APPLICATIONS

- Check if nodes on a network are connected (telephones, communication, paths)
- Driving directions
- Formulate a plan. For example, how to fill in a sudoku puzzle. Each node is a partially completed puzzle and each edge fills in one new square according to the rules of sudoku
- Compute the "pieces" or "components" of a graph clustering, structure of the web graph, etc.

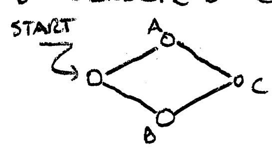
GENERIC GRAPH SEARCH ALGORITHM

Basis of graph search which can be refined with breadth first search or depth first search (will be discussed later).

GOALS:

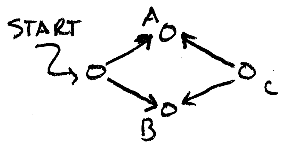
1) Find everything findable from a given starting vertex

UNDIRECTED GRAPH



can travel in any direction to get to the node in question
All nodes findable from the starting vertex

DIRECTED GRAPH



can only travel from tail to head to get to the node in question. o⟶o, NOT o⟵o
Only nodes A and B findable from the starting vertex

2) Don't explore anything twice with a goal of $O(m+n)$ (m: EDGES, n: NODES)

ALGORITHM PSEUDOCODE:

```
Given graph, G; starting vertex, s
  Initialize s as explored (isExplored = true), all other nodes unexplored
  While possible to access an unexplored node:
    Choose an edge (u, v) with u as explored and v as unexplored
    Mark v as explored (isExplored = true)
```

(con't on next sheet)

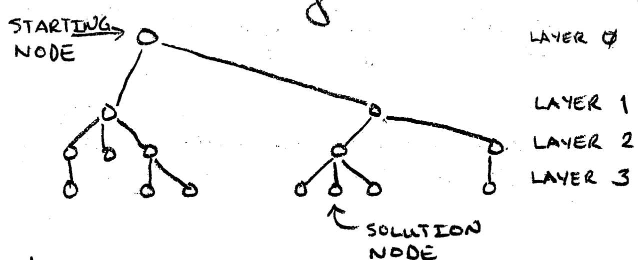
GENERIC GRAPH SEARCH ALGORITHM (cont)

After a graph search algorithm runs if a node, v, is explored than there is a path in graph, G, from the starting vertex, s, to v whether the graph is directed or undirected.

This is true because it is impossible to have v marked as explored after the algorithm runs if there is no connection, the algorithm has a while loop to ensure all nodes that can be explored get explored before terminating.

BREADTH FIRST SEARCH (BFS) OVERVIEW

- Explores nodes in "layers". Check all nodes connected to the starting node, check all nodes connected to the nodes you just checked, continue checking until you reach your solution. The number of layers checked is the shortest path distance, i edges, from the starting vertex to the solution node.

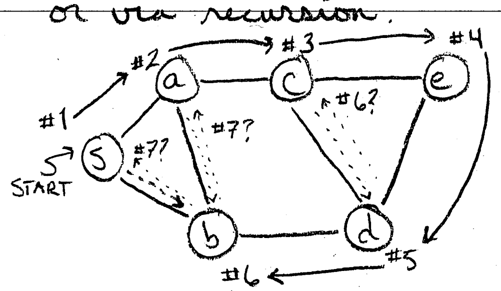


SHORTEST DISTANCE = 3 EDGES

- Better for undirected graphs
- Runs in $O(m+n)$ time using a queue (first-in, first-out, FIFO)

DEPTH FIRST SEARCH (DFS) OVERVIEW

- Explores aggressively like a maze, backtracking only when necessary
- Computes topological ordering of a directed acyclic graph
  Fancy way of saying determines order of precedence for a one-way system, that is, a has to happen before b, or you can't put a roof on a house if the frame has not yet been built.
- Better for directed graphs
- Run in $O(m+n)$ time using a stack (last-in, first-out, LIFO) ~~or via recursion.~~



BREADTH FIRST SEARCH ALGORITHM

PSEUDOCODE:

```
1 BFS (graph, G; starting vertex, s):
2   Set all nodes as unexplored
3   Mark s as explored
4   Let Q = queue data structure (FIFO), initialized with s
5   While Q ≠ Ø:
6     Remove the first node of the queue, call it v
7     For each edge, (v, w):
8       If w is unexplored:
9         Mark w as explored
10        Add w to the Q (gets added to the end)
```

At the end of a BFS if an arbitrary point, v, is marked as explored the graph, G, has a path from the starting vertex, s, to v. This was proved with the generic form of the graph search algorithm on page 2.

Also, the running time of the while loop is $O(m_s + n_s)$, where:

m_s = # of edges reachable from s.
n_s = # of nodes " " ".

We only deal with reachable nodes, unreachable nodes never get considered.

FINDING THE SHORTEST PATH

Compute dist(v), the fewest number of edges on a path from s to v.

Between lines 4 and 5 of the Pseudocode add dist(v) initialization.

```
If s = v
  Then dist(v) = Ø
  Else dist(v) = ∞  # We don't know if a
                      connection exists
```

Between lines 7 and 8 of the Pseudocode, when considering edge (v, w)

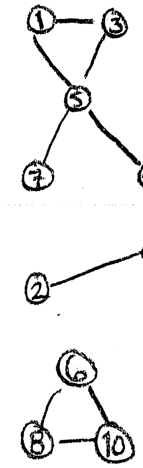
```
If w is unexplored
  Then dist(w) = dist(v) + 1
```

Now at termination, dist(v) = i, where i represents the i^{th} layer in which v was found and is the shortest path from s to v.

BREADTH FIRST SEARCH ALGORITHM (con't)

FIND CONNECTED COMPONENTS OF AN UNDIRECTED GRAPH

Given an undirected graph
Find the connected components, the portions of the graph that are all connected. For example, from the graph on the right, there are three connected components (1, 3, 5, 7, 9), (2, 4), and (6, 8, 10).



We solve this problem using equivalence relations. For an object to have equivalence it must possess three characteristics:

REFLEXIVE: Everything must be related to itself. In a graph a node is reflexive because it always possesses a path to itself.

SYMMETRIC: If an element a is related to an element b, element b must be related to element a. In an undirected graph if there is a path from node u to node v, there exists a symmetrical path from node v to node u.

TRANSITIVE: If an element a is related to an element b and b is related to an element c, a must be related to c. In the undirected graph above node 1 has a connection to node 5 and node 5 has a connection to node 9, therefore node 1 is connected to node 9 through (transitively) node 5.

For an object to be a member of a connected component in a graph it must possess an equivalence relation with other nodes in the graph.

This allows us to compute all the connected components in a graph. An application of which would allow us to check if a network has become disconnected (ping a node).

(con't on next sheet)

BREADTH FIRST SEARCH ALGORITHM (con't)

FIND CONNECTED COMPONENTS OF AN UNDIRECTED GRAPH (con't)

PSEUDOCODE TO COMPUTE ALL COMPONENTS

```
Set all nodes to unexplored  # Assume labeled 1 to n
Initialize an object to store connected components
For i = 1 to n
  Initialize an object to hold members of a connected component
  If i is not yet explored  # From BreadthFirstSearch
    Then BreadthFirstSearch (Graph G, vertex i)  # PAGE 3
    Append members into the connected component
  Append connected component to the store of connected components
```

Runtime is $O(m+n)$

DEPTH FIRST SEARCH ALGORITHM

Depth First Search (DFS) uses a stack instead of a queue and other modifications from BFS.

PSEUDOCODE (RECURSIVE):

```
DFS (Graph, G; starting vertex, s)
  Mark s as explored
  For every edge (s, v):
    If v is unexplored:
      Then DFS (G, v)
```

At the end of a DFS if an arbitrary point, v, is marked as explored the graph, G, has a path from the starting vertex, s, to v. This is proved with the generic form of the graph search algorithm on page 2.

The running time is similar to BFS at $O(m_s + n_s)$, where:

m_s = # of edges reachable from s.
n_s = # of nodes " " ".

(con't on next sheet)

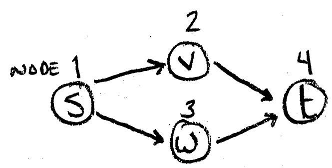
DEPTH FIRST SEARCH ALGORITHM (cont)

TOPOLOGICAL SORT

A topological ordering of a directed graph, G, is a labeling, f, of G's nodes such that:

a) The label of an arbitrary point, v, represented by f(v) is in the set of nodes within the directed graph from 1 to n.

b) For edge, (u, v), in G $f(u) < f(v)$ or node 1 must be found (or ordered) in G before nodes 2 or 6.

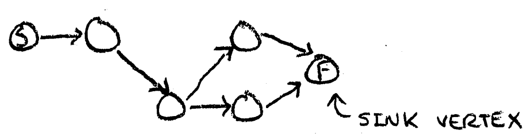


DIRECTED GRAPH, G

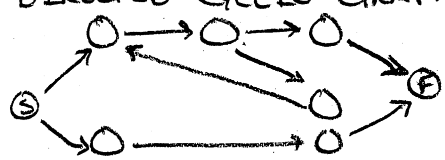
Useful for sequencing tasks while respecting all precedence constraints. Think project schedules (You can't put a car's engine in before the frame is built!), course prerequisites, etc.

Remember, this is only effective for acyclic graphs, that is, no node loops back to (shares an edge) with a previous node.

DIRECTED ACYCLIC GRAPH



DIRECTED CYCLIC GRAPH



Therefore, every directed acyclic graph has at least one sink vertex (final nodes where no farther outgoing arcs exist)

PSEUDOCODE FOR STRAIGHTFORWARD APPROACH

```
Let v be a sink vertex of G
Set f(v) = n
Recurse on G with v removed as an element of G
```

This works because when v is assigned to position i, all outgoing arcs are already deleted which lead to later vertices in the order.

(cont on next sheet)

Depth First Search Algorithm (con't)

Topological Sort (con't)

The straightforward (naïve) approach to topological sort can be improved with a Depth First Search (DFS).

Pseudocode

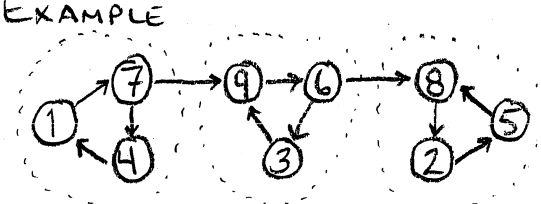
```
DFS_Loop (graph, G)
  Mark all nodes unexplored
  current_label = n   # To keep track of ordering
  For each vertex in G:
    If v is not explored:
      DFS (G, v, current_label)

DFS (graph, G; starting vertex, s; integer current_label)
  Mark s as explored
  For every edge (s, w):
    If w not yet explored:
      DFS (G, v, current_label)
  Set f(s) = current_label
  current_label --
```

Running time is $O(m+n)$

Finding Strongly Connected Components

A strongly connected component is any region of a directed graph from which one node can access any other connected node and return to the starting node. This is due to equivalence (see pg 4).

Example



The smaller sets: (8, 5, 2), (1, 7, 4), and (9, 6, 3) are strongly connected components of the graph. Any node in those component sets can get to and from any node within its set. Nodes outside of the sets are not strongly connected as node 9 can access node 5, but you cannot return to node 9 from node 5.

(Con't on next sheet)

DEPTH FIRST SEARCH ALGORITHM (con't)

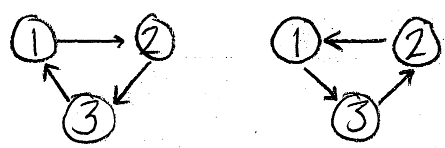
FINDING STRONGLY CONNECTED COMPONENTS (con't)

Order of searching nodes matters when attempting to discover strongly connected components (SCCs). This can be done in linear runtime with:

KOSARAJU'S TWO-PASS ALGORITHM

Given a directed graph, G

1) Let $G^{rev} = G$, but with all arcs reversed
   $G(1,2) \rightarrow G^{rev}(2,1)$



2) Run DFS_Loop from TOPOLOGICAL SORT on page 7 with G^{rev}.
   This step gives an ordering of nodes used as input for the next step.

   Let $f(v)$ be the "finishing order" of each vertex in G

3) Run DFS_Loop with G
   This step discovers the SCCs one-by-one by processing nodes in decreasing order of the finishing order.
   So, if a graph has nodes A, B, and C and the finishing orders of the nodes are A = 2, B = 3, C = 1 this step will start searching from B, then A, and finally C for SCCs.
   SCCs get classified as nodes with the same "leader".
   A "leader" is the first node (starting node) used to reveal an SCC. For example, in a graph with two SCCs one SCC will have a "leader" node of say node 6, and the other SCC will have a "leader" node of node 2. Those were the two nodes that were used to discover its respective SCC and all nodes in each SCC are associated with their respective "leader" node.

(con't on next sheet)

Depth First Search Algorithm (con't)

Finding Strongly Connected Components (con't)

Kosaraju's Two-Pass Algorithm (con't)

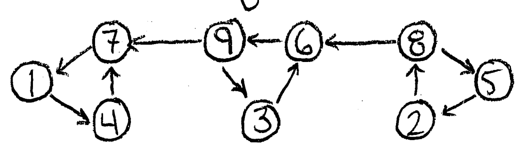
Pseudocode, Assume nodes labelled from 1 to n

```
DFS_Loop (graph G)
  t = Ø   # Used to set finishing order in first pass and
          # represents the number of nodes processed
  s = null # Used to represent leaders in the second pass
           # and represents the current source (starting)
           # vertex
  For i = n down to 1:
    If i is not yet explored:
      s = i
      DFS (G, i, t, s)

DFS (graph, G; node, i; finish order, t; starting vertex, s)
  Mark i as explored  # For entirety of the DFS_Loop
  Set leader(i) = s
  For each arc, (i, j), in G:
    If j is not yet explored:
      DFS (G, j, s, t)
  t ++
  Set f(i) = t  # f(i) represents i's finishing order
```

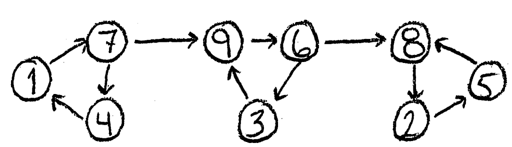
Example

Step 1: Create G^{rev} from G

Given, G:



G^{REV}



(Con't on next sheet)

DEPTH FIRST SEARCH ALGORITHM (con't)

FINDING STRONGLY CONNECTED COMPONENTS (con't)

KOSARAJU'S TWO PASS ALGORITHM (con't)

EXAMPLE (con't)

STEP 2: Run DFS-Loop on G^{rev} to find the finishing order, $f(v)$, for each vertex.

```
t = Ø, i = 9   # 9 has not yet been explored
s = 9
DFS(G^rev, 9, Ø, 9)
  Mark 9 as explored
  Set Leader(9) = 9
  For each arc, (9, j), in G^rev   # (9,6)
    6 has not yet been explored
    DFS(G^rev, 6, Ø, 9)
      Mark 6 as explored
      Set leader(6) = 9   # Leader nodes irrelevant in Step 2
      For each arc, (6, j) in G^rev   # (6, 3), (6, 8)
        3 has not yet been explored
        DFS(G^rev, 3, Ø, 9)
          Mark 3 as explored
          Set leader(3) = 9
          For each arc, (3, j) in G^rev   # (3,9)
            9 has been explored
          Increment (t++)  t = 1
          Set f(3) = 1
```

# node, i	1	2	3	4	5	6	7	8	9
# f(i)	?	?	1	?	?	?	?	?	?

```
        8 has not yet been explored
        DFS(G^rev, 8, Ø, 9)
          Mark 8 as explored
          Set leader(8) = 9
          For each arc, (8, j) in G^rev   # (8, 2)
            2 has not yet been explored
            DFS(G^rev, 2, Ø, 9)
              Mark 2 as explored
              Set leader(2) = 9
              For each arc, (2, j) in G^rev   # (2, 5)
                5 has not yet been explored
                DFS(G^rev, 5, Ø, 9)
                  Mark 5 as explored
                  Set leader(5) = 9
```

(con't on next sheet)

Depth First Search Algorithm (con't)

Finding Strongly Connected Components (con't)

Kosaraju's Two Pass Algorithm (con't)

Example (con't)

Step 2 (con't)

For each arc, (5, j) in G^{rev} # (5,8)
8 has been explored
Increment t (t++), t = 2
Set f(5) = 2

# node, i	1	2	3	4	5	6	7	8	9
# f(i)	?	?	1	?	2	?	?	?	?

Increment t (t++), t = 3
Set f(2) = 3

# node, i	1	2	3	4	5	6	7	8	9
# f(i)	?	3	1	?	2	?	?	?	?

Increment t (t++), t = 4
Set f(8) = 4

# node, i	1	2	3	4	5	6	7	8	9
# f(i)	?	3	1	?	2	?	?	4	?

Increment t (t++), t = 5
Set f(6) = 5

# node, i	1	2	3	4	5	6	7	8	9
# f(i)	?	3	1	?	2	5	?	4	?

Increment t (t++), t = 6

# node, i	1	2	3	4	5	6	7	8	9
# f(i)	?	3	1	?	2	5	?	4	6

t = 6, i = 8 # 8 has been explored

t = 6, i = 7 # 7 has not yet been explored
DFS(G^{rev}, 7, 6, 7)
Mark 7 as explored
Set leader (7) = 7
For each arc (7, j) in G^{rev} # (7,9), (7, 4)
9 has been explored
4 has not yet been explored
DFS(G^{rev}, 4, 6, 7)
Mark 4 as explored
Set leader (4) = 7
For each arc, (4, j) in G^{rev} # (4, 1)
1 has not yet been explored
DFS(G^{rev}, 1, 6, 7)

(Con't on next sheet)

Depth First Search Algorithm (con't)

Finding Strongly Connected Components (con't)

Kosaraju's Two Pass Algorithm (con't)

Example (con't)

Step 2 (con't)

Mark 1 as explored
Set leader (1) = 7
For each arc (1, j) in G^{rev} # (1, 7)
7 has been explored
Increment t (t++), t = 7
Set f(1) = 7

# node, i	1	2	3	4	5	6	7	8	9
# f(i)	7	3	1	?	2	5	?	4	6

Increment t (t++), t = 8
Set f(4) = 8

# node, i	1	2	3	4	5	6	7	8	9
# f(i)	7	3	1	8	2	5	?	4	6

Increment t (t++), t = 9
Set f(7) = 9

# node, i	1	2	3	4	5	6	7	8	9
# f(i)	7	3	1	8	2	5	9	4	6

t = 9, i = 6 ... # 6 to 1 have been explored

Step 3: Run DFS-Loop on G to find the SCCs using the finishing order in decreasing value to explore the graph.

f(v)	9	8	7	6	5	4	3	2	1
node, v	7	4	1	9	6	8	2	5	3

t = ∅ # not needed this pass, will ignore
s = null # Represents leader nodes
For i = f(n) from n down to 1: # f(9) = node 7
7 has not yet been explored
s = 7
DFS(G, 7, t, 7)
Mark 7 as explored # For entirety of step 3
Set leader (7) = 7
For each arc, (7, j) in G: # (7, 1)
1 has not yet been explored
DFS(G, 1, t, 7)

(con't on next sheet)

DEPTH FIRST SEARCH ALGORITHM (con't)

FINDING STRONGLY CONNECTED COMPONENTS (con't)

KOSARAJU'S TWO-PASS ALGORITHM (con't)

EXAMPLE (con't)

STEP 3 (con't)

Mark 1 as explored
Set leader(1) = 7
For each arc, (1, j) in G: # (1,4)
4 has not yet been explored
DFS(G, 4, t, 7)
Mark 4 as explored
Set leader(4) = 7
For each arc (4, j) in G: #(4, 7)
7 has been explored

# node	1	2	3	4	5	6	7	8	9
# leader	7	?	?	7	?	?	7	?	?

s = 7
For i = f(n) from n down to 1: # f(8) = node 4
4 has been explored

s = 7
For i = f(n) from n down to 1: # f(7) = node 1
1 has been explored

s = 7
For i = f(n) from n down to 1: # f(6) = node 9
9 has not yet been explored
s = 9
DFS(G, 9, t, 9)
Mark 9 as explored
Set leader(9) = 9

# node	1	2	3	4	5	6	7	8	9
# leader	7	?	?	7	?	?	7	?	9

For each arc (9, j) in G: # (9, 3), (9, 7)
3 has not yet been explored
DFS(G, 3, t, 9)
Mark 3 as explored
Set leader (3) = 9

# node	1	2	3	4	5	6	7	8	9
# leader	7	?	9	7	?	?	7	?	9

(con't on next sheet)

DEPTH FIRST SEARCH ALGORITHM (cont)

FINDING STRONGLY CONNECTED COMPONENTS (cont)

KOSARAJU'S TWO-PASS ALGORITHM (cont)

EXAMPLE (cont)

STEP 3 (cont)

- For each arc, (3, j), in G: # (3,6)
  - 6 has not yet been explored
    - DFS(G, 6, t, 9)
      - Mark 6 as explored
      - Set leader(6) = 9

# node	1	2	3	4	5	6	7	8	9
# leader	7	?	9	7	?	9	7	?	9

      - For each arc, (6, j), in G: # (6, 9)
        - 9 has been explored
  - 7 has been explored
- s = 9
- For i = f(n) from n down to 1: # f(5) = 6
  - 6 has been explored
- s = 9
- For i = f(n) from n down to 1: # f(4) = 8
  - 8 has not yet been explored
  - s = 8
  - DFS(G, 8, t, 8)
    - Mark 8 as explored
    - Set leader(8) = 8

# node	1	2	3	4	5	6	7	8	9
# leader	7	?	9	7	?	9	7	8	9

    - For each arc, (8, j), in G: # (8, 5), (8, 6)
      - 5 has not yet been explored
      - DFS(G, 5, t, 8)
        - Mark 5 as explored
        - Set leader(5) = 8

# node	1	2	3	4	5	6	7	8	9
# leader	7	?	9	7	8	9	7	8	9

        - For each arc (5, j) in G: # (5, 2)
          - 2 not yet explored
          - DFS(G, 2, t, 8)
            - Mark 2 as explored
            - Set leader(2) = 8

# node	1	2	3	4	5	6	7	8	9
# leader	7	8	9	7	8	9	7	8	9

            - For each arc (2, j) in G: # (2, 8)
              - 8 has been explored

(cont on next sheet)

DEPTH FIRST SEARCH ALGORITHM (cont)

FINDING STRONGLY CONNECTED COMPONENTS (cont)

KOSARAJU'S TWO-PASS ALGORITHM (cont)

EXAMPLE (cont)

STEP 3 (cont)

s = 8
For i = f(n) from n down to 1: # f(3) = 2
2 has been explored

s = 8
For i = f(n) from n down to 1: # f(2) = 5
5 has been explored

s = 8
For i = f(n) from n down to 1: # f(1) = 3
3 has been explored

Now all nodes have been searched and the leader associated with each node is the SCC set the node is a part of

Leader = 7, SCC set = (1, 4, 7)
Leader = 8, SCC set = (2, 5, 8)
Leader = 9, SCC set = (3, 6, 9)