CSE 548: Algorithms

Basic Graph Algorithms

R. Sekar
Overview

Graphs provide a concise representation of a range of problems:

- Map coloring – more generally, resource contention problems
- Networks — communication, traffic, social, biological, ...

![Map and Graph](image)

**Figure 3.1** (a) A map and (b) its graph.
### Definition and Representations

#### Definition

- **A graph** $G = (V, E)$, where $V$ is a set of vertices, and $E$ a set of edges. 
- **An edge** $e$ of the form $(v_1, v_2)$ is said to span vertices $v_1$ and $v_2$. 
- **The edges in a directed graph are directed.**
Definition

- **A graph** $G = (V, E)$, where $V$ is a set of vertices, and $E$ a set of edges.
- An edge $e$ of the form $(v_1, v_2)$ is said to span vertices $v_1$ and $v_2$.
- The edges in a directed graph are directed.
- A $G' = (V', E')$ is called a subgraph of $G$ if $V' \subseteq V$ and $E' = \{(u, v) \in E | u, v \in V'\}$. 
Definition and Representations

**Definition**

- A graph $G = (V, E)$, where $V$ is a set of vertices, and $E$ a set of edges.
- An edge $e$ of the form $(v_1, v_2)$ is said to span vertices $v_1$ and $v_2$.
- The edges in a directed graph are directed.
- A $G' = (V', E')$ is called a subgraph of $G$ if $V' \subseteq V$ and $E' = \{(u, v) \in E | u, v \in V'\}$.

**Adjacency matrix**

A graph $(V = \{v_1, \ldots, v_n\}, E)$ can be represented by an $n \times n$ matrix $a$, where $a_{ij} = 1$ iff $(v_i, v_j) \in E$. 
Definition and Representations

Definition

- A graph $G = (V, E)$, where $V$ is a set of vertices, and $E$ a set of edges.
- An edge $e$ of the form $(v_1, v_2)$ is said to span vertices $v_1$ and $v_2$.
- The edges in a directed graph are directed.
- A $G' = (V', E')$ is called a subgraph of $G$ if $V' \subseteq V$ and $E' = \{(u, v) \in E | u, v \in V'\}$.

Adjacency matrix

A graph $(V = \{v_1, \ldots, v_n\}, E)$ can be represented by an $n \times n$ matrix $a$, where $a_{ij} = 1$ iff $(v_i, v_j) \in E$.

Adjacency list

Each vertex $v$ is associated with a linked list consisting of all vertices $u$ such that $(v, u) \in E$. 
Definition and Representations

Definition

- A graph $G = (V, E)$, where $V$ is a set of vertices, and $E$ a set of edges.
- An edge $e$ of the form $(v_1, v_2)$ is said to span vertices $v_1$ and $v_2$.
- The edges in a directed graph are directed.
- A $G' = (V', E')$ is called a subgraph of $G$ if $V' \subseteq V$ and $E' = \{(u, v) \in E | u, v \in V'\}$.

Adjacency matrix

A graph $(V = \{v_1, \ldots, v_n\}, E)$ can be represented by an $n \times n$ matrix $a$, where $a_{ij} = 1$ iff $(v_i, v_j) \in E$

Adjacency matrix uses $O(n^2)$ storage; adjacency list uses $O(|V| + |E|)$ storage.

Adjacency list

Each vertex $v$ is associated with a linked list consisting of all vertices $u$ such that $(v, u) \in E$. 
Depth-First Search (DFS)

- A technique for traversing all vertices in the graph.
- Very versatile, forms the linchpin of many graph algorithms.
Depth-First Search (DFS)

- A technique for traversing all vertices in the graph.
- Very versatile, forms the linchpin of many graph algorithms.

\[
dfs(V, E)
\]

\[
\text{foreach } v \in V \text{ do } \text{visited}[v] = false
\]

\[
\text{foreach } v \in V \text{ do}
\]

\[
\text{if not } \text{visited}[v] \text{ then } \text{explore}(V, E, v)
\]
Depth-First Search (DFS)

- A technique for traversing all vertices in the graph.
- Very versatile, forms the linchpin of many graph algorithms.

```plaintext
dfs(V, E)
    foreach v ∈ V do visited[v] = false
    foreach v ∈ V do
        if not visited[v] then explore(V, E, v)

explore(V, E, v)
    visited[v] = true
    previsit(v) /*A placeholder for now*/
    foreach (v, u) ∈ E do
        if not visited[u] then explore(V, E, u)
    postvisit(v) /*Another placeholder*/
```
If a maze is represented as a graph, then DFS of the graph amounts to an exploration and mapping of the maze.
A graph and its DFS tree

DFS uses $O(|V|)$ space and $O(|E| + |V|)$ time.
A connected component of a graph is a maximal subgraph where there is path between any two vertices in the subgraph, i.e., it is a maximal connected subgraph.
DFS Numbering

Associate post and pre numbers with each visited node by defining \textit{previsit} and \textit{postvisit}

\begin{align*}
\text{previsit}(v) &= \text{clock} \\
\text{pre}[v] &= \text{clock} \\
\text{clock} &= \text{clock} + 1
\end{align*}
DFS Numbering

Associate post and pre numbers with each visited node by defining *previsit* and *postvisit*

\[
\text{previsit}(v) = \begin{align*}
\text{pre}[v] &= \text{clock} \\
\text{clock} &= \text{clock} + 1
\end{align*}
\]

\[
\text{postvisit}(v) = \begin{align*}
\text{post}[v] &= \text{clock} \\
\text{clock} &= \text{clock} + 1
\end{align*}
\]
DFS Numbering

Associate post and pre numbers with each visited node by defining $\text{previsit}$ and $\text{postvisit}$

$\text{previsit}(v)$$
\begin{align*}
\text{pre}[v] &= \text{clock} \\
\text{clock} &= \text{clock} + 1
\end{align*}$

$\text{postvisit}(v)$$
\begin{align*}
\text{post}[v] &= \text{clock} \\
\text{clock} &= \text{clock} + 1
\end{align*}$

Property

For any two vertices $u$ and $v$, the intervals $[\text{pre}[u], \text{post}[u]]$ and $[\text{pre}[v], \text{post}[v]]$ are either disjoint, or one is contained entirely within another.
Figure 3.7 DFS on a directed graph.

A
B
C
D
E
F
G
H

3.3 Depth-first search in directed graphs

3.3.1 Types of edges

Our depth-first search algorithm can be run verbatim on directed graphs, taking care to traverse edges only in their prescribed directions. Figure 3.7 shows an example and the search tree that results when vertices are considered in lexicographic order.

In further analyzing the directed case, it helps to have terminology for important relationships between nodes of a tree. A is the root of the search tree; everything else is its descendant. Similarly, E has descendants F, G, and H, and conversely, is an ancestor of these three nodes.

The family analogy is carried further: C is the parent of D, which is its child.

For undirected graphs we distinguished between tree edges and nontree edges. In the directed case, there is a slightly more elaborate taxonomy:

- **Tree edges**: Edges that are part of a path from the root to a leaf.
- **Back edges**: Edges that go from a child to its ancestor.
- **Forward edges**: Edges that go from a parent to a child.
- **Cross edges**: Any remaining edge.

These timings will soon take on larger significance. Meanwhile, you might have noticed from Figure 3.4 that:

**Property**

For any nodes u and v, the two intervals \([pre(u), post(u)]\) and \([pre(v), post(v)]\) are either disjoint or one is contained within the other. Why? Because \([pre(u), post(u)]\) is essentially the time during which vertex u was on the stack. The last-in, first-out behavior of a stack explains the rest.
**DFS and Edge Types**

**DFS tree**

- **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; they connect nodes that are not related by ancestor-descendant relationships.

### Table: pre/post ordering for $(u, v)$

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

**No cross edges in undirected graphs!**

Back and forward edges merge.
Directed Acyclic Graphs (DAGs)

A directed graph that contains no cycles.
Often used to represent (acyclic) dependencies, partial orders,...

**Property (DAGs and DFS)**

- A directed graph has a cycle iff its DFS reveals a back edge.
- In a dag, every edge leads to a vertex with lower post number.
- Every dag has at least one source and one sink.
Topological Sort

A way to linearize DAGs while ensuring that for every vertex, all its ancestors appear before itself.

Applications: spreadsheet recomputation of formulas, Make (and other compile/build systems) and Task scheduling/project management.
### Topological Sort

**topoSort**(V, E)

```plaintext
while |V| ≠ 0
  if there is a vertex v in V with in-degree of 0
    output v
    V = V − {v}; E = E − {e ∈ E|e is incident on v}
  else output “graph is cyclic”; break
```

---

**Intro DFS DAGs SCC BFS Paths and Matrices**

---

Chapter 6

Dynamic programming

In the preceding chapters we have seen some elegant design principles—such as divide-and-conquer, graph exploration, and greedy choice—that yield de finitive algorithms for a variety of important computational tasks. The drawback of these tools is that they can only be used on very specfic types of problems. We now turn to the two sledgehammers of the algorithms craft, dynamic programming and linear programming, techniques of very broad applicability that can be invoked when more specialized methods fail. Predictably, this generality often comes with a cost in ef ciency.

#### 6.1 Shortest paths in dags, revisited

At the conclusion of our study of shortest paths (Chapter 4), we observed that the problem is especially easy in directed acyclic graphs (dags). Let's recapitulate this case, because it lies at the heart of dynamic programming.

The special distinguishing feature of a dag is that its nodes can be linearized; that is, they can be arranged on a line so that all edges go from left to right (Figure 6.1). To see why this helps with shortest paths, suppose we want to figure out distances from node S to the other nodes. For concreteness, let's focus on node D. The only way to get to it is through its predecessors, B or C; so to find the shortest path to D, we need only compare these two routes:

\[
\text{dist}(D) = \min\{\text{dist}(B) + 1, \text{dist}(C) + 3\}.
\]

A similar relation can be written for every node. If we compute these dist values in the

---

![Diagram](image.png)
Topological Sort

topoSort(V, E)

while |V| \neq 0

if there is a vertex v in V with in-degree of 0

output v

V = V \setminus \{v\}; E = E \setminus \{e \in E | e \text{ is incident on } v\}

else output “graph is cyclic”; break

Correctness:

- If there is no vertex with in-degree 0, it is not a DAG
- When the algorithm outputs v, it has already output v’s ancestors
Topological Sort

\textit{topoSort}(V, E)

\textbf{while} \(|V| \neq 0\)

\hspace{1em} \textbf{if} there is a vertex \(v\) in \(V\) with in-degree of 0

\hspace{2em} \textbf{output} \(v\)

\hspace{2em} \(V = V - \{v\};\ E = E - \{e \in E| e \text{ is incident on } v\}\)

\textbf{else output} “graph is cyclic”; \textbf{break}

Correctness:

- If there is no vertex with in-degree 0, it is not a DAG
- When the algorithm outputs \(v\), it has already output \(v\)’s ancestors

Performance: What is the runtime? Can it be improved using DFS properties of DAGs?
Strongly Connected Components (SCC)

For directed graphs, SCCs are the equivalent of connected components in undirected graphs.

**Definition (SCC)**

- Two vertices $u$ and $v$ in a directed graph are **connected** if there is a path from $u$ to $v$ and vice-versa.

- A directed graph is **strongly connected** if any pair of vertices in the graph are connected.

- A **subgraph** of a directed graph is said to be an SCC if it is a maximal subgraph that is strongly connected.
3.4.2 An efficient algorithm

The decomposition of a directed graph into its strongly connected components is very informative and useful. It turns out, fortunately, that it can be found in linear time by making further use of depth-first search. The algorithm is based on some properties we have already seen but which we will now pinpoint more closely.

Property 1: 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. Figure 3.9 has two sink strongly connected components. Starting explore at node $K$, for instance, will completely traverse the larger of them and then stop.

This suggests a way of finding one strongly connected component, but still leaves open two major problems: (A) how do we find a node that we know for sure lies in a sink strongly connected component and (B) how do we continue once this first component has been discovered?

Let's start with problem (A). There is not an easy, direct way to pick out a node that is guaranteed to lie in a sink strongly connected component. But there is a way to get a node in a source strongly connected component.

Property 2: The node that receives the highest post number in a depth-first search must lie in a source strongly connected component. This follows from the following more general property.

Property 3: If $C$ and $C'$ are strongly connected components, and there is an edge from a node...
**Property**

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

---

**Figure 3.9**

(a) A directed graph and its strongly connected components. (b) The meta-graph.

---

**Property 1**

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

**Property 2**

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

This follows from the following more general property.

**Property 3**

If $C$ and $C'$ are strongly connected components, and there is an edge from a node in $C$ to a node in $C'$, then $C$ and $C'$ are not strongly connected components.
Towards an Algorithm for Computing SCCs

- Pick a sink SCC.
- Output all nodes in this SCC.
  - We can just do a DFS starting from any node $v$ in this SCC!
    - Because this is a sink SCC, this DFS cannot reach any other SCC, so will only output this SCC.
- Delete these nodes from the graph and repeat the whole process.
Towards an Algorithm for Computing SCCs

- Pick a sink SCC.
- Output all nodes in this SCC.
  - We can just do a DFS starting from any node $v$ in this SCC!
    - Because this is a sink SCC, this DFS cannot reach any other SCC, so will only output this SCC.
- Delete these nodes from the graph and repeat the whole process.

But how do we find a node in the sink SCC?
Towards an Algorithm for Computing SCCs

Property

- When \( \text{explore}(u) \) returns, it has visited all (and only) the nodes reachable from \( u \).

- If \( C \) and \( C' \) are SCCs and there is an edge from \( C \) to \( C' \) then:
  
  the highest post number in \( C \) will be larger than the highest post number in \( C' \).
Towards an Algorithm for Computing SCCs

Property

- When \( \text{explore}(u) \) returns, it has visited all (and only) the nodes reachable from \( u \).
- If \( C \) and \( C' \) are SCCs and there is an edge from \( C \) to \( C' \) then:
  the highest post number in \( C \) will be larger than the highest post number in \( C' \).

Corollary

The node that receives the highest post number after DFS must be in a source SCC.
Towards an Algorithm for Computing SCCs

Property

- When explore(u) returns, it has visited all (and only) the nodes reachable from u.
- If C and C' are SCCs and there is an edge from C to C' then:
  the highest post number in C will be larger than the highest post number in C'.

Corollary

The node that receives the highest post number after DFS must be in a source SCC.

Property

For a graph G, let $G_R$ denote the graph formed by reversing every edge in G. Then
- The SCCs of G and $G_R$ are identical.
- A source SCC of $G_R$ is a sink SCC of G.
An Algorithm for Computing SCCs

1. Construct $G_R$ from $G$ by reversing every edge in the given graph $G$.
2. The node $v$ with the highest post number is in a source SCC of $G_R$.
   - So, $v$ must be in a sink SCC of $G$.
3. Invoke $\text{explore}(v)$ in $G$ to output this sink SCC.
4. Delete these nodes from $G$ and $G_R$, and repeat from Step 2.
An Algorithm for Computing SCCs

1. Construct $G_R$ from $G$ by reversing every edge in the given graph $G$.

2. The node $v$ with the highest post number is in a source SCC of $G_R$.
   - So, $v$ must be in a sink SCC of $G$.

3. Invoke $explore(v)$ in $G$ to output this sink SCC.

4. Delete these nodes from $G$ and $G_R$, and repeat from Step 2.

Can we do all this in linear time?
Breadth-first Search (BFS)

- Traverse the graph by “levels”
  - \( BFS(v) \) visits \( v \) first
  - Then it visits all immediate children of \( v \)
  - Then it visits children of children of \( v \), and so on.
Breadth-first Search (BFS)

- Traverse the graph by “levels”
  - \(\text{BFS}(v)\) visits \(v\) first
  - Then it visits all immediate children of \(v\)
  - then it visits children of children of \(v\), and so on.

- As compared to DFS, BFS uses a queue (rather than a stack) to remember vertices that still need to be explored
BFS Algorithm

\( \text{bfs}(V, E, s) \)

\[
\begin{align*}
\textbf{foreach } u \in V \textbf{ do } & \text{ visited}[u] = false \\
q = \{s\}; & \text{ visited}[s] = true \\
\textbf{while } q \text{ is nonempty } \textbf{ do } \\
& u = \text{deque}(q) \\
\textbf{foreach edge } (u, v) \in E \textbf{ do } \\
& \textbf{if not } \text{ visited}[v] \textbf{ then } \\
& \quad \text{queue}(q, v); \text{ visited}[v] = true
\end{align*}
\]
**BFS Algorithm Illustration**

In Figure 4.3, we present a breadth-first search (BFS) algorithm. The BFS starts with a queue and visits nodes in order of their distance from the starting point. The algorithm proceeds as follows:

1. Initialize a queue and set the distance of the starting point to 0.
2. If the queue is empty, return; otherwise, dequeue a node.
3. For each of the node's neighbors, if the neighbor has not been visited before, set its distance to the current node's distance plus 1 and enqueue it.
4. Repeat until the queue is empty.

The BFS algorithm is very efficient for finding the shortest path in an unweighted graph. It's similar to DFS but uses a queue instead of a stack, which gives it the property of visiting nodes in increasing order of their distance from the starting point.

In our example (Figure 4.1), we apply the BFS algorithm and see how the nodes are visited in increasing order of their distance from the starting point, which is the node 'S'. The nodes are visited in the order: S, D, E, C, A, B. This confirms that BFS works correctly and efficiently for finding the shortest path in an unweighted graph.
BFS Algorithm Illustration

<table>
<thead>
<tr>
<th>Order of visitation</th>
<th>Queue contents after processing node</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>[S]</td>
</tr>
<tr>
<td>A</td>
<td>[A C D E]</td>
</tr>
<tr>
<td>C</td>
<td>[C D E B]</td>
</tr>
<tr>
<td>D</td>
<td>[D E B]</td>
</tr>
<tr>
<td>E</td>
<td>[E B]</td>
</tr>
<tr>
<td>B</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

Breadth-first search (BFS) directly implements this simple reasoning (Figure 4.3). Initially, the starting point is the only node on the queue. Then, nodes are processed in order of their distance from the starting point, which is achieved using a queue. This suggests an iterative algorithm. As these nodes are processed (ejected off the front of the queue), their neighbors are added to the queue, and so on. A convenient way to compute distances is through breadth-first search. Each vertex is put on the queue exactly once, when it is first encountered. This ensures that vertices are visited in increasing order of their distance from the starting point. This is a broader, shallower search, rather like the propagation of a wave upon water. And it is achieved using almost exactly the same code as DFS—but with a queue in place of a stack.
Shortest Paths and BFS

BFS automatically computes shortest paths!

\[ \text{bfs}(V, E, s) \]

\begin{enumerate}
\item \textbf{foreach} \( u \in V \) \textbf{do} \( \text{dist}[u] = \infty \)
\item \( q = \{s\}; \text{dist}[s] = 0 \)
\item \textbf{while} \( q \) \textbf{is nonempty} \textbf{do}
  \begin{enumerate}
  \item \( u = \text{deque}(q) \)
  \item \textbf{foreach} \text{edge} \((u, v) \in E\) \textbf{do}
    \begin{enumerate}
    \item \textbf{if} \( \text{dist}[v] = \infty \) \textbf{then}
      \begin{enumerate}
      \item \( \text{queue}(q, v); \text{dist}[v] = \text{dist}[u] + 1 \)
      \end{enumerate}
    \end{enumerate}
  \end{enumerate}
\end{enumerate}
\end{enumerate}

But not all paths are created equal! We would like to compute shortest weighted path — a topic of future lecture.
Shortest Paths and BFS

BFS automatically computes shortest paths!

\[ \text{bfs}(V, E, s) \]

\[
\begin{align*}
\text{foreach } u \in V \text{ do } & \quad \text{dist}[u] = \infty \\
q = \{s\}; \text{dist}[s] = 0 \\
\text{while } q \text{ is nonempty } & \text{ do} \\
& \quad u = \text{deque}(q) \\
& \quad \text{foreach edge } (u, v) \in E \text{ do} \\
& \hspace{1em} \text{if } \text{dist}[v] = \infty \text{ then} \\
& \hspace{2em} \text{queue}(q, v); \text{dist}[v] = \text{dist}[u] + 1
\end{align*}
\]

But not all paths are created equal! We would like to compute shortest weighted path — a topic of future lecture.
Graph paths and Boolean Matrices

A graph and its boolean matrix representation

\[
A = \begin{bmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]
Let $A$ be the adjacency matrix for a graph $G$, and $B = A \times A$. Now, $B_{ij} = 1$ iff there is path in the graph of length 2 from $v_i$ to $v_j$. 

Let $C = A + B$. Then $C_{ij} = 1$ iff there is path of length $\leq 2$ between $v_i$ and $v_j$.

Define $A^* = A_0 + A_1 + A_2 + \cdots$. If $D = A^*$ then $D_{ij} = 1$ iff $v_j$ is reachable from $v_i$. 

**Example**

\[
A = \begin{bmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

\[
A^2 = \begin{bmatrix}
1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

\[
A^3 = \begin{bmatrix}
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]
Graph paths and Boolean Matrices

- Let $A$ be the adjacency matrix for a graph $G$, and $B = A \times A$. Now, $B_{ij} = 1$ iff there is a path in the graph of length 2 from $v_i$ to $v_j$.

- Let $C = A + B$. Then $C_{ij} = 1$ iff there is a path of length $\leq 2$ between $v_i$ and $v_j$. 

\[
A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \\
A^2 = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \\
A^3 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}
\]
Let $A$ be the adjacency matrix for a graph $G$, and $B = A \times A$. Now, $B_{ij} = 1$ iff there is path in the graph of length 2 from $v_i$ to $v_j$.

Let $C = A + B$. Then $C_{ij} = 1$ iff there is path of length $\leq 2$ between $v_i$ and $v_j$.

Define $A^* = A^0 + A^1 + A^2 + \cdots$. If $D = A^*$ then $D_{ij} = 1$ iff $v_j$ is reachable from $v_i$.
Redefine operations on matrix elements so that $+$ becomes $\text{min}$, and $\ast$ becomes integer addition.
Shortest paths and Matrix Operations

- Redefine operations on matrix elements so that $+$ becomes $\text{min}$, and $\ast$ becomes integer addition.

- $D = A^*$ then $D_{ij} = k$ iff the shortest path from $v_j$ to $v_i$ is of length $k$