Föreläsning 20

Graphs and graph searches

TDDD86: DALP

Utskriftsversion av Föreläsing i *Datastrukturer, algoritmer och programmeringsparadigm*
02 December 2024

IDA, Linköpings universitet

20.1

Content

Contents

1 Graphs **1**
1.1 Introduction . . . 1
1.2 ADT graph . . . 3
1.3 Data structures . . . 3

2 Search in undirected graphs **5**
2.1 DFS . . . 5
2.2 BFS . . . 10
2.3 DFS vs BFS . . . 12

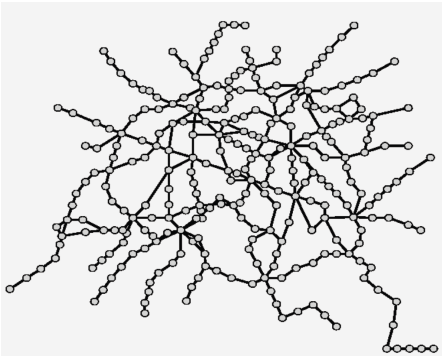
20.2

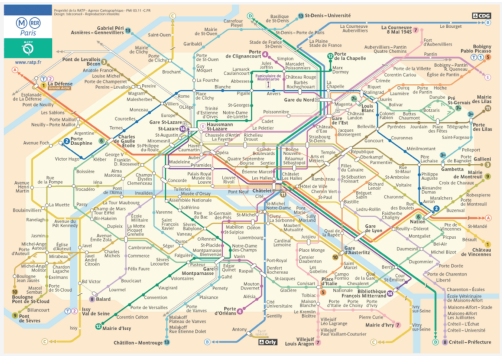
1 Graphs

1.1 Introduction

Definition

- A graph is a pair (V, E), where
  - V is a set of nodes or vertices
  - E is a set of pairs of nodes or edges
  - Nodes and edges can store data



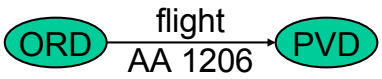


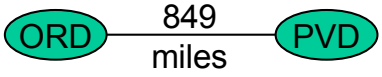
20.3

Types of edges

- Directed edge
  - ordered pair of nodes (u, v)
  - u is the start node, v is the end node
- undirected edge

  - unordered pair $\{u, v\}$
- In a directed graph, all edges are directed
- In an undirected graph, all edges are undirected





20.4

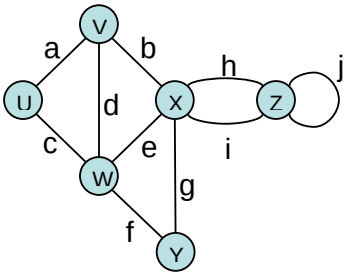
Why study graphs?

- Thousands of practical applications
- Multitude of graph algorithms
- Interesting abstraction with many applications
- Branch of computer science and discrete mathematics with many challenges

20.5

Terminology

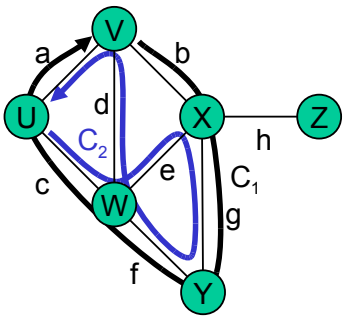
- An edge has endpoints (a has endpoints U and V)
- Edges that end in a node n are said to be incidents (a, d and b are incident to V)
- Nodes can be adjacent (U and V are adjacent)
- Nodes have degrees (X has degree 5)
- Parallel edges (h and i are parallel edges)
- Loops (j is a loop)



20.6

More terminology

- A cycle is a circular sequence of alternating nodes and edges. Each edge is preceded and followed by its endpoints.
- A simple cycle is a cycle where all nodes and edges are distinct.
- $C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$ is a simple cycle.
- $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, U)$ is *not* a simple cycle.



20.7

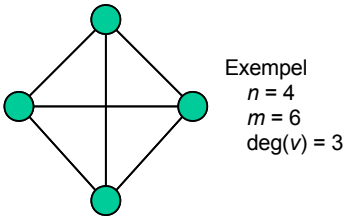
Properties

Property 1
$\sum_v \deg(v) = 2m$ Proof: each edge is counted twice

Property 2
In an undirected graph without loops and parallel edges, we have $m \leq n(n-1)/2$ Proof: Each node has a max degree of $(n-1)$

Notation

- n number of nodes
- m number of edges
- $\deg(v)$ is the degree of node v



20.8

Some algorithmic graph problems

- Path. Is there a path between s and t?
- Shortest path. What is the shortest path between s and t?
- Cycle. Is there a cycle in the graph?
- Euler tour. Is there a cycle that uses each edge exactly once?
- Hamiltonian cycle. Is there a cycle that uses each node exactly once?
- Connectivity. Is there a path from each node to each other node?
- MST. What is the best way to connect all nodes?
- Bi-connected graph. Is it possible to obtain a disconnected graph by removing a single node?
- Planar. Is it possible to draw a graph without having edges crossing each other?
- Isomorphism. Are two graphs identical except for renaming?

Challenge. Which of these problems are simple? difficult? impossible to solve efficiently?

20.9

1.2 ADT graph

Important methods for undirected graphs

- Node och edges
  - are positions
  - store labels
- Access methods
  - endVertices(e): an array with e's two endpoints
  - opposit(v,e): the node opposit v wrt. e
  - areAdjacent(v,w): **true** iff v and w are adjacent
  - replace(v,x): replace label in node v with x
  - replace(e,x): replace label in edge e with x

20.10

Important methods for undirected graphs

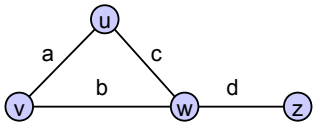
- Update methods
  - insertVertex(o): inserts a node with label o
  - insertEdge(v,w,o): insert an edge (v,w) with label o
  - removeVertex(v): remove node v (and its incident edges)
  - removeEdge(e): remove edge e
- Iteration methods
  - incidentEdges(v): edges incident to v
  - vertices$()$: all nodes in the graph
  - edges$()$: all edges in the graph

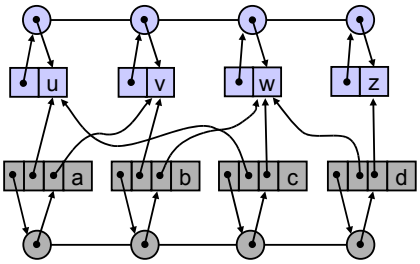
20.11

1.3 Data structures

Data-structure 1: Edge list

- A nodes' sequence with a sequence of positions for node objects
- An edges' sequence with a sequence of positions for edge objects
- A node object stores the label and a reference to the position in the nodes' sequence
- An edge object stores the label, a reference to start node object, a reference to the end node object and a reference to the position in the edges' sequence

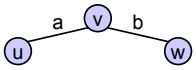


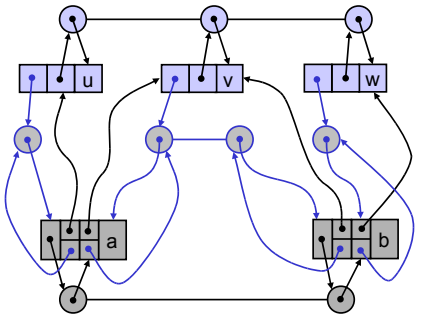


20.12

Data-structure 2: adjacent list

- Additional structure to the edge list
- Each node is associated to a list of its incident edges and references to the incident edge objects
- Edge objects are augmented with references to associated positions in the sequences of incident edges associated to its endpoints

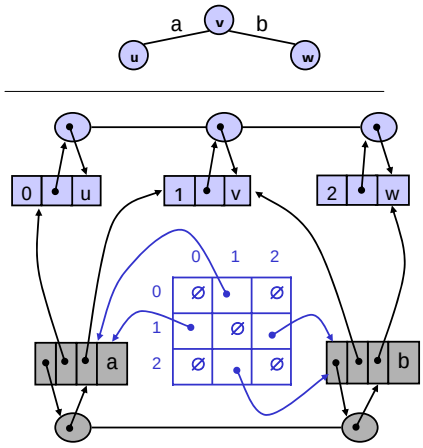




20.13

Data-structure 3: adjacent matrix

- Add extra structure to the edges' list
- Node objects augmented with integer keys (indices) associated with the nodes
- Two dimensional adjacent array
  - Reference to edge object for nodes that are adjacent
  - **null** for nodes that are not adjacent



20.14

Asymptotic performance

n noder, ***m*** bågar inga parallella kanter inga öglor	Båglista	Grannlista	Grann-matris
minne	$O(n + m)$	$O(n + m)$	$O(n^2)$
incidentEdges(***v***)	$O(m)$	$O(\deg(v))$	$O(n)$
areAdjacent (***v***, ***w***)	$O(m)$	$O(\min(\deg(v),\deg(w))$	$O(1)$
insertVertex(***o***)	$O(1)$	$O(1)$	$O(n^2)$
insertEdge(***v***, ***w***, ***o***)	$O(1)$	$O(1)$	$O(1)$
removeVertex(***v***)	$O(m)$	$O(\deg(v))$	$O(n^2)$
removeEdge(***e***)	$O(1)$	$O(1)$	$O(1)$

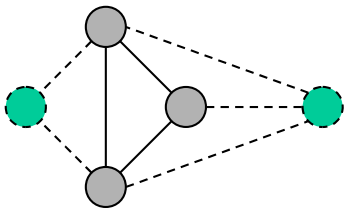
20.15

2 Search in undirected graphs

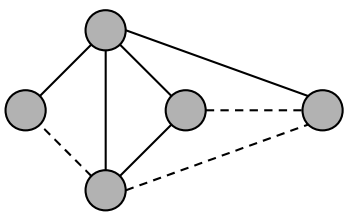
2.1 DFS

Sub-graphs

- A sub-graph S of a graph G is a graph s.t.:
  - Nodes of S are subset of the nodes in G
  - Edges in S are subset of the edges in G
- A spanning sub-graph of G is a sub-graph that includes all nodes in G



Delgraf

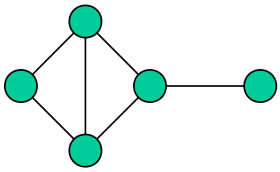


Spännande delgraf

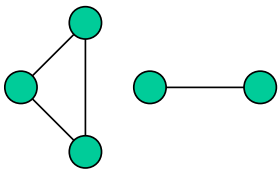
20.16

Connectivity

- A graph is connected if there is a path between each pair of nodes
- A connected component of a graph G is a maximal connected sub-graph of G



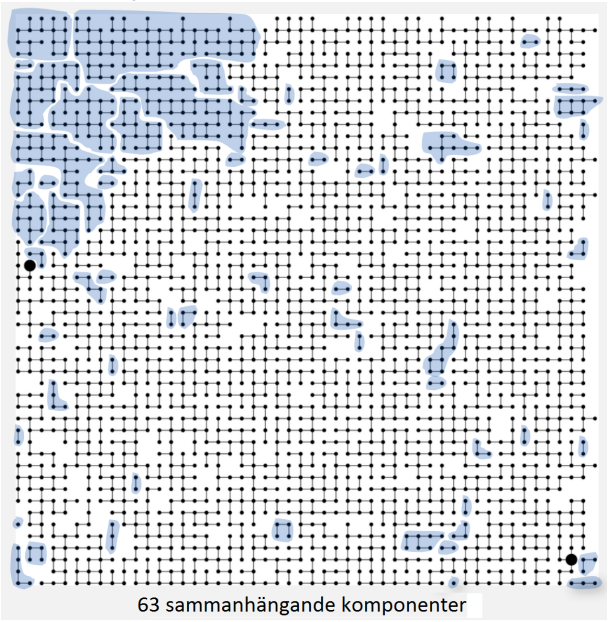
Sammanhängande graf



Ej sammanhängande graf med två sammanhängande komponenter

20.17

Connected components



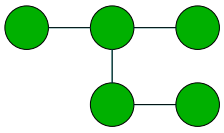
63 sammanhängande komponenter

20.18

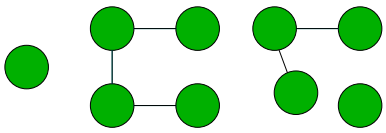
Trees and forests

- A (free) tree is an undirected graph T such that:
  - T is connected
  - T does not have any cycles
  - This definition is different from the one for rooted trees
- A forest is an undirected graph without cycles
- The connected components of a forest are trees

20.19



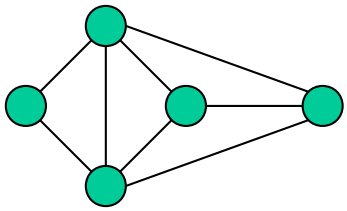
Träd



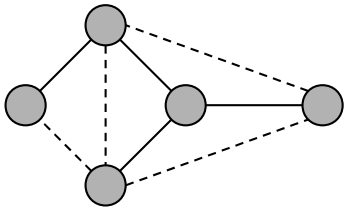
Skog

Spanning trees and forests

- A spanning tree for a connected graph is a spanning sub-graph that is a tree
- A spanning tree is not unique if the original graph is not a tree
- Spanning trees have applications in design of communication networks
- A spanning forest for a graph is a spanning sub-graph that is a forest



Graf



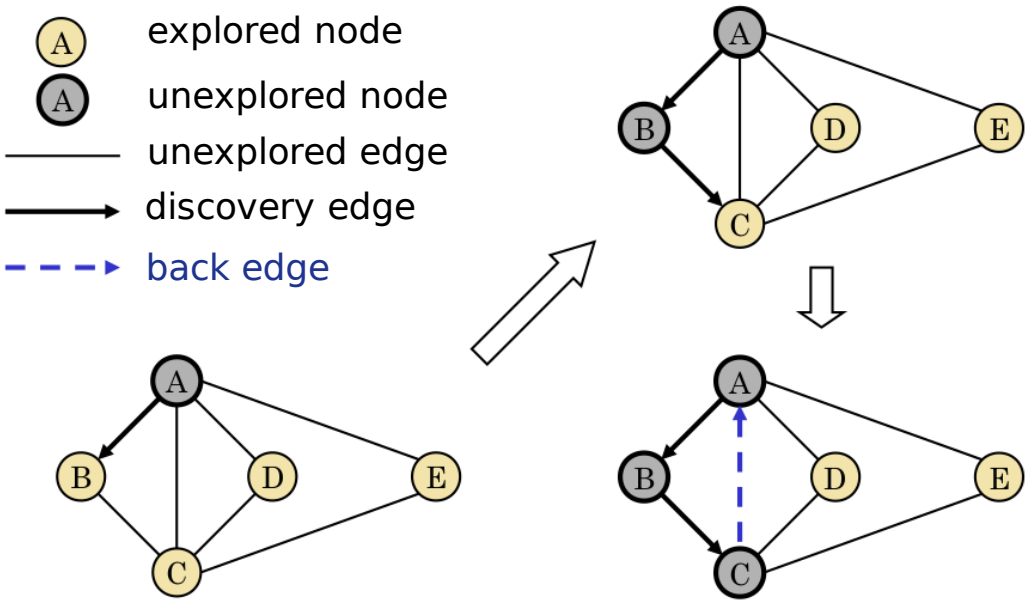
Spännande träd

DFS: Depth first search

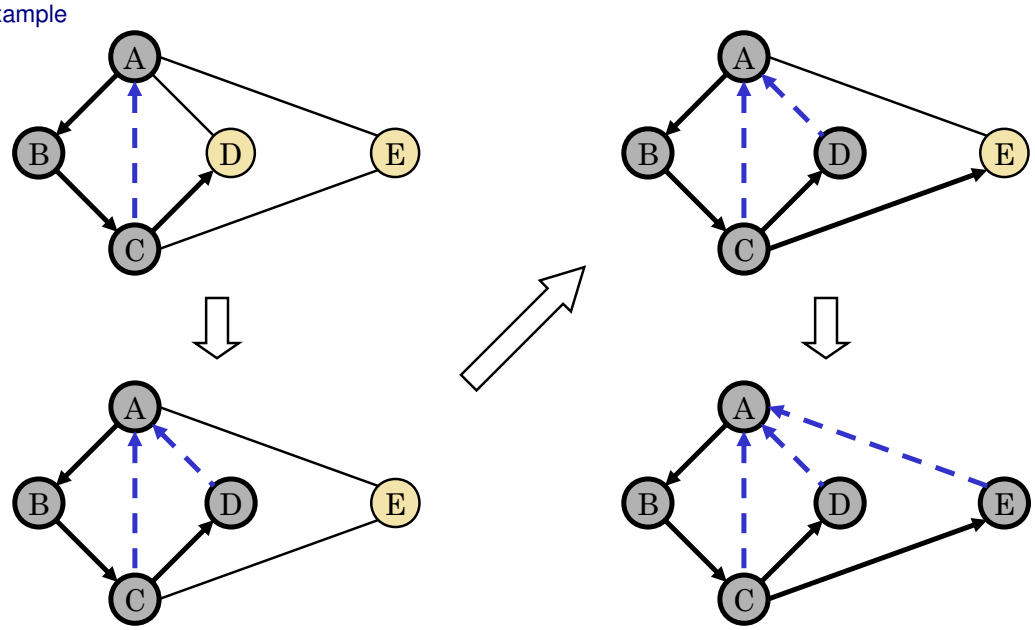
- Depth first search (DFS) is a generic technique for traversing a graph
- DFS in a graph G
  - Visits all nodes and edges in G
  - Checks whether G is connected
  - Computes connected components in G
  - Computes a spanning forest for G
- DFS on a graph with n nodes and m edges takes $O(n+m)$ time
- DFS can be augmented to solve other graph problems
  - Find a path between two given nodes in a graph
  - Find a cycle in a graph

Algorithm for DFS

```
procedure DFS(G)
    for all u ∈ G.VERTICES() do
        SETLABEL(u, UNEXPLORED)
    for all e ∈ G.EDGES() do
        SETLABEL(e, UNEXPLORED)
    for all v ∈ G.VERTICES() do
        if GETLABEL(v) = UNEXPLORED then
            DFS(G, v)
procedure DFS(G, v)
    SETLABEL(v, VISITED)
    for all e ∈ G.INCIDENTEDGES(v) do
        if GETLABEL(e) = UNEXPLORED then
            w ←OPPOSITE(v, e)
            if GETLABEL(w) = UNEXPLORED then
                SETLABEL(e, DISCOVERY)
                DFS(G, w)
            else
                SETLABEL(e, BACK)
```

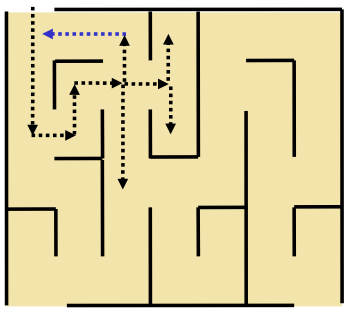


Example



DFS and labyrinth exploration

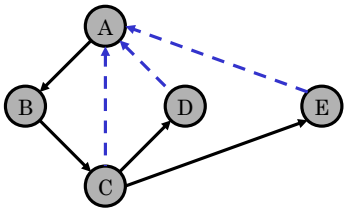
- Algorithm for DFS resembles a classical strategy for exploring a labyrinth
  - We mark each crossing and dead end we encounter (nodes)
  - We mark each corridor we walk through (edges)
  - We keep how to get back to the start node (recursion stack)



Properties

Property 1
DFS(G,v) visits all nodes and edges in the connected part of G that includes v

Property 2
The "discovery"-edges that are marked by a DFS(G,v) execution result in a spanning tree for the connected component of G containing v



Analysis of DFS

- Marking/checking marking of node/edge takes $O(1)$ time
- Each node is marked twice:
  - one time as *UNEXPLORED*
  - one time as *VISITED*
- Each edge is marked twice:
  - once as *UNEXPLORED*
  - once as *DISCOVERY* or *BACK*
- The method incidentEdges is called once for each node
- DFS is executed in time $O(n+m)$ given the graph is represented as an adjacent list
  - Recall $\sum_v deg(v) = 2m$

```
procedure DFS(G)
    for all u ∈ G.VERTICES() do
        SETLABEL(u, UNEXPLORED)
    for all e ∈ G.EDGES() do
        SETLABEL(e, UNEXPLORED)
    for all v ∈ G.VERTICES() do
        if GETLABEL(v) = UNEXPLORED then
            DFS(G, v)
```

```
procedure DFS(G, v)
    SETLABEL(v, VISITED)
    for all e ∈ G.INCIDENTEDGES(v) do
        if GETLABEL(e) = UNEXPLORED then
            w ← OPPOSITE(v, e)
            if GETLABEL(w) = UNEXPLORED then
                SETLABEL(e, DISCOVERY)
                DFS(G, w)
            else
                SETLABEL(e, BACK)
```


Find paths

- We can specialize DFS to find a path between two given nodes v and z
- We call DFS(G,v) with v as a start node
- We use a stack S to maintain the path from the start node to the current node
- As soon as we find the target node z, we return the content of the stack as the path

```
procedure PATHDFS(G, v, z)
    SETLABEL(v, VISITED)
    S.PUSH(v)
    if v = z //found path then
        print labels in S
    else
        for all e ∈ G.INCIDENTEDGES(v) do
            if GETLABEL(e) = UNEXPLORED then
                w ← OPPOSITE(v, e)
                if GETLABEL(w) = UNEXPLORED then
                    SETLABEL(e, DISCOVERY)
                    S.PUSH(e)
                    PATHDFS(G, w, z)
                    S.POP() // e
                else
                    SETLABEL(e, BACK)
    S.POP() // v
```


Find cycles

- We can specialize the DFS algorithm to find simple cycles
- We use a stack S to maintain a path to the start node from the current node
- As soon as we encounter an edge (v,w) that leads to an ancestor we return the content of the cycle from the stack

```
procedure CYCLEDFS(G, v, z)
    SETLABEL(v, VISITED)
    S.PUSH(v)
    for all e ∈ G.INCIDENTEDGES(v) do
        if GETLABEL(e) = UNEXPLORED then
            w ←OPPOSITE(v, e)
            S.PUSH(e)
            if GETLABEL(w) = UNEXPLORED then
                SETLABEL(e, DISCOVERY)
                CYCLEDFS(G, w)
            else // found cycle
                GETLABEL(w) = VISITED
                // w has to be in S
                print labels in S between w and v
            S.POP() // e
    S.POP() // v
```

20.29

2.2 BFS

BFS: Breadth first search

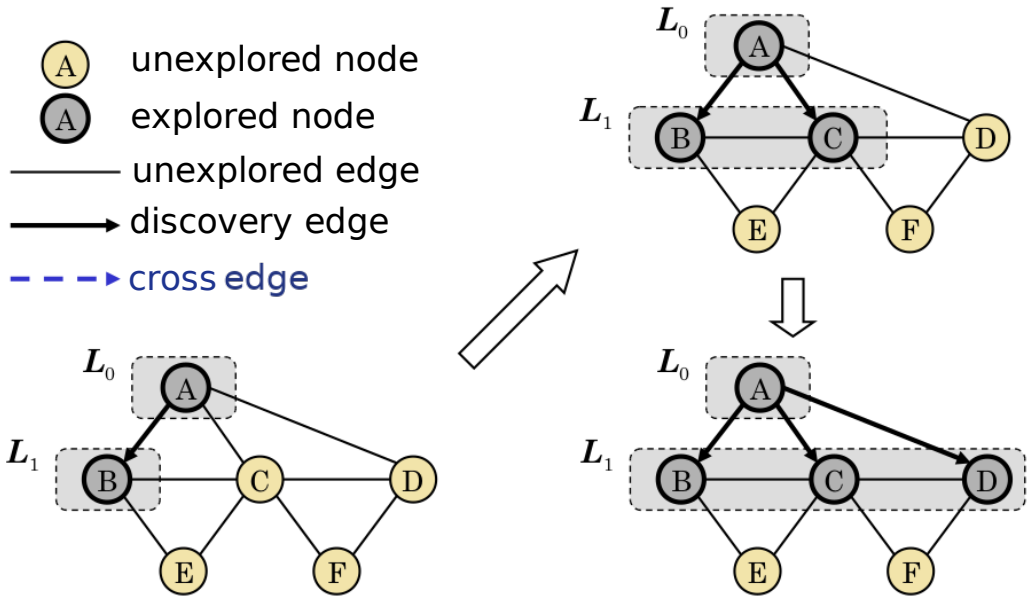
- BFS is a generic technique to traverse a graph
- BFS in a graph G
  - Visits all nodes and edges in G
  - Checks whether G is connected
  - Computes connected components in G
  - Computes a spanning forest in G
- BFS on a graph with n nodes and m edges takes $O(n+m)$ time
- BFS can be augmented to solve other graph problems:
  - Find and return a shortest path between two given nodes in a graph
  - Find a simple cycle in a graph, if any exists

20.30

Algorithm for BFS

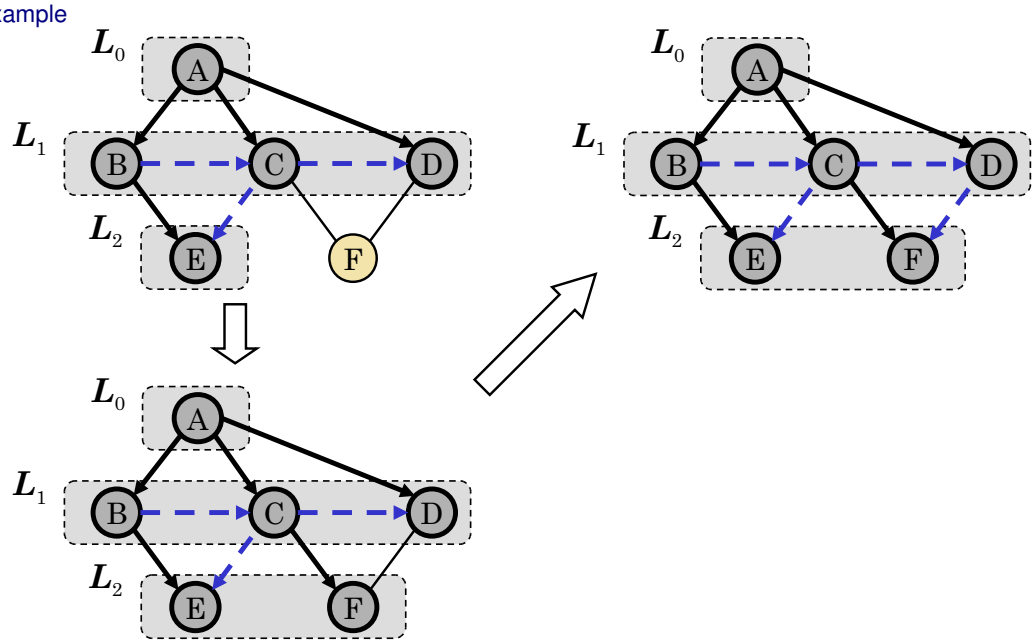
```
procedure BFS(G)
    mark all nodes/edges with UNEXPLORED
    for all v ∈ G.VERTICES() do
        if GETLABEL(v) = UNEXPLORED then BFS(G, v)
procedure BFS(G, s)
    L_0 ← new empty sequence; L_0.INSERTLAST(s); SETLABEL(s, VISITED); i ← 0
    while ¬L_i.ISEMPTY() do
        L_{i+1} ← new empty sequence
        for all v ∈ L_i.ELEMENTS() do
            for all e ∈ G.INCIDENTEDGES(v) do
                if GETLABEL(e) = UNEXPLORED then
                    w ←OPPOSITE(v, e)
                    if GETLABEL(w) = UNEXPLORED then
                        SETLABEL(e, DISCOVERY)
                        SETLABEL(w, VISITED)
                        L_{i+1}.INSERTLAST(w)
                    else
                        SETLABEL(e, CROSS)
        i ← i + 1
```

20.31



20.32

Example



20.33

Properties

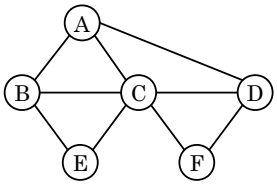
Let G_s be the connected component of G that includes s

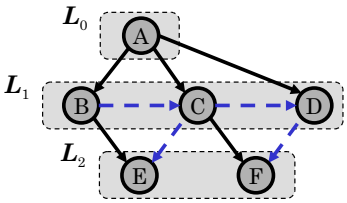
Property 1
BFS(G,s) visits all nodes and edges in G_s

Property 2
"discovery"-edges marked by BFS(G,s) are a spanning tree T_s for G_s

Property 3
For each node v in L_i:

- The path in T_s from s to v has i edges
- Any path from s to v in G_s has at least i edges





20.34

Analysis of BFS

- Mark/check marking of a node/edge takes $O(1)$ time
- Each node is marked twice
  - once as *UNEXPLORED*
  - once as *VISITED*
- Each edge is marked twice
  - once as *UNEXPLORED*
  - once as *DISCOVERY* or *CROSS*
- Each node is inserted once in a sequence L_i
- Method incidentEdges is called once for each node
- BFS executes in $O(n+m)$ time given the graph is represented with adjacent lists
  - Recall $\sum_v deg(v) = 2m$

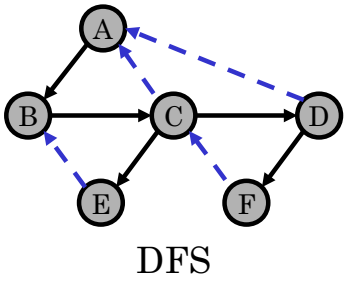
```
procedure BFS(G)
    mark all nodes/edges with UNEXPLORED
    for all v ∈ G.VERTICES() do
        if GETLABEL(v) = UNEXPLORED then
            BFS(G, v)
```

```
procedure BFS(G, s)
    L_0 ← new empty sequence
    L_0.INSERTLAST(s)
    SETLABEL(s, VISITED)
    i ← 0
    while ¬L_i.ISEMPTY() do
        L_{i+1} ← new empty sequence
        for all v ∈ L_i.ELEMENTS() do
            for all e ∈ G.INCIDENTEDGES(v) do
                if GETLABEL(e) = UNEXPLORED then
                    w ←OPPOSITE(v, e)
                    if GETLABEL(w) = UNEXPLORED then
                        SETLABEL(e, DISCOVERY)
                        SETLABEL(w, VISITED)
                        L_{i+1}.INSERTLAST(w)
                    else
                        SETLABEL(e, CROSS)
        i ← i + 1
```

20.35

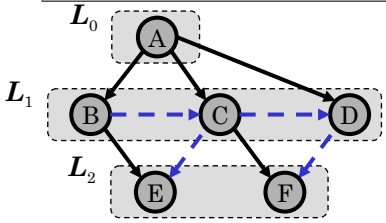
2.3 DFS vs BFS

Applications



DFS

Applications	DFS	BFS
Spaning tree, connected components, paths, cycles	√	√
Shortest path		√
2-connected components	√	



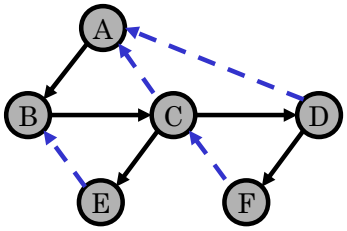
BFS

20.36

Edges to already visited nodes

edge to ancestor

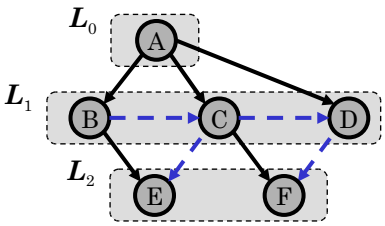
- w is an ancestor to v in the tree of "discovery"-edges



DFS

crossing edge

- w is in the same level as v or in the next level in the tree of ”discovery”-edges



BFS