CS 341: Algorithms

Douglas R. Stinson

David R. Cheriton School of Computer Science
University of Waterloo

Winter, 2015

Table of Contents

Graphs and Digraphs

A **graph** is a pair $G = (V, E)$. V is a set whose elements are called **vertices** and E is a set whose elements are called **edges**. Each edge joins two distinct vertices. An edge can be represented as a set of two vertices, e.g., $\{u, v\}$, where $u \neq v$. We may also write this edge as uv or vu.

We often denote the number of vertices by n and the number of edges by m. Clearly $m \leq \binom{n}{2}$.

A **directed graph** or **digraph** is also a pair $G = (V, E)$. The elements of E are called **directed edges** or **arcs** in a digraph. Each arc joins two vertices, and an arc can be represented as a ordered pair, e.g., (u, v). The arc (u, v) is directed from u (the **tail**) to v (the **head**), and we allow $u = v$.

If we denote the number of vertices by n and the number of arcs by m, then $m \leq n^2$.

Data Structures for Graphs: Adjacency Matrices

There are two main data structures to represent graphs: an **adjacency matrix** and a set of **adjacency lists**.

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$. The **adjacency matrix** of G is an n by n matrix $A = (a_{u,v})$, which is indexed by V, such that

$$a_{u,v} = \begin{cases} 1 & \text{if } \{u, v\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

There are exactly $2m$ entries of A equal to 1.

If G is a digraph, then

$$a_{u,v} = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{otherwise.} \end{cases}$$

For a digraph, there are exactly m entries of A equal to 1.

Data Structures for Graphs: Adjacency Lists

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$.

An **adjacency list representation** of G consists of n linked lists.

For every $u \in V$, there is a linked list (called an **adjacency list**) which is named $Adj[u]$.

For every $v \in V$ such that $uv \in E$, there is a node in $Adj[u]$ labelled v. (This definition is used for both directed and undirected graphs.)

In an undirected graph, every edge uv corresponds to nodes in two adjacency lists: there is a node v in $Adj[u]$ and a node u in $Adj[v]$.

In a directed graph, every edge corresponds to a node in only one adjacency list.

Breadth-first Search of an Undirected Graph

A **breadth-first search** of an undirected graph begins at a specified vertex s.

The search “spreads out” from s, proceeding in **layers**.

First, all the neighbours of s are **explored**.

Next, the neighbours of those neighbours are explored.

This process continues until all vertices have been explored.

A **queue** is used to keep track of the vertices to be explored.

Breadth-first Search

Algorithm: $\mathit{BFS}(G, s)$

for each $v \in V(G)$

do $\begin{cases} colour[v] \leftarrow \textbf{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

$colour[s] \leftarrow \textbf{gray}$

$\mathit{InitializeQueue}(Q)$

$\mathit{Enqueue}(Q, s)$

while $Q \neq \emptyset$

do $\begin{cases} u \leftarrow \mathit{Dequeue}(Q) \\ \textbf{for each } v \in Adj[u] \\ \quad \textbf{do } \begin{cases} \textbf{if } colour[v] = \textbf{white} \\ \quad \textbf{then } \begin{cases} colour[v] = \textbf{gray} \\ \pi[v] \leftarrow u \\ \mathit{Enqueue}(Q, v) \end{cases} \end{cases} \\ colour[u] \leftarrow \textbf{black} \end{cases}$

Properties of Breadth-first Search

A vertex is **white** if it is **undiscovered**.

A vertex is **gray** if it has been **discovered**, but we are still processing its adjacent vertices.

A vertex becomes **black** when all the adjacent vertices have been processed.

If G is **connected**, then every vertex eventually is coloured black.

When we explore an edge $\{u, v\}$ starting from u:

- if v is **white**, then uv is a **tree edge** and $\pi[v] = u$ is the **predecessor** of v in the **BFS tree**
- otherwise, uv is a **cross edge**.

The BFS tree consists of all the tree edges.

Every vertex $v \neq s$ has a unique predecessor $\pi[v]$ in the BFS tree.

Shortest Paths via Breadth-first Search

Algorithm: $\mathit{BFS}(G, s)$

for each $v \in V(G)$ **do** $\begin{cases} colour[v] \leftarrow \textbf{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

$colour[s] \leftarrow \textbf{gray}$

$\boxed{dist[s] \leftarrow 0}$

$\mathit{InitializeQueue}(Q)$

$\mathit{Enqueue}(Q, s)$

while $Q \neq \emptyset$

$$\textbf{do} \begin{cases} u \leftarrow \mathit{Dequeue}(Q) \\ \textbf{for each } v \in Adj[u] \\ \quad \textbf{do} \begin{cases} \textbf{if } colour[v] = \textbf{white} \quad \textbf{then} \begin{cases} colour[v] = \textbf{gray} \\ \pi[v] \leftarrow u \\ \mathit{Enqueue}(Q, v) \\ \boxed{dist[v] \leftarrow dist[u] + 1} \end{cases} \end{cases} \\ colour[u] \leftarrow \textbf{black} \end{cases}$$

Distances in Breadth-first Search

If $\{u, v\}$ is **any edge**, then $|dist[u] - dist[v]| \leq 1$.

If uv is a **tree edge**, then $dist[v] = dist[u] + 1$.

$dist[u]$ is the length of the **shortest path** from s to u.

This is also called the **distance** from s to u.

Bipartite Graphs and Breadth-first Search

A graph is **bipartite** if the vertex set can be partitioned as $V = X \cup Y$, in such a way that all edges have one endpoint in X and one endpoint in Y.

A graph is bipartite if and only if it does not contain an **odd cycle**.

BFS can be used to test if a graph is bipartite:

- if we encounter an edge $\{u, v\}$ with $dist[u] = dist[v]$, then G is not bipartite, whereas
- if no such edge is found, then define $X = \{u : dist[u] \text{ is even}\}$ and $Y = \{u : dist[u] \text{ is odd}\}$; then X, Y forms a bipartition.

Depth-first Search of a Directed Graph

A **depth-first search** uses a **stack** (or **recursion**) instead of a queue.

We define predecessors and colour vertices as in BFS.

It is also useful to specify a **discovery time** $d[v]$ and a **finishing time** $f[v]$ for every vertex v.

We increment a **time counter** every time a value $d[v]$ or $f[v]$ is assigned.

We eventually visit all the vertices, and the algorithm constructs a **depth-first forest**.

Depth-first Search

Algorithm: $\mathit{DFS}(G)$

for each $v \in V(G)$

do $\begin{cases} colour[v] \leftarrow \mathbf{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

$time \leftarrow 0$

for each $v \in V(G)$

do $\begin{cases} \textbf{if } colour[v] = \mathbf{white} \\ \quad \textbf{then } \mathit{DFSvisit}(v) \end{cases}$

Depth-first Search (cont.)

Algorithm: *DFSvisit*(v)

$colour[v] \leftarrow \mathbf{gray}$
$time \leftarrow time + 1$
$d[v] \leftarrow time$
comment: $d[v]$ is the discovery time for vertex v
for each $w \in Adj[v]$

$$\textbf{do} \begin{cases} \textbf{if } colour[w] = \mathbf{white} \\ \quad \textbf{then } \begin{cases} \pi[w] \leftarrow v \\ \textit{DFSvisit}(w) \end{cases} \end{cases}$$

$colour[v] \leftarrow \mathbf{black}$
$time \leftarrow time + 1$
$f[v] \leftarrow time$
comment: $f[v]$ is the finishing time for vertex v

Classification of Edges in Depth-first Search

- uv is a **tree edge** if $u = \pi[v]$
- uv is a **forward edge** if it is not a tree edge, and v is a descendant of u in a tree in the depth-first forest
- uv is a **back edge** if u is a descendant of v in a tree in the depth-first forest
- any other edge is a **cross edge**.

Properties of Edges in Depth-first Search

In the following table, we indicate the colour of a vertex v when an edge uv is discovered, and the relation between the start and finishing times of u and v, for each possible type of edge uv.

edge type	colour of v	discovery/finish times
tree	**white**	$d[u] < d[v] < f[v] < f[u]$
forward	**black**	$d[u] < d[v] < f[v] < f[u]$
back	**gray**	$d[v] < d[u] < f[u] < f[v]$
cross	**black**	$d[v] < f[v] < d[u] < f[u]$

Observe that two intervals $(d[u], f[u])$ and $(d[v], f[v])$ never **overlap**. Two intervals are either **disjoint** or **nested**. This is sometimes called the parenthesis theorem.

Topological Orderings and DAGs

A directed graph G is a **directed acyclic graph**, or **DAG**, if G contains no directed cycle.

A directed graph $G = (V, E)$ has a **topological ordering**, or **topological sort**, if there is a linear ordering $<$ of all the vertices in V such that $u < v$ whenever $uv \in E$.

Some interesting/useful facts:

- A DAG contains a vertex of indegree 0.
- A directed graph G has a topological ordering if and only if it is a DAG.
- A directed graph G is a DAG if and only if a DFS of G has no back edges.
- If uv is an edge in a DAG, then a DFS of G has $f[v] < f[u]$.

Topological Ordering via Depth-first Search

Algorithm: *DFS*(G)

$\boxed{\textit{InitializeStack}(S)}$

$\boxed{DAG \leftarrow true}$

for each $v \in V(G)$

do $\begin{cases} colour[v] \leftarrow \textbf{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

$time \leftarrow 0$

for each $v \in V(G)$

do $\begin{cases} \textbf{if } colour[v] = \textbf{white} \\ \quad \textbf{then } \textit{DFSvisit}(v) \end{cases}$

$\boxed{\textbf{if } DAG \ \ \textbf{then return } (S) \ \ \textbf{else return } (DAG)}$

Topological Ordering via Depth-first Search (cont.)

Algorithm: *DFSvisit*(v)

$colour[v] \leftarrow \mathbf{gray}$

$time \leftarrow time + 1$

$d[v] \leftarrow time$

comment: $d[v]$ is the discovery time for vertex v

for each $w \in Adj[v]$

$$\textbf{do} \begin{cases} \textbf{if } colour[w] = \mathbf{white} \\ \quad \textbf{then } \begin{cases} \pi[w] \leftarrow v \\ \textit{DFSvisit}(w) \end{cases} \\ \boxed{\textbf{if } colour[w] = \mathbf{gray} \quad \textbf{then } DAG \leftarrow false} \end{cases}$$

$colour[v] \leftarrow \mathbf{black}$

$\boxed{\textit{Push}(S, v)}$

$time \leftarrow time + 1$

$f[v] \leftarrow time$

comment: $f[v]$ is the finishing time for vertex v

Strongly Connected Components of a Digraph G

For two vertices x and y of G, define $x \sim y$ if $x = y$; or if $x \neq y$ and there exist directed paths from x to y **and** from y to x.

The relation $\sim$ is an **equivalence relation**.

The **strongly connected components** of G are the equivalence classes of vertices defined by the relation $\sim$.

The **component graph** of G is a directed graph whose vertices are the strongly connected components of G. There is an arc from C_i to C_j if and only if there is an arc in G from some vertex of C_i to some vertex of C_j.

For a strongly connected component C, define $f[C] = \max\{f[v] : v \in C\}$ and $d[C] = \min\{d[v] : v \in C\}$.

Some interesting/useful facts:

- The component graph of G is a DAG.
- If C_i, C_j are strongly connected components, and there is an arc from C_i to C_j in the component graph, then $f[C_i] > f[C_j]$.

An Algorithm to Find the Strongly Connected Components

step 1 Perform a depth-first search of G, recording the finishing times $f[v]$ for all vertices v.

step 2 Construct a directed graph H from G by **reversing** the direction of all edges in G.

step 3 Perform a depth-first search of H, considering the vertices in **decreasing** order of the values $f[v]$ computed in step 1.

step 4 The strongly connected components of G are the trees in the depth-first forest constructed in step 3.

Depth-first Search of H

Assume that $f[v_{i_1}] > f[v_{i_2}] > \cdots > f[v_{i_n}]$.

Algorithm: *DFS*(H)

for $j \leftarrow 1$ **to** n
  do $colour[v_{i_j}] \leftarrow$ **white**
$scc \leftarrow 0$
for $j \leftarrow 1$ **to** n
  do $\begin{cases} \textbf{if } colour[v_{i_j}] = \textbf{white} \\ \quad \textbf{then } \begin{cases} scc \leftarrow scc + 1 \\ \mathit{DFSvisit}(H, v_{i_j}, scc) \end{cases} \end{cases}$
return $(comp)$
comment: $comp[v]$ is the strongly connected component containing v

DFSvisit for H

Algorithm: *DFSvisit*(H, v, scc)

$colour[v] \leftarrow$ **gray**

$comp[v] \leftarrow scc$

for each $w \in Adj[v]$

do $\begin{cases} \textbf{if } colour[w] = \textbf{white} \\ \quad \textbf{then } \textit{DFSvisit}(H, w, scc) \end{cases}$

$colour[v] \leftarrow$ **black**

Minimum Spanning Trees

A **spanning tree** in a connected, undirected graph $G = (V, E)$ is a subgraph T that is a tree which contains every vertex of V.

T is a spanning tree of G if and only if T is an acyclic subgraph of G that has $n - 1$ edges (where $n = |V|$).

Problem

Minimum Spanning Tree

Instance: *A connected, undirected graph $G = (V, E)$ and a* **weight function** $w : E \rightarrow \mathbb{R}$.

Find: *A spanning tree T of G such that*

$$\sum_{e \in T} w(e)$$

is minimized (this is called a **minimum spanning tree***, or* **MST***).*

Kruskal's Algorithm

Assume that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$, where $m = |E|$).

Algorithm: *Kruskal*(G, w)

$A \leftarrow \emptyset$

for $j \leftarrow 1$ **to** m

do $\begin{cases} \textbf{if } A \cup \{e_j\} \text{ does not contain a cycle} \\ \quad \textbf{then } A \leftarrow A \cup \{e_j\} \end{cases}$

return (A)

Prim's Algorithm (idea)

We initially choose an arbitrary vertex u_0 and define $A = \{e\}$, where e is the **minimum weight** edge incident with u_0.

A is always a **single tree**, and at each step we select the minimum weight edge that joins a vertex in VA to a vertex not in VA.

Remark: VA denotes the set of vertices in the tree A.

For a vertex $v \notin VA$, define

$$\begin{aligned} N[v] &= \text{a minimum weight edge } \{u, v\} \text{ such that } u \in VA \\ W[v] &= w(N[v], v). \end{aligned}$$

Assume $w(u, v) = \infty$ if $\{u, v\} \notin E$.

Prim's Algorithm

Algorithm: *Prim*(G, w)

$A \leftarrow \emptyset$

$VA \leftarrow \{u_0\}$, where u_0 is arbitrary

for all $v \in V \backslash \{u_0\}$

do $\begin{cases} W[v] \leftarrow w(u_0, v) \\ N[v] \leftarrow u_0 \end{cases}$

while $|A| < n - 1$

do $\begin{cases} \text{choose } v \in V \setminus VA \text{ such that } W[v] \text{ is minimized} \\ VA \leftarrow VA \cup \{v\} \\ u \leftarrow N[v] \\ A \leftarrow A \cup \{uv\} \\ \textbf{for all } v' \in V \setminus VA \\ \quad \textbf{do } \begin{cases} \textbf{if } w(v, v') < W[v'] \\ \quad \textbf{then } \begin{cases} W[v'] \leftarrow w(v, v') \\ N[v'] \leftarrow v \end{cases} \end{cases} \end{cases}$

return (A)

A General Greedy Algorithm to Find an MST

Algorithm: *GreedyMST*(G, w)

$A \leftarrow \emptyset$

while $|A| < n - 1$

do $\begin{cases} \text{let } (S, V \backslash S) \text{ be a cut that respects } A \\ \text{let } e \text{ be a minimum weight crossing edge} \\ A \leftarrow A \cup \{e\} \end{cases}$

return (A)

Some Relevant Definitions for Proof of Correctness

Let $G = (V, E)$ be a graph. A **cut** is a partition of V into two non-empty (disjoint) sets, i.e., a pair $(S, V \backslash S)$, where $S \subseteq V$ and $1 \leq |S| \leq n - 1$.

Let $(S, V \backslash S)$ be a cut in a graph $G = (V, E)$. An edge $e \in E$ is a **crossing edge** with respect to the cut $(S, V \backslash S)$ if e has one endpoint in S and one endpoint in $V \backslash S$.

Let $A \subseteq E$. A cut $(S, V \backslash S)$ **respects** the set of edges A provided that no edge in A is a crossing edge.

Single Source Shortest Paths

Problem

Single Source Shortest Paths

Instance: *A directed graph* $G = (V, E)$, *a non-negative* **weight function** $w : E \to \mathbb{R}^+ \cup \{0\}$, *and a* **source vertex** $u_0 \in V$.

Find: *For every vertex* $v \in V$, *a directed path* P *from* u_0 *to* v *such that*

$$w(P) = \sum_{e \in P} w(e)$$

is minimized.

The term **shortest path** really means **minimum weight path**.

We are asked to find n different shortest paths, one for each vertex $v \in V$.

If all edges have weight 1, we can just use *BFS* to solve this problem.

Dijkstra's Algorithm (Main Ideas)

S is a subset of vertices such that the shortest paths from u_0 to all vertices in S are known; initially, $S = \{u_0\}$.

For all vertices $v \in S$, $D[v]$ is the weight of the shortest path P_v from u_0 to v, and all vertices on P_v are in the set S.

For all vertices $v \notin S$, $D[v]$ is the weight of the shortest path P_v from u_0 to v in which all interior vertices are in S.

For $v \neq u_0$, $\pi[v]$ is the **predecessor** of v on the path P_v.

At each stage of the algorithm, we choose $v \in V \backslash S$ so that $D[v]$ is minimized, and then we add v to S.

Then the arrays D and π are updated appropriately.

Dijkstra's Algorithm

Algorithm: *Dijkstra*(G, w, u_0)

$S \leftarrow \{u_0\}$
$D[u_0] \leftarrow 0$
for all $v \in V \backslash \{u_0\}$
  do $\begin{cases} D[v] \leftarrow w(u_0, v) \\ \pi[v] \leftarrow u_0 \end{cases}$
while $|S| < n$
  do {
    choose $v \in V \backslash S$ such that $D[v]$ is minimized
    $S \leftarrow S \cup \{v\}$
    for all $v' \in V \backslash S$
      do {
        if $D[v] + w(v, v') < D[v']$
          then $\begin{cases} D[v'] \leftarrow D[v] + w(v, v') \\ \pi[v'] \leftarrow v \end{cases}$
return (D, π)

Finding the Shortest Paths

Algorithm: *FindPath*(u_0, π, v)

$path \leftarrow v$

$u \leftarrow v$

while $u \neq u_0$

do $\begin{cases} u \leftarrow \pi[u] \\ path \leftarrow u \parallel path \end{cases}$

return $(path)$

Shortest Paths in a DAG

If G is a DAG, we perform a topological ordering of the vertices. Suppose the resulting ordering is $v_1, \ldots, v_n$. Then we find all the shortest paths in G with source v_1.

Note: This algorithm is correct even if there are **negative-weight edges**.

Algorithm: *DAG Shortest paths*(G, w, v_1)

for $j \leftarrow 1$ **to** n

do $\begin{cases} D[v_1] \leftarrow \infty \\ \pi[v_j] \leftarrow undefined \end{cases}$

$D[v_1] \leftarrow 0$

for $j \leftarrow 1$ **to** $n - 1$

do $\begin{cases} \textbf{for all } v' \in Adj[v_j] \\ \quad \textbf{do} \begin{cases} \textbf{if } D[v_j] + w(v_j, v') < D[v'] \\ \quad \textbf{then} \begin{cases} D[v'] \leftarrow D[v_j] + w(v_j, v') \\ \pi[v'] \leftarrow v_j \end{cases} \end{cases} \end{cases}$

return (D, π)

All-Pairs Shortest Paths

Problem

All-Pairs Shortest Paths

Instance: *A directed graph* $G = (V, E)$, *and a* **weight matrix** W, *where* $W[i, j]$ *denotes the weight of edge* ij, *for all* $i, j \in V$, $i \neq j$.

Find: *For all pairs of vertices* $u, v \in V$, $u \neq v$, *a directed path* P *from* u *to* v *such that*

$$w(P) = \sum_{ij \in P} W[i, j]$$

is minimized.

We allow edges to have negative weights, but we assume there are no negative-weight directed cycles in G.

First Solution

Algorithm: *SlowAllPairsShortestPath*(W)

$L_1 \leftarrow W$
for $m \leftarrow 2$ **to** $n-1$
  do {
    for $i \leftarrow 1$ **to** n
      do {
        for $j \leftarrow 1$ **to** n
          do {
            $\ell \leftarrow \infty$
            for $k \leftarrow 1$ **to** n
              do $\ell \leftarrow \min\{\ell, L_{m-1}[i,k] + W[k,j]\}$
            $L_m[i,j] \leftarrow \ell$
return (L_{n-1})

Second Solution

Algorithm: *FasterAllPairsShortestPath*(W)

$L_1 \leftarrow W$
$m \leftarrow 2$
while $m < n-1$
  do {
    for $i \leftarrow 1$ **to** n
      do {
        for $j \leftarrow 1$ **to** n
          do {
            $\ell \leftarrow \infty$
            for $k \leftarrow 1$ **to** n
              do $\ell \leftarrow \min\{\ell, L_{m/2}[i,k] + L_{m/2}[k,j]\}$
            $L_m[i,j] \leftarrow \ell$
          }
      }
    $m \leftarrow 2m$
  }
return (L_m)

Third Solution

Algorithm: *FloydWarshall*(W)

$D_0 \leftarrow W$
for $m \leftarrow 1$ **to** n
do $\begin{cases} \textbf{for } i \leftarrow 1 \textbf{ to } n \\ \textbf{do } \begin{cases} \textbf{for } j \leftarrow 1 \textbf{ to } n \textbf{ do} \\ D_m[i,j] \leftarrow \min\{D_{m-1}[i,j], D_{m-1}[i,m] + D_{m-1}[m,j]\} \end{cases} \end{cases}$
return (D_n)