4 Basic graph theory and algorithms

References: [DPV06, Ros11].

4.1 Basic graph definitions

Definition 4.1. A *graph* $G = (V, E)$ is a set V of *vertices* and a set E of *edges*. Each edge $e \in E$ is associated with two vertices u and v from V, and we write $e = (u, v)$. We say that u *is adjacent to* v, u *is incident to* v, and u *is a neighbor of* v.

Graphs are a common abstraction to represent data. Some examples include: road networks, where the vertices are cities and there is an edge between any two cities that share a highway; protein interaction networks, where the vertices are proteins and the edges represent interactions between proteins; and social networks, where the nodes are people and the edges represent friends.
Sometimes we want to associate a direction with the edges to indicate a one-way relationship. For example, consider a predator-prey network where the vertices are animals and an edge represents that one animal hunts the other. We want to express that the fox hunts the mouse but not the other way around. This naturally leads to *directed graphs*:

Definition 4.2. A *directed graph* $G = (V, E)$ is a set V of *vertices* and set E of *edges*. Each edge $e \in E$ is an ordered pair of vertices from V. We denote the edge from u to v by (u, v) and say that u *points to* v. This could be useful, for example, in making a graph representation of a road network where some streets were one-way.

Here are some other common definitions that you should know:

Definition 4.3. A *weighted graph* $G = (V, E, w)$ is a graph (V, E) with an associated weight function $w : E \to \mathbb{R}$. In other words, each edge e has an associated weight $w(e)$. This could be useful incorporating lengths of roads in a graph representation of a road network.

Definition 4.4. In an undirected graph, the *degree* of a node $u \in V$ is the number of edges $e = (u, v) \in E$. We will write d_u to denote the degree of node u.

Lemma 4.5. ***The handshaking lemma****.* *In an undirected graph, there are an even number of nodes with odd degree.*

Proof. Let d_v be the degree of node v.

$$2|E| = \sum_{v \in G} d_v = \sum_{v \in G : d_v \text{ even}} d_v + \sum_{v \in G : d_v \text{ odd}} d_v$$

The terms $2|E|$ and $\sum_{v \in G : d_v \text{ even}} d_v$ are even, so $\sum_{v \in G : d_v \text{ odd}} d_v$ must also be even. $\square$

Definition 4.6. In a directed graph, the *in-degree* of a node $u \in V$ is the number of edges that point to u, i.e., the number of edges $(v, u) \in E$. The *out-degree* of u is the number of edges that point away from u, i.e., the number of edges $(u, v) \in E$.

Definition 4.7. A *path* on a graph is a sequence of nodes $v_1, v_2, \ldots v_k$ such that the edge $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, k - 1$ (see Figure 2).

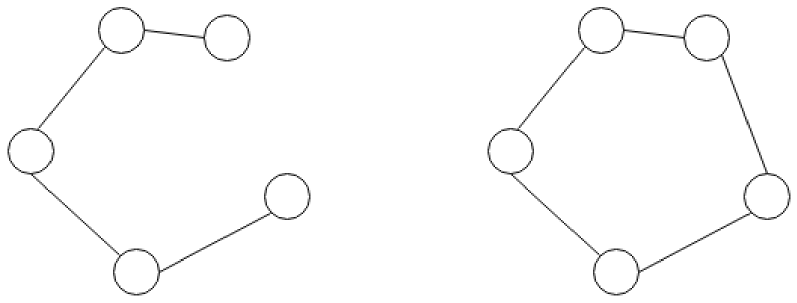


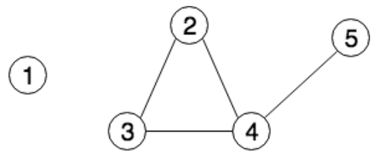
Figure 2: An example of a path (left) and cycle (right).

Definition 4.8. A *cycle* is a path $v_1, v_2, \ldots v_k$ with $k \geq 3$ such that $v_1 = v_k$ (see Figure 2).

Definition 4.9. A *complete graph* is a graph where for every $u, v \in V$, $u \neq v$, the edge $(u, v) \in E$.

Definition 4.10. The *complement* of a graph G is the graph G' such that $(u, v) \in G' \iff (u, v) \notin G$.

4.2 Storing a graph



$$\begin{array}{c|c}
 & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}
\end{array}$$

1	
2	3,4
3	2,4
4	2,3,4
5	4

Figure 3: Example graph with adjacency matrix and adjacency list.

How do we actually store a graph on a computer? There are two common methods: an *adjacency matrix* or an *adjacency list*. In the adjacency matrix, we store a matrix A of size $|V| \times |V|$ with entry $A_{ij} = 1$ if edge (i, j) is in the graph, and $A_{ij} = 0$ otherwise. The main advantage of an adjacency matrix is that we can query the existence of an edge in $\Theta(1)$ time—we just look up the entry in the matrix. The disadvantage is that storage is $\Theta(|V|^2)$. In many graphs, the number of edges is far less than $|V|^2$, which makes adjacency matrices impractical for many problems. However, adjacency matrices are still a useful tools for understanding graphs.
Adjacency lists, on the other hand, only need memory proportional to the size of the graph. They consist of a length-$|V|$ array of linked lists. The ith linked list contains all nodes adjacent to node i (also called node i's *neighbor list*). We can access the neighbor list of i in $\Theta(1)$ time, but querying for an edge takes $O(d_{\max})$ time, where $d_{\max} = \max_j d_j$. You can see examples of these two storage methods in Figure 3.

4.3 Graph exploration

Definition 4.11. We say that node v is *reachable* from node u if there is a path from u to v.

We now cover two ways of exploring a graph: depth-first search (DFS) and breadth-first search (BFS). The goal of these algorithms is to find all nodes reachable from a given node, or simply to explore all nodes in a graph. We will describe the algorithms for undirected graphs, but they generalize to directed graphs.

4.3.1 Depth-first search

In depth first search we explore a graph by starting at a vertex and then following a path of unexplored vertices away from this vertex as far as possible (and repeating).
This construction of DFS comes from [DPV06]. We will use an auxiliary routine called `Explore(v)`, where v specifies a starting node. We use a vector C to keep track of which nodes have already been explored. The routine is described in Algorithm 7. We have included functions `Pre-visit()` and `Post-visit()` before and after exploring a node. We will reserve these functions for book-keeping to help us understand other algorithms. For example, `Pre-visit()` may record the order in which nodes were explored.

```
Data: Graph $G = (V, E)$, starting node $v$, vector of covered nodes $C$
Result: $C[u] = true$ for all nodes $u$ reachable from $v$
$C[v] \leftarrow true$
for $(v, w) \in E$ do
    if $C[w]$ is not true then
        Pre-visit(w)
        Explore(w)
        Post-visit(w)
    end
end
```

Algorithm 7: Function `Explore(v)`

Given the explore routine, we have a very simple DFS algorithm. We present it in Algorithm 8. The cost of DFS is $O(|V| + |E|)$. We only call `Explore()` on a given node once. Thus, we only consider the neighbors of each node once. Therefore, we only look at the edges of a given node once.

```
Data: Graph $G = (V, E)$
Result: depth-first-search exploration of $G$
for $v \in V$ do
    // Initialized covered nodes
    $C[v] \leftarrow false$
end
for $v \in V$ do
    if $C[v]$ is not true then
        Explore(v)
    end
end
```

Algorithm 8: Function `DFS(G)`

We now look at an application of DFS: topological ordering of DAGs.

Definition 4.12. A *directed acyclic graph (DAG)* is a directed graph that has no cycles.

DAGs characterize an important class of graphs. One example of a DAG is a schedule where some tasks can't start before others have finished. There is an edge (u, v) if task u must finish before task v. If a cycle exists, then the schedule cannot be completed. We now investigate how to order the tasks to create a valid sequential schedule.

Definition 4.13. A *topological ordering* of the nodes v is an order such that if `order[v]` $<$ `order[w]`, then there is no path from w to v.

Example 4.14. *Suppose we define the post-visit function to simply update a count and record the counter, as in Algorithm 9. Ordering the nodes by decreasing* `post[v]` *after a DFS search gives a topological ordering of a DAG.*

Proof. Suppose not. Then there are nodes v and w such that post$[v] >$ post$[w]$ and there is a path from w to v. If we called the explore function on w first, then post$[v] <$ post$[w]$ as there is a path from w to v. This is a contradiction. Suppose instead that we called the explore function on v first. If w was reachable from v, then there is a cycle in the graph. Hence, w must not be reachable from v. But then post$[w]$ would be larger than post$[v]$ since we explored v first. Again, we have a contradiction. ☐

```
// Post-visit()
// count is initialized to 0 before running the algorithm
post[v] $\leftarrow$ count
count $\leftarrow$ count + 1
```

Algorithm 9: Post-visit function to find topological orderings

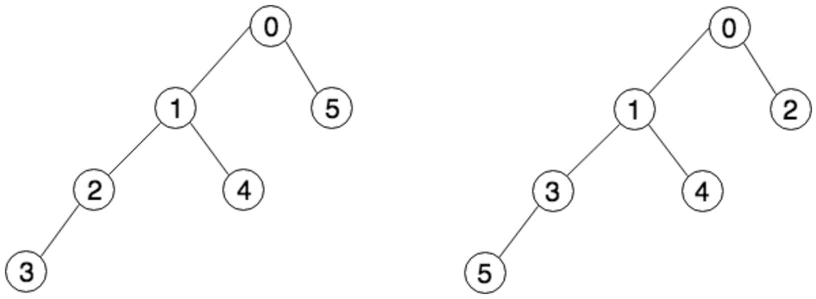


Figure 4: Left: a tree's nodes numbered by DFS. Right: the same tree's nodes numbered by BFS.

4.3.2 Breadth-first search

With DFS, we went as far along a path away from the starting node as we could. With breadth-first-search (BFS), we instead look at all neighbors first, then neighbors of neighbors, and so on. For an example of the difference between these two search methods see Figure 4. For BFS, we use a queue data structure. A *queue* is an object that supports the following two operations:

- `enqueue(x)`: puts the data element x at the back of the queue

- `dequeue()`: returns the oldest element in the queue

Algorithm 10 describes BFS and also keeps track of the distances of nodes from a source node s to all nodes reachable from s.

```
Data: Graph $G = (V, E)$, source node $s$
Result: For all nodes $t$ reachable from $s$, dist[t] is set to the length of the smallest path
        from $s$ to $t$. dist[t] is set to $\infty$ for nodes not reachable from $s$
for $v \in V$ do
    dist[v] $\leftarrow \infty$
end
dist[s] $\leftarrow 0$
Initialize queue $Q$
$Q$.enqueue($s$)
while $Q$ is not empty do
    $u \leftarrow Q$.dequeue()
    for $(u, v) \in E$ do
        if dist[$v$] is $\infty$ then
            $Q$.enqueue($v$)
            dist[$v$] $\leftarrow$ dist[$u$] + 1
        end
    end
end
```

Algorithm 10: Breadth-first search

4.4 Connected components

Definition 4.15. An *induced subgraph* of $G = (V, E)$ specified by $V' \subset V$ is the graph $G' = (V', E')$, where $E' = \{(u, v) \in E | u, v \in V'\}$.

Definition 4.16. In an undirected graph G, a *connected component* is a maximal induced subgraph $G' = (V', E')$ such that for all nodes $u, v \in V'$, there is a path from u to v in G'.

By "maximal", we mean that there is no V'' such that $V' \subset V''$ and the induced subgraph specified by V'' is also a connected component.

Definition 4.17. In a directed graph G, a *strongly connected component* is a maximal induced subgraph $G' = (V', E')$ such that for all nodes $u, v \in V'$, there are path from u to v and from v to u in G'.

Definition 4.18. An undirected graph is *connected* if G is a connected component. A directed graph is *strongly connected* if G is a strongly connected component.

We now present a simple algorithm for finding connected components in an undirected graph. First, we update the DFS procedure (Algorithm 8) to increment a counter `count` before calling `Explore()`, with initialization to 0. Then the previsit function assigns connected component numbers.

```
// Pre-visit()
connected_component[v] $\leftarrow$ count
```

Algorithm 11: Pre-visit for connected components in undirected graphs

4.5 Trees

Definition 4.19. A *tree* is a connected, undirected graph with no cycles.

Definition 4.20. A *rooted tree* is a tree in which one node is called the root, and edges are directed away from the root.

Note that an undirected tree can be transformed into a rooted tree from any vertex in the tree since there are no cycles.

Definition 4.21. In a rooted tree, for each directed edge (u, v), u is the *parent* of v and v is the *child* of u.

Note that the parent of a node v must be unique.

Definition 4.22. In a rooted tree, a *leaf* is a node with no children.

Note that all finite trees must have at least one leaf node (we can find them by a topological ordering). We can also see this fact by starting at an arbitrary vertex and following any path, which must stop somewhere, and where it does, we have found a leaf node. In fact, every tree has at least 2 leaf nodes, because we can start from the leaf node instead of an arbitrary node as before; following an arbitrary path necessarily leads to a second leaf node (cannot be the same as the original because there are no cycles).

Theorem 4.23. *A tree with n vertices has $n - 1$ edges.*

Proof. We go by induction. With $n = 1$, there are no edges. Consider a graph with k nodes. Let v be a leaf node and consider removing v and (u, v) from the tree, where u is the parent of v. The remaining graph is a tree, with one fewer edge. By induction, it must have $k - 2$ edges. Therefore, the original tree had $k - 1$ edges. □

4.5.1 Minimum spanning trees

Definition 4.24. A spanning tree of an undirected graph $G = (V, E)$ is a tree $T = (V, E')$, where $E' \subset E$.

Definition 4.25. The minimum spanning tree $T = (V, E', w)$ of a weighted, connected undirected graph $G = (V, E, w)$ is a spanning tree where the sum of the edge weights of E' is minimal.

Two algorithms for finding minimum spanning trees are Prim's algorithm (Algorithm 12) and Kruskal's algorithm (Algorithm 13). Both procedures are straightforward. We will analyze Kruskal's algorithm for correctness to get used to this type of analysis. Note that you will go over Kruskal's algorithm in CME 305 and by the end of the class you will see more generally why it works (because greedy algorithms work for matroid optimazation). You can find a proof of correctness for Prim's algorithm at https://en.wikipedia.org/wiki/Prim%27s_algorithm#Proof_of_correctness.

```
**Data:** Connected weighted undirected graph $G = (V, E, w)$
**Result:** Minimum spanning tree $T = (V, E')$
$S \leftarrow \{v\}$ for any arbitrary node $v \in V$
$E' \leftarrow \emptyset$
while $|S| < |V|$ do
    $(u, v) \leftarrow \arg\min_{u \in S, v \notin S} w((u, v))$
    $S \leftarrow S \cup \{u\}$
    $E' \leftarrow E' \cup \{(u, v)\}$
end
$T \leftarrow (V, E')$
```

Algorithm 12: Prim's algorithm for minimum spanning trees

```
**Data:** Connected weighted undirected graph $G = (V, E, w)$
**Result:** Minimum spanning tree $T = (V, E')$
$E' \leftarrow \emptyset$
foreach $e = (u, v) \in E$ by increasing $w(e)$ do
    if adding $(u, v)$ to $E$ does not creates a cycle then
        $E' \leftarrow E' \cup \{(u, v)\}$
    end
end
$T \leftarrow (V, E')$
```

Algorithm 13: Kruskal's algorithm for minimum spanning trees

Theorem 4.26. *When all of the weights of G are distinct, the output T from Kruskal's algorithm is a minimum weight spanning tree for the connected graph G.*

Proof. First we notice that T must be a spanning tree of G because by construction it has no cycles and must be connected and spanning. Otherwise we would have multiple connected components, and because G is connected the algorithm must have considered at least one edge connecting these disconnected components and added at least one (no cycles would have formed). So T is a spanning tree of G.
Now we must show that T is a minimum-weight spanning tree. For contradiction, assume that T isn't a minimum-weight spanning tree. Let T' be a minimum-weight spanning tree of G (so we are assuming $w(T') < w(T)$.
Let e be the first edge added to T during Kruskal's algorithm that's not in T'. Then $T' \cup \{e\}$ contains a cycle. Therefore one of the edges in this cycle must not be in T, because T is a tree and therefore contains no cycles. Call this edge f. Then $T'' = (T' \cup \{e\}) \setminus f$ is also a spanning tree. Because T' is a minimum spanning tree, we know that $w(e) > w(f)$. But we also know $w(e) < w(f)$ because otherwise we would have added f to T before e in Kruskal's algorithm. Therefore we have a contradiction. Therefore T is a minimum spanning tree.

□

Note that if we use the correct data structures for our graph, the major step in this algorithm is sorting the edges by weight. Therefore, the running time of Kruskal's algorithm is $O(|E|\log(|E|)$.

Exercise 4.27. *Adapt the proof of correctness for Kruskal's algorithm to work even when not all of the edge weights of G are distinct.*

4.6 Cycles

Definition 4.28. A *Hamiltonian cycle* of a graph G is a cycle that visits each node of G exactly once.

Definition 4.29. A *Hamiltonian path* of a graph G is a path that visits each node of G exactly once.

Definition 4.30. A *Eulerian cycle* of a graph G is a cycle that visits each edge of G exactly once.

Definition 4.31. A *Eulerian path* of a graph G is a path that visits each edge of G exactly once.

Exercise 4.32. *Show the following:*

- *A connected graph is Eulerian (has an Eulerian cycle) if and only if the degree of every vertex is even.*
- *A graph contains an Eulerian path if and only if it is connected and has exactly two vertices of odd degree.*

Example 4.33. *Suppose the minimum degree of a graph G is k. Show that the graph contains a cycle of length at least $k + 1$.*

Proof. Consider the longest path in G. Let $u, v \in V$ be the endpoints of this path. We know that all the neighbors of u and v must be on this path, otherwise this would not be the longest path. Let v_i be the neighbor of u that's ith farthest away from u on the path out of all of u's neighbors. Then we know that following the path from u to v_k will include at least k edges. We know that $(u, v_k) \in E$ because v_k is a neighbor of u. Therefore traveling from v_k back to u is a cycle of length $\geq k + 1$. We can make this same argument to show that v is also in at least one cycle of length $\geq k + 1$. Drawing a picture can help you understand this example. $\square$

Note that this technique of considering the longest path of a graph when working under assumptions about the minimum degree can be very useful. You will see in class that using this technique (along with some other ideas) that if the minimum degree of a graph is greater than or equal to $\frac{|V|}{2}$, then the graph contains a Hamiltonian cycle, usually a very hard property to determine.

Exercise 4.34. *If C is a cycle, and e is an edge connecting two nonadjacent nodes of C, then we call e a* chord. *Show that if every node of a graph G has degree at least 3, then G contains a cycle with a chord.*

4.7 Cuts

Definition 4.35. A *cut* $C = (A, B)$ of a graph $G = (V, E)$ is a partition of V into two subsets A and B. The *cut-set* of C is the set of edges that cross the cut, i.e. $\{(u, v) \in C | u \in A, v \in B\}$.

We are almost always interested in the number of edges (or total weight of the edges in the case of directed graphs) in the cut-set.

Definition 4.36. An *s-t minimum cut* is the cut $C^* = (S, T)$ with $s \in S$ and $t \in T$ such that the cut-set is of minimum weight.

Definition 4.37. A *global minimum cut* is a cut $C^* = (A, B)$ in the graph G such that both A and B are non-empty and the cut-set has minimum weight.

Definition 4.38. A *global maximum cut* is a cut $C^* = (A, B)$ in the graph G such that both A and B are non-empty and the cut-set has maximum weight.

You will see in class that $s-t$-minimum cuts are very related to network flow problems and are very useful for scheduling problems among other applications. Both kinds of minumum cuts are easy to compute, while maximum cuts are very hard.

4.8 Other definitions and examples

Definition 4.39. A *bipartite graph* is a graph whose vertices can be divided into two disjoint sets A and B such that every edge connects a vertex in A to one in B.

Although we said earlier that max cuts are typically difficult to compute, they are very easy for bipartite graphs. A global maximum cut of a bipartite graph with nonnegative weights is simply $C(A, B)$.

Exercise 4.40. *Show that every tree is bipartite.*

Note that we can further prove that a graph is bipartite if and only if it contains no cycles of odd length.

Definition 4.41. An *independent set* is a set of vertices $I \subseteq V$ in a graph such that no two in the set are adjacent.

Definition 4.42. A *vertex cover* of a graph is a set of vertices $C \subseteq V$ such that each edge of the graph is incident to at least one vertex in C.

Definition 4.43. An *edge cover* of a graph is a set of edges $C \subseteq E$ such that every vertex of the graph is incident to at least one edge in C.

Note that we can see immediately from the above definitions that each side of a bipartite graph will necessarily be both an independent set and a vertex cover. Using this fact and Exercise 4.40, we can see also that every tree on n nodes has a vertex cover of size at most $\lceil \frac{n-1}{2} \rceil$.

Definition 4.44. A *matching* M in a graph G is a set of edges such that no two edges share a common vertex.

Definition 4.45. A *maximal matching* is a matching M of a graph G with the property that if any edge of G not already in M is added to M, the set is no longer a valid matching.

Definition 4.46. A *maximum matching* of a graph G is the matching that contains the greatest number of edges.

We see that all maximum matchings must also be maximal, but the opposite is certainly not true.

Definition 4.47. A *perfect matching* of a graph G is a matching such that for any vertex v in G there exists an edge in the matching that is adjacent to v.

Example 4.48. *Assume that you are given a graph $G = (V, E)$, a matching M in G and an independent set S in G. Show that $|M| + |S| \leq |V|$.*

Proof. Let M^* be a maximum matching of G and S^* be a maximum independent set of G. For each of the $|M^*|$ matched pairs, $|S^*|$ can include at most one of the vertices, so $|S^*| \leq |V| - |M^*|$. We have $|S^*| + |M^*| \leq |V|$ and because M^* and S^* are the maximum size matching and independent set respectively, we have $|S| + |M| \leq |V|$ for all matchings and independent sets. $\square$