


Overview

- Goal: Determine the shortest path between an initial and goal node in a graph
 - General Information:
 - A* is an extension of Dijkstra's algorithm and employs an additional heuristic.
 - Typical implementations employ a priority queue (i.e., open set)
 - If the heuristic is monotone, nodes only have to be active once → like Dijkstra
- 
- $$h(n_1) \leq d(n_1, n_2) + h(n_2)$$

A* Heuristic

- A* contains a cost function $f(n) = g(n) + h(n)$, where $h(n)$ is the heuristic.
 - When $f(n) = g(n)$, it becomes truly greedy
 - Search considers what it believes the best path from current to goal → Dijkstra's algorithm
- If $h(n)$ is admissible (always be less than or equal to the cost to move to the goal or the cost calculated during Dijkstra's algorithm $g(n)$), then A* is guaranteed to find the shortest path.
- If we make $h(n)$ too small, we will slow down the speed of finding the shortest path.

Possible A* Heuristic

- Shortest number of nodes to goal – we will use this for our examples
- Distance from the current node to the goal node
 - Manhattan Distance (the 1-norm)
 - Euclidean Distance (the 2-norm)
 - Euclidean Distance Squared
- Problem-specific

- $h(n)$ is the heuristic cost function, which returns the estimated cost of a path from n to q_{goal} .
 - $f(n) = g(n) + h(n)$ is the estimated cost of shortest path from q_{start} to q_{goal} .
- The algorithm can be found in algorithm 24.

H.2.2 Discussion: Completeness, Efficiency, and Optimality

Here is an informal proof of completeness for A^* . A^* generates a search tree by definition, has no cycles. Furthermore, there are a finite number of acyclic paths in the tree, assuming a bounded world. Since A^* uses a tree, it only considers paths. Since the number of acyclic paths is finite, the most work that can

Algorithm 24 A^* Algorithm

Input: A graph

Output: A path between start and goal nodes

- 1: **repeat**
 - 2: Pick n_{best} from O such that $f(n_{\text{best}}) \leq f(n), \forall n \in O$.
 - 3: Remove n_{best} from O and add to C .
 - 4: If $n_{\text{best}} = q_{\text{goal}}$, EXIT.
 - 5: Expand n_{best} : for all $x \in \text{Star}(n_{\text{best}})$ that are not in C .
 - 6: **if** $x \notin O$ **then**
 - 7: add x to O .
 - 8: **else if** $g(n_{\text{best}}) + c(n_{\text{best}}, x) < g(x)$ **then**
 - 9: update x 's backpointer to point to n_{best}
 - 10: **end if**
 - 11: **until** O is empty
-

A* Algorithm

1. Initialize the distance to n_{init} as 0 and the heuristic as its actual calculated value. Initialize temporary distances and heuristics for the remaining nodes to ∞ . Calculate the cost for each node by adding the distance and heuristic. Additionally, initialize the closest neighbor for each node as NULL.
2. Create an open set containing n_{init} and an empty visited set.
3. Set the node with the smallest cost in the open set to active.
4. If the active node is n_g , return the cost and the shortest path.
5. For each neighbor of the active node not in the visited set, calculate the distance and the heuristic cost. If the new distance is smaller than the temporary distance, update the cost of the node and set the closest neighbor to the active node. If the neighbor is not in the open set, add it.
6. Once all neighbors have been visited, add the active node to the visited set.
7. If n_g is not the active node, repeat Steps 3 through 6.

```

function Astar( $G, n_{init}, n_{goal}$ )
   $O = \{n_{init}\}$  % open set, a priority queue
   $C = \emptyset$  % closed set
  for each node  $n$  in graph  $G$  do:
     $prev[n] = NULL$  (set backpointer set as null set)
     $f[n] = \infty$ 
     $g[n] = \infty$ 
  end for
   $g[n_{init}] = 0$ 
   $f[n_{init}] = heuristic\_cost\_estimate(n_{init}, n_{goal})$ 
  while  $O$  is not empty do:
    Pick  $n_{best}$  from  $O$  such that  $f(n_{best}) \leq f(n), \forall n \in O$ .
    Remove  $n_{best}$  from  $O$ 
    Add  $n_{best}$  to  $C$ .
    if  $n_{best} = n_{goal}$  then:
      exit
    end if
    for each neighbor,  $x$ , of  $n_{best}$  do:
      if  $x \in C$  then:
        continue
      end if
       $g\_temp \doteq g[n_{best}] + dist\_between(n_{best}, x)$ 
      if  $x \notin O$  then:
         $openset.add(x)$ : add  $x$  to open set  $O$ 
      else if  $g\_temp \geq g[x]$  then:
        continue
      end if
       $prev[x] \doteq n_{best}$ 
       $g[x] \doteq g\_temp$ 
       $f[x] \doteq g[x] + heuristic\_cost\_estimate(x, n_{goal})$ 
    end for
  end while

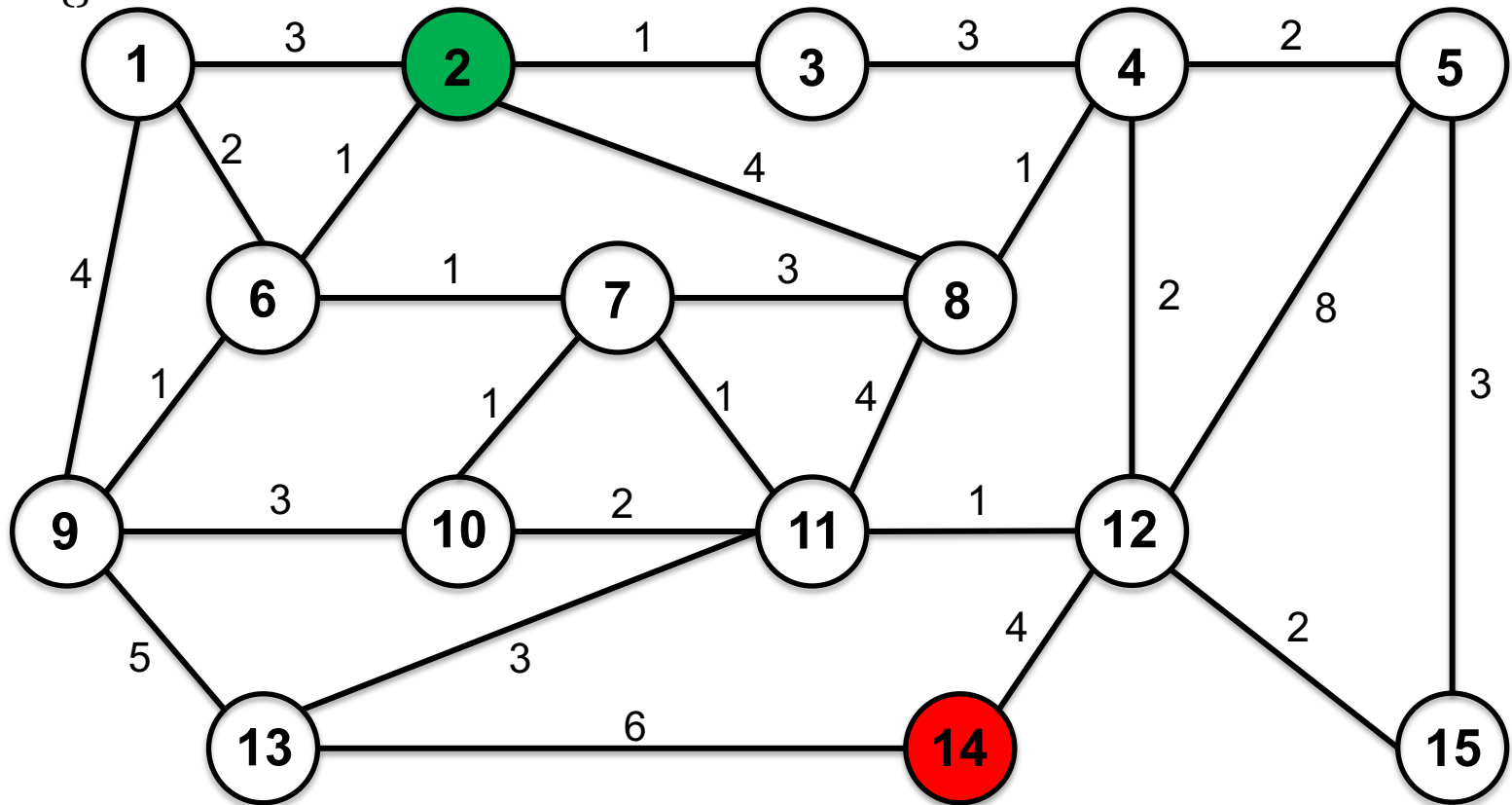
```

Then using $prev$ and others, reconstruct the shortest path.

Example (Undirected Graph)

$openSet = \{2\}$

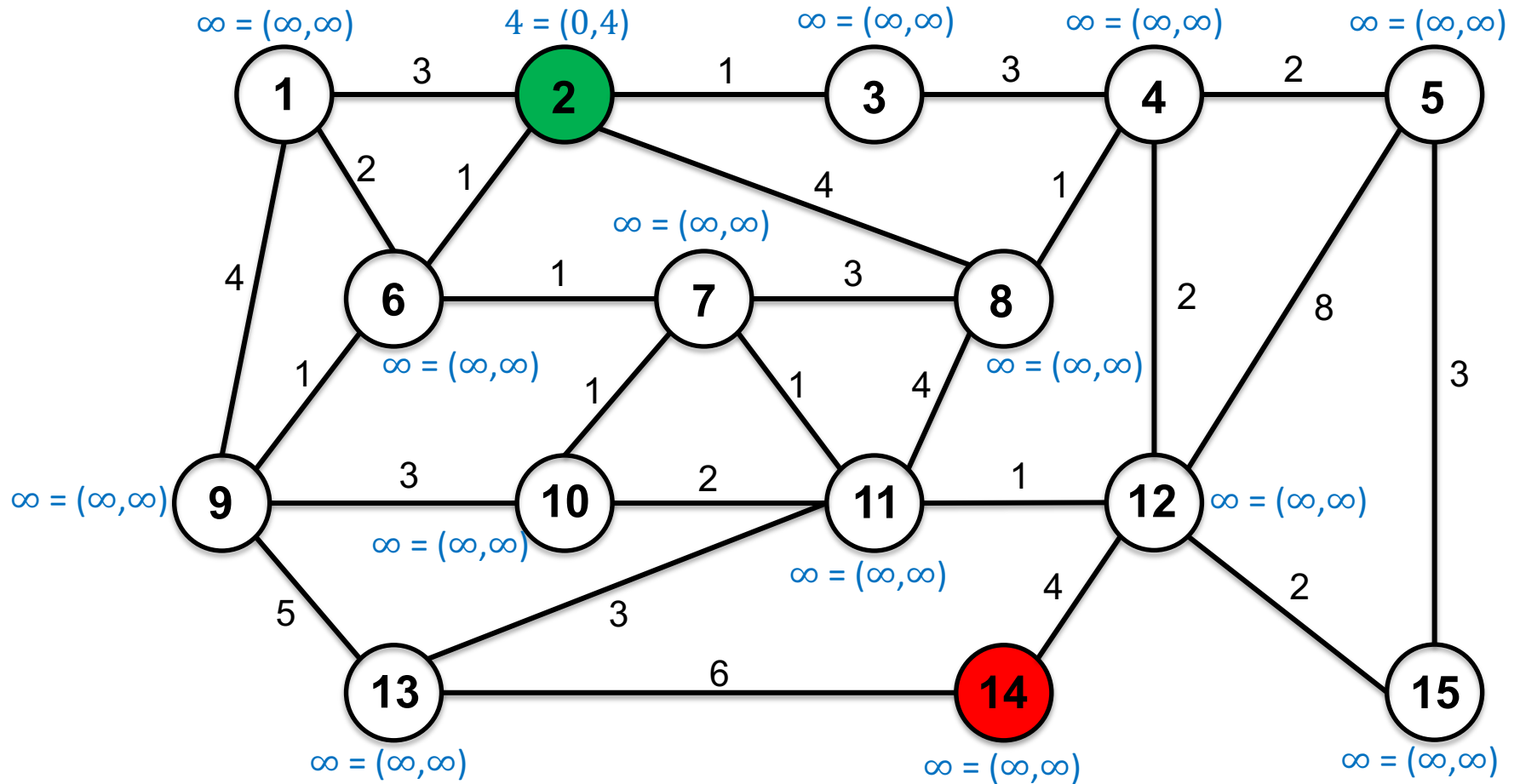
$visited = \{\}$



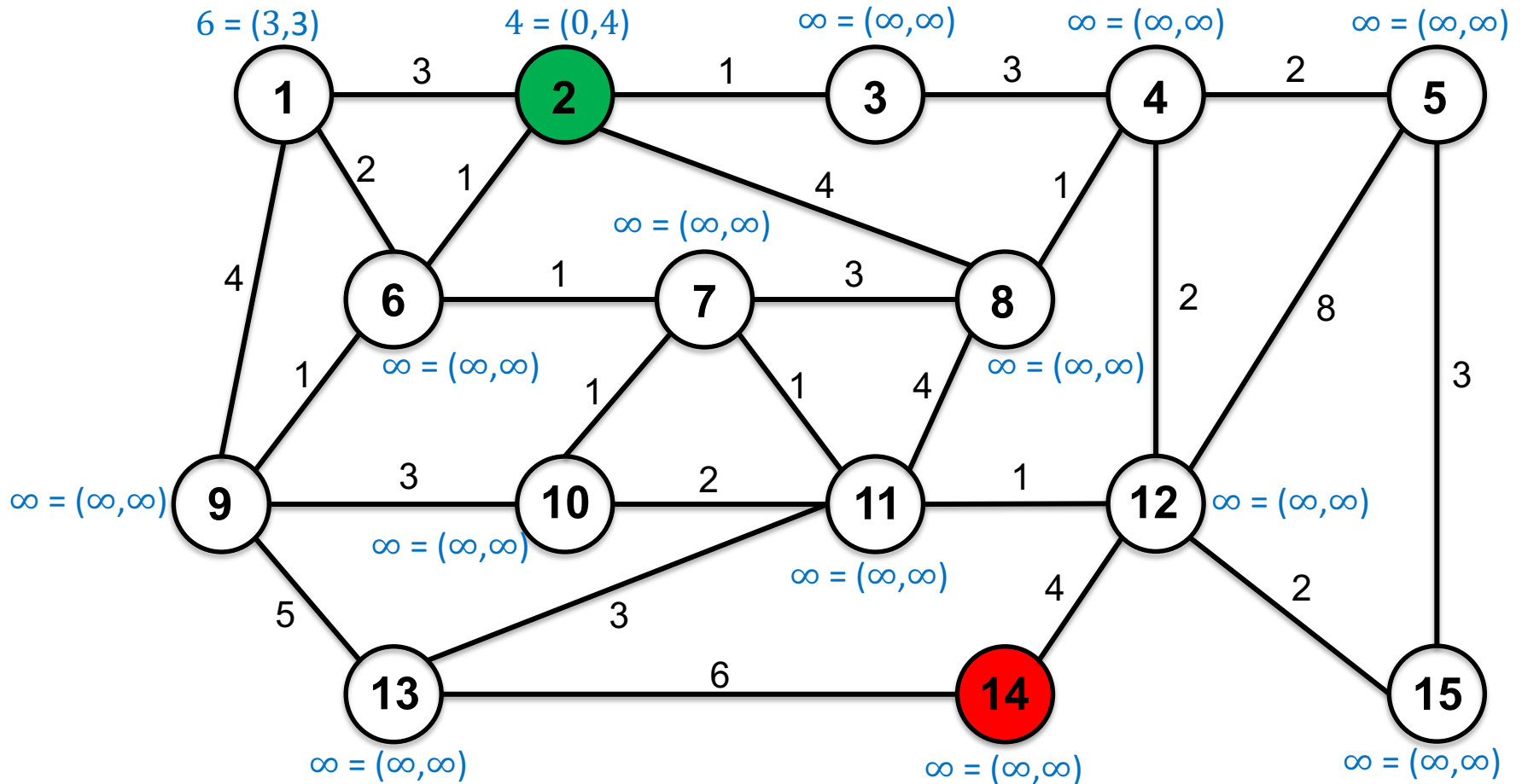
Example (Undirected Graph)

$openSet = \{2\}$

$visited = \{\}$



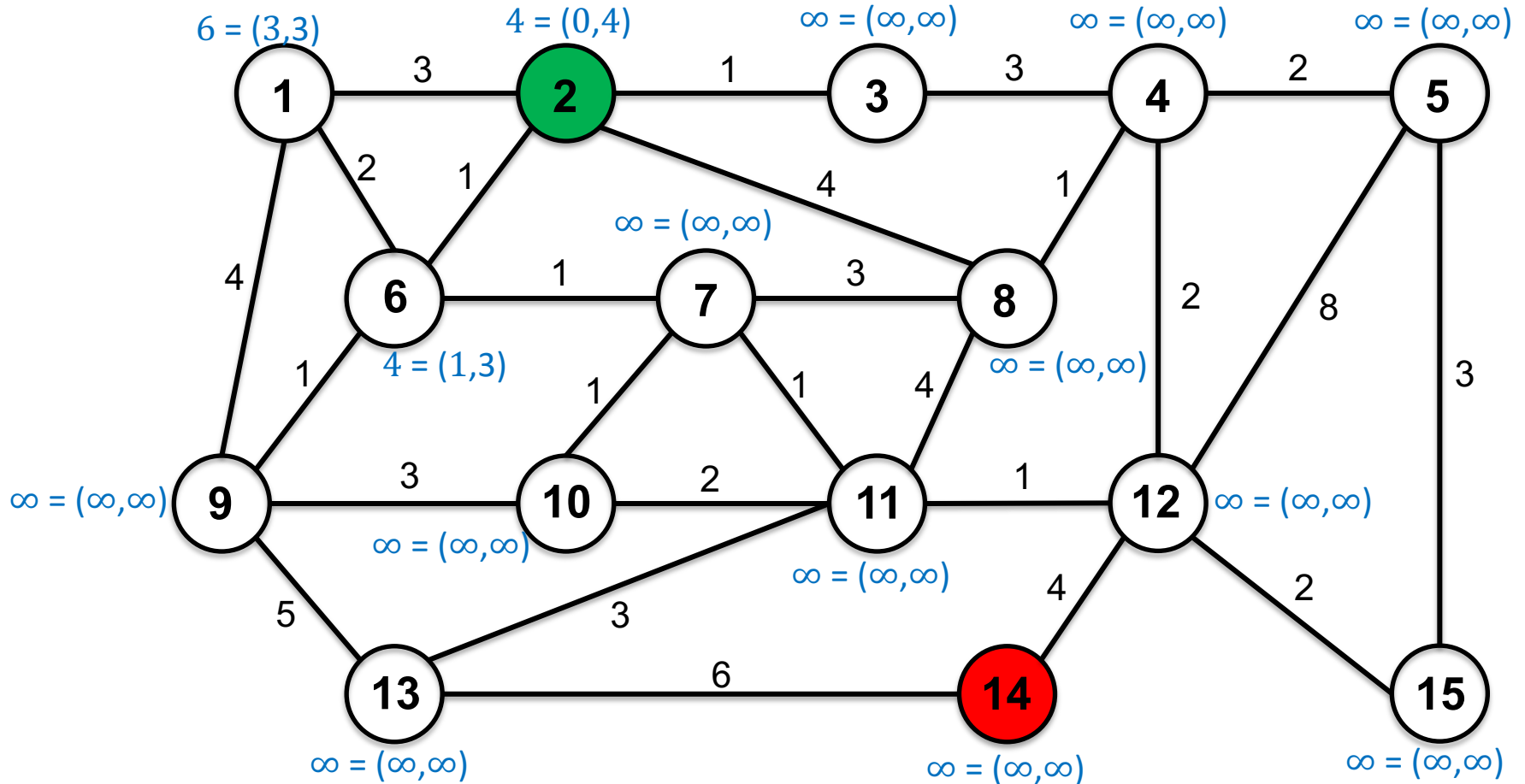
Example (Undirected Graph)

$$openSet = \{1\}$$
$$visited = \{2\}$$


Example (Undirected Graph)

$openSet = \{1,6\}$

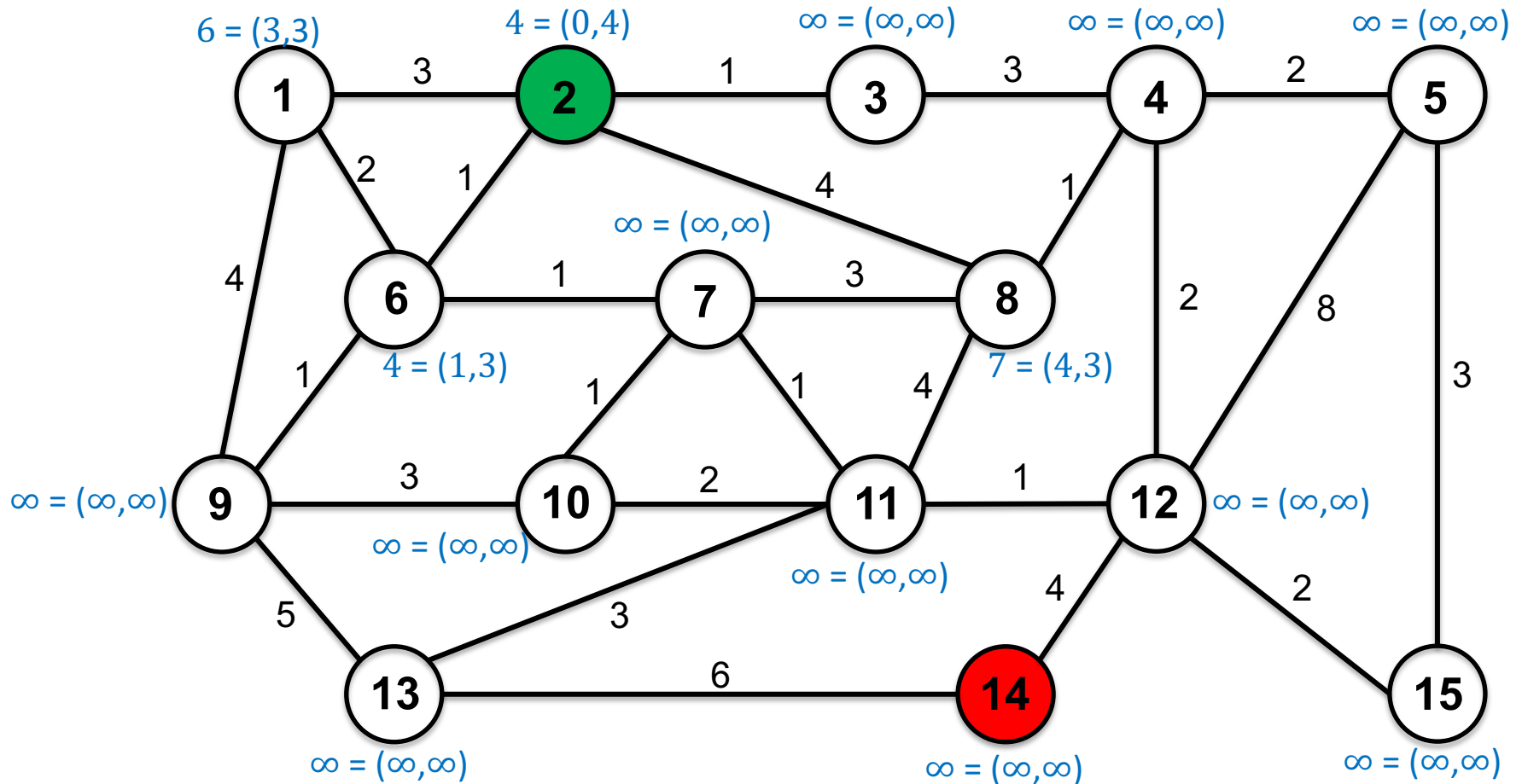
$visited = \{2\}$



Example (Undirected Graph)

$openSet = \{1,6,8\}$

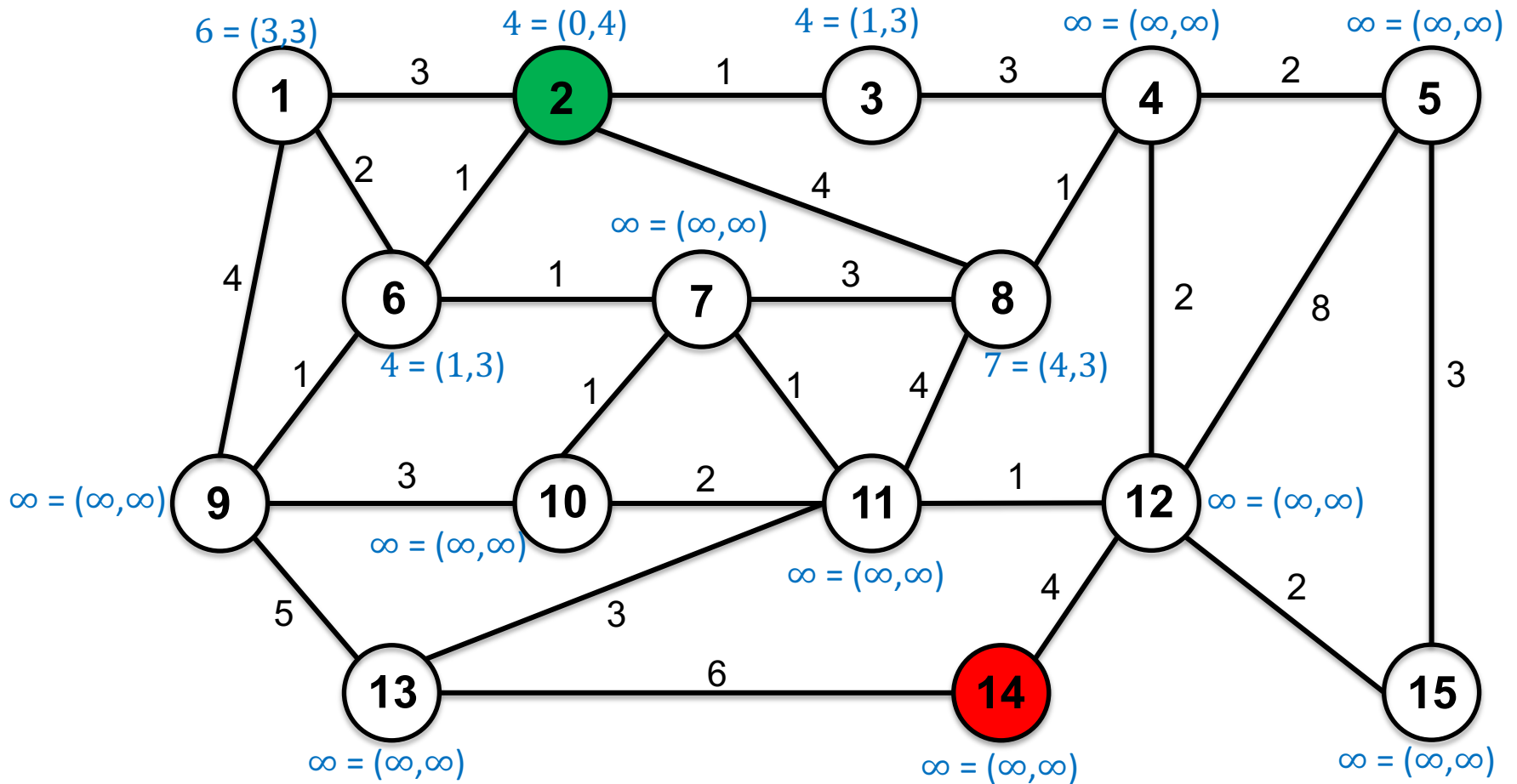
$visited = \{2\}$



Example (Undirected Graph)

$openSet = \{1, 6, 8, 3\}$

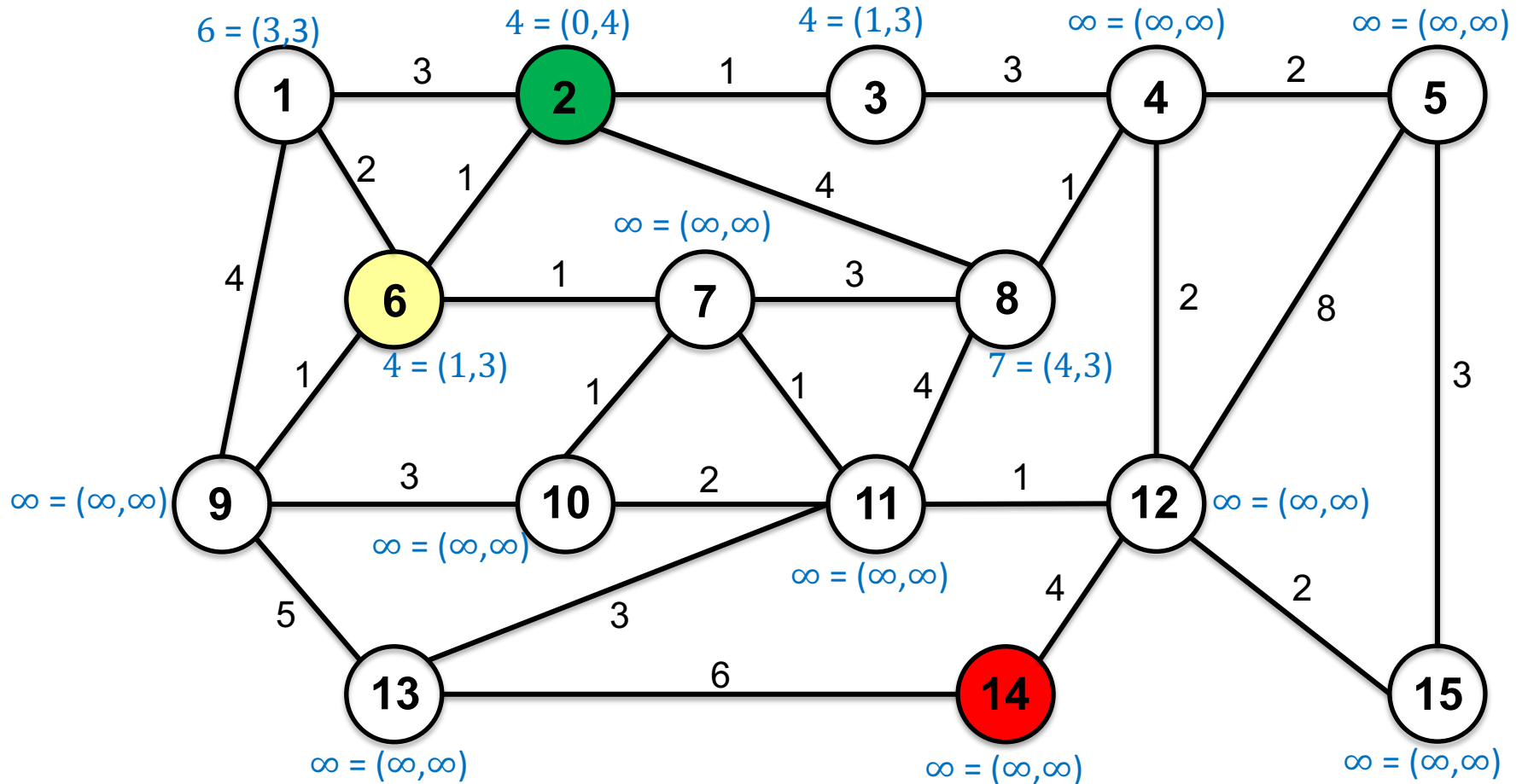
$visited = \{2\}$



Example (Undirected Graph)

$openSet = \{1, 8, 3\}$

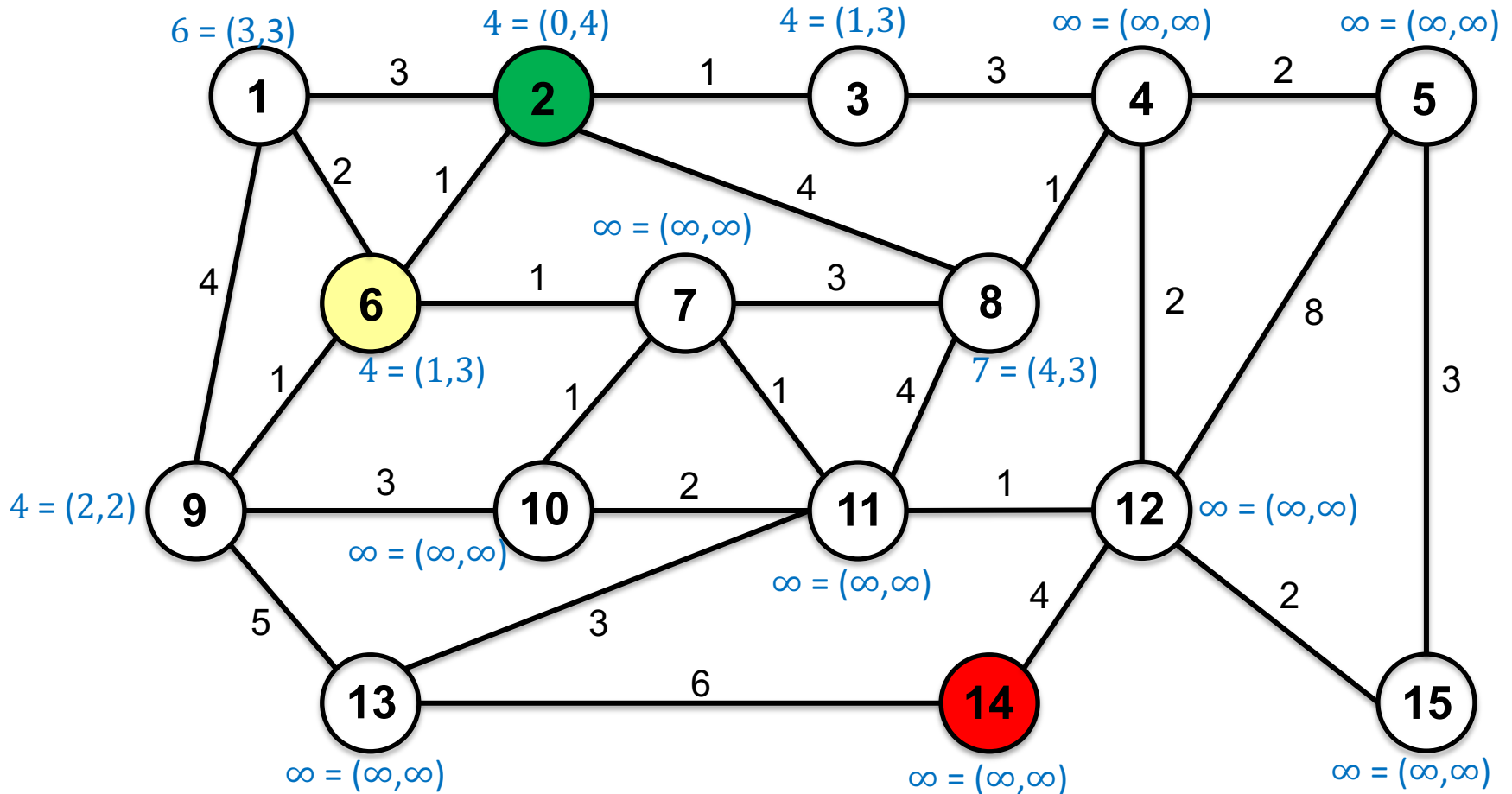
$visited = \{2, 6\}$



Example (Undirected Graph)

$openSet = \{1, 8, 3, 9\}$

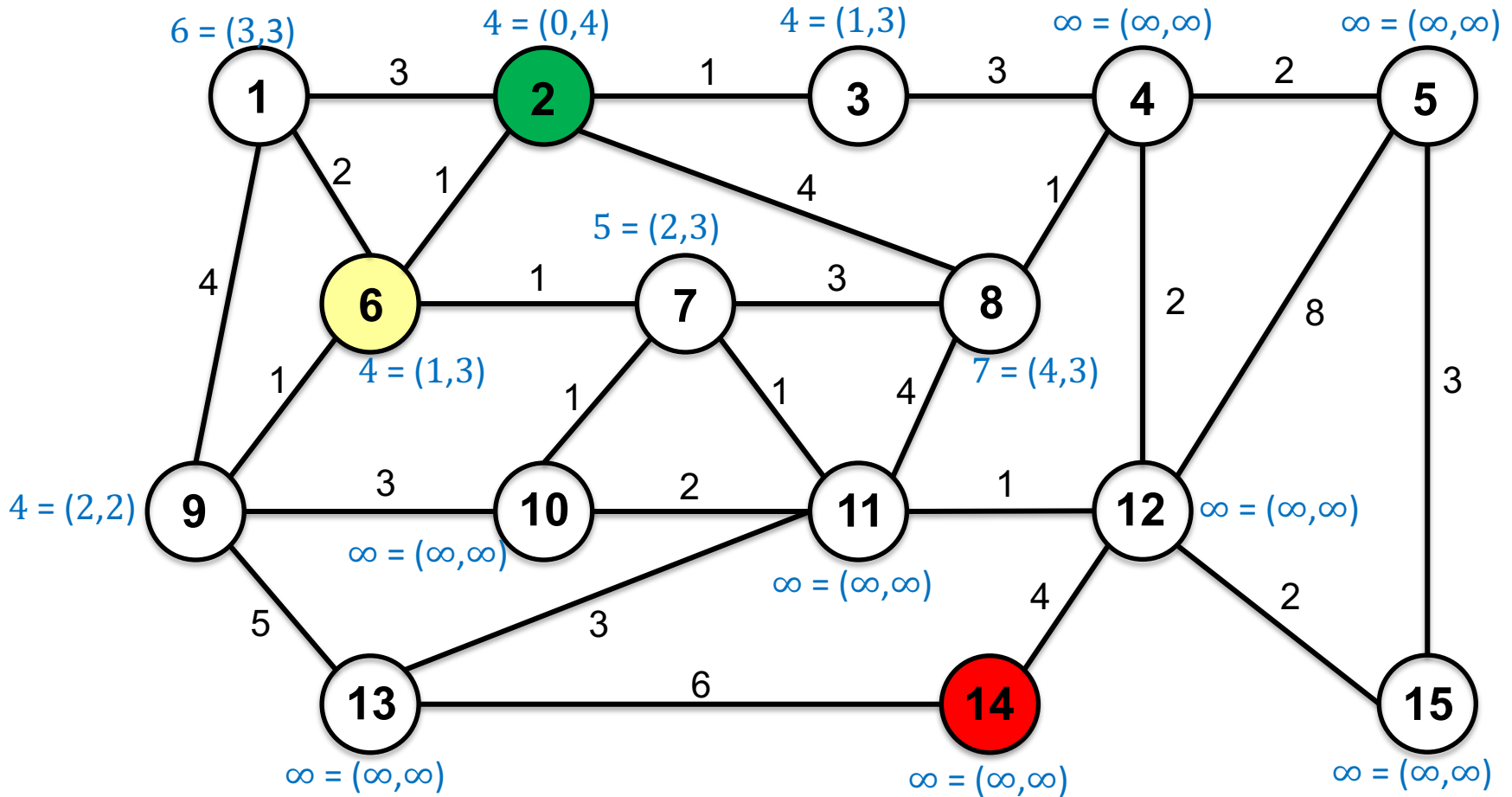
$visited = \{2, 6\}$



Example (Undirected Graph)

$openSet = \{1, 8, 3, 9, 7\}$

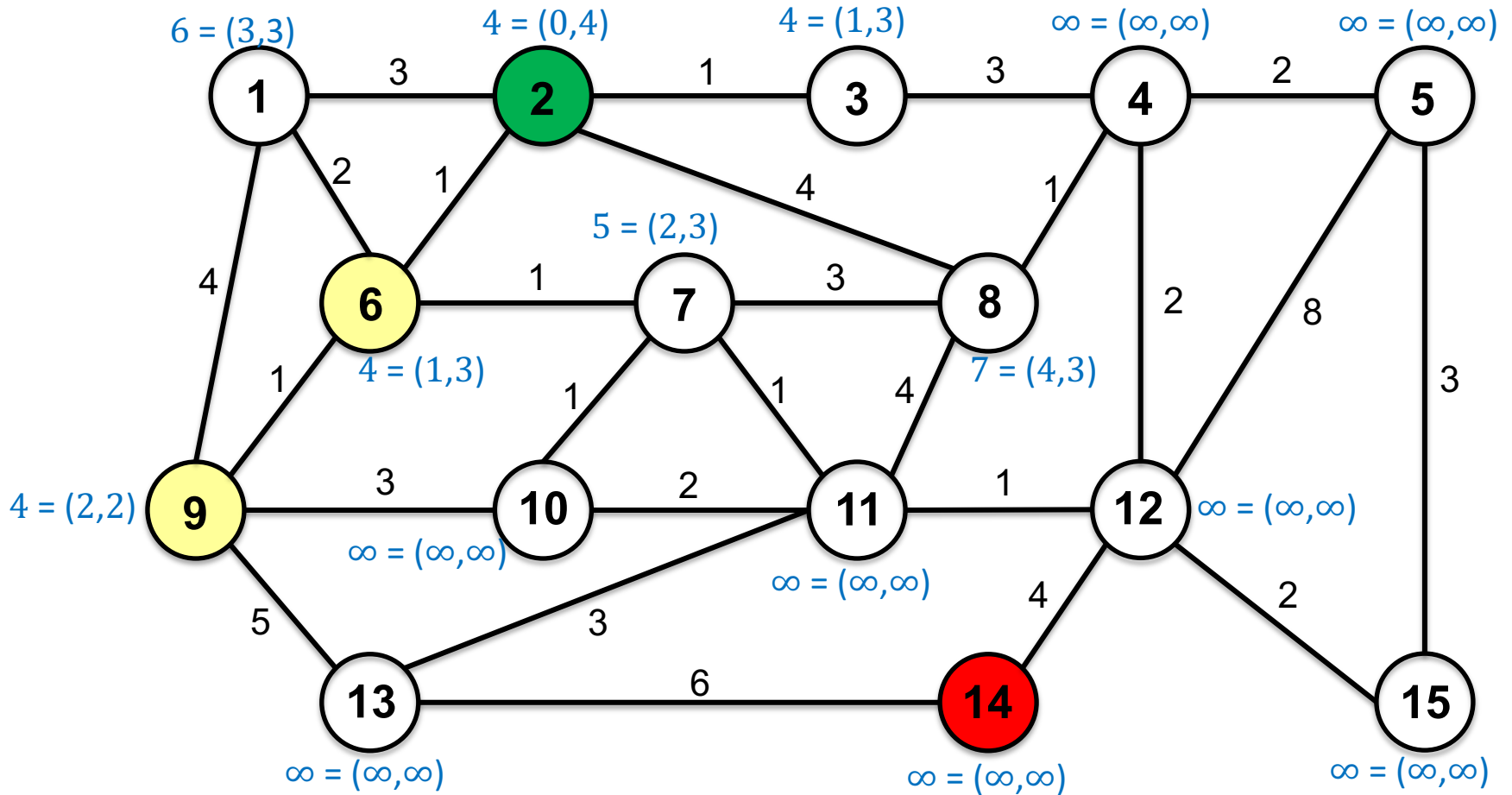
$visited = \{2, 6\}$



Example (Undirected Graph)

$openSet = \{1, 8, 3, 7\}$

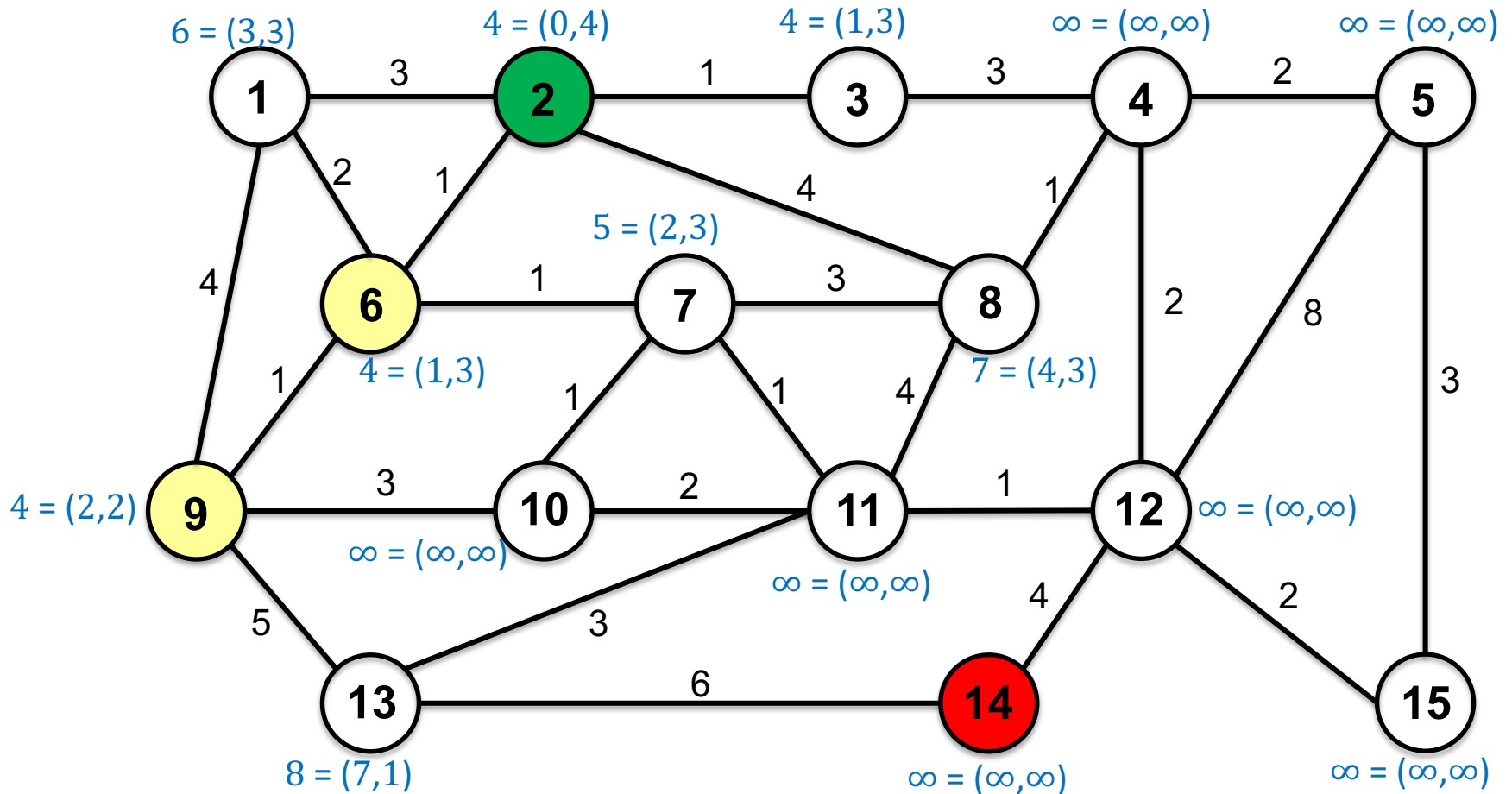
$visited = \{2, 6, 9\}$



Example (Undirected Graph)

$openSet = \{1, 8, 3, 7, 13\}$

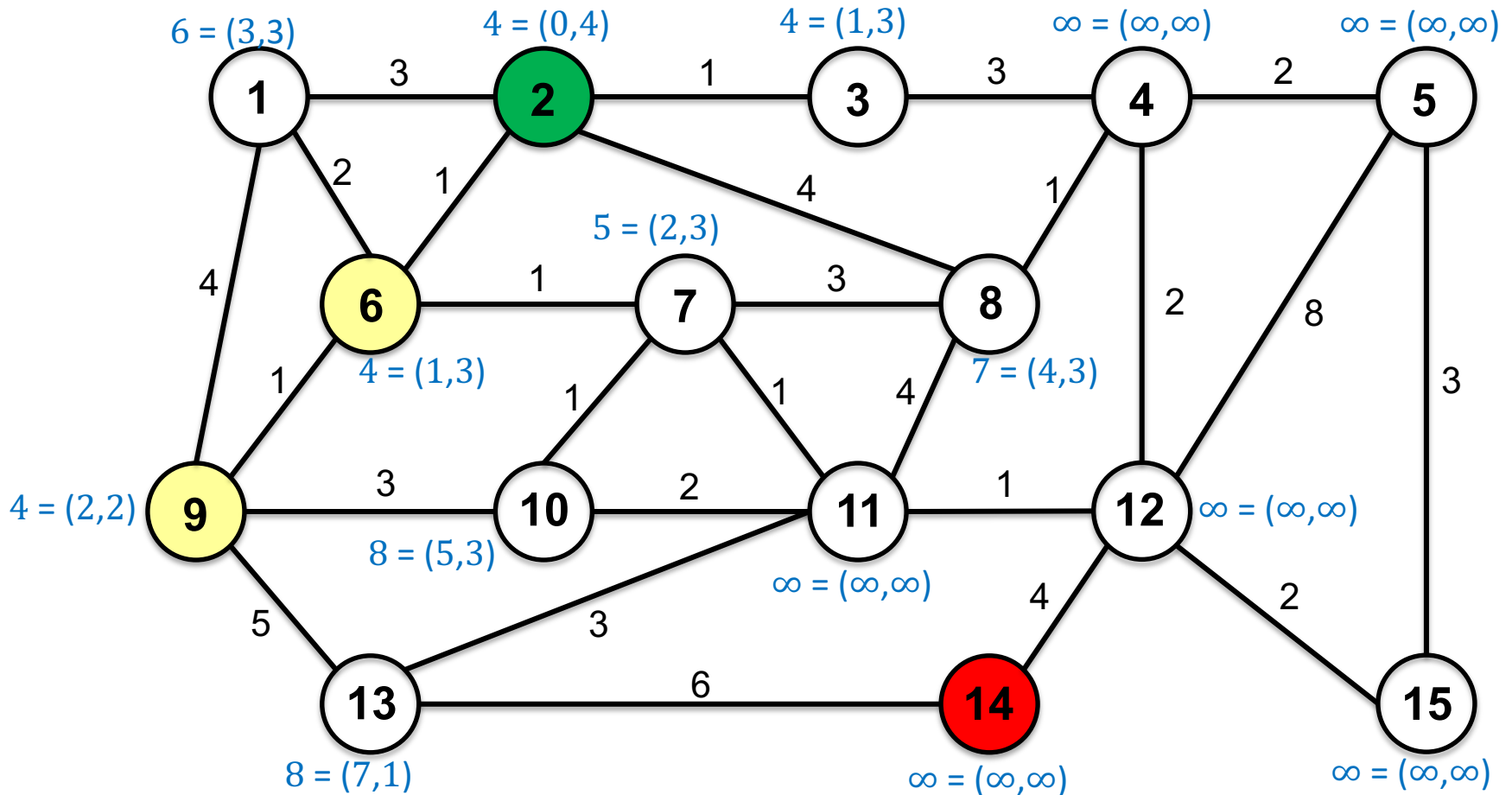
$visited = \{2, 6, 9\}$



Example (Undirected Graph)

$openSet = \{1, 8, 3, 7, 13, 10\}$

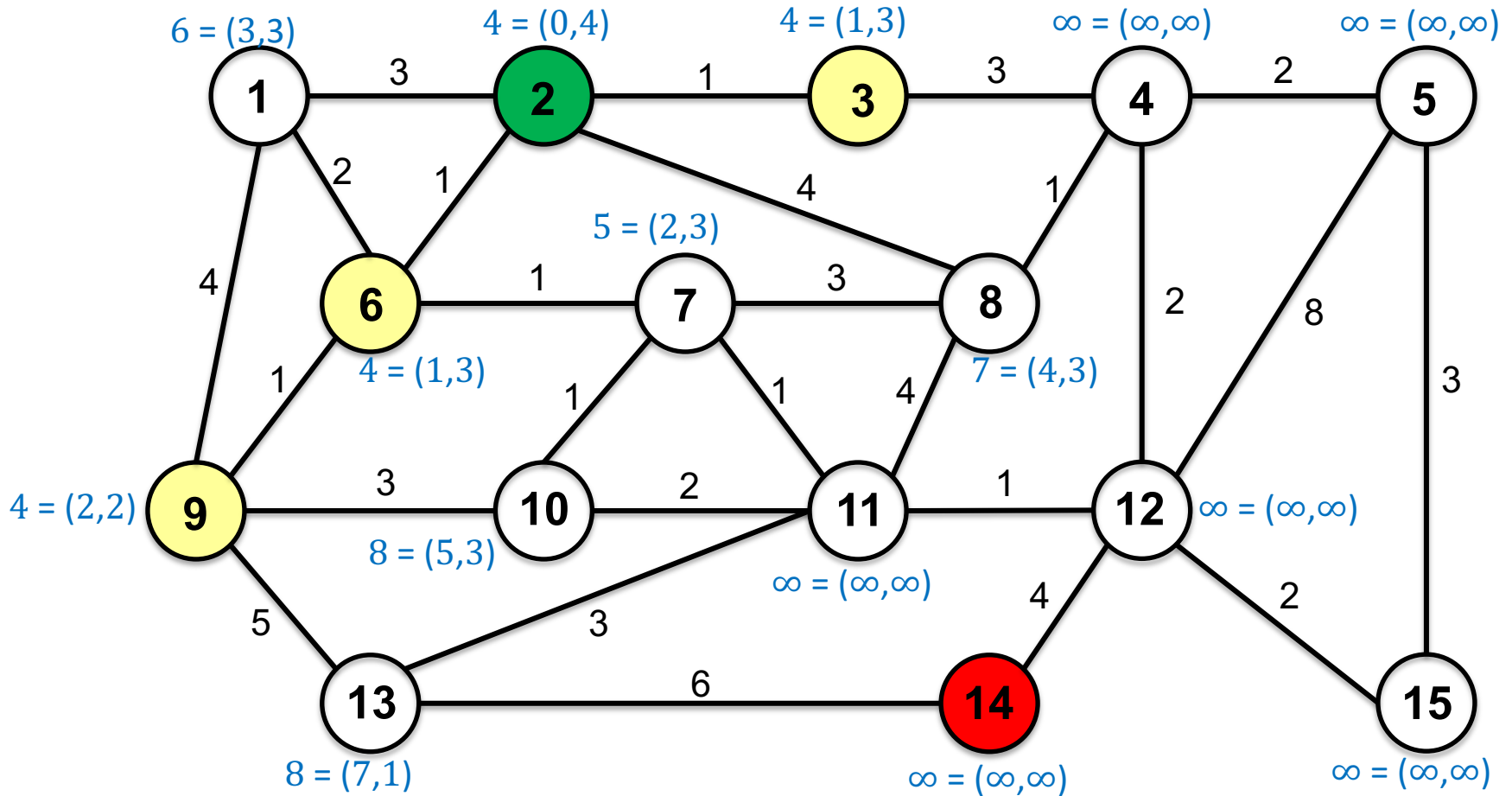
$visited = \{2, 6, 9\}$



Example (Undirected Graph)

$openSet = \{1, 8, 7, 13, 10\}$

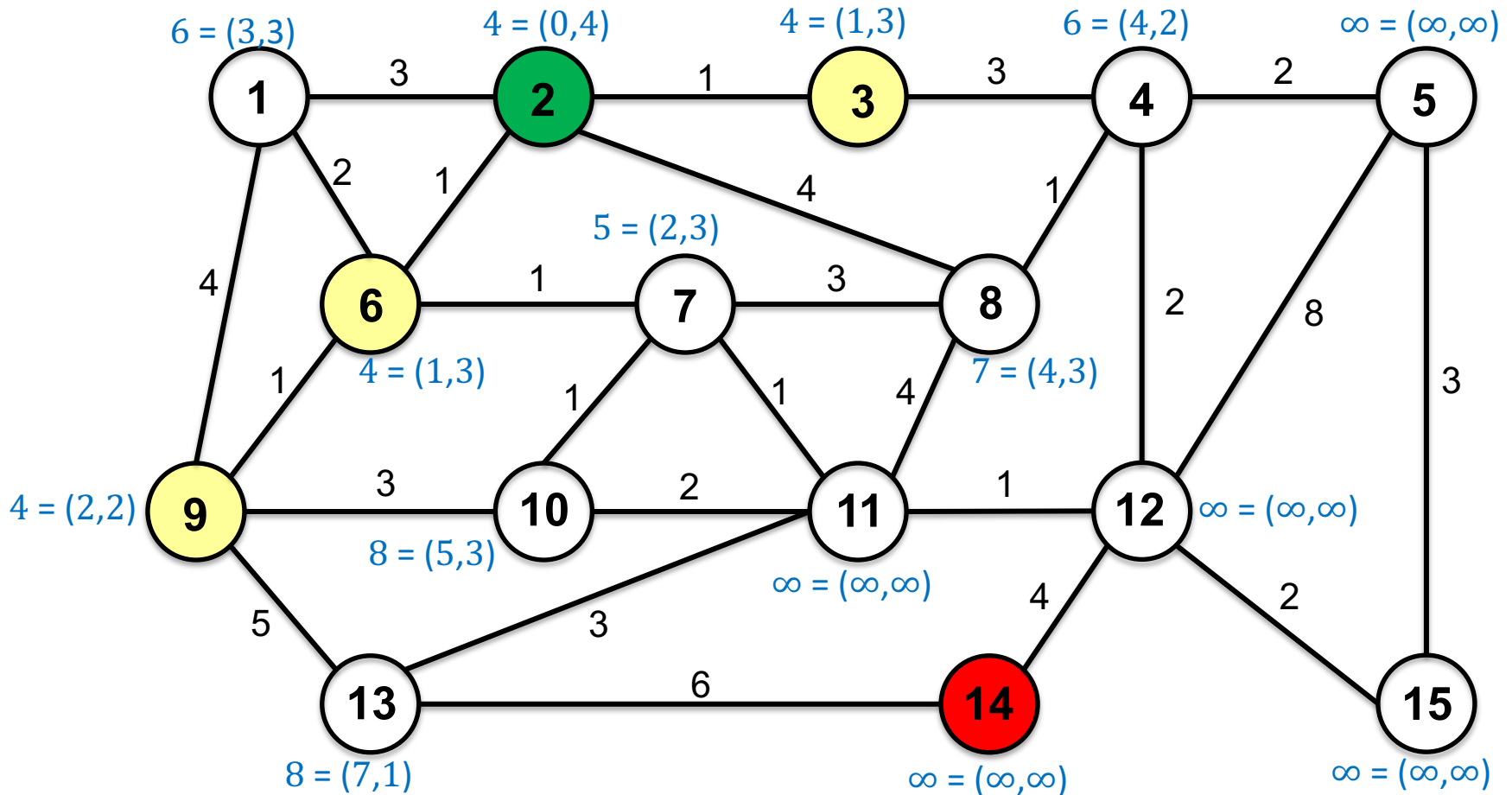
$visited = \{2, 6, 9, 3\}$



Example (Undirected Graph)

$openSet = \{1, 8, 7, 13, 10, 4\}$

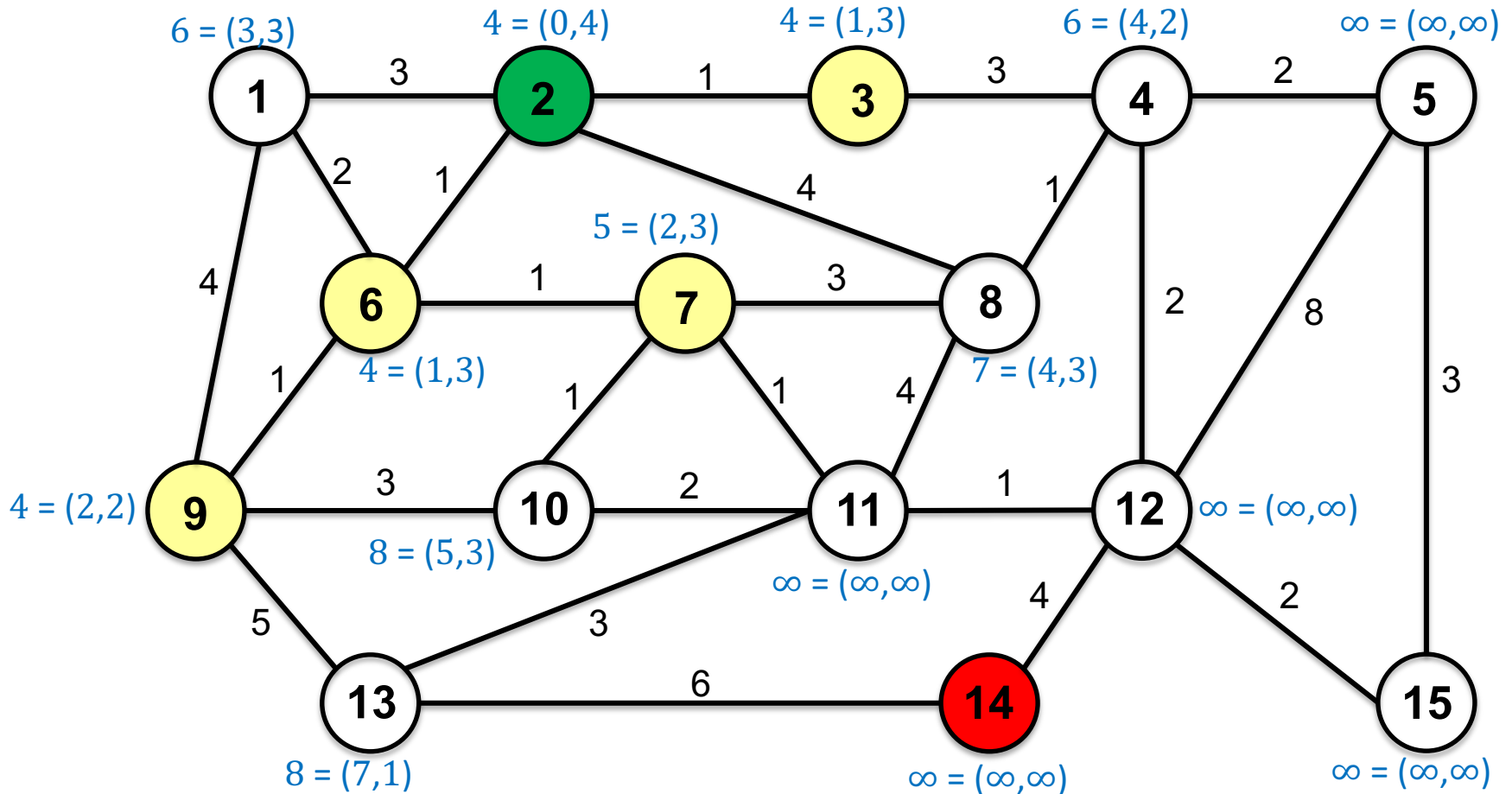
$visited = \{2, 6, 9, 3\}$



Example (Undirected Graph)

$openSet = \{1, 8, 13, 10, 4\}$

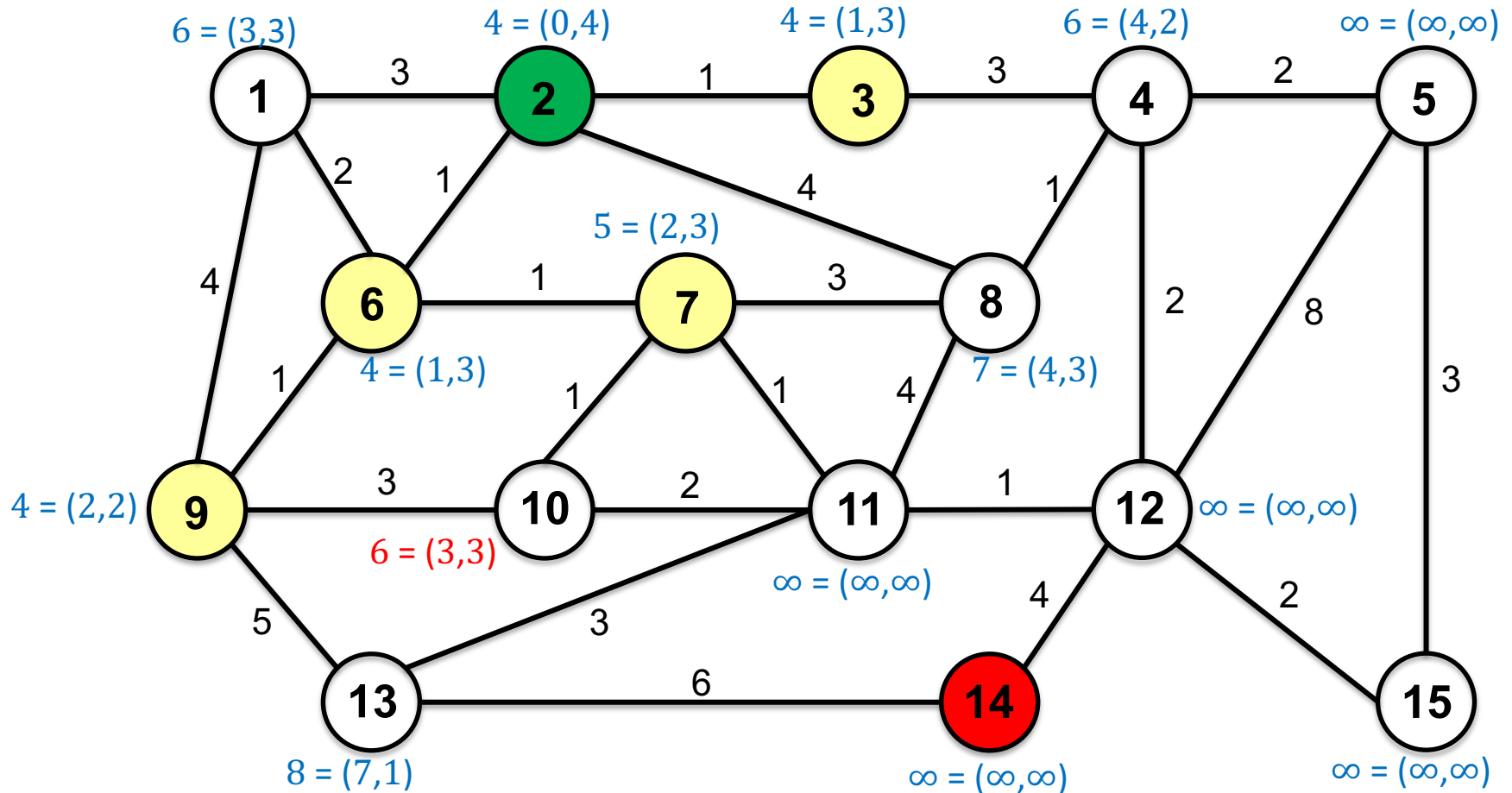
$visited = \{2, 6, 9, 3, 7\}$



Example (Undirected Graph)

$openSet = \{1, 8, 13, 10, 4\}$

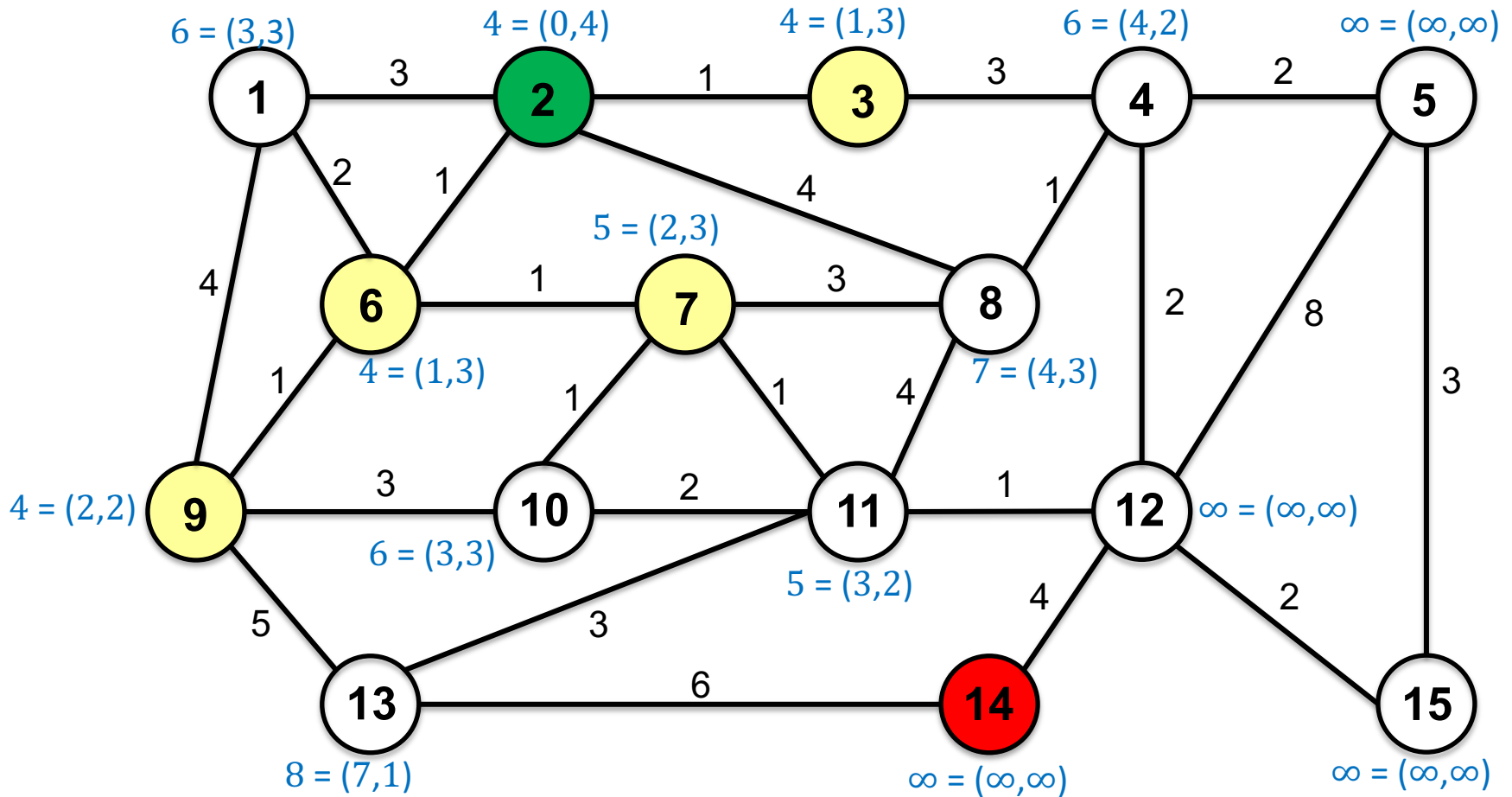
$visited = \{2, 6, 9, 3, 7\}$



Example (Undirected Graph)

$openSet = \{1, 8, 13, 10, 4, 11\}$

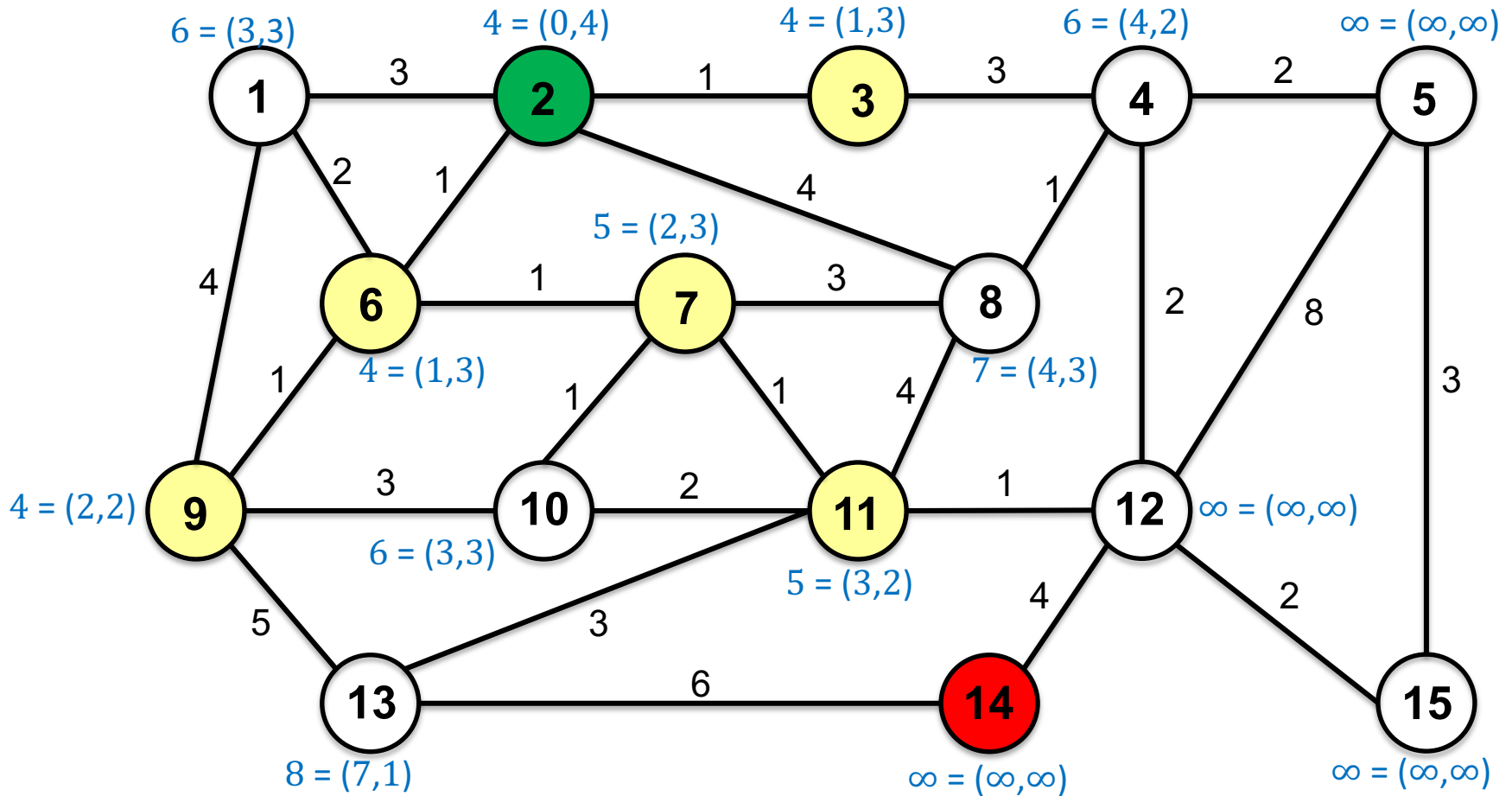
$visited = \{2, 6, 9, 3, 7\}$



Example (Undirected Graph)

openSet = {1, 8, 13, 10, 4}

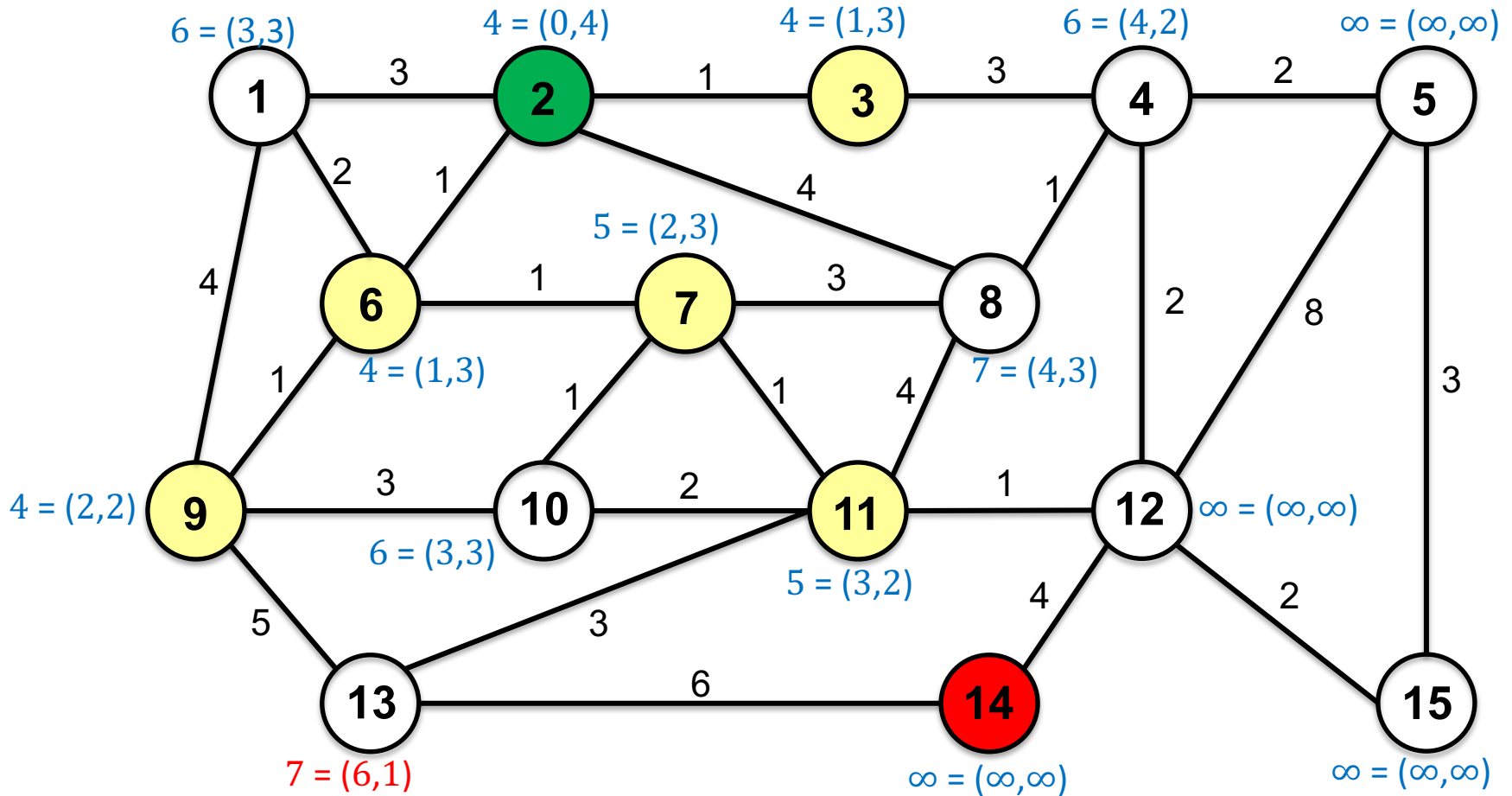
visited = {2, 6, 9, 3, 7, 11}



Example (Undirected Graph)

$openSet = \{1, 8, 13, 10, 4\}$

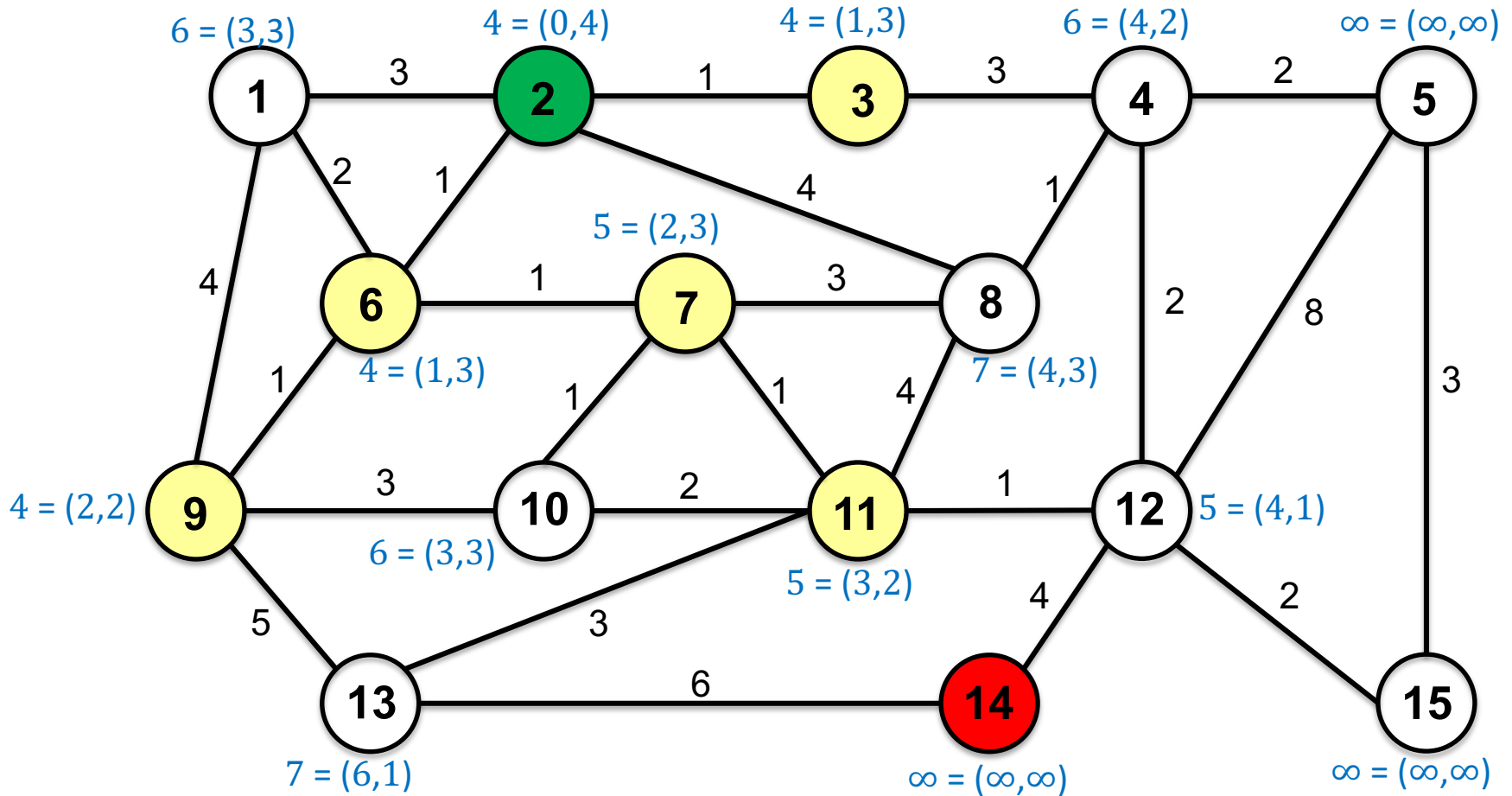
$visited = \{2, 6, 9, 3, 7, 11\}$



Example (Undirected Graph)

$openSet = \{1, 8, 13, 10, 4, 12\}$

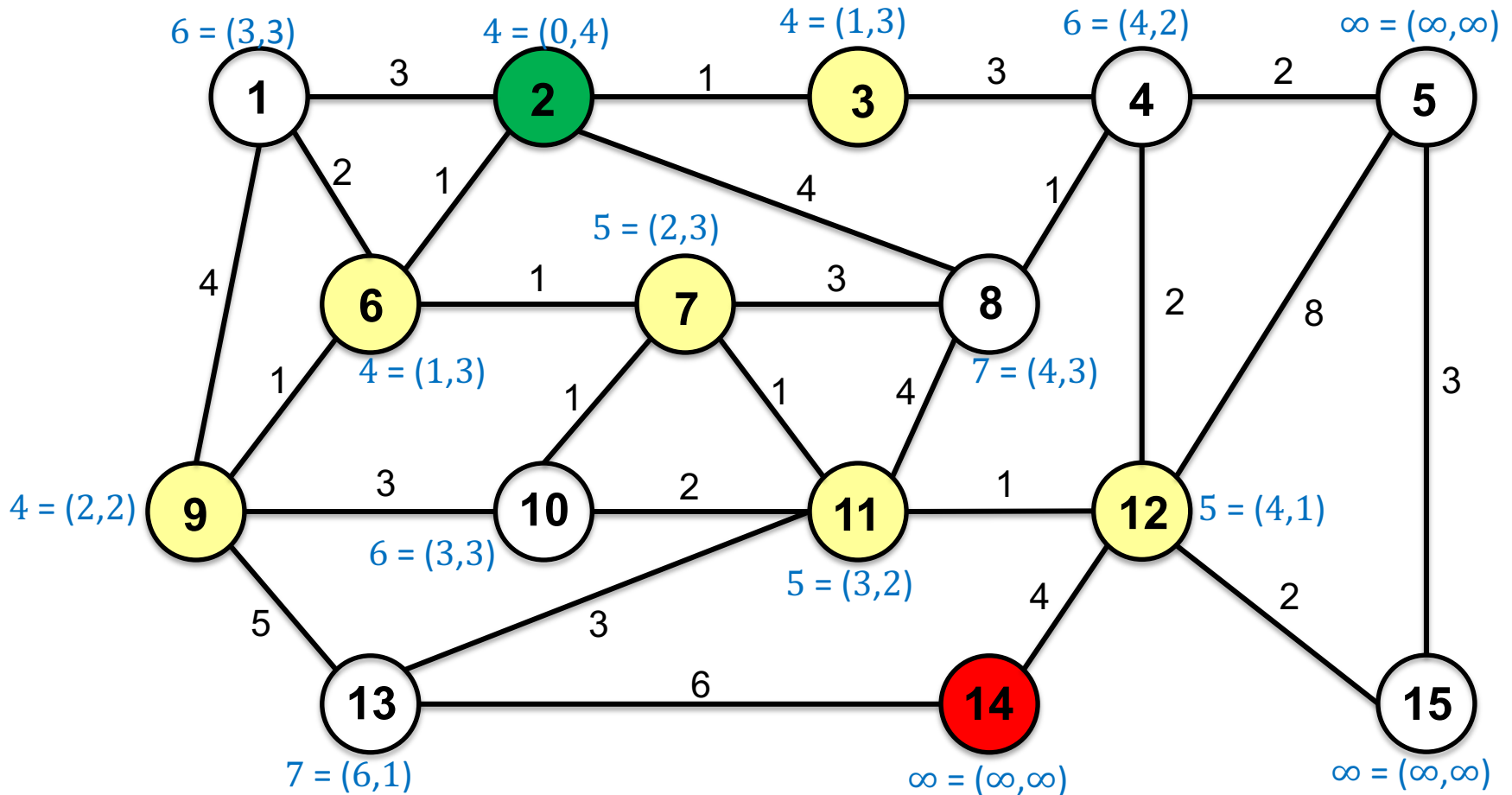
$visited = \{2, 6, 9, 3, 7, 11\}$



Example (Undirected Graph)

$openSet = \{1, 8, 13, 10, 4\}$

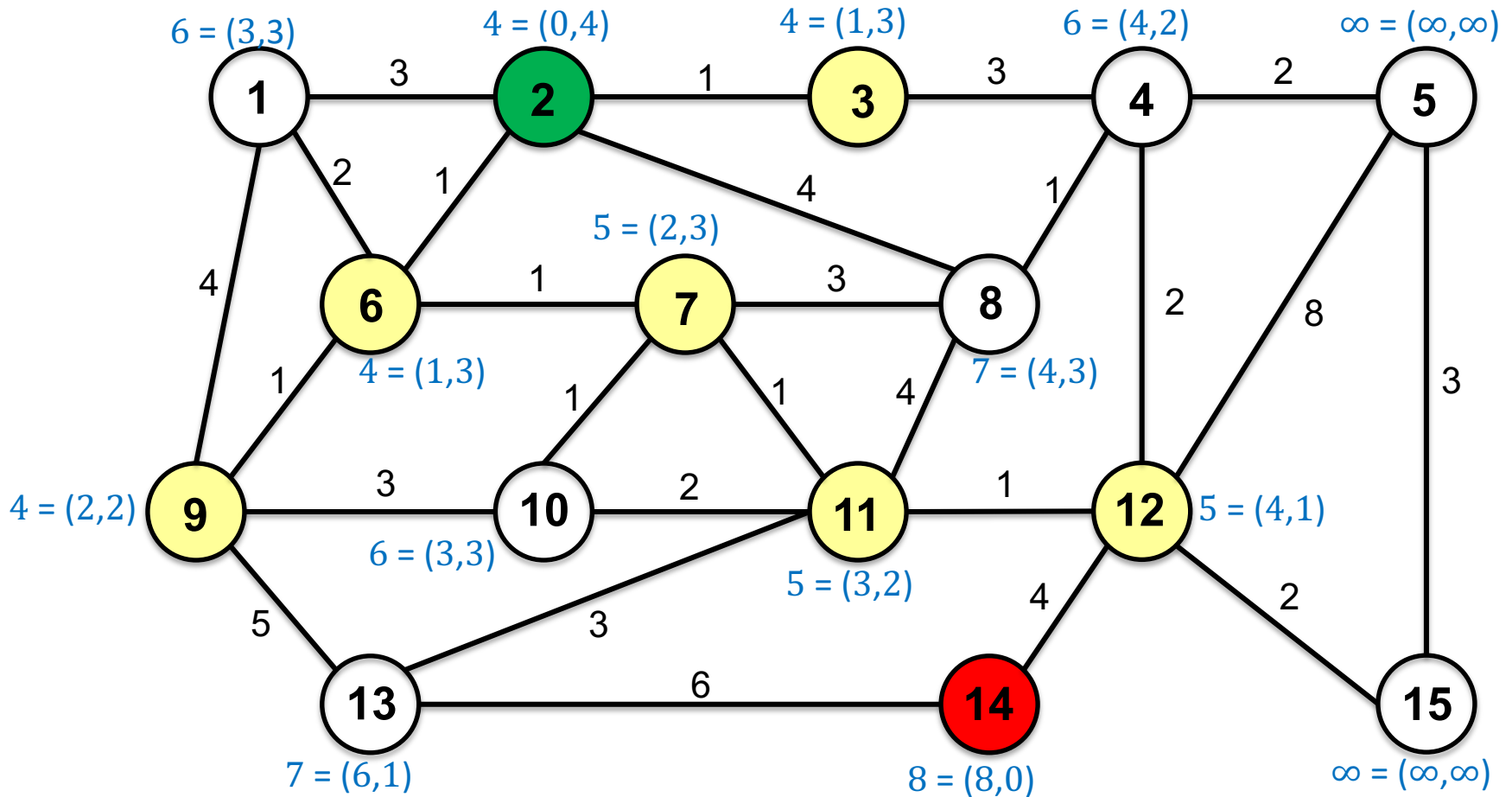
$visited = \{2, 6, 9, 3, 7, 11, 12\}$



Example (Undirected Graph)

$openSet = \{1, 8, 13, 10, 4, 14\}$

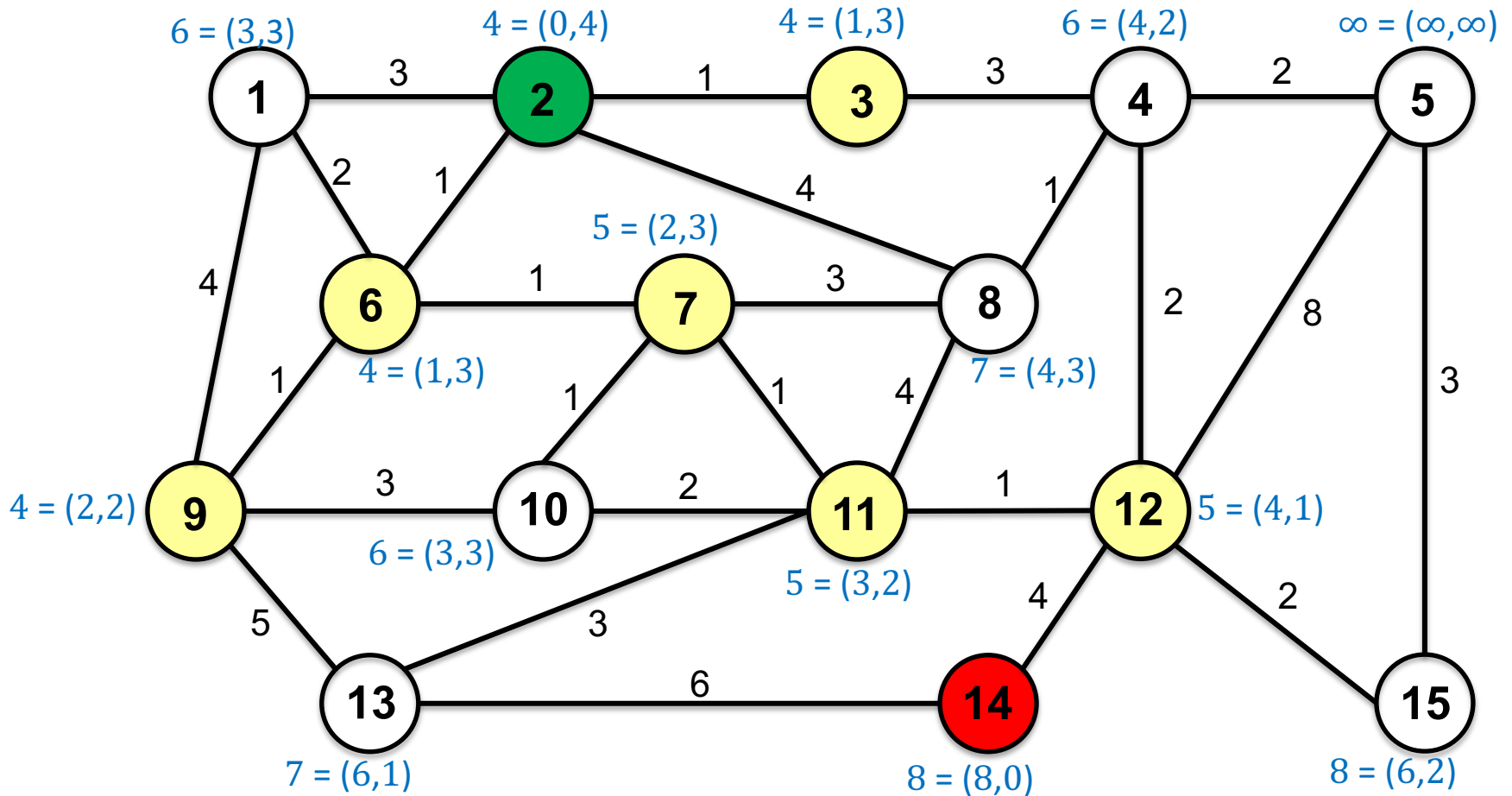
$visited = \{2, 6, 9, 3, 7, 11, 12\}$



Example (Undirected Graph)

$openSet = \{1, 8, 13, 10, 4, 14, 15\}$

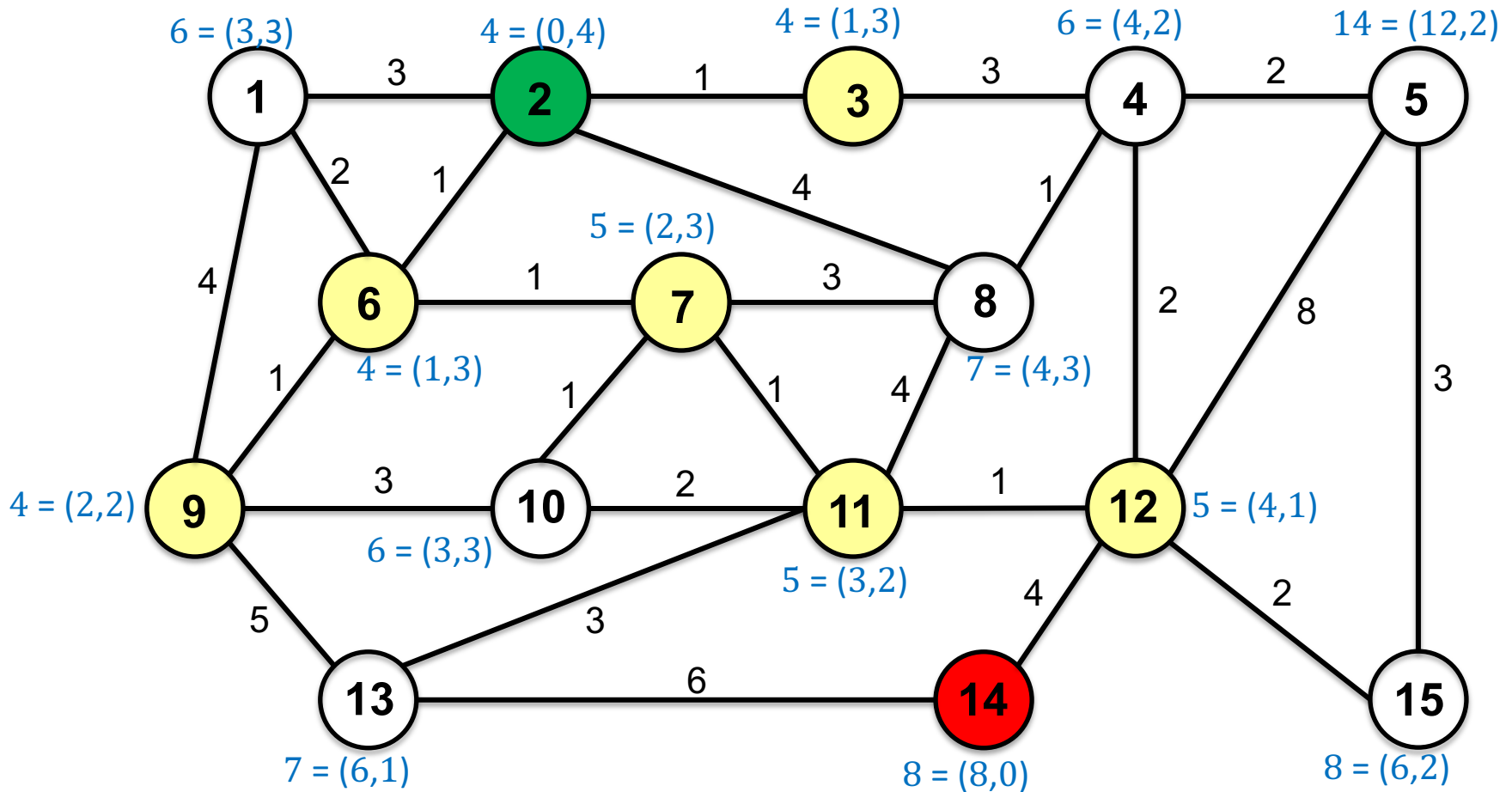
$visited = \{2, 6, 9, 3, 7, 11, 12\}$



Example (Undirected Graph)

$openSet = \{1, 8, 13, 10, 4, 14, 15, 5\}$

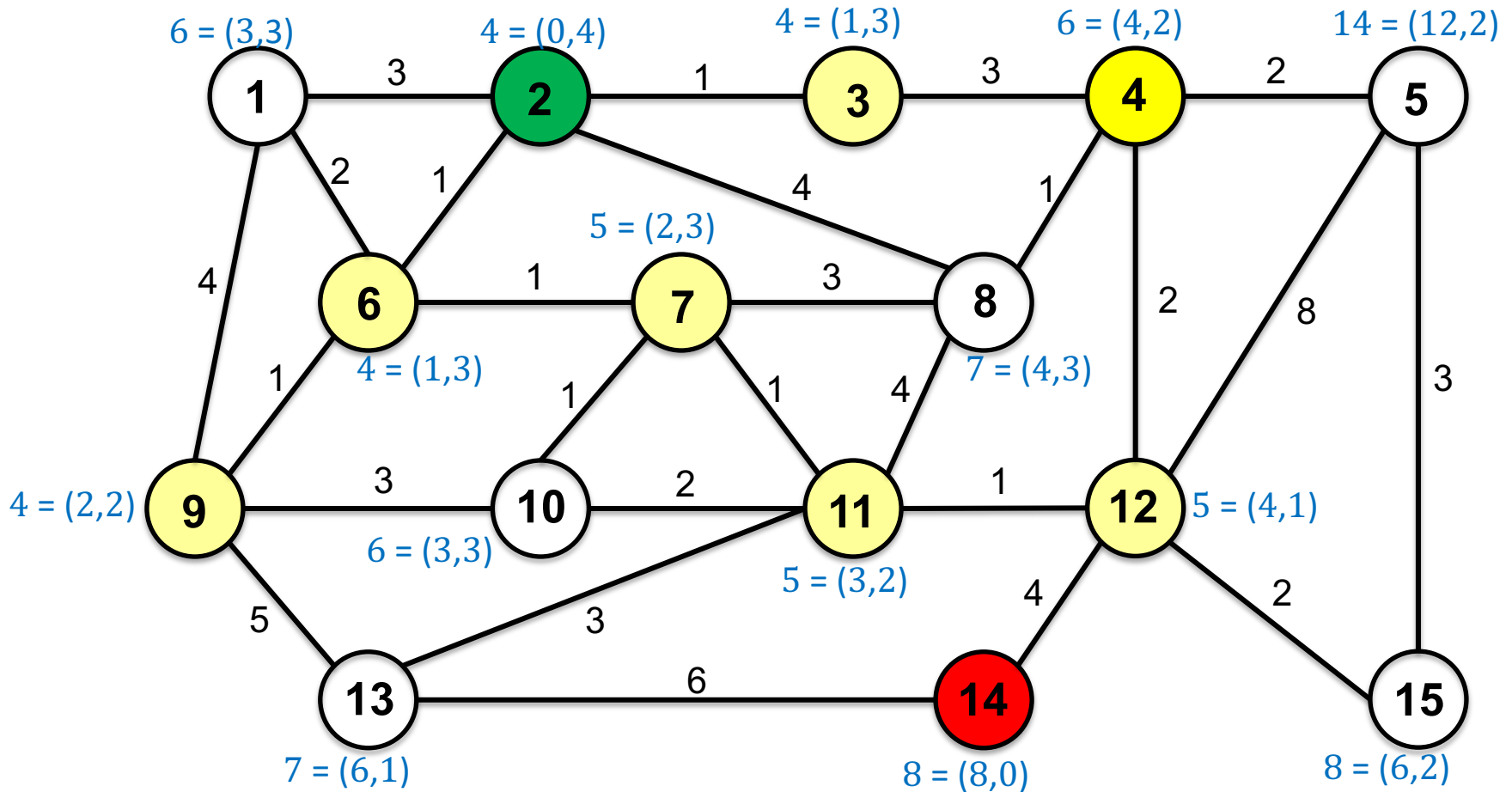
$visited = \{2, 6, 9, 3, 7, 11, 12\}$



Example (Undirected Graph)

$openSet = \{1, 8, 13, 10, 14, 15, 5\}$

$visited = \{2, 6, 9, 3, 7, 11, 12, 4\}$

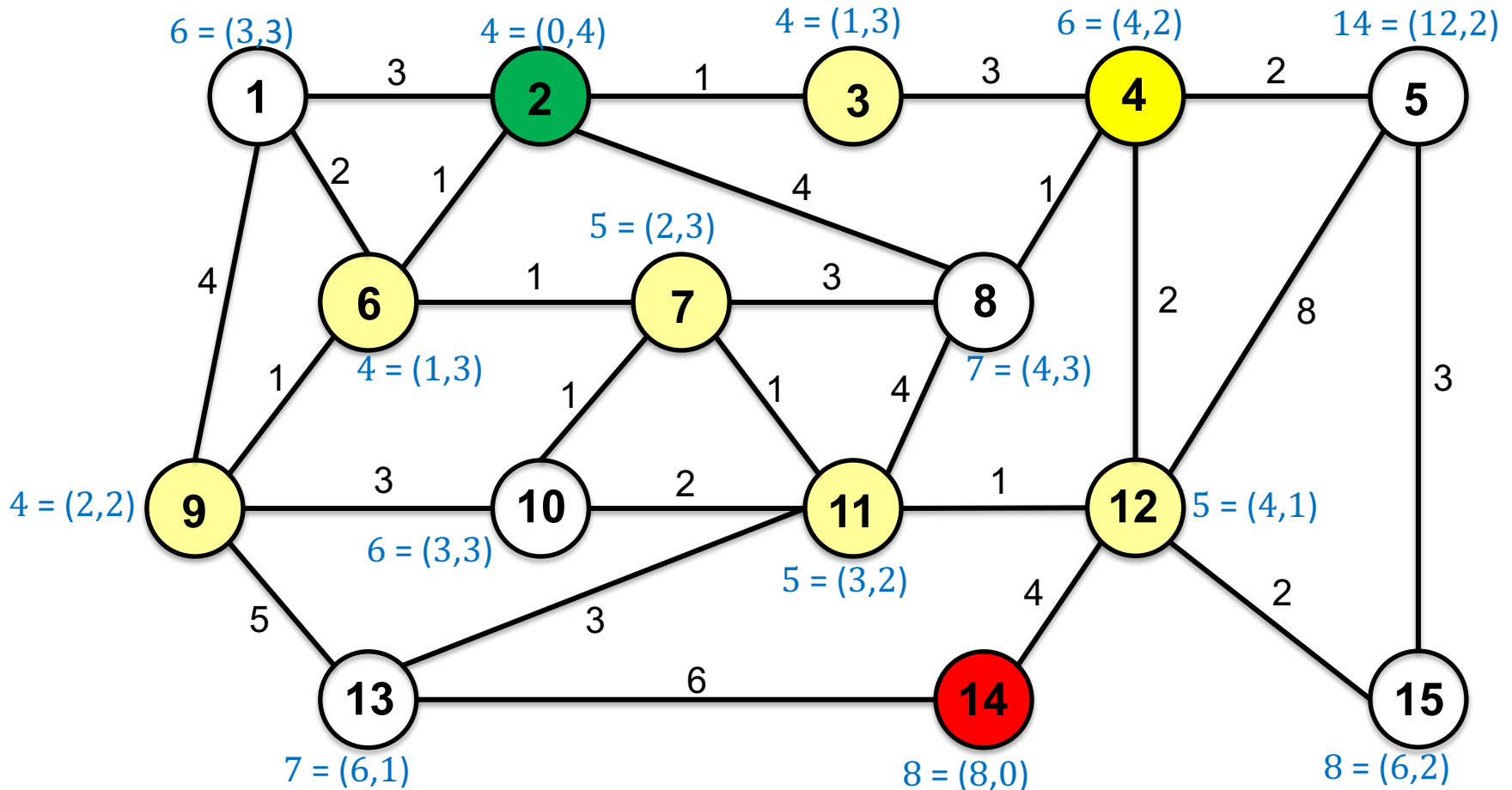


Example (Undirected Graph)

$openSet = \{1, 8, 13, 10, 14, 15, 5\}$

$visited = \{2, 6, 9, 3, 7, 11, 12, 4\}$

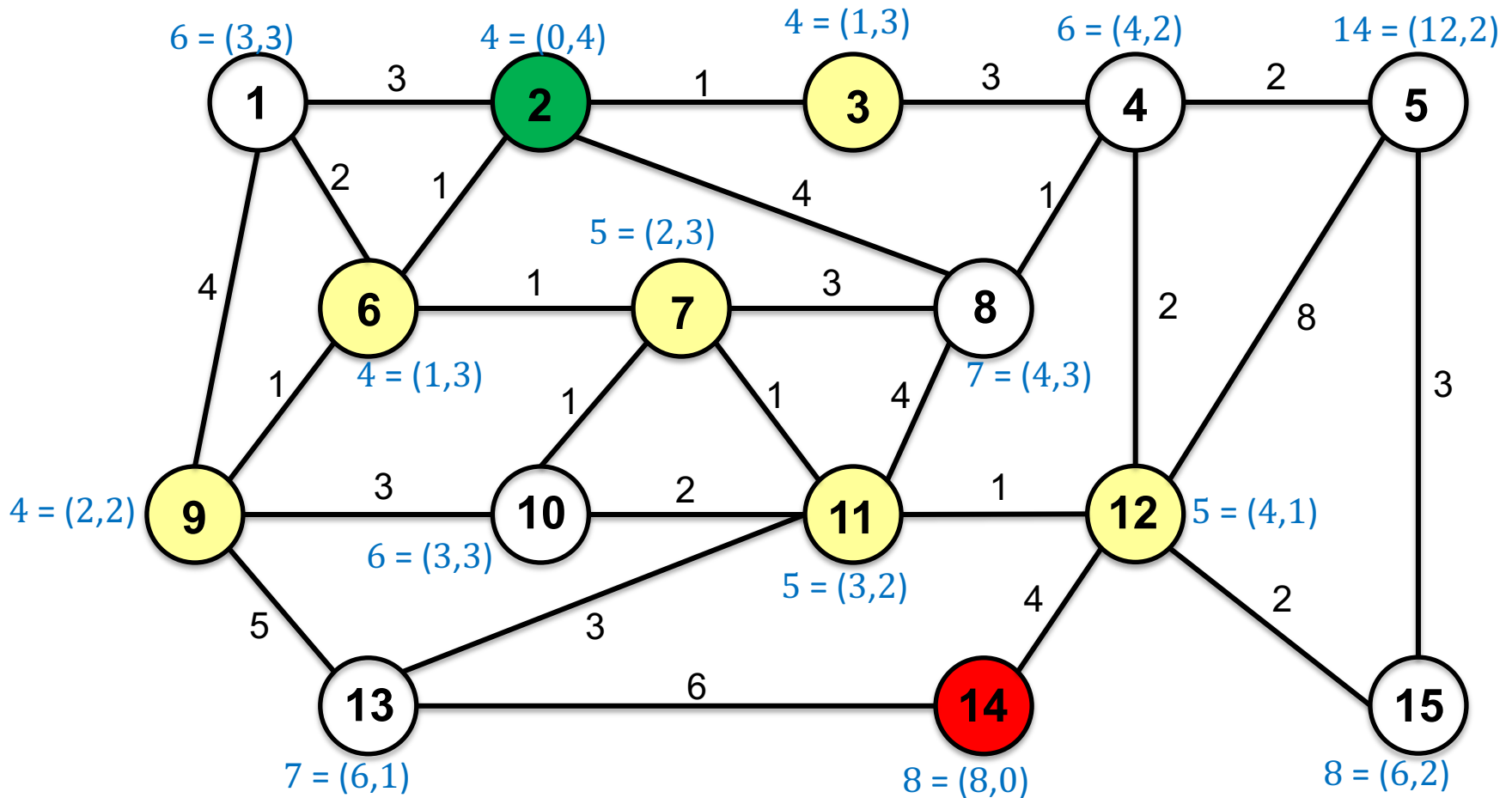
Repeat more!!



Example (Undirected Graph)

openSet = {1, 8, 13, 10, 4, 15, 5, ...}

visited = {2, 6, 9, 3, 7, 11, 12, 14}



Example (Undirected Graph)

openSet = {1, 8, 13, 10, 4, 15, 5}

visited = {2, 6, 9, 3, 7, 11, 12, 14}

