Module 04: Graph Algorithms

Analysis and Design of Algorithms

Ammar Sherif

Nile University

What are Graphs?

Graphs

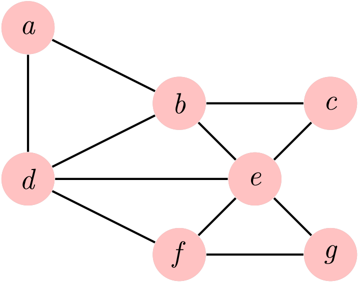
From a mathematical perspective, consist of a nodes/vertices and edges.

What are Graphs?

Graphs

From a mathematical perspective, consist of a nodes/vertices and edges.

- Nodes/Vertices

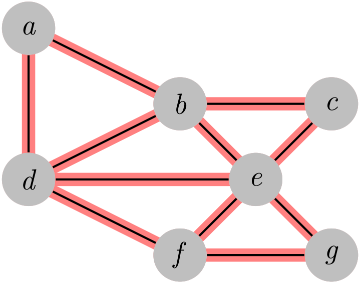


What are Graphs?

Graphs

From a mathematical perspective, consist of a nodes/vertices and edges.

- Nodes/Vertices
- Edges

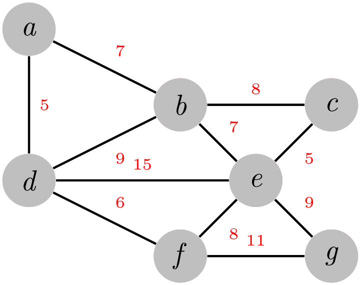


What are Graphs?

Graphs

From a mathematical perspective, consist of a nodes/vertices and edges.

- Nodes/Vertices
- Edges
- Weighted Graph

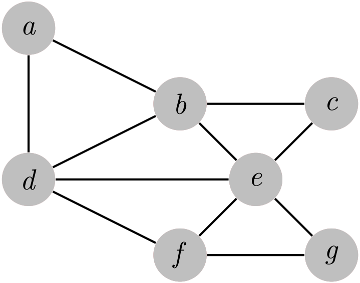


What are Graphs?

Graphs

From a mathematical perspective, consist of a nodes/vertices and edges.

- Nodes/Vertices
- Edges
- Weighted Graph
- Unweighted Graph

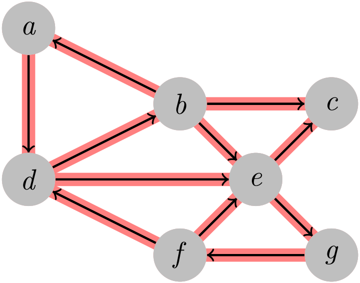


What are Graphs?

Graphs

From a mathematical perspective, consist of a nodes/vertices and edges.

- Nodes/Vertices
- Edges
- Weighted Graph
- Unweighted Graph
- Directed Graph

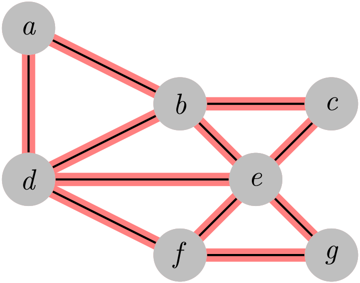


What are Graphs?

Graphs

From a mathematical perspective, consist of a nodes/vertices and edges.

- Nodes/Vertices
- Edges
- Weighted Graph
- Unweighted Graph
- Directed Graph
- Undirected Graph



Why Graphs & Graph Algorithms?

- Shortest Path and Route planning

Why Graphs & Graph Algorithms?

- Shortest Path and Route planning
- Robotics

Why Graphs & Graph Algorithms?

- Shortest Path and Route planning
- Robotics
  - Warehouses

Why Graphs & Graph Algorithms?

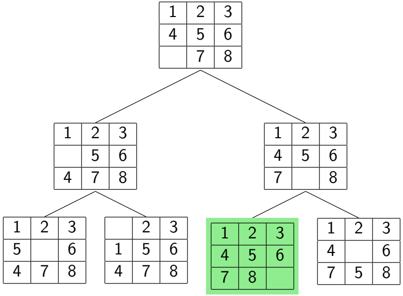
- Shortest Path and Route planning
- Robotics
  - Warehouses
  - Space Robots

Why Graphs & Graph Algorithms?

- Shortest Path and Route planning
- Robotics
  - Warehouses
  - Space Robots
  - Rescue Robots

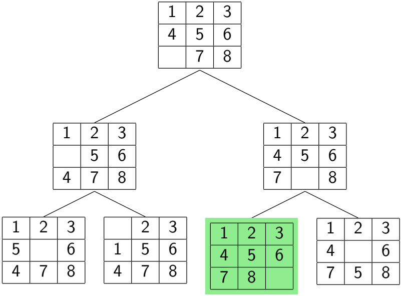
Why Graphs & Graph Algorithms?

- Shortest Path and Route planning
- Robotics
  - Warehouses
  - Space Robots
  - Rescue Robots
- Games



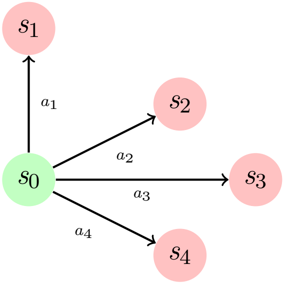
Why Graphs & Graph Algorithms?

- Shortest Path and Route planning
- Robotics
  - Warehouses
  - Space Robots
  - Rescue Robots
- Games
- Optimization Problems



Why Graphs & Graph Algorithms?

- Shortest Path and Route planning
- Robotics
  - Warehouses
  - Space Robots
  - Rescue Robots
- Games
- Optimization Problems
- Any decision-based problem



Graph Representation

Algorithms are implemented via programming, so how to represent graphs?

What our representation should provide?

Graph Representation

Algorithms are implemented via programming, so how to represent graphs?

What our representation should provide?

- Know the neighbors

Graph Representation

Algorithms are implemented via programming, so how to represent graphs?

What our representation should provide?

- Know the neighbors
- Weights of links

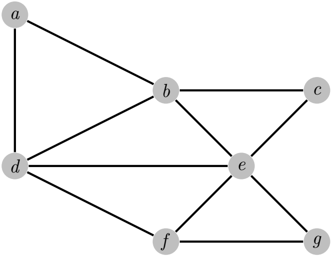
Graph Representation

Algorithms are implemented via programming, so how to represent graphs?

What our representation should provide?

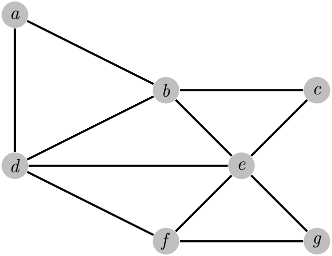
- Know the neighbors
- Weights of links
- list of nodes/edges

Graph Representation



Graph Representation

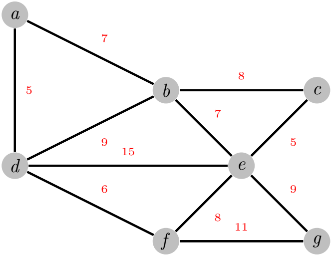
- Adjacency Matrix



	a	b	c	d	e	f	g
a	–	1		1			
b	1	–	1	1	1		
c		1	–		1		
d	1	1		–	1	1	
e		1	1	1	–	1	1
f				1	1	–	1
g					1	1	–

Graph Representation

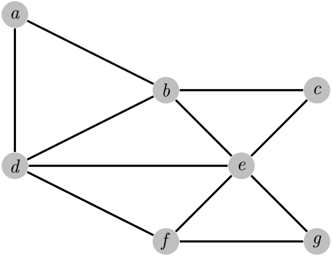
- Adjacency Matrix

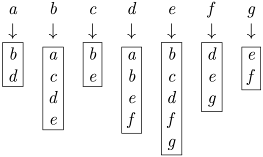


	a	b	c	d	e	f	g
a	–	7		5			
b	7	–	8	9	7		
c		8	–		5		
d	5	9		–	15	8	
e		7	5	15	–	8	9
f				8	8	–	11
g					9	11	–

Graph Representation

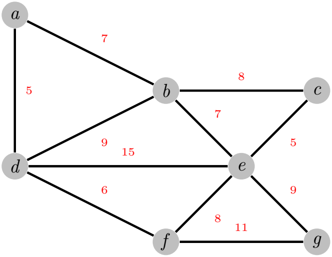
- Adjacency Matrix
- Adjacency List





Graph Representation

- Adjacency Matrix
- Adjacency List

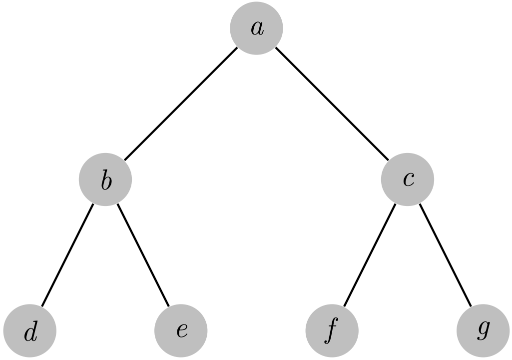


a	b	c	d	e	f	g
$b:7$	$a:7$	$b:8$	$a:5$	$b:7$	$d:8$	$e:9$
$d:5$	$c:8$	$e:5$	$b:9$	$c:5$	$e:8$	$f:11$
	$d:9$		$e:15$	$d:15$	$g:11$	
	$e:7$		$f:8$	$f:8$		
				$g:9$		

1. Graphs & Graph Algorithms
2. Unweighted Graphs
   - Depth First Search (DFS)
   - Breadth First Search (BFS)
3. Weighted Graphs
4. Path Construction

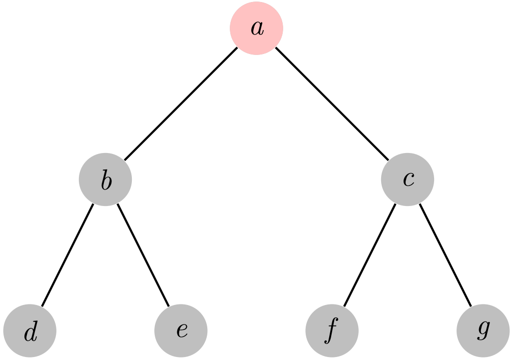
Depth First Search (DFS)

Depth has the max priority



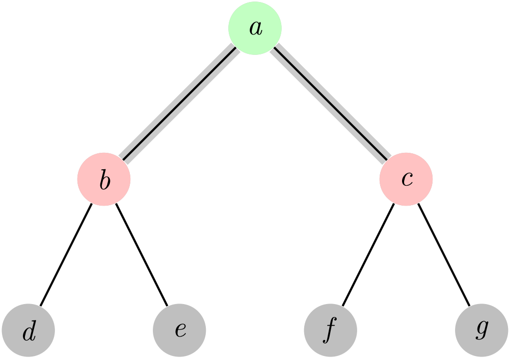
Depth First Search (DFS)

Depth has the max priority



Depth First Search (DFS)

Depth has the max priority

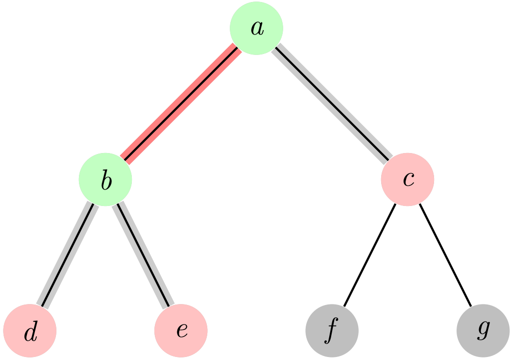


Depth First Search (DFS)

Depth has the max priority

Child $\longleftarrow$ Parent

$b \longleftarrow a$

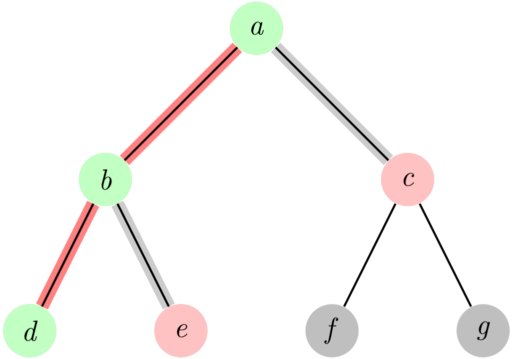


Depth First Search (DFS)

Depth has the max priority

Child $\longleftarrow$ Parent

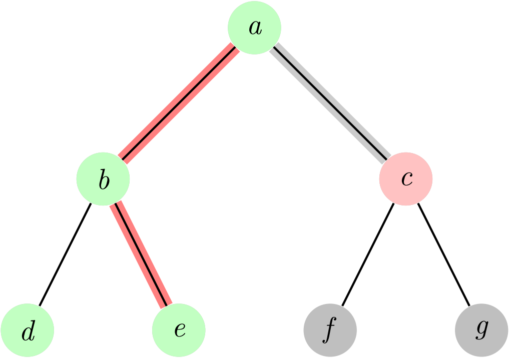
$b \longleftarrow a$
$d \longleftarrow b$



Depth First Search (DFS)

Depth has the max priority
Child $\longleftarrow$ Parent

$b \longleftarrow a$
$d \longleftarrow b$
$e \longleftarrow b$

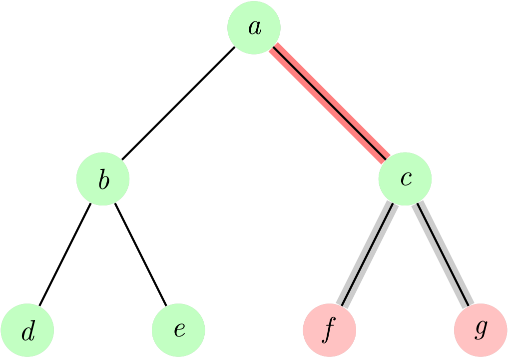


Depth First Search (DFS)

Depth has the max priority

Child $\longleftarrow$ Parent

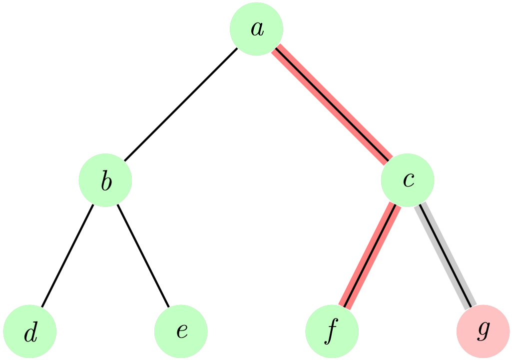
$b \longleftarrow a$
$d \longleftarrow b$
$e \longleftarrow b$
$c \longleftarrow a$



Depth First Search (DFS)

Depth has the max priority
Child $\longleftarrow$ Parent

$b \longleftarrow a$
$d \longleftarrow b$
$e \longleftarrow b$
$c \longleftarrow a$
$f \longleftarrow c$

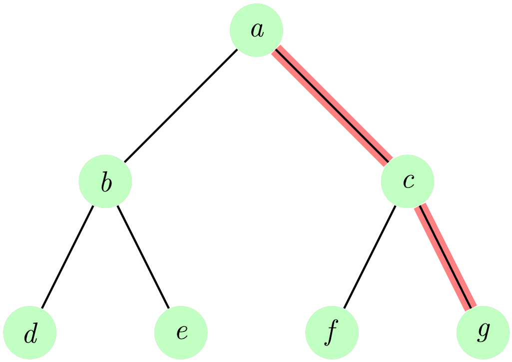


Depth First Search (DFS)

Therefore, the below table summarizes how did we get to any node through our traversal

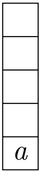
Child $\longleftarrow$ Parent

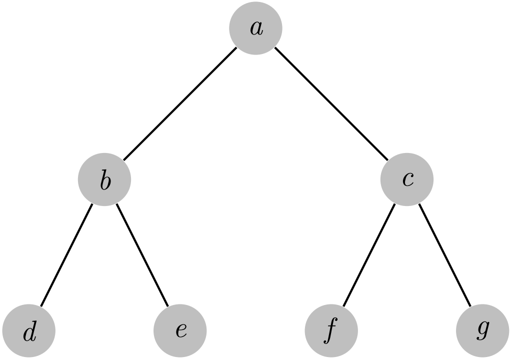
$b \longleftarrow a$
$d \longleftarrow b$
$e \longleftarrow b$
$c \longleftarrow a$
$f \longleftarrow c$
$g \longleftarrow c$



Depth First Search (DFS)

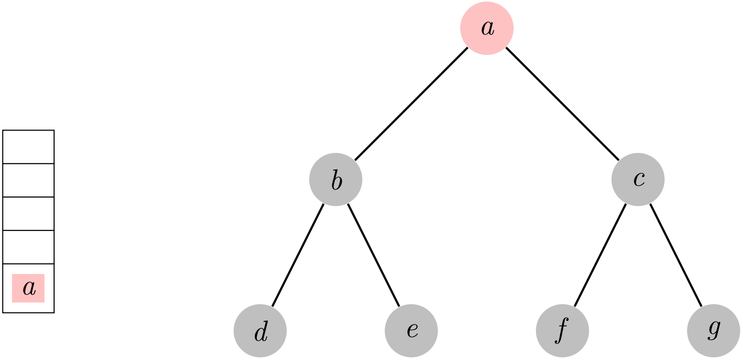
Let's do it again, to notice the **pattern** of nodes to be visited





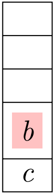
Depth First Search (DFS)

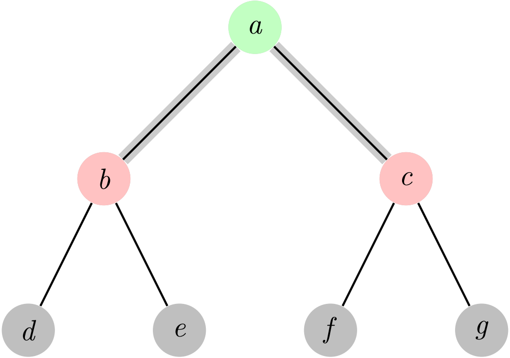
Let's do it again, to notice the **pattern** of nodes to be visited



Depth First Search (DFS)

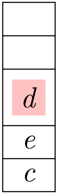
Let's do it again, to notice the **pattern** of nodes to be visited

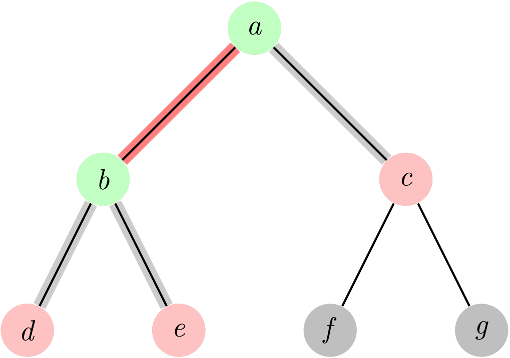




Depth First Search (DFS)

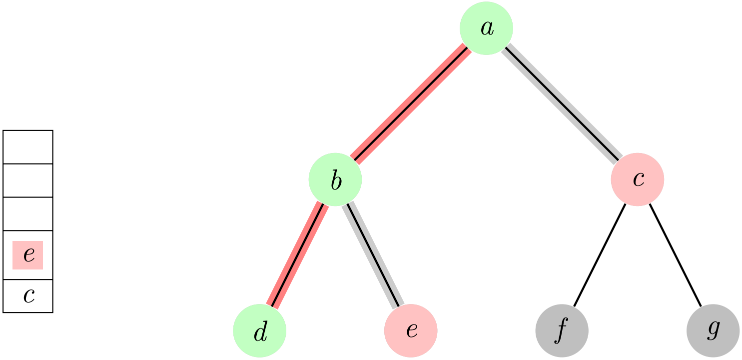
Let's do it again, to notice the **pattern** of nodes to be visited





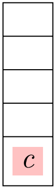
Depth First Search (DFS)

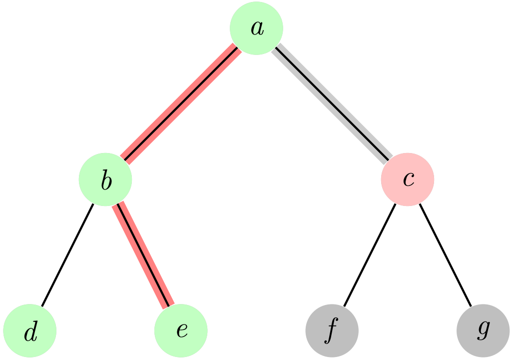
Let's do it again, to notice the **pattern** of nodes to be visited



Depth First Search (DFS)

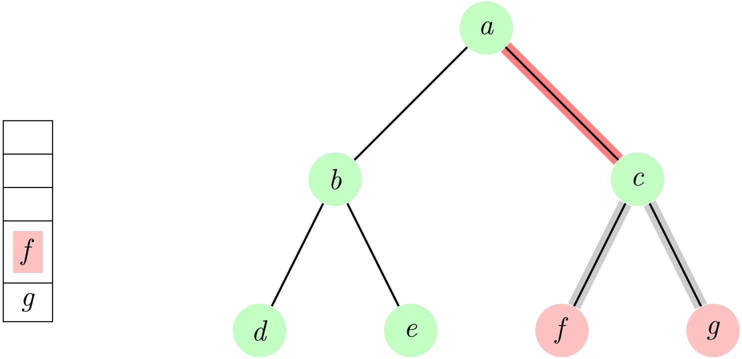
Did you get the pattern of nodes to be visited?





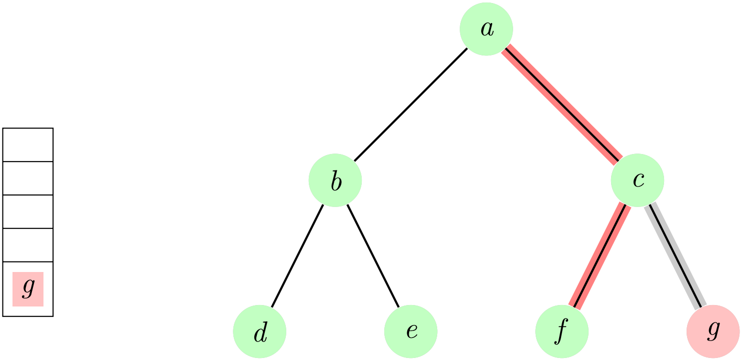
Depth First Search (DFS)

Did you get the pattern of nodes to be visited? **Last** inserted element is **first** to explore.



Depth First Search (DFS)

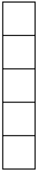
Did you notice what might be this structure? Hint: Last-in First-out

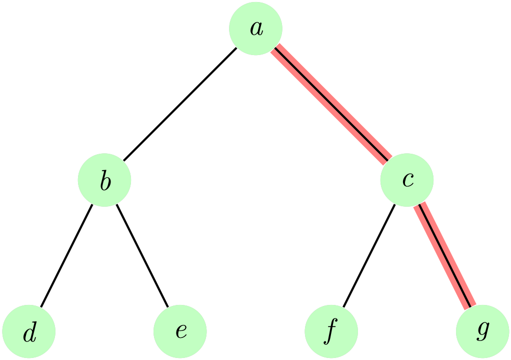


Depth First Search (DFS)

Did you notice what might be this structure? Hint: Last-in First-out

Yes, it is *Stack*





Depth First Search (DFS)

Algorithm 1: $\textsc{Depth-First}(root)$

```
def S to be Stack;
visited ← {};
S.push(root);
while S ≠ ϕ do
    node ← S.pop();
    if node ∉ visited then
        visited ← visited ∪ {node};
        for n ∈ adjacent(node) do
            S.push(n);
        end
    end
end
```

1. Graphs & Graph Algorithms

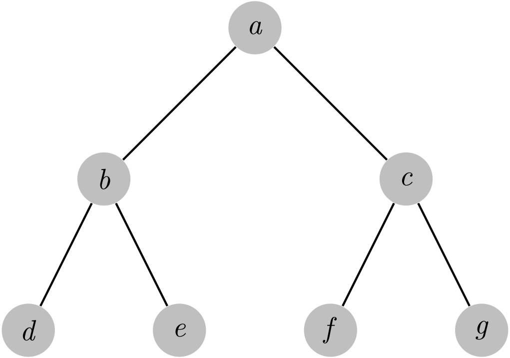
2. Unweighted Graphs
    - Depth First Search (DFS)
    - Breadth First Search (BFS)

3. Weighted Graphs

4. Path Construction

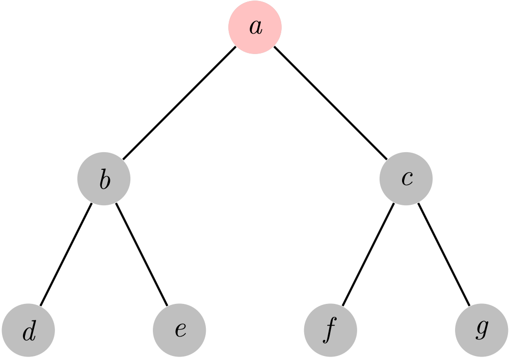
Breadth First Search (BFS)

Breadth has the max priority



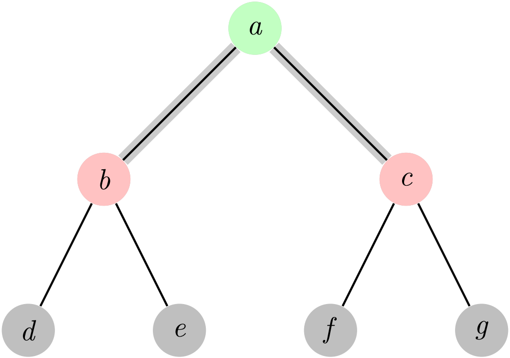
Breadth First Search (BFS)

Breadth has the max priority



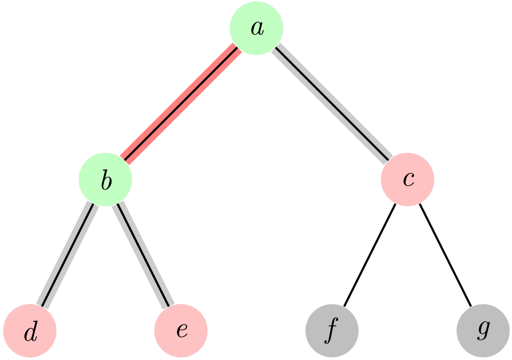
Breadth First Search (BFS)

Breadth has the max priority



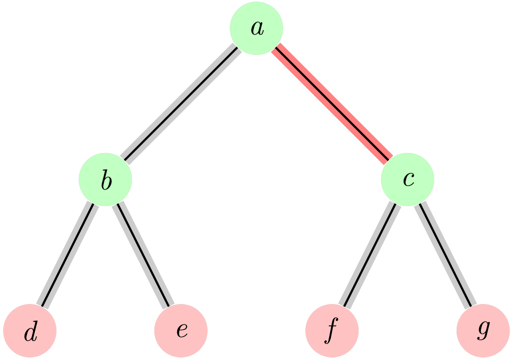
Breadth First Search (BFS)

Breadth has the max priority



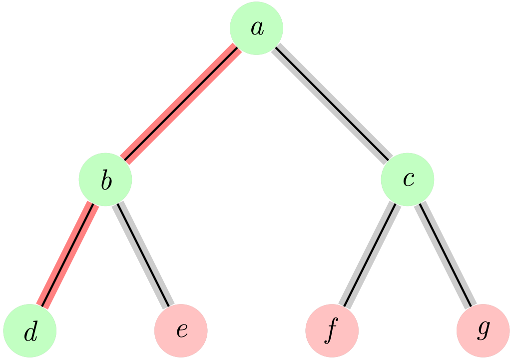
Breadth First Search (BFS)

Breadth has the max priority



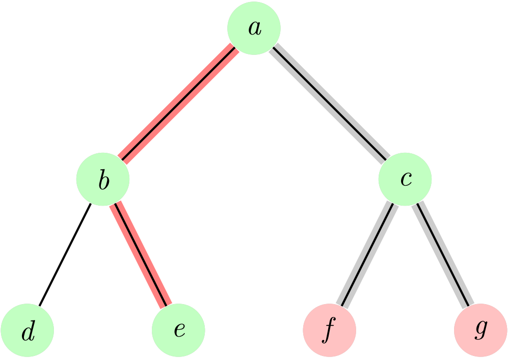
Breadth First Search (BFS)

Breadth has the max priority



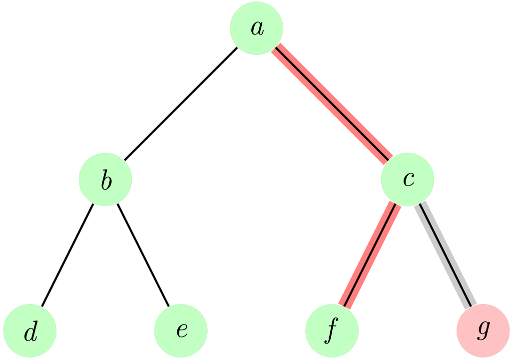
Breadth First Search (BFS)

Breadth has the max priority



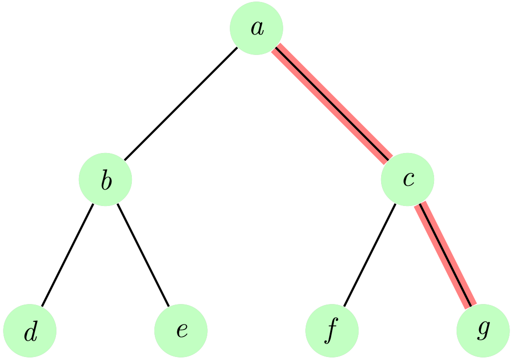
Breadth First Search (BFS)

Breadth has the max priority



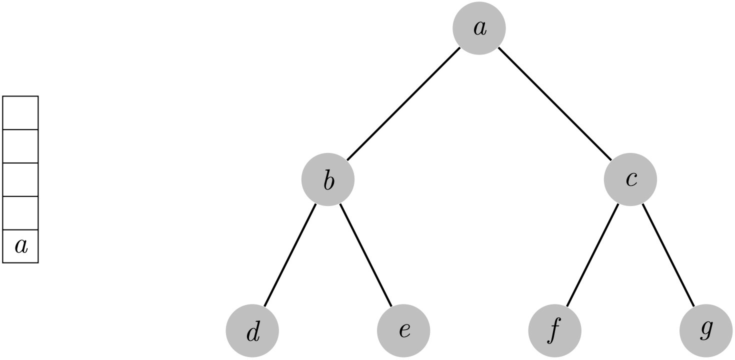
Breadth First Search (BFS)

Breadth has the max priority



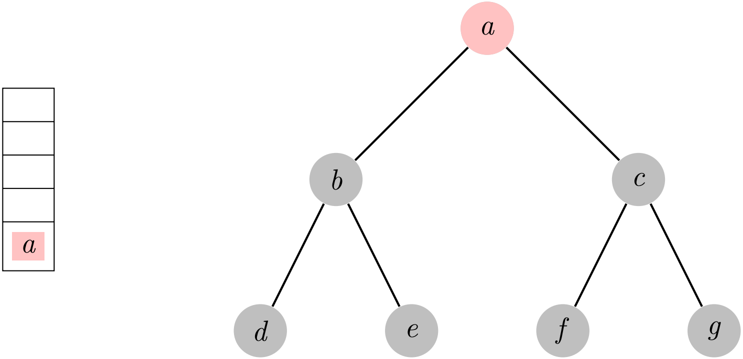
Breadth First Search (BFS)

Let's do it again, to notice the **pattern** of nodes to be visited



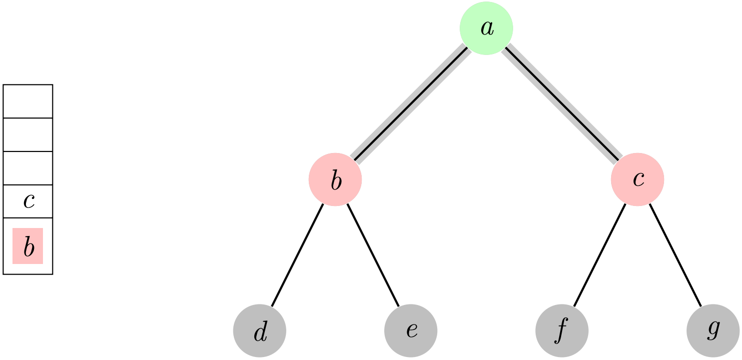
Breadth First Search (BFS)

Let's do it again, to notice the **pattern** of nodes to be visited



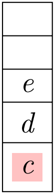
Breadth First Search (BFS)

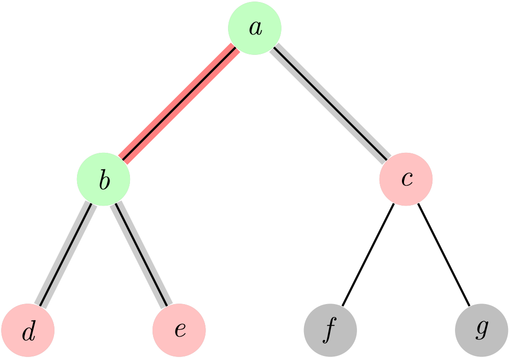
Let's do it again, to notice the **pattern** of nodes to be visited



Breadth First Search (BFS)

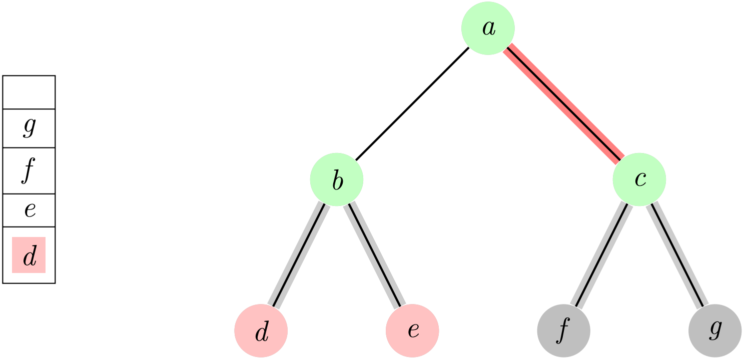
Let's do it again, to notice the **pattern** of nodes to be visited





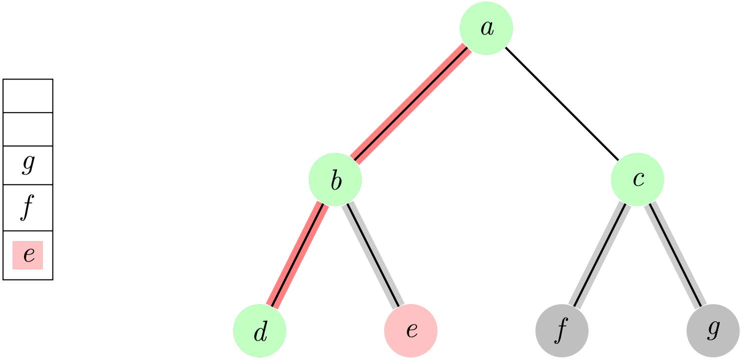
Breadth First Search (BFS)

Let's do it again, to notice the **pattern** of nodes to be visited



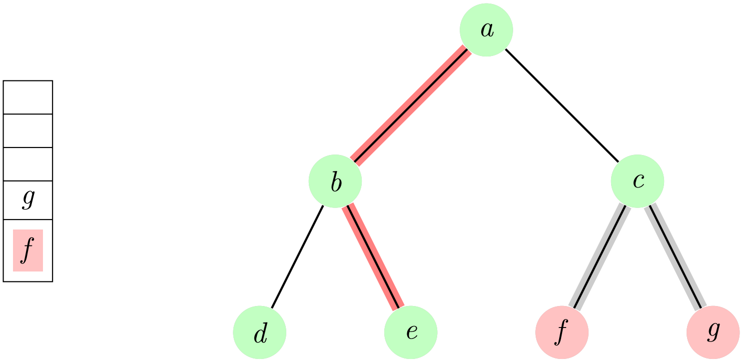
Breadth First Search (BFS)

Did you get the pattern of nodes to be visited?



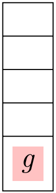
Breadth First Search (BFS)

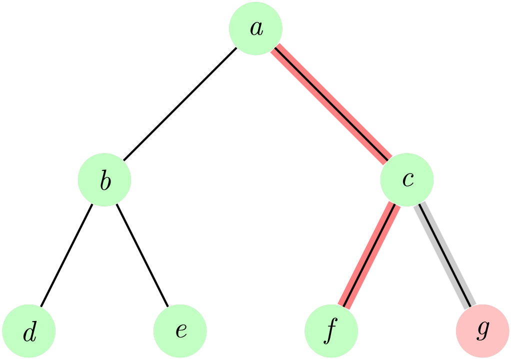
Did you get the pattern of nodes to be visited? **First** inserted element is **first** to explore.



Breadth First Search (BFS)

Did you notice what might be this structure? Hint: First-in First-out

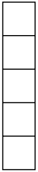


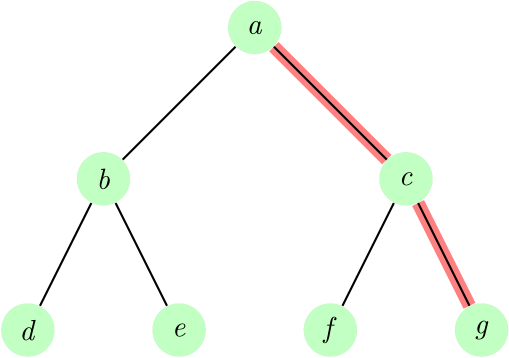


Breadth First Search (BFS)

Did you notice what might be this structure? Hint: First-in First-out

Yes, it is *Queue*





Breadth First Search (BFS)

Algorithm 2: BREADTH-FIRST($root$)

def S to be Queue;
$visited \leftarrow \{\}$;
S.enqueue($root$);
while $S \neq \phi$ **do**
    $node \leftarrow S$.dequeue();
    if $node \notin visited$ **then**
        $visited \leftarrow visited \cup \{node\}$;
        for $n \in adjacent(node)$ **do**
            S.enqueue(n);
        end
    end
end

1. Graphs & Graph Algorithms

2. Unweighted Graphs

3. Weighted Graphs
   - Branch and Bound
   - Search Space Pruning
   - Branch and Bound + Visited List
   - Branch and Bound + Heuristics
   - A* Algorithm
   - Heuristic Design

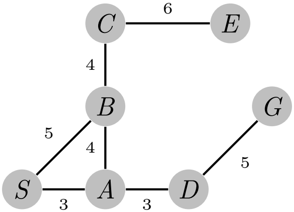
4. Path Construction

Branch and Bound

Right now we have weights, now what should we prioritize?

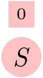
Branch and Bound

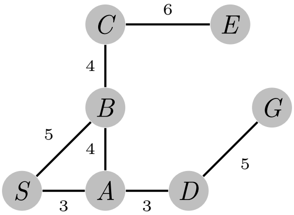
Right now we have weights, now what should we prioritize? Min/Max **weights**



Branch and Bound

Right now we have weights, now what should we prioritize? Min/Max **weights**

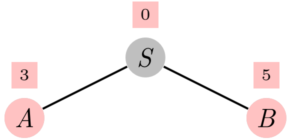


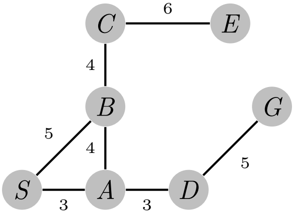


$\boxed{S:0}$

Branch and Bound

Right now we have weights, now what should we prioritize? Min/Max **weights**



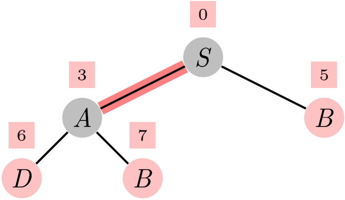


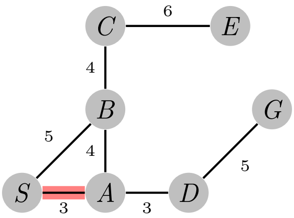
iterations/visits: 1

$A:3$	$B:5$

Branch and Bound

Right now we have weights, now what should we prioritize? Min/Max **weights**



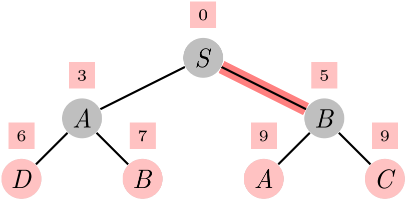


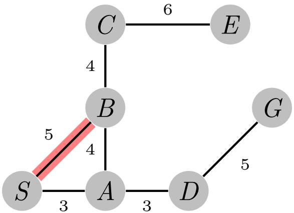
iterations/visits: 2

$B:5$	$D:6$	$B:7$

Branch and Bound

Can you notice the pattern of the structure?



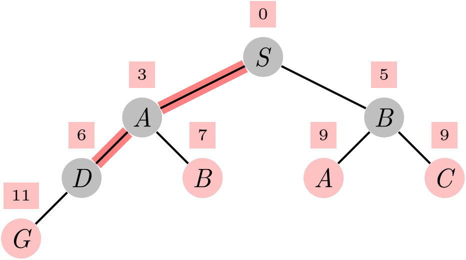


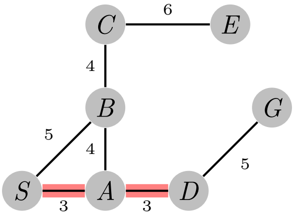
iterations/visits: 3

$D : 6$	$B : 7$	$A : 9$	$C : 9$

Branch and Bound

Can you notice the pattern of the structure? Also, we found G, so shall we stop?



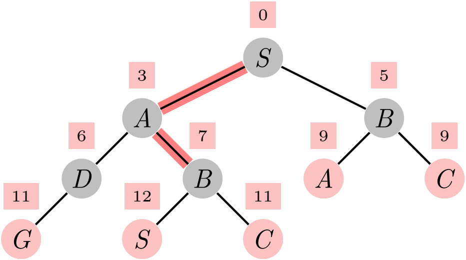


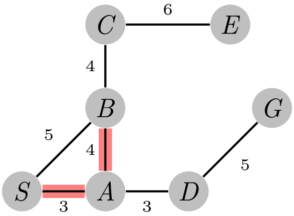
iterations/visits: 4

$B:7$	$A:9$	$C:9$	$G:11$

Branch and Bound

Have you noticed what happened?



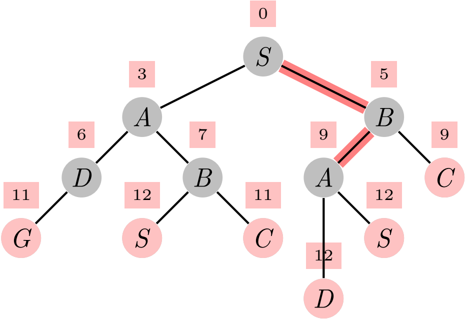


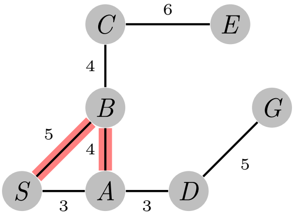
iterations/visits: 5

$A:9$	$C:9$	$C:11$	$G:11$	$S:12$

Branch and Bound

Now, what is such structure?



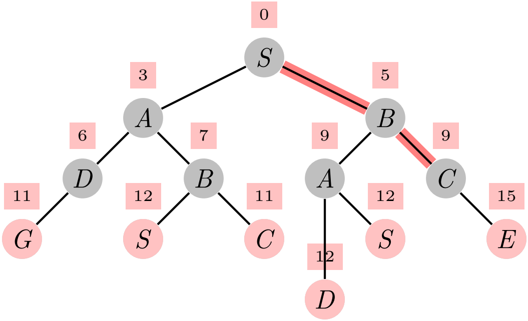


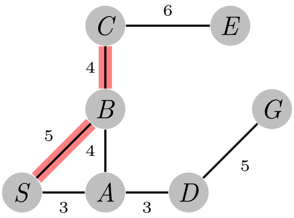
iterations/visits: 6

$C:9$	$C:11$	$G:11$	$S:12$	$D:12$	$S:12$

Branch and Bound

Now, what is such structure?



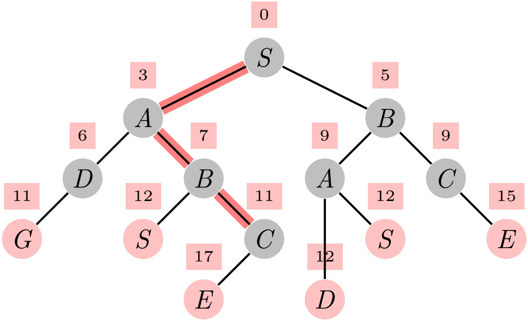


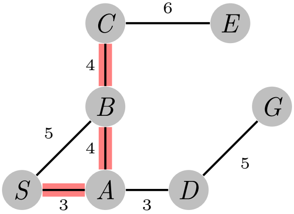
iterations/visits: 7

$C:11$	$G:11$	$S:12$	$D:12$	$S:12$	$E:15$

Branch and Bound

Now, what is such structure?



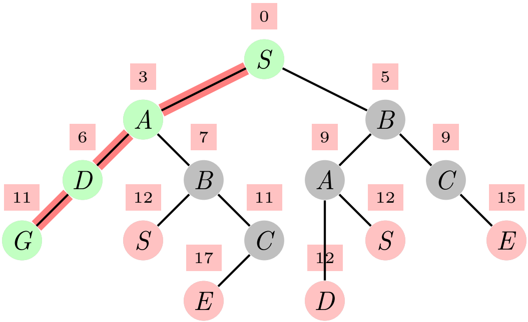


iterations/visits: 8

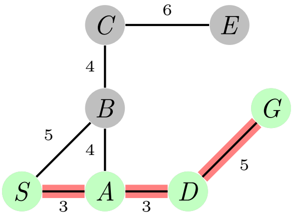
$G:11$	$S:12$	$D:12$	$S:12$	$E:15$	$E:17$

Branch and Bound

Yes, it is a **Priority Queue**, where the priority is the overall path cost.



$S:12$	$D:12$	$S:12$	$E:15$	$E:17$



iterations/visits: 9

Branch and Bound

Algorithm 3: $\text{Branch-Bound}(root, goal)$

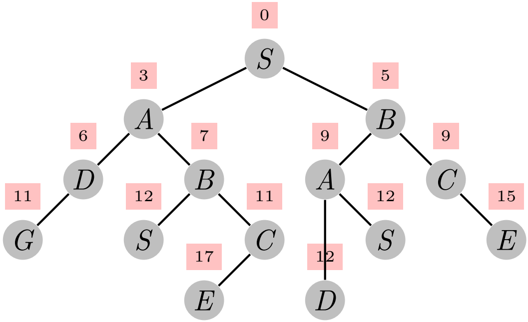
```
def PQ to be Priority Queue
PQ.enqueue(root, 0)
node ← root
while PQ ≠ φ ∧ node ≠ goal do
    node, path_cost ← PQ.dequeue()
    for n ∈ adjacent(node) do
        // loop over the links and their costs
        PQ.enqueue(n, path_cost + cost(n))
    end
end
```

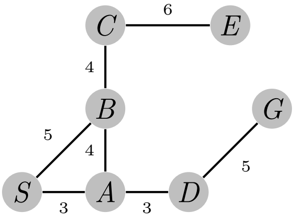
Search Space Pruning

One question over the previous algorithm is: **cannot we do better?**

Search Space Pruning

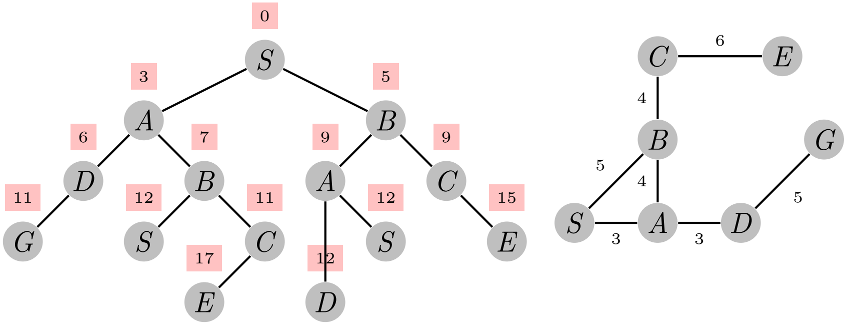
To answer this, we need to check the produced **search space**, on the left.





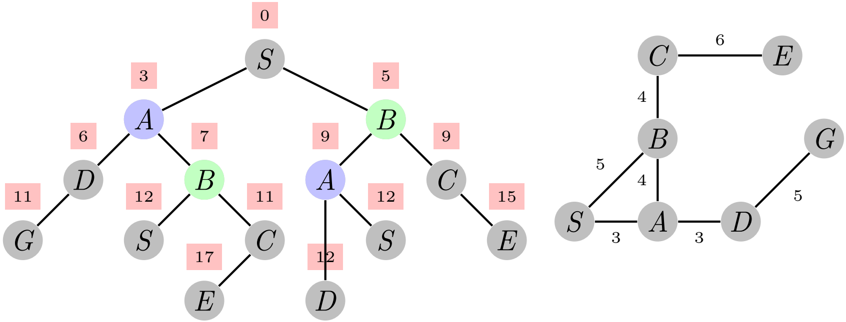
Search Space Pruning

We know the **denser** the search space, more nodes to visit, the more time our algorithms takes. Hence, ***pruning*** it, eliminating some of the nodes, should improve it, how can we do this?



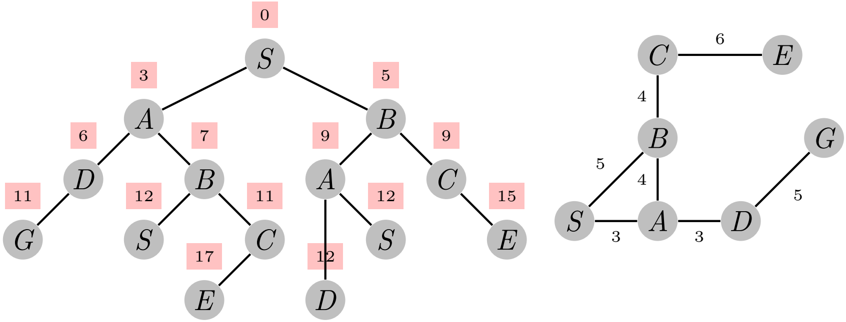
Search Space Pruning

One way is to remove duplicates; do you think we need to check A or B twice? We can do this using a simple **visited list**



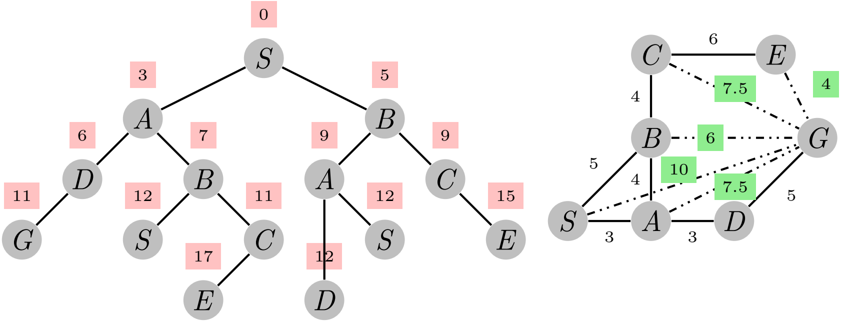
Search Space Pruning

Could you think of other techniques to prune the tree?



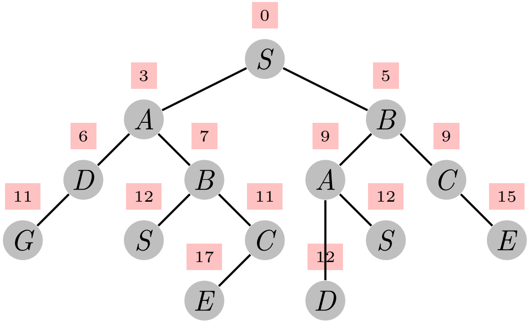
Search Space Pruning

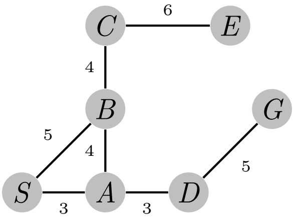
One solution is if we could have an estimate of where the goal is; this may make us more oriented towards searching in particular regions than others; this is briefly, the **heuristics**.



Search Space Pruning

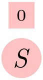
We are covering both the **visited list** and **heuristic** techniques in the next subsections, applying them to branch and bound.

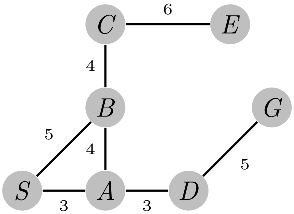




Adding visited list to Branch and Bound

Let's follow on the same graph, but while using a **visited list**.



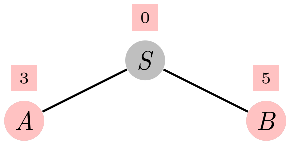


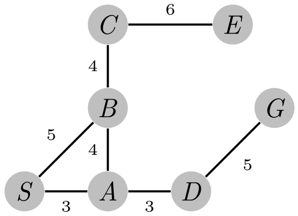
PQ: $\boxed{S : 0}$

$\boldsymbol{visited}$:

Adding visited list to Branch and Bound

Let's follow on the same graph, but while using a **visited list**.





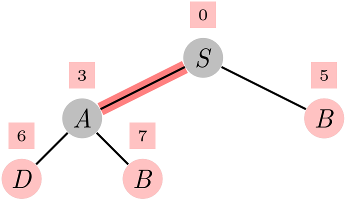
iterations/visits: 1

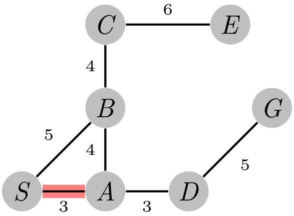
PQ: $\boxed{A:3}\boxed{B:5}$

$visited$: $\boxed{S}$

Adding visited list to Branch and Bound

Let's follow on the same graph, but while using a **visited list**.





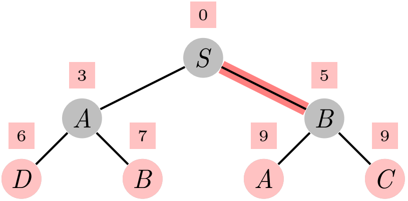
iterations/visits: 2

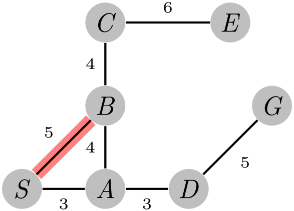
PQ: $\boxed{B:5}\boxed{D:6}\boxed{B:7}$

$visited$: $\boxed{S}\boxed{A}$

Adding visited list to Branch and Bound

Let's follow on the same graph, but while using a **visited list**.



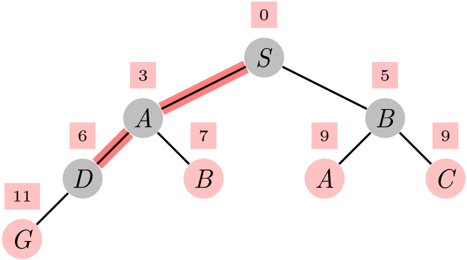


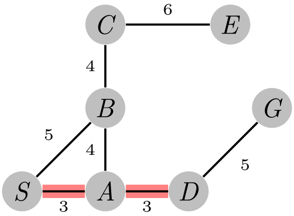
iterations/visits: 3

PQ:	$D:6$	$B:7$	$A:9$	$C:9$
visited:	S	A	B	

Adding visited list to Branch and Bound

Now, the queue has B, should we visit it next?



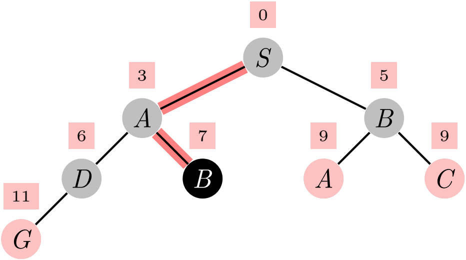


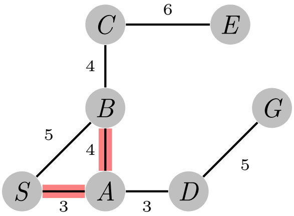
iterations/visits: 4

PQ:	$B:7$	$A:9$	$C:9$	$G:11$
$visited$:	S	A	B	D

Adding visited list to Branch and Bound

Well, no: B is already visited, so there is already a shorter path to it.





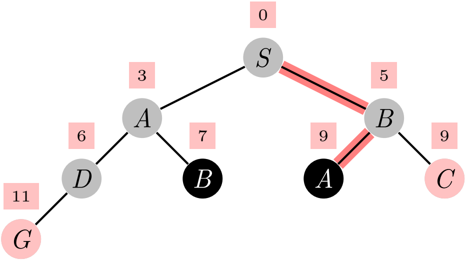
iterations/visits: 5

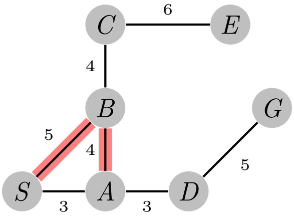
PQ: $A:9$ | $C:9$ | $G:11$

$visited$: S | A | B | D

Adding visited list to Branch and Bound

Again, the next was A, so it is blocked as well.





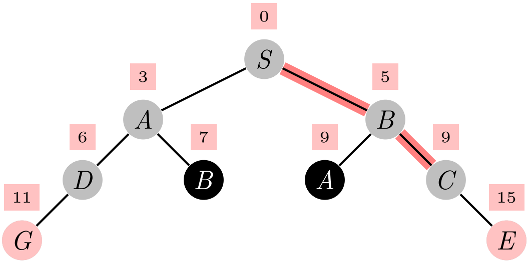
iterations/visits: 6

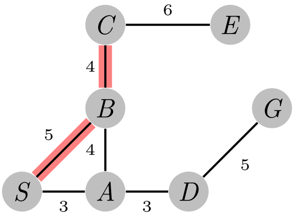
PQ: $\boxed{C:9}\boxed{G:11}$

$visited$: $\boxed{S}\boxed{A}\boxed{B}\boxed{D}$

Adding visited list to Branch and Bound

We, then, proceed



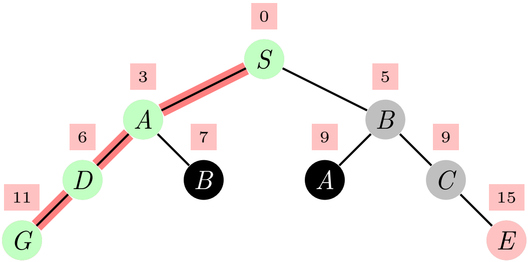


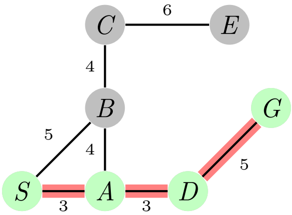
iterations/visits: 7

PQ:	$G:11$	$E:15$			
visited:	S	A	B	D	C

Adding visited list to Branch and Bound

Finally, we got our solution. Could you see the pruning effect?





iterations/visits: 8

PQ: $\boxed{E : 15}$

$\boldsymbol{visited}$: | S | A | B | D | C |

Adding visited list to Branch and Bound

Algorithm 4: $\textsc{Branch-Bound-Visited}(root, goal)$

```
def PQ to be Priority Queue
visited ← {}
PQ.enqueue(root, 0)
node ← root
while PQ ≠ φ ∧ node ≠ goal do
    node, path_cost ← PQ.dequeue()
    if node ∉ visited then
        for n ∈ adjacent(node) do
            // loop over the links and their costs
            PQ.enqueue(n, path_cost + cost(n))
        end
    end
end
```

Branch and Bound + Heuristics

Now, we start with showing what heuristics are, first. Heuristics are **optimistic estimation** to the future cost.

Branch and Bound + Heuristics

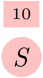
Now, we start with showing what heuristics are, first. Heuristics are **optimistic estimation** to the future cost. Although they are not accurate, but because they capture some of the information, we use them within our algorithms to **guide** our search to **more promising areas** than others, leading to faster findings.

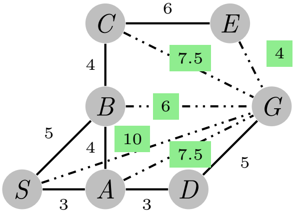
Branch and Bound + Heuristics

We will be using the same map, where our heuristic values, in dashed lines, represent the **direct Euclidean distance** between the nodes. They are not accurate because shorter distances does not necessarily mean we are close to our goal.

Branch and Bound + Heuristics

Our cost, priority, = actual cost to current node + the estimate to the goal. Let's do it without a visited list.

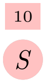


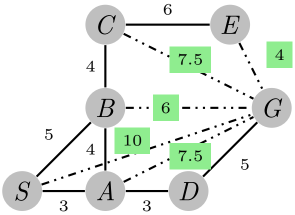


PQ: $\boxed{S : 10}$

Branch and Bound + Heuristics

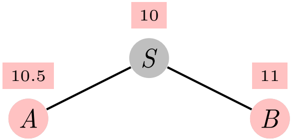
Note our initial cost $= 10$, which is 0 actual cost $+10$ as estimate

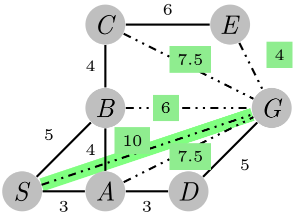




Branch and Bound + Heuristics

Again, the cost = true cost + heuristic



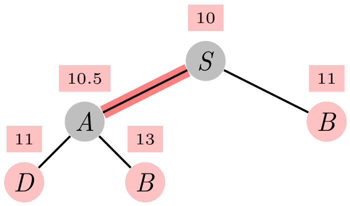


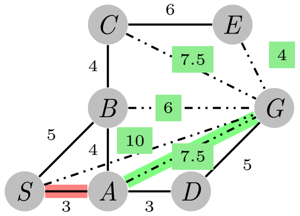
iterations/visits: 1

PQ: $\boxed{A : 10.5}\boxed{B : 11}$

Branch and Bound + Heuristics

Again, the cost = true cost + heuristic





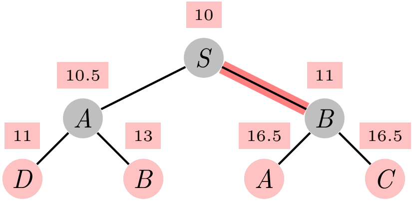
iterations/visits: 2

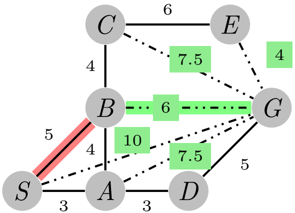
PQ:

$B:11$	$D:11$	$B:13$

Branch and Bound + Heuristics

Again, the cost = true cost + heuristic





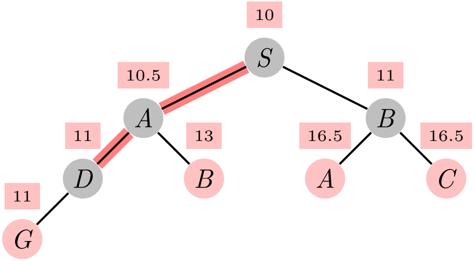
iterations/visits: 3

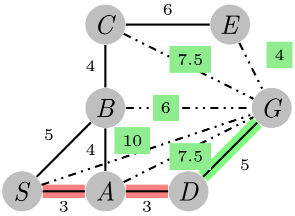
PQ:

$D : 11$	$B : 13$	$A : 16.5$	$C : 16.5$

Branch and Bound + Heuristics

Again, the cost = true cost + heuristic



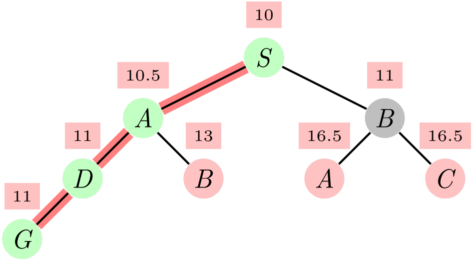


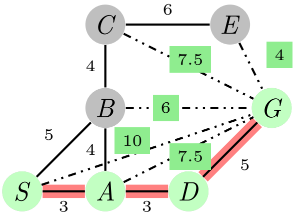
iterations/visits: 4

PQ:	$G:11$	$B:13$	$A:16.5$	$C:16.5$

Branch and Bound + Heuristics

and we got it in just 5 iterations.





iterations/visits: 5

PQ: | $B : 13$ | $A : 16.5$ | $C : 16.5$ |

Branch and Bound + Heuristics

Algorithm 5: BRANCH-BOUND-HEURISTICS(*root*, *goal*)

```
def PQ to be Priority Queue
PQ.enqueue(root, h(root))
node ← root
while PQ ≠ ϕ ∧ node ≠ goal do
    node, node_cost ← PQ.dequeue()
    path_cost ← node_cost − h(node)
    for n ∈ adjacent(node) do
        // h(n) is the heuristic estimate from n to goal
        total_cost ← path_cost + cost(n) + h(n)
        PQ.enqueue(n, total_cost)
    end
end
```

A* Algorithm

We studied two cool search pruning techniques:

- visited list
- heuristics

Now, you might ask what happens if we used both of these cool techniques.

A* Algorithm

We studied two cool search pruning techniques:

- visited list
- heuristics

Now, you might ask what happens if we used both of these cool techniques. We get an even more **awesome** algorithm, which is the **A***. It is typically using both techniques with the Branch-and-Bound algorithm. Let's check the pseudo-code

A* Algorithm

Algorithm 6: A-Star($root$, $goal$)

$visited \leftarrow \{\}$, $node \leftarrow root$
$PQ.\text{enqueue}(root, \text{h}(root))$
while $PQ \neq \phi \wedge node \neq goal$ **do**
    $node,\ node\_cost \leftarrow PQ.\text{dequeue}()$
    if $node \notin visited$ **then**
        $visited \leftarrow visited \cup \{node\}$
        $path\_cost \leftarrow node\_cost - \text{h}(node)$
        for $n \in adjacent(node)$ **do**
            $PQ.\text{enqueue}(n, path\_cost + \text{cost}(n) + \text{h}(n))$
        end
    end
end

1. Graphs & Graph Algorithms
2. Unweighted Graphs
3. Weighted Graphs
4. Path Construction

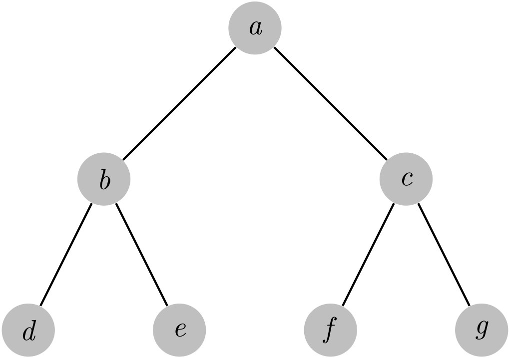
Path Construction from Traversing Algorithm

Cool; now, we have shown how many traversing algorithms, *which visits the nodes*, and checked the pseudo-code, but and important question is

how to construct a path from a traversing algorithm? (DFS for example)

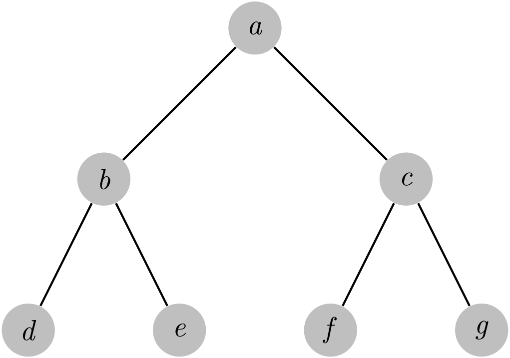
Path Construction from Traversing Algorithm

First, to differentiate between traversing and path, we revisit the same DFS tree as before; the ordering of the visited nodes is a, b, d, e, c, f, g.



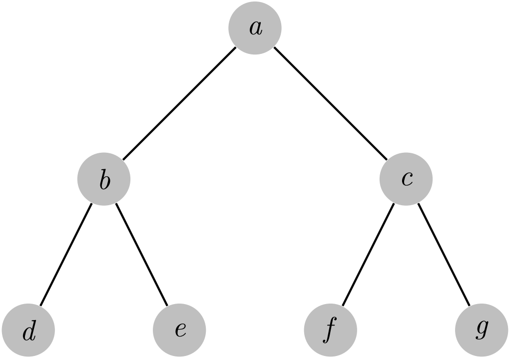
Path Construction from Traversing Algorithm

the ordering of the visited nodes is a, b, d, e, c, f, g. Does this mean that the **path** according to our algorithm from $a \rightarrow f$ is a, b, d, e, c, f?



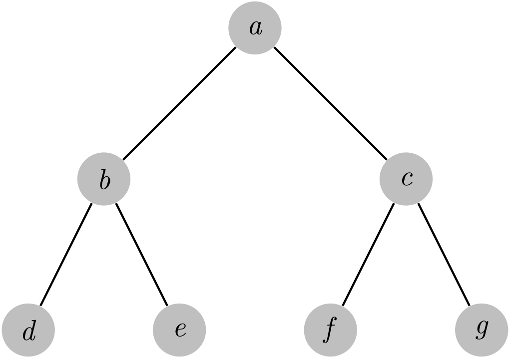
Path Construction from Traversing Algorithm

the ordering of the visited nodes is a, b, d, e, c, f, g. Does this mean that the **path** according to our algorithm from $a \rightarrow f$ is a, b, d, e, c, f? well, **No**. This is just a traversing order.



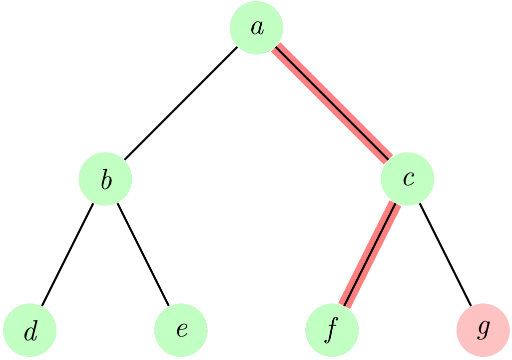
Path Construction from Traversing Algorithm

In that case, what is the path constructed from $a \rightarrow f$ according to our algorithm?



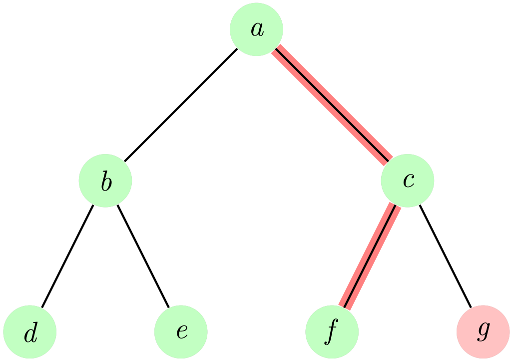
Path Construction from Traversing Algorithm

In that case, what is the path constructed from $a \rightarrow f$ according to our algorithm? It is a, c, f as we see from our previous execution.



Path Construction from Traversing Algorithm

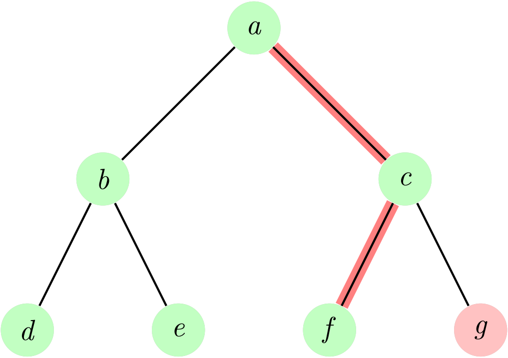
so, how could we construct such path from our algorithm?



Path Construction from Traversing Algorithm

so, how could we construct such path from our algorithm?

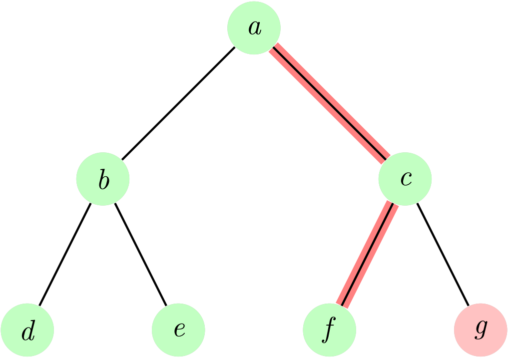
- store the whole paths, instead of just nodes



Path Construction from Traversing Algorithm

so, how could we construct such path from our algorithm?

- store the whole paths, instead of just nodes; Storage hungry



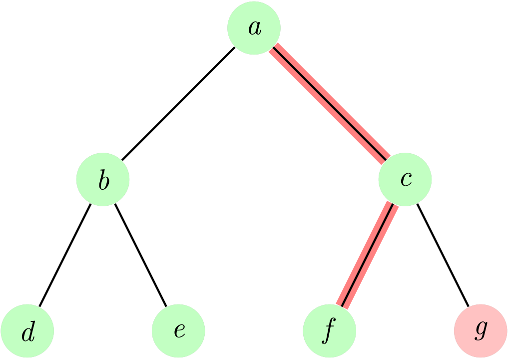
Path Construction from Traversing Algorithm

so, how could we construct such path from our algorithm?

- using **parent-child** structure.

Child $\longleftarrow$ Parent

$b \longleftarrow a$
$d \longleftarrow b$
$e \longleftarrow b$
$c \longleftarrow a$
$f \longleftarrow c$
$g \longleftarrow c$

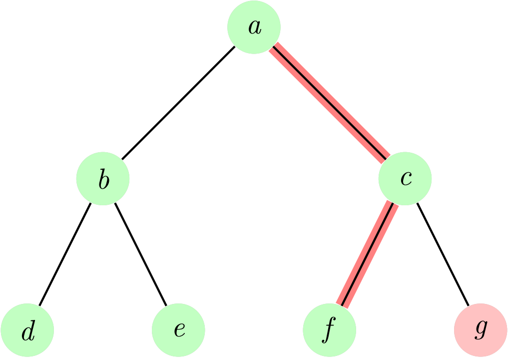


Path Construction from Traversing Algorithm

start from your goal, f, and move backward until getting your start.

Child $\longleftarrow$ Parent

$b \longleftarrow a$
$d \longleftarrow b$
$e \longleftarrow b$
$c \longleftarrow a$
$f \longleftarrow c$
$g \longleftarrow c$

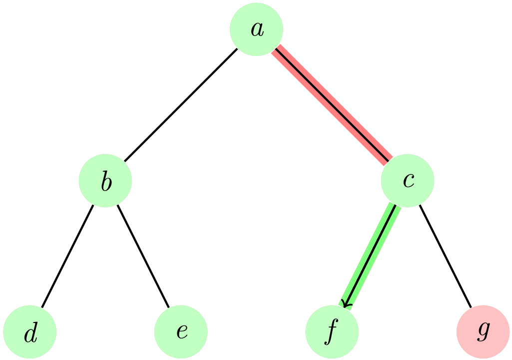


Path Construction from Traversing Algorithm

Path: $c \to f$

Child $\longleftarrow$ Parent

$b \longleftarrow a$
$d \longleftarrow b$
$e \longleftarrow b$
$c \longleftarrow a$
$f \longleftarrow c$
$g \longleftarrow c$

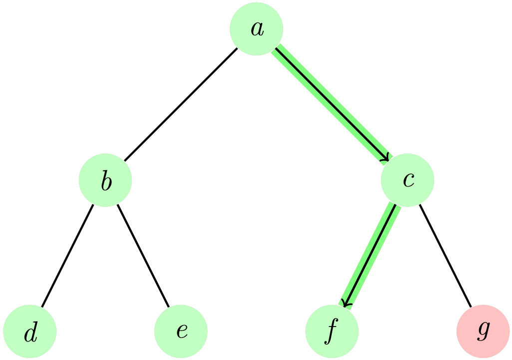


Path Construction from Traversing Algorithm

Path: $\boxed{a \rightarrow c \rightarrow f}$

Child $\longleftarrow$ Parent

$b \longleftarrow a$
$d \longleftarrow b$
$e \longleftarrow b$
$c \longleftarrow a$
$f \longleftarrow c$
$g \longleftarrow c$

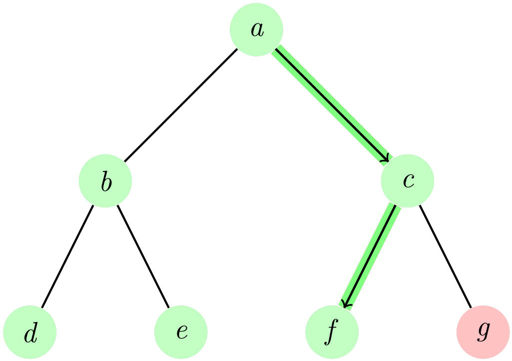


Path Construction from Traversing Algorithm

We can this technique for all the traversing algorithms, mentioned within this module.

Child $\longleftarrow$ Parent

$b \longleftarrow a$
$d \longleftarrow b$
$e \longleftarrow b$
$c \longleftarrow a$
$f \longleftarrow c$
$g \longleftarrow c$



Thanks!