Algorithms (VII)

Yijia Chen
Shanghai Jiaotong University

Review of the Previous Lecture

Depth-first search in undirected graphs

Exploring graphs

EXPLORE(G, v)
Input: $G = (V, E)$ is a graph; $v \in V$
Output: `visit`(u) is set to `true` for all nodes u reachable from v

1. `visited`$(v) =$ `true`
2. PREVIST(v)
3. **for** each edge $(v, u) \in E$ **do**
4. **if** not `visited`(u) **then** EXPLORE(u)
5. POSTVISIT(v)

Theorem

EXPLORE(G, v) *is correct, i.e., it visits exactly all nodes that are reachable from* v.

Edge types

Those edges in G that are traversed by EXPLORE are *tree edges*.

The rest are *back edges*.

Depth-first search

```
DFS(G)

1. for all v ∈ V do
2.         visited(v) = false
3. for all v ∈ V do
4.         if not visited(v) then EXPLORE(v)
```

Theorem

DFS has a running time of $O(|V| + |E|)$.

Connectivity in undirected graphs

Definition

An undirected graph is **connected**, if there is a path between any pair of vertices.

Definition

A **connected component** is a subgraph that is internally connected but has no edges to the remaining vertices.

When EXPLORE is started at a particular vertex, it identifies precisely the connected component containing that vertex.

Each time the DFS outer loop calls EXPLORE, a new connected component is picked out.

Connectivity in undirected graphs (cont'd)

Thus depth-first search is trivially adapted to check if a graph is connected.

More generally, to assign each node v an integer `ccnum`$[v]$ identifying the connected component to which it belongs.

All it takes is

PREVISIT(v)

`ccnum`$[v] = cc$

where cc needs to be initialized to zero and to be incremented each time the DFS procedure calls EXPLORE.

Previsit and postvisit orderings

For each node, we will note down the times of two important events:

- the moment of first discovery (corresponding to PREVISIT);
- and the moment of final departure (POSTVISIT).

PREVISIT(v)

$\texttt{pre}[v] = \text{clock}$

$\text{clock} = \text{clock} + 1$

POSTVISIT(v)

$\texttt{post}[v] = \text{clock}$

$\text{clock} = \text{clock} + 1$

Lemma

For any nodes u and v, the two intervals $[\texttt{pre}(u), \texttt{post}(u)]$ and $[\texttt{pre}(u), \texttt{post}(u)]$ are either disjoint or one is contained within the other.

Depth-first search in directed graphs

Types of edges

DFS yields a **search tree/forests**.

- root.
- descendant and ancestor.
- parent and child.

- **Tree edges** are actually part of the DFS forest..
- **Forward edges** lead from a node to a nonchild descendant in the DFS tree.
- **Backedges** lead to an ancestor in the DFS tree.
- **Cross edges** lead to neither descendant nor ancestor; they therefore lead to a node that has already been completely explored (that is, already postvisited).

Types of edges (cont'd)

`pre`/`post` ordering for (u, v)	Edge type
$[_u \quad [_v \quad]_v \quad]_u$	Tree/forward
$[_v \quad [_u \quad]_u \quad]_v$	Back
$[_v \quad]_v \quad [_u \quad]_u$	Cross

Directed acyclic graphs (DAG)

Definition

A cycle in a directed graph is a circular path

$$v_0 \to v_1 \to v_2 \to \cdots v_k \to v_0.$$

Lemma

A directed graph has a cycle if and only if its depth-first search reveals a back edge.

Directed acyclic graphs (cont'd)

Linearization/Topologically Sort: Order the vertices such that every edge goes from a small vertex to a large one.

Lemma

In a dag, every edge leads to a vertex with a lower post number.

Hence there is a linear-time algorithm for ordering the nodes of a dag.

Since a dag is linearized by decreasing `post` numbers, the vertex with the smallest `post` number comes last in this linearization, and it must be a **sink** – no outgoing edges. Symmetrically, the one with the highest `post` is a **source**, a node with no incoming edges.

Lemma

Every dag has at least one source and at least one sink.

The guaranteed existence of a source suggests an alternative approach to linearization:

1. Find a source, output it, and delete it from the graph.
2. Repeat until the graph is empty.

Strongly connected components

Defining connectivity for directed graphs

Definition

Two nodes u and v of a directed graph are **connected** if there is a path from u to v and a path from v to u.

This relation partitions V into disjoint sets that we call **strongly connected components**.

Lemma

Every directed graph is a dag of its strongly connected components.

An efficient algorithm

Lemma

If the EXPLORE *subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited.*

Therefore, if we call explore on a node that lies somewhere in a sink strongly connected component (a strongly connected component that is a sink in the meta-graph), then we will retrieve exactly that component.

We have two problems:

(A) How do we find a node that we know for sure lies in a sink strongly connected component?

(B) How do we continue once this first component has been discovered?

An efficient algorithm (cont'd)

Lemma

The node that receives the highest `post` *number in a depth-first search must lie in a source strongly connected component.*

Lemma

If C and C' are strongly connected components, and there is an edge from a node in C to a node in C', then the highest `post` *number in C is bigger than the highest* `post` *number in C'.*

Hence the strongly connected components can be linearized by arranging them in decreasing order of their highest `post` numbers.

Solving problem A

Consider the **reverse graph** G^R, the same as G but with all edges reversed.

G^R has exactly the same strongly connected components as G.
So, if we do a depth-first search of G^R, the node with the highest `post` number will come from a source strongly connected component in G^R, which is to say a sink strongly connected component in G.

Solving problem B

Once we have found the first strongly connected component and deleted it from the graph, the node with the highest post number among those remaining will belong to a sink strongly connected component of whatever remains of G.

Therefore we can keep using the post numbering from our initial depth-first search on G^R to successively output the second strongly connected component, the third strongly connected component, and so on.

The linear-time algorithm

1. Run depth-first search on G^R.
2. Run the undirected connected components algorithm on G, and during the depth-first search, process the vertices in decreasing order of their `post` numbers from step 1.

Chapter 4. Paths in graphs

Distances

DFS does not necessarily find the shortest paths.

Definition

The **distance** between two nodes is the length of the shortest path between them.

Breadth-first search

The algorithm

BFS(G, s)

Input: Graph $G = (V, E)$, directed or undirected; vertex $s \in V$

Output: For all vertices u reachable from s, $\texttt{dist}(u)$ is set to the distance from s to u.

```
1.  for all u ∈ V do
2.          dist(u) = ∞
3.  dist(s) = 0
4.  Q = [s] (queue containing just s)
5.  while Q is not empty do
6.          u = eject(Q)
7.          for all edge (u, v) ∈ E do
8.                  if dist(v) = ∞ then
9.                          inject(Q, v)
10.                         dist(v) = dist(u) + 1
```

Correctness and efficiency

Lemma

For each $d = 0, 1, 2, \ldots$, there is a moment at which (1) all nodes at distance $\leq d$ from s have their distances correctly set; (2) all other nodes have their distances set to ∞; and (3) the queue contains exactly the nodes at distance d.

Lemma

BFS has a running time of $O(|V| + |E|)$.

Lengths on edges

BFS treats all edges as having *the same length*, which is rarely true in applications where shortest paths are to be found.

Every edge $e \in E$ with a length ℓ_e.
If $e = (u, v)$, we will sometimes also write

$$\ell(u, v) \quad \text{or} \quad \ell_{uv}.$$

Dijkstra's algorithm

An adaption of breadth-first search

BFS finds shortest paths in any graph whose edges have unit length. Can we adapt it to a more general graph $G = (V, E)$ whose *edge lengths ℓ_e are positive integers*?

A simple trick:

> *For any edge $e = (u, v)$ of E, replace it by ℓ_e edges of length 1, by adding $\ell_e - 1$ dummy nodes between u and v.*

It might take time

$$O\left(|V| + \sum_{e \in E} \ell_e\right),$$

which is bad in case we have edges with high length.

Alarm clocks

- Set an alarm clock for node s at time 0.
- Repeat until there are no more alarms:
  Say the next alarm goes off at time T, for node y. Then:
  - The distance from s to u is T.
  - For each neighbor v of u in G:
    - If there is no alarm yet for v, set one for time $T + \ell(u, v)$.
    - If v's alarm is set for later than $T + \ell(u, v)$, then reset it to this earlier time.

Priority queue

Priority queue is a data structure usually implemented by *heap*.

- *Insert.* Add a new element to the set.
- *Decrease-key.* Accommodate the decrease in key value of a particular element.
- *Delete-min.* Return the element with the smallest key, and remove it from the set.
- *Make-queue.* Build a priority queue out of the given elements, with the given key values. (In many implementations, this is significantly faster than inserting the elements one by one.)

Dijkstra's shortest-path algorithm

DIJKSTRA(G, ℓ, s)

Input: Graph $G = (V, E)$, directed or undirected; positive edge length $\{\ell_e \mid e \in E\}$; vertex $s \in V$

Output: For all vertices u reachable from s, $\texttt{dist}(u)$ is set to the distance from s to u.

```
1.  for all u ∈ V do
2.          dist(u) = ∞
3.          prev(u) = nil
4.  dist(s) = 0
5.  H = makequeue(V) (using dist-values as keys)
6.  while H is not empty do
7.          u = deletemin(H)
8.          for all edge (u, v) ∈ E do
9.                  if dist(v) > dist(u) + ℓ(u, v) then
10.                         dist(v) = dist(u) + ℓ(u, v)
11.                         prev(v) = u
12.                         decreasekey(H, v)
```

An alternative derivation

1. Initialize $\texttt{dist}(s) = 0$, other $\texttt{dist}(\cdot)$ to ∞
2. $R = \{\ \}$ (the "known region")
3. **while** $R \neq V$ **do**
4. Pick the node $v \notin R$ with smallest $\texttt{dist}(\cdot)$
5. Add v to R
6. **for all** edge $(v, z) \in E$ **do**
7. **if** $\texttt{dist}(z) > \texttt{dist}(v) + \ell(v, z)$ **then**
8. $\texttt{dist}(z) = \texttt{dist}(v) + \ell(v, z)$

Key property

At the end of each iteration of the while loop, the following conditions hold:

(1) there is a value d such that all nodes in R are at distance $\leq d$ from s and all nodes outside R are at distance $\geq d$ from s;

(2) for every node u, the value `dist`(u) is the length of the shortest path from s to u whose intermediate nodes are constrained to be in R (if no such path exists, the value is ∞).

Running time

Since `makequeue` takes at most as long as $|V|$ `insert` operations, we get a total of $|V|$ `deletemin` and $|V| + |E|$ `insert`/`decreasekey` operations.

The time needed for these varies by implementation; for instance, a *binary heap* gives an overall running time of

$$O((|V| + |E|) \log |V|).$$

Which heap is best?

Implementation	`deletemin`	`insert`/ `decreasekey`	$\lvert V\rvert \times$ `deletemin`+ $(\lvert V\rvert + \lvert E\rvert) \times$ `insert`
Array	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
Binary heap	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$	$O((\lvert V\rvert + \lvert E\rvert) \log \lvert V\rvert)$
d-ary heap	$O(\frac{d \log \lvert V\rvert}{\log d})$	$O(\frac{\log \lvert V\rvert}{\log d})$	$O(\frac{(d\lvert V\rvert+\lvert E\rvert) \log \lvert V\rvert}{\log d})$
Fibonacci heap	$O(\log \lvert V\rvert)$	$O(1)$ (amortized)	$O(\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert)$

Priority queue implementations

Array

The simplest implementation of a priority queue is as an *unordered array* of key values for all potential elements (the vertices of the graph, in the case of Dijkstra's algorithm).

Initially, these values are set to ∞.

An `insert` or `decreasekey` is fast, because it just involves adjusting a key value, an O(1) operation.

To `deletemin`, on the other hand, requires a linear-time scan of the list.

Binary heap

Here elements are stored in *a complete binary tree*.

In addition, a special ordering constraint is enforced:

> *the key value of any node of the tree is less than or equal to that of its children.*

In particular, therefore, the root always contains the smallest element.

To `insert`, place the new element at the bottom of the tree (in the first available position), and let it "*bubble up*."
The number of swaps is at most the height of the tree $\lfloor \log_2 n \rfloor$, when there are n elements.

A `decreasekey` is similar, except the element is already in the tree, so we let it bubble up from its current position.

To `deletemin`, return the root value. Then remove this element from the heap, take the last node in the tree (in the rightmost position in the bottom row) and place it at the root. Then let it "*sift down*." Again this takes $O(\log n)$ time.

d-ary heap

A d-**ary heap** is identical to a binary heap, except that nodes have *d children*.

This reduces the height of a tree with n elements to

$$\Theta(\log_d n) = \Theta((\log n)/(\log d))$$

`Inserts` are therefore speeded up by a factor of $\Theta(\log d)$.
`Deletemin` operations, however, take a little longer, namely $O(d \log_d n)$.