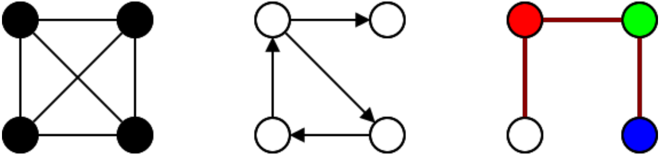
Graphs: Intro, DFS & BFS

Pedro Ribeiro

DCC/FCUP

2022/2023

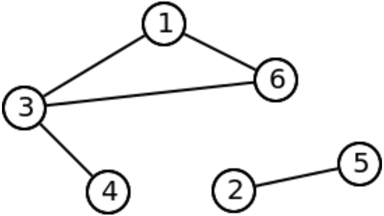


Concept

Graph Definition

Formally, a **graph** is:

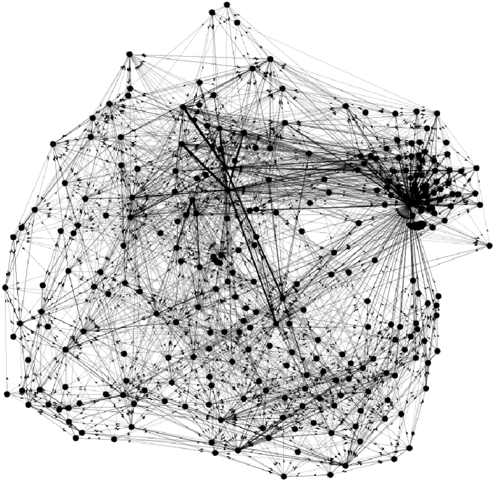
- A set of **nodes**/**vertices** (**V**).
- A set of **links**/**edges** (**E**), that connect pairs of vertices



- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{(1, 6), (1, 3), (3, 6), (3, 4), (2, 5)\}$

What are graphs for?

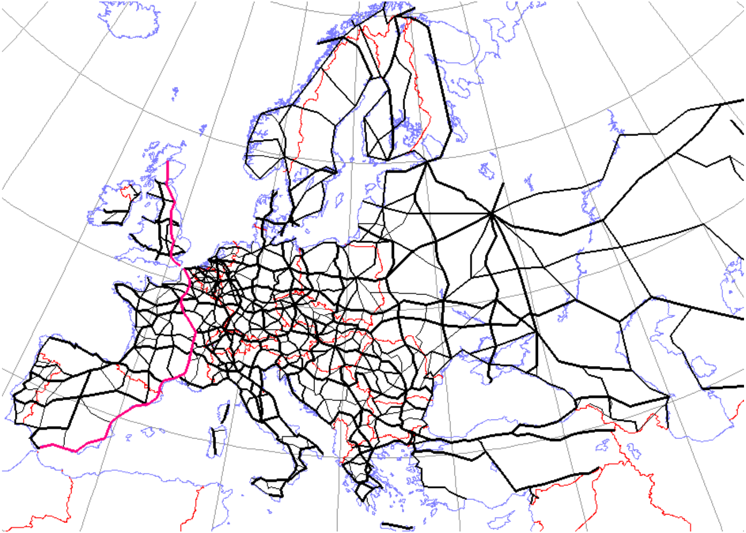
- Graphs are **ubiquitous** in Computer Science and they are present, implicitly or explicitly in many algorithms.
- They can be used in a multitude of applications.



Graph Examples

Networks that exist in the real "physical" world

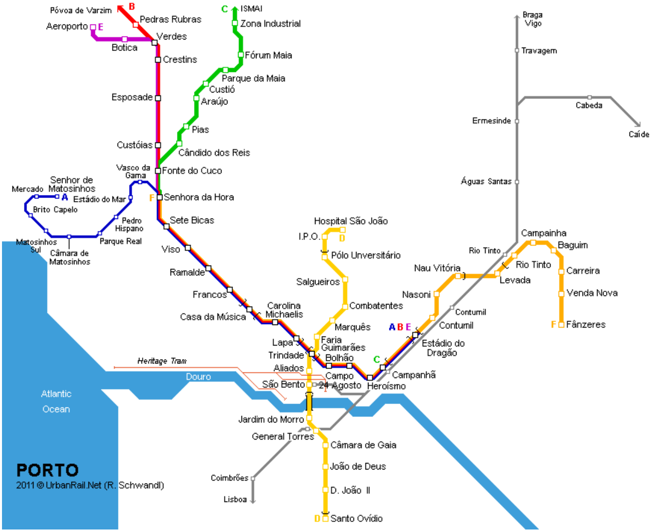
- Road Network



Graph Examples

Networks that exist in the real "physical" world

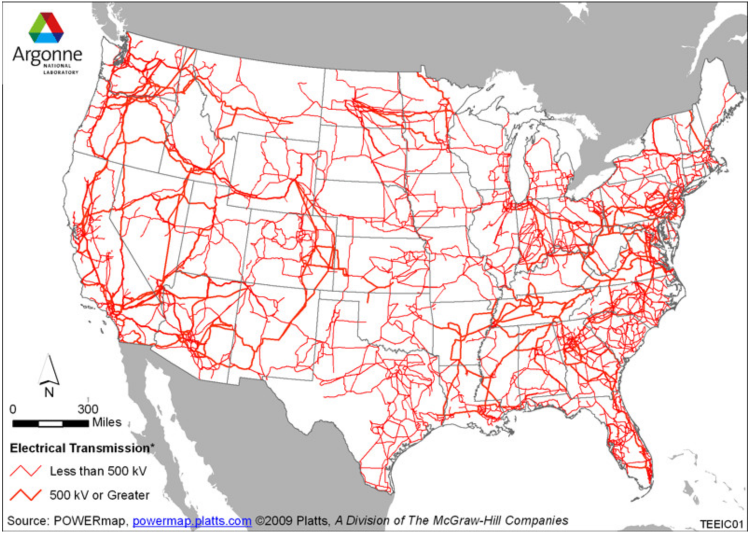
- Public Transportation (ex: subway, train)



Graph Examples

Networks that exist in the real "physical" world

- Power Grid



Source: POWERmap, powermap.platts.com ©2009 Platts, *A Division of The McGraw-Hill Companies*

Graph Examples

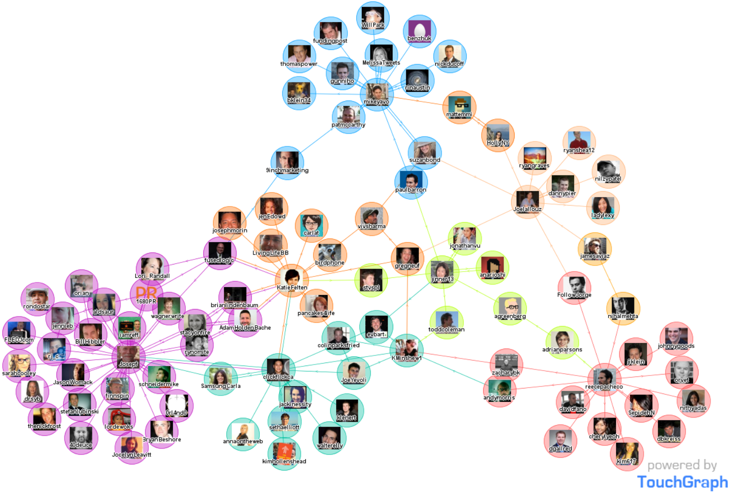
Networks that exist in the real ”physical” world

- Computer Network

Graph Examples

Social Network

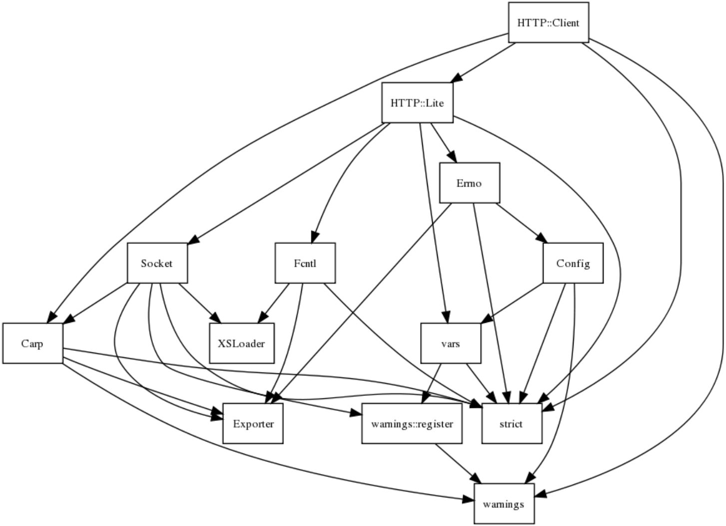
- Facebook (others: Twitter, emails, co-authorship of articles, ...)



Graph Examples

Software Networks

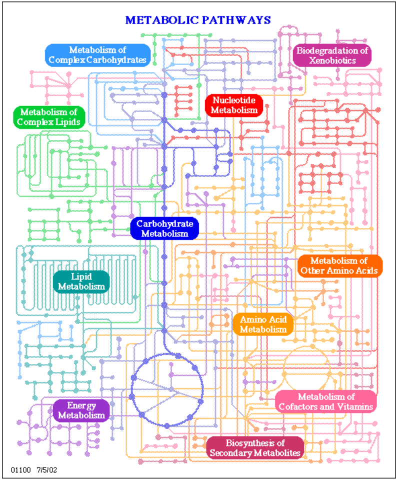
- Module Dependencies (other examples: state, information flow, ...)



Graph Examples

Biological Networks

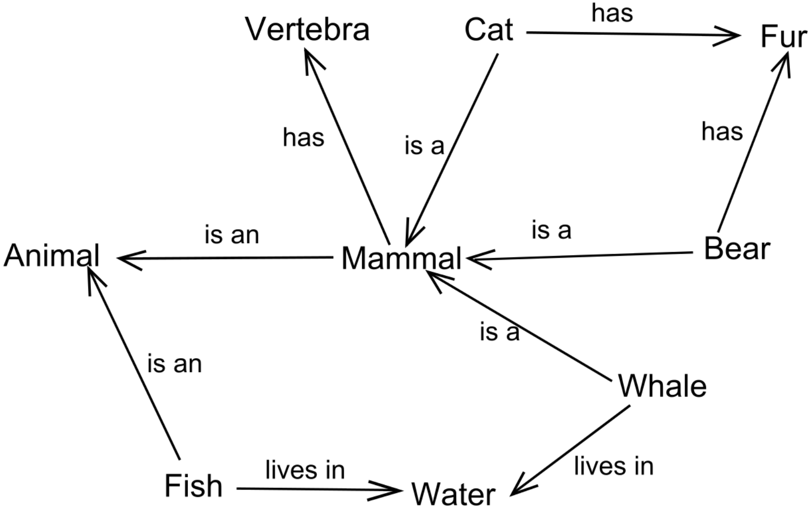
- Metabolic Networks (other examples: protein interaction, brain networks, food webs, phylogenetic trees, ...)



Graph Examples

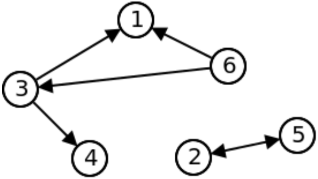
Other Graphs

- Semantic Networks (other examples: world wide web, ...)

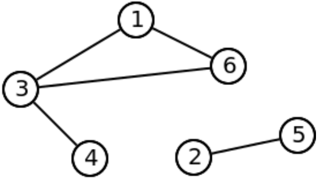


Terminology

- **Directed** graph - each link has a starting node (**origin**) and an **end** node (order matters!). Usually we use arrows to indicate the direction.
- **Undirected** graphs - There is no origin or end, but just a connection



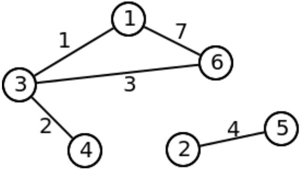
Directed Graph



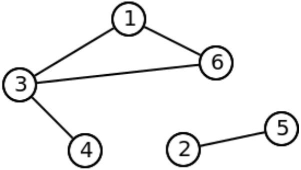
Undirected Graph

Terminology

- **Weighted** graph - there is a vale associated with each link (it could be distance, cost, ...)
- **Unweighted** - there are no weights associated with a link



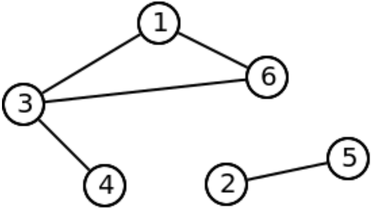
Weighted Graph



Unweighted Graph

Terminology

- **Degree** - number of connections of a node
- In directed graphs we can distinguish between **indegree** and **outdegree**



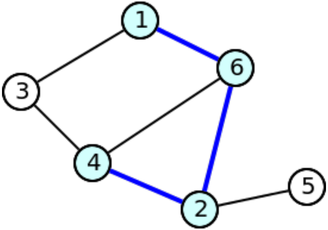
1 has degree 2
2 has degree 1
3 has degree 3
4 has degree 1
5 has degree 1
6 has degree 2

Terminology

- **Adjacent/neighbor** node: two nodes are neighbors if they are linked
- **Trivial graph**: graph with no edges and a single node
- **Self-loop**: link from a node to itself
- **Simple graph:** graph without self-loops and without repeated links (we are mostly going to work with simple graphs)
- **Multigraph:** graph with multiple links between the same node pair
- **Dense graph:** with many links when compared with the maximum possible - $|E|$ of the order of $\mathcal{O}(|V|^2)$
- **Sparse graph:** with few links when compared with the maximum possible - $|E|$ with lower order than $\mathcal{O}(|V|^2)$

Terminology

- **Path**: sequence alternating nodes and edges, such that two consecutive nodes are linked. In simple graphs we typically describe a path using just the nodes.

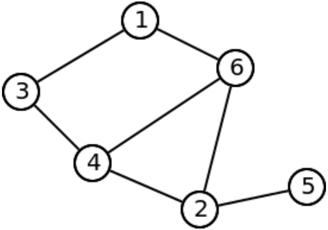


$1 \rightarrow 6 \rightarrow 2 \rightarrow 4$

- **Cycle**: path that starts and ends on the same node (ex: for the above graph, $1 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 1$ is a cycle)
- **Acyclic graph**: graph without cycles

Terminology

- **Size** of a path: number of edges in the path
- **Cost** of a path: if the graph is weighted, we can talk about the cost, which is the sum of the edge weights
- **Distance**: size/cost of the smallest path between two nodes
- **Diameter** of a graph: max distance between two nodes of a graph



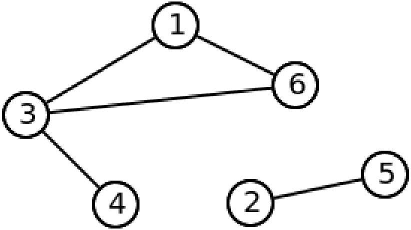
Diameter = 3

	1	**2**	**3**	**4**	**5**	**6**
1	0	2	1	2	3	1
2	2	0	2	1	1	1
3	1	2	0	1	3	2
4	2	1	1	0	2	1
5	3	1	3	2	0	2
6	1	1	2	1	2	0

Distances between nodes

Terminology

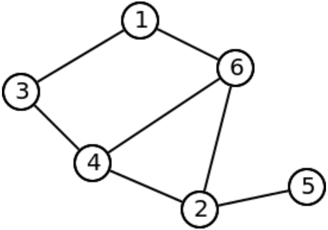
- **Connected Component**: Subset of nodes where there is at least one path between each of them
- **Connected Graph**: Graph with just one connected component (there is a path between all pairs of nodes)



Graph with two connected components: $\{1, 3, 4, 6\}$ e $\{2, 5\}$

Terminology

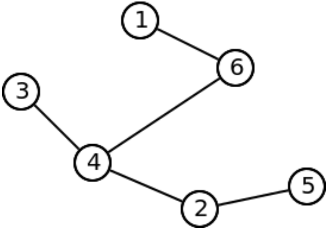
- **Subgraph**: subset of nodes and the edges between them
- **Complete** graph: with links between all pairs of nodes
- **Clique**: a complete subgraph
- **Triangle**: a clique with 3 nodes



Subgraph examples: $\{1,3\}$, $\{1,6,2\}$, $\{2,4,5,6\}$, etc
Example clique: $\{2,4,6\}$ (a triangle)

Terminology

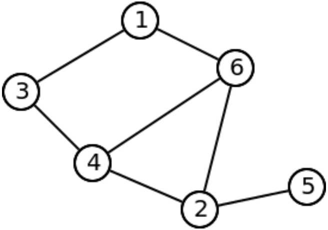
- **Tree**: simple, connected acyclic graph
  (if it has n nodes, then it will have $n - 1$ edges)
- **Forest**: set of multiple disconnected trees



Graph Representation

How to represent a graph?

- **Adjacency Matrix**: $|V| \times |V|$ matrix where the (i, j) cell indicates if there is a link between nodes i and j (if the graph is weighted we can store the weight)
- **Adjacency list**: each node stores a list of its neighbors (if the graph is weighted we have to store pairs (destination,weight))



	1	**2**	**3**	**4**	**5**	**6**
1			X			X
2				X	X	X
3	X			X		
4		X	X			X
5		X				
6	X	X		X		

Adjacency Matrix

1: 3, 6
2: 4, 5, 6
3: 1, 4
4: 2, 3, 6
5: 2
6: 1, 2, 4

Adjacency List

Graph Representation

Some pros and cons:

- **Adjacency Matrix**:
  - Very simple to implement
  - Quick to check if there is a connection between two nodes - $\mathcal{O}(\mathbf{1})$
  - Slow to traverse the neighbors - $\mathcal{O}(|\mathbf{V}|)$
  - Lots of memory wasted (in sparse graphs) - $\mathcal{O}(|\mathbf{V}|^{\mathbf{2}})$
  - Weighted graph implies simply to store the weight in the matrix
  - Adding/Removing edges is simply changing a cell - $\mathcal{O}(\mathbf{1})$

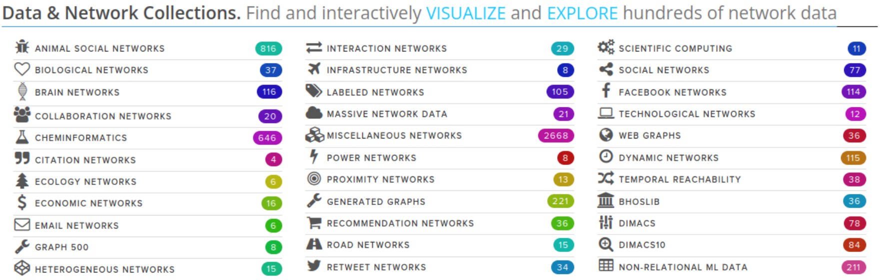
- **Adjacency List**:
  - Slow to see if there is a link between u and v - $\mathcal{O}(\mathbf{degree}(\mathbf{u}))$
  - Quick to traverse the neighbors - $\mathcal{O}(\mathbf{degree}(\mathbf{u}))$
  - Efficient usage of memory - $\mathcal{O}(|\mathbf{V}| + |\mathbf{E}|)$
  - Weighted graph implies adding an attribute to the list
  - Removing edge (u, v) implies traversing the list - $\mathcal{O}(\mathbf{degree}(\mathbf{u}))$

    Note: we can use for instante BSTs (set/map) to improve the efficiency of searching and removing to $\mathcal{O}(\log degree(u))$

Graph datasets

Here are some interesting websites with graphs

- **Network Repository:** `http://networkrepository.com/`
- **Konect:** `http://konect.cc/`
- **SNAP:** `https://snap.stanford.edu/data/`

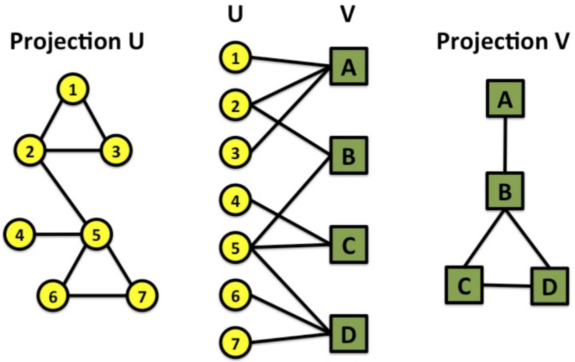


Data & Network Collections. Find and interactively VISUALIZE and EXPLORE hundreds of network data

ANIMAL SOCIAL NETWORKS	816	INTERACTION NETWORKS	29	SCIENTIFIC COMPUTING	11
BIOLOGICAL NETWORKS	37	INFRASTRUCTURE NETWORKS	8	SOCIAL NETWORKS	77
BRAIN NETWORKS	116	LABELED NETWORKS	105	FACEBOOK NETWORKS	114
COLLABORATION NETWORKS	20	MASSIVE NETWORK DATA	21	TECHNOLOGICAL NETWORKS	12
CHEMINFORMATICS	646	MISCELLANEOUS NETWORKS	2668	WEB GRAPHS	36
CITATION NETWORKS	4	POWER NETWORKS	8	DYNAMIC NETWORKS	115
ECOLOGY NETWORKS	6	PROXIMITY NETWORKS	13	TEMPORAL REACHABILITY	38
ECONOMIC NETWORKS	16	GENERATED GRAPHS	221	BHOSLIB	36
EMAIL NETWORKS	6	RECOMMENDATION NETWORKS	36	DIMACS	78
GRAPH 500	8	ROAD NETWORKS	15	DIMACS10	84
HETEROGENEOUS NETWORKS	15	RETWEET NETWORKS	34	NON-RELATIONAL ML DATA	211

Bipartite Graphs

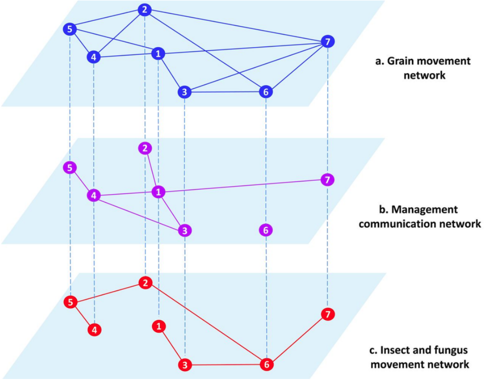
- A **bipartite graph** is a graph whose nodes can be divided into two disjoint sets U and V such that every edge connects a node in U to one in V



- Many (real world) networks come from projections (ex: actors and movies, diseases and genes)

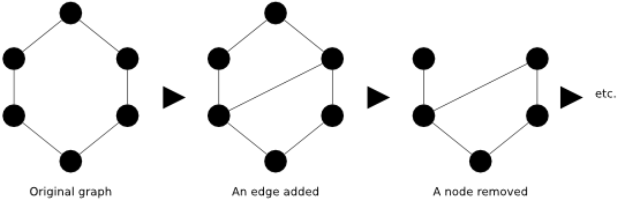
Other Graph Types: Multilayer / Multiplex

- Graphs can have different layers



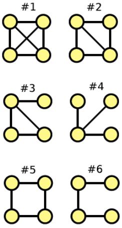
Other Graph Types: Temporal Networks

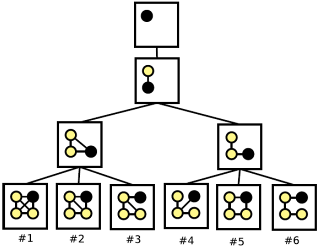
- Graphs can evolve over time



Network Science / Graph Mining

My main research area





PhD Thesis (2011):

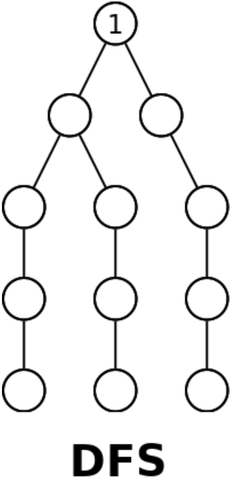
Efficient and Scalable Algorithms for Network Motifs Discovery

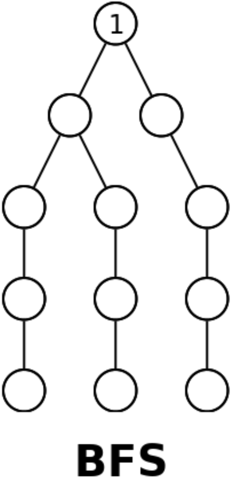
Publications: `http://www.dcc.fc.up.pt/~pribeiro/pubs_by_year.html`

Graph Traversal

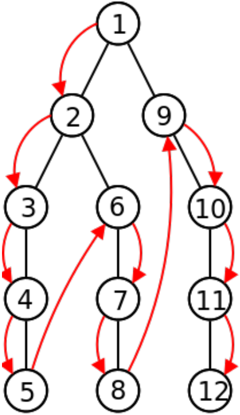
- One of the most important tasks is to **traverse** a graph, that is, **pass trough all its nodes** using the existing **links**
- We call this **graph traversal** (or **graph search**)
- There are two basic traversal types that differ on **the order in which the nodes are traversed**:
  - **Depth-First Search - DFS**
    Traverse the entire subgraph connected to a neighbor before entering the next neighbor node
  - **Breadth-First Search - BFS**
    Traverse the nodes by increasing distance of number of links to reach them

Graph Traversal

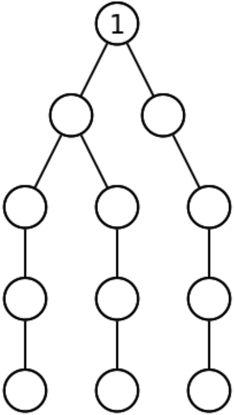




Graph Traversal

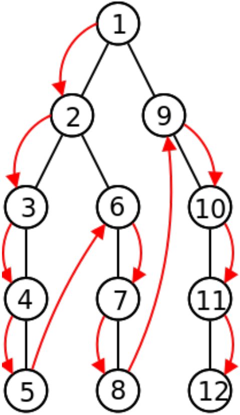


DFS

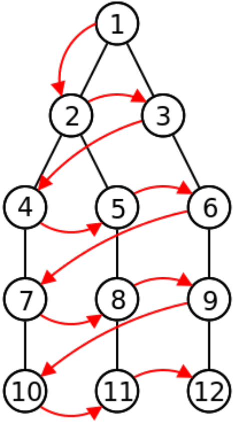


BFS

Graph Traversal



DFS



BFS

Graph Traversal

- In their essence, DFS and BFS do the "same":
  traverse all the nodes
- When to use one or the other depends on the **order that betters suits the problem** that you are solving
- Let's see how to **implement** both and give examples of applications

Depth-First Search - DFS

The "backbone" of a DFS:

DFS (recursive version)

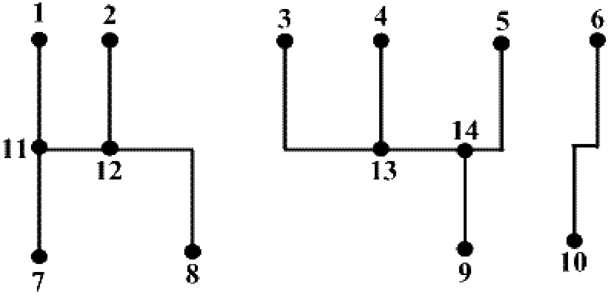
```
dfs(node $v$):
    mark $v$ as visited
    For all neighbors $w$ of $v$ do
        If $w$ has not yet been visited then
            dfs($w$)
```

Complexity:

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

Example Application: Connected Components

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



Example Application: Connected Components

The "backbone" of a program to solve it:

Finding connected components

```
counter ← 0
set all nodes as **not visited**
**For** all nodes $v$ of the graph **do**
    **If** $v$ has not yet been visited **then**
        counter++
        dfs($v$)
write($contador$)
```

Temporal complexity:

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

Implicit Graphs

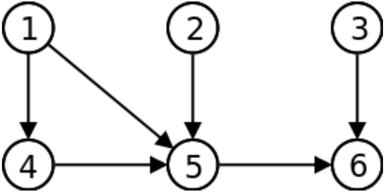
- We do not always need to explicitly store the graph
- Example: find the number of "blobs" (connected spots) in a matrix. Two cells are adjacent if they are connected vertically or horizontally.

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

- To solve we simply need to do $dfs(x, y)$ to visit the cell (x, y) where the neighbors are $(x+1, y), (x-1, y), (x, y+1)$ and $(x, y-1)$
- Using DFS to "color" the connected components is known as doing a **Flood Fill**.

Topological Sorting

- Given a DAG G (directed acyclic graph), find an order of nodes such that u comes before v if and only if there is no edge (v, u)
- 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 are other possible valid orders)



A classic example of application is to decide in which order to execute a set of tasks with precedences.

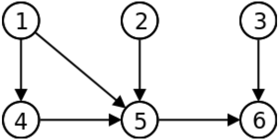
Topological Sorting

- How to solve this problem with DFS? What is the relationship between topological sorting and the DFS node order?

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

```
order ← empty
set all nodes as not visited
For all nodes $v$ of the graph do
   If $v$ has not yet been visited then
      dfs($v$)
write(order)

dfs(node $v$):
   mark $v$ as visited
   For all neighbors $w$ of $v$ do
      If $w$ has not yet been visited then
         dfs($w$)
   add $v$ to the begginning of order
```

Topologic Sorting



Example of execution:

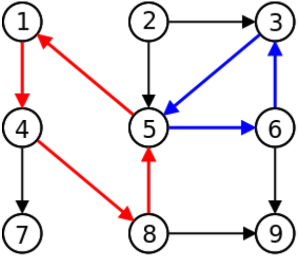
- $order = \emptyset$

- 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(3) | $order = 3, 2, 1, 4, 5, 6$

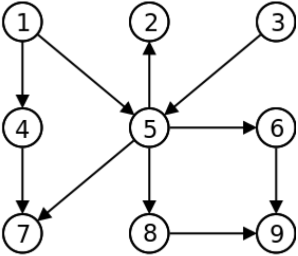
- $order = 3, 2, 1, 4, 5, 6$

Cycle Detection

- Find if a (directed) graph G us **acyclic**
- Example: the left graph has a cycle; the right graph doesn't



Graph with cycles



Acyclic Graph

Cycle Detection

Let's use 3 "colors":

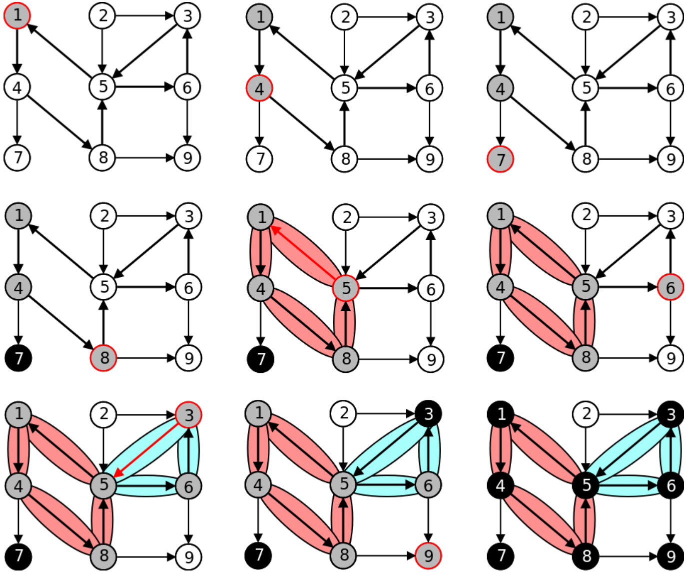
- **White** - node visited node
- **Gray** - node being visited (we are exploring its descendants)
- **Black** - node already visited (we visited all its descendants)

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

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

dfs(node $v$):
   color[$v$] ← gray
   For all neighbors $w$ of $v$ do
      If color[$w$] = gray then
         write("Cycle found!")
      Else if color[$w$] = white then
         dfs($w$)
   color[$v$] ← black
```

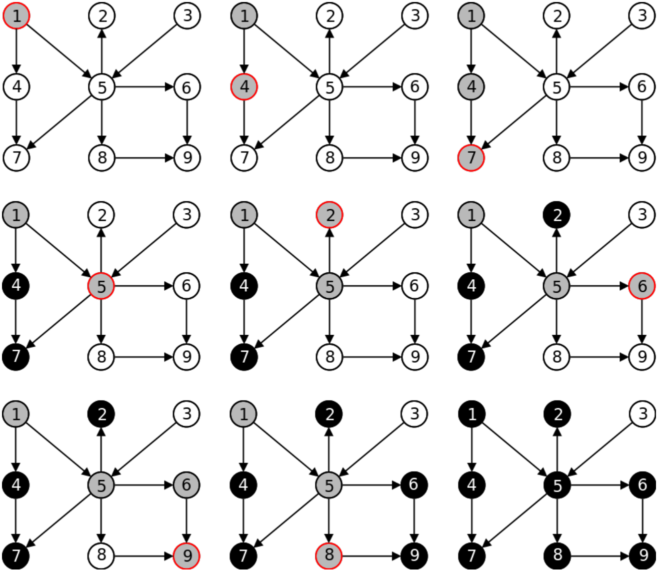
Cycle Detection

Example (starting on node 1) - graph with two cycles



Cycle Detection

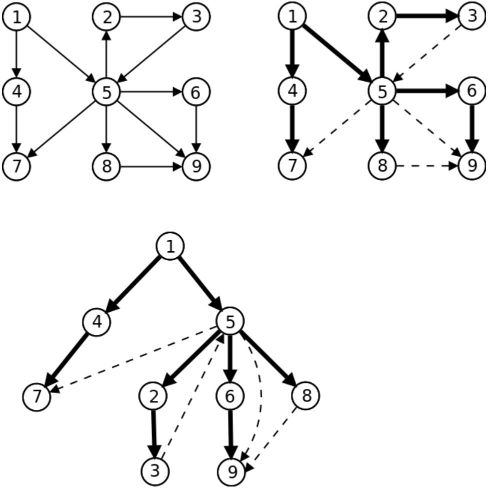
Example (starting on node 1) - acyclic graph



Classifying DFS Edges

Another "angle" of DFS

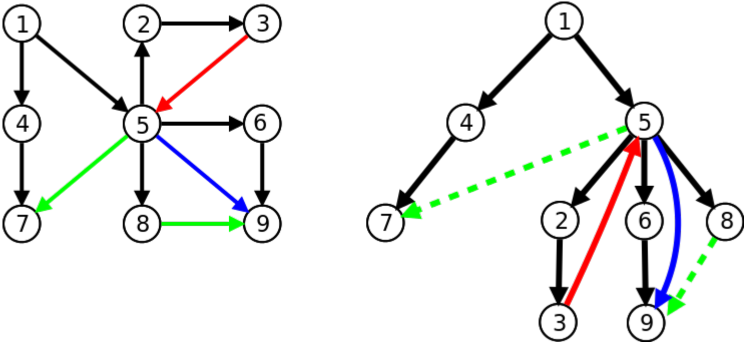
- A DFS implicitly creates a **search tree**, that corresponds to the traversed edges



Classifying DFS Edges

Another "angle" of DFS

- A DFS visit separates the edges into 4 categories
  - **Tree Edges** - Edges from the DFS tree
  - **Back Edges** - Edge from a node to one of its tree ancestors
  - **Forward Edges** - Edge from a node to one of its tree descendants
  - **Cross Edges** - All other edges (from one branch to another)



Classifying DFS Edges

Another ”angle” of DFS

- Example application: finding cycles is finding... **Back Edges**!
- Knowing the edge types may help to solve problem!

- Note: an undirected graph has only **Tree Edges** and **Back Edges**.

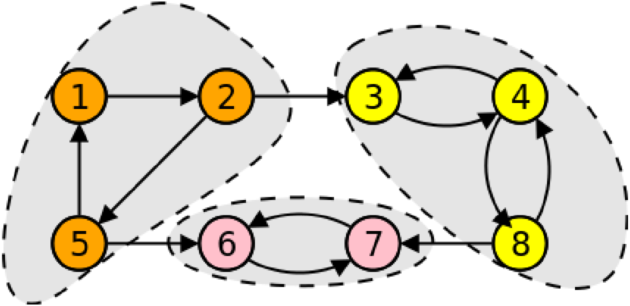
Strongly Connected Components

A more complex DFS application

- Decompose a graph into its **strongly connected component**

A **strongly connected component** (SCC) its a maximal subgraph where there is a (directed) path between each of its nodes.

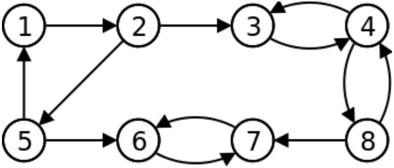
An example graph with 3 SCCs:

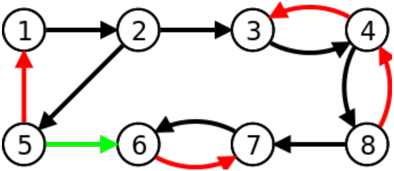


Strongly Connected Components

A more complex DFS application

- How to compute SCCs?
- Let's try to use our knowledge about DFS edge types:

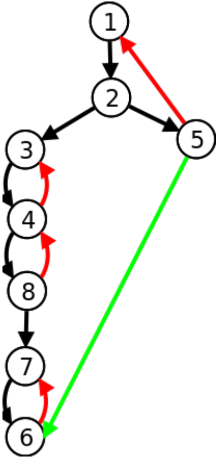




Strongly Connected Components

A more complex DFS application

- Let's look at the generated tree:

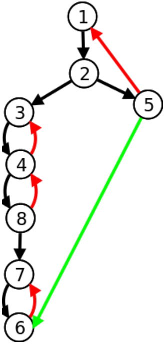


- What is the "lowest" ancestor reachable by a node?
    - 1: it's 1
    - 2: it's 1
    - 5: it's 1
    - 3: it's 3
    - 4: it's 3
    - 8: it's 3
    - 7: it's 7
    - 6: it's 7
- *Et voilà!* Here are our SCCs!

Strongly Connected Components

A more complex DFS application

- Let's add 2 attributes to the nodes in a DFS visit:
  - **num(i)**: order in which *i* is visited
  - **low(i)**: smallest $num(i)$ reachable by the 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 complex DFS application

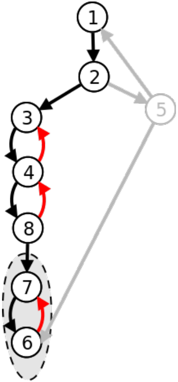
Main ideas of **Tarjan Algorithm** to find SCCs:

- Make a **DFS** and in each node *i*:
    - Keep pushing the nodes to a **stack S**
    - Compute and store the values of **num**(**i**) and **low**(**i**).
    - If when finishing the visit of a node *i* we have that **num**(**i**) = **low**(**i**), then *i* is the "root" of a SCC. In that case, remove all the elements in the stack until reaching *i* and report those elements as belonging to a SCC!

Strongly Connected Components

A more complex DFS application

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



State of Stack **S**:

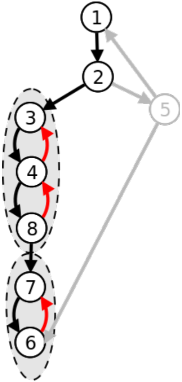
6
7
8
4
3
2
1

Remove elements from stack until reaching **7**; output SCC: {6, 7}

Strongly Connected Components

A more complex DFS application

Example of execution: in the moment we leave *dfs*(3), we find that *num*(3) = *low*(3) (3 is the "root" of a SCC)



State of Stack **S**:

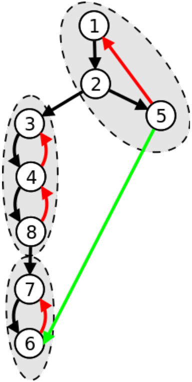
8
4
3
2
1

Remove elements from stack until reaching **3**; output SCC: `{8, 4, 3}`

Strongly Connected Components

A more complex DFS application

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



State of Stack **S**:

5
2
1

Remove elements from stack until reaching **1**; output SCC:: `{5, 2, 1}`

Strongly Connected Components

Tarjan Algorithm for SCCs

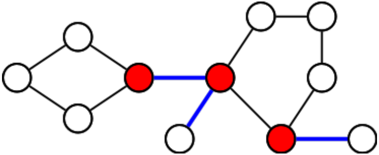
```
index ← 0 ; $S \leftarrow \emptyset$
For all nodes $v$ of the graph do
  If $num[v]$ is still undefined then
    dfs_scc($v$)

dfs_scc(node $v$):
  $num[v] \leftarrow low[v] \leftarrow index$ ; $index \leftarrow index + 1$ ; S.push($v$)
  /* Traverse edges of $v$ */
  For all neighbors $w$ of $v$ do
    If $num[w]$ is still undefined then /* Tree Edge */
      dfs_scc($w$) ; $low[v] \leftarrow min(low[v], low[w])$
    Else if $w$ is in $S$ then  /* Back Edge */
      $low[v] \leftarrow min(low[v], num[w])$
  /* We know that we are at the root of an SCC */
  If $num[v] = low[v]$ then
    Start new SCC $C$
    Repeat
      $w \leftarrow$ S.pop() ; Add $w$ to $C$
    Until $w = v$
    Write $C$
```

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 said to be **biconnected**.

Articulation Points

A more complex DFS application

- Finding **articulation points** is a very useful problem
  - For instance, a "robust" graph should not have articulation points that when "attacked" will disconnect them.

- How to compute? A possible (naive) **algorithm**:
  1. Make a DFS and count the number of connected components
  2. Remove a node from the original graph and execute a new DFS, counting again the connnected components. If this number increased, them the node is an articulation point.
  3. Repeat step 2 for all nodes in the graph

- What would be the **complexity** of this method? $\mathcal{O}(|V|(|V|+|E|))$, because we will make $|V|$ calls to DFS, each one taking $|V|+|E|$.

- **It is possible to do much better... using a single DFS**!

Articulation Points

A more complex DFS application

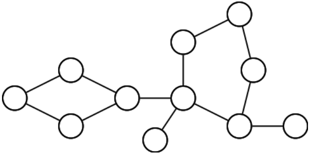
An idea:

- Apply DFS to the graph and obtain the **DFS tree**
- If a **node** v **has a child** w **without any path to an ancestor of** v**, then** v **is an articulation point**! (since removing it would disconnect w from the resto of the graph)
  - This corresponds to verify if $low[w] \geq num[v]$
- The only exception is the **root** of the DFS tree. If it has more than one child in the tree... it is also an articulation point!

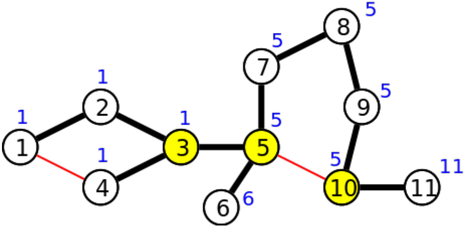
Articulation Points

A more complex DFS application

- An example graph:

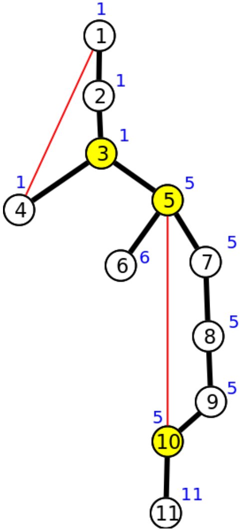


- *num*[*i*] - numbers inside the node
- *low*[*i*] - blue numbers
- articulation points: yellow nodes



Articulation Points

A more complex DFS application



- 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 a tree edge

Articulation Points

Algorithm very similar to Tarjan, but with different DFS:

Algorithm to find articulation points

```
dfs_art(node $v$):
  $num[v] \leftarrow low[v] \leftarrow index$ ; $index \leftarrow index + 1$ ; S.push($v$)
  For all neighbors $w$ of $v$ do
    If $num[w]$ 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 have used colors (gray means it is in the stack)

Breadth-First Search - BFS

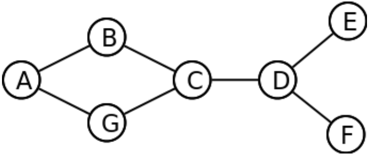
- A breadth-first search (BFS) is very similar to a DFS. It only changes the order in which the nodes are visited!
- Instead of using recursion, we will keep explicitly a queue of not visited nodes (q)

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

```
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 still unprocessed nodes */
        $u \leftarrow q.dequeue()$ /* remove first element of $q$ */
        For all neighbors $w$ of $u$ do
            If $w$ has not yet been visited then /* new node! */
                    $q.enqueue(w)$
                    mark $w$ as visited
```

Breadth-First Search - BFS

- An example:



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

Breadth-First Search - BFS

Computing distances

- Almost everything than can be done with DFS can also be done with BFS!
- An important difference is that with BFS we visit the nodes in increasing order of distance (in terms of number of edges) to the initial node!
- In this way, BFS an be used to compute **shortest distances** between nodes on a **unweighted graph** (with ot without direction).
- Let's see what really changes in the code

Breadth-First Search - BFS

Computing distances

- In red the lines that were added. Em *node.distance* stores the distance to node v.

BFS - Computing distances

```
bfs(node $v$):
  $q \leftarrow \emptyset$ /* Queue of non visited nodes */
  $q.enqueue(v)$
  $v.distance \leftarrow 0$ /* distance from $v$ to itself it's zero */
  mark $v$ as visited
  While $q \neq \emptyset$ /* while there are still unprocessed nodes */
    $u \leftarrow q.dequeue()$ /* remove first element of $q$ */
    For all neighbors $w$ of $u$ do
      If $w$ has not yet been 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 in any graph type
- Consider for instance that you want to know the **minimum distance between** points **A** and **B** on a 2D maze:

```
########                            ########
#A.....#                            #A12345#
####.###          --->              ####4###
#B.....#   BFS starting in A        #876567#
########                            ########
```

  - A node is a cell (x, y)
  - Neighbors are $(x+1, y)$, $(x-1, y)$, $(x, y+1)$ e $(x, y-1)$
  - Everything ele in the BFS is the same! (time: $\mathcal{O}(rows \times cols)$)
  - To store on the queue we need to represent a coordinates pair (e.g.: struct in C, pair or class in C++, class in Java).