# Machine Learning with Graphs: Graph Algorithms 1/2 - Definitions, Problems

Arlei Silva

Spring 2022

We will formally define a few types of graphs and introduce graph problems and algorithms useful for machine learning.

## Definitions and Notation

A *graph* is a tuple $G = (V, E)$, where $V$ is a set of *vertices/nodes* $\{v_1, v_2, \dots v_n\}$, and $E$ is a set of $m$ *edges/links/arcs* $E = \{e_1, \dots e_m\}$, where $e_i = \{u, v\}$ is an unnordered tuple and $u, v \in V$. If $\{u, v\} \in E$, we say that $u$ and $v$ are *neighbors* (or adjacent). The set of all neighbors of $v$ is given by $N(v) = \{u | \{u, v\} \in E\}$. The *degree* $d(v) = N(v)$ of a vertex $v$ is its number of neighbors. A graph with $|E| = \binom{n}{2}$—i.e. $d(v) = n - 1, \forall v \in V$—is called a *complete graph*.

We call $G' = (V', E')$ a *subgraph* of $G$ if $V' \subset V$ and $E' \subset E$. A *k-clique* of $G$ is a subgraph $G'$ of $G$ where $|V'| = k$ and $G'$ is a complete graph. A *triangle* is a 3-clique. Two graphs $G = (V, E)$ and $G' = (V', E')$ are said to be *isomorphic* if there is a bijection $\phi : V \to V'$ such that $\{u, v\} \in E$ iff $\{\phi(u), \phi(v)\} \in E'$. A subraph $G'$ of $G$ is *subgraph isomorphic* to $G$ if $G'$ is isomorphic to a subgraph of $G$, i.e. there is a bijection $\phi : V' \to V$ such that if $\{u, v\} \in E'$ then $\{\phi(u), \phi(v)\} \in E$.

A path is a set of vertices that can be ordered as $(v_1, v_2, \dots v_k)$ such that $\{v_i, v_{i+1}\} \in E, \forall 1 \leq i < n$. The start $(v_1)$ and end $(v_k)$ vertices in a path are called *source* and *destination*, respectively. A cycle is a path with the additional edge $\{v_1, v_k\}$.

A *path graph* is a graph formed by a single path and a *cycle graph* is a graph formed by a single cycle. A *tree* is a graph with no cycles. A *chordal graph* is a graph where every cycle with 4 or more vertices has an additional edge connecting two of its vertices (a chord). In a (square) *grid graph*, each vertex $v_{i,j}$ can be assigned to positive integer positions $(i, j)$ of the 2-D plane and there is an edge $\{v_{i,j}, v_{i',j'}\}$ if either (1) $i = i'$ and $|j - j'| = 1$ or (2) $j = j'$ and $|i - i'| = 1$. A grid graphs are a special case of a *lattice* or *mesh graph*, which is any graph that forms a regular tiling in $\mathbb{R}^n$.

A *simple graph* is a graph that is undirected, unweighted, and does not contain loops or multiple edges. If multiple edges are allowed, the graph is

called a *multigraph*.

A *directed graph* is a generalization of graphs where edges are ordered tuples $e_i = (u, v)$. Vertices in directed graphs have *in-neighbors* $N_i(v) = \{u \in V | (u, v) \in E\}$ and *out-neighbors* $N_o(v) = \{u \in V | (v, u) \in E\}$, with corresponding definitions for *in-degree*, *out-degree*, and paths.

A *weighted graph* $G = (V, E, W)$ is another generalization of graphs where $W : E \to \mathbb{R}$ is an edge weighting function. For convenience, we will use the notation $w_{u,v} = W(u, v)$. Weights can represent either costs or similarities. There are intuitive generalizations for most definitions for simple graphs. For instance, the *weighted degree* is defined as $d(v) = \sum_{u \in N(v)} w_{u,v}$.

## Some Graph Problems

*Search problems* requires traversing the vertices $V$ reachable from a source vertex $v$ through a path. Search algorithms are building blocks of many other graph algorithms, such as *cycle detection, planarity testing, max-flow*, etc. Search algorithms vary by the visiting order of vertices, which can be exploited in machine learning on graphs to generate different types of node embeddings [2].

A *Minimum Spanning Tree (MST)* of a graph $G = (V, E)$ is a tree $G' = (V, E')$ with minimum cost $c = \sum_{\{u,v\} \in E'} w_{u,v}$. The *minimum spanning tree problem* consists of computing an MST from $G$. As we will see later in the course, trees play a key role in graphical models.

A *shortest path* $P$ between vertices $u$ and $v$ is a path $(v_1, \dots v_k)$ with minimum cost $c = \sum_{u,v \in P} w_{u,v}$. The *shortest path problem* consists of computing a shortest path between a pair of vertices in $G$. Shortest paths are of interest in network science as they reveal relevant information not only about specific pairs of nodes but the graph as a role. For instance, the *diameter* of a graph is its longest shortest path.

## Complexity Analysis

The usual way to analyze the expected amount of resources (running time, space, communication, etc.) needed by an algorithm is using *asymptotic notation*. It gives how fast the need for resources grows as a function of the input. For machine learning algorithms, modern hardware, and typical applications, running time is the most important cost considered. It is not uncommon to assume that the entire dataset fits into the main memory and, when that is not the case, sampling or distributed algorithms are popular alternatives.

There are different types of asymptotic notation with different purposes (e.g., $\Theta$, $\Omega$, $O$, $\omega$, $o$). We will focus on *worst-case analysis* using *Big O*-notation. We say that a function $f(n)$ is $O(g(n))$ if there exist real constants $c, n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$. Here, $n$ is the "size" of the input (e.g. number of vertices/edges in the graph) and $g(n)$ will be the time/memory/communication cost of running the algorithm with the given input.

# Graph Algorithms

*Breadth-first search* (*BFS*) is a classical algorithm for graph search. It receives the input graph $G$ and a source vertex $s$. The algorithm visits every vertex reachable by $s$ in $G$. The visiting order is such that vertices at $k$ hops from $s$ are discovered before those at $k+1$ hops. We show the pseudocode for BFS in Algorithm 1. The time complexity of the algorithm is $O(n+m)$. Lines 1-3 runs in time $O(n)$ and lines 6-14 runs in time $O(m)$.

**Algorithm 1** $BFS(\text{graph } G, \text{source } s)$

```
for v ∈ V do
  visited[v] ← false
end for
Q ← empty queue
Q.insert(s)
while Q ≠ ∅ do
  v ← Q.get()
  for u ∈ N(v) do
    if visited[u] = false then
      visited[u] ← true
      Q.insert(u)
    end if
  end for
end while
```

*Depth-first search* (*DFS*) is another algorithm for searching a graph. Similar to *BFS*, it visits the set of vertices reachable by a source vertex $s$ in $G$. The pseudocode for the algorithm is shown in Algorithms 2 and 3. The time complexity of *DFS* is $O(n+m)$. Lines 1-3 in Algorithm 2 takes $O(n)$ time and *DFS-visit* takes $O(m)$ times—each edge is discovered twice.

**Algorithm 2** $DFS(\text{graph } G, \text{source } s)$

```
1: for v ∈ N(s) do
2:     visited[v] ← false
3: end for
4: DFS-visit(s)
```

The *Kruskal's* algorihtm is a *greedy* algorithm for computing MSTs. Its pseudocode is shown in Algorithm 4. The idea is to initialize the solution as a forest where each vertex is its own tree. Edges are added to the solution one by one. The select edge must be one with lowest cost for which the addition does not violate the tree propety—i.e not creating a loop in the current forest. Possible violations can be checked efficiently using a *disjoint set data structure* (a.k.a union-find) with operations *make-set*, *find-set* and *union*. In particular, *find-set* identifies a representative vertex in a set and *union* merges two sets, both in

**Algorithm 3** $DFS\text{-}visit(\text{graph } G, \text{vertex } v)$

```
1: for u ∈ N(v) do
2:    if visited[u] = false then
3:       DFS-visit(G, u)
4:    end if
5: end for
6: visited[v] ← true
```

time $O(\log(n))$. The total time complexity of the algorithm is $O(m \log(n))$, as the loop from lines 4-10 runs $m$ times.

**Algorithm 4** $Kruskal's(\text{graph } G = (V, E, W))$

```
1: T ← ∅
2: S ← make-set(V)
3: sorted-edges ← edges in E sorted by nondecreasing weights W
4: for e ∈ [1, ... m] do
5:    (u, v) ← sorted-edges_e
6:    if S.find-set(u) ≠ S.find-set(v) then
7:       T ← T ∪ {(u, v)}
8:       S.union(S.find-set(u), S.find-set(v))
9:    end if
10: end for
11: return T
```

*Prim's* algorithm is also greedy. It starts with a tree containing a single arbitrary vertex $r$. Then, at each step, the cheapest edge connecting the current tree and a vertex not in the tree is selected. The main steps of the algorithm are shown in Algorithm 5. The cheapest candidate edge can be identified efficiently—time $O(\log(n))$—using a min-priority queue $Q$. Besides the edge costs (values), the queue also stores the endpoint of the corresponding edge in the current tree ($Q.pred$). The loop in lines 7-15 is executed $O(m)$ times and thus the total time complexity is $O(m \log(n))$.

The last graph algorithm we will cover here is *Dijkstra's*, which computes the shortest paths from a source vertex $r$ to every vertex in the graph. The algorithm assumes that edge weights/costs are nonnegative. A key property applied by efficient algorithms for shortest paths is the *optimal substructure property*. It says that *subpaths of the shortest paths should also be the shortest*. We will first describe the *Relax* function (Algorithm 6). It tries to improve the current shortest path $d[v]$ from the source to $v$ by going through a neighbor $u \in N(v)$, if that is the case, the predecessor $pred[v]$ of $v$ in the shortest path is updated to $u$. Algorithm 7 describes the full algorithm. The shortest paths and predecessors are initialized as $\infty$ and *null*, respectively. At each step, the algorithm selects a vertex $u$ with minimum distance $d[u]$ to the source using a min-priority queue $Q$. Notice that in the first iteration, $Q.get\_min()$ will return

**Algorithm 5** $Prim's(\text{graph } G = (V, E, W), \text{root } r)$

```
1: Q ← empty min-priority queue
2: for v ∈ V do
3:    Q.value(v) ← ∞
4:    Q.pred(v) ← null
5: end for
6: Q.value(r) ← 0
7: while Q ≠ ∅ do
8:    v ← Q.min-value()
9:    for u ∈ N(v) do
10:      if u ∈ Q and Q.value(u) > w_{u,v} then
11:         Q.value(u) ← w_{u,v}
12:         Q.pred(u) ← v
13:      end if
14:   end for
15: end while
16: T ← {(v, Q.pred(v))|v ∈ V − {r}
17: return T
```

the source $s$ itself. Moreover, the algorithm maintains the invariant the $SP$ will always contain the shortest subpath starting from the source $s$. In each iteration, the function $Relax$ is applied to every edge adjacent to $u$. The running time of the algorithm is $O(m + n\log(n))$ if the min-priority queue is implemented using a *fibonacci heap*—with $O(\log(n))$ amortized time for $Q.get\_min()$.

**Algorithm 6** $Relax(u, v, W)$

```
1: if d[v] > d[u] + w(u, v) then
2:    d[v] = d[u] + w(u, v)
3:    pred[v] = u
4: end if
```

# References


[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[2] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

**Algorithm 7** $Dijkstra's(\text{graph } G = (V, E, W), \text{source } s)$

```
1: for v ∈ V do
2:     d[v] ← ∞
3:     pred[v] ← null
4: end for
5: d[s] ← 0
6: Q ← V
7: SP ← ∅
8: while Q ≠ ∅ do
9:     u ← Q.get_min()
10:    SP ← SP ∪ u
11:    for v ∈ N(u) do
12:        Relax(u, v, W)
13:    end for
14: end while
15: return SP
```