Graphs: Searching (Advanced)

Week 3

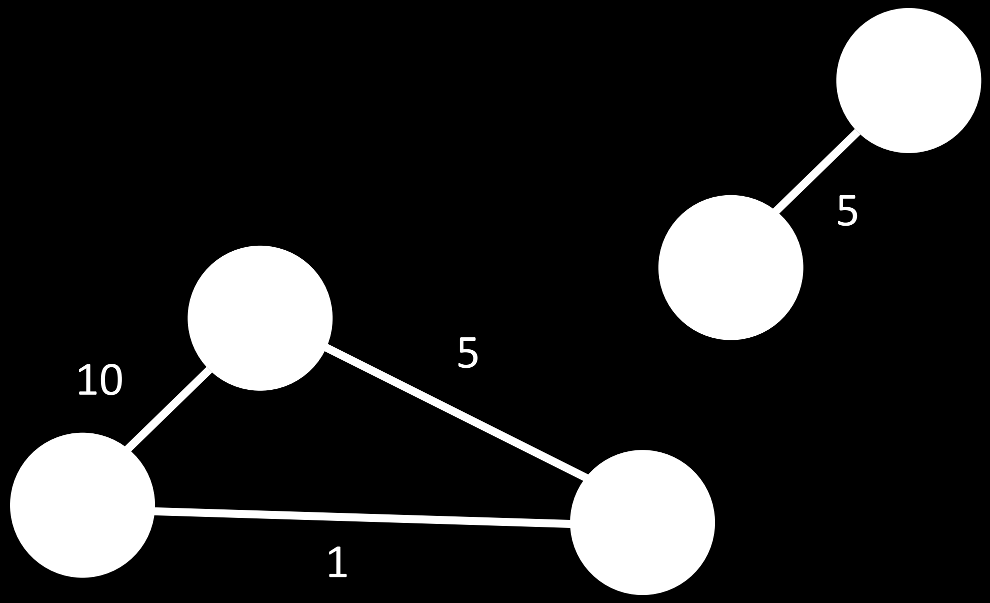
Last Time

- Depth-First Search (DFS)
  - Expands depth-first (keep going deeper until it can’t anymore)
  - Finds a path if one exists
  - No guarantees on path length
  - Can be implemented with a stack
- Breadth-First Search (BFS)
  - Expands breadth-first (hopes to find the target node in the first level!)
  - Finds a shortest path if the nodes are connected
  - Can be implemented with a queue
- Both algorithms are implemented similarly

Weighted Edges

- Last class, considered graphs with unweighted edges
    - i.e. all edges with weight 1
- In real life, most things have an associated comparative **cost**

Today



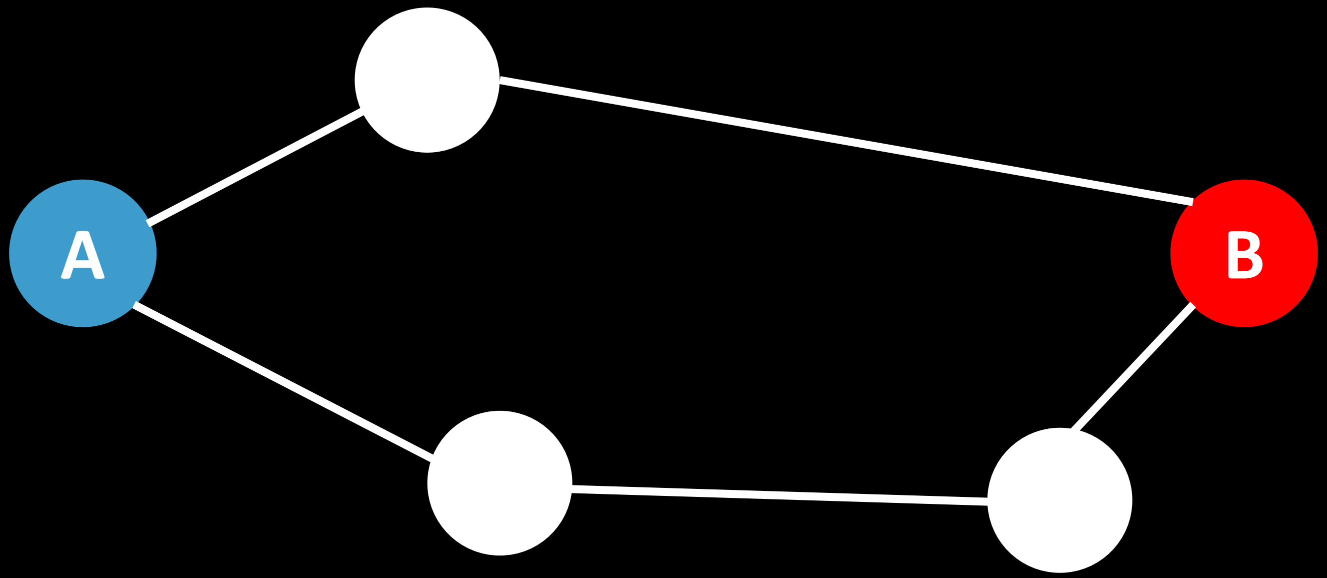
Undirected + Weighted

Could be Cyclic

Could be Disconnected

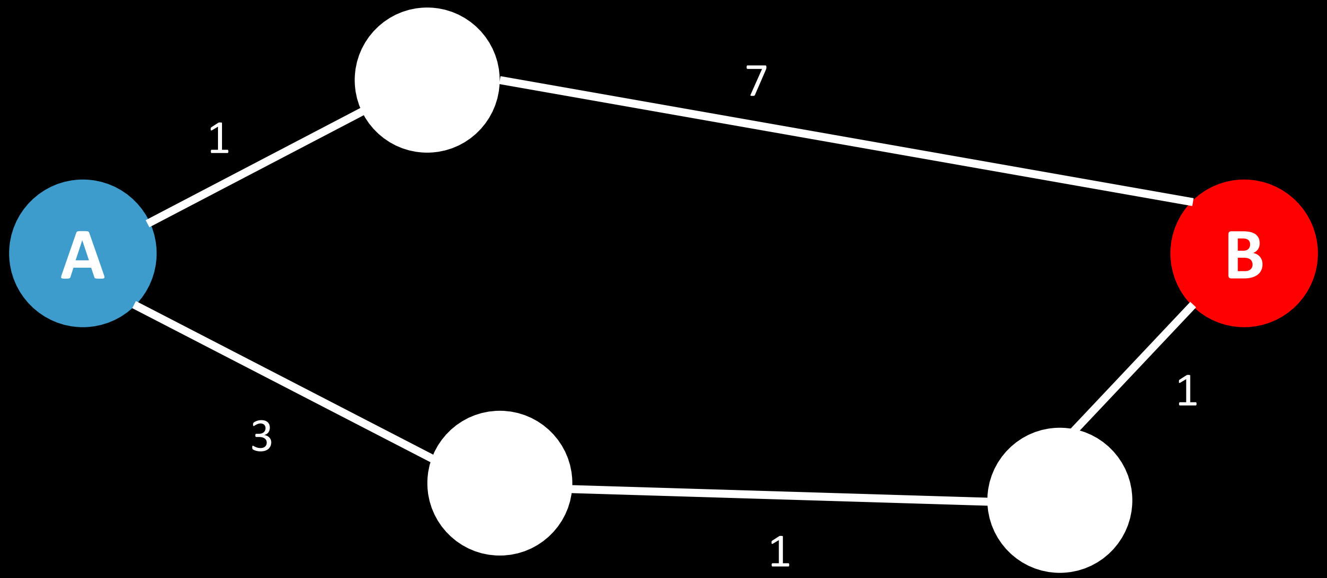
Weighted Edges

- Simple Example



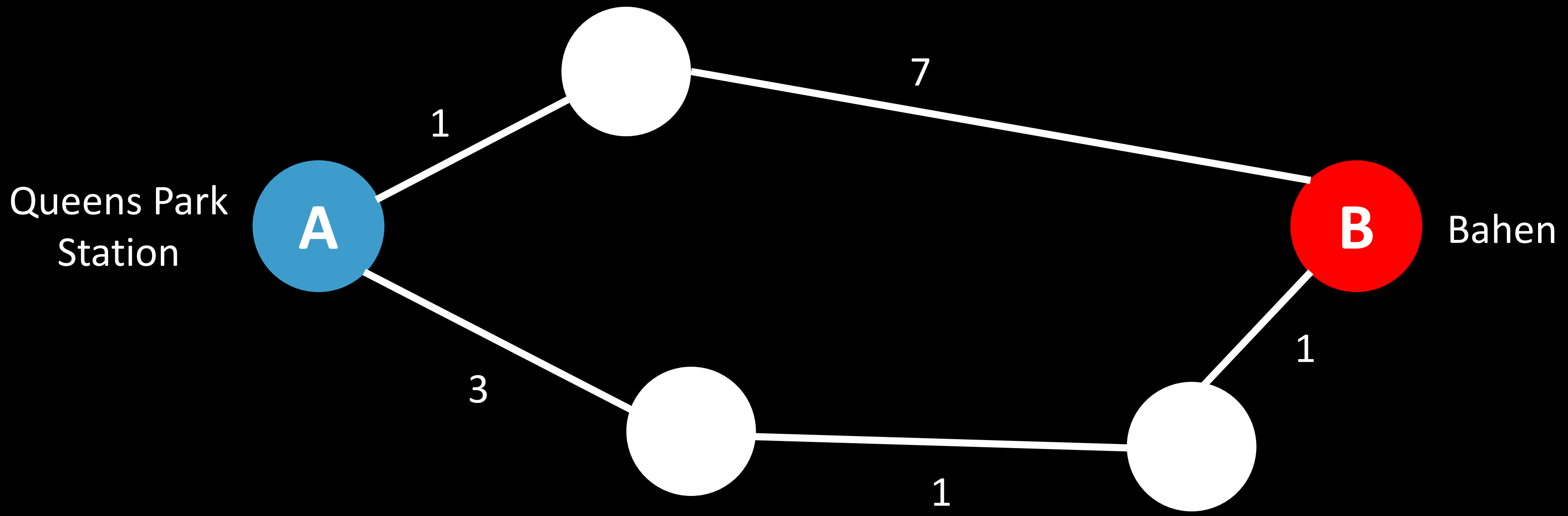
Weighted Edges

- Simple Example



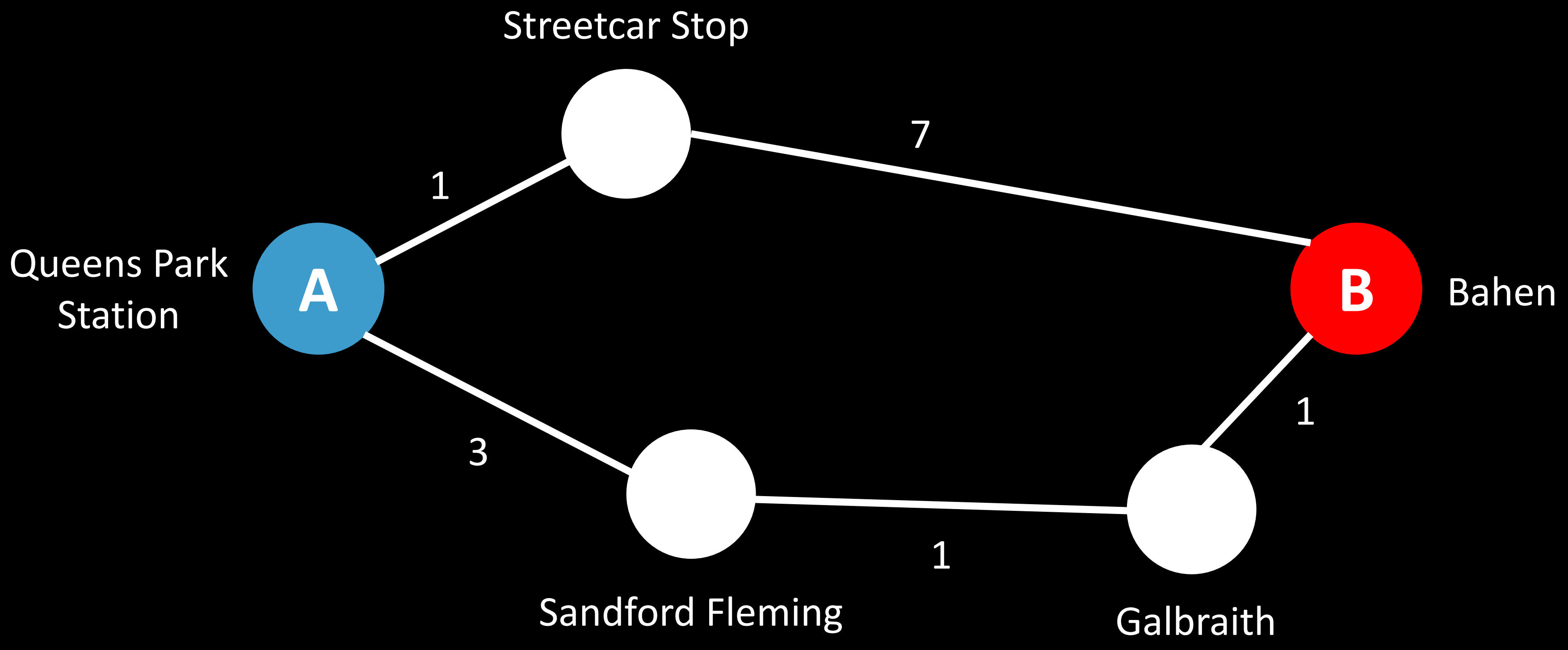
Weighted Edges

- Simple Example



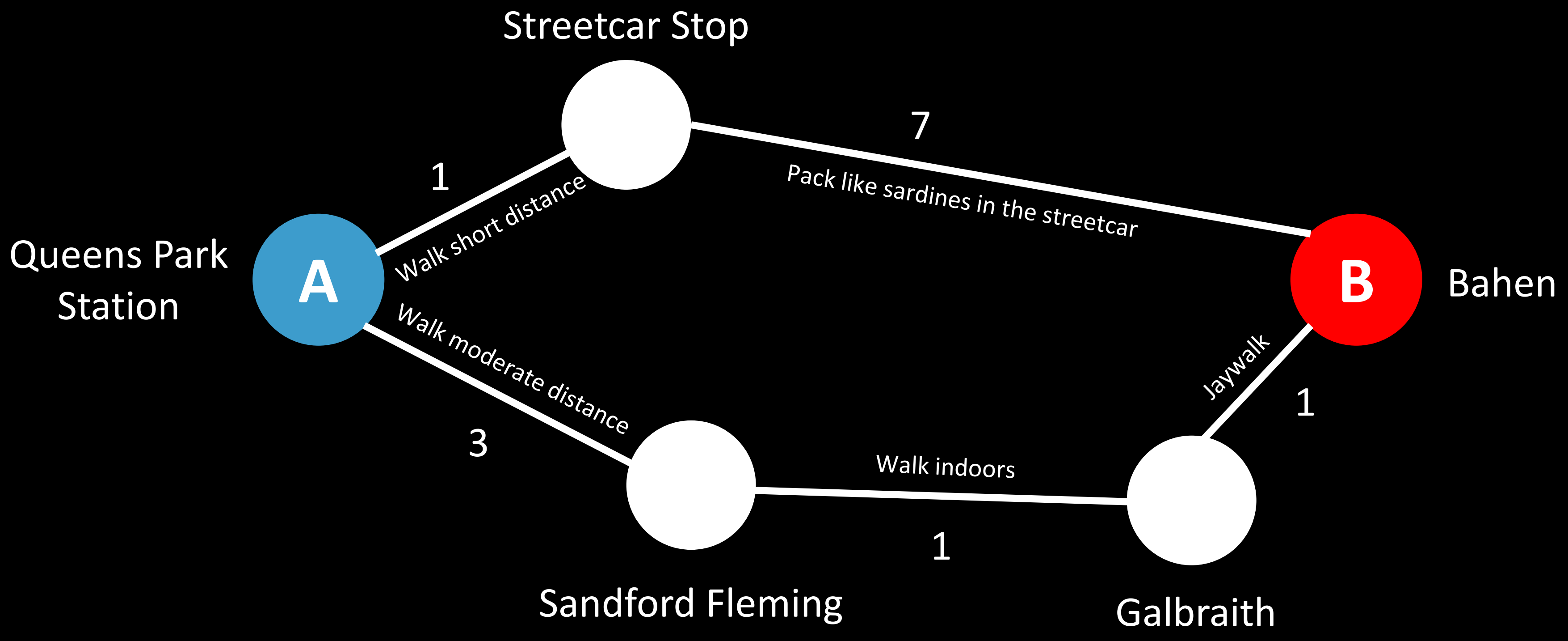
Weighted Edges

- Simple Example



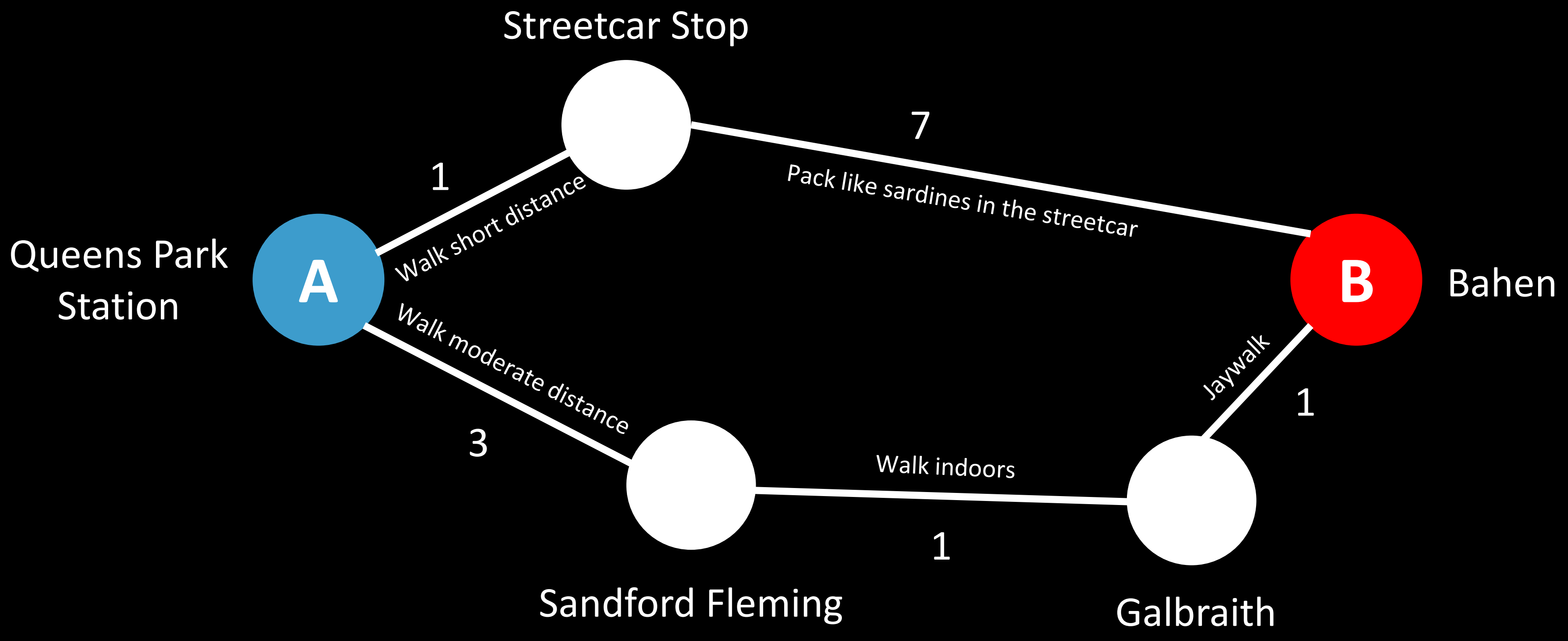
Weighted Edges

- Simple Example



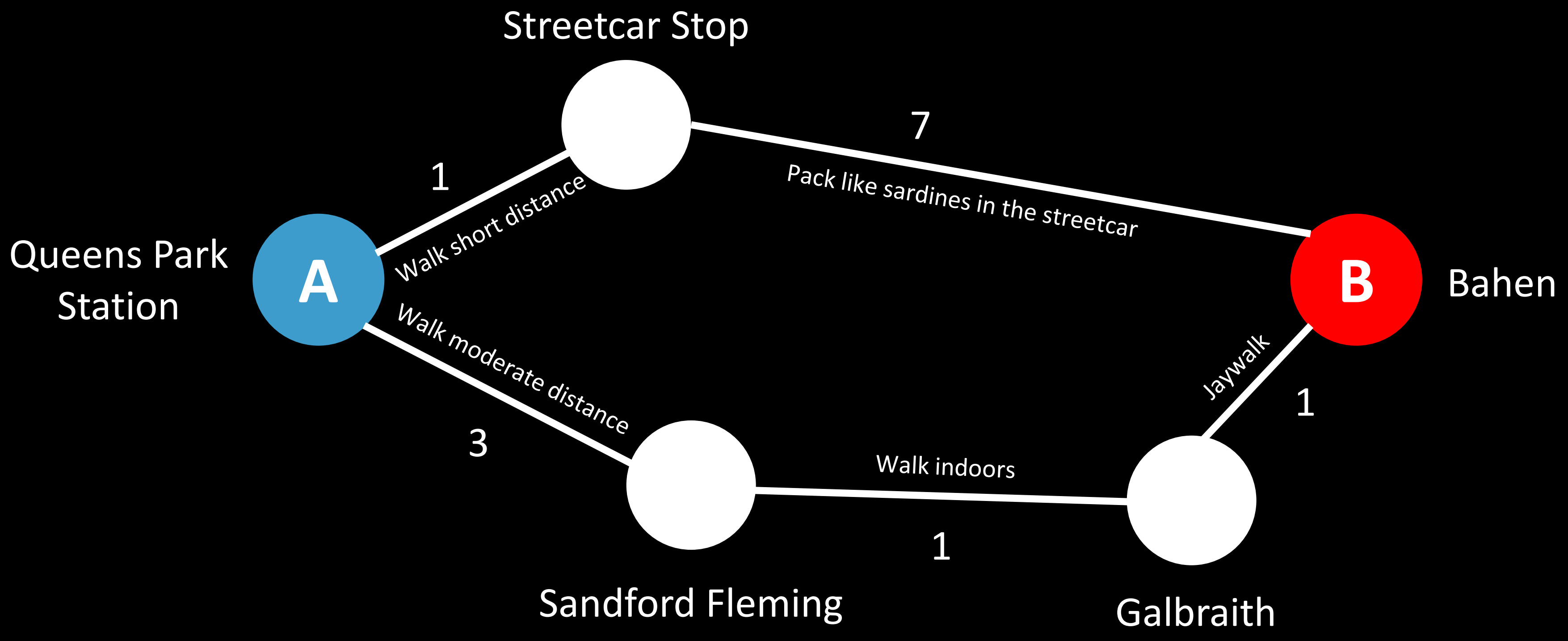
Weighted Edges

- Simple Example – how to find the path with **minimal cost**?

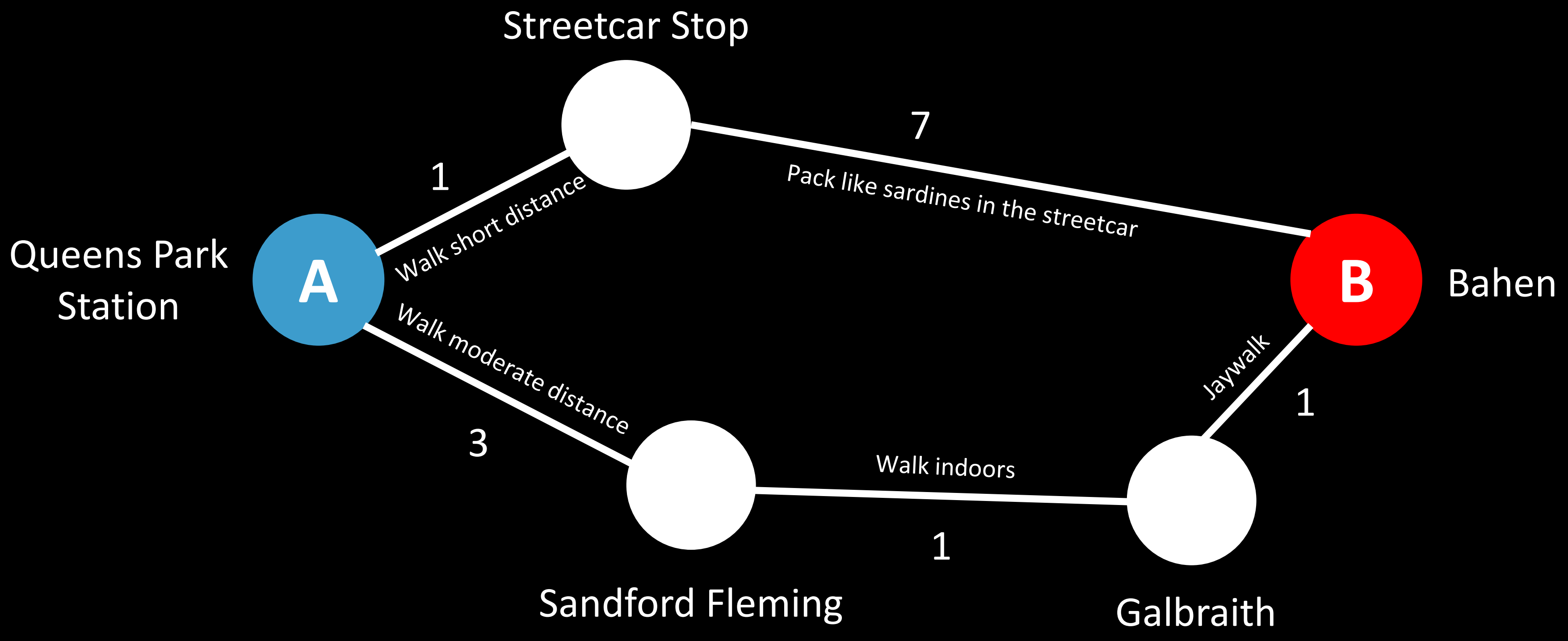


Weighted Edges

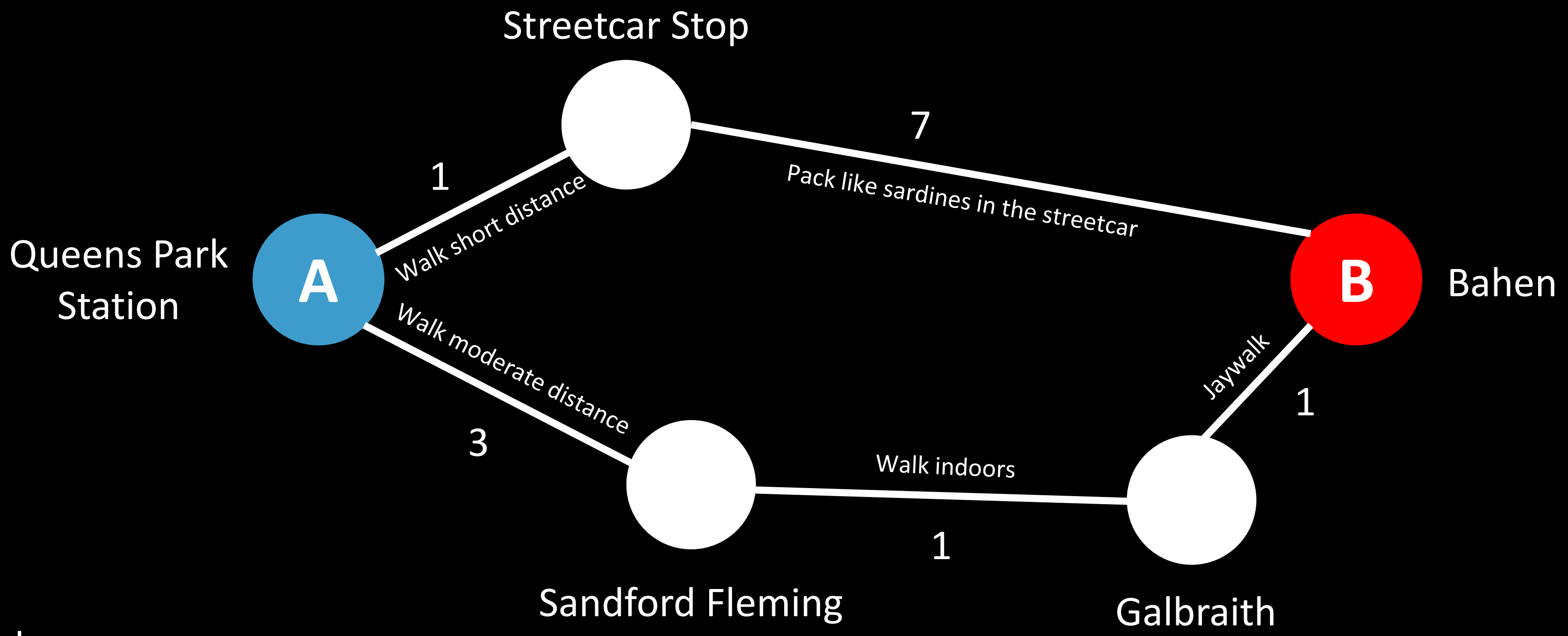
- Q: how to find the path with **minimal cost**?



Weighted Edges



Weighted Edges

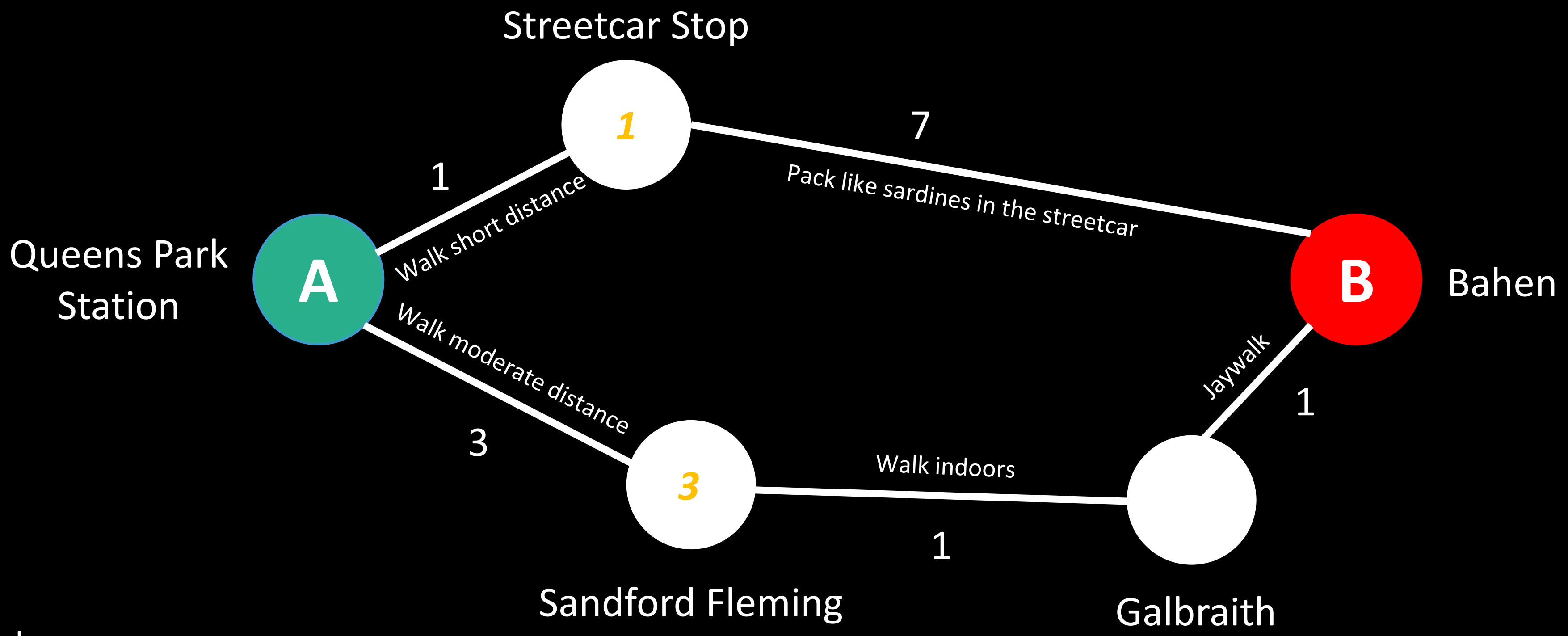


⬜ Unvisited nodes

🟩 Visited nodes

d Distance

Weighted Edges

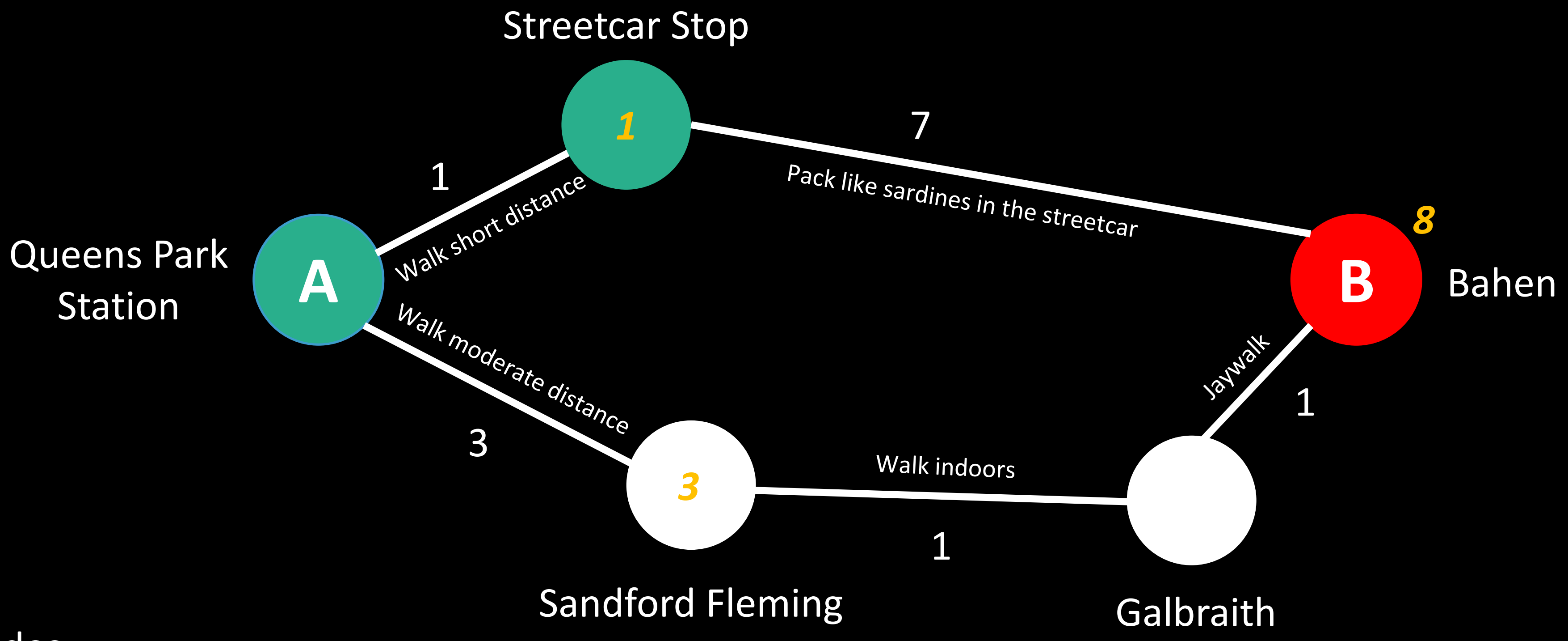


⬜ Unvisited nodes

🟩 Visited nodes

d Distance

Weighted Edges

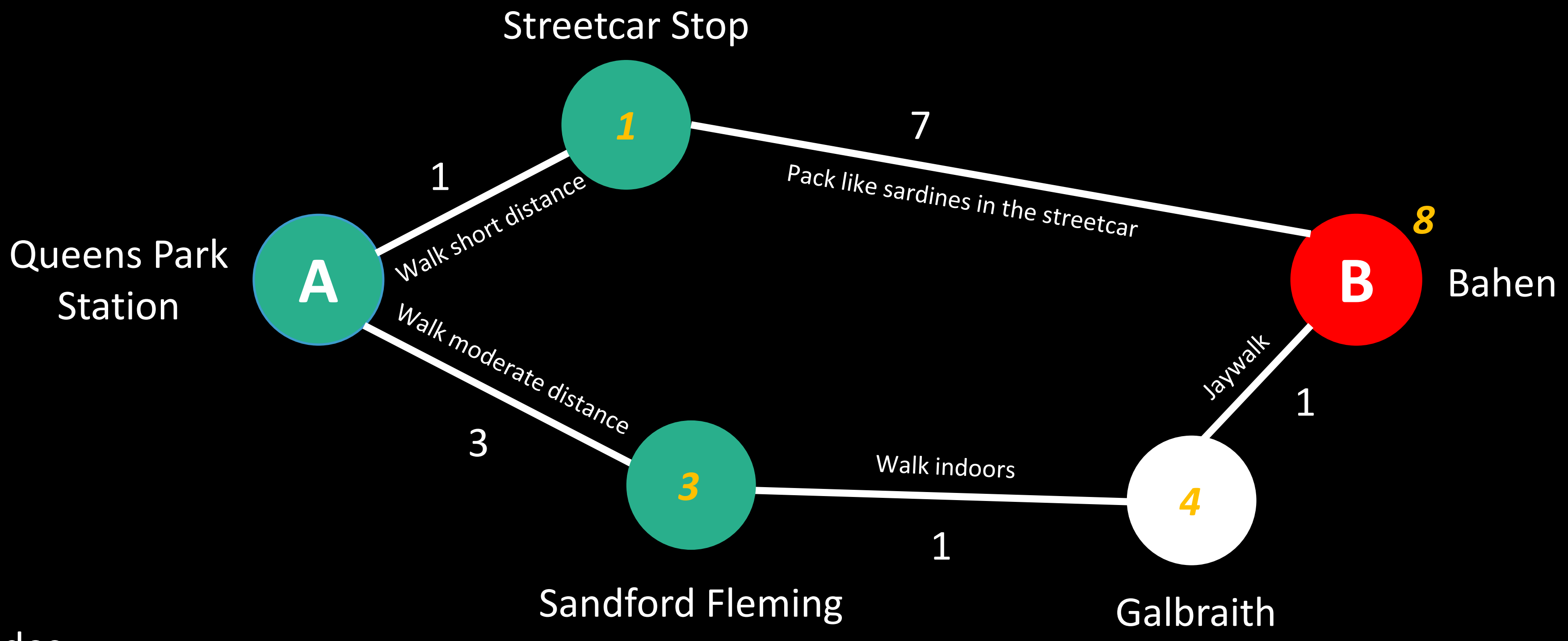


⬜ Unvisited nodes

🟩 Visited nodes

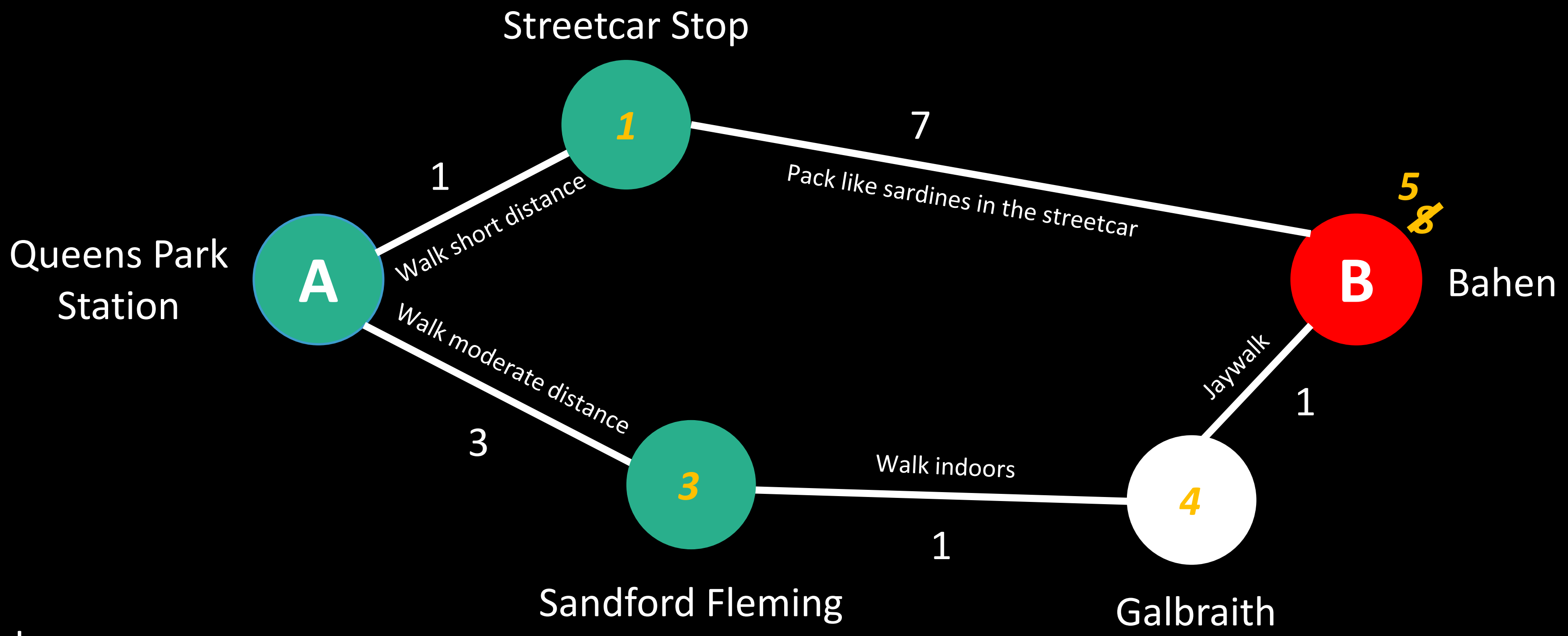
d Distance

Weighted Edges



- ⬜ Unvisited nodes
- 🟩 Visited nodes
- *d* Distance

Weighted Edges

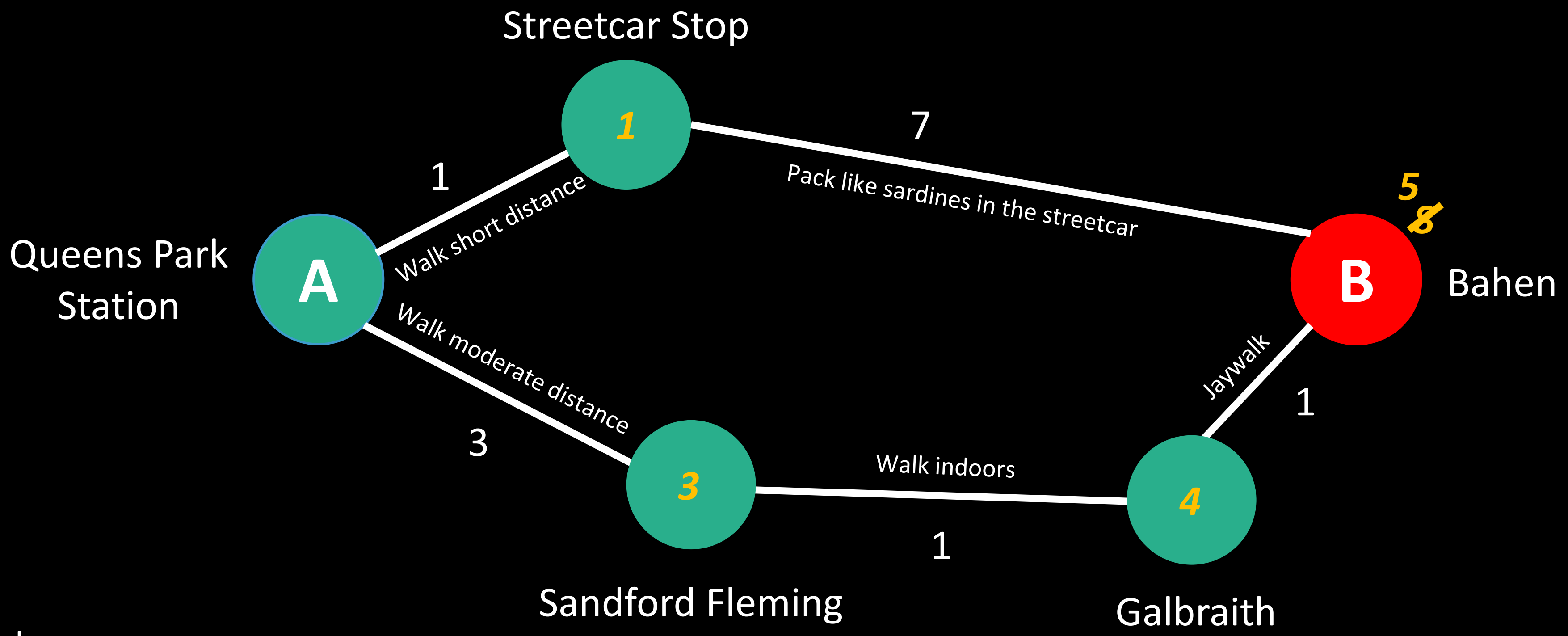


⬜ Unvisited nodes

🟩 Visited nodes

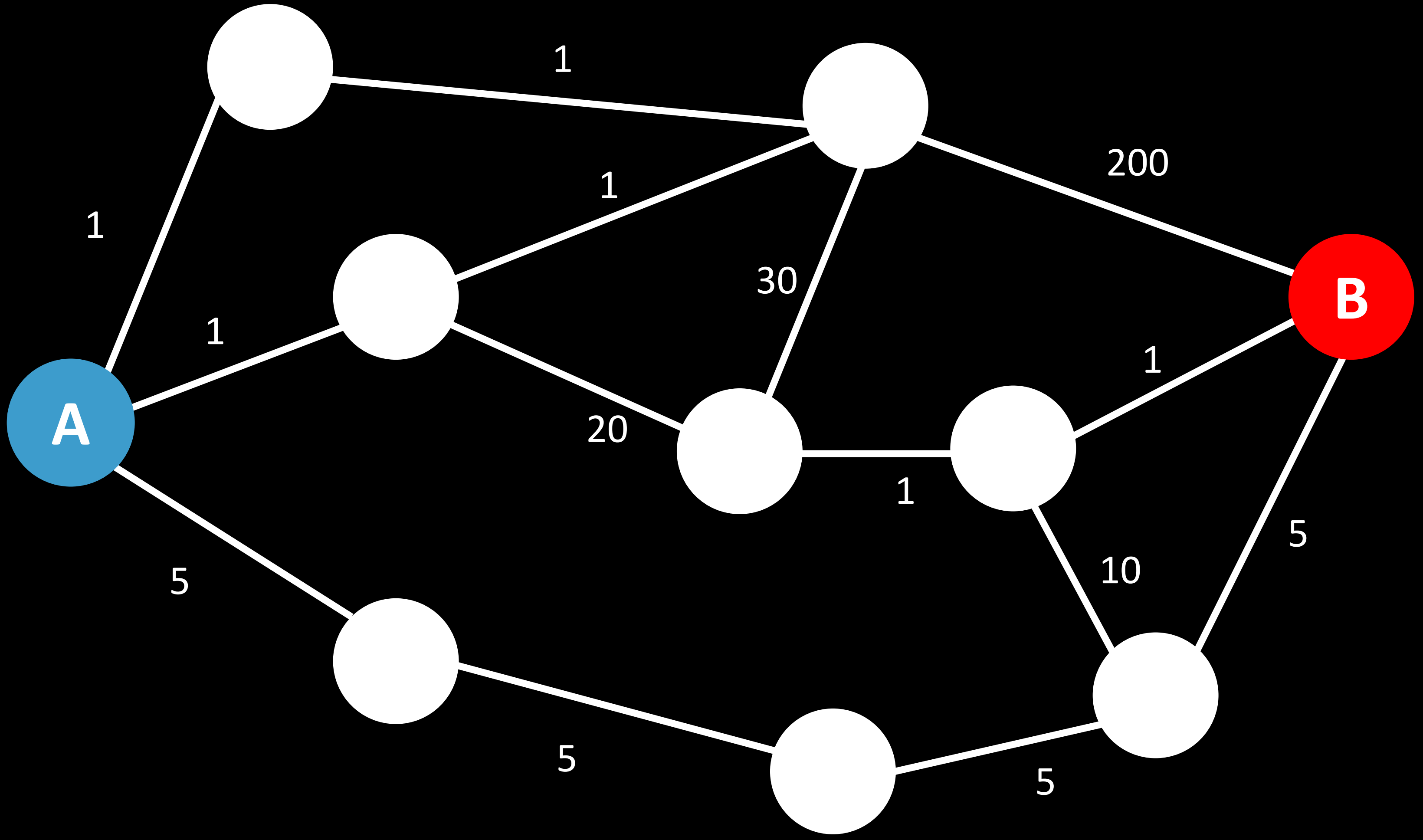
d Distance

Weighted Edges

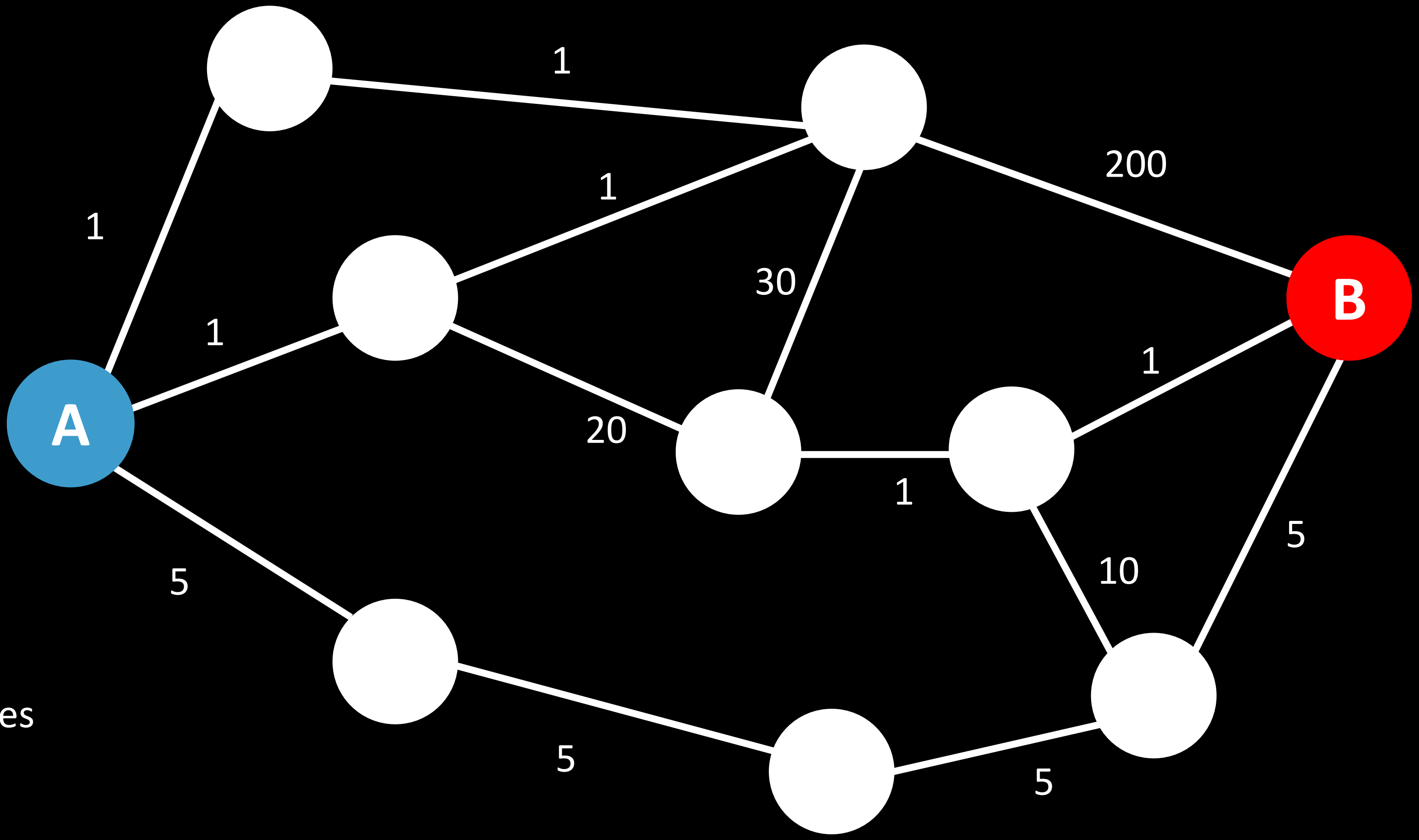


- ⬜ Unvisited nodes
- 🟩 Visited nodes
- *d* Distance

A more illustrative example...



A more illustrative example…

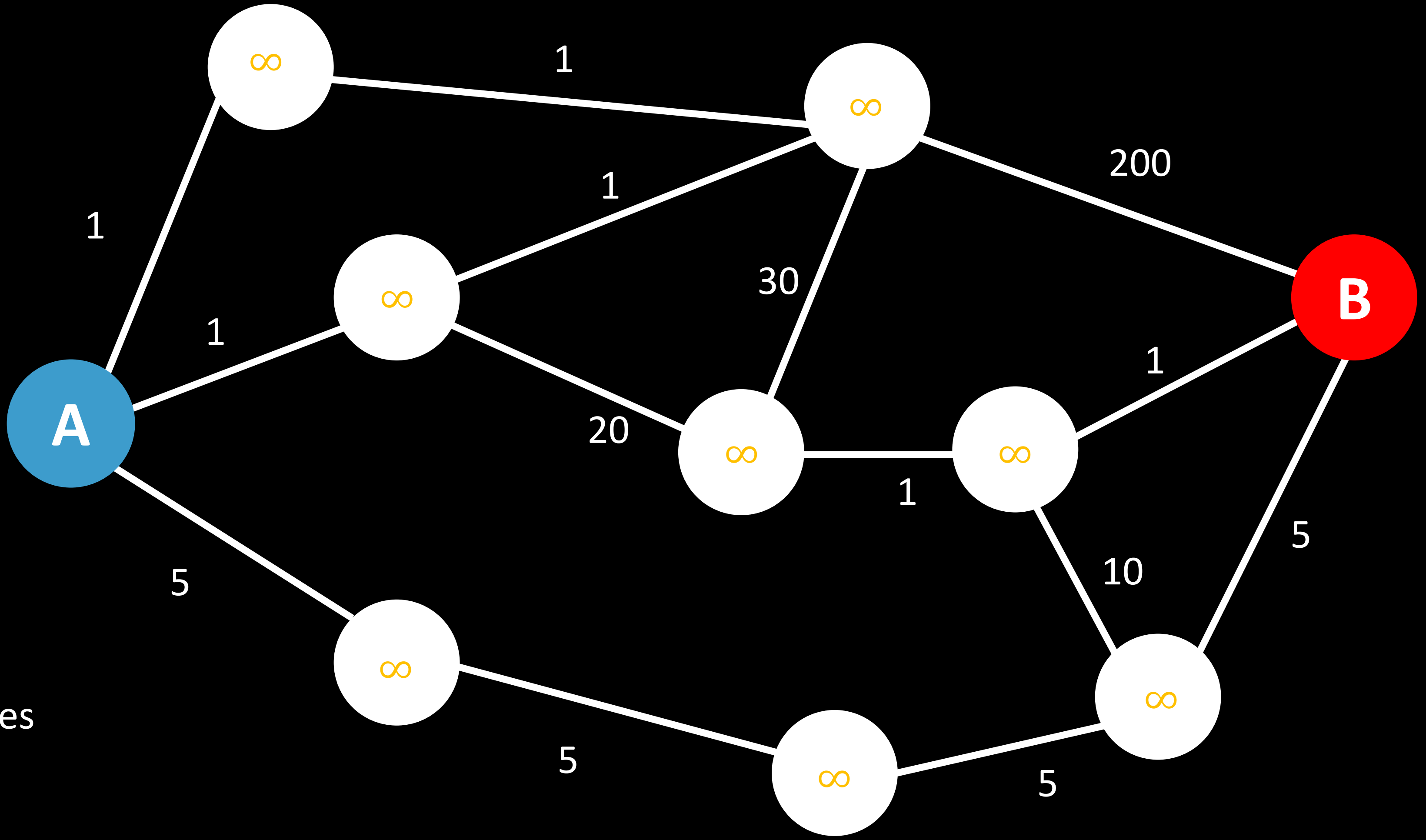


⬜ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

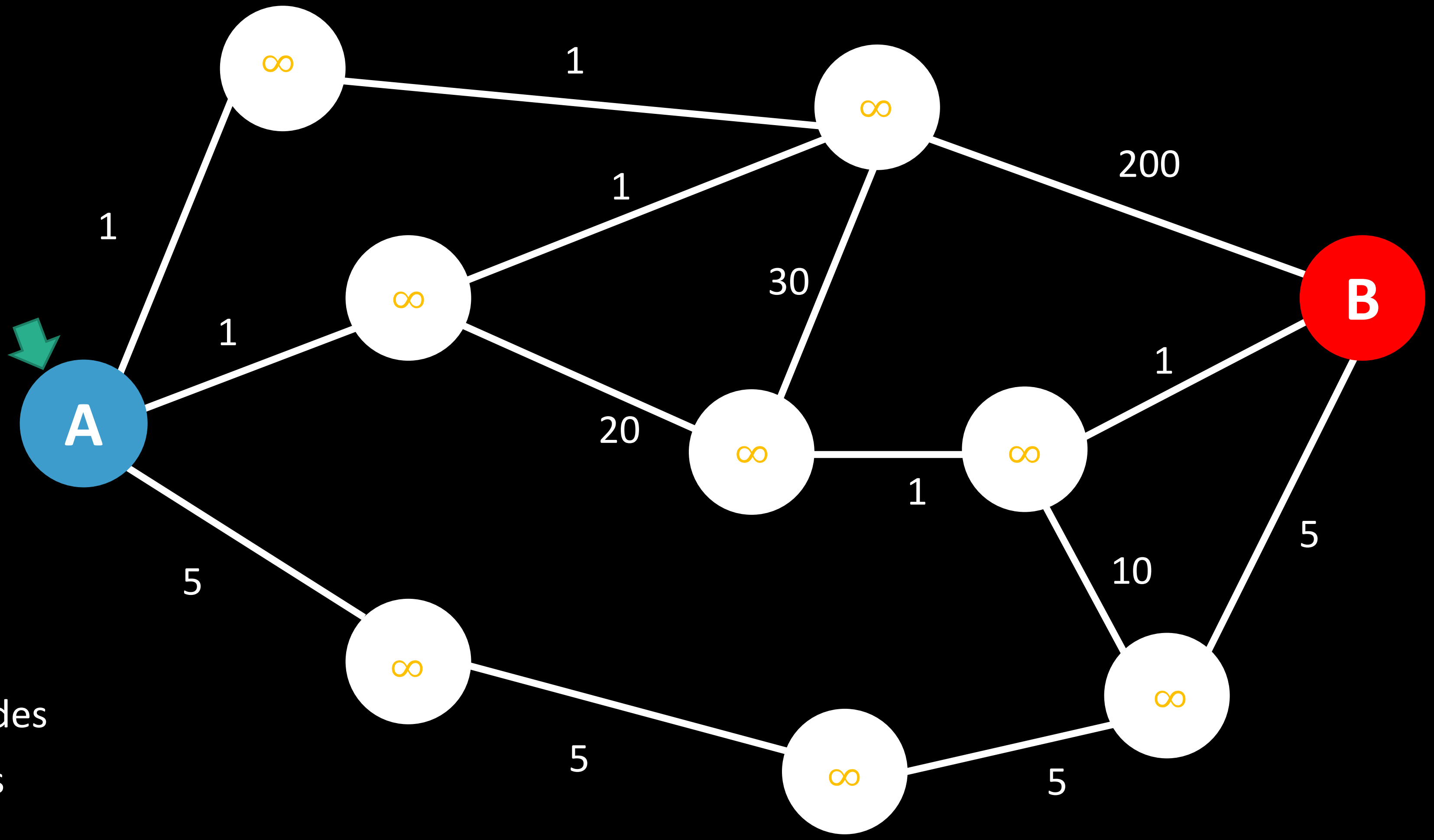


⬜ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

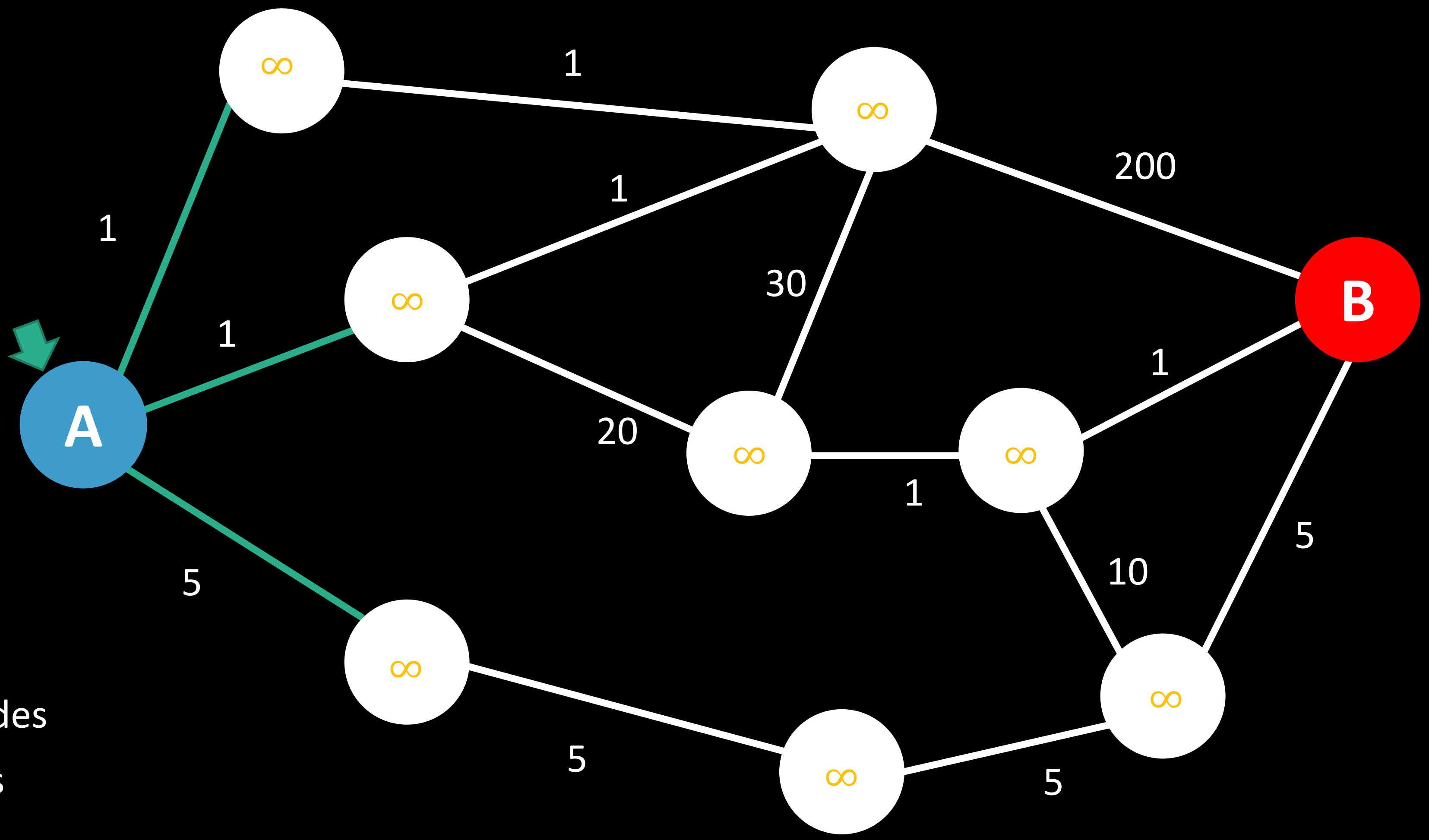


⬜ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

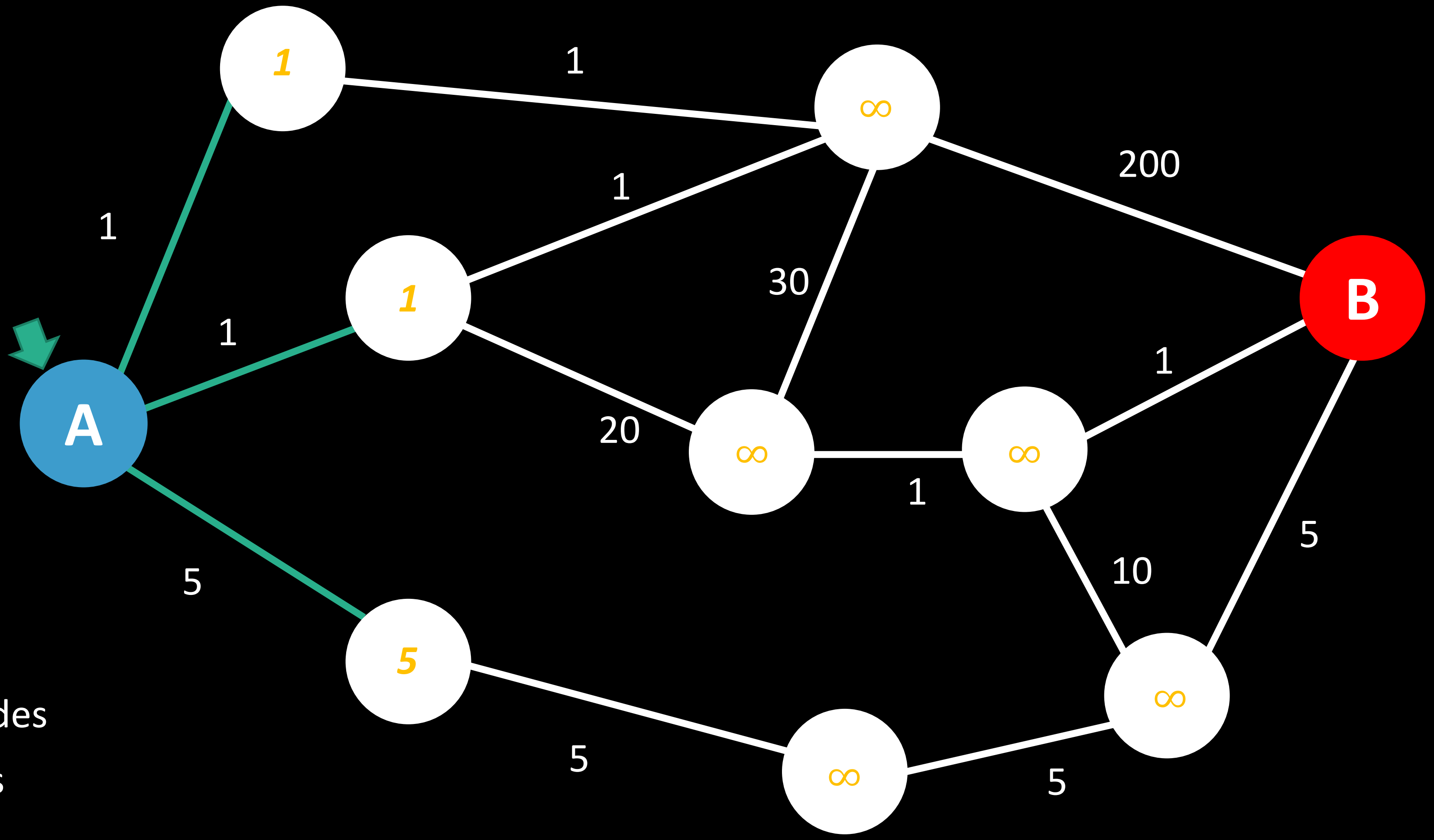


■ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

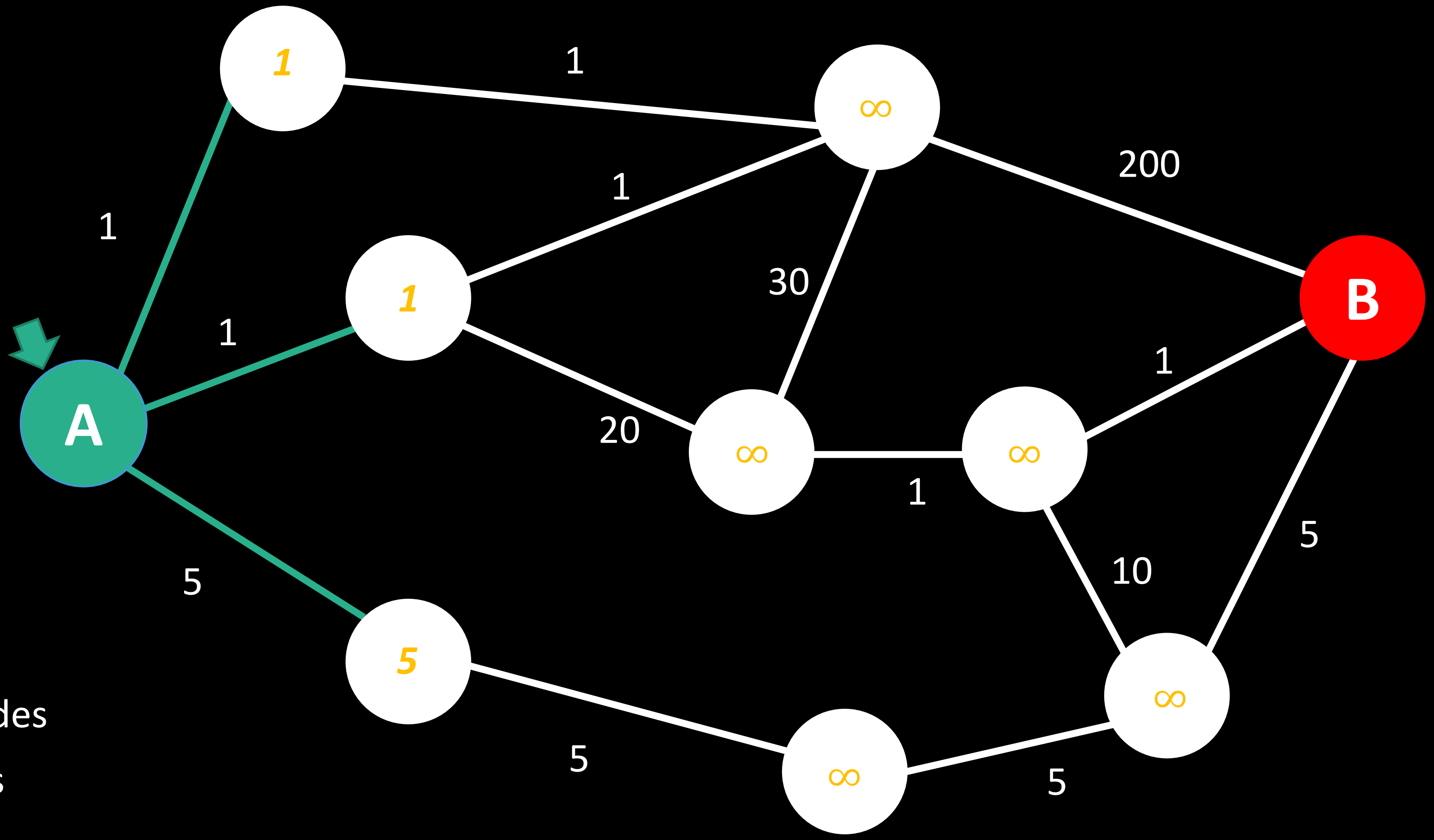


■ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example...

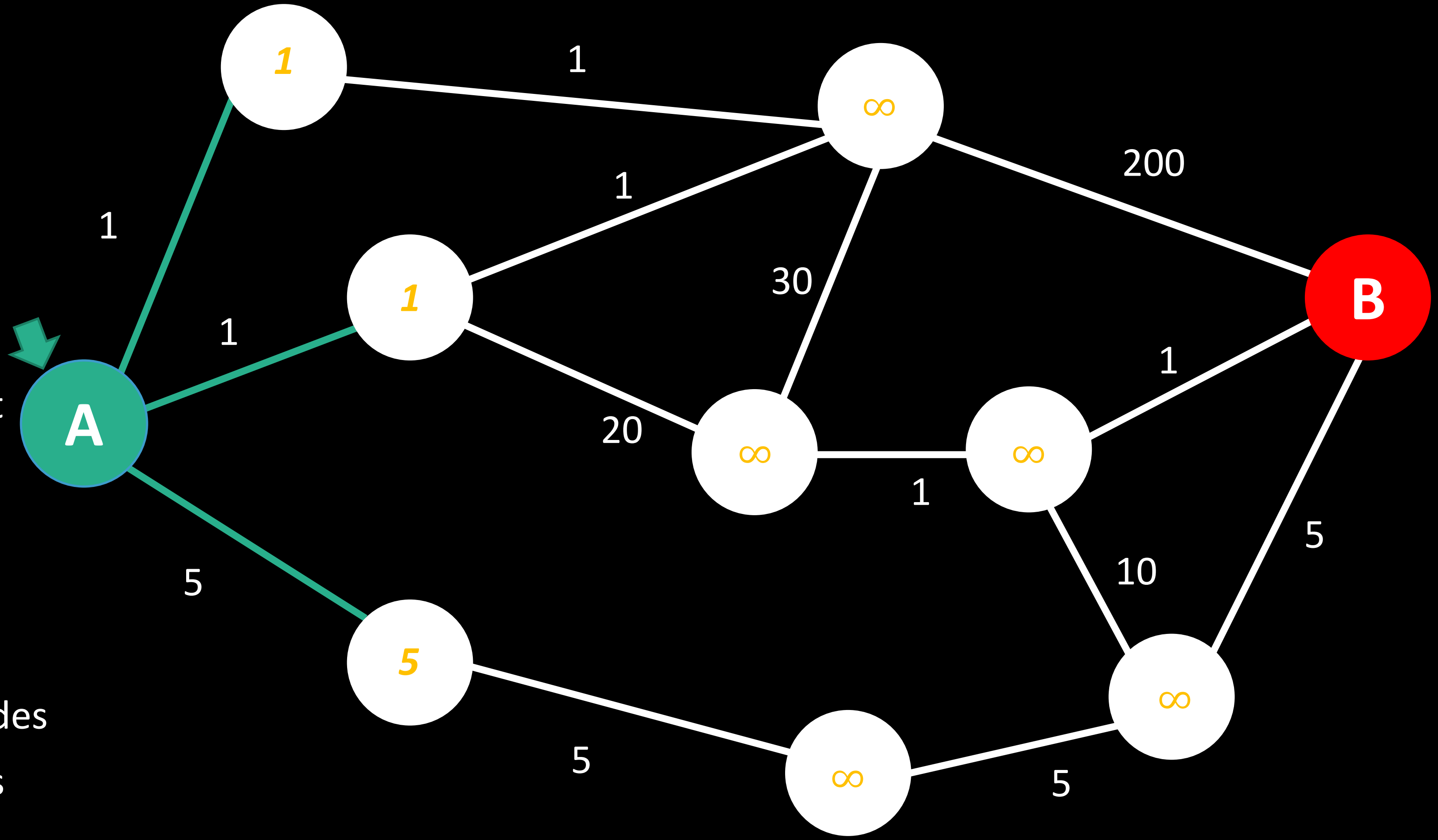


⬜ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example...



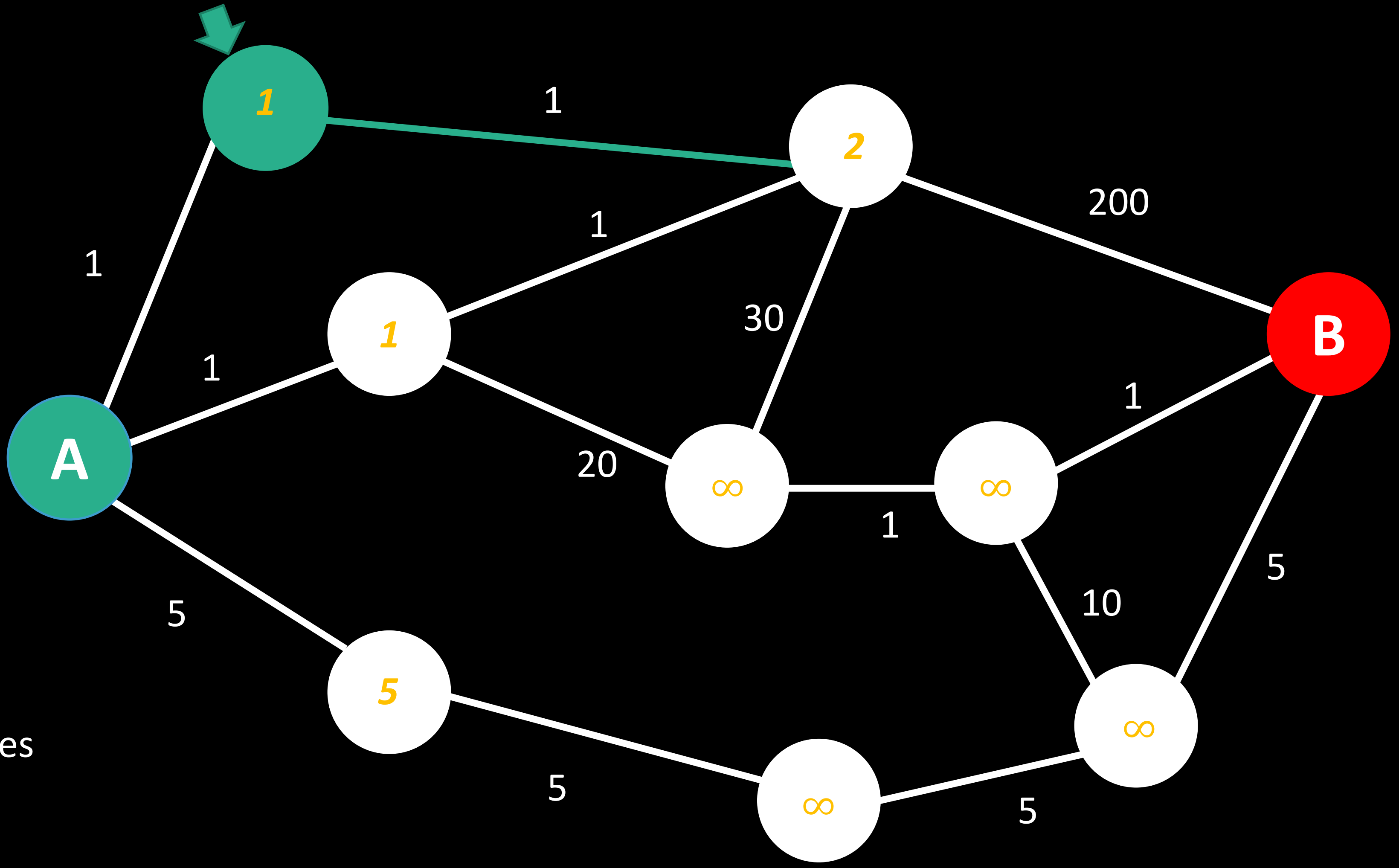
Minimal cost path is 1

⬜ Unvisited nodes

🟩 Visited nodes

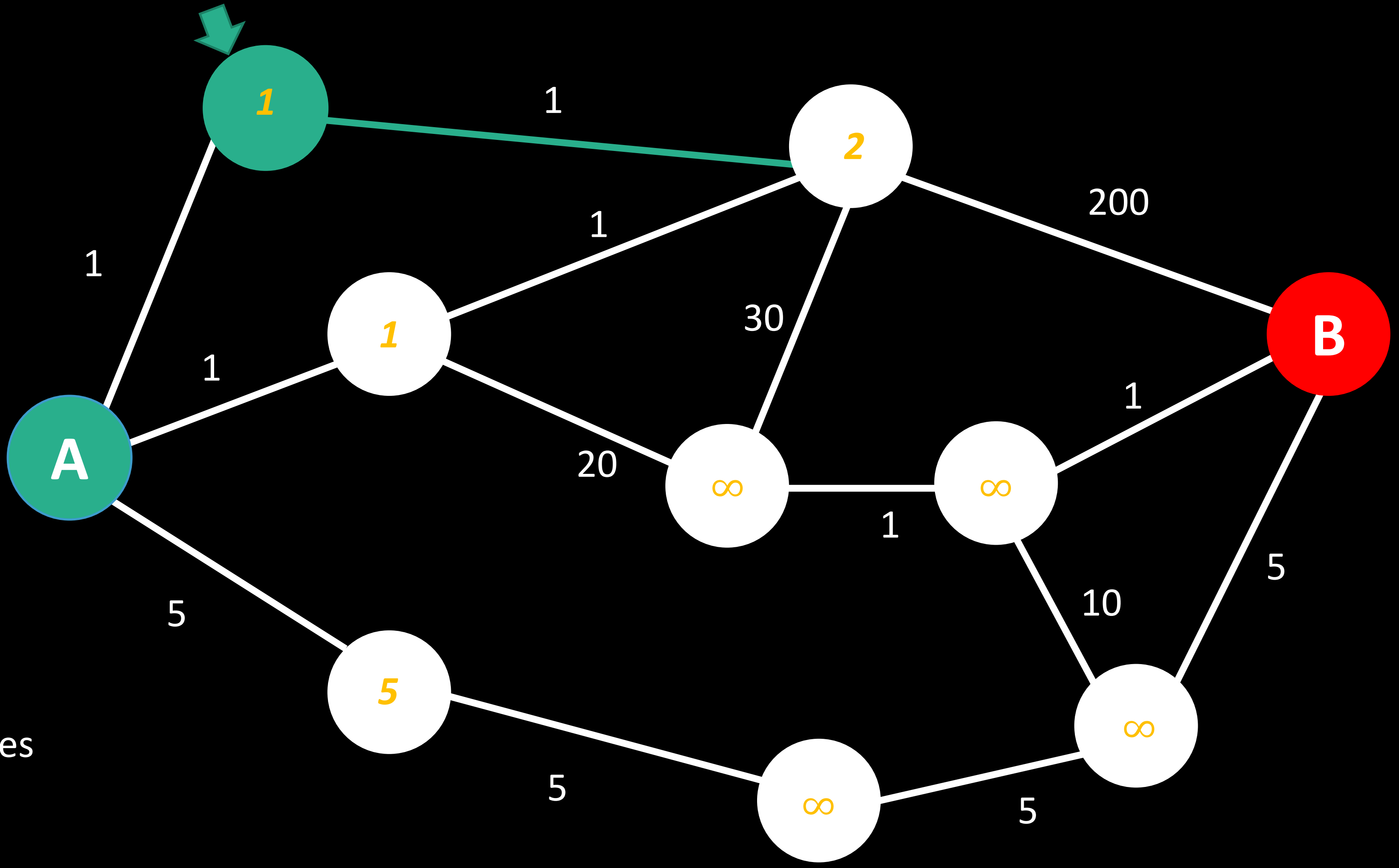
d Distance

A more illustrative example...



- ■ Unvisited nodes
- 🟩 Visited nodes
- *d* Distance

A more illustrative example…



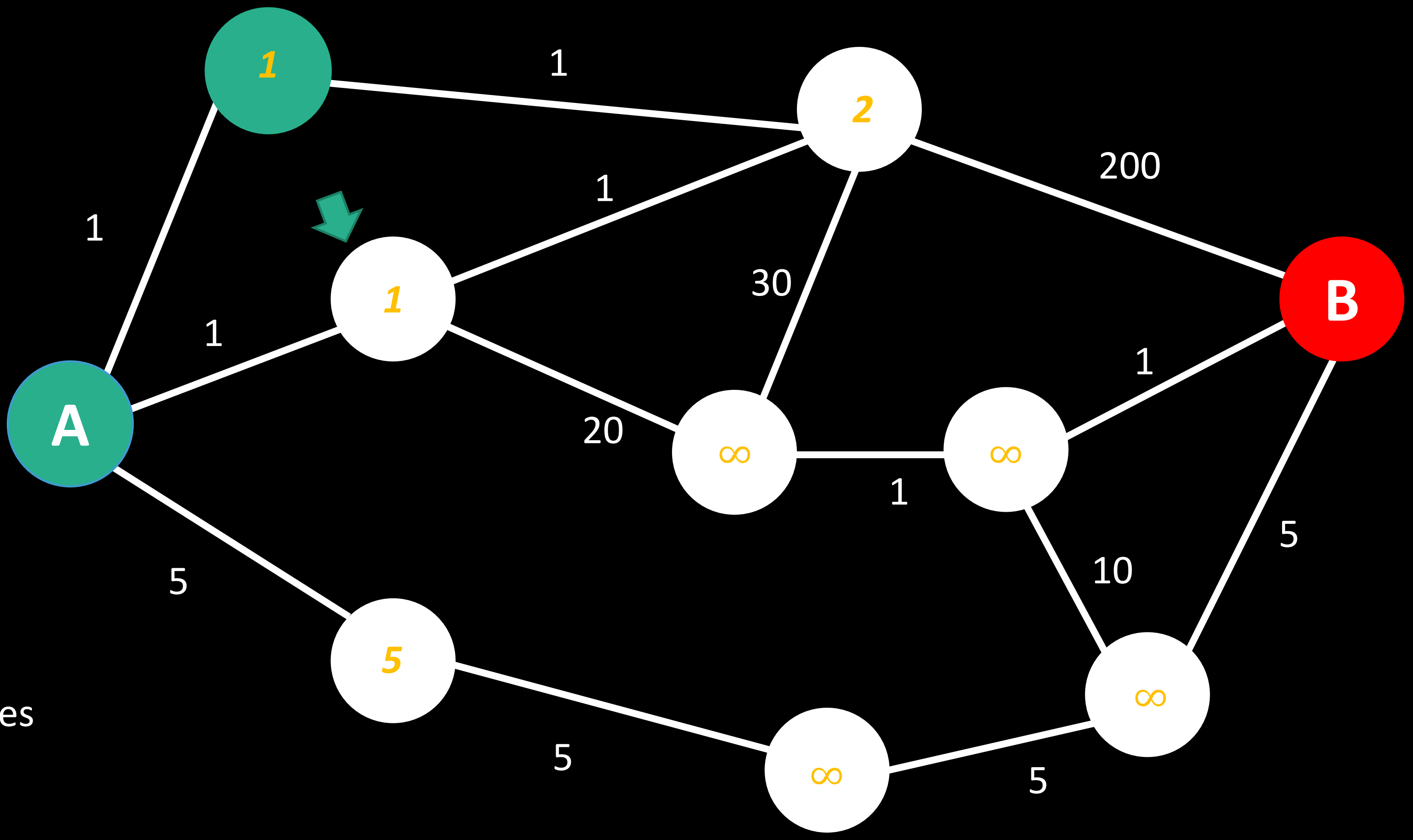
Minimal cost path is 1

■ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…



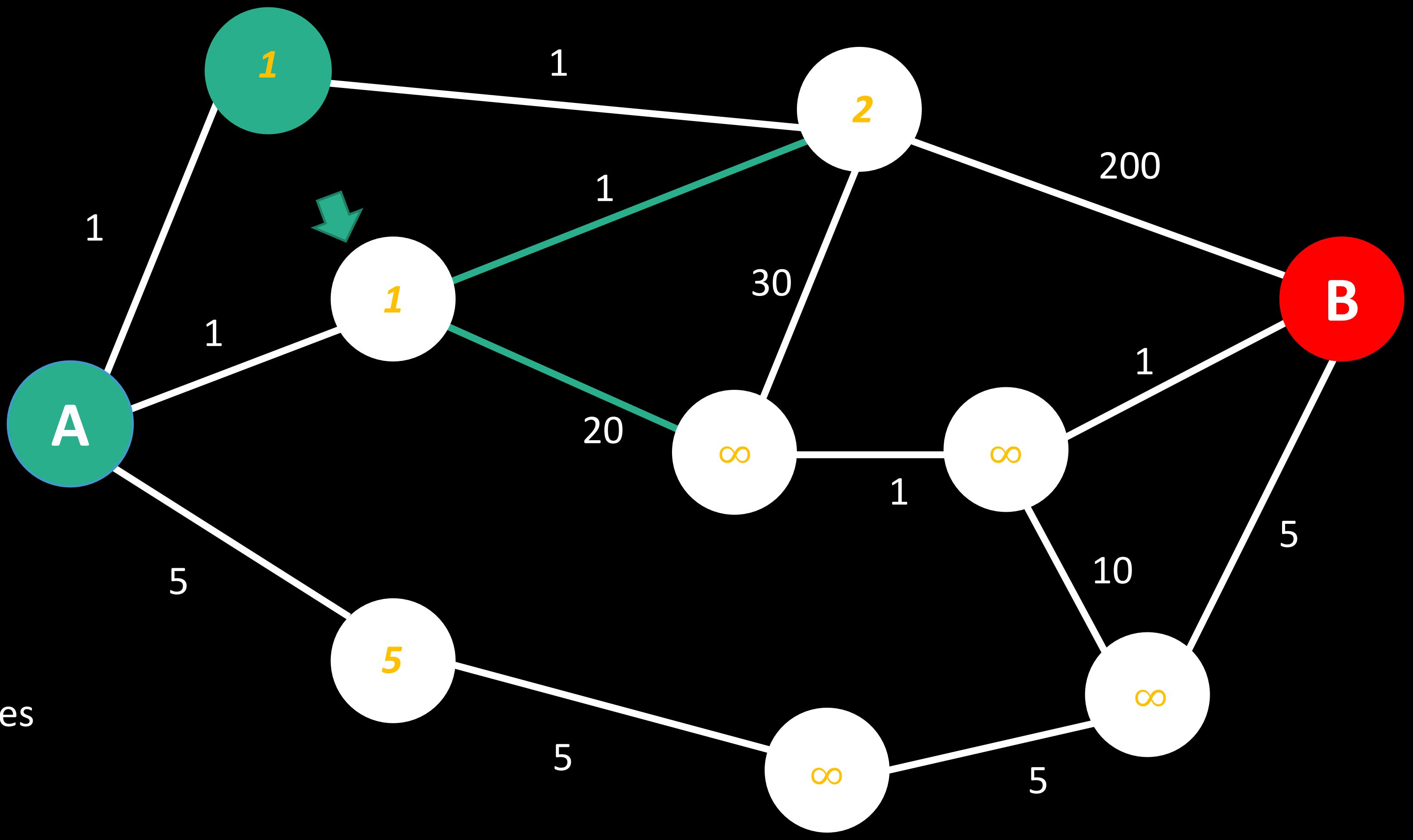
Minimal cost path is 1

□ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

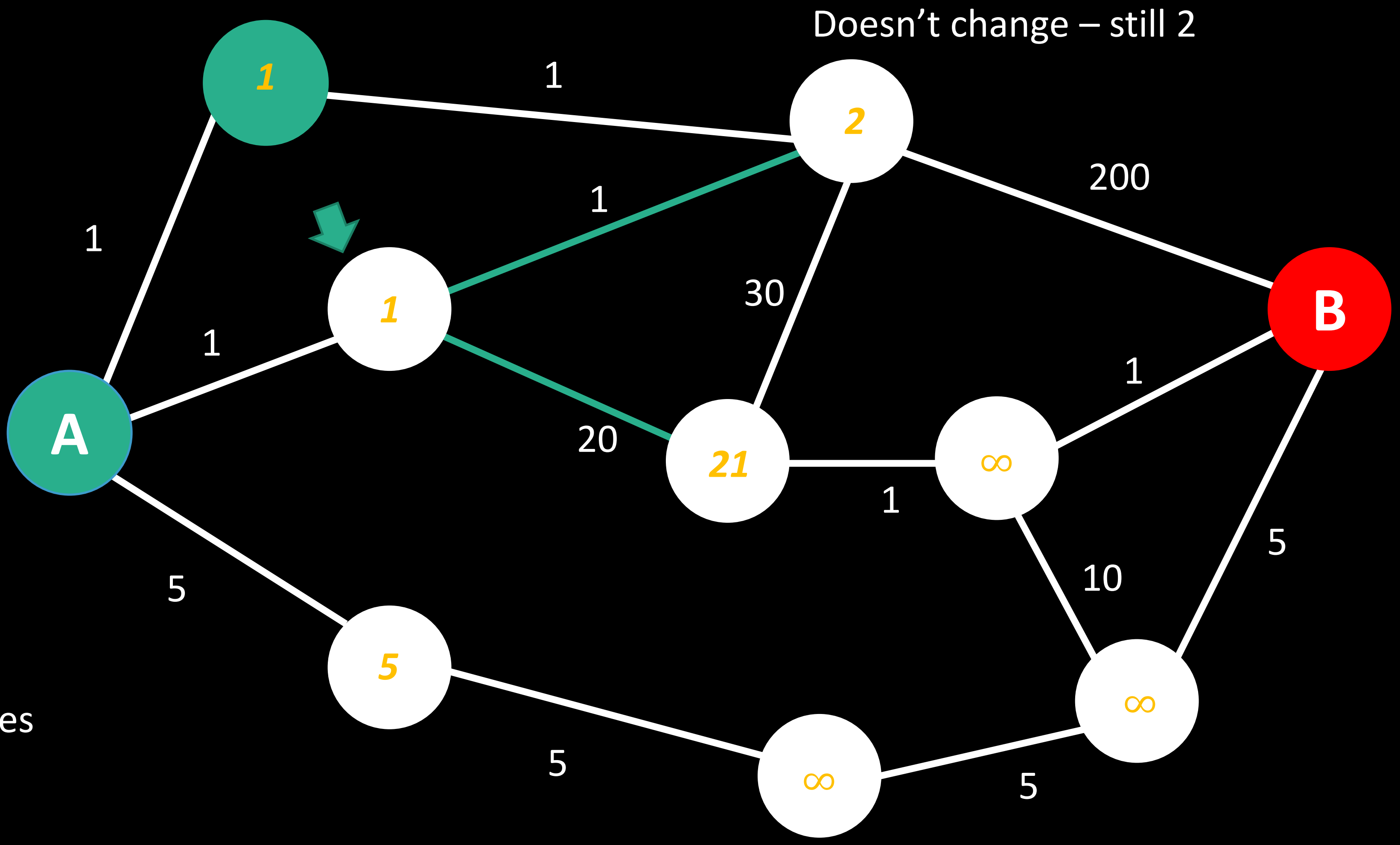


■ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

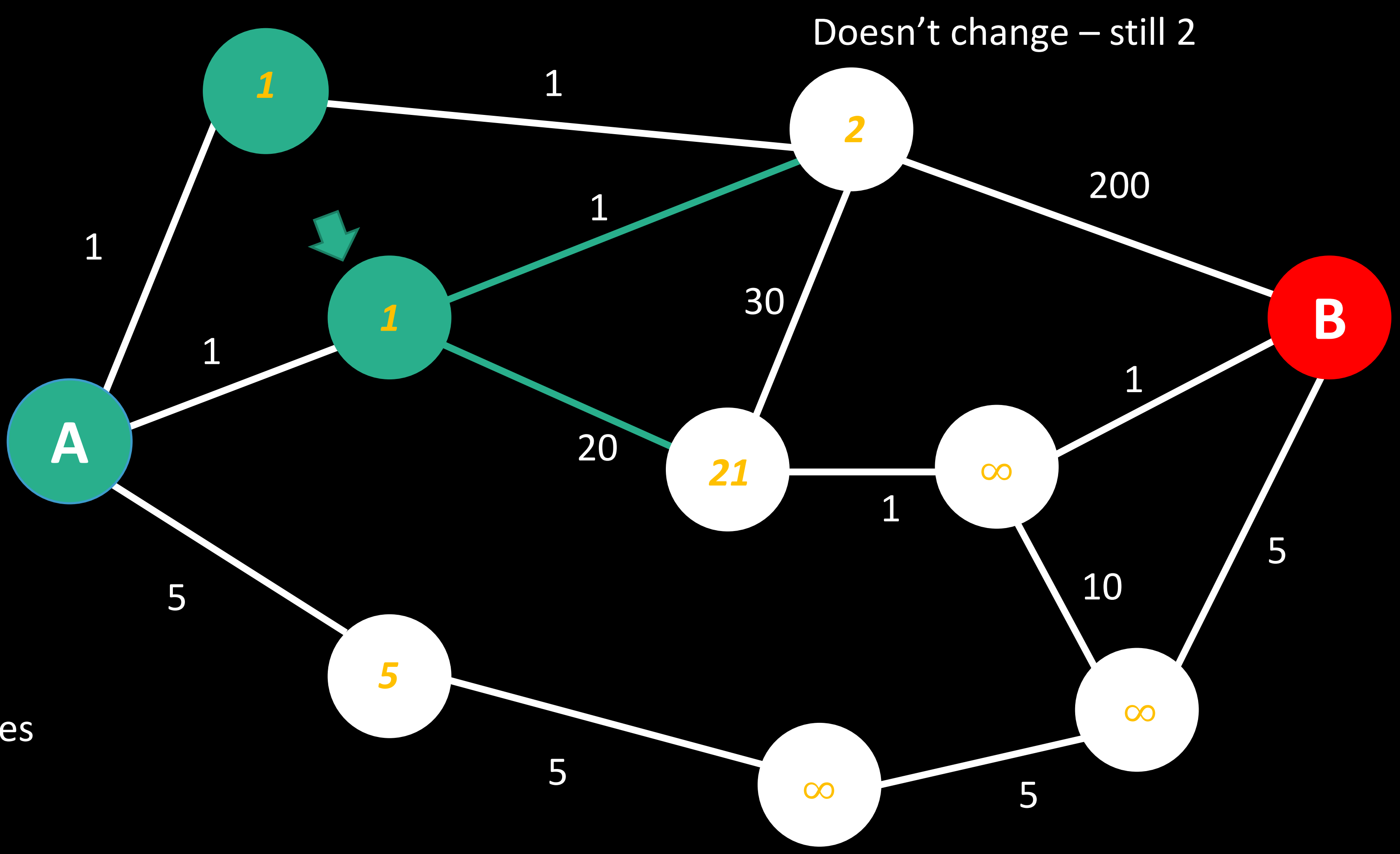


⬜ Unvisited nodes

🟩 Visited nodes

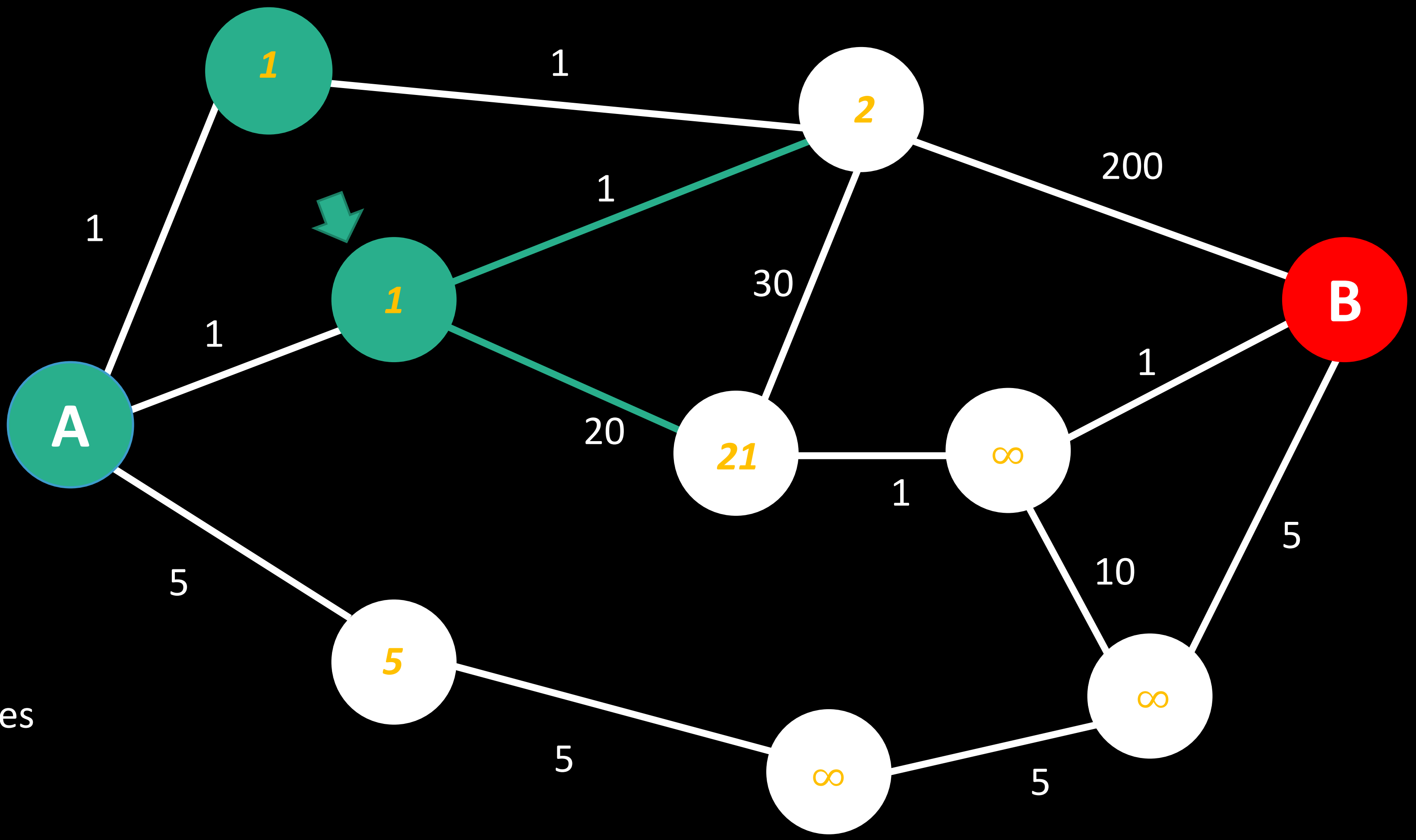
d Distance

A more illustrative example...



- ■ Unvisited nodes
- 🟩 Visited nodes
- *d* Distance

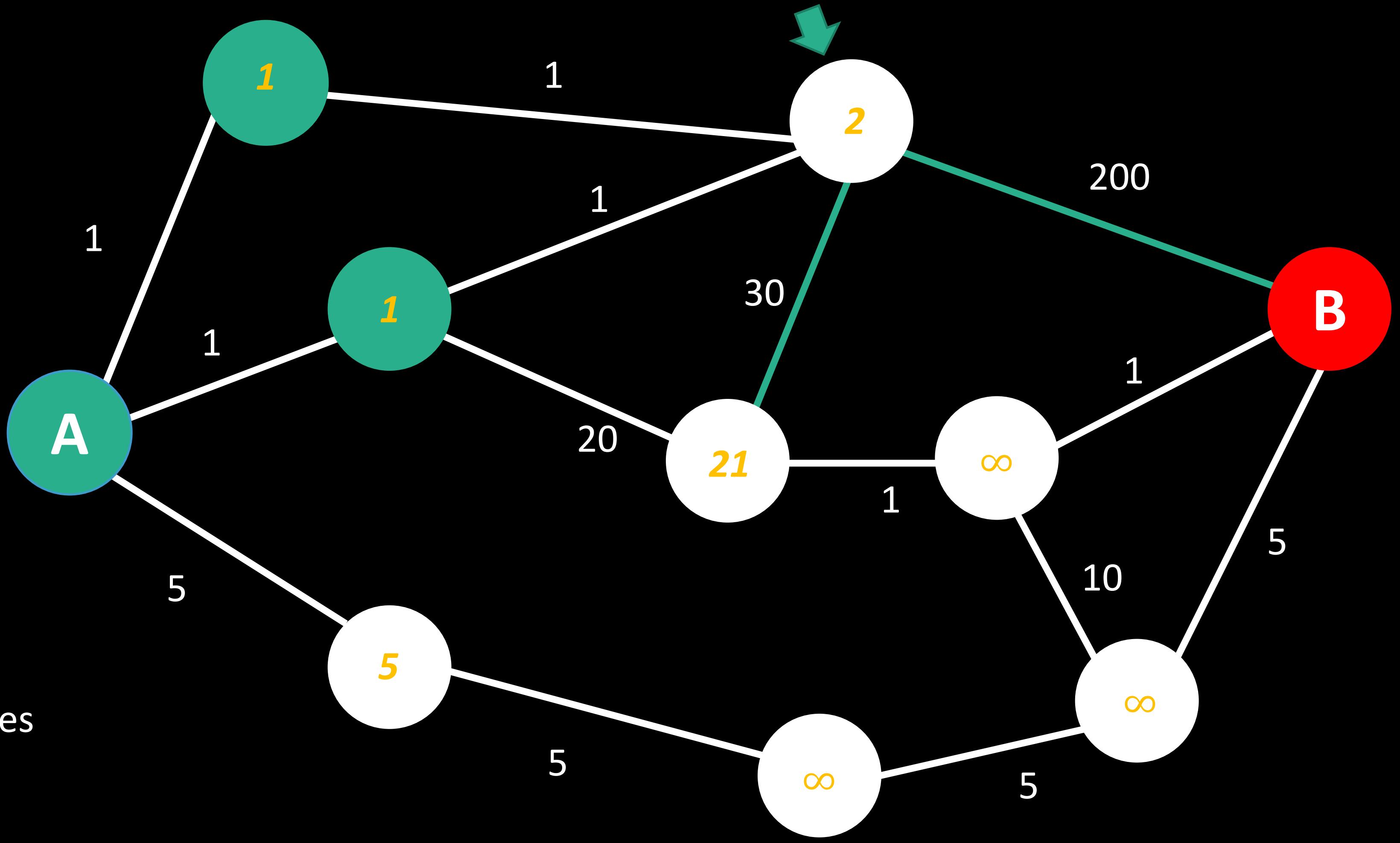
A more illustrative example…



Minimal cost path is 2

- ⬜ Unvisited nodes
- 🟩 Visited nodes
- *d* Distance

A more illustrative example...

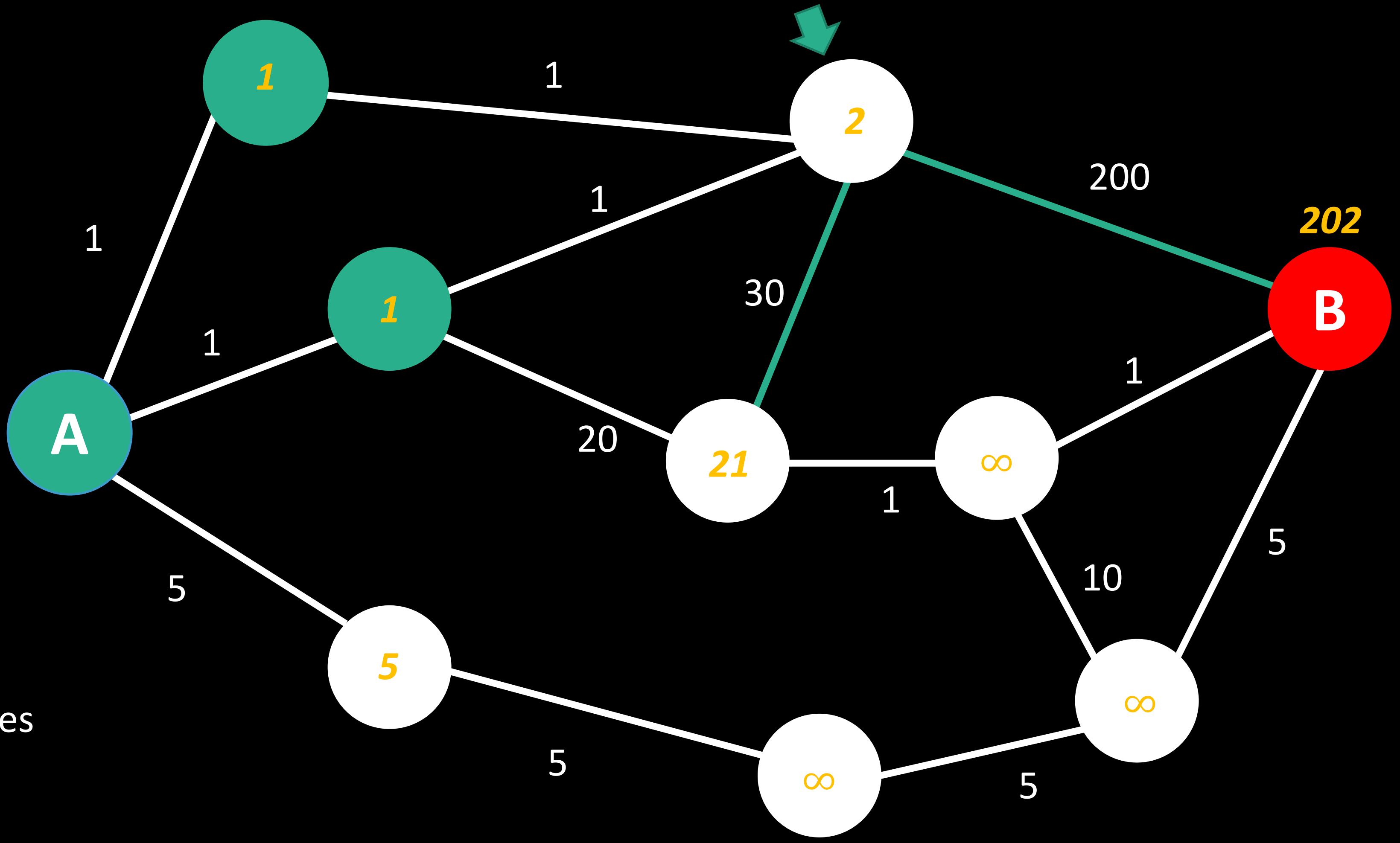


⬜ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

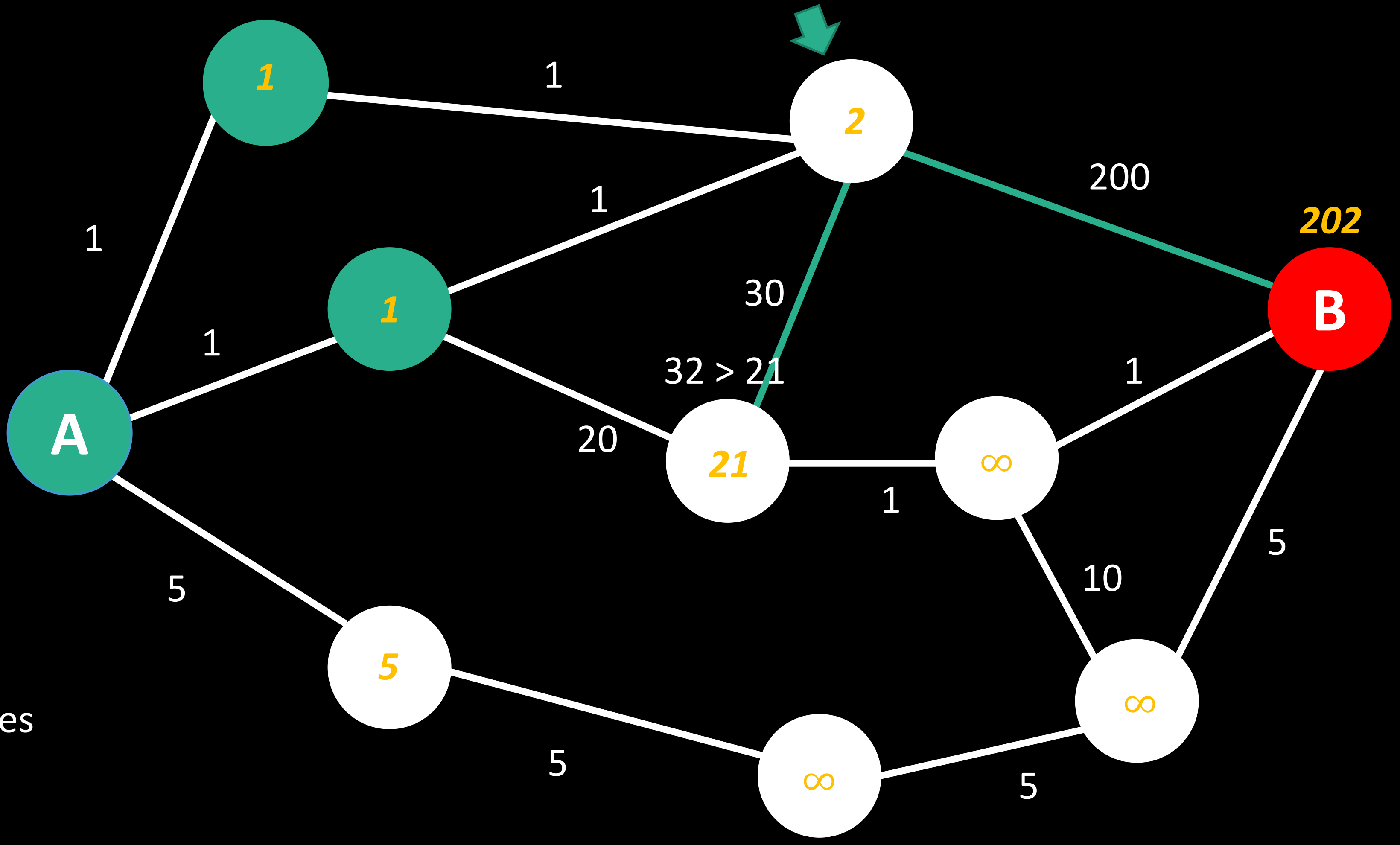


■ Unvisited nodes

🟩 Visited nodes

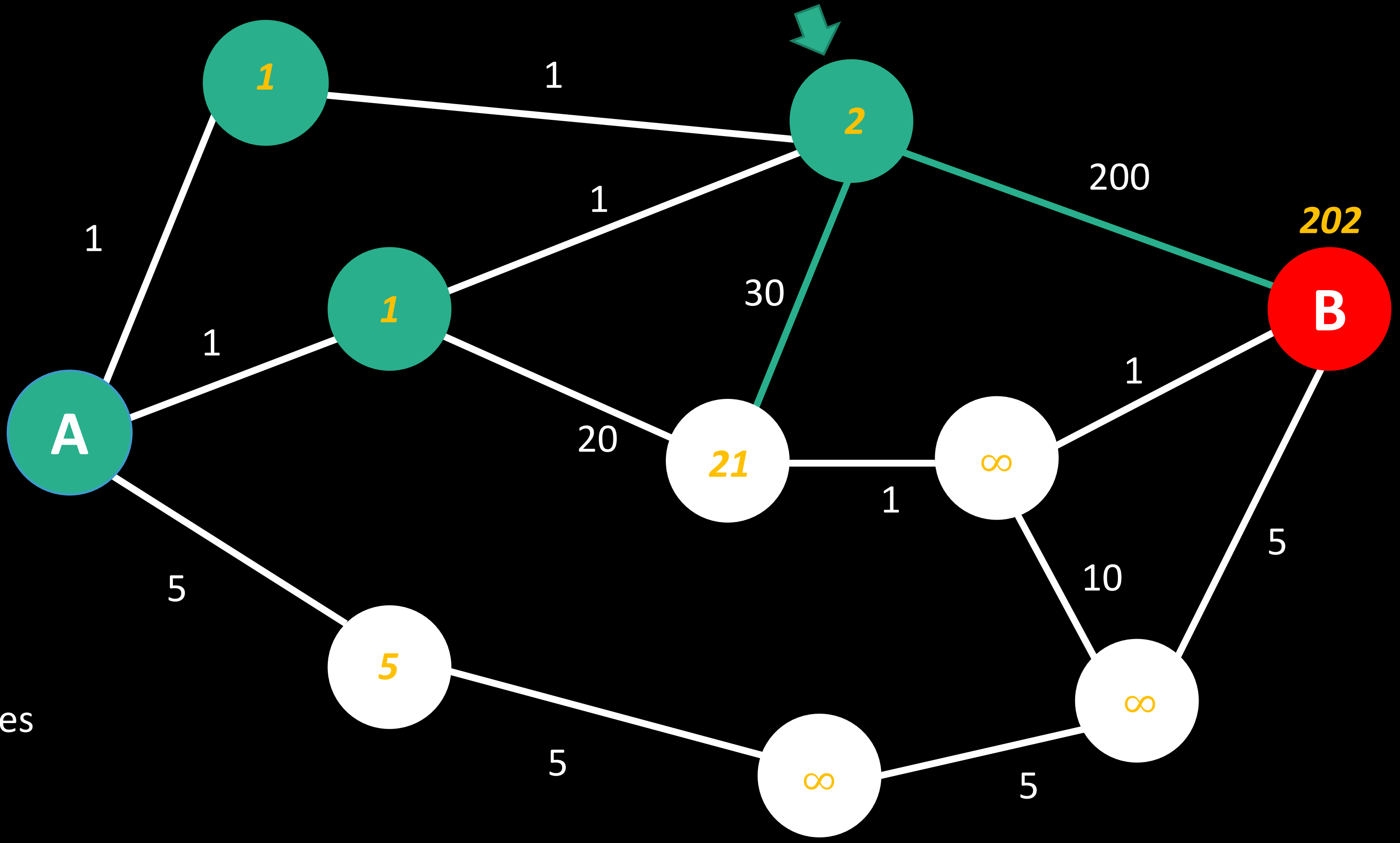
d Distance

A more illustrative example...



- ⬜ Unvisited nodes
- 🟩 Visited nodes
- *d* Distance

A more illustrative example…

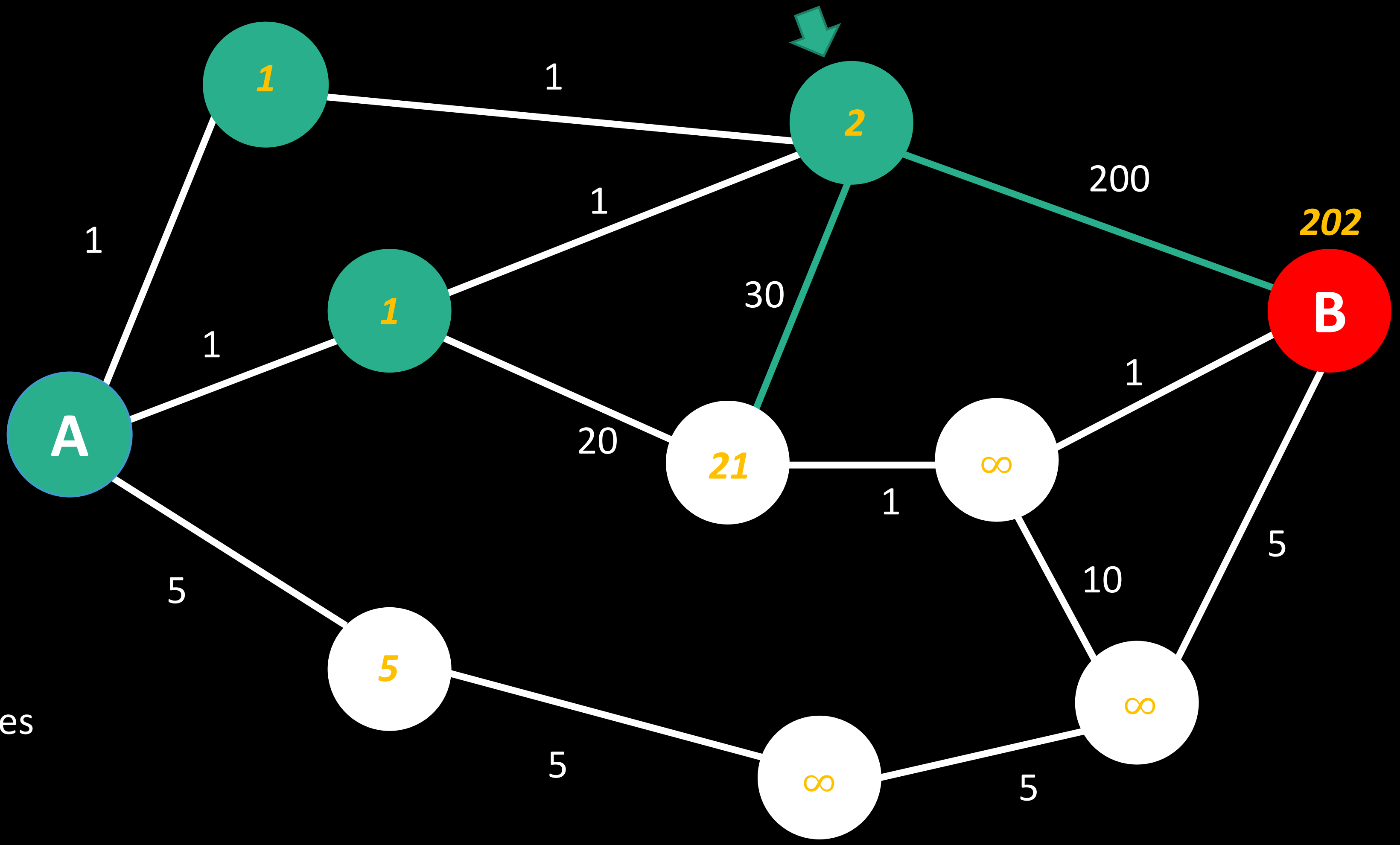


■ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…



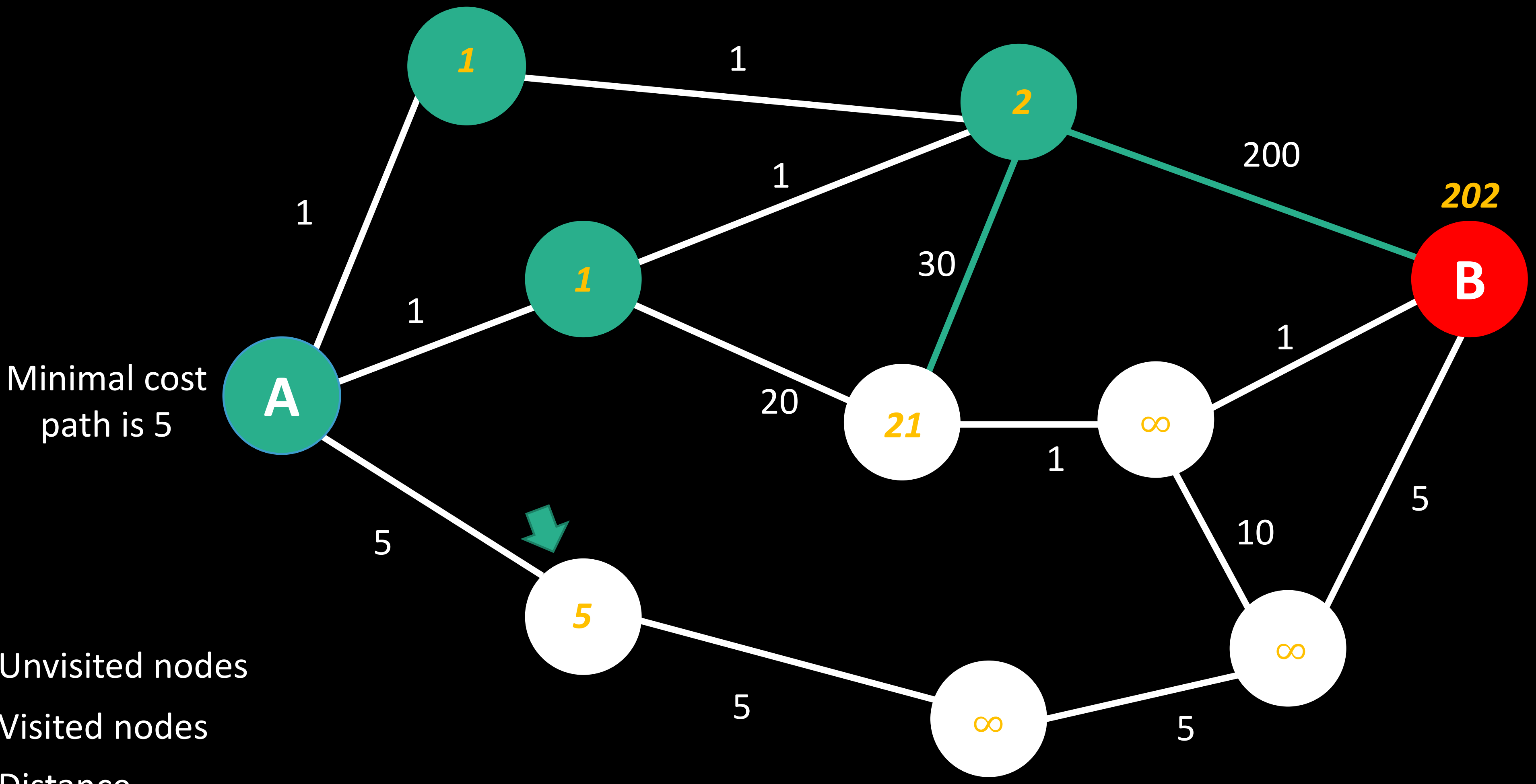
Minimal cost
path is 5

⬜ Unvisited nodes

🟩 Visited nodes

d Distance

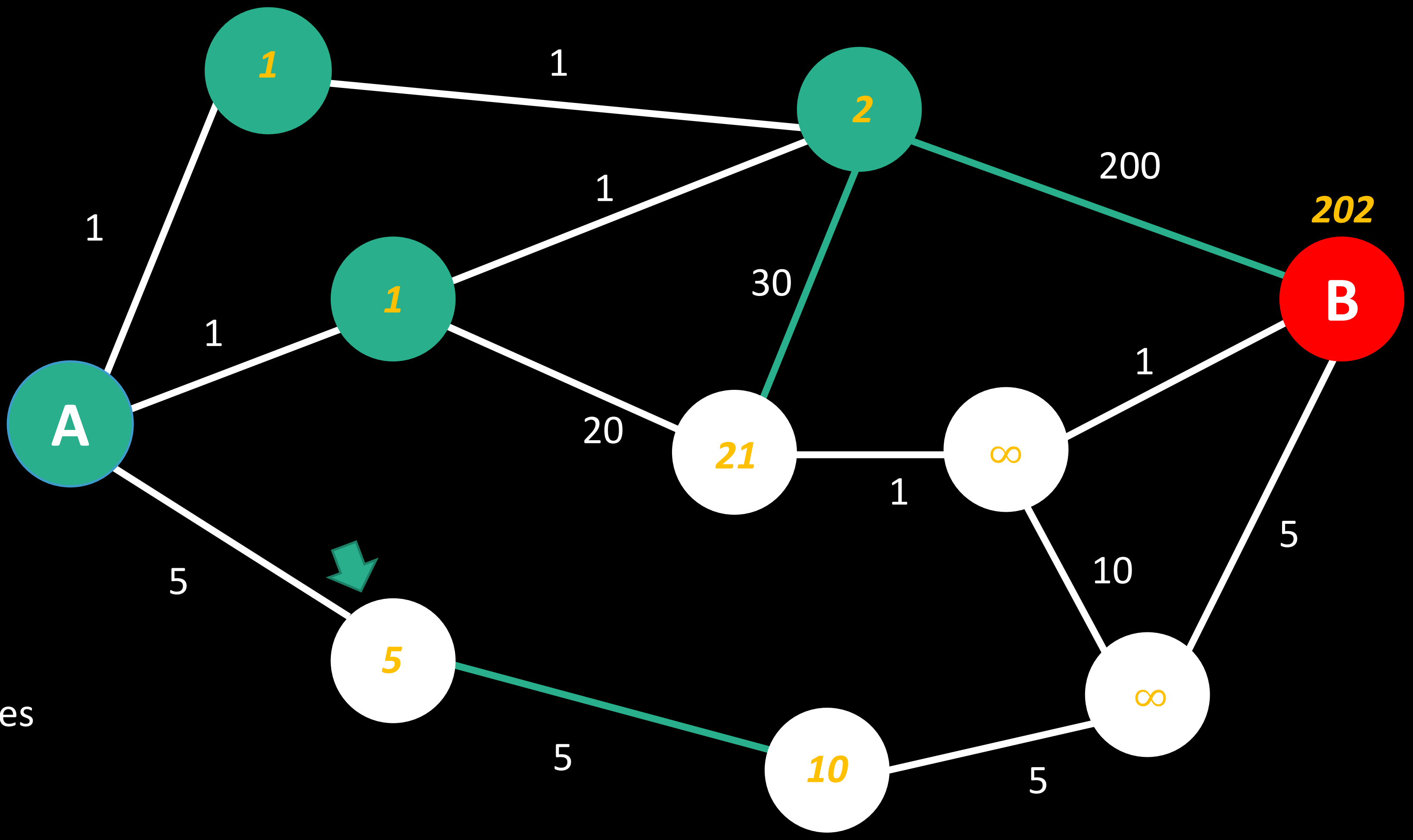
A more illustrative example…



Minimal cost path is 5

- ⬜ Unvisited nodes
- 🟩 Visited nodes
- *d* Distance

A more illustrative example…

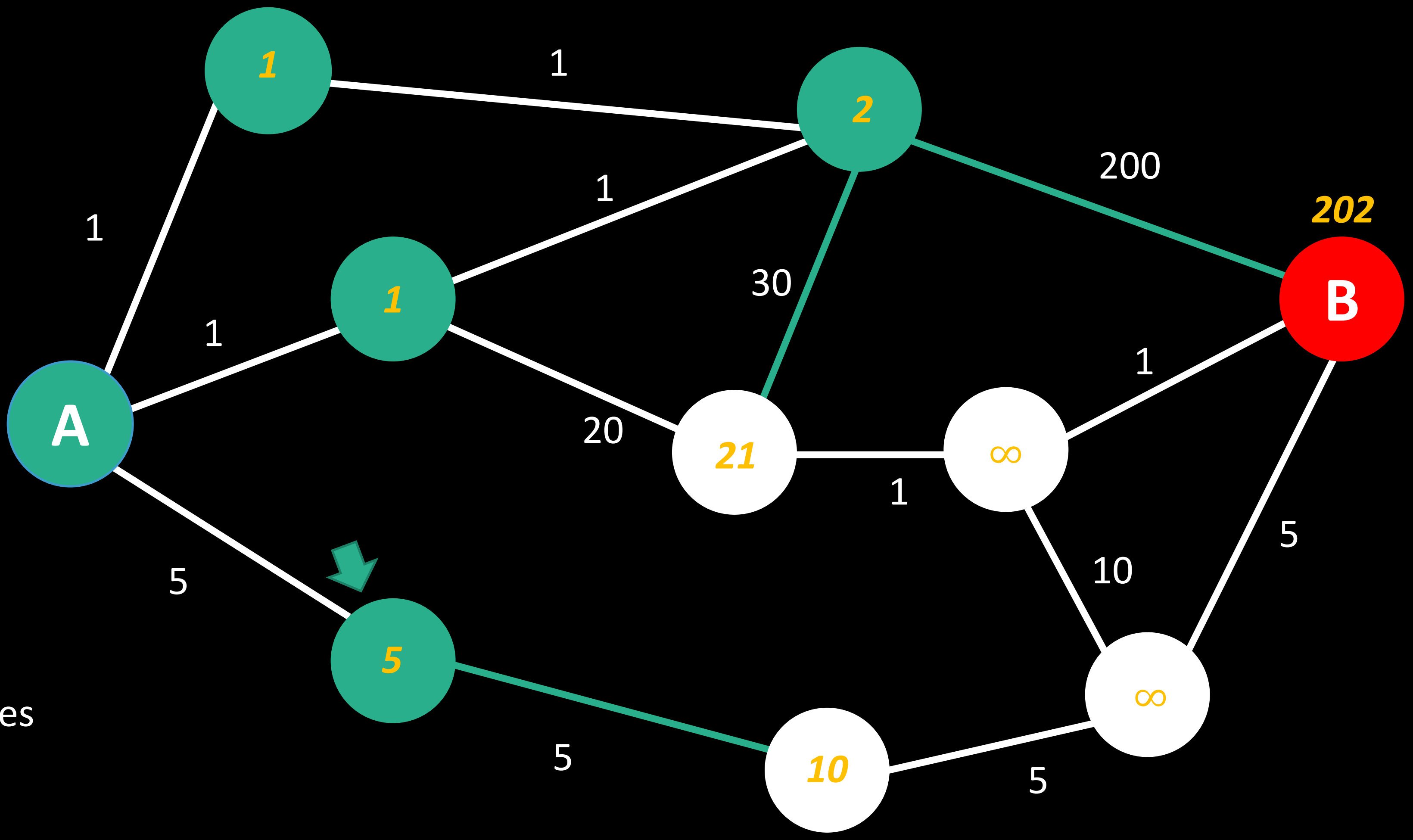


■ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

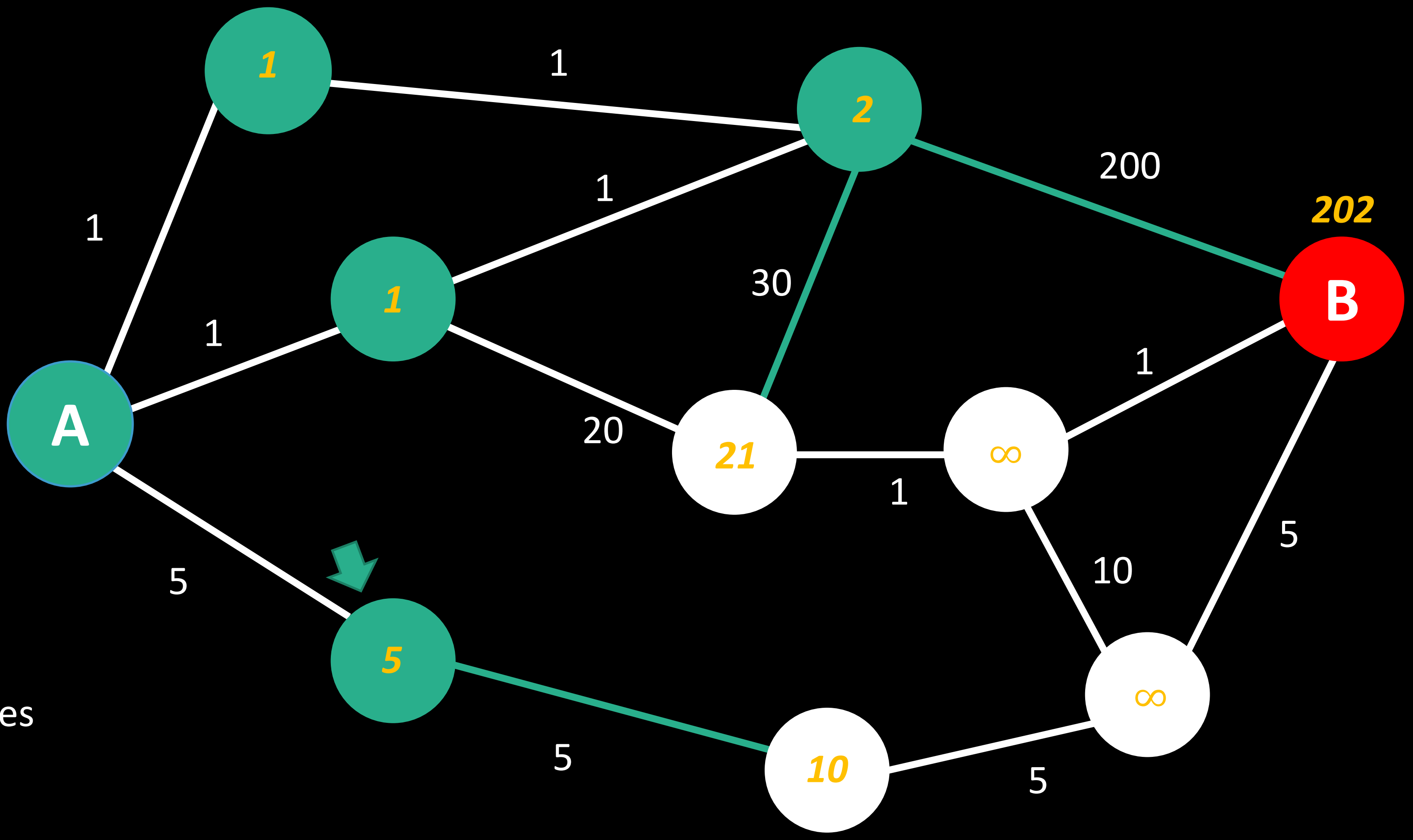


■ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…



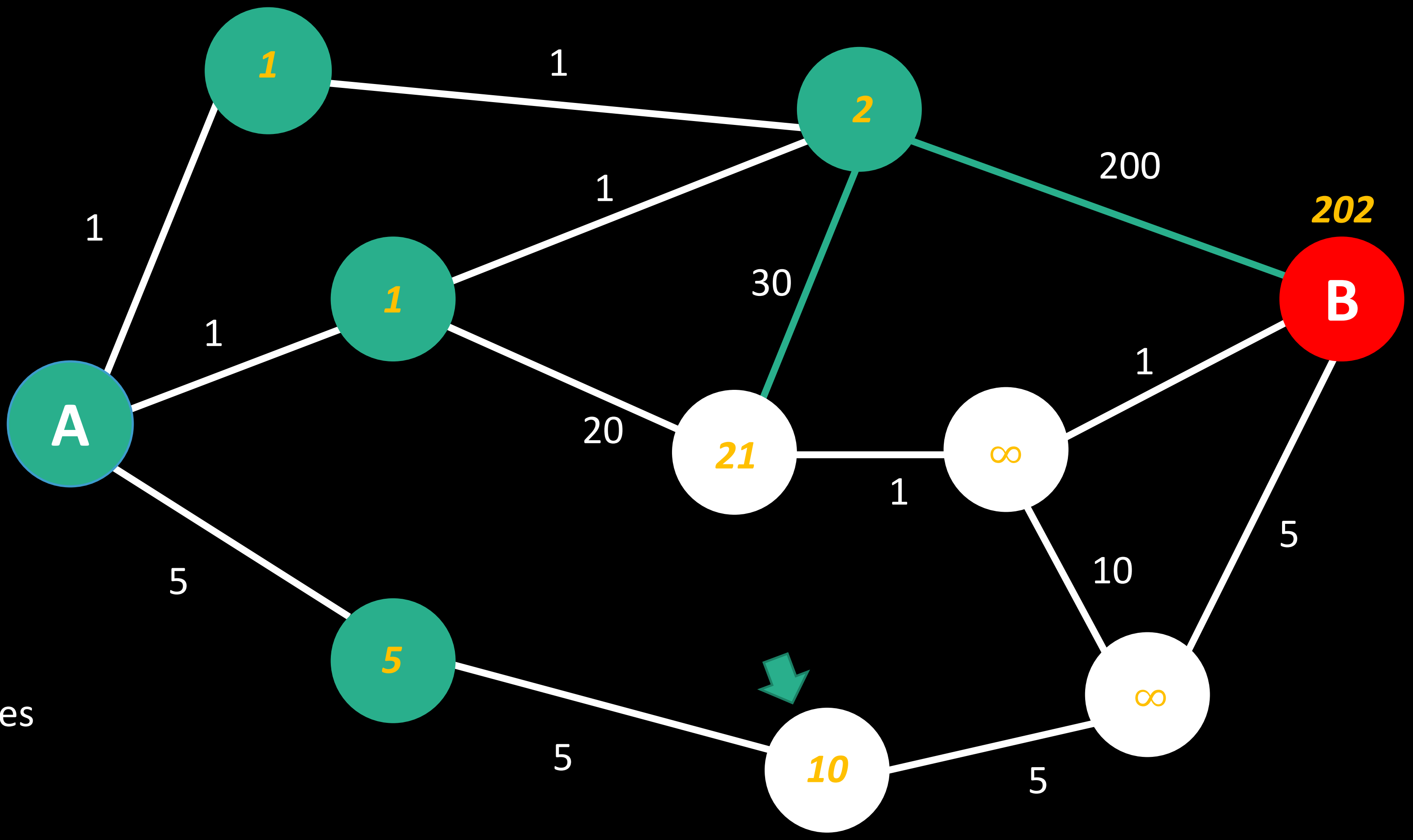
Minimal cost path is 10

■ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…



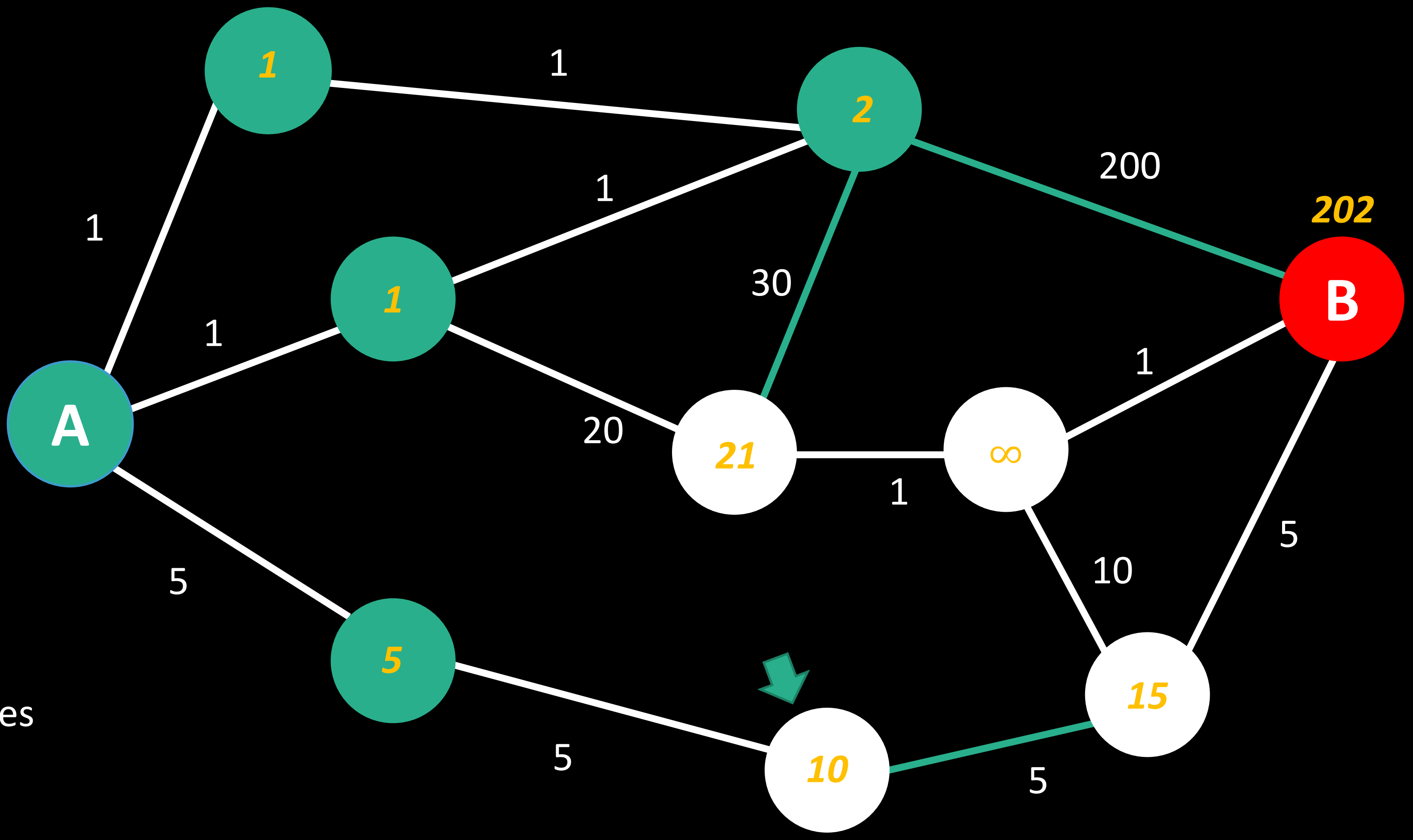
Minimal cost path is 10

◻ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

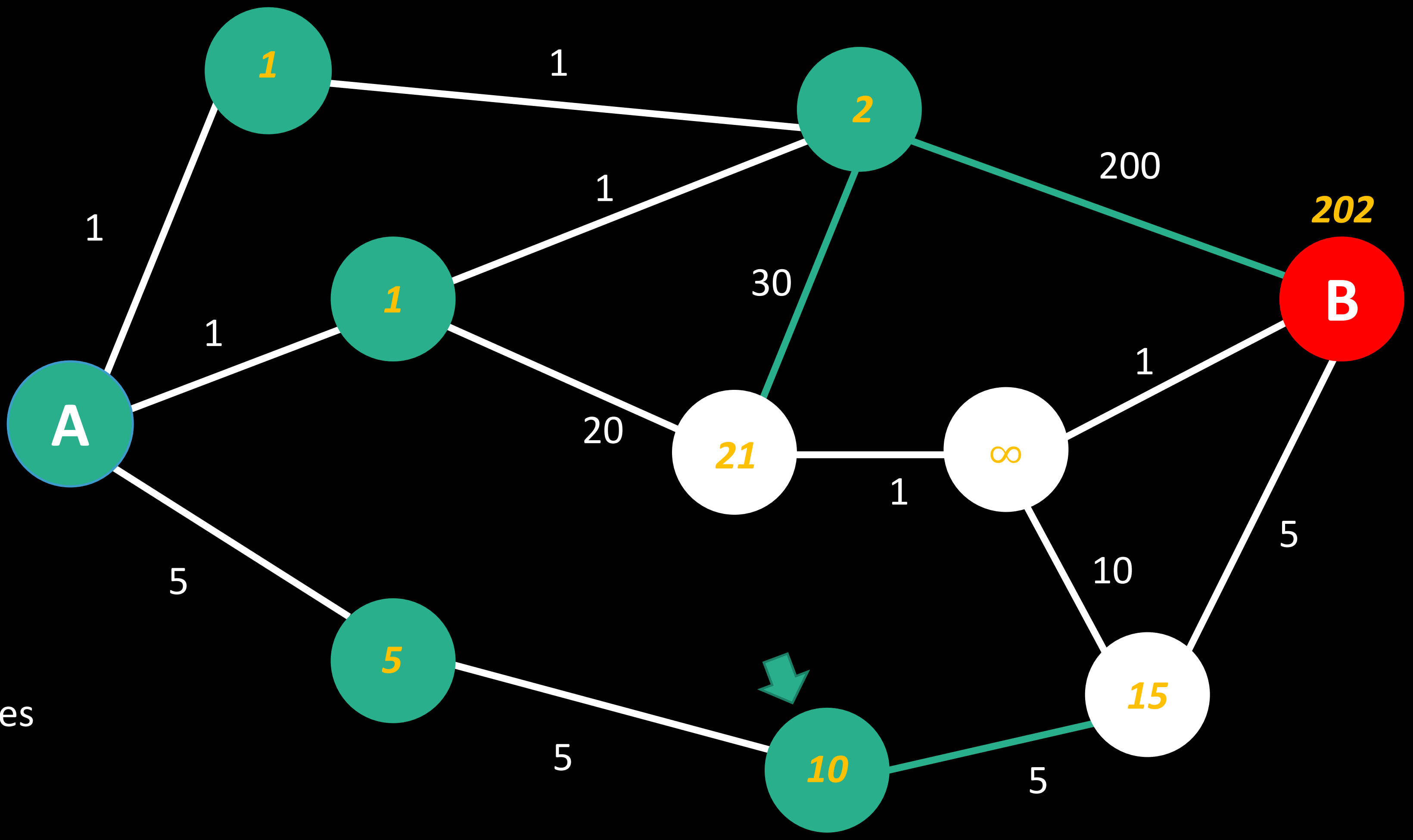


■ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

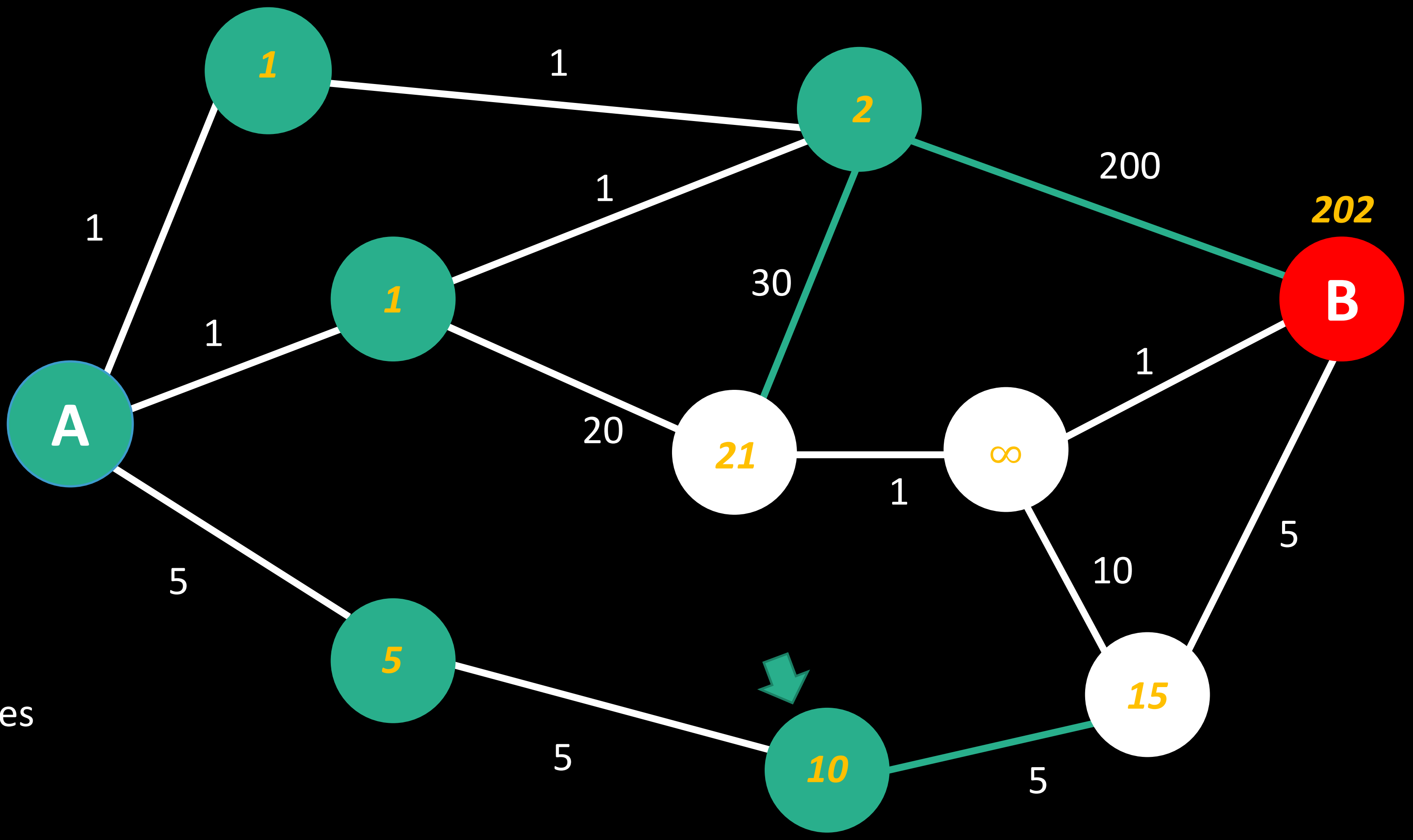


⬜ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example...



Minimal cost path is 15

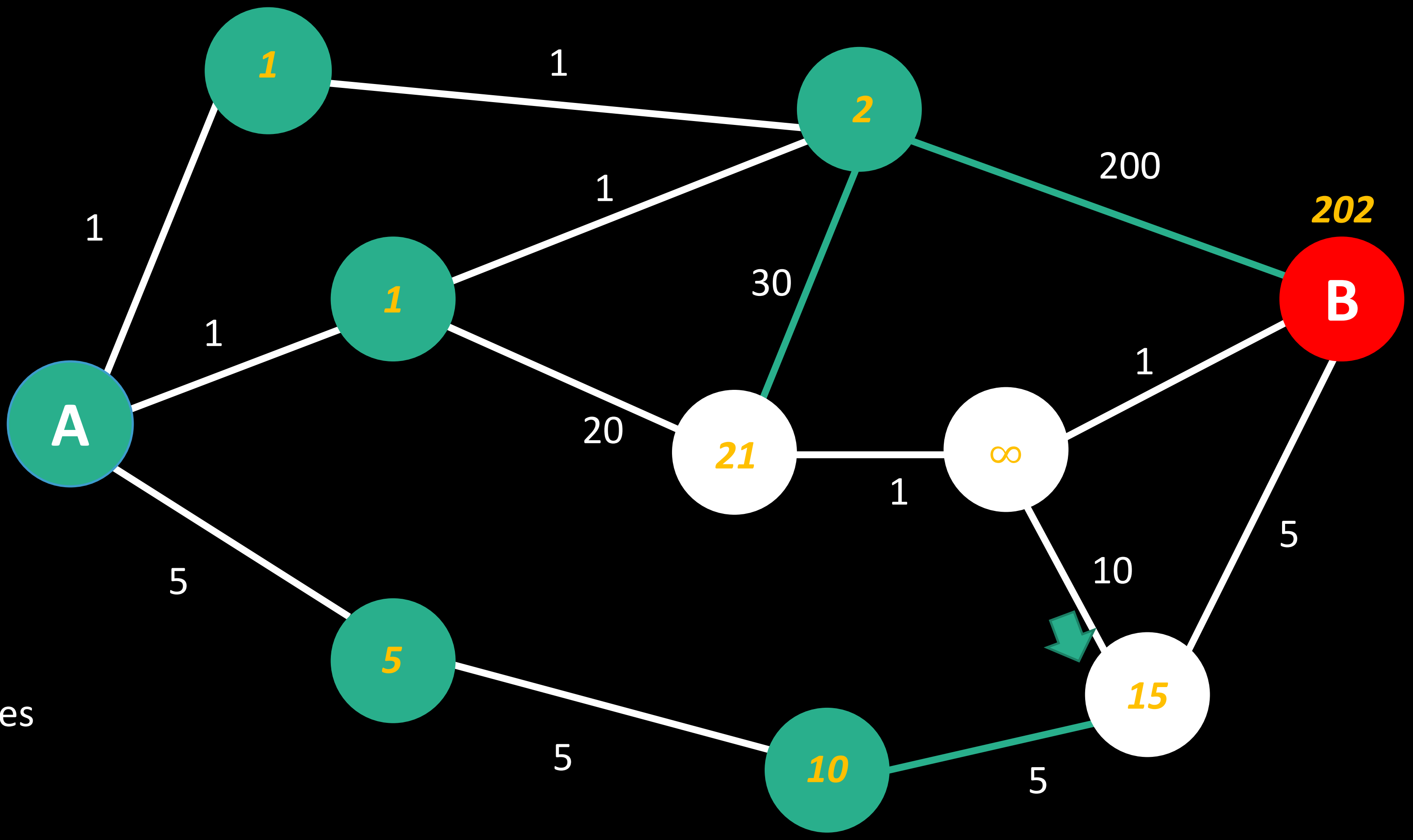
⬜ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

Minimal cost path is 15

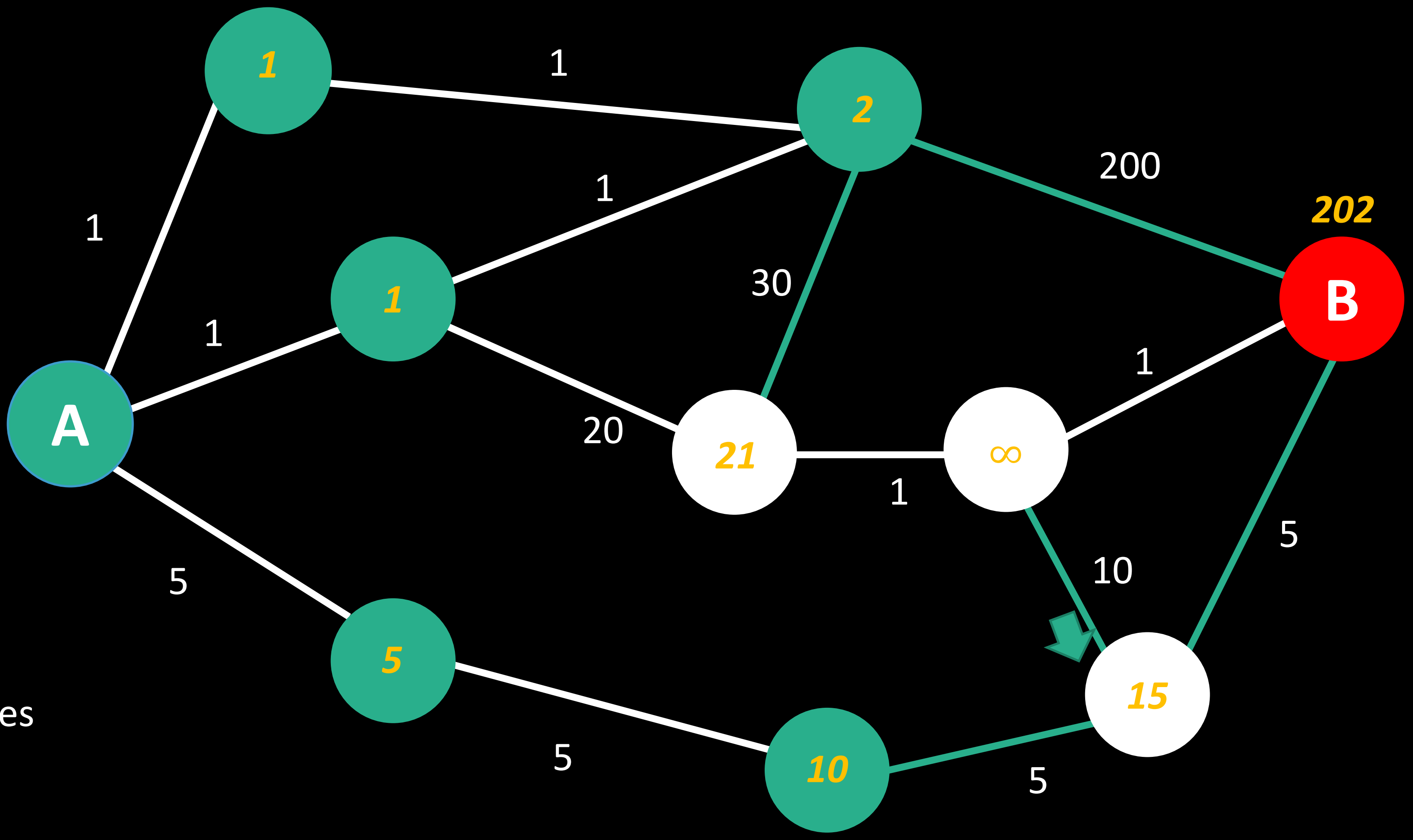


■ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

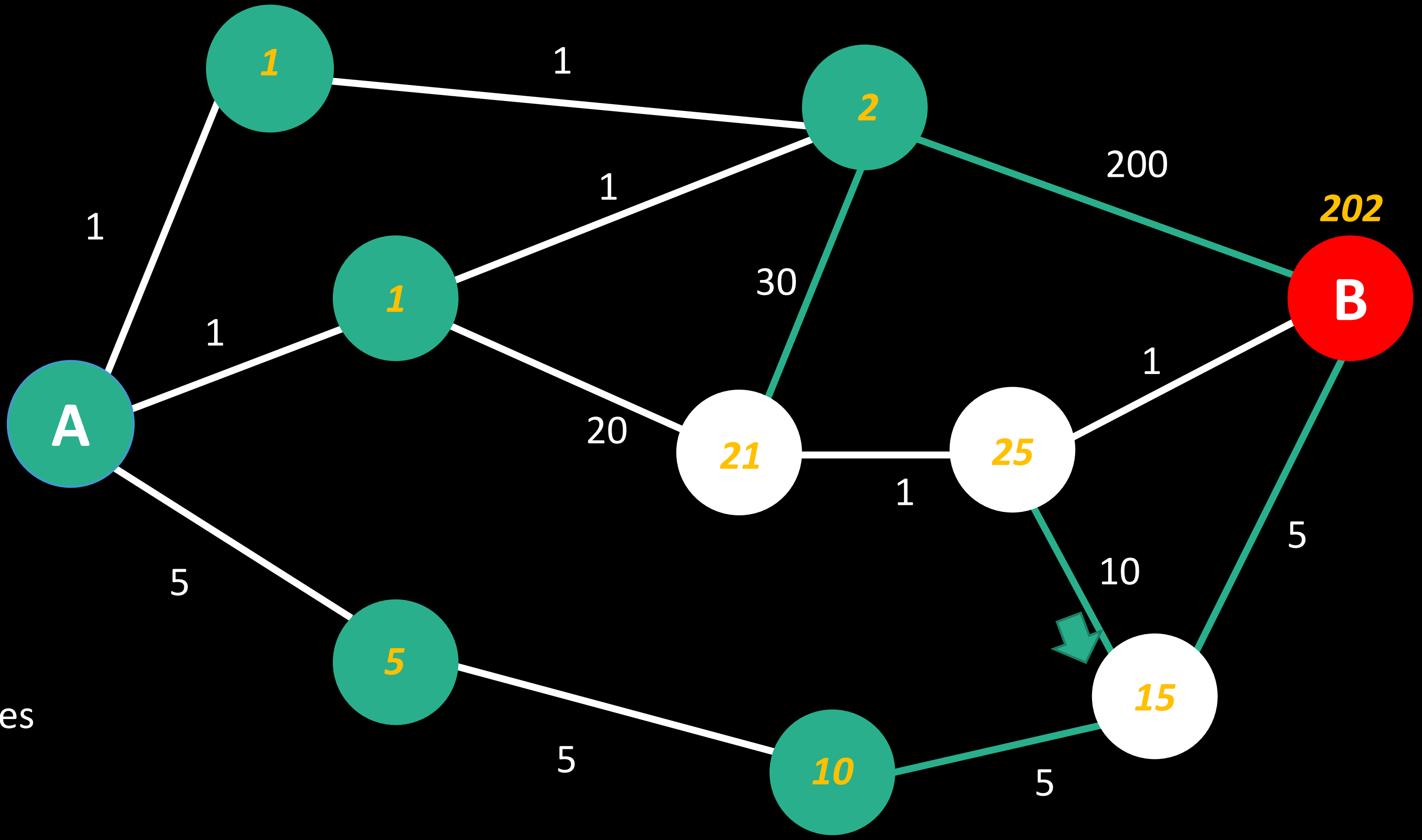


⬜ Unvisited nodes

🟩 Visited nodes

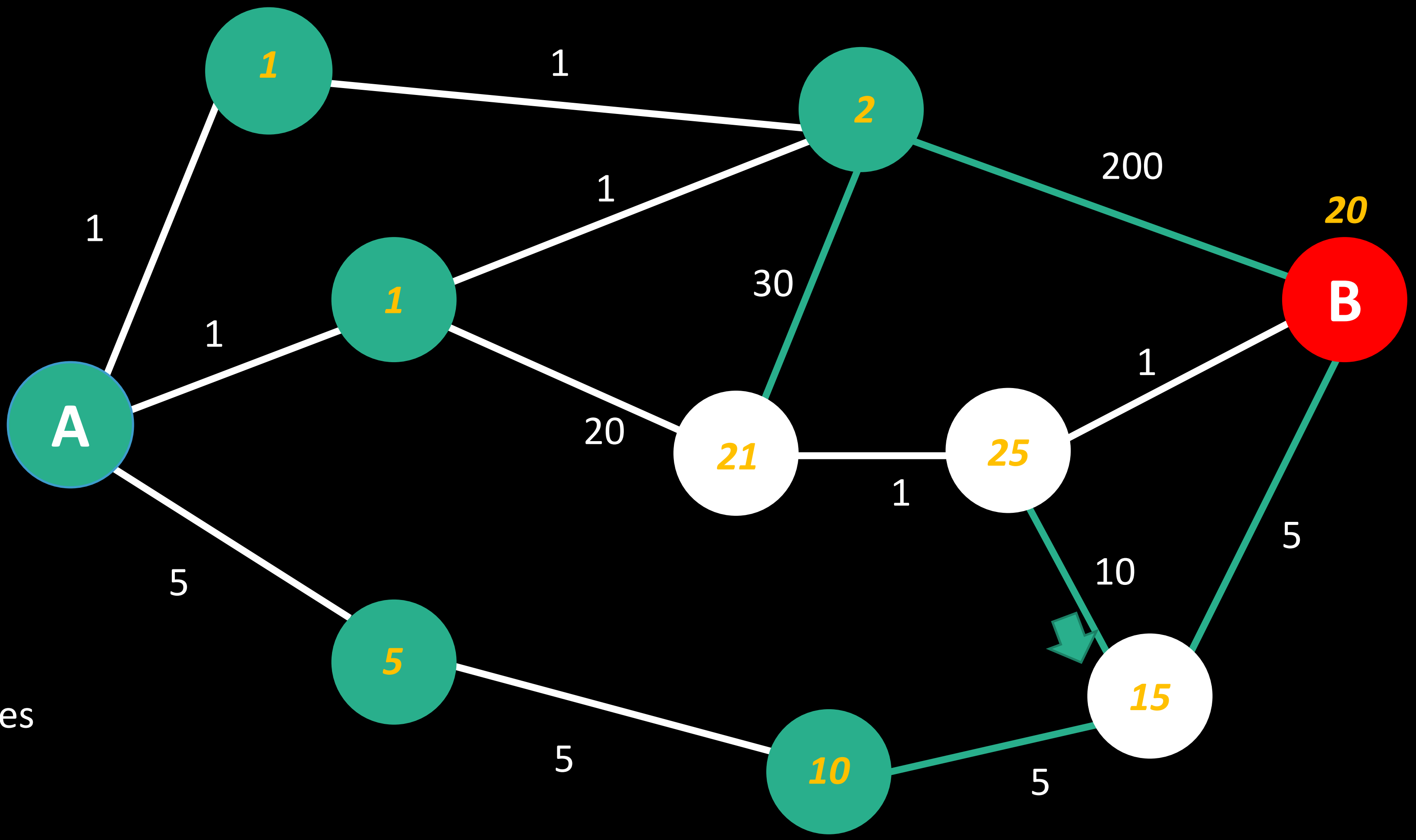
d Distance

A more illustrative example…



- ■ Unvisited nodes
- 🟩 Visited nodes
- *d* Distance

A more illustrative example…

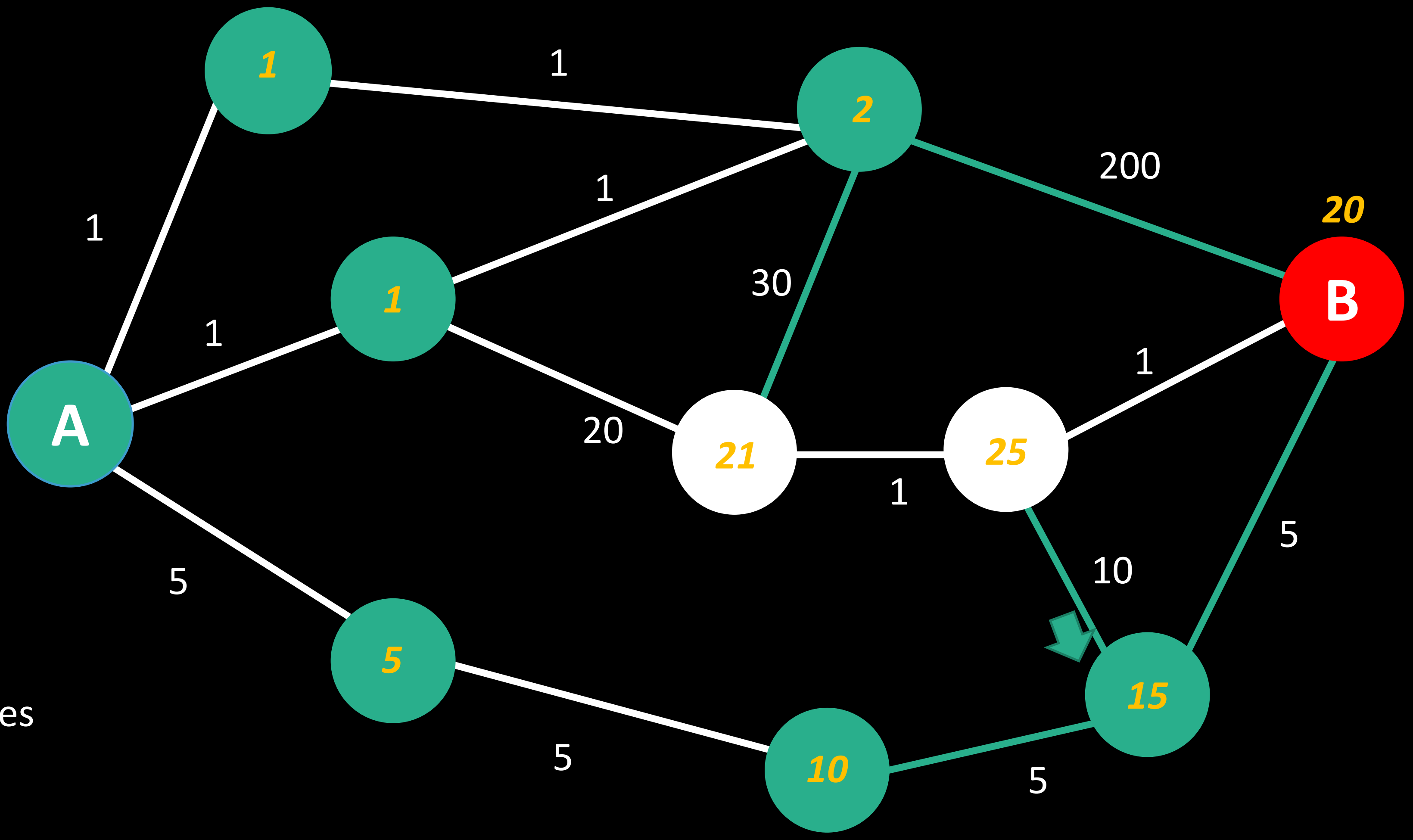


⬜ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

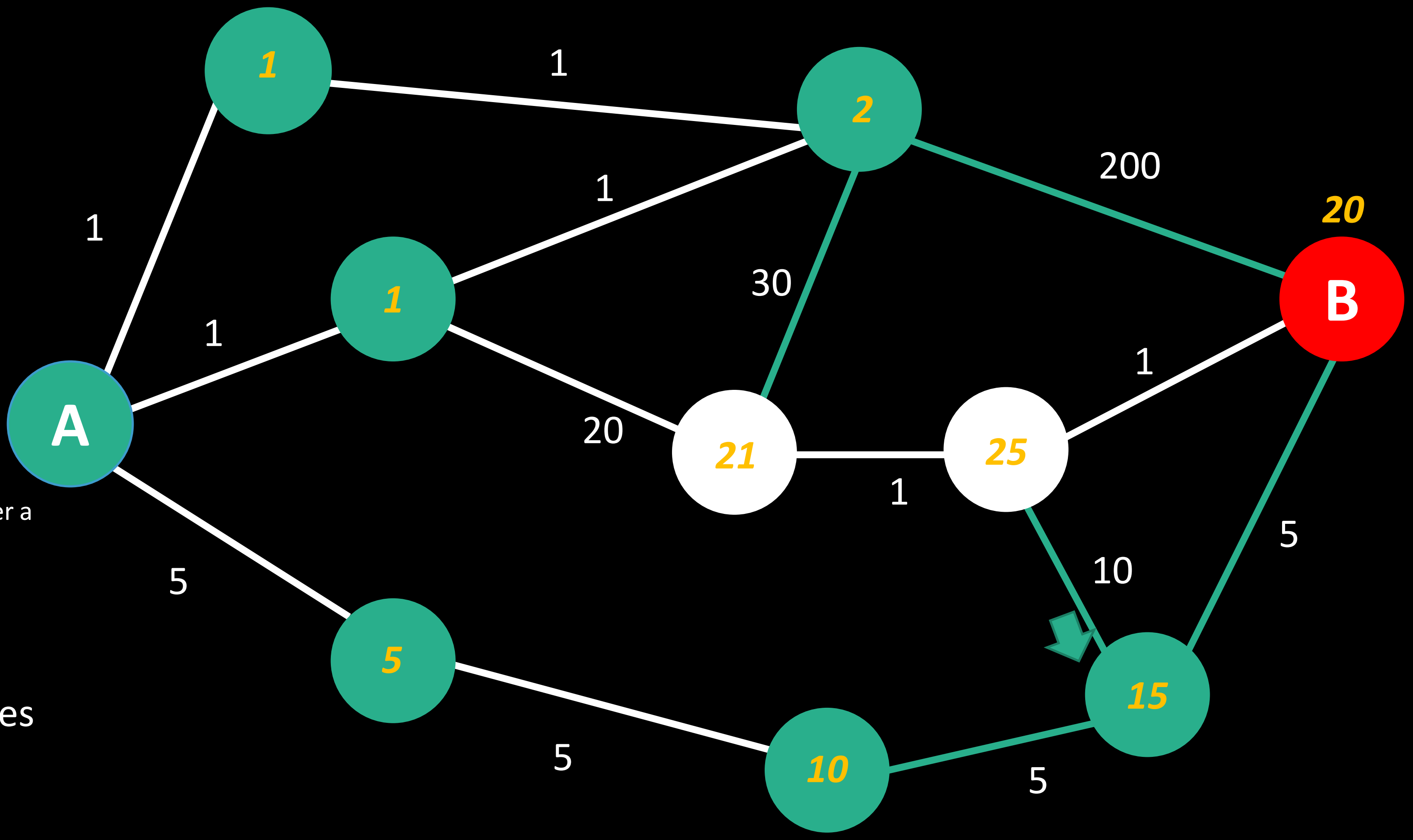


⬜ Unvisited nodes

🟩 Visited nodes

d Distance

A more illustrative example…

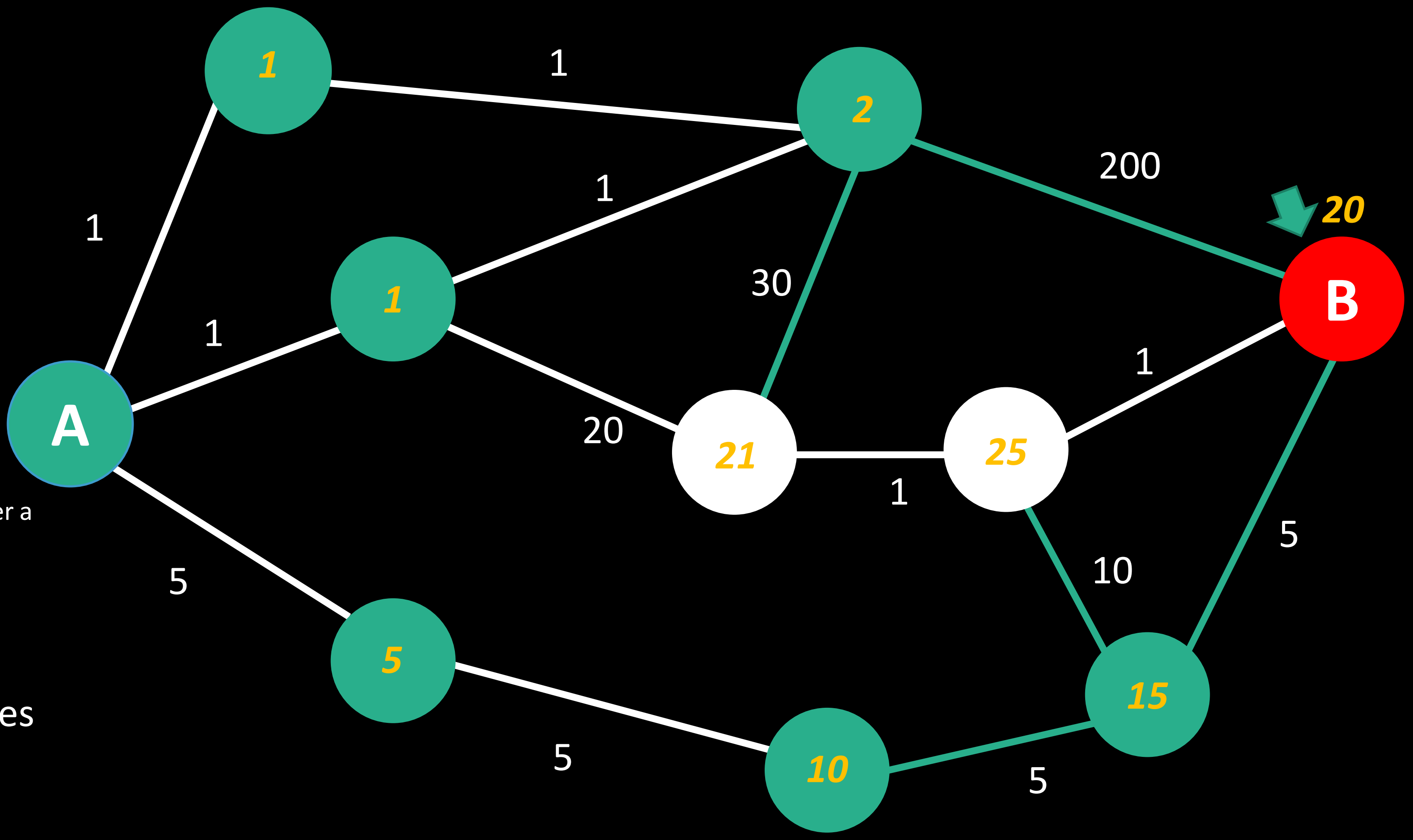


Minimal cost path is **20**

No other path could offer a total cost < 20

- ⬜ Unvisited nodes
- 🟩 Visited nodes
- *d* Distance

A more illustrative example…



Minimal cost path is **20**

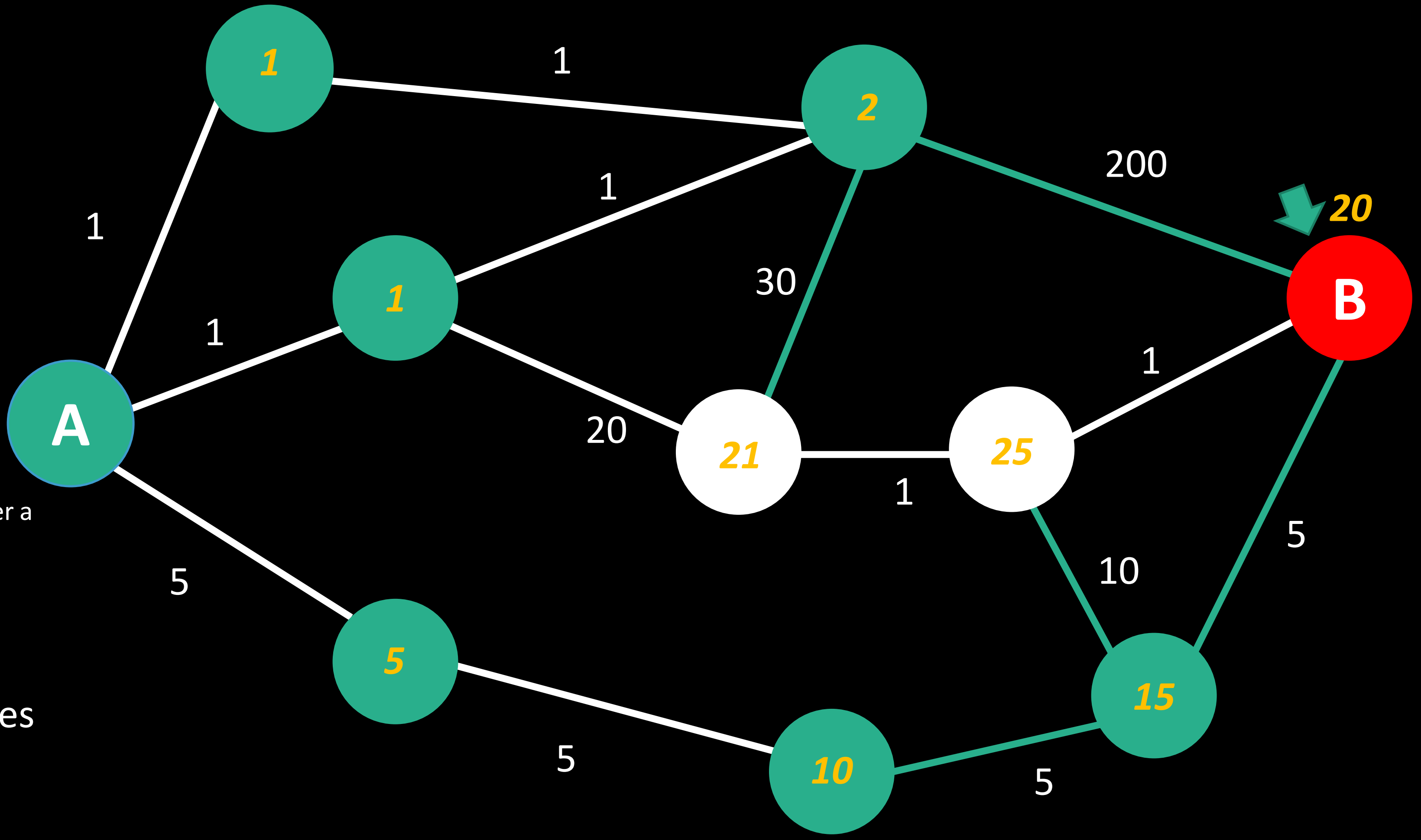
No other path could offer a total cost < 20

- ⬜ Unvisited nodes
- 🟩 Visited nodes
- *d* Distance

A more illustrative example...

Minimal cost path is **20**

No other path could offer a total cost < 20



Target node found, terminate algorithm

■ Unvisited nodes

🟩 Visited nodes

d Distance

Djikstra’s Algorithm

- Key idea: always choose the least expensive **total** path
  - Any other path is known to cost more
- Djikstra’s identifies the **optimal** path
  - Proof will be left as outside of the scope of this course
- Keep track of:
  - Distance from node A to node V
  - Which nodes have been visited already
  - The previous node (as we did in BFS)

Djikstra's Algorithm

1. Initialize all costs to ∞
2. Expand starting node v
   - Calculate the cost of the path between the starting node and all its neighbors
   - Add node v to N, remove node v from Ω
   - Update the cost of all the neighboring nodes
3. Choose the next node to visit, denoted u
   - Choose $u \in \Omega$ such that $\forall u_i \in \Omega, u = \mathrm{argmin}_{\mathrm{u_i}}(\{dist(u_i, v)\})$
4. Repeat step 2-3, substituting u for v
   - Unvisited nodes may have their costs updated
   - Previously visited nodes are never visited again

Set of all visited nodes: N
Set of all unvisited nodes: Ω

Djikstra’s Algorithm

- Terminate when target node is visited
    - Or, when we have not yet visited target node but all remaining unvisited nodes have cost ∞
    - i.e. node is part of unconnected part of graph
- Consider a very large, connected graph
    - Expensive (in time, and potentially space) to find optimal solution
    - Save time using A* search
        - (Potentially) exploits knowledge about the problem to converge to optimum solution faster
        - You may use it for the projects, will not be covered in class

Graph Algorithms

Implementation

Implementing DFS

- How do we implement DFS?

Implementing DFS

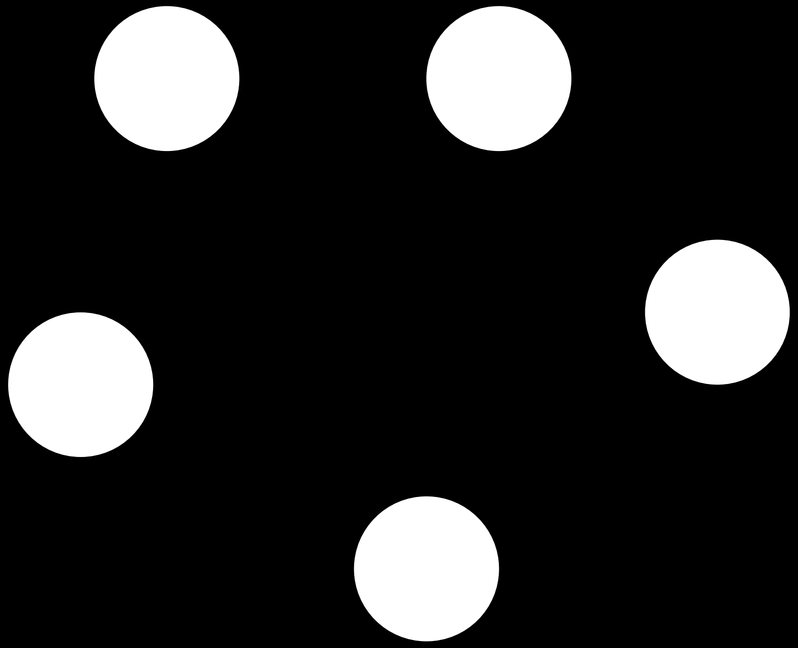
- How do we implement DFS?
- How do we represent a graph?

Implementing Graphs

- What about the way we implemented trees?

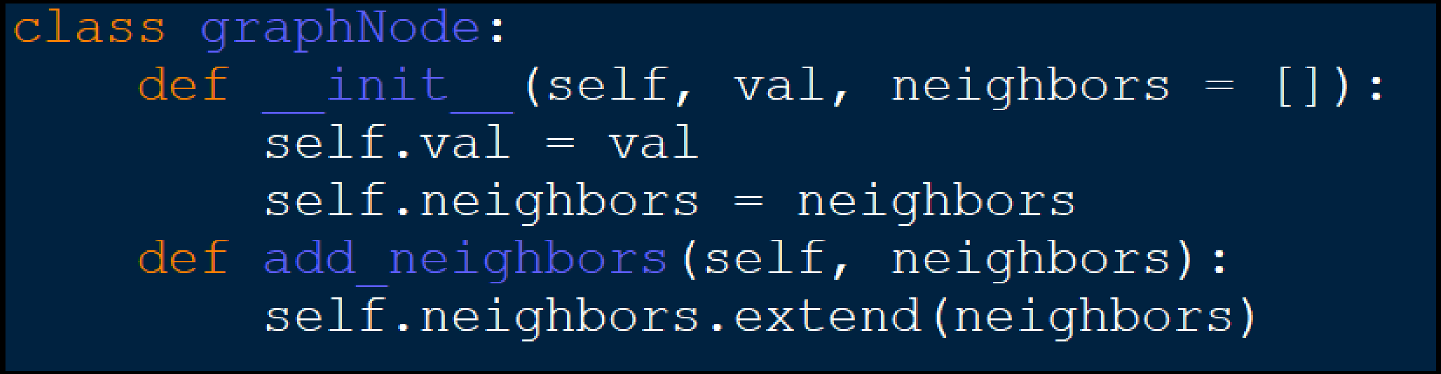
Implementing Graphs

- What about the way we implemented trees?

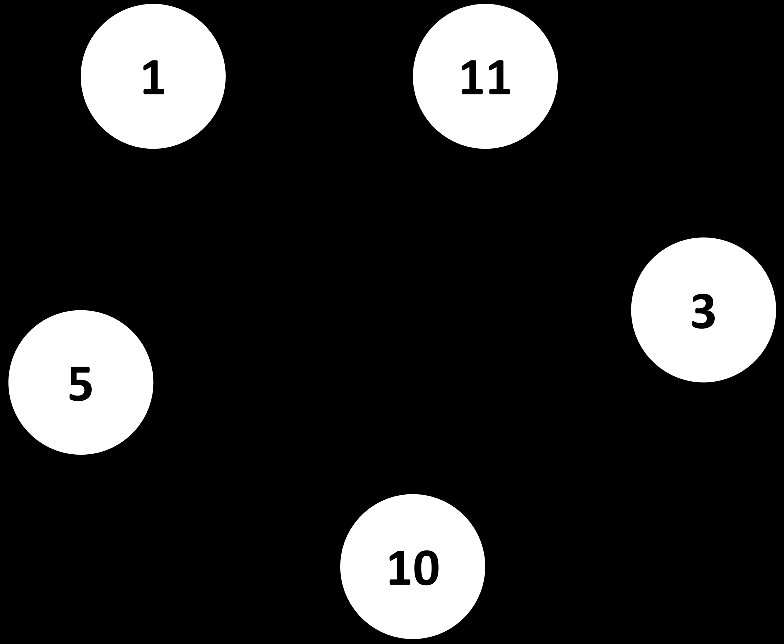


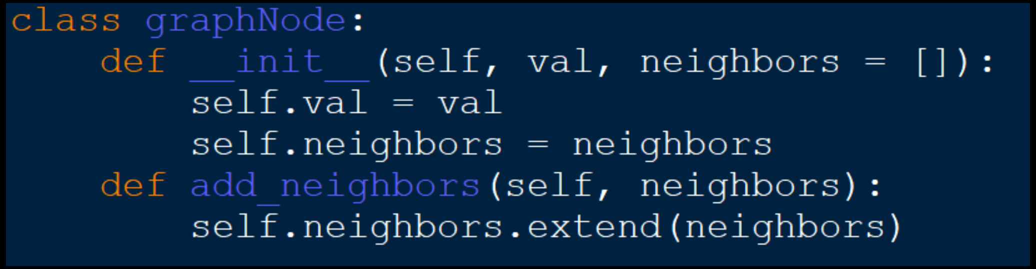
Implementing Graphs

- What about the way we implemented trees?



```
class graphNode:
    def __init__(self, val, neighbors = []):
        self.val = val
        self.neighbors = neighbors
    def add_neighbors(self, neighbors):
        self.neighbors.extend(neighbors)
```

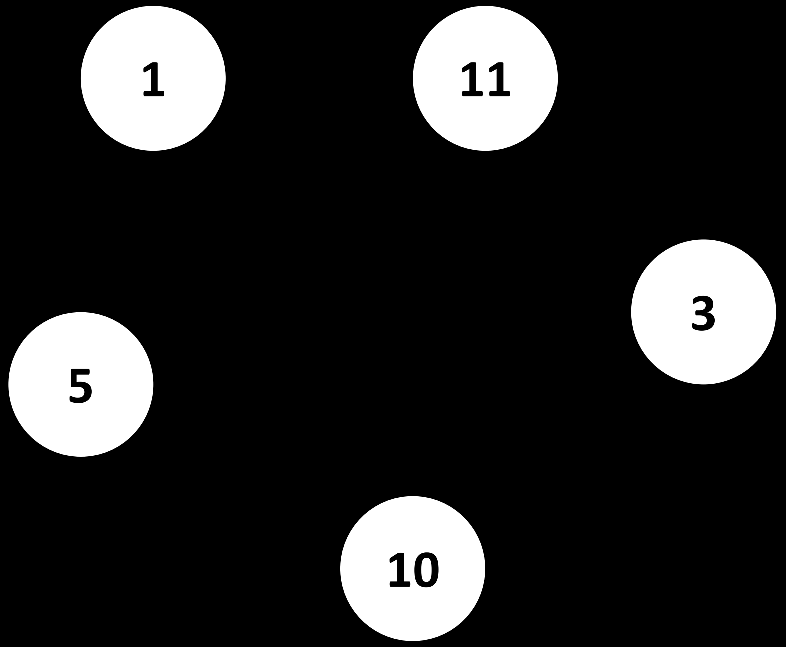


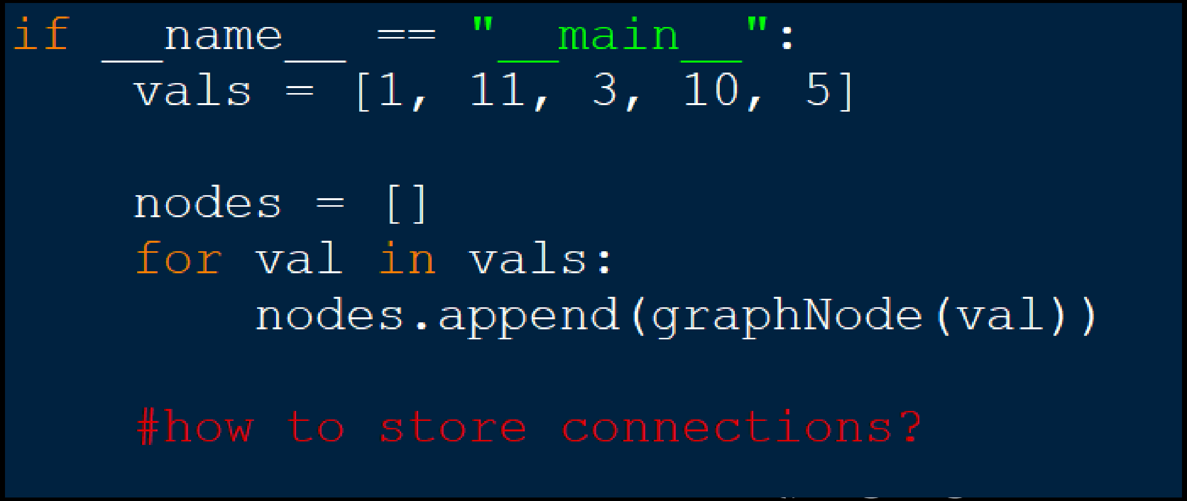


```
class graphNode:
    def __init__(self, val, neighbors = []):
        self.val = val
        self.neighbors = neighbors
    def add_neighbors(self, neighbors):
        self.neighbors.extend(neighbors)
```

Implementing Graphs

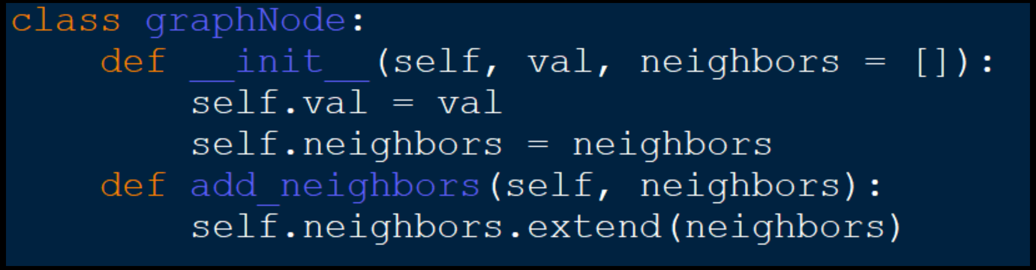
- What about the way we implemented trees?





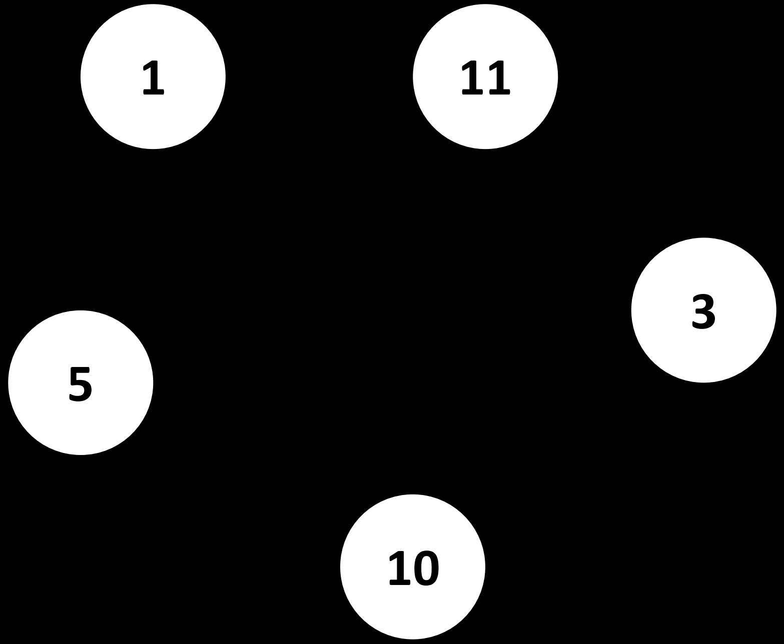
```
if __name__ == "__main__":
    vals = [1, 11, 3, 10, 5]

    nodes = []
    for val in vals:
        nodes.append(graphNode(val))

    #how to store connections?
```

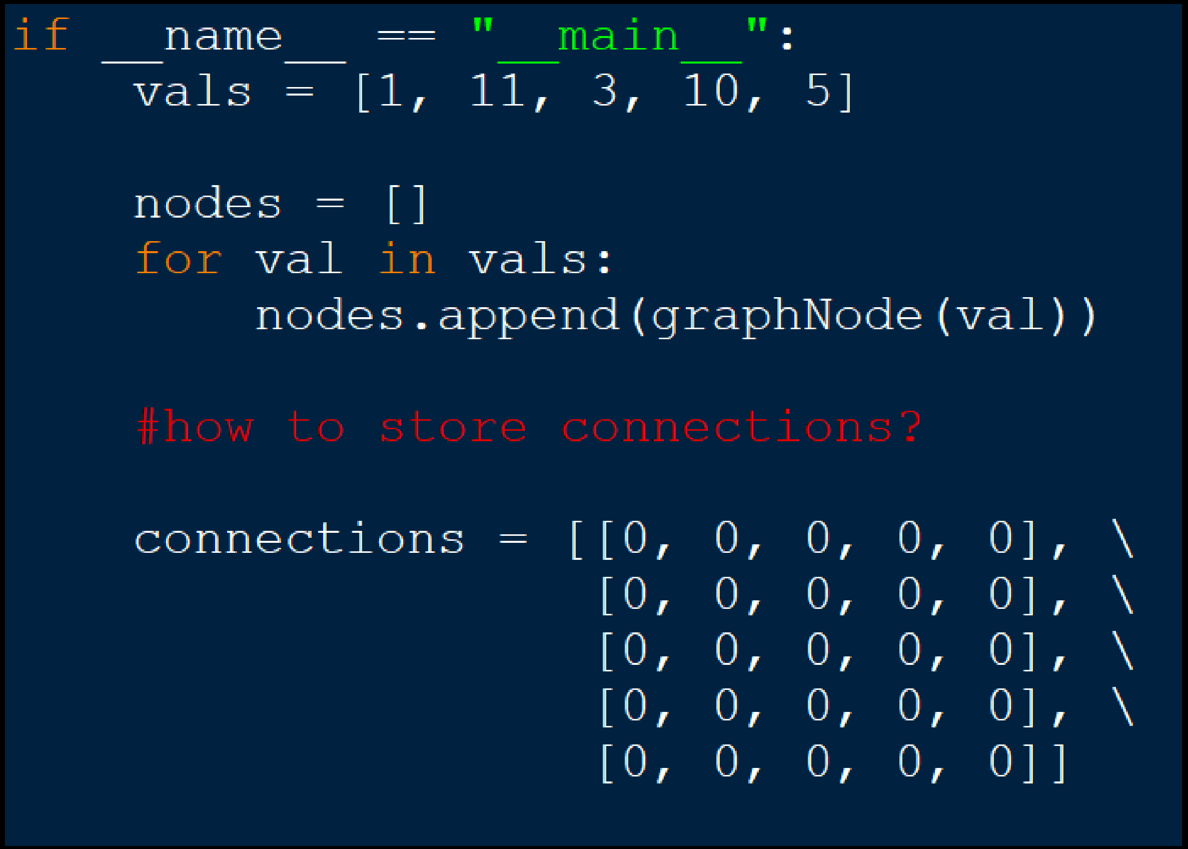
Implementing Graphs



```python
class graphNode:
    def __init__(self, val, neighbors = []):
        self.val = val
        self.neighbors = neighbors
    def add_neighbors(self, neighbors):
        self.neighbors.extend(neighbors)
```

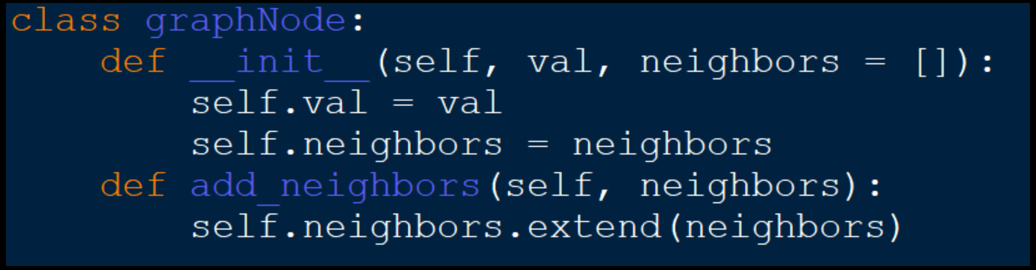
- What about the way we implemented trees?





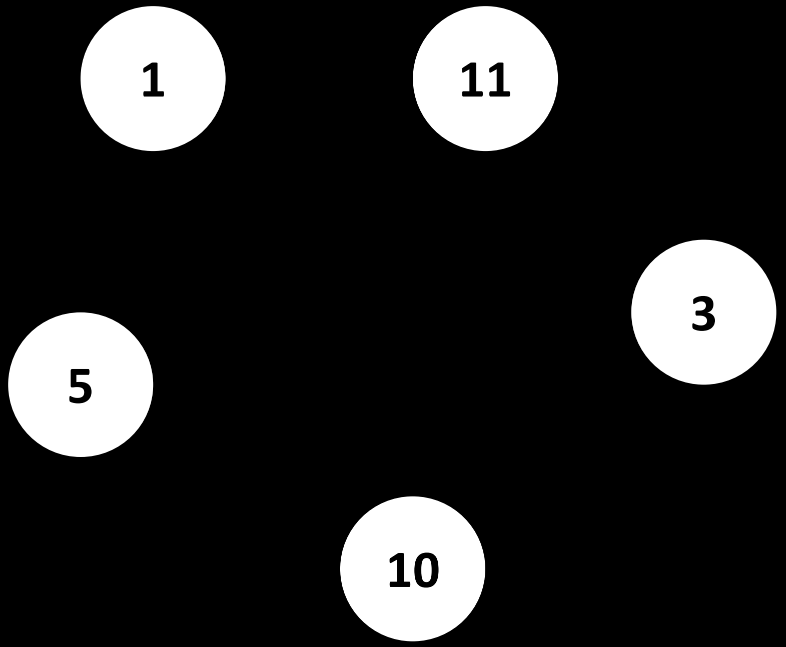
```python
if __name__ == "__main__":
    vals = [1, 11, 3, 10, 5]

    nodes = []
    for val in vals:
        nodes.append(graphNode(val))

    #how to store connections?

    connections = [[0, 0, 0, 0, 0], \
                   [0, 0, 0, 0, 0], \
                   [0, 0, 0, 0, 0], \
                   [0, 0, 0, 0, 0], \
                   [0, 0, 0, 0, 0]]
```

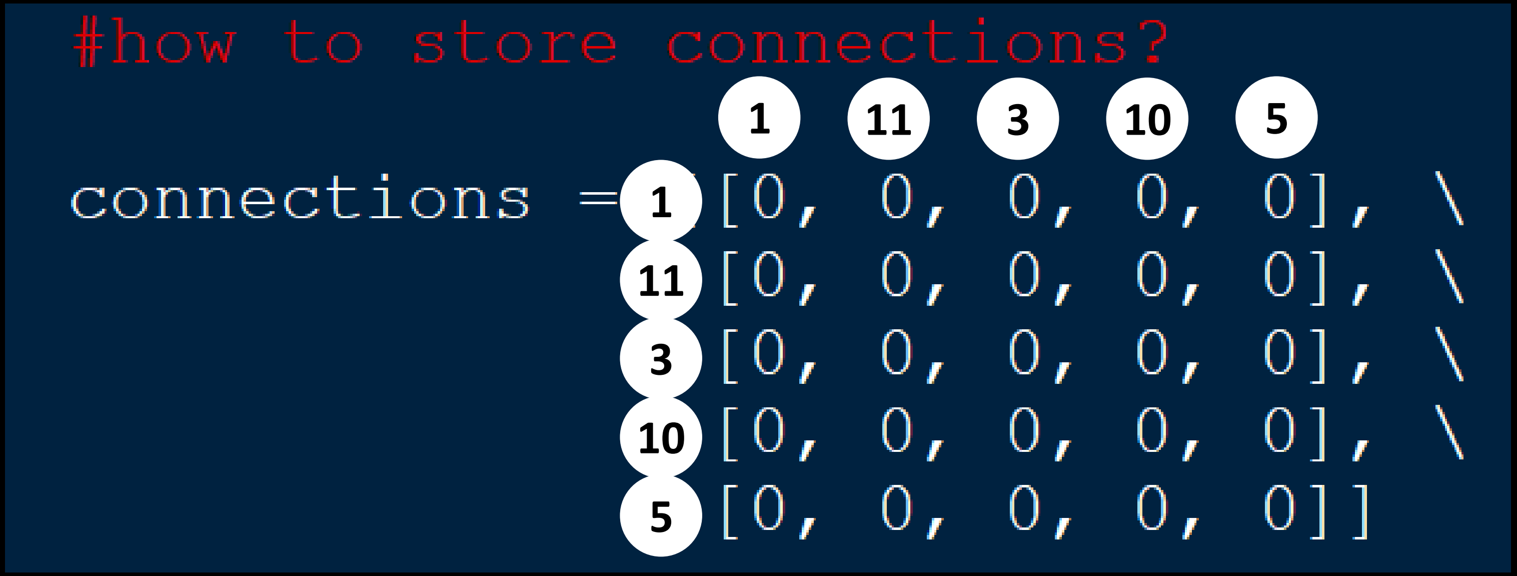
Implementing Graphs



```
class graphNode:
    def __init__(self, val, neighbors = []):
        self.val = val
        self.neighbors = neighbors
    def add_neighbors(self, neighbors):
        self.neighbors.extend(neighbors)
```

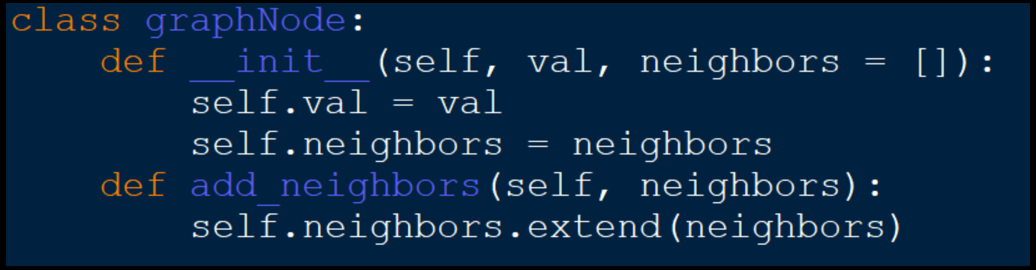
- What about the way we implemented trees?





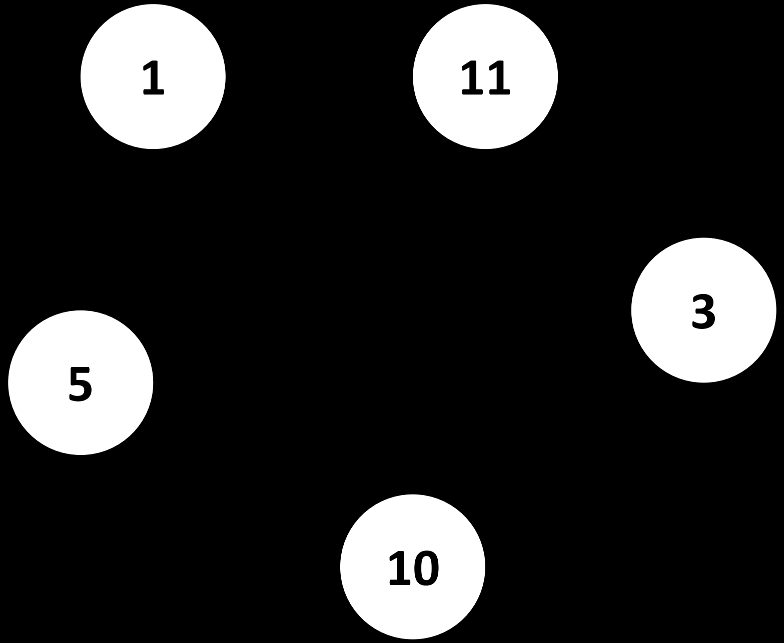
```
#how to store connections?

connections = [[0, 0, 0, 0, 0], \
               [0, 0, 0, 0, 0], \
               [0, 0, 0, 0, 0], \
               [0, 0, 0, 0, 0], \
               [0, 0, 0, 0, 0]]
```

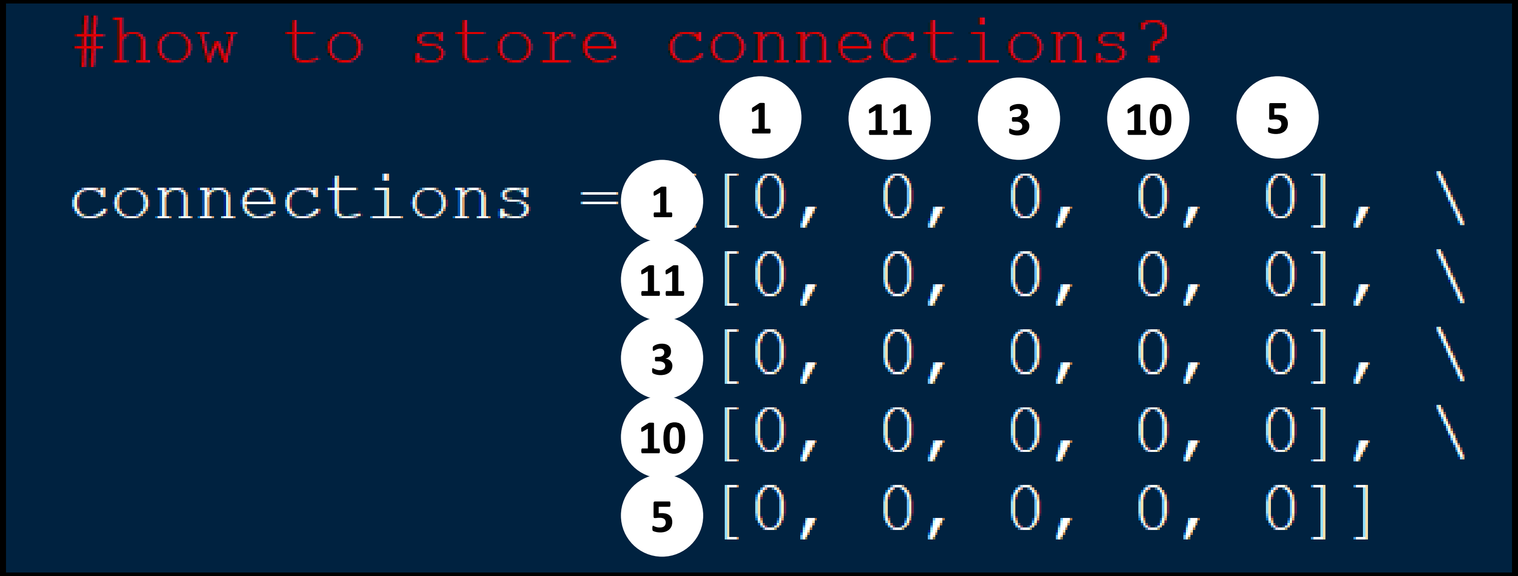
Implementing Graphs



```
class graphNode:
    def __init__(self, val, neighbors = []):
        self.val = val
        self.neighbors = neighbors
    def add_neighbors(self, neighbors):
        self.neighbors.extend(neighbors)
```

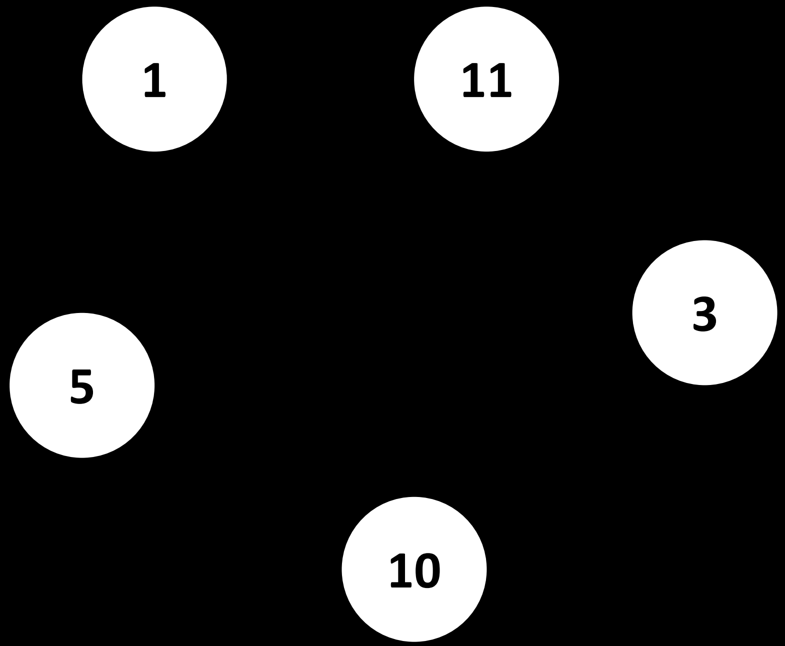
- What about the way we implemented trees?
- Wait a minute… what does the class offer?

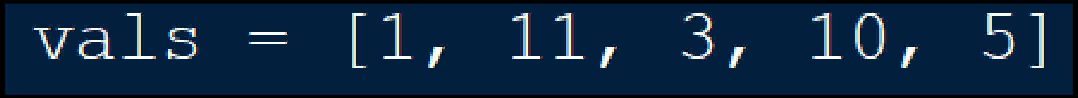




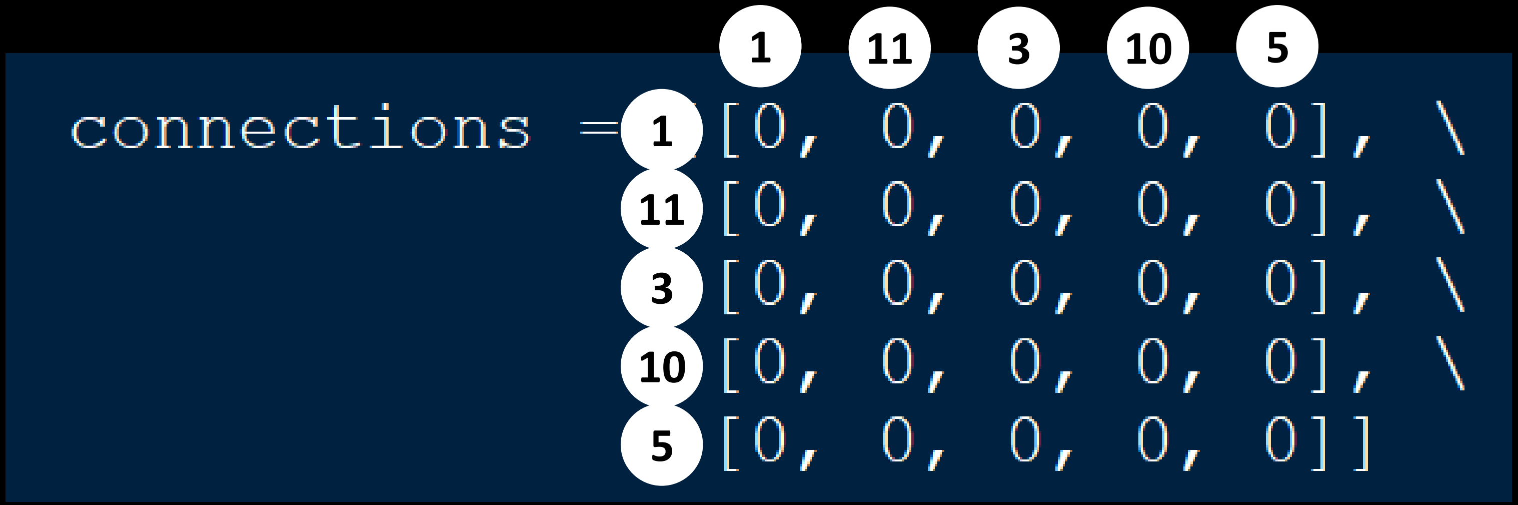
```
#how to store connections?
                      1   11    3   10    5
connections =  1  [0,  0,  0,  0,  0], \
              11  [0,  0,  0,  0,  0], \
               3  [0,  0,  0,  0,  0], \
              10  [0,  0,  0,  0,  0], \
               5  [0,  0,  0,  0,  0]]
```

Implementing Graphs

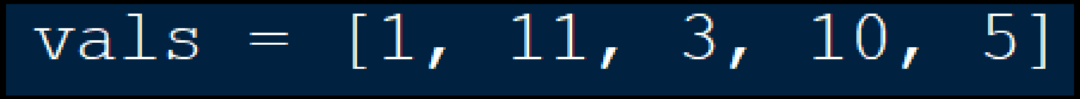




```
vals = [1, 11, 3, 10, 5]
```

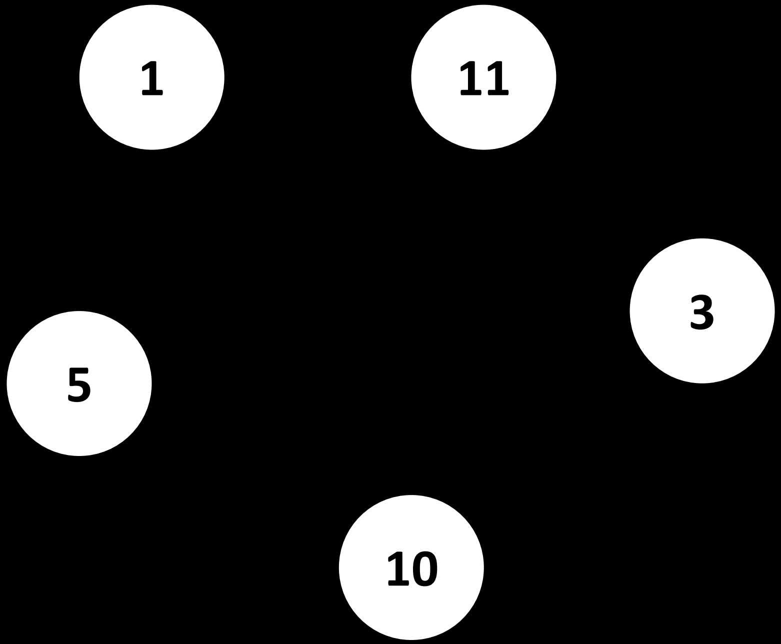


```
connections = [[0, 0, 0, 0, 0], \
               [0, 0, 0, 0, 0], \
               [0, 0, 0, 0, 0], \
               [0, 0, 0, 0, 0], \
               [0, 0, 0, 0, 0]]
```

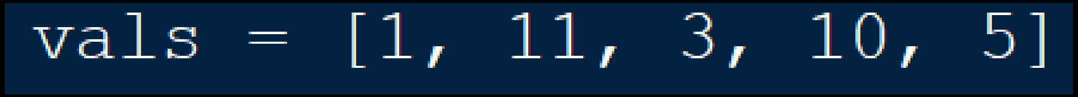


```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs



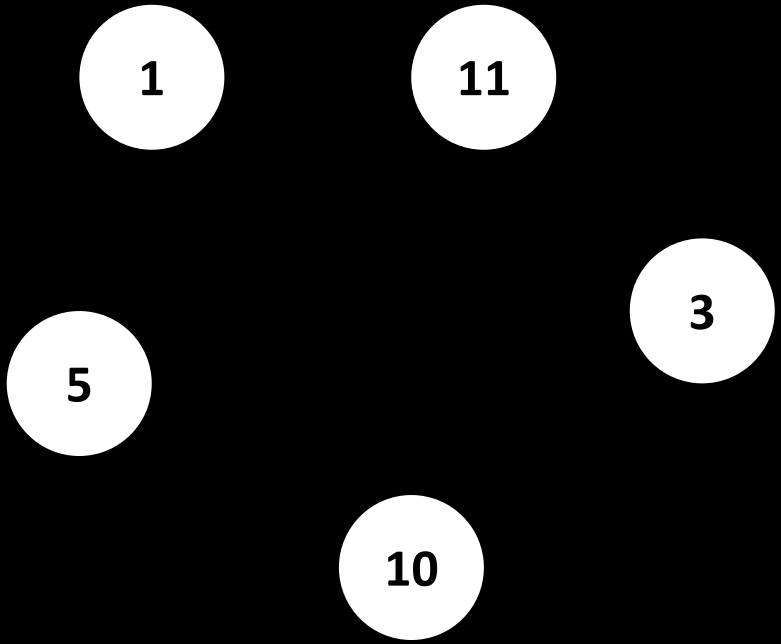
	1	11	3	10	5
1					
11					
3					
10					
5					



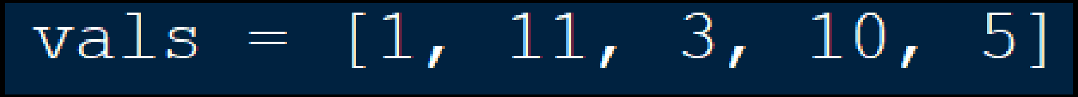
```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

Case: Nothing is connected



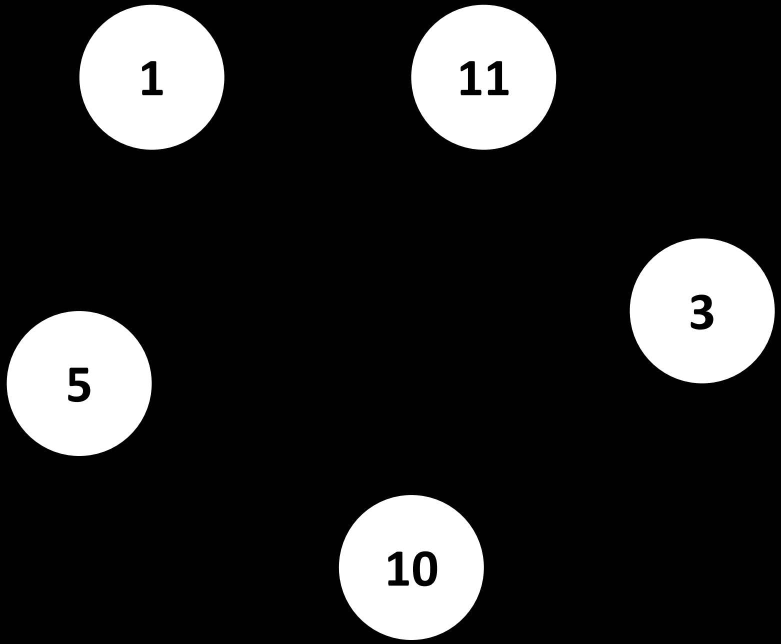
	1	11	3	10	5
1	0	0	0	0	0
11	0	0	0	0	0
3	0	0	0	0	0
10	0	0	0	0	0
5	0	0	0	0	0



```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

Case: Maxed out connections

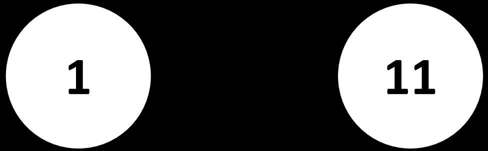


	1	11	3	10	5
1					
11					
3					
10					
5					

Implementing Graphs

Case: Maxed out connections

```
vals = [1, 11, 3, 10, 5]
```

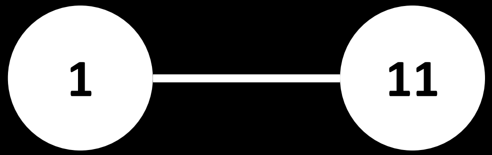


	1	11
1		
11		

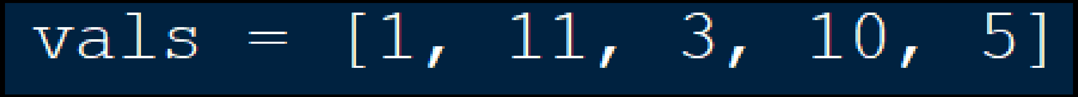
Implementing Graphs

Case: Maxed out connections (?)

```
vals = [1, 11, 3, 10, 5]
```



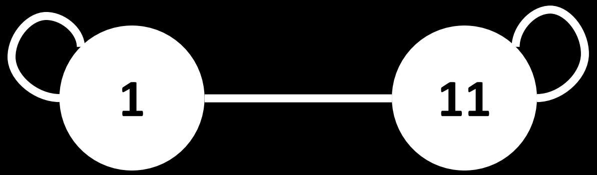
	1	11
1		
11		



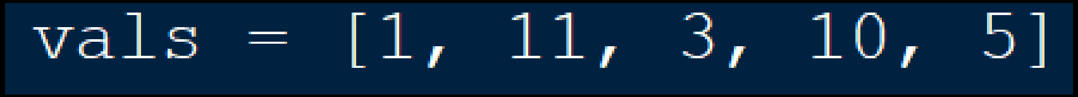
```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

Case: Maxed out connections



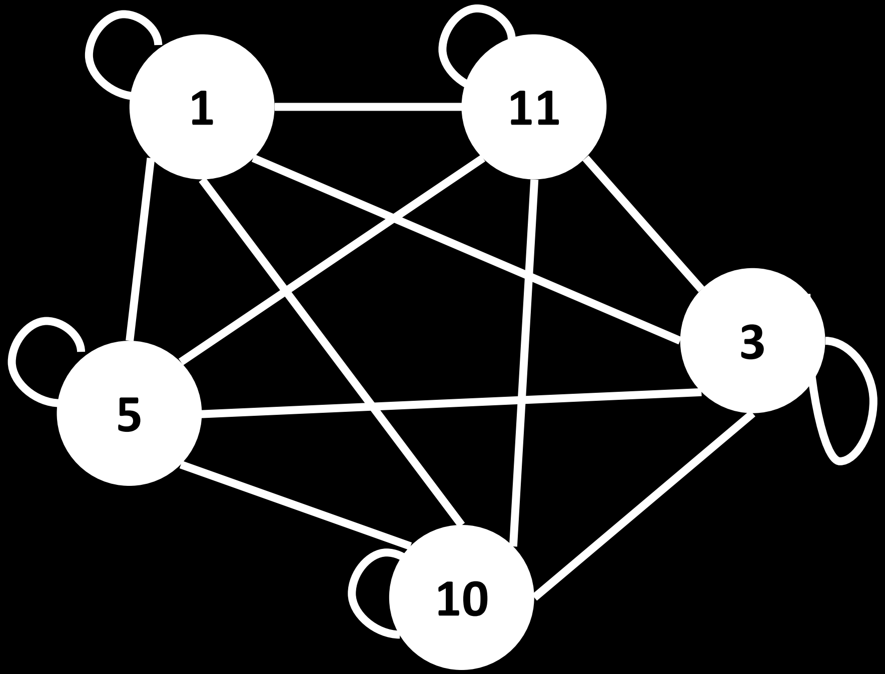
	1	11
1	1	1
11	1	1



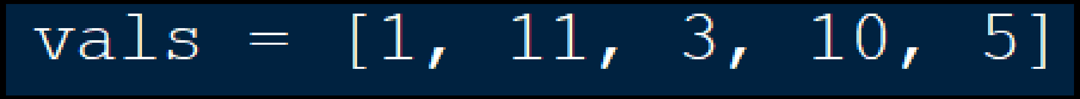
```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

Case: Maxed out connections



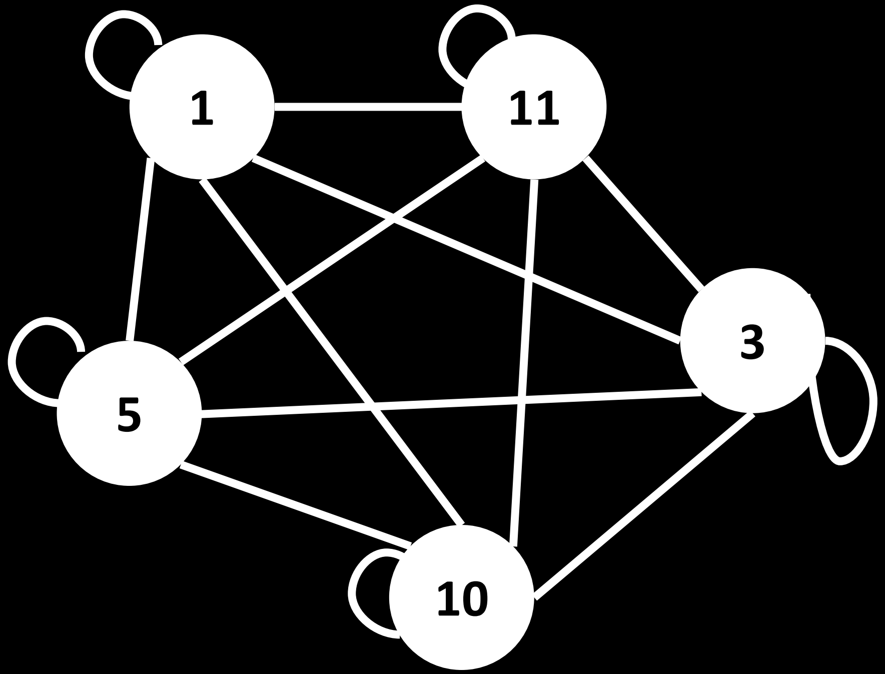
	1	11	3	10	5
1					
11					
3					
10					
5					



```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

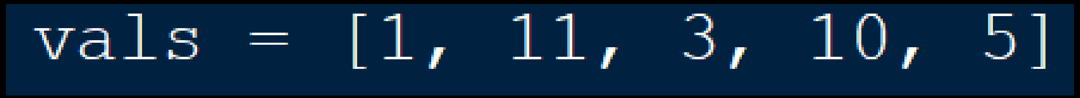
Case: Maxed out connections



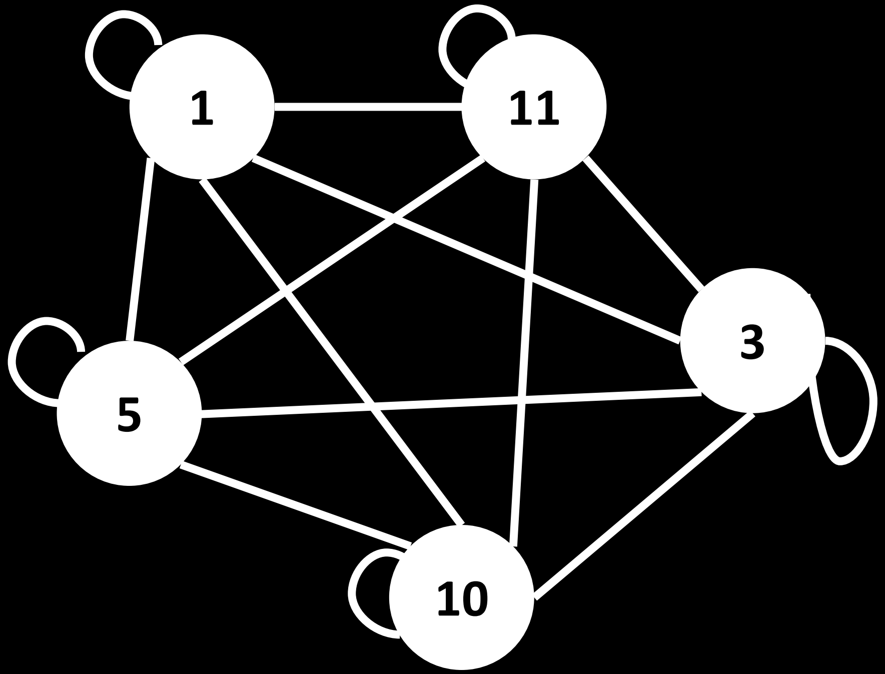
	1	11	3	10	5
1	1	1	1	1	1
11	1	1	1	1	1
3	1	1	1	1	1
10	1	1	1	1	1
5	1	1	1	1	1

Implementing Graphs

Case: Maxed out connections

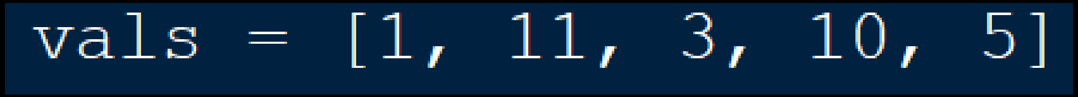


```
vals = [1, 11, 3, 10, 5]
```



	1	11	3	10	5
1	1	1	1	1	1
11	1	1	1	1	1
3	1	1	1	1	1
10	1	1	1	1	1
5	1	1	1	1	1

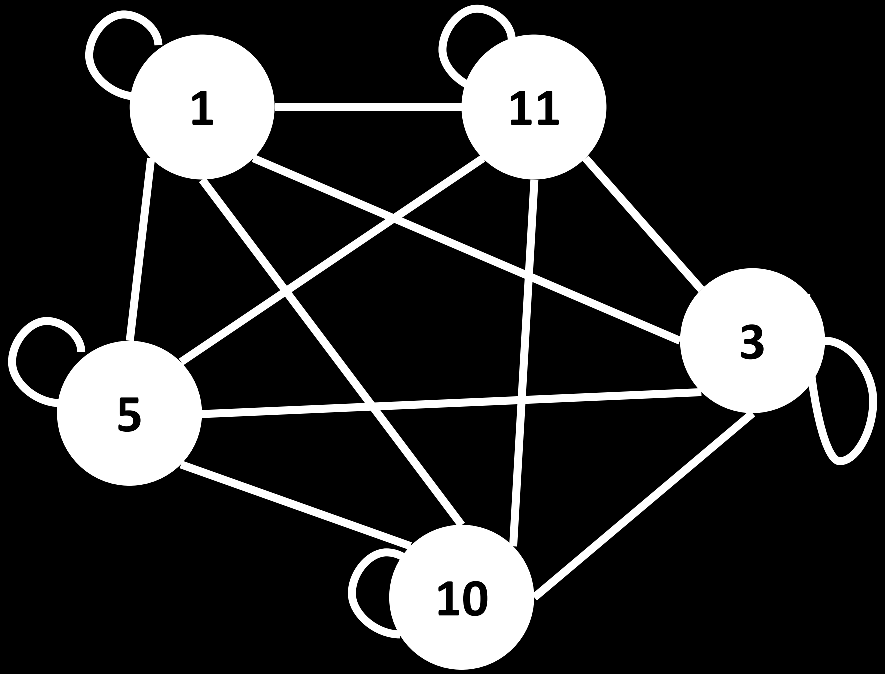
For an undirected graph:
If 1 is connected to 11,
11 is connected to 1!



```
vals = [1, 11, 3, 10, 5]
```

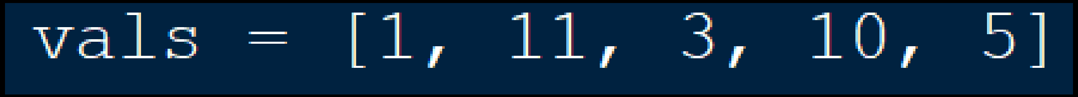
Implementing Graphs

Case: Maxed out connections



	1	11	3	10	5
1	1	1	1	1	1
11	1	1	1	1	1
3	1	1	1	1	1
10	1	1	1	1	1
5	1	1	1	1	1

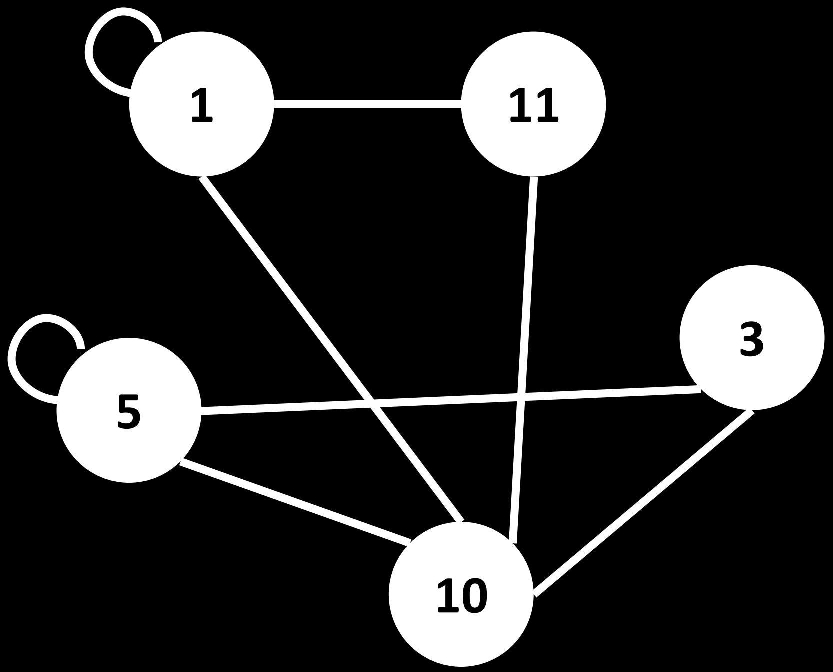
For an undirected graph:
If 1 is connected to 11,
11 is connected to 1!

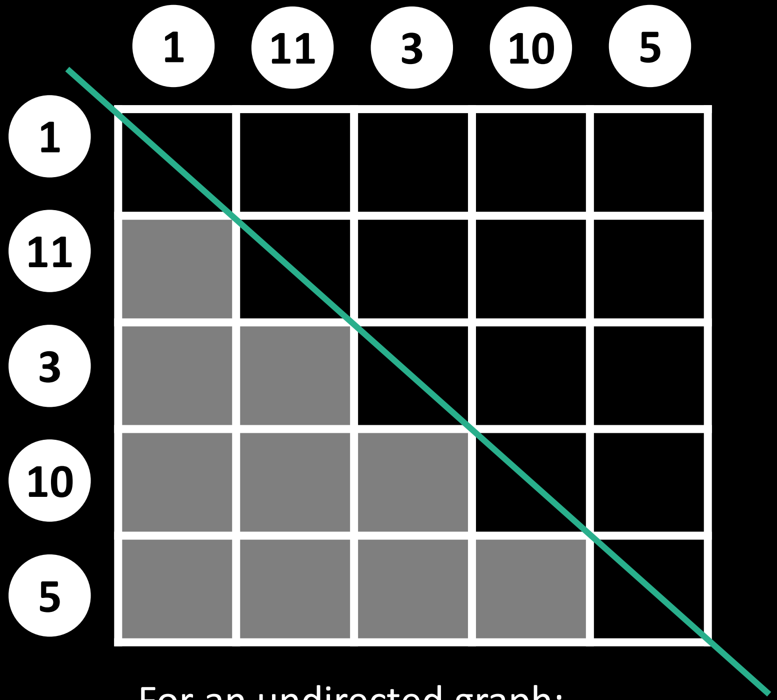


```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

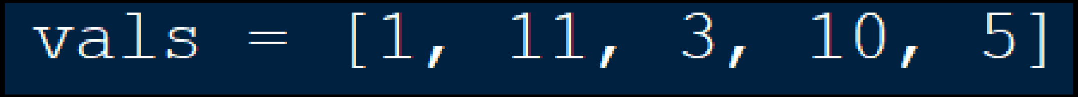
Fill out the **adjacency matrix** for the following graph





	1	11	3	10	5
1					
11	▨				
3	▨	▨			
10	▨	▨	▨		
5	▨	▨	▨	▨	

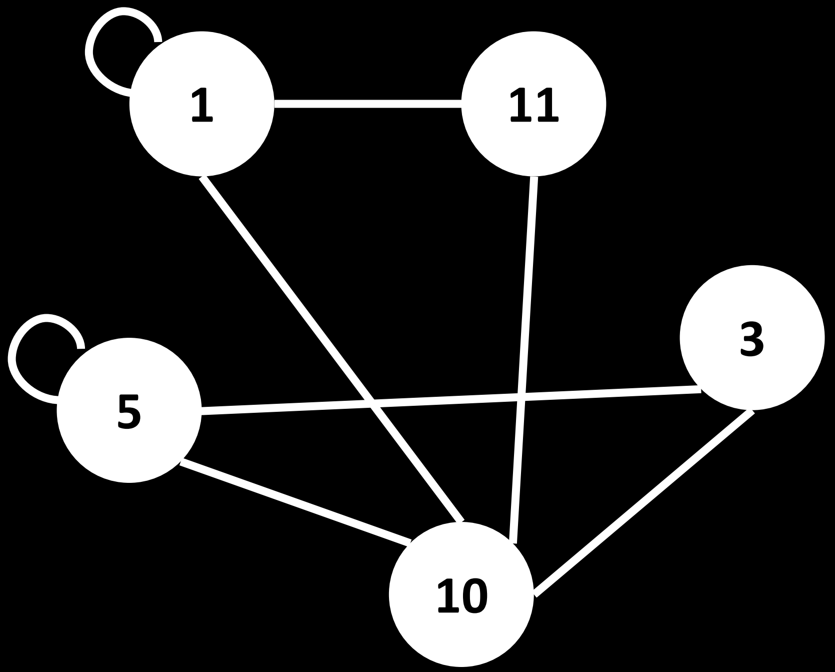
For an undirected graph:
If 1 is connected to 11,
11 is connected to 1!



```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

Fill out the **adjacency matrix** for the following graph



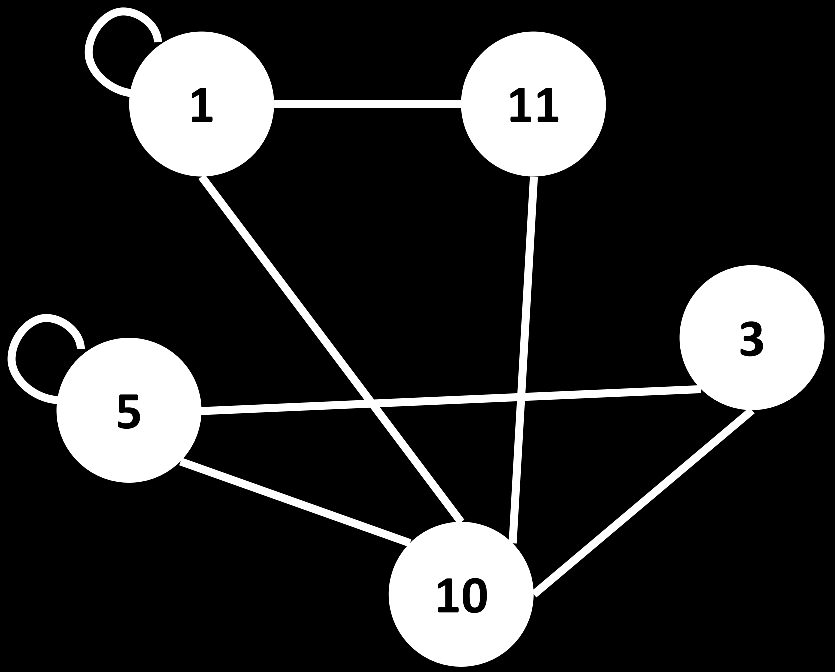
	1	11	3	10	5
1	1	1	0	1	0
11		0	0	1	0
3			0	1	1
10				0	1
5					1

For an undirected graph:
If 1 is connected to 11,
11 is connected to 1!

```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

Fill out the **adjacency matrix** for the following graph



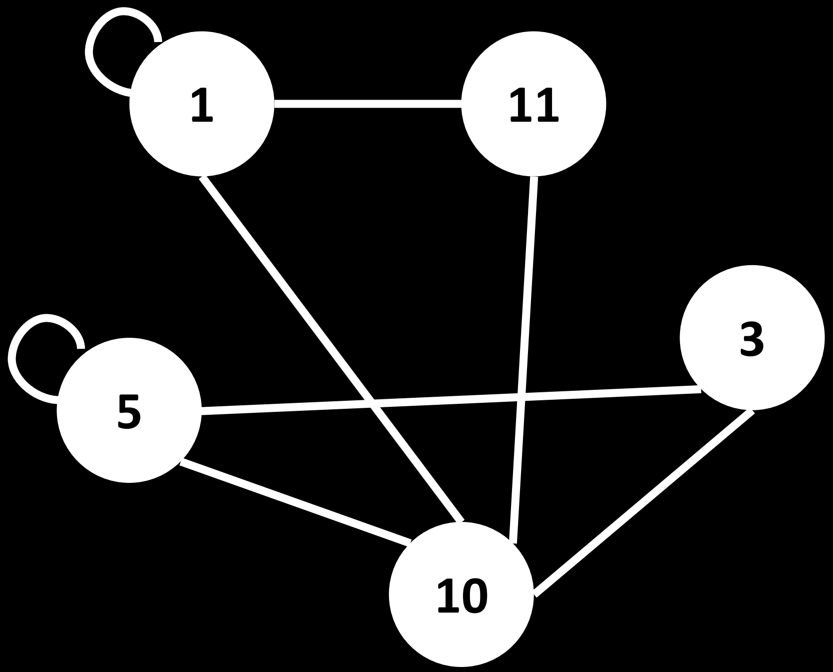
	1	11	3	10	5
1	1	1	0	1	0
11	1	0	0	1	0
3	0	0	0	1	1
10	1	1	1	0	1
5	0	0	1	1	1

For an undirected graph:
If 1 is connected to 11,
11 is connected to 1!

```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

Fill out the **adjacency matrix** for the following graph



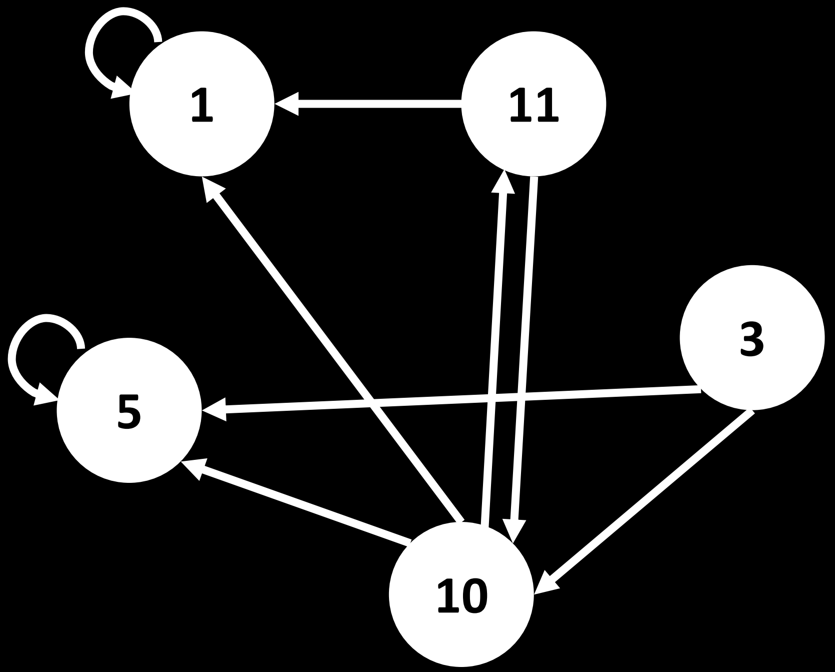
	1	11	3	10	5
1	1	1	0	1	0
11	1	0	0	1	0
3	0	0	0	1	1
10	1	1	1	0	1
5	0	0	1	1	1

The transpose of the adjacency matrix is equal to itself.

```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

Fill out the **adjacency matrix** for the following graph

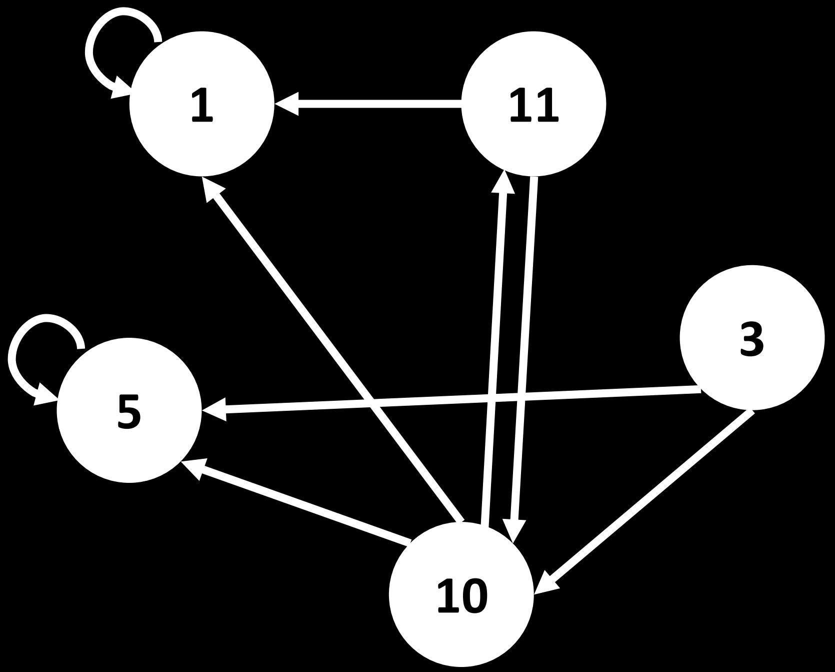


	1	11	3	10	5
1					
11					
3					
10					
5					

```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

Fill out the **adjacency matrix** for the following graph

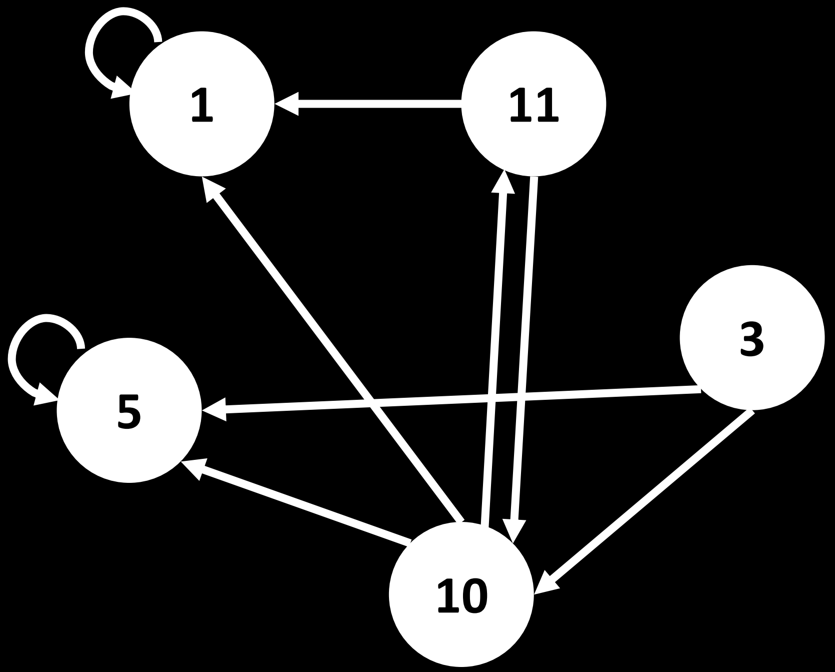


	1	11	3	10	5
1	1	0	0	0	0
11					
3					
10					
5					

Implementing Graphs

```
vals = [1, 11, 3, 10, 5]
```

Fill out the **adjacency matrix** for the following graph

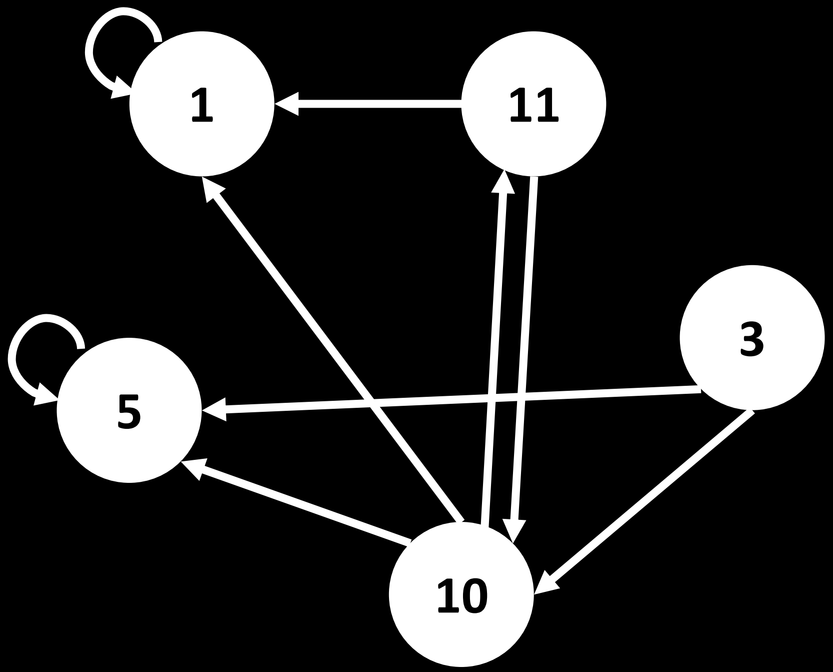


	1	11	3	10	5
1	1	0	0	0	0
11	1	0	0	1	0
3					
10					
5					

```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

Fill out the **adjacency matrix** for the following graph

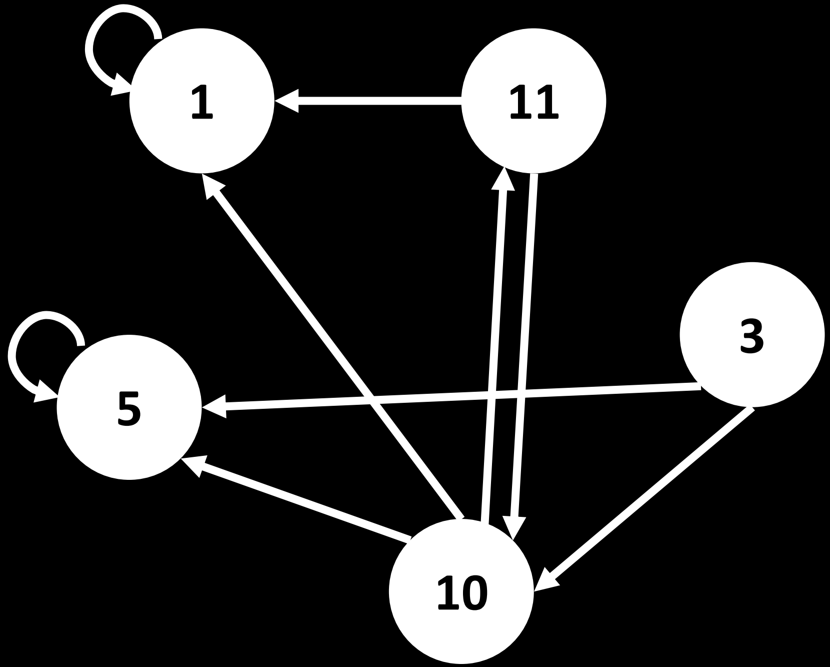


	1	11	3	10	5
1	1	0	0	0	0
11	1	0	0	1	0
3	0	0	0	1	1
10					
5					

```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

Fill out the **adjacency matrix** for the following graph

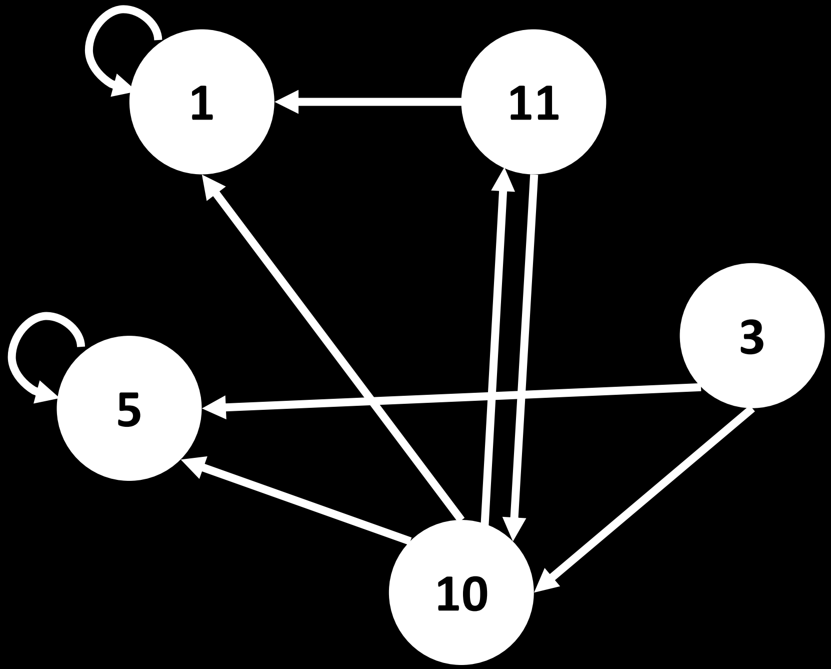


	1	11	3	10	5
1	1	0	0	0	0
11	1	0	0	1	0
3	0	0	0	1	1
10	0	1	0	0	1
5					

```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

Fill out the **adjacency matrix** for the following graph

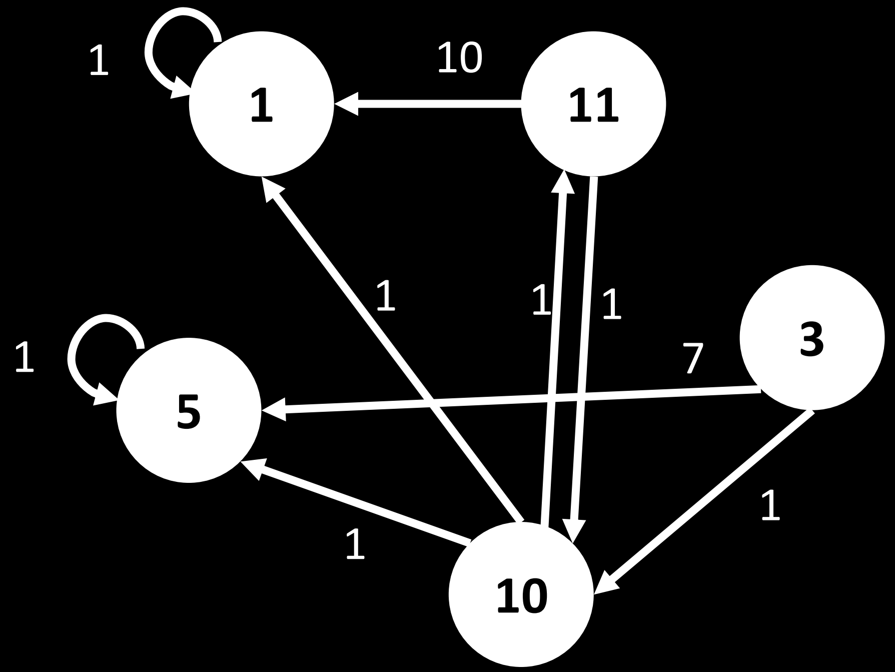


	1	11	3	10	5
1	1	0	0	0	0
11	1	0	0	1	0
3	0	0	0	1	1
10	0	1	0	0	1
5	0	0	0	0	1

```
vals = [1, 11, 3, 10, 5]
```

Implementing Graphs

Fill out the **adjacency matrix** for the following graph



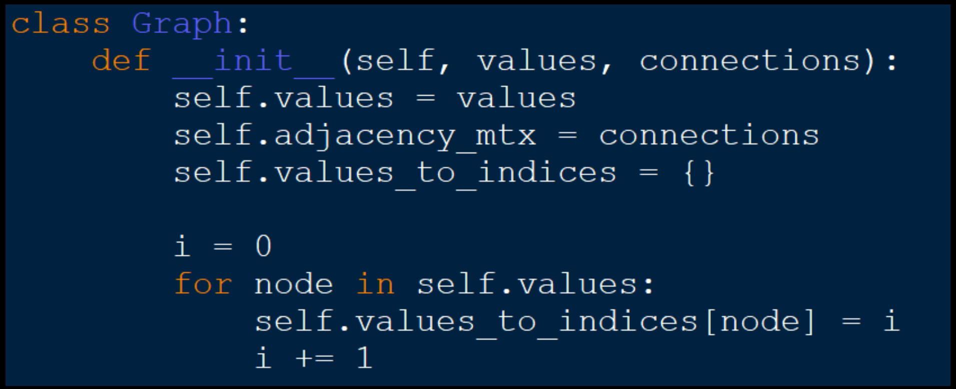
	1	11	3	10	5
1	1	0	0	0	0
11	10	0	0	1	0
3	0	0	0	1	7
10	0	1	0	0	1
5	0	0	0	0	1

Implementing DFS

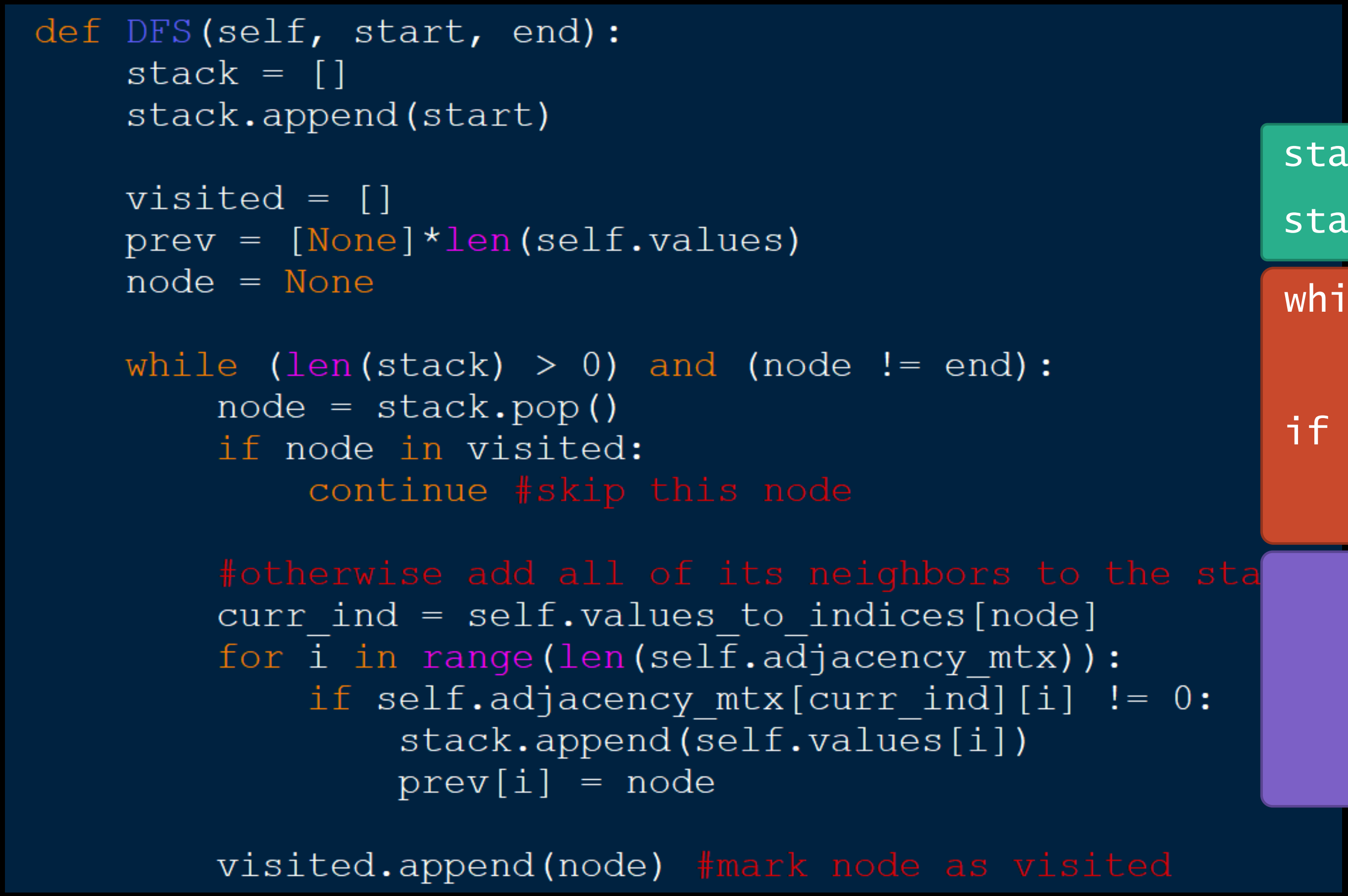
Depth-First Search PseudoCode

```
stack = []
stack.push(starting_node)

while (stack not empty) and (target not found):
    node = stack.pop()
    if node is visited:
        skip
    if node = target:
        target = found
    for all unvisited neighbors of node:
        neighbor.prev = node
        stack.push(neighbor)
    node.visited = True
```

Implementing DFS



```python
class Graph:
    def __init__(self, values, connections):
        self.values = values
        self.adjacency_mtx = connections
        self.values_to_indices = {}

        i = 0
        for node in self.values:
            self.values_to_indices[node] = i
            i += 1
```



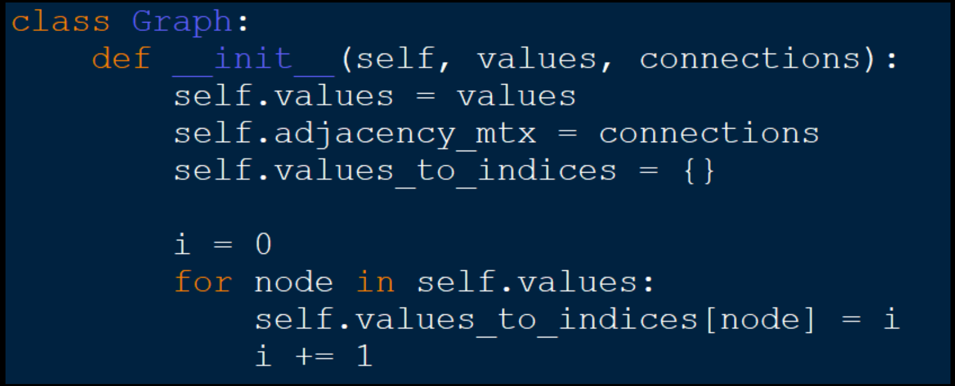
```python
def DFS(self, start, end):
    stack = []
    stack.append(start)

    visited = []
    prev = [None]*len(self.values)
    node = None

    while (len(stack) > 0) and (node != end):
        node = stack.pop()
        if node in visited:
            continue #skip this node

        #otherwise add all of its neighbors to the sta
        curr_ind = self.values_to_indices[node]
        for i in range(len(self.adjacency_mtx)):
            if self.adjacency_mtx[curr_ind][i] != 0:
                stack.append(self.values[i])
                prev[i] = node

        visited.append(node) #mark node as visited
```

```
stack = []
stack.push(starting_node)
```

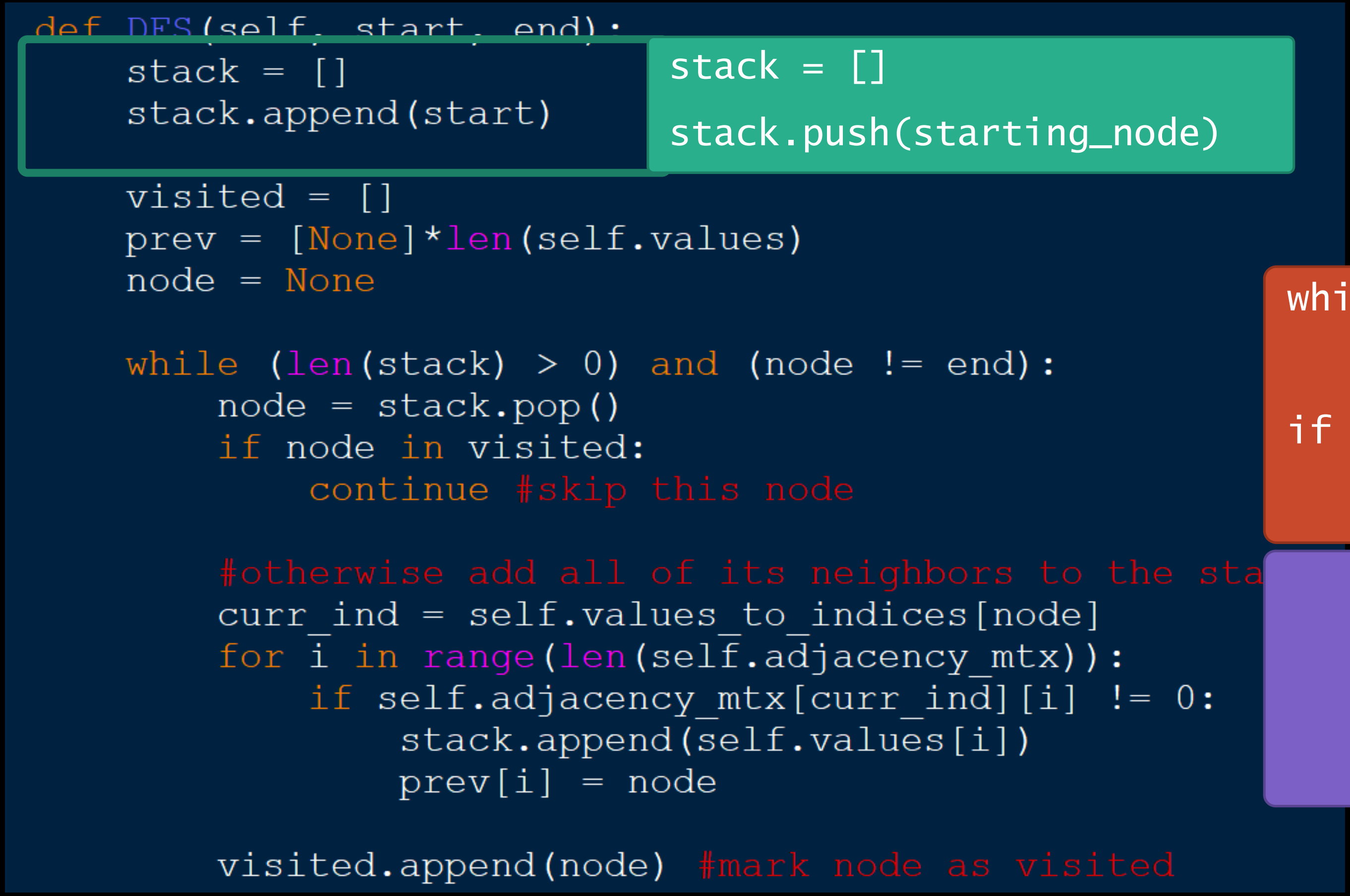
```
while (stack not empty) and (target not found):
        node = stack.pop()
if node is visited:
        skip
```

```
        for all unvisited neighbors of node:
                stack.push(neighbor)
                neighbor.prev = node
        node.visited = True
```

Implementing DFS



```
class Graph:
    def __init__(self, values, connections):
        self.values = values
        self.adjacency_mtx = connections
        self.values_to_indices = {}

        i = 0
        for node in self.values:
            self.values_to_indices[node] = i
            i += 1
```



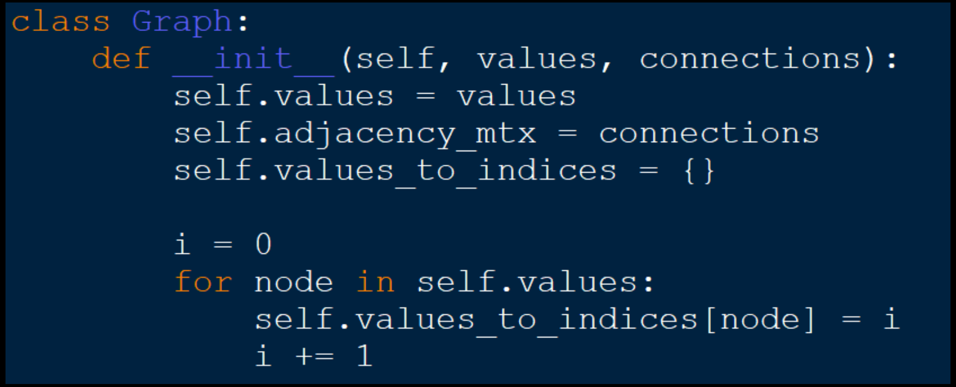
```
def DFS(self, start, end):
    stack = []
    stack.append(start)

    visited = []
    prev = [None]*len(self.values)
    node = None

    while (len(stack) > 0) and (node != end):
        node = stack.pop()
        if node in visited:
            continue #skip this node

        #otherwise add all of its neighbors to the sta
        curr_ind = self.values_to_indices[node]
        for i in range(len(self.adjacency_mtx)):
            if self.adjacency_mtx[curr_ind][i] != 0:
                stack.append(self.values[i])
                prev[i] = node

        visited.append(node) #mark node as visited
```

```
stack = []
stack.push(starting_node)
```

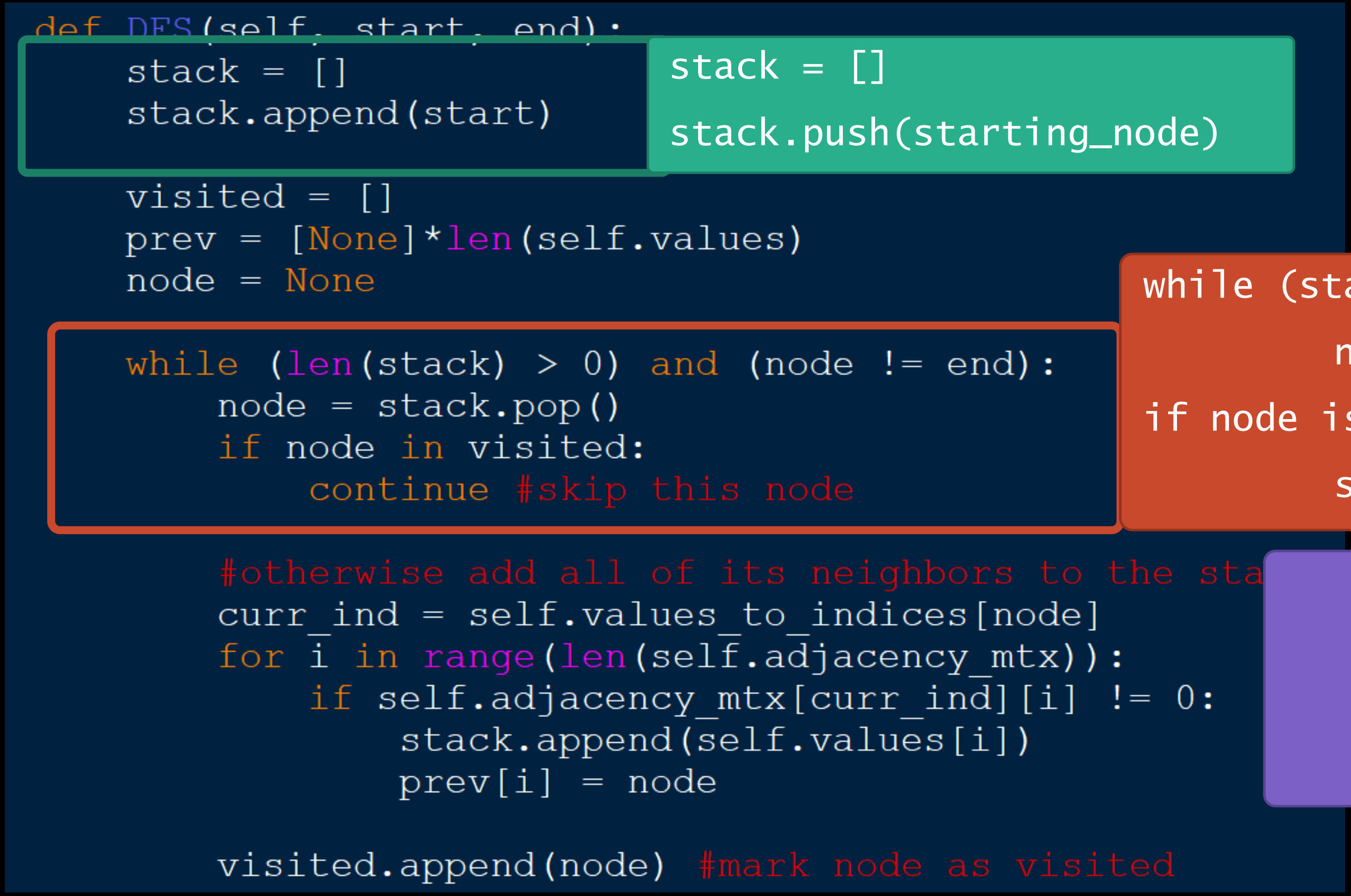
```
while (stack not empty) and (target not found):
        node = stack.pop()
if node is visited:
        skip
```

```
        for all unvisited neighbors of node:
                stack.push(neighbor)
                neighbor.prev = node
        node.visited = True
```

Implementing DFS



```python
class Graph:
    def __init__(self, values, connections):
        self.values = values
        self.adjacency_mtx = connections
        self.values_to_indices = {}

        i = 0
        for node in self.values:
            self.values_to_indices[node] = i
            i += 1
```



```python
def DFS(self, start, end):
    stack = []
    stack.append(start)

    visited = []
    prev = [None]*len(self.values)
    node = None

    while (len(stack) > 0) and (node != end):
        node = stack.pop()
        if node in visited:
            continue #skip this node

        #otherwise add all of its neighbors to the sta
        curr_ind = self.values_to_indices[node]
        for i in range(len(self.adjacency_mtx)):
            if self.adjacency_mtx[curr_ind][i] != 0:
                stack.append(self.values[i])
                prev[i] = node

        visited.append(node) #mark node as visited
```

```
stack = []
stack.push(starting_node)
```

```
while (stack not empty) and (target not found):
        node = stack.pop()
if node is visited:
        skip
```

```
        for all unvisited neighbors of node:
                stack.push(neighbor)
                neighbor.prev = node
        node.visited = True
```

Implementing DFS

```
class Graph:
    def __init__(self, values, connections):
        self.values = values
        self.adjacency_mtx = connections
        self.values_to_indices = {}

        i = 0
        for node in self.values:
            self.values_to_indices[node] = i
            i += 1
```

```
def DFS(self, start, end):
    stack = []
    stack.append(start)

    visited = []
    prev = [None]*len(self.values)
    node = None

    while (len(stack) > 0) and (node != end):
        node = stack.pop()
        if node in visited:
            continue #skip this node

        #otherwise add all of its neighbors to the sta
        curr_ind = self.values_to_indices[node]
        for i in range(len(self.adjacency_mtx)):
            if self.adjacency_mtx[curr_ind][i] != 0:
                stack.append(self.values[i])
                prev[i] = node

        visited.append(node) #mark node as visited
```

```
stack = []
stack.push(starting_node)
```

```
while (stack not empty) and (target not found):
        node = stack.pop()
if node is visited:
        skip
```

```
for all unvisited neighbors of node:
        stack.push(neighbor)
        neighbor.prev = node
node.visited = True
```

Steps to DS & A design & implementation

1. Choose a structure to impose on your data
   - Exploit known qualities or desired outcome
   - Reference classic abstract data types: Graphs ⊃ Trees ⊃ Heaps
   - And corresponding algorithms: BFS/DFS, Traversals, Heap operations
2. Design an algorithm – discretized steps for the machine to take
3. Write pseudocode for the algorithm
4. Code the algorithm in a specific language, implementing the ADT using a data structure which makes sense for your application

Homework

- Implement BFS
- Implement a visualizer to ensure the BFS/DFS are working as expected
- You will implement Djikstra’s for Lab 1

What’s next?

- Time complexity analysis of BFS and DFS