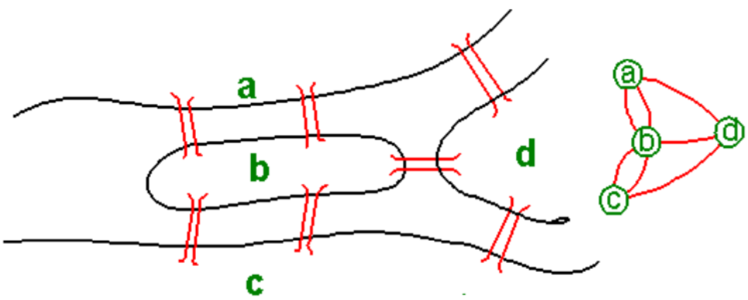
Data Structures

Elementary Graph Algorithms

BFS, DFS & Topological Sort

Tzachi (Isaac) Rosen

The 7 Bridges of Konigsberg



Tzachi (Isaac) Rosen

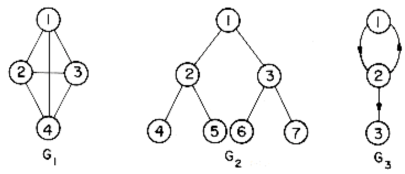
Graphs

- A **graph**, G = (V, E), consists of two sets:
  - V is a finite non-empty set of **vertices**.
  - E is a set of pairs of vertices, called **edges**.

Graphs

- In an **undirected graph**
  - The pair of vertices are unordered pairs.
  - Thus, the pairs (v_1, v_2) and (v_2, v_1) are the same.
  - No reflective (self) edges.
- In a **directed graph**
  - The edges are represented by a directed pair (v_1, v_2).
  - Therefore (v_2, v_1) and (v_1, v_2) are two different edges
  - v_1 is the **tail** and v_2 the **head** of the edge.

Graphs

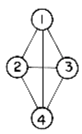


- G_1 and G_2 are undirected.
- G_3 is a directed graph.

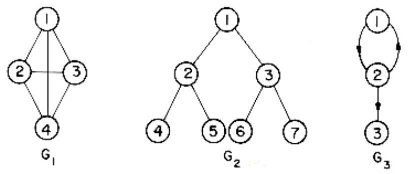
- G_1 = ({1,2,3,4}, {(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)})
- G_2 = ({1,2,3,4,5,6,7}, {(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)})
- G_3 = ({1,2,3}, {<1,2>, <2,1>, <2,3>})

Graphs

- The **maximum number edges** in
  – An **undirected graph** with n vertices is n(n - 1)/2
  – A **directed graph** with n vertices is n^2
- An n vertex undirected graph with exactly n(n-1)/2 edges is said to be **complete**

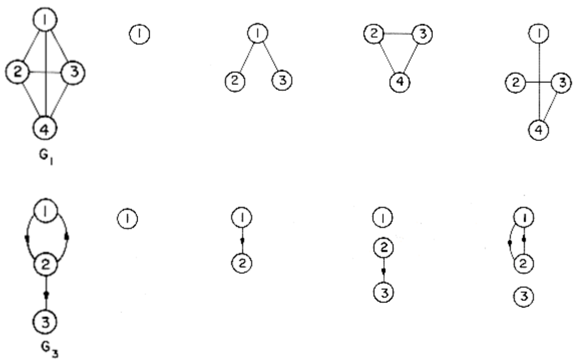


Graphs

- If (v_1,v_2) is an edge of a graph G, then
  - We shall say the vertices v_1 and v_2 are **adjacent**
  - And that the edge (v_1,v_2) is **incident** on vertices v_1 and v_2
- If (v_1,v_2) is a directed edge, then vertex v_1 will be said to be **adjacent to** v_2, while v_2 is **adjacent from** v_1.
- The **degree** of a vertex is
  - The number of edges incident to that vertex
- In case G is a directed graph, we define
  - the **in-degree** of a vertex v to be the number of edges for which v is the head.
  - The **out-degree** is defined to be the number of edges for which v is the tail.
- 

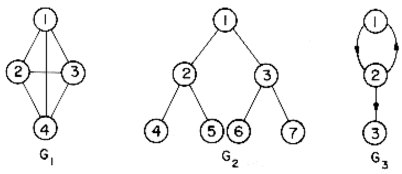
Graphs

- A **subgraph** of a graph G is a grapl G' = (V’, E’) such that
  - $V' \subseteq V$
  - $G' \subseteq G$.



Graphs

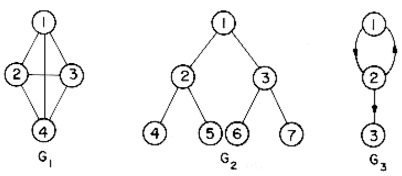
- A **path** from vertex v_p to vertex v_q ($v_p \rightsquigarrow v_q$) in graph G is
  - A **sequence of vertices** $v_p, v_{i1}, v_{i2}, \ldots, v_{in}, v_q$ such that
  - $(v_p, v_{i1}), (v_{i1}, v_{i2}), \ldots, (v_{in}, v_q)$ are edges
- The **length** of a path is **the number of edges** on it
- A **simple path** is a path in which
  - **All vertices** except possibly the first and last **are distinct**



Tzachi (Isaac) Rosen

Graphs

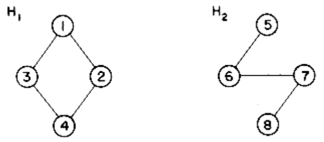
- A **cycle** is a path in which
  - The first and last vertices are the same
- A **simple cycle** – same with simple path
- When the graph is **directed**, we add the prefix "directed" to the terms



Tzachi (Isaac) Rosen

Graphs

- In an undirected graph, G, two **vertices** v_1 and v_2 are said to be **connected**
  - If there is a path in G from v_1 to v_2.
  - Since G is undirected, this means there must also be a path from v_2 to v_1.
- An **undirected graph** is said to be **connected**
  - If for every pair of distinct vertices v_i, v_i in there is a path from v_i to v_j in G
- A **(connected) component** of an undirected graph is a maximal connected subgraph
- A **tree** is a connected acyclic undirected graph



Graphs

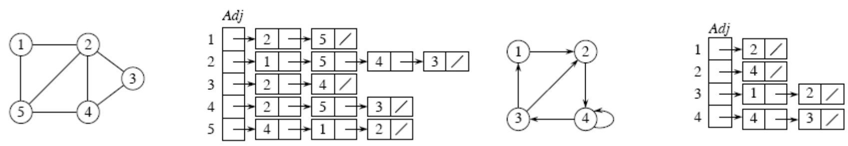
Simple cycle | Nonsimple cycle | Free Tree | Forest | DAG

Graph Representation

- Two common ways to represent a graph (either directed or undirected):
  - **Adjacency lists**.
  - **Adjacency matrix**.

Adjacency Lists

- Array Adj of |V| lists,
  - One per vertex.
  - Vertex u's list has all vertices v such that (u, v) ∈ E.
  - Works for both directed and undirected graphs.

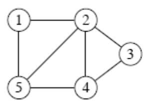


- **Space**: Θ(V + E).
  - When expressing the running time, we'll drop the cardinality.
- **Time to list** all vertices adjacent to u: Θ(degree(u)).
- **Time to determine** if (u, v) ∈ E: O(degree(u)).

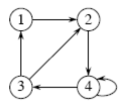
Adjacency Matrix

- $|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E , \\ 0 & \text{otherwise} . \end{cases}$$



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1

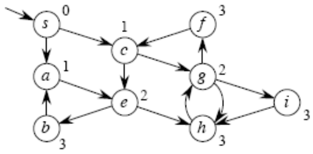
- Space: Θ(V^2).
- **Time to list all vertices** adjacent to u: Θ(V).
- Time to determine if (u, v) ∈ E: O(1).

Breadth-First Search

- Given a graph G = (V, e) and a vertex s ∈ E
  - Which vertex is reachable from s
  - What is the shortest distance to it
  - What is the shortest path

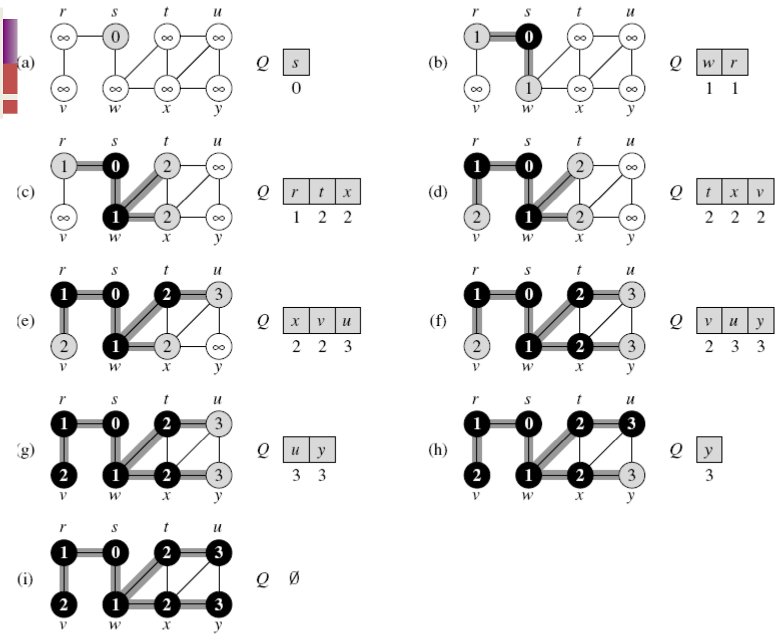
Breadth-First Search

- Algorithm Idea:
  - Send a wave out from s:
    - First hits all vertices 1 edge from s.
    - From there, hits all vertices 2 edges from s.
    - Etc.
  - Use FIFO queue Q to maintain wave front.
    - v ∈ Q if and only if wave has hit v but has not come out of v yet.



Breadth-First Search

- **Input**:
  - **Graph** G = (V, E), either directed or undirected.
  - **Source vertex** s ∈ V.
- **Output**:
  - d[v] = **distance** to all v ∈ V from s.
    - If v ∈ V is not reachable from s, d[v] = ∞.
  - π[v] = u such that (u, v) is **last edge on shortest** path s ⇝ v.
    - If v ∈ V is not reachable from s, π[v] will be **null**.
    - The set of edges {(π[v], v) : v ≠ s} forms a **tree**, such that u is v's **predecessor**.
- **Auxiliary Means**:
  - every vertex has a **color:**
    - White - **undiscovered**
    - Gray - **discovered, but not finished** (not done exploring from it)
    - Black - **finished** (have found everything reachable from it)



Tzachi (Isaac) Rosen

Breadth-First Search

BFS(G,s)

  for (each vertex u ∈ V[G] - {s}) **do**

    color[u] = **white**, d[u] = ∞, π[u] = **null**

  Q = {s}, color[s] = **gray**, d[s] = 0 , π[s] = **null**

  while (Q ≠ Ø) **do**

    u = **dequeue**(Q)

    for (each v ∈ Adj[u]) **do**

      if (color[v] = **white**) **then**

        enqueue(Q, v), color[v] = **gray**, d[v] = d[u] + 1, π[v] = u

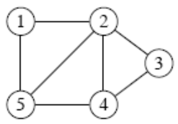
    color[u] = **black**

Complexity: O(V + E)

Tzachi (Isaac) Rosen

Breadth-First Search

- The **shortest-path distance** from s to v, δ(s, v), is:
  - The minimum length of the paths from s to v, or else
  - ∞, if there is no path from s to v.
- A **path** of length δ(s, v) from s to v is said to be a **shortest path** from s to v.

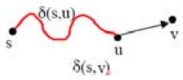


Breadth-First Search

- **Theorem**:
  - Suppose that BFS is run on graph G = (V, E) from a source s ∈ V, then
  - **It discovers every vertex** v ∈ V that is reachable from s.
  - Upon termination, **d[v] = δ(s, v)** for all v ∈ V.
  - For any vertex v ≠ s that is reachable from s,
    - **One of the shortest paths** from s to v is a shortest path from s to π[v] followed by the edge (π[v], v).
    - Hence, the path (s, π[…π[v]]), …, (π[π[v]], π[v]), (π[v], v) is one of the shortest path from s to v.

Breadth-First Search

- Proof:
  - We will use the following two facts:
    - **Fact 1**: Upon termination, for each $v \in V$, $d[v] \geq \delta(s, v)$.
    - **Fact 2**: If vertex v_i is enqueued before v_j during the execution then $d[v_i] \leq d[v_j]$.
  - Assume, for the purpose of contradiction, that there is $v \in V$ such that upon termination $d[v] \neq \delta(s, v)$.
  - If there are more then one, take v with minimum $\delta(s, v)$.
  - Clearly $v \neq s$.
  - By fact 1, $d[v] \geq \delta(s, v)$, and thus, from the assumption, $d[v] > \delta(s, v)$.
  - Vertex v must be reachable from s, for if it is not, then $\delta(s, v) = \infty \geq d[v]$.
  - Let u be the vertex immediately preceding v on a shortest path from s to v, so that $\delta(s, v) = \delta(s, u) + 1$.



  - Since $\delta(s, u) < \delta(s, v)$, and because of how we chose v, we have $d[u] = \delta(s, u)$.
  - Putting these together, we have $d(v) > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1$.

Breadth-First Search

- Consider the time when vertex u dequeue from Q.
- At this time, vertex v is either white, gray, or black.
  - If v is white, then the BFS sets $d[v] = d[u] + 1$, contradiction.
  - If v is black, then it was already removed from the queue and, by fact 2, we have $d[v] \leq d[u]$, contradiction.
  - If v is gray, then it was painted gray upon dequeuing some vertex w, which was removed from Q earlier than u and for which $d[v] = d[w] + 1$.
  - By fact2, however, $d[w] \leq d[u]$, and so we have $d[v] \leq d[u] + 1$, contradiction.
- Thus we conclude that $d[v] = \delta(s, v)$ for all $v \in V$.
- All vertices reachable from s must be discovered, for if they were not, they would have infinite d values.
- if $\pi[v] = u$, then $d[v] = d[u] + 1$. Thus, we can obtain a shortest path from s to v by taking a shortest path from s to $\pi[v]$ and then traversing the edge $(\pi[v], v)$.
- By induction, the path $(s, \pi[\ldots\pi[v]]), \ldots, (\pi[\pi[v]], \pi[v]), (\pi[v], v)$ is one of the shortest path from s to v.

Printing The Shortest Path

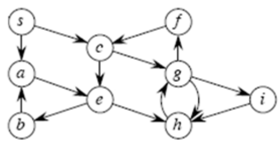
PrintPath (G, s, v)
  if (v = s) **then**
    print s
  else if (π[v] = null) **then**
    print "no path from" s "to" v "exists"
  else
    printPath (G, s, π[v])
    print v

Depth-First Search

- Given a graph G = (V, E)
  - Draw G as a forest of sub graphs
  - Say which vertex is a descendant of another in the forest.
  - Detect cycles
  - Topological Sort

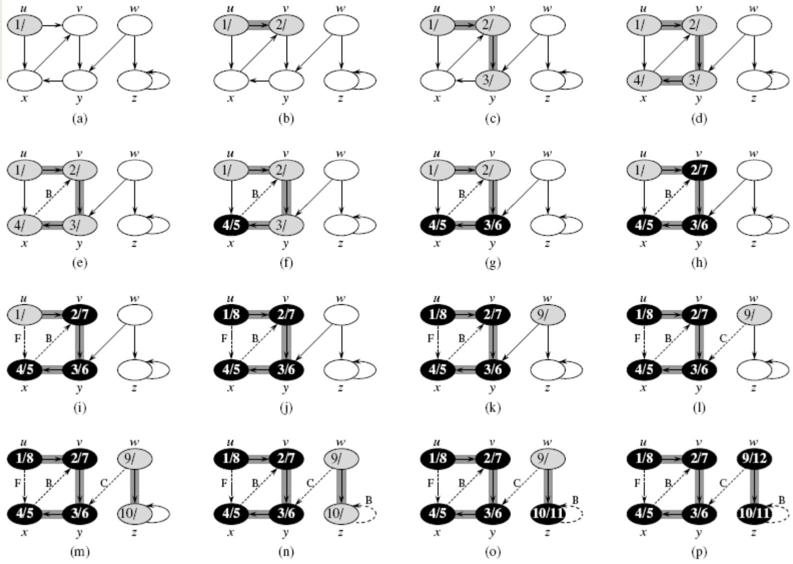
Depth-First Search

- Algorithm Idea:
  - Search deeper in the graph whenever possible:
    - **Start** from an arbitrary unvisited vertex.
    - **Explore** out one of the undiscovered edge of the most recently discovered vertex v.
    - When all of v's edges have been explored, **backtracks**, and continue.
    - **Start** all over again.



Depth-First Search

- **Input**:
  - **Graph** G = (V, E), either directed or undirected.
- **Output**:
  - d[v] = **discovery time** & f [v] = **finishing time**.
    - A unique integer from 1 to 2|V| such that $1 \le d[v] < f[v] \le 2|V|$.
  - $\pi[v] = u$ such that u is the **predecessor** to v in the visit order.
    - The predecessor subgraph $G\pi = (V, E\pi)$, where $E\pi=\{(\pi[v],v) : \pi[v] \neq null\}$ forms a **forest** composed of several trees.
- **Auxiliary Means**:
  - every vertex has a **color:**
    - White - **undiscovered**
    - Gray - **discovered, but not finished** (not done exploring from it)
    - Black - **finished** (have found everything reachable from it)



Depth-First Search

dfs (G)
  for (each vertex u ∈ V[G]) **do**
    color[u] = **white**
    π[u] = **null**
  time = 0
  for (each vertex u ∈ V[G]) **do**
    if (color[u] = white) **then**
      dfsVisit(u)

dfsVisit (u)
  color[u] = **gray**
  d[u] = time = time+1
  for (each v ∈ Adj[u]) **do**
    if (color[v] = **white**) **then**
      π[v] = u
      dfsVisit(v)
  color[u] = **black**
  f[u] = time = time+1

- No source vertex given.
- Will explore every edge.

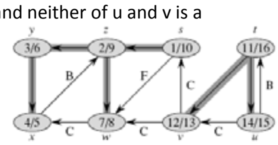
Complexity: Θ(V + E).

Depth-First Search

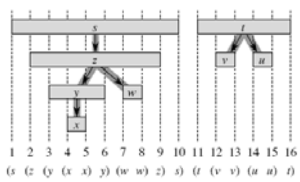
- When one vertex is a descendant of another in the forest that was constructed by the DFS.
  - Parenthesis Theorem
  - White-path Theorem

Parenthesis Theorem

- **Theorem** (Parenthesis theorem):
  - For all u, v, exactly one of the following holds:
    1. d[u] < f [u] < d[v] < f [v] or d[v] < f [v] < d[u] < f [u] and neither of u and v is a descendant of the other.
    2. d[u] < d[v] < f [v] < f [u] and v is a descendant of u.
    3. d[v] < d[u] < f [u] < f [v] and u is a descendant of v.
  - So d[u] < d[v] < f [u] < f [v] cannot happen.



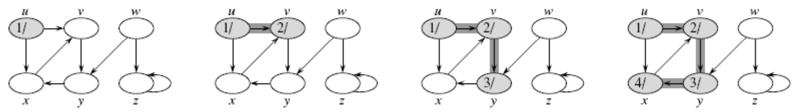
- Like parentheses:
  - OK: ( ) [ ] ([ ]) [( )]
  - Not OK: ([ )] [( ])



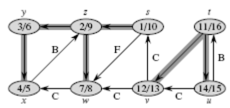
- **Corollary (Nesting of descendants' intervals)**:
  - v is a proper descendant of u if and only if d[u] < d[v] < f [v] < f [u].

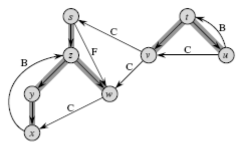
White-path Theorem

- Theorem (White-path theorem):
  - v is a descendant of u if and only if at time d[u], there is a path u ⇝ v consisting of only white vertices (Except for u, which was just colored gray)



Classification of Edges





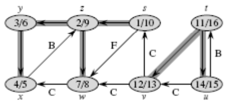
- **Tree edge**:
  - In the constructed forest.
  - Found by exploring (u, v).
- **Back edge**:
  - (u, v), where u is a descendant of v.
- **Forward edge**:
  - (u, v), where v is a descendant of u, but not a tree edge.
- **Cross edge**:
  - any other edge.
  - Can go between vertices in same depth-first tree or in different depth-first trees.

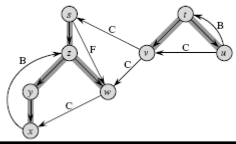
- In an undirected graph, there may be some ambiguity since (u, v) & (v, u) are the same edge.
  - Classify by the first type above that matches.

- **Theorem**:
  - In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

Classification of Edges

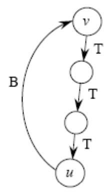
- Edge (u, v) can be classified by the color of v when the edge is first explored:
  - WHITE - indicates a **tree edge**
  - GRAY - indicates a **back edge**
  - BLACK indicates a **forward or cross edge**.
    - (u, v) is a **forward edge** if d[u] < d[v]
    - (u, v) is a **cross edge** if d[u] > d[v].





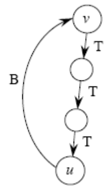
Detection of Cycles

- Lemma:
  - A directed graph G is acyclic if and only if a DFS of G yields no back edges.
- Proof:
  - Back edge ⇒ Cycle
    - Suppose there is a back edge (u, v).
    - Then v is ancestor of u in the constructed forest.
    - Therefore, there is a path v ⇝ u, so v ⇝ u → v is a cycle.



Detection of Cycles

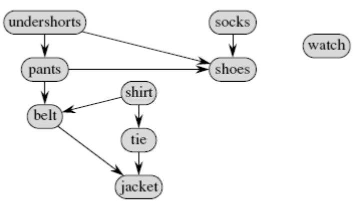
- Cycle ⇒back edge.
  - Suppose G contains cycle C.
  - Let v be the first vertex discovered in c, and let (u, v) be the preceding edge in C.
  - At time d[v], vertices of C form a white path v ⇝ u
    - since v is the first vertex discovered in c.
  - By white-path theorem, u is descendant of v in depth-first forest.
  - Therefore, (u, v) is a back edge.



Tzachi (Isaac) Rosen

Topological Sort

- **Directed acyclic graph** (dag) is a directed graph with no cycles
- Good for modeling a **partial order:**
  - a > b and b > c ⇒ a > c.
  - May have a and b such that neither a > b nor b > c.
- **Topological sort of a dag**: a linear ordering of vertices such that if (u, v) ∈ E, then u appears somewhere before v.

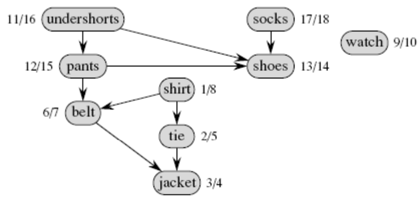


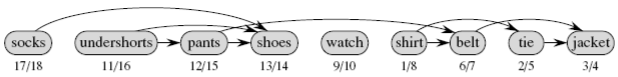
Tzachi (Isaac) Rosen

Topological Sort

topologicalSort (V, E)
// Assume G is a DAG
Call dfs(V, E) to compute finishing times f[v] for all v ∈ V
Output vertices in order of decreasing finish times

Complexity: Θ(V + E)





Tzachi (Isaac) Rosen

Topological Sort

- **Correctness**:
  - Just need to show if (u, v) ∈ E, then f [v] < f [u].
- When we explore (u, v), what are the colors of u and v?
  - u is gray.
  - v can't be gray.
    - Because then (u, v) is a back edge, but G is a dag.
  - If v is white.
    - By parenthesis theorem, d[u] < d[v] < f[v] < f[u].
  - If v is black.
    - Then v is already finished, but u doesn't, therefore, f [v] < f [u].

Tzachi (Isaac) Rosen