## Föreläsning 20

Graphs and graph searches

## TDDD86: DALP

Utskriftsversion av Föreläsing i Datastrukturer, algoritmer och programmeringsparadigm 4 December 2023

IDA, Linköpings universitet

Content

## Contents

1 Graphs ..... 1
1.1 Introduction ..... 1
1.2 ADT graph ..... 3
1.3 Data structures ..... 4
2 Search in undirected graphs ..... 5
2.1 DFS ..... 5
2.2 BFS ..... 10
2.3 DFS vs BFS ..... 12

## 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


Types of edges

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

$$
\text { - unordered pair }\{u, v\}
$$

- In a directed graph, all edges are directed
- In an undirected graph, all edges are undirected


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


## 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)


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.



## Properties

Property 1
$\sum_{v} \operatorname{deg}(v)=2 m$ 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
- $\operatorname{deg}(v)$ is the degree of node $v$

Exempel
$n=4$
$m=6$
$\operatorname{deg}(v)=3$

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?

### 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 $x$


## 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


### 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



## 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


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


Asymptotic performance

| $\boldsymbol{n}$ noder, $\boldsymbol{m}$ bågar <br> inga parallella kanter <br> inga öglor | Båglista | Grannlista | Grann- <br> matris |
| :---: | :---: | :---: | :---: |
| minne | $O(\boldsymbol{n}+\boldsymbol{m})$ | $O(\boldsymbol{n}+\boldsymbol{m})$ | $O\left(\boldsymbol{n}^{2}\right)$ |
| incidentEdges $(\boldsymbol{v})$ | $O(\boldsymbol{m})$ | $O(\operatorname{deg}(\boldsymbol{v}))$ | $O(\boldsymbol{n})$ |
| areAdjacent $(\boldsymbol{v}, \boldsymbol{w})$ | $O(\boldsymbol{m})$ | $O(\min (\operatorname{deg}(\boldsymbol{v}), \operatorname{deg}(\boldsymbol{w}))$ | $O(1)$ |
| insertVertex $(\mathbf{o})$ | $O(1)$ | $O(1)$ | $O\left(\boldsymbol{n}^{2}\right)$ |
| insertEdge $(\boldsymbol{v}, \boldsymbol{w}, \mathbf{o})$ | $O(1)$ | $O(1)$ | $O(1)$ |
| removeVertex $(\boldsymbol{v})$ | $O(\boldsymbol{m})$ | $O(\operatorname{deg}(\boldsymbol{v}))$ | $O\left(\boldsymbol{n}^{2}\right)$ |
| removeEdge $(\boldsymbol{e})$ | $O(1)$ | $O(1)$ | $O(1)$ |

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

## 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

Connected components


63 sammanhängande komponenter

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




## 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 \(\operatorname{DFS}(G)\)
    for all \(u \in G\).VERTICES() do
        SETLABEL ( \(u, U N E X P L O R E D)\)
    for all \(e \in G\). \(\operatorname{EDGES}()\) do
        SETLABEL( \(e\), UNEXPLORED)
    for all \(v \in G\).VERTICES() do
        if \(\operatorname{GEtLabel}(v)=U N E X P L O R E D\) then
            \(\operatorname{DFS}(G, v)\)
    procedure \(\operatorname{DFS}(G, v)\)
    SETLABEL( \(v\), VISITED)
    for all \(e \in G\).IncidentEdges \((v)\) do
        if \(\operatorname{GETLABEL}(e)=U N E X P L O R E D\) then
            \(w \leftarrow \operatorname{OPPOSITE}(v, e)\)
            if \(\operatorname{GetLabel}(w)=U N E X P L O R E D\) then
                    SETLABEL \((e\), DISCOVERY)
                    \(\operatorname{DFS}(G, w)\)
                else
                    \(\operatorname{SETLABEL}(e, B A C K)\)
```



-     -         - back edge


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

$\operatorname{DFS}(G, v)$ visits all nodes and edges in the connected part of $G$ that includes $v$

## Property 2

The "discovery"-edges that are marked by a $\operatorname{DFS}(G, v)$ 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
procedure $\operatorname{DFS}(G)$
for all $u \in G$. VERTICES() do SETLABEL( $u, U N E X P L O R E D)$
for all $e \in G . \operatorname{EDGES}()$ do SETLABEL ( $e$, UNEXPLORED)
for all $v \in G$.VERTICES() do if $\operatorname{GEtLabel}(v)=U N E X P L O R E D$ then $\operatorname{DFS}(G, v)$
- Each edge is marked twice:
- once as UNEXPLORED
- once as DISCOVERY or BACK
- The method incidentEdges is called once for each node
procedure $\operatorname{DFS}(G, v)$ $\operatorname{SETLABEL}(v$,VISITED)
for all $e \in G$.IncidentEdges $(v)$ do if $\operatorname{GETLABEL}(e)=U N E X P L O R E D$ then
$w \leftarrow \operatorname{OPPositE}(v, e)$
if $\operatorname{GETLABEL}(w)=$ UNEXPLORED then SETLABEL (e,DISCOVERY) $\operatorname{DFS}(G, w)$
else
SETLABEL $(e, B A C K)$
$\qquad$

Find paths

- We can specialize DFS to find a path between two given nodes $v$ and $z$
- We call $\operatorname{DFS}(G, v)$ 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\inG.incidentEdges(v) do
            if GETLABEL}(e)=UNEXPLORED the
                w\leftarrow\operatorname{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 \(e \in G\).IncidentEdges \((v)\) do
        if \(\operatorname{GETLABEL}(e)=U N E X P L O R E D\) then
            \(w \leftarrow \operatorname{OPPOSITE}(v, e)\)
            \(S\).PUSH (e)
            if \(\operatorname{GETLABEL}(w)=U N E X P L O R E D\) then
                \(\operatorname{SETLABEL}(e, D I S C O V E R Y)\)
                CYCLEDFS \((G, w)\)
            else // found cycle
                \(\operatorname{GETLABEL}(w)=V I S I T E D\)
                    // \(w\) has to be in \(S\)
                print labels in \(S\) between \(w\) and \(v\)
            S.POP() //e
    \(S . \operatorname{POP}() / / v\)
```


### 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

```
Algorithm for BFS
    procedure BFS(G)
    mark all nodes/edges with UNEXPLORED
    for all v\inG.VERTICES() do
        if GETLABEL}(v)=UNEXPLORED then BFS (G,v
    procedure BFS(G,s)
        L
        while}\neg\mp@subsup{L}{i}{}\mathrm{ .ISEMPTY() do
            Li+1}\leftarrow\leftarrow\mathrm{ new empty sequence
            for all v\in Li}\mathrm{ .ELEMENTS() do
            for all }e\inG.INCIDENTEDGES(v) do
                    if GETLABEL}(e)=UNEXPLORED the
                    w\leftarrowOPPOSITE }(v,e
                        if GETLABEL}(w)=UNEXPLORED then
                                SETLABEL(e,DISCOVERY)
                                SETLABEL(w,VISITED)
                                Li+1.INSERTLAST(w)
                    else
                                    SETLABEL(e,CROSS)
            i\leftarrowi+1
```

(A) unexplored node
(A) explored node

- unexplored edge
$\longrightarrow$ discovery edge

-     -         - cross edge


Example


## Properties

Let $G_{s}$ be the connected component of $G$ that includes $s$

## Property 1

$\operatorname{BFS}(G, s)$ visits all nodes and edges in $G_{s}$

## Property 2

"discovery"-edges marked by $\operatorname{BFS}(G, s)$ 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



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

$$
-\operatorname{Recall} \sum_{v} \operatorname{deg}(v)=2 m
$$

```
procedure BFS(G)
    mark all nodes/edges with UNEXPLORED
    for all }v\inG.\mathrm{ vertices() do
        if GETLABEL}(v)=UNEXPLORED then
        BFS(G,v)
    procedure BFS(G,s)
    L
    Lo.INSERTLAST(s)
    SETLABEL(s,VISITED)
    SETLA
    while}\neg\mp@subsup{L}{i}{}\mathrm{ .ISEMPTY() do
    while}\neg\mp@subsup{L}{i}{}\mathrm{ .ISEMPTY() do
        Li+1}\leftarrow\leftarrow\mathrm{ new empty sequence
        for all v\inL L
            for all e\inG.incidentEdges(v) do
                if GETLABEL}(e)=UNEXPLORED then
                    w\leftarrow\operatorname{OPPOSITE}(v,e)
                    if GETLABEL}(w)=UNEXPLORED the
                        SETLAbel(e,DISCOVERY)
                        SETLABEL( }w,\mathrm{ VISITED)
                            Li+1}\cdot\mathrm{ INSERTLAST( }w
                            else
                            SETLABEL(e,CROSS
```

        \(\leftarrow i+1\)
    
### 2.3 DFS vs BFS

## Applications

| Applications | DFS | BFS |
| :---: | :---: | :---: |
| Spaning tree, connected components, paths, cycles | $\sqrt{ }$ | $\sqrt{ }$ |
| Shortest path |  | $\sqrt{ }$ |
| 2-connected components | $\checkmark$ |  |



DFS


BFS

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


