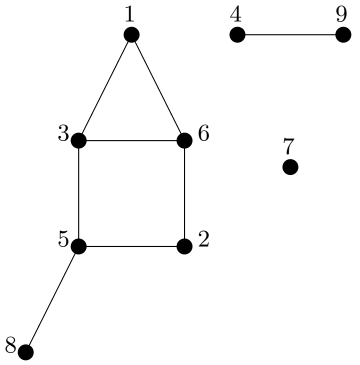
Agenda:

- Graphs — basic definitions
- Graphs Representation
- Graph Traversals: Breadth First Search (BFS) and applications

Reading:

- CLRS: 589-602

A Graph:



Q: What Are Graphs?

- Basically, nodes and edges.
- What nodes represent differs from one application to another
  - People
  - Computers
  - Webpages
  - Cites on a map
  - Proteins
  - Airports
  - ...
- Edges connect two vertices.
  They represent the idea that u and v have some *direct* interaction.
- Again, the exact nature of the interaction depends on the application.
  - u is a friend of v
  - u and v can sends messages directly to one another
  - Webpage u has a link to webpage v
  - Cities u and v are connected by a highway
  - Protein u transmutates to protein v
  - You can fly directly from airport u to airport v
  - ...
- Note that if there is no edge between u and v it *doesn't* necessarily mean there is no interaction between u and v
  - Perhaps we can reach from u to v through other nodes...
  - ... and perhaps not

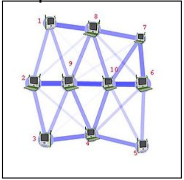
Q: Why Study Graphs?

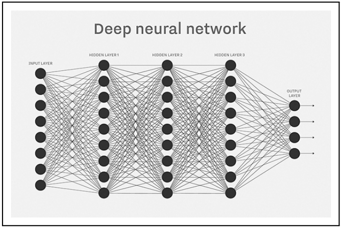
- A1: Many problems can be cast down to graph problems.
- Example: Sorting
  - Nodes: elements in A
  - Edges: a directed edge $A[i] \to A[j]$ if $A[i] \leq A[j]$.
  - Sorting: find a path with n nodes.
  - Observe, just constructing this graph takes $\Theta(n^2)$ time...
- Example: Puzzles can often be reduced to graphs, like Towers of Hanoi —
  - Nodes: all possible legal configurations of the disks on all 3 pegs
  - (Undirected) Edges: connect configuration i to j if there's an action taking us from i to j.
  - Now find a path (=sequence of moves) taking you from the start-configuration (all disks on peg A) to the end-configuration (all disks on C).
- Casting the problem as a graph should be one of the first go-to solutions.
  - Sometime yields the best approach, sometimes doesn't
  - But it is always good to keep in mind
  - and keep in mind the time it takes to create the representing graph.

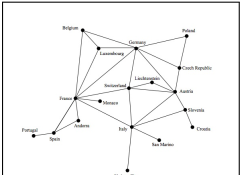
- A2: Because they are there...
  - Offer many interesting and complex challenges
  - 1,229 books in the UAlberta libraries alone under "graph theory"...

Casting Problems as Graphs

computer networks

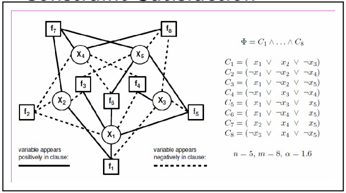




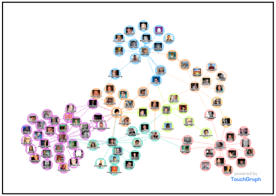


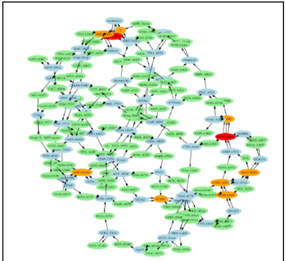
Maps

Constraint Satisfaction



Social Network





Protein Reaction

Basic Graph Definitions $G = (V, E)$

- V Nodes / Vertices
  - A set of n elements with unique identifiers: usually numbers from $\{1, ..., n\}$.
- E Edges
  - Undirected Graph: each edge is a multiset of exactly two nodes $e = \{u, v\}$
    - We say e connects u and v
    - We say u and v are adjacent, or neighbors
  - Directed Graph (digraph): each edge/arc – tuple of two nodes $e = \langle u, v \rangle$.
    - We say e leaves u and enters v; or e is from u to v.
    - We say u and v are adjacent; u is an in-neighbor of v; v is an out-neighbor of u.
- Loops / self-loops: an edge connecting a node to itself.
  - Unless specified otherwise: assume no self loops, and no multiple edges
- Degree of v: $\deg(v) =\#$ edges that touch v = # neighbors of v
  - In a digraph: separated into in-degree (#in-neighbors) and out-degree (#out-neighbors)
- A path: a sequence of nodes $v_0, v_1, ..., v_k$ such there exists k edges $e_1, ..., e_k$ where e_i connects v_{i-1} to v_i. ($k + 1$ nodes, k edges)
- A simple path: a path where all nodes are unique
  - k is the length of the path (length is measured in edges)
  - We often assume a path is a simple path from v_0 to v_k
- A cycle: a path where $v_0 = v_k$.
- A simple cycle: a cycle where all nodes but v_0 and v_k are unique
  - k is the length of the cycle
  - We often assume a cycle is a simple cycle
- An acyclic graph is a graph that has no cycles.

Basic Graph Definitions

- Size of the graph $|G| = |V| = n$
- We often use the notation $V(G), E(G)$.
- "A n-nodes and m-edges graph" — means $|V(G)| = n$ and $|E(G)| = m$.
  - Undirected graph: $m \leq \binom{n}{2}$.
  - Directed graph: $m \leq n(n-1)$.
  - The empty graph: no edge belongs to E
  - The complete graph: all edges belong to E
- Degrees and edges:
  - Undirected graph: $\deg(v) =$#edges adjacent to v.
    Directed: $in\_\deg(v) =$#edges entering v; $out\_\deg(v) =$#edges leaving v.
  - The Handshake Lemma: In an undirected graph $\sum_{v \in V} \deg(v) = 2m$
  - In a directed graph $\sum_{v \in V} in\_\deg(v) = m = \sum_{v \in V} out\_\deg(v)$
- $G' = (V', E')$ is a sub-graph of $G = (V, E)$ if $V' \subset V$ and $E \subset E'$.
  - Removing an edge e from G results in the subgraph $(V, E \setminus \{e\})$
- The induced subgraph on $V' \subset V$ is the graph $G[V'] = G|_{V'} = (V', E')$ where $e \in E'$ iff $e \in E$ and both its vertices are in V'
  - Removing a node v from G results in the induced graph $G[V \setminus \{v\}]$
- U is an independent set in G if $G[U]$ is the empty graph.
- U is a clique in G if $G[U]$ is the complete graph.

Connectivity in an Undirected Graph

- u is connected to v ($u \sim v$) if there exists a path from u to v.
- G is a connected graph if for every $u, v \in V, \quad u \sim v$
- $C \subset V$ is the connected component of u ($CC(u)$) if it is the maximal set C such that $u \in C$ and $G[C]$ is connected.
  - We often identify C with $G[C]$
- Connectivity is an *equivalence relation*
  - Reflexivity: for every u we have $u \sim u$ by a path of length 0
  - Symmetry: for every u and v, $u \sim v$ iff $v \sim u$
  - Transitivity: for every u, v, w, if $u \sim v$ and $v \sim w$ then $u \sim w$
- So the connected component that contains a node u, denoted $CC(u) = \{v \in V : \; u \sim v\}$, is *well-defined and unique*
  - So $u \sim v$ iff $CC(u) = CC(v)$.
- Thus the different connected components of G form a partition of $V(G)$.
  - every edge $e \in E$ belongs to a unique CC and no edge connects two components.
- What are all the connected components of the *complete* graph?
  - A single component: $C = V(G)$.
- What are all the connected components of the *empty* graph?
  - n different singletons: $\{v_1\}, \{v_2\}, \{v_3\}, ..., \{v_n\}$.
- Connectivity induces a metric (distances)
  - $d(u, v) \stackrel{\text{def}}{=}$ min length of a simple path from u to v; or ∞ if $u \nsim v$
  - Reflexivity: $d(u, v) = 0$ iff $u = v$, for any $u, v \in V$
  - Symmetry: $d(u, v) = d(v, u)$, for any $u, v \in V$
  - Triangle inequality: $d(u, w) \leq d(u, v) + d(v, w)$, for any $u, v, w \in V$

Forests and Trees

- A forest F is an acyclic graph.
- A tree T is a connected acyclic graph, and we say T spans the vertices $V(T)$.
  - The connected components of a forest are trees, each spanning all the vertices in its connected component
- All of the following definitions of a tree are equivalent:
  - A maximal acyclic graph
    - Adding any edge to T results in a cycle
  - A minimal connected graph
    - Remove an edge from T and it is no longer connected
  - A connected and acyclic graph
  - An acyclic graph with $n-1$ edges
  - A connected graph with $n-1$ edges
- A graph G is connected iff it has a spanning tree:
  a subgraph T which is a tree with $V(T) = V(G)$.

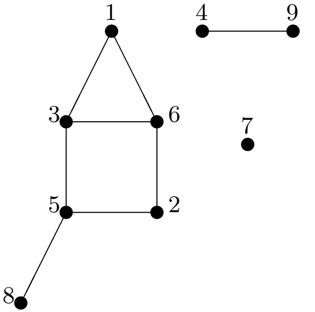
Strong Connectivity in a Digraph

- v is reachable from u ($u \to v$) if there exists a path from u to v.
  - Defines distances: $d(u,v) \overset{\text{def}}{=}$ min length of path $u \to v$; or ∞ if no such path exists.
  - Not symmetric: $d(u,v) \neq d(v,u)$ in general (so common to use $d(u \to v)$)
- u and v are strongly connected ($u \sim v$) if there exists a path from u to v and a path from v to u.
  Exists a (directed) cycle containing both u and v.
- G is a strongly connected graph if for every $u, v \in V, \quad u \sim v$
- $C \subset V$ is the strongly-connected component of u ($SCC(u)$) if it is the maximal set C such that $u \in C$ and $G[C]$ is strongly-connected.
- Strong-connectivity is an *equivalence relation*
  - Reflexivity: for every u we have $u \sim u$ by a path of length 0
  - Symmetry: for every u and v, $u \sim v$ iff $v \sim u$
  - Transitivity: for every u, v, w, if $u \sim v$ and $v \sim w$ then $u \sim w$
- So $C = SCC(u)$ is unique $SCC(u) = \{v \in V : \ u \sim v\}$.
  - So $u \sim v$ iff $SCC(u) = SCC(v)$.
- Thus the different SCCs of G form a partition of $V(G)$
  - There could be an edge between two strongly-connected components, but no edge back
  - Given G, think of the graph H where $V(H)$ are the SCCs of G and there is an edge $\langle C_i, C_j \rangle$ if for some $u \in C_i$ and $v \in C_j$ there is an edge $\langle u, v \rangle \in E$.
    Then H is acyclic.
    (This called "contracting the nodes each C_i")

Representing Graphs

- Representing the nodes
  - We will assume that all nodes are stored in an array / a hash-table
  - And that each node is accessible in $O(1)$ time
  - And that we can traverse all nodes in $O(n)$ time
    - independent of m, the number of edges
  - Nodes will have different attributes / fields as required
    - degree, color, parent, distance, etc...
    - So the code "`if` ($v.color$ =`WHITE`)" takes $O(1)$-time
    - Not the same as "`if exists some` v `with` $v.color$ =`WHITE`" which takes naively $O(n)$-times to check, unless we do something clever...
- Representing the edges
  - We will not necessarily assume there exists an array of edges
    ... though later we will show how to traverse all edges
  - But rather that the edges are given in one of two representations:
  - Adjacency matrix: an $n \times n$-matrix where the i, j-entry contains e if such an edge exists or 0 o/w.
  - Adjacency lists: each node has an array / a list of all the edges that are adjacent to it
  - Some operations are more efficient in the adjacency-matrix model, some operations are more efficient in the adjacency-list model.
  - NOTE: Edges may have attributes too (color, weight, capacity, label, etc...)
    Regardless of the representation we use, we assume that once we reach an edge e we can access its attributes in $O(1)$-time

An example:



node →	array/list
1 →	3 6
2 →	6 5
3 →	5 1 6
4 →	9
5 →	2 8 3
6 →	3 2 1
7 →	
8 →	5
9 →	4

	1	2	3	4	5	6	7	8	9
1	0	0	1	0	0	1	0	0	0
2	0	0	0	0	1	1	0	0	0
3	1	0	0	0	1	1	0	0	0
4	0	0	0	0	0	0	0	0	1
5	0	1	1	0	0	0	0	1	0
6	1	1	1	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	1	0	0	0	0
9	0	0	0	1	0	0	0	0	0

Comparison between the two representations

	Adjacency Lists	Adjacency Matrix
Space (so good for)	$O(m)$ sparse graphs	$O(n^2)$ dense graphs
Accessing a node v Traversing all nodes	$O(1)$ $O(n)$	$O(1)$ $O(n)$
Accessing an edge $e = (u, v)$ (finding if e exists)	$O(\|\Gamma(u)\|)$ # neighbors(u)	$O(1)$
Finding some neighbor of v	$O(1)$	$O(n)$
Traversing all edges/vertices adjacent to a node u	$O(\|\Gamma(u)\|)$	$O(n)$
Traversing all edges	$O(m)$	$O(n^2)$

Comments about Graph Representations

- If G is undirected, then the adjacency matrix is symmetric.
- Sometimes, in runtime analysis it is easier to use a max-degree bound $\Delta = \max_v deg(v)$ (since all lists have length $\leq \Delta$)
- We do not assume the lists are sorted according to the neighbors' identifiers, the neighbors' attirbutes or the edges' attributes. We will be responsible to sort them or keep them in order (using Priority Queues)
- Example: find if u and v are of distance$=2$
  - This means that exists some w s.t $(u, w) \in E$ and $(w, v) \in E$.
  - $O(n)$ in the matrix model
  - In the adjacency lists model (undirected graphs or if we keep incoming edges for each node):
    - Naïvely: for each neighbor x of u, check if x is in the adjacency list of v. Runtime $O(|\Gamma(u)| \times |\Gamma(v)|)$.
    - Better runtime: Sort first the list for u and for v, then iterate both, $O\left(|\Gamma(u)| \log(|\Gamma(u)|) + |\Gamma(v)| \log(|\Gamma(v)|)\right)$
    - One more way: construct a $\{0, 1\}$ array for all the nodes, and see if they are connected to u and v (i.e., build the respective row from the adjacency matrix) in $O(n)$ time.
    - Finally, you can use a hash-table: build a hash-table with the vertices adjacent to u, and try to `Find()` in it each vertex adjacent to v. This takes $O((|\Gamma(u)| + |\Gamma(v)|) \cdot t)$ where t is the time it takes to hash.

- A bipartite graph is a graph where V can be partitioned into two disjoint sets $V = R \cup L$, such that all edges have one right- and one left-vertex.
- A bipartite graph can be represented also by a $|R| \times |L|$-matrix.

Graph Traversal

- The most elementary graph algorithm:
- Goal: visit all vertices, by following the edge structure of the graph
- Via graph traversals we find all vertices connected/reachable from a given vertex u, find distances, connected components, characterize edges, etc.
  - *E.g.*, maze traversal — is there a path "enter" $\rightarrow$ "exit"?
- There are two main principled ways to traverse the graph
  - Breadth First Search (BFS)
    - We start at v, then first visit all of its neighbors, then visit all of its neighbors' neighbors, then neighbors' neighbors' neighbors and so on.
    - Think of a balloon sitting at v and inflating until it shadows the entire graph
  - Depth First Search (DFS)
    - We start at v, take a path for as far as it takes us, then go up the path and take any other branches we can, until we exhaust all paths from v.
    - Think of water being poured on v until the entire graph is flooded.
- Both use the notion of a node color — representing its state

$$\underset{\text{Never saw } v}{\texttt{WHITE}} \xrightarrow[\text{previsit time}]{\text{discovery}} \underset{\text{Working on } v}{\texttt{GRAY}} \xrightarrow[\text{postvisit time}]{\text{finish}} \underset{\text{Done with } v}{\texttt{BLACK}}$$

  - All vertices start as `WHITE` and end as `BLACK`
  - The order in which we make these $2n$ color changes is of importance! (the time in which a vertex turns gray and when it turns black)

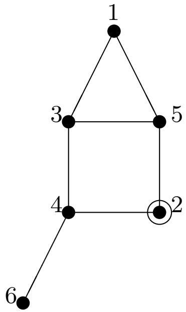
Breadth First Search (BFS):

- Assume for now all nodes are connected to s
- Pseudocode:

```
procedure BFS-Visit(G, s)          ** G = (V, E),  s ∈ V start vertex
foreach v ∈ V do
   v.color ← WHITE                 **unknown yet
   v.dist ← ∞                      **distance from s
   v.predec ← nil                  **predecessor
Initialize a queue Q               **waiting vertex queue
s.color ← GRAY                     **in queue Q
s.dist ← 0
enqueue(Q, s)
while (Q ≠ ∅) do
   u ← dequeue(Q)
   foreach neighbor v of u do
      if (v.color = WHITE) then
         v.color ← GRAY            **discovered v
         v.dist ← u.dist + 1
         v.predec ← u
         enqueue(Q, v)
   u.color ← BLACK                 **done with u
```

BFS example:

- $V = \{1, 2, 3, 4, 5, 6\}$
  $E = \{\{1, 3\}, \{1, 5\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{3, 5\}, \{4, 6\}\}$
  $s = 2$



Adjacency lists

1:	3	5	
2:	4	5	
3:	1	4	5
4:	2	3	6
5:	1	2	3
6:	4		

BFS example:

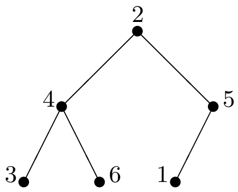
	1	2	3	4	5	6	Q
color	W	G	W	W	W	W	$\{2\}$
distance	∞	0	∞	∞	∞	∞	
parent	NIL	NIL	NIL	NIL	NIL	NIL	
color	W	B	W	G	G	W	$\{4, 5\}$
distance	∞	0	∞	1	1	∞	
parent	NIL	NIL	NIL	2	2	NIL	
color	W	B	G	B	G	G	$\{5, 3, 6\}$
distance	∞	0	2	1	1	2	
parent	NIL	NIL	4	2	2	4	
color	G	B	G	B	B	G	$\{3, 6, 1\}$
distance	2	0	2	1	1	2	
parent	5	NIL	4	2	2	4	
color	G	B	B	B	B	G	$\{6, 1\}$
distance	2	0	2	1	1	2	
parent	5	NIL	4	2	2	4	
color	G	B	B	B	B	B	$\{1\}$
distance	2	0	2	1	1	2	
parent	5	NIL	4	2	2	4	
color	B	B	B	B	B	B	$\emptyset$
distance	2	0	2	1	1	2	
parent	5	NIL	4	2	2	4	

BFS example:

- Adjacency lists:

1:	3	5	
2:	4	5	
3:	1	4	5
4:	2	3	6
5:	1	2	3
6:	4		

- BFS tree:
  - root is the start vertex s
  - parent of u is predecessor $u.predec$
  - left-to-right child order *depends* on neighbor ordering (in u's list)



Properties of BFS

- Each u that is reachable from s is visited,
  enqueued exactly once (turns `GRAY`)
  and dequeued exactly once (turns `BLACK`)
- For any u denote $d(u, s)$ the true distance between s and u, and $u.dist$ as the $distance$ given by BFS.
  Claim: $u.dist = d(u, s)$, and the path from u to s using the predecessors is a shortest path.
  - Prove by induction on d that all nodes u with $d(u, s) = d$ are assigned $u.dist = d$.
  - Base case $d = 0$ and we only have s to consider.
  - Induction step. Let u be a node s.t. $d(u, s) = d + 1$. On *all* shortest-paths from s to u the next-to-last node must be of distance d from s, so by IH it was assigned $dist = d$; and in particular – it had to be turned gray and enqueued by the BFS. So, among all next-to-last-nodes let x be the first node to be enqueued. This means u is discovered by the edge (x, u), which means $u.dist = x.dist + 1 = d + 1$.

Properties of BFS

- Each u that is reachable from s is visited,
  enqueued exactly once (turns `GRAY`)
  and dequeued exactly once (turns `BLACK`)
- For any u denote $d(u, s)$ the true distance between s and u, and $u.dist$ as the $distance$ given by BFS.
  Claim: $u.dist = d(u, s)$, and the path from u to s using the predecessors is a shortest path.
- BFS creates layers $L_i = \{u : u.dist = i\}$ such that for any edge (u, v) we have $L(v) - L(u) \leq 1$.
  - For an undirected graph — all edges are between the same or adjacent layers.
- For any u, v, if $L(u) < L(v)$, then u turns `GRAY` before v, enqueued before v and turns `BLACK` before v.
- At any moment, all vertices in the queue belong to the same or adjacent layers. (But never layers at distance ≥ 2)

BFS runtime analysis:

- $n = |V|$, $m = |E|$
- Analysis:
  - each vertex enqueued exactly once: WHITE $\rightarrow$ GRAY
  - each vertex dequeued exactly once: GRAY $\rightarrow$ BLACK
  - running time:
    1. adjacency list representation:
       $\Theta(n + \sum_{v \in V} \texttt{degree}(v)) = n + 2m) = \Theta(n + m)$
    2. adjacency matrix representation:
       $\Theta(n + \sum_{v \in V} n = n + n^2) = \Theta(n^2)$
  - space complexity: (in addition to the list / matrix representation)
    1. Each node has a color attribute $\Omega(n)$
    2. Since each vertex is enqueued exactly once, the queue size never passed $O(n)$
    3. So $\Theta(n)$.
- Warning: vertices in other connected components wouldn't be discovered!!!

Breadth First Search (BFS):

- procedure BFS(G)   ** $G = (V, E)$

```
foreach v ∈ V do
    v.color ← WHITE              **unknown yet
    v.dist ← ∞                   **distance from s
    v.predec ← nil               **predecessor
foreach v ∈ V do
        if (v.color =WHITE) then
           BFS-visit(G, v).
```

- procedure BFS-visit(G, s)   ** $G = (V, E)$, $s \in V$ start vertex

```
Initialize a queue Q             **waiting vertex queue
s.color ← GRAY                   **in queue Q
s.dist ← 0
enqueue(Q, s)
while (Q ≠ ∅) do
    u ← dequeue(Q)
    foreach neighbor v of u do
        if (v.color = WHITE) then
           v.color ← GRAY        **discovered v
           v.dist ← u.dist + 1
           v.predec ← u
           enqueue(Q, v)
    u.color ← BLACK              **done with u
```

- Runtime?
- HW: In an undirected graph — adjust BFS to assign each vertex a label such that the labels indicate the connected components of G.