**Graphs**

**What is a Graph?**

|  |
| --- |
|    |
|  |

Informal definition:

* A *graph* is a mathematical abstraction used to represent "connectivity information".
* A graph consists of *vertices* and *edges* that connect them, e.g.,



* It shouldn't be confused with the "bar-chart" or "curve" type of graph.

|  |
| --- |
|    |

Formally:

* A graph *G = (V, E)* is:
	+ a set of vertices *V*
	+ and a set of edges *E = { (u, v): u* and *v* are vertices }.
* Two types of graphs:
	+ **Undirected graphs**: the edges have no direction.
	+ **Directed graphs**: the edges have direction.
* Example: undirected graph



* + Edges have no direction.
	+ If an edge connects vertices *1* and *2*, either convention can be used:
		- No duplication: only one of *(1, 2)* or *(2, 1)* is allowed in *E*.
		- Full duplication: both *(1, 2)* and *(2, 1)* should be in *E*.
* Example: directed graph



* + Edges have direction (shown by arrows).
	+ The edge *(3, 6)* is not the same as the edge *(6, 3)* (both exist above).

|  |
| --- |
|    |

Depicting a graph:

* The picture with circles (vertices) and lines (edges) is *only a depiction*
=> a graph is purely a mathematical abstraction.
* Vertex labels:
	+ Can use letters, numbers or anything else.
	+ Convention: use integers starting from 0.
	=> useful in programming, e.g. degree[i] = degree of vertex i.
* Edges can be drawn "straight" or "curved".
* The geometry of drawing has no particular meaning:



|  |
| --- |
|    |

Graph conventions:

* What's allowed (but unusual) in graphs:
	+ Self-loops (occasionally used).
	+ Multiple edges between a pair of vertices (rare).
	+ Disconnected pieces (frequent in some applications).

Example:



* What's not (conventionally) allowed:
	+ Mixing undirected and directed edges.
	+ Re-using labels in vertices.
	+ Bidirectional arrows.
* Most common:
	+ No multiple edges.
	+ No self-loops.
* Other terms used:
	+ Vertices: nodes, terminals, endpoints.
	+ Edges: links, arcs.

Definitions:

* **Degrees**:
	+ Undirected graph: the **degree** of a vertex is the number of edges incident to it.
	+ Directed graph: the **out-degree** is the number of (directed) edges leading out, and the **in-degree** is the number of (directed) edges terminating at the vertex.
* **Neighbors**:
	+ Two vertices are **neighbors** (or are **adjacent**) if there's an edge between them.
	+ Two edges are **neighbors** (or are **adjacent**) if they share a vertex as an endpoint.
* **Paths**:
	+ Undirected: a sequence of vertices in which successive vertices are adjacent.
	+ Directed: a sequence of vertices in which every pair of successive vertices has this property: there's a directed edge from the first to the second.
	+ A **simple path** does not repeat any vertices (and therefore edges) in the sequence.
	+ A **cycle** is a simple path with the same vertex as the first and last vertex in the sequence.
* **Connectivity:**
	+ Undirected: Two vertices are **connected** if there is a path that includes them.
	+ Directed: Two vertices are **strongly-connected** if there is a (directed) path from one to the other.
* **Components:**
	+ A **subgraph** is a subset of vertices together with the edges from the original graph that connects vertices in the subset.
	+ Undirected: A **connected component** is a subgraph in which every pair of vertices is connected.
	+ Directed: A **strongly-connected component** is a subgraph in which every pair of vertices is strongly-connected.
	+ A **maximal component** is a connected component that is not a proper subset of another connected component.
* **Digraph**: another name for a *directed graph*.

Example:



|  |
| --- |
|    |

More definitions:

* **Euler tour**: A cycle that traverses all edges exactly once (but may repeat vertices).



Known result: Euler tour exists if and only if all vertices have even degree.

* **Hamiltonian tour**: A cycle that traverses all vertices exactly once.



Known result: testing existence of a Hamiltonian tour is (very) difficult.

* **Euler path**: A path that traverses all edges exactly once.
* **Hamiltonian path**: A path that traverses all vertices exactly once.
* **Trees**:
	+ A **tree** is a connected graph with no cycles.
	+ A **spanning tree** of a graph is a connected subgraph that is a tree.



* **Weighted graphs**:
	+ Sometimes, we include a "weight" (number) with each edge.
	+ Weight can signify length (for a geometric application) or "importance".
	+ Example:



|  |
| --- |
|    |

**Graph Data Structures**

|  |
| --- |
|    |

First, an idea that doesn't work:

* We have already represented trees (like binary trees) with node instances and pointers between instances.
* Idea: use a node instance for each vertex, and a pointer from one vertex to another if an edge exists between them.

The two fundamental data structures:

* Adjacency matrix.
	+ Key idea: use a 2D matrix.
	+ Row *i* has "neighbor" information about vertex *i*.
	+ Undirected: adjMatrix[i][j] = 1 if and only if there's an edge between vertices *i* and *j*.
	adjMatrix[i][j] = 0 otherwise.
	+ Directed: adjMatrix[i][j] = 1 if and only if there's an edge from *i* to *j*.
	adjMatrix[i][j] = 0 otherwise.
	+ Example: undirected



 0 **1** **1** 0 0 0 0 0

 **1** 0 **1** 0 0 0 0 0

 **1** **1** 0 **1** 0 **1** 0 0

 0 0 **1** 0 **1** 0 **1** 0

 0 0 0 **1** 0 0 **1** 0

 0 0 **1** 0 0 0 **1** **1**

 0 0 0 **1** **1** 0 0 0

 0 0 0 0 0 **1** 0 0

Note: adjMatrix[i][j] == adjMatrix[j][i] (convention for undirected graphs).

* + Example: directed



 0 **1** 0 0 0 0 0 0

 0 0 **1** 0 0 0 0 0

 **1** 0 0 **1** 0 **1** 0 0

 0 0 0 0 **1** 0 **1** 0

 0 0 0 0 0 0 **1** 0

 0 0 **1** 0 0 0 0 **1**

 0 0 0 **1** 0 0 0 0

 0 0 0 0 0 0 0 0

* Adjacency list.
	+ Key idea: use an array of vertex-lists.
	+ Each vertex list is a list of neighbors.
	+ Example: undirected



* + Example: directed



* + Convention: in each list, keep vertices in order of insertion
	=> add to rear of list
* Both representations allow complete construction of the graph.
* Advantages of matrix:
	+ Simple to program.
	+ Some matrix operations (multiplication) are useful in some applications (connectivity).
	+ Efficient for dense (lots of edges) graphs.
* Advantages of adjacency list:
	+ Less storage for sparse (few edges) graphs.
	+ Easy to store additional information in the data structure.
	(e.g., vertex degree, edge weight)

**Breadth-First Search**

|  |
| --- |
|    |

About graph search:

* "Searching" here means "exploring" a particular graph.
* Searching will help reveal properties of the graph
e.g., is the graph connected?
* Usually, the input is: vertex set and edges (in no particular order).

|  |
| --- |
|    |

Key ideas in breadth-first search: (undirected)

* Mark all vertices as "unvisited".
* Initialize a queue (to empty).
* Find an unvisited vertex and apply breadth-first search to it.
* In breadth-first search, add the vertex's neighbors to the queue.
* Repeat: extract a vertex from the queue, and add its "unvisited" neighbors to the queue.

|  |
| --- |
|    |

Example:

* Initially, place vertex 0 in the queue.



* Dequeue 0
=> mark it as visited, and add its unvisited neighbors to queue:



* Dequeue 1
=> mark it as visited, and add its unvisited neighbors to queue:



* Dequeue 2
=> mark it as visited, and add its unvisited neighbors to queue:



* Dequeue 2
=> it's already visited, so ignore.
* Continuing ...



* Breadth-first search tree, and visit order:



* + Exploring an edge: examining an unvisited neighbor.
	+ If an unvisited neighbor gets on the queue for the first time, the edge is called a "tree edge".
	+ Putting the tree edges and all vertices together results in: the *breadth-first search tree*.
	+ For a particular graph and its implementation, the tree produced is unique.
	+ However, starting from another vertex will result in another tree, that may be just as useful.

Searching an unconnected graph:

* The connected components are explored in order:
* Example:



The tree, and visit order:



|  |
| --- |
|    |
|   |

Applications:

* Connectivity:
	+ Breadth-first search identifies connected components.
	+ However, depth-first search is preferred (required for directed graphs).
* Shortest paths:
	+ A path between two vertices in the tree is the shortest path in the graph.
* Optimization algorithms:
	+ Various problems result in "graph search space".
	+ BFS together with "exploration rules" is often used to search for solutions (e.g., branch-and-bound exploration).

Note: BFS works on a weighted graph by ignoring the weights and only using connectivity information (i.e., is there an edge or not?).

**Depth-First Search on Undirected Graphs**

Key ideas:

* Mark all vertices as "unvisited".
* Visit first vertex.
* Recursively visit its "unvisited" neighbors.

|  |
| --- |
|    |

Example:

* Start with vertex 0 and mark it visited.



* Visit the first neighbor 1, mark it visited.



* Explore 1's first neighbor, 2.



* Continuing until all vertices are visited ...



* + Vertices are marked in order of visit.
	+ An edge to an unvisited neighbor that gets visited next is in the depth-first search tree.

|  |
| --- |
|    |
|   |

* Completion order:
	+ In breadth-first search, once a vertex is processed, it is never processed again.
	+ In depth-first, we also encounter a vertex after returning from the recursive call.
	=> we can record a *completion order*.



|  |
| --- |
|    |
|    |

**Depth-First Search in Directed Graphs**

Key ideas:

* A straightforward depth-first search is similar to the undirected version
=> only explore edges going outward from a vertex in a directed graph.
* In addition to "back" and "down" edges, it is useful to identify "cross" edges.

|  |
| --- |
|    |

Example:

* Consider: (slightly different from previous example)



* Applying DFS gives:



|  |
| --- |
|    |