Unweighted graph algorithms

Serge Kruk

October 11, 2022

Warnings

We will go fast!

These algorithms are rather simple, yet useful and classical. You must know them.

- Know how to run them by hand
- Know when to use them
- Know what data structure should be used

Warnings

Code is (almost) all FAKE!

Contrary to all the code I presented up to now, the code of this section is all pseudo-code. The reasons for this will become clear as we progress. (So my usual "Code this to understand deeply" is lifted, temporarily.)

Graph

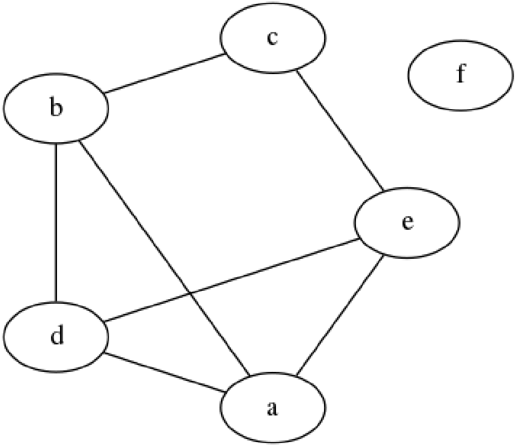
Recall that a graph is a tuple $G = (V, E)$ where

- V is the set of vertices (anything, really)
- E is the set of edges, couples of vertices.
- Vertices are also known as nodes.
- Edges are known as arcs when they are directed.

We sometimes write $V(G)$ $(E(G))$ to indicate the vertices (edges) of graph G when more than one graph is under consideration.

Graphical representation

$$G = (\{a, b, c, d, e, f\}, \{(a, b), (a, d), (a, e), (b, c), (b, d), (c, e), (d, e)\})$$



Vocabulary

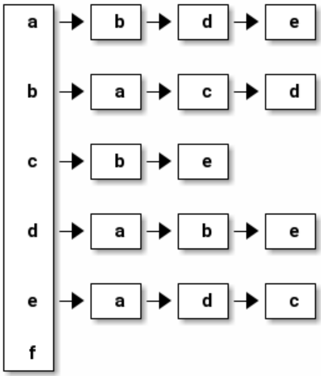
Given the previous graph

- Node a is adjacent to nodes b, d, e.
- Edge (a, b) is incident to node a and node b.
- The neibourhood of node a is the set of nodes $\{b, d, e\}$.
- The degree of node a is 3, of node c is 2 and of node f is 0.

Possible implementations

- Linked list
- Adjacency matrix
- Incidence matrix

Graph using linked lists



Discuss: What would you use to implement the leftmost structure?

Simple-minded implementation

- Use a hash table indexed by nodes
- Each entry points to a list of neighbours
- Need to implement functions to return
  - All nodes
  - All edges
  - Neighbours of a given node

Simple-minded implementation

- Initialize an empty graph
- Add nodes (without neighbours)
- Add neighbours of a node
- What is the degree of a node?
- Is this an edge?

```python
def new_graph():
  return {}
def add_nodes(G,nodes):
  for node in nodes:
    G[node]=set()
  return G
def add_neighbours(G,node,neighbours):
  for v in neighbours:
    G[node].add(v)
```

Example

```python
G = new_graph()
add_nodes(G,[0,1,2,3])
print(G)
add_neighbours(G,1,[2,3])
print(G)
add_neighbours(G,2,[3])
print(G)
```

```
{0: set(), 1: set(), 2: set(), 3: set()}
{0: set(), 1: {2, 3}, 2: set(), 3: set()}
{0: set(), 1: {2, 3}, 2: {3}, 3: set()}
```

Needed functions

- Get me all nodes
- Get me the neighbours of a node
- Get me all edges in the graph

```
def neighbours(G,v):
  return G[v]                 # O(1)
def nodes(G):
  return list(G)              # O(|V|)
def edges(G):                 # O(|E|)
  all=[]
  for u in nodes(G):
    for v in neighbours(G,u):
      all.append((u,v))
  return all
```

Example

```
print(nodes(G))
print(neighbours(G,1))
print(edges(G))
```

```
[0, 1, 2, 3]
{2, 3}
[(1, 2), (1, 3), (2, 3)]
```

Implementation as adjacency matrix

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Characterization:

- Both rows and columns are indexed by vertices.
- 1 at position (i, j) iff there is an edge between vertices i and j.
- Matrix is symmetric if edges are undirected.
- Compute $A^2, A^3, \ldots$. They contain interesting combinatorial properties.

Implementation as node-arc incidence matrix

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Characterization:

- Rows are indexed by vertices.
- Columns are indexed by edges (or arcs).
- If the graph is undirected, columns have entries 1 on exactly two rows: the vertices of that edge.
- If the graph is directed, columns have -1 (arc tail) and $= 1$ (arc head).

Consequences of implementation on efficiency

Assume $G(V, E)$ and dense matrix structure.

	LL	AM	IM	Hash
IsEdge(x,y)	$O(\|V\|)$	$O(1)$	$O(\|E\|)$	?
Degree(x)	$O(\|V\|)$	$O(\|V\|)$	$O(\|E\|)$	$O(1)$
Neighbours(x)	$O(\|V\|)$	$O(\|V\|)$	$O(\|E\|)$	$O(1)$
Memory use	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$	$O(\|E\| \cdot \|V\|)$	?

Data structure review: Queue

Two operations:

- enqueue (add an element to the queue)
- dequeue (extract oldest element of the queue)

Can we do these in $O(1)$?

Implementation of a Queue

Simple-minded implementation with no value other than pedagogical

```
def new_queue():
  return []
def empty_queue(q):
  return len(q)==0
def enqueue(q,e):
  return q.append(e)
def dequeue(q):
  return q.pop(0)
```

Simple-minded tests

```
>>> q=[]
>>> enqueue(q,101)
>>> enqueue(q,104)
>>> enqueue(q,204)
>>> q
[101, 104, 204]
>>> dequeue(q)
101
>>> q
[104, 204]
>>>
```

First algorithms: systematically visiting every node.

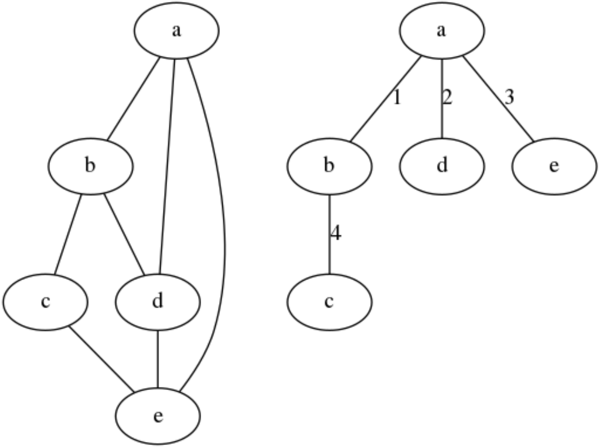
- Breadth First Search
- Depth First Search

Breadth First Search (BFS)

- Pick/obtain a node (known as the root).
- Visit its neighbours (in some order).
- In the order you visited the neighbours, recurse (or iterate).

BFS(G,a)

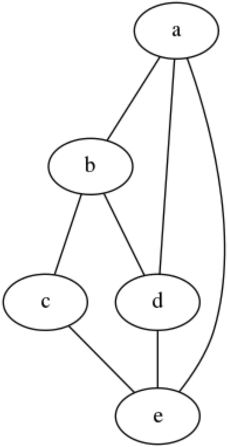
- Order of the visit: a,b,d,e,c
- Note the implicit spanning tree

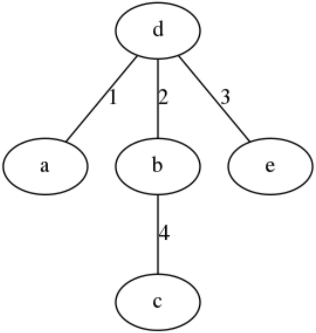


Illustration

BFS(G,d)

- Order of the visit: d,a,b,e,c
- Note the different spanning tree





Implement a BFS

```python
def bfs(G,r):
    # G is a graph.  r is a node of this graph.
```

Implement a BFS

```
def bfs(G,r,visited=None,process=None):
    # G is a graph.  r is a node of this graph.
    q=new_queue()
    if visited is None:
        visited = [False]*len(nodes(G))
    enqueue(q,r)
    visited[r] = True
    while not empty_queue(q):
        v = dequeue(q)
        if process is not None:
            process(v)
        for u in neighbours(G,v):
            if not visited[u]:
                enqueue(q,u)
                visited[u] = True
```

- We pass the process as a parameter.
- Runtime?

Runtime

- The only honest answers is "I don't know and neither do you."
- The while loop executes $|V|$ times
  - Why? Because we push each node exactly once in the queue
- Inside the while loop we do
  - Call `Process` (Runtime unkown)
  - Call `Neighbours` (runtime depends on graph data structure)
- Therefore the correct answer is $O(|V| \cdot (P + N))$ where
  - P is the runtime of one call to `process`
  - N is the runtime of one call to `neighbours`

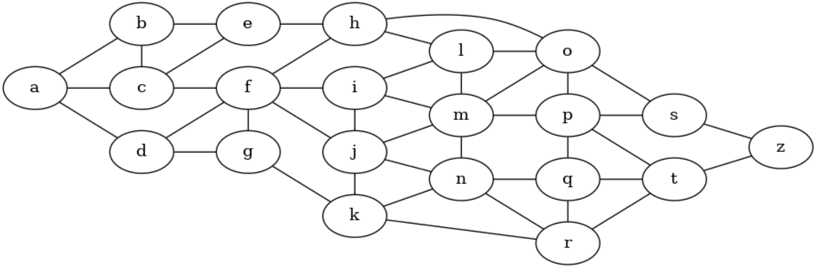
Small test

```
G={0:set([1,2,3]), 1:set([2,3]), 2:set([3,4,5]),
   3:set([4,5]), 4:set([5]), 5:set()}
bfs(G,1,process=print)
```

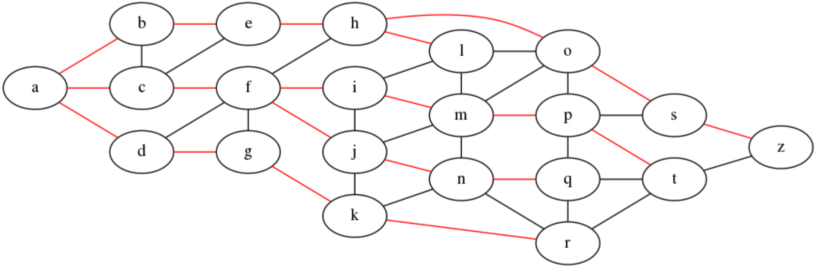
```
1
2
3
4
5
```

Homework/Test question

- Run BFS starting on node *a* and draw the resulting tree.



- You could have used the code presented to solve this and check.



Applications of BFS

- Find connected components
- Shortest path in unweighted graphs
- Testing for bipartite property
- Cuthill-McKee (numerical analysis)

Find connected components

Problem

Given a graph, find the number of components it contains.

Finding connected components

```
def components(G):
    count = 0
    visited = [False]*len(nodes(G))
    for v in nodes(G):
        if not visited[v]:
            count = count + 1
            bfs(G,v,visited)
    return count
```

- How does this work?
- Runtime?
- What if you wanted to return the components?

Example

```
G={0:set([1,2,3]), 1:set([2,3]), 2:set([3,4,5]),
   3:set([4,5]), 4:set([5]), 5:set()}
print('components ', components(G))
G={0:set([1,2,3]), 1:set([2,3]), 2:set([3,4,5]),
   3:set([4,5]), 4:set([5]), 5:set(),
   6:set([7,8]),7:set(), 8:set()}
print('components ', components(G))
```

```
components  1
components  2
```

Bipartite?

Problem

Can you colour the nodes of a graph using red and blue so that any two adjacent nodes are of a different colour?

Bipartite? (Fake code)

```
def bipartite(G):
    for v in vertices(G):
        if not visited[v]:
            colour[v] = red
            BFS(G,v)

def process(x,y):
    if colour[x] == colour[y]:
        abort('Not bipartite')
    else:
        colour[y] = complement(colour[x])
```

Where do you add the call to `process`?

Implement a bipartite check (Fake code)

```
def bfs(G,r,process=None):
    # G is a graph.  r is a node of this graph.
    enqueue(r)
    visited[r] = True
    while v = dequeue():
        for u in neighbors(G,v):
            if process:
                process(v,u)
            if not visited[u]:
                enqueue(u)
                visited[u] = True
```

Shortest path (unweighted graph)

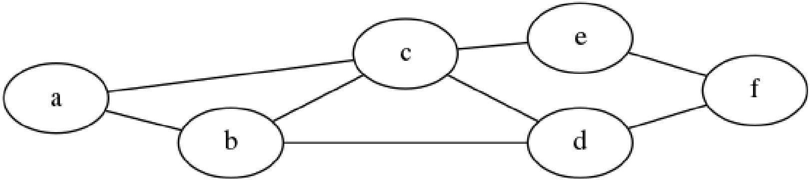
Problem

Given an undirected and unweighted graph, find the shortest paths from one identified node to all others.

Key idea:

By constructing a breadth first tree, we are getting from the start node to every other node with as few hops as possible.

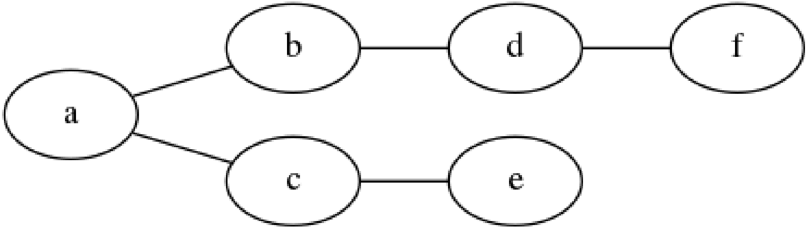
Shortest paths tree rooted at node a.



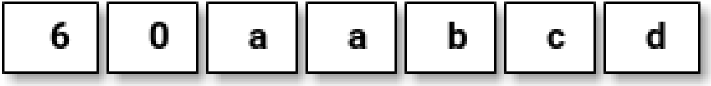
Questions

- Find the rooted tree.
- In what data structure would you keep this rooted tree?

Rooted tree data structure



Good data structure for a rooted tree:



Rooted tree data structure

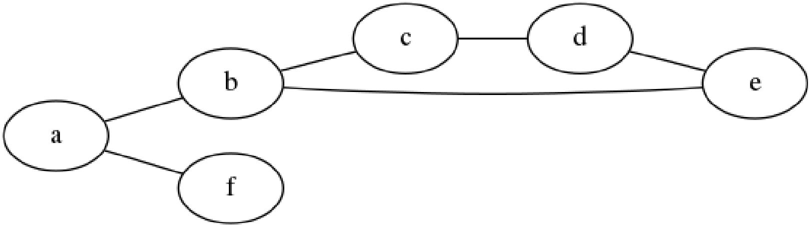
Details

- A vector of length equal to number of vertices+1
- Number of vertices in position 0, then, at every position
  - X for no parent (root)
  - Parent node

Depth First Search

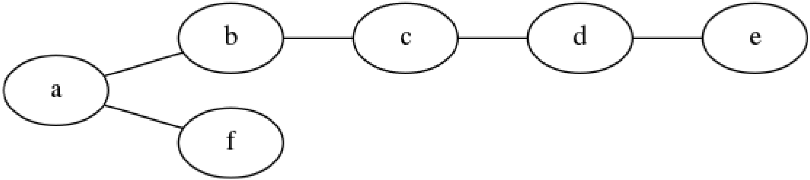
- Pick/obtain a node (known as the root).
- For each of its neighbors
  - Visit the neighbour
  - Start a DFS on that neighbour

Example graph



DFS(G,a)

- Order of the visit : a, b, c, d, e, f



Implement a DFS

```
def dfs(G,r):
```

Implement a DFS

```
def dfs(G,r):
    push(r)
    while u = pop():
        visited[u] = True  # Pre-order
        for v in neighbour(G,u):
            if not visited[v]:
                push(v)
        visited[u] = True  # Post-order
```

- Note: doing nothing here! Add process, as required.
- Runtime?

Required data structure

Stack:

- Push (an element on the stack)
- Pop (the last element pushed)

Can you do these in $O(1)$?

But using a stack is what recursion does implicitly!

Discuss

DFS via recursion

```
def DFS(G,r):
    visited[r] = True
    for v in neighbor(G,r):
        if not visited[v]:
            DFS(G,v)
```

Look Ma, no stack!

Applications

- Connected components.
- Topological sorting.
- Finding cycles in directed graphs.
- Transitive closures.

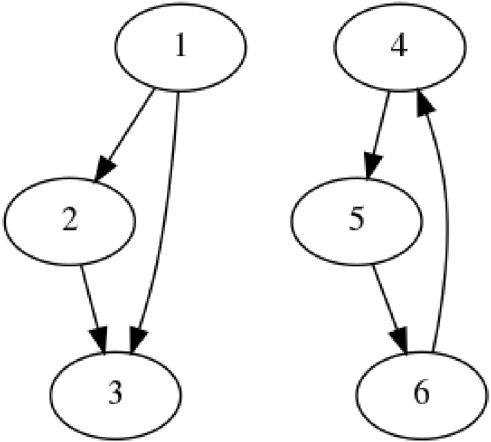
Topological sort

Definition

A topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

Examples

Left graph has a topsort 1,2,3; the right graph has not.



Why such a sort?

- To order activities in a complex process (building a house).
- To order instructions to be executed in a program running on a multi-core machine.

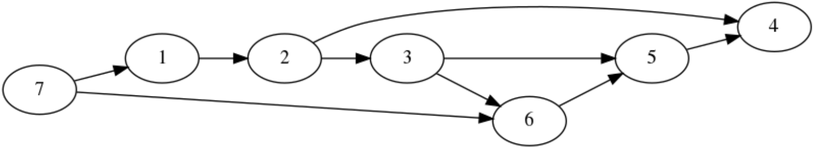
Topsort (Part I)

```
def topsort(G):
    stack=[]
    visited=[0]*(len(nodes(G))+1)
    for v in nodes(G):
        if visited[v]==0:
            if visit(G,v,stack,visited)==False:
                return None
    return stack
```

Topsort (Part II)

```
def visit(G,r,stack,visited):
    if visited[r]==2:
        return True
    elif visited[r]==1:
        return False
    visited[r]=1
    for v in neighbors(G,r):
        if visit(G,v,stack,visited)==False:
            return False
    stack.insert(0,r)
    visited[r] = 2
    return True
```

Example




```
G={7:[1,6],1:[2],2:[3,4],3:[6,5],4:[],5:[4],6:[5]}
print topsort(G)
```

```
[7, 1, 2, 3, 6, 5, 4]
```

Trace the code to understand.

Related result

A digraph has a topological ordering if and only if it has no cycles.

Proof: If it has no cycles, our algorithm will produce a topological order. If it has a cycle our code will detect it.

```
G={3:[1,2], 1:[2], 2:[]}
print(G,' -> ',topsort(G))
G={3:[1], 1:[2], 2:[3]}
print(G,' -> ',topsort(G))
```

```
{3: [1, 2], 1: [2], 2: []}  ->  [3, 1, 2]
{3: [1], 1: [2], 2: [3]}  ->  None
```

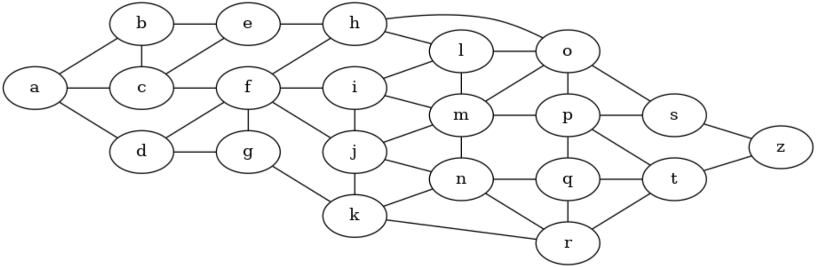
Different traversals of the graph.

- Push onto a queue before recursive call: Pre-order
- Push onto a queue after recursive call: Post-order
- Push onto a stack after recursive call: Reverse Post-order

Morals

- Graph algorithms are simple from a birds-eye view.
- The devil is in the details
  - You cannot claim a runtime without complete code or detailed assumptions.
  - You cannot claim correctness without complete code.
- Every algorithm requires a number of different data structures.

Homework/Test questions



- Draw the tree of a BFS started on node *a*.
- Draw the tree of a DFS started on node *a*.

Homework/Test questions

- State an efficient algorithm to find the diameter of a graph, the length of the largest shortest path in the graph. (No coding required; bird-s eye view sufficient.)
- Trace the Topsort algorithm by displaying the successive changes to the `stack` variable as it executes the call on Slide 45.
- Is the DFS of a graph unique if you fix the root?
- Is the BFS of a graph unique if you fix the root?
- Is the number of edges in a DFS fixed?