Theory of Algorithms I

Graph Algorithms, Revisited

Guoqiang Li

School of Software, Shanghai Jiao Tong University
Instructor and Teaching Assistants

• Guoqiang LI
  • Homepage: https://basics.sjtu.edu.cn/~liguoqiang
  • Course page: https://basics.sjtu.edu.cn/~liguoqiang/teaching/SE222/
  • Email: li.g@outlook.com
  • Office: Rm. 1212, Building of Software
  • Phone: 3420-4167

• TA:
  • Yuan HANG: amberbabatt (AT) gmail (DOT)com
  • Office hour: Tue. 14:00-17:00 @ Software Building 3203
  • Office hour: Wed. 13:30-15:00 @ Software Building 1212 (need reserve!)
Text Book

- Algorithms: Design Techniques and Analysis
  - M. H. Alsuwalyel
Text Book

- **Algorithms**
  - Sanjoy Dasgupta  
    University of California  
  - San Diego Christos Papadimitriou  
    University of California at Berkeley  
  - Umesh Vazirani  
    University of California at Berkeley  

- Available at:
  http://www.cs.berkeley.edu/~vazirani/algorithms.html
Which one comes first, computer or algorithms?
Al Khwarizmi

In the 12th century, Latin translations of his work on the Indian numerals, introduced the decimal system to the Western world. (Source: Wikipedia)
Algorithms

Al Khwarizmi laid out the basic methods for

- adding,
- multiplying,
- dividing numbers,
- extracting square roots,
- calculating digits of $\pi$.

These procedures were precise, unambiguous, mechanical, efficient, correct.

They were algorithms, a term coined to honor the wise man after the decimal system was finally adopted in Europe, many centuries later.
DFS in Graphs, Revisited
EXPLORE \((G, v)\)

**input**: \(G = (V, E)\) is a graph; \(v \in V\)

**output**: \(visited(u)\) to \(true\) for all nodes \(u\) reachable from \(v\)

\(visited(v) = true;\)

PREVISIT\((v);\)

for each edge \((v, u) \in E\) do

| if not \(visited(u)\) then EXPLORE \((G, u)\); |

end

POSTVISIT\((v);\)
Types of Edges in Undirected Graphs

Those edges in $G$ that are traversed by \text{EXPLORE} are \text{tree edges}.

The rest are \text{back edges}.
Depth-First Search

\[
\text{DFS}(G) \\
\text{for all } v \in V \text{ do} \\
\quad \text{visited}(v) = \text{false}; \\
\text{end} \\
\text{for all } v \in V \text{ do} \\
\quad \text{if not visited}(v) \text{ then Explore}(G, v); \\
\text{end}
\]
Previsit and Postvisit Orderings

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

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

\[
\begin{align*}
\text{PREVISIT}(v) & : \begin{align*} 
pre[v] &= \text{clock} \\
clock &= \text{clock} + +
\end{align*} \\
\text{POSTVISIT}(v) & : \begin{align*} 
post[v] &= \text{clock} \\
clock &= \text{clock} + +
\end{align*}
\end{align*}
\]

**Lemma**

For any nodes \( u \) and \( v \), the two intervals \([pre(u), post(u)]\) and \([pre(u), post(u)]\) are either disjoint or one is contained within the other.
Previsit and Postvisit Orderings

(a)

A - B
E - F
I - J

(b)

A
B
C
D
E
F
G
H
I
J
K
L
1,10
2,3
4,9
5,8
6,7
11,22
23,24
12,21
13,20
14,17
15,16
18,19
20,20
21,21
22,22
23,23
24,24
25,25
26,26
Types of Edges in Directed Graphs

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.
- Back edges lead to an ancestor in the DFS tree.
- Cross edges lead to neither descendant nor ancestor.
Directed Graphs

A → B
A → C
B → D
C → D
E → F
E → G
F → H
G → H

A → B
B → C
C → D
D → A
E → F
E → G
F → H
G → H

A
B
C
D
E
F
G
H

1,16
2,11
3,10
4,7
5,6
8,9
12,15
13,14
### Types of Edges

<table>
<thead>
<tr>
<th>pre/post ordering for ((u, v))</th>
<th>Edge type</th>
</tr>
</thead>
<tbody>
<tr>
<td>([u \quad v \quad v \quad u])</td>
<td>Tree/forward</td>
</tr>
<tr>
<td>([v \quad u \quad u \quad v])</td>
<td>Back</td>
</tr>
<tr>
<td>([v \quad v \quad u \quad u])</td>
<td>Cross</td>
</tr>
</tbody>
</table>

Q: Is that all?
Directed Acyclic Graphs (DAG)

**Definition:**
A cycle in a directed graph is a circular path

\[ v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots v_k \rightarrow v_0 \]

**Lemma:**
A directed graph has a cycle if and only if its depth-first search reveals a back edge.
Directed Acyclic Graphs (DAG)
Directed Acyclic Graphs (DAG)

**Linearization/Topologically Sort:** Order the vertices such that every edge goes from an earlier vertex to a later one.

**Q:** What types of dags can be linearized?

**A:** All of them.

DFS tells us exactly how to do it: perform tasks in decreasing order of their post numbers.

The only edges \((u, v)\) in a graph for which \(\text{post}(u) < \text{post}(v)\) are **back edges**, and we have seen that a DAG cannot have **back edges**.
Directed Acyclic Graphs (DAG)

Lemma:
In a DAG, every edge leads to a vertex with a lower post number.
There is a **linear-time algorithm** for ordering the nodes of a DAG.

**Acyclicity**, **linearizability**, and the absence of back edges during a depth-first search - are the same thing.

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 (SCC)
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 (SCC)**.

**Lemma:**
Every directed graph is a DAG of its **SCC**.
Strongly Connected Components

(a) 
\[
\begin{align*}
A & \rightarrow B \\
B & \rightarrow C \\
C & \rightarrow B \\
D & \rightarrow E \\
E & \rightarrow F \\
F & \rightarrow E \\
G & \rightarrow H \\
H & \rightarrow G \\
I & \rightarrow J \\
J & \rightarrow K \\
K & \rightarrow J \\
L & \rightarrow I
\end{align*}
\]

(b) 
\[
\begin{align*}
A & \rightarrow B,E \\
B,E & \rightarrow C,F \\
D & \rightarrow G,H,I \\
G,H,I & \rightarrow J,K,L
\end{align*}
\]
Lemma:
If the EXPLORE subroutine at node $u$, then it will terminate precisely when all nodes reachable from $u$ have been visited.

If we call explore on a node that lies somewhere in a sink SCC, then we will retrieve exactly that component.

We have two problems:

1. How do we find a node that we know for sure lies in a sink SCC?
2. How do we continue once this first component has been discovered?
Lemma:
The node that receives the highest post number in a depth-first search must lie in a source SCC.

Lemma:
If $C$ and $C'$ are SCC, 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 SCCs 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 SCCs as $G$.

If we do a depth-first search of $G^R$, the node with the highest post number will come from a source SCC in $G^R$. It is a sink SCC in $G$. 
Strongly Connected Components
Once we have found the first SCC and deleted it from the graph, the node with the highest post number among those remaining will belong to a sink SCC 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 SCC, and so on.
The Linear-Time Algorithm

1. Run depth-first search on $G^R$.
2. Run the EXPLORE algorithm on $G$, and during the depth-first search, process the vertices in decreasing order of their post numbers from step 1.
Strongly Connected Components

Diagram showing a directed graph with strongly connected components.
How the SCC algorithm works when the graph is very, very huge?
Exercises
Suppose a CS curriculum consists of $n$ courses, all of them mandatory. The prerequisite graph $G$ has a node for each course, and an edge from course $v$ to course $w$ if and only if $v$ is a prerequisite for $w$. Find an algorithm that works directly with this graph representation, and computes the minimum number of semesters necessary to complete the curriculum (assume that a student can take any number of courses in one semester). The running time of your algorithm should be linear.
Give an efficient algorithm which takes as input a directed graph \( G = (V, E) \), and determines whether or not there is a vertex \( s \in V \) from which all other vertices are reachable.
BFS in Graphs, Revisited
Breadth-First Search

BFS \((G, v)\)

**input**: Graph \(G = (V, E)\), directed or undirected; Vertex \(v \in V\)

**output**: For all vertices \(u\) reachable from \(v\), \(dist(u)\) is the set to the distance from \(v\) to \(u\)

```plaintext
for all \(u \in V\) do
    \(\text{dist}(u) = \infty\);
end

\(\text{dist}[v] = 0\);

\(Q = [v]\) queue containing just \(v\);

while \(Q\) is not empty do
    \(u = \text{Eject}(Q)\);
    for all edge \((u, s) \in E\) do
        if \(\text{dist}(s) = \infty\) then
            \(\text{Inject}(Q, s)\);
            \(\text{dist}[s] = \text{dist}[u] + 1\);
        end
    end
end
```
Dijkstra’s Shortest-Path Algorithm

**DIJKSTRA** $(G, l, s)$

**input**: Graph $G = (V, E)$, directed or undirected; positive edge length $\{l_e | e \in E\}$; Vertex $s \in V$

**output**: For all vertices $u$ reachable from $s$, $dist(u)$ is the set to the distance from $s$ to $u$

```
for all $u \in V$ do
    dist(u) = \infty;
    prev(u) = nil;
end

dist(s) = 0;

H = makequeue(V) \\ using dist-values as keys;

while H is not empty do
    u = delete_min(H);
    for all edge $(u, v) \in E$ do
        if dist(v) > dist(u) + l(u, v) then
            dist(v) = dist(u) + l(u, v);
            prev(v) = u;
            decreasekey(H, v);
        end
    end
end
```
An Example

A: 0  B: 4  C: 2  D: $\infty$  E: $\infty$

A: 0  B: 3  C: 2  D: 5  E: 6

A: 0  B: 3  C: 2  D: 5  E: 6

A: 0  B: 3  C: 2  D: 6  E: 7
Shortest Paths in the Presence of Negative Edges
Negative Edges

Dijkstra’s algorithm works in part because the shortest path from the starting point $s$ to any node $v$ must pass exclusively through nodes that are closer than $v$.

This no longer holds when edge lengths can be negative.

Q: What needs to be changed in order to accommodate this new complication?

A crucial invariant of Dijkstra’s algorithm is that the $\text{dist}$ values it maintains are always either overestimates or exactly correct.

They start off at $\infty$, and the only way they ever change is by updating along an edge:

\[
\text{UPDATE } ((u, v) \in E) \\
dist(v) = \min\{\dist(v), \dist(u) + l(u, v)\};
\]
Update

\[
\text{UPDATE } ((u, v) \in E) \\
dist(v) = \min\{dist(v), dist(u) + l(u, v)\};
\]

This **UPDATE** operation is simply an expression of the fact that the distance to \( v \) cannot possibly be more than the distance to \( u \), plus \( l(u, v) \). It has the following properties,

1. It gives the correct distance to \( v \) in the particular case where \( u \) is the second-last node in the shortest path to \( v \), and \( dist(u) \) is correctly set.

2. It will never make \( dist(v) \) too small, and in this sense it is safe. For instance, a slew of extraneous update’s can’t hurt.
Update

\textbf{UPDATE } ((u, v) \in E)

\[ \text{dist}(v) = \min\{\text{dist}(v), \text{dist}(u) + l(u, v)\}; \]

Let

\[ s \rightarrow u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow \ldots \rightarrow u_k \rightarrow k \]

be a shortest path from \( s \) to \( t \).

This path can have at most \(|V| - 1\) edges (\textbf{why?}).
Bellman-Ford Algorithm

If we don’t know all the shortest paths beforehand, how can we be sure to update the right edges in the right order?

We simply update all the edges, $|V| - 1$ times!
Bellman-Ford Algorithm

SHORTEST-PATHS \((G, l, s)\)

**input**: Graph \(G = (V, E)\), edge length \(\{l_e \mid e \in E\}\); Vertex \(s \in V\)

**output**: For all vertices \(u\) reachable from \(s\), \(dist(u)\) is the set to the distance from \(s\) to \(u\)

\[
\text{for all } u \in V \text{ do} \\
\qquad \text{dist}(u) = \infty; \\
\qquad \text{prev}(u) = \text{nil}; \\
\text{end}
\]

\(\text{dist}[s] = 0;\)

repeat \(|V| - 1\) times: \(\text{for } e \in E \text{ do} \\
\qquad \text{UPDATE}(e); \\
\text{end}\)

**Running time**: \(O(|V| \cdot |E|)\)
Bellman-Ford Algorithm
Negative Cycles

If the graph has a negative cycle, then it doesn’t make sense to even ask about shortest path.

Q: How to detect the existence of negative cycles:

Instead of stopping after $|V| - 1$, iterations, perform one extra round.

There is a negative cycle if and only if some $dist$ value is reduced during this final round.
Shortest Paths in Dags
Graphs without Negative Edges

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

• graphs without negative edges,
• and graphs without cycles.

We already know how to efficiently handle the former.

We will now see how the single-source shortest-path problem can be solved in just linear time on directed acyclic graphs.

As before, we need to perform a sequence of updates that includes every shortest path as a subsequence.

In any path of a DAG, the vertices appear in increasing linearized order.
A Shortest-Path Algorithm for DAG

DAG-SHORTEST-PATHS \( (G, l, s) \)

**input**: Graph \( G = (V, E) \), edge length \( \{l_e \mid e \in E\} \); Vertex \( s \in V \)

**output**: For all vertices \( u \) reachable from \( s \), \( \text{dist}(u) \) is the set to the distance from \( s \) to \( u \)

for all \( u \in V \) do
  \[
  \text{dist}(u) = \infty;
  \]
  \[
  \text{prev}(u) = \text{nil};
  \]
end

\( \text{dists} = 0; \)
linearize \( G; \)
for each \( u \in V \) in linearized order do
  for all \( e \in E \) do
    \[
    \text{UPDATE}(e);
    \]
  end
end
A Shortest-Path Algorithm for DAG

Note that the scheme doesn’t require edges to be positive.

In particular, we can find longest paths in a DAG by the same algorithm: just negate all edge lengths.
Exercises
Professor Fake suggests the following algorithm for finding the shortest path from node $s$ to node $t$ in a directed graph with some negative edges: add a large constant to each edge weight so that all the weights become positive, then run Dijkstra’s algorithm starting at node $s$, and return the shortest path found to node $t$. 
You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights along with a particular node $v_0 \in V$. Give an efficient algorithm for finding shortest paths between all pairs of nodes, with the one restriction that these paths must all pass through $v_0$. 
Homework

[DPV07]. 3.7, 3.11, 3.28, 4.11, 4.12, 4.16
[Als99]. 9.18, 9.33,