



Graph visits

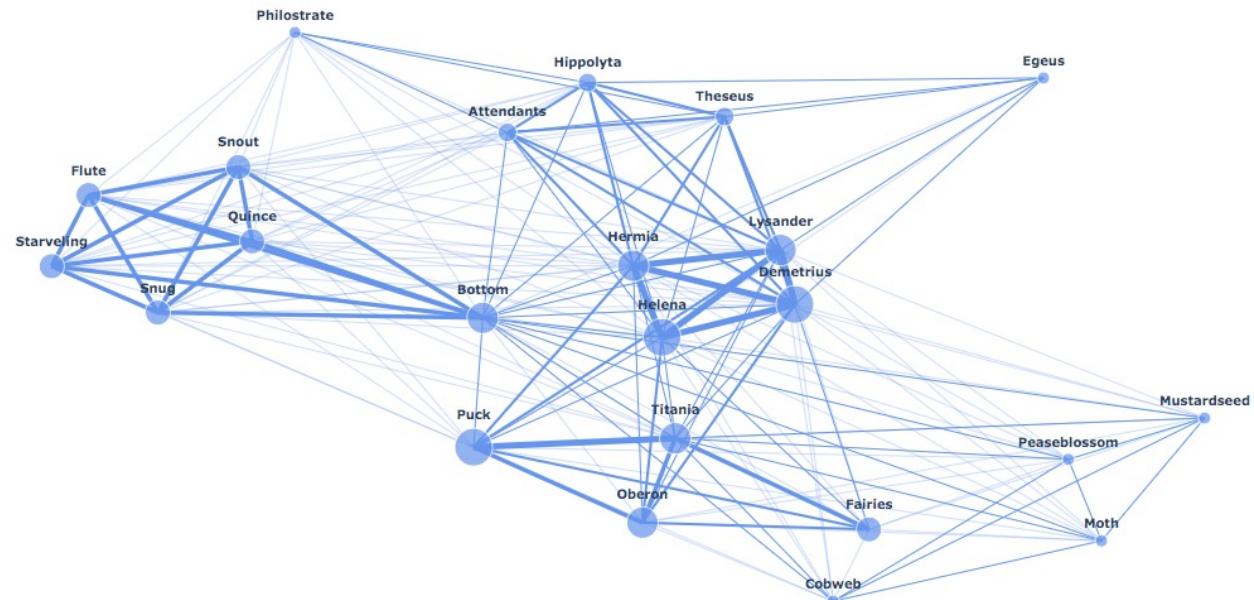
How to explore graphs

Fulvio Corno

Giuseppe Averta

Carlo Masone

Francesca Pistilli





GRAPH VISITS

Representing and visiting graphs

Visit Algorithms

- Visit =
 - Systematic exploration of a graph
 - Starting from a ‘source’ vertex
 - Reaching all reachable vertices
- Main strategies
 - Breadth-first visit (“in ampiezza”)
 - Depth-first visit (“in profondità”)

Quando diciamo che visitiamo un grafo intendiamo un algoritmo in grado di esplorare in maniera sistematica questa struttura dati. L’idea è: noi partiamo da un nodo **source**, e andiamo a controllare quali nodi del grafo sono raggiungibili dal nodo source, assumendo di avere degli archi che potranno essere orientati oppure no.

Per fare questa cosa abbiamo due principali strategie:

- posso andare a cercare i nodi successivi uno dopo l’altro: prendo un arco dal nodo source che mi porterà al secondo nodo, e poi continuerò ad andare in profondità.
- posso visitare in ampiezza: inizio considerando tutti i nodi vicini al nodo source, e così via per ogni nodo, considero sempre prima i nodi vicini piuttosto che andare in profondità.

Breadth-First Visit

- Also called Breadth-first search (BFV or BFS)
- All reachable vertices are visited “by levels”
 - L – level of the visit
 - S_L – set of vertices in level L
 - $L=0, S_0=\{ v_{\text{source}} \}$
 - Repeat while S_L is not empty:
 - $S_{L+1} = \text{set of all vertices:}$
 - not visited yet, and
 - adjacent to at least one vertex in S_L
 - $L=L+1$

Breadth-First è una strategia di ricerca in cui visitiamo tutti i nodi a livelli:

• partiamo dal nodo source e visitiamo tutti i vicini; una volta fatto questo, prendiamo uno di questi nodi vicini e di nuovo andiamo a considerare tutti i nodi vicini a questo secondo nodo, e così via aggiungiamo alla nostra lista di nodi visitati e quindi nodi raggiungibili tutti i nodi che noi visitiamo in maniera progressiva livello dopo livello.

S_L : insieme di vertici livello L
 S_0 : nodo "source"

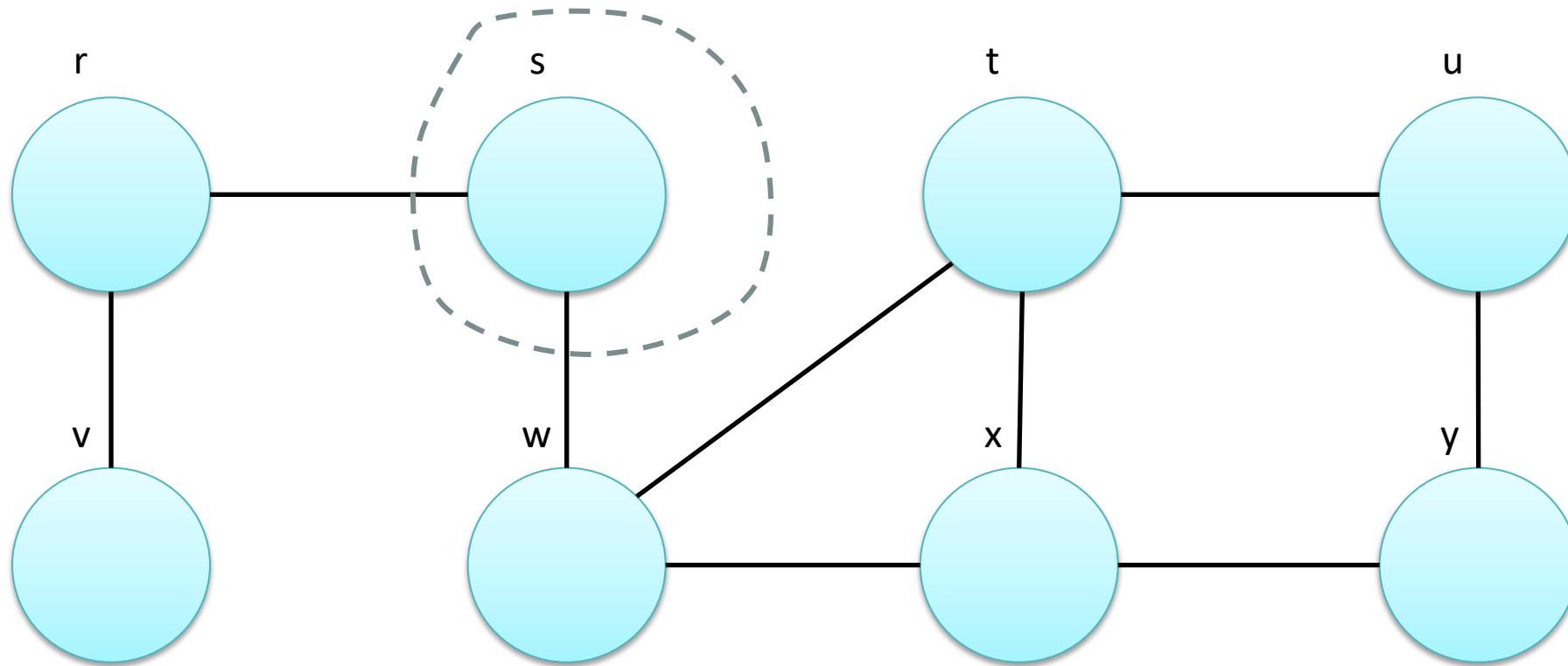
• A ogni iterazione, aumenta L di 1 e visita tutti i nodi vicini ai nodi che ho esplorato all'iterazione precedente

N.B.: Per coprire tutti i nodi, il grafo deve essere **connesso!**

Example

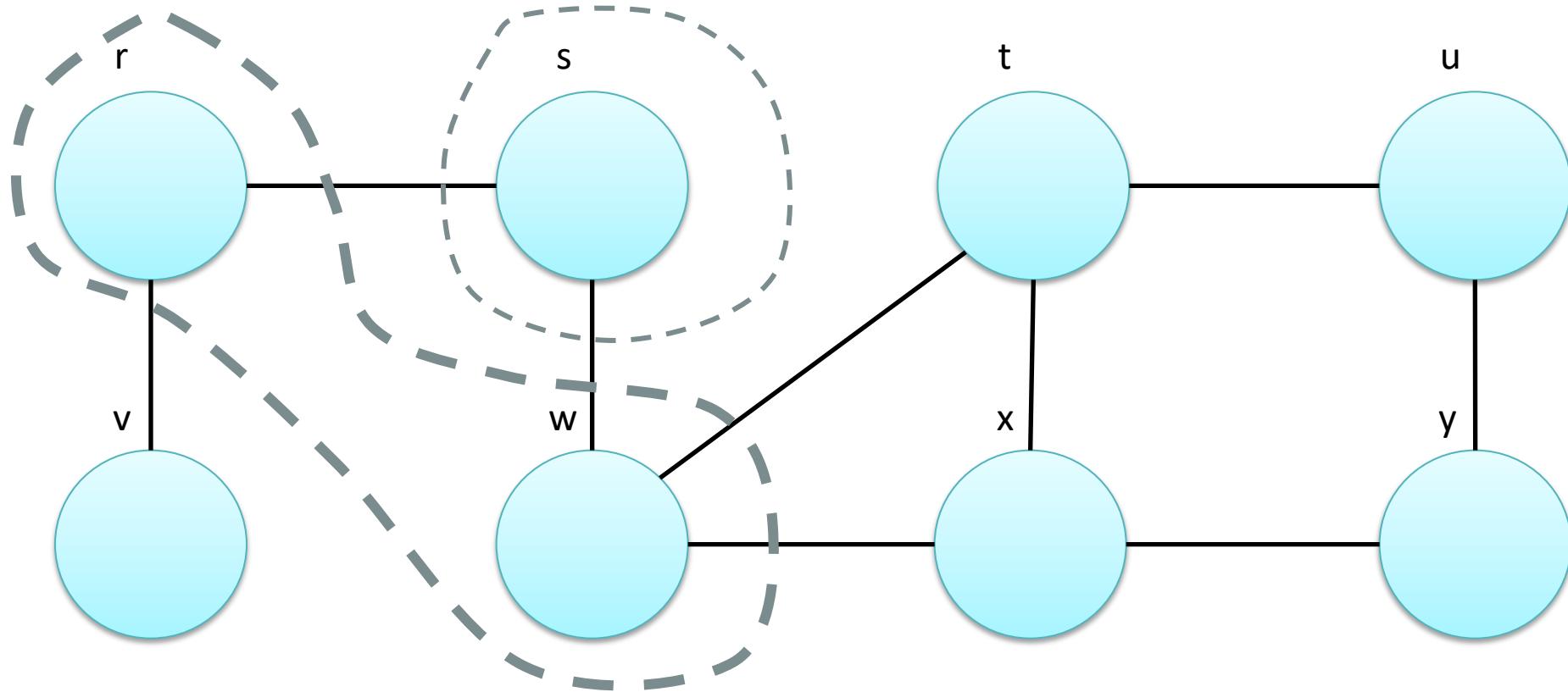
Partendo da **s**, visitiamo prima i nodi vicini quindi **r e w**

Source = s
L = 0
 $S_0 = \{s\}$



Example

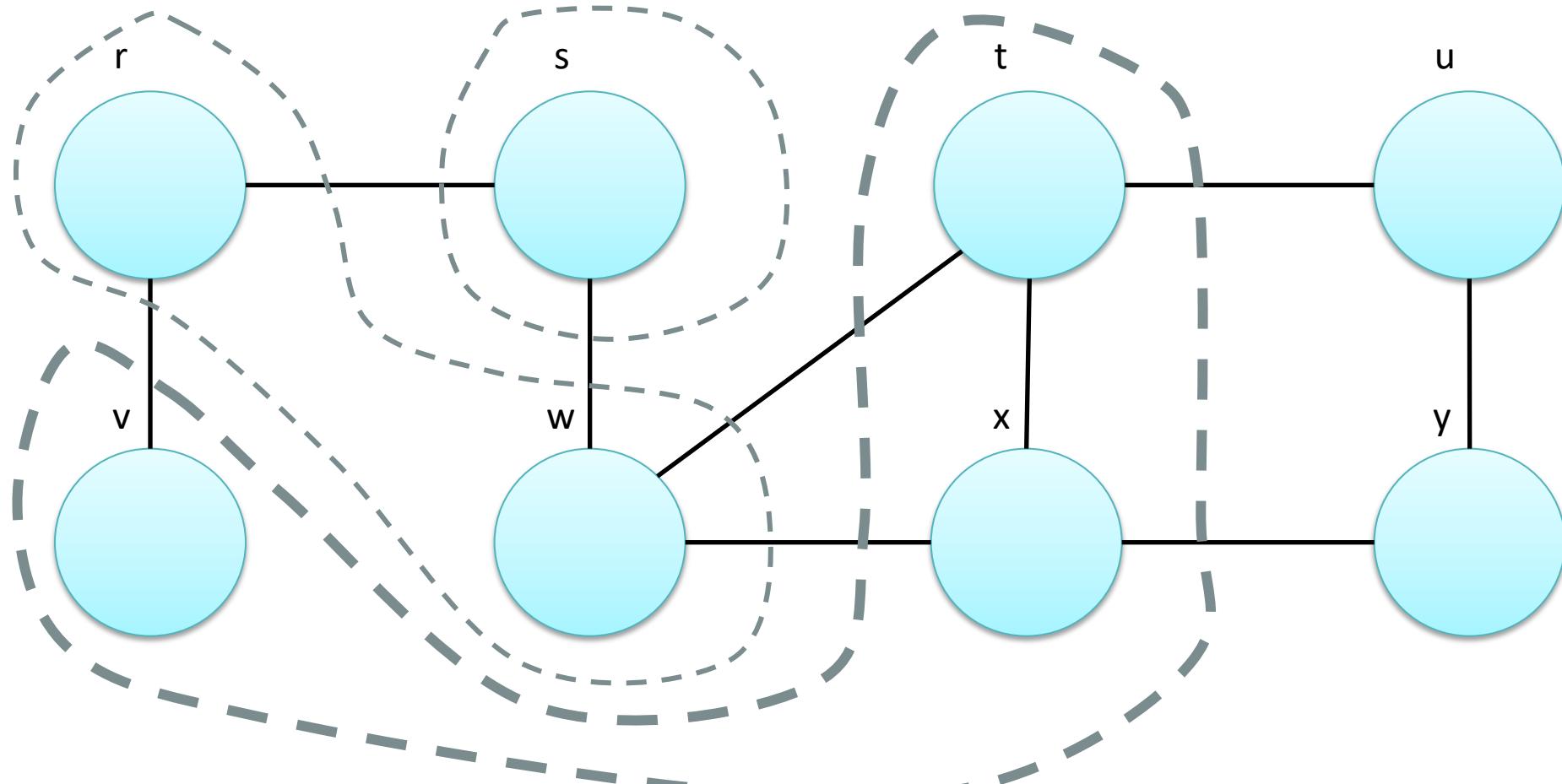
$L = 1$
 $S_0 = \{s\}$
 $S_1 = \{r, w\}$



Example

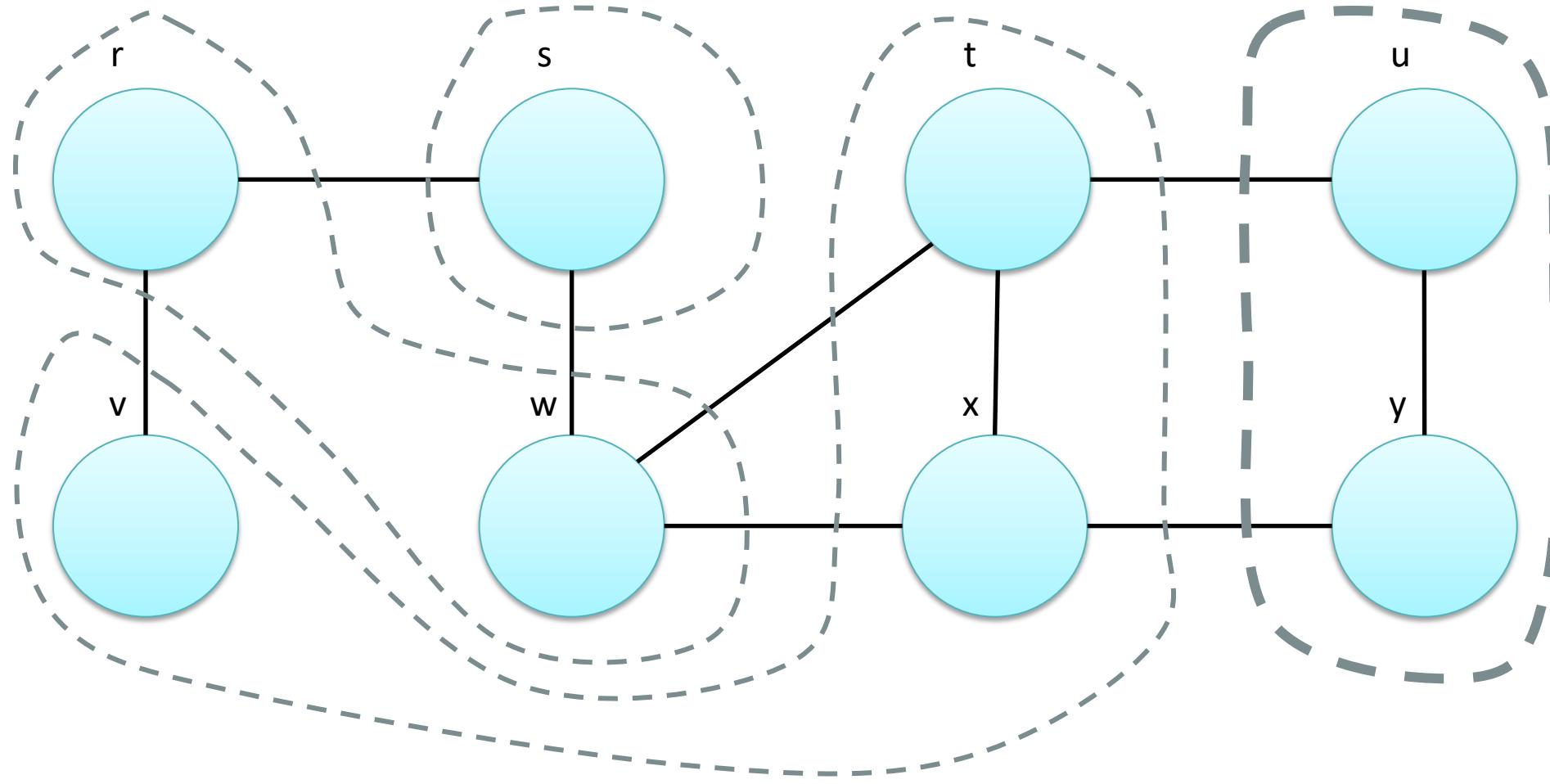
- Al livello 2, visito i nodi vicini ai nodi del livello 1: $S_2 = \{r, w\}$.

$$\begin{aligned}L &= 2 \\S_1 &= \{r, w\} \\S_2 &= \{v, t, x\}\end{aligned}$$



Example

$L = 3$
 $S_2 = \{v, t, x\}$
 $S_3 = \{u, y\}$



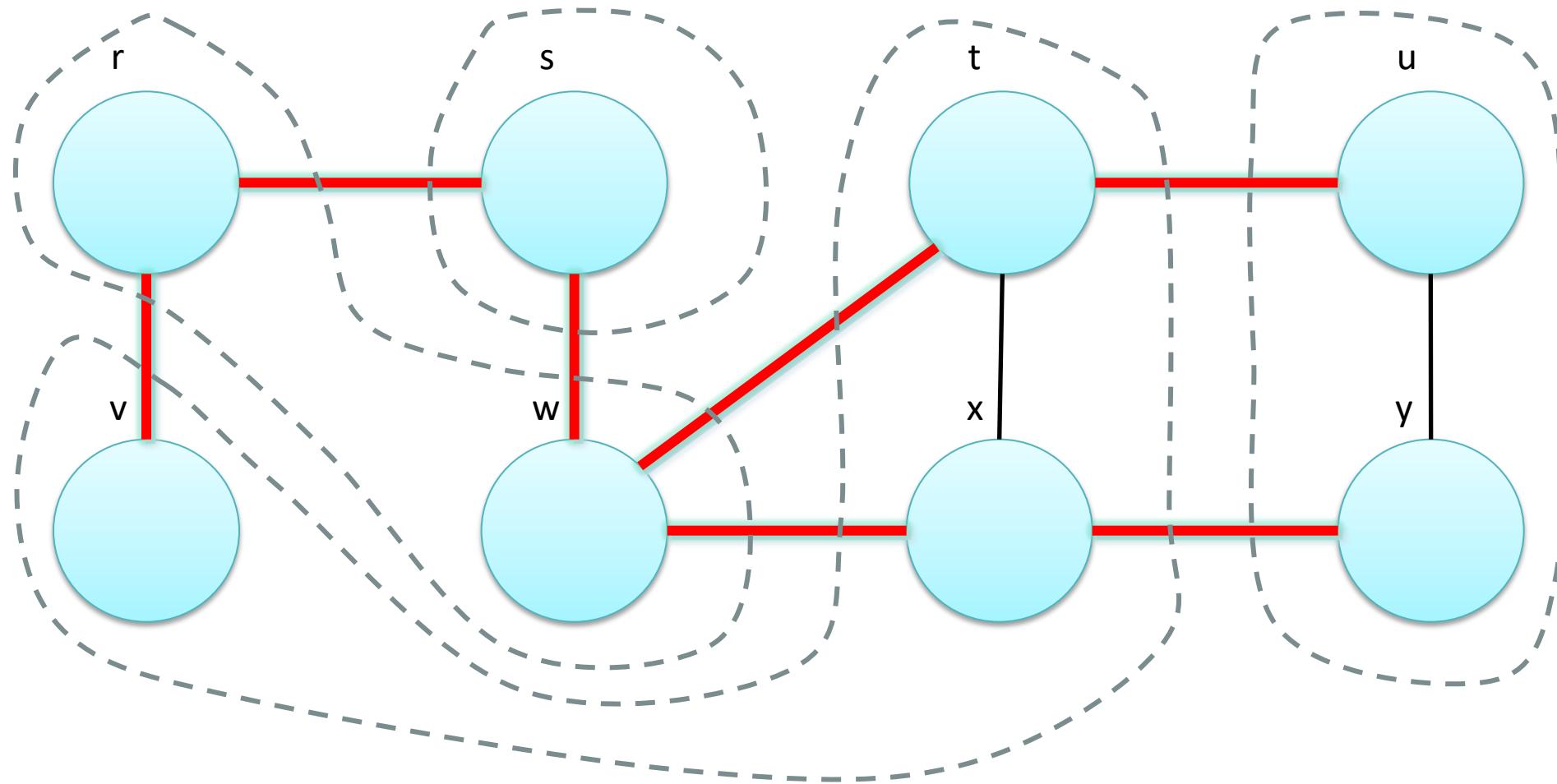
BFS Tree

Quindi l'algoritmo breadth-first quello che ci restituisce di fatto è un albero che a partire dal nodo-source ci porta verso le foglie.
Ovviamente i nodi che saranno inclusi saranno tutti nodi della componente connessa, perché riusciremo a visitarli tutti, ma in generale non abbiamo garanzia di passare verso tutti gli archi, perché se ci sono dei loop nel grafo, noi prenderemo solo un sottoinsieme degli archi del grafo.

- The result of a BFV identifies a “visit tree” in the graph:
 - The tree root is the source vertex
 - Tree nodes are all graph vertices
 - (in the same connected component of the source)
 - Tree are a subset of graph edges
 - Those edges that have been used to “discover” new vertices.

BFS Tree

• Gli archi rossi sono gli edges che ho visitato con l'algoritmo breadth-first!



Minimum (shortest) paths

L'algoritmo breadth-first ha anche un alto vantaggio: perchè l'output di questo algoritmo ci fornisce tutti i cammini minimi dal nodo source verso tutti gli altri nodi; essendo costruito su più livelli, si ha la garanzia che il percorso che ti porta a un nodo nell'albero di visita sia quello più corto, IN NUMERO DI TERMINI DI ARCHI!
Ovviamente non sarà necessariamente il percorso migliore nel caso di archi pesati, perchè non considero il peso degli archi, ma per quanto riguarda il numero di archi attraversati ho la garanzia che sarà sempre il minimo possibile!

- Shortest path: the minimum number of edges on any path between two vertices
- The BFS procedure computes all minimum paths for all vertices, starting from the source vertex
- NB: unweighted graph : path length = number of edges

Depth First Visit

L'algoritmo depth-first fa il contrario: partiamo sempre da un nodo source, scegliamo a caso con una certa euristica un nodo vicino, e aggiungiamo quell'arco; dopo di che ricerchiamo di nuovo i vicini del nodo dove siamo capitati, ne scegliamo uno a caso e lo visitiamo, finché non finiamo il percorso (e quindi abbiamo visitato tutti i nodi, sempre che il grafo sia effettivamente连通的).
Nel momento in cui abbiamo finito il percorso torniamo indietro di uno step e continuamo a visitare il grafo.

- Also called Depth-first search (DFV or DFS)
- Opposite approach to BFS
- At every step, visit one (yet unvisited) vertex, adjacent to the last visited one
- If no such vertex exist, go back one step to the previously visited vertex
- Lends itself to recursive implementation
 - Similar to tree visit procedures

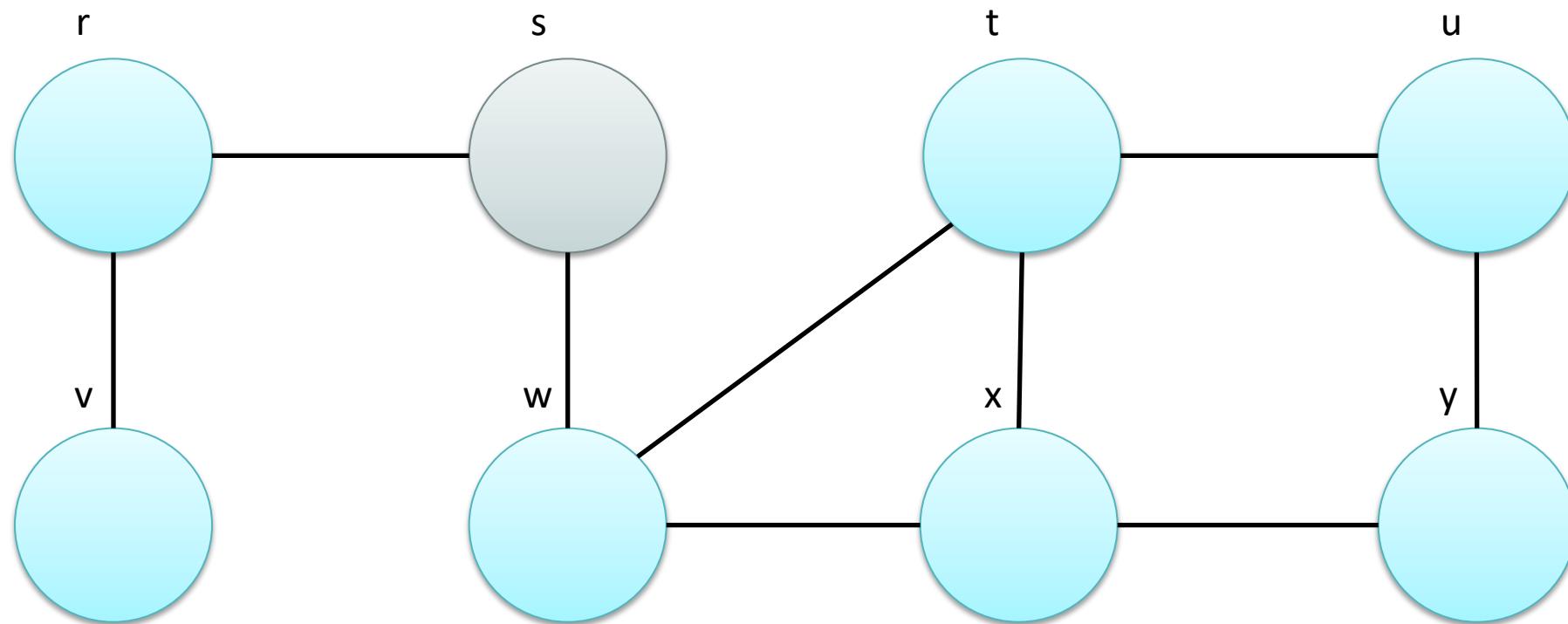
DFS Algorithm

- $\text{DFS}(\text{Vertex } v)$
 - For all ($w : \text{adjacent_to}(v)$)
 - If(not visited (w))
 - Visit (w)
 - $\text{DFS}(w)$
 - Start with: $\text{DFS}(\text{source})$

• Algoritmo Depth-First implementato in pseudocodice:
è come se fosse un metodo ricorsivo!

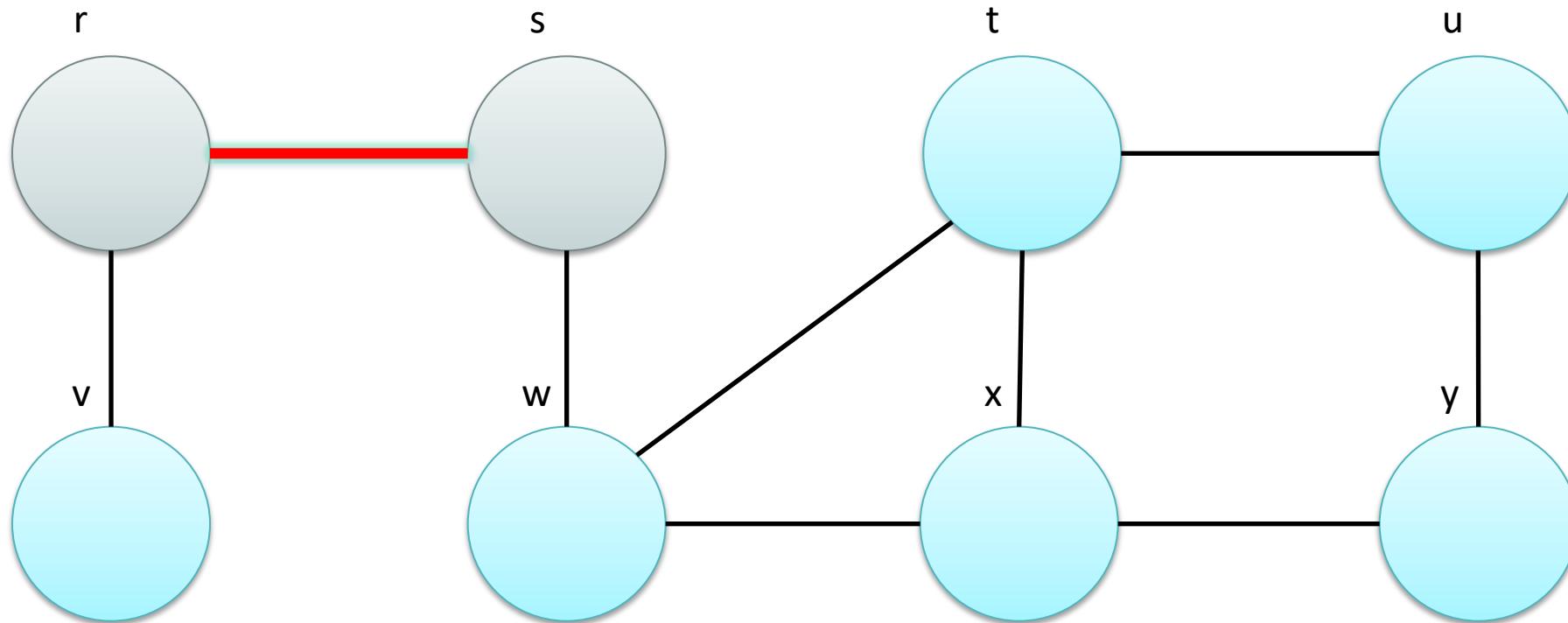
Example

Source = s



Example

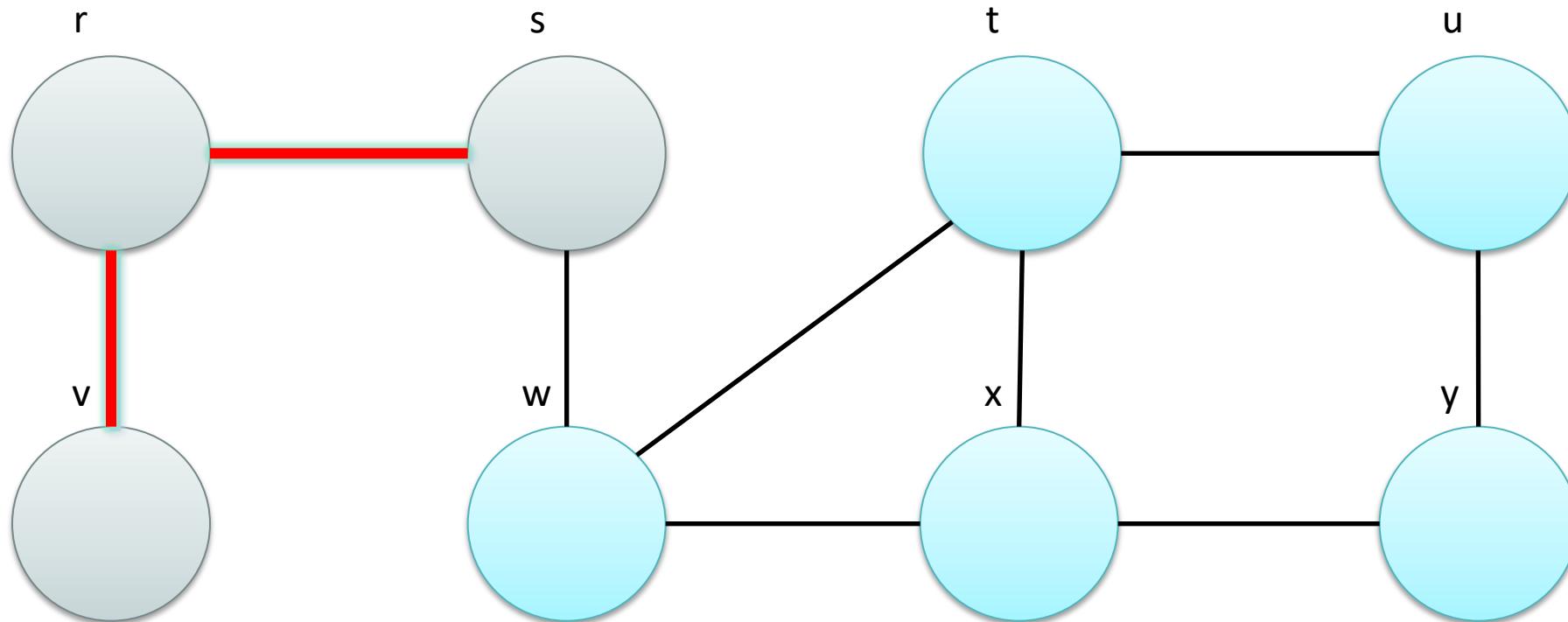
Source = s
Visit r



Example

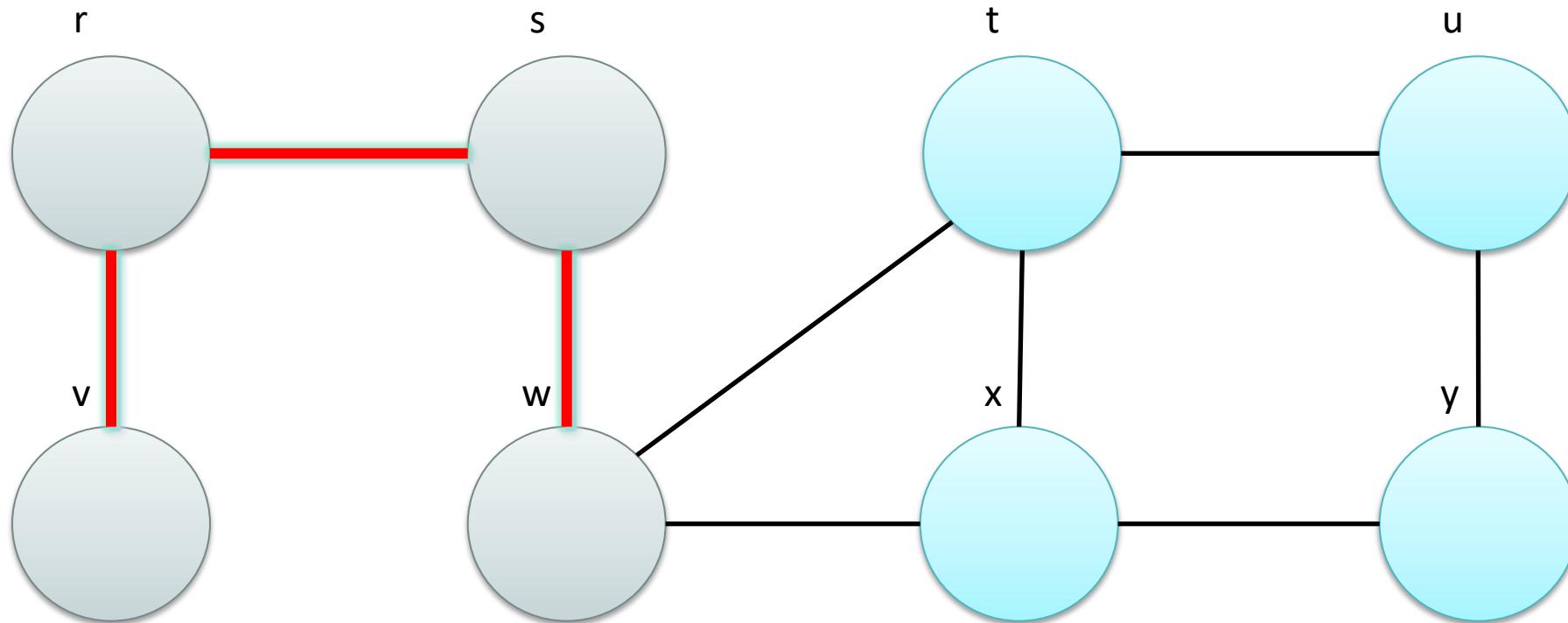
• Dato che **v** NON ha vicini, torno indietro di qualche step fino ad arrivare a un nodo che effettivamente abbia dei nodi vicini.

Source = s
Visit r
Visit v



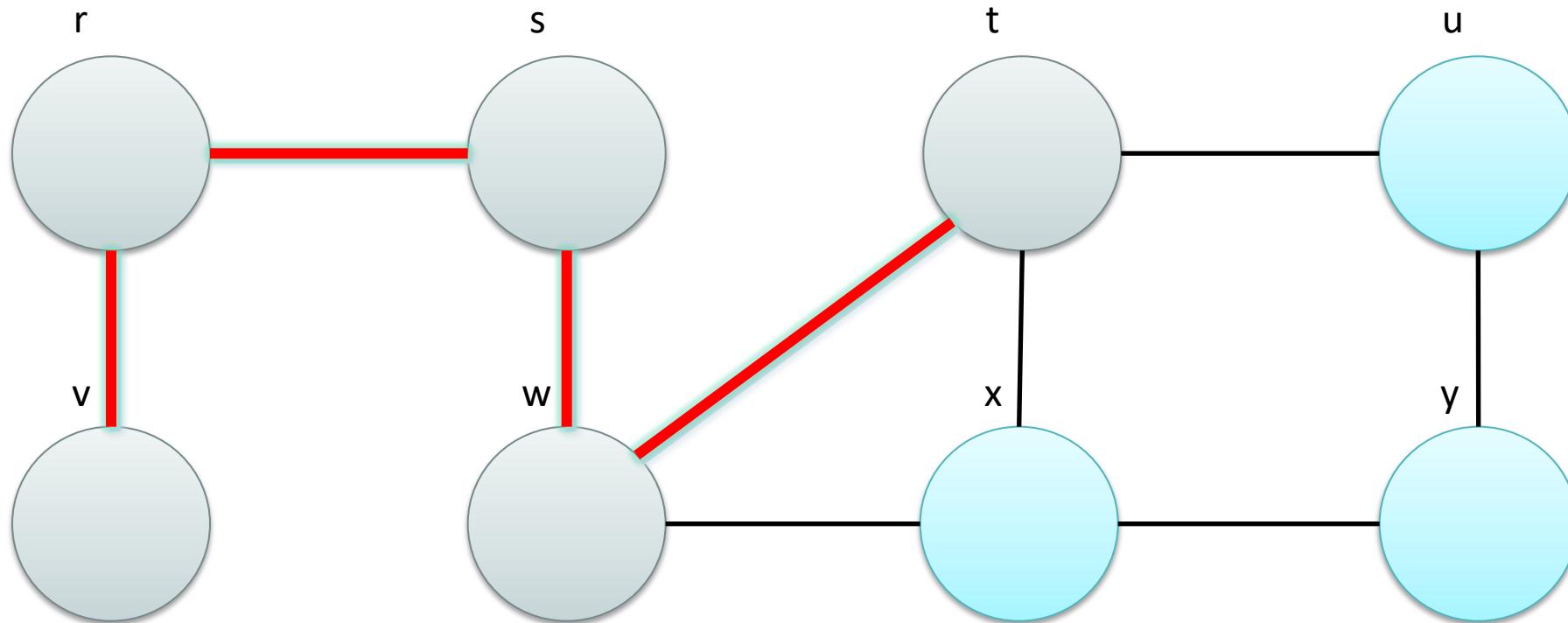
Example

Source = s
Back to r
Back to s
Visit w



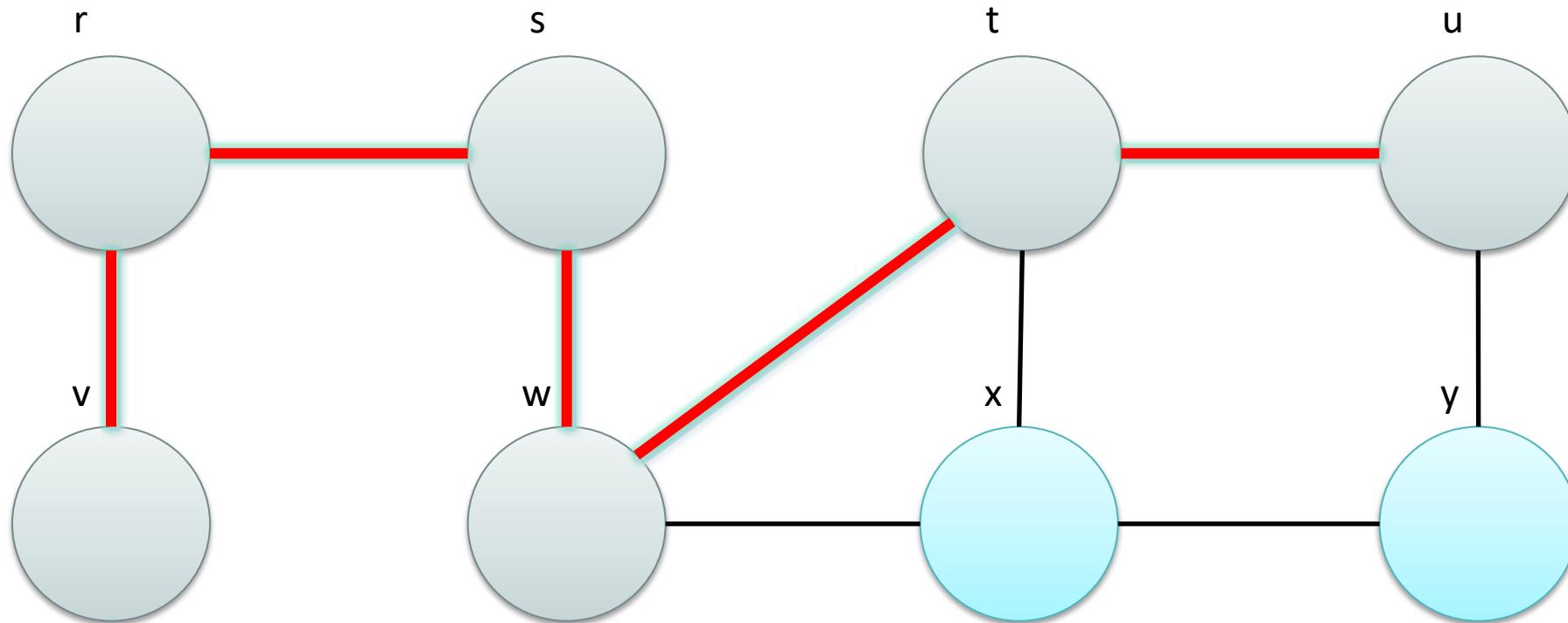
Example

Source = s
Visit w
Visit t



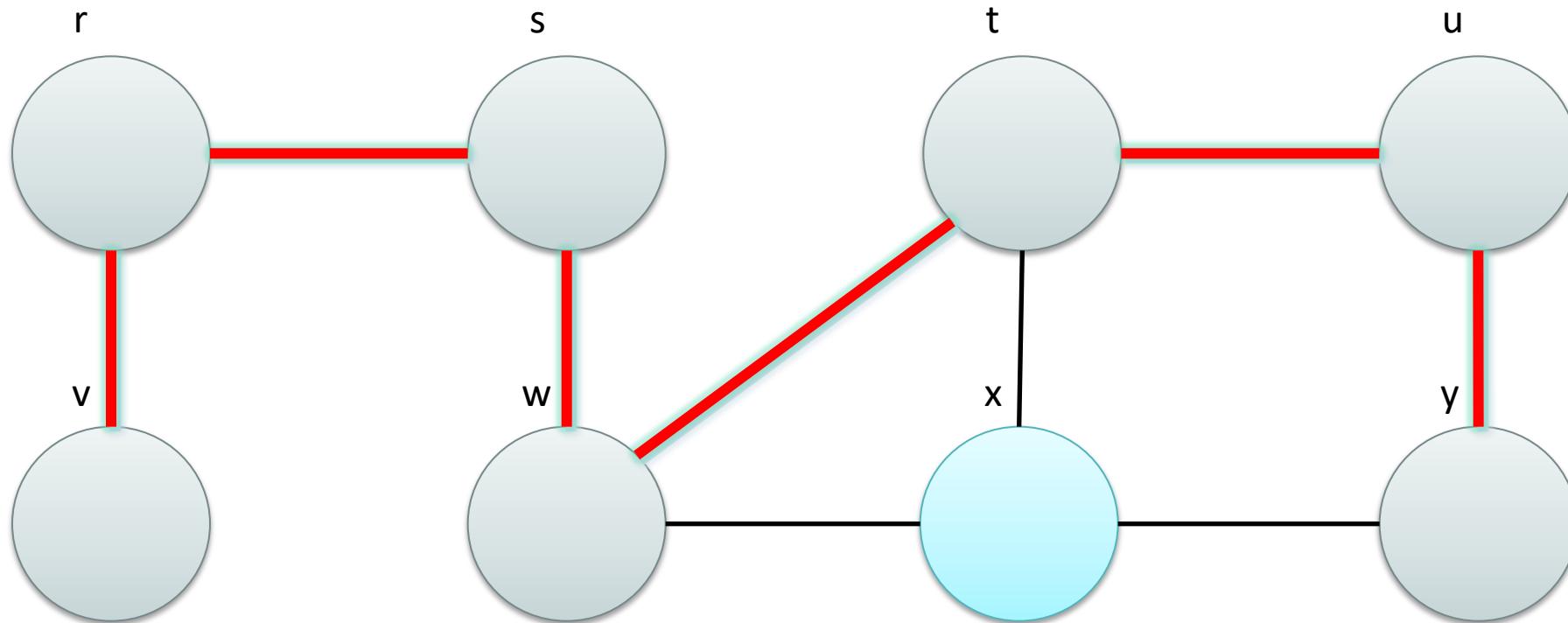
Example

Source = s
Visit w
Visit t
Visit u



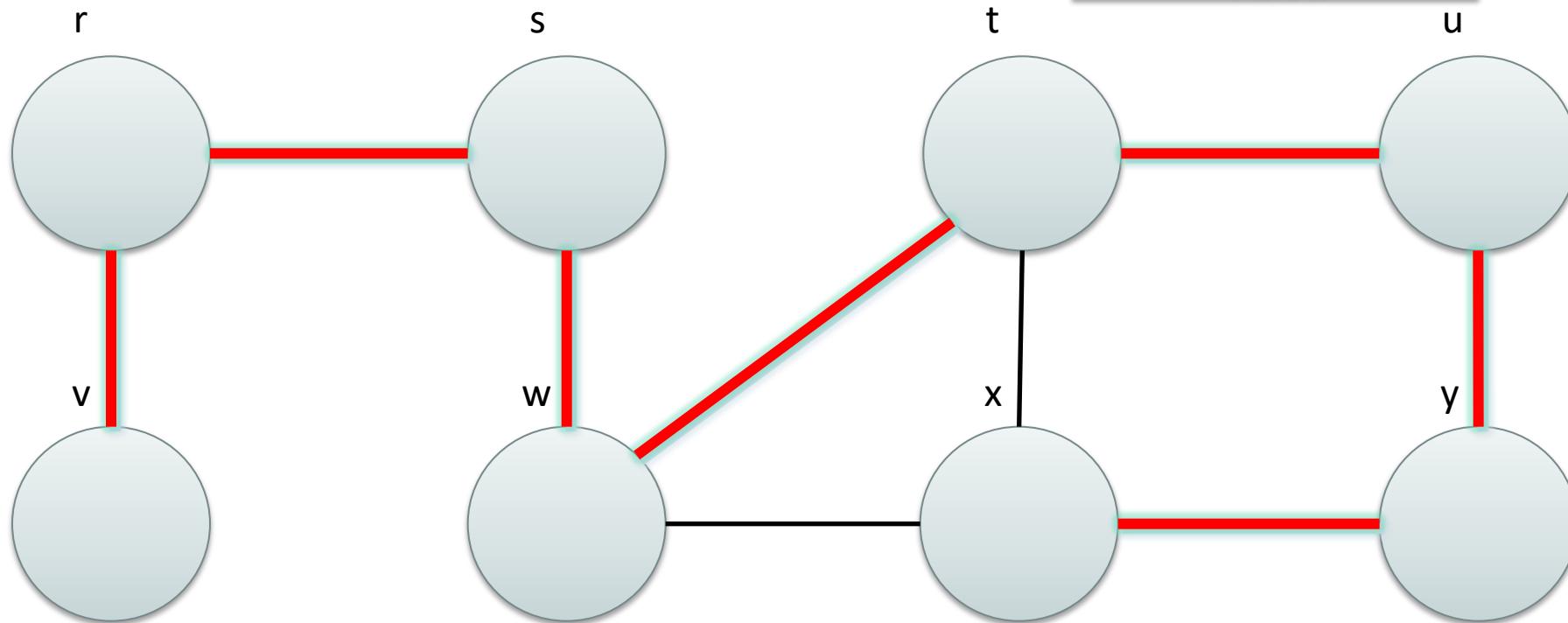
Example

Source = s
Visit w
Visit t
Visit u
Visit y

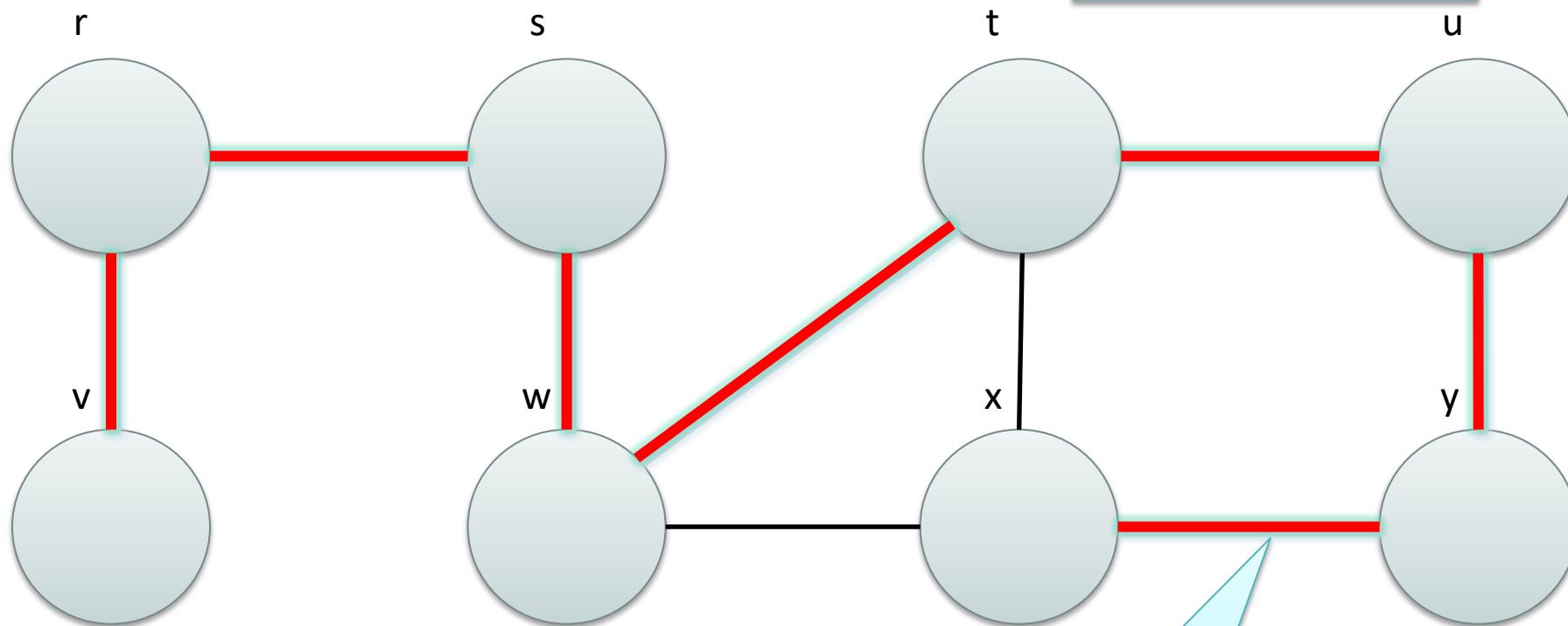


Example

Source = s
Visit w
Visit t
Visit u
Visit y
Visit x



Example



Source = s
Back to y
Back to u
Back to t
Back to w

Back to s = STOP

DFS tree

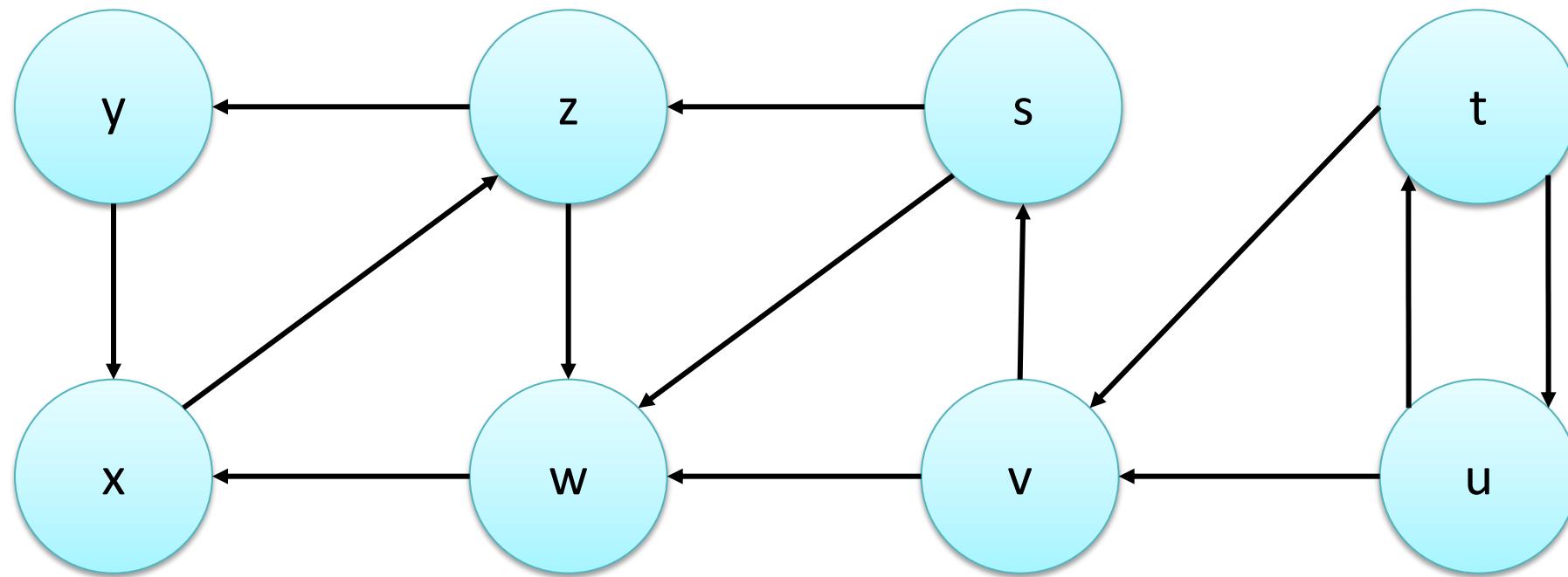
Albero DFS!

N.B:

L'algoritmo DFS non ci garantisce di avere cammini minimi, a differenza dell'algoritmo BFS!

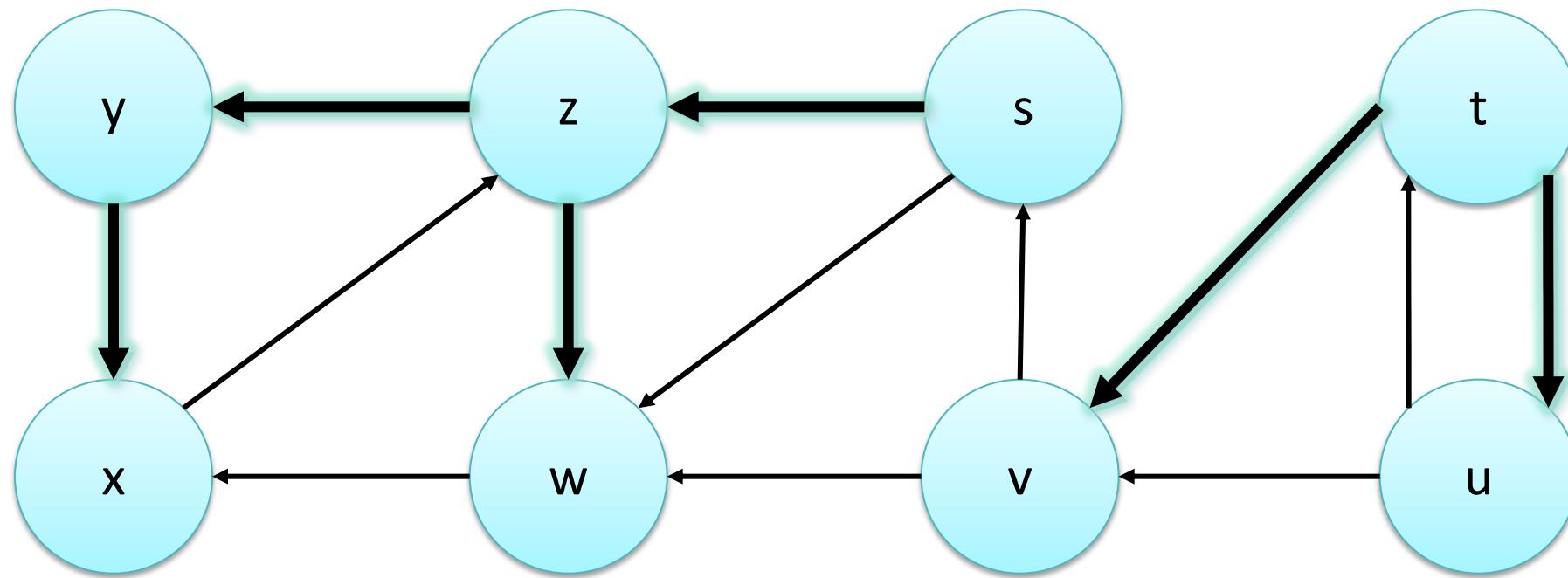
Example

Directed graph



Example

DFS visit
(sources: s, t)



Complexity

- Visits have linear complexity in the graph size
 - BFS : $O(V+E)$
 - DFS : $\Theta(V+E)$
- N.B. for dense graphs, $E = O(V^2)$

Entrambi gli algoritmi hanno una complessità che è abbastanza limitata, che è dell'ordine del numero dei nodi + il numero degli archi, e ovviamente dipende da quanto è denso il grafo (numero di grafi rispetto al numero di archi massimo possibili)

Resources

- Maths Encyclopedia: <http://mathworld.wolfram.com/>
- Basic Graph Theory with Applications to Economics
<http://www.isid.ac.in/~dmishra/mpdoc/lecgraph.pdf>
- Application of Graph Theory in real world
- <http://prezi.com/tseh1wvpves-/application-of-graph-theory-in-real-world/>



Representing and visiting graphs

VISITS IN NETWORKX

Traversal

- Visits are called “traversals”
- NetworkX already provides implementations for BFV and DFV, together with other visits strategies

NetworkX ci implementa già molti metodi!

dfs/bfs-tree: restituisce un grafo che ha gli stessi nodi del grafo di partenza ma come archi solo l'albero di visita risultante dall'algoritmo rispettivo!

Graph traversal methods

Traversal

Depth First Search

Basic algorithms for depth-first searching the nodes of a graph.

`dfs_edges(G[, source, depth_limit, ...])`

Restituisce sotto forma di dizionario il risultato di una visita di tipo DFS del grafo.

Iterate over edges in a depth-first-search (DFS).

`dfs_tree(G[, source, depth_limit, ...])`

Returns oriented tree constructed from a depth-first-search from source.

`dfs_predecessors(G[, source, depth_limit, ...])`

Returns dictionary of predecessors in depth-first-search from source.

`dfs_successors(G[, source, depth_limit, ...])`

Returns dictionary of successors in depth-first-search from source.

`dfs_preorder_nodes(G[, source, depth_limit, ...])`

Generate nodes in a depth-first-search pre-ordering starting at source.

`dfs_postorder_nodes(G[, source, ...])`

Generate nodes in a depth-first-search post-ordering starting at source.

`dfs_labeled_edges(G[, source, depth_limit, ...])`

Iterate over edges in a depth-first-search (DFS) labeled by type.

Breadth First Search

Basic algorithms for breadth-first searching the nodes of a graph.

`bfs_edges(G, source[, reverse, depth_limit, ...])`

Iterate over edges in a breadth-first-search starting at source.

`bfs_layers(G, sources)`

Returns an iterator of all the layers in breadth-first search traversal.

`bfs_tree(G, source[, reverse, depth_limit, ...])`

Returns an oriented tree constructed from a breadth-first-search starting at source.

`bfs_predecessors(G, source[, depth_limit, ...])`

Returns an iterator of predecessors in breadth-first-search from source.

`bfs_successors(G, source[, depth_limit, ...])`

Returns an iterator of successors in breadth-first-search from source.

`descendants_at_distance(G, source, distance)`

Returns all nodes at a fixed `distance` from `source` in `G`.

`generic_bfs_edges(G, source[, neighbors, ...])`

Iterate over edges in a breadth-first search.

Example



License

- These slides are distributed under a Creative Commons license “Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)”
- You are free to:
 - Share — copy and redistribute the material in any medium or format
 - Adapt — remix, transform, and build upon the material
 - The licensor cannot revoke these freedoms as long as you follow the license terms.
- Under the following terms:
 - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
 - NonCommercial — You may not use the material for commercial purposes.
 - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
 - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- <https://creativecommons.org/licenses/by-nc-sa/4.0/>