Lecture 18: Greedy Algorithms + Midterm Review

Tim LaRock

larock.t@northeastern.edu

bit.ly/cs3000syllabus

Business

Homework 5 due tonight at midnight Boston time, solutions will be released tomorrow morning

No class tomorrow, midterm review moved to today

Extra credit assignment available as of yesterday

- Optional
- 6 points on the final exam
- Available until Sunday June 21st

Midterm 2 to be released tomorrow night, due Friday night

- Topics: Graph algorithms and network flow

Greedy Algorithms

- For some problems, we can think of simple decision making rules that intuitively guide us towards a solution
  - Best-first search: We want to find shortest paths/minimum trees, so only choose edges that can be included in these solutions!
- Applying this idea does not always work as intended!
  - Maximum flow: We tried assigning flow based on best-first search, but we showed that the algorithm will get stuck if it is not able to modify the flow!
- Algorithms that rely on repeatedly making optimal *local decisions* to eventually reach an optimal global solution are called *greedy algorithms*

Example: Files on Tape

Before any of us were born, computers used to exist on magnetic tape.

Imagine we have such a tape, split in to segments we will call “blocks”, where each block contains data from a single file. Each file is referred to by an integer index i, and has length in blocks $L[i]$.

1	1	1	2	2	3	3	3	4	4

To read file k, the tape head needs to first skip all of the files before k. Therefore, the *cost* of accessing file k can be written as

$$cost(k) = \sum_{i=1}^{k} L[i]$$

Example: Files on Tape

1	1	1	2	2	3	3	3	4	4

Assuming all files are equally likely to be accessed, we can write the *expected* (equivalently, average) cost of accessing file *k* as

$$\mathbb{E}[cost] = \frac{1}{n}\sum_{i=1}^{n} cost(i) = \frac{1}{n}\sum_{k=1}^{n}\sum_{i=1}^{k} L[i]$$

$$\mathbb{E}[cost] = \frac{1}{4} \cdot (cost(1) + cost(2) + cost(3) + cost(4))$$

Example: Files on Tape

$$= \frac{1}{4} \cdot (3 + 5 + 8 + 10) = \frac{26}{4}$$

1	1	1	2	2	3	3	3	4	4

Assuming all files are equally likely to be accessed, we can write the *expected* (equivalently, average) cost of accessing file *k* as

$$\mathbb{E}[cost] = \frac{1}{n}\sum_{i=1}^{n} cost(i) = \frac{1}{n}\sum_{k=1}^{n}\sum_{i=1}^{k} L[i]$$

What order should we keep the files in?

1	1	1	2	2	3	3	3	4	4

$\mathbb{E}[cost] = \frac{26}{4}$

We can modify the order of the files on the tape, resulting in a permutation π where $\pi(i)$ returns the index of the file in the ith block. We can then rewrite the *expected* (average) cost of accessing file *k* as

$$\mathbb{E}[cost(\pi)] = \frac{1}{n}\sum_{k=1}^{n}\sum_{i=1}^{k} L[\pi(i)]$$

Intuitively: To minimize average cost, we should store the smallest files first, otherwise we will need to unnecessarily spend time skipping the large files to read smaller ones!

2	2	4	4	1	1	1	3	3	3

What order should we keep the files in?

1	1	1	2	2	3	3	3	4	4

$\mathbb{E}[cost] = \frac{26}{4}$

We can modify the order of the files on the tape, resulting in a permutation π where $\pi(i)$ returns the index of the file in the ith block. We can then rewrite the *expected* (average) cost of accessing file *k* as

$$\mathbb{E}[cost(\pi)] = \frac{1}{n}\sum_{k=1}^{n}\sum_{i=1}^{k} L[\pi(i)]$$

Intuitively: To minimize average cost, we should store the smallest files first, otherwise we will need to unnecessarily spend time skipping the large files to read smaller ones!

2	2	4	4	1	1	1	3	3	3

$$\mathbb{E}[cost(\pi)] = \frac{2+4+7+10}{4} = \frac{23}{4}$$

Greedy Algorithm for Storing Files

Input: A set of files labeled $1 \ldots n$ with lengths $L[i]$
Output: An ordering of the files on the tape

Repeat until all files are on the tape:

1. Find the unwritten file with minimum length (break ties arbitrarily)
2. Write that file to the tape

Greedy Algorithm for Storing Files

Input: A set of files labeled $1 \ldots n$ with lengths $L[i]$
Output: An ordering of the files on the tape

Repeat until all files are on the tape:

1. Find the unwritten file with minimum length (break ties arbitrarily)
2. Write that file to the tape

How can we show this is optimal?

Proof of optimality

1	1	1	2	2	3	3	3	4	4

Claim: $\mathbb{E}[cost(\pi)]$ is minimized when $L[\pi(i)] \leq L[\pi(i+1)]$ for all i.

Proof of optimality

1	1	1	2	2	3	3	3	4	4

Claim: $\mathbb{E}[cost(\pi)]$ is minimized when $L[\pi(i)] \leq L[\pi(i+1)]$ for all i.

Proof:

Let $\mathrm{a} = \pi(i)$ and $b = \pi(i+1)$ and suppose $L[a] > L[b]$ for some index i.

Proof of optimality

1	1	1	2	2	3	3	3	4	4

Claim: $\mathbb{E}[cost(\pi)]$ is minimized when $L[\pi(i)] \leq L[\pi(i+1)]$ for all i.

Proof:

Let $\mathrm{a} = \pi(i)$ and $b = \pi(i+1)$ and suppose $L[a] > L[b]$ for some index i.

If we swap the files a and b on the tape, then the cost of accessing a increases by $L[b]$ and the cost of accessing b decreases by $L[a]$.

Proof of optimality

1	1	1	2	2	3	3	3	4	4

Claim: $\mathbb{E}[cost(\pi)]$ is minimized when $L[\pi(i)] \leq L[\pi(i+1)]$ for all i.

Proof:

Let $\mathrm{a} = \pi(i)$ and $b = \pi(i+1)$ and suppose $L[a] > L[b]$ for some index i.

If we swap the files a and b on the tape, then the cost of accessing a increases by $L[b]$ and the cost of accessing b decreases by $L[a]$.

Overall, the swap changes the expected cost by $\frac{L[b]-L[a]}{n}$.

Proof of optimality

1	1	1	2	2	3	3	3	4	4

Claim: $\mathbb{E}[cost(\pi)]$ is minimized when $L[\pi(i)] \leq L[\pi(i+1)]$ for all i.

Proof:

Let a $= \pi(i)$ and $b = \pi(i+1)$ and suppose $L[a] > L[b]$ for some index i.

If we swap the files a and b on the tape, then the cost of accessing a increases by $L[b]$ and the cost of accessing b decreases by $L[a]$.

Overall, the swap changes the expected cost by $\frac{L[b]-L[a]}{n}$.

This change represents an improvement because $L[b] < L[a]$.

Proof of optimality

1	1	1	2	2	3	3	3	4	4

Claim: $\mathbb{E}[cost(\pi)]$ is minimized when $L[\pi(i)] \leq L[\pi(i+1)]$ for all i.

Proof:

Let a $= \pi(i)$ and $b = \pi(i+1)$ and suppose $L[a] > L[b]$ for some index i.

If we swap the files a and b on the tape, then the cost of accessing a increases by $L[b]$ and the cost of accessing b decreases by $L[a]$.

Overall, the swap changes the expected cost by $\frac{L[b]-L[a]}{n}$.

This change represents an improvement because $L[b] < L[a]$.

Average cost for example above: $\frac{26}{4}$

Average cost after swapping files 1 and 2: $\frac{1}{4}(2+5+8+10) = \frac{25}{4}$

$$\frac{26}{4} + \frac{2-3}{4} = \frac{26-1}{4} = \frac{25}{4}$$

Proof of optimality

1	1	1	2	2	3	3	3	4	4

Claim: $\mathbb{E}[cost(\pi)]$ is minimized when $L[\pi(i)] \leq L[\pi(i+1)]$ for all i.

Proof:

Let $\mathrm{a} = \pi(i)$ and $b = \pi(i+1)$ and suppose $L[a] > L[b]$ for some index i.

If we swap the files a and b on the tape, then the cost of accessing a increases by $L[b]$ and the cost of accessing b decreases by $L[a]$.

Overall, the swap changes the expected cost by $\frac{L[b]-L[a]}{n}$.

This change represents an improvement because $L[b] < L[a]$.

Thus, if the files are out of length-order, we can decrease expected cost by swapping pairs to put them in order.

Wrap-up

Greedy algorithms repeatedly apply a simple rule to eventually find an optimal solution

Inductive Exchange Arguments are strategies for proving correctness of some greedy algorithms

Next Week:

- Data Compression with Huffman Codes
- Proof strategies for greedy algorithms
  - Inductive exchange
  - Greedy-stays-ahead

Midterm 2 Review/Q&A

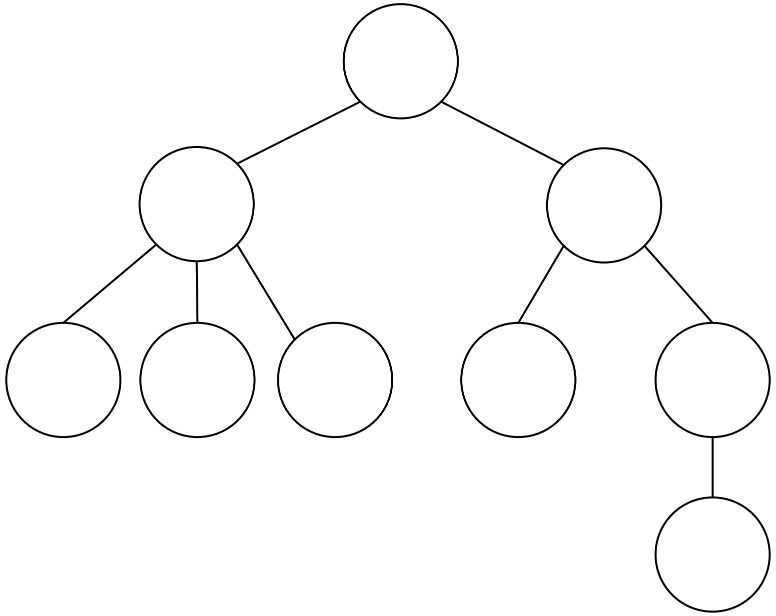
Topics

- Graph Algorithms
  - Reachability, connectivity, graph traversal
    - DFS and BFS
    - Typology of edges in a whatever-first-search tree
      - tree, forward, backward, cross
    - Post-ordering of nodes in a traversal
  - Topological orderings/Directed Acyclic Graphs (DAGs)
    - Reverse post-ordering is a topological ordering iff the graph is a DAG!
  - Shortest paths
    - Using BFS/DFS or Dijkstra (best-first-search)
    - Single-source vs. all-pairs
    - Betweenness centrality
  - Minimum Spanning Trees
    - Cut property and Cycle property
    - Boruvka: Add all safe edges across each cut, then recurse
    - Prim: Best first search: Repeatedly add T's safe edge to itself
- Network Flow
  - Max flow/min cut duality
  - Augmenting Paths and the residual graph
  - Ford-Fulkerson algorithm
  - Reduction to many other problems

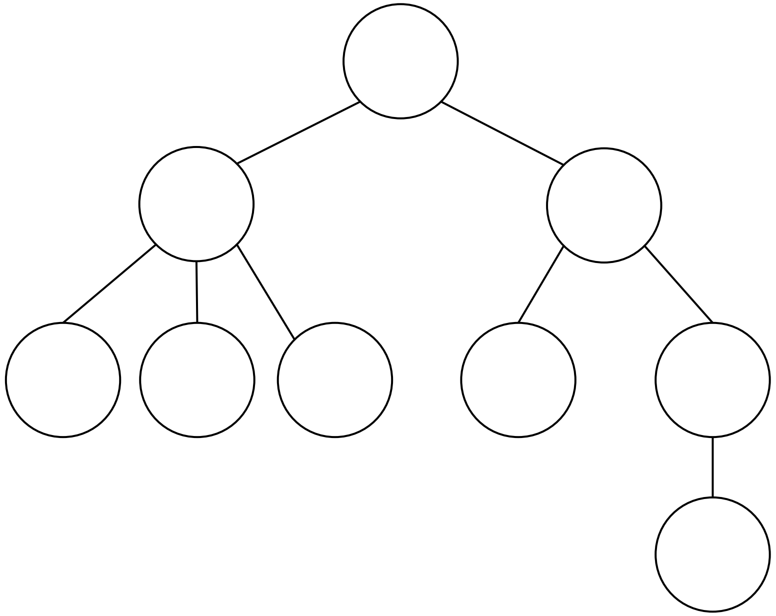
Graph Traversal

Breadth vs. Depth vs. Best (first search)

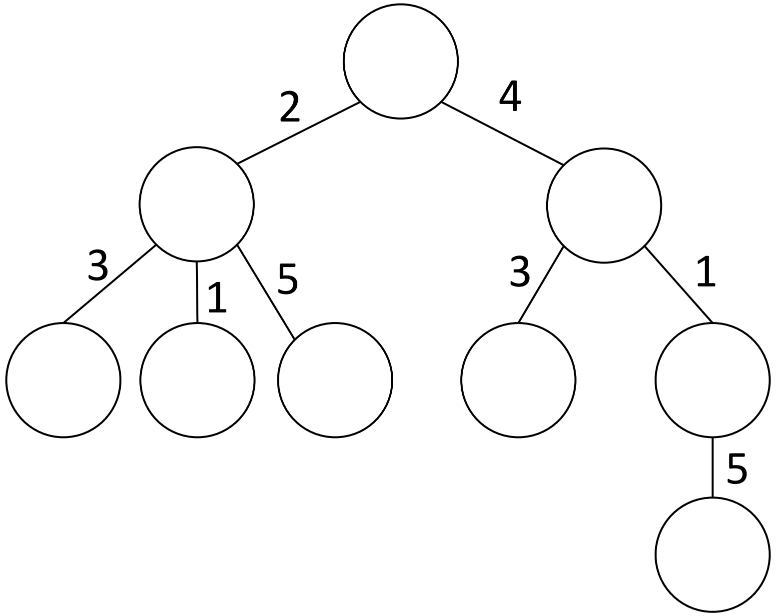
BFS



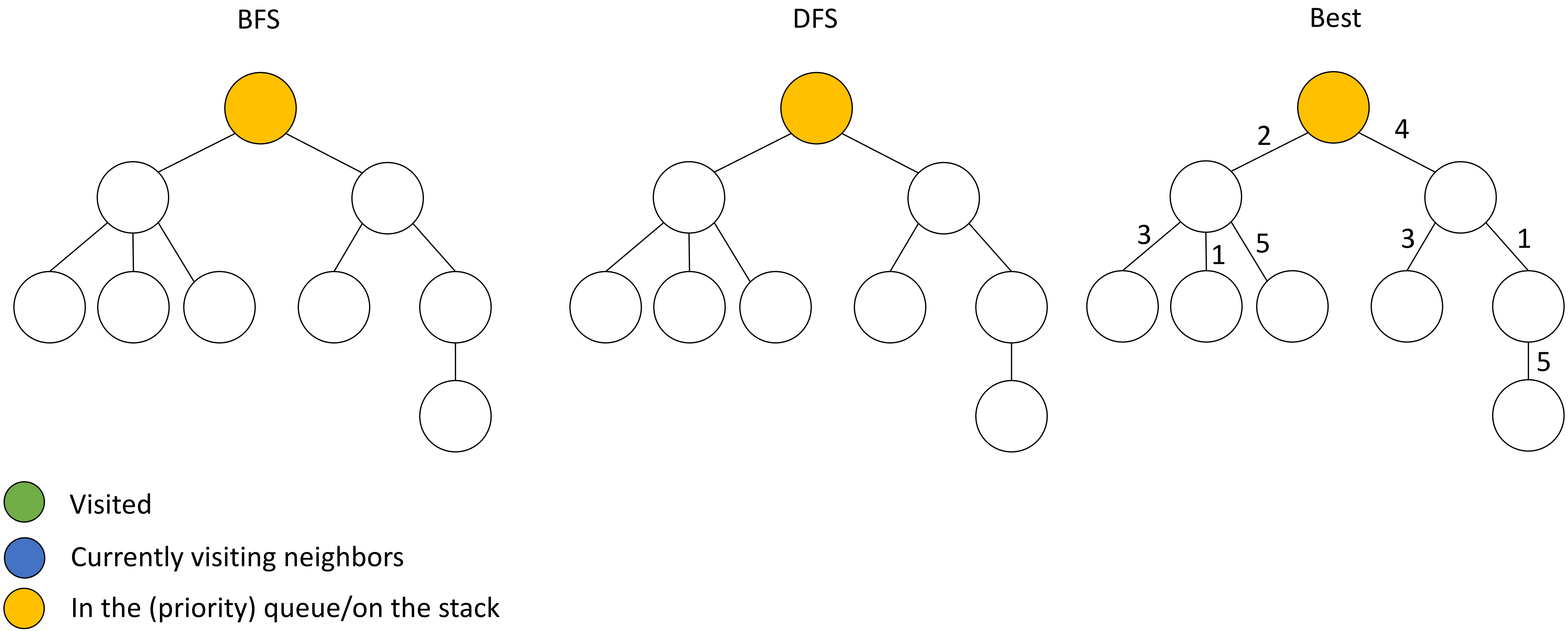
DFS



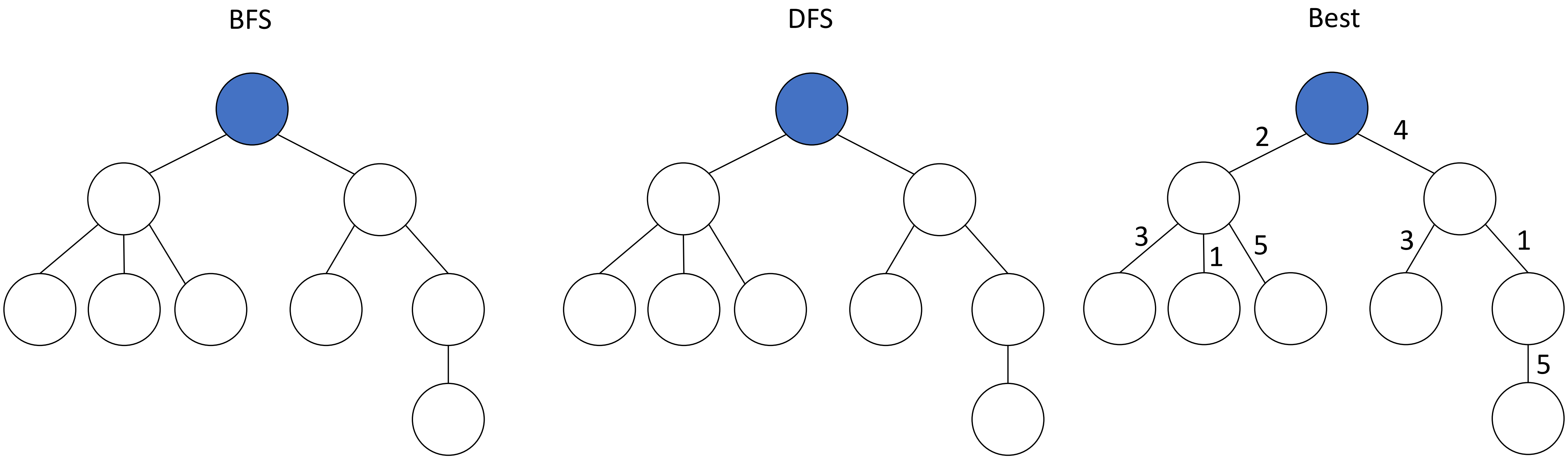
Best



Breadth vs. Depth vs. Best (first search)



Breadth vs. Depth vs. Best (first search)

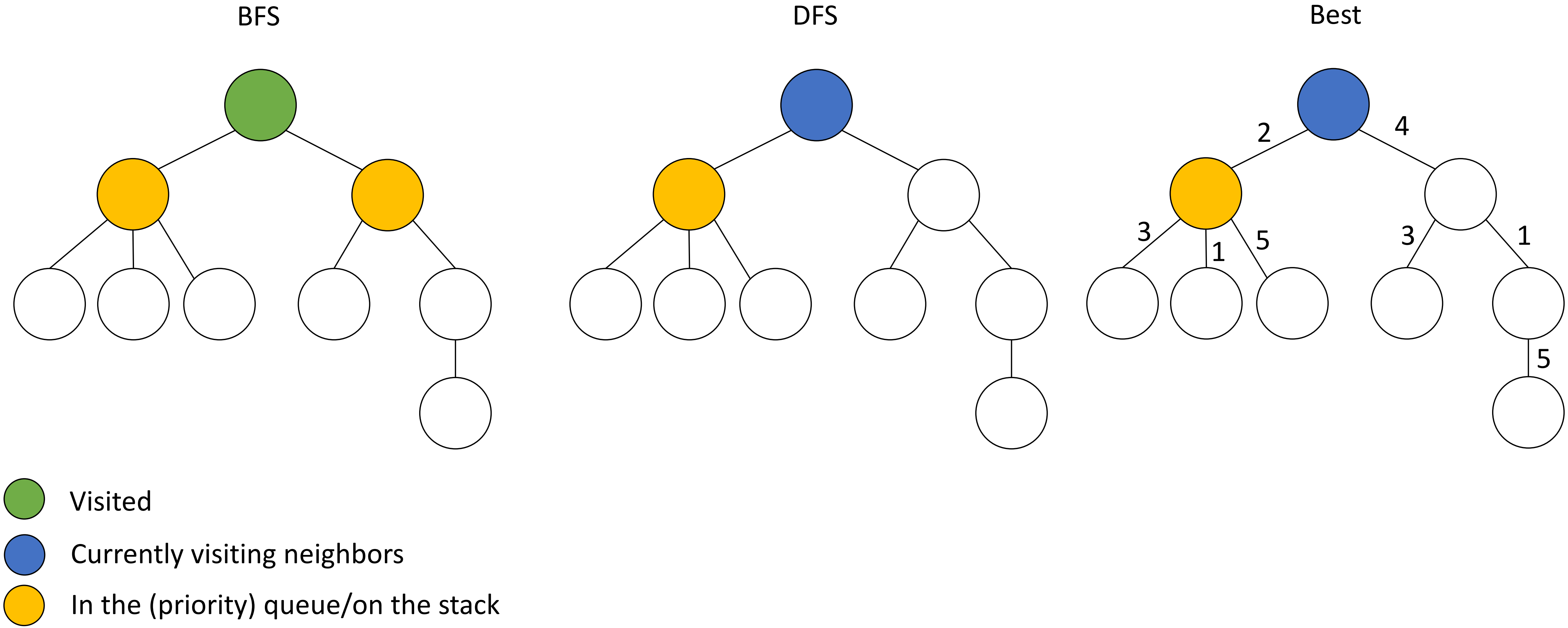


Visited

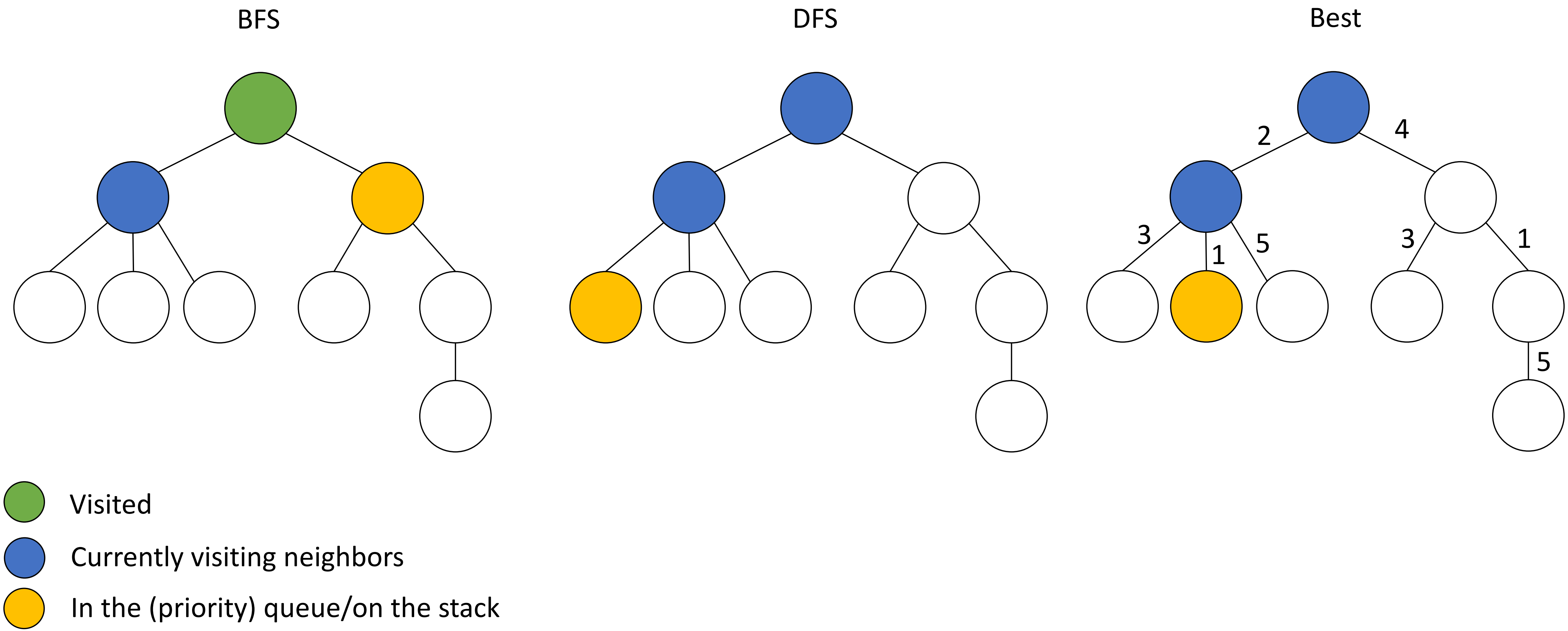
Currently visiting neighbors

In the (priority) queue/on the stack

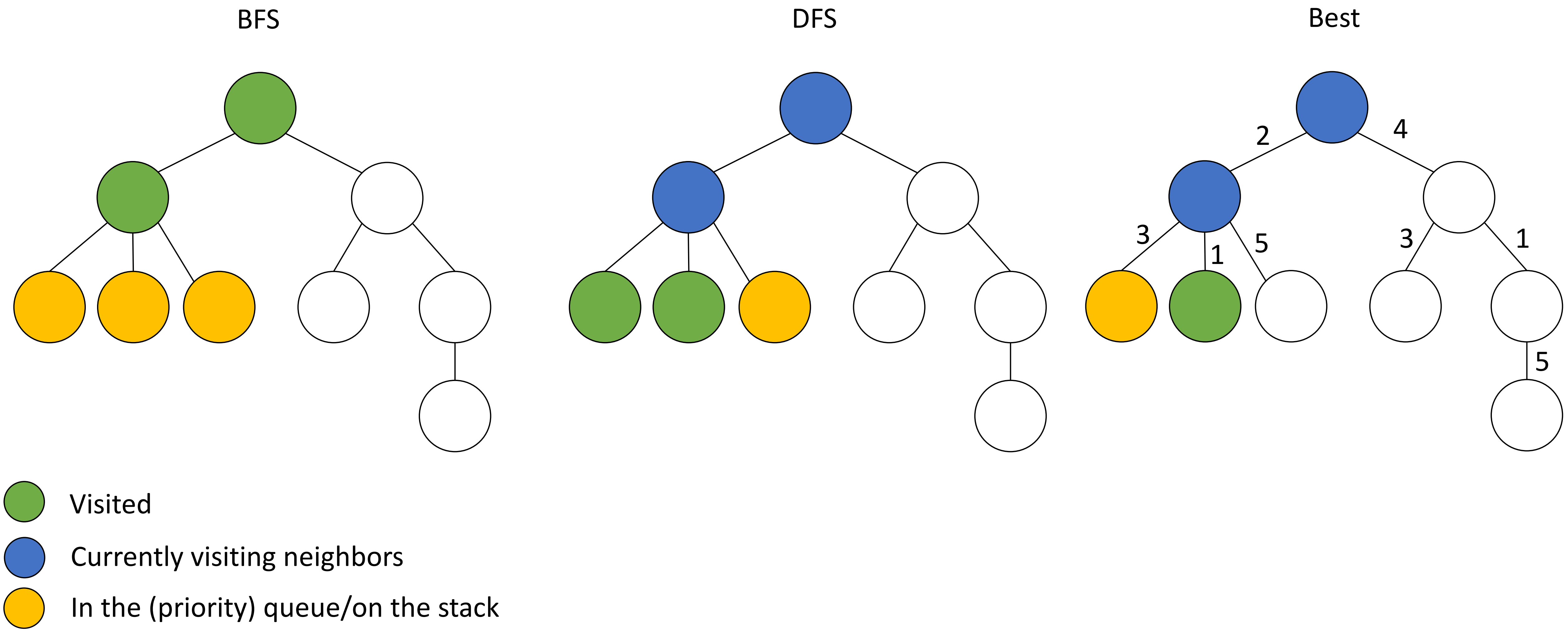
Breadth vs. Depth vs. Best (first search)



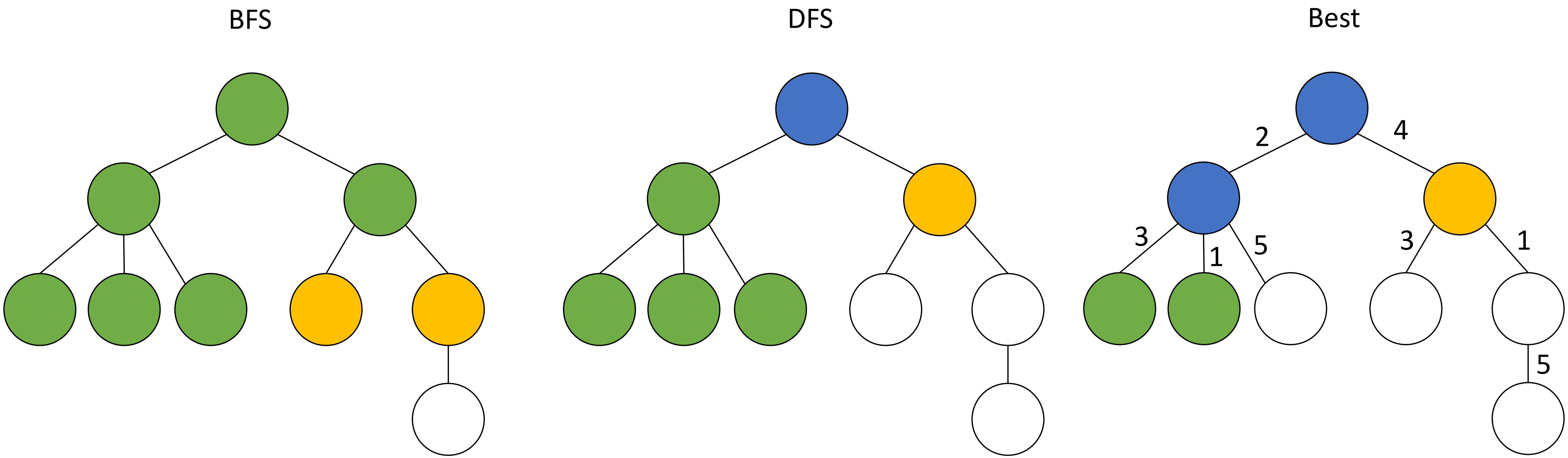
Breadth vs. Depth vs. Best (first search)



Breadth vs. Depth vs. Best (first search)



Breadth vs. Depth vs. Best (first search)

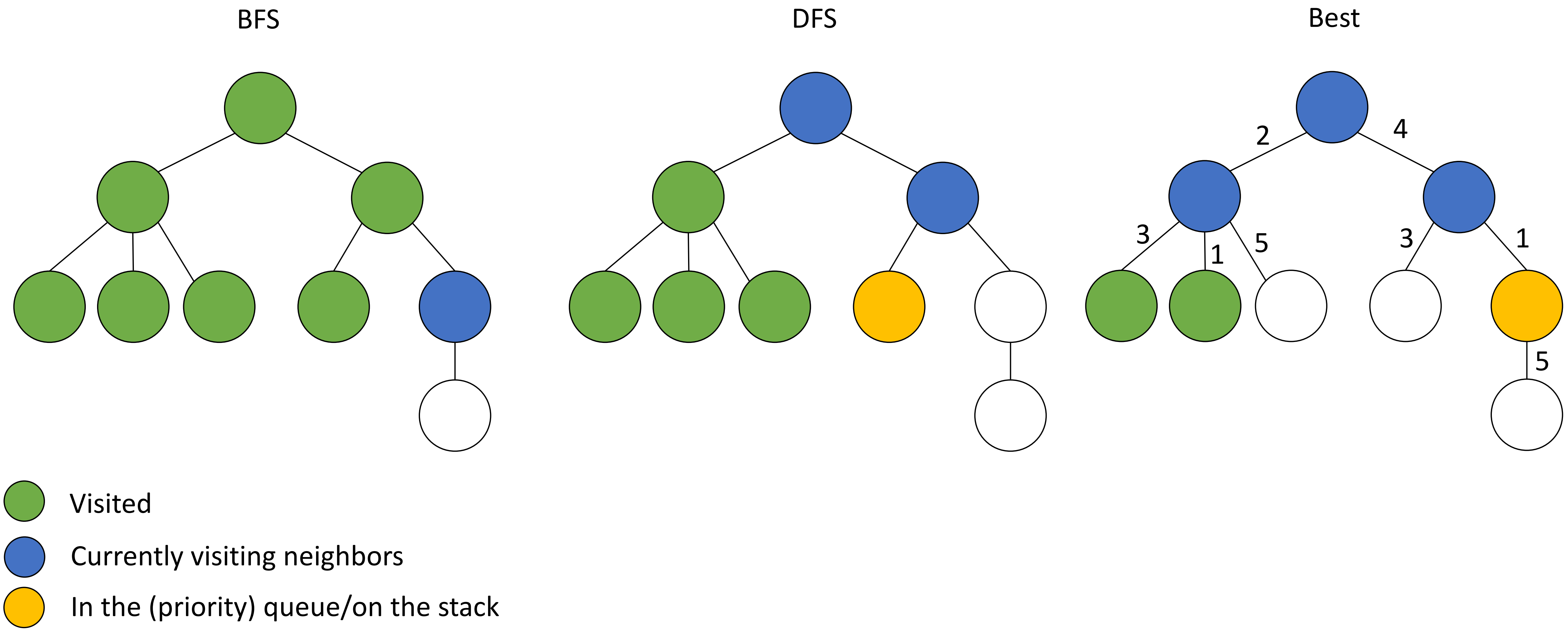


Visited

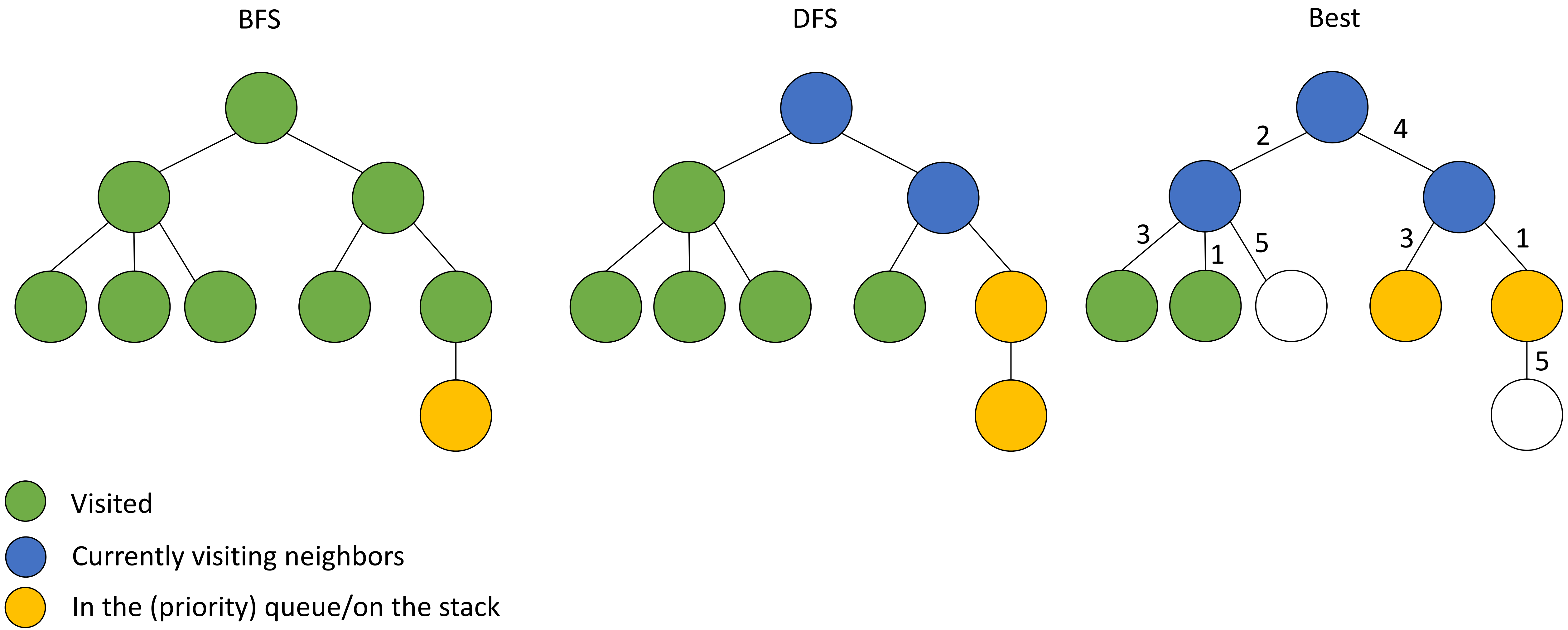
Currently visiting neighbors

In the (priority) queue/on the stack

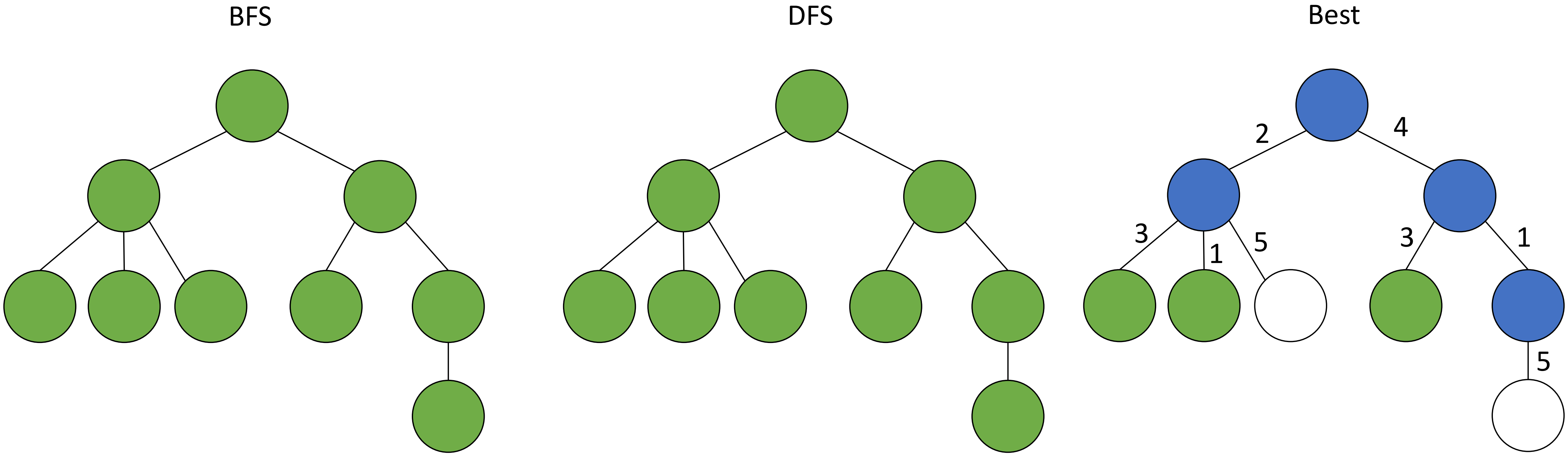
Breadth vs. Depth vs. Best (first search)



Breadth vs. Depth vs. Best (first search)



Breadth vs. Depth vs. Best (first search)



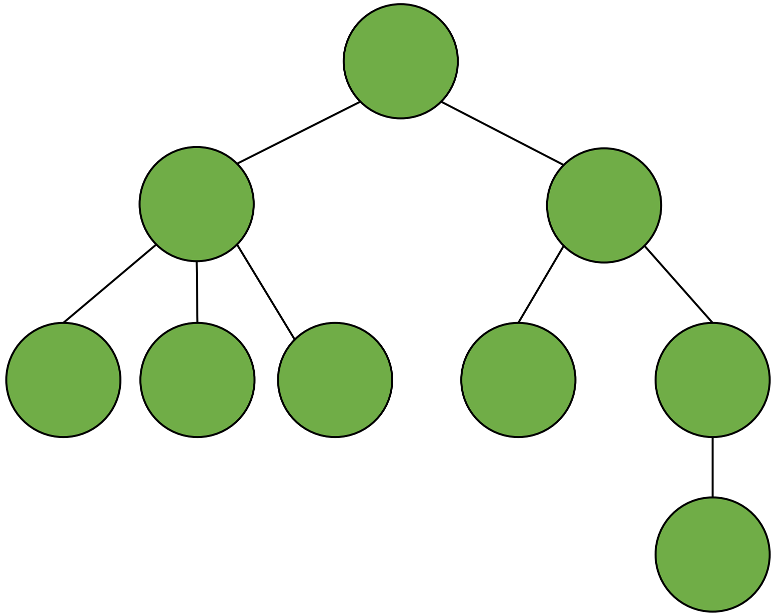
Visited

Currently visiting neighbors

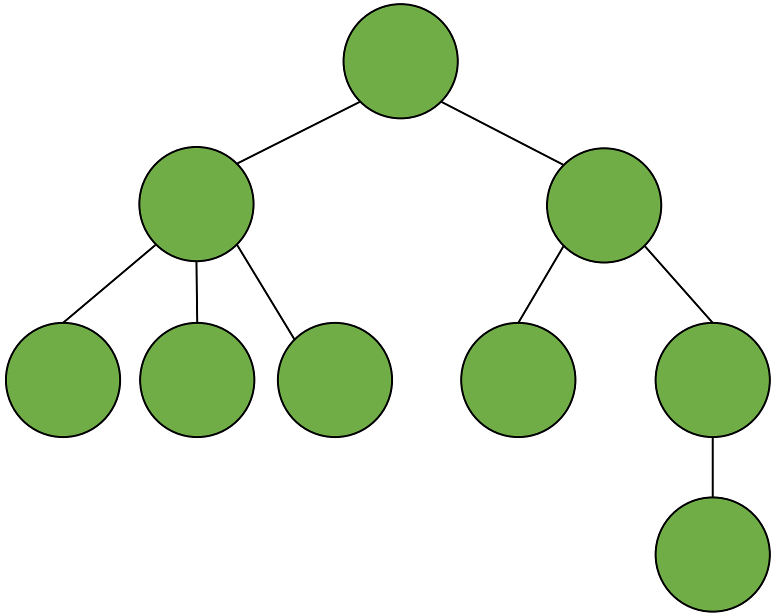
In the (priority) queue/on the stack

Breadth vs. Depth vs. Best (first search)

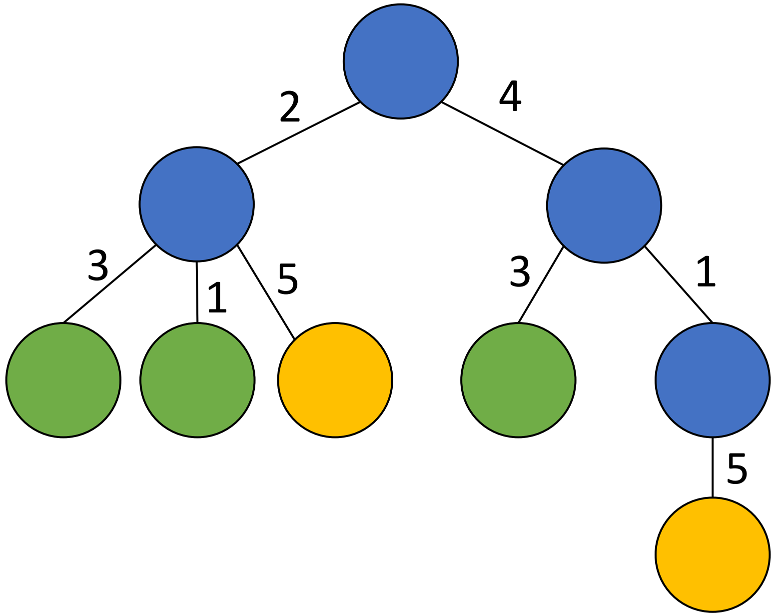
BFS

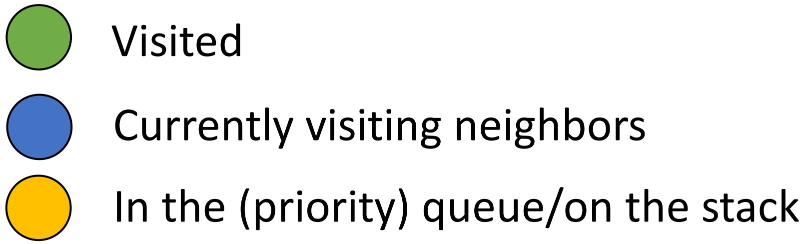


DFS

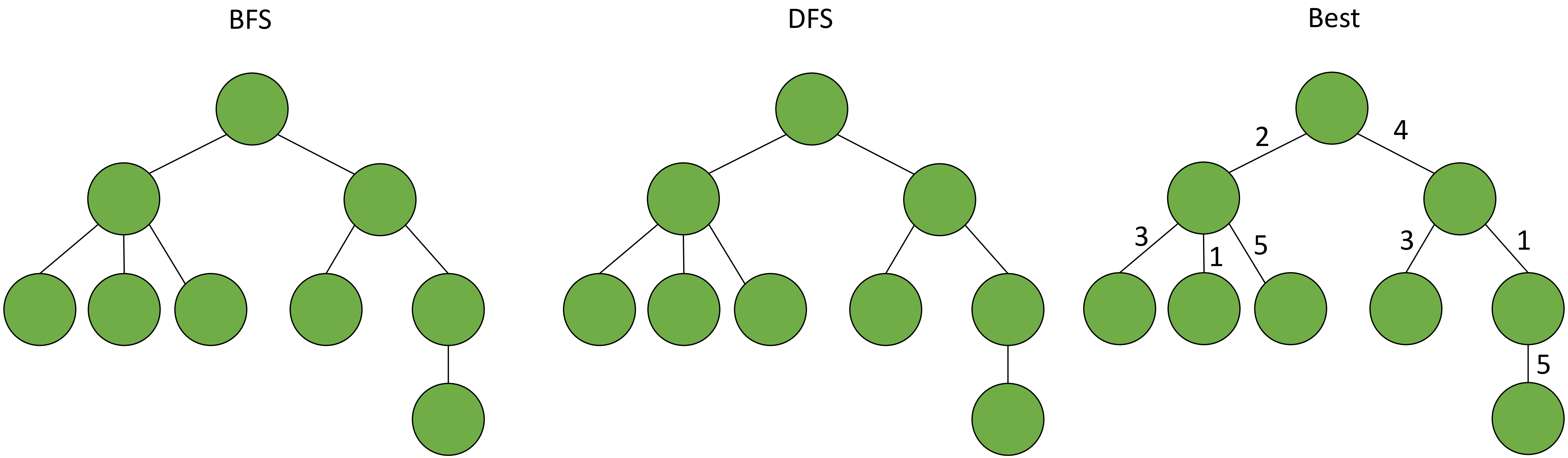


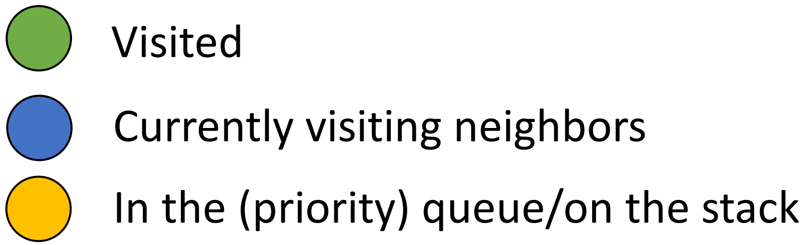
Best





Breadth vs. Depth vs. Best (first search)





Post-Ordering, DAGs, and Topological Ordering

Post-Ordering

A *post-ordering* of a graph $G = (V, E)$ is an ordering of the nodes based on "when" DFS from each node finished.

To get a post-order, we maintain a global clock variable that is initialized to 1.

Every time we finish calling DFS on all of a node's neighbors, we set its post-order value to the current value of clock, then increment clock.

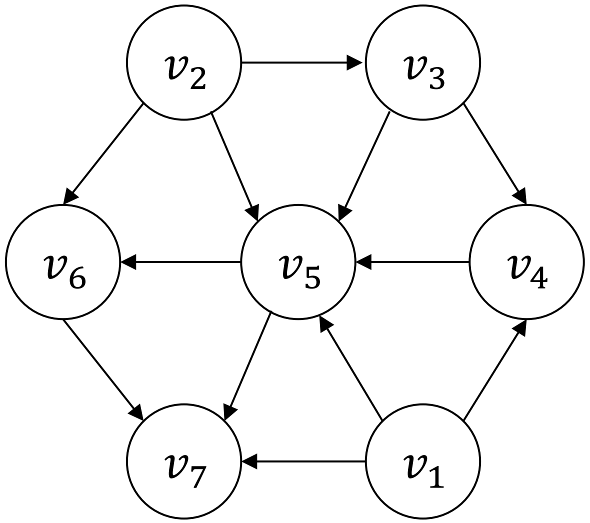
Recursive DFS with post-ordering

```
G = (V,E) is a graph
visited[u] = 0 for all u ∈ V
clock = 1

DFS(u):
  visited[u] = 1
  For v ∈ Neighbors(u):
    If visited[v] = 0:
      parent[v] = u
      DFS(v)

  post-visit(u)
```

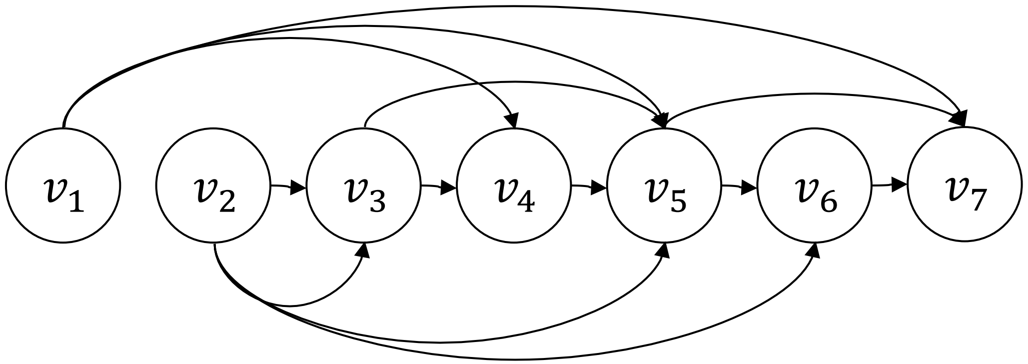
```
post-visit(u):
  set postorder[u] = clock
  clock ← clock + 1
```

Directed Acyclic Graph (DAG)

- A directed graph with no cycles
- Represent precedence relationships
  - “this” comes before “that”
  - “this” is prior to “that”



A *topological ordering* of a directed graph is a labeling of the nodes so that all edges point “forward”, meaning for all directed edges $(v_i, v_j), j > i$

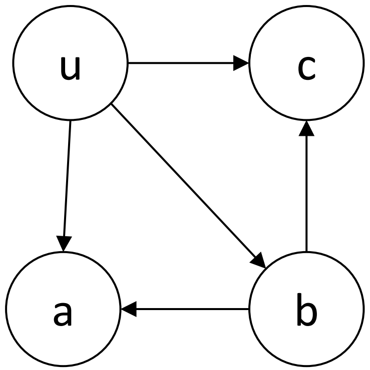


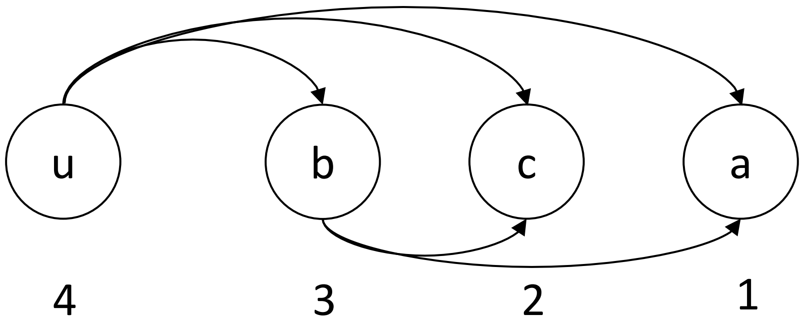
Key point: A reverse post-ordering of the nodes in a DAG is a topological ordering!

Topological Ordering

Ordering nodes by decreasing post-order gives a topological ordering.

Example:





Vertex	u	a	b	c
Postorder	4	1	3	2

Minimum Spanning Trees

Minimum Spanning Trees

A *spanning tree* is a set of edges $T \in E$ is a subgraph of a graph $G = (V, E)$ that (i) is a tree and (ii) contains all of the nodes $v \in V$.

A *minimum spanning tree* for a connected, weighted, undirected graph $G = (V, E, \{w_e\})$, where $w_e \in \mathbb{R}$ is a weight associated with each edge $e \in E$, is a spanning tree T with minimum weight $w(T)$:

$$w(T) = \sum_{e \in T} w_e$$

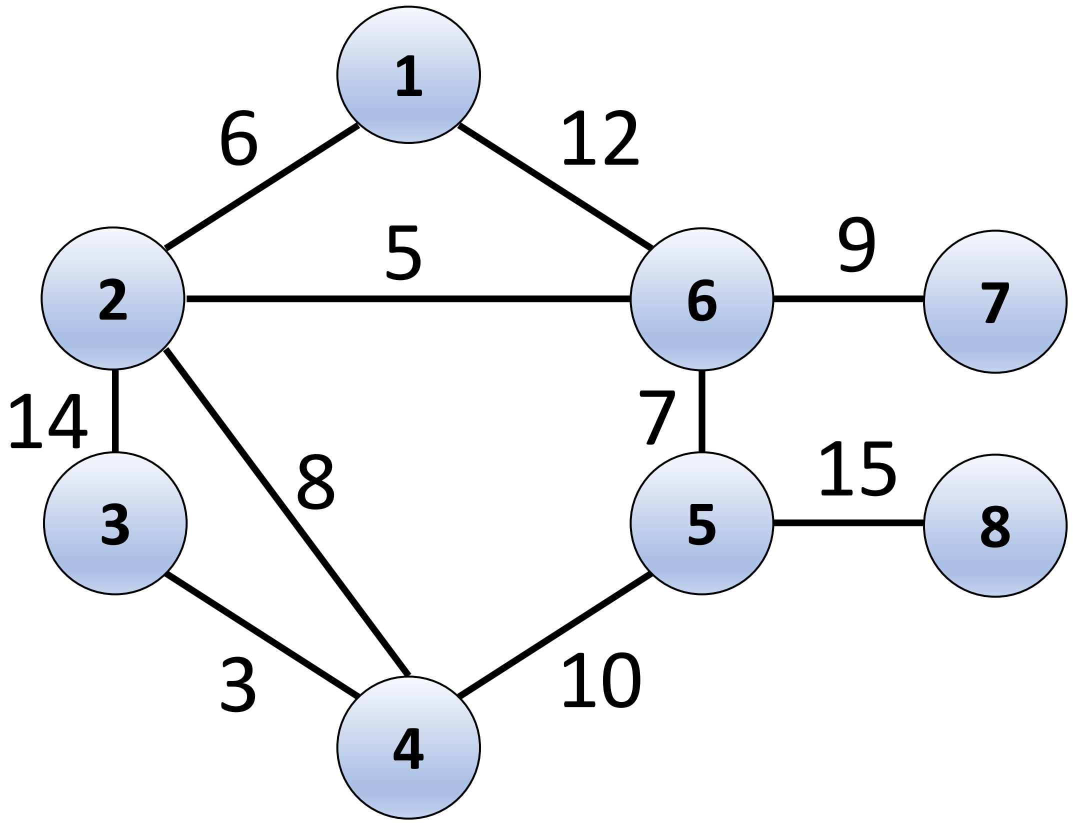
Borůvka's Algorithm

- **Borůvka:**
  - Let $T = \emptyset$
  - Repeat until T is connected:
    - Let $C_1, \dots, C_k$ be the connected components of (V, T)
    - Let $e_1, \dots, e_k$ be the safe edge for the cuts $C_1, \dots, C_m$
    - Add $e_1, \dots, e_k$ to T

- **Correctness:** every edge we add is safe

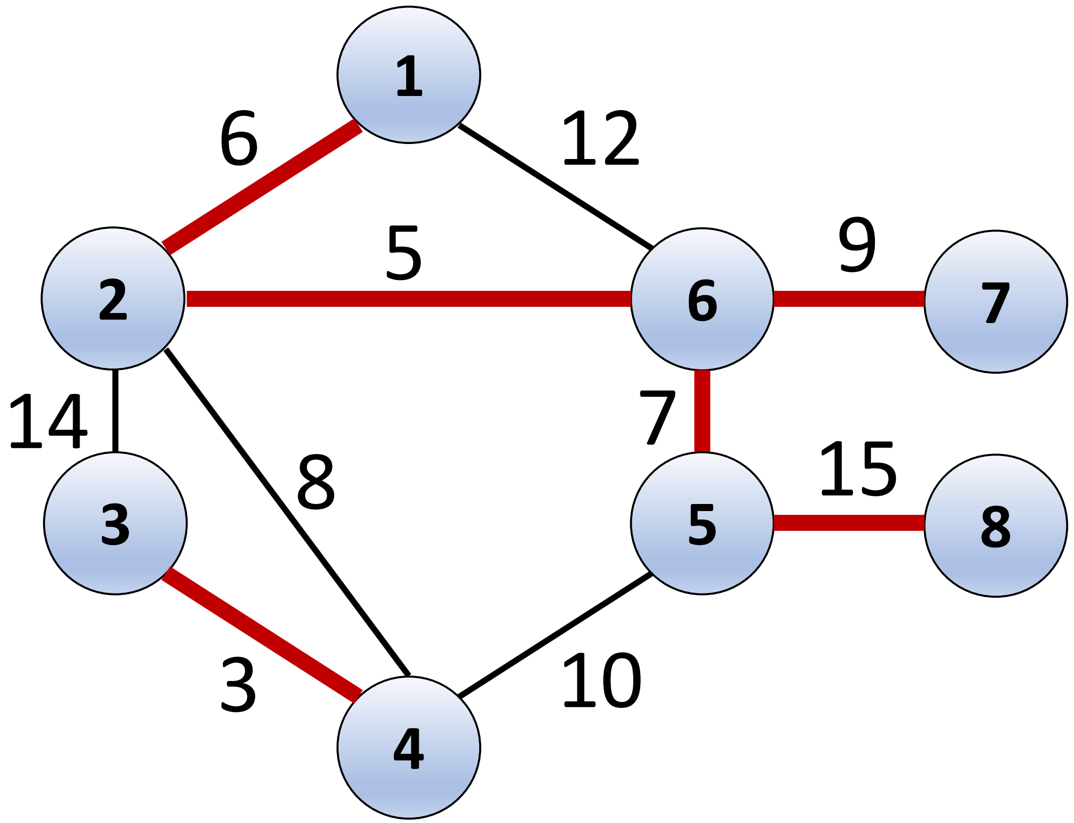
Borůvka's Algorithm

Label Connected Components



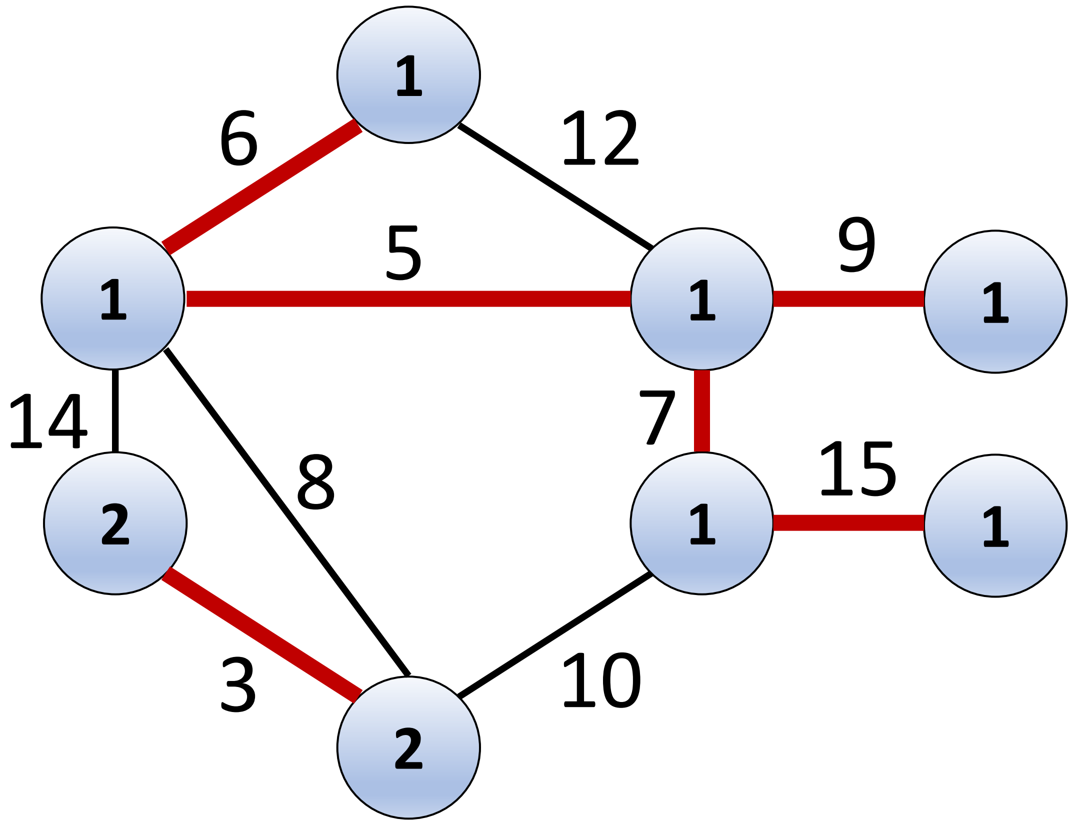
Borůvka's Algorithm

Add Safe Edges



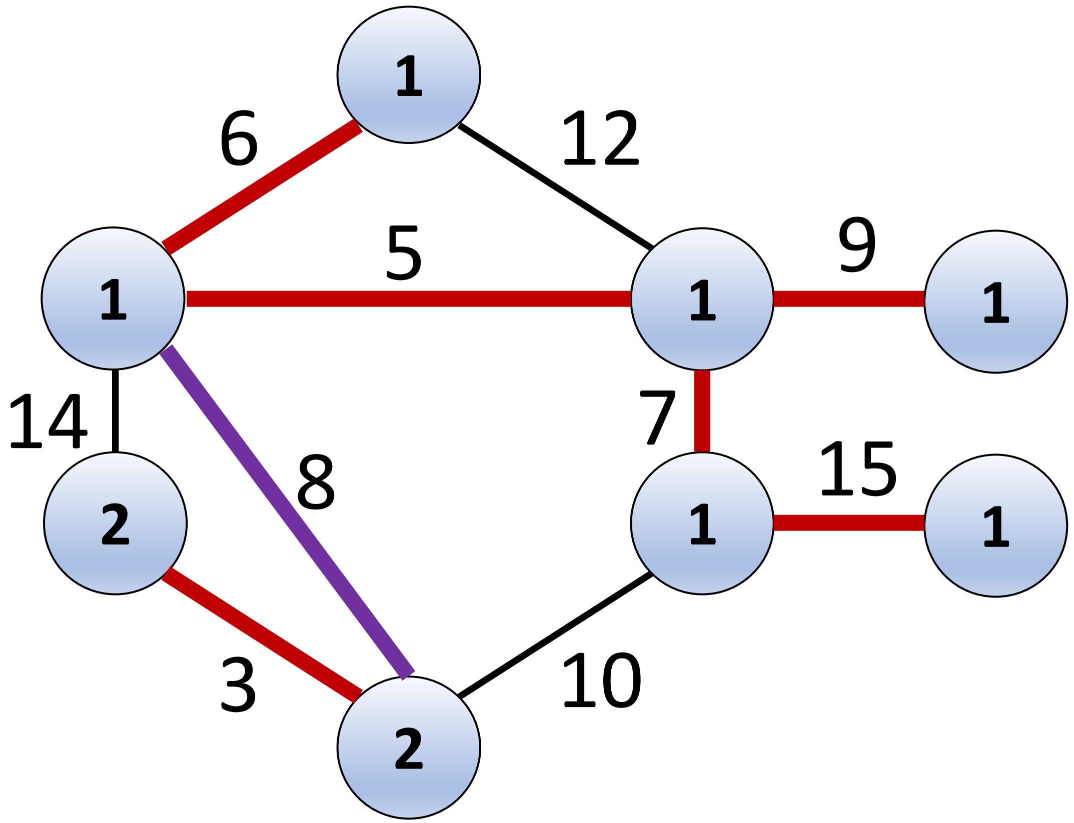
Borůvka's Algorithm

Label Connected Components



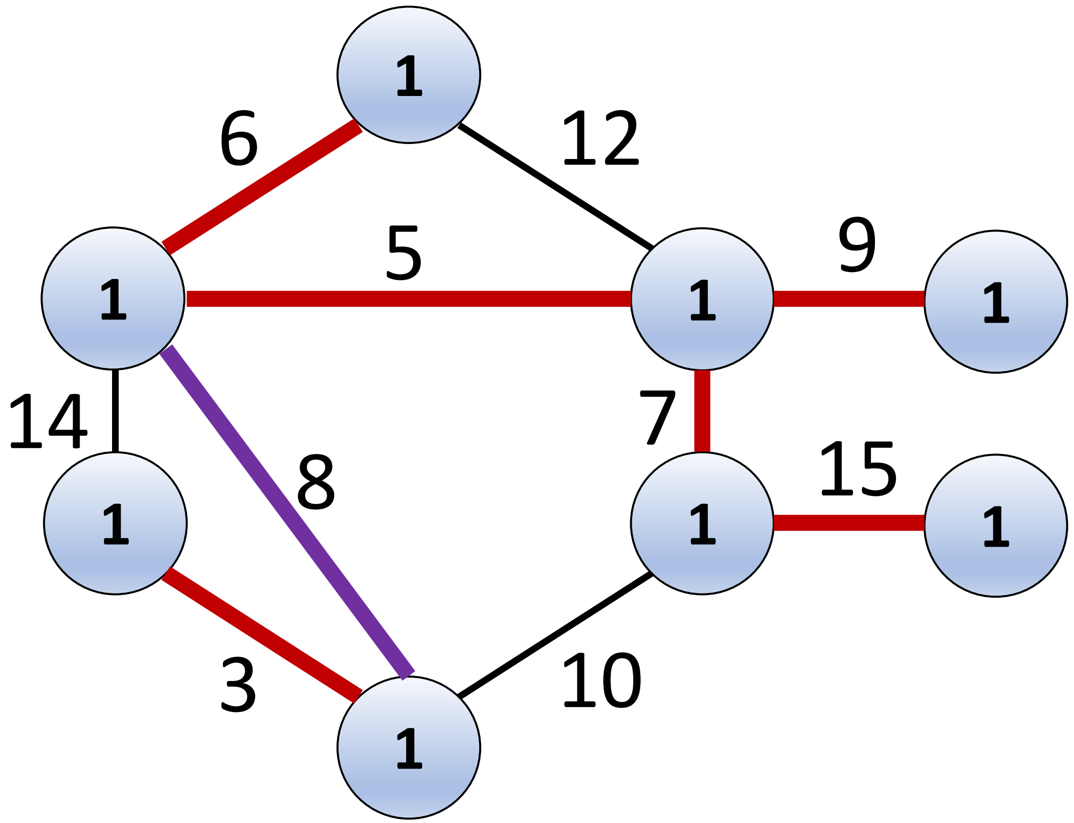
Borůvka's Algorithm

Add Safe Edges



Borůvka's Algorithm

Done!

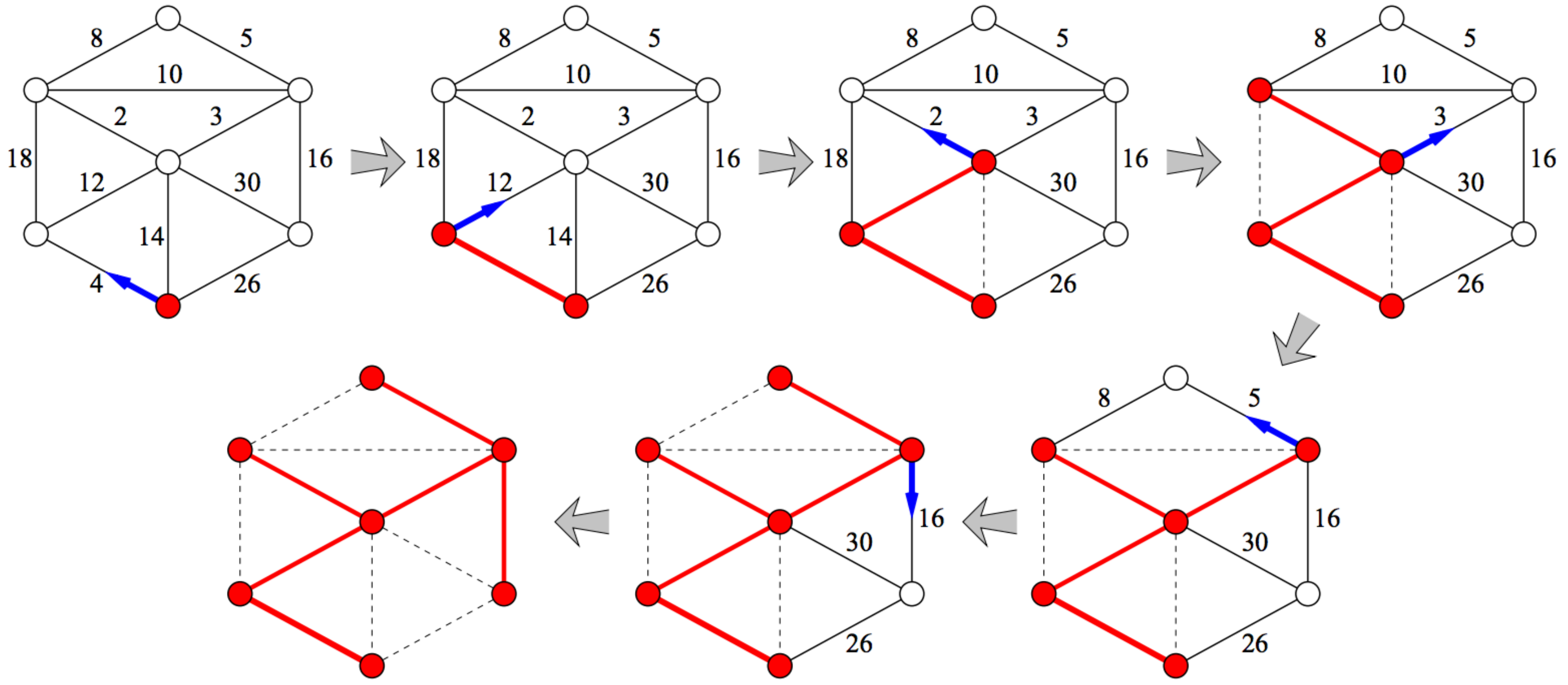


Prim's Algorithm

- **Prim Informal**
  - Let $T = \emptyset$
  - Let s be some arbitrary node and $S = \{s\}$
  - Repeat until $S = V$
    - Find the cheapest edge $e = (u, v)$ cut by S. Add e to T and add v to S

- **Correctness:** every edge we add is safe

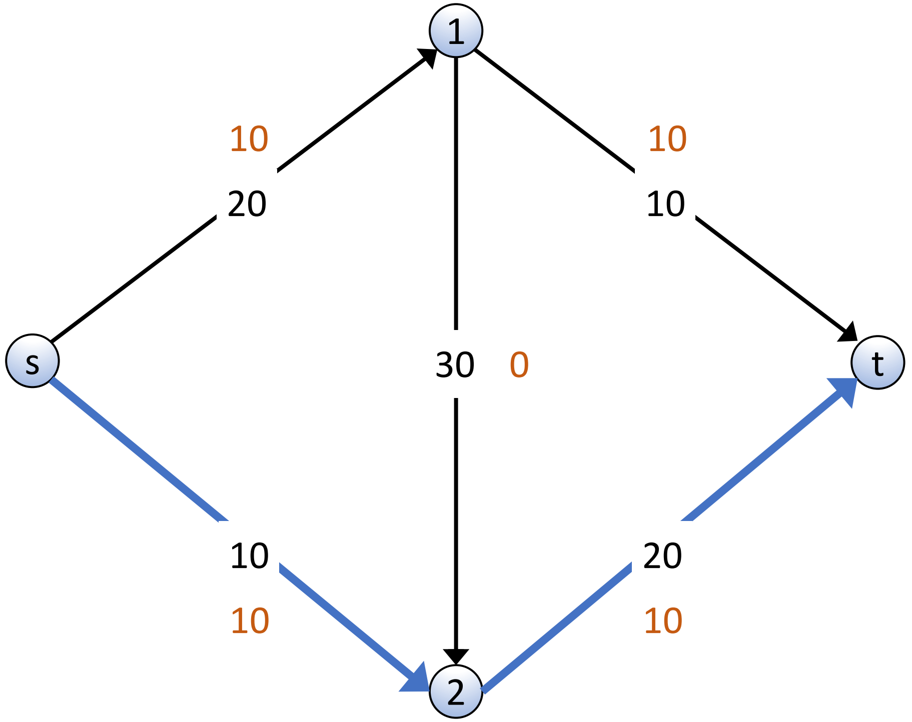
Prim’s Algorithm



Network Flow

Augmenting Paths

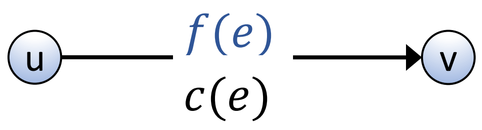
- Given a network $G = (V, E, s, t, \{c(e)\})$ and a flow f, an **augmenting path** P is an $s \to t$ path such that $f(e) < c(e)$ for every edge $e \in P$



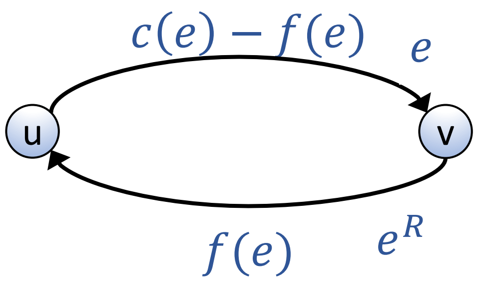
Adding uniform flow on an augmenting path results in a new valid s-t flow!

Residual Graphs

- Original edge: $e = (u, v) \in E$.
  - Flow $f(e)$, capacity $c(e)$



- Residual edge
  - Allows “undoing” flow
  - $e = (u, v)$ and $e^R = (v, u)$.
  - Residual capacity



- Residual graph $G_f = (V, E_f)$
  - Edges with positive residual capacity.
  - $E_f = \{e : f(e) < c(e)\} \cup \{e^R : c(e) > 0\}$.

Augmenting Paths in Residual Graphs

- Let G_f be a **residual graph**
- Let P be an augmenting path in the **residual graph**
- **Fact:** $f' = \text{Augment}(G_f, P)$ is a valid flow

```
Augment(G_f, P)
    b ← the minimum capacity of an edge in P
    for e ∈ P
        if e ∈ E:    f(e) ← f(e) + b
        else:        f(e) ← f(e) - b
    return f
```

Note: This is the same process as the recurrence in Erickson 10.3!

$$f'(u \to v) = \begin{cases} f(u \to v) + F & \text{if } u \to v \in P \\ f(u \to v) - F & \text{if } v \to u \in P \\ f(u \to v) & \text{otherwise} \end{cases}$$

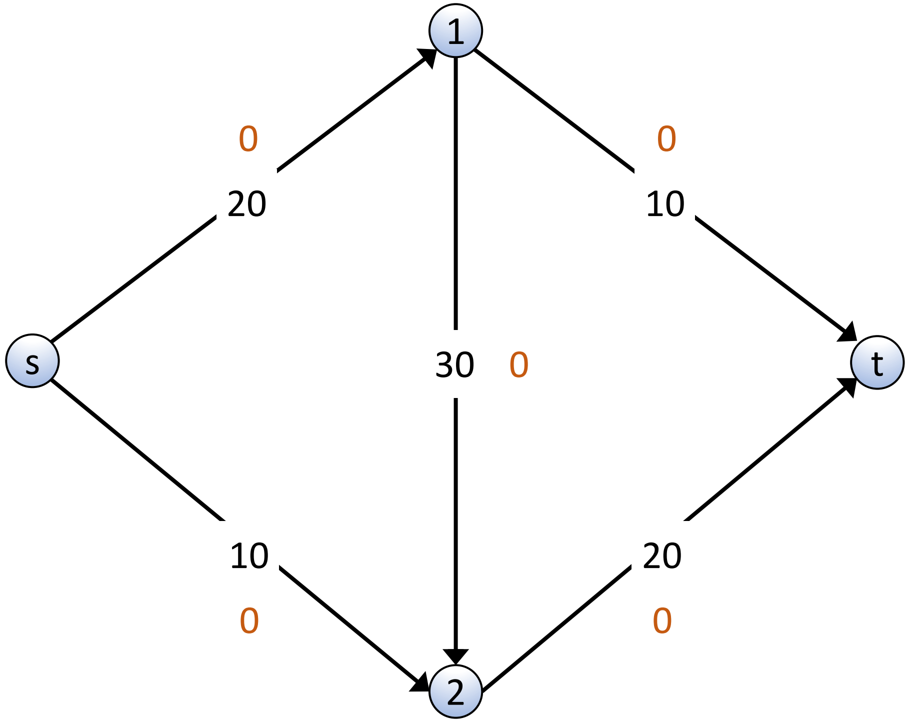
Ford-Fulkerson Algorithm

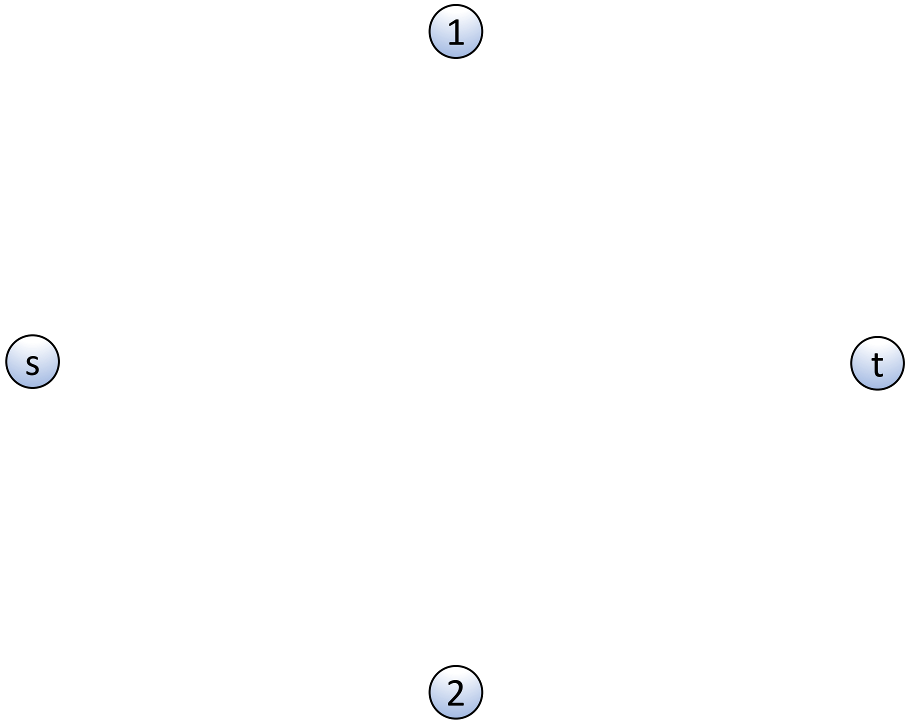
```
FordFulkerson(G,s,t,{c(e)})
    for e ∈ E: f(e) ← 0
    G_f is the residual graph

    while (there is an s-t path P in G_f)
        f ← Augment(G_f,P)
        update G_f

    return f
```

```
Augment(G_f, P)
    b ← the minimum capacity of an edge in P
    for e ∈ P
        if e ∈ E:    f(e) ← f(e) + b
        else:        f(e) ← f(e) - b
    return f
```

Ford-Fulkerson Algorithm

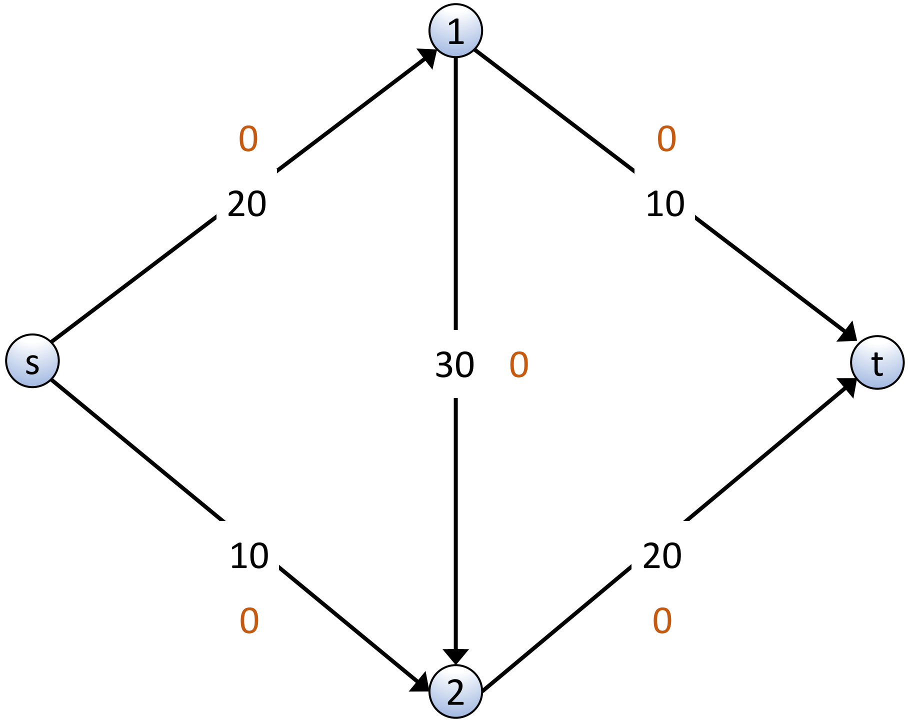
- Start with $f(e) = 0$ for all edges $e \in E$
- Find an **augmenting path** P in the **residual graph**
- Repeat until you get stuck

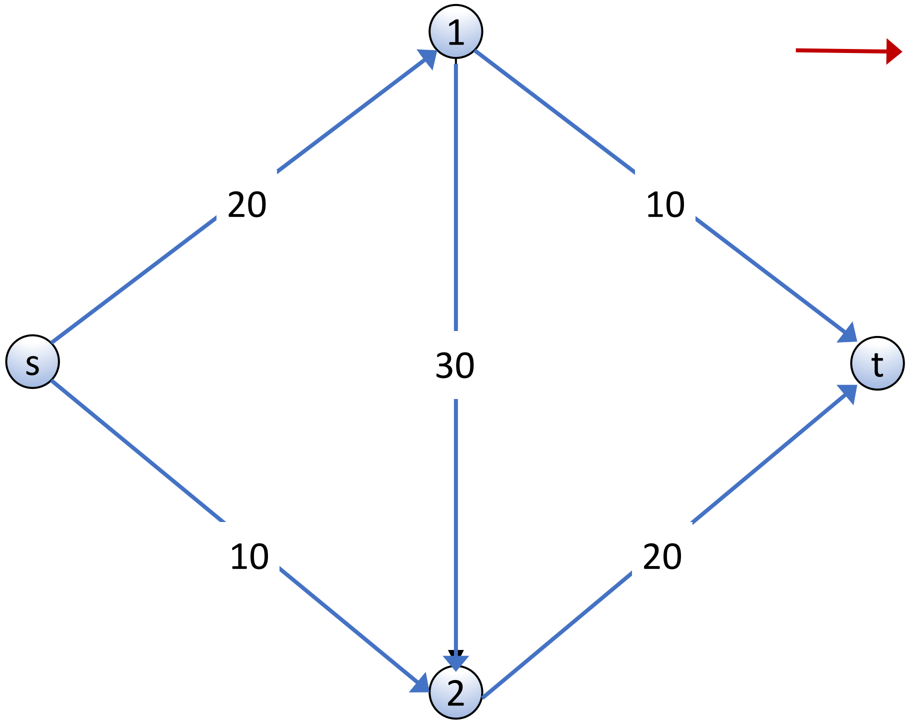




Ford-Fulkerson Algorithm

- Start with $f(e) = 0$ for all edges $e \in E$
- Find an **augmenting path** P in the **residual graph**
- Repeat until you get stuck



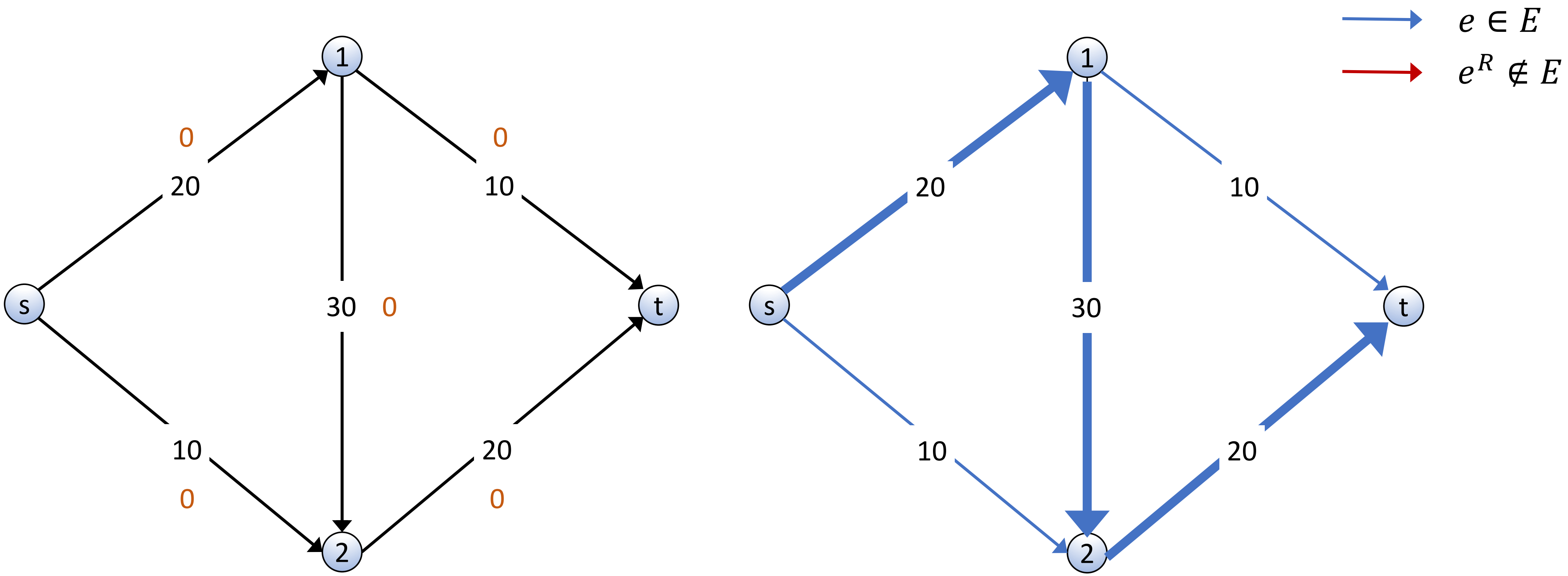


$e \in E$

$e^R \notin E$

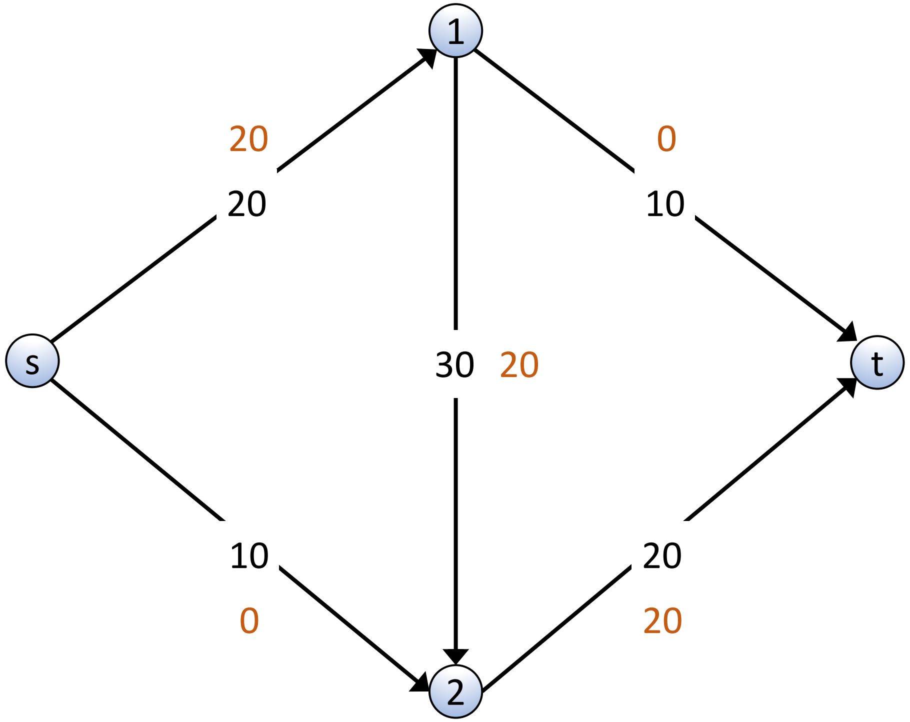
Ford-Fulkerson Algorithm

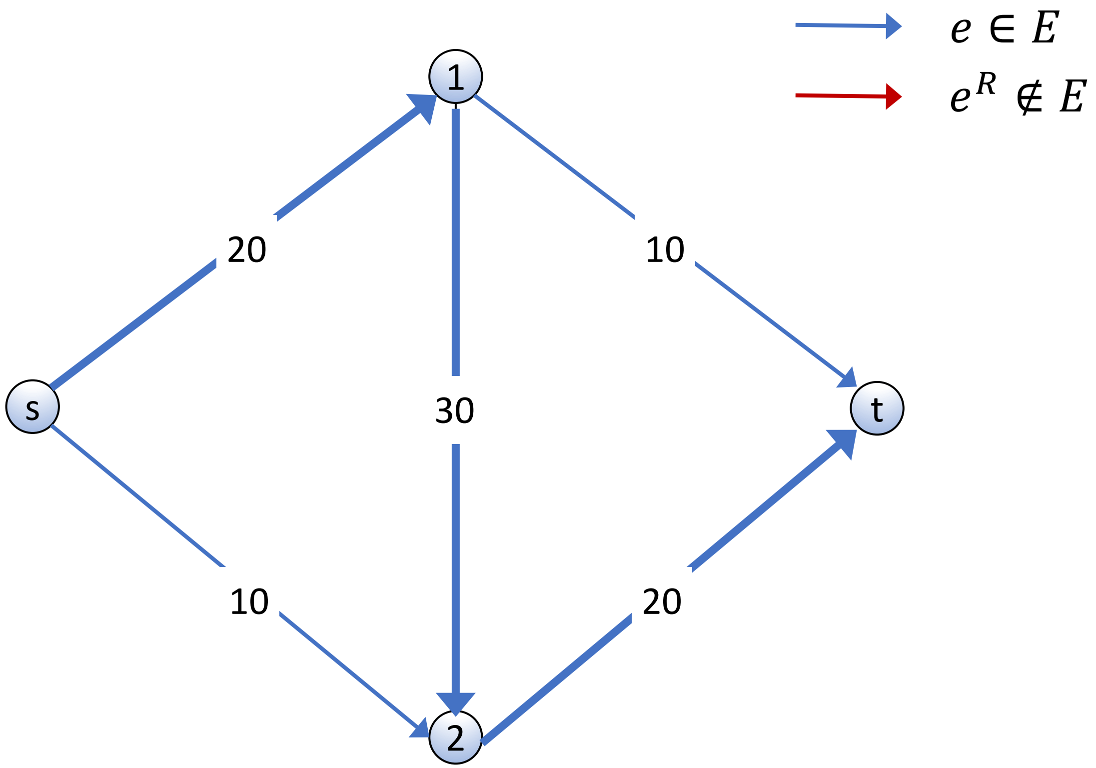
- Start with $f(e) = 0$ for all edges $e \in E$
- Find an **augmenting path** P in the **residual graph**
- Repeat until you get stuck



Ford-Fulkerson Algorithm

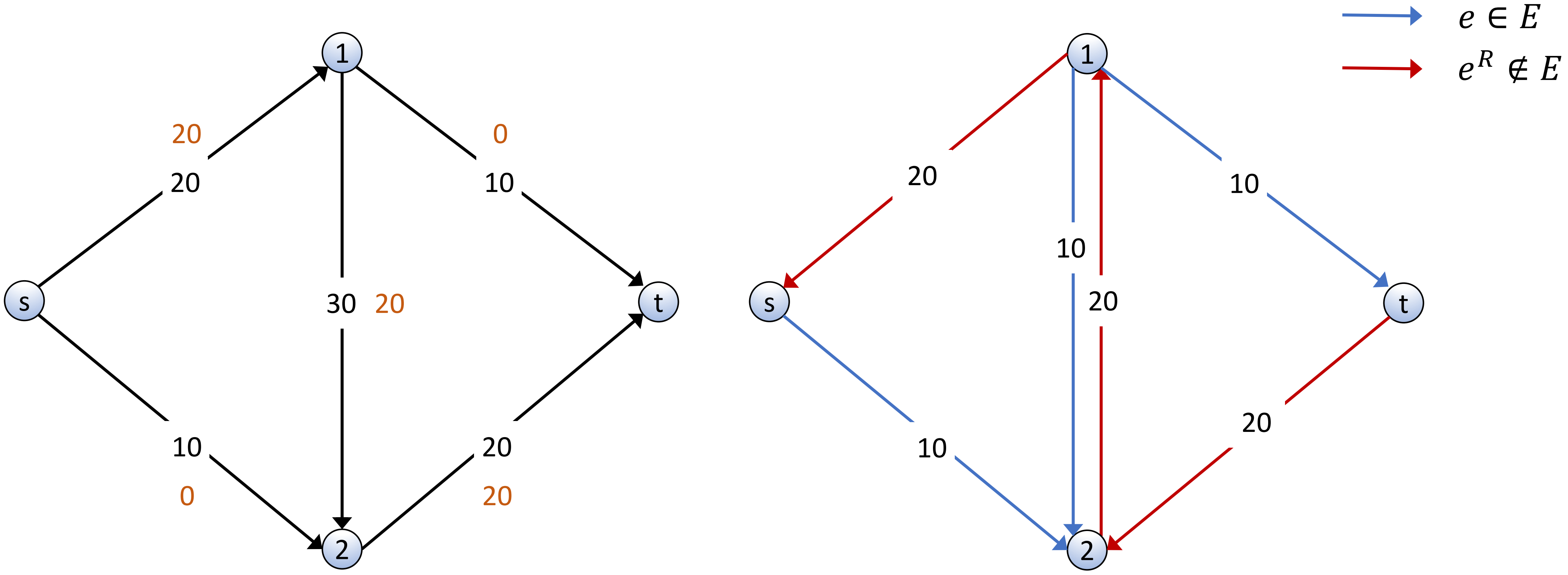
- Start with $f(e) = 0$ for all edges $e \in E$
- Find an **augmenting path** P in the **residual graph**
- Repeat until you get stuck





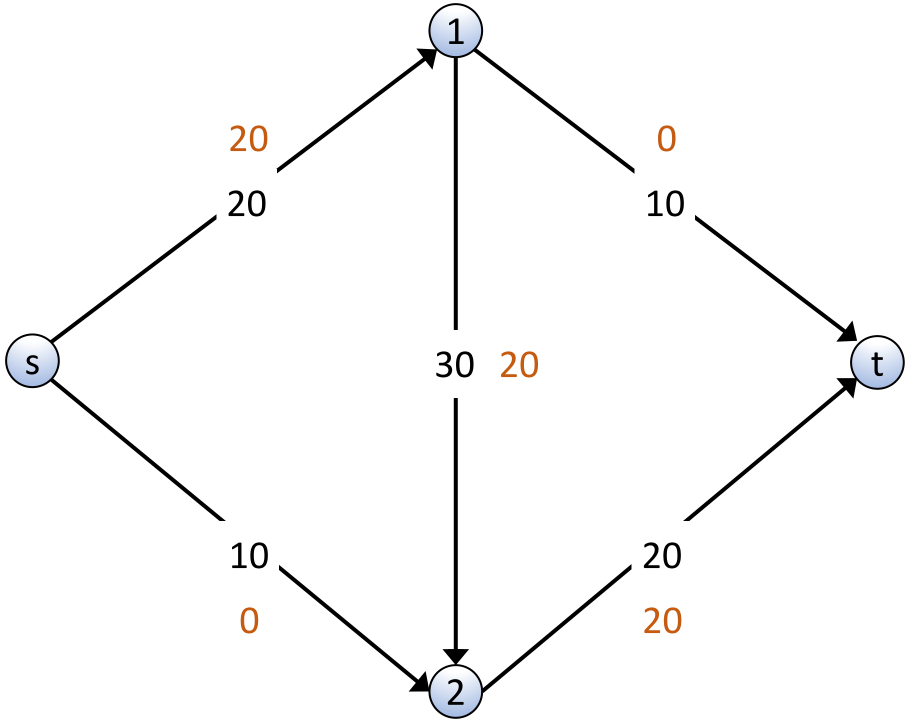
Ford-Fulkerson Algorithm

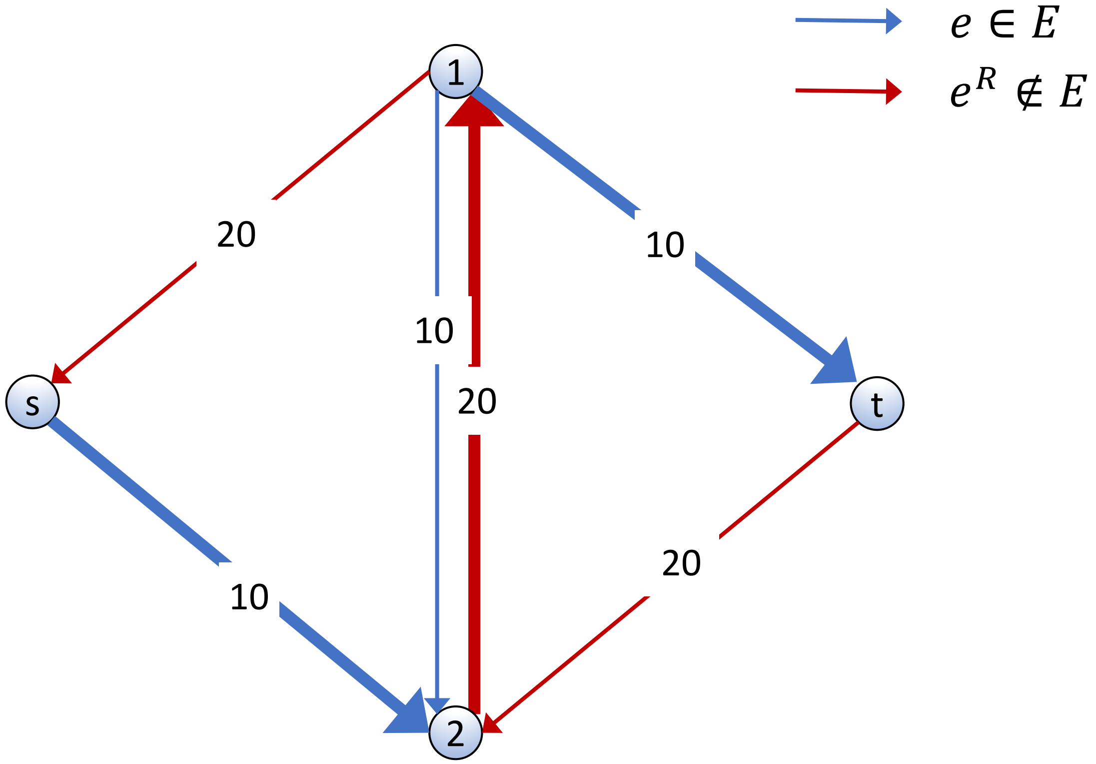
- Start with $f(e) = 0$ for all edges $e \in E$
- Find an **augmenting path** P in the **residual graph**
- Repeat until you get stuck



Ford-Fulkerson Algorithm

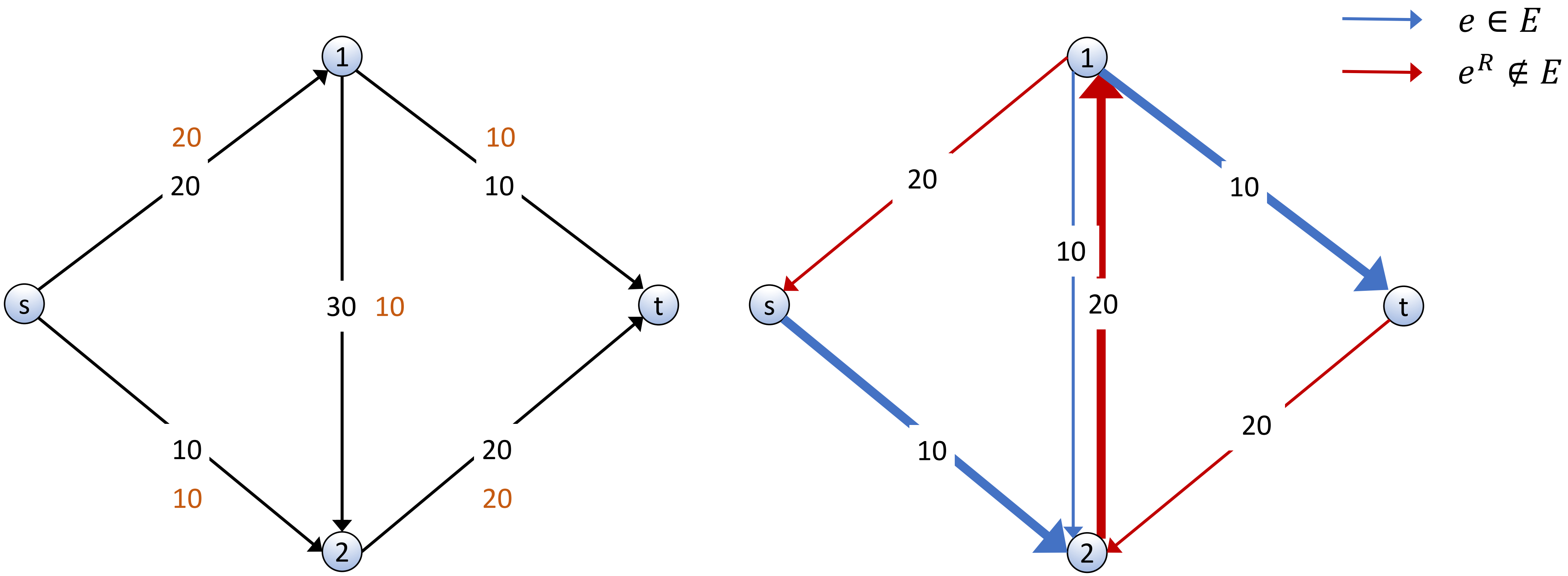
- Start with $f(e) = 0$ for all edges $e \in E$
- Find an **augmenting path** P in the **residual graph**
- Repeat until you get stuck





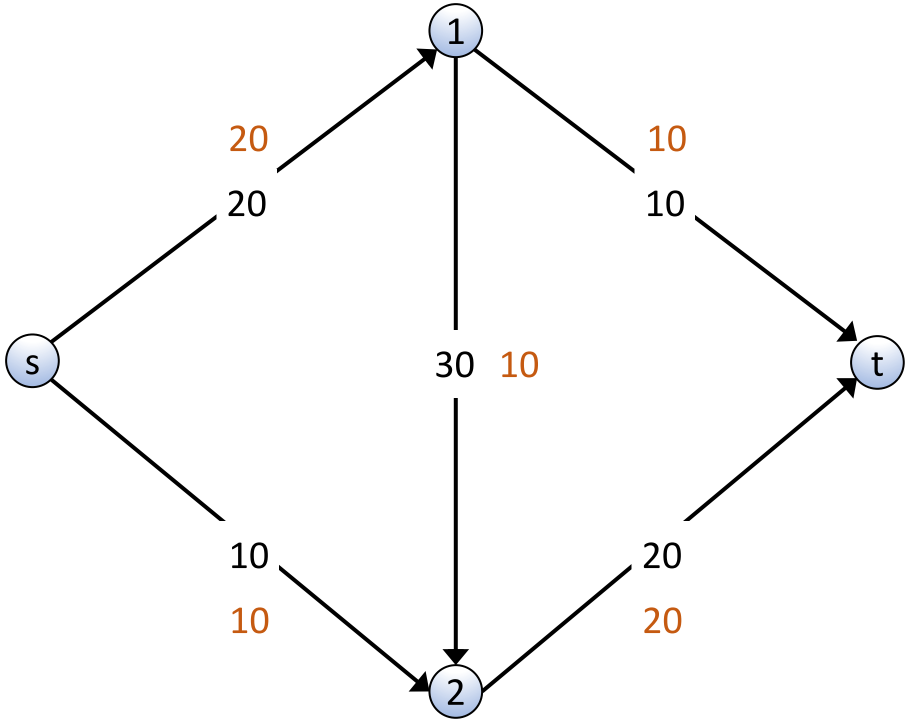
Ford-Fulkerson Algorithm

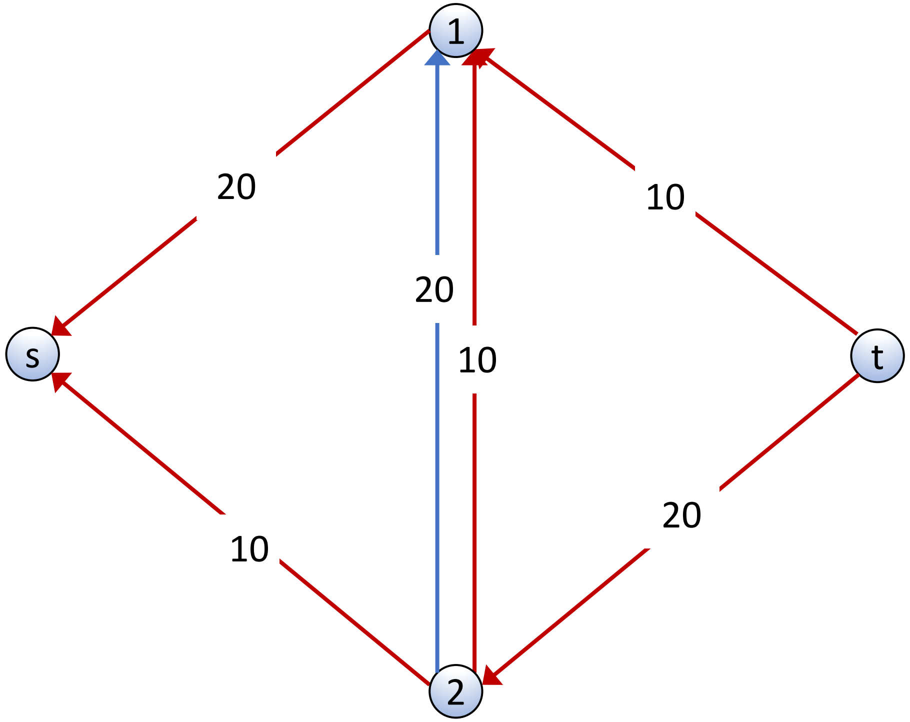
- Start with $f(e) = 0$ for all edges $e \in E$
- Find an **augmenting path** P in the **residual graph**
- Repeat until you get stuck



Ford-Fulkerson Algorithm

- Start with $f(e) = 0$ for all edges $e \in E$
- Find an **augmenting path** P in the **residual graph**
- Repeat until you get stuck





Network Flow Summary

- **The Ford-Fulkerson Algorithm solves maximum s-t flow**
  - Running time $O\left(m \cdot val(f^*)\right)$ in networks with integer capacities

- **Strong MaxFlow-MinCut Duality: max flow = min cut**
  - The value of the maximum s-t flow equals the capacity of the minimum s-t cut
  - If f^* is a maximum s-t flow, then the set of nodes reachable from s in G_{f^*} gives a minimum cut
  - Given a max-flow, can find a min-cut in time $O(n + m)$

- **Every graph with integer capacities has an integer maximum flow**
  - Ford-Fulkerson will return an integer maximum flow

More questions?

Wrap-up

No class tomorrow!

Homework 5 due tonight, solutions out tomorrow morning

- Get in touch ASAP (not 10PM) if you need more time!

Midterm 2 released Wednesday 8PM and due Friday 8PM Boston time!