SCC0602 - Algoritmos e Estruturas de Dados I

Graphs

Professor: André C. P. L. F. de Carvalho, ICMC-USP
PAE: Rafael Martins D'Addio
Monitor: Joao Pedro Rodrigues Mattos

Today

- Abstract data types
  - Queues
- Graphs
- Graph representations
- Traversing graphs
  - Breadth-First Search
  - Depth-First Search
- Topological sort

Abstract Data Types (ADTs)

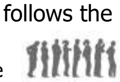
- Mathematical entity that defines data structures separating:
- Specification
  - What are the values the data structure can assume and the possible operations on these values
- Implementation
  - How the values and operations are implemented

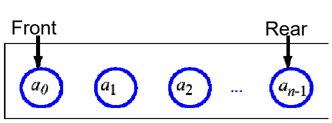
Abstract Data Types (ADTs)

- Examples of ADTs
  - Vector
  - List (Sequence)
    - Doubly linked list implementation
  - Dynamic set ADTs
    - Stack
    - Queue
    - Deque
    - Priority Queue

Queue ADT

- Insertion and removal of elements follows the **first-in-first-out** (FIFO) principle
  - Elements may be inserted at any time
    - But only the oldest element in the queue can be removed
  - Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)





Queue

Queue ADT

- Constructor:
  - **make** ():*Queue - Creates an empty queue*
- Access functions:
  - **size** (*S:Queue*):*integer - returns size of the queue*
  - **isEmpty** (*S:Queue*):*Boolean - tells if queue is empty*
  - **front** (*S:Queue*):*element - return element at the front*
- Manipulation procedures:
  - **Enqueue**(*S:Queue, o:element*):*Queue* - Inserts object *o* at the rear of the queue
  - **Dequeue** (*S:Queue*):*Queue* - Removes the object from the front of the queue

Array Implementation

- Create a queue using an array in a circular fashion
- A maximum size *N* is specified
- The queue has an N-element array *Q* and two integer variables:
  - *f*, index of the front element (head, for dequeue)
  - *r*, index of the element after the rear element (tail, for enqueue)

Array Implementation

- "wrapped around" configuration

- what does *f=r* mean?

An Array Implementation

- Pseudo code

Algorithm ***size()***
return *(N-f+r)* ***mod*** *N*

Algorithm ***isEmpty()***
return *size()=0*

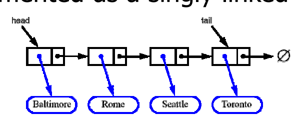
Algorithm ***front()***
if *isEmpty()* ***then***
  return *Error*
return *Q[f]*

Algorithm ***dequeue()***
if *isEmpty()* ***then***
  return *Error*
Q[f]=null
*f=(f+1)****modN***

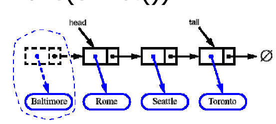
Algorithm ***enqueue(o)***
if *size = N - 1* ***then***
  return *Error*
Q[r]=o
*r=(r +1)****mod*** *N*

Using List ADT

- List ADT implemented as a singly linked list can be used

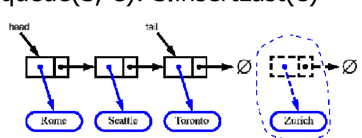


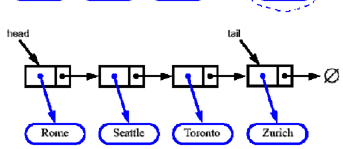
- Dequeue(S): S.remove(S.first())



Using List ADT

- Enqueue(S, e): S.insertLast(e)

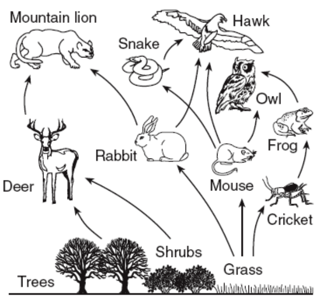




Graphs

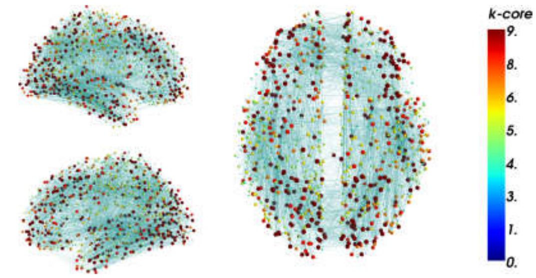
- Used in many applications
  - Electronic circuits
  - Energy distribution
  - Transport between places
  - Communication networks
  - Relationtions between
    - Components
    - People
    - Proteins

Food Chain



André de Carvalho - ICMC/USP 13

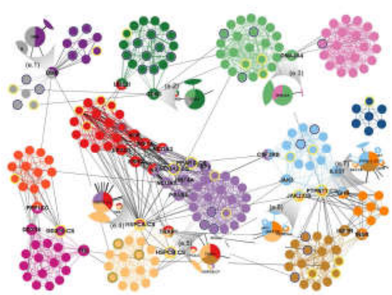
Brain neural connections



k-core: maximal subgraph with minimum degree ≥ k

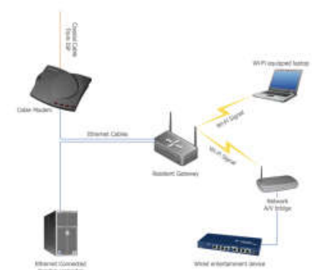
André de Carvalho - ICMC/USP 14

Protein interaction network



André de Carvalho - ICMC/USP 15

Home network

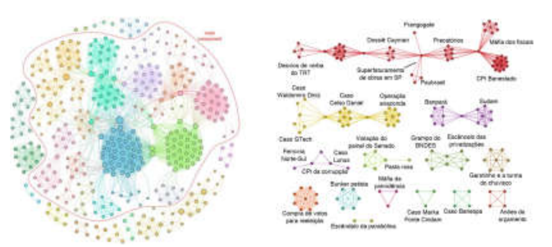


André de Carvalho - ICMC/USP 16

Social network

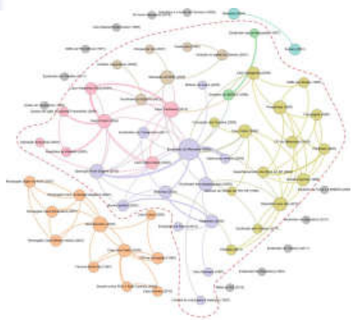
André de Carvalho - ICMC/USP 17

Corruption network



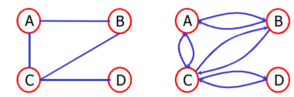
André de Carvalho - ICMC/USP 18

Corruption network



Graphs

- A **graph** G = (V,E) is composed of:
  - V: set of **vertices**
  - E ⊂ V× V: set of **edges** connecting graph **vertices**
    - Each **edge** ***e*** = (*u*,*v*), ***e*** ∈ ***E***, connects a pair of vertices
- A graph can be **directed** or un**directed**
  - In a undirected graph an edge between *u* and *v* is represented by both (*u*,*v*) ∈ E and (*v*,*u*) ∈ E



V = {A, B, C, D}

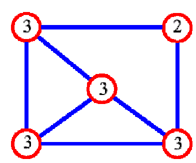
E = {(A,B), (B,A), (A,C), (C,A), (C,D), (D,C), (B,C), (C,B)}

In either case

- May think of vertices storing other information
  - Attributes (name, IP address, …)
  - Information for algorithms that will be performed on the graph
- We will want to be able to do the following operations:
  - Edge Membership: Is edge e in E?
  - Neighbor Query: Who are the neighbors of vertex v?

Graph terminology

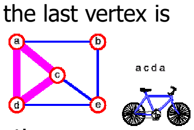
- A vertex *v* is **adjacent** to a vertex *u* iff (*u*,*v*) ∈ E
- **Degree** of a vertex: # of adjacent vertices



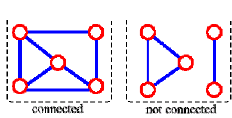
- **Path**: a sequence of vertices $v_1, v_2, \ldots v_k$, such that v_{i+1} is adjacent to v_i for $i = 1 \ldots k-1$

Graph terminology

- **Simple path**: a path with no repeated vertices
- **Cycle**: a simple path in which the last vertex is the same as the first vertex

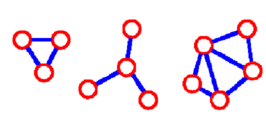


- **Connected** graph: any two vertices are connected by some path



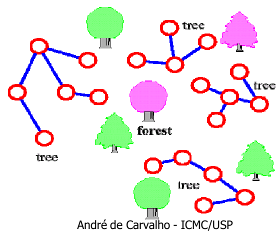
Graph terminology

- **Subgraph**: a subset of vertices and edges forming a graph
- **Connected component**: a maximal connected subgraph
  - Ex.: graph with 3 connected components



Graph terminology

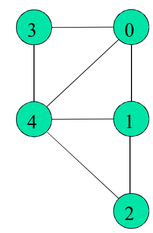
- **(free) tree**: connected graph without cycles
- **forest**: collection of trees



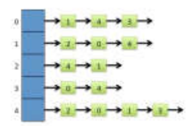
Data structures for graphs

- Adjacency matrix

$$\begin{array}{c|c} & \begin{matrix}0&1&2&3&4\end{matrix}\\ \begin{matrix}0\\1\\2\\3\\4\end{matrix} & \begin{bmatrix}0&1&0&1&1\\1&0&1&0&1\\0&1&0&0&1\\1&0&0&0&1\\1&1&1&1&0\end{bmatrix}\end{array}$$

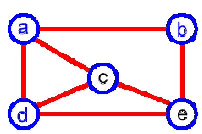


- Adjacency list



Adjacency matrix

- Matrix M with entries for all pairs of vertices
  - M[i,j] = true if there is an edge (i,j) in the graph
  - M[i,j] = false if there is no edge (i,j) in the graph
  - Space = $O(|V|^2)$

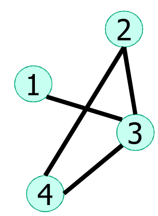


$$\begin{array}{c|c} & \begin{matrix}a&b&c&d&e\end{matrix}\\ \begin{matrix}a\\b\\c\\d\\e\end{matrix} & \begin{bmatrix}0&1&1&1&0\\1&0&0&0&1\\1&0&0&1&1\\1&0&1&0&1\\0&1&1&1&0\end{bmatrix}\end{array}$$

Adjacency matrix

- Option 1: adjacency matrix

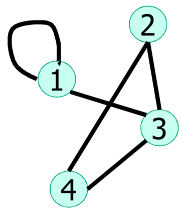
$$\begin{array}{c|c} & \begin{matrix}1&2&3&4\end{matrix}\\ \begin{matrix}1\\2\\3\\4\end{matrix} & \begin{bmatrix}0&0&1&0\\0&0&1&1\\1&1&0&1\\0&1&1&0\end{bmatrix}\end{array}$$



Adjacency matrix

- Option 1: adjacency matrix

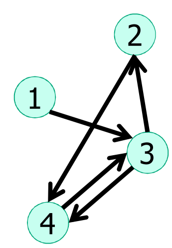
$$\begin{array}{c|c} & \begin{matrix}1&2&3&4\end{matrix}\\ \begin{matrix}1\\2\\3\\4\end{matrix} & \begin{bmatrix}1&0&1&0\\0&0&1&1\\1&1&0&1\\0&1&1&0\end{bmatrix}\end{array}$$



Adjacency matrix

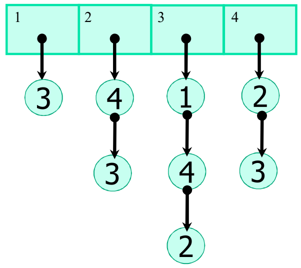
- Option 1: adjacency matrix

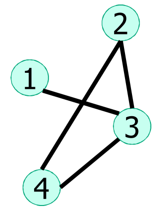
$$\begin{array}{c|c} & \text{Destination}\\ & \begin{matrix}1&2&3&4\end{matrix}\\ \text{Source}\ \begin{matrix}1\\2\\3\\4\end{matrix} & \begin{bmatrix}0&0&1&0\\0&0&0&1\\0&1&0&1\\0&0&1&0\end{bmatrix}\end{array}$$



Adjacency list

- Option 2: linked lists

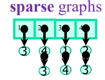




How would you modify this for directed graphs?

Graph representation

Generally better for **sparse** graphs

Suppose there are n vertices and m edges	$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$	
Edge membership Is e = {u,v} in E?	O(1)	O(deg(v)) or O(deg(u))
Neighbor query Give me v's neighbors	O(n)	O(deg(v))
Space requirements	O(n^2)	O(n + m)

We'll assume this representation for the rest of the class

Pseudocode assumptions

- **Graph** ADT with an operation
  - **V**(): *VertexSet*
- A looping construct "**for each** $v \in V$", where V is of a type *VertexSet,* and v is of a type *Vertex*
- **Vertex** ADT with operations:
  - **adjacent**(): *VertexSet*
  - **d**(): *int* and **setd**(d: *int*)
  - f(): *int* and **setf**(f: *int*)
  - **parent**() (ou π()): *Vertex* and **setparent**(p: *Vertex*)
  - **color**():{white, gray, black} and **setcolor**(c:{white, gray, black})

Graph searching algorithms

- Systematic search of every edge and vertex of the graph
- Graph G = (V,E) is either directed or undirected
- Applications
  - Compilers
  - Computer Graphics
  - Maze-solving
  - Mapping
  - Networks: routing, searching, clustering, etc.

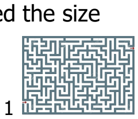
Graph search algorithms

- Searching a graph:
  - Systematically follow the edges of a graph to visit all the vertices of the graph
- Used to discover the structure of a graph
- Standard graph-searching algorithms
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

Breadth-First Search (BFS)

- Traverses a **connected component** of a graph, defining a **spanning tree** with useful properties
- In undirected graphs, similar to walk in a labyrinth with a **string**

- The starting vertex *s,* it is assigned a distance 0
- In the first round, the string is unrolled the size of 1 edge
  - All of the edges that are only one edge away from the starting vertex are visited (**discovered**) and assigned distances of 1



Breadth-First Search (BFS)

- In the second round, all new edges that can be reached by unrolling the string 2 edges are visited and assigned a distance of 2
- The walking continues until every vertex has been assigned a level
- The label of any vertex *v* corresponds to the length of the shortest path from *s* to *v*
  - In terms of the number of edges

André de Carvalho - ICMC/USP 37

Breadth-First Search (BFS)

- **Input:** Graph $G = (V, E)$, directed or undirected, and ***source vertex*** $s \in V$
- **Output:**
  - d[v]: distance (smallest # of edges or shortest path) from *s* to *v*, for all $v \in V$
    - $d[v] = \infty$ if *v* is not reachable from *s*
  - π[v]: *u, parent (predecessor) of v* where (u, v) is the last edge on the shortest path $s \rightsquigarrow v$
  - Builds a breadth-first tree with root *s* and all reachable vertices

André de Carvalho - ICMC/USP 38

Breadth-First Search (BFS)

- Definitions
  - Path between vertices *u* and *v* :
    - Sequence of vertices $(v_1, v_2, \ldots, v_k)$ such that $u = v_1$ and $v = v_k$, and $(v_i, v_{i+1}) \in E$, for all $1 \le i \le k\text{-}1$
  - Length of the path:
    - Number of edges in the path
  - Path is simple if no vertex is repeated

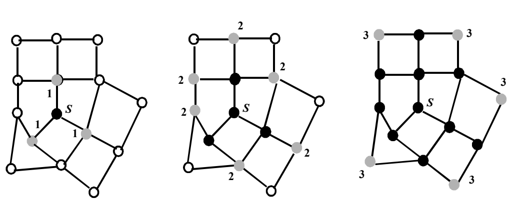
André de Carvalho - ICMC/USP 39

Breadth-First Search (BFS)

- Expands the frontier between discovered and undiscovered vertices uniformly across its breadth
  - A vertex is "discovered" the first time it is encountered during the search
  - A vertex is "finished" if all vertices adjacent to it have been discovered
- Colors the vertices to keep track of progress
  - White – Undiscovered
  - Gray – Discovered but not finished
  - Black – Finished
    - Colors are required only to reason about the algorithm
    - Can be implemented without colors

André de Carvalho - ICMC/USP 40

Breadth-First Search (BFS)



● Finished

● Discovered

○ Undiscovered

André de Carvalho - ICMC/USP 41

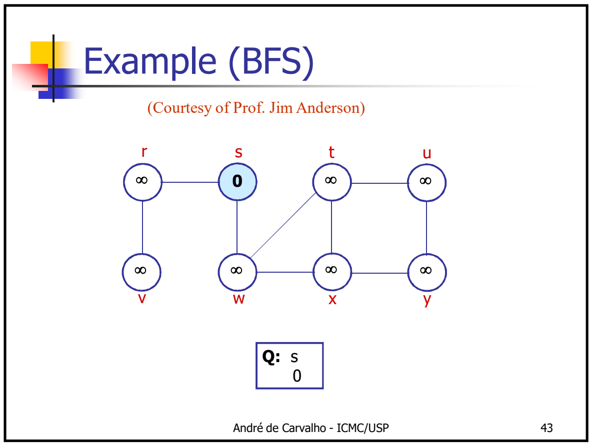
Breadth-First Search algorithm

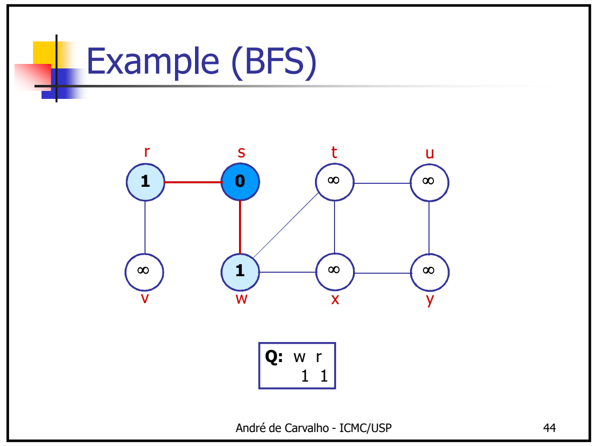
```
BFS(G,s)
for each vertex u in V[G] – {s} do
    color[u] ← WHITE
    d[u] ← ∞
    π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← ∅
ENQUEUE (Q,s)
while Q ≠ ∅ do
   u ← DEQUEUE (Q)
   for each v in Adj[u] do
       if color[v] = WHITE
       then color[v] ← GRAY
               d[v] ← d[u] + 1
               π[v] ← u
               ENQUEUE (Q,v)
   color[u] ← BLACK
```

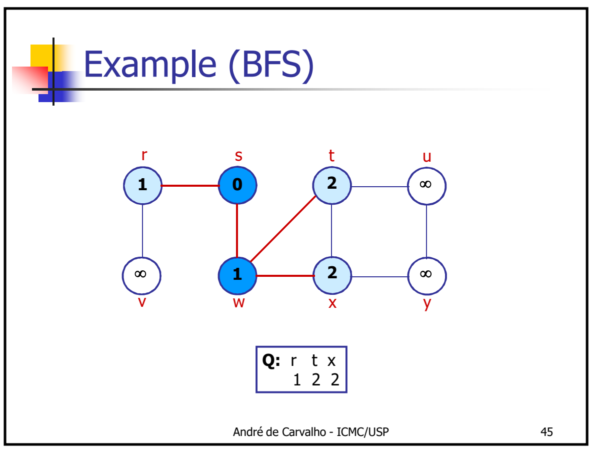
WHITE: undiscovered
GRAY: discovered
BLACK: finished

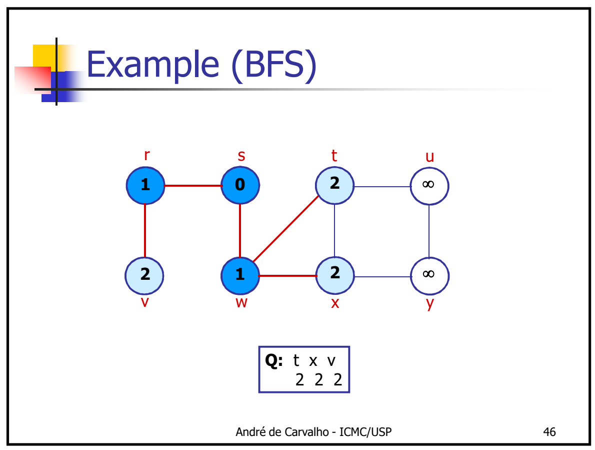
Q: queue of discovered vertices
color[v]: color of v
d[v]: distance from s to v
π[u]: predecessor of u

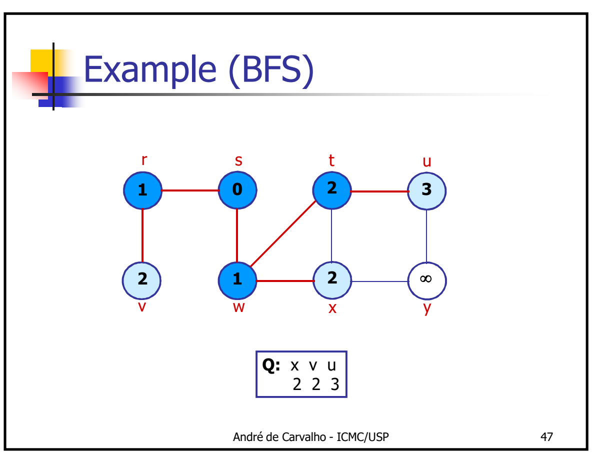
André de Carvalho - ICMC/USP 42

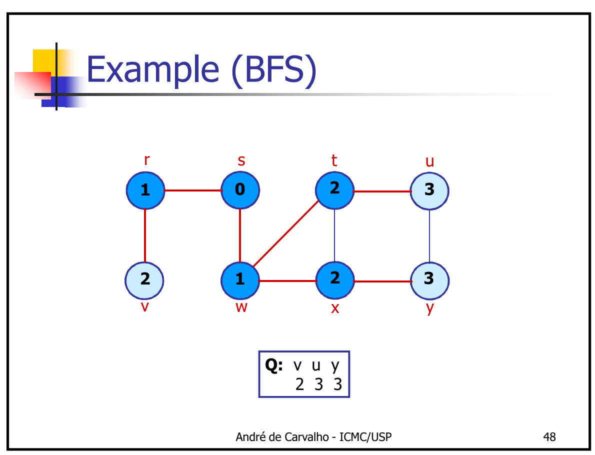

Example (BFS)
(Courtesy of Prof. Jim Anderson)
r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q: s
0
André de Carvalho - ICMC/USP
43


Example (BFS)
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q: w r
1 1
André de Carvalho - ICMC/USP
44

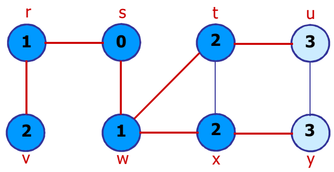

Example (BFS)
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q: r t x
1 2 2
André de Carvalho - ICMC/USP
45


Example (BFS)
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q: t x v
2 2 2
André de Carvalho - ICMC/USP
46


Example (BFS)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q: x v u
2 2 3
André de Carvalho - ICMC/USP
47

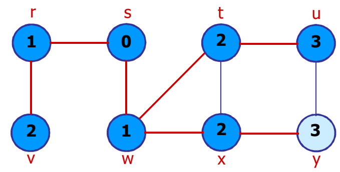

Example (BFS)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q: v u y
2 3 3
André de Carvalho - ICMC/USP
48

Example (BFS)



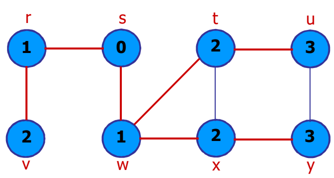
Q: u y
3 3

Example (BFS)



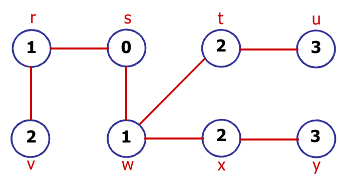
Q: y
3

Example (BFS)



Q: ∅

Example (BFS)



BF Tree

BFS running time

- Given a graph G = (V,E)
  - Initializing the algorithm takes O(V)
  - Vertices are enqueued if their color is white
  - Assuming that en- and dequeuing takes O(1) time the total cost of this operation is O(V)
  - Adjacency list of a vertex is scanned when the vertex is dequeued (and only then…)
  - The sum of the lengths of all lists is Θ(E) Consequently, O(E) time is spent on scanning them
- **Running time of BFS is O(V+E)**
  - Linear in the size of the adjacency list representation of G

BFS properties

- Given a graph G = (V,E), BFS **discovers all vertices reachable from a source vertex *s***
- It computes the **shortest distance** to all reachable vertices
- It computes a **breadth-first tree** that contains all such reachable vertices
- For any vertex *v* reachable from *s*, the path in the breadth first tree from s to v is the **shortest path** in G

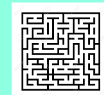
Depth-First Search (DFS)

- In undirected graphs, similar to walk in a labyrinth with a **string** and a **can of paint**



Using DFS to escape from a Labyritnth

1 Start at vertex s, tying the end of our string to s and paint s "visited"
2 Label s as the current vertex, named u
3 Travel along an arbitrary edge (u,v)
4 If edge (u,v) leads us to an already visited vertex v
Then Return to u
Else Unroll string and move to v and paint v "visited"
Set v as the current vertex and go to 3



If gets to a point where all incident edges on u lead to visited vertices
Then Backtrack by rolling the string to a previously visited vertex v
Set v as the current vertex and go to 3
If all incident edges on v lead to visited vertices
Then Backtrack as before
Continue to backtrack along the travelled path, finding and exploring unexplored edges, and repeating the procedure
If backtrack to vertex s and there are no more unexplored edges incident on s
Then DFS search is finished

Depth-First Search (DFS)

- Explore edges out of the most recently discovered vertex *v*
  - When all edges of *v* have been explored, backtrack to explore other edges leaving the vertex from which *v* was discovered (its *predecessor*)
  - "Search as deep as possible first."
  - Continue until all vertices reachable from the original source are discovered
  - If any undiscovered vertices remain, one of them is chosen as a new source and search is repeated from this source

Depth-First Search (DFS)

- **Input:** $G = (V, E)$, directed or undirected
  - No source vertex is given
- **Output:**
  - 2 timestamps on each vertex
    - Integers between 1 and 2|V|
      - $d[v]$ = ***discovery time*** (when *v* turns from white to gray)
      - $f[v]$ = ***finishing time*** (when *v* turns from gray to black)
  - $\pi[v]$: predecessor of *v*
  - *Vertex u* if *v* was discovered in *u* adjacency list
- Uses the same coloring scheme used for vertices as BFS

Pseudo-code

DFS(G)

*for each vertex u ∈ V[G] **do***
color[u] ← white
π[u] ← NIL
time ← 0
*for each vertex u ∈ V[G] **do***
if color[u] = white
then DFS-Visit(u)

Uses a global timestamp *time*.

DFS-Visit(u)

/ White vertex u has been discovered.*
color[u] ← GRAY
time ← time + 1
d[u] ← time
*for each v ∈ Adj[u] **do***
if color[v] = WHITE
then π[v] ← u
DFS-Visit(v)
/ Blacken u; it visited all its adj list*
color[u] ← BLACK
time ← time + 1
f[u] ← time

DFS Algorithm

- Initialize: color all vertices white
- Visit each and every white vertex using DFS-Visit
- Call from DFS to DFS-Visit(u) roots a new tree of the **depth-first forest** at vertex u
- When DFS finishes, every vertex has:
  - A discovery time *d*
  - A finishing time *f*

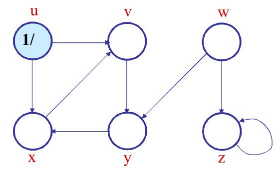
DFS Timestamping

- Vertex *u* is
  - White before time *d*[*u*]
  - Gray between time *d*[*u*] and time *f*[*u*]
  - Black after *f*[*u*]
- Structure throughout the search:
  - Gray vertices form a linear chain
    - Chain corresponds to a stack of vertices that were not exhaustively explored
    - DFS-Visit started but did not finished

André de Carvalho - ICMC/USP 61

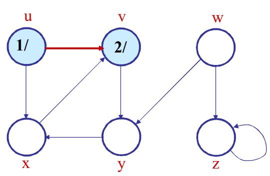
Example (DFS)

(Courtesy of Prof. Jim Anderson)



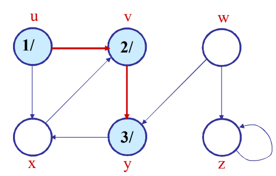
André de Carvalho - ICMC/USP 62

Example (DFS)



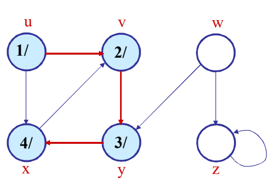
André de Carvalho - ICMC/USP 63

Example (DFS)



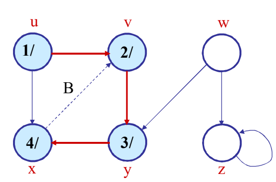
André de Carvalho - ICMC/USP 64

Example (DFS)



André de Carvalho - ICMC/USP 65

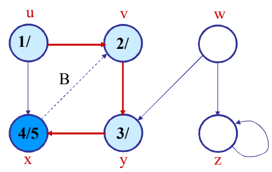
Example (DFS)



Non tree edges:
B: back edge
C: cross edge
F: forward edge

André de Carvalho - ICMC/USP 66

Example (DFS)

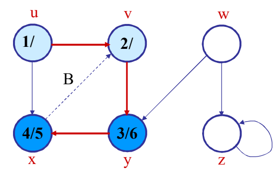


Non tree edges:
B: back edge
C: cross edge
F: forward edge

Discovery time / finish time

André de Carvalho - ICMC/USP 67

Example (DFS)

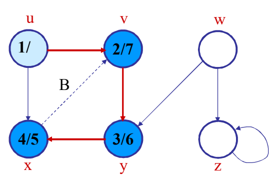


Non tree edges:
B: back edge
C: cross edge
F: forward edge

Discovery time / finish time

André de Carvalho - ICMC/USP 68

Example (DFS)

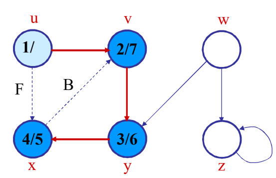


Non tree edges:
B: back edge
C: cross edge
F: forward edge

Discovery time / finish time

André de Carvalho - ICMC/USP 69

Example (DFS)

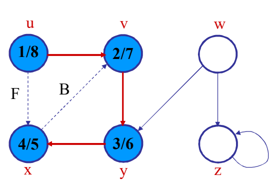


Non tree edges:
B: back edge
C: cross edge
F: forward edge

Discovery time / finish time

André de Carvalho - ICMC/USP 70

Example (DFS)

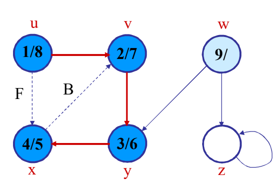


Non tree edges:
B: back edge
C: cross edge
F: forward edge

Discovery time / finish time

André de Carvalho - ICMC/USP 71

Example (DFS)

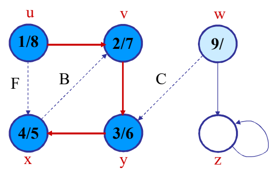


Non tree edges:
B: back edge
C: cross edge
F: forward edge

Discovery time / finish time

André de Carvalho - ICMC/USP 72

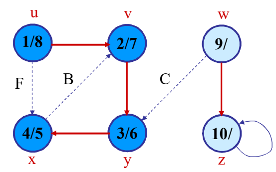
Example (DFS)



Non tree edges:
B: back edge
C: cross edge
F: forward edge

Discovery time / finish time

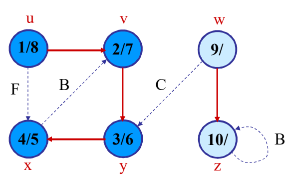
Example (DFS)



Non tree edges:
B: back edge
C: cross edge
F: forward edge

Discovery time / finish time

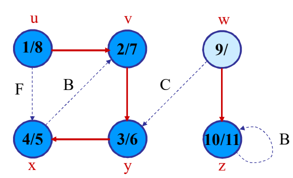
Example (DFS)



Non tree edges:
B: back edge
C: cross edge
F: forward edge

Discovery time / finish time

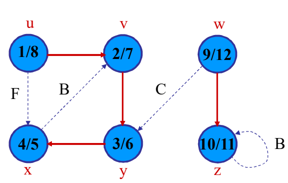
Example (DFS)



Non tree edges:
B: back edge
C: cross edge
F: forward edge

Discovery time / finish time

Example (DFS)



Non tree edges:
B: back edge
C: cross edge
F: forward edge

Discovery time / finish time

DFS Algorithm running time

- Running time
  - The loops in DFS take time Θ(V) each
    - Excluding the time to execute DFS-Visit
  - DFS-Visit is called once for every vertex
    - It is only called for white vertices, when it immediately paints the vertex with gray
    - For each DFS-visit, a loop interates over all *v*.**adjacent**()

$$\sum_{v \in V} |v.adjacent()| = \Theta(E)$$

    - Total cost for DFS-Visit is Θ(E)
- **Running time of DFS is Θ(V+E)**

Generic Graph Search

```
GenericGraphSearch(G,s)
for each vertex u ∈ G.V()
    u.setcolor(white)
    u.setparent(NIL)
s.setcolor(gray)
GrayVertices.init()
GrayVertices.add(s)
while not GrayVertices.isEmpty()
    u ← GrayVertices.remove()
    for each v ∈ u.adjacent() do
        if v.color() = white then
            v.setcolor(gray)
            v.setparent(u)
            GrayVertices.add(v)
    u.setcolor(black)
```

- BFS, when `GrayVertices` is a Queue
- DFS, when `GrayVertices` is a Stack

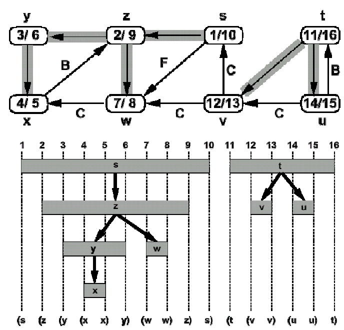
BFS versus DFS

- BFS
  - Search wider
  - Find shortest-path distances to a given source
  - Forms a tree
    - One source
  - FIFO (queue)
- DFS
  - Search deeper
  - It is often a subroutine in another algorithm
  - Forms a tree forest
    - One or more sources
  - LIFO (stack)

DFS Parenthesis Theorem

- Discovery and finish times have a parenthesis structure
  - Represents discovery of *u* with left parenthesis
    - "(u"
  - Represents finishing of *u* with right parenthesis
    - "u)"
  - History of discoveries and finishings makes a well-formed expression
    - Parenthesis are properly nested
    - Example: (s (a (b b) a) s)

DFS Parenthesis Theorem



Theorem 22.7

For all *u*, *v*, exactly one of the following holds:

1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither *u* nor *v* is a descendant of the other.
2. $d[u] < d[v] < f[v] < f[u]$ and *v* is a descendant of *u*.
3. $d[v] < d[u] < f[u] < f[v]$ and *u* is a descendant of *v*.

DFS Parenthesis Theorem

- For all *u*, *v*, one of the following holds:
  - $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither *u* nor *v* is a descendant of the other
  - $d[u] < d[v] < f[v] < f[u]$ and *v* is a descendant of *u*
  - $d[v] < d[u] < f[u] < f[v]$ and *u* is a descendant of *v*

DFS Parenthesis Theorem

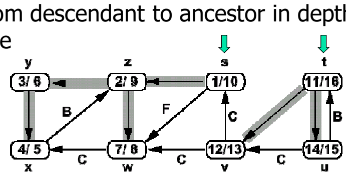
- Intuition for proof:
  - Any two intervals are either disjoint or enclosed (one inside the other)
  - Overlapping intervals would mean:
    - Finishing ancestor, before finishing descendant or
    - Starting descendant without starting ancestor

DFS Edge Classification

- DFS can be used to classify the edges of the graph G = (V,E)
  - Tree (T) edges
  - Back (B) edges
  - Forward (F) edges
  - Cross (C) edges
- Edge classification can provide important information about the graph
  - E.g. a direct graph is acyclic if and only if DFS produces no back edges

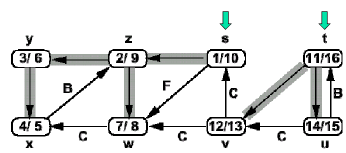
DFS Edge Classification

- Tree edge (from gray to white)
  - Edges in depth-first forest
- Back edge (from gray to gray)
  - From descendant to ancestor in depth-first tree



DFS Edge Classification

- Forward edge (from gray to black)
  - Non-tree edge from ancestor to descendant in depth-first tree
    - Cross edge (from gray to black)
  - Remainder: between trees or subtrees

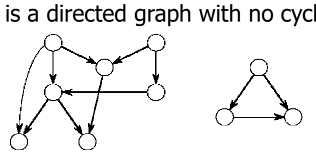


DFS Edge Classification

- Edge type for edge (*u, v*) can be identified when it is first explored by DFS
- Tree and back edges are the most important
- Most algorithms do not distinguish between forward and cross edges
- Undirected graphs:
  - Classification is difficult , since (u,v) = (v,u)
    - Edge is classified according to the vertex encountered first
  - Forward and cross edges never occur

Directed Acyclic Graphs

- A DAG is a directed graph with no cycles



- Often used to indicate precedence among events
  - Event ***a*** must happen before event ***b***
- Example: parallel code execution
- Total order can be introduced using **Topological Sorting**

Topological Sort (TS)

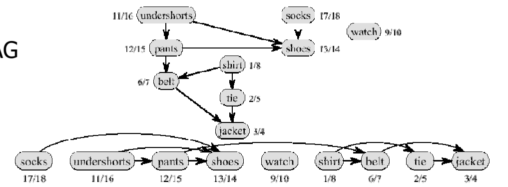
- TS of a DAG G = (V,E) is a linear ordering of all its vertices, such that:
  - If G contains an edge (u,v), u must appear before v in the ordering
    - No ordering is possible in a cyclic graph
  - Like ordering all vertices in a horizontal line so that all directed edges go from left to right
  - First vertex always has in-degree equal to 0
    - Vertex with no incoming edges

Topological Sort Example

- Precedence relations: an edge from *u* to *v* means that *u* must be finished before start *v*
- Intuition: conclude a task when all necessary previous tasks are concluded

DAG

TS



Topological Sort Algorithm

Topological-Sort(G)
1 call DFS(G) to compute finishing times f[v] for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 return the linked list of vertices

The linked lists has a total ordering

Topological Sort Running Time

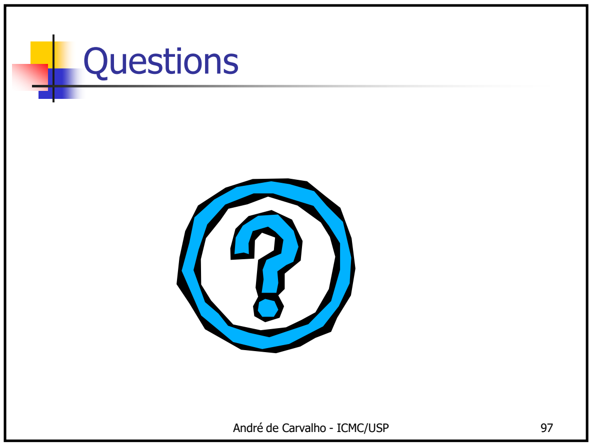
- Running time
  - DFS: $O(V+E)$ time
  - insert each of the $|V|$ vertices to the front of the linked list: $O(1)$ per insertion
- The total running time is $O(V+E)$

Next Lecture

- Strongly connected components
- Transpose of directed graphs
- Algorithm to find strongly connected components

Acknowledgement

- A large part of this material were adapted from
  - Simonas Šaltenis, Algorithms and Data Structures, Aalborg University, Denmark
  - Mary Wootters, Design and Analysis of Algorithms, Stanford University, USA
  - George Bebis, Analysis of Algorithms CS 477/677, University of Nevada, Reno
  - David A. Plaisted, Information Comp 550-001, University of North Carolina at Chapel Hill


Questions
André de Carvalho - ICMC/USP
97