Advanced Algorithms: Lecture 1

Contents

1 Introduction to graphs **1**

2 Representation of Graphs: Adjacency Matrix and Adjacency List **3**

3 Graph traversals **4**

4 Greedy Algorithms **5**

5 Solution of the exercises **9**

1 Introduction to graphs

A graph $G = (V, E)$ consists of a finite set of vertices V and a finite set of edges E. An edge represents a binary relation between the vertices. Although there are many varieties of graph concepts studied in the literature, two main ones will be used throughout this course. These correspond to graphs whose edges are directed or undirected. Graphs with directed edges are called directed graphs or simply, digraphs. Graphs with undirected edges are called undirected graphs. In an *undirected graph* an edge *unordered* pair of vertices $\{u, v\}$. While in a *directed graph* an edge is an *ordered* pair of vertices (u, v). Observe that you can have directed edges of the form (a, a), these edges are called *loops*. In this course we will consider only directed graphs without loops, hence we will consider only edges (u, v) with distinct u, v.
We will say that an edge (u, v) is *incident* to u and v and the vertices u and v are called *adjacent*.
A graph is usually depicted visually, by drawing the elements of the vertices set as boxes or circles, and drawing the elements of the edge set as lines or arcs between the boxes or circles. In Fig. 1 we present two different drawings of the same graph $G = (V, E)$ with $V = \{a, b, c, d, e\}$ and $E = \Big\{\{a, b\}, \{b, c\}, \{c, d\}, \{d, e\}, \{a, e\}\Big\}$.

A graph that can be drawn in a plan such that the edges intersect only in the endpoints is called *planar*. The graph in Fig. 1 is planar as its representation on the left has no crossing edges.
Given a graph G, the *degree* of a vertex v in G (denoted by $d_G(v)$) is the number of edges incident to it. In symbols for a vertex $v \in V$ the degree $d_G(v)$ is defined as $d_G(v) = |\{u : \{u, v\} \in E\}|$.
If G is directed we define the *out-degree* (*in-degree*) of of a vertex v as follows:

$$d^+_G(v) = |\{u : (v, u) \in E(G)\}| \qquad \text{and} \qquad d^-_G(v) = |\{u : (u, v) \in E(G)\}|$$

For a graph $G = (V, E)$ we usually refer by n to the number of vertices and m the number of edges. In other words, $n = |V|$ and $m = |E|$.

Figure 1: The same graph represented in two different drawings.

Exercise 1.1 *For a graph $G = (V, E)$ on n vertices (i.e. $|V| = n$) what is the minimum and the maximum degree that a vertex can have? Ans.*

A *walk* in a graph is a sequence $v_1, e_1, \ldots, e_{k-1}, v_k$ of alternating vertices and edges, beginning and ending with vertices, such that for each $1 \leq i \leq k$ the edge e_i has endpoints v_i and v_{i+1}.
A *path* is a walk with no repeated vertices.
A *cycle* is a path starting and ending in the same vertex. Observe that the graph depicted in Fig. 1 is a cycle.

Connectivity Imagine a map of a country in which cities are connected by roads, railways, etc. Its main purpose is to show whether and how to travel from one place to another. This is related to an important graph property: the connectivity.
Two vertices u and v in a graph G are said *connected* if there exists a path starting from u and ending in v in G. A graph is connected if every two of its vertices are connected.

Exercise 1.2 *Show that deleting any edge that belongs to a cycle in a graph does not disconnect the graph.*

A graph that is not connected is called *disconnected*. A disconnected graph can be "broken" into connected graphs called *components*.

Subgraphs of a graph Consider $G = (V, E)$ we say that $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. *A spanning subgraph* of G is a subgraph that is connected and contains all the vertices of G. The graph $G' = (V', E')$ is an *induced subgraph* of G if $V' \subseteq V$ and E' contains all the edges in E whose endpoints are in V'. For an example see Fig. 2.

Important graphs Here we see some important types of graphs. An *empty* graph is a graph with no vertices and no edges (that is $E = V = \emptyset$). A *complete* graph (sometime also called a *clique*) is a graph that has all the edges between vertices. A complete graph on n vertices is usually denoted by K_n. A graph that has no cycle is called *acyclic*.

Exercise 1.3 *How many edges does K_n have? Ans.*

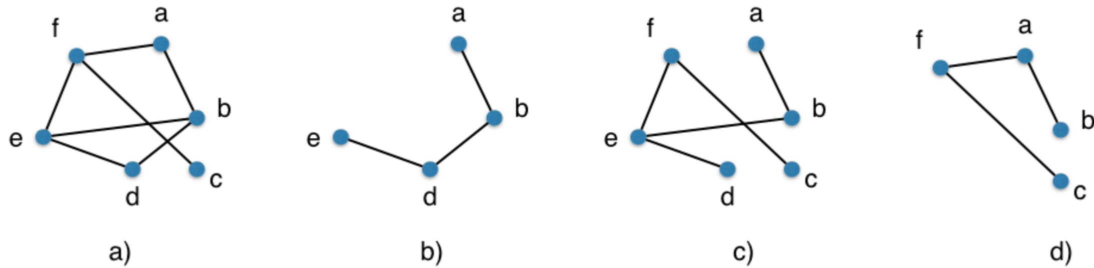

Figure 2: a) A graph G, b) A subgraph of G, c) A spanning subgraph of G, d) An induced subgraph of G.

Imagine a city in winter where all the streets were blocked by snow. To assure the travelling the streets have to be cleaned. However, the city cannot afford economically to clean all of them, however it wants to clean all the streets so that it is possible to travel by vehicle between all places in the city along cleaned streets. How can we do this?

Clearly this problem can be modeled as a graph. We want to find a subgraph that is connected and has no cycle. This because as we have seen in Exercise 1.2 deleting an edge from a cycle does not disconnect the graph. Hence, the street that corresponds to that edge is not necessary useful for reaching some place. In fact to find the best solution we have to look for a spanning subgraph that is acyclic and connected. This is called a tree.
A graph that is connected and has no cycle is called a *tree*. We will usually denote a tree by T rather than by G. Sometimes we will need to distinguish a vertex in a tree, in that case we will talk about *rooted trees* and the distinguished vertex will be called *a root*. The *level* of a vertex in a rooted tree is the number of edges from that vertex to the root. The *height* of a rooted tree is the maximum level of any vertex in the tree.
A *leaf* is a vertex of degree 1.

Exercise 1.4 *Show that every treee has a leaf. Ans.*

2 Representation of Graphs: Adjacency Matrix and Adjacency List

The two main graph representations we use when talking about graph problems are the adjacency list and the adjacency matrix.

Adjacency Matrix A graph $G = (V, E)$ on n vertices can be represented as a $n \times n$ matrix M such that $M[i, j] = 1$ if $\{v_i, v_j\} \in E$ and $M[i, j] = 0$ otherwise.
Observe that an adjacency matrix takes up $\Theta(n^2)$ storage.

Adjacency List An adjacency list is a vector of lists. Each cell in the vector corresponds to a vertex v and contains a list of edges $\{u, v\}$ that are incident to u. Thus, an adjacency list takes up $\Theta(n + m)$ space.

Exercise 2.1 *Try to answer the followings:*

- *Thinking in terms of storage when is more convenient to store a graph as an adjacency list? an adjacency matrix?*

- *How much time does it take to check whether an edge is present in a graph, if the graph is stored using: (a) adjacency list, (b) adjacency matrix?*

3 Graph traversals

The myth of Theseus and the Minotaur is one of the most known myths of the Greek Mythology. King Minos of Crete had built a giant maze, a Labyrinth, and, at the centre of the maze, he kept a terrifying creature, - the Minotaur. The humans that enter the labyrinth would get lost in the corridors of the maze until eventually die from hunger or from the Minotaur. A graph traversal is a systematic way to visit every node of the graph. In this section we explore two main methods to traverse a graph, namely *Breadth First Search* (BFS) and *Depth First Search* (DFS).

Breadth First Search Algorithm

Algorithm$BFS(G, s)$
Require: A graph $G = (V, E)$ and a vertex s
Ensure: Returns in the vector $parent[]$, the edges of the spanning tree produced.

```
 1: /* Initialization part */
 2: for all u ∈ V do
 3:     state[u] ← "unvisited"; /* will contain the status of a vertex*/
 4:     parent[u] ← NULL; /* will contain the parent of u in the visit */
 5: end for
 6: state[s] ← "visited";
 7: p[s] ← "root";
 8: ADD(Q, s); /* The queue Q will be handled in a FIFO (first in first out) manner */
 9: while Q is NOT EMPTY do
10:     u ← Head(Q); /* this operation also removes u from Q */
11:     for all v ∈ Adjacent(u) do
12:         if state[v] == "unvisited" then
13:             state[v] ← "visited";
14:             ADD(Q, v);
15:             parent[v] ← u;
16:         end if
17:     end for
18: end while
19: return parent[.];
```

Figure 3: Algorithm $BFS(G, s)$

Depth First Search Algorithm

Algorithm$DFS(G, s)$

Require: A graph $G = (V, E)$ and a vertex s

1: /* Initialization part */
2: **for all** $u \in V$ **do**
3: $state[u] \leftarrow$ "*unvisited*"; /* will contain the status of a vertex*/
4: **end for**
5: $p[s] \leftarrow$ "*root*";
6: $DFS_Visit(G, s)$

Figure 4: Algorithm $DFS(G, s)$

Algorithm$DFS_Visit(G, u)$

1: $state[u] \leftarrow$ "*visited*";
2: **for all** $v \in Adj(u)$ **do**
3: **if** $state[v] ==$ "*unvisited*" **then**
4: $state[v] \leftarrow$ "*visited*";
5: $DFS_VISIT(G, v)$
6: **end if**
7: **end for**

Figure 5: Algorithm $DFS_Visit(G, s)$

4 Greedy Algorithms

Both the DFS and the BFS visits on a graph produce a spanning tree. However sometimes we may have weights on the edges of a graph. A simple example is the one when the graph represents a map of a city (i.e. vertices are places and edges are streets connecting places) and the weight of an edge is simply the length of the street in km. More formally, an edge-weighted graph $G = (V, E, w)$ with $w : E \rightarrow R$ being the weight function. In this case we may be interested to find a spanning tree that has minimum total weight.

Definition 4.1 *Given an edge-weighted graph $G = (V, E, w)$ and a subgraph G', the weight of G', denoted as $w(G')$, is given by:*

$$w(G') = \Sigma_{e \in E(G')} w(e)$$

The *minimum spanning tree* (MST) of a graph G is the spanning tree of minimum weight.
Observe that both the DFS and the BFS will give you a spanning tree, but this may not be a MST.
In this section we will show two greedy algorithms that find a MST for a graph. Generally speaking a greedy algorithm produces a solution by making at each step the choice that it seems optimal. Note that this does not necessarily guarantee that the optimal choices that are done locally will lead to an optimal global solution.
As we have edge-weighted graphs we may think that we can have an array E that contains all the edges of G and for each edge $e = \{x, y\}$ it has three parts: $E[e].a, E[e].b, E[e].w$, which represent x, y and $w(e)$, respectively .

Kruskal's Algorithm

The main idea of the Kruskal's algorithm (1956) is the following: At each step of the algorithm choose an edge $e \in E$ of minimum cost. Then if e does not create cycles with the edges that we have already chosen (*i.e.* the edges in SOL) then add e to the current solution (*i.e.* the edges in $SOL \leftarrow SOL \cup \{e\}$). In implementing the algorithm it is efficient to order the edges initially in a non decreasing order, thus at each step I can pick the first edge of the list. This will require $O(m \log m)$ time. After this we have to check if e forms a cycle with the edges we have already taken. We could do a BFS/DFS to check for cycles, however a more elegant way to do this is the following: observe that the edges of SOL determine connected components of G, thus an edge $e = \{x, y\}$ does not create a cycle if and only if x and y belong to different components. To keep the information about this components we use a vector CC, such that for all i, $CC[i]$ is the label of the connected component determined by SOL where i belongs. As a label of a connected component we could use the label of a vertex. In Algorithm 6 we present the pseudocode. To analyse the time complexity of the algorithm notice that the ordering of the edges is done once and takes time $O(m \log m)$. Then every iteration of the second FOR cycle, requires time : (a) $O(1)$ if the edge creates cycle, (b) $O(n)$ if the edge does not create cycle, as the result of the merging of the components. Note that, it can happen at most $n-1$ times that the edge we check does not create cycle, so the iterations that require time $O(n)$ are $n-1$. Hence the total time of the FOR is $O(n^2)$, thus the total time for this algorithm will be $O(m \log m + n^2)$.

Algorithm$Kruskal(G)$
Require: A graph G /* A connected edge weighted graph G*/
Ensure: Returns a minimum spanning tree.

1: /* Initialization part */
2: $SOL \leftarrow \emptyset$
3: $E \leftarrow$ array containing all the edges of G and for each edge $e = \{x, y\}$ it has three parts: $E[e].a, E[e].b, E[e].w$
4: ORDER E according to the weights in non decreasing order
5: **for all** $u \in V$ **do**
6: $CC[u] \leftarrow u$; /* initially each vertex is in a single connected component composed by itself */
7: **end for**
8: **for all** $i = 1$ to m **do**
9: $\{u, v\} \leftarrow \{E[i].u, E[i].v\}$; /* gets the edge of minimum cost, i.e. the first of the list */
10: /* if the edge does not create cycles */
11: **if** $CC[u] \neq CC[v]$ **then**
12: $SOL \leftarrow SOL \cup \{\{u, v\}\}$;
13: $c \leftarrow CC[v]$;
14: **for all** $w \in V$ **do**
15: **if** $CC[w] == c$ **then**
16: $CC[w] \leftarrow CC[u]$;/* we have to merge the two components $CC[u]$ and $CC[v]$ */
17: **end if**
18: **end for**
19: **end if**
20: **end for**
21: **return** SOL;

Figure 6: Algorithm $Kruskal(G)$

Prim's algorithm

The algorithm of Prim was discovered on 1957. The main idea is the following: At each step, the algorithm tries to find the edge of minimum cost that extends the tree constructed in the previous steps. In Algorithm 7 we present the pseudocode. Concerning the complexity of the algorithm we need to specify how to implement the procedure to find an edge $\{u, v\}$ of minimum weight among the edges with $u \in C$ and $v \notin C$. Notice that this is equivalent with finding the vertex $v \notin C$ such that by adding v to C through the edge $\{u, v\}$ with $u \in C$ we obtain a locally optimal solution. Indeed, if we define the cost of a vertex $v \notin C$ as the minimum weight of an edge that connects v to some vertex in C, then it is clear that we try to find a vertex of minimum weight that extends the tree. This is very similar to the Dijkstra algorithm and hence we can use the same data structures we used there (e.g. Fibonaccci heaps). In a very simple way we could use an array to keep the costs of the vertices. Finding a vertex of minimum weight will cost $O(n)$ and hence the complexity of this implementation will be $O(n^2)$.

Algorithm$Prim(G)$
Require: A graph G /* A connected edge weighted graph G*/
Ensure: Returns a minimum spanning tree.

1: $SOL \leftarrow \emptyset$
2: Choose a vertex s in G;
3: $C \leftarrow s$
4: /* we continue until we have a spanning tree */
5: **while** $C \neq V$ **do**
6:     let $\{u, v\}$ be the edge of minimum weight among the edges with $u \in C$ and $v \notin C$
7:     $SOL \leftarrow SOL \cup \{\{u, v\}\}$
8:     $C \leftarrow C \cup \{v\}$
9: **end while**
10: **return** SOL;

Figure 7: Algorithm $Prim(G)$

We mentioned that not always the greedy approach leads to an optimal solution. However both the algorithms of Kruskal and Prim guarantee that the solution produces is optimal. In order to illustrate how to prove this, we show the following theorem

Theorem 4.1 *Given an edge-weighted graph G, the algorithm of Prim, correctly produces a MST of G.*

Proof: First we need to show that the algorithm finishes in a finite number of steps. This is true as at each step of the WHILE, we add a vertex to C and as the set V is finite this process will finish. Note that if $C \neq V$, and the graph is connected then it always exists an edge $\{u, v\}$ with $u \in C, v \notin C$, hence it is always possible to find a vertex $v \notin C$ to be added.
Now we need to show in each step of the algorithm we are extending our partial solution to an optimal one. In other words for each step of the algorithm $h = 0, \ldots, k$ there exists an optimal solution SOL^*, such that if SOL_h is the solution produced by the algorithm at the end of step h, $SOL_h \subseteq SOL^*$. We prove this by induction on h. For $h = 0$ the claim is trivially true as $SOL_0 = \emptyset \subseteq SOL^*$. Suppose the claim holds for all the steps from 0 to h, that is $SOL_h \subseteq SOL^*$, we prove that it holds also for step $h + 1$. In other words we need to show that $SOL_{h+1} \subseteq SOL^*$. From the induction hypothesis there exists an optimal solution SOL^* such that $SOL_h \subseteq SOL^*$. If $SOL_{h+1} \subseteq SOL^*$ then we are done. Otherwise this means that the edge added

by the algorithm in the step $h+1$ is not present in SOL^*. Let $\{u, v\}$ be this edge. Thus, $\{u, v\} \in SOL_{h+1}$ and $\{u, v\} \notin SOL^*$. Consider now the graph formed by the edges in $SOL^* \cup \{\{u, v\}\}$. As SOL^* induces a spanning tree, adding $\{u, v\}$ creates necessarily a cycle that passes through $\{u, v\}$. We denote this cycle $u = x_1, \dots, x_k = v$. We know that as the algorithm chooses $\{u, v\}$, $u \in C_h$ and $v \notin C_h$. Starting from $u = x_1$ let x_i be the first vertex such that $x_i \notin C_h$. Clearly x_i exists as in the worst case it will be v. Then we know that $\{x_{i-1}, x_i\} \in E$ and $x_{i-1} \in C$ and $x_i \notin C$. Hence, $\{x_{i-1}, x_i\}$ is an edge that the algorithm must have considered during the iteration $h+1$. Thus, $w(\{x_{i-1}, x_i\}) \geq w(\{u, v\}$. If we consider the graph H formed by the edges of the set $SOL^* \cup \{\{u, v\}\} - \{\{x_{i-1}, x_i\}\}$, we have that H is a spanning tree (because we removed an edge from a cycle does not disconnect the graph) and $w(H) \leq w(SOL^*)$. Hence, H is also a minimum spanning tree. Thus, the proof is concluded as we found a minimum spanning tree H that contains all the edges of SOL_{h+1}.

□

5 Solution of the exercises

Solution 1 *Exercise 1.1* 0 *and* $n - 1$.

Solution 2 *Exercise 1.3* $n(n - 1)/2$.

Solution 3 *Exercise 1.4*

Solution 4 *Exercise* **??** *It follows as the sum of all degrees must be even.*

References

[1] J.A. Bondy and U.S.R. Murty, Graph Theory with Applications, North-Holland, 1976. (This book is out of print. You can download their personal copy from the web page: http://book.huihoo.com/pdf/graph-theory-With-applications/pdf/GTWA.pdf)