Carleton University
Alina Shaikhet

Graphs

part 2

Definitions

Every **path** is a sequence of edges connecting a sequence of vertices.

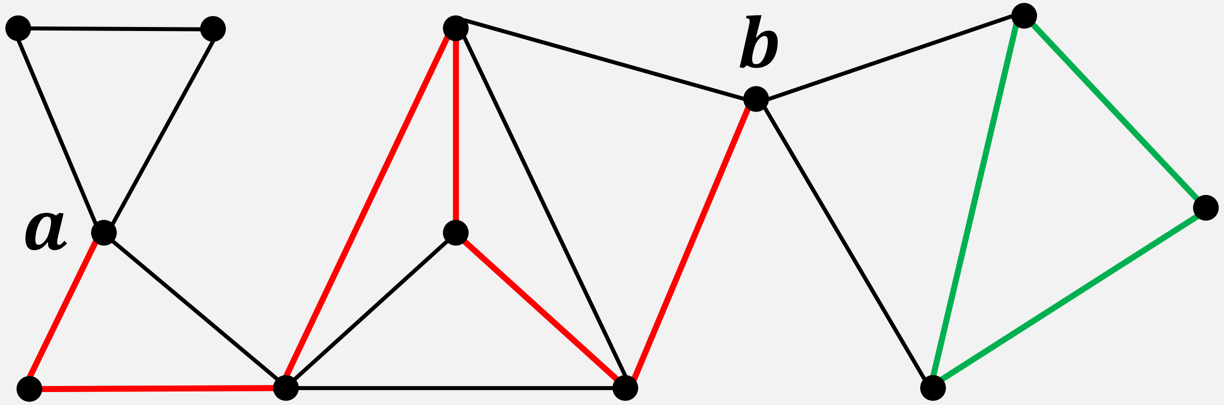
The **length** of the path is the number of edges in the path.

Every **cycle** (or **circuit**) is a sequence of edges that starts and ends at the same vertex.

A **simple path** does not traverse any edge more than once.

An undirected graph is **connected** if there is a path between any pair of vertices.

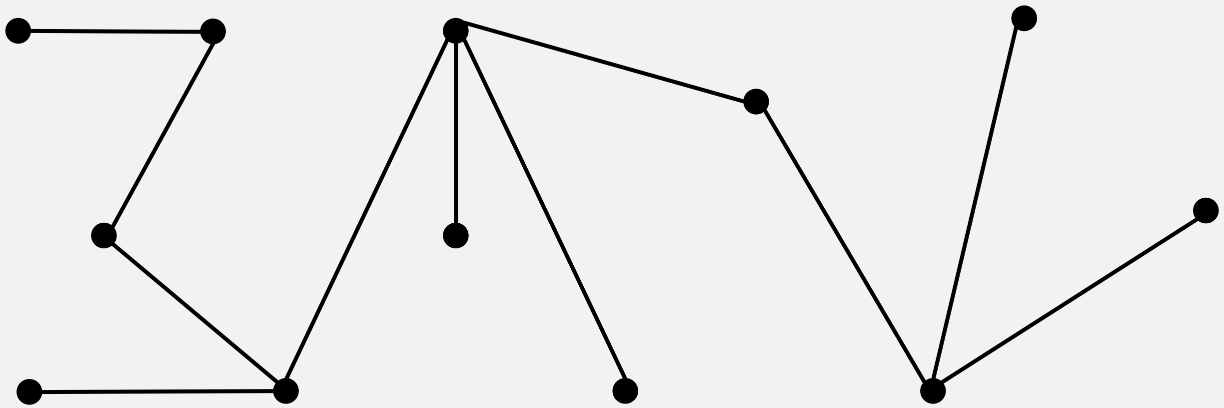
A graph is called **disconnected** otherwise.



Tree

A connected undirected graph without any cycles is known as a **tree**

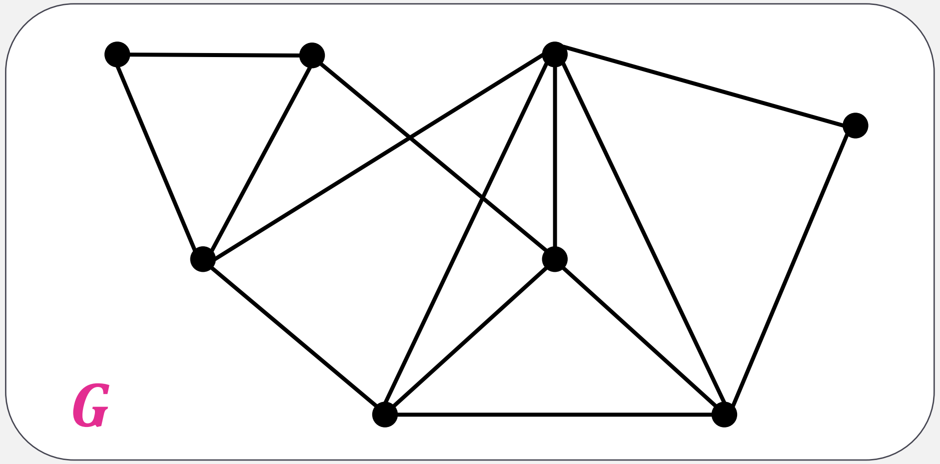
There exists a **unique** simple path between every pair of vertices in a tree.

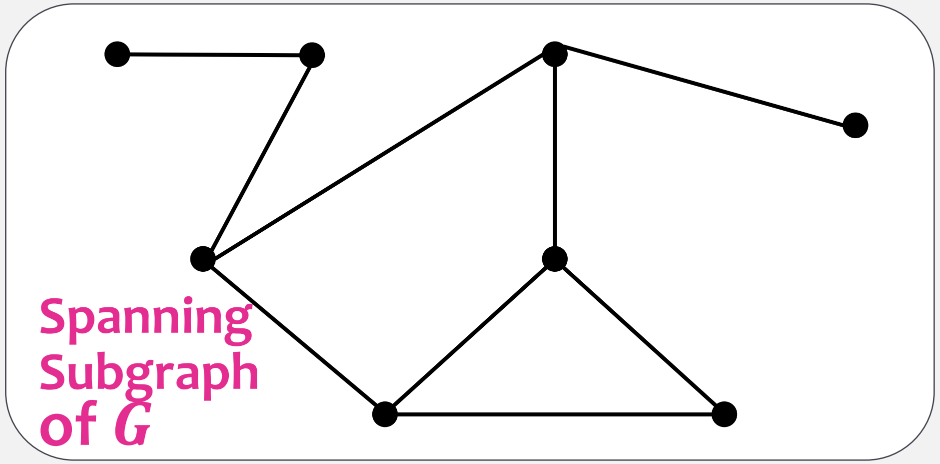


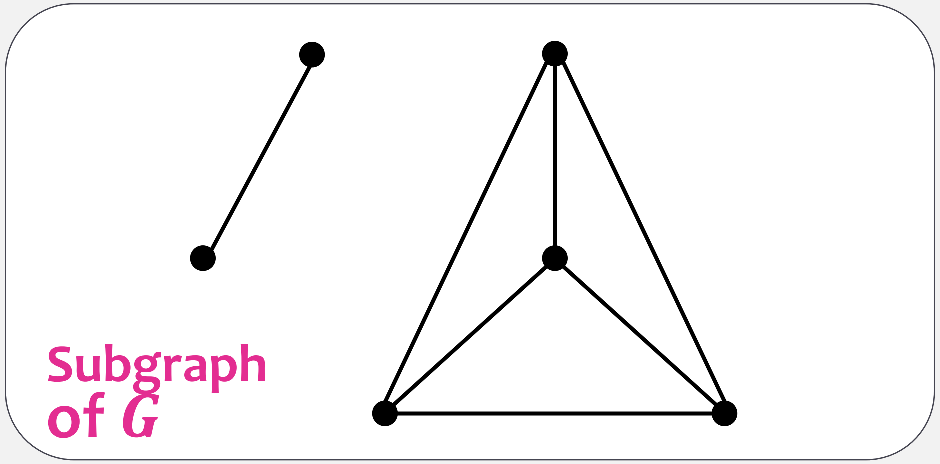
Subgraphs

Given a graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, a **subgraph** $\boldsymbol{G}' = (\boldsymbol{V}', \boldsymbol{E}')$ of $\boldsymbol{G}$ is another graph such that $\boldsymbol{V}' \subseteq \boldsymbol{V}$ and $\boldsymbol{E}' \subseteq \boldsymbol{E}$, where $\{\boldsymbol{a}, \boldsymbol{b}\} \in \boldsymbol{E}'$ only if $\boldsymbol{a}, \boldsymbol{b} \in \boldsymbol{V}'$.

$\boldsymbol{G}'$ is a **spanning subgraph** of $\boldsymbol{G}$ if $\boldsymbol{V}' = \boldsymbol{V}$



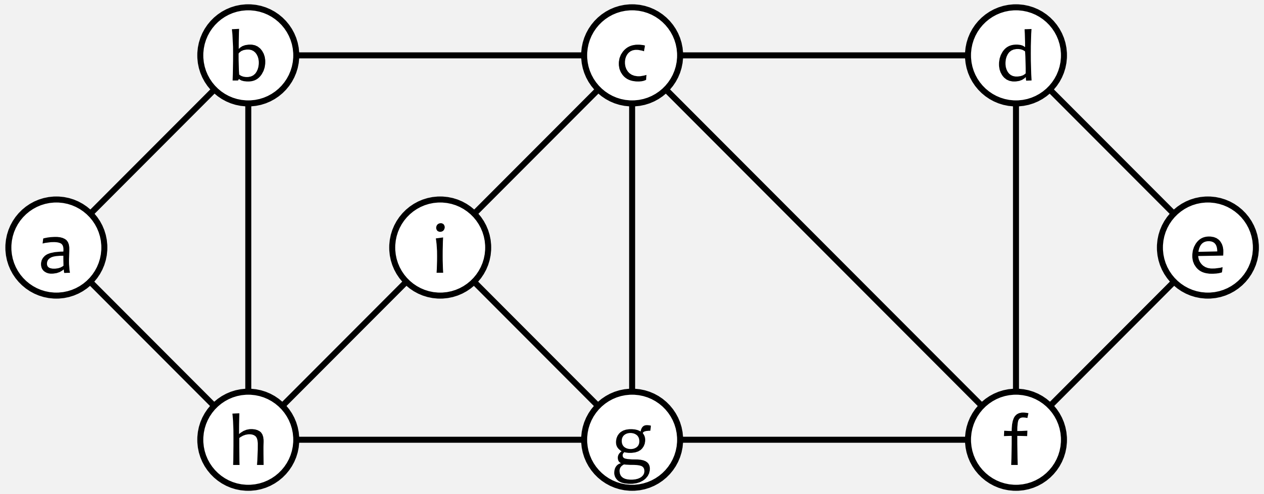




Tree

A **spanning tree** contains every vertex of the graph (since it is **spanning**) and no more edges than necessary (since it is a **tree**)

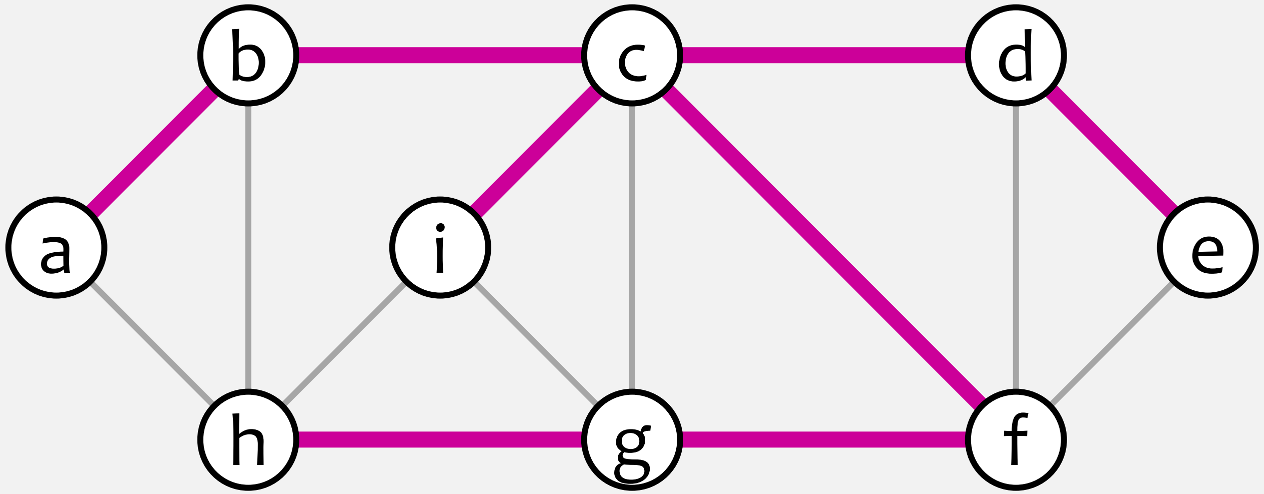
What is a spanning tree of this graph?



Tree

A **spanning tree** contains every vertex of the graph (since it is **spanning**) and no more edges than necessary (since it is a **tree**)

What is a spanning tree of this graph?



Exploring (searching) a Graph

We often need to visit all vertices in a graph.

Exploration/Search starts at a specific vertex and then visits other vertices.

Exploration/Search may terminate if a specific vertex is reached, or it may continue until every vertex is visited

There are two very common ways of exploring a graph:

breadth-first search (BFS) and **depth-first search (DFS)**

Depth-First Search (DFS)

Given: $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ and vertex $\boldsymbol{v_i}$.

Goal: Find all the vertices that can be reached from $\boldsymbol{v_i}$.

- Mark all vertices as unvisited
- Start at the given vertex $\boldsymbol{v_i}$ or at an arbitrary vertex if $\boldsymbol{v_i}$ is not specified
- Move to an unvisited neighbour. When given choice, always choose the **smallest** element to visit (smallest number, letter, etc.)
- Repeat until every neighbour is visited

Out of Neighbours?

Then return to the most recently visited vertex and visit those neighbours...

Depth-First Search (DFS)

Algorithm DFS(G, v_i):

for all $v \in V$:
  visited(v) = **false**;
T is a tree with single vertex v_i
explore(v_i, T, G);
Output T;

The depth-first-search algorithm is similar to the standard algorithm for traversing binary trees; it first fully explores one subtree before returning to the current node and then exploring the other subtree.

Algorithm explore(v, T, G):

visited(v) = **true**;
for each edge $\{v, u\} \in E$: ← **for each neighbor of v**
  if visited(u) = **false**: ← **u is not in T**
    add u and $\{v, u\}$ to T;
    explore(u, T, G);

Recursive code will fail for many large graphs by causing a **stack overflow**. An alternative implementation is to replace the recursion stack with an explicit stack. (refer to the **ods** textbook)

Depth-First Search (DFS)

- **explore**() is called exactly once for each vertex reachable from $\boldsymbol{v_i}$ (as a part of a recursive call). Only on $\boldsymbol{v_i}$ it is called explicitly.
- time spent for **explore**$(\boldsymbol{v})$, excluding recursive call, is $\boldsymbol{O}(\mathbf{1} + \boldsymbol{deg}(\boldsymbol{v}))$.

Total time:

$$\boldsymbol{O}\left(|\boldsymbol{V}| + \sum_{\boldsymbol{v} \epsilon \boldsymbol{V}} (\mathbf{1} + \boldsymbol{deg}(\boldsymbol{v})) \right)$$

$$\boldsymbol{O}(|\boldsymbol{V}| + |\boldsymbol{E}|)$$

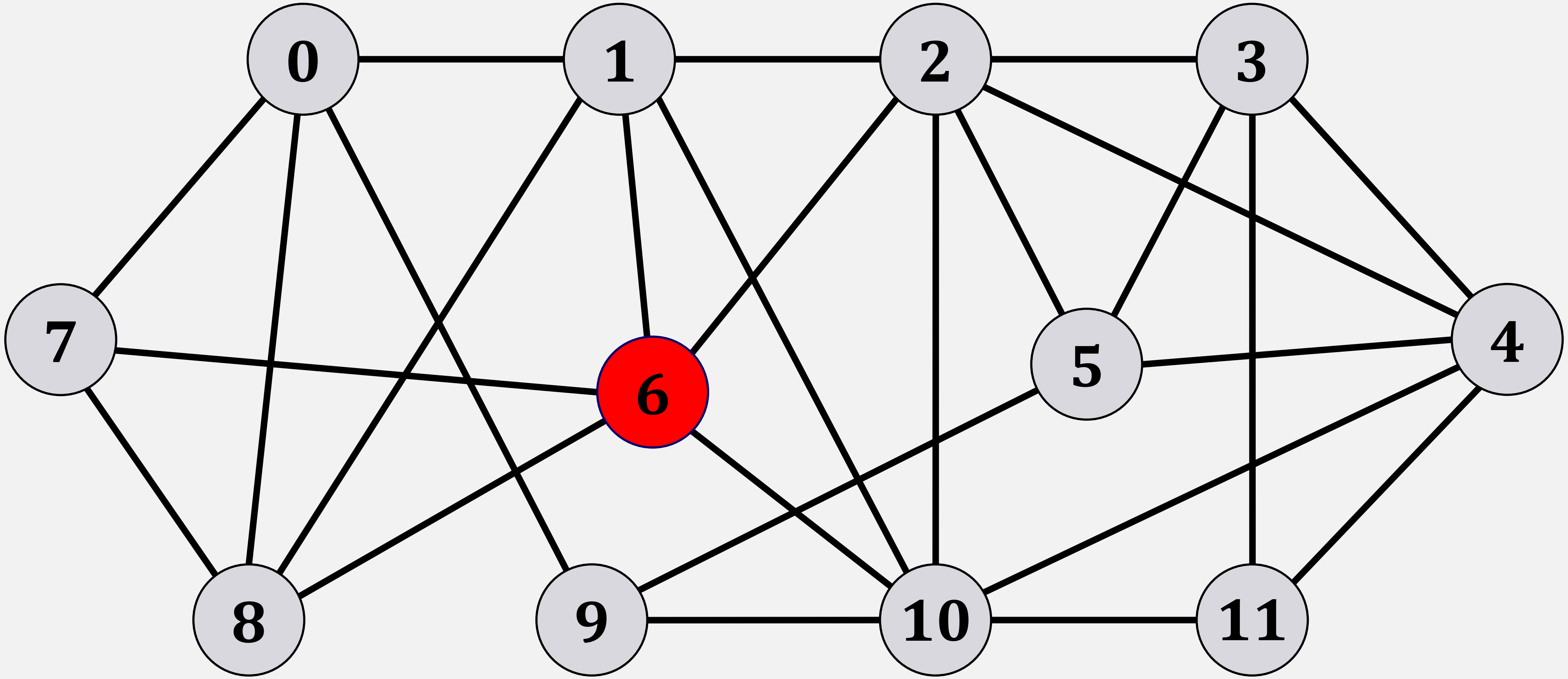
Algorithm **explore**$(\boldsymbol{v}, \boldsymbol{T}, \boldsymbol{G})$:

```
visited(v) = true;
for each edge {v, u} ∈ E:
    if visited(u) = false:        ⟵ u is not in T
        add u and {v, u} to T;
        explore(u, T, G);
```

Alina Shaikhet – COMP2406 – Carleton University

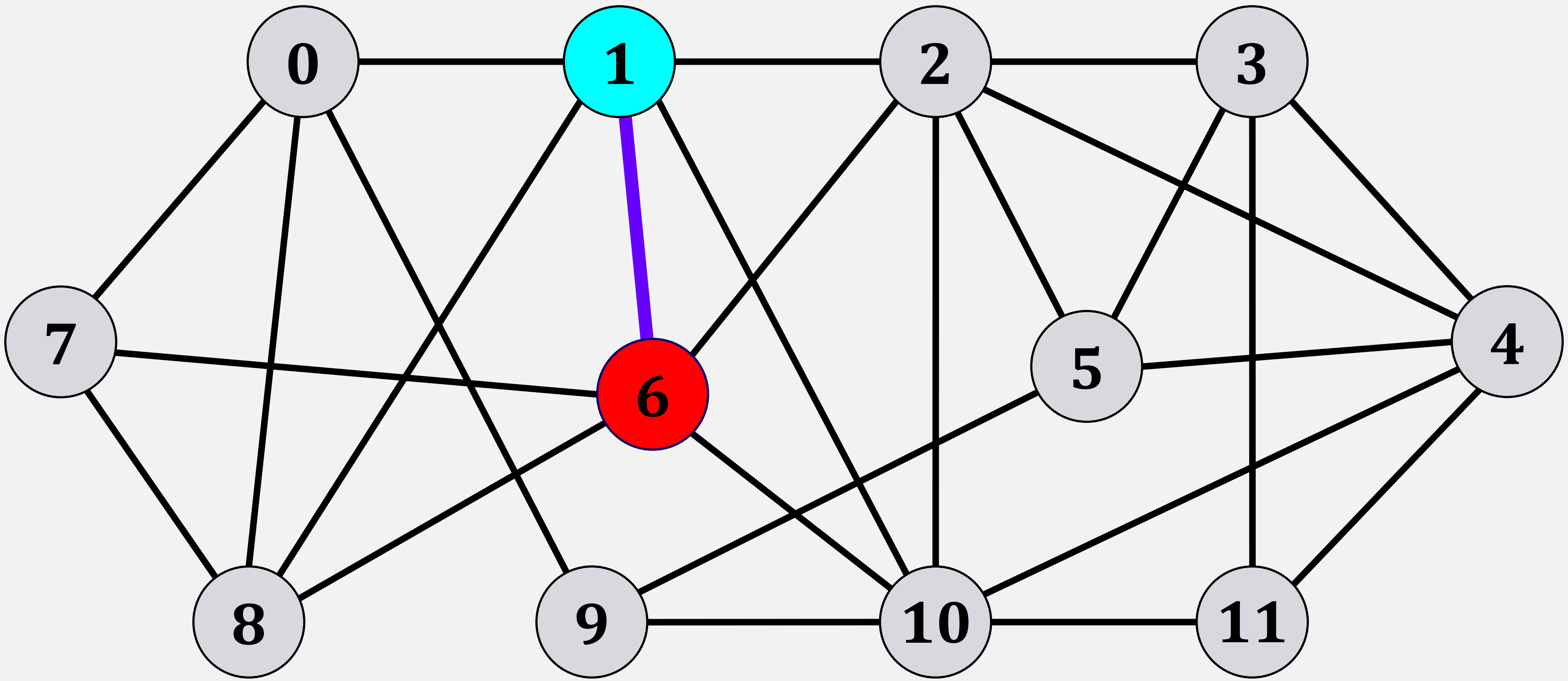
Depth-First Search (DFS)

Traversal: 6



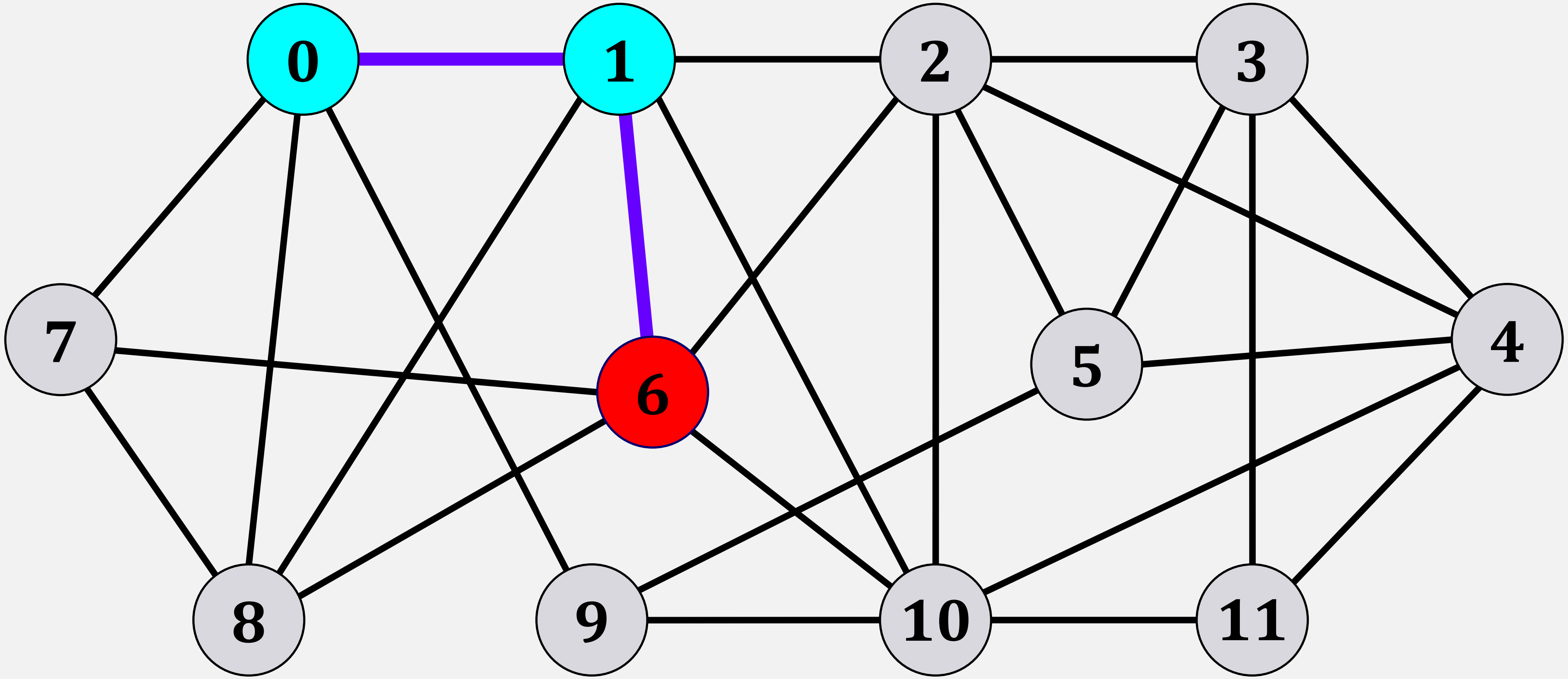
Depth-First Search (DFS)

Traversal: **6, 1**



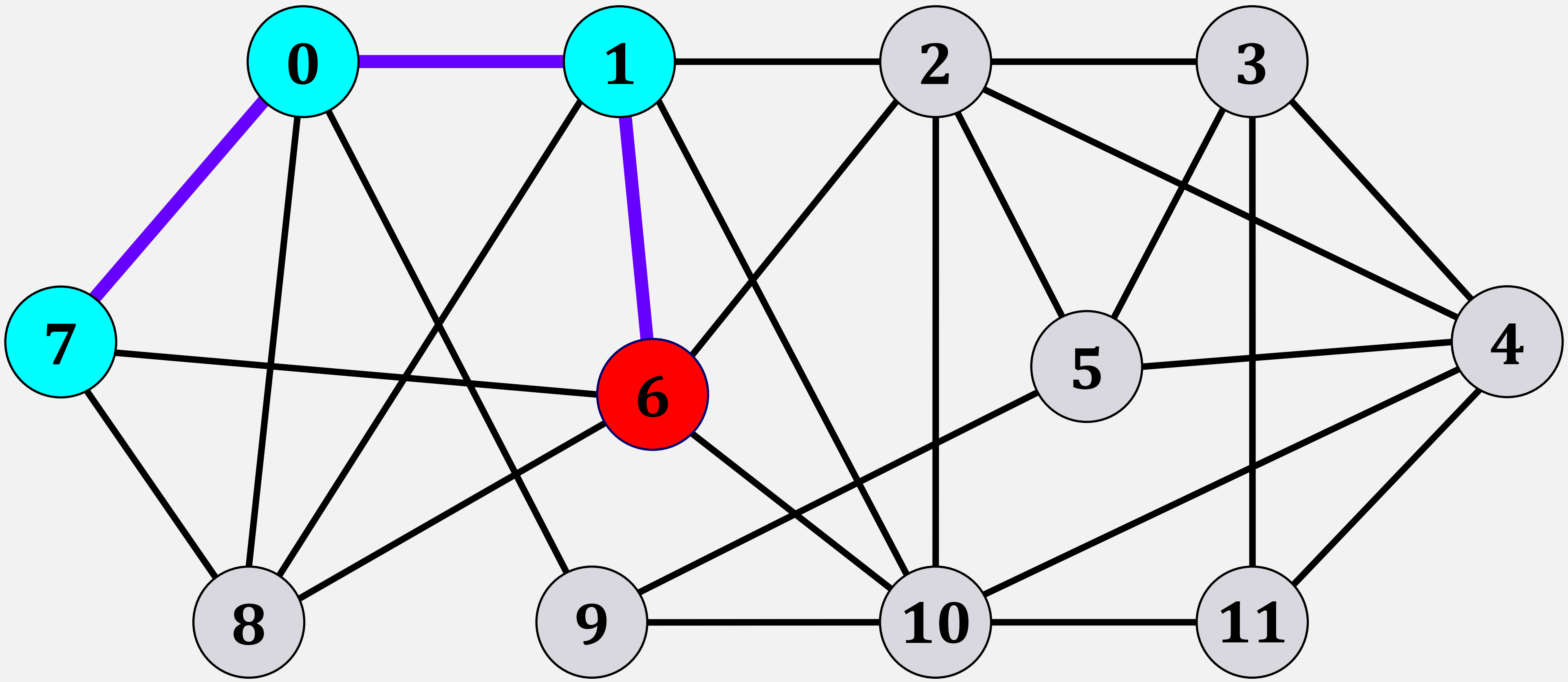
Depth-First Search (DFS)

Traversal: **6, 1, 0**



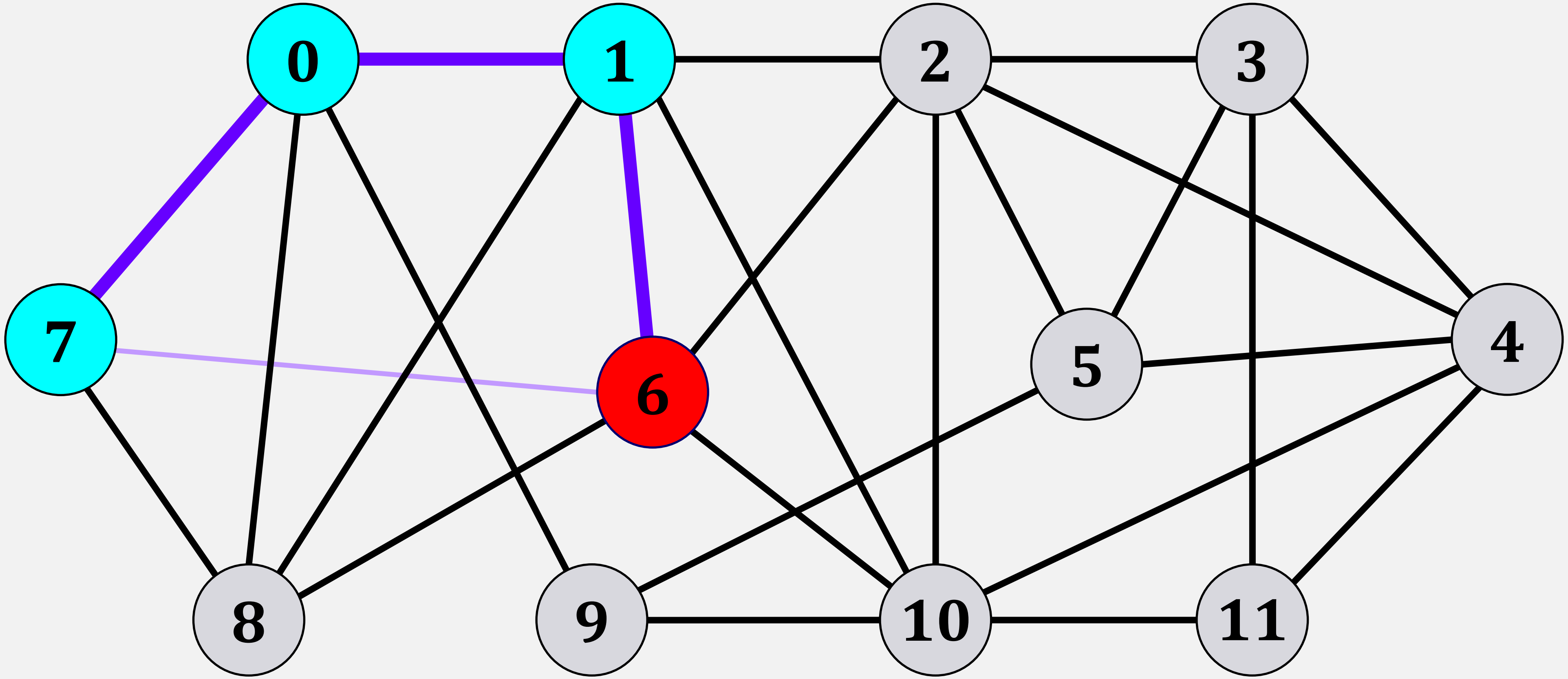
Depth-First Search (DFS)

Traversal: **6, 1, 0, 7**



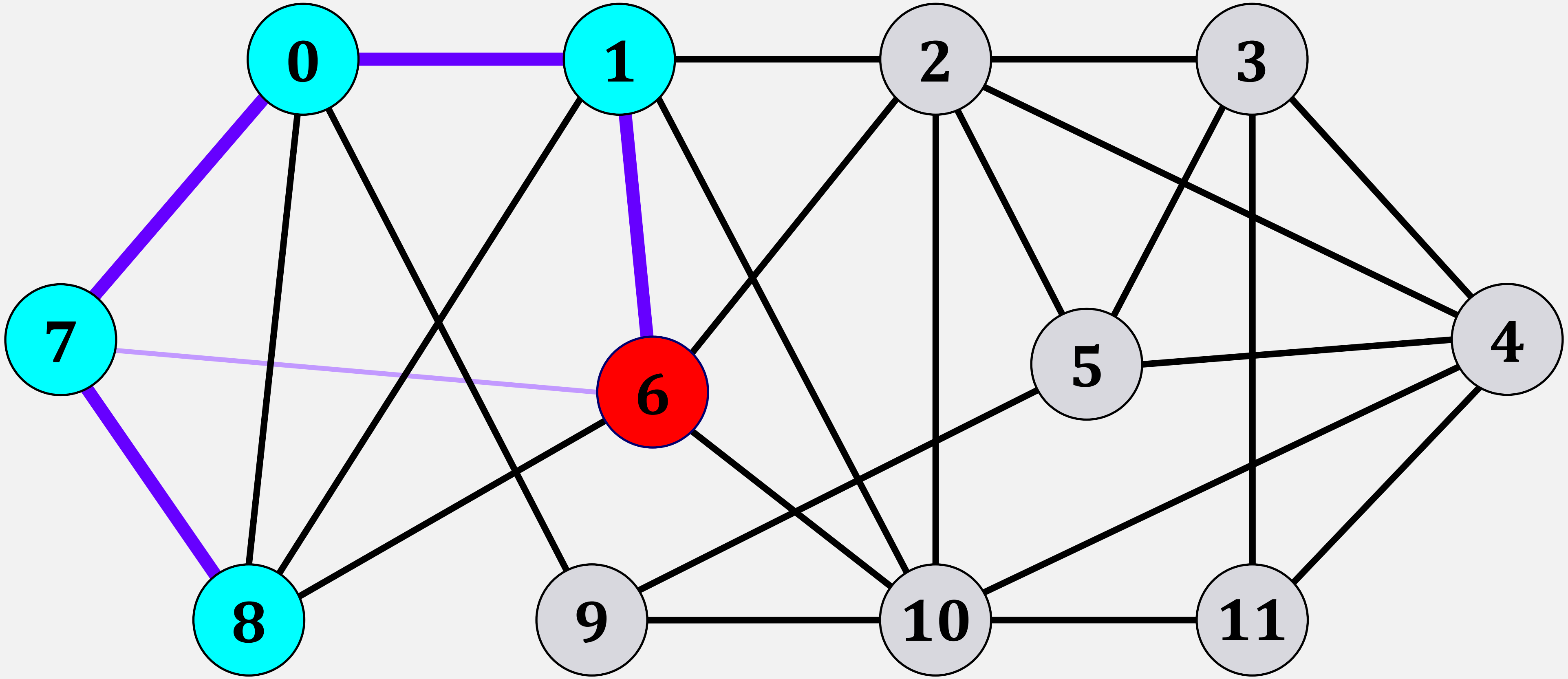
Depth-First Search (DFS)

Traversal: 6, 1, 0, 7



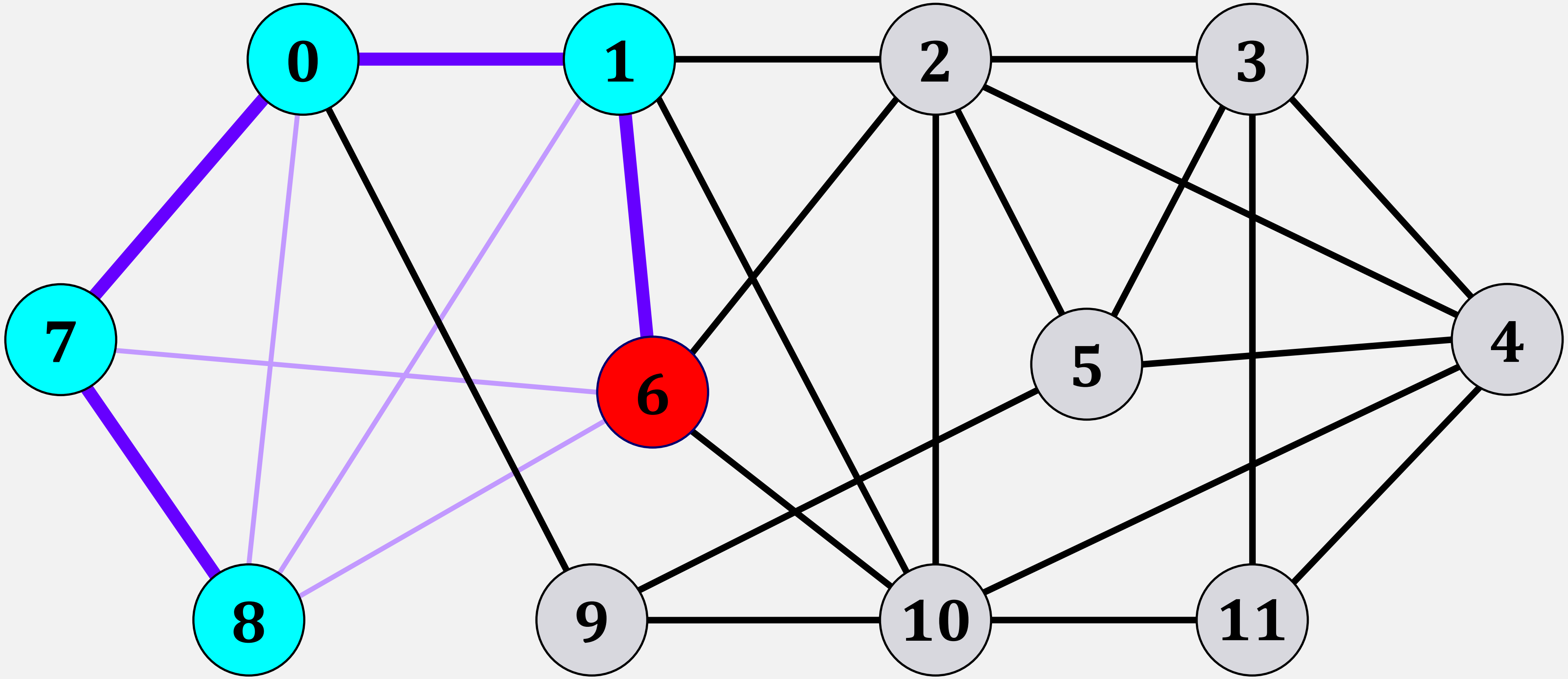
Depth-First Search (DFS)

Traversal: 6, 1, 0, 7, 8



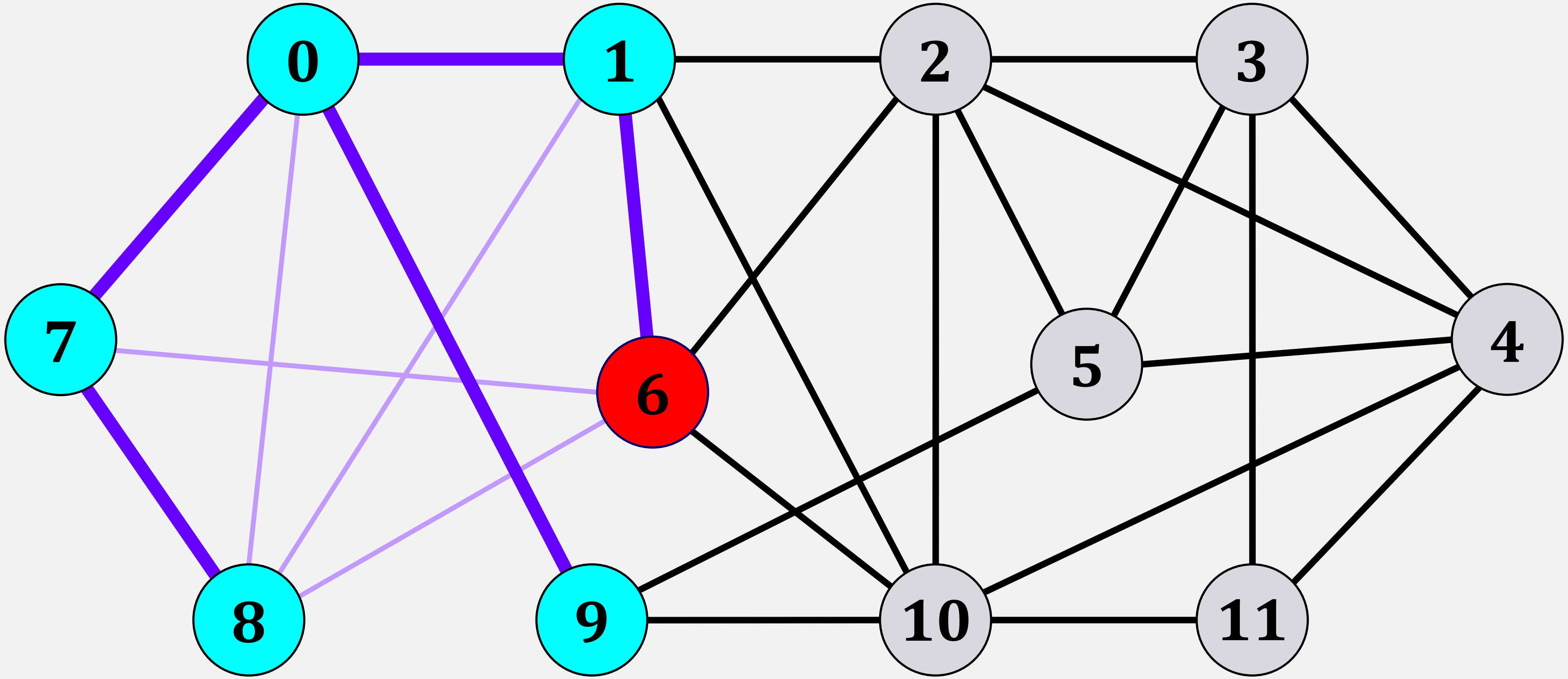
Depth-First Search (DFS)

Traversal: **6, 1, 0, 7, 8**



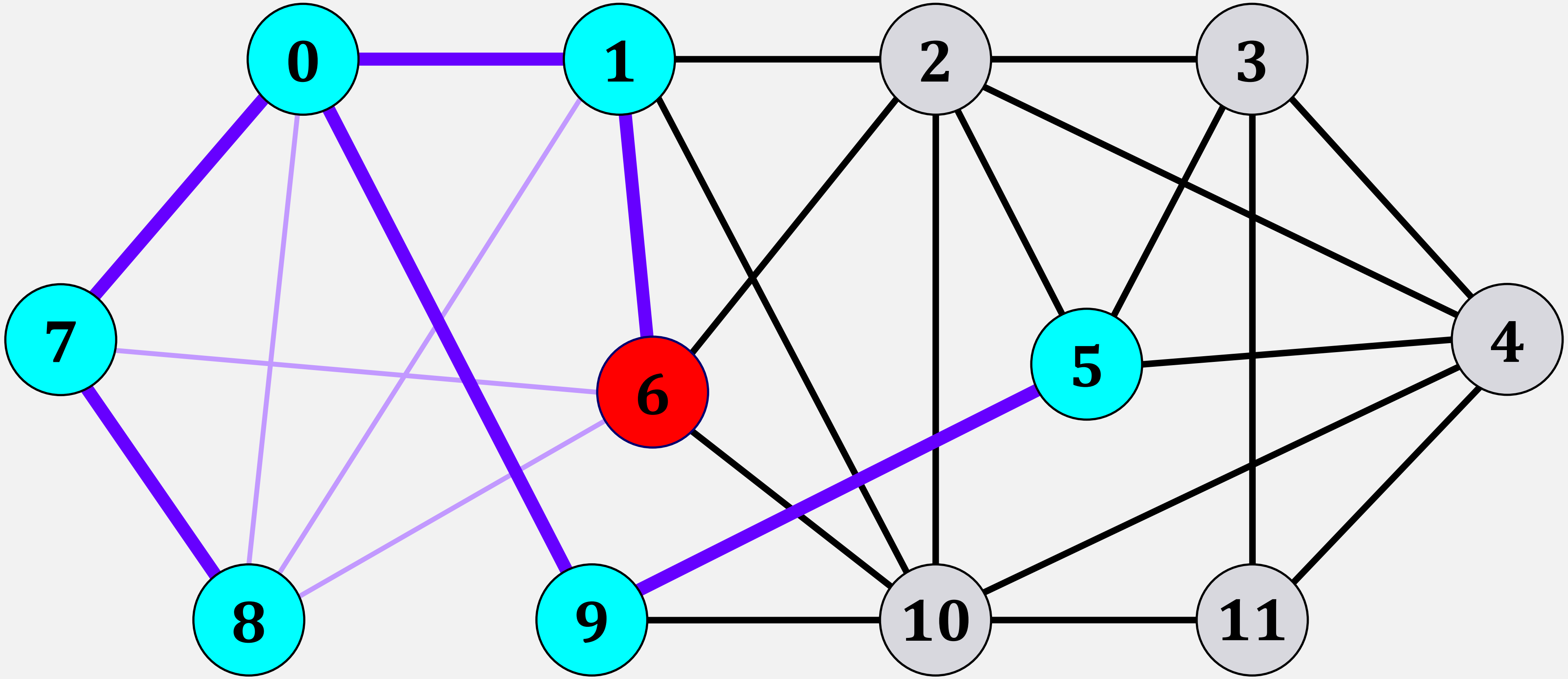
Depth-First Search (DFS)

Traversal: **6, 1, 0, 7, 8, 9**



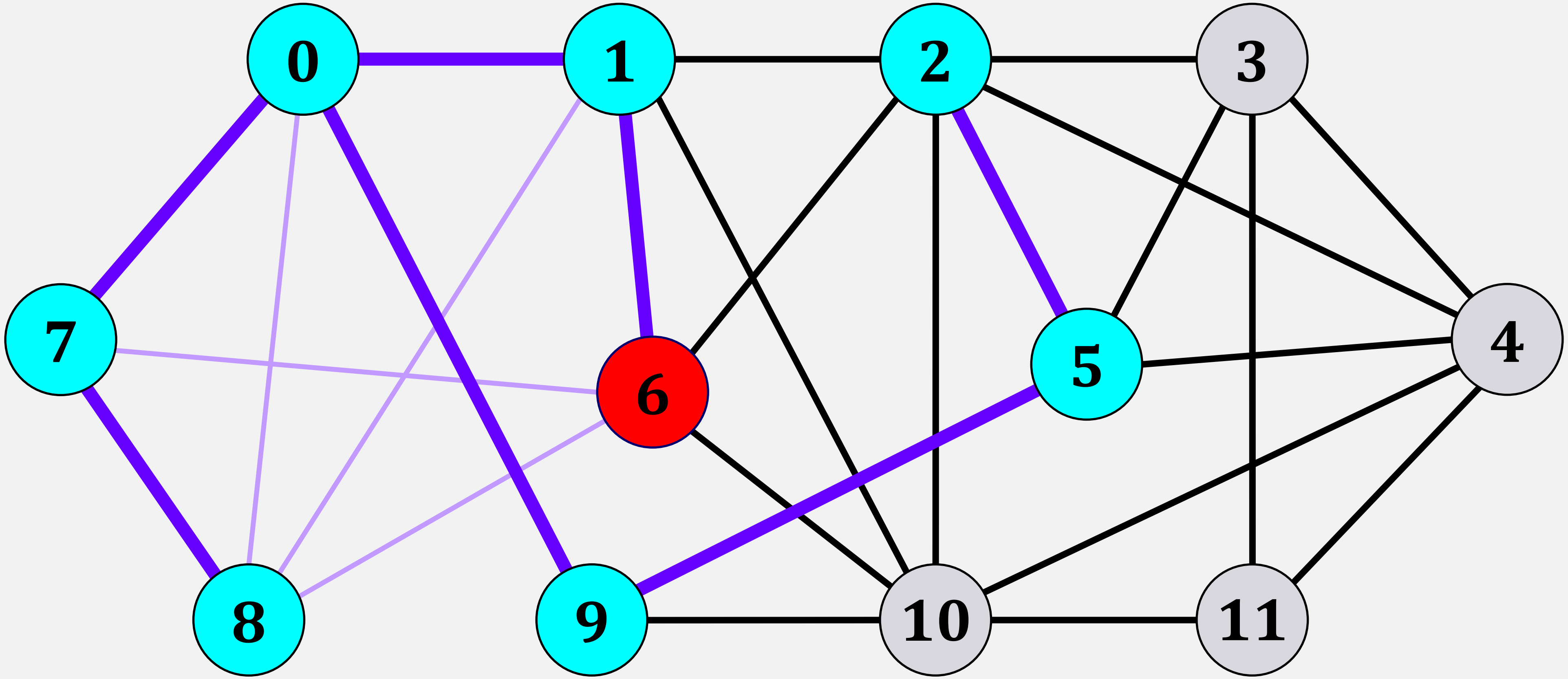
Depth-First Search (DFS)

Traversal: 6, 1, 0, 7, 8, 9, 5



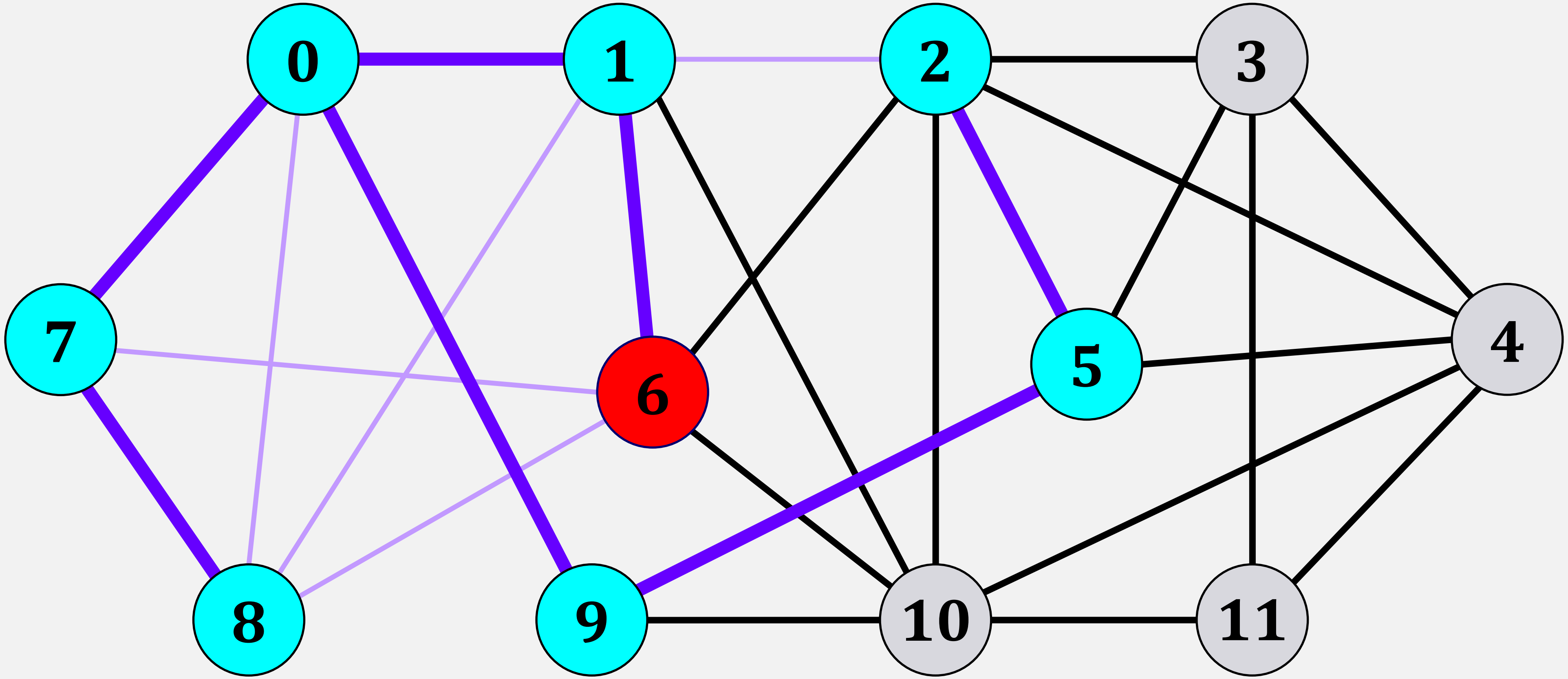
Depth-First Search (DFS)

Traversal: 6, 1, 0, 7, 8, 9, 5, 2



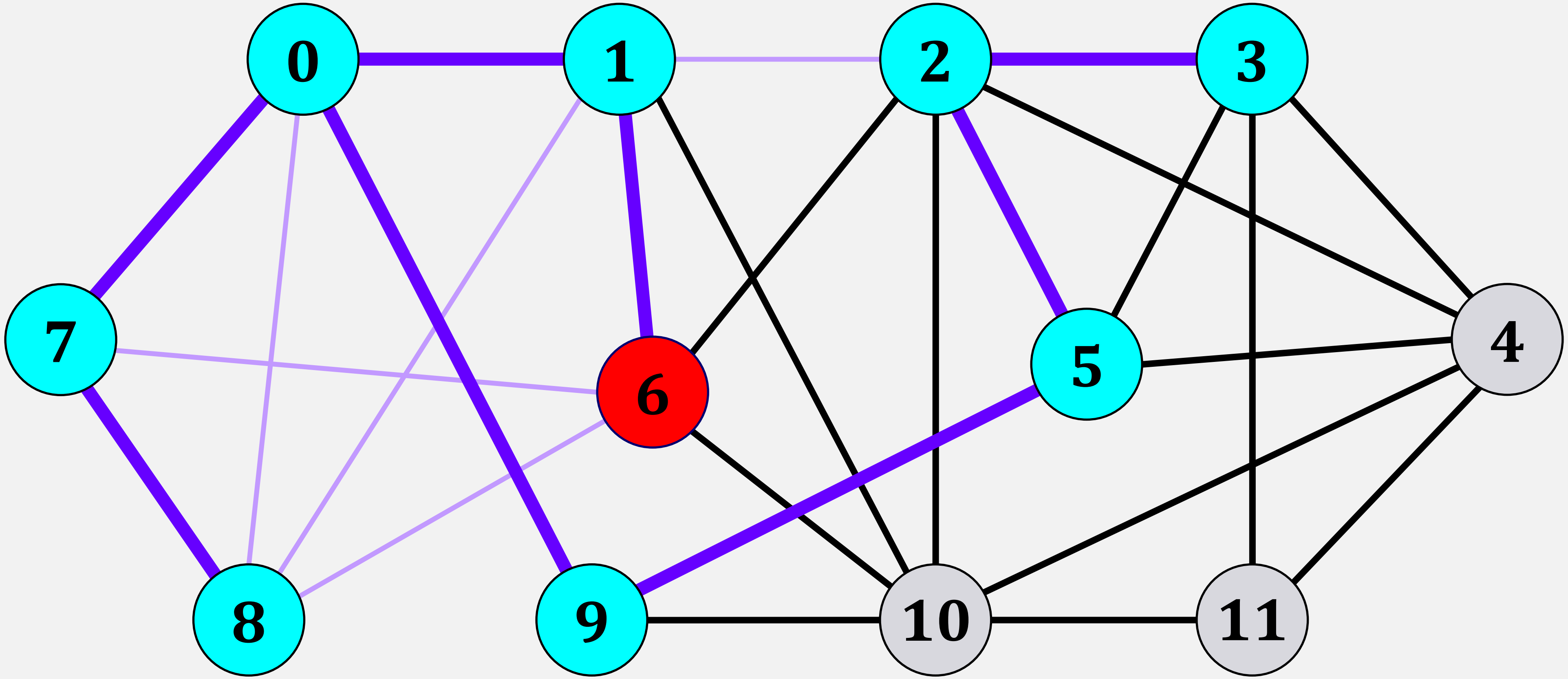
Depth-First Search (DFS)

Traversal: 6, 1, 0, 7, 8, 9, 5, 2



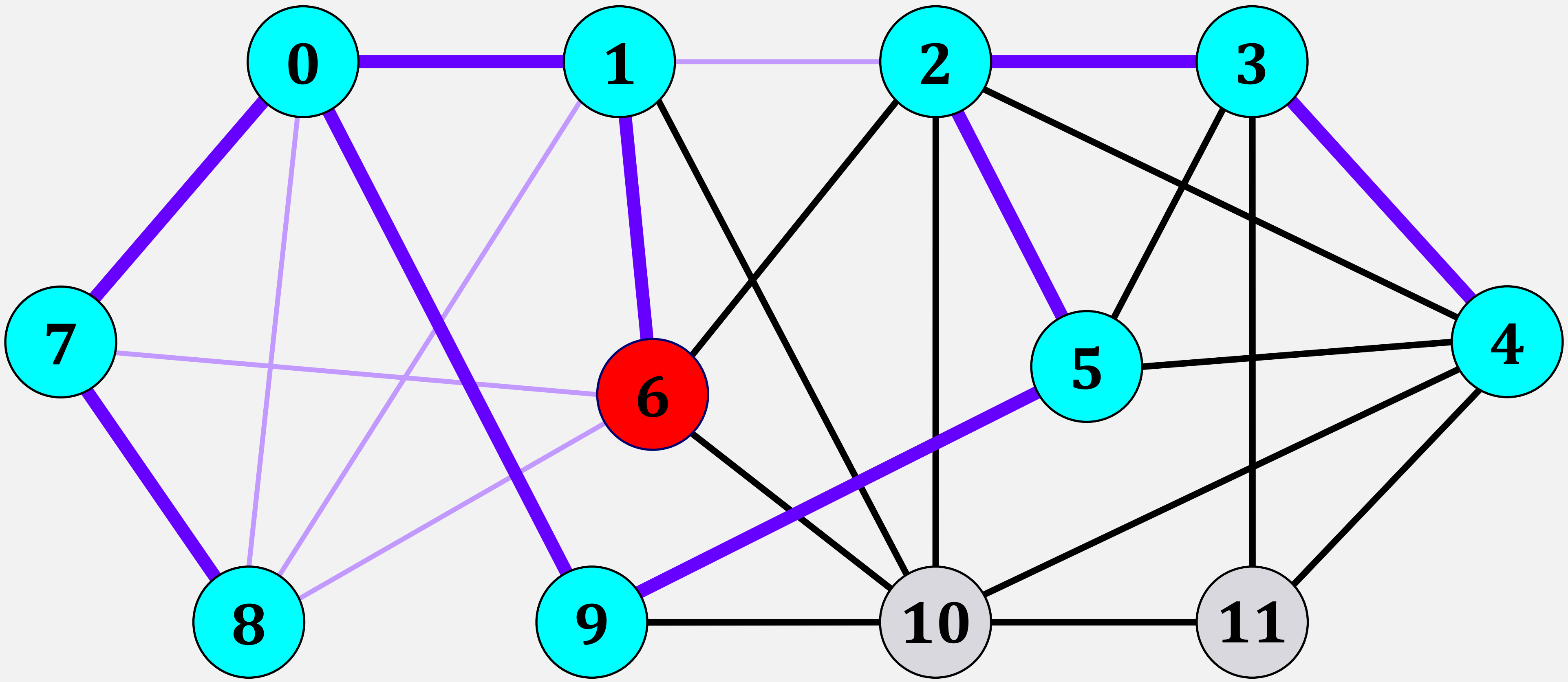
Depth-First Search (DFS)

Traversal: 6, 1, 0, 7, 8, 9, 5, 2, 3



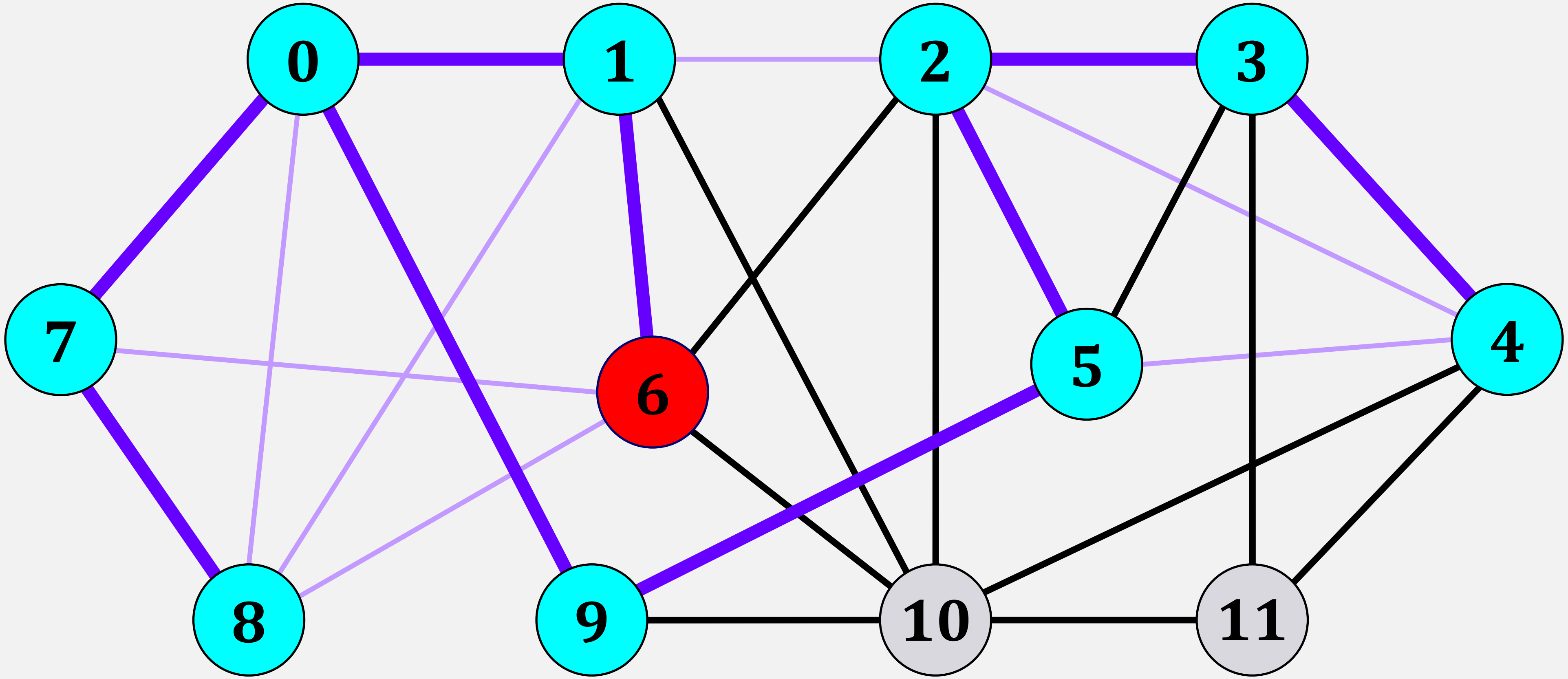
Depth-First Search (DFS)

Traversal: 6, 1, 0, 7, 8, 9, 5, 2, 3, 4



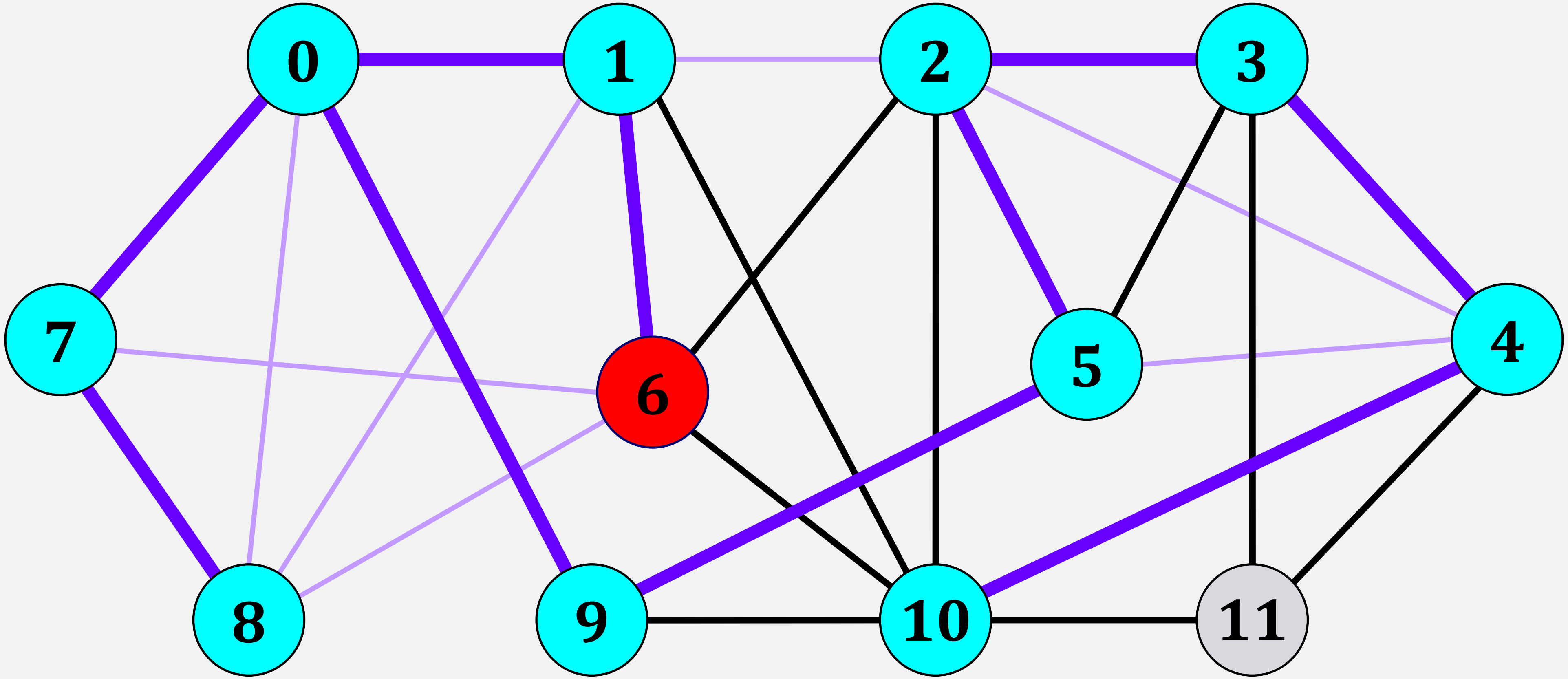
Depth-First Search (DFS)

Traversal: **6, 1, 0, 7, 8, 9, 5, 2, 3, 4**



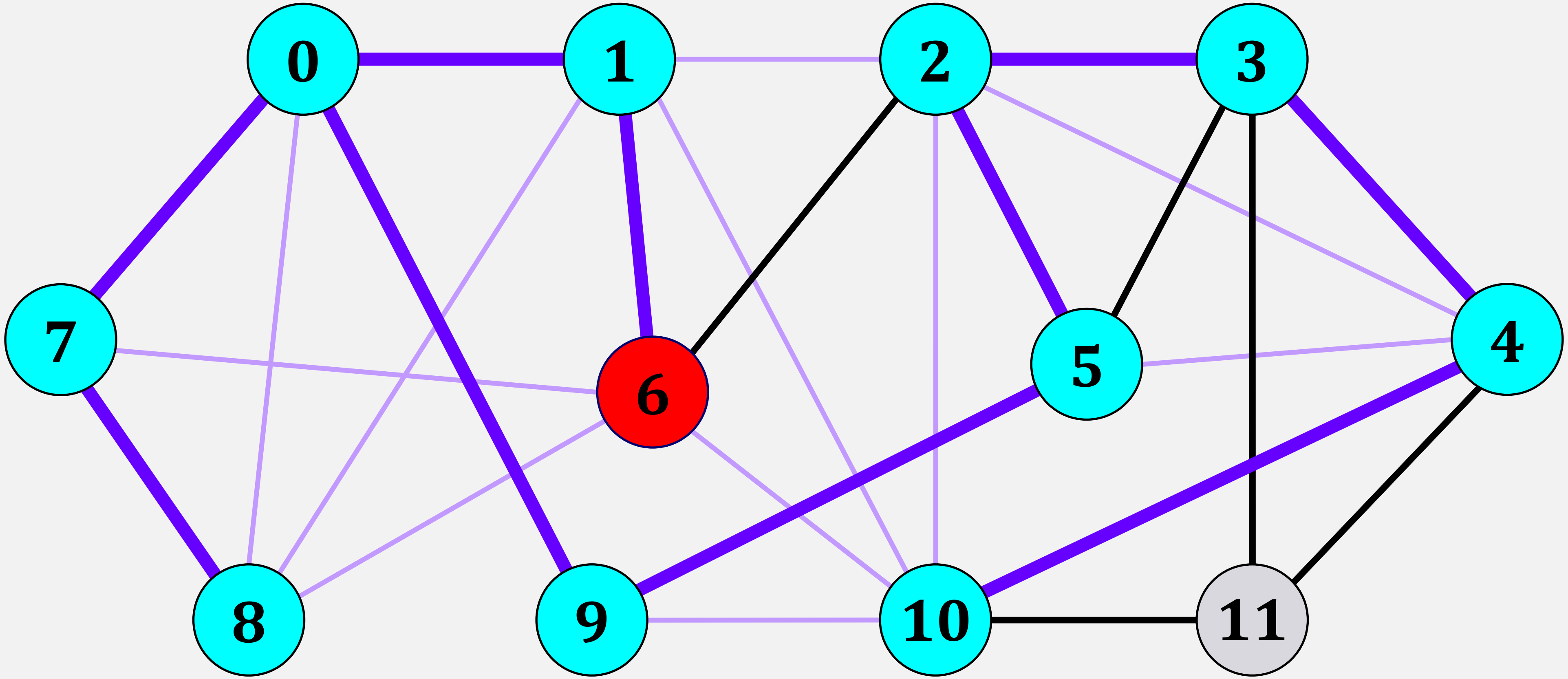
Depth-First Search (DFS)

Traversal: 6, 1, 0, 7, 8, 9, 5, 2, 3, 4, 10



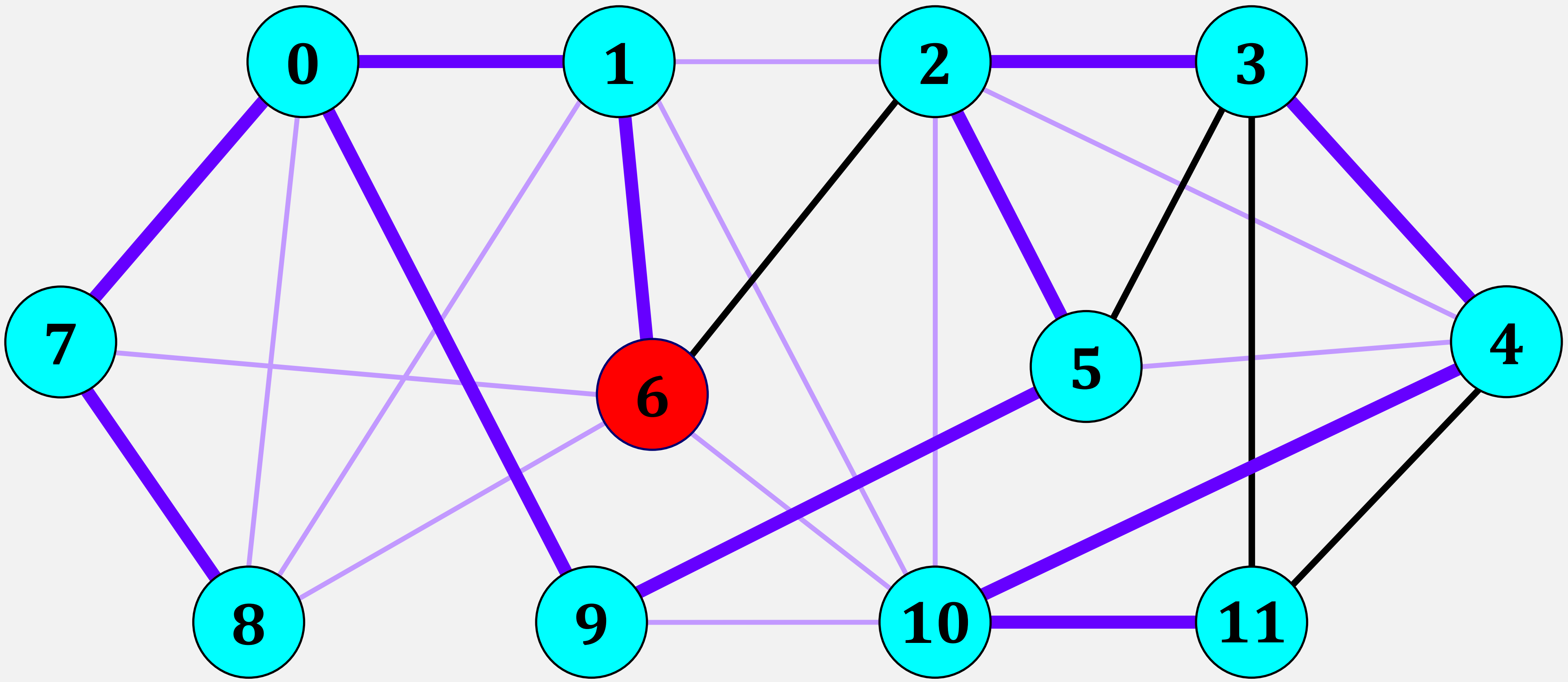
Depth-First Search (DFS)

Traversal: 6, 1, 0, 7, 8, 9, 5, 2, 3, 4, 10



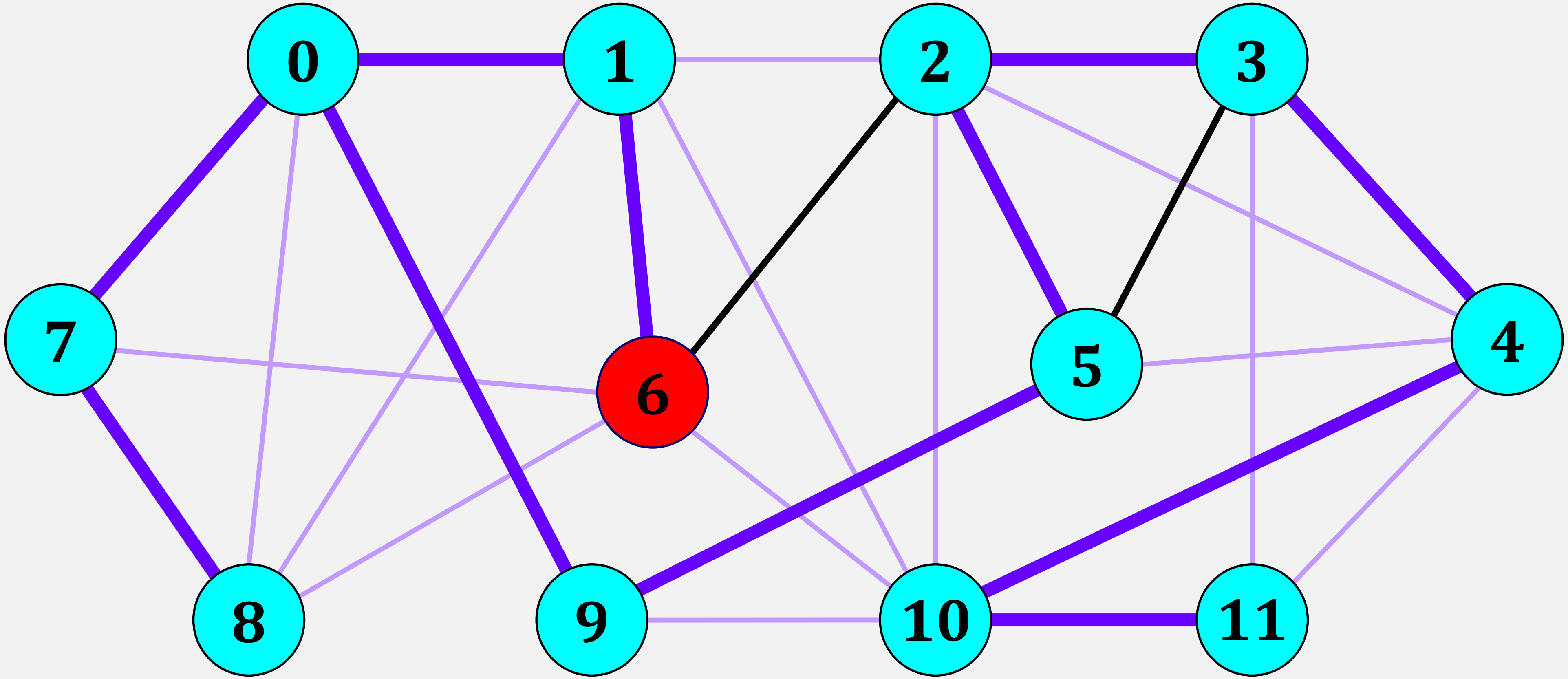
Depth-First Search (DFS)

Traversal: 6, 1, 0, 7, 8, 9, 5, 2, 3, 4, 10, 11



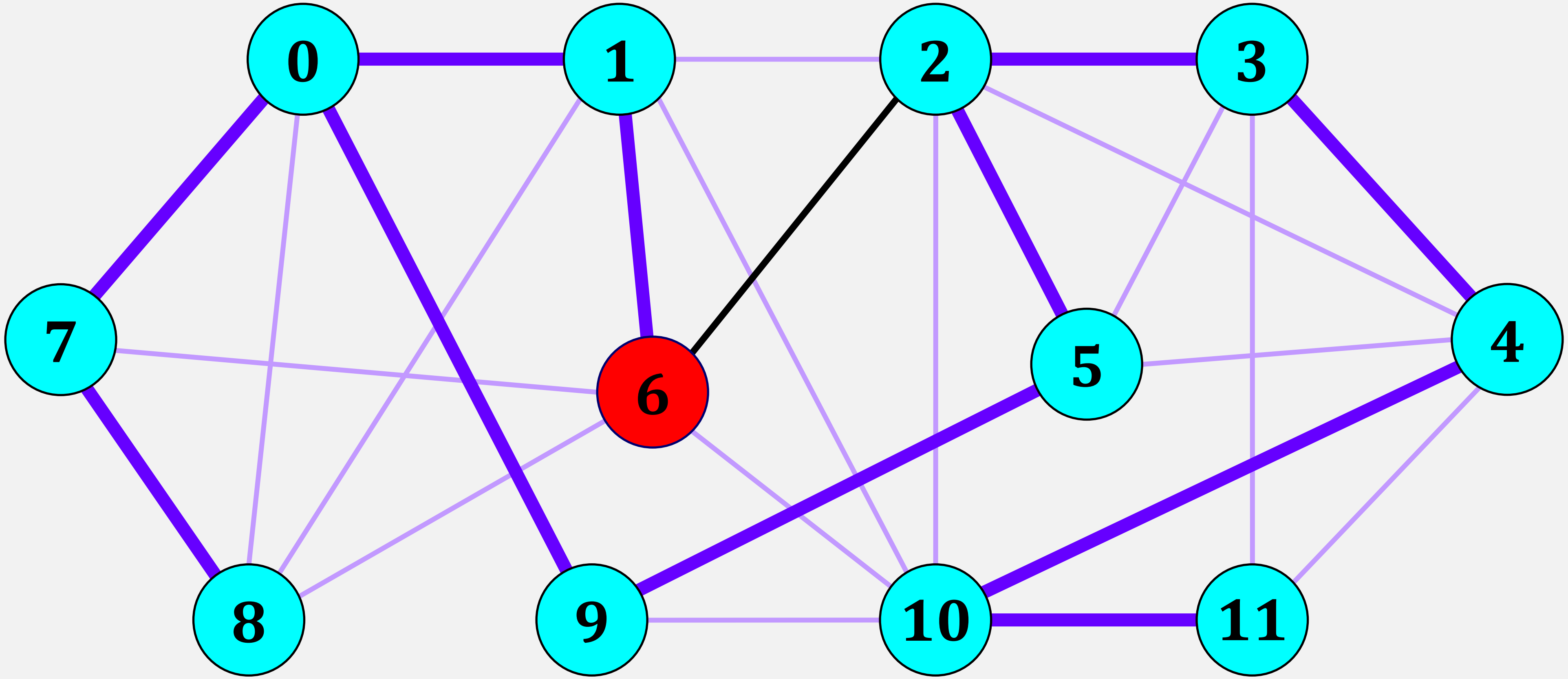
Depth-First Search (DFS)

Traversal: 6, 1, 0, 7, 8, 9, 5, 2, 3, 4, 10, 11



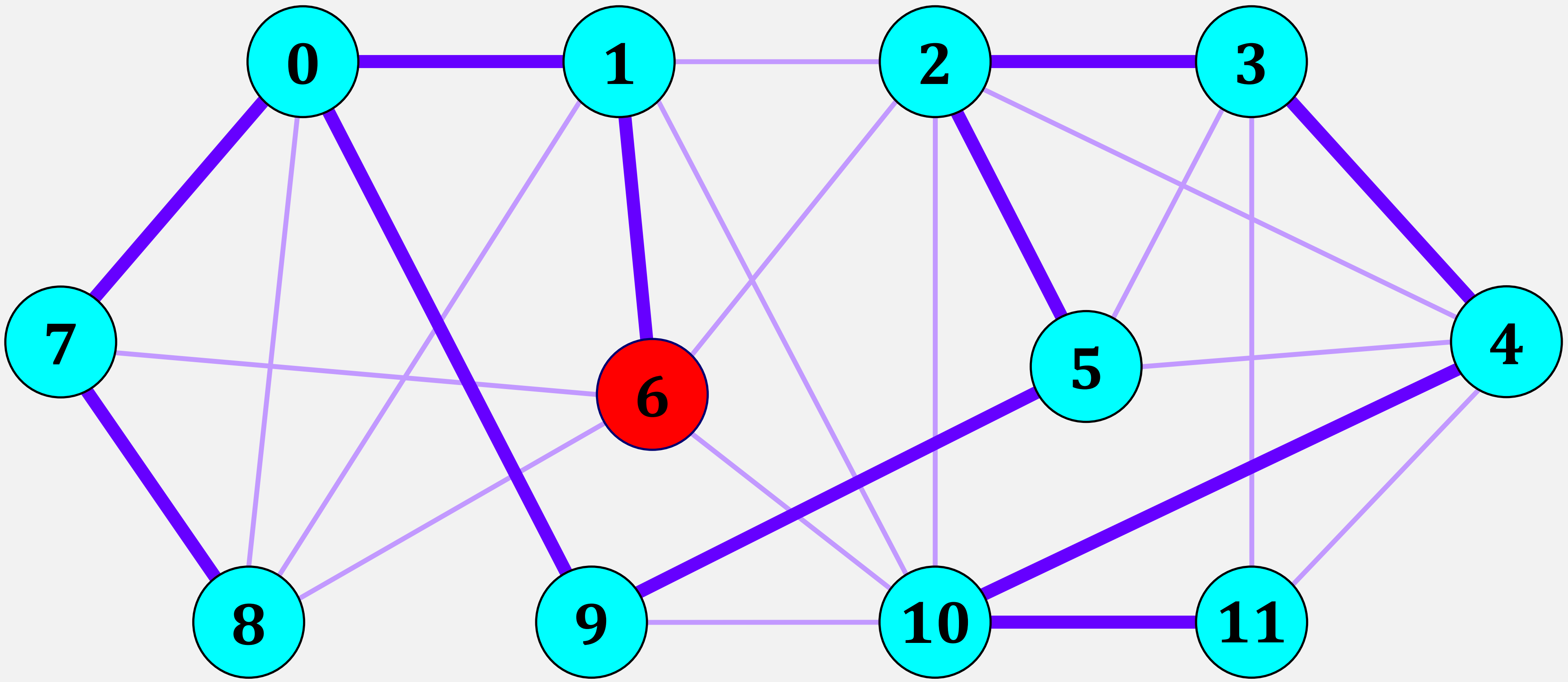
Depth-First Search (DFS)

Traversal: 6, 1, 0, 7, 8, 9, 5, 2, 3, 4, 10, 11



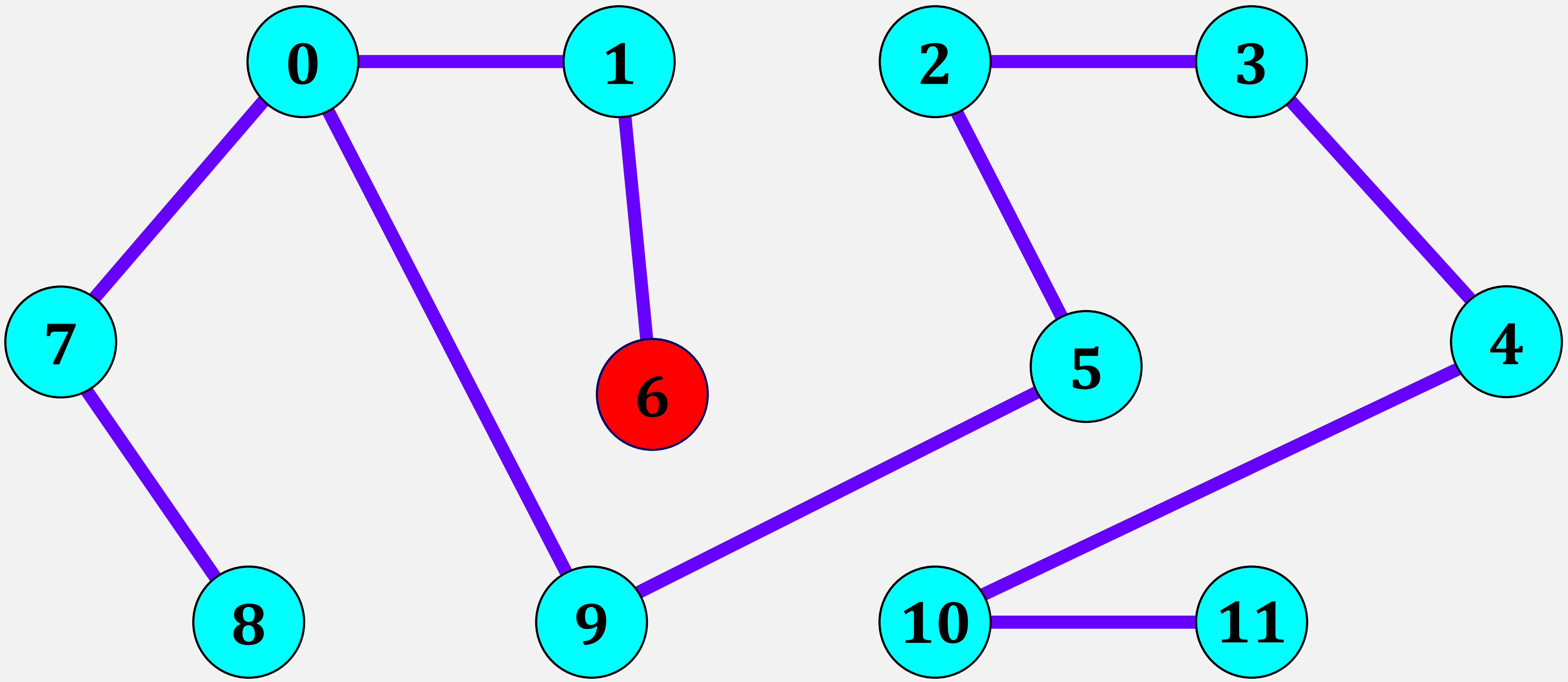
Depth-First Search (DFS)

Traversal: 6, 1, 0, 7, 8, 9, 5, 2, 3, 4, 10, 11



Depth-First Search (DFS)

Traversal: 6, 1, 0, 7, 8, 9, 5, 2, 3, 4, 10, 11



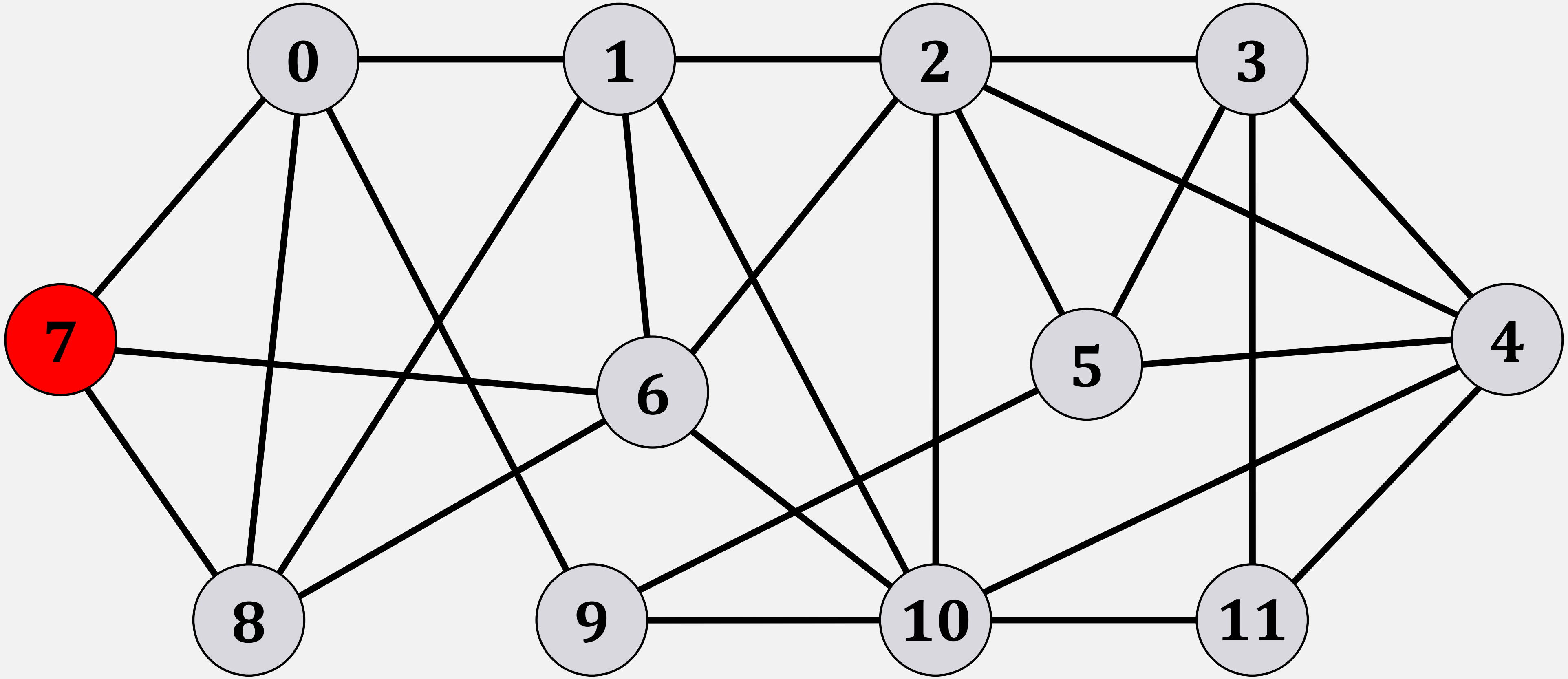
Spanning subtree generated by **DFS** with root **6**

Depth-First Search (DFS)

Spanning subtree generated by **DFS** depends on the **starting** vertex

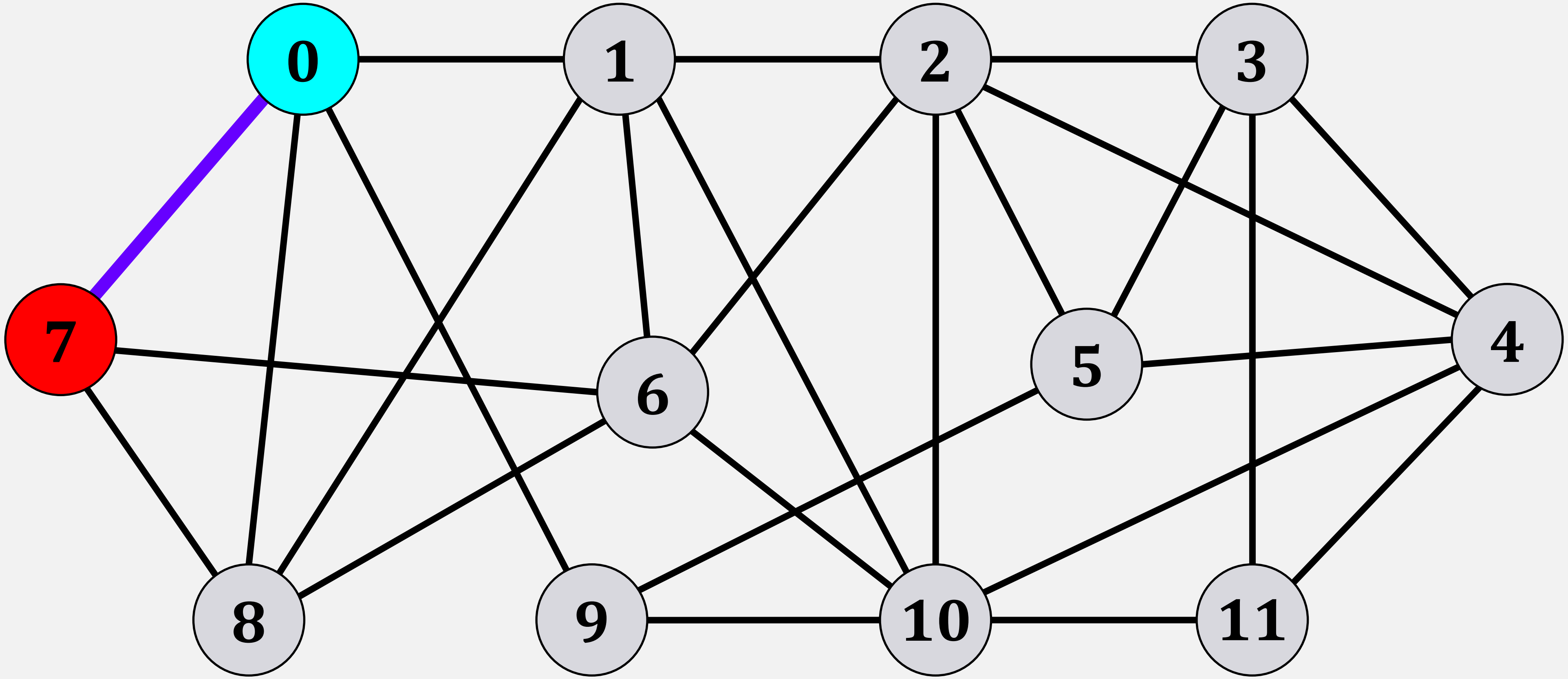
Depth-First Search (DFS)

Traversal: 7



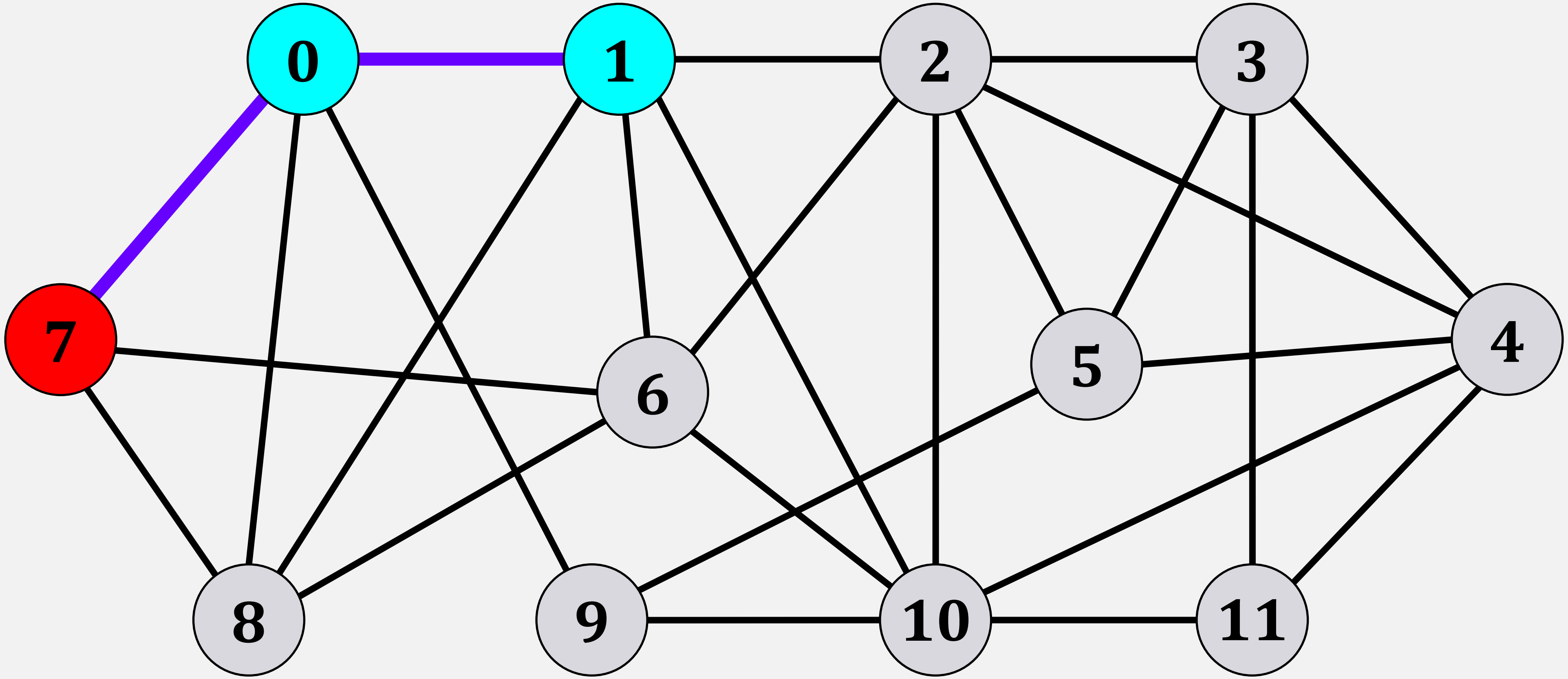
Depth-First Search (DFS)

Traversal: 7, 0



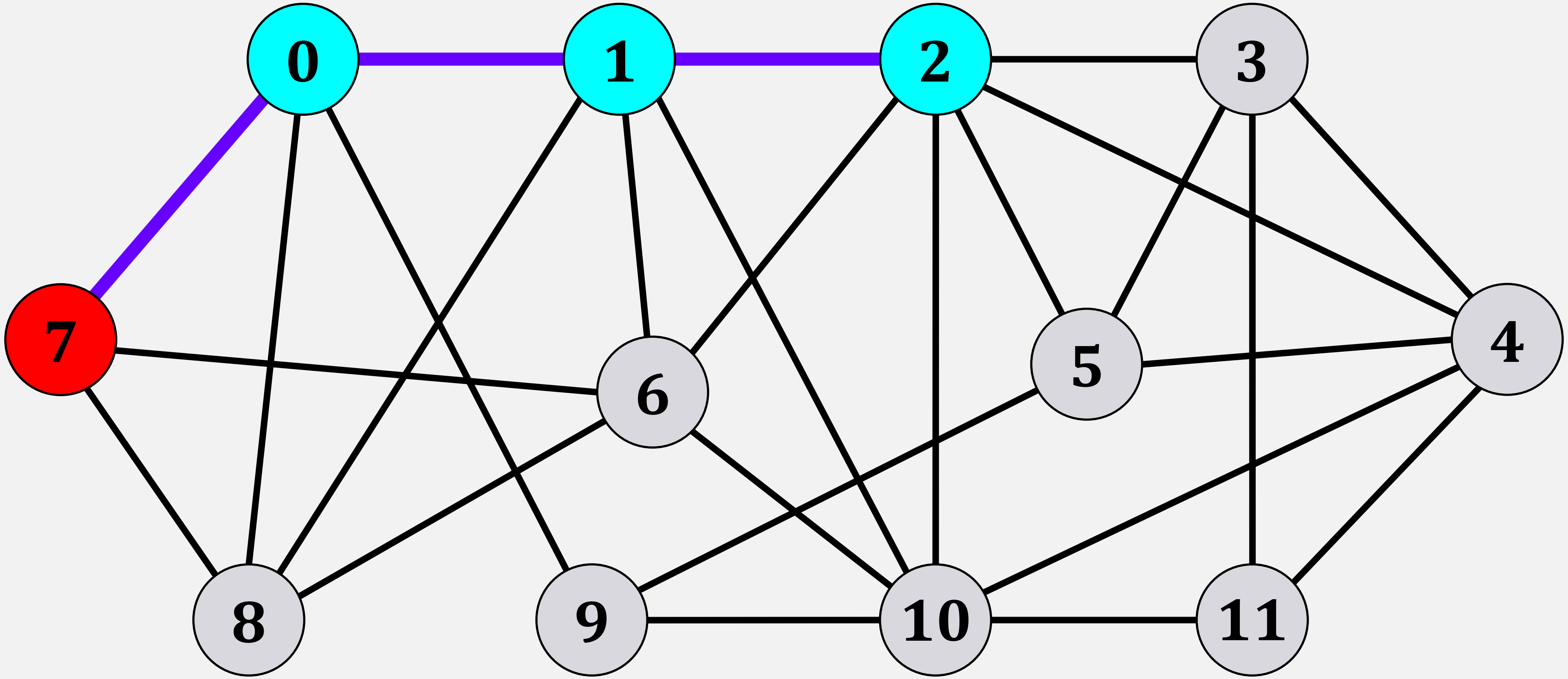
Depth-First Search (DFS)

Traversal: **7, 0, 1**



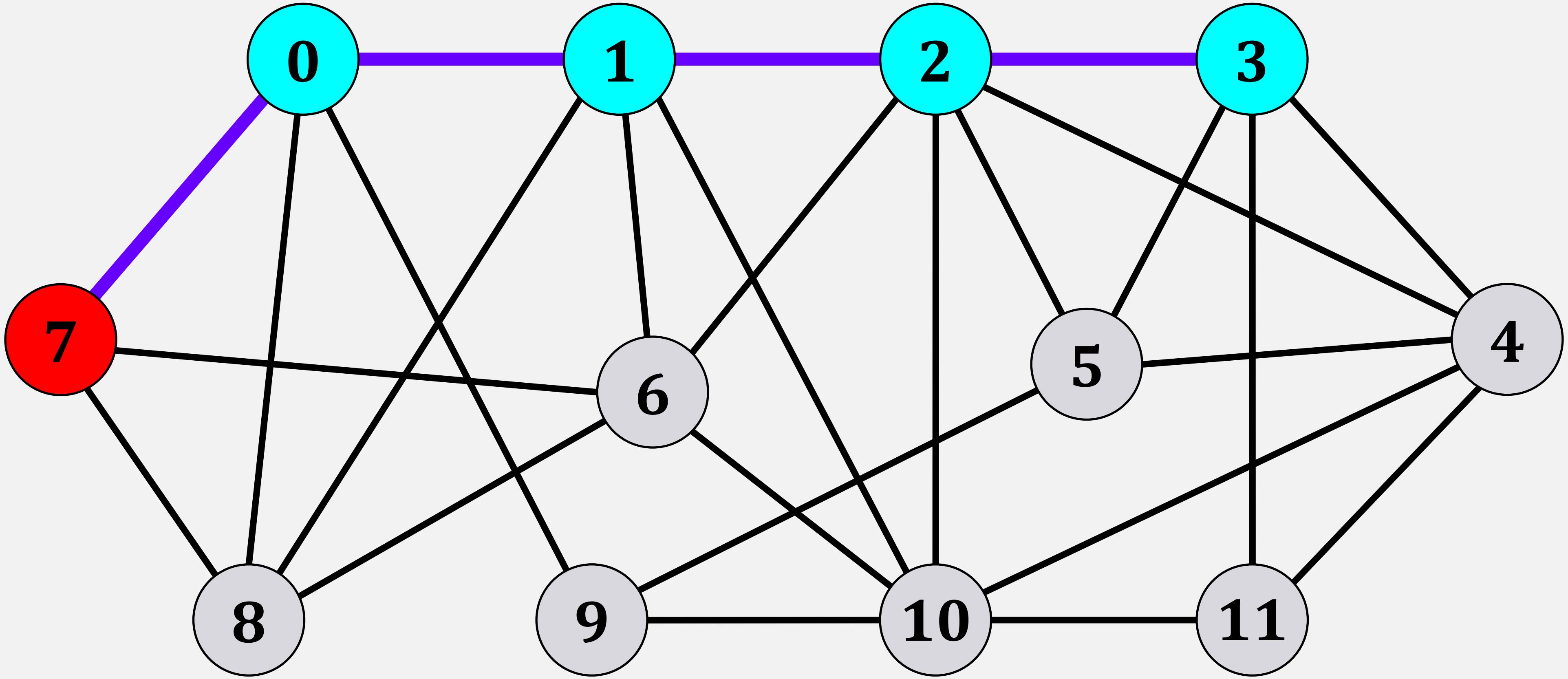
Depth-First Search (DFS)

Traversal: 7, 0, 1, 2



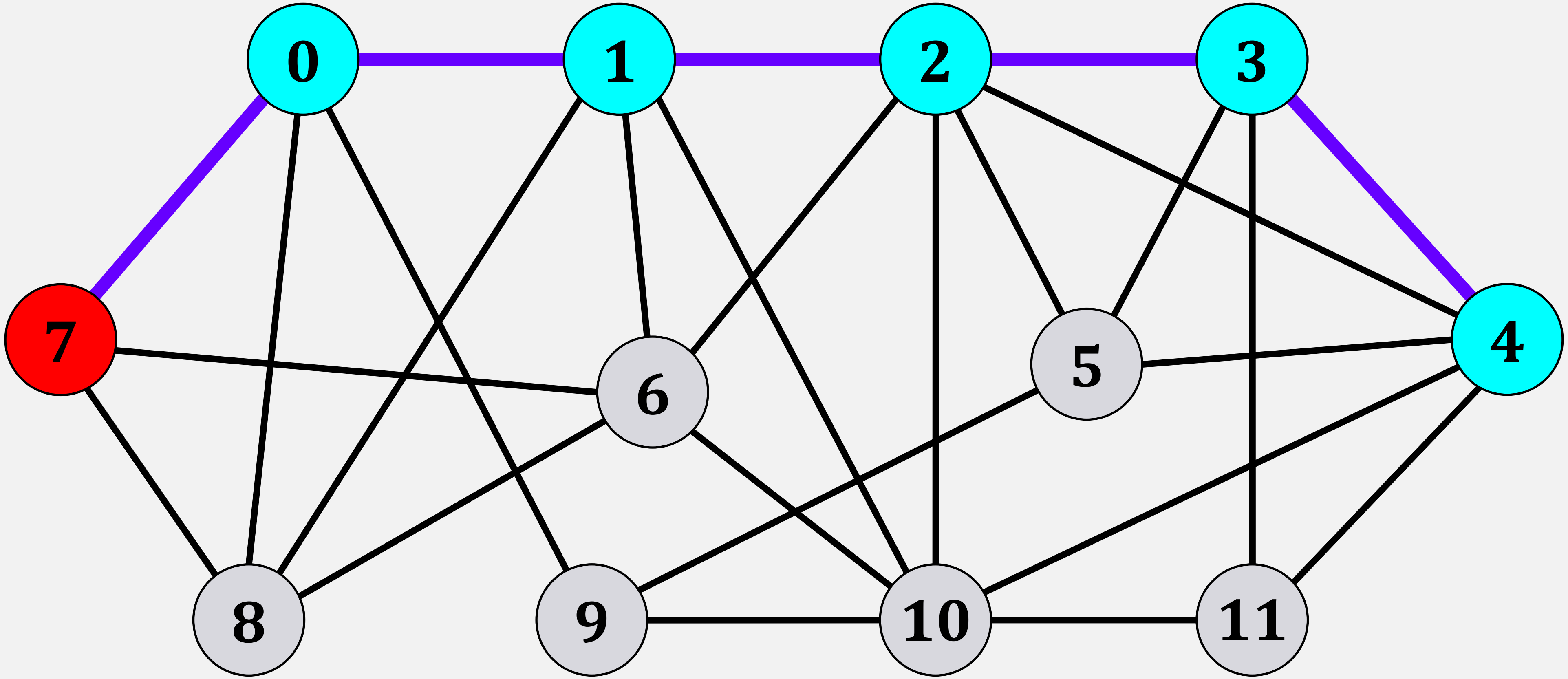
Depth-First Search (DFS)

Traversal: 7, 0, 1, 2, 3



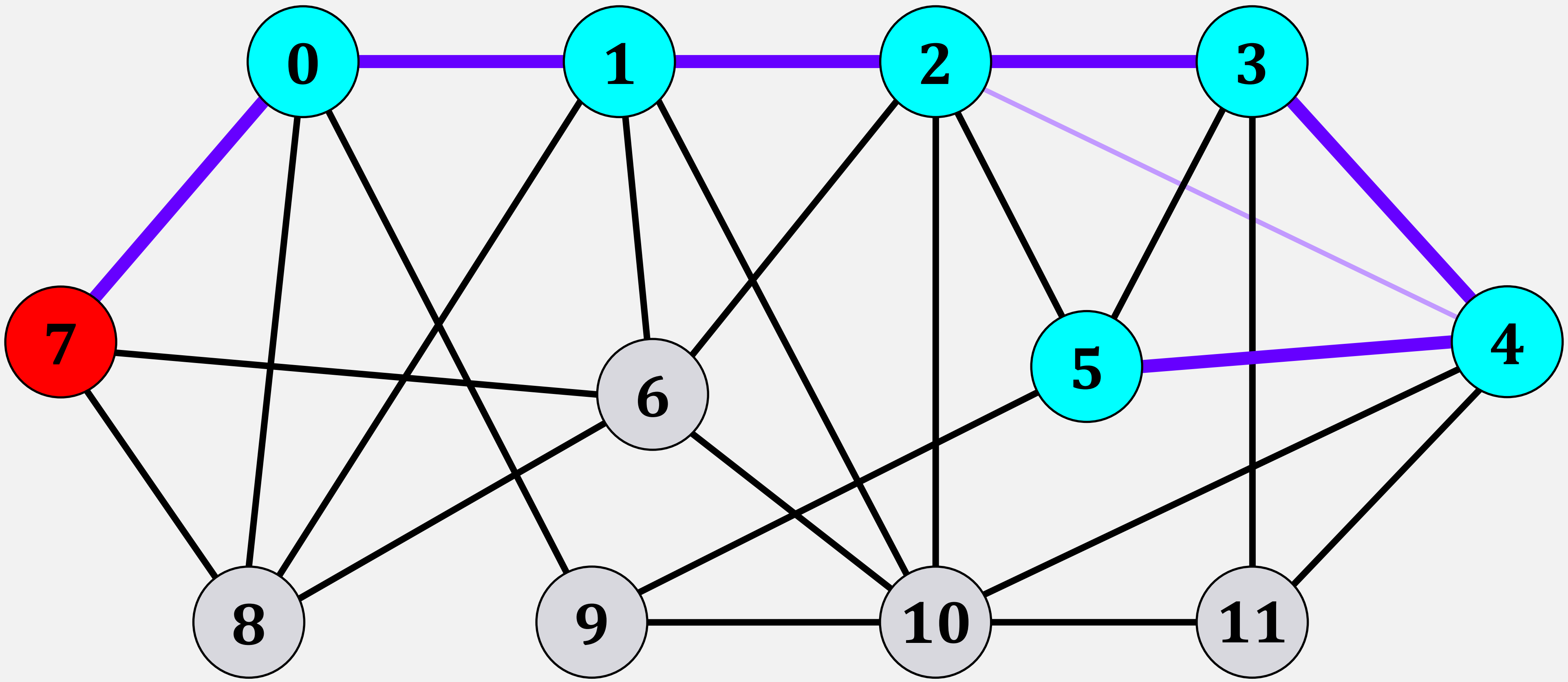
Depth-First Search (DFS)

Traversal: 7, 0, 1, 2, 3, 4



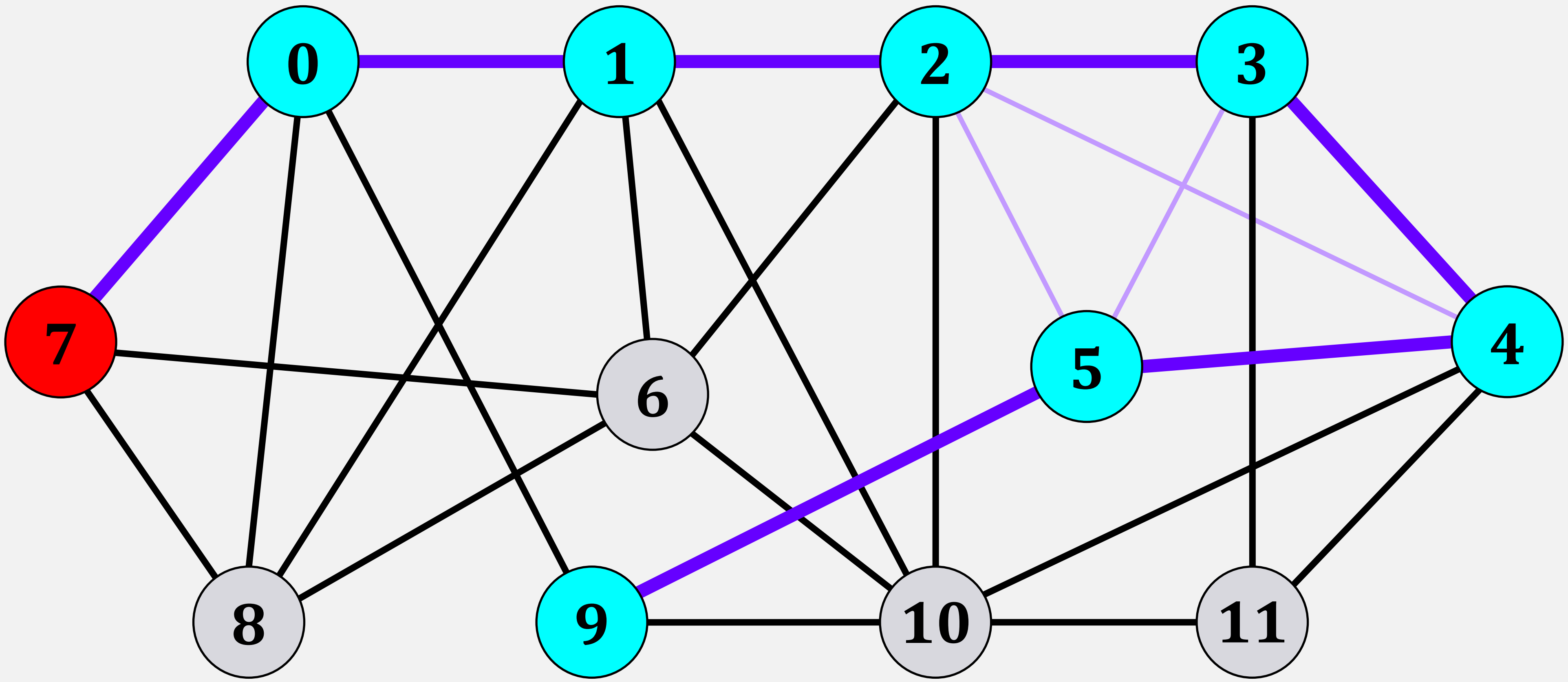
Depth-First Search (DFS)

Traversal: 7, 0, 1, 2, 3, 4, 5



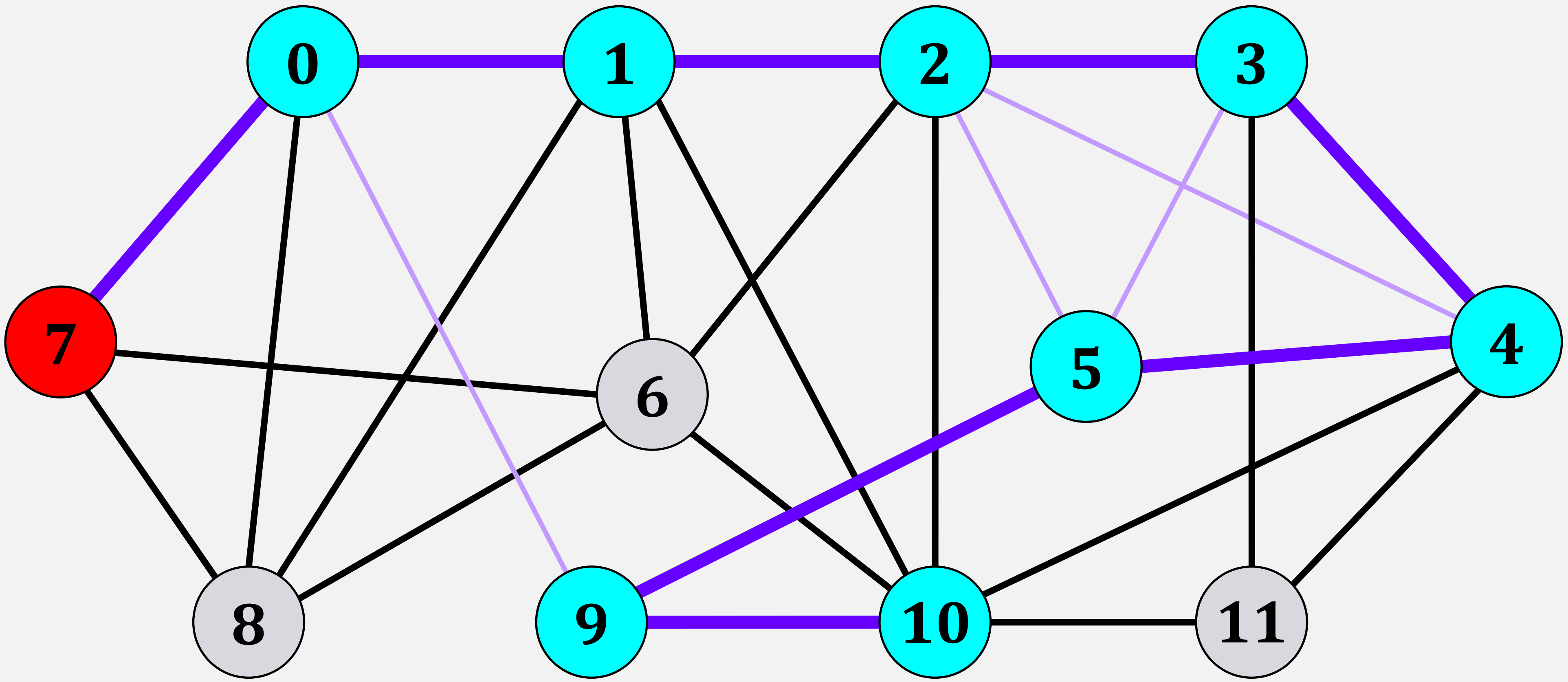
Depth-First Search (DFS)

Traversal: **7, 0, 1, 2, 3, 4, 5, 9**



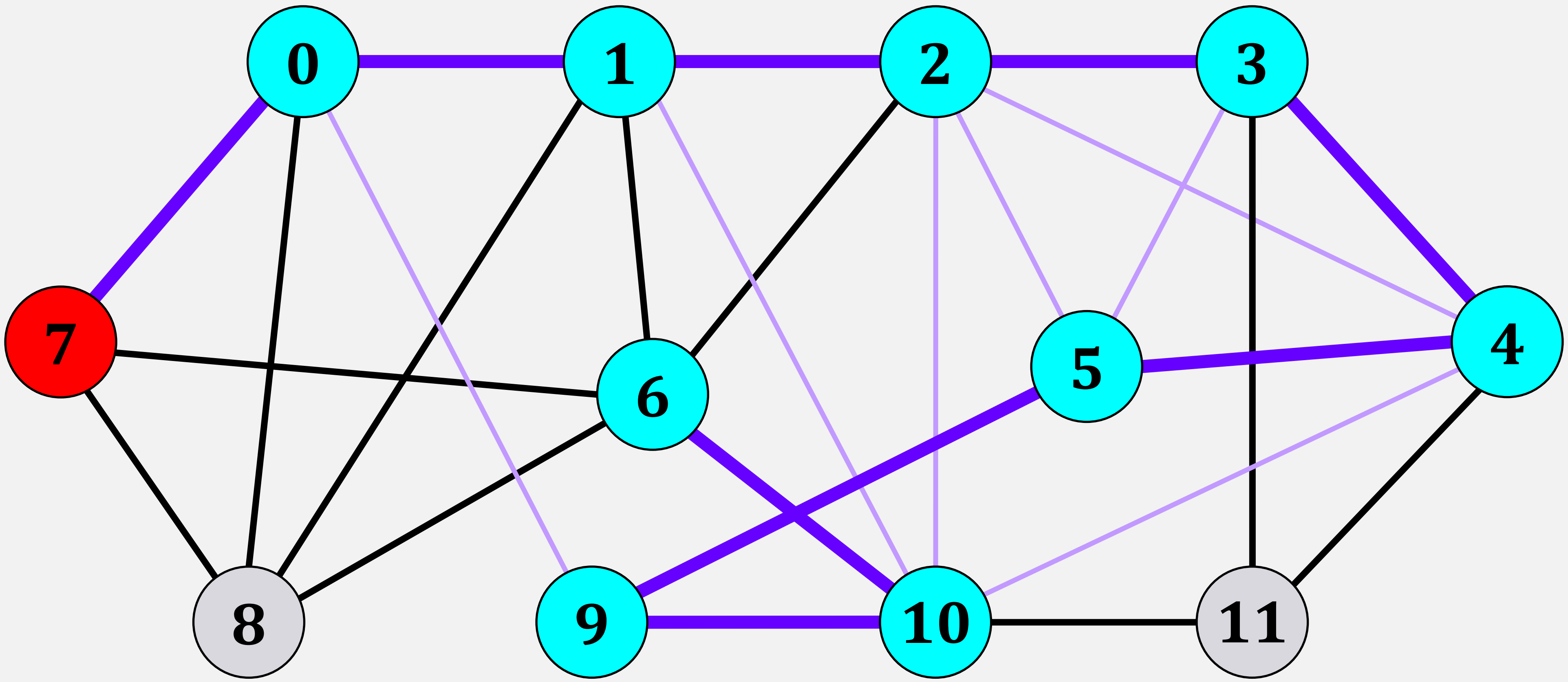
Depth-First Search (DFS)

Traversal: 7, 0, 1, 2, 3, 4, 5, 9, 10



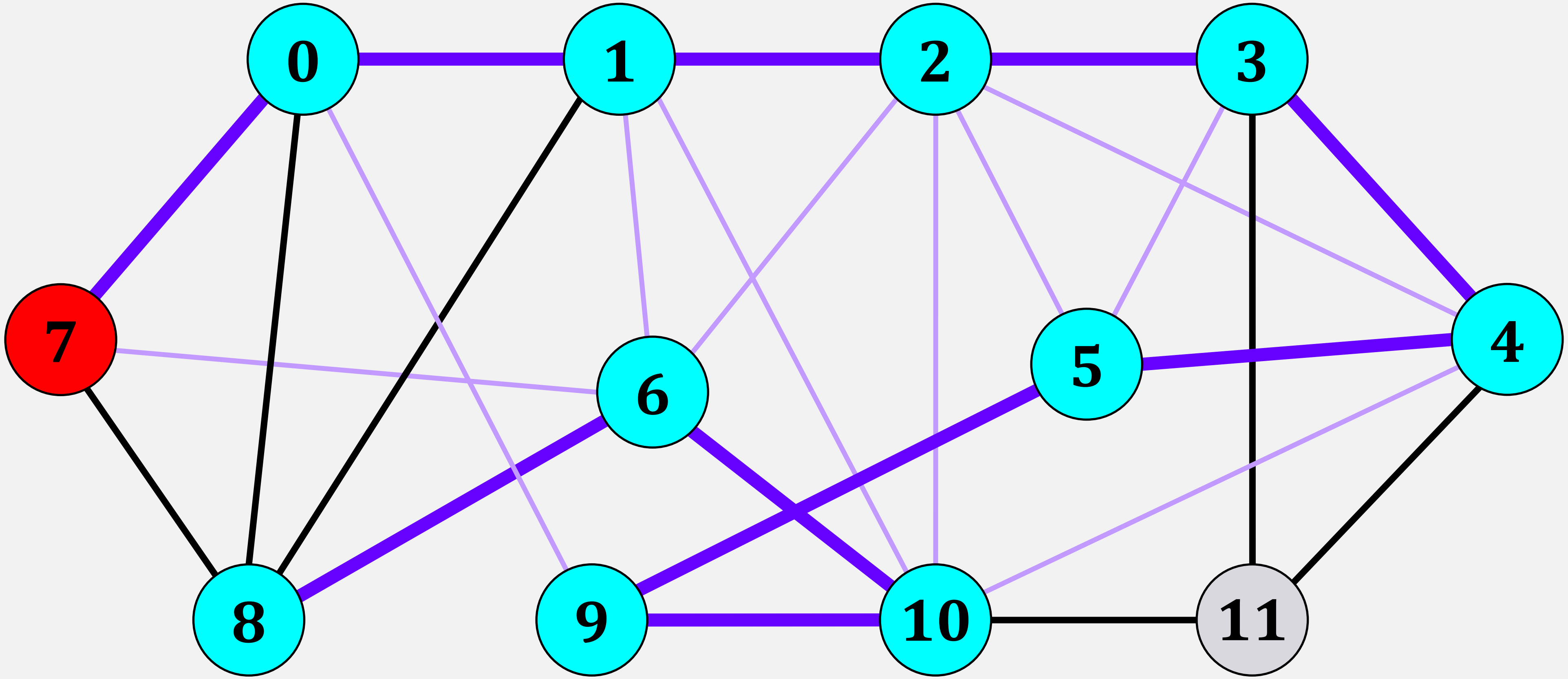
Depth-First Search (DFS)

Traversal: 7, 0, 1, 2, 3, 4, 5, 9, 10, 6



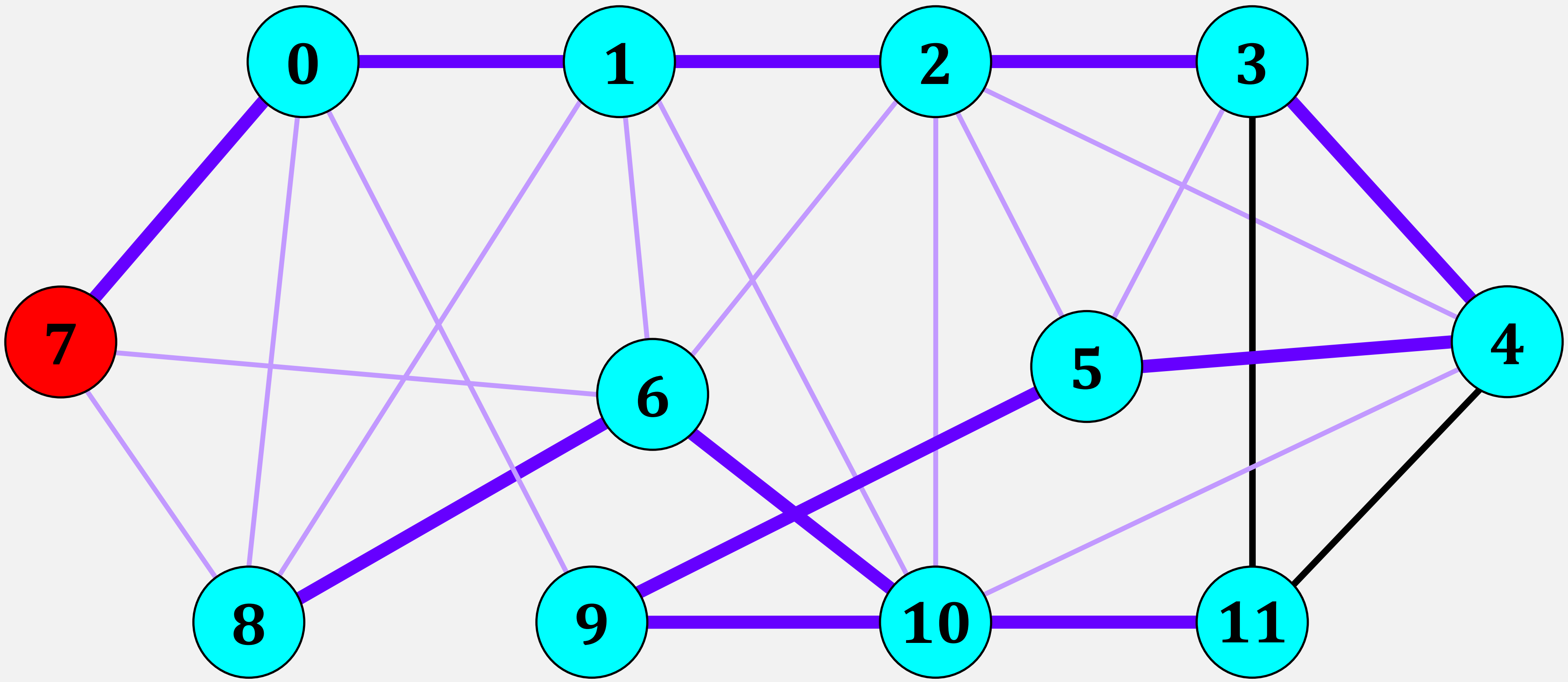
Depth-First Search (DFS)

Traversal: 7, 0, 1, 2, 3, 4, 5, 9, 10, 6, 8



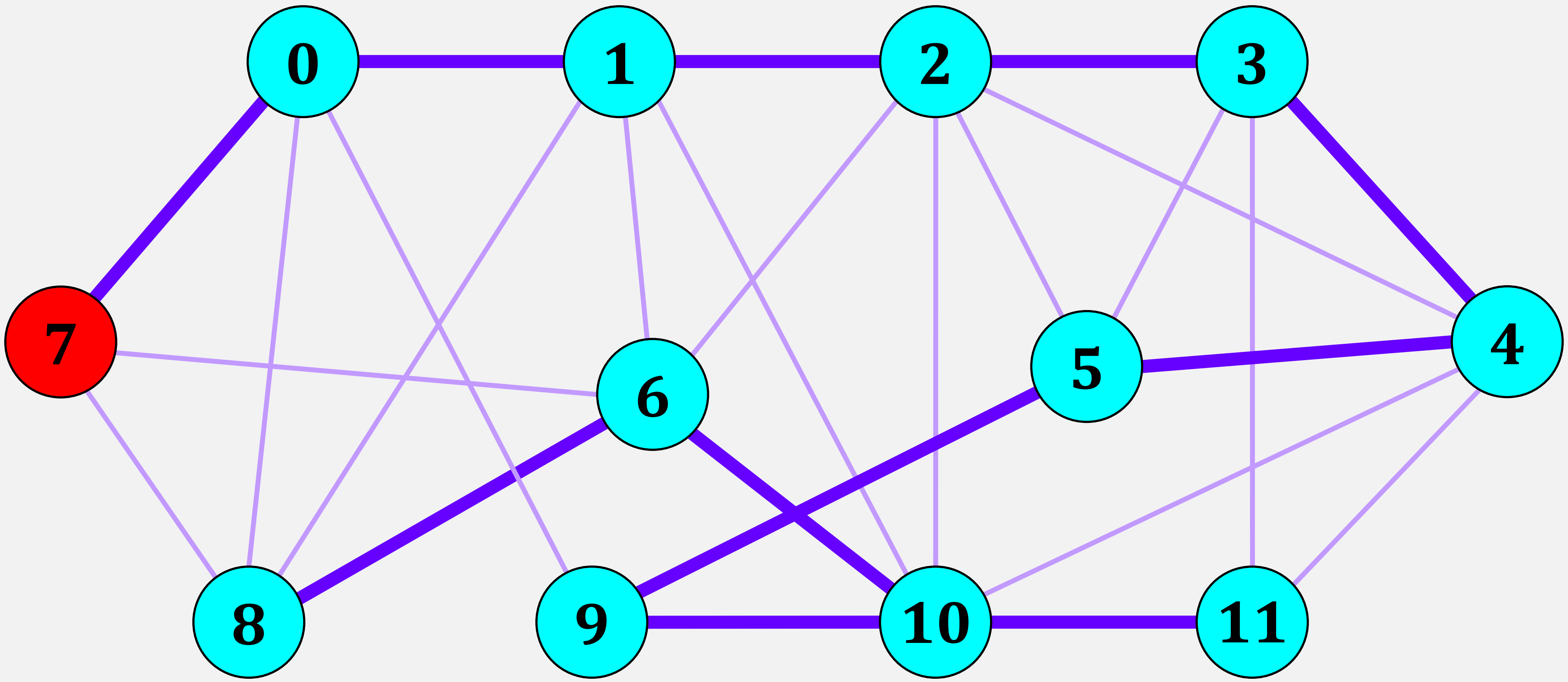
Depth-First Search (DFS)

Traversal: 7, 0, 1, 2, 3, 4, 5, 9, 10, 6, 8, 11



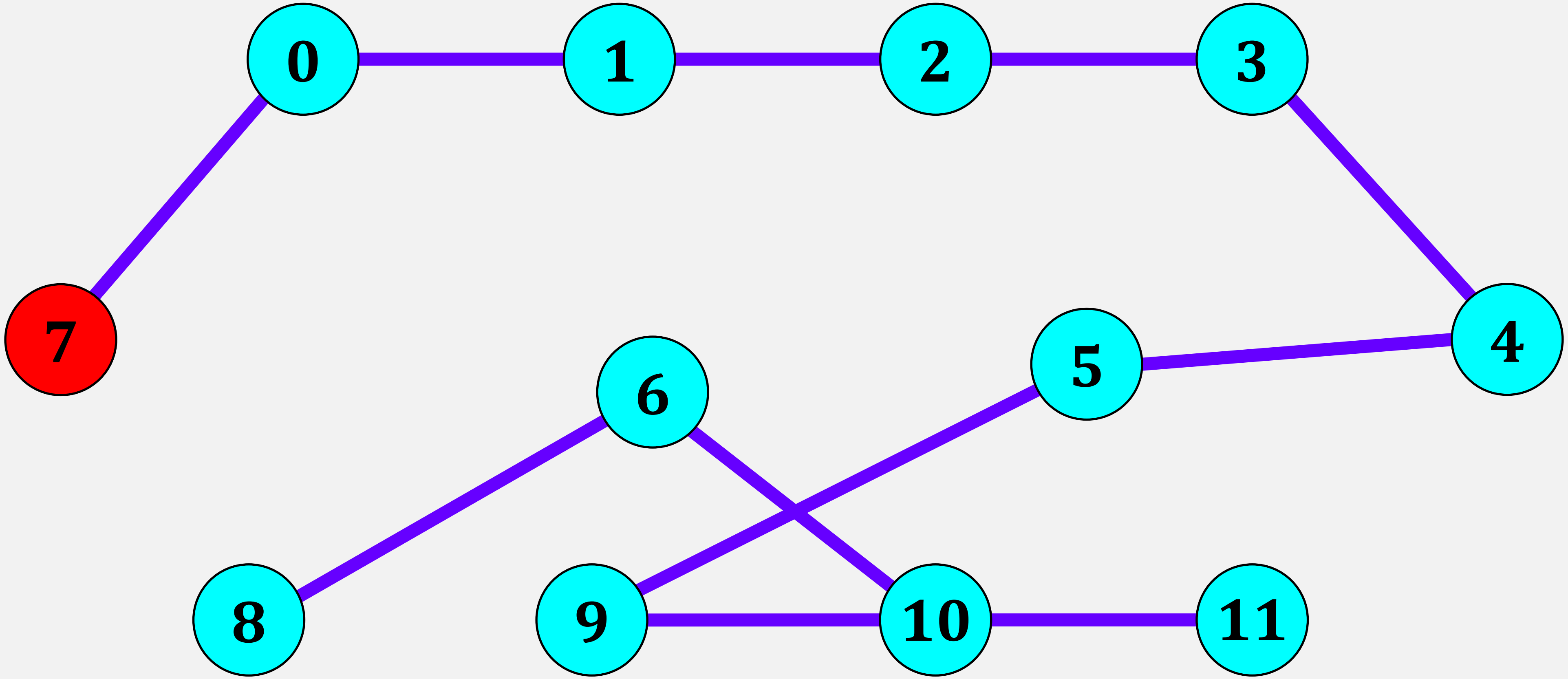
Depth-First Search (DFS)

Traversal: 7, 0, 1, 2, 3, 4, 5, 9, 10, 6, 8, 11



Depth-First Search (DFS)

Traversal: **7, 0, 1, 2, 3, 4, 5, 9, 10, 6, 8, 11**



Spanning subtree generated by **DFS** with root **7**

Theorem 12.4

When given as input a Graph, $\boldsymbol{g}$, that is implemented using the **AdjacencyLists** data structure, the **dfs**$(\boldsymbol{g}, \boldsymbol{r})$ and the non-recursive **dfs2**$(\boldsymbol{g}, \boldsymbol{r})$ algorithms each run in $\boldsymbol{O}(\boldsymbol{n} + \boldsymbol{m})$ time.

Breadth-First Search (BFS)

Given: $\boldsymbol{G = (V, E)}$ and vertex $\boldsymbol{v_i}$.

Goal: Find all the vertices that can be reached from $\boldsymbol{v_i}$.

- Starting from some node $\boldsymbol{v_i}$, visit all of $\boldsymbol{v_i}$'s neighbours (vertices adjacent to $\boldsymbol{v_i}$) that we have not seen yet.
- Then visit all the neighbours of the neighbours of $\boldsymbol{v_i}$ that have not been visited yet.
- Then visit the neighbours of the neighbours of the neighbours…

When given a choice, always choose the smallest element to visit (smallest number, letter, etc.)

Breadth-First Search (BFS)

Given: $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ and vertex $\boldsymbol{v_i}$.

Goal: Find all the vertices that can be reached from $\boldsymbol{v_i}$.

- Keep a **queue** of **visited** vertices (including the start)
- Add all adjacent vertices to the **queue** (at the end), provided that these neighbours have never been in **queue** before.

- **Remove** vertices from the **front** of the **queue**, and
- **Add** their unvisited neighbours to the **end** of the **queue**

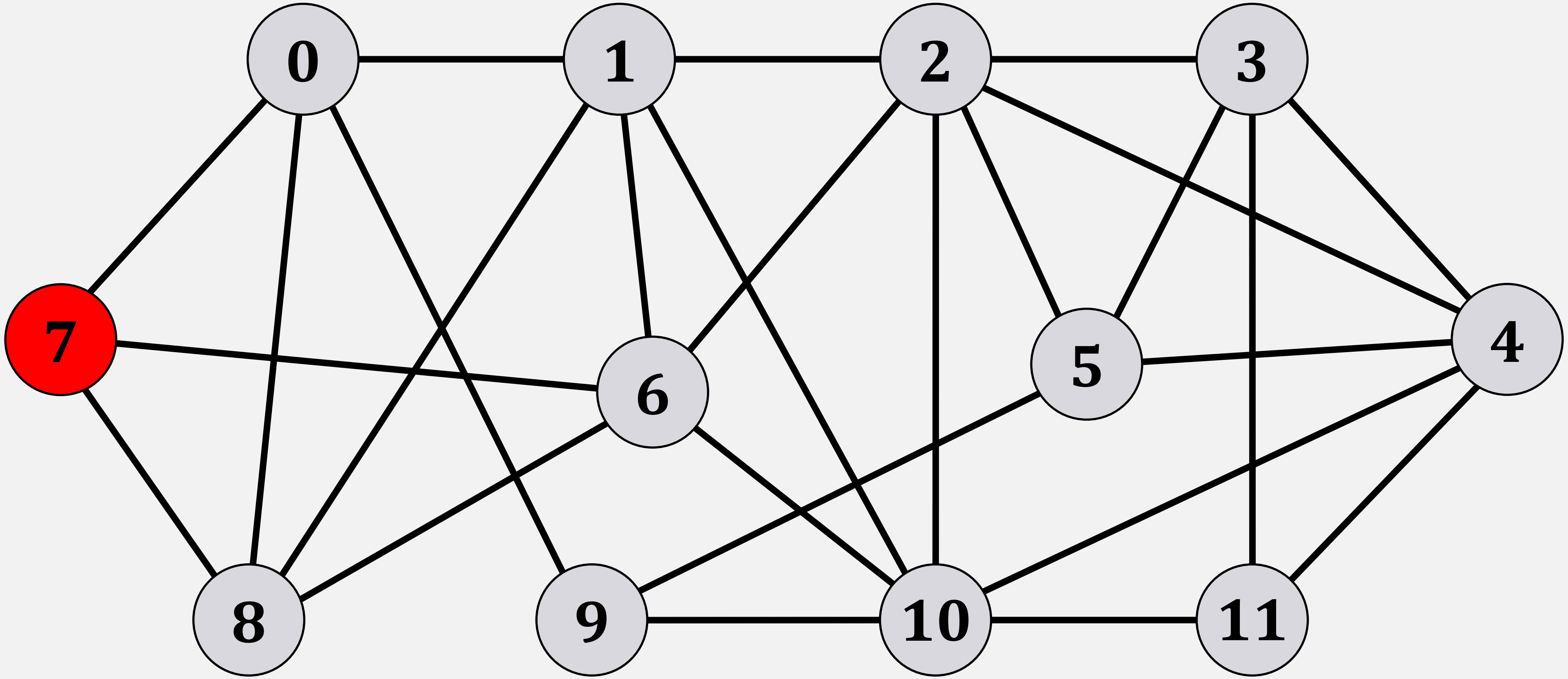
Breadth-First Search (BFS) $O(|V| + |E|)$

the algorithm never adds the same vertex to **q** more than once

```
void bfs(Graph g, int r) {
  boolean[] seen = new boolean[g.nVertices()];
  Queue<Integer> q = new SLList<Integer>();
  q.add(r);
  seen[r] = true;
  while (!q.isEmpty()) {
    int i = q.remove();
    for (Integer j : g.outEdges(i)) {
      if (!seen[j]) {
        q.add(j);
        seen[j] = true;
      }
    }
  }
}
```

Breadth-First Search (BFS)

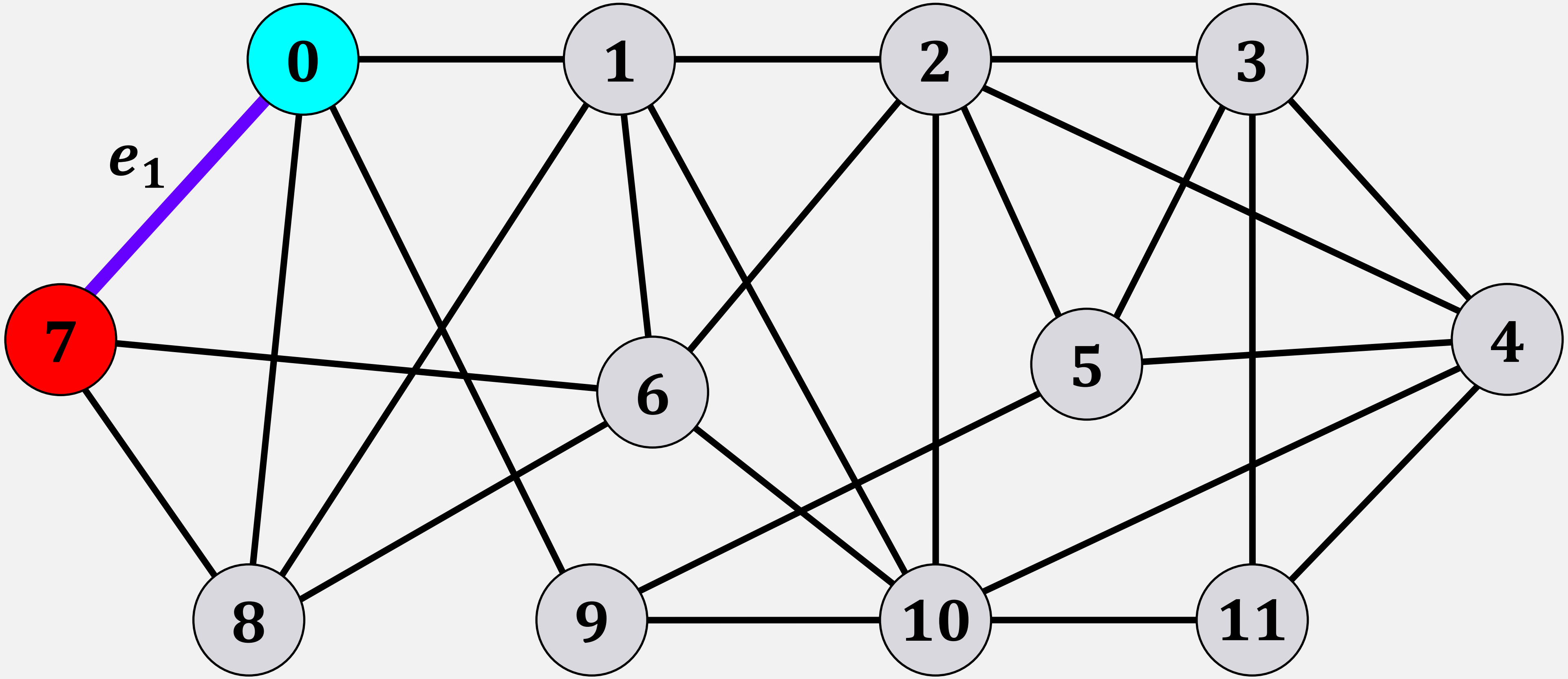
traversal → q contains **7,**



Perform a BFS starting with vertex **7**

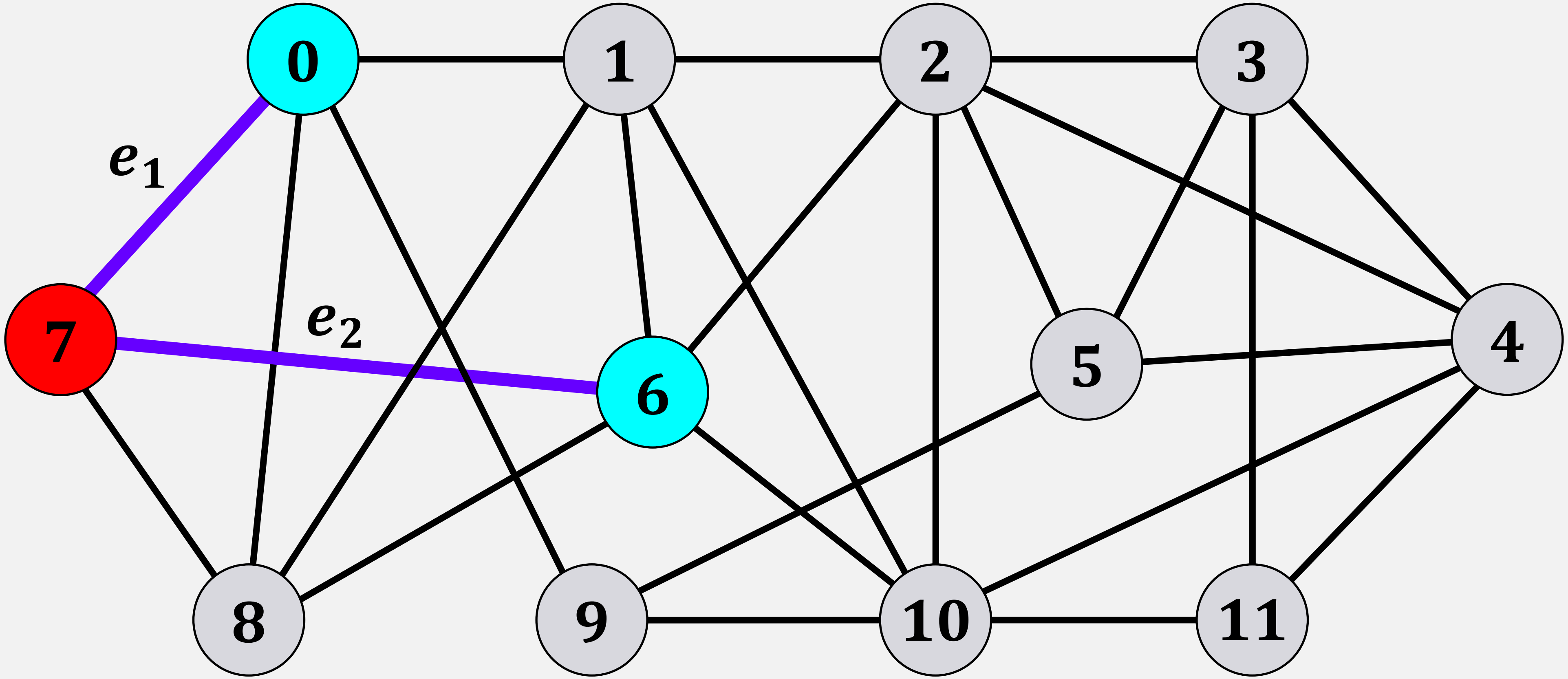
Breadth-First Search (BFS)

traversal → q contains 7, 0,



Breadth-First Search (BFS)

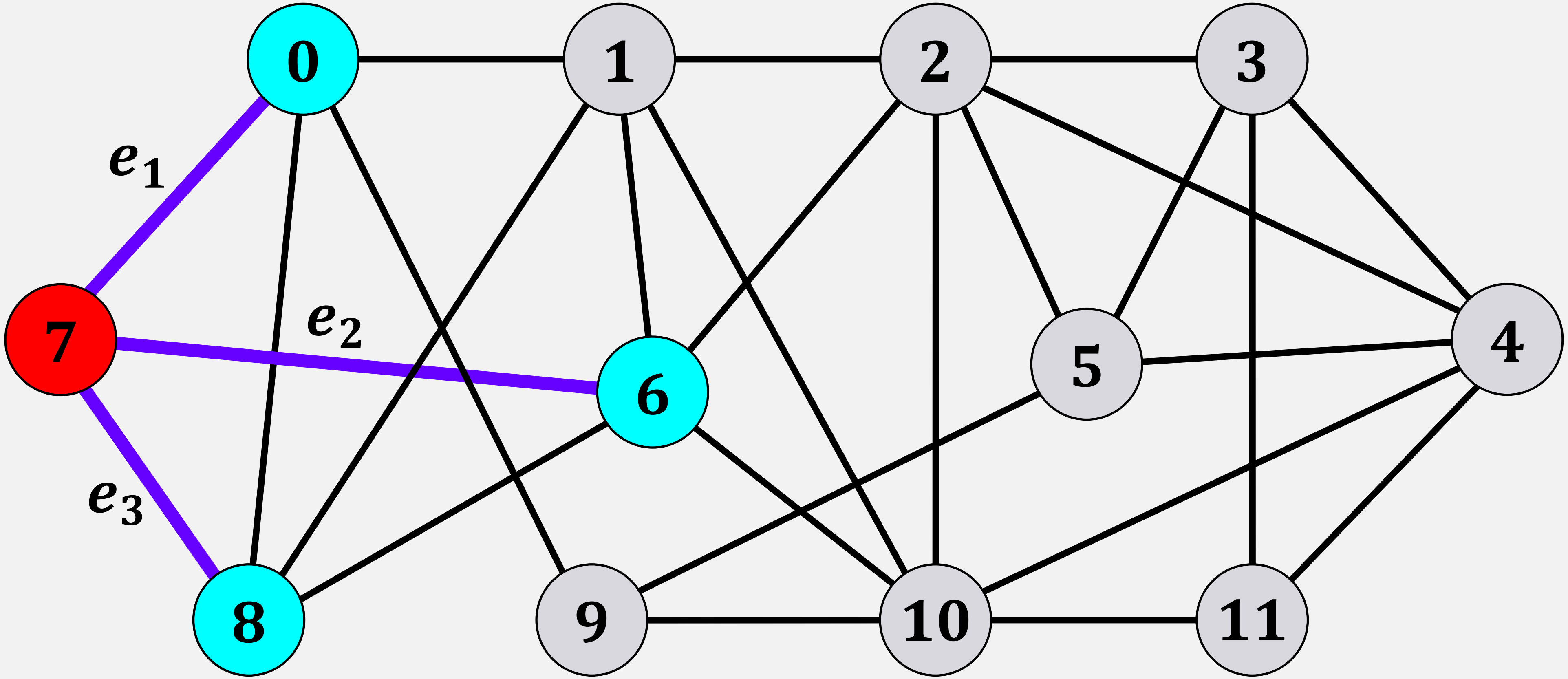
traversal → q contains 7, **0, 6,**



Alina Shaikhet – COMP2406 – Carleton University

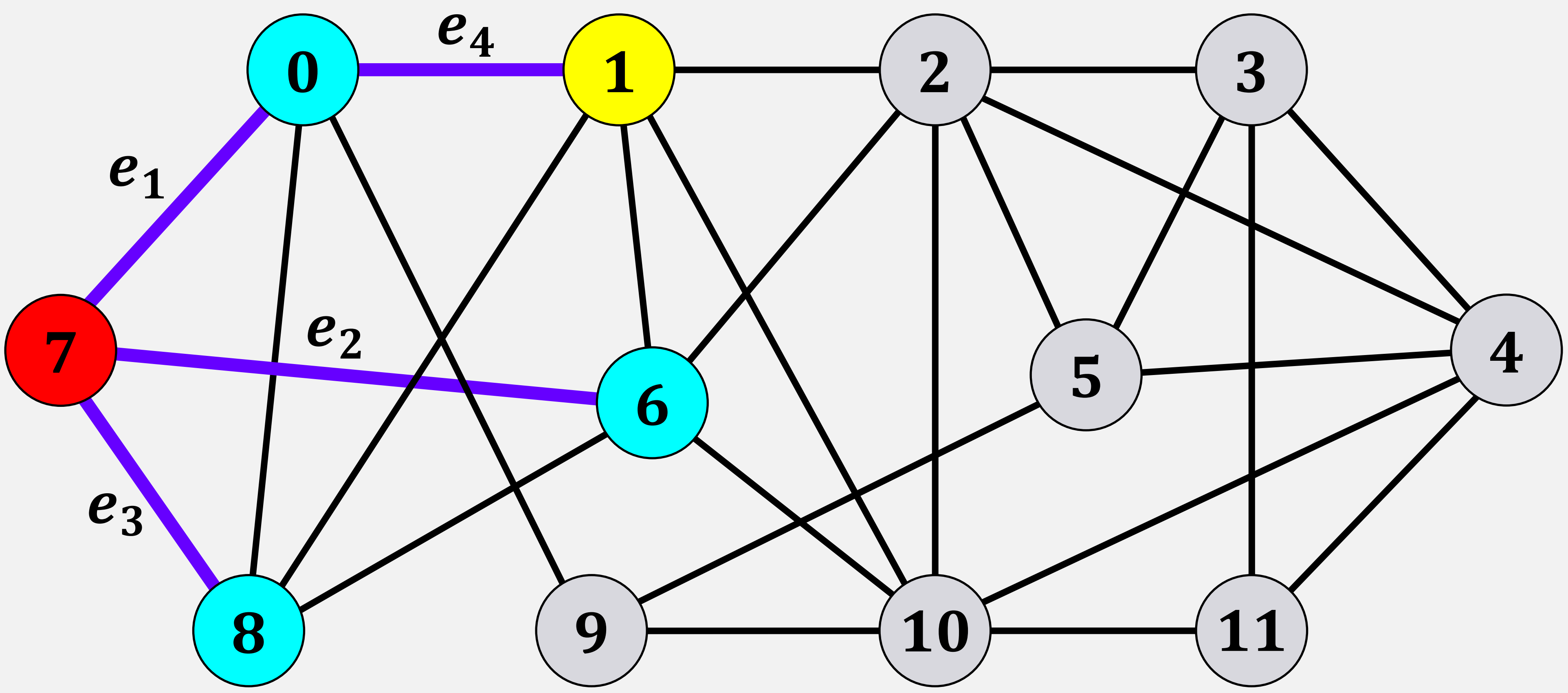
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8,



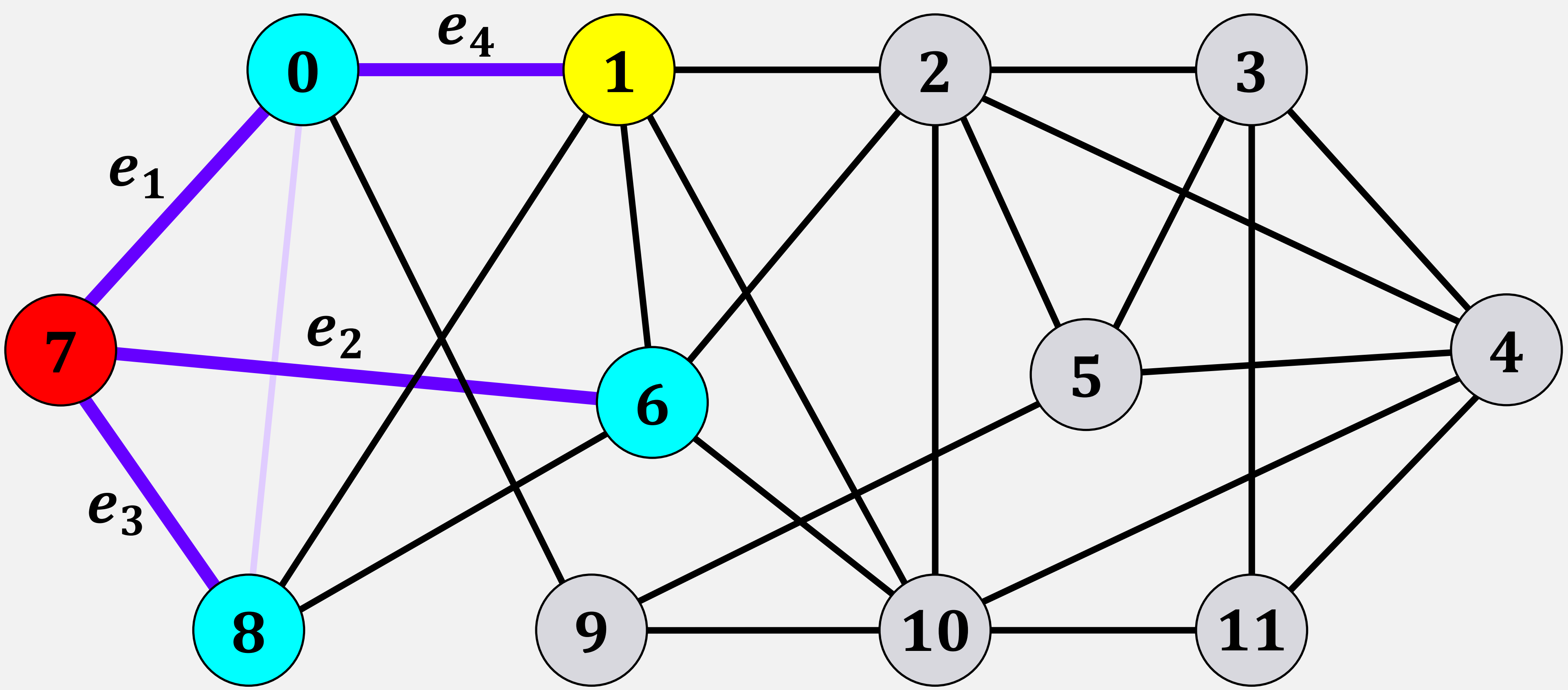
Breadth-First Search (BFS)

traversal → q contains $7, 0, 6, 8, 1,$



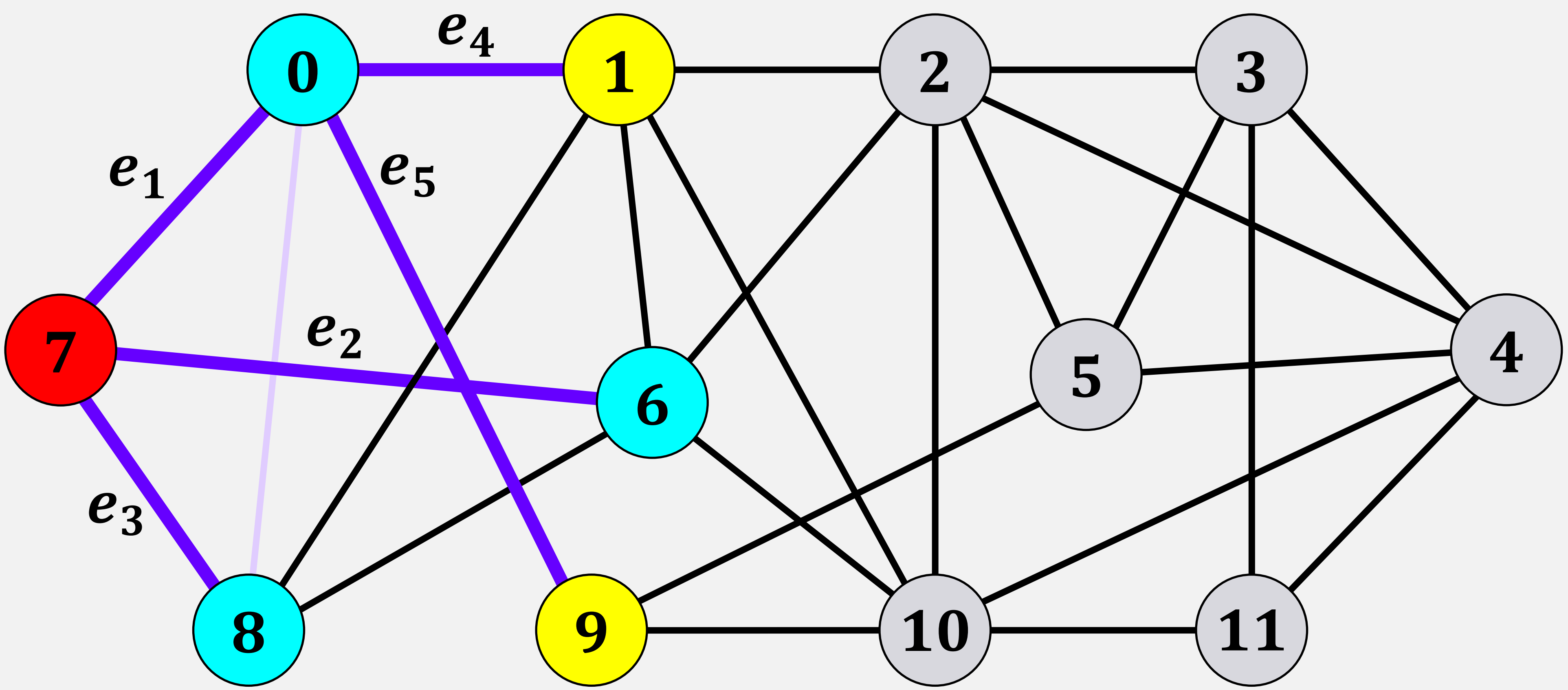
Breadth-First Search (BFS)

traversal → q contains 7, 0, **6, 8, 1,**



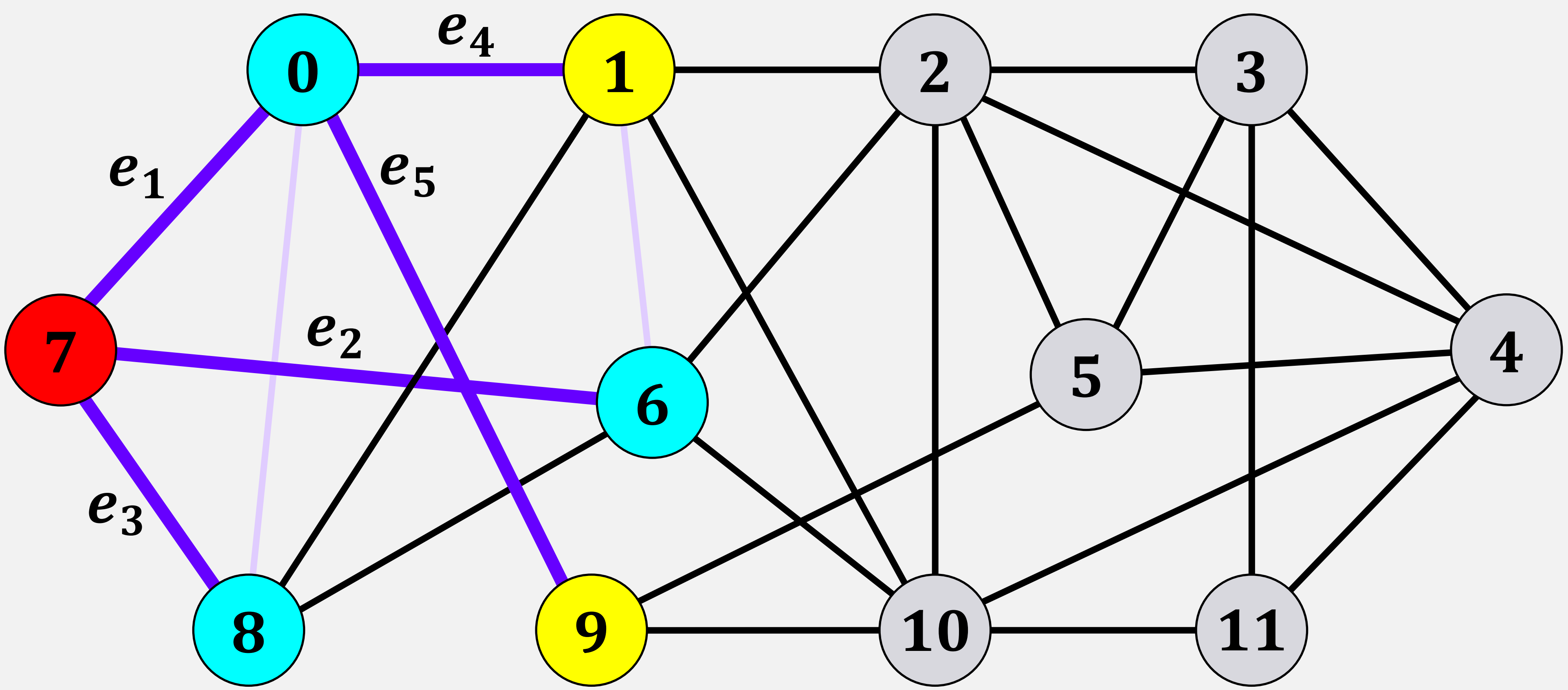
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, 9,



Breadth-First Search (BFS)

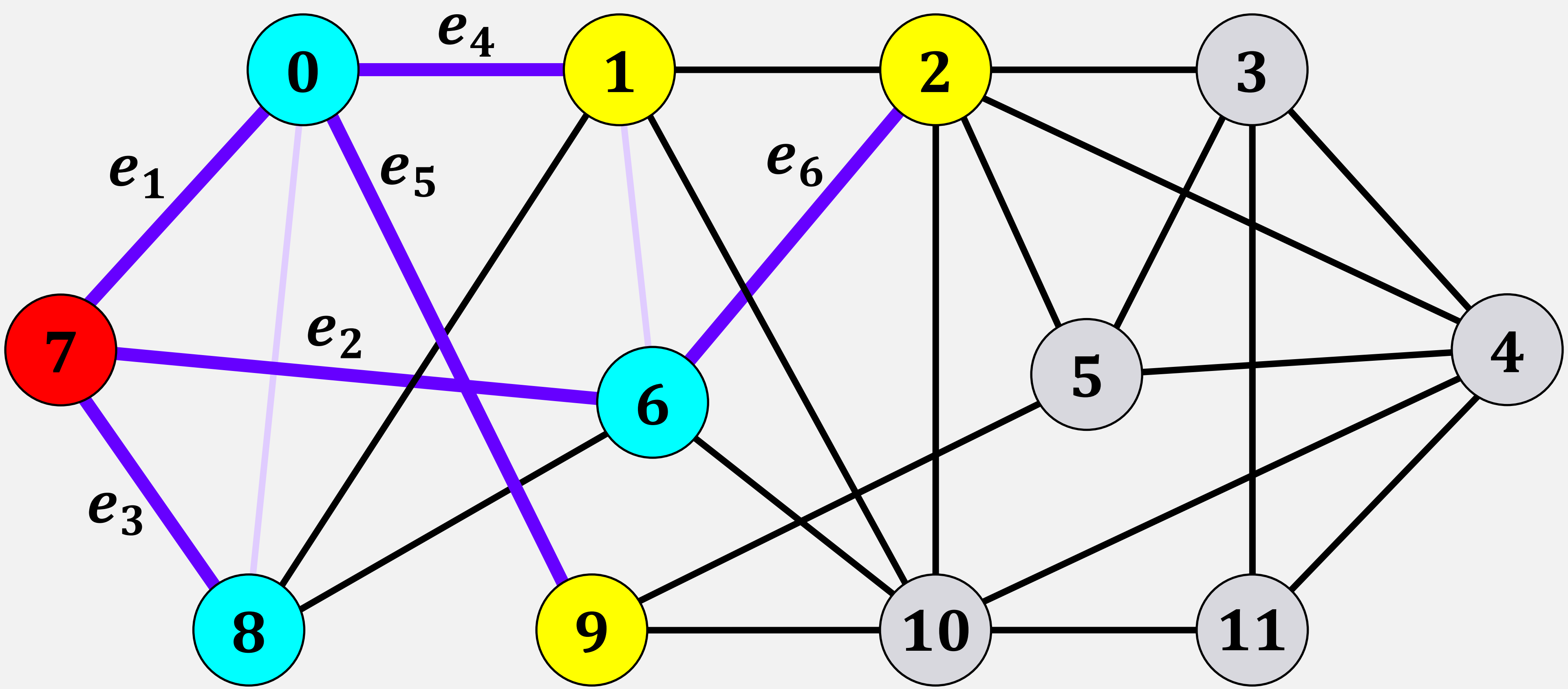
traversal → q contains 7, 0, 6, **8, 1, 9,**



Alina Shaikhet – COMP2406 – Carleton University

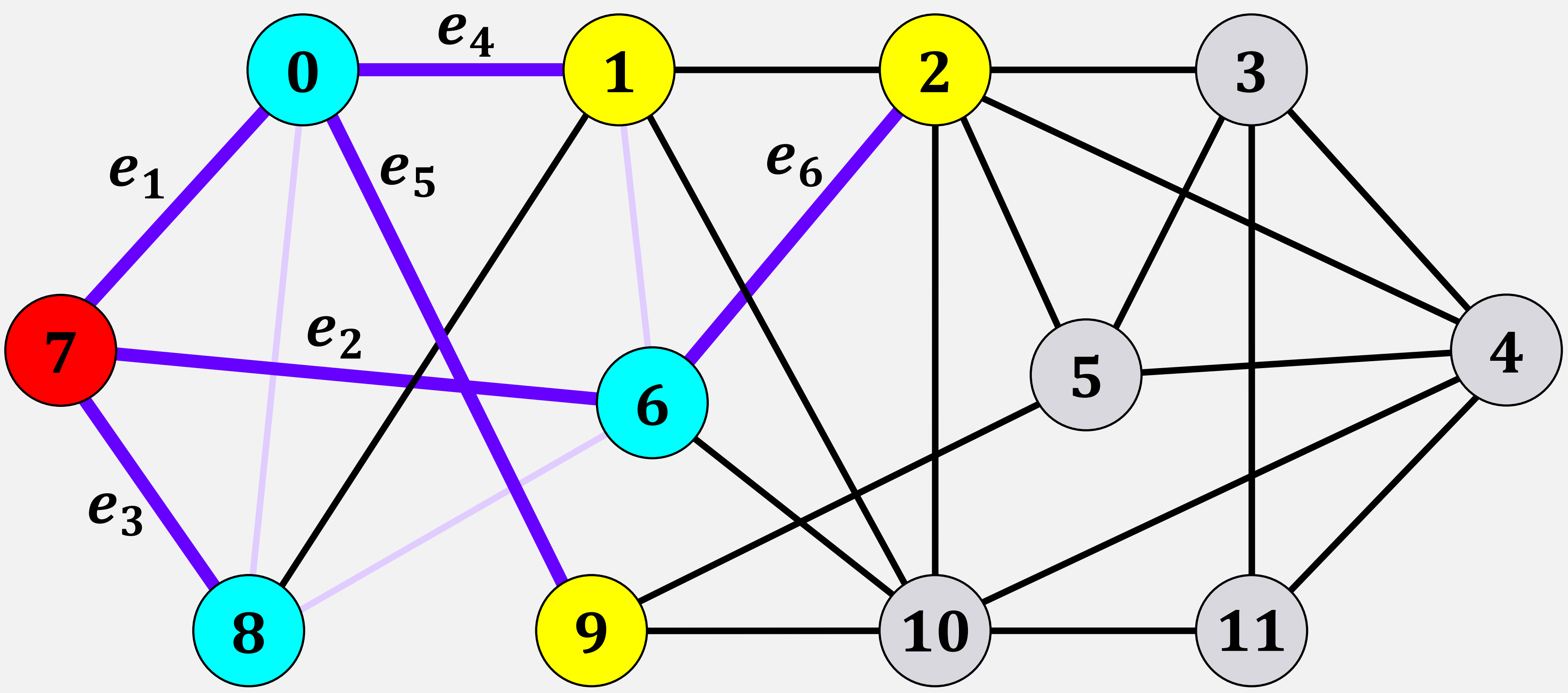
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, **8, 1, 9, 2,**



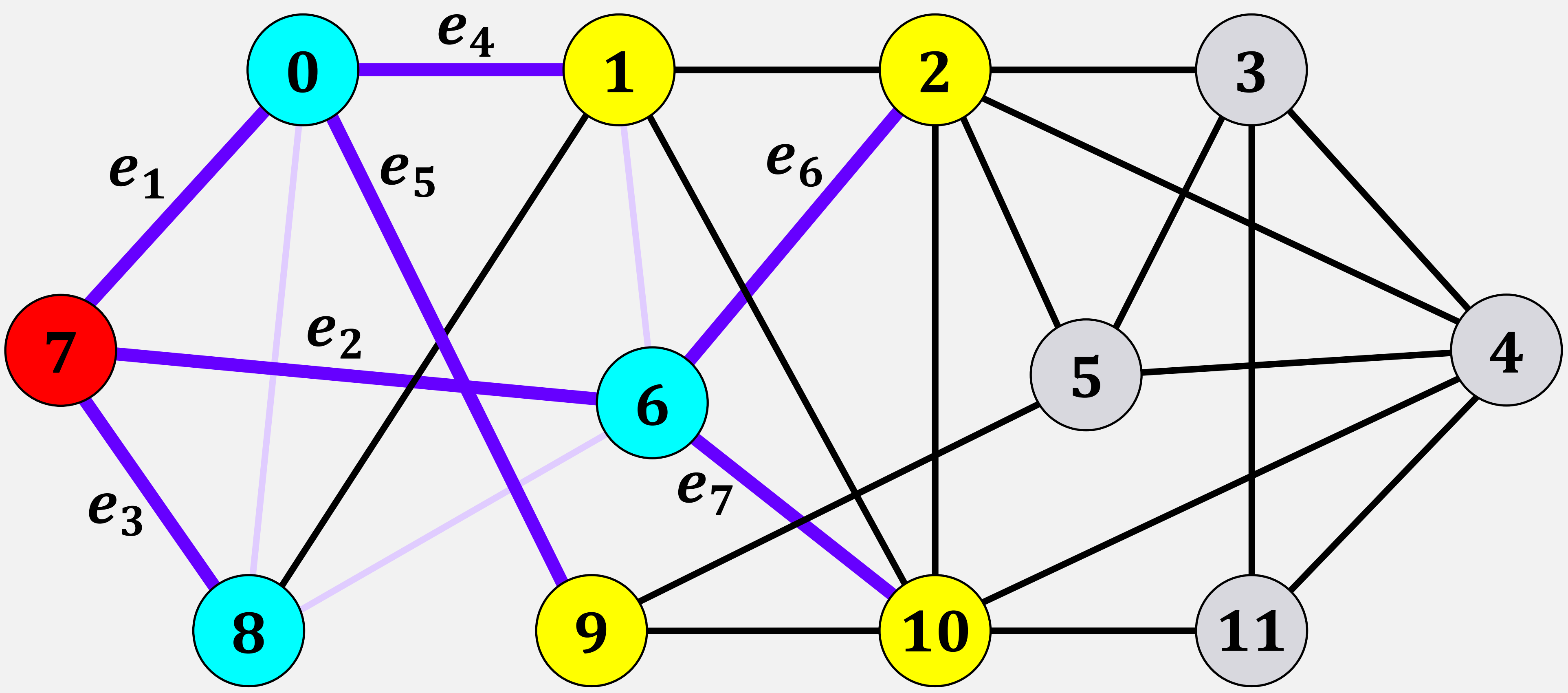
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, 9, 2,



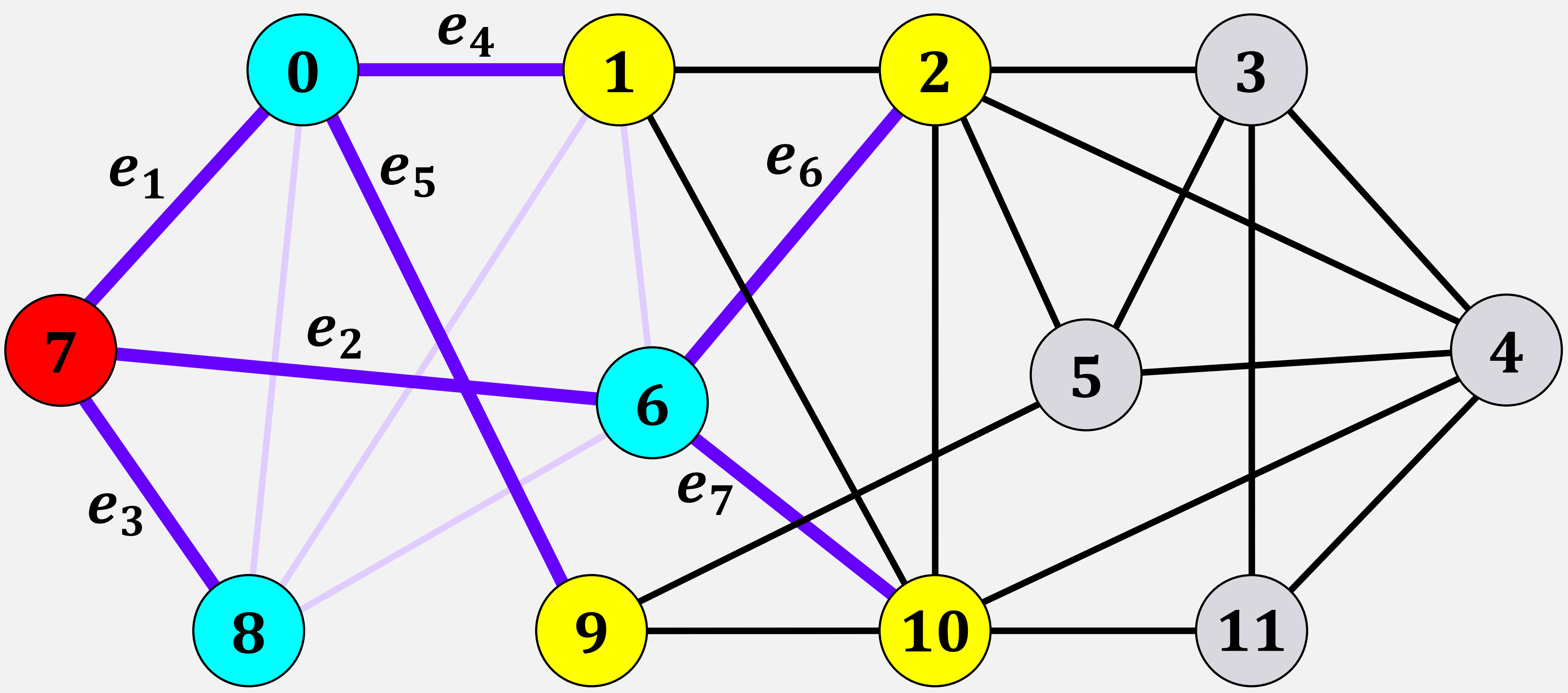
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, **8, 1, 9, 2, 10,**



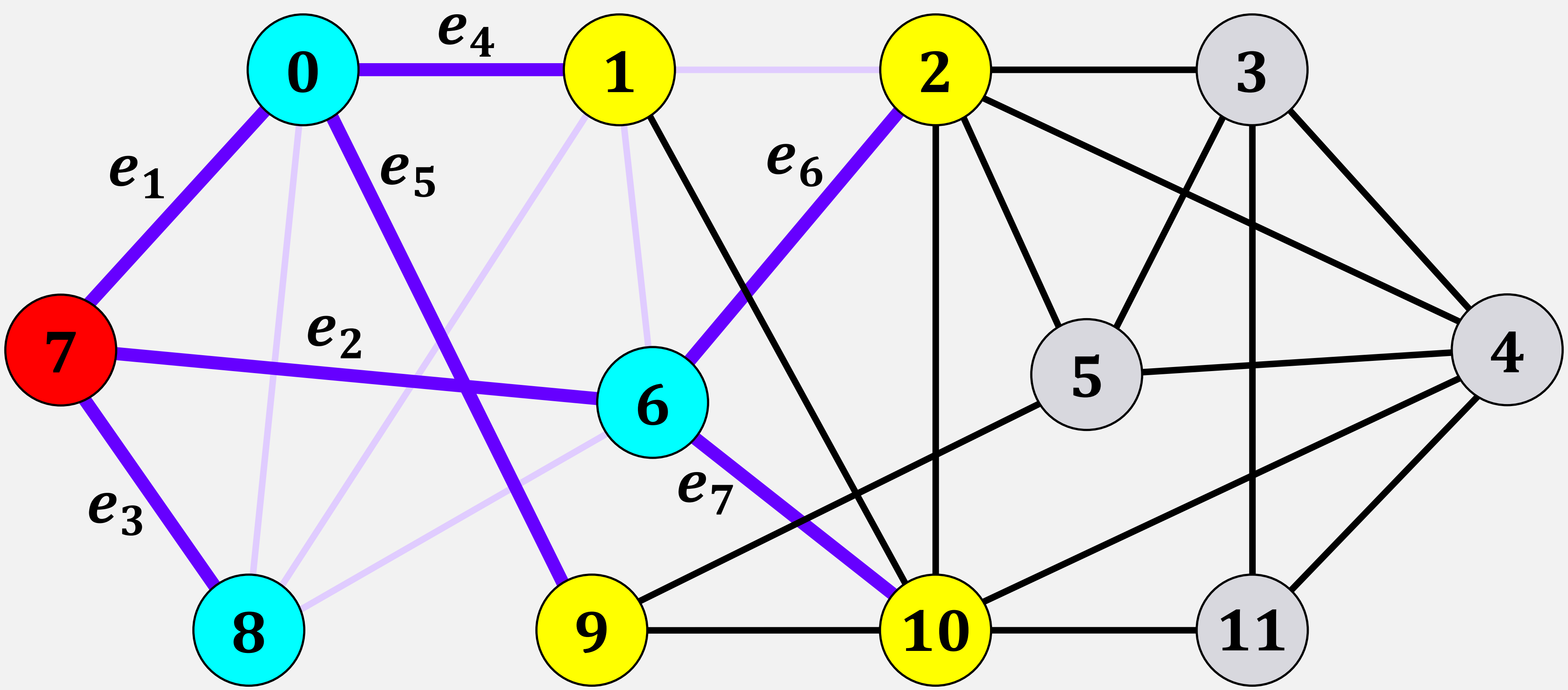
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, **1, 9, 2, 10,**



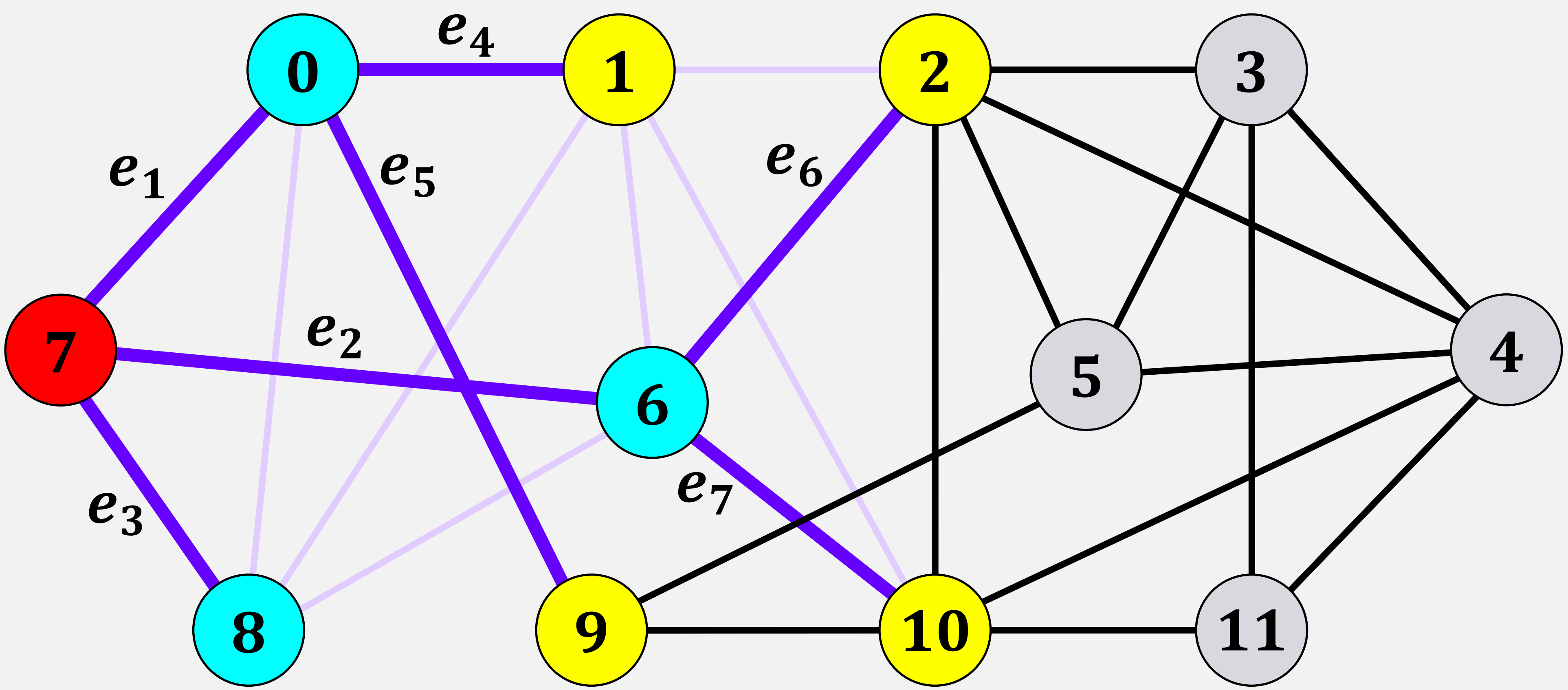
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, **9, 2, 10,**



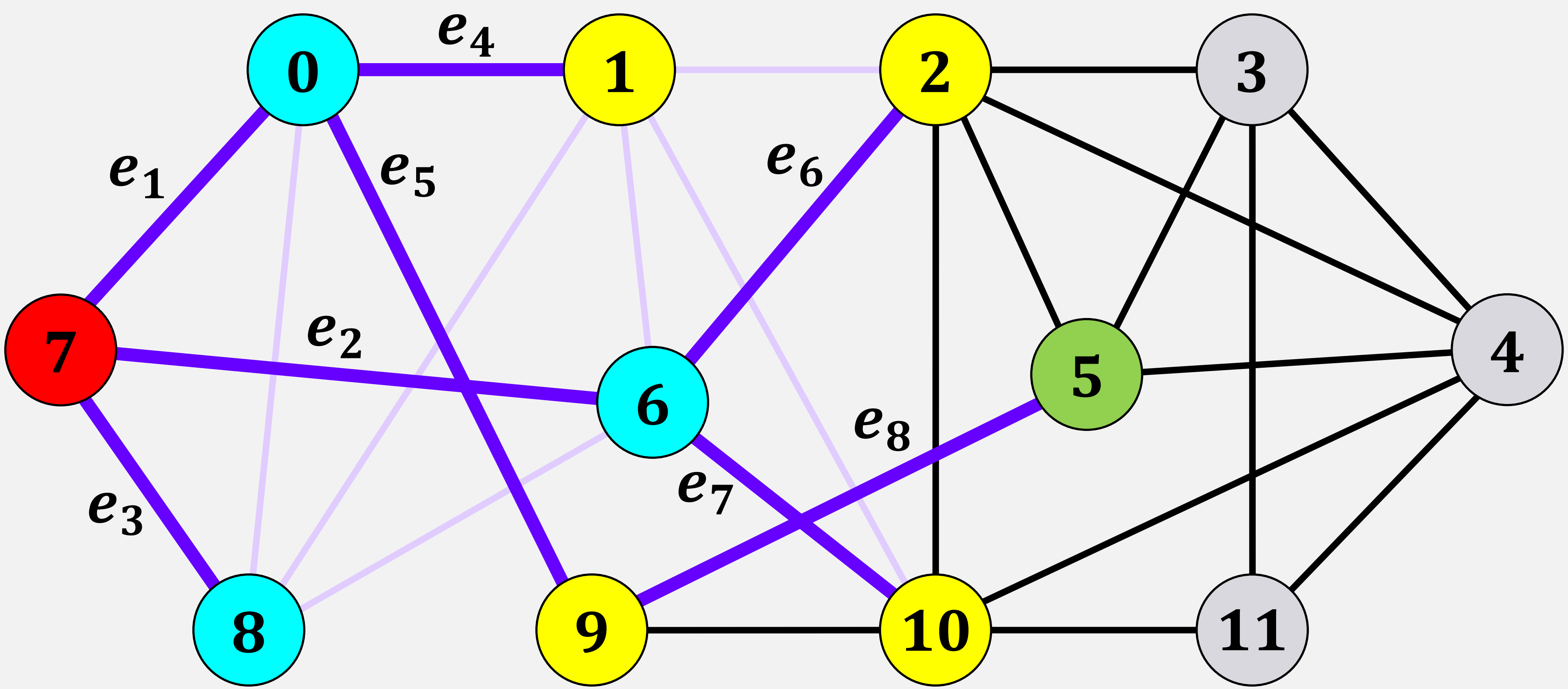
Breadth-First Search (BFS)

traversal → q contains $7, 0, 6, 8, 1,$ **9, 2, 10,**



Breadth-First Search (BFS)

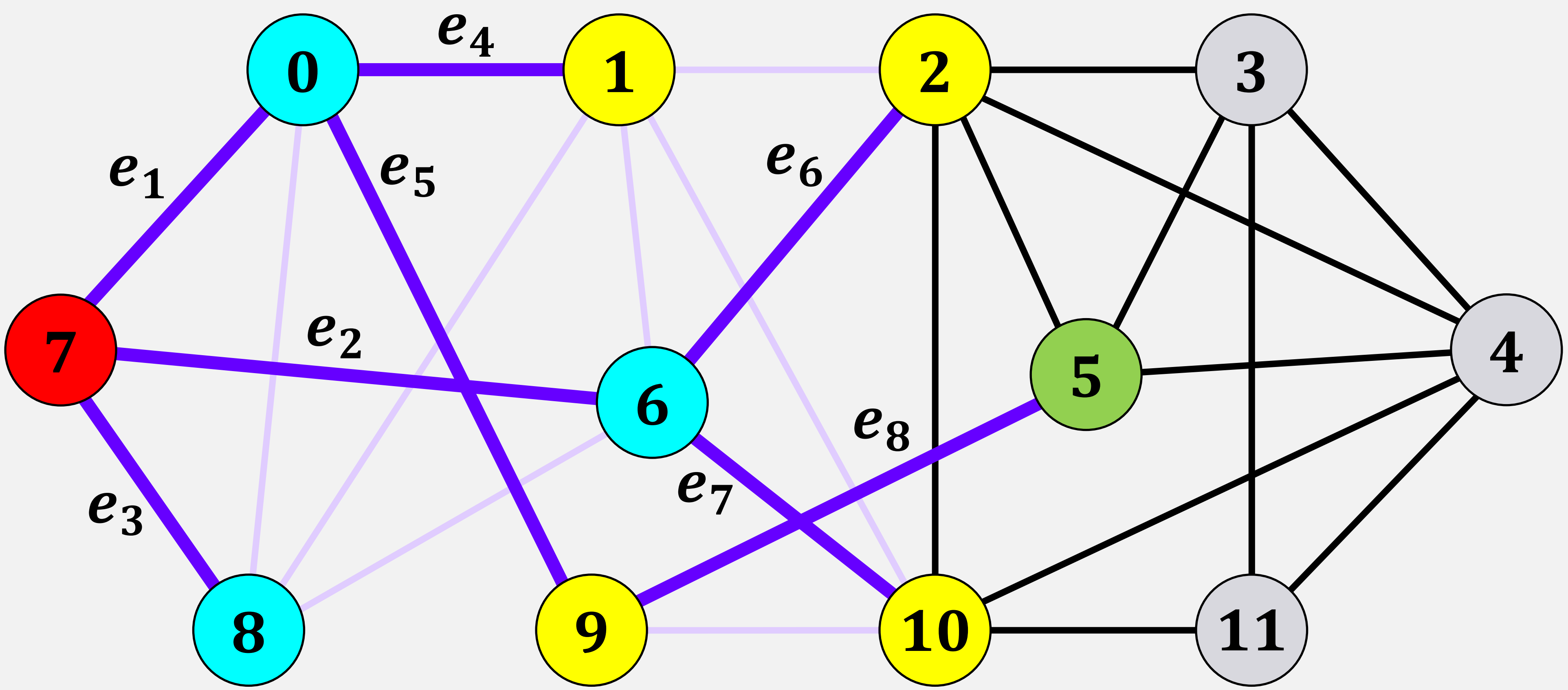
traversal → q contains 7, 0, 6, 8, 1, 9, **2, 10, 5,**



Alina Shaikhet – COMP2406 – Carleton University

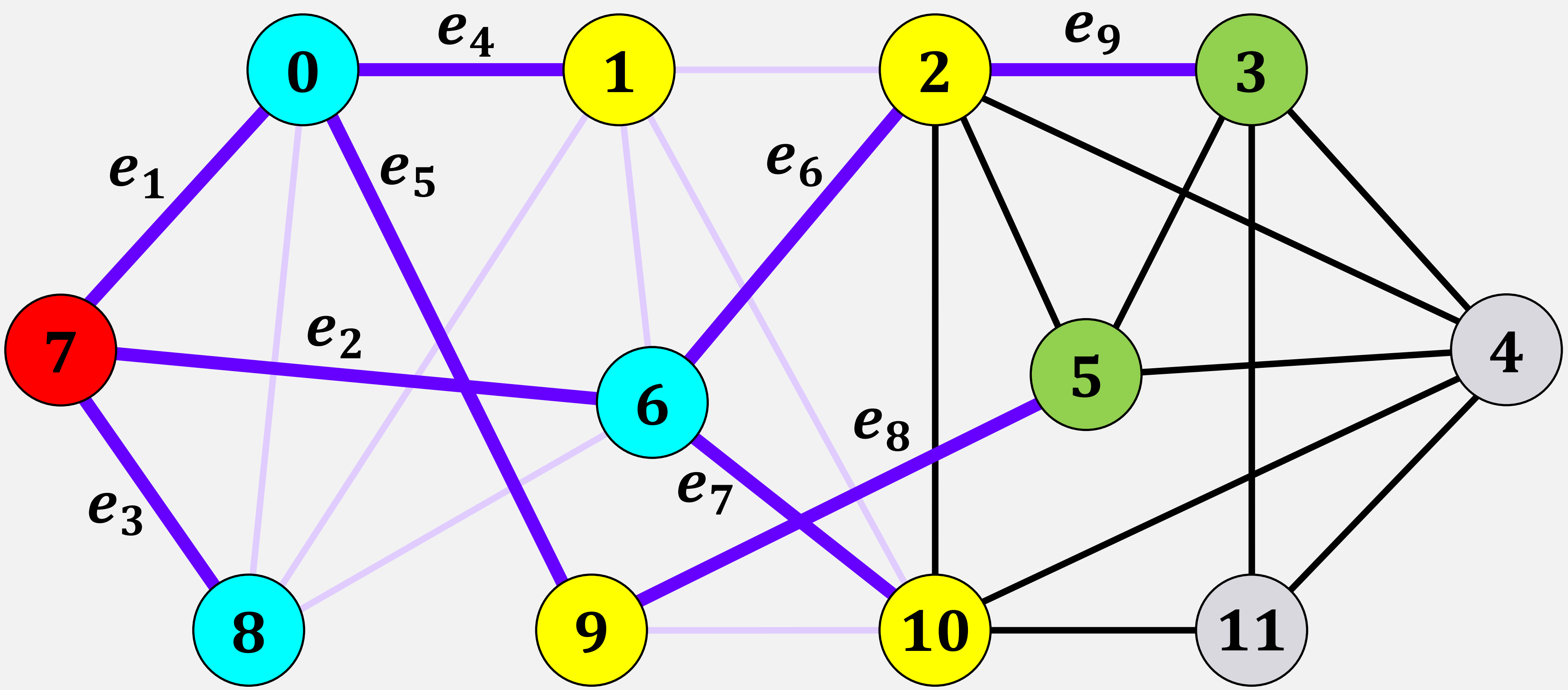
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, 9, **2, 10, 5,**



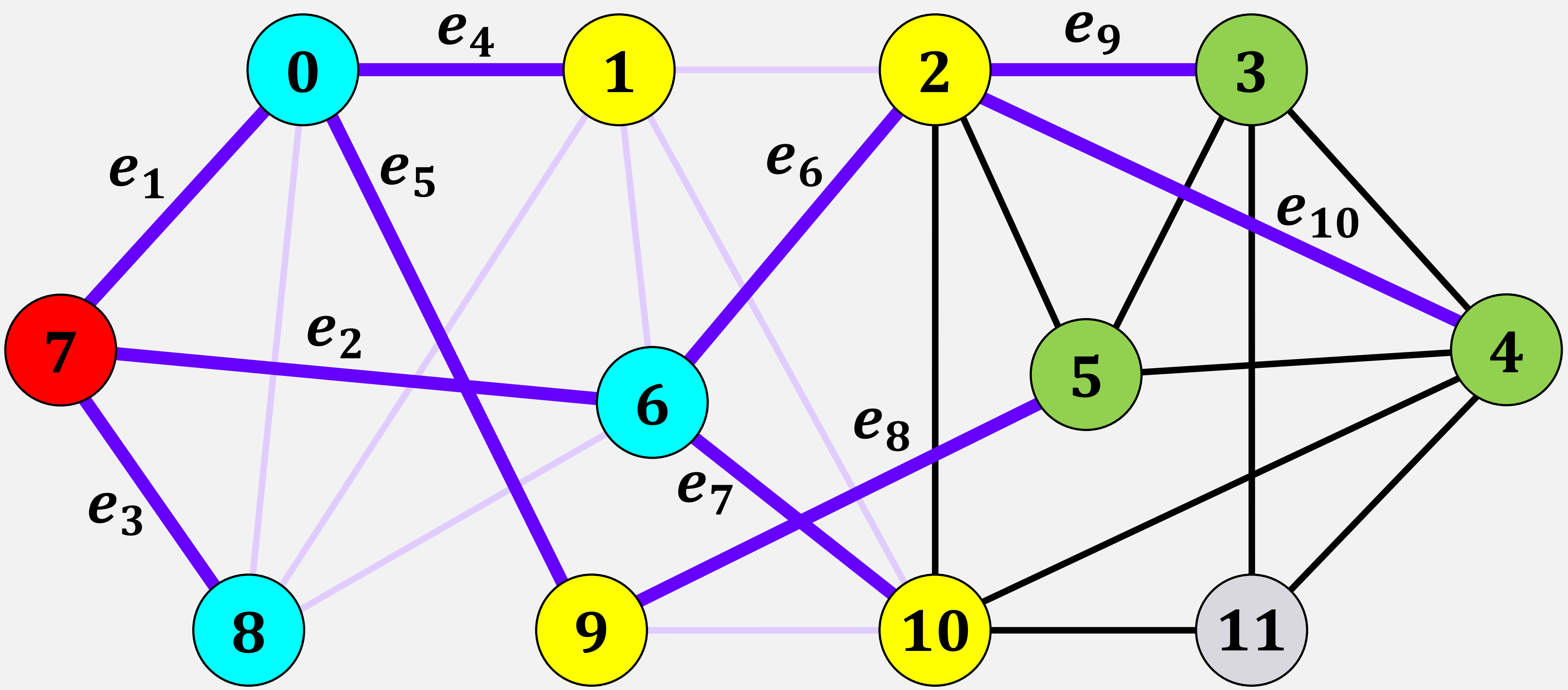
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, 9, 2, **10, 5, 3,**



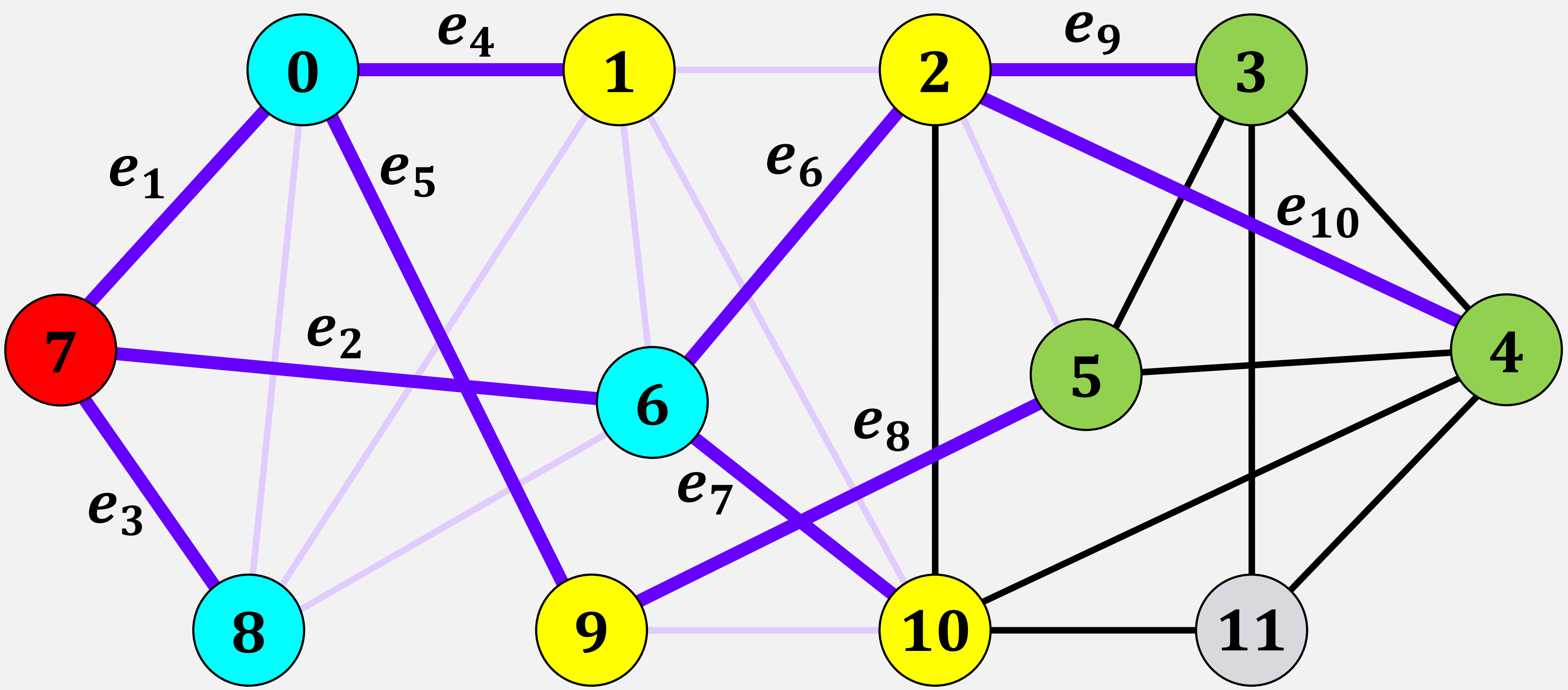
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, 9, 2, **10, 5, 3, 4,**



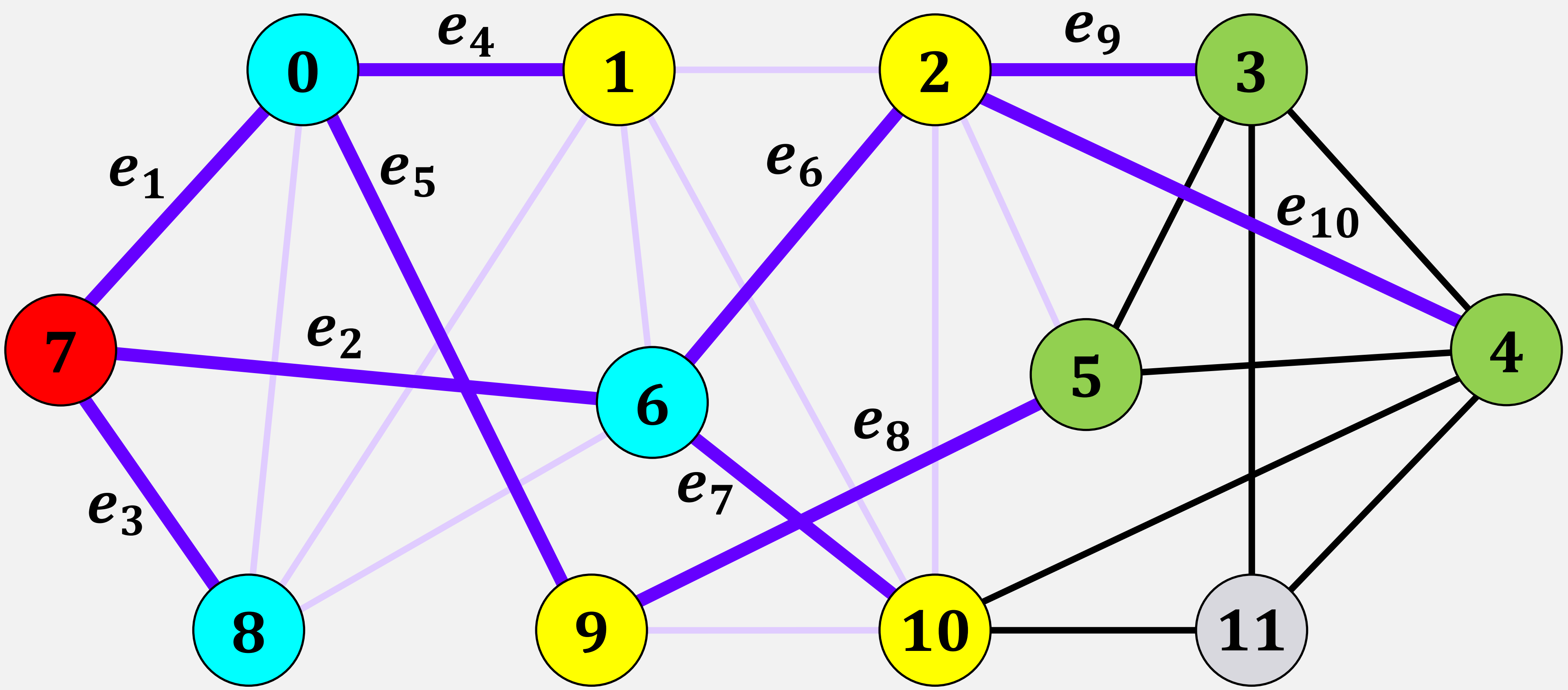
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, 9, 2, **10, 5, 3, 4,**



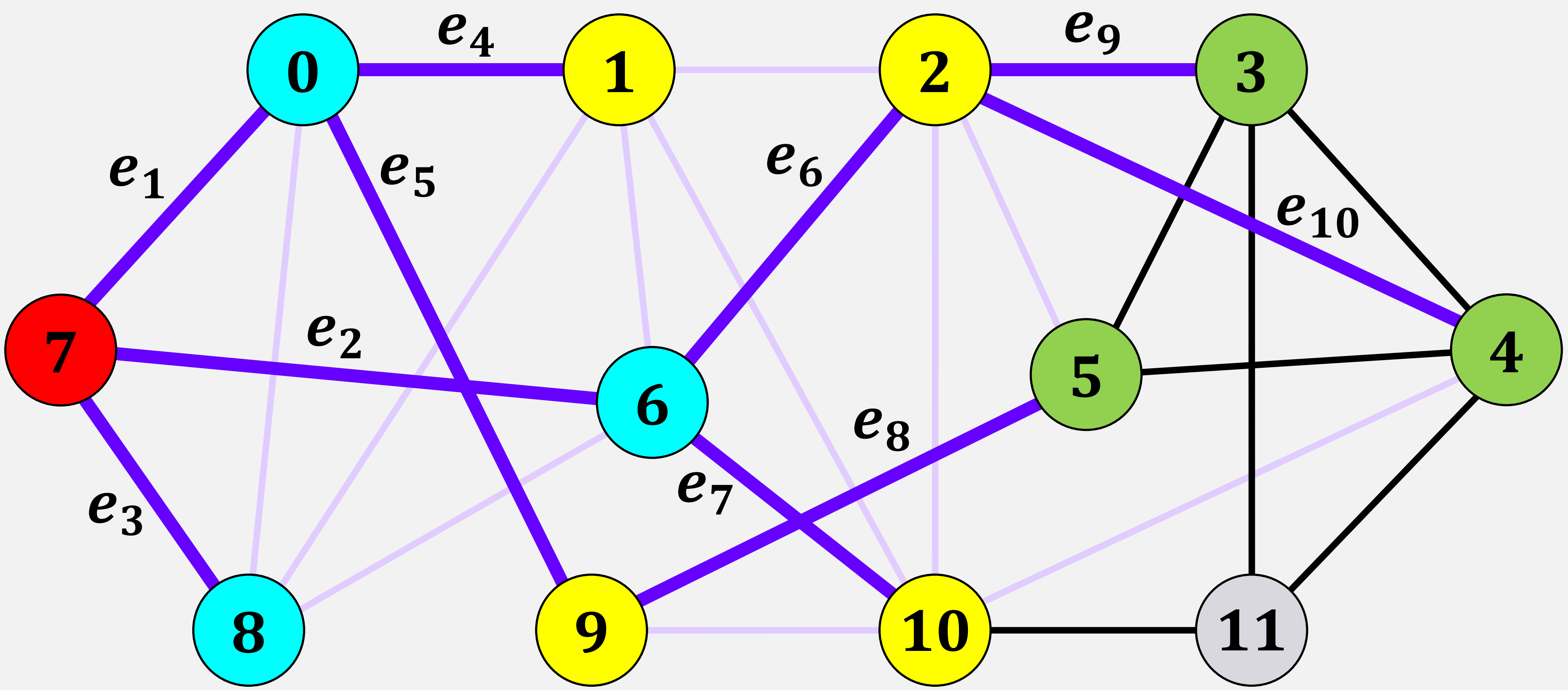
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, 9, 2, **10, 5, 3, 4,**



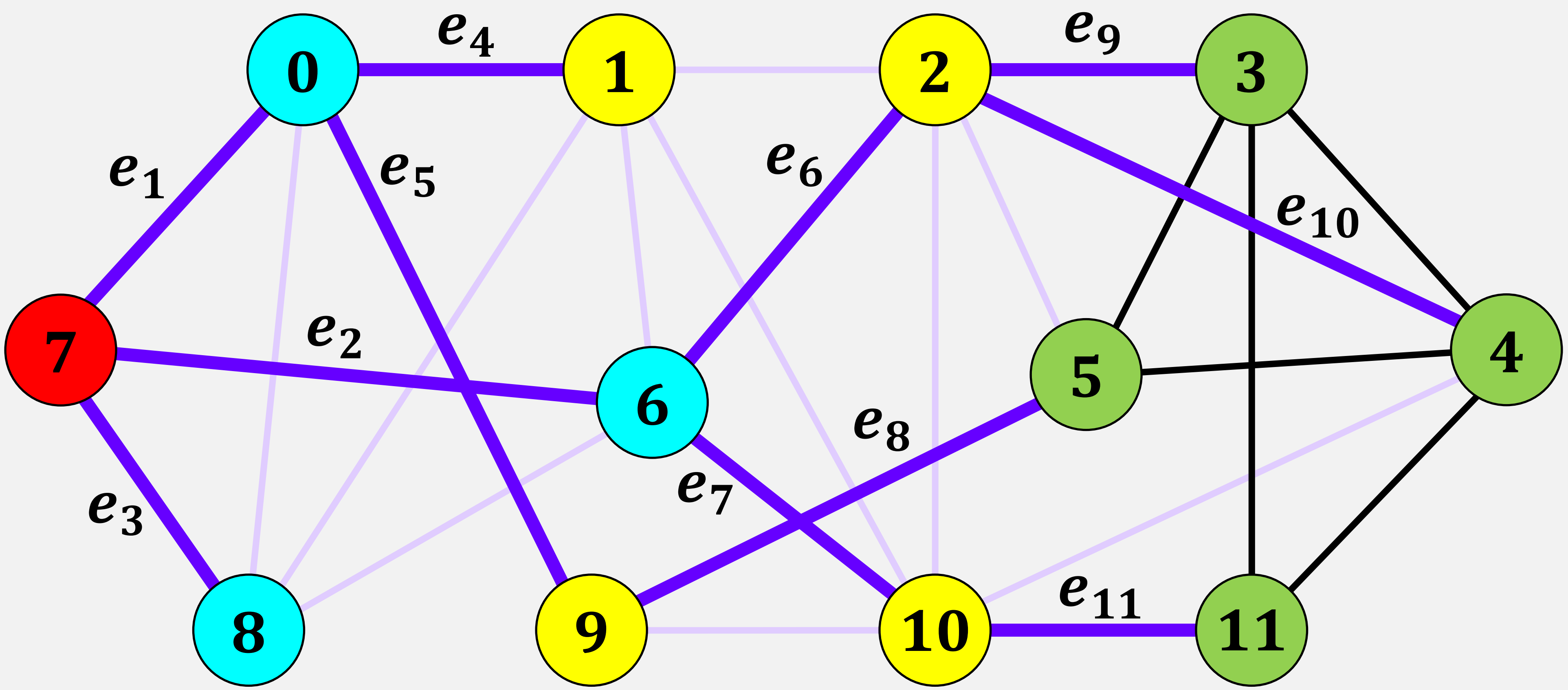
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, 9, 2, 10, **5, 3, 4,**



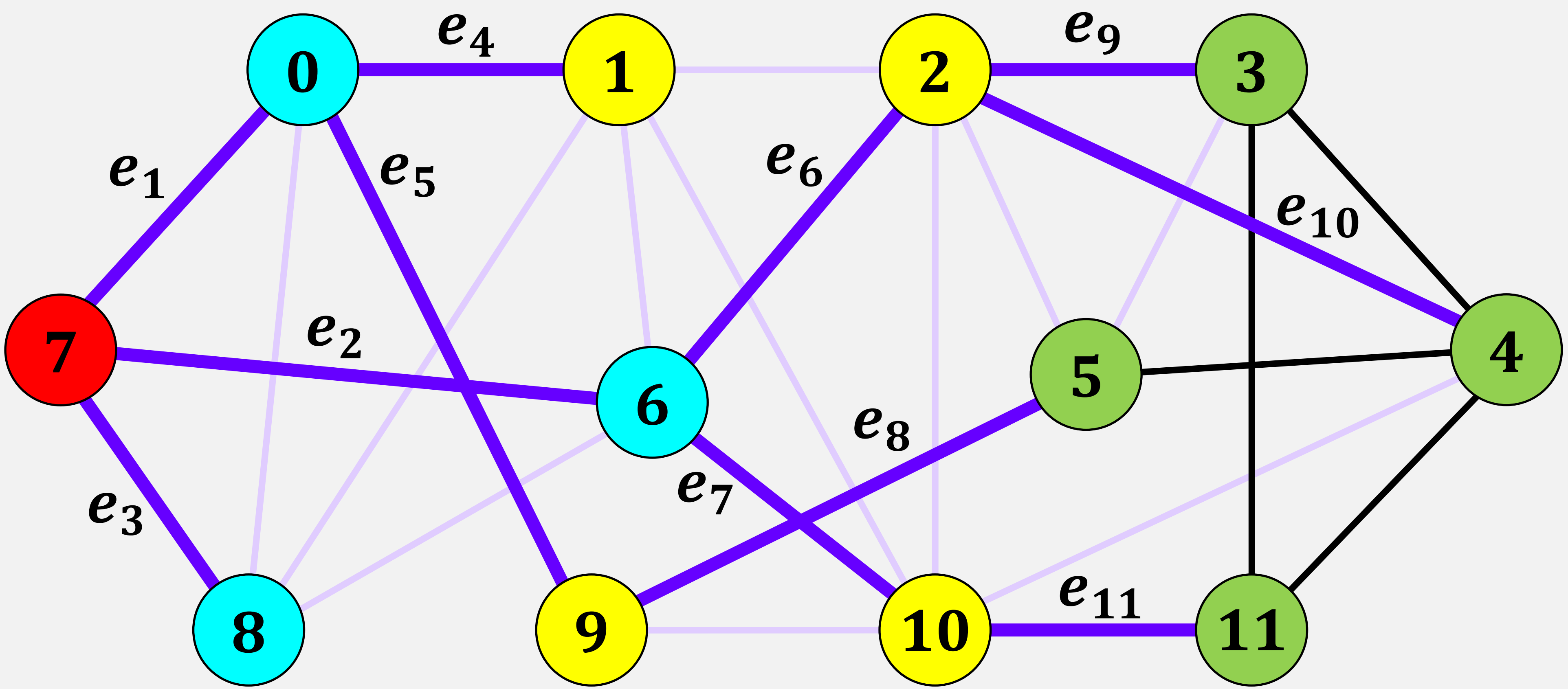
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, 9, 2, 10, **5, 3, 4, 11,**



Breadth-First Search (BFS)

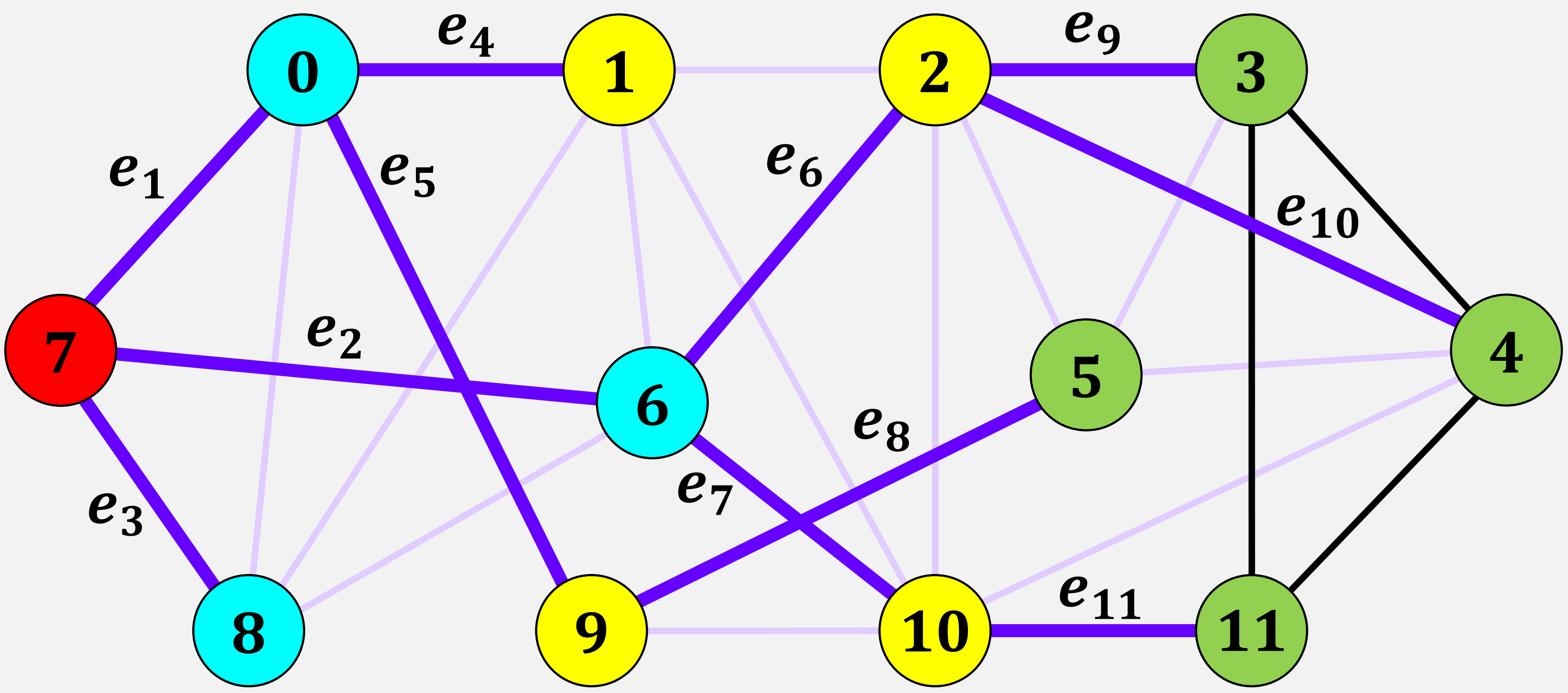
traversal → q contains 7, 0, 6, 8, 1, 9, 2, 10, 5, **3, 4, 11,**



Alina Shaikhet – COMP2406 – Carleton University

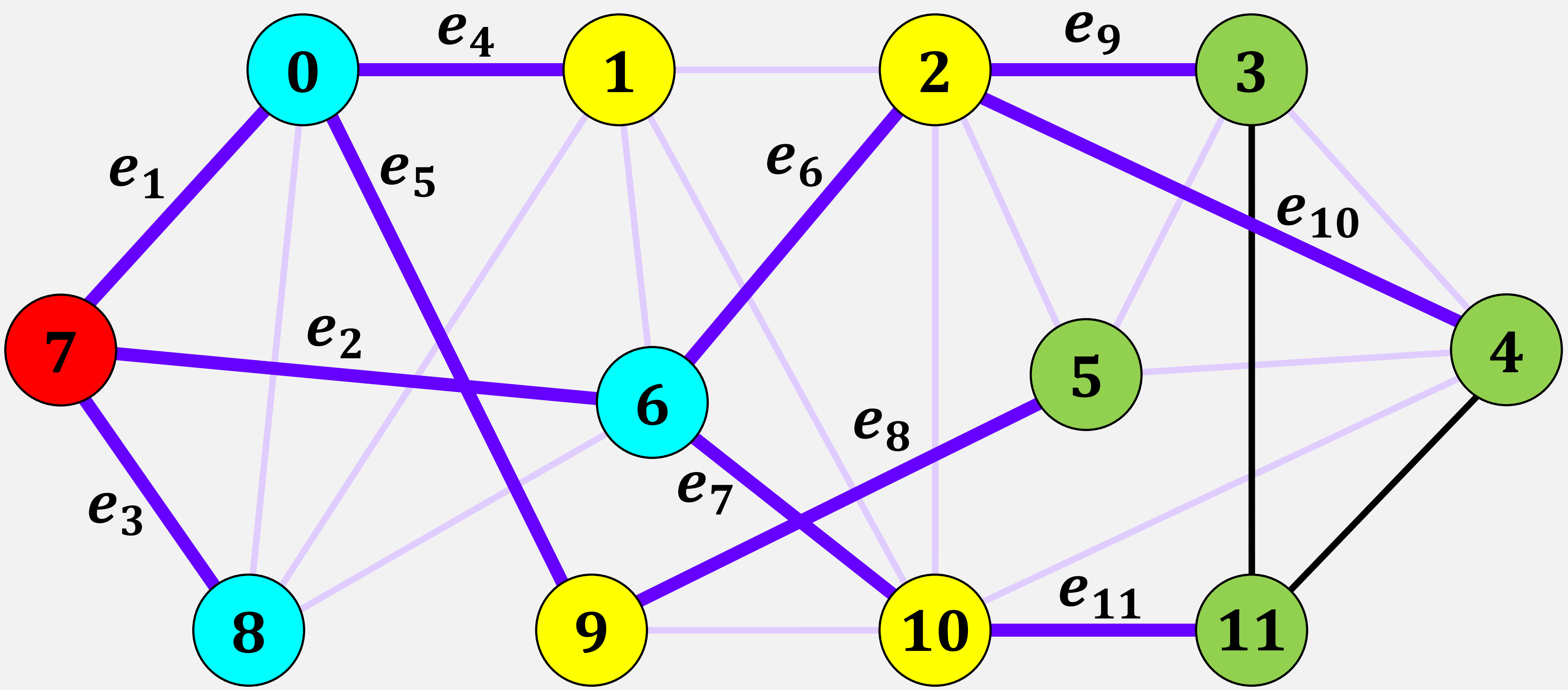
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, 9, 2, 10, 5, **3, 4, 11,**



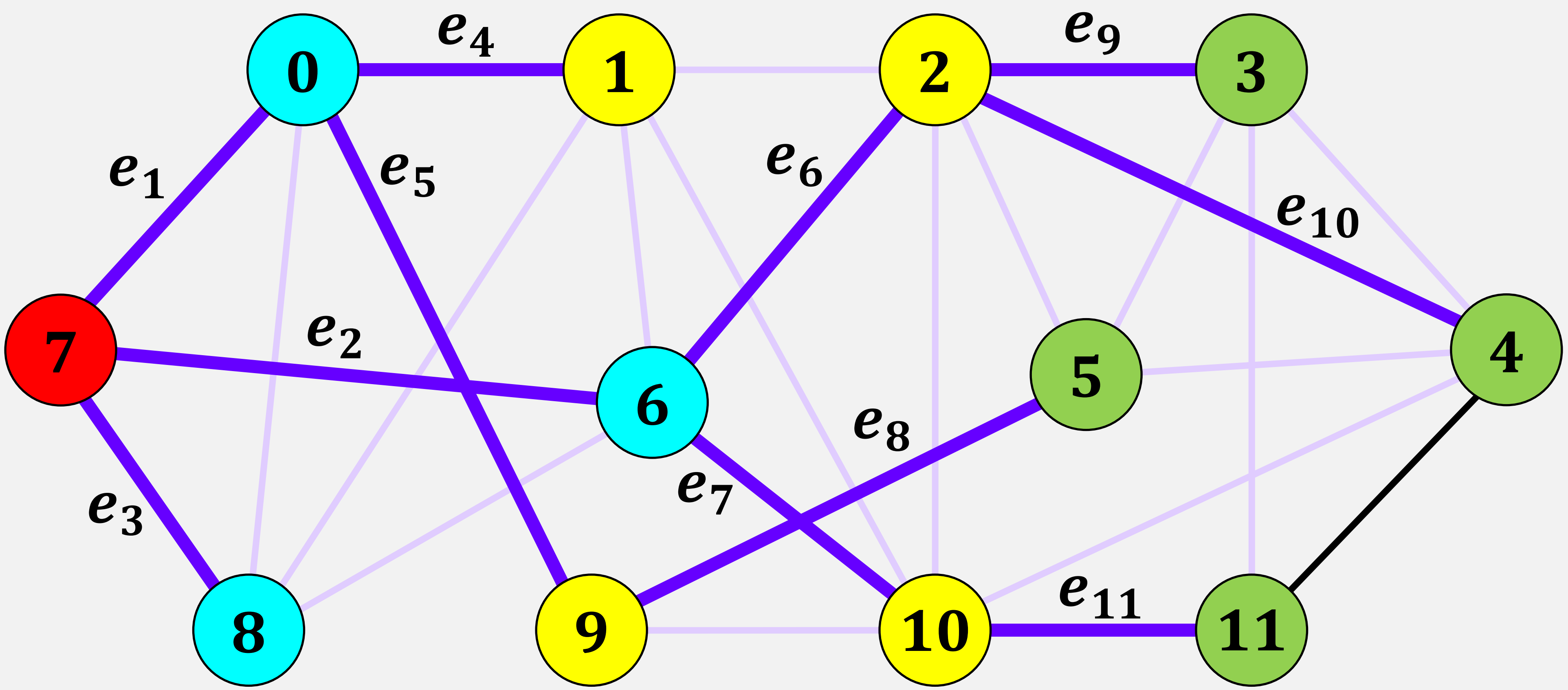
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, 9, 2, 10, 5, 3, **4, 11,**



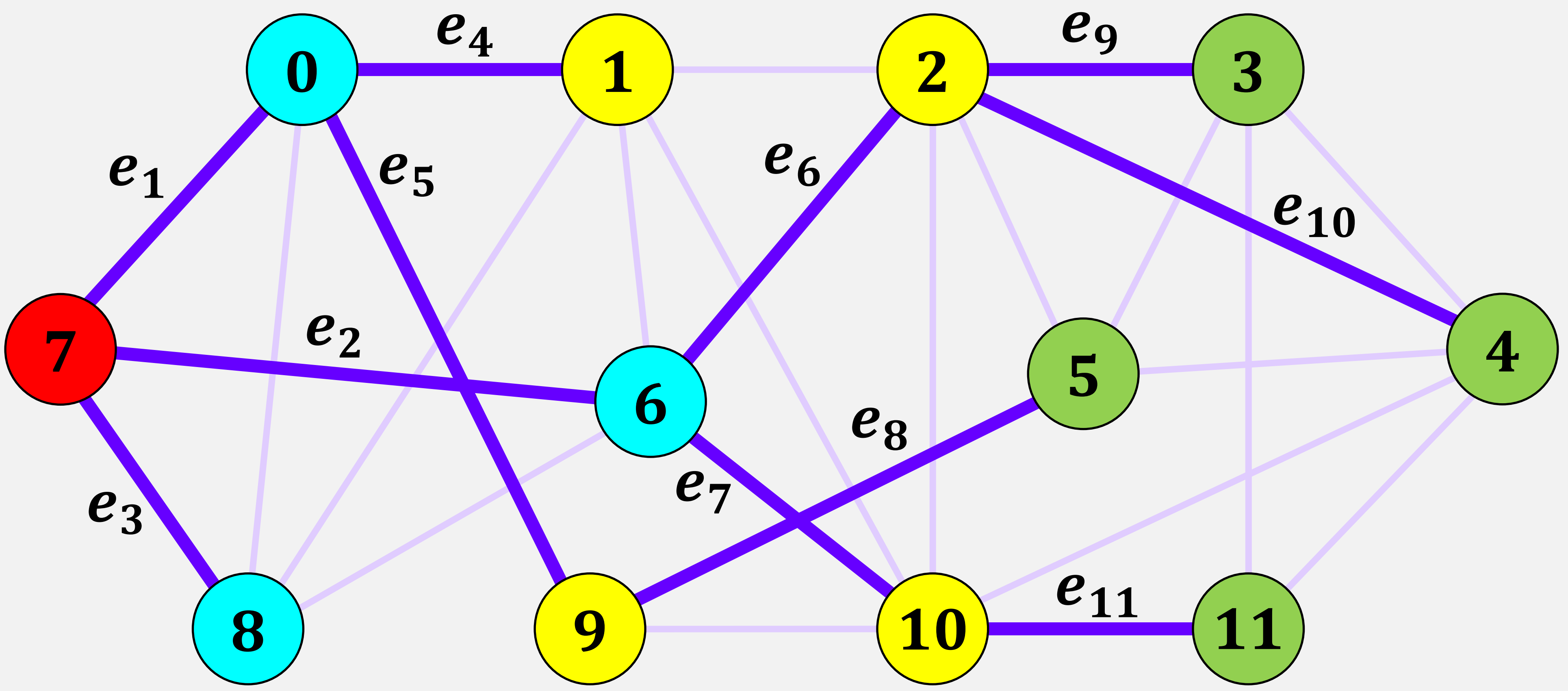
Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, 9, 2, 10, 5, 3, **4, 11,**

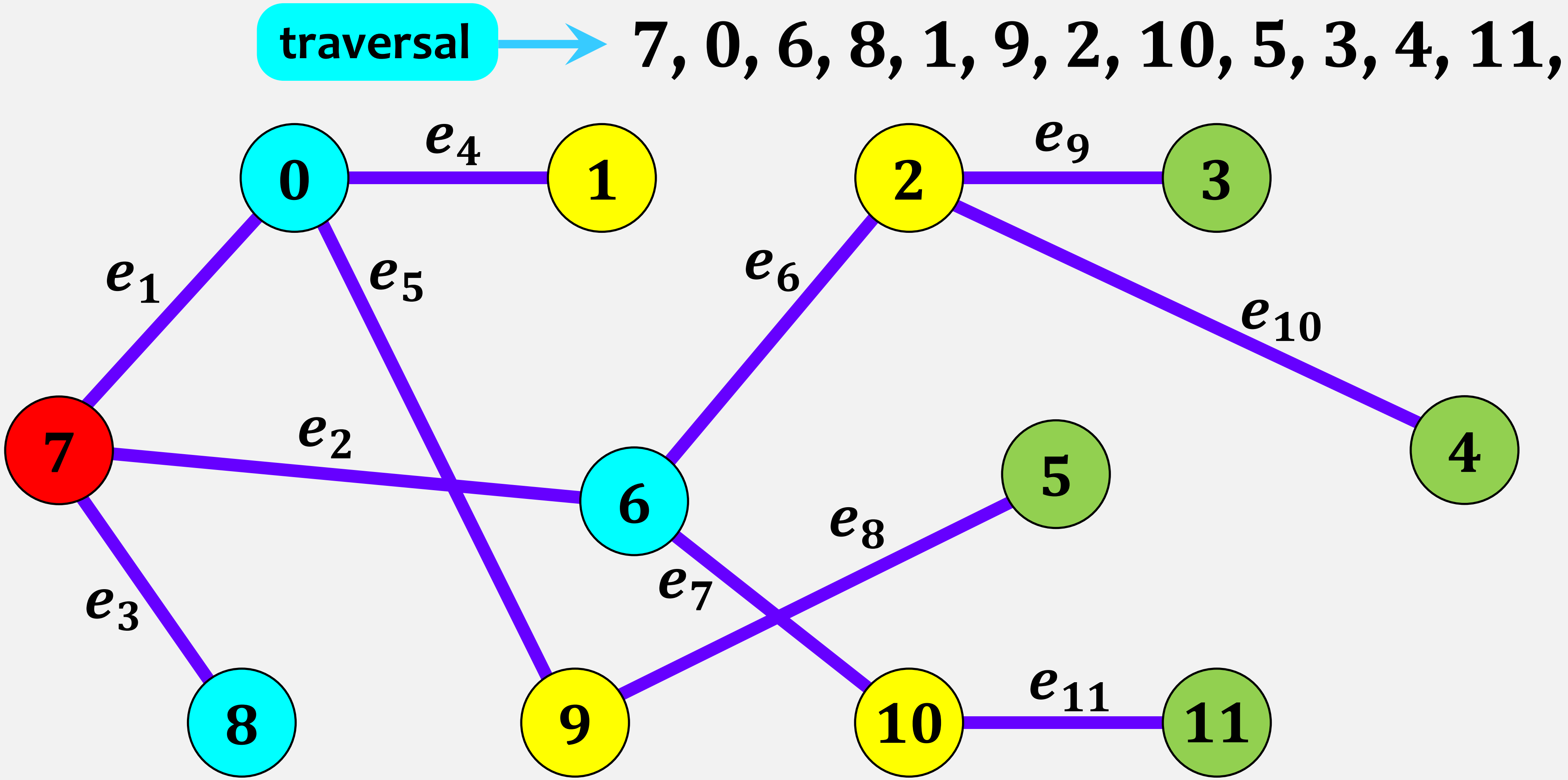


Breadth-First Search (BFS)

traversal → q contains 7, 0, 6, 8, 1, 9, 2, 10, 5, 3, 4, **11**,



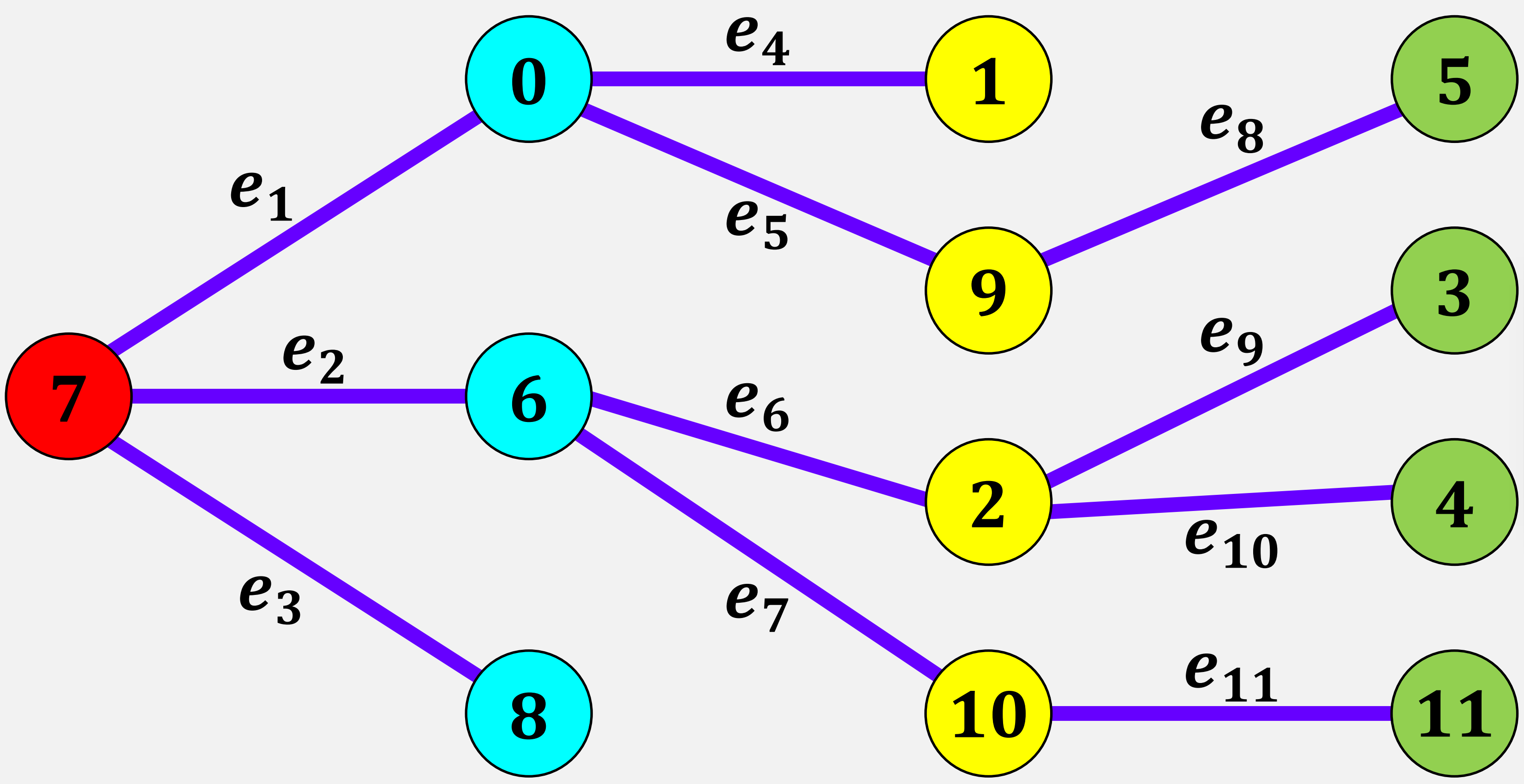
Breadth-First Search (BFS)



Spanning subtree generated by BFS

Breadth-First Search (BFS)

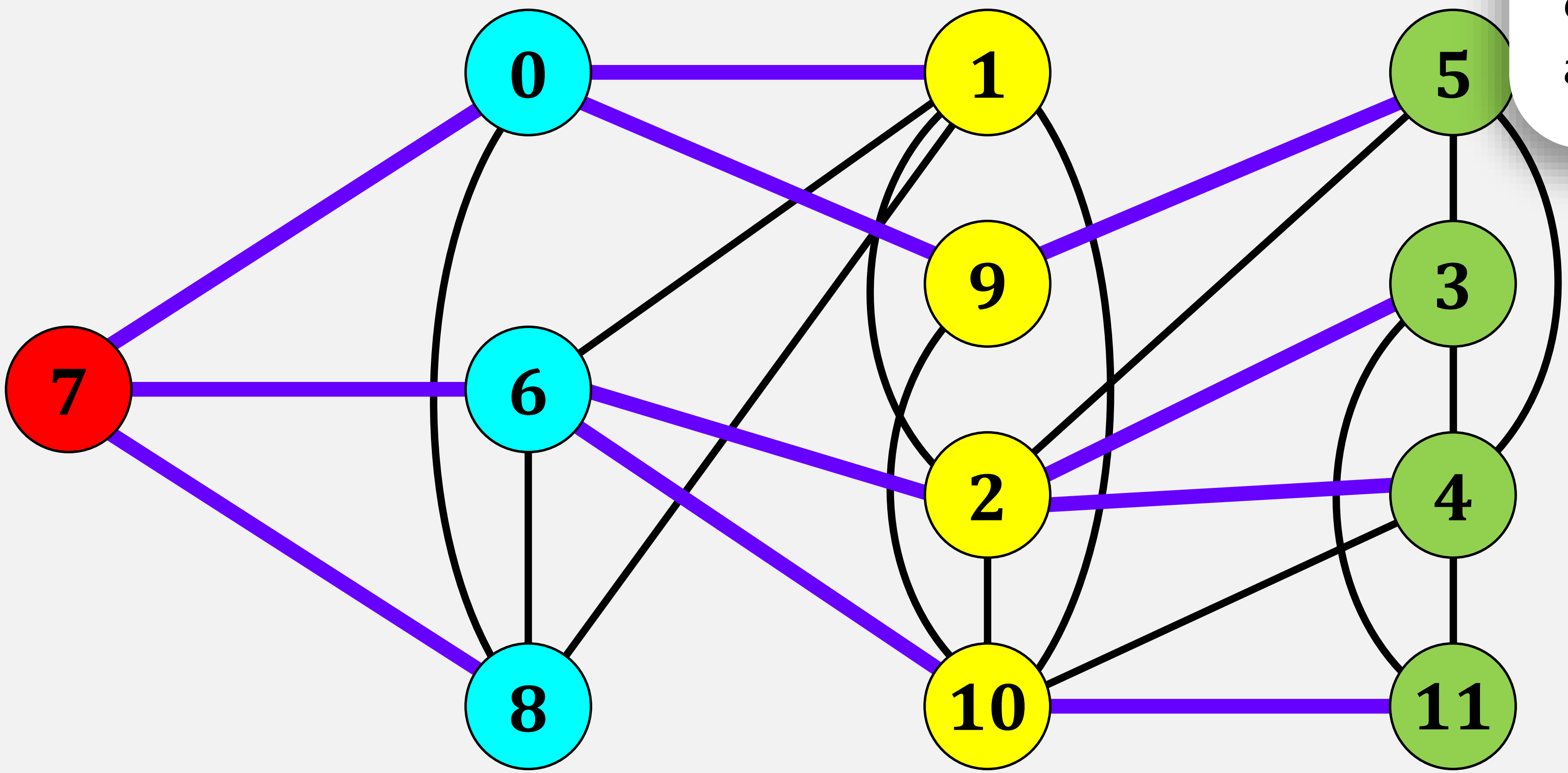
traversal → **7, 0, 6, 8, 1, 9, 2, 10, 5, 3, 4, 11,**



We've found the **shortest path** from **7** to every other node in $\boldsymbol{G}$

Spanning subtree generated by BFS

Breadth-First Search (BFS)



For each edge in G that is not in the BFS tree:
The depth of its endpoints differs by at most 1.

Breadth-First Search (BFS)

Although the Breadth-First Search might appear to be more complicated than DFS (and, in many ways, it is)...

...it produces the shortest path (in terms of number of edges traversed) to any vertex

Given a vertex in the graph a BFS will find a shortest path from that vertex to every other vertex in the graph.

Theorem 12.3

When given as input a Graph, $\boldsymbol{g}$, that is implemented using the **AdjacencyLists** data structure, the **bfs**$(\boldsymbol{g}, \boldsymbol{r})$ algorithm runs in $\boldsymbol{O}(\boldsymbol{n} + \boldsymbol{m})$ time.