Introduction to graph theory and algorithms for sparse matrices

Bora Uçar

RO:MA, LIP, ENS Lyon, France

CR-08: Combinatorial scientific computing, September 2014
http://perso.ens-lyon.fr/bora.ucar/CR08/
1 Definitions and some problems

2 Basic algorithms
   - Breadth-first search
   - Depth-first search
   - Topological sort
   - Strongly connected components

3 Questions
A graph $G = (V, E)$ consists of a finite set $V$, called the vertex set and a finite, binary relation $E$ on $V$, called the edge set.

**Three standard graph models**

**Undirected graph:** The edges are unordered pair of vertices, i.e., \( \{u, v\} \in E \) for some $u, v \in V$.

**Directed graph:** The edges are ordered pair of vertices, that is, $(u, v)$ and $(v, u)$ are two different edges.

**Bipartite graph:** $G = (U \cup V, E)$ consists of two disjoint vertex sets $U$ and $V$ such that for each edge $(u, v) \in E$, $u \in U$ and $v \in V$.

An ordering or labelling of $G = (V, E)$ having $n$ vertices, i.e., $|V| = n$, is a mapping of $V$ onto $1, 2, \ldots, n$. 
Matrices and graphs: Rectangular matrices

The rows/columns and nonzeros of a given sparse matrix correspond (with natural labelling) to the vertices and edges, respectively, of a graph.

The set of rows corresponds to one of the vertex set $R$, the set of columns corresponds to the other vertex set $C$ such that for each $a_{ij} \neq 0$, $(r_i, c_j)$ is an edge.
Matrices and graphs: Square unsymmetric pattern

The rows/columns and nonzeros of a given sparse matrix correspond (with natural labelling) to the vertices and edges, respectively, of a graph.

Square unsymmetric pattern matrices

\[
A = \begin{pmatrix}
1 & 2 & 3 \\
1 & \times & \times \\
2 & \times & \times \\
3 & & \times \\
\end{pmatrix}
\]

Graph models

- Bipartite graph as before.
- Directed graph

The set of rows/cols corresponds the vertex set \( V \) such that for each \( a_{ij} \neq 0 \), \((v_i, v_j)\) is an edge. Transposed view possible too, i.e., the edge \((v_i, v_j)\) directed from column \( i \) to row \( j \). Usually self-loops are omitted.
Matrices and graphs: Square unsymmetric pattern

A special subclass
Directed acyclic graphs (DAG): A directed graphs with no loops (maybe except for self-loops).

DAGs
We can sort the vertices such that if \((u, v)\) is an edge, then \(u\) appears before \(v\) in the ordering.

Question: What kind of square matrices have a DAG?
Matrices and graphs: Symmetric pattern

The rows/columns and nonzeros of a given sparse matrix correspond (with natural labelling) to the vertices and edges, respectively, of a graph.

Square symmetric pattern matrices

\[
A = \begin{pmatrix}
1 & 2 & 3 \\
1 & \times & \\
2 & \times & \times & \times \\
3 & \times & \times
\end{pmatrix}
\]

Graph models

- Bipartite and directed graphs as before.
- Undirected graph

The set of rows/cols corresponds the vertex set \( V \) such that for each \( a_{ij}, a_{ji} \neq 0 \), \( \{v_i, v_j\} \) is an edge. No self-loops; usually the main diagonal is assumed to be zero-free.
Definitions: Edges, degrees, and paths

Many definitions for directed and undirected graphs are the same. We will use \((u, v)\) to refer to an edge of an undirected or directed graph to avoid repeated definitions.

- An edge \((u, v)\) is said to incident on the vertices \(u\) and \(v\).
- For any vertex \(u\), the set of vertices in \(\text{adj}(u) = \{v : (u, v) \in E\}\) are called the neighbors of \(u\). The vertices in \(\text{adj}(u)\) are said to be adjacent to \(u\).
- The degree of a vertex is the number of edges incident on it.
- A path \(p\) of length \(k\) is a sequence of vertices \(\langle v_0, v_1, \ldots, v_k \rangle\) where \((v_{i-1}, v_i) \in E\) for \(i = 1, \ldots, k\). The two end points \(v_0\) and \(v_k\) are said to be connected by the path \(p\), and the vertex \(v_k\) is said to be reachable from \(v_0\).
- Simple path: has no vertex repetitions.
Definitions: Components

- An undirected graph is said to be connected if every pair of vertices is connected by a path.

- The connected components of an undirected graph are the equivalence classes of vertices under the “is reachable” from relation.

- A directed graph is said to be strongly connected if every pair of vertices are reachable from each other.

- The strongly connected components of a directed graph are the equivalence classes of vertices under the “are mutually reachable” relation.
Definitions and some problems
Basic algorithms
Questions

Definitions: Trees and spanning trees

A tree is a connected, acyclic, undirected graph. If an undirected graph is acyclic but disconnected, then it is a forest.

Properties of trees

- Any two vertices are connected by a unique path.
- $|E| = |V| - 1$

A rooted tree is a tree with a distinguished vertex $r$, called the root.

There is a unique path from the root $r$ to every other vertex $v$. Any vertex $y$ in that path is called an ancestor of $v$. If $y$ is an ancestor of $v$, then $v$ is a descendant of $y$.

The subtree rooted at $v$ is the tree induced by the descendants of $v$, rooted at $v$.

A spanning tree of a connected graph $G = (V, E)$ is a tree $T = (V, F)$, such that $F \subseteq E$. 
Ordering of the vertices of a rooted tree

- A **topological ordering** of a rooted tree is an ordering that numbers children vertices before their parent.
- A **postorder** is a topological ordering which numbers the vertices in any subtree consecutively.

Connected graph G

 root: v
child: x, w
child of x: u
child of w: y, z
child of y: v
child of z: w

Rooted spanning tree with topological ordering

- Topological ordering: 1 3 4 5 6
- Postordering: u w x y z v

Rooted spanning tree with postordering

- Topological ordering: 1 3 4 5 6
- Postordering: u w x y z v
Postordering the vertices of a rooted tree – I

The following recursive algorithm will do the job:

\[[p\text{order}]]=[\text{PostOrder}(T, r)]
\quad \text{for each child } c \text{ of } r \text{ do}
\quad \quad p\text{order} \leftarrow [p\text{order}, \text{PostOrder}(T, c)]
\quad \quad p\text{order} \leftarrow [p\text{order}, r]

We need to run the algorithm for each root \( r \) when \( T \) is a forest.

Usually recursive algorithms are avoided, as for a tree with large number of vertices can cause stack overflow.
Postordering the vertices of a rooted tree – II

\[ \text{porder} = \text{PostOrder}(T, r) \]

\[
\begin{align*}
\text{porder} & \leftarrow [.] \\
\text{seen}(v) & \leftarrow \text{False} \text{ for all } v \in T \\
\text{seen}(r) & \leftarrow \text{True} \\
\text{Push}(S, r) \\
\text{while } \text{NotEmpty}(S) \text{ do} \\
& \quad v \leftarrow \text{Top}(S) \\
& \quad \text{if } \exists \text{ a child } c \text{ of } v \text{ with } \text{seen}(c) = \text{False} \text{ then} \\
& \quad \quad \text{seen}(c) \leftarrow \text{True} \\
& \quad \quad \text{Push}(S, c) \\
& \quad \text{else} \\
& \quad \quad \text{Pop}(S) \\
& \quad \quad \text{porder} \leftarrow [\text{porder}, v]
\end{align*}
\]

Again, have to run for each root, if \( T \) is a forest.

Both algorithms run in \( \mathcal{O}(n) \) time for a tree with \( n \) nodes.
A permutation matrix is a square \((0, 1)\)-matrix where each row and column has a single 1.

If \(P\) is a permutation matrix, \(PP^T = I\), i.e., it is an orthogonal matrix. Let,

\[
A = \begin{pmatrix}
1 & 2 & 3 \\
\times & \times & \\
2 & \times & \times \\
3 & \times & \times
\end{pmatrix}
\]

and suppose we want to permute columns as \([2, 3, 1]\). Define \(p_{2,1} = 1\), \(p_{3,2} = 1\), \(p_{1,3} = 1\), and \(B = AP\) (if column \(j\) to be at position \(i\), set \(p_{ji} = 1\))

\[
B = \begin{pmatrix}
2 & 3 & 1 \\
1 & \times & \times \\
2 & \times & \times \\
3 & \times & \times
\end{pmatrix} = \begin{pmatrix}
1 & 2 & 3 \\
\times & \times & \\
\times & \times & \\
\times & \times &
\end{pmatrix} \times \begin{pmatrix}
1 & 2 & 3 \\
1 & \times & \times \\
2 & \times & \times \\
3 & \times & \times
\end{pmatrix} = \begin{pmatrix}
1 & 2 & 3 \\
1 & 1 & 1 \\
2 & 1 & 1 \\
3 & 1 & 1
\end{pmatrix}
\]
Given a matrix \( A \), we want to permute the rows according to a given permutation. Let \( \pi(i) \) denote the new position of row \( i \). Write a simple for-loop to create the permutation matrix \( P \) so that the matrix \( PA \) is permuted according to \( \pi \). Verify your solution on the matrix in the previous slide (it is always good to verify, no?).
**Permutation matrices**

**Question**: For $P$ and $Q$ permutation matrices, what is the output matrix of the multiplication $PQ$?

**Question**: Let $A$ be a square matrix and $P$ be a permutation matrix. What is $PAP^T$?

**Question**: Let $A$ be a square matrix and $P$ be a permutation matrix. What can you say about the diagonal elements of $PAP^T$?

**Question**: Let $A$ be a square matrix and $P$ be a permutation matrix. What can you say about the graphs of $A$ and $PAP^T$?
A matching in a graph is a set of edges no two of which share a common vertex. We will be mostly dealing with matchings in bipartite graphs.

In matrix terms, a matching in the bipartite graph of a matrix corresponds to a set of nonzero entries no two of which are in the same row or column.

A vertex is said to be matched if there is an edge in the matching incident on the vertex, and to be unmatched otherwise. In a perfect matching, all vertices are matched.

The cardinality of a matching is the number of edges in it. A maximum cardinality matching or a maximum matching is a matching of maximum cardinality. Solvable in polynomial time.
Matching in bipartite graphs and permutations

Given a square matrix whose bipartite graph has a perfect matching, such a matching can be used to permute the matrix to have the matching entries are along the main diagonal.

\[
\begin{pmatrix}
1 & 2 & 3 \\
\times & \cdot & \times \\
2 & \times & \cdot \\
3 & \cdot & \times \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
2 & 3 & 1 \\
\cdot & \times & \times \\
\times & \cdot & \times \\
\times & \cdot & \times \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 2 & 3 \\
\times & \cdot & \times \\
2 & \times & \cdot \\
3 & \cdot & \times \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 2 & 3 \\
\times & \cdot & \times \\
2 & \times & \cdot \\
3 & \cdot & \times \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 2 & 3 \\
\times & \cdot & \times \\
2 & \times & \cdot \\
3 & \cdot & \times \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 2 & 3 \\
\times & \cdot & \times \\
2 & \times & \cdot \\
3 & \cdot & \times \\
\end{pmatrix}
\]
Let $A$ be an $n \times n$ sparse matrix having a perfect matching in its bipartite graph. Let $M$ be such a matching, where $\text{colmatch}(j)=i$ show that the column $j$ is matched to row $i$ in $M$.

Write a pseudocode to create a permutation matrix $Q$ such that $AQ$ has the matching entries along its diagonal.

Write a pseudocode to create a permutation matrix $P$ such that $PA$ has the matching entries along its diagonal.
Reducible matrix: An \( n \times n \) square matrix \( A \) is reducible if there exists an \( n \times n \) permutation matrix \( P \) such that

\[
PAP^T = \begin{pmatrix} A_{11} & A_{12} \\ O & A_{22} \end{pmatrix},
\]

where \( A_{11} \) is an \( r \times r \) submatrix, \( A_{22} \) is an \( (n - r) \times (n - r) \) submatrix, where \( 1 \leq r < n \).

Irreducible matrix: There is no such a permutation matrix.

Theorem: An \( n \times n \) square matrix is irreducible iff its directed graph is strongly connected.

Proof: Follows by definition.
Why care irreducibility

Definition 1.15. For $n \geq 2$, an $n \times n$ complex matrix $A$ is reducible if there exists an $n \times n$ permutation matrix $P$ such that

$$PAP^T = \begin{bmatrix} A_{1,1} & A_{1,2} \\ O & A_{2,2} \end{bmatrix},$$

where $A_{1,1}$ is an $r \times r$ submatrix and $A_{2,2}$ is an $(n-r) \times (n-r)$ submatrix, where $1 \leq r < n$. If no such permutation matrix exists, then $A$ is irreducible. If $A$ is a $1 \times 1$ complex matrix, then $A$ is irreducible if its single entry is nonzero, and reducible otherwise.

It is evident from Definition 1.15 that the matrix of (1.38) is reducible.

The term irreducible (unzerlegbar) was introduced by Frobenius (1912); it is also called un-reduced and indecomposable in the literature. The motivation for calling matrices such as those in (1.39) reducible is quite clear, for if we seek to solve the matrix equation $\tilde{A}x = k$, where $\tilde{A} := PAP^T$ is the partitioned matrix of (1.39), we can partition the vectors $x$ and $k$ similarly so that the matrix equation $\tilde{A}x := k$ can be written as

$$A_{1,1}x_1 + A_{1,2}x_2 = k_1,$$
$$A_{2,2}x_2 = k_2.$$

Thus, by solving the second equation for $x_2$, and with this known solution for $x_2$, solving the first equation for $x_1$, we have reduced the solution of the original matrix equation to the solution of two lower-order matrix equations.

The matrix $A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ is irreducible (we just do not have many alternatives to permute it symmetrically and obtain the required form).

Yet, clearly it is not hard to solve $Ax = b$; the solution is $x_2 = b_1$ and $x_1 = b_2$.

Here comes something more general than irreducibility.
Definitions: Fully indecomposability

**Fully indecomposable matrix:** There is no permutation matrices $P$ and $Q$ such that

$$PAQ = \begin{pmatrix} A_{11} & A_{12} \\ O & A_{22} \end{pmatrix},$$

with the same condition on the blocks and their sizes as above.

**Theorem:** An $n \times n$ square matrix $A$ is fully indecomposable iff for some permutation matrix $Q$, the matrix $AQ$ is irreducible and has a zero-free main diagonal.

**Proof:** We will come later in the semester to the “if” part.

Only if part (by contradiction): Let $B = AQ$ be an irreducible matrix with zero-free main diagonal. $B$ is fully indecomposable iff $A$ is (why?). Therefore we may assume that $A$ is irreducible and has a zero-free diagonal. Suppose, for the sake of contradiction, $A$ is not fully indecomposable.
Fully indecomposable matrices

There is no permutation matrices $P$ and $Q$ such that

$$PAQ = \begin{pmatrix} A_{11} & A_{12} \\ O & A_{22} \end{pmatrix},$$

with the same condition on the blocks and their sizes as above.

Proof cont.: Let $P_1AQ_1$ be of the form above with $A_{11}$ of size $r \times r$. We may write $P_1AQ_1 = A'Q'$, where $A' = P_1AP_1^T$ with zero-free diagonal (why?), and $Q' = P_1Q_1$ is a permutation matrix which has to permute (why?) the first $r$ columns (of $A'$) among themselves, and similarly the last $n - r$ columns among themselves. Hence, $A'$ is in the above form, and $A$ is reducible: contradiction. \qed
## Definitions: Cliques and independent sets

### Clique

In an undirected graph \( G = (V, E) \), a set of vertices \( S \subseteq V \) is a clique if for all \( s, t \in S \), we have \((s, t) \in E\).

**Maximum clique**: A clique of maximum cardinality (finding a maximum clique in an undirected graph is \( \mathsf{NP}\)-complete).

**Maximal clique**: A clique is a maximal clique, if it is not contained in another clique.

In a symmetric matrix \( A \), a clique corresponds to a subset of rows \( R \) and the corresponding columns such that the matrix \( A(R, R) \) is full.

### Independent set

A set of vertices is an **independent set** if none of the vertices are adjacent to each other. Can we find the largest one in polynomial time?

In a symmetric matrix \( A \), an independent set corresponds to a subset of rows \( R \) and the corresponding columns such that the matrix \( A(R, R) \) is either zero, or diagonal.
Definitions: More on cliques

Clique: In an undirected graph $G = (V, E)$, a set of vertices $S \subseteq V$ is a clique if for all $s, t \in S$, we have $(s, t) \in E$.

In a symmetric matrix $A$ corresponds to a subset of rows $R$ and the corresponding columns such that the matrix $A(R, R)$ is full.

Clique in bipartite graphs: Bi-cliques

In a bipartite graph $G = (U \cup V, E)$, a pair of sets $\langle R, C \rangle$ where $R \subseteq U$ and $C \subseteq V$ is a bi-clique if for all $a \in R$ and $b \in C$, we have $(a, b) \in E$.

In a matrix $A$, corresponds to a subset of rows $R$ and a subset of columns $C$ such that the matrix $A(R, C)$ is full.

The maximum node bi-clique problem asks for a bi-clique of maximum size (e.g., $|R| + |C|$), and it is polynomial time solvable, whereas the maximum edge bi-clique problem (e.g., asks for a maximum $|R| \times |C|$) is NP-complete.
**Definitions: Hypergraphs**

**Hypergraph**: A hypergraph $H = (V, N)$ consists of a finite set $V$ called the vertex set and a set of non-empty subsets of vertices $N$ called the hyperedge set or the net set. A generalization of graphs.

For a matrix $A$, define a hypergraph whose vertices correspond to the rows and whose nets correspond to the columns such that vertex $v_i$ is in net $n_j$ iff $a_{ij} \neq 0$ (the column-net model).

A sample matrix

$$
\begin{pmatrix}
1 & 2 & 3 & 4 \\
1 & \times & \times & \times \\
2 & \times & \times \\
3 & \times & \times & \times
\end{pmatrix}
$$

The column-net hypergraph model
Basic graph algorithms

Searching a graph: Systematically following the edges of the graph so as to visit all the vertices.
- Breadth-first search,
- Depth-first search.

Topological sort (of a directed acyclic graph): It is a linear ordering of all the vertices such that if \((u, v)\) directed is an edge, then \(u\) appears before \(v\) in the ordering.

Strongly connected components (of a directed graph; why?): Recall that a strongly connected component is a maximal set of vertices for which every pair its vertices are reachable. We want to find them all.

We will use some of the course notes by Cevdet Aykanat (http://www.cs.bilkent.edu.tr/~aykanat/teaching.html)
Breadth-first search: Idea

Graph $G = (V, E)$, directed or undirected with adjacency list representation.

**GOAL:** Systematically explores edges of $G$ to
- discover every vertex reachable from the source vertex $s$
- compute the shortest path distance of every vertex from the source vertex $s$
- produce a breadth-first tree (BFT) $G_\Pi$ with root $s$
  - BFT contains all vertices reachable from $s$
  - the unique path from any vertex $v$ to $s$ in $G_\Pi$ constitutes a shortest path from $s$ to $v$ in $G$

**IDEA:** Expanding frontier across the breadth-greedy-
- propagate a wave 1 edge-distance at a time
- using a FIFO queue: $O(1)$ time to update pointers to both ends
Maintains the following fields for each \( u \in V \)
- \( \text{color}[u] \): color of \( u \)
  - \( \text{WHITE} \): not discovered yet
  - \( \text{GRAY} \): discovered and to be or being processed
  - \( \text{BLACK} \): discovered and processed
- \( \Pi[u] \): parent of \( u \) (NIL of \( u = s \) or \( u \) is not discovered yet)
- \( d[u] \): distance of \( u \) from \( s \)

Processing a vertex = scanning its adjacency list
Breadth-first search: Algorithm

\[\text{BFS}(G, s)\]

\[
\begin{align*}
&\text{for each } u \in V - \{s\} \text{ do} \\
&\quad \text{color}[u] \leftarrow \text{WHITE} \\
&\quad \Pi[u] \leftarrow \text{NIL}; d[u] \leftarrow \infty \\
&\quad \text{color}[s] \leftarrow \text{GRAY} \\
&\quad \Pi[s] \leftarrow \text{NIL}; d[s] \leftarrow 0 \\
&\quad Q \leftarrow \{s\} \\
&\quad \text{while } Q \neq \emptyset \text{ do} \\
&\quad \quad u \leftarrow \text{head}[Q] \\
&\quad \quad \text{for each } v \text{ in } \text{Adj}[u] \text{ do} \\
&\quad \quad \quad \text{if } \text{color}[v] = \text{WHITE} \text{ then} \\
&\quad \quad \quad \quad \text{color}[v] \leftarrow \text{GRAY} \\
&\quad \quad \quad \Pi[v] \leftarrow u \\
&\quad \quad \quad d[v] \leftarrow d[u] + 1 \\
&\quad \quad \quad \text{ENQUEUE}(Q, v) \\
&\quad \quad \text{DEQUEUE}(Q) \\
&\quad \quad \text{color}[u] \leftarrow \text{BLACK}
\end{align*}
\]
Breadth-first search: Example

Sample Graph:

- **FIFO** just after queue \( Q \), processing vertex
- \( \langle a \rangle \)
Breadth-first search: Example

FIFO queue \(Q\) just after processing vertex

\[\langle a \rangle, \langle a, b, c \rangle, a\]
Breadth-first search: Example

- **FIFO** queue $Q$ just after processing vertex
- $\langle a \rangle$, $\langle a,b,c \rangle$, $\langle a,b,c,f \rangle$
- $a$, $b$
Breadth-first search: Example

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the root node (or an arbitrary node of a graph, sometimes referred to as a 'source') and explores the neighbor nodes first, before moving to the next level neighbors.

### BFS Example

Consider the graph below with source node `s`:

- **Vertices:** `a, b, c, d, e, f, g, h, i`
- **Edges:** Directed edges between the vertices

#### Initial State
- **Source:** `s`
- **Queue:** Initially empty

#### Process:

1. **Step 1:**
   - `s` is dequeued and marked as processed.
   - `Q` is updated to `[a]`.
   - Process `a`.

2. **Step 2:**
   - `a` is dequeued and marked as processed.
   - `Q` is updated to `[b, c]`.
   - Process `a`.

3. **Step 3:**
   - `b` is dequeued and marked as processed.
   - `Q` is updated to `[c, d]`.
   - Process `b`.

4. **Step 4:**
   - `c` is dequeued and marked as processed.
   - `Q` is updated to `[d, e]`.
   - Process `c`.

5. **Step 5:**
   - `d` is dequeued and marked as processed.
   - `Q` is updated to `[e, f]`.
   - Process `d`.

6. **Step 6:**
   - `e` is dequeued and marked as processed.
   - `Q` is updated to `[f, i]`.
   - Process `e`.

7. **Step 7:**
   - `f` is dequeued and marked as processed.
   - `Q` is updated to `[i, h]`.
   - Process `f`.

8. **Step 8:**
   - `i` is dequeued and marked as processed.
   - `Q` is updated to `[h]`.
   - Process `i`.

9. **Step 9:**
   - `h` is dequeued and marked as processed.
   - `Q` is updated to `[ ]`.
   - Process `h`.

#### BFS Process:

- **Queue Q:** `[a, b, c, d, e, f, i, h]`
- **Processing order:** `s, a, b, c, d, e, f, i, h`

#### Final State:

- All vertices are processed.
- The graph is scanned in a breadth-first manner.

### FIFO after processing a vertex

- `a`: `a`
- `a,b,c`: `a`
- `a,b,c,f`: `b`
- `a,b,c,f,e`: `c`
Breadth-first search: Example

FIFO just after queue $Q$ processing vertex

\[
\begin{align*}
\langle a \rangle &\quad - \\
\langle a, b, c \rangle &\quad a \\
\langle a, b, c, f \rangle &\quad b \\
\langle a, b, c, f, e \rangle &\quad c \\
\langle a, b, c, f, e, g, h \rangle &\quad f
\end{align*}
\]
Breadth-first search: Example

FIFO just after queue $Q$ processing vertex

$\langle a \rangle$ -
$\langle a, b, c \rangle$ a
$\langle a, b, c, f \rangle$ b
$\langle a, b, c, f, e \rangle$ c
$\langle a, b, c, f, e, g, h \rangle$ f
$\langle a, b, c, f, e, g, h, d, i \rangle$ e

all distances are filled in after processing e
Breadth-first search: Example

FIFO just after queue $Q$ processing vertex

$\langle a \rangle$ -
$\langle a,b,c \rangle$ a
$\langle a,b,c,f \rangle$ b
$\langle a,b,c,f,e \rangle$ c
$\langle a,b,c,f,e,g,h \rangle$ f
$\langle a,b,c,f,e,g,h,d,i \rangle$ g
Breadth-first search: Example

FIFO just after queue Q processing vertex

\[
\begin{align*}
\langle a \rangle & \quad - \\
\langle a,b,c \rangle & \quad a \\
\langle a,b,c,f \rangle & \quad b \\
\langle a,b,c,f,e \rangle & \quad c \\
\langle a,b,c,f,e,g,h \rangle & \quad f \\
\langle a,b,c,f,e,g,h,d,i \rangle & \quad h
\end{align*}
\]
Breadth-first search: Example

FIFO just after queue $Q$ processing vertex

\[
\begin{aligned}
\langle a \rangle & \quad - \\
\langle a,b,c \rangle & \quad a \\
\langle a,b,c,f \rangle & \quad b \\
\langle a,b,c,f,e \rangle & \quad c \\
\langle a,b,c,f,e,g,h \rangle & \quad f \\
\langle a,b,c,f,e,g,h,d,i \rangle & \quad d
\end{aligned}
\]
Breadth-first search: Example

Algorithm terminates: all vertices are processed
Breadth-first search: Analysis

Running time: \( O(V+E) = \) considered linear time in graphs

- initialization: \( \Theta(V) \)
- queue operations: \( O(V) \)
  - each vertex enqueued and dequeued at most once
  - both enqueue and dequeue operations take \( O(1) \) time
- processing gray vertices: \( O(E) \)
  - each vertex is processed at most once and
    \[ \sum_{u \in V} |Adj[u]| = \Theta(E) \]
Definitions and some problems
Basic algorithms
Questions

Breadth-first search: The paths to the root

\[ \text{BFS}(G, s) \text{, where } V_\Pi = \{ \nu \in V : \Pi[\nu] \neq \text{NIL} \} \cup \{s\} \text{ and } \]
\[ E_\Pi = \{ (\Pi[\nu], \nu) \in E : \nu \in V_\Pi - \{s\} \} \]
is a breadth-first tree such that

- \( V_\Pi \) consists of all vertices in \( V \) that are reachable from \( s \)
- \( \forall \nu \in V_\Pi \), unique path \( p(\nu, s) \) in \( G_\Pi \) constitutes a \( \text{sp}(s, \nu) \) in \( G \)

**PRINT-PATH**(G, s, \( \nu \))

- if \( \nu = s \) then print \( s \)
- else if \( \Pi[\nu] = \text{NIL} \) then
  - print no “\( s \rightarrow \nu \) path”
- else
  - **PRINT-PATH**(G, s, \( \Pi[\nu] \))
  - print \( \nu \)

Prints out vertices on a \( s \rightarrow \nu \) shortest path
Breadth-first search: The BFS tree

Breadth-First Tree Generated by BFS

BFS(G, a) terminated

BFT generated by BFS(G, a)
Breadth-first search: SpMxV and BFS

Write BFS starting from a vertex $i$ using SpMxV.
Depth-first search: Idea

- Graph $G=(V,E)$ directed or undirected
- Adjacency list representation
- **Goal**: Systematically explore every vertex and every edge
- **Idea**: search deeper whenever possible
  - Using a LIFO queue (Stack; FIFO queue used in BFS)
Depth-first search: Key components

- Maintains several fields for each $v \in V$
- Like BFS, colors the vertices to indicate their states. Each vertex is
  - Initially white,
  - grayed when discovered,
  - blackened when finished
- Like BFS, records discovery of a white $v$ during scanning $\text{Adj}[u]$ by $\pi[v] \leftarrow u$
Depth-first search: Key components

- Unlike BFS, predecessor graph $G_{\pi}$ produced by DFS forms spanning forest
- $G_{\pi} = (V, E_{\pi})$ where
  
  $$E_{\pi} = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\}$$
- $G_{\pi}$ = depth-first forest (DFF) is composed of disjoint depth-first trees (DFTs)
Depth-first search: Key components

- DFS also timestamps each vertex with two timestamps
- $d[v]$ records when $v$ is first discovered and grayed
- $f[v]$ records when $v$ is finished and blackened
- Since there is only one discovery event and finishing event for each vertex we have $1 \leq d[v] < f[v] \leq 2|V|$

<table>
<thead>
<tr>
<th>Time axis for the color of a vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image%E9%B9%AFimage" alt="Time axis diagram" /></td>
</tr>
</tbody>
</table>
Depth-first search: Algorithm

**DFS(G)**

for each \( u \in V \) do

\[ \text{color}[u] \leftarrow \text{white} \]

\[ \pi[u] \leftarrow \text{NIL} \]

\[ \text{time} \leftarrow 0 \]

for each \( u \in V \) do

if \( \text{color}[u] = \text{white} \) then

DFS-VISIT(G, u)

**DFS-VISIT(G, u)**

\[ \text{color}[u] \leftarrow \text{gray} \]

\[ d[u] \leftarrow \text{time} \leftarrow \text{time} + 1 \]

for each \( v \in \text{Adj}[u] \) do

if \( \text{color}[v] = \text{white} \) then

\[ \pi[v] \leftarrow u \]

DFS-VISIT(G, v)

\[ \text{color}[u] \leftarrow \text{black} \]

\[ f[u] \leftarrow \text{time} \leftarrow \text{time} + 1 \]
Depth-first search: Analysis

- Running time: $\Theta(V+E)$
- Initialization loop in DFS: $\Theta(V)$
- Main loop in DFS: $\Theta(V)$ exclusive of time to execute calls to DFS-Visit
  - DFS-Visit is called exactly once for each $v \in V$ since
    - DFS-Visit is invoked only on white vertices and
    - DFS-Visit($G, u$) immediately colors $u$ as gray
  - For loop of DFS-Visit($G, u$) is executed $|\text{Adj}[u]|$ time
  - Since $\Sigma |\text{Adj}[u]| = E$, total cost of executing loop of DFS-Visit is $\Theta(E)$
Depth-first search: Example

A directed graph with labeled nodes and arrows indicating the direction of edges. The nodes are labeled with letters (S, x, y, z, w, v, t, u). The arrows show the flow from one node to another. The depth-first search algorithm is illustrated by visiting nodes in a specific order, typically starting from a chosen node (such as S) and exploring as far as possible along each branch before backtracking.
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example

[Diagram of a directed graph showing nodes labeled S, W, X, Y, Z, 1, 2, 3, 4, 5, and their connections. The nodes are shaded and colored to illustrate the depth-first search process.]

Definitions and some problems
Basic algorithms
Depth-first search
Topological sort
Strongly connected components
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example

Graph representation of depth-first search.
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example

A directed graph with nodes labeled from 1 to 12, and edges connecting them. The graph shows the flow of a depth-first search algorithm, starting from node S (2, 7) and proceeding through the nodes in a specific order. The nodes are colored to indicate the order of exploration, with black indicating visited nodes and white indicating not visited yet.
Depth-first search: Example

![Graph Example](image-url)
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example
Depth-first search: Example
Depth-first search: DFT and DFF

**DFS(G) terminated**

**Depth-first forest (DFF)**
Depth-first search: Parenthesis theorem

**Thm:** In any DFS of $G=(V,E)$, let $\text{int}[v] = [d[v], f[v]]$ then exactly one of the following holds for any $u$ and $v \in V$

- $\text{int}[u]$ and $\text{int}[v]$ are entirely disjoint
- $\text{int}[v]$ is entirely contained in $\text{int}[u]$ and $v$ is a descendant of $u$ in a DFT
- $\text{int}[u]$ is entirely contained in $\text{int}[v]$ and $u$ is a descendant of $v$ in a DFT
Depth-first search: Parenthesis theorem

Parenthesis Theorem

\((x \ (s \ (w \ w) \ (v \ v) \ s) \ (y \ (t \ t) \ y) \ x) \ (z \ (u \ u) \ z)\)
Depth-first search: Edge classification

Tree Edge: discover a new (WHITE) vertex

▷ GRAY to WHITE ◀

Back Edge: from a descendant to an ancestor in DFT

▷ GRAY to GRAY ◀

Forward Edge: from ancestor to descendant in DFT

▷ GRAY to BLACK ◀

Cross Edge: remaining edges (btwn trees and subtrees)

▷ GRAY to BLACK ◀

Note: ancestor/descendent is wrt Tree Edges
Depth-first search: Edge classification

- How to decide which GRAY to BLACK edges are forward, which are cross

Let BLACK vertex $v \in \text{Adj}[u]$ is encountered while processing GRAY vertex $u$
- $(u,v)$ is a forward edge if $d[u] < d[v]$
- $(u,v)$ is a cross edge if $d[u] > d[v]$
Depth-first search: Edge classification example

The diagram illustrates a graph with nodes labeled from 1 to 7, and directed edges connecting these nodes. The nodes are colored to indicate different classifications during the depth-first search process.
Depth-first search: Edge classification example
Depth-first search: Edge classification example
Depth-first search: Edge classification example
Depth-first search: Edge classification example

Depth-first search:

Example

x y z
s t
wv u
1
2
3 4
T T B
Depth-first search: Edge classification example
Depth-first search: Edge classification example

Graph:
- Nodes: s, t, w, v, u, x, y, z
- Edges: s→t, t→w, w→v, v→u, u→z, z→x, x→y, y→z

Node States:
- s, t: White
- w, v: Gray
- u, z: Black

Edge Classification:
- T: Forward
- B: Back
- C: Cross
- S: None
Depth-first search: Edge classification example
Depth-first search: Edge classification example
Depth-first search: Edge classification example
Depth-first search: Edge classification example

Diagram of a graph with nodes labeled from 1 to 8 and edges labeled with T and B. The graph is annotated with edge classifications such as T and B, indicating the type of edge during the depth-first search.
Depth-first search: Edge classification example
Depth-first search: Edge classification example
Depth-first search: Edge classification example
Depth-first search: Edge classification example

Graph with vertices labeled and edges marked with labels indicating classification.
Depth-first search: Edge classification example
Depth-first search: Edge classification example
Depth-first search: Edge classification example
Depth-first search: Edge classification example

Definitions and some problems
Basic algorithms
Questions

Breadth-first search
Depth-first search
Topological sort
Strongly connected components

Depth-First Search: Example

```
x y z
s t
wv u
2
34 56
7 9 10
8 11 12
13
```

Depth-First Search: Edge classification example
Depth-first search: Edge classification example

- Nodes: S, T, X, Y, Z, W, V, U
- Edges labeled with T, F, B, C
- Example graph with depth-first search classification
Depth-first search: Edge classification example
Depth-first search: Edge classification example
Depth-first search: Undirected graphs

Edge classification

Any DFS on an undirected graph produces only **Tree** and **Back** edges.
Depth-first search: Non-recursive algorithm

\[
[\pi, d, f] = \text{DFS}(G, v)
\]

\[\text{top} \leftarrow 1\]
\[\text{stack}(\text{top}) \leftarrow v\]
\[d(v) \leftarrow \text{ctime} \leftarrow 1\]

while \( \text{top} > 0 \) do

\[u \leftarrow \text{stack}(\text{top})\]

if there is a vertex \( w \in \text{Adj}(u) \) where \( \pi(w) \) is not set then

\[\text{top} \leftarrow \text{top} + 1\]
\[\text{stack}(\text{top}) \leftarrow w\]
\[\pi(w) \leftarrow u\]
\[d(w) \leftarrow \text{ctime} \leftarrow \text{ctime} + 1\]

else

\[f(u) \leftarrow \text{ctime} \leftarrow \text{ctime} + 1\]

\[\text{top} \leftarrow \text{top} - 1\]
Topological sort (of a directed acyclic graph): It is a linear ordering of all the vertices such that if \((u, v)\) is a directed edge, then \(u\) appears before \(v\) in the ordering.

Ordering is not necessarily unique.
Topological sort: Example

- under short
- socks
- watch
- pants
- shoes
- shirt
- belt
- tie
- jacket

11/16
12/15
13/14
9/10
1/8
6/7
2/5
3/4
Topological sort: Algorithm

The algorithm

- run $\text{DFS}(G)$
- when a vertex is finished, output it
- vertices are output in the reverse topologically sorted order

Runs in $O(V + E)$ time — a linear time algorithm.

The algorithm: Correctness

If $(u, v) \in E$, then $f[u] > f[v]$

Proof: Consider the color of $v$ during exploring the edge $(u, v)$, where $u$ is Gray.

$v$ cannot be Gray (otherwise a Back edge in an acyclic graph !!!).

If $v$ is White, then $u$ is an ancestor of $v$, hence $f[u] > f[v]$.

If $v$ is Black, $f[v]$ is computed already, $f[u]$ is going to be computed, hence $f[u] > f[v]$. □
Transitive closure of a directed graph

The transitive closure of a digraph $G = (V, E)$ is the graph $G' = (V, E')$ where $E' = \{(j, i) : \text{there is a } j \rightarrow i \text{ path in } G\}$.

The digraph $G = (V, E)$ is transitively closed if it is equal to its own transitive closure.
The **strongly connected components** of a directed graph are the equivalence classes of vertices under the “are mutually reachable” relation.

For a graph $G = (V, E)$, the transpose is defined as $G^T = (V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$.

Constructing $G^T$ from $G$ takes $O(V + E)$ time with adjacency list (like the CSR or CSC storage format for sparse matrices) representation.

Notice that $G$ and $G^T$ have the same SCCs.
Strongly connected components: Algorithm

1. Run $\text{DFS}(G)$ to compute finishing times for all $u \in V$
2. Compute $G^T$
3. Call $\text{DFS}(G^T)$ processing vertices in main loop in decreasing $f[u]$ computed in Step (1)
4. Output vertices of each $\text{DFT}$ in $\text{DFF}$ of Step (3) as a separate $\text{SCC}$
Lemma 1: no path between a pair of vertices in the same SCC, ever leaves the SCC

Proof: let \( u \) and \( v \) be in the same SCC \( \Rightarrow u \leftrightarrow v \)

let \( w \) be on some path \( u \rightarrow w \rightarrow v \Rightarrow u \rightarrow w \)

but \( v \rightarrow u \Rightarrow \exists \) a path \( w \rightarrow v \rightarrow u \Rightarrow w \rightarrow u \)

therefore \( u \) and \( w \) are in the same SCC

QED
Strongly connected components: Example

CS473 – Lecture 16 Cevdet Aykanat - Bilkent University
Computer Engineering Department
5
SCC: Example
a b c
e
d
f g h
(1) Run \texttt{DFS}(G) to compute finishing times for all \( u \in V \)
(1) Run **DFS(G)** to compute finishing times for all $u \in V$
(1) Run DFS(G) to compute finishing times for all \( u \in V \)

```
SCC: Example
a b c
e d f g h
1 2 10
2 7 3 4 5 6
8 9 11
```

Diagram:

- Vertices: a, b, c, d, e, f, g, h
- Edges: a to b, b to c, c to d, d to g, g to h, h to e, e to f, f to 3, 3 to 4, 4 to 11, 11 to 10, 10 to 1, 1 to 2, 2 to 7, 7 to 5, 5 to 6

- Vertices 1, 2, 3, 4, 5, 6, 7, 10 are in one strongly connected component (SCC).
- Vertices 8, 9, 11 are in another SCC.
- Vertices a, b, c, d, e, f, g, h are in a third SCC.
Strongly connected components: Example

Vertices sorted according to the finishing times:

\[\langle b, e, a, c, d, g, h, f \rangle\]
Strongly connected components: Example

(2) Compute $G^T$
(3) Call **DFS**\((G^T)\) processing vertices in main loop in decreasing **f**[**u**] order: \(\langle b, e, a, c, d, g, h, f \rangle\)
(3) Call $\text{DFS}(G^T)$ processing vertices in main loop in decreasing $f[u]$ order: $\langle b, e, a, c, d, g, h, f \rangle$
(3) Call $\text{DFS}(G^T)$ processing vertices in main loop in decreasing $f[u]$ order: $\langle b, e, a, c, d, g, h, f \rangle$
(3) Call $\text{DFS}(G^T)$ processing vertices in main loop in decreasing $f[u]$ order: $\langle b, e, a, c, d, g, h, f \rangle$
(3) Call **DFS**\((G^T)\) processing vertices in main loop in decreasing \(f[u]\) order: \([b, e, a, c, d, g, h, f]\)
Strongly connected components: Example

(3) Call $\text{DFS}(G^T)$ processing vertices in main loop in decreasing $f[u]$ order: $\langle b, e, a, c, d, g, h, f \rangle$
(3) Call \textbf{DFS}(G^T) \text{ processing vertices in main loop in decreasing } f[u] \text{ order: } \langle b, e, a, c, d, g, h, f \rangle
(3) Call $\text{DFS}(G^T)$ processing vertices in main loop in decreasing $f[u]$ order: $\langle b, e, a, c, d, g, h, f \rangle$
Strongly connected components: Example

(4) Output vertices of each DFT in DFF as a separate SCC

\[ C_b = \{b, a, e\} \]
\[ C_g = \{g, f\} \]
\[ C_h = \{h\} \]
Strongly connected components: Example

Acyclic component graph

- $a, b, e$
- $f, g$
- $c, d$
- $h$

$C_b$, $C_g$, $C_c$, $C_h$
In any DFS($G$), all vertices in the same SCC are placed in the same DFT.

In the DFS($G$) step of the algorithm, the last vertex finished in an SCC is the first vertex discovered in the SCC.

Consider the vertex $r$ with the largest finishing time. It is a root of a DFT. Any vertex that is reachable from $r$ in $G^T$ should be in the SCC of $r$ (why?)
SCC and reducibility

To detect if there exists a permutation matrix \( P \) such that

\[
PAP^T = \begin{pmatrix} A_{11} & A_{12} \\ O & A_{22} \end{pmatrix},
\]

where \( A_{11} \) is an \( r \times r \) submatrix, \( A_{22} \) is an \( (n-r) \times (n-r) \) submatrix, where \( 1 \leq r < n \):

run SCC on the directed graph of \( A \) to identify each strongly connected component as an irreducible block (more than one SCC?). Hence \( A_{11} \), too, can be in that form (how many SCCs?).
SCC and fully indecomposability

To detect if there exists permutation matrices \( P \) and \( Q \) such that

\[
PAQ = \begin{pmatrix} A_{11} & A_{12} \\ O & A_{22} \end{pmatrix},
\]

where \( A_{11} \) is an \( r \times r \) submatrix, \( A_{22} \) is an \( (n - r) \times (n - r) \) submatrix, where \( 1 \leq r < n \):

1) Find a zero-free diagonal (by permuting the columns with \( Q \) corresponding to a perfect matching),

2) Run SCC on the directed graph of \( B = AQ \) to (finds a \( P \) such that \( PBP^T \) is in the form above).
Could not get enough of it: Questions

How would you describe the following in the language of graphs

- the structure of $PAP^T$ for a given square sparse matrix $A$ and a permutation matrix $P$,
- the structure of $PAQ$ for a given square sparse matrix $A$ and two permutation matrices $P$ and $Q$,
- the structure of $A^k$, for $k > 1$, where $A$ has a zero diagonal. What about $A^k$ when $A$ has a zero-free diagonal?
- the structure of $AA^T$,
- the structure of the vector $b$, where $b = Ax$ for a given sparse matrix $A$, and a sparse vector $x$. 
Could not get enough of it: Questions

Can you define:
- the row-net hypergraph model of a matrix.
- a matching in a hypergraph (is it a hard problem?).

Can you relate:
- the DFS or BFS on a tree to a topological ordering? postordering?