Graphs - Recitation 9

December 28, 2022

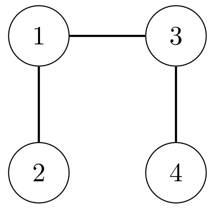
1 Graphs

This is an important section, as you'll be seeing graph's A LOT, both in this course and in courses to follow.

1.1 Definitions, Examples and Basics

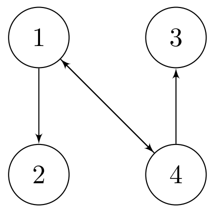
Definition 1. *A* ***non-directed graph*** *G is a pair of two sets - $G = (V, E)$ - V being a set of vertices and E being a set of couples of vertices, which represent edges ("links") between vertices.*

Example: $G = (\{1,2,3,4\},\ \{\{1,2\},\{1,3\},\{3,4\}\})$ is the following graph:



Definition 2. *A* ***directed graph*** *G is a pair of two sets - $G = (V, E)$ - V being a set of vertices and $E \subseteq V \times V$ being a set of directed edges ("arrows") between vertices.*

Example: $G = (\{1,2,3,4\},\ \{(1,2),(1,4),(4,1),(4,3)\})$ is the following graph (note that it has arrows):



Definition 3. *A* ***weighted graph*** *composed by a graph $G = (V, E)$ (either non-directed or directed) and a wight function $w : E \to \mathbb{R}$. Usally (but not necssery) we will think about the quantity $w(e)$ where $e \in E$ as the length of the edge.*

Now that we see graphs with our eyes, we can imagine all sorts of uses for them... For example, the can represent the structure of the connections between friends on facebook, or they can even represent which rooms in your house have doors between them.

Remark 1. *Note that directed graphs are a* ***generalization*** *of non-directed graphs, in the sense that every non-directed graph can be represented as a directed graph. Simply take every non-directed edge $\{v, u\}$ and turn it into two directed edges $(v, u), (u, v)$.*

Remark 2. *Note that most of the data structures we discussed so far - Stack, Queue, Heap, BST - can all be implemented using graphs.*

Now let's define some things in graphs:

Definition 4. *(Path, circle, degree)*

1. *A* ***simple path*** *in the graph G is a series of unique vertices (that is, no vertex appears twice in the series) $v_1, v_2, ..., v_n$ that are connected with edges in that order.*

2. *A* ***simple circle*** *in the graph G is a simple path such that $v_1 = v_n$.*

3. *The* ***distance*** *between two vertices $v, u \in V$ is the length of the shortest path between them (∞ if there is no such path).*

Remark 3. *Note that for all $u, v, w \in V$ the triangle inequality holds regarding path lengths. That is:*

$$dist(u, w) \leq dist(u, v) + dist(v, w)$$

Definition 5. *(connectivity)*

1. *Let $G = (V, E)$ be a non-directed graph. A* ***connected component*** *of G is a subset $U \subseteq V$ of maximal size in which there exists a path between every two vertices.*

2. *A non-directed graph G is said to be a* ***connected*** *graph if it only has one connected component.*

3. *Let $G = (V, E)$ be a directed graph. A* ***strongly connected component*** *of G is a subset $U \subseteq V$ of maximal size in which for any pair of vertices $u, v \in U$ there exist both directed path from u to v and a directed path form v to u.*

Proposition 1. *Let $G = (V, E)$ be some graph. If G is connected, then $|E| \geq |V| - 1$*

Proof. We will perform the following cool process: Let $\{e_1, ..., e_m\}$ be an enumeration of E, and let $G_0 = (V, \emptyset)$. We will build the graphs $G_1, G_2, ...G_m = G$ by adding edges one-by-one. Formally, we define -

$$\forall i \in [m] \quad G_i = (V, \{e_1, ..., e_i\})$$

G_0 has exactly $|V|$ connected components, as it has no edges at all. Then G_1 has $|V| - 1$. From there on, any edges does one of the following:

1. Keeps the number of connected components the same (the edge closes a cycle) '

2. Lowers the number of connected components by 1 (the edges does not close a cycle)

So in general, the number of connected components of G_i is $\geq |V| - i$. Now, if $G_m = G$ is connected, it has just one connected component! This means:

$$1 \geq |V| - |E| \implies |E| \geq |V| - 1$$

$\square$

1.2 Graph Representation

Okay, so now we know what graphs are. But how can we represent them in a computer? There are two mainly options. The first one is by **array of adjacency lists**. Given some graph G, every slot in the array will contain a linked list. Each linked list will represent a list of some node's neighbors. The second option is to store edges in an **adjacency matrix**, a $|V| \times |V|$ binary matrix in which the v, u-cell eqauls 1 if there is an edge connecting u to v. In the example below the matrix is denoted by A_G. Note that the running time anlysis might depands on the underline representaion.

Question. What is the memmory cost of each of the representations? Note that while holding an adjacency matrix requires to store $|V|^2$ bits regardless the size of E, Mainting the edges by adjacency lists costs linear memmory at number of edges and therefore only $\Theta(|V| + |E|)$ bits.

Example. Consider the following directed graph:

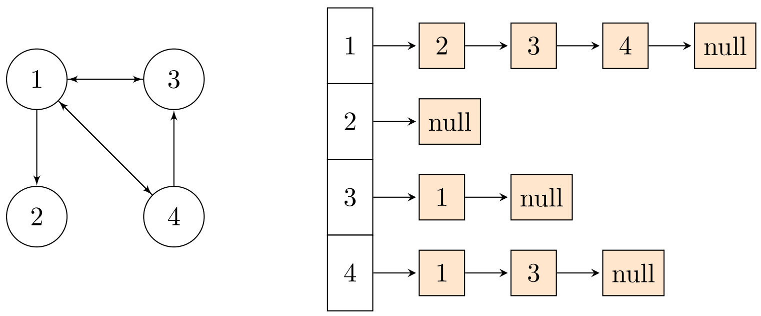
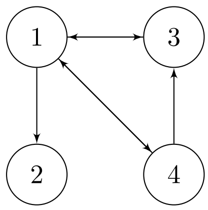


Figure 3: Persenting G by array of adjacency lists.



$$A_G = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 4: Presenting G by adjacency matrix.

1.3 Breadth First Search (BFS)

One natural thing we might want to do is to travel around inside a graph. That is, we would like to visit all of the vertices in a graph in some order that relates to the edges.
One reason to want such a thing is the following objective: Given some start vertex v, we would like to know how long is the shortest path from s to every vertex $v \in V$.
For this we present the following algorithm, which receives the connected graph G and the starting node s as input. Note that in addition to our definition of the graph G, we will now also save some property *dist* for every vertex, and also a flag called *visited*.

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever the search discovers a white vertex v in the course of scanning the adjacency list of a gray vertex u ,the vertex v and the edge $(u,\ v)$ are added to the tree. We say that u is the predecessor or parent of v in the breadth-first tree. Since every vertex reachable from s is discovered at most once, each vertex reachable from s has exactly one parent. (There is one exception: because s is the root of the breadth-first tree, it has no parent.) Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on the simple path in the tree from the root s to vertex v ,then u is an ancestor of v and v is a descendant of u. The breadth-first-search procedure BFS on the following page assumes that the graph $G = (V, E)$ is represented using adjacency lists. It denotes the queue by Q, and it attaches three additional attributes to each vertex v in the graph:

1. v.color is the color of v: WHITE, GRAY, or BLACK.
2. v.d holds the distance from the source vertex s to v,as computed by the algorithm.
3. $v.\pi$ is v's predecessor in the breadth-first tree. If v has no predecessor because it is the source vertex or is undiscovered, then $v.\pi$ NIL.

BFS(G, s)

```
1  for v ∈ V do
2      set v.dist ← ∞, v.visited ← False
3  set s.dist ← 0
4  Q ← new Queue
5  Q.Enqueue(s)
6  s.visited ← True
7  while Q is not empty do
8      u ← Q.Dequeue()
9      for neighbor w of u do
10         if w.visited is False then
11             w.visited← True
12             w.dist ← min(w.dist, u.dist+1)
13             Q.Enqueue(w)
```

Example: In the tirgul video...

Correctness: The example should be enough to explain the correctness. A concrete proof can be found in the book, page 597.

Runtime: We can analyse the runtime line-by-line:

- Lines 1-2: $|V|$ operations, all in $O(1)$ runtime, for a total of $O(|V|)$.
- Lines 3-6: $O(1)$
- Lines 7-8: First we need to understand the number of times the *while* loop iterates. We can see that every vertex can only enter the queue ONCE (since it is then tagged as "visited"), and therefore it runs $\leq |V|$ times. All operations are $O(1)$, and we get a total of $O(|V|)$.
- Lines 9-13: Next, we want to understand the number of times this for loop iterates. The for loop starts iterating once per vertex, and then the number of its iterations is the same as the number of neighbours that this vertex has. Thus, it runs $O(|E|)$ times.

So all in all we get a runtime of $O(|V| + |E|)$

1.4 Usage of BFS

Now we have in our hands a way to travel through a graph using the edges. How else can we use it?

Exercise: Present and analyse an algorithm $CC(G)$ which receives some undirected graph G and outputs the number of connected components in G in $O(|V| + |E|)$.

Solution: Consider the following algorithm: (Now read the algorithm, it may be in the next page because LaTeX is dumb...)

CC(G)

```
count ← 0
for v ∈ V do
    if v.visited = False then
        count←count+1
        BFS(G, v)
return count
```

Correctness: We know that given some vertex $v \in V$, $BFS(G, v)$ marks all of the nodes in the connected component of v as marked (Since BFS marks all vertices in a connected graph). So each time that BFS is invoked, it is because some vertex v has not been visited, meaning it is not in the same connected component as any of the previous vertices.

Runtime: Let's denote all of the connected components in G by $(G_i = (V_i, E_i))_{i=1}^{k}$. For each connected component, the runtime of BFS will be $O(|V_i| + |E_i|)$. So all in all, we get $O(|V| + |E|)$.

1.5 Correctness

We must not forget to prove BFS's correctness! Specifically, we want to prove that after calling $BFS(G, s)$, we have that for all nodes v in s's connected component, v.visited = True. We will prove something stronger:

Proposition 2. *Let $G = (V, E)$ and let $s \in V$. For each $v \in V$ denote $d_v = dist(s, v)$. After $BFS(G, s)$ is called, all nodes with $d_v = k$ are inserted to the queue, and they are insterted before all nodes with $d_v = k+1$.*

Proof. We prove this by induction on k.

Base: $k = 0$. Only $d_s = 0$, and it is in fact inserted first.

Hypothesis: Assume correctness for all $l < k$.

Step: Let v be a node such that $d_v = k$ (if there is no such v we are done). By I.H for $k - 1$, we know that all nodes u with $d_u = k - 1$ were inserted into the queue by now. By definition of the distance function, there exists some node u_0 such that $d_{u_0} = k - 1$ and $(u_0, v) \in E$. When u_0 is dequeued, v is inserted into the queue.

So far we have proved that all nodes of distance k are eventually inserted to the queue. We still need to show that this is done before all nodes of distance $k + 1$.

Assume towards contradiction that v is inserted before w, where $d_v = k + 1$, $d_w = k$. If v was accessed through u and w was accessed through x, then $d_u = k$ and $d_x = k - 1$. Since a queue is LIFO, we get that u was inserted before x, but this is a contradiction to the I.H.

□

Remark 4. *If v is not in the connected component of s then $d_v = \infty$.*

Corollary 1.1. *After $BFS(G, s)$ is run, v.visited = True for all v in the connected component of s!*

1.6 Depth First Search (DFS)

As its name implies, depth-first search searches "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until all vertices that are reachable from the original source vertex have been discovered. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, repeating the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

DFS(G)

```
DFS( G):
for v ∈ V do
    vi.visited ← False
time ← 1 for v ∈ V do
    if  not v.visited then
        π (v) v ← null
        Explore( G, v )

Explore(G, v):
for (v, u) ∈ E do
    Previsit(v) if  not u.visited then
        π (u) ← v
        Explore( G, u )
    Postvisit(v)

Previsit(v):
pre(v) ← time
time ← time +1

Postvisit (v):
post(v) ← time
time ← time +1
```

Properties of depth-first search. Depth-first search yields valuable information about the structure of a graph. Perhaps the most basic property of depth-first search is that the predecessor subgraph G_π does indeed form a forest of trees, since the structure of the depth-first trees exactly mirrors the structure of recursive calls of explore-function . That is, $u = \pi(v)$ if and only if explore(G, v) was called during a search of u's adjacency list. Additionally, vertex v is a descendant of vertex u in the depth-first forest if and only if v is discovered during the time in which u is gray. Another important property of depth-first search is that discovery and finish times have parenthesis structure. If the explore procedure were to print a left parenthesis "(u" when it discovers vertex u and to print a right parenthesis r"u)" when it finishes u, then the printed expression would be well formed in the sense that the parentheses are properly nested.

The following theorem provides another way to characterize the parenthesis structure.

Parenthesis theorem In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

1. the intervals [pre(u), post(u)] and [pre(v), post(v)] are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest.
2. the interval [pre(u), post(u)] is contained entirely within the interval [pre(v), post(v)], and u is a descendant of v in a depth-first tree, or
3. the interval [pre(v), post(v)] is contained entirely within the interval [pre(u), post(u)], and v is a descendant of u in a depth-first tree.

Proof. We begin with the case in which pre(u) $<$ pre(v). We consider two subcases, according to whether pre(v) $<$ post(u). The first subcase occurs when pre(v) $<$ post(u), so that v was discovered while u was still gray, which implies that v is a descendant of u. Moreover, since v was discovered after u, all of its outgoing edges are explored, and v is finished, before the search returns to and finishes u. In this case, therefore, the interval [pre(v), post(v)] is entirely contained within the interval [pre(u), post(u)]. In the other subcase, post(u) < pre(v), and by defintion, pre(u) < post(u) < pre(v) < post(v), and thus the intervals [pre(u), post(u)] and [pre(v), post(v)] are disjoint. Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.

Corollary. Nesting of descendants' intervals. Vertex v is a proper descendant of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if pre(u) < pre(v) < post(v) < post(u).