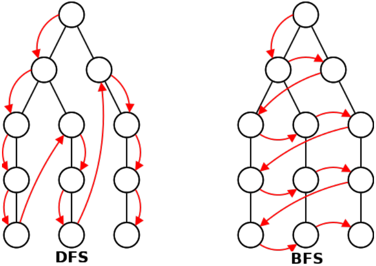
Graph Traversal

Pedro Ribeiro

DCC/FCUP

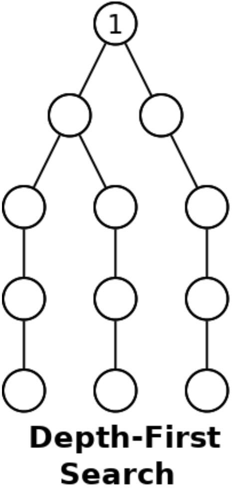
2019/2020

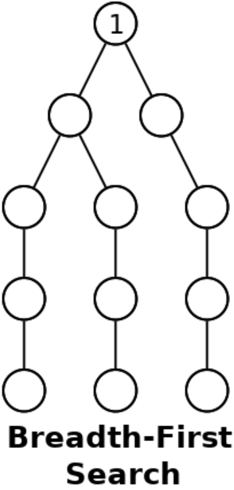


Graph Traversal

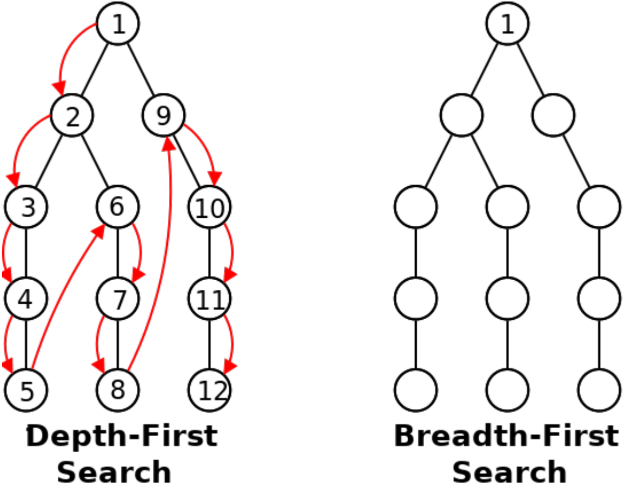
- One of the most important graph related tasks its to how to **traverse** it, that is, **passing trough all nodes** using the **connections between them**
- We call this a **graph traversal** (or graph search)
- There are two main graph traversal algorithms, that differ on the **order of traversal**:
  - **Depth-First Search (DFS)**
    Traverse all the graph connected to an adjacent node before entering the next adjacent node
  - **Breadth-First Search (BFS)**
    Traverse the nodes by increasing order of its distance in number of edges to the source node

Graph Traversal

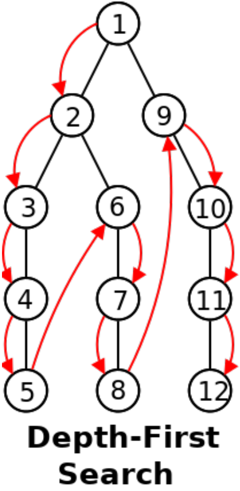


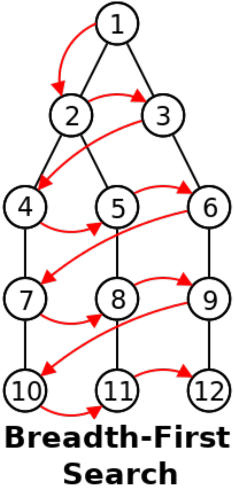


Pesquisa em Grafos



Pesquisa em Grafos





Pesquisa em Grafos

- On its essence, DFS and BFS are doing the "same": **traverse all nodes**
- When to use one or the other depends on the problem and on the **order on which we want to traverse the nodes**
- We will see how to **implement** both and we will give example applications

DFS

The "skeleton" of a DFS:

DFS (recursive version)

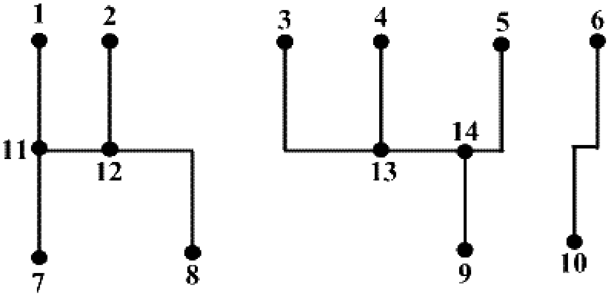
dfs(node v):
  mark v as visited
  For all nodes w adjacent to v **do**
    If w was not yet visited **then**
      dfs(w)

Complexity:

- Temporal:
  - Adjacency List: $\mathcal{O}(|V| + |E|)$
  - Adjacency Matrix: $\mathcal{O}(|V|^2)$
- Spatial: $\mathcal{O}(|V|)$

Connected Components

- Finding **connected components** of a graph G
- Example: the following graph has **3 connected components**



Connected Components

The "skeleton" of a program to solve this:

Finding connected components

count $\leftarrow 0$
mark all nodes as **not visited**
For all nodes v of the graph **do**
  If v is not yet visited **then**
    count $\leftarrow$ count $+ 1$
    dfs(v)
write($count$)

Temporal complexity:

- Adjacency list: $\mathcal{O}(|V| + |E|)$
- Adjacency matrix: $\mathcal{O}(|V|^2)$

Implicit Graphs

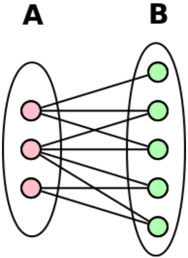
- We do not have to always explicitly store the graph.
- Example: finding the number of "blobs" (connected areas) on matrix. Two cells are adjacent if they are connected vertically or horizontally.

```
#.##..##                        1.22..33
#.....##                        1.....33
...##...   -->  4 blobs  -->    ...44...
...##...                        ...44...
```

- To solve we simply do $dfs(x, y)$ to visit position (x, y), where the adjacent nodes are $(x+1, y), (x-1, y), (x, y+1)$ e $(x, y-1)$
- Calling a DFS to "color" the connected components is known as doing a **Flood Fill**.

Bipartite Graphs

- A **bipartite graph** is a graph where we can divide the nodes in two groups A and where each edge connects a node from A into a node from B:
  - There cannot be any edge from A to A
  - There cannot be any edge from B to B

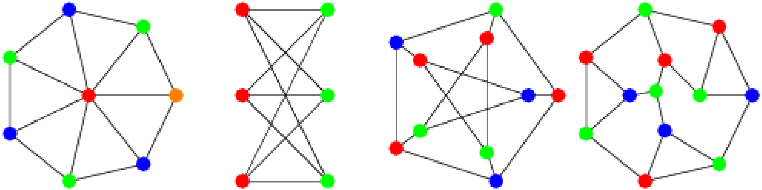


- Many real graph are of this type. Some examples:
  - Products and buyers
  - Movies and actors
  - Books and authors
  - ...

Bipartite graphs

Coloring Graphs

- The problem of **graph coloring** implies discovering a color allocation such that two neighbor nodes never have the same color.



- Given a graph, what is the minimum number of colors we need? (this is the *chromatic number* of a graph)
    - For a general graph this an hard problem and there are no known polynomial solutions.
      (it is one of the original 21 NP-complete problems)

Bipartite Graphs
DFS algorithm

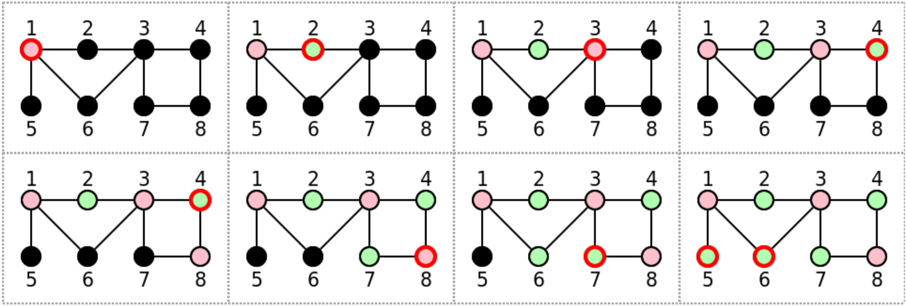
- Knowing if a graph is bipartite is a particular case of graph coloring
- Bipartite graph $\leftrightarrow$ **can we color with 2 colors?**
- We can adapt *dfs* to test for this:

Algorithm to test if a graph is bipartite

Make a dfs from node v and paint that node with a certain color
For each neighbor node w of v:

- If w was not visited, do dfs(w) and paint w with a different color than v
- If w was already visited, check if the color is different
  - If the color is the same, the graph is not bipartite!

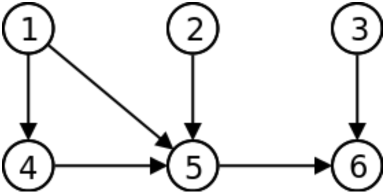
- Black node: not visited
- Red node: group A
- Green node: group B



Topological Sorting

- Given a directed and acyclic graph G, find a node ordering such that u comes before v if and only if there is no (v, u) edge.

- Example: for the graph below, a possible topological sorting would be: $1, 2, 3, 4, 5, 6$ (or $1, 4, 2, 5, 3, 6$ - there might be many possible topological sortings)



A classical example application is to decide in which order you can execute task that have precedences.

Topological Sorting

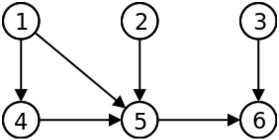
- How to solve this problem with DFS? What is the relationship of the order in which DFS visits the nodes with a topological sorting?

Topological Sorting - $\mathcal{O}(|V| + |E|)$ (list) or $\mathcal{O}(|V|^2)$ (matrix)

order $\leftarrow$ empty list
mark all nodes as **not visited**
For all nodes v of the graph **do**
   If v is not yet visited **then**
      dfs(v)
write(*order*)

dfs(node v):
   mark v as visited
   For all nodes w adjacent to v **do**
      If w is not yet visited **then**
         dfs(w)
   add v to the beginning of list *order*

Topological Sorting



Example of execution:

- $order = \emptyset$

- start dfs(1) | $order = \emptyset$
- start dfs(4) | $order = \emptyset$
- start dfs(5) | $order = \emptyset$
- start dfs(6) | $order = \emptyset$
- end dfs(6) | $order = 6$
- end dfs(5) | $order = 5, 6$
- end dfs(4) | $order = 4, 5, 6$
- end dfs(1) | $order = 1, 4, 5, 6$
- start dfs(2) | $order = 1, 4, 5, 6$
- end dfs(2) | $order = 2, 1, 4, 5, 6$
- start dfs(3) | $order = 2, 1, 4, 5, 6$
- end dfs(2) | $order = 3, 2, 1, 4, 5, 6$

- $order = 3, 2, 1, 4, 5, 6$

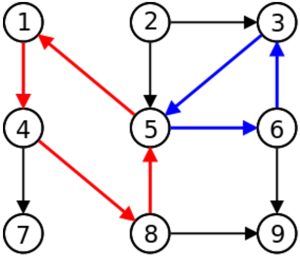
Topological Sorting

- The temporal complexity is $\mathcal{O}(|V| + |E|)$ (list) because we only pass once trough each node and edge.

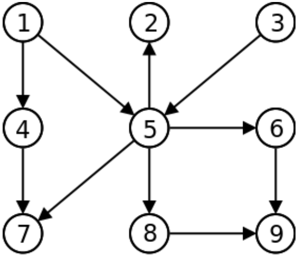
- An algorithm without DFS would be, on a **greedy** fashion, look for a node with in-degree zero, add it to the order and then remove it from the graph, repeating the same process afterwards.

Cycle Detection

- Find if a (directed) graph G if **acyclic** (does not contain cycles)
- Example: the graph on the left contains cycles, the one on the right doesn't



Graph with Cycles



Acyclic Graph

Cycle Detection

Let's use 3 "colors":

- **White** - Node not visited
- **Gray** - Node being visited (we are still exploring descendants)
- **Black** - Node already visited (we visited all descendants)

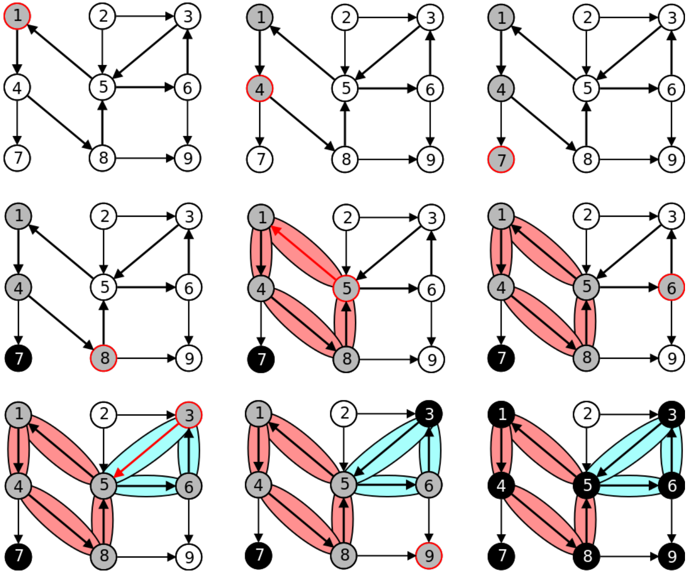
Cycle Detection - $\mathcal{O}(|V| + |E|)$ (list) or $\mathcal{O}(|V|^2)$ (matrix)

color[$v \in V$] $\leftarrow$ white
For all nodes v of the graph **do**
  If $cor[v]$ = white **then**
    dfs(v)

dfs(node v):
  color[v] $\leftarrow$ gray
  For all nodes w adjacent to v **do**
    If color[w] = gray **then**
      write("Cycle found!")
    Else if color[w] = white **then**
      dfs(w)
  color[v] $\leftarrow$ black

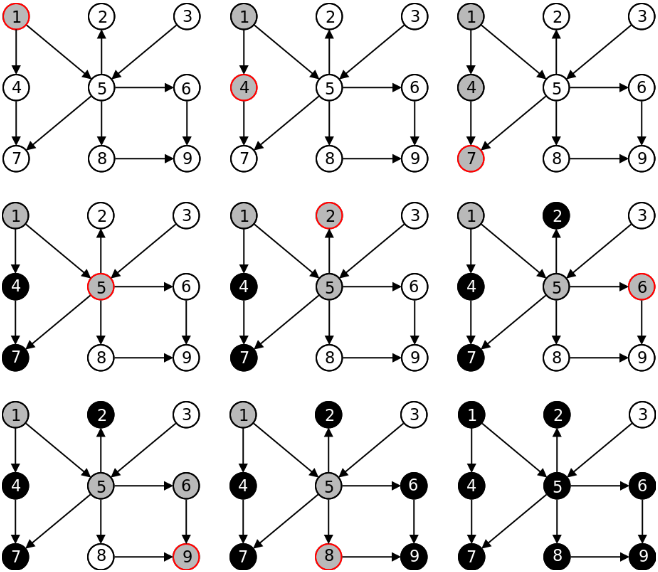
Cycle Detection

Example of execution (Starting on node 1) - Graph with 2 cycles



Cycle Dtection

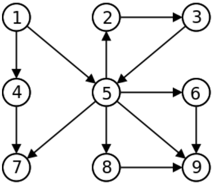
Example of execution (Starting on node 1) - Acyclic graph

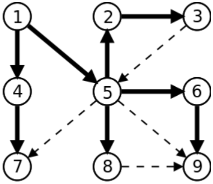


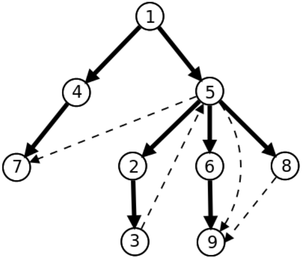
Classifying edges in DFS

Another "angle" for DFS

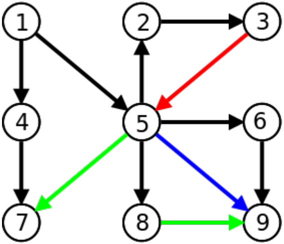
- A DFS implicitly creates a **search tree** that corresponds to the edges that were traversed when exploring the nodes

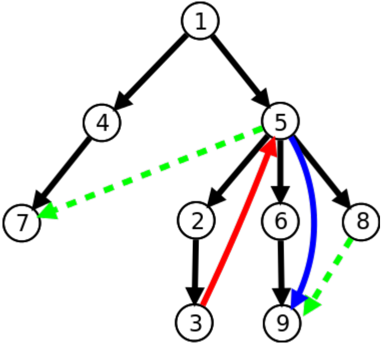






- A visit with DFS classifies edges in 4 categories
  - **Tree Edges** - Edges on DFS tree
  - **Back Edges** - Edge from a node to a predecessor in the tree
  - **Forward Edges** - Edges to a descendant in the tree
  - **Cross Edges** - All the others (from a branch to another branch)





Classifying edges in DFS

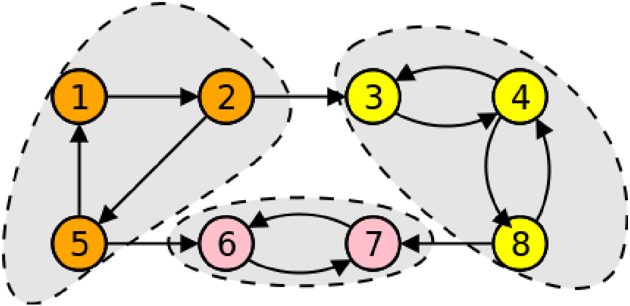
Another "angle" for DFS

- An example application: finding cycles is discovering... **Back Edges**!
- Knowing these edge typs helps to solve problems!
- Note: a undirected graph only has **Tree Edges** and **Back Edges**.

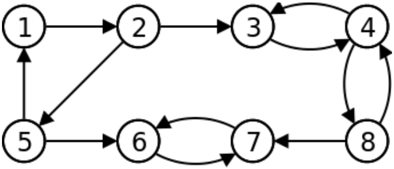
- Decompose a graph in its **strongly connected components**

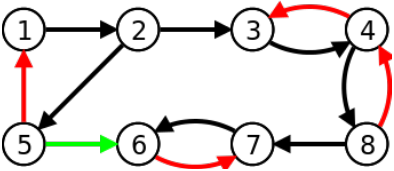
A **strongly connected component** (SCC) its a maximal subgraph where there is a connected (directed) path between all node pairs of that subgraph.

An example graph and its three SCCs:



- How to compute SCCs?
- Let's use our edge types to help:

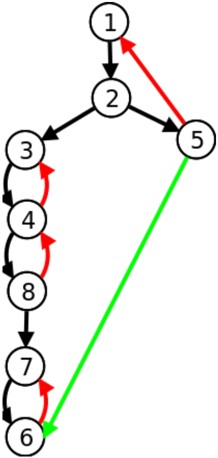




Strongly Connected Components

A more elaborated application of DFS

- Let's take a good look to the DFS tree:

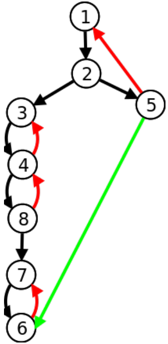


- What is the "lowest" ancestor of a node that is achievable by it?
  - 1: it's again 1
  - 2: it's 1
  - 5: it's 1
  - 3: it's again 3
  - 4: it's 3
  - 8: it's 3
  - 7: it's again 7
  - 6: it's 7
- *Et voilà!* here are our SCCs!

Strongly Connected Components

A more elaborated application of DFS

- Let's add 2 more properties to a node on a DFS visit:
  - **num(i)**: order in which i is visited
  - **low(i)**: lowest $num(i)$ achievable by a subtree that starts in i. It's the minimum between:
    - $num(i)$
    - smallest $num(v)$ between all back edges (i, v)
    - smallest $low(v)$ between all tree edges (i, v)



i	$num(i)$	$low(i)$
1	1	1
2	2	1
3	3	3
4	4	3
5	8	1
6	7	6
7	6	6
8	5	4

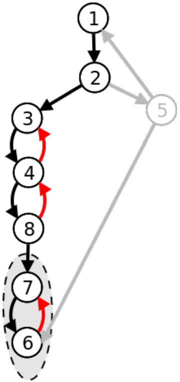
Strongly Connected Components

A more elaborated application of DFS

The idea **Tarjan's algorithm** to discover SCCs:

- Make a **DFS** and in each node i:
  - Put the nodes on a **stack S**
  - Compute and store the values of $\mathbf{num(i)}$ and $\mathbf{low(i)}$.
  - If when exiting the visit to i we have $\mathbf{num(i)} = \mathbf{low(i)}$, then i is the "root" of a SCC. In that case, remove everything from the stack until i and report those elements as a SCC!

Example of execution: when we exit *dfs*(7), we find that *num*(7) = *low*(7) (7 is the "root" of a SCC)



State of stack **S**:

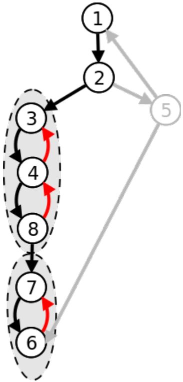
6
7
8
4
3
2
1

We remove from the stack until **7**, and we output the SCC: {6, 7}

Strongly Connected Components

A more elaborated application of DFS

Example of execution: when we exit *dfs*(3), we find that $num(3) = low(3)$ (3 is the "root" of a SCC)



State of stack **S**:

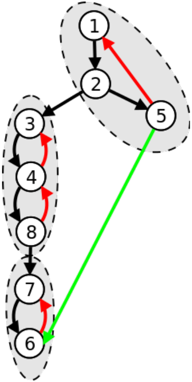
8
4
3
2
1

We remove from the stack until **3**, and we output the SCC: `{8, 4, 3}`

Strongly Connected Components

A more elaborated application of DFS

Example of execution: when we exit $dfs(1)$, we find that $num(1) = low(1)$ (1 is the "root" of a SCC)



State of stack **S**:

5
2
1

We remove from the stack until **1**, and we output the SCC: {5, 2, 1}

Strongly Connected Components

A more elaborated application of DFS

Tarjan's algorithm for SCCs - $\mathcal{O}(|V| + |E|)$ (list)

index $\leftarrow 0$; $S \leftarrow \emptyset$
For all nodes v of the graph **do**
   If $num[v]$ is not yet defined **then**
      dfs_cfc(v)

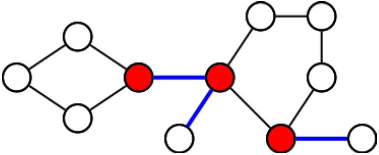
dfs_cfc(node v):
   $num[v] \leftarrow low[v] \leftarrow index$; $index \leftarrow index + 1$; S.push(v)
   /* Traverse edges of v */
   For all nodes w adjacent to v **do**
      If $num[w]$ is not yet defined **then** /* Tree Edge */
         dfs_cfc(w) ; $low[v] \leftarrow min(low[v], low[w])$
      Else if w is in S **então** /* Back Edge */
         $low[v] \leftarrow min(low[v], num[w])$
   If $num[v] = low[v]$ **then** /* We know that we are on a SCC "root" */
      Start new SCC C
      Repeat
         $w \leftarrow$ S.pop() ; Add w to C
      Until $w = v$

Articulation Points and Bridges

An **articulation point** is a **node** whose removal increases the number of connected components

A **bridge** is an **edge** whose removal increases the number of connected components

Example (in red the articulation points; in blue the bridges):



A graph without articulation points is caleed **biconnected**.

Articulation Points

A more elaborated application of DFS

- Finding the **articulation points** is very useful
  - For instance, a graph that is "robust" to attacks should not have articulation points that when "attacked" will disconnect the graph.
- How to compute? A possible "naive" **algorithm**:
  1. Make one DFS and count connected components
  2. Remove from the original graph a node and execute a new DFS, counting again connected components. If the number increases, then it is an articulation points.
  3. Repeat step 2 for all nodes.
- What would be the **complexity** of this method? $\mathcal{O}(|V| \times (|V| + |E|))$, as we will make $|V|$ calls to a DFS, an each call takes $|V| + |E|$.
- **It is possible to do much better... making one single DFS**!

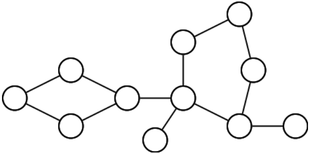
One idea:

- Apply DFS on the graph and obtain the **DFS tree**
- If a **node v has a child w that does not have any path to an ancestor of v, then v is an articulation point**! (since removing it disconnects w from the rest of the graph)
  - This corresponds to see if $low[u] \geq num[v]$
- The only exception is the **root** of the tree. If it has more than one child... then it is also an articulation point!

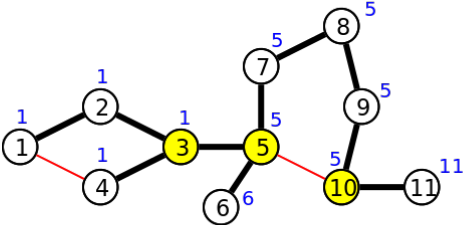
Articulation Points

A more elaborated application of DFS

- An example graph:

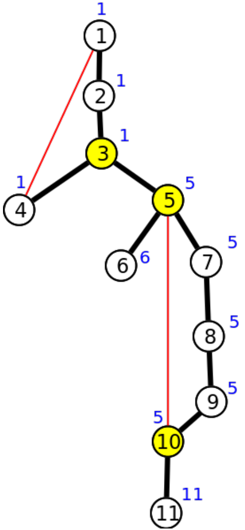


- *num*[*i*] - numbers inside the node
- *low*[*i*] - numbers in blue
- articulation points: nodes in yellow



Articulation Points

A more elaborated application of DFS



- 3 is an articulation point:
  $low[5] = 5 \geq num[3] = 3$
- 5 is an articulation point:
  $low[6] = 6 \geq num[5] = 5$
  ou
  $low[7] = 5 \geq num[5] = 5$
- 10 is an articulation point:
  $low[11] = 11 \geq num[10] = 10$
- 1 is not an articulation point:
  it only has one tree edge

Articulation Points

A more elaborated application of DFS

Algorithm very similar to SCC, but with different DFS:

Finding articulation points - $\mathcal{O}(|V| + |E|)$ (list)

dfs_art(nde v):
  $num[v] \leftarrow low[v] \leftarrow index$; $index \leftarrow index + 1$; S.push(v)
  For all nodes w adjacent to a v **do**
    If $number[w]$ is not yet defined **then** /* Tree Edge */
      dfs_art(w) ; $low[v] \leftarrow min(low[v], low[w])$
      If $low[w] \geq num[v]$ **then**
        write(v + "is an articulation point")
    Else if w is in S **then** /* Back Edge */
      $low[v] \leftarrow min(low[v], num[w])$
  S.pop()

Instead of a stack, we could use the colors (grey means it is in the stack)

Breadth-First Search (BFS)

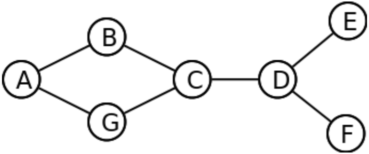
- A Breadth-First Search (BFS) is very similar to a DFS. The only thing that changes is the order in which we visit the nodes.
- Instead of using recursion, it is more common to explicitly keep a queue of non visited nodes (q).

"Skeleton" of a BFS - $\mathcal{O}(|V| + |E|)$ (list)

bfs(node v):
    $q \leftarrow \emptyset$ /* Queue of non visited nodes */
    $q.enqueue(v)$
    mark v as visited
    While $q \neq \emptyset$ /* While there are nodes to visit */
        $u \leftarrow q.dequeue()$ /* Remove first node of q */
        For all nodes w adjacent to u **do**
            If w was not yet visited **then** /* new node */
                $q.enqueue(w)$
                mark w as visited

Breadth-First Search (BFS)

- An example:



1. Initially $q = \{A\}$
2. We remove **A**, we add non visited neighbors ($q = \{B, G\}$)
3. We remove **B**, we add non visited neighbors ($q = \{G, C\}$)
4. We remove **G**, we add non visited neighbors ($q = \{C\}$)
5. We remove **C**, we add non visited neighbors ($q = \{D\}$)
6. We remove **D**, we add non visited neighbors ($q = \{E, F\}$)
7. We remove **E**, we add non visited neighbors ($q = \{F\}$)
8. We remove **F**, we add non visited neighbors ($q = \{\}$)
9. q empty, BFS finished

Breadth-First Search (BFS)

Computing Distances

- Almost anything that can be done with DFS can also be made with BFS
- An important difference is that with BFS we visit the nodes on increasing order of distance to the source (in terms of number of edges)
- In that sense, BFS can compute **shortest paths** between nodes in unweighted graphs.
- Let's see what really changes in the code

Breadth-First Search (BFS)

Computing Distances

- In red the new lines. *node.distance* store the distance to v.

BFS with distances - $\mathcal{O}(|V| + |E|)$ (list)

bfs(node v):
  $q \leftarrow \emptyset$ /* Queue of non visited nodes */
  $q.enqueue(v)$
  $v.distance \leftarrow 0$ /* distance of v to itself is zero */
  mark v as visited
  While $q \neq \emptyset$ /* While there are nodes to visit */
    $u \leftarrow q.dequeue()$ /* Remove first node of q */
    For all nodes w adjacent to u **do**
      If w was not yet visited **then** /* new node */
        $q.enqueue(w)$
        mark w as visited
        $w.distance \leftarrow u.distance + 1$

Breadth-First Search (BFS)

More applications

- BFS can be applied to any graph type
- Consider for example that you want the **shortest distance** between a **starting** cell (S) and an **ending** cell (E) on a 2D maze:

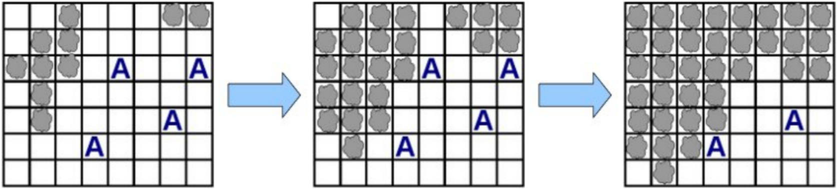
```
########                      ########
#S.....#                      #S12345#
####.###        --->          ####4###
#E.....#     BFS from S       #876567#
########                      ########
```

  - A node in this graph is the position (x, y)
  - The adjacent nodes are $(x+1, y)$, $(x-1, y)$, $(x, y+1)$ and $(x, y-1)$
  - The rest of the BFS remains the same (we take $\mathcal{O}(rows \times cols)$)
  - To store on a queue we need to use a pair (of coordinates)

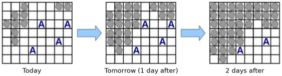
Breadth-First Search (BFS)

More applications

- Let's see a problem from ONI'2010 qualification
- Problem inspired on the eruption of **Eyjafjallajökull volcano**, whose ash cloud caused so many problems in europe's air traffic
- Imagine that the position of the **ash clouds** is given on a matrix, and that in each time unit the cloud expands by one cell horizontally and vertically. `A`'s represent airports.



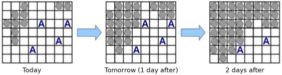
Today | Tomorrow (1 day after) | 2 days after



- The problem asks for:
  - What is **the first airport** being covered by ashes
  - How much time before **all** airports are covered by ashes
- Let $dist(A_i)$ be the distance of i until any cell with ash
- The problem asks for the smallest and largest $dist(A_i)$
- One way would be to make one BFS from each airport
  $\mathcal{O}(num\_airports \times rows \times cols)$
- Another way would be to make one BFS from each ash cell
  $\mathcal{O}(num\_ashes \times linhas \times colunas)$
- Can we do better, using a single BFS?

Breadth-First Search (BFS)

More applications



- Idea: initialize the BFS queue with all the ashes
- Everything else remains the same

```
...#...       ..1#1..      .21#12.      321#123      321#123
..##...       .1##1..      21##12.      21##123      21##123
.####..  ->   1####1.  ->  1####12  ->  1####12  ->  1####12
.......       11111..      111112.      1111123      1111123
##.....       ##1....      ##122..      ##1223.      ##12234
```

- The distances are what we want
- Each cell will only be traversed once $\mathcal{O}(rows \times cols)$

Breadth-First Search (BFS)

More applications

- One last problem where the graph does not "explicitly" exists *[original problem from IOI'1996]*
- Consider the following puzzle (a kind of "2D Rubik's cube")
  - Initial puzzle position is:

1	2	3	4
8	7	6	5

  - In each iteration we can do one of the following moves:
    - **Move A**: swap the two rows

8	7	6	5
1	2	3	4

    - **Move B**: shift the rectangle to the right

4	1	2	3
5	8	7	6

    - **Move C**: rotation (clockwise) of the 4 "middle" cells

1	7	2	4
8	6	3	5

  - How many moves do we need to reach a given position?

- Can be solved with... **BFS**!
- The initial node is... the initial position.
- The adjacent nodes are... the positions we can go to using a single move (A, B or C).
- When we reach the desired position... we necessarily know the shortest distance (nr moves) to get there
- The "hardest" part is to represent the positions :)