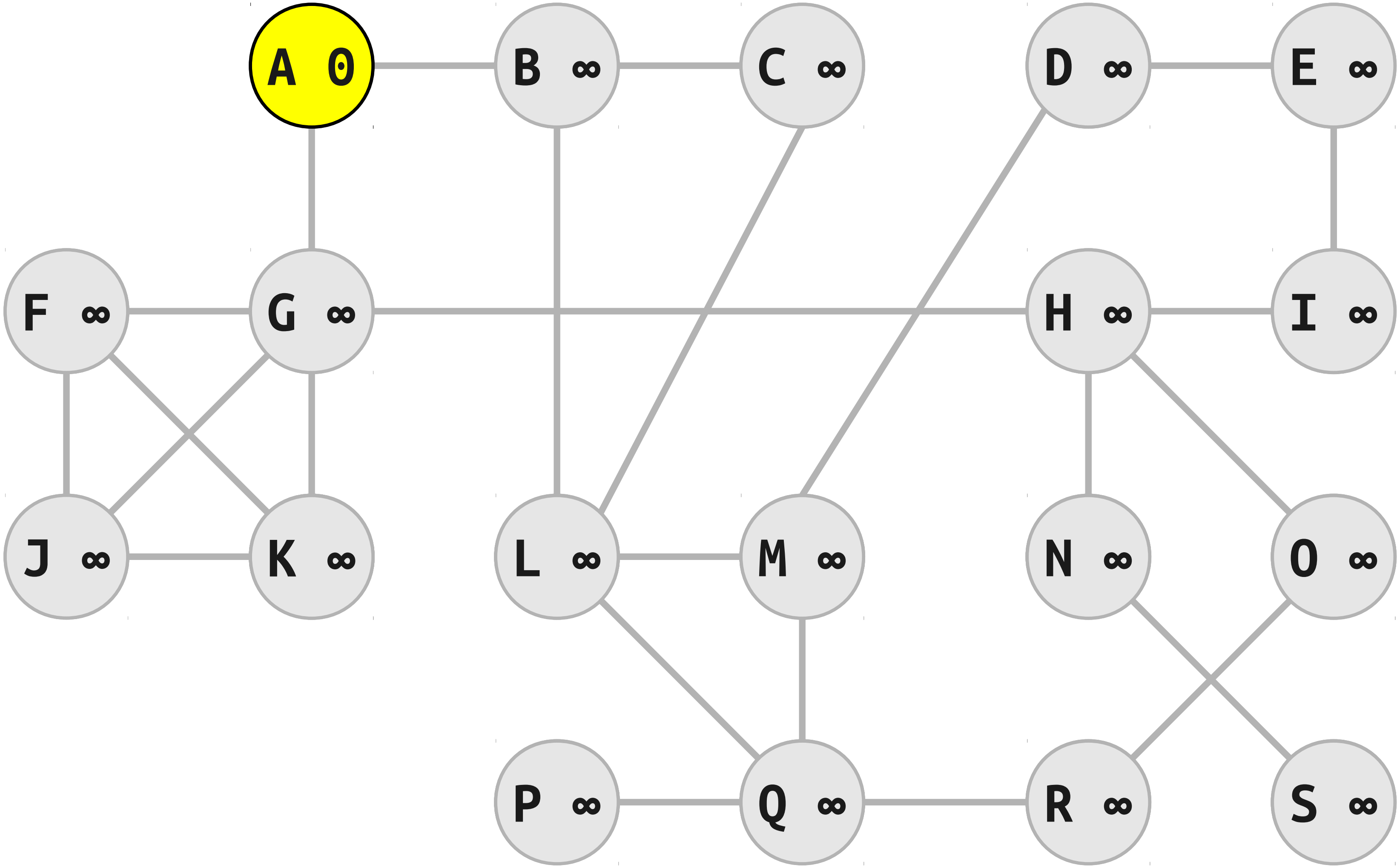
Breadth-First Search

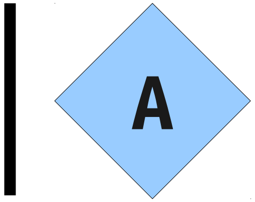




All nodes at distance ≤ **0** from *A* have the right distance set.

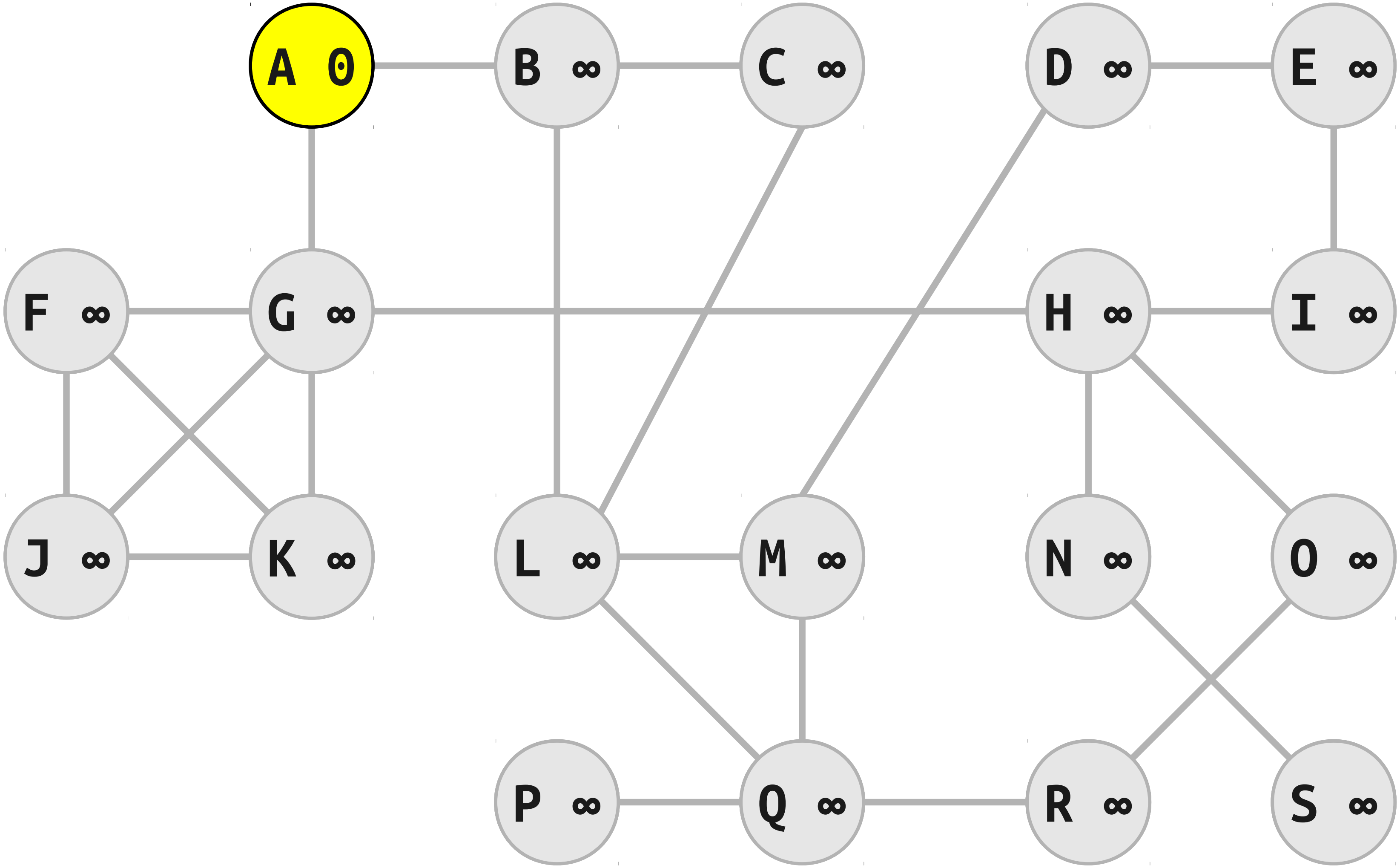
Breadth-First Search

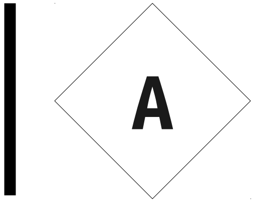




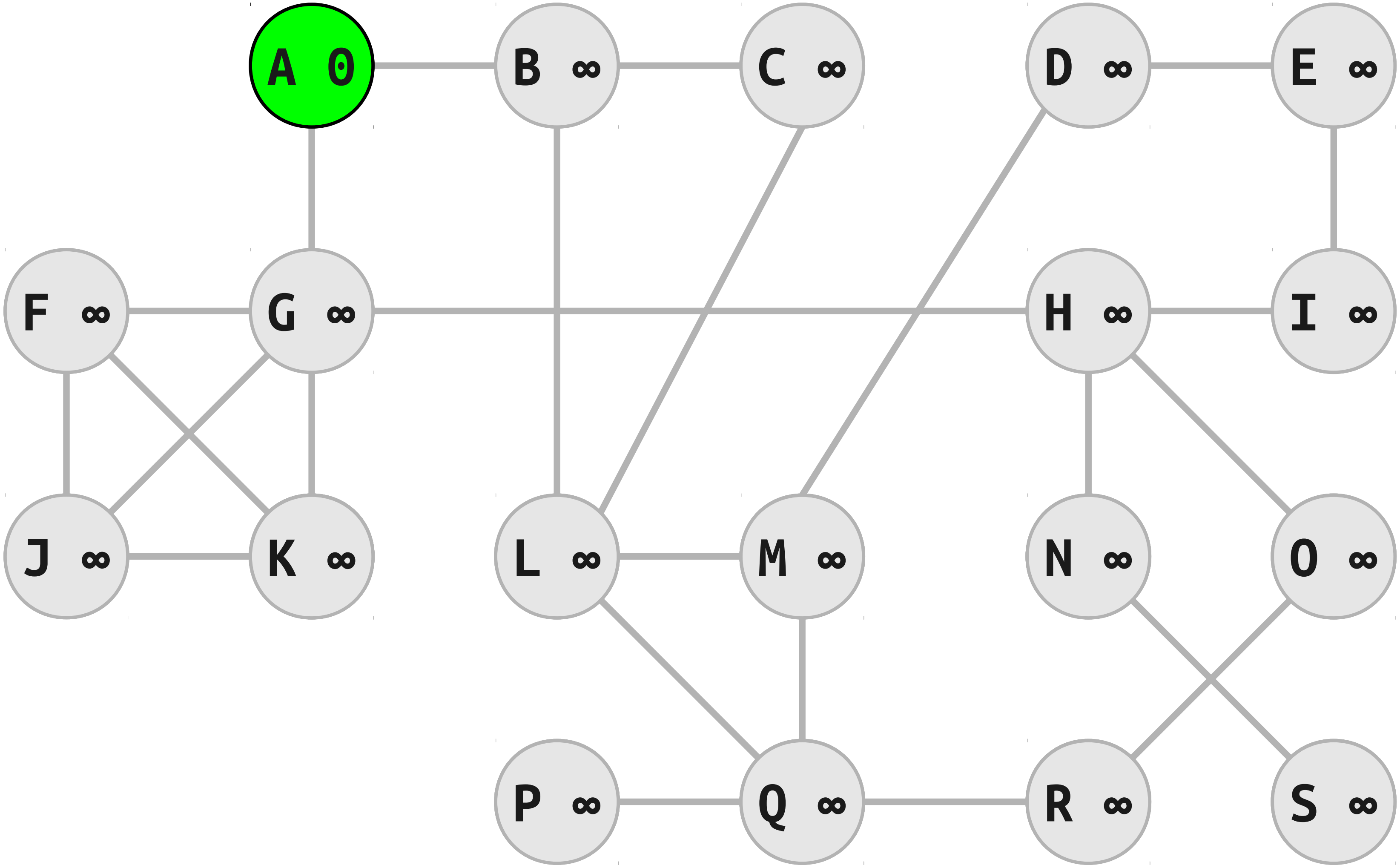
All nodes at distance > **0** from *A* have distance set to ∞

Breadth-First Search

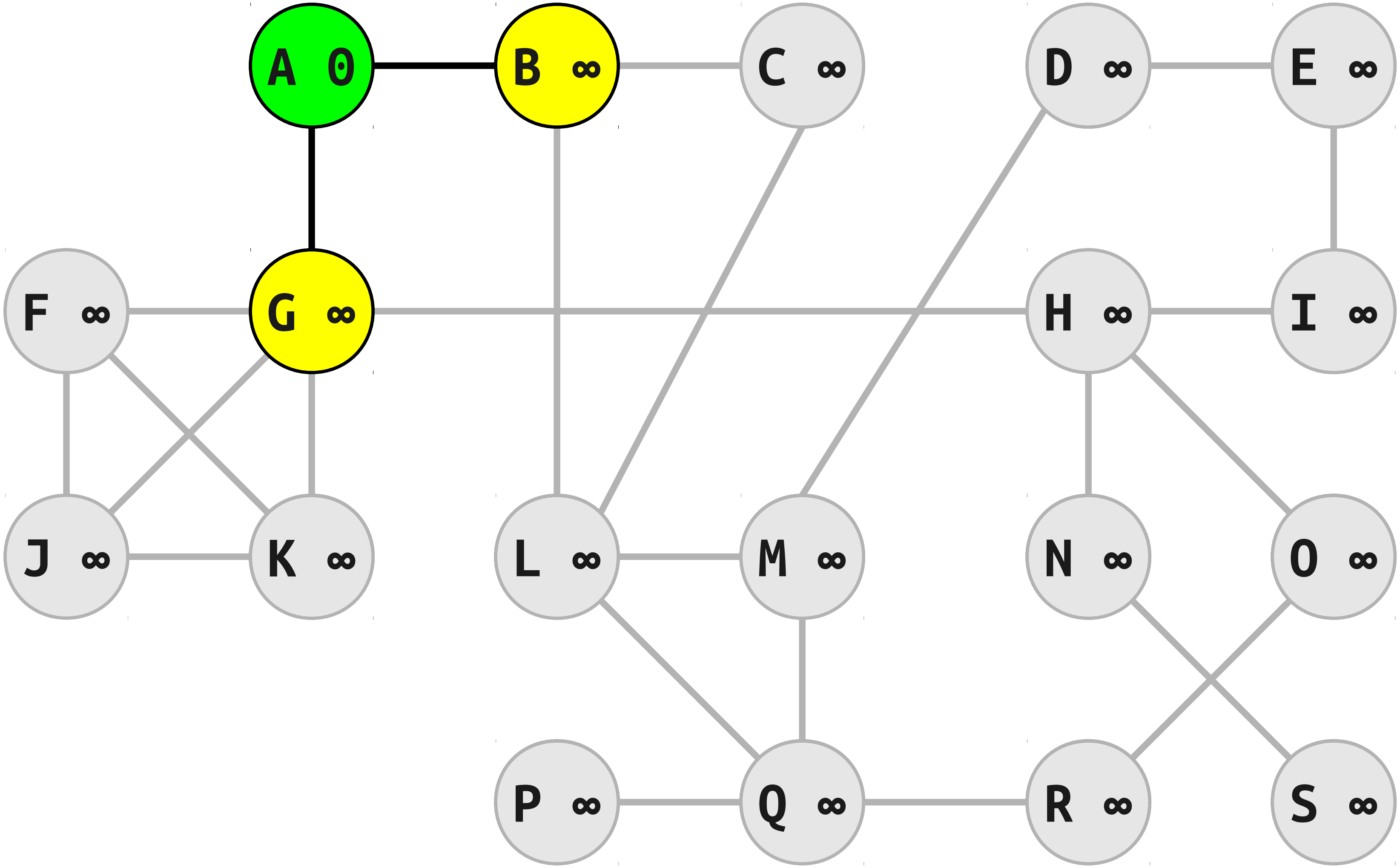




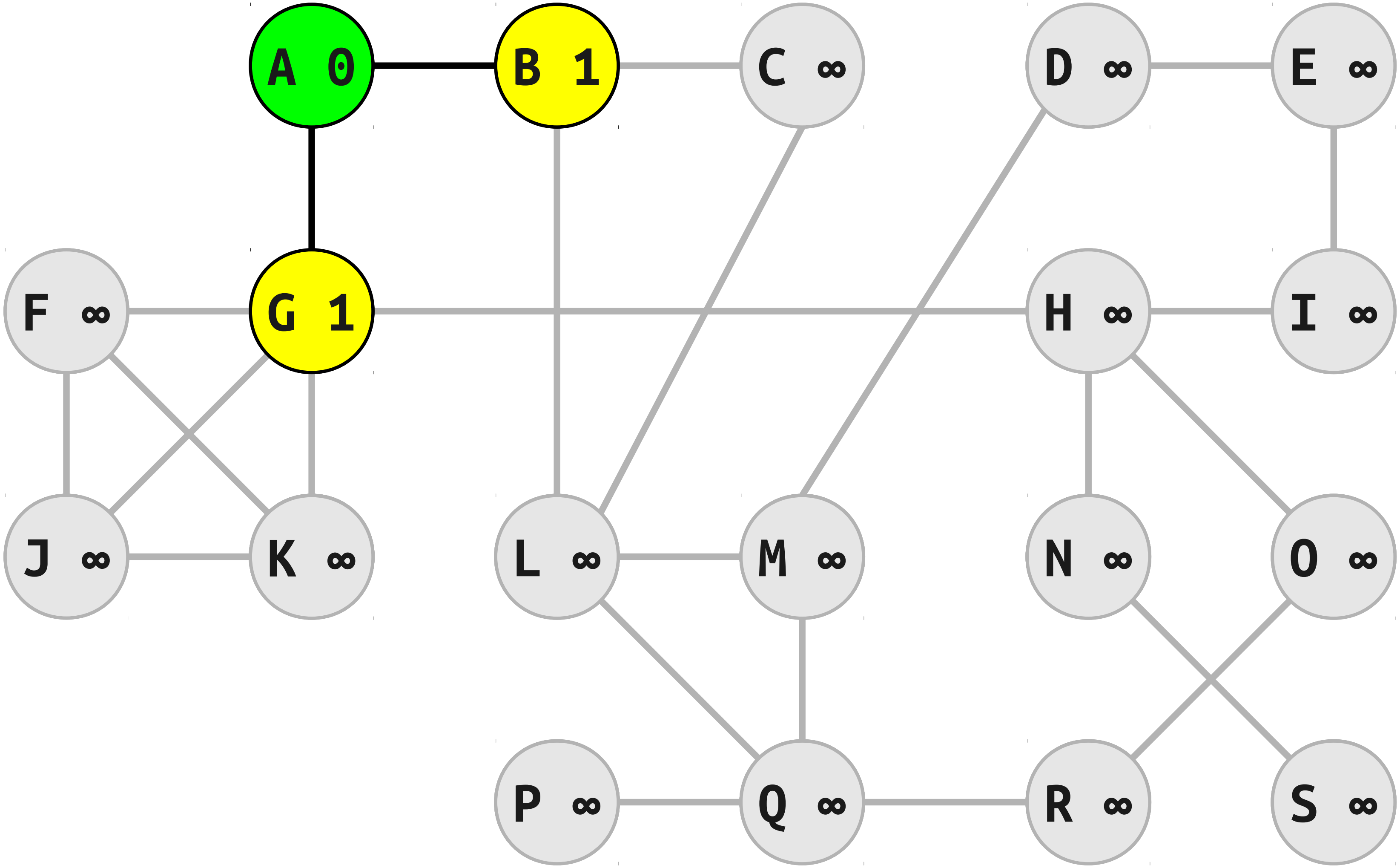
Breadth-First Search



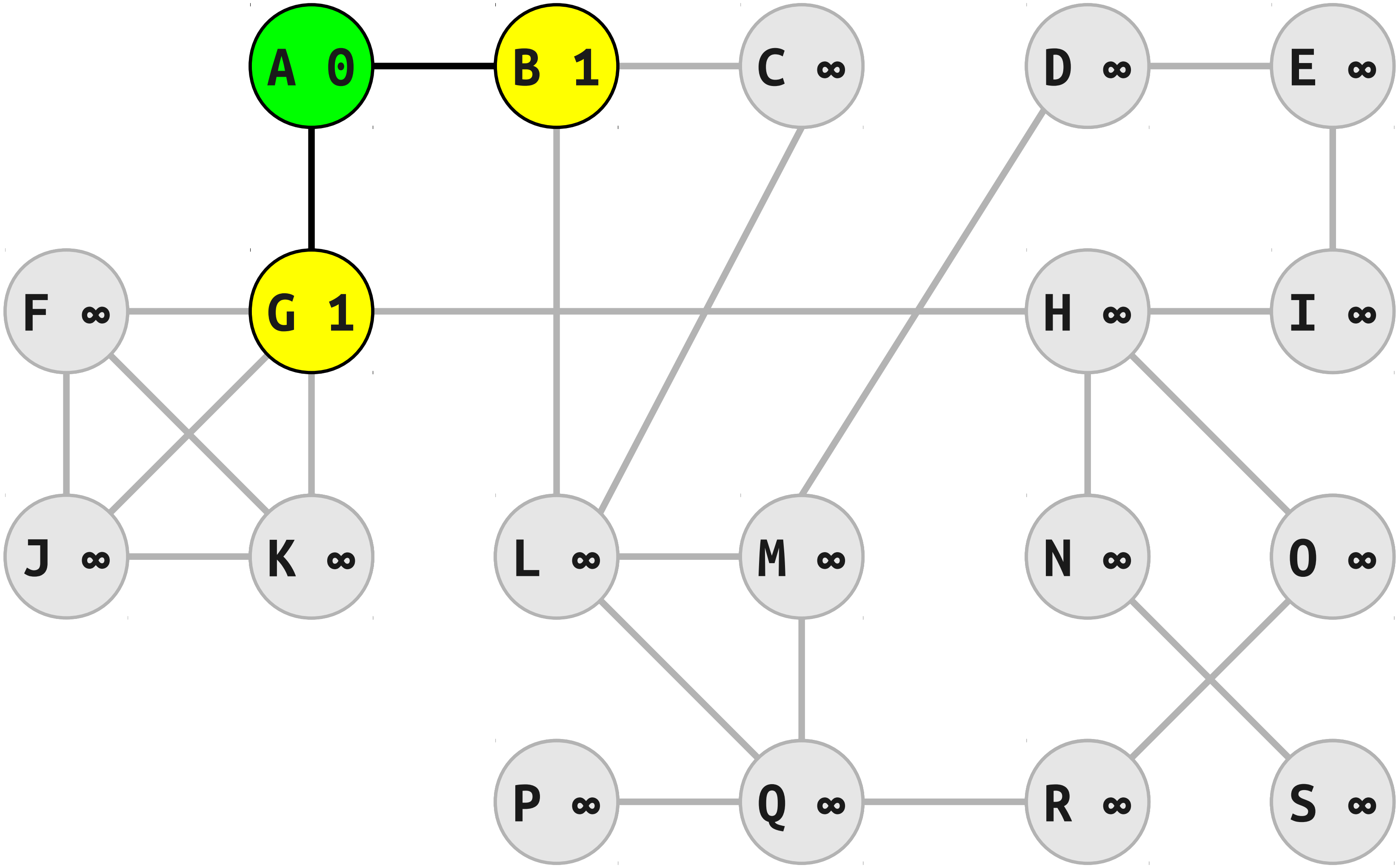
Breadth-First Search

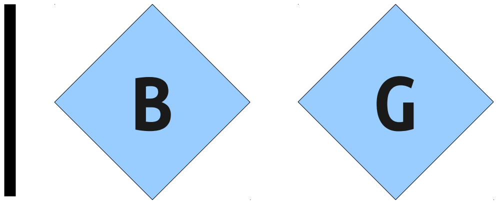


Breadth-First Search

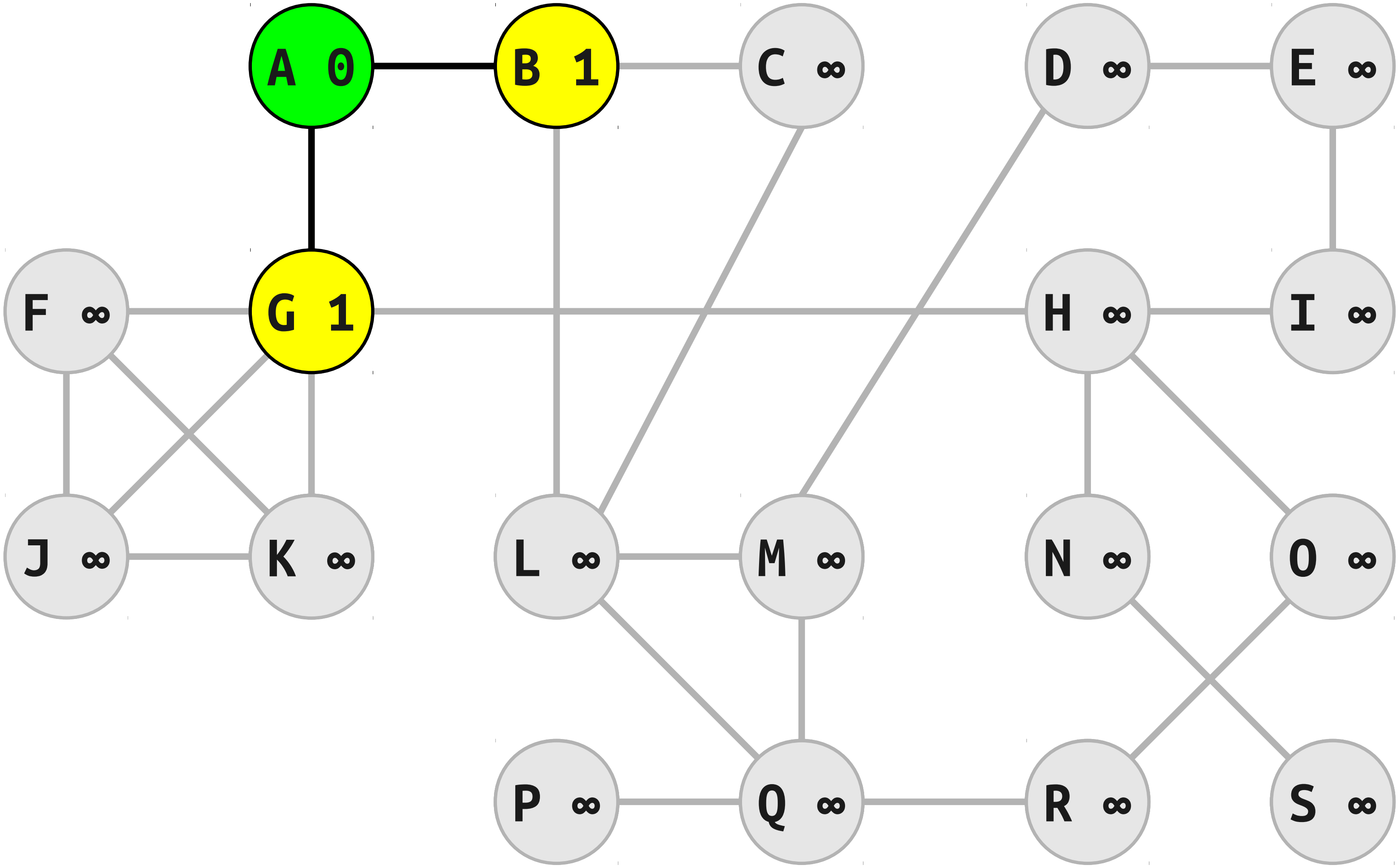


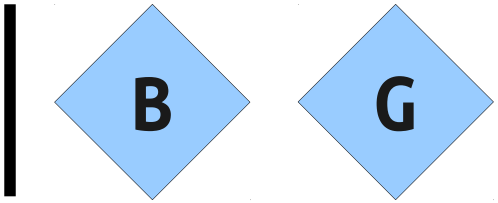
Breadth-First Search





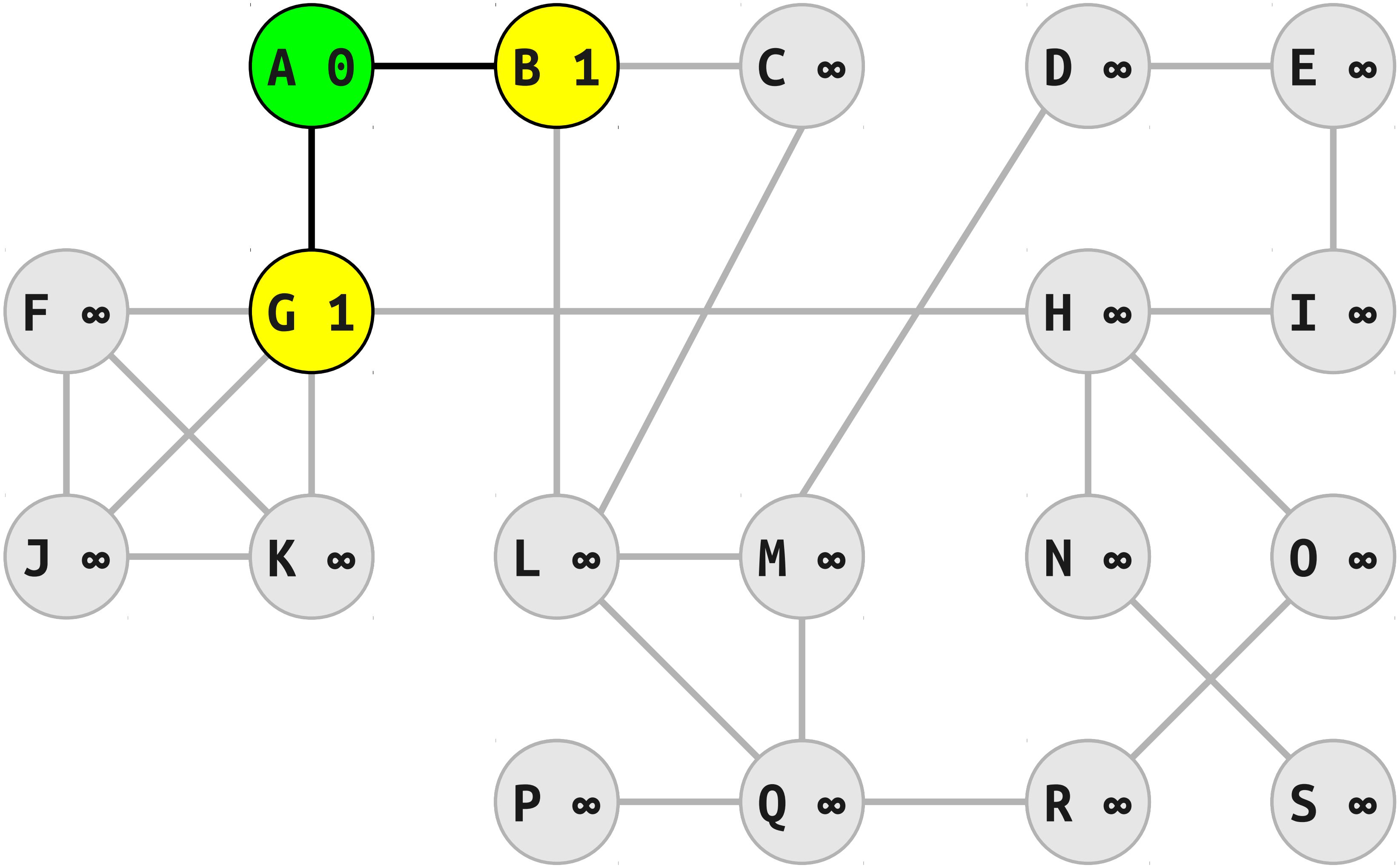
Breadth-First Search

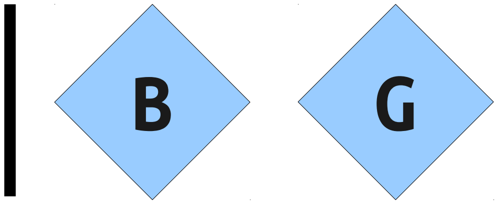




All nodes in the queue are at distance **1** from *A*.

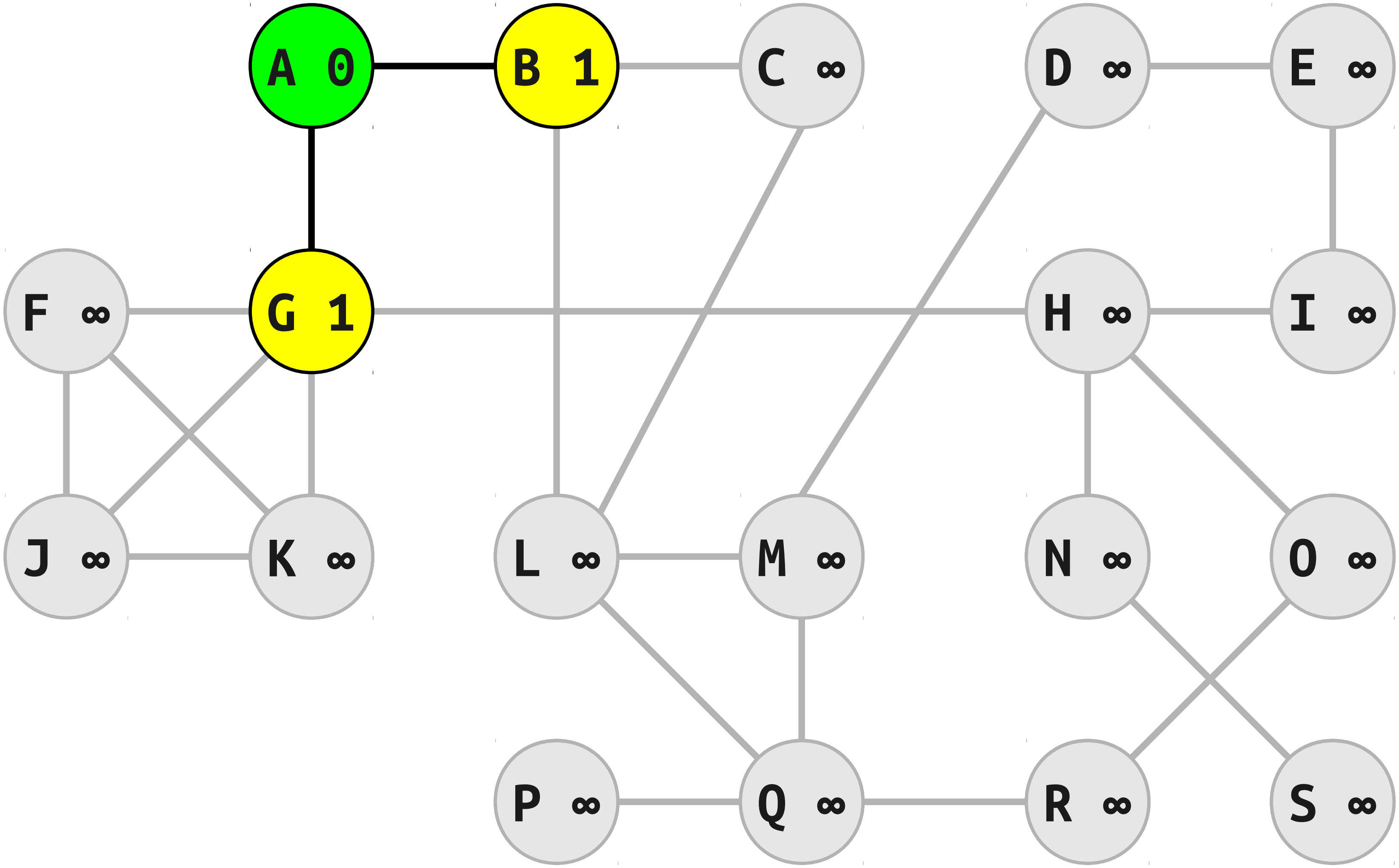
Breadth-First Search

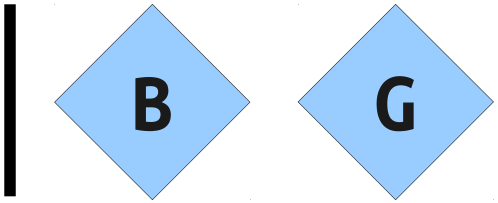




All nodes at distance 1 from *A* are in the queue.

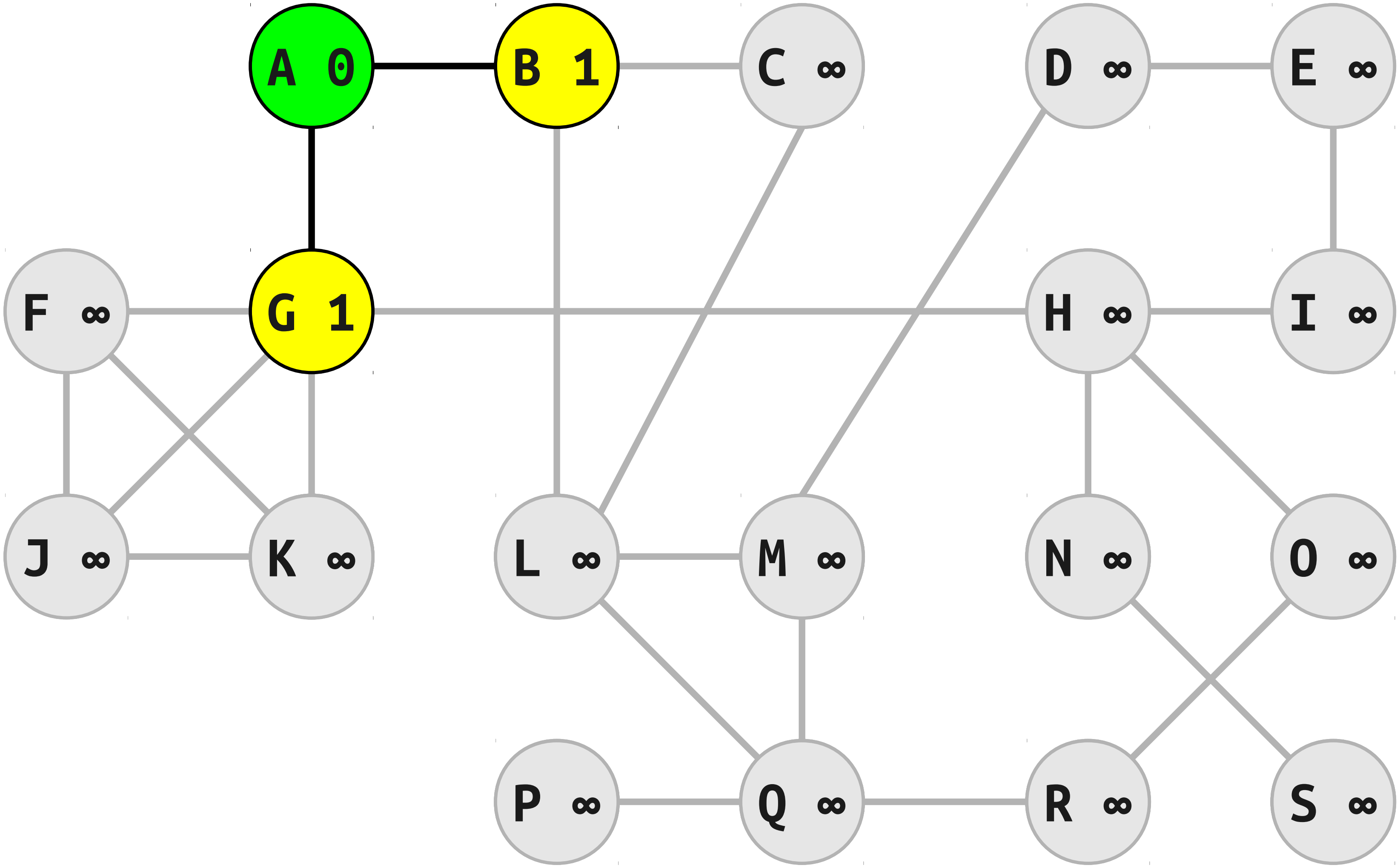
Breadth-First Search

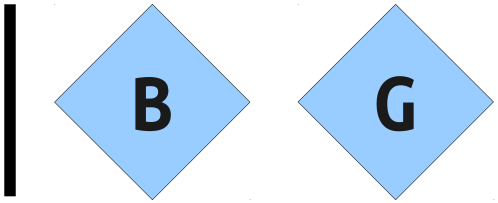




All nodes at distance ≤ 1 from *A* have the right distance set.

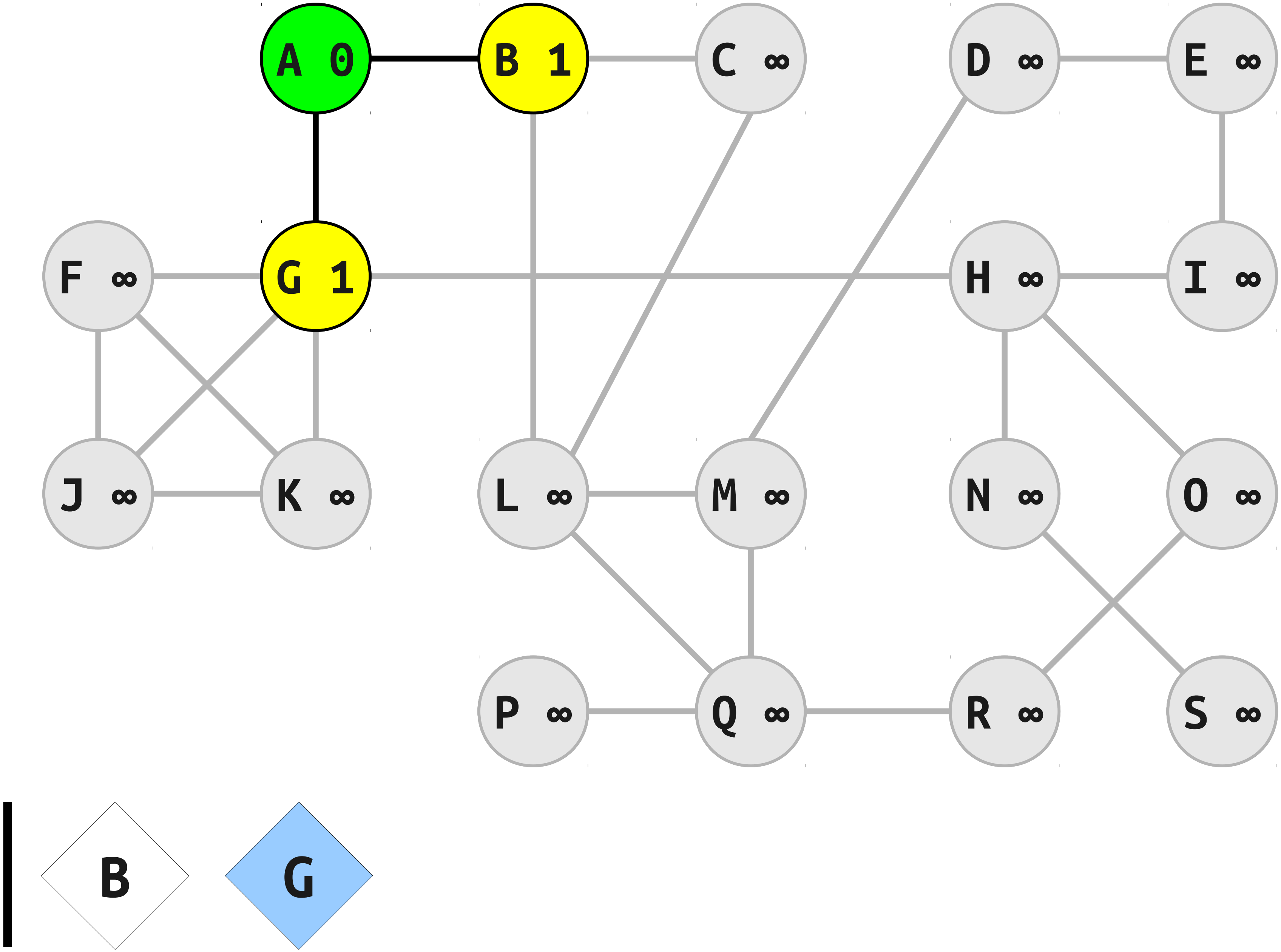
Breadth-First Search



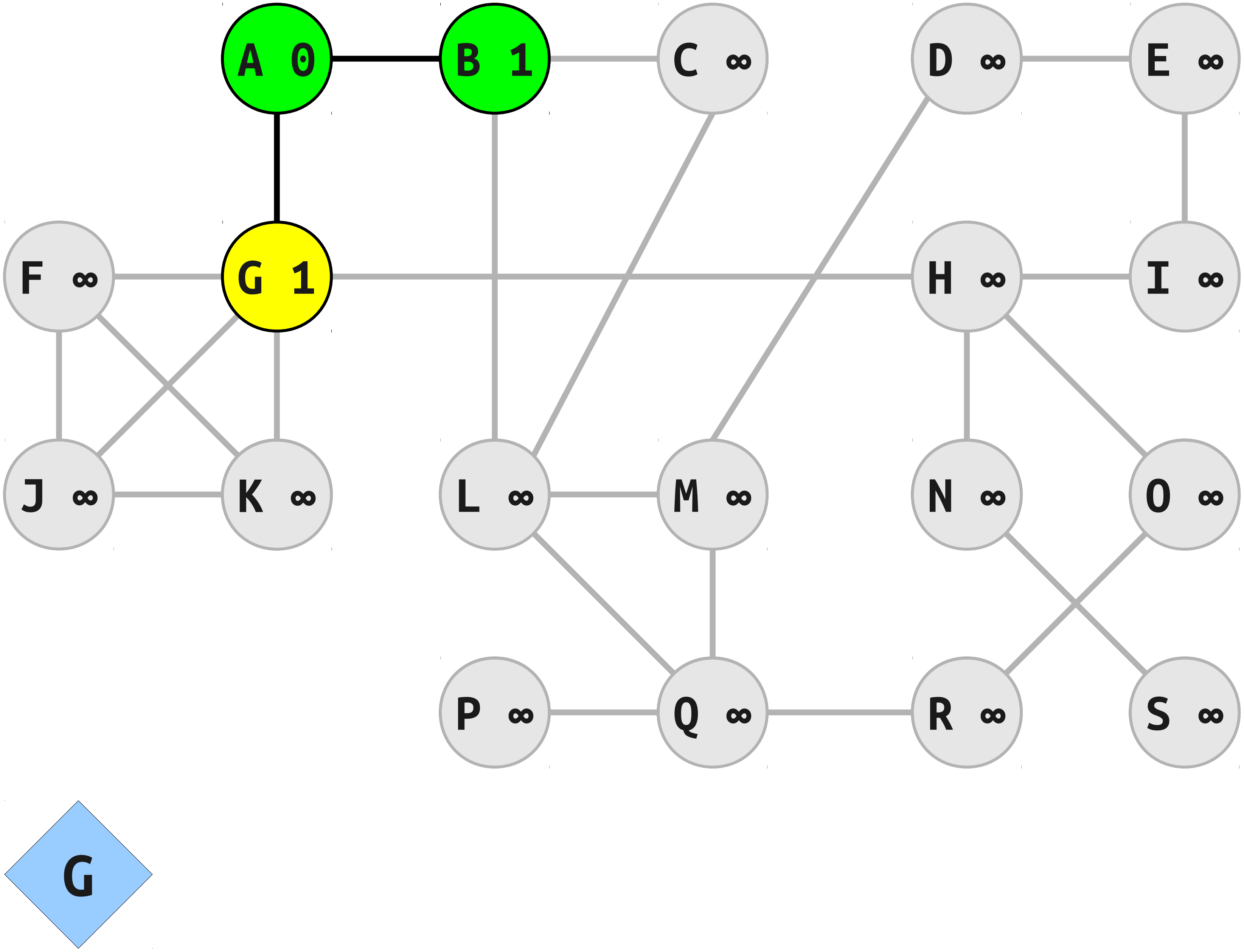


All nodes at distance > 1 from *A* have distance set to ∞

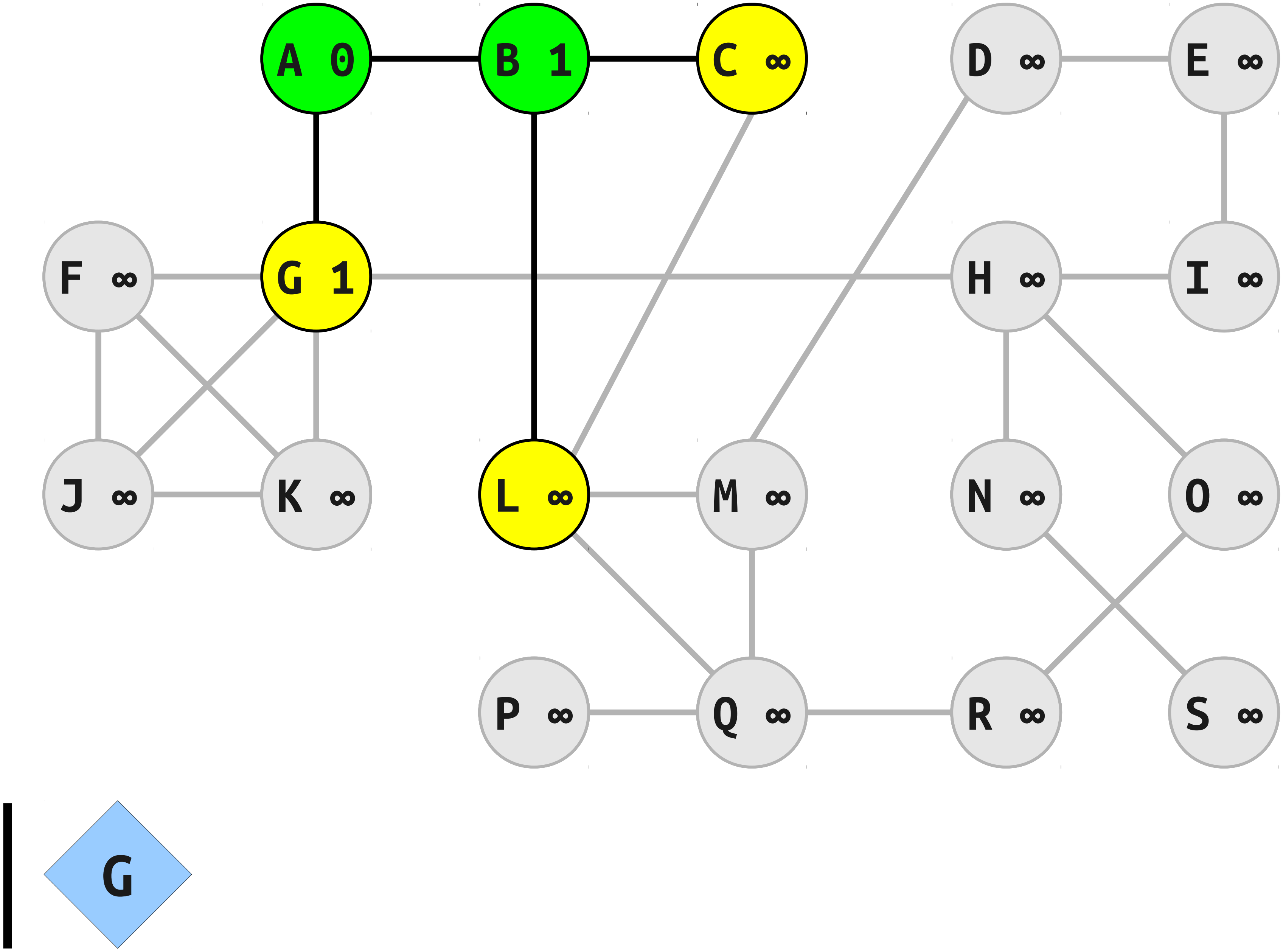
Breadth-First Search



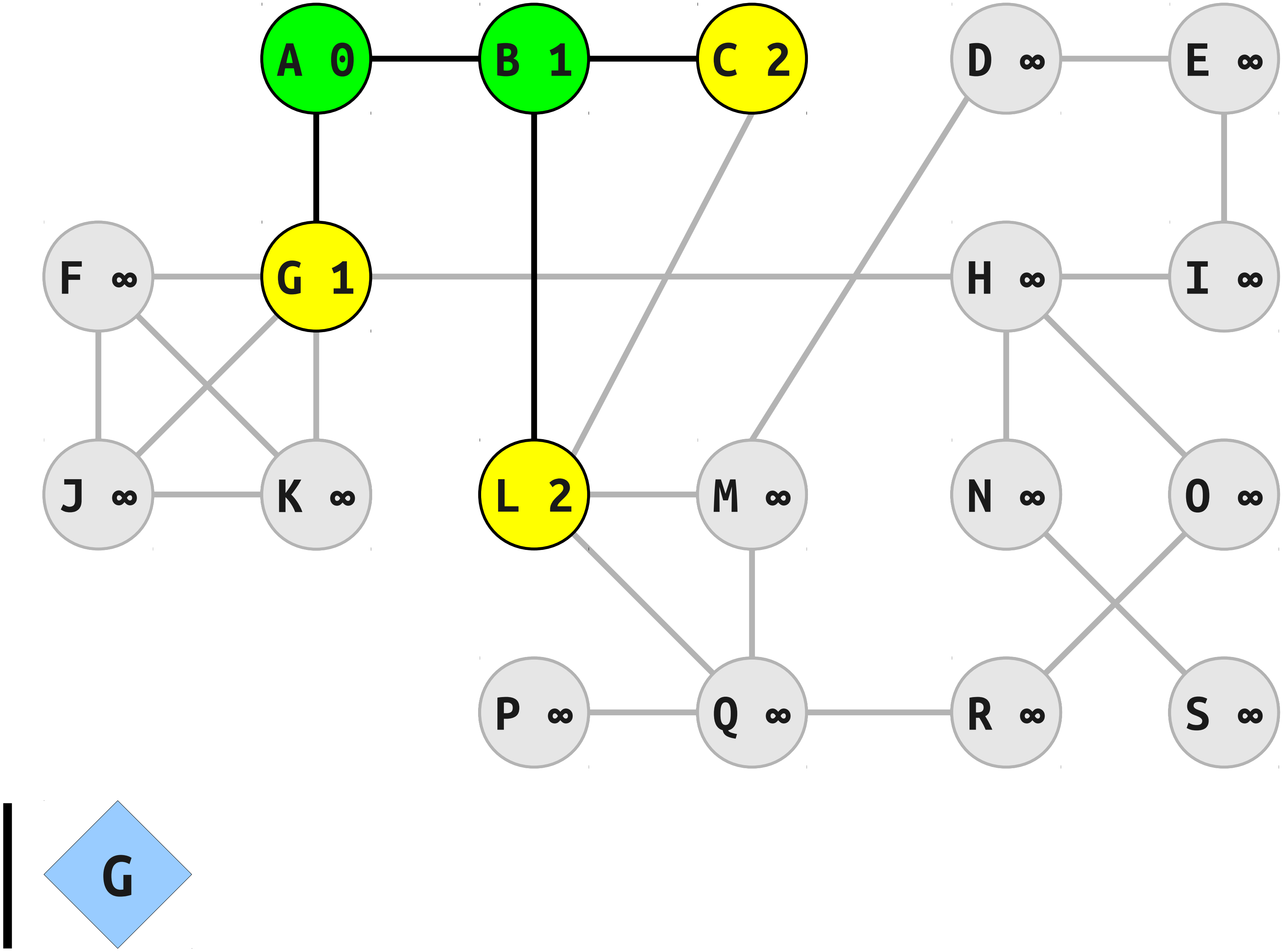
Breadth-First Search



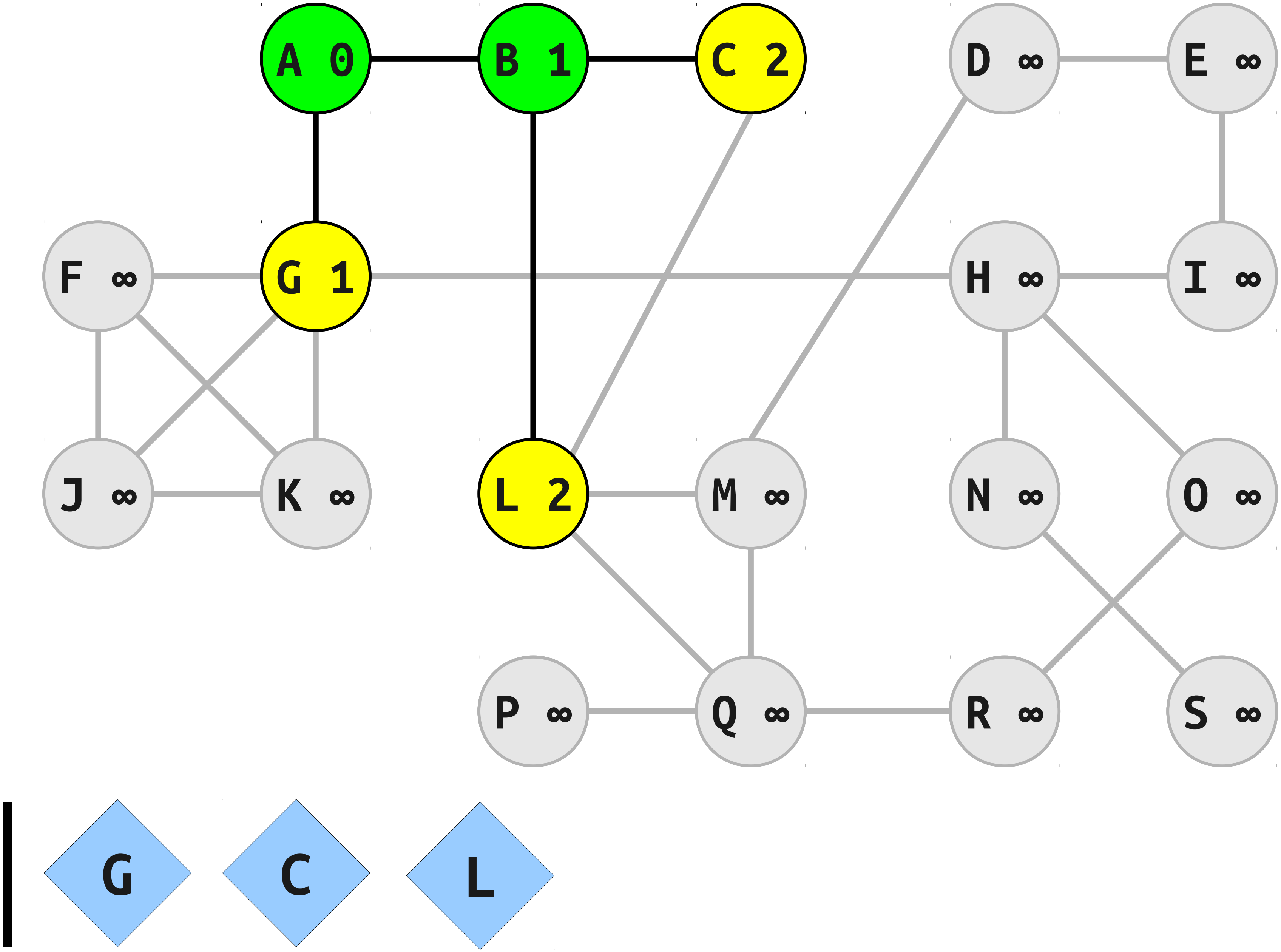
Breadth-First Search



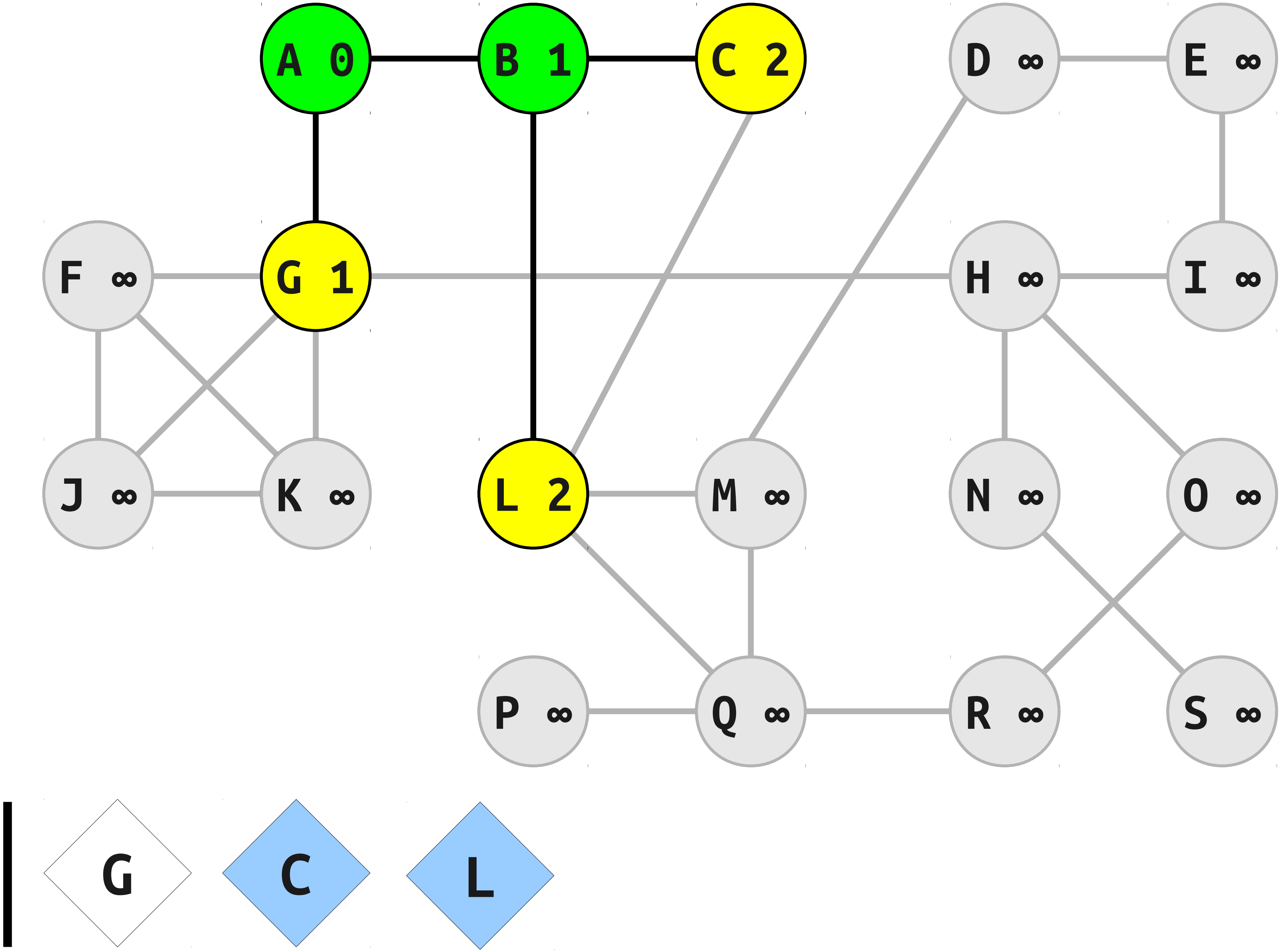
Breadth-First Search



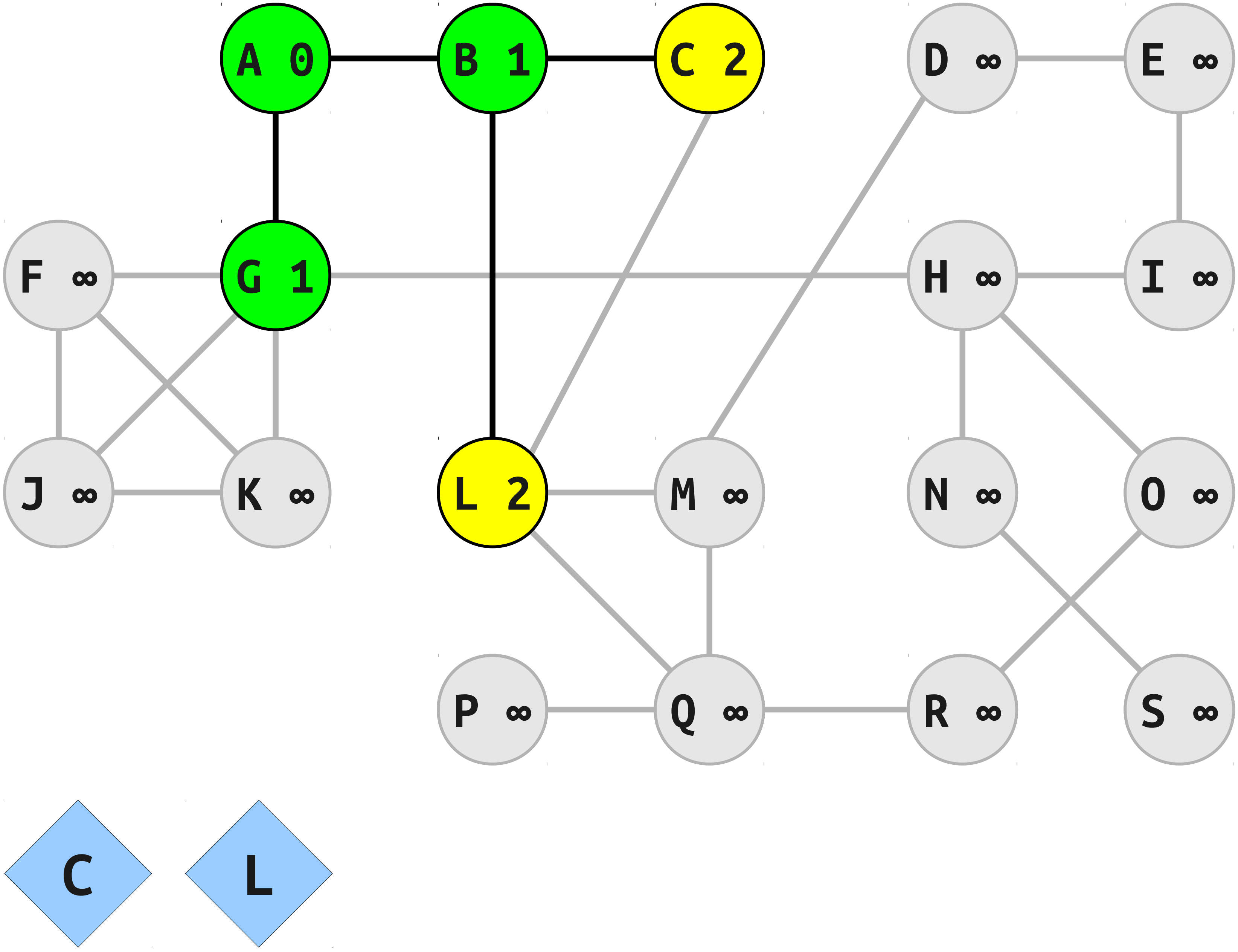
Breadth-First Search



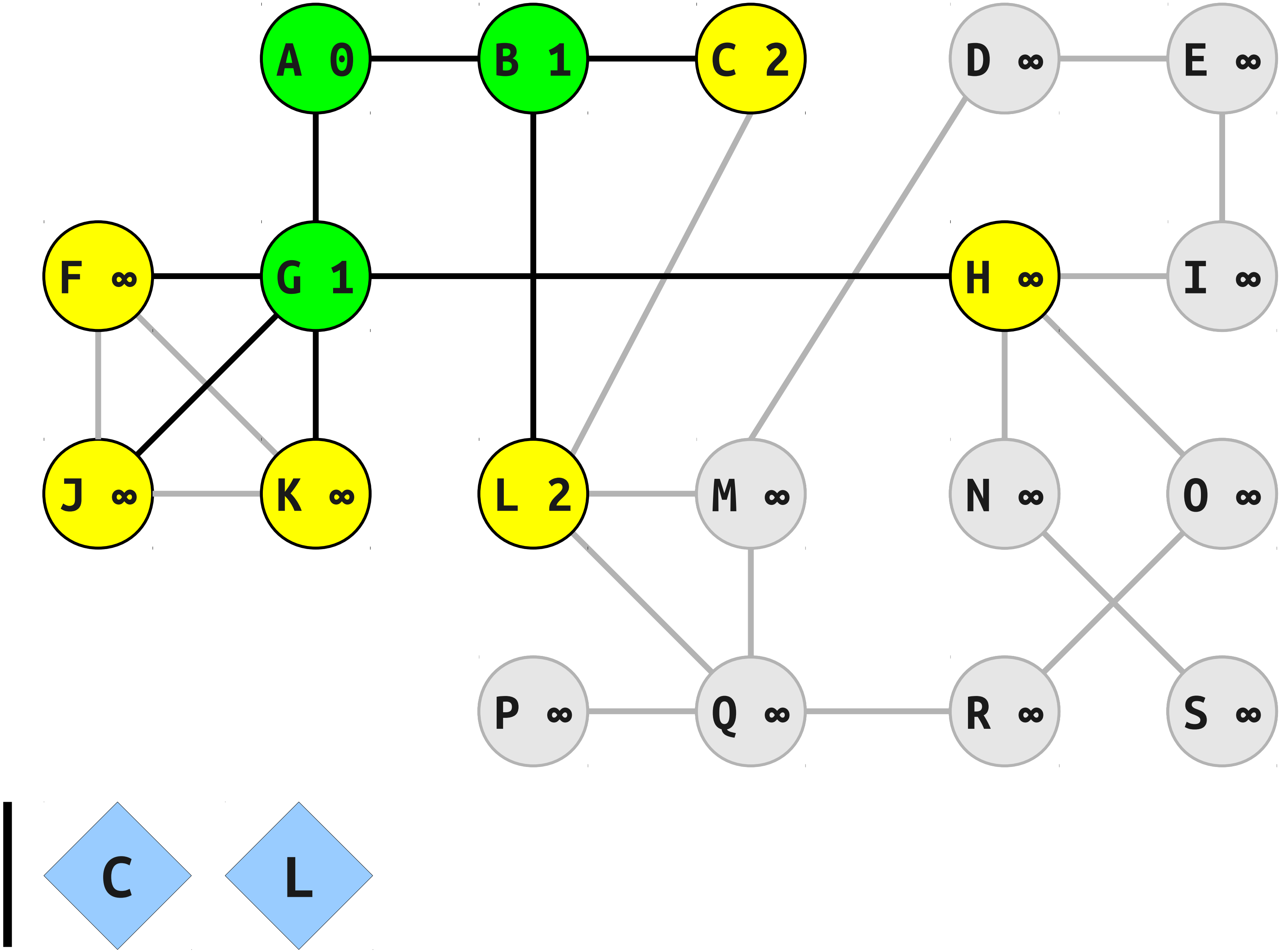
Breadth-First Search



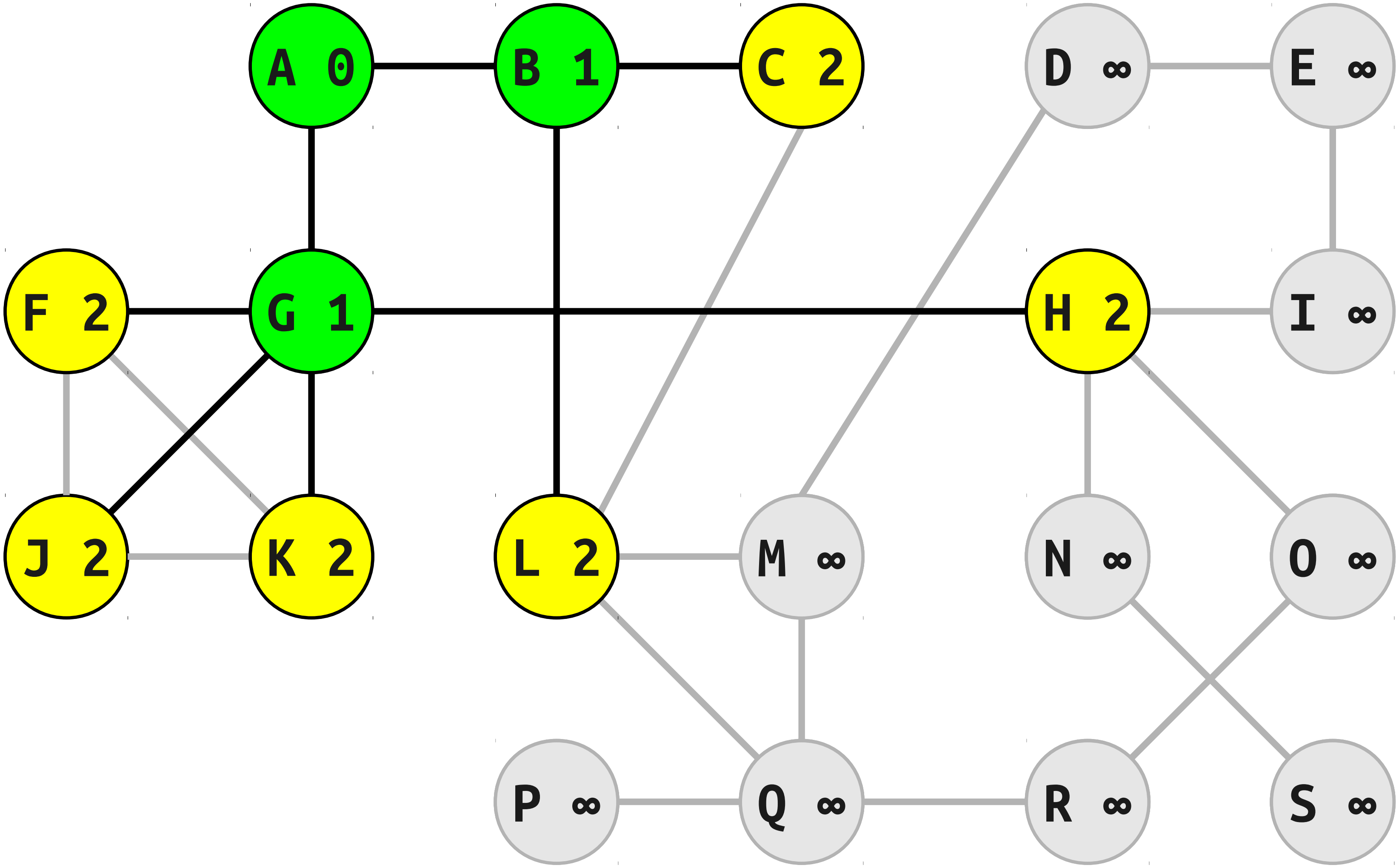
Breadth-First Search

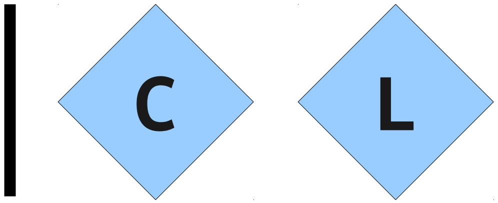


Breadth-First Search

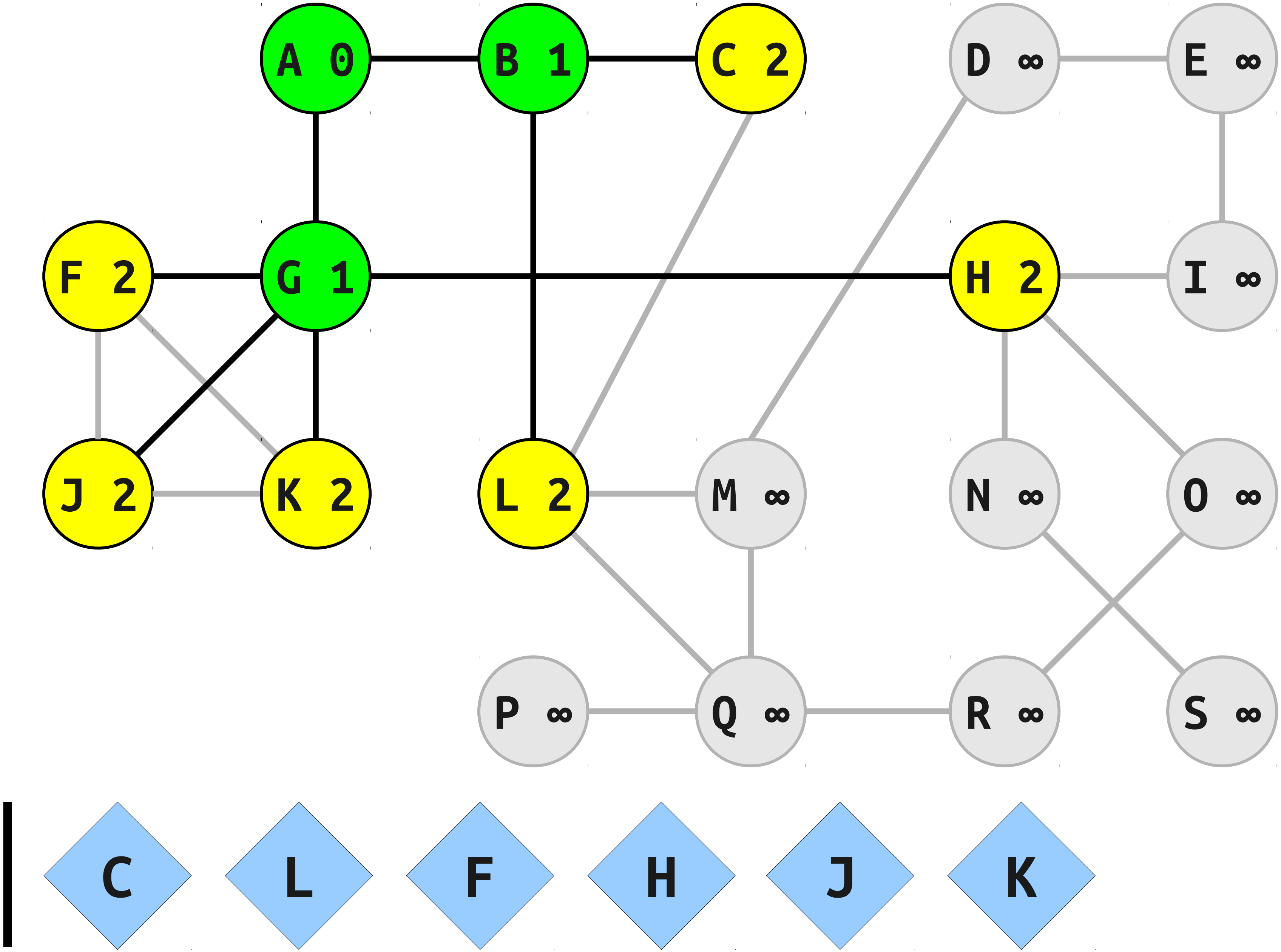


Breadth-First Search

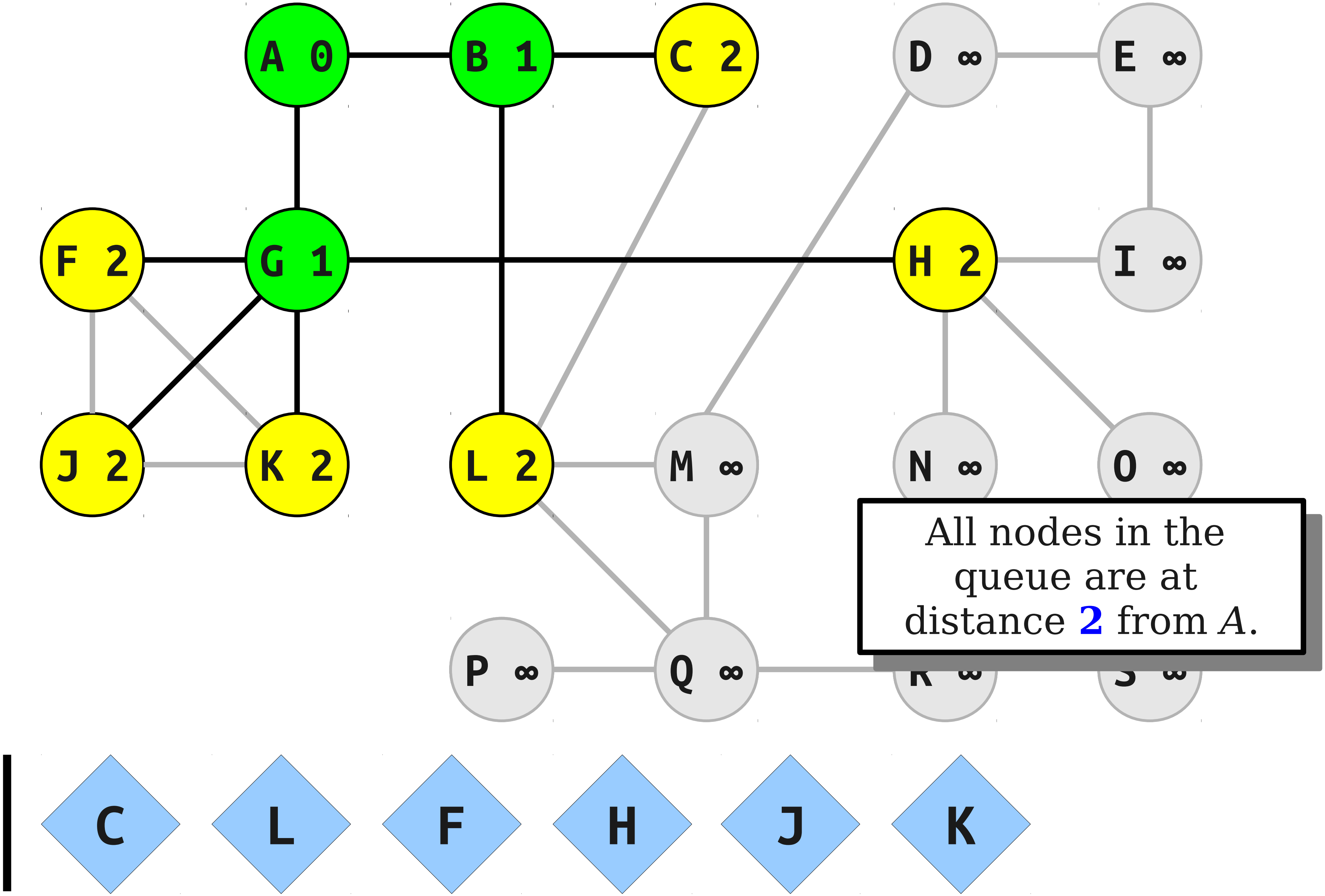




Breadth-First Search

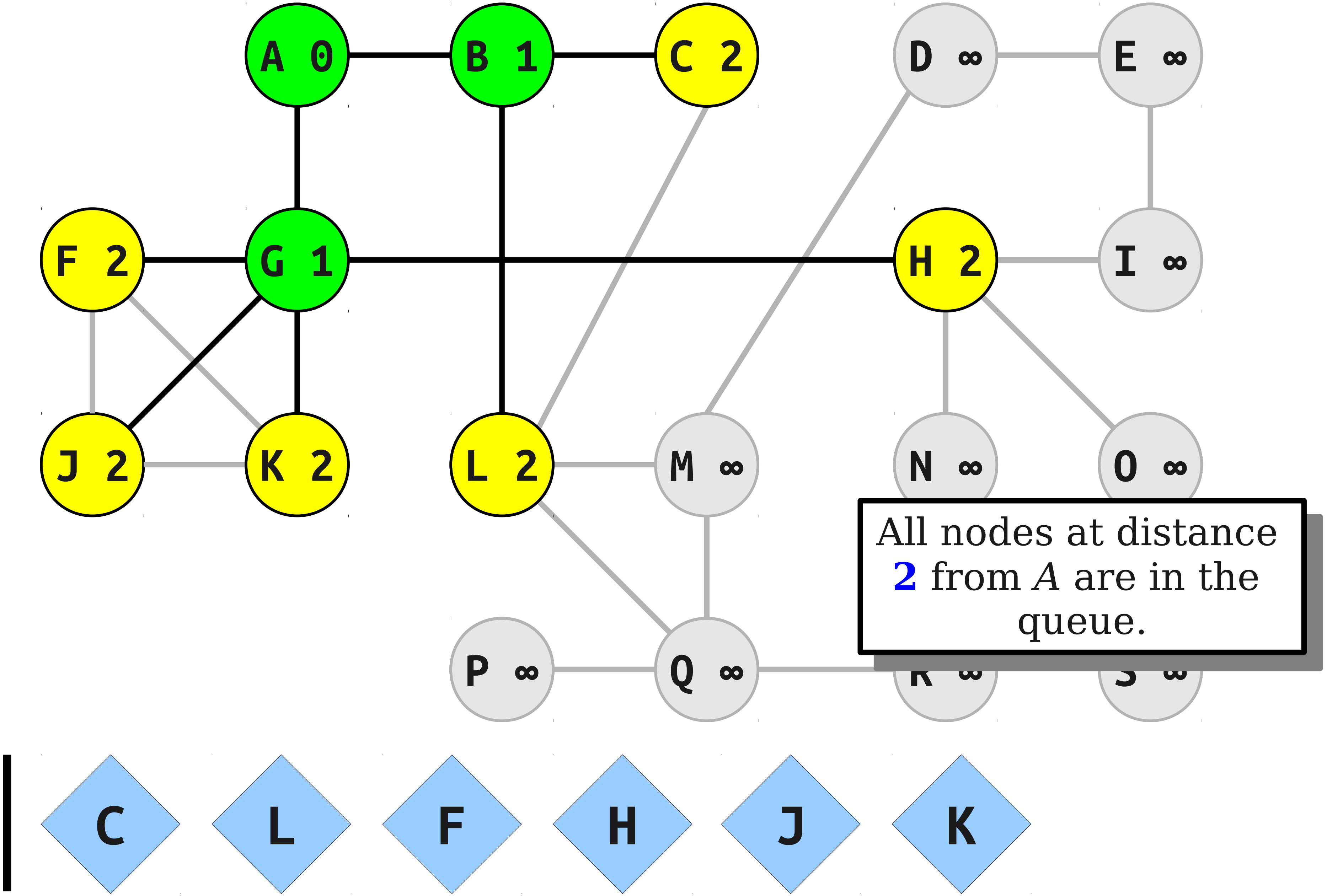


Breadth-First Search



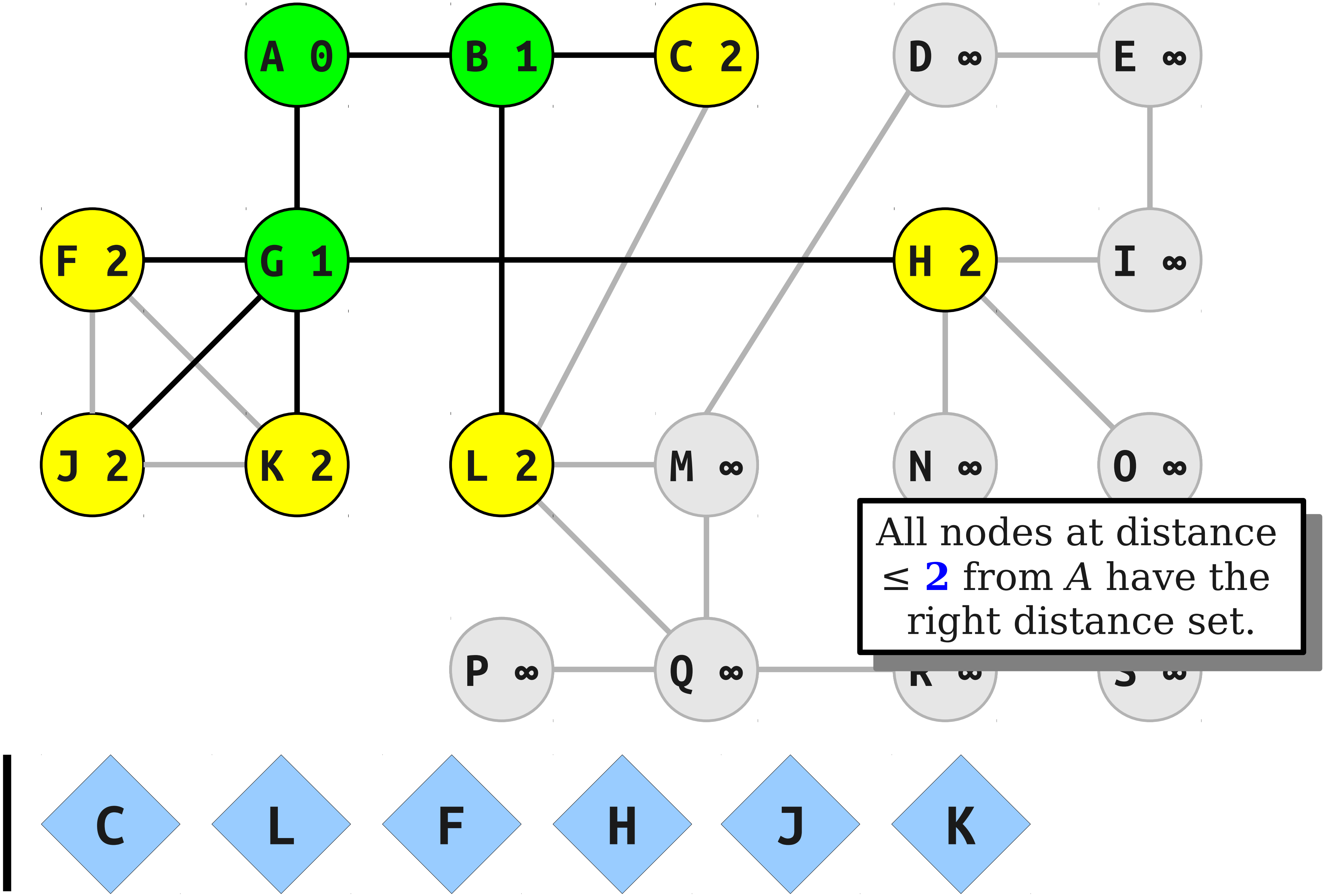
All nodes in the queue are at distance **2** from *A*.

Breadth-First Search



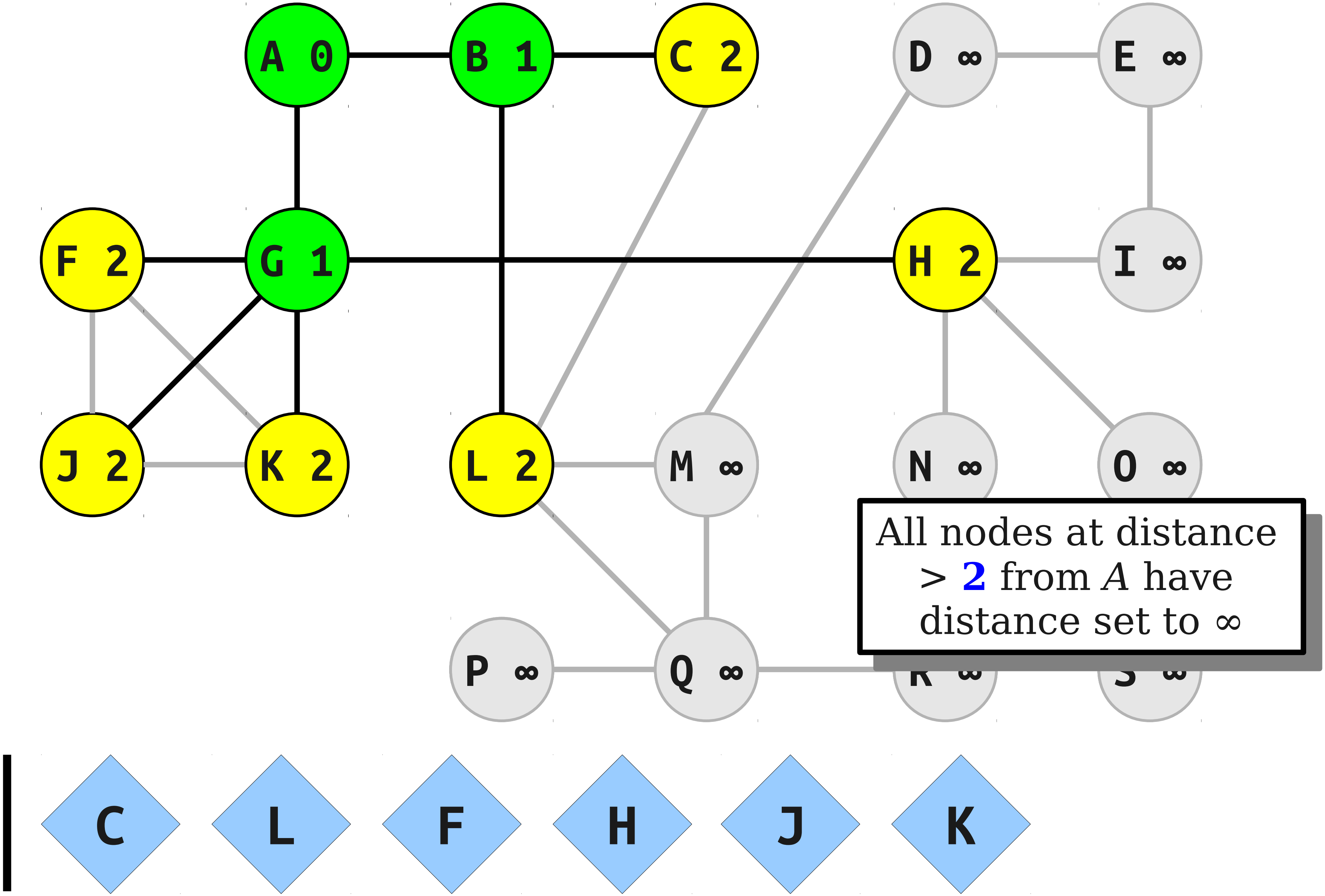
All nodes at distance 2 from *A* are in the queue.

Breadth-First Search



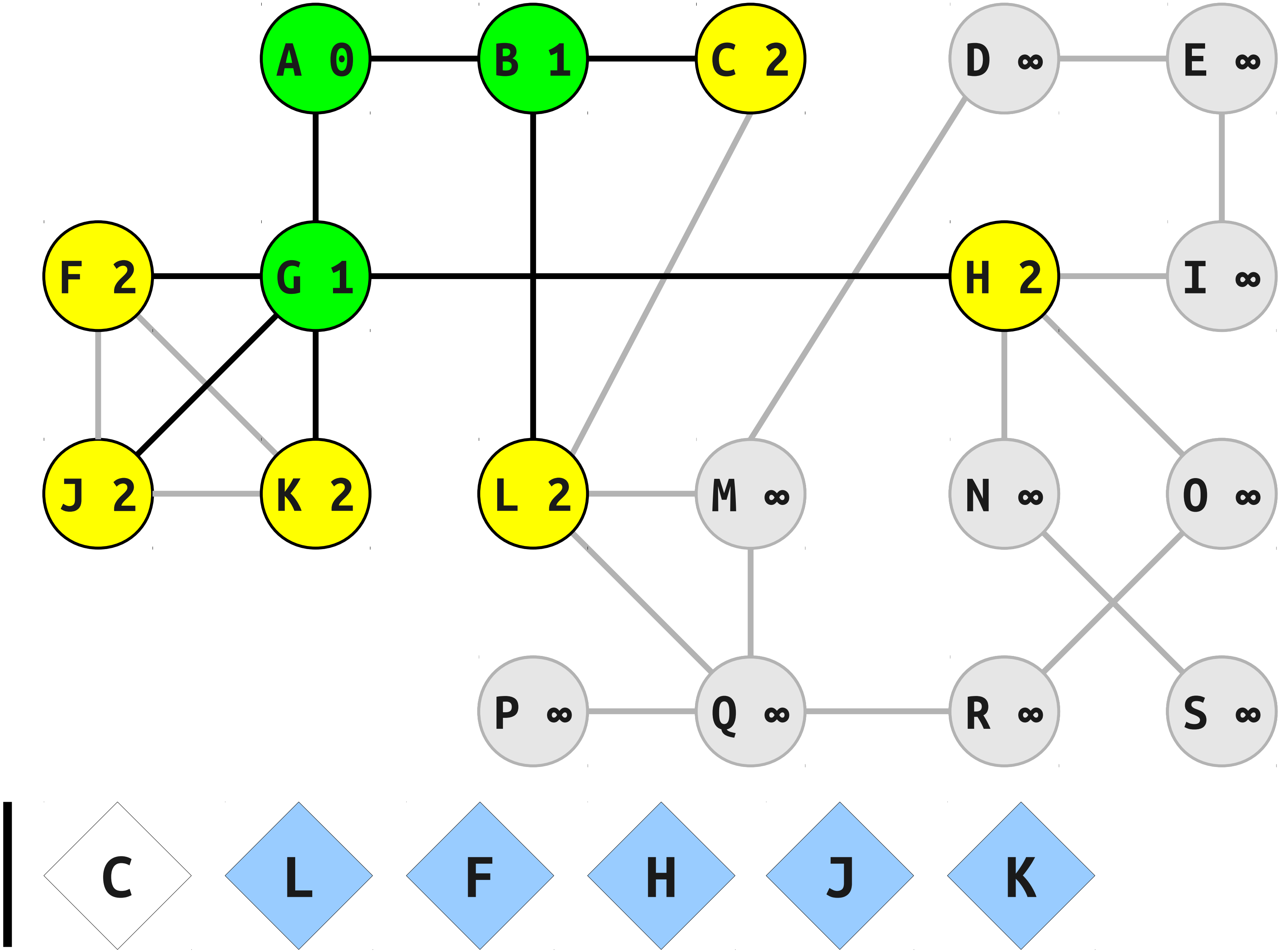
All nodes at distance ≤ **2** from *A* have the right distance set.

Breadth-First Search

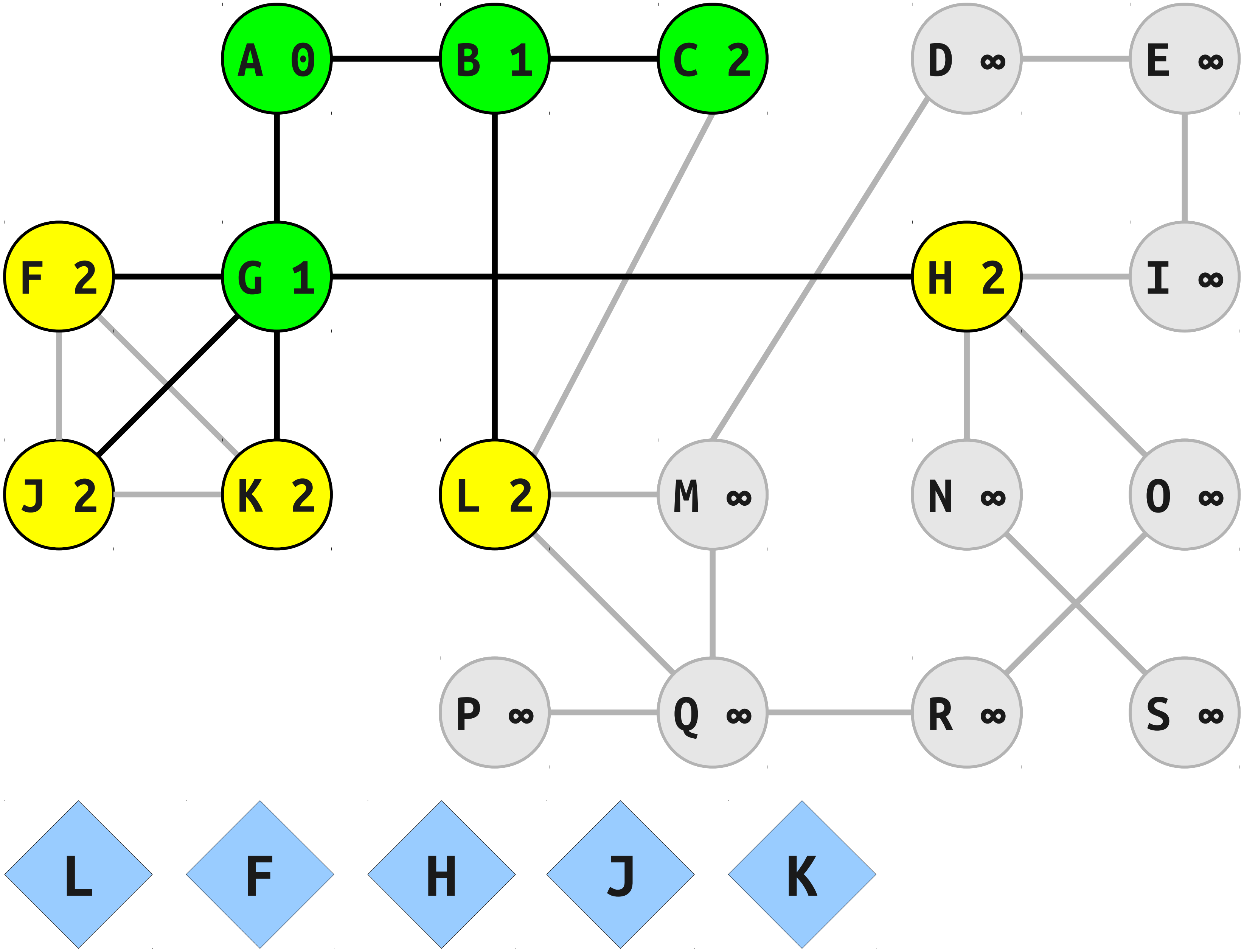


All nodes at distance > **2** from *A* have distance set to ∞

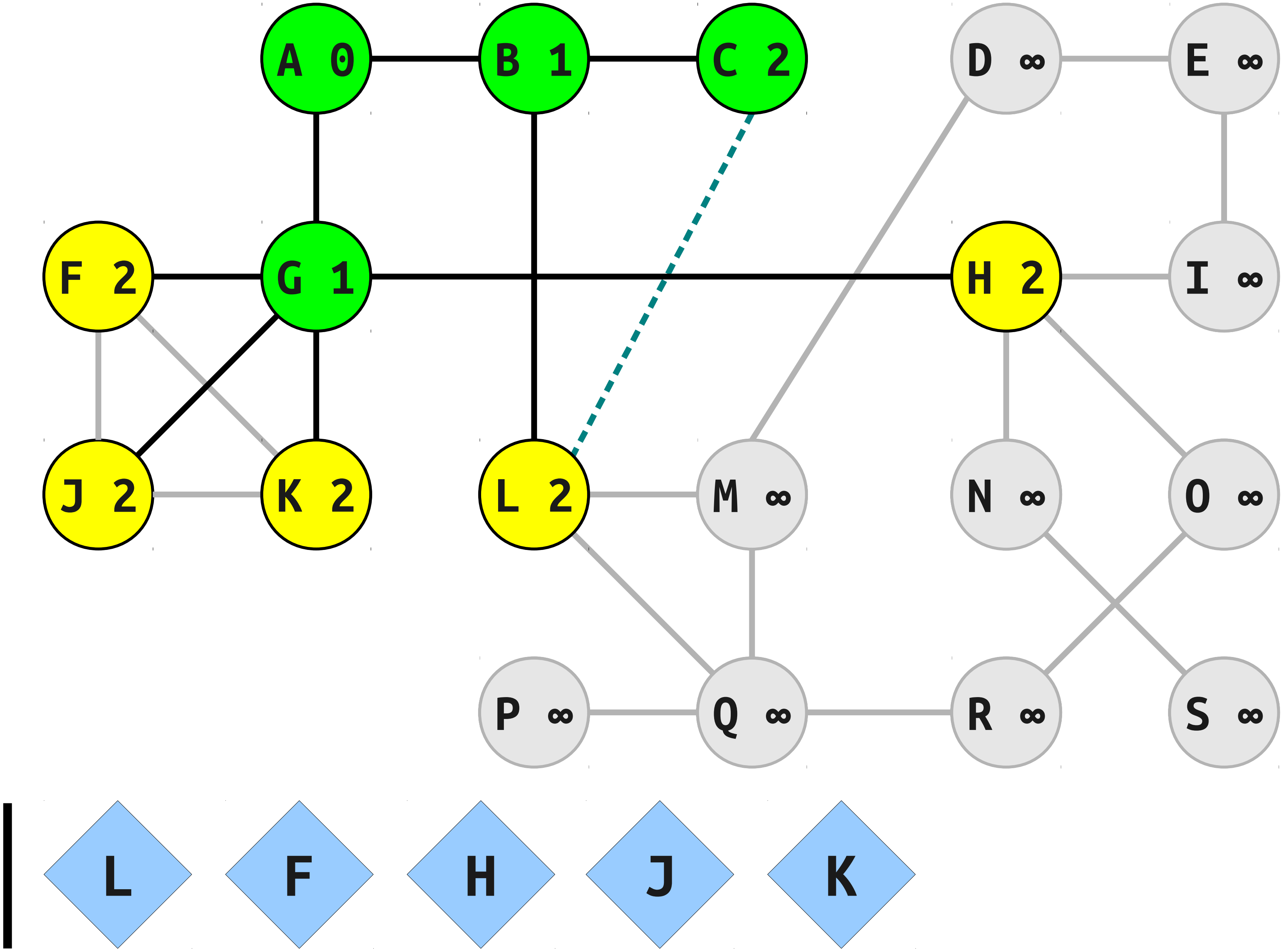
Breadth-First Search



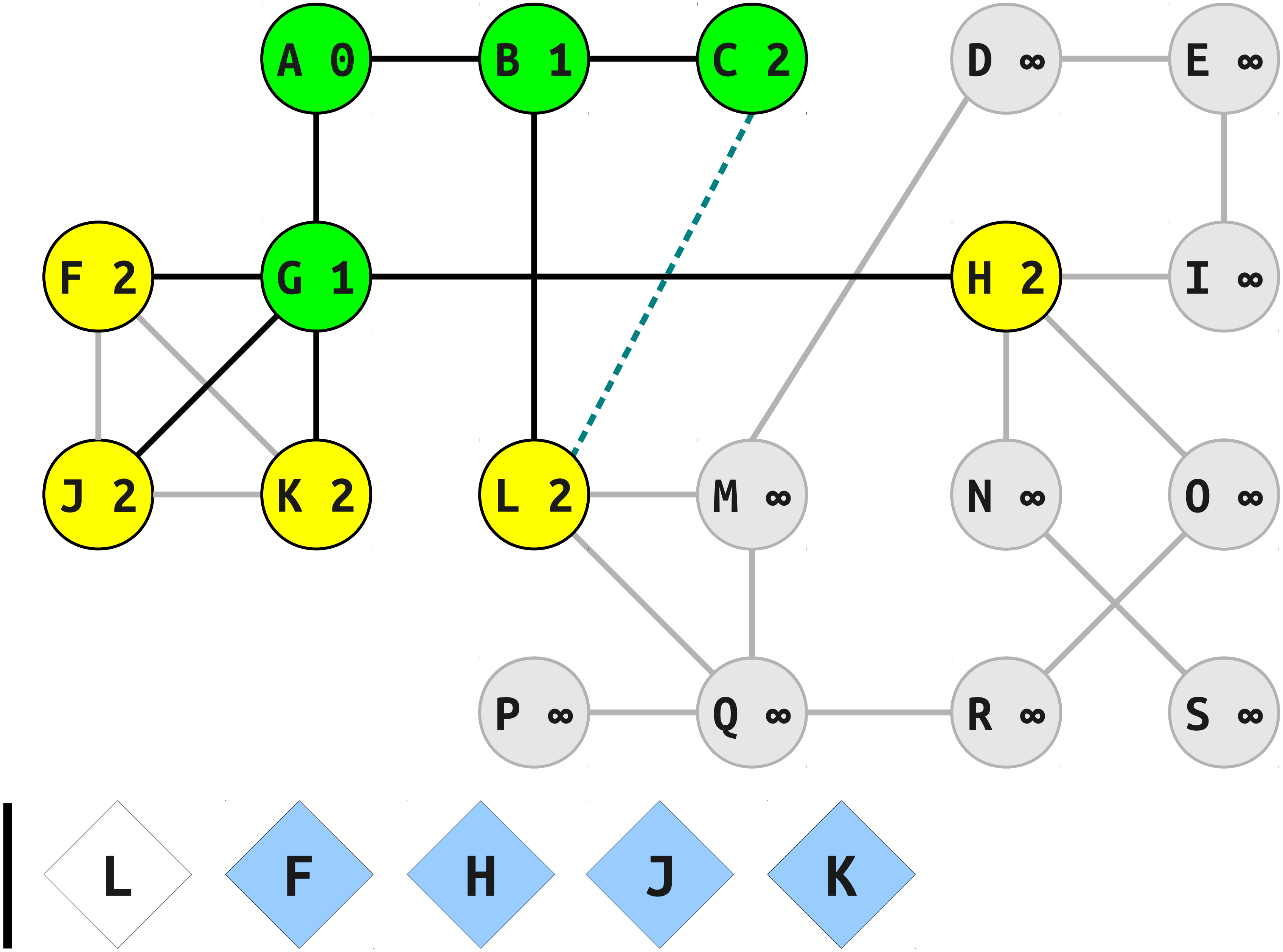
Breadth-First Search



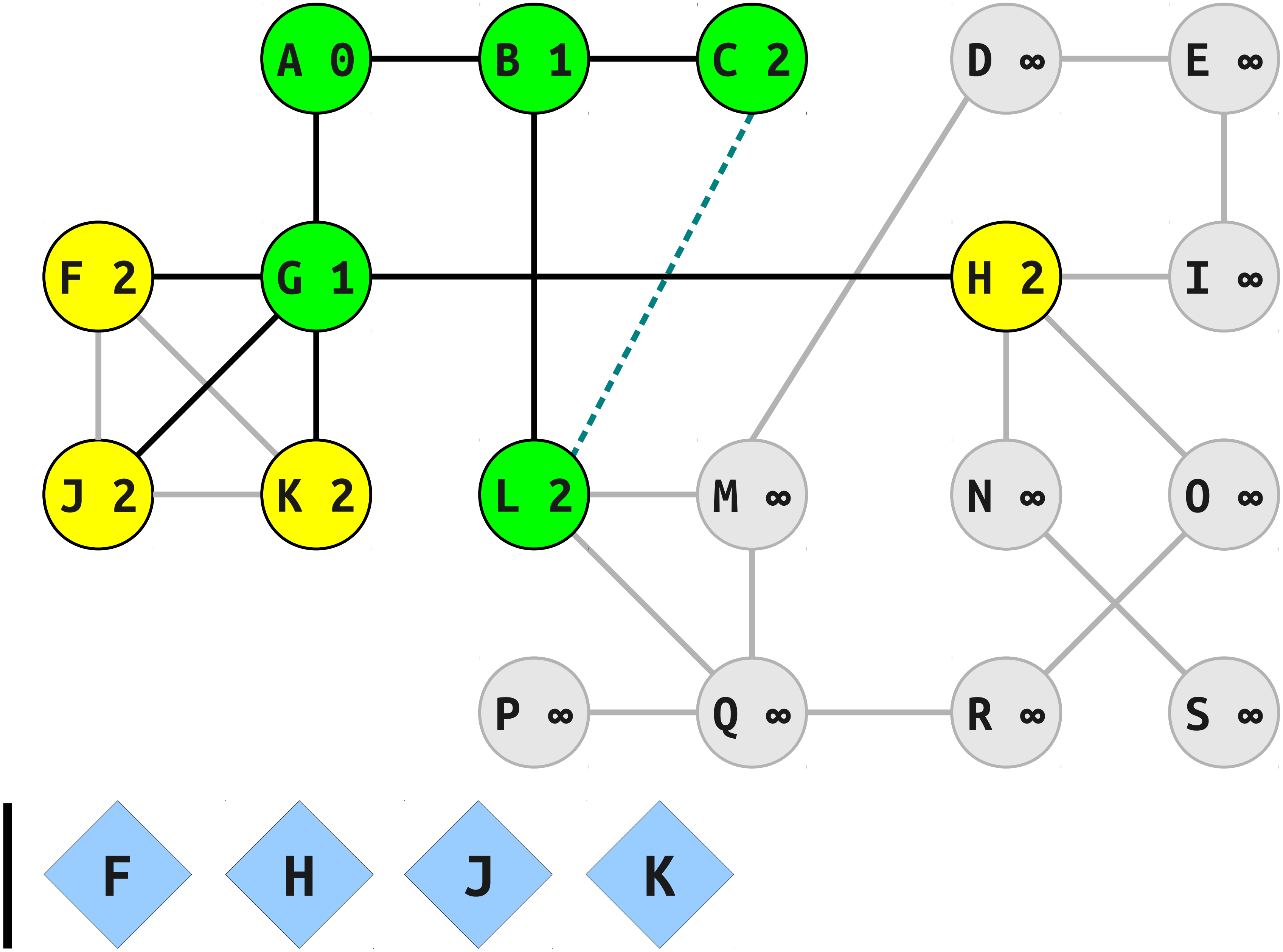
Breadth-First Search



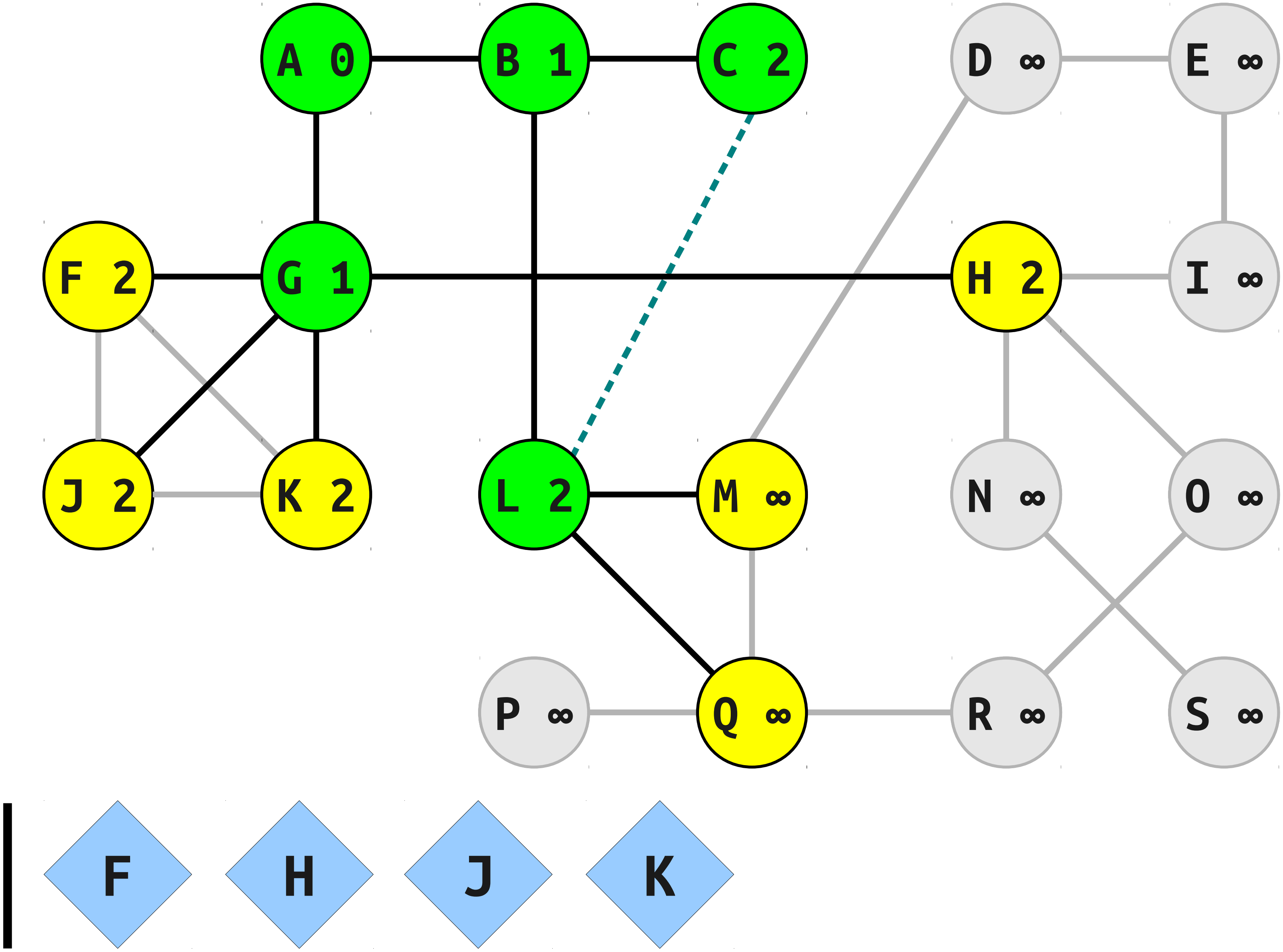
Breadth-First Search



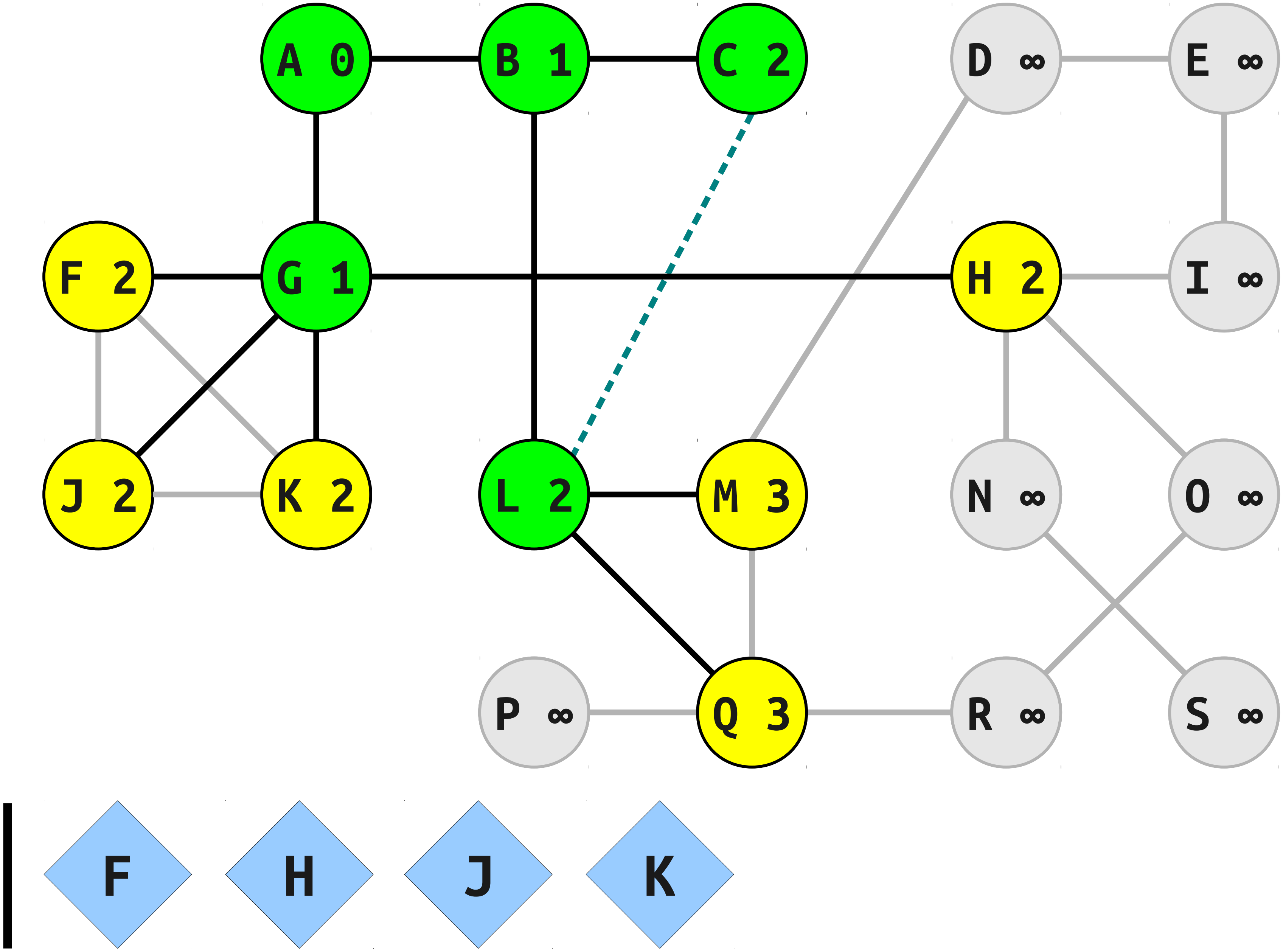
Breadth-First Search



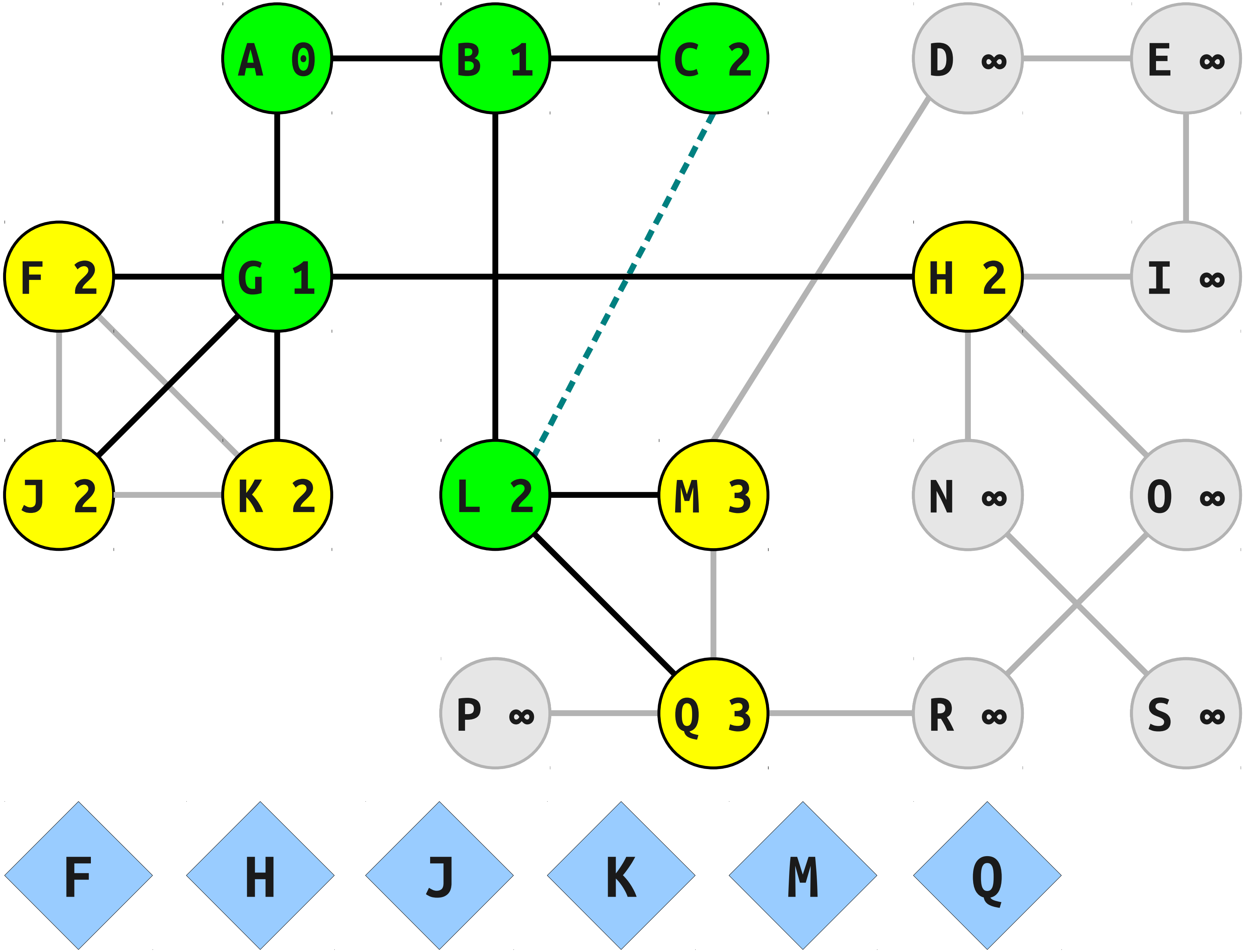
Breadth-First Search



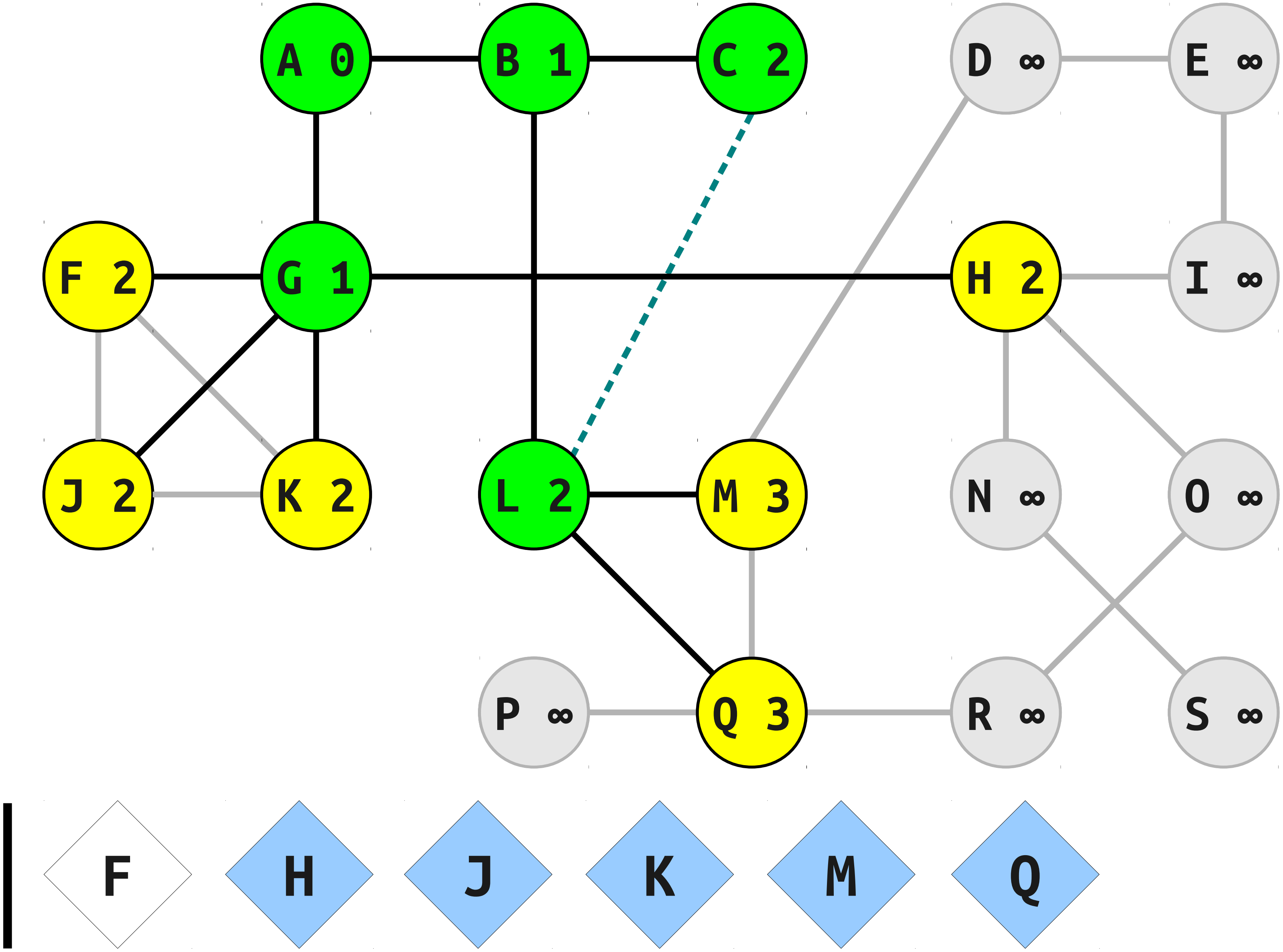
Breadth-First Search



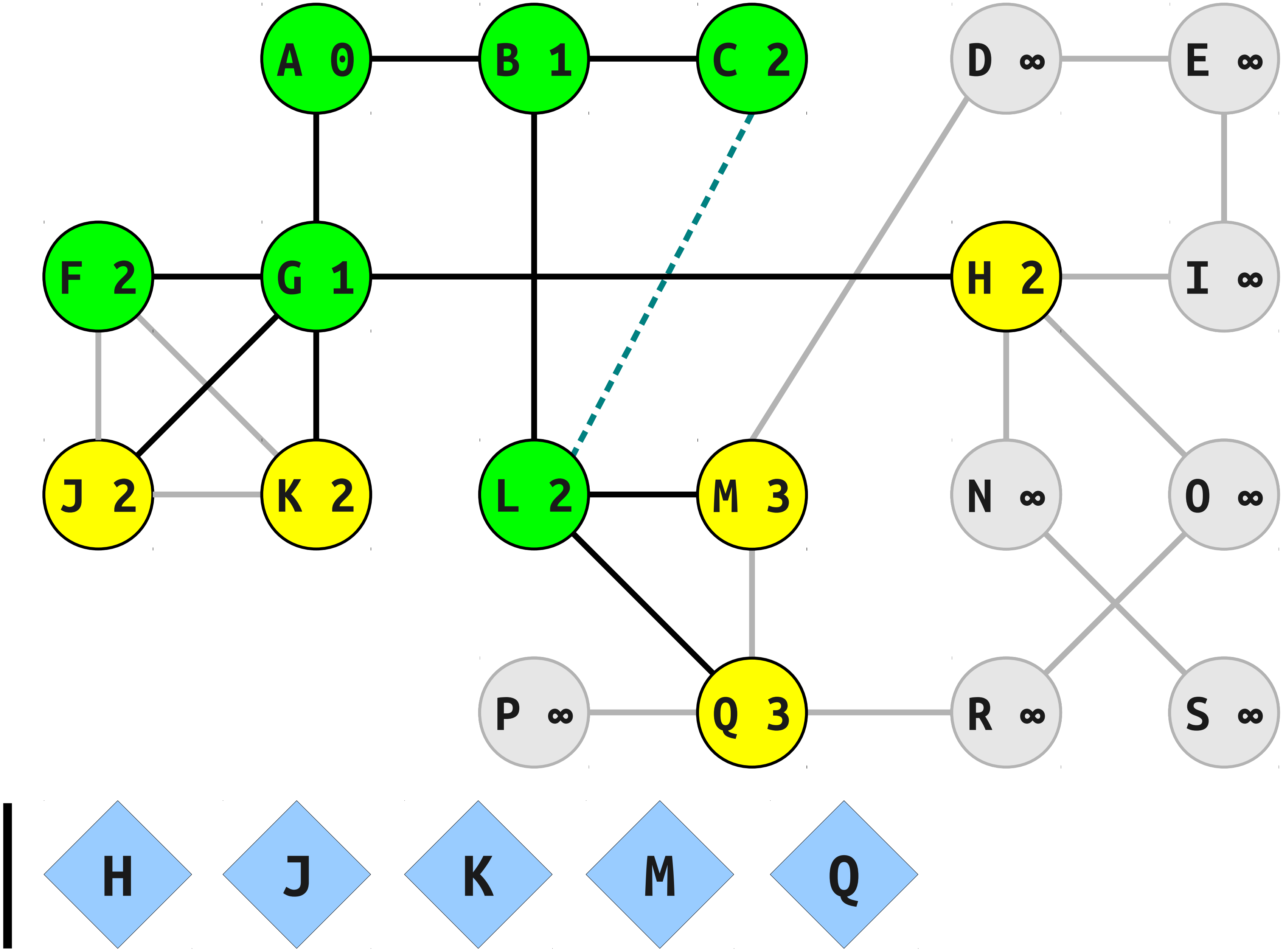
Breadth-First Search



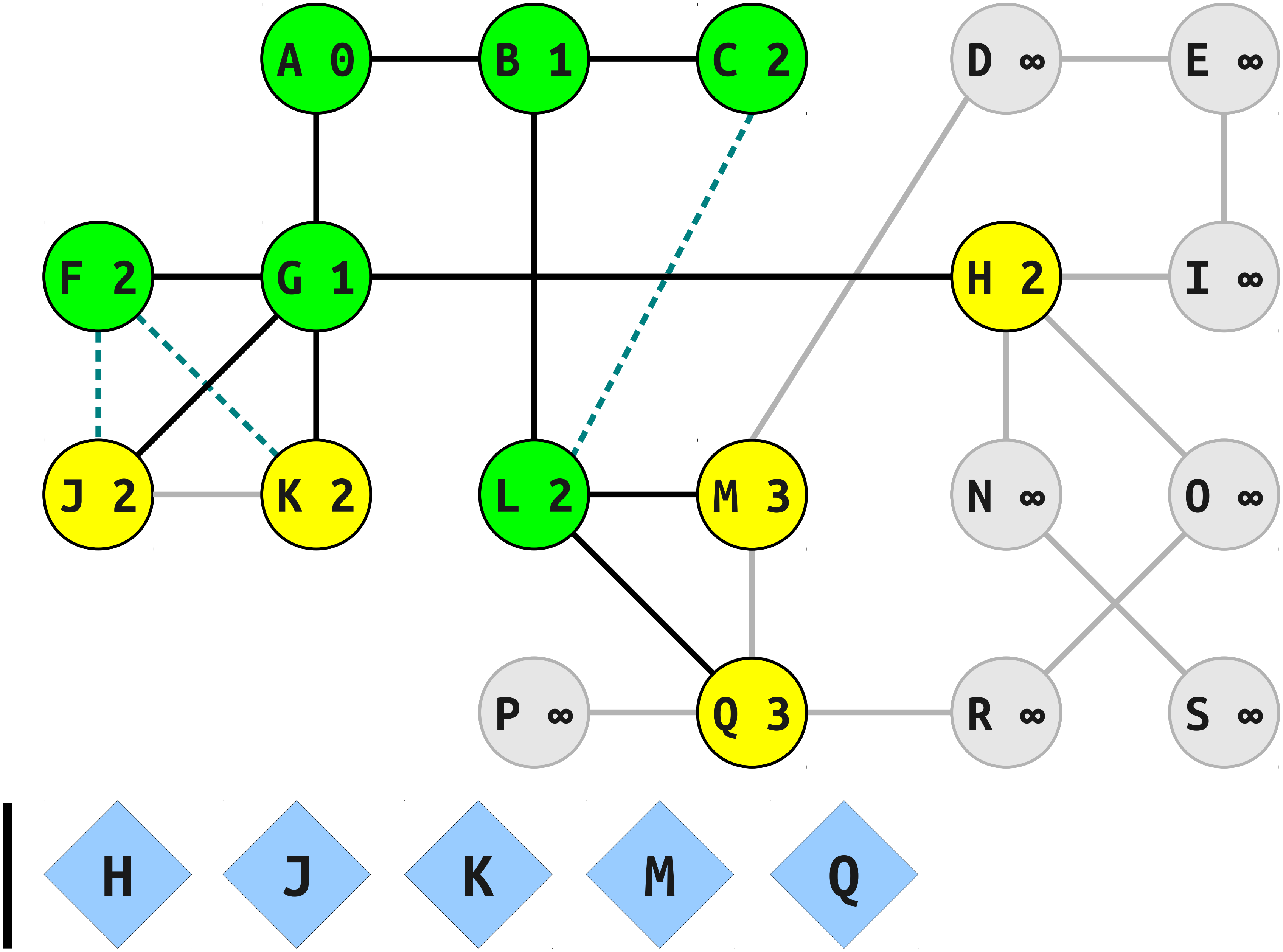
Breadth-First Search



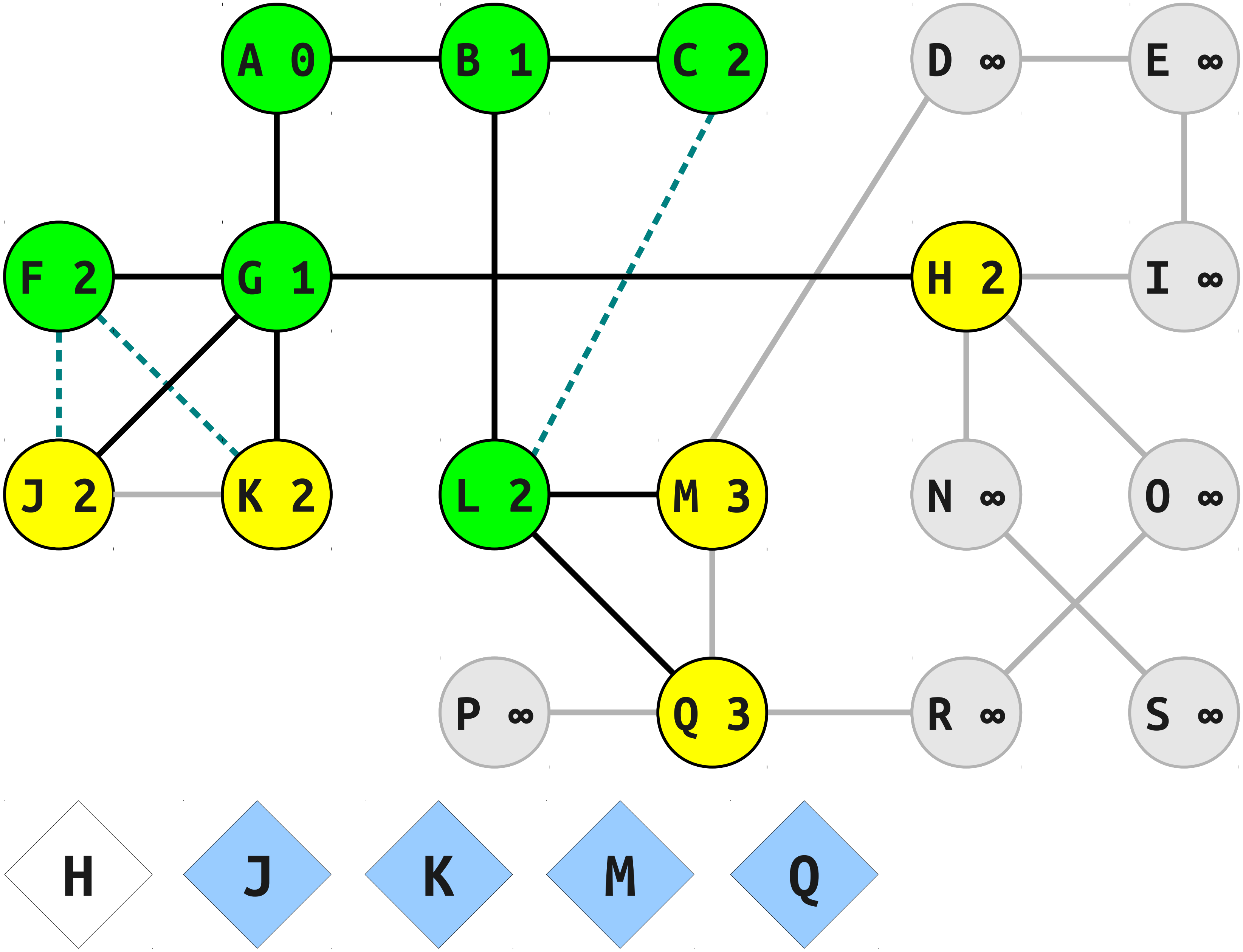
Breadth-First Search



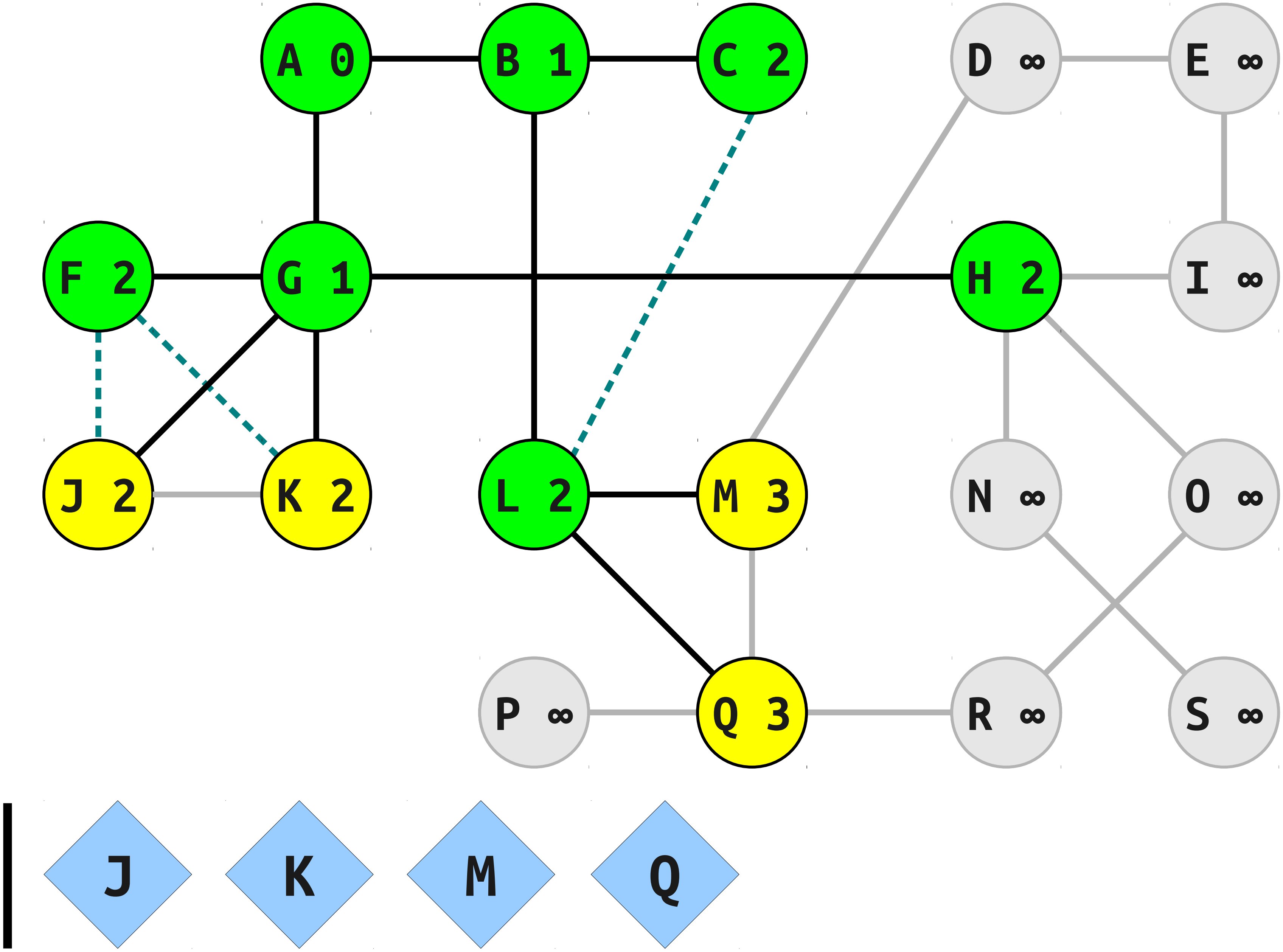
Breadth-First Search



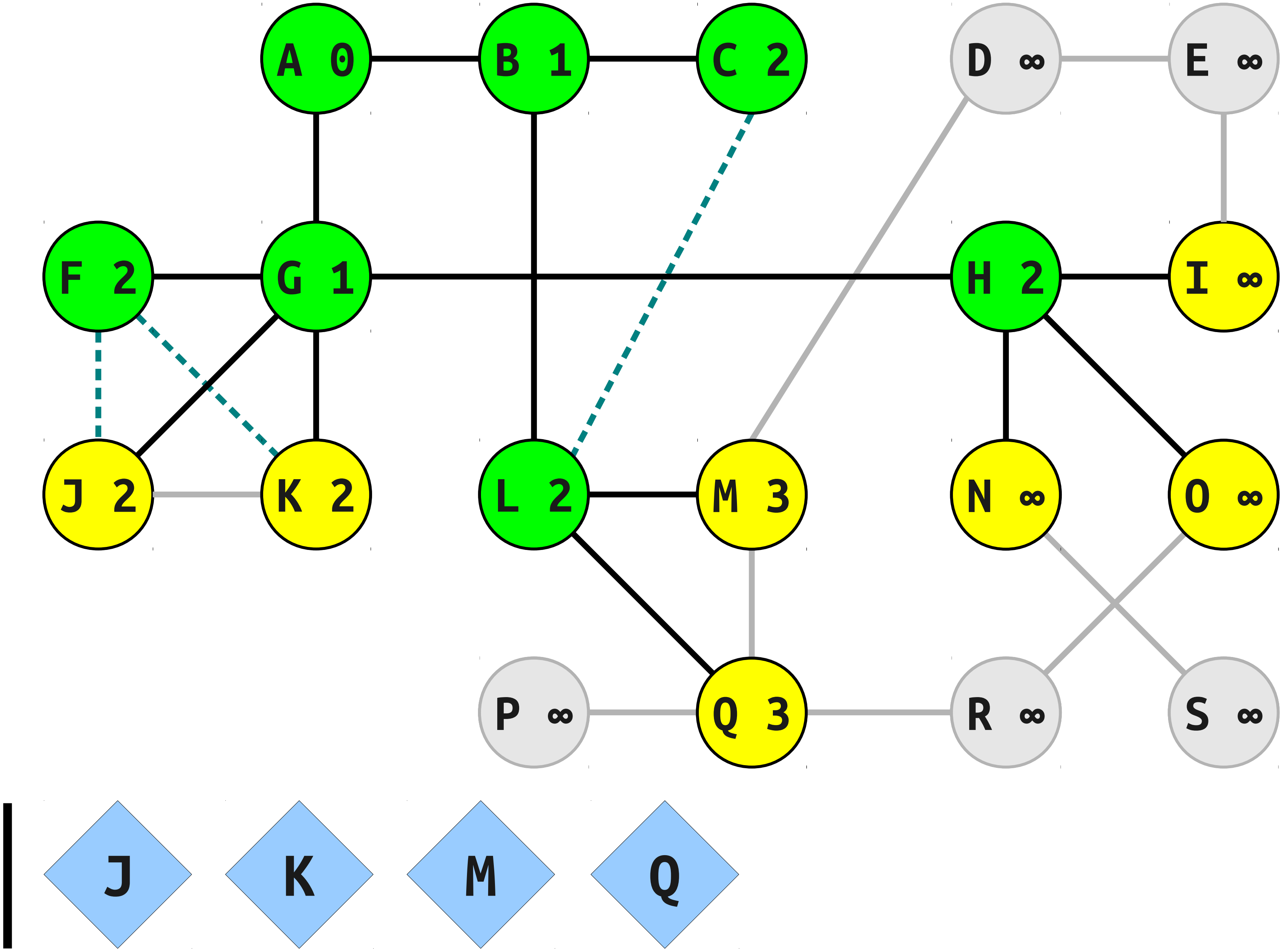
Breadth-First Search



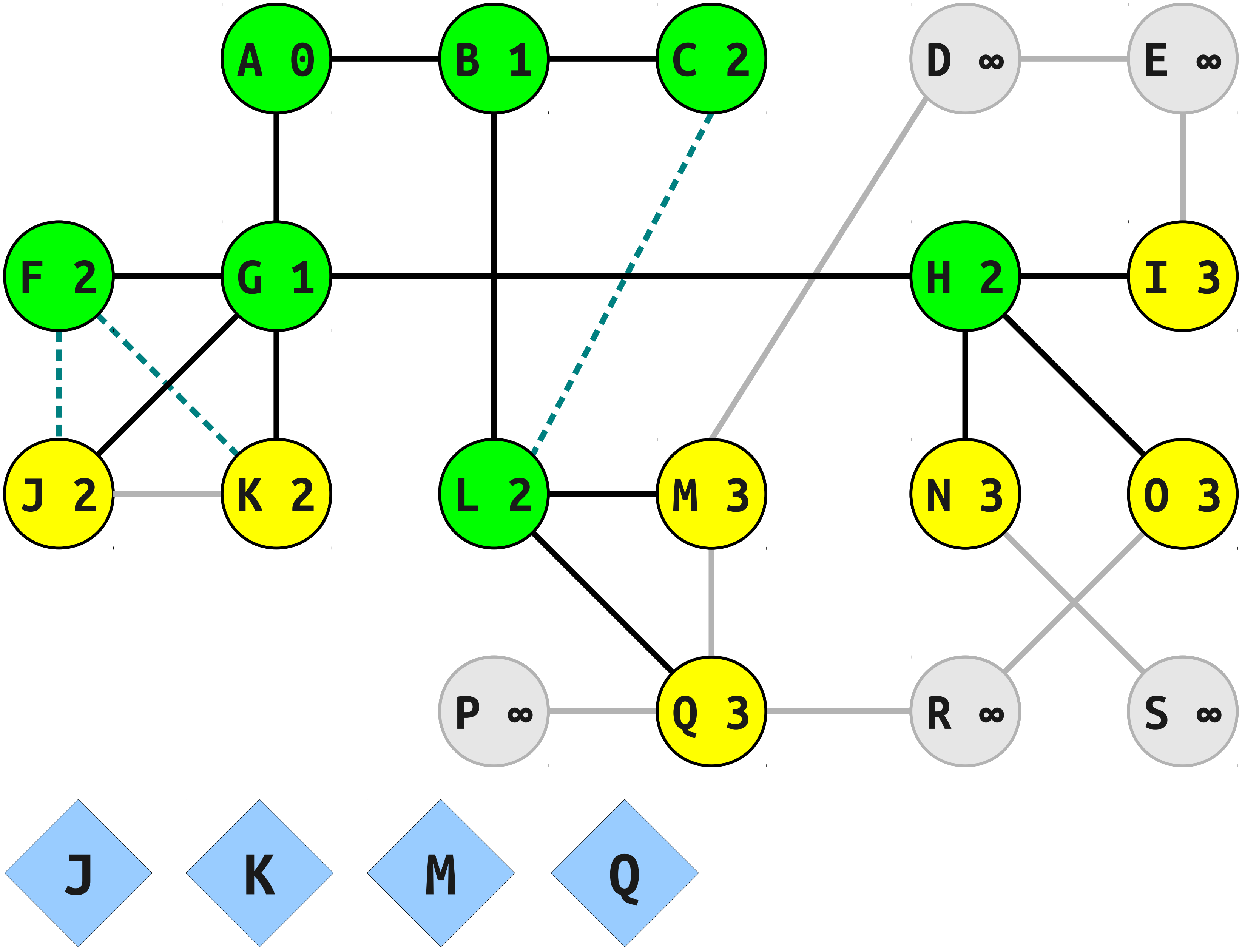
Breadth-First Search



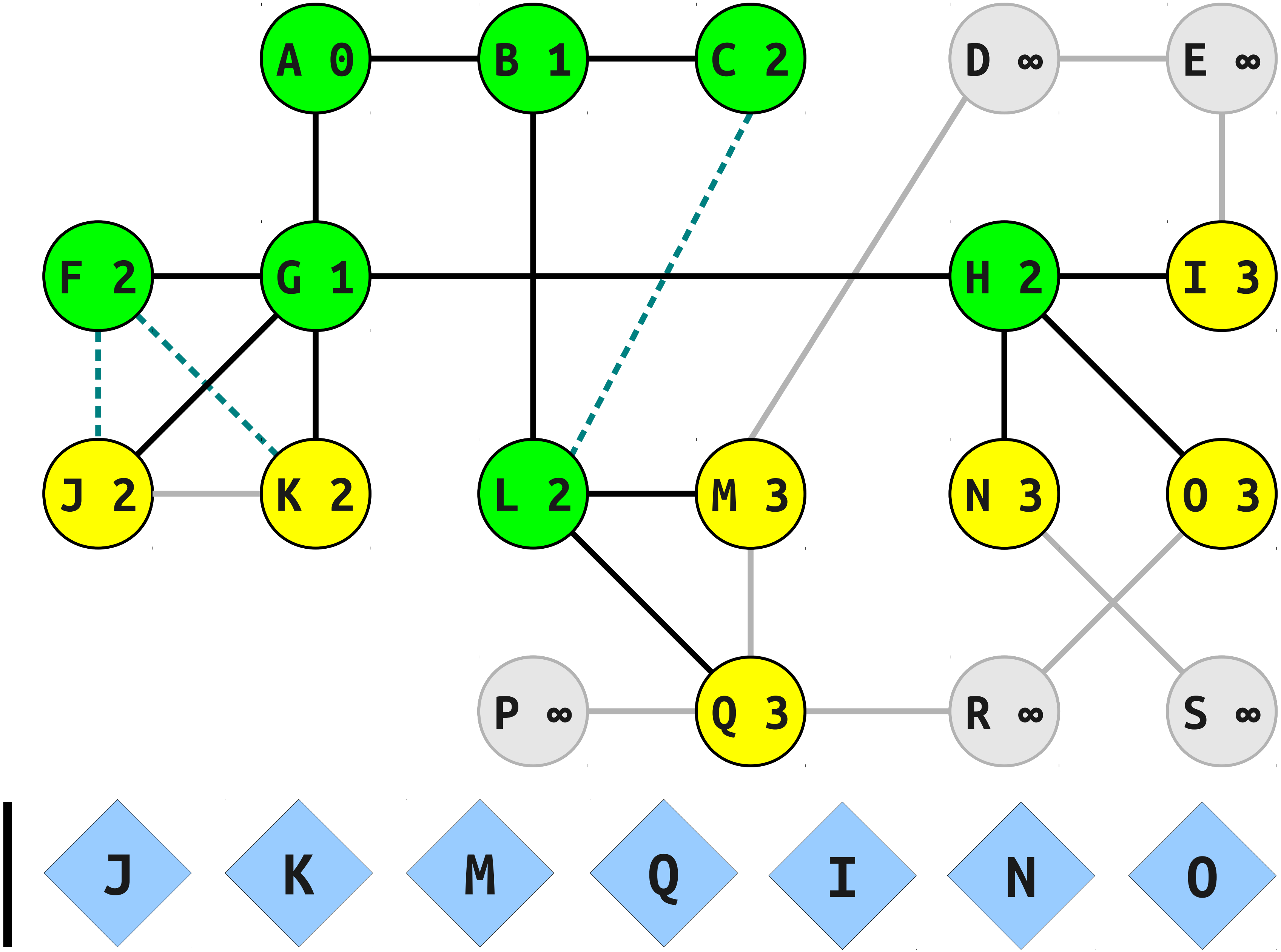
Breadth-First Search



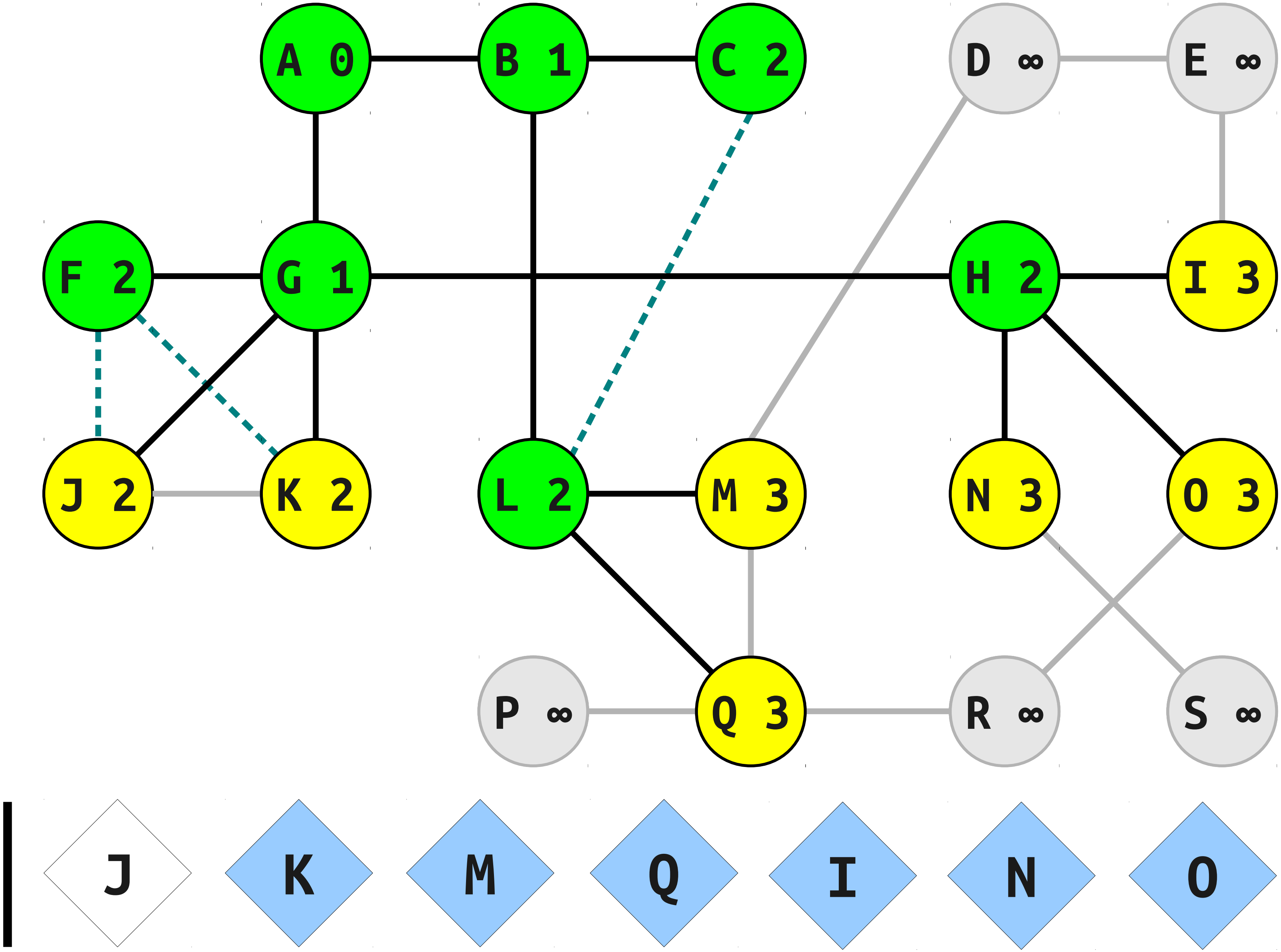
Breadth-First Search



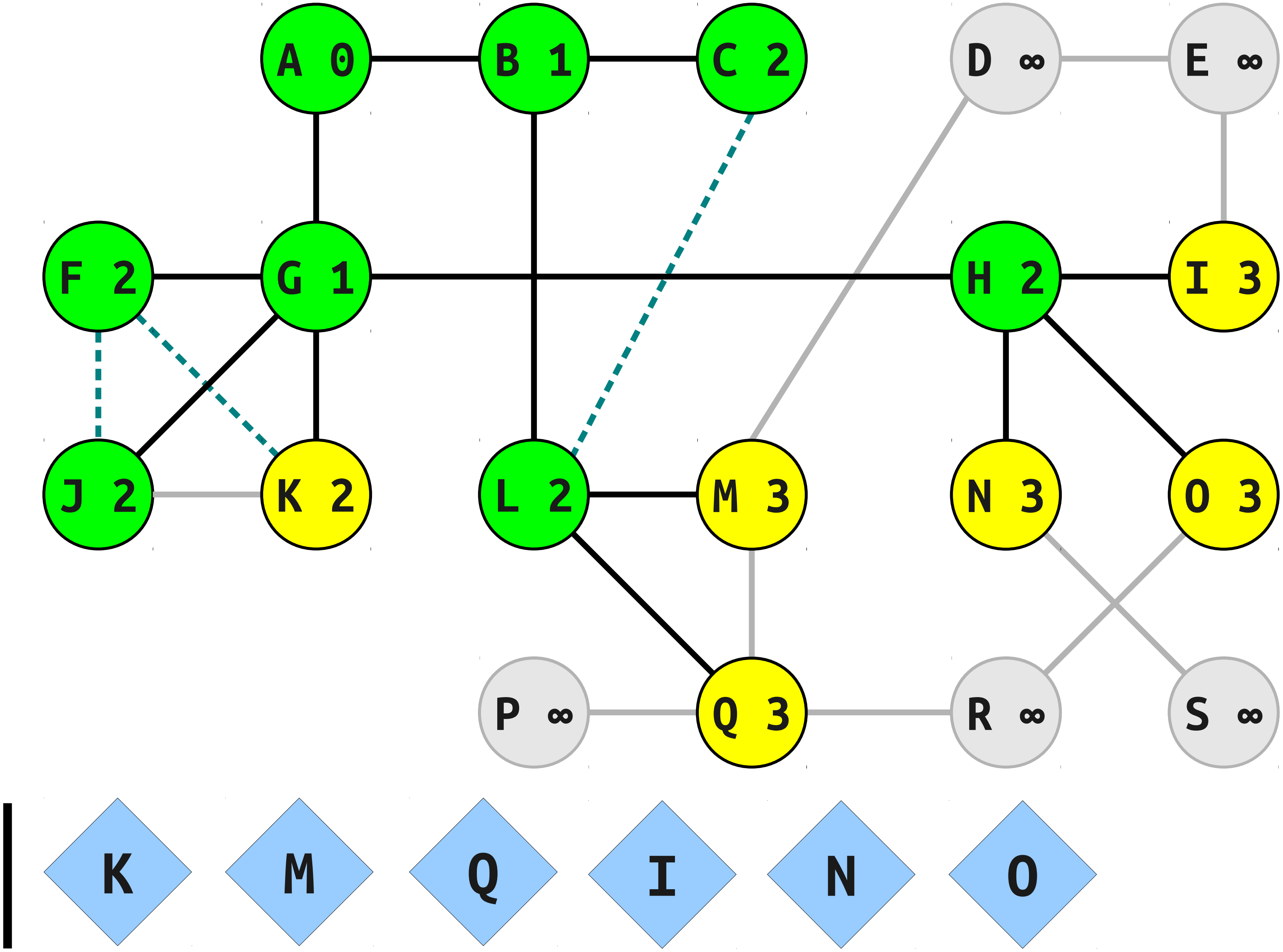
Breadth-First Search



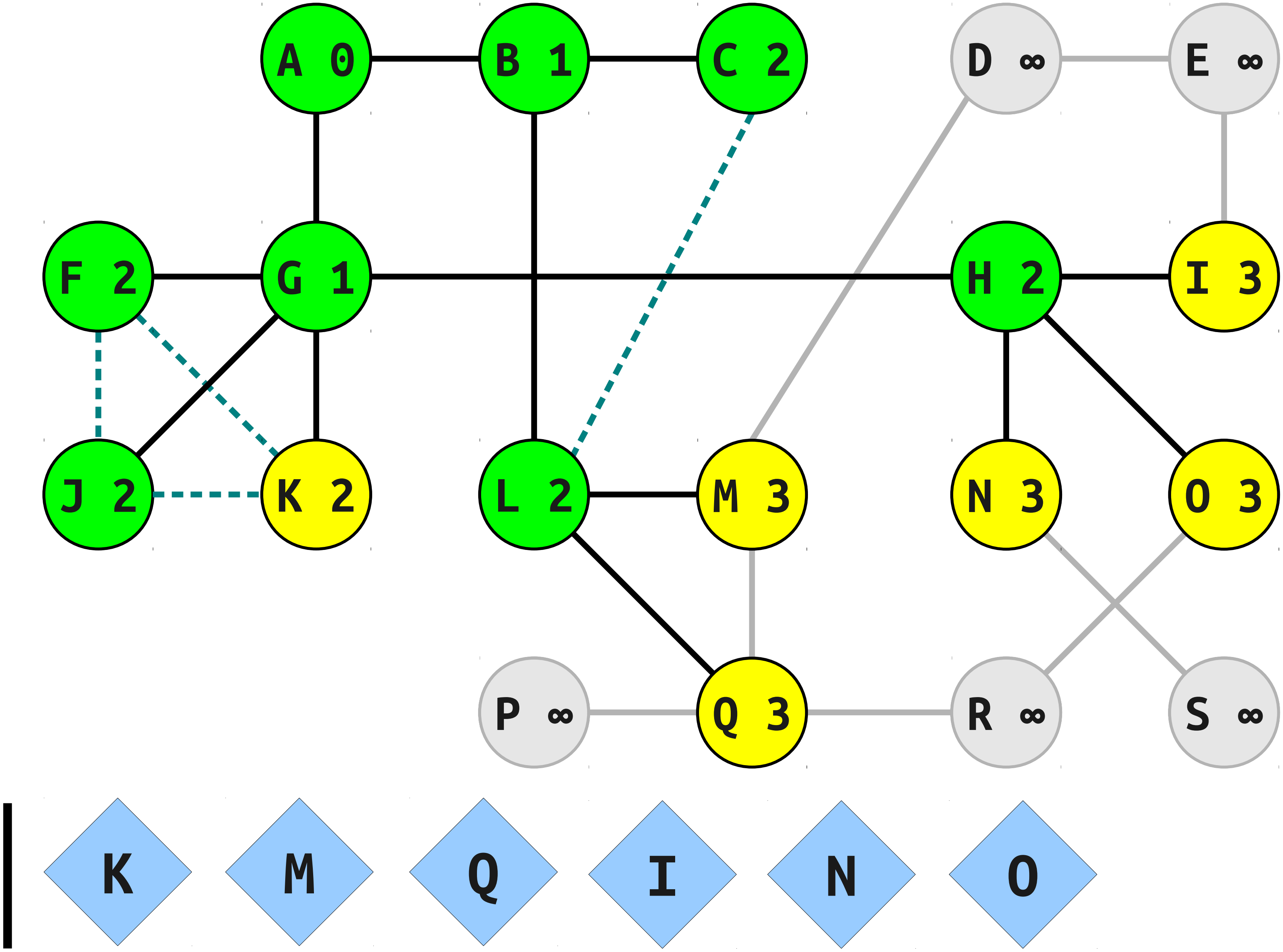
Breadth-First Search



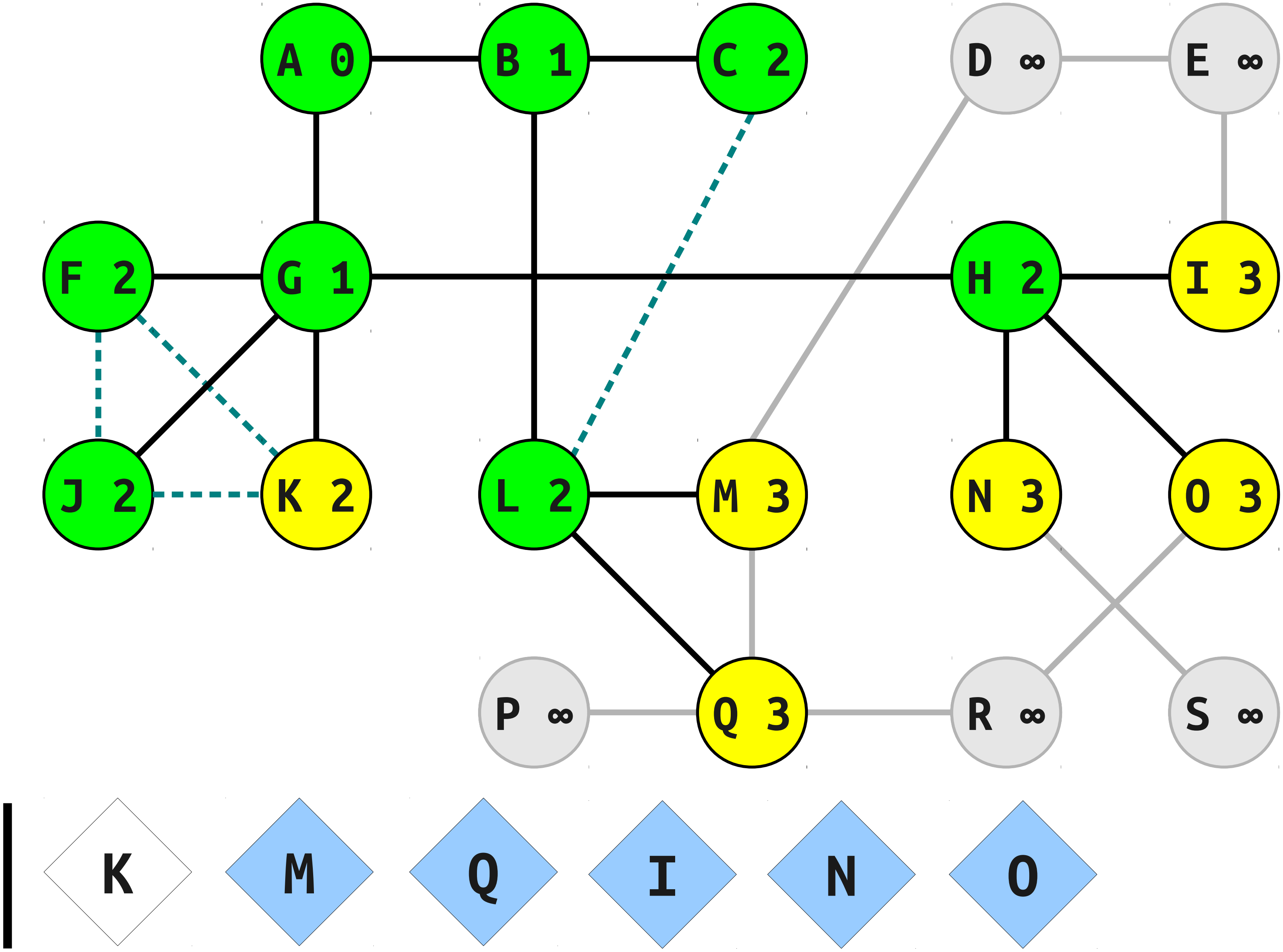
Breadth-First Search



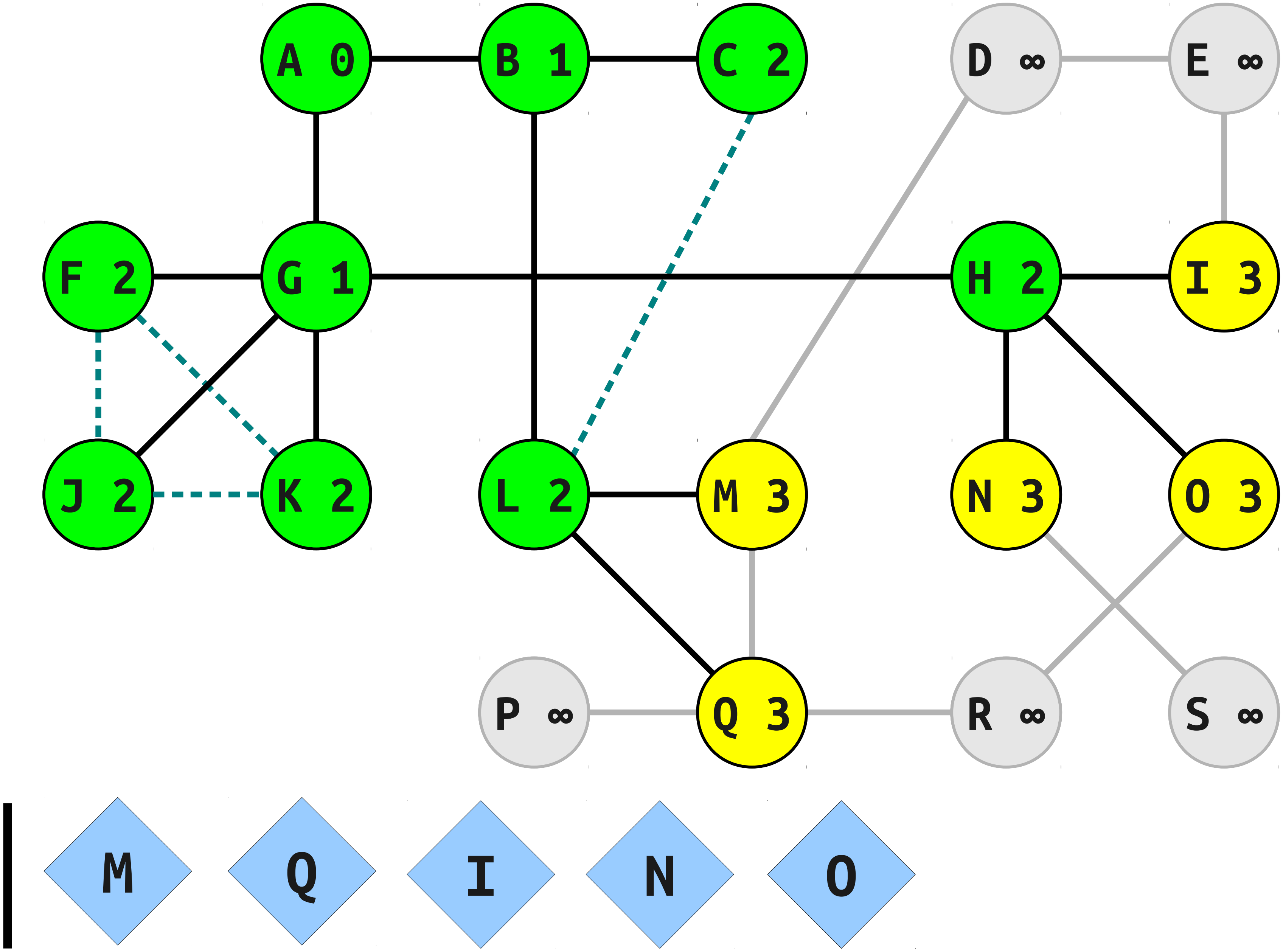
Breadth-First Search



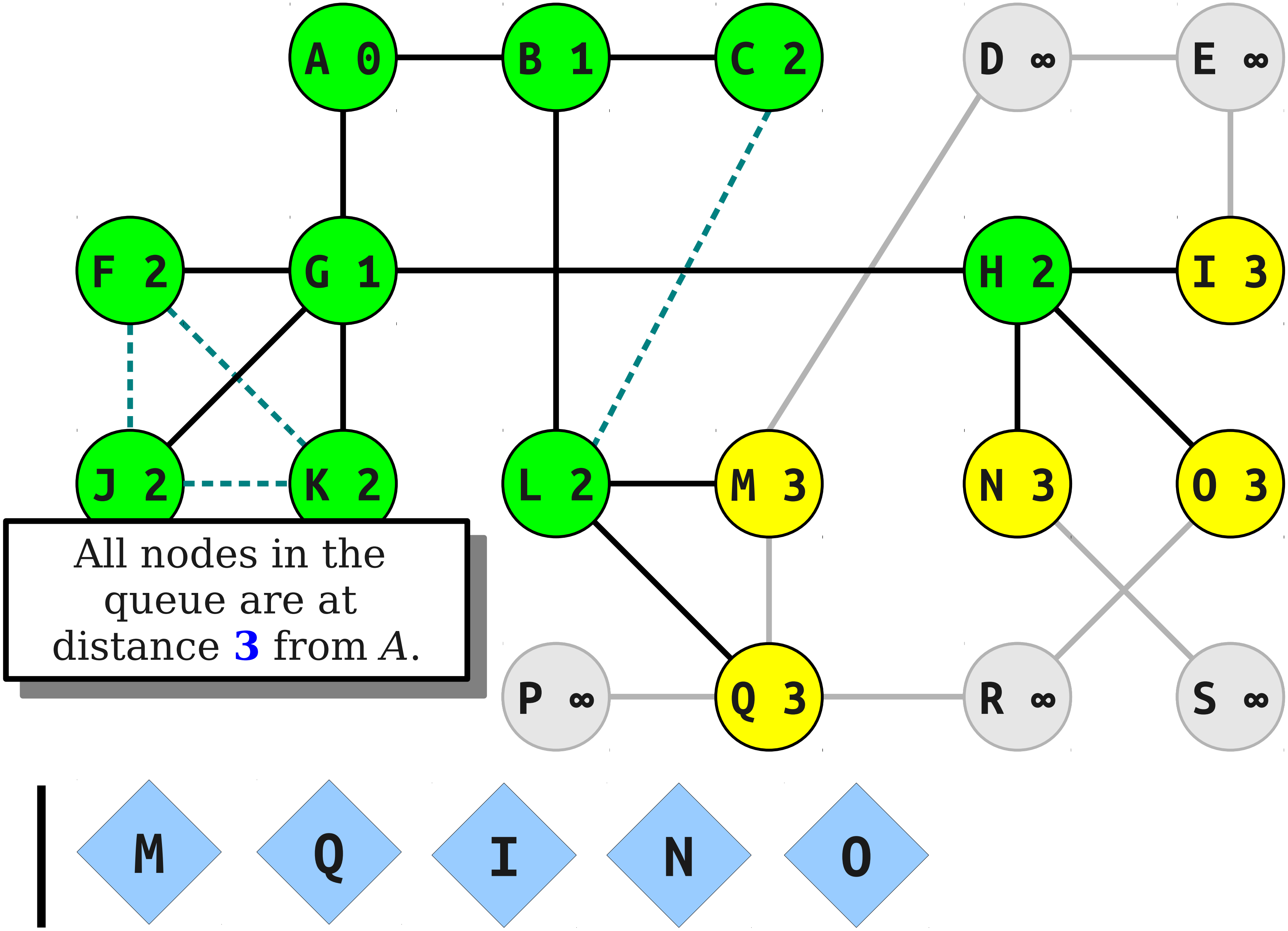
Breadth-First Search



Breadth-First Search

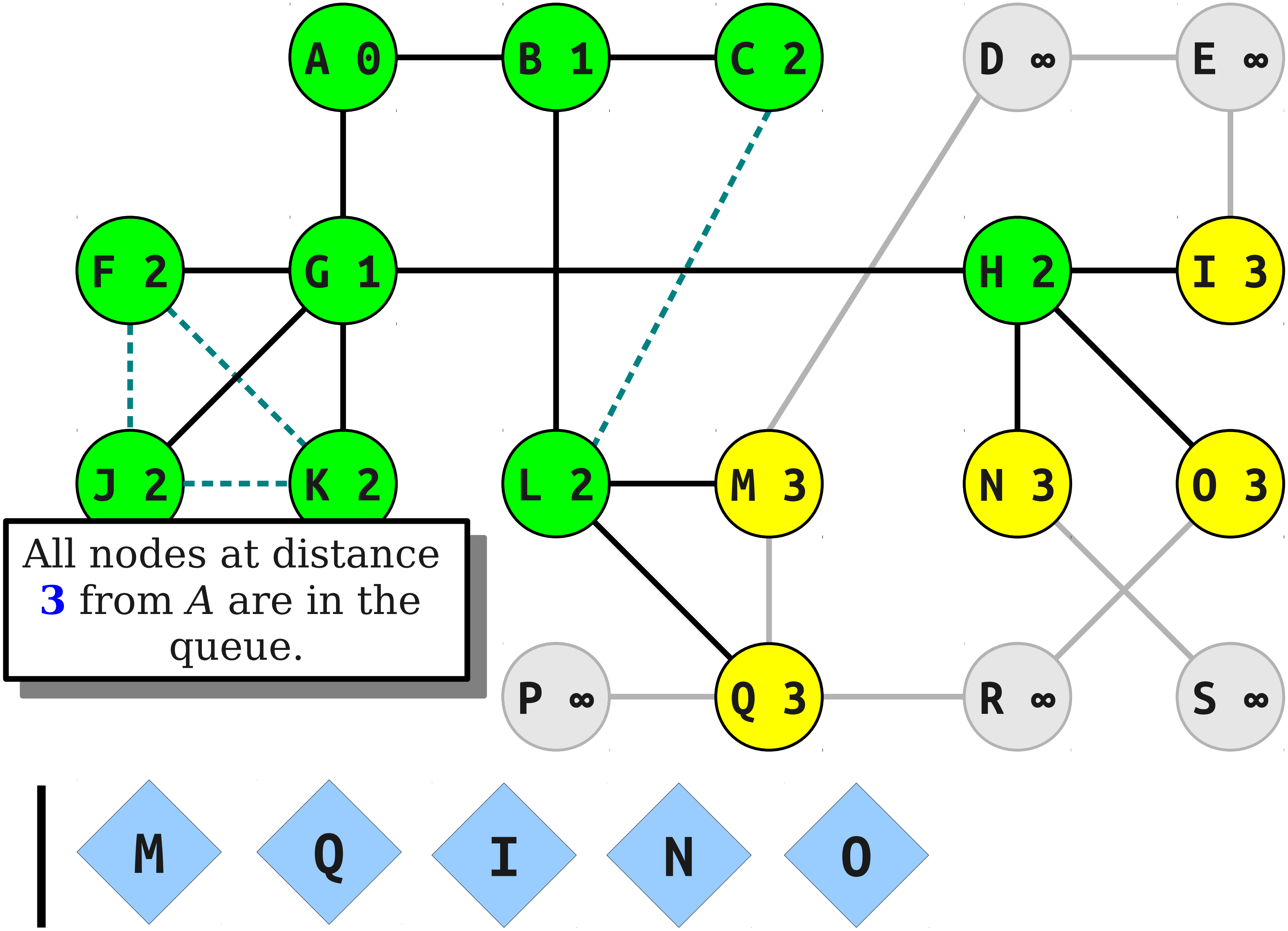


Breadth-First Search



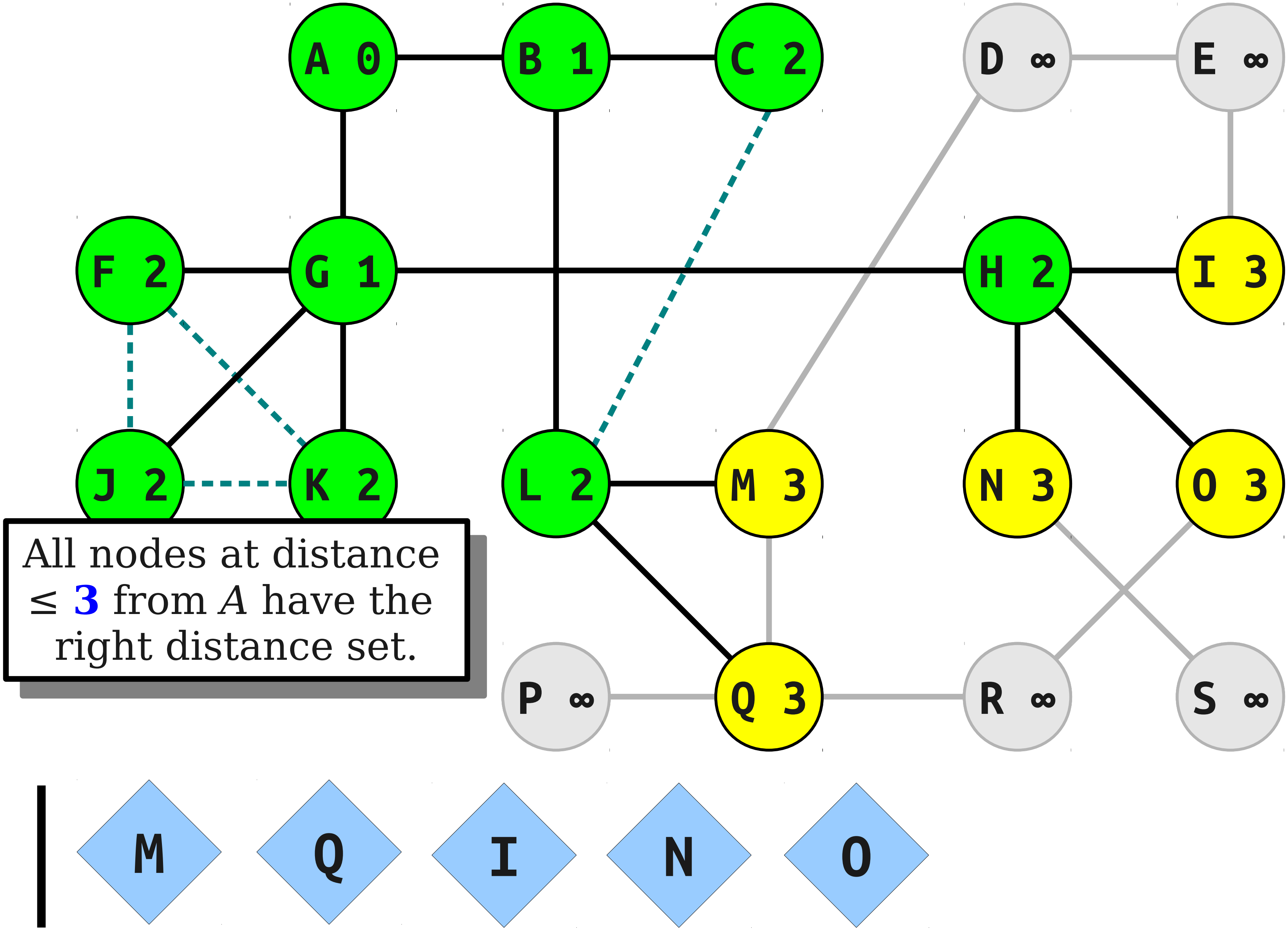
All nodes in the queue are at distance **3** from *A*.

Breadth-First Search



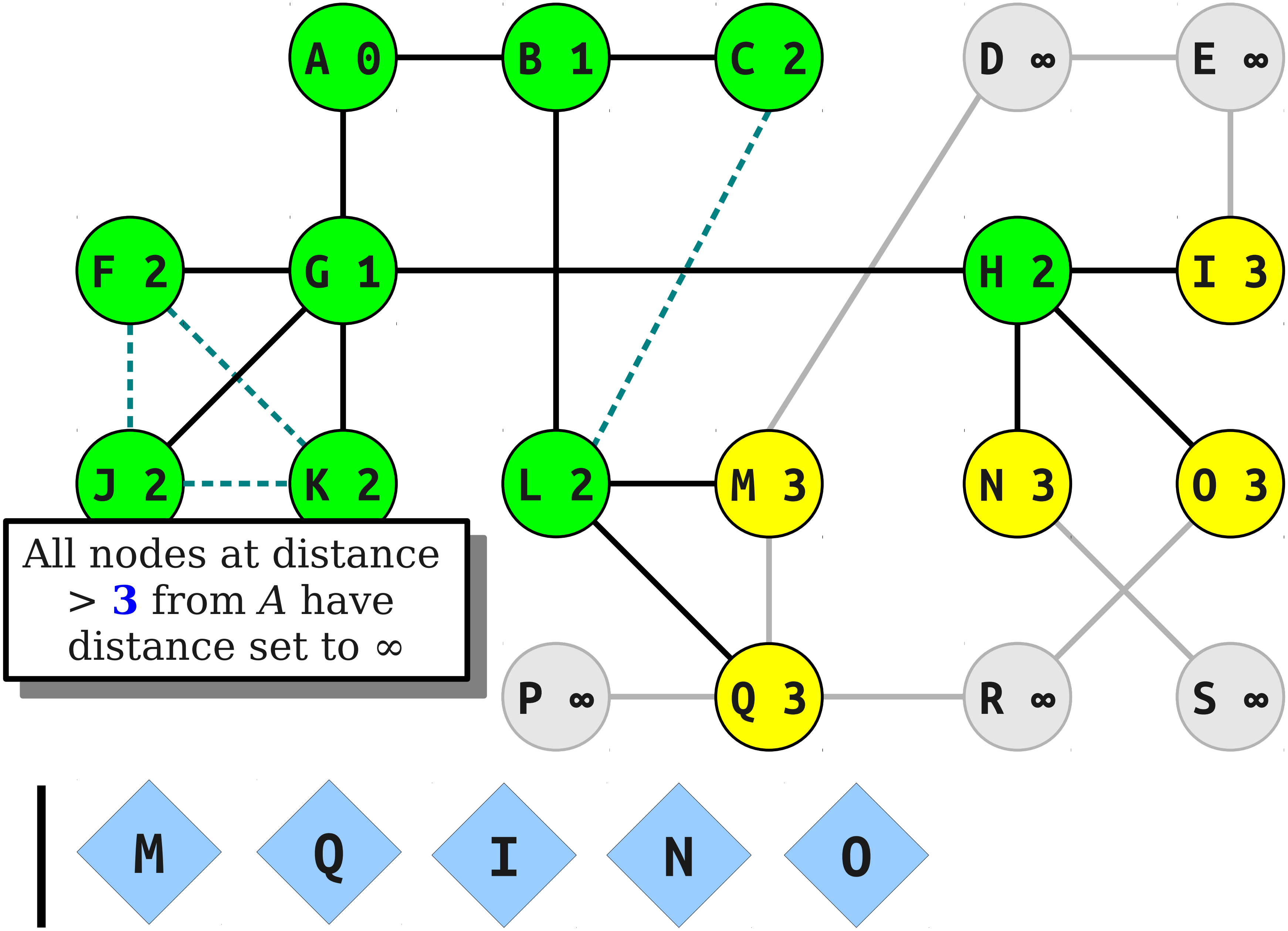
All nodes at distance 3 from *A* are in the queue.

Breadth-First Search



All nodes at distance ≤ 3 from *A* have the right distance set.

Breadth-First Search



All nodes at distance > **3** from *A* have distance set to ∞

Theorem: Breadth-first search always terminates with dist[v] = d(s, v) for all $v \in V$.

Theorem: Breadth-first search always terminates with dist[v] = d(s, v) for all $v \in V$.

Proof: Define "round n" of BFS to be an instance where at the start of the loop, all nodes v in the queue satisfy dist[v] = n.

Theorem: Breadth-first search always terminates with dist[v] = d(s, v) for all $v \in V$.

Proof: Define "round n" of BFS to be an instance where at the start of the loop, all nodes v in the queue satisfy dist[v] = n. We will prove in an lemma the following are always true after the first n rounds:

(1) For any node v, d(s, v) = n iff v is in the queue.
(2) All nodes v where d(s, v) $\leq n$ have dist[v] = d(s, v).
(3) All nodes v where d(s, v) $> n$ have dist[v] = ∞

Theorem: Breadth-first search always terminates with dist[v] = d(s, v) for all $v \in V$.

Proof: Define "round n" of BFS to be an instance where at the start of the loop, all nodes v in the queue satisfy dist[v] = n. We will prove in an lemma the following are always true after the first n rounds:

(1) For any node v, d(s, v) = n iff v is in the queue.
(2) All nodes v where d(s, v) $\leq n$ have dist[v] = d(s, v).
(3) All nodes v where d(s, v) $> n$ have dist[v] = ∞

Let k be the maximum finite distance of any node from node s.

Theorem: Breadth-first search always terminates with dist[v] = d(s, v) for all $v \in V$.

Proof: Define "round n" of BFS to be an instance where at the start of the loop, all nodes v in the queue satisfy dist[v] = n. We will prove in an lemma the following are always true after the first n rounds:

(1) For any node v, d(s, v) = n iff v is in the queue.
(2) All nodes v where d(s, v) ≤ n have dist[v] = d(s, v).
(3) All nodes v where d(s, v) > n have dist[v] = ∞

Let k be the maximum finite distance of any node from node s. Note the following:

Theorem: Breadth-first search always terminates with dist[v] = d(s, v) for all $v \in V$.

Proof: Define "round n" of BFS to be an instance where at the start of the loop, all nodes v in the queue satisfy dist[v] = n. We will prove in an lemma the following are always true after the first n rounds:

(1) For any node v, d(s, v) = n iff v is in the queue.
(2) All nodes v where d(s, v) ≤ n have dist[v] = d(s, v).
(3) All nodes v where d(s, v) > n have dist[v] = ∞

Let k be the maximum finite distance of any node from node s. Note the following:

- Any node v where d(s, v) is finite satisfies d(s, v) ≤ k, and any node v where d(s, v) > k satisfies d(s, v) = ∞. This follows from the fact that we picked the maximum possible finite k.

Theorem: Breadth-first search always terminates with dist[v] = d(s, v) for all $v \in V$.

Proof: Define "round n" of BFS to be an instance where at the start of the loop, all nodes v in the queue satisfy dist[v] = n. We will prove in an lemma the following are always true after the first n rounds:

(1) For any node v, d(s, v) = n iff v is in the queue.
(2) All nodes v where d(s, v) $\leq n$ have dist[v] = d(s, v).
(3) All nodes v where d(s, v) $> n$ have dist[v] = ∞

Let k be the maximum finite distance of any node from node s. Note the following:

- Any node v where d(s, v) is finite satisfies d(s, v) $\leq k$, and any node v where d(s, v) $> k$ satisfies d(s, v) = ∞. This follows from the fact that we picked the maximum possible finite k.
- There must be nodes at distances 0, 1, 2, …, k from s. A simple inductive argument using property (1) shows that there will be exactly $k + 1$ rounds, corresponding to distances 0, 1, …, k.

Theorem: Breadth-first search always terminates with $\mathrm{dist}[v] = \mathrm{d}(s, v)$ for all $v \in V$.

Proof: Define "round n" of BFS to be an instance where at the start of the loop, all nodes v in the queue satisfy $\mathrm{dist}[v] = n$. We will prove in an lemma the following are always true after the first n rounds:

(1) For any node v, $\mathrm{d}(s, v) = n$ iff v is in the queue.
(2) All nodes v where $\mathrm{d}(s, v) \leq n$ have $\mathrm{dist}[v] = \mathrm{d}(s, v)$.
(3) All nodes v where $\mathrm{d}(s, v) > n$ have $\mathrm{dist}[v] = \infty$

Let k be the maximum finite distance of any node from node s. Note the following:

- Any node v where $\mathrm{d}(s, v)$ is finite satisfies $\mathrm{d}(s, v) \leq k$, and any node v where $d(s, v) > k$ satisfies $\mathrm{d}(s, v) = \infty$. This follows from the fact that we picked the maximum possible finite k.

- There must be nodes at distances $0, 1, 2, \ldots, k$ from s. A simple inductive argument using property (1) shows that there will be exactly $k + 1$ rounds, corresponding to distances $0, 1, \ldots, k$.

So consider $\mathrm{dist}[v]$ for any node v after the algorithm terminates (that is, after $k+1$ rounds).

Theorem: Breadth-first search always terminates with dist[v] = d(s, v) for all $v \in V$.

Proof: Define "round n" of BFS to be an instance where at the start of the loop, all nodes v in the queue satisfy dist[v] = n. We will prove in an lemma the following are always true after the first n rounds:

(1) For any node v, d(s, v) = n iff v is in the queue.
(2) All nodes v where d(s, v) $\leq n$ have dist[v] = d(s, v).
(3) All nodes v where d(s, v) $> n$ have dist[v] = ∞

Let k be the maximum finite distance of any node from node s. Note the following:

- Any node v where d(s, v) is finite satisfies d(s, v) $\leq k$, and any node v where d(s, v) $> k$ satisfies d(s, v) = ∞. This follows from the fact that we picked the maximum possible finite k.
- There must be nodes at distances 0, 1, 2, …, k from s. A simple inductive argument using property (1) shows that there will be exactly k + 1 rounds, corresponding to distances 0, 1, …, k.

So consider dist[v] for any node v after the algorithm terminates (that is, after k+1 rounds). If d(s, v) is finite, then d(s, v) $\leq k \leq k$+1, and so by (1) we have dist[v] = d(s, v).

Theorem: Breadth-first search always terminates with dist[v] = d(s, v) for all $v \in V$.

Proof: Define "round n" of BFS to be an instance where at the start of the loop, all nodes v in the queue satisfy dist[v] = n. We will prove in an lemma the following are always true after the first n rounds:

(1) For any node v, d(s, v) = n iff v is in the queue.
(2) All nodes v where d(s, v) $\leq n$ have dist[v] = d(s, v).
(3) All nodes v where d(s, v) $> n$ have dist[v] = ∞

Let k be the maximum finite distance of any node from node s. Note the following:

- Any node v where d(s, v) is finite satisfies d(s, v) $\leq k$, and any node v where d(s, v) $> k$ satisfies d(s, v) = ∞. This follows from the fact that we picked the maximum possible finite k.
- There must be nodes at distances 0, 1, 2, …, k from s. A simple inductive argument using property (1) shows that there will be exactly $k + 1$ rounds, corresponding to distances 0, 1, …, k.

So consider dist[v] for any node v after the algorithm terminates (that is, after k+1 rounds). If d(s, v) is finite, then d(s, v) $\leq k \leq k$+1, and so by (1) we have dist[v] = d(s, v). If d(s, v) = ∞, then d(s, v) $> k + 1$, so by (2) we have dist[v] = ∞.

Theorem: Breadth-first search always terminates with dist[v] = d(s, v) for all $v \in V$.

Proof: Define "round n" of BFS to be an instance where at the start of the loop, all nodes v in the queue satisfy dist[v] = n. We will prove in an lemma the following are always true after the first n rounds:

(1) For any node v, d(s, v) = n iff v is in the queue.
(2) All nodes v where d(s, v) $\leq n$ have dist[v] = d(s, v).
(3) All nodes v where d(s, v) $> n$ have dist[v] = ∞

Let k be the maximum finite distance of any node from node s. Note the following:

- Any node v where d(s, v) is finite satisfies d(s, v) $\leq k$, and any node v where d(s, v) $> k$ satisfies d(s, v) = ∞. This follows from the fact that we picked the maximum possible finite k.

- There must be nodes at distances 0, 1, 2, …, k from s. A simple inductive argument using property (1) shows that there will be exactly $k + 1$ rounds, corresponding to distances 0, 1, …, k.

So consider dist[v] for any node v after the algorithm terminates (that is, after k+1 rounds). If d(s, v) is finite, then d(s, v) $\leq k \leq k$+1, and so by (1) we have dist[v] = d(s, v). If d(s, v) = ∞, then d(s, v) $> k + 1$, so by (2) we have dist[v] = ∞. Thus d(s, v) = dist[v] for all $v \in V$, as required.

Theorem: Breadth-first search always terminates with dist[v] = d(s, v) for all $v \in V$.

Proof: Define "round n" of BFS to be an instance where at the start of the loop, all nodes v in the queue satisfy dist[v] = n. We will prove in an lemma the following are always true after the first n rounds:

(1) For any node v, d(s, v) = n iff v is in the queue.
(2) All nodes v where d(s, v) ≤ n have dist[v] = d(s, v).
(3) All nodes v where d(s, v) > n have dist[v] = ∞

Let k be the maximum finite distance of any node from node s. Note the following:

- Any node v where d(s, v) is finite satisfies d(s, v) ≤ k, and any node v where d(s, v) > k satisfies d(s, v) = ∞. This follows from the fact that we picked the maximum possible finite k.

- There must be nodes at distances 0, 1, 2, …, k from s. A simple inductive argument using property (1) shows that there will be exactly k + 1 rounds, corresponding to distances 0, 1, …, k.

So consider dist[v] for any node v after the algorithm terminates (that is, after k+1 rounds). If d(s, v) is finite, then d(s, v) ≤ k ≤ k+1, and so by (1) we have dist[v] = d(s, v). If d(s, v) = ∞, then d(s, v) > k + 1, so by (2) we have dist[v] = ∞. Thus d(s, v) = dist[v] for all $v \in V$, as required. ■

Lemma: After n rounds, the following hold:

(1) For any node v, $d(s, v) = n$ iff v is in the queue.
(2) All nodes v where $d(s, v) \leq n$ have $dist[v] = d(s, v)$.
(3) All nodes v where $d(s, v) > n$ have $dist[v] = \infty$

Proof: By induction n. After 0 rounds, $dist[s] = 0$, $dist[v] = \infty$ for any $v \neq s$, and the queue holds only s. Since s is the only node at distance 0, (1) – (3) hold.

For the inductive step, assume for some n that (1) – (3) hold after n rounds. We will prove (1) – (3) hold after $n + 1$ rounds. We need to show the following:

(a) For any node v, $d(s, v) = n + 1$ iff v is in the queue.
(b) All nodes v where $d(s, v) \leq n + 1$ have $dist[v] = d(s, v)$.
(c) All nodes v where $d(s, v) > n + 1$ have $dist[v] = \infty$

To prove (a), note that at the end of round n, all nodes of distance n will have been dequeued, so we need to show all nodes v where $d(s, v) = n + 1$ are enqueued and nothing else is. Note that if a node u is enqueued in round $n + 1$, then at the start of round $n + 1$ $dist[u] = \infty$ (so by (2) and (3), its distance is at least $n + 1$) and u must have been adjacent to a node v in the queue (by (1), $d(s, v) = n$). Thus there is a path of length $n + 1$ to u (take the path of length n to v, then follow the edge to u), and there is no shorter path, so this is the shortest path to u. Thus, $d(s, u) = n + 1$. Also note that if a node u satisfies $d(s, u) = n + 1$, then by (3) at the start of round $n + 1$ it must have $dist[u] = \infty$. Also, it must be adjacent to some node at distance n, which by (1) must be in the queue at the start of the round. Thus at the end of round $n + 1$, u will be enqueued and $dist[u]$ set to $n + 1$.

By our above argument, we know that (a) must hold. Since we didn't change any dist values for nodes at distance n or less, and we set dist values for all enqueued nodes to $n + 1$, (b) holds. Finally, since we only changed labels for nodes at distance $n + 1$, (c) holds as well. This completes the induction. ■

Question 1: How do we prove this always finds the right distances?

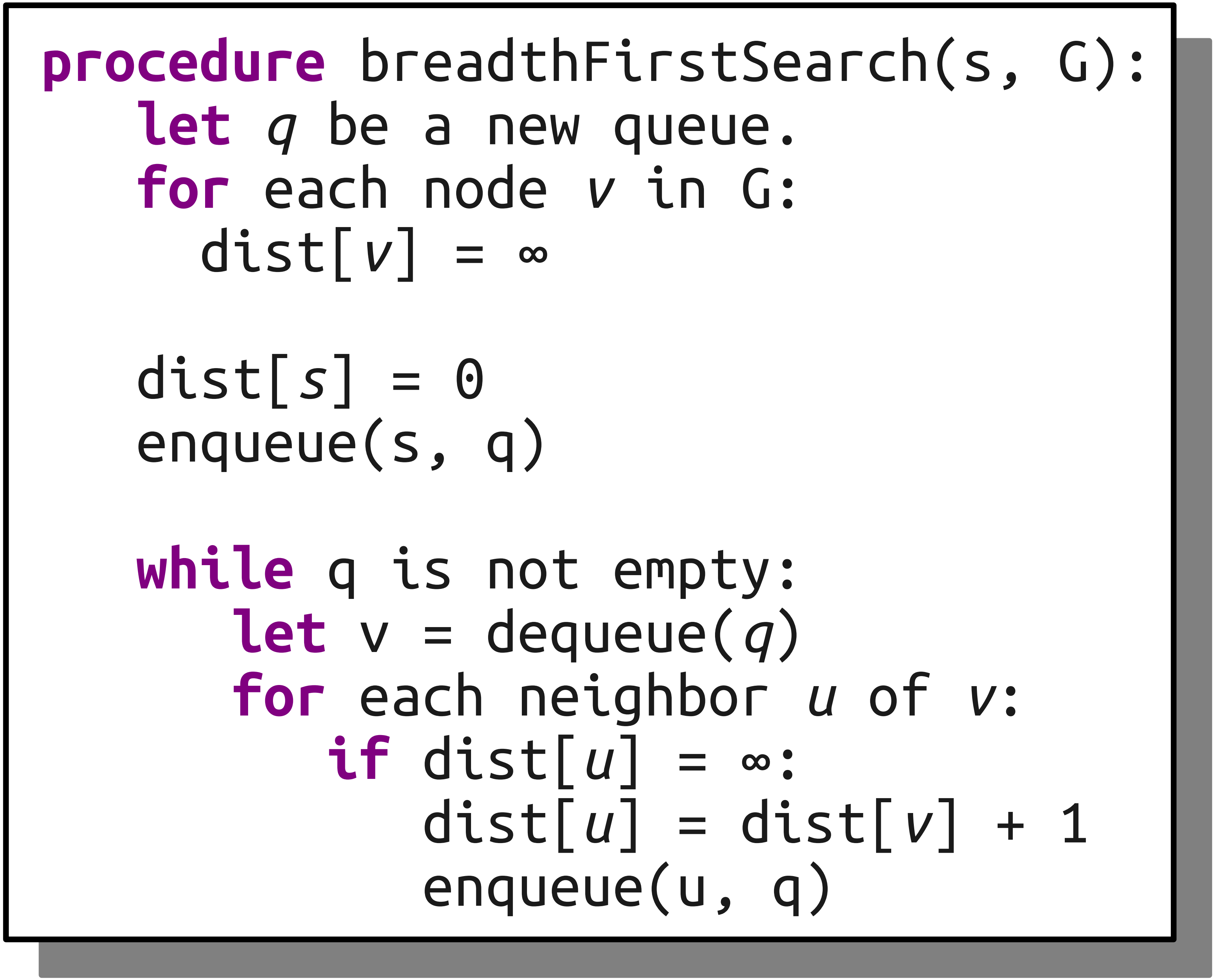
Question 2: How *efficiently* does this find the right distances?

Question 1: How do we prove this always finds the right distances?

Question 2: How *efficiently* does this find the right distances?

Graph Terminology

- When analyzing algorithms on a graph, there are (usually) two parameters we care about:
  - The number of nodes, denoted ***n***. ($n = |V|$)
  - The number of edges, denoted ***m***. ($m = |E|$)
- Note that $m = O(n^2)$. *(Why?)*
- A graph is called **dense** if $m = \Theta(n^2)$. A graph is called **sparse** if it is not dense.



```
procedure breadthFirstSearch(s, G):
   let q be a new queue.
   for each node v in G:
      dist[v] = ∞

   dist[s] = 0
   enqueue(s, q)

   while q is not empty:
      let v = dequeue(q)
      for each neighbor u of v:
         if dist[u] = ∞:
            dist[u] = dist[v] + 1
            enqueue(u, q)
```

```
procedure breadthFirstSearch(s, G):
   let q be a new queue.
   for each node v in G:
     dist[v] = ∞

   dist[s] = 0
   enqueue(s, q)

   while q is not empty:
      let v = dequeue(q)
      for each neighbor u of v:
         if dist[u] = ∞:
            dist[u] = dist[v] + 1
            enqueue(u, q)
```

O(1)

```
procedure breadthFirstSearch(s, G):
    let q be a new queue.
    for each node v in G:
      dist[v] = ∞

    dist[s] = 0
    enqueue(s, q)

    while q is not empty:
        let v = dequeue(q)
        for each neighbor u of v:
            if dist[u] = ∞:
               dist[u] = dist[v] + 1
               enqueue(u, q)
```

O(1)

```
procedure breadthFirstSearch(s, G):
   let q be a new queue.
   for each node v in G:
      dist[v] = ∞

   dist[s] = 0
   enqueue(s, q)

   while q is not empty:
      let v = dequeue(q)
      for each neighbor u of v:
         if dist[u] = ∞:
            dist[u] = dist[v] + 1
            enqueue(u, q)
```

```
       procedure breadthFirstSearch(s, G):
O(1)      let q be a new queue.
O(n)      for each node v in G:
             dist[v] = ∞

          dist[s] = 0
          enqueue(s, q)

          while q is not empty:
             let v = dequeue(q)
             for each neighbor u of v:
                if dist[u] = ∞:
                   dist[u] = dist[v] + 1
                   enqueue(u, q)
       procedure breadthFirstSearch(s, G):
O(1)      let q be a new queue.
O(n)      for each node v in G:
             dist[v] = ∞

          dist[s] = 0
          enqueue(s, q)

          while q is not empty:
             let v = dequeue(q)
             for each neighbor u of v:
                if dist[u] = ∞:
                   dist[u] = dist[v] + 1
                   enqueue(u, q)
       procedure breadthFirstSearch(s, G):
O(1)      let q be a new queue.
O(n)      for each node v in G:
             dist[v] = ∞

O(1)      dist[s] = 0
          enqueue(s, q)

          while q is not empty:
             let v = dequeue(q)
             for each neighbor u of v:
                if dist[u] = ∞:
                   dist[u] = dist[v] + 1
                   enqueue(u, q)
```

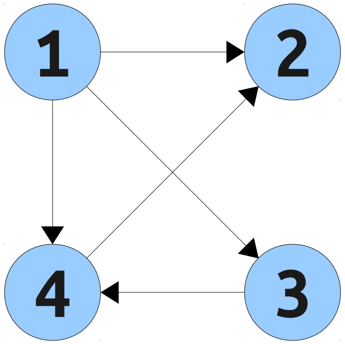
```
        procedure breadthFirstSearch(s, G):
O(1)        let q be a new queue.
O(n)        for each node v in G:
                dist[v] = ∞

O(1)        dist[s] = 0
            enqueue(s, q)

            while q is not empty:
                let v = dequeue(q)
                for each neighbor u of v:
                    if dist[u] = ∞:
                        dist[u] = dist[v] + 1
                        enqueue(u, q)
```

```
       procedure breadthFirstSearch(s, G):
O(1)      let q be a new queue.
O(n)      for each node v in G:
            dist[v] = ∞

O(1)      dist[s] = 0
          enqueue(s, q)

          while q is not empty:
             let v = dequeue(q)
             for each neighbor u of v:
                if dist[u] = ∞:
                   dist[u] = dist[v] + 1
                   enqueue(u, q)
       procedure breadthFirstSearch(s, G):
O(1)       let q be a new queue.
O(n)       for each node v in G:
             dist[v] = ∞

O(1)       dist[s] = 0
           enqueue(s, q)

           while q is not empty:
              let v = dequeue(q)
              for each neighbor u of v:
                 if dist[u] = ∞:
                    dist[u] = dist[v] + 1
                    enqueue(u, q)
```

How are our graphs represented?

Adjacency Matrices

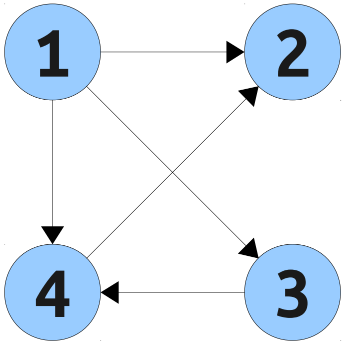
- An **adjacency matrix** is a representation of a graph as an $n \times n$ matrix M of 0s and 1s, where
  - $M_{uv} = 1$ if $(u, v) \in E$.
  - $M_{uv} = 0$ otherwise.



$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Adjacency Matrices

- An **adjacency matrix** is a representation of a graph as an $n \times n$ matrix M of 0s and 1s, where
  - $M_{uv} = 1$ if $(u, v) \in E$.
  - $M_{uv} = 0$ otherwise.

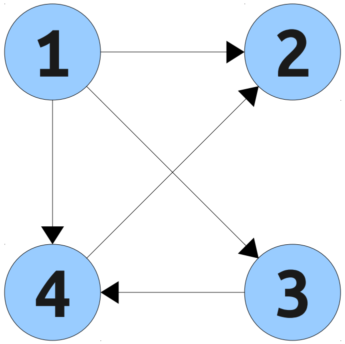


$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

- Memory usage:

Adjacency Matrices

- An **adjacency matrix** is a representation of a graph as an $n \times n$ matrix M of 0s and 1s, where
  - $M_{uv} = 1$ if $(u, v) \in E$.
  - $M_{uv} = 0$ otherwise.

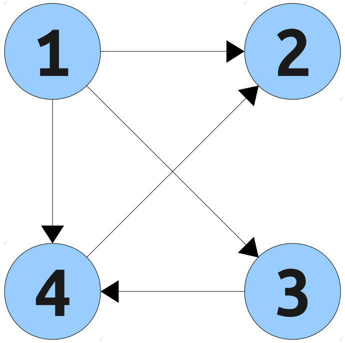


$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

- Memory usage: **$\Theta(n^2)$**.

Adjacency Matrices

- An **adjacency matrix** is a representation of a graph as an $n \times n$ matrix M of 0s and 1s, where
  - $M_{uv} = 1$ if $(u, v) \in E$.
  - $M_{uv} = 0$ otherwise.

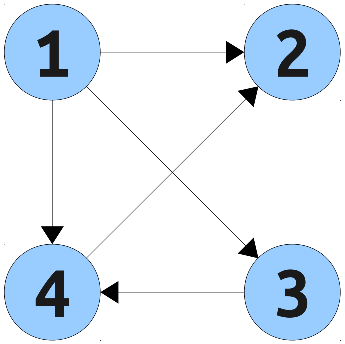


$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

- Memory usage: **$\Theta(n^2)$**.
- Time to check if an edge exists:

Adjacency Matrices

- An **adjacency matrix** is a representation of a graph as an $n \times n$ matrix M of 0s and 1s, where
  - $M_{uv} = 1$ if $(u, v) \in E$.
  - $M_{uv} = 0$ otherwise.

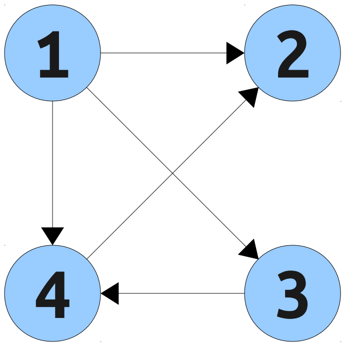


$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

- Memory usage: $\boldsymbol{\Theta(n^2)}$.
- Time to check if an edge exists: **O(1)**

Adjacency Matrices

- An **adjacency matrix** is a representation of a graph as an $n \times n$ matrix M of 0s and 1s, where
  - $M_{uv} = 1$ if $(u, v) \in E$.
  - $M_{uv} = 0$ otherwise.

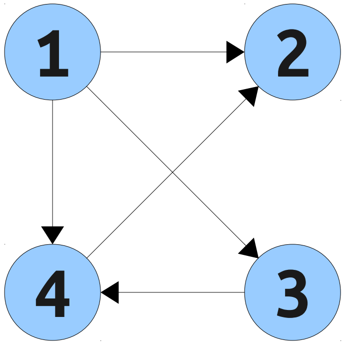


$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

- Memory usage: **$\Theta(n^2)$**.
- Time to check if an edge exists: **O(1)**
- Time to find all outgoing edges for a node:

Adjacency Matrices

- An **adjacency matrix** is a representation of a graph as an $n \times n$ matrix M of 0s and 1s, where
    - $M_{uv} = 1$ if $(u, v) \in E$.
    - $M_{uv} = 0$ otherwise.



$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

- Memory usage: **$\Theta(n^2)$**.
- Time to check if an edge exists: **O(1)**
- Time to find all outgoing edges for a node: **$\Theta(n)$**

```
      procedure breadthFirstSearch(s, G):
O(1)      let q be a new queue.
O(n)      for each node v in G:
            dist[v] = ∞

O(1)      dist[s] = 0
          enqueue(s, q)

          while q is not empty:
              let v = dequeue(q)
              for each neighbor u of v:
                  if dist[u] = ∞:
                      dist[u] = dist[v] + 1
                      enqueue(u, q)
      procedure breadthFirstSearch(s, G):
O(1)      let q be a new queue.
O(n)      for each node v in G:
              dist[v] = ∞

O(1)      dist[s] = 0
          enqueue(s, q)

          while q is not empty:
              let v = dequeue(q)
            ⎧ for each neighbor u of v:
Θ(n)       ⎨     if dist[u] = ∞:
            ⎪         dist[u] = dist[v] + 1
            ⎩         enqueue(u, q)
```

```
         procedure breadthFirstSearch(s, G):
O(1)         let q be a new queue.
O(n)         for each node v in G:
                 dist[v] = ∞

O(1)         dist[s] = 0
             enqueue(s, q)

             while q is not empty:
                 let v = dequeue(q)
                 for each neighbor u of v:
Θ(n)                 if dist[u] = ∞:
                         dist[u] = dist[v] + 1
                         enqueue(u, q)
```

```
            procedure breadthFirstSearch(s, G):
$O(1)$          let q be a new queue.
$O(n)$          for each node v in G:
                    dist[v] = ∞

$O(1)$          dist[s] = 0
                enqueue(s, q)

$O(n^2)$        while q is not empty:
                    let v = dequeue(q)
                    for each neighbor u of v:
$\Theta(n)$             if dist[u] = ∞:
                            dist[u] = dist[v] + 1
                            enqueue(u, q)
```

$O(1)$

$O(n)$

$O(1)$

$+O(n^2)$

$\Theta(n)$

```
procedure breadthFirstSearch(s, G):
    let q be a new queue.
    for each node v in G:
        dist[v] = ∞

    dist[s] = 0
    enqueue(s, q)

    while q is not empty:
        let v = dequeue(q)
        for each neighbor u of v:
            if dist[u] = ∞:
                dist[u] = dist[v] + 1
                enqueue(u, q)
```

```
             procedure breadthFirstSearch(s, G):
$O(1)$           let $q$ be a new queue.
$O(n)$           for each node $v$ in G:
                     dist[$v$] = ∞

$O(1)$           dist[$s$] = 0
                 enqueue(s, q)

$+O(n^2)$        while q is not empty:
$O(n^2)$             let v = dequeue($q$)
                     for each neighbor $u$ of $v$:
        $\Theta(n)$          if dist[$u$] = ∞:
                             dist[$u$] = dist[$v$] + 1
                             enqueue(u, q)
```

```
procedure breadthFirstSearch(s, G):
    let q be a new queue.
    for each node v in G:
        dist[v] = ∞

    dist[s] = 0
    enqueue(s, q)

    while q is not empty:
        let v = dequeue(q)
        for each neighbor u of v:
            if dist[u] = ∞:
                dist[u] = dist[v] + 1
                enqueue(u, q)
```

$O(1)$

$O(n)$

$O(1)$

$+O(n^2)$

$O(n^2)$

$\Theta(n)$

Why isn't the runtime $\Theta(n^2)$?

Linear Time on Graphs

- With an adjacency matrix, BFS runs in time **$O(n^2)$**. Is that efficient?
- In a graph with n nodes and m edges, we say that an algorithm runs in **linear time** iff the algorithm runs in time $O(m + n)$.
  - This is linear in the number of “pieces” of the graph, which is the number of nodes plus the number of edges.
- On a dense graph, this implementation of BFS runs in linear time:

$$O(n^2) = O(n^2 + n) = O(m + n)$$

- On sparser graphs (say, $m = O(n)$), though, this is not linear time:

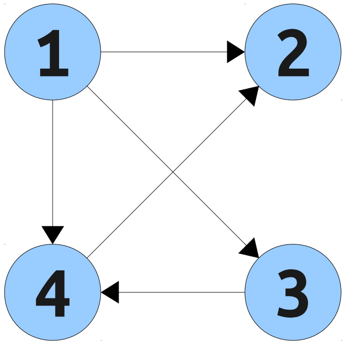
$$O(n^2) \neq O(n) = O(m + n)$$

The Issue

- Our algorithm is slow because this step always takes Θ(n) time:

```
for each neighbor u of v:
```

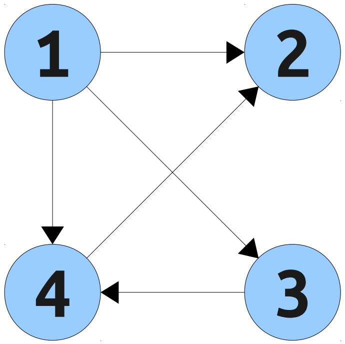
- Can we refine our data structure for storing the graph so that we can easily find all edges incident to a node?

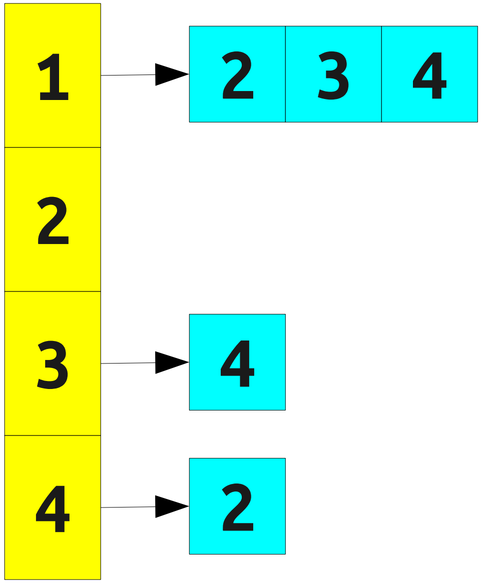


$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Adjacency Lists

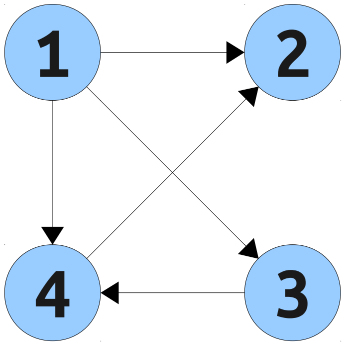
- An **adjacency list** is a representation of a graph as an array A of n lists. The list $A[u]$ holds all nodes v where (u, v) is an edge.

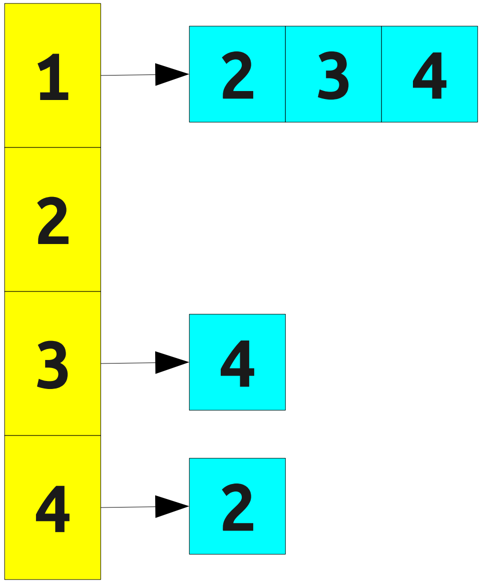




Adjacency Lists

- An **adjacency list** is a representation of a graph as an array *A* of *n* lists. The list *A*[*u*] holds all nodes *v* where (*u*, *v*) is an edge.

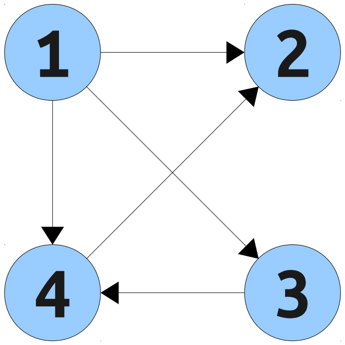


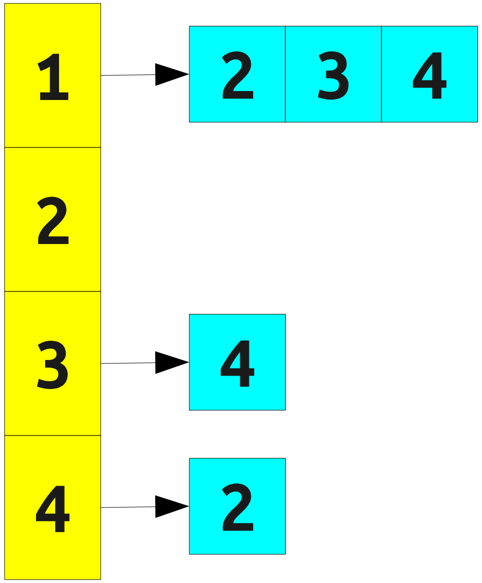


- Memory usage:

Adjacency Lists

- An **adjacency list** is a representation of a graph as an array *A* of *n* lists. The list *A*[*u*] holds all nodes *v* where (*u*, *v*) is an edge.

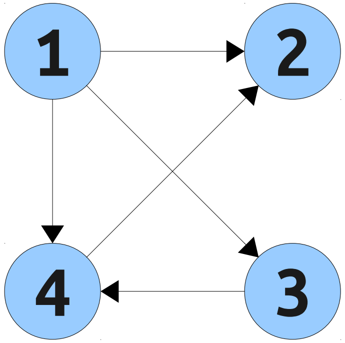


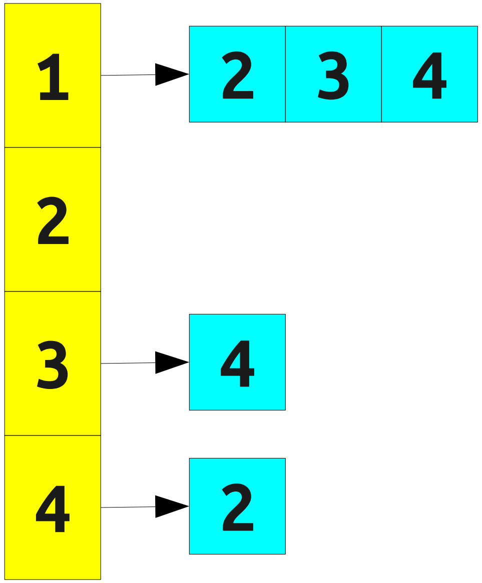


- Memory usage: **Θ(*n* + *m*)**.

Adjacency Lists

- An **adjacency list** is a representation of a graph as an array A of n lists. The list $A[u]$ holds all nodes v where (u, v) is an edge.

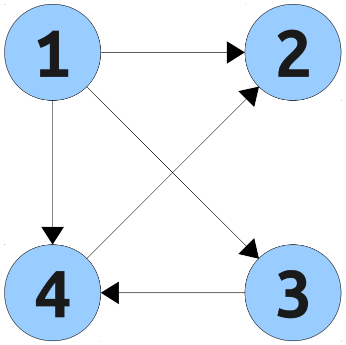


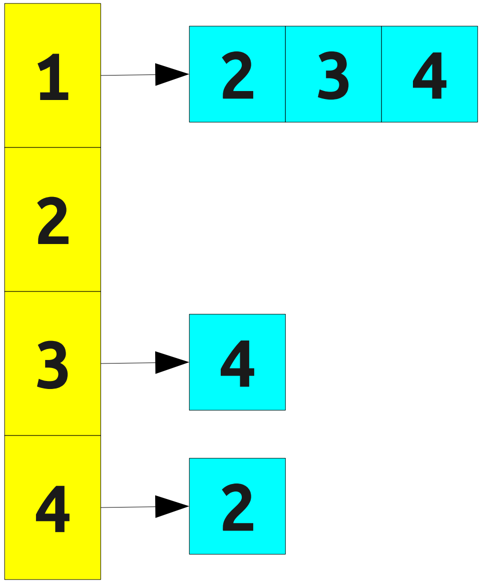


- Memory usage: $\boldsymbol{\Theta(n + m)}$.
- Time to check if edge (u, v) exists:

Adjacency Lists

- An **adjacency list** is a representation of a graph as an array *A* of *n* lists. The list *A*[*u*] holds all nodes *v* where (*u*, *v*) is an edge.

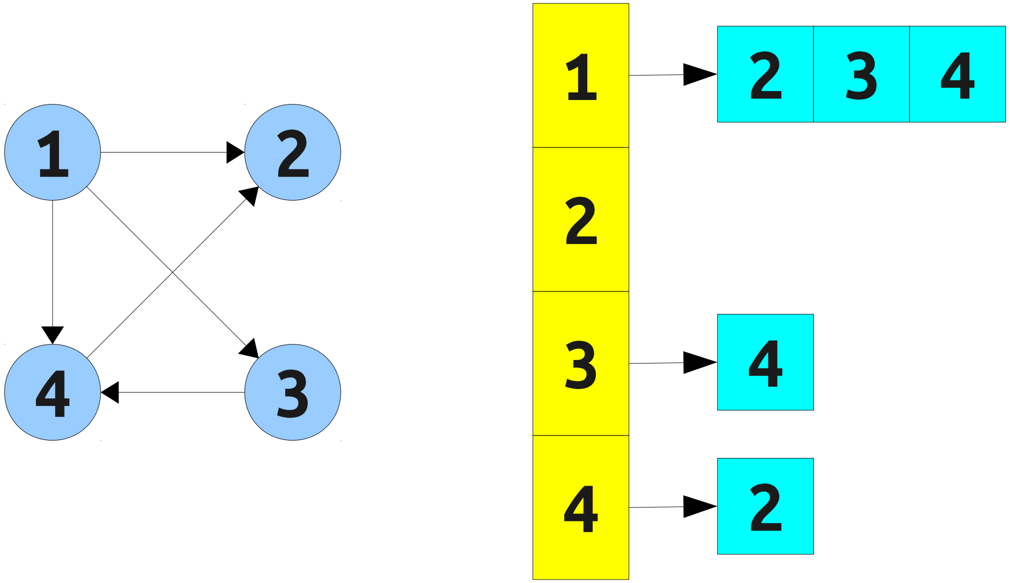




- Memory usage: **$\Theta(n + m)$**.
- Time to check if edge (*u*, *v*) exists: **$O(\deg^+(u))$**

Adjacency Lists

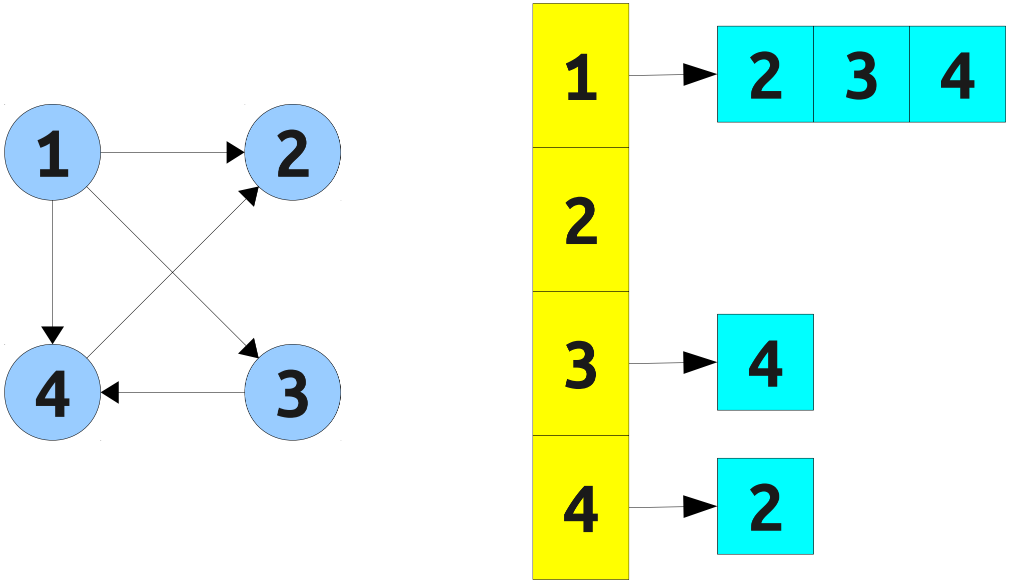
- An **adjacency list** is a representation of a graph as an array A of n lists. The list $A[u]$ holds all nodes v where (u, v) is an edge.



- Memory usage: $\boldsymbol{\Theta(n + m)}$.
- Time to check if edge (u, v) exists: $\mathbf{O(deg^+(}\boldsymbol{u}\mathbf{))}$
- Time to find all outgoing edges for a node u:

Adjacency Lists

- An **adjacency list** is a representation of a graph as an array A of n lists. The list $A[u]$ holds all nodes v where (u, v) is an edge.



- Memory usage: $\boldsymbol{\Theta(n + m)}$.
- Time to check if edge (u, v) exists: $\mathbf{O(deg^+(}\boldsymbol{u}\mathbf{))}$
- Time to find all outgoing edges for a node u: $\boldsymbol{\Theta}\mathbf{(deg^+(}\boldsymbol{u}\mathbf{))}$

```
       procedure breadthFirstSearch(s, G):
O(1)      let q be a new queue.
O(n)      for each node v in G:
            dist[v] = ∞

O(1)      dist[s] = 0
          enqueue(s, q)

          while q is not empty:
             let v = dequeue(q)
             for each neighbor u of v:
                if dist[u] = ∞:
                   dist[u] = dist[v] + 1
                   enqueue(u, q)
```

```
        procedure breadthFirstSearch(s, G):
O(1)        let q be a new queue.
O(n)        for each node v in G:
                dist[v] = ∞

O(1)        dist[s] = 0
            enqueue(s, q)

            while q is not empty:
                let v = dequeue(q)
                for each neighbor u of v:
                    if dist[u] = ∞:
O(n)                    dist[u] = dist[v] + 1
                        enqueue(u, q)
```

```
      procedure breadthFirstSearch(s, G):
O(1)     let q be a new queue.
O(n)     for each node v in G:
            dist[v] = ∞

O(1)     dist[s] = 0
         enqueue(s, q)

         while q is not empty:
            let v = dequeue(q)
            for each neighbor u of v:
O(n)           if dist[u] = ∞:
                  dist[u] = dist[v] + 1
                  enqueue(u, q)
```

```
          procedure breadthFirstSearch(s, G):
$O(1)$        let q be a new queue.
$O(n)$        for each node v in G:
                 dist[v] = ∞

$O(1)$        dist[s] = 0
              enqueue(s, q)

$O(n^2)$      while q is not empty:
                 let v = dequeue(q)
                 for each neighbor u of v:
                    if dist[u] = ∞:
          $O(n)$       dist[u] = dist[v] + 1
                       enqueue(u, q)
```

```
            procedure breadthFirstSearch(s, G):
$O(1)$          let q be a new queue.
$O(n)$          for each node v in G:
                    dist[v] = ∞

$O(1)$          dist[s] = 0
                enqueue(s, q)

$+O(n^2)$       while q is not empty:
                    let v = dequeue(q)
                    for each neighbor u of v:
      $O(n)$            if dist[u] = ∞:
                            dist[u] = dist[v] + 1
                            enqueue(u, q)
```

```
               procedure breadthFirstSearch(s, G):
$O(1)$             let q be a new queue.
$O(n)$             for each node v in G:
                       dist[v] = ∞

$O(1)$             dist[s] = 0
                   enqueue(s, q)

$+O(n^2)$          while q is not empty:
                       let v = dequeue(q)
$O(n^2)$               for each neighbor u of v:
                           if dist[u] = ∞:
          $O(n)$               dist[u] = dist[v] + 1
                               enqueue(u, q)
```

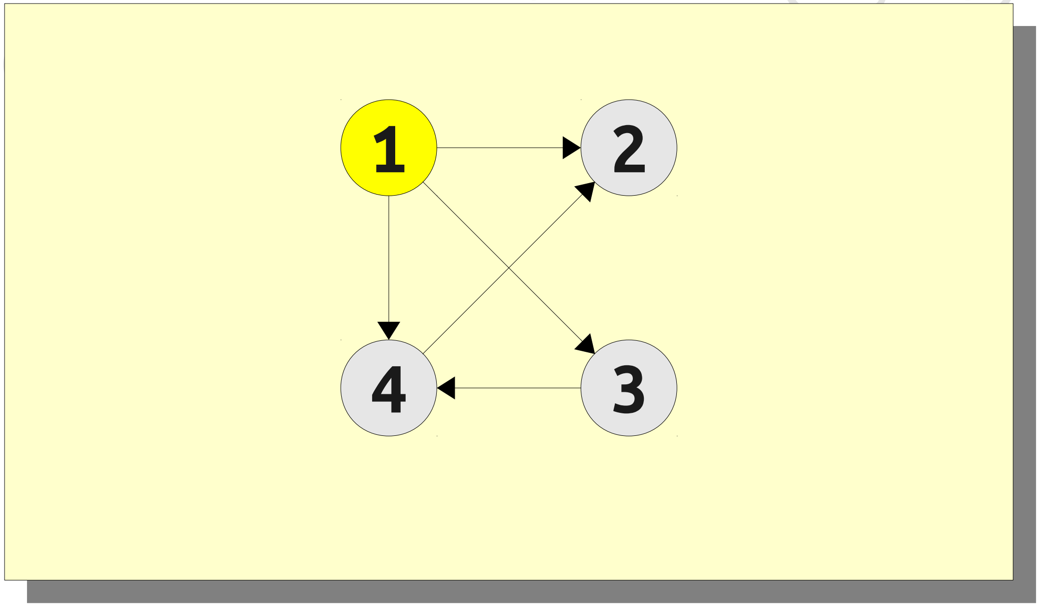
A Better Analysis

```
       procedure breadthFirstSearch(s, G):
O(1)      let q be a new queue.
O(n)      for each node v in G:
            dist[v] = ∞

O(1)      dist[s] = 0
          enqueue(s, q)

          while q is not empty:
             let v = dequeue(q)
             for each neighbor u of v:
                if dist[u] = ∞:
                   dist[u] = dist[v] + 1
                   enqueue(u, q)
```

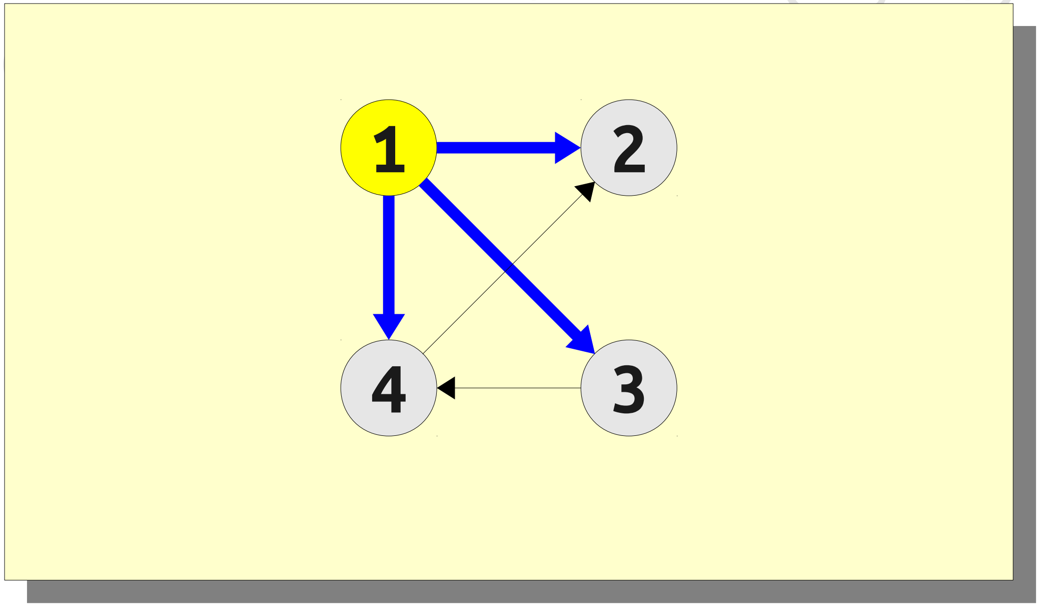
```
     procedure breadthFirstSearch(s, G):
O(1)    let q be a new queue.
O(n)    for each node v in G:
          dist[v] = ∞

O(1)    dist[s] = 0
        enqueue(s, q)

        while q is not empty:
           let v = dequeue(q)
           for each neighbor u of v:
              if dist[u] = ∞:
                 dist[u] = dist[v] + 1
                 enqueue(u, q)
     procedure breadthFirstSearch(s, G):
O(1)     let q be a new queue.
O(n)     for each node v in G:
           dist[v] = ∞

O(1)     dist[s] = 0
         enqueue(s, q)

O(n)     while q is not empty:
             let v = dequeue(q)
             for each neighbor u of v:
                 if dist[u] = ∞:
                     dist[u] = dist[v] + 1
                     enqueue(u, q)
```

```
       procedure breadthFirstSearch(s, G):
O(1)     let q be a new queue.
O(n)     for each node v in G:
           dist[v] = ∞

O(1)     dist[s] = 0
         enqueue(s, q)

O(n)     while q is not empty:
           let v = dequeue(q)
           for each neighbor u of v:
             if dist[u] = ∞:
               dist[u] = dist[v] + 1
               enqueue(u, q)
```

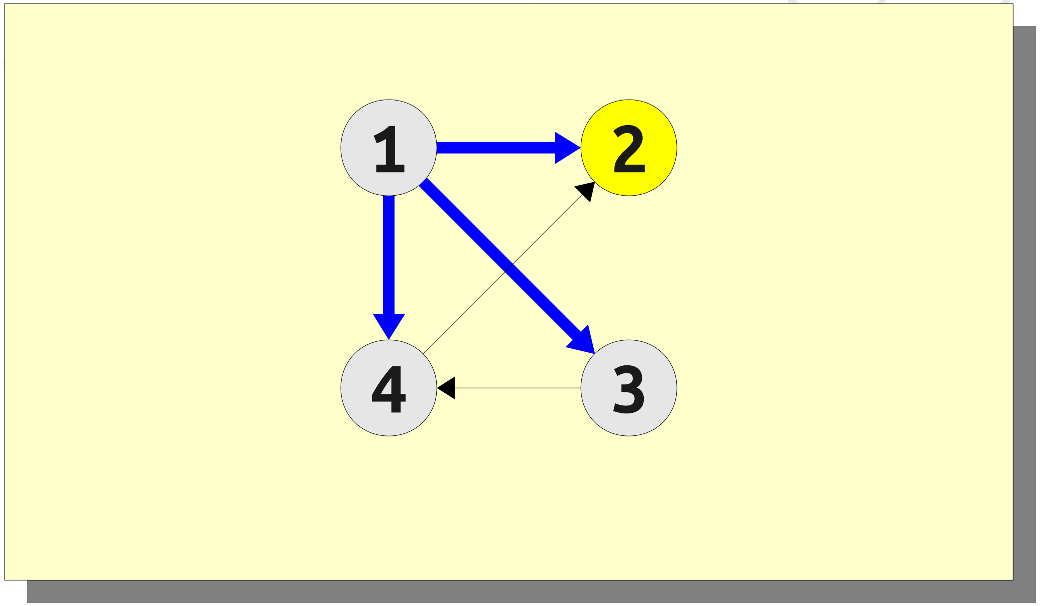
```
        procedure breadthFirstSearch(s, G):
O(1)        let q b
O(n)        for eac
              dist[

O(1)        dist[s]
            enqueue

O(n)        while q is not empty:
                let v = dequeue(q)
                for each neighbor u of v:
                    if dist[u] = ∞:
                        dist[u] = dist[v] + 1
                        enqueue(u, q)
```



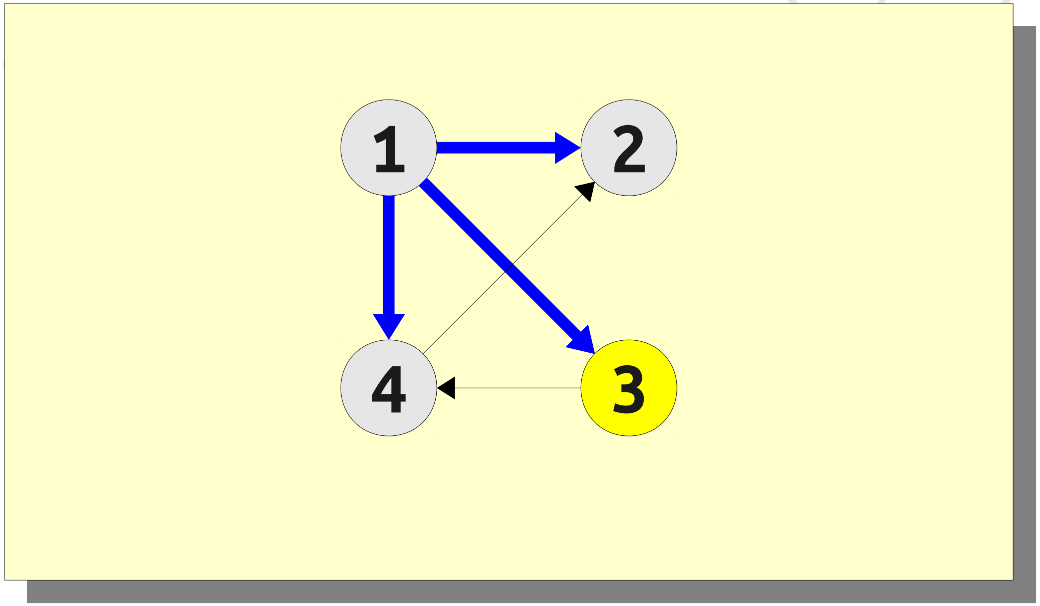
```
        procedure breadthFirstSearch(s, G):
$O(1)$      let q b
$O(n)$      for eac
                dist[

$O(1)$      dist[s]
            enqueue

$O(n)$      while q is not empty:
                let v = dequeue(q)
                for each neighbor u of v:
                    if dist[u] = ∞:
                        dist[u] = dist[v] + 1
                        enqueue(u, q)
```



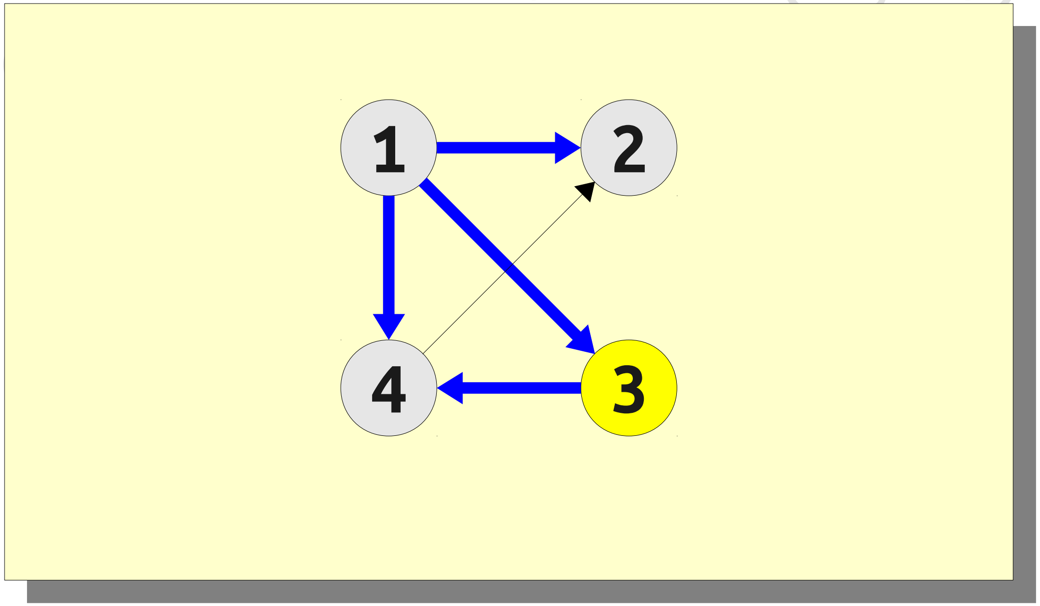
```
       procedure breadthFirstSearch(s, G):
O(1)       let q b
O(n)       for eac
              dist[

           dist[s]
O(1)       enqueue

O(n)       while q is not empty:
              let v = dequeue(q)
              for each neighbor u of v:
                 if dist[u] = ∞:
                    dist[u] = dist[v] + 1
                    enqueue(u, q)
```



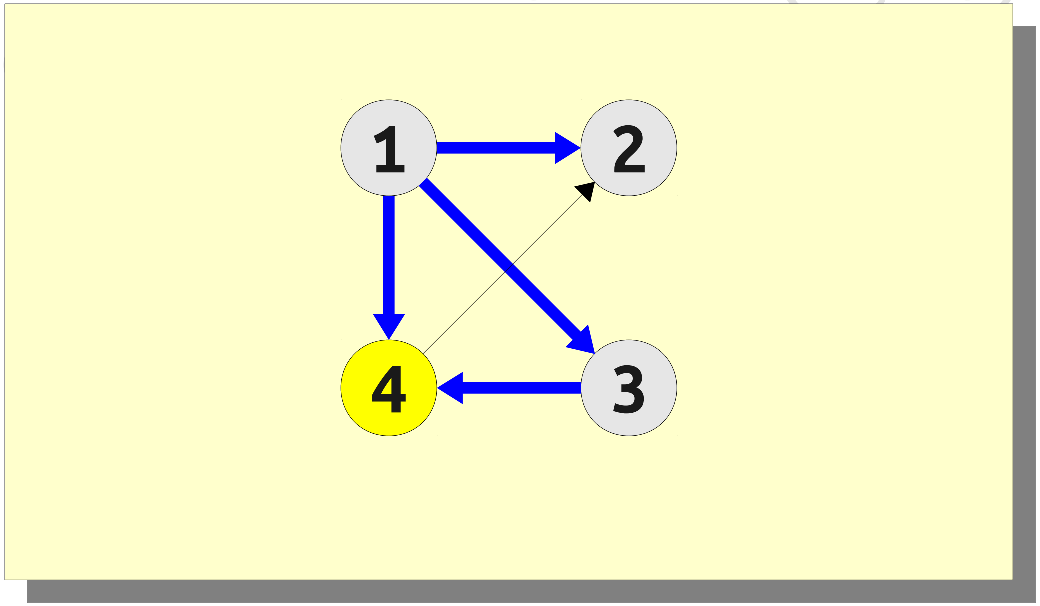
```
       procedure breadthFirstSearch(s, G):
O(1)       let q b
O(n)       for eac
               dist[

O(1)       dist[s]
           enqueue

O(n)       while q is not empty:
               let v = dequeue(q)
               for each neighbor u of v:
                   if dist[u] = ∞:
                       dist[u] = dist[v] + 1
                       enqueue(u, q)
```



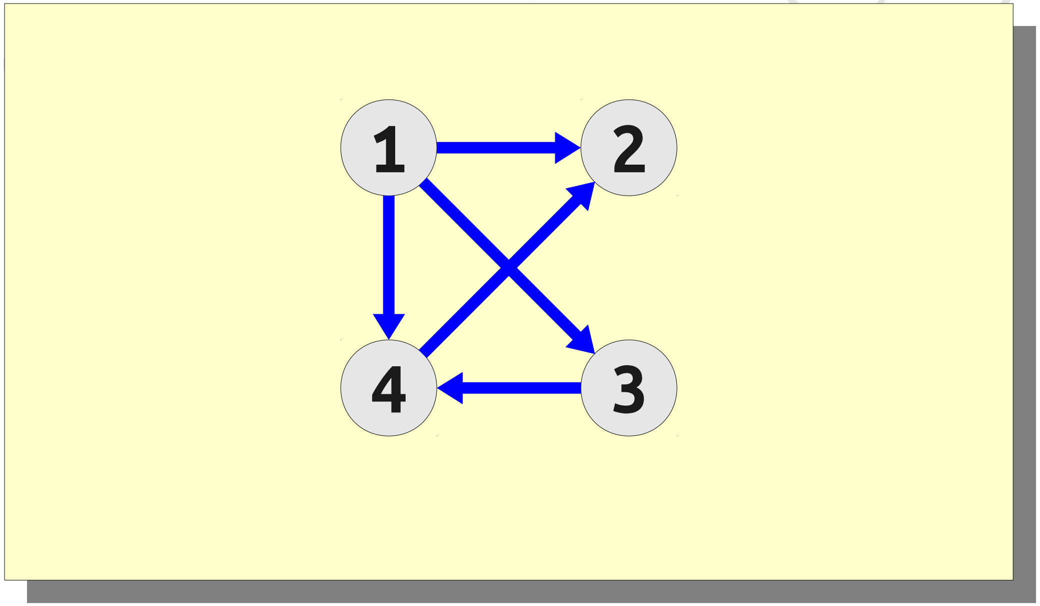
```
        procedure breadthFirstSearch(s, G):
O(1)        let q b
O(n)        for eac
               dist[

O(1)        dist[s]
            enqueue

O(n)        while q is not empty:
               let v = dequeue(q)
               for each neighbor u of v:
                  if dist[u] = ∞:
                     dist[u] = dist[v] + 1
                     enqueue(u, q)
```



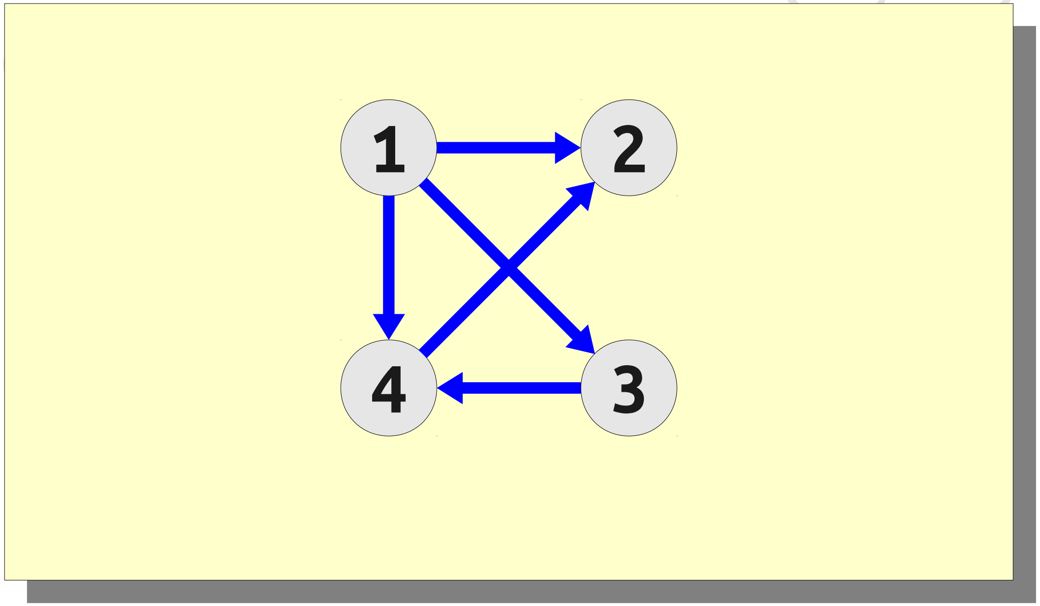
```
        procedure breadthFirstSearch(s, G):
O(1)        let q b
O(n)        for eac
                dist[

O(1)        dist[s]
            enqueue

O(n)        while q is not empty:
                let v = dequeue(q)
                for each neighbor u of v:
                    if dist[u] = ∞:
                        dist[u] = dist[v] + 1
                        enqueue(u, q)
```



```
        procedure breadthFirstSearch(s, G):
O(1)        let q b
O(n)        for eac
                dist[

            dist[s]
O(1)        enqueue

O(n)        while q is not empty:
                let v = dequeue(q)
                for each neighbor u of v:
                    if dist[u] = ∞:
                        dist[u] = dist[v] + 1
                        enqueue(u, q)
```



```
             procedure breadthFirstSearch(s, G):
O(1)             let q b
O(n)             for eac
                     dist[

O(1)             dist[s]
                 enqueue

O(n)             while q is not empty:
                     let v = dequeue(q)
                     for each neighbor u of v:
O(m + n)                 if dist[u] = ∞:
                             dist[u] = dist[v] + 1
                             enqueue(u, q)
```



```
               procedure breadthFirstSearch(s, G):
$O(1)$             let $q$ be a new queue.
$O(n)$             for each node $v$ in G:
                       dist[$v$] = ∞

$O(1)$             dist[$s$] = 0
                   enqueue(s, q)

$O(n)$             while q is not empty:
                       let v = dequeue($q$)
                       for each neighbor $u$ of $v$:
$O(m + n)$                 if dist[$u$] = ∞:
                               dist[$u$] = dist[$v$] + 1
                               enqueue(u, q)
```

A Better Analysis

- Using adjacency lists, BFS runs in time **O(m + n)**.
    - This is linear time!
- **Key Idea**: Do a more precise accounting of the work done by an algorithm.
    - Determine how much work is done *across all iterations* to determine total work.
    - Don't just find worst-case runtime and multiply by number of iterations.
- Going forward, we will use adjacency lists rather than adjacency matrices as our graph representation unless stated otherwise.

Next Time

- Dijkstra's Algorithm
- Depth-First Search
- Directed Acyclic Graphs