Graph Algorithms

Advanced Algorithms and Data Structures - Lecture 6

Venanzio Capretta
Thursday 5 November 2020

School of Computer Science, University of Nottingham
A (directed) graph consists of

- a set of vertices: \{0, 1, 2, 3, 4\}
- a set of edges between the vertices:
  \{ (0, 1), (0, 3), (1, 4), (2, 0), (2, 3), (3, 0), (3, 1), (3, 4), (4, 1), (4, 2) \}
There are several ways of representing graphs as data structures:
There are several ways of representing graphs as data structures:

**List of edges:**

\[ [(0, 1), (0, 3), (1, 4), (2, 0), (2, 3), (3, 0), (3, 1), (3, 4), (4, 1), (4, 2)] \]

We assume the set of vertices is implicit:

the vertices are the ones given as source or target of edges
Adjacency List:

For every vertex \( i \rightarrow \) a list of vertices \( j \) for which there is an edge \((i, j)\)

If the vertices are numbered \( \{0, \ldots, n - 1\} \), we can leave the source unspecified (it’s the index in the list)

List of lists: \([ [1, 3], [4], [0, 3], [0, 1, 4], [1, 2] ]\)
**Adjacency Matrix**: An $n \times n$ matrix of Booleans

The $(i, j)$ entry is true if there is an edge from $i$ to $j$
Space Complexity

The amount of memory necessary to store a graph depends on the representation

- With an adjacency list we need $\Theta(V + E)$ space where $V$ is the number of vertices and $E$ is the number of edges.
- With an adjacency matrix we need $\Theta(V^2)$ space independently of the number of edges.
Space Complexity

The amount of memory necessary to store a graph depends on the representation

- With an adjacency list we need $\Theta(V + E)$ space
  where $V$ is the number of vertices and $E$ is the number of edges
- With an adjacency matrix we need $\Theta(V^2)$ space
  independently of the number of edges

Which one is more convenient depends on the number of edges:

- **Sparse Graphs:**
  the number of edges is much smaller than the possible maximum $V^2$
  It is more convenient to use a adjacency list
- **Dense Graphs:**
  the number of edges is close to the possible maximum $V^2$
  It is more convenient to use a adjacency matrix
Space Complexity

The amount of memory necessary to store a graph depends on the representation

- With an adjacency list we need $\Theta(V + E)$ space
  where $V$ is the number of vertices and $E$ is the number of edges
- With an adjacency matrix we need $\Theta(V^2)$ space
  independently of the number of edges

Which one is more convenient depends on the number of edges:

- **Sparse Graphs:**
  the number of edges is much smaller than the possible maximum $V^2$
  It is more convenient to use a adjacency list
- **Dense Graphs:**
  the number of edges is close to the possible maximum $V^2$
  It is more convenient to use a adjacency matrix

Exercise: Write conversion functions between the two representations
Minimum Length Problem

Given two vertices $i$ and $j$ in a graph, find a path from $i$ to $j$ with the least number of edges.

From 0 to 3:
There is a path of length 4: $0 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 3$
But the direct path has length 1: $0 \rightarrow 3$
We may solve the problem efficiently using Dynamic Programming. Verify that the conditions for DP are met:

**Optimal Substructure**

Suppose a path $\pi : i \rightsquigarrow j$ goes through an intermediate vertex $k$:

$$i \overset{\pi_1}{\rightsquigarrow} k \overset{\pi_2}{\rightsquigarrow} j$$

If $\pi$ is a minimum path from $i$ to $j$, then
- $\pi_1$ is a minimum path from $i$ to $k$ and
- $\pi_2$ is a minimum path from $k$ to $j$
I’m trying to find a minimum path from $i$ to $j$
I use an intermediate vertex $k$; subproblems: $i \leadsto k$, $k \leadsto j$
Overlapping Subproblems

The same subproblem may occur in different branches of the computation:

I’m trying to find a minimum path from $i$ to $j$
I use an intermediate vertex $k$; subproblems: $i \leadsto k$, $k \leadsto j$
Computing $i \leadsto k$ may involve paths going from $v$ to $w$
Overlapping Subproblems

The same subproblem may occur in different branches of the computation:

I’m trying to find a minimum path from $i$ to $j$
I use an intermediate vertex $k$; subproblems: $i \leadsto k$, $k \leadsto j$
Computing $i \leadsto k$ may involve paths going from $v$ to $w$
Computing $k \leadsto j$ may also involve paths going from $v$ to $w$ (not both)
Overlapping Subproblems

The same subproblem may occur in different branches of the computation:

I’m trying to find a minimum path from $i$ to $j$
I use an intermediate vertex $k$; subproblems: $i \leadsto k$, $k \leadsto j$
Computing $i \leadsto k$ may involve paths going from $v$ to $w$
Computing $k \leadsto j$ may also involve paths going from $v$ to $w$ (not both)
The subproblem $v \leadsto w$ is recomputed several times
Exercise: Write a DP algorithm to solve the shortest path problem
Similar problem: Find the longest *simple* path between two nodes

(*simple* = contains no cycles)

Longest Path from 0 to 3, length 4:

0 → 1 → 4 → 2 → 3

With cycles we could make it as long as we want, ex length 8:

0 → 1 → 4 → 2 → 0 → 1 → 4 → 2 → 3
Can DP also be applied to this problem? Optimal Substructure?
Can DP also be applied to this problem? Optimal Substructure?

- Optimal solution for 0 $\leadsto$ 3: 0 $\rightarrow$ 1 $\rightarrow$ 4 $\rightarrow$ 2 $\rightarrow$ 3
- It goes through 1, subproblems: 0 $\leadsto$ 1 and 1 $\leadsto$ 3
Can DP also be applied to this problem? Optimal Substructure?

- Optimal solution for $0 \rightarrow 3$: $0 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 3$
- It goes through 1, subproblems: $0 \rightarrow 1$ and $1 \rightarrow 3$
- Optimal solution for $0 \rightarrow 1$: $0 \rightarrow 3 \rightarrow 4 \rightarrow 1$
- Optimal solution for $1 \rightarrow 3$: $1 \rightarrow 4 \rightarrow 2 \rightarrow 3$

We can’t put the subproblem together: cycles!
The Maximum Length Problem does not have Optimal Substructure
We can’t apply Dynamic Programming to find an efficient algorithm
In fact, this is an NP-complete problem
Weighted Graphs

We assign to every edge a weight:

![weighted graph]

Every edge is assigned a real number, its weight.

We can easily modify the adjacency list and adjacency matrix representations to include weights.
Weighted Graph Representations

- **Adjacency List**

  The entries in the list are pairs of target-vertices and edge-weights

  \[
  \begin{align*}
  0 \rightarrow & \ [(1, 1.0), (3, 2.0)] & \ [(1, 1.0), (3, 2.0)] \\
  1 \rightarrow & \ [(4, 4.0)] & \ [(4, 4.0)] \\
  2 \rightarrow & \ [(0, 10.0), (3, 5.0)] & \ [(0, 10.0), (3, 5.0)] \\
  3 \rightarrow & \ [(0, 3.0), (1, 9.0), (4, 2.0)] & \ [(0, 3.0), (1, 9.0), (4, 2.0)] \\
  4 \rightarrow & \ [(1, 6.0), (2, 7.0)] & \ [(1, 6.0), (2, 7.0)]
  \end{align*}
  \]
Weighted Graph Representations

• **Adjacency List**
  The entries in the list are pairs of target-vertices and edge-weights

  0 → [(1, 1.0), (3, 2.0)]
  1 → [(4, 4.0)]
  2 → [(0, 10.0), (3, 5.0)]
  3 → [(0, 3.0), (1, 9.0), (4, 2.0)]
  4 → [(1, 6.0), (2, 7.0)]

• **Adjacency Matrix**
  The entries in the matrix are weights instead of Booleans

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>
  0  | ·  | 1.0| ·  | 2.0| ·  |
  1  | ·  | ·  | ·  | ·  | 4.0|
  2  | 10.0| ·  | ·  | 5.0| ·  |
  3  | 3.0| 9.0| ·  | ·  | 2.0|
  4  | ·  | 6.0| 7.0| ·  | ·  |
Shortest path problem
Find a path such that the sum of the weights of its edges has the minimum possible value

We assume the weights to be non-negative
(If we allow negatives, finding the shortest is as hard as the longest path)

The version with no weights is a special case: all edges have weight 1.0
Shortest path problem
Find a path such that the sum of the weights of its edges has the minimum possible value

We assume the weights to be non-negative
(If we allow negatives, finding the shortest is as hard as the longest path)

The version with no weights is a special case: all edges have weight 1.0

Two versions:

- **Single-Source Shortest Paths**
  Fix a source vertex,
  find the shortest paths from that source to all vertices
Shortest Path Problems

**Shortest path problem**
Find a path such that the sum of the weights of its edges has the minimum possible value

We assume the weights to be non-negative
(If we allow negatives, finding the shortest is as hard as the longest path)

The version with no weights is a special case: all edges have weight 1.0

Two versions:

- **Single-Source Shortest Paths**
  Fix a source vertex,
  find the shortest paths from that source to all vertices

- **All-Pairs Shortest Paths**
  Find the shortest path between all pairs of two vertices
In the solution of the single-source shortest paths problem

- We call $w_{i,j}$ the weight of an edge from $i$ to $j$;
  If there is no edge $w_{i,j} = \infty$
- We keep an estimate $dist_i$ of
  the minimum length of a path from the source $s$ to the vertex $i$
Relaxation

In the solution of the **single-source shortest paths** problem

- We call $w_{i,j}$ the weight of an edge from $i$ to $j$;
  
  If there is no edge $w_{i,j} = \infty$

- We keep an estimate $\text{dist}_i$ of
  
  the minimum length of a path from the source $s$ to the vertex $i$

We will use an auxiliary **relaxation** algorithm to update the distances:

- Suppose we have estimated $\text{dist}_i$ without using the vertex $k$
  (That is, our estimate of $\text{dist}_i$ uses paths that don’t include $k$)

- If at one point we found the minimum distance $\text{dist}_k$,
  (so $\text{dist}_i$ is just an estimate, while $\text{dist}_k$ is the correct value)

- We can use $k$ to update the estimate $\text{dist}_i$
In the solution of the **single-source shortest paths** problem

- We call $w_{i,j}$ the weight of an edge from $i$ to $j$; If there is no edge $w_{i,j} = \infty$
- We keep an estimate $\text{dist}_i$ of the minimum length of a path from the source $s$ to the vertex $i$

We will use an auxiliary *relaxation* algorithm to update the distances:

- Suppose we have estimated $\text{dist}_i$ without using the vertex $k$ (That is, our estimate of $\text{dist}_i$ uses paths that don’t include $k$)
- If at one point we found the minimum distance $\text{dist}_k$, (so $\text{dist}_i$ is just an estimate, while $\text{dist}_k$ is the correct value)
- We can use $k$ to update the estimate $\text{dist}_i$

**Relaxation:** If $\text{dist}_k + w_{k,i} < \text{dist}_i$ then update $\text{dist}_i \leftarrow \text{dist}_k + w_{k,i}$
In our algorithm we will keep a *queue of vertices* whose distance $\text{dist}(v)$ has been estimated but not yet fixed.
Priority Queues

In our algorithm we will keep a **queue of vertices** whose distance $\text{dist}_i$ has been estimated but not yet fixed.

This will be a **Priority Queue**

A data type which represent a set of **keys** (vertices) with **values** (estimated distances) supporting the following operations:

- **Insert** a new element in the queue with associated value
- **Extract** the element with the minimum value
- **Update** the value of an element in the list
In our algorithm we will keep a queue of vertices whose distance has been estimated but not yet fixed. This will be a Priority Queue, a data type which represents a set of keys (vertices) with values (estimated distances) supporting the following operations:

- **Insert** a new element in the queue with associated value
- **Extract** the element with the minimum value
- **Update** the value of an element in the list

The algorithm iterates extracting the vertex with the minimum distance and updating the remaining vertex-distances using relaxation.
Priority Queues

In our algorithm we will keep a queue of vertices whose distance $\text{dist}_i$ has been estimated but not yet fixed.

This will be a Priority Queue.
A data type which represent a set of keys (vertices) with values (estimated distances) supporting the following operations:

- **Insert** a new element in the queue with associated value
- **Extract** the element with the minimum value
- **Update** the value of an element in the list

The algorithm iterates extracting the vertex with the minimum distance and updating the remaining vertex-distances using relaxation.

For now we can use a naive representation of queues as list of pairs or (balanced) search trees.

We will see efficient tree representations (Heaps) in future lectures: Leftist Heaps, Fibonacci Heaps.
Dijkstra’s Algorithm

Let the source vertex be \( s \)
Keep a vector \( \text{dist} \) that, for every vertex \( i \), contain an approximation \( \text{dist}_i \) of the length of the shortest path from \( s \) to \( i \)
Keep an queue \( Q \) of vertices whose distance from \( s \) has not yet been fully computed

**Dijkstra’s Algorithm:**

- Initialize the distance: \( \text{dist}_i = \infty \) for all \( i \), except \( \text{dist}_s = 0.0 \)
- Initialize the queue: \( Q = V \) all vertices
- Repeat while \( Q \) is not empty
  - Extract from \( Q \) the vertex \( i \) with the minimum \( \text{dist}_i \)
  - Relax the distances of all remaining elements of \( Q \) using \( i \)
To compute the minimum distances between all pairs of vertices, we could apply Dijkstra’s algorithm repeatedly, running the source through all vertices.
To compute the minimum distances between all pairs of vertices, we could apply Dijkstra’s algorithm repeatedly, running the source through all vertices.

A better method is the **Floyd-Warshall Algorithm**. It uses a form of Dynamic Programming. It works also with negative weights as long as there are no negative cycles.
All-pairs shortest path

To compute the minimum distances between all pairs of vertices, we could apply Dijkstra’s algorithm repeatedly, running the source through all vertices.

A better method is the Floyd-Warshall Algorithm. It uses a form of Dynamic Programming. It works also with negative weights as long as there are no negative cycles.

**Idea:** Use an growing set of intermediate vertices to construct better and better paths.

The intermediate vertices of a path $i_0 \to i_1 \to \cdots \to i_{m-1} \to i_m$ are $\{i_1, \ldots, i_{m-1}\}$. 
Floyd-Warshall Algorithm

Let $V_n$ be the set of vertices $\{0, \ldots, n - 1\}$

So $V_0 = \emptyset$, $V_1 = \{0\}$, $V_2 = \{0, 1\}$, etc.

$V_n$ is the set of all vertices

For every $k$, we compute the minimum distances $\text{dist}^{(k)}_{i,j}$ of a path from $i$ to $j$ that uses only elements of $V_k$ as intermediate vertices
Floyd-Warshall Algorithm

Let $V_n$ be the set of vertices $\{0, \ldots, n - 1\}$

So $V_0 = \emptyset$, $V_1 = \{0\}$, $V_2 = \{0, 1\}$, etc.

$V_n$ is the set of all vertices

For every $k$, we compute the minimum distances $\text{dist}^{(k)}_{i,j}$ of a path from $i$ to $j$ that uses only elements of $V_k$ as intermediate vertices

- $\text{dist}^{(0)}_{i,j} = w_{i,j}$ (if there is no edge)
- A minimum path from $i$ to $j$ that only uses intermediate vertices from $V_{k+1}$ either goes through $k$ or not
  - If it doesn’t go through $k$, then it only uses $V_k$ and $\text{dist}^{(k+1)}_{i,j} = \text{dist}^{(k)}_{i,j}$
  - If it goes through $k$, then it is made of a path from $i$ to $k$ and a path from $k$ to $j$; these paths do not use $k$ as internal vertex, so $\text{dist}^{(k+1)}_{i,j} = \text{dist}^{(k)}_{i,k} + \text{dist}^{(k)}_{k,j}$
- So $\text{dist}^{(k+1)}_{i,j} = \min(\text{dist}^{(k)}_{i,j}, \text{dist}^{(k)}_{i,k} + \text{dist}^{(k)}_{k,j})$
Floyd-Warshall Algorithm

Let $V_n$ be the set of vertices $\{0, \ldots, n - 1\}$

So $V_0 = \emptyset$, $V_1 = \{0\}$, $V_2 = \{0, 1\}$, etc.

$V_n$ is the set of all vertices

For every $k$, we compute the minimum distances $dist^{(k)}_{i,j}$ of a path from $i$ to $j$ that uses only elements of $V_k$ as intermediate vertices

- $dist^{(0)}_{i,j} = w_{i,j}$ ($dist^{(0)}_{i,j} = \infty$ if there is no edge)
- A minimum path from $i$ to $j$ that only uses intermediate vertices from $V_{k+1}$ either goes through $k$ or not
  - If it doesn’t go through $k$, then it only uses $V_k$ and $dist^{(k+1)}_{i,j} = dist^{(k)}_{i,j}$
  - If it goes through $k$, then it is made of a path from $i$ to $k$ and a path from $k$ to $j$; these paths do not use $k$ as internal vertex, so $dist^{(k+1)}_{i,j} = dist^{(k)}_{i,k} + dist^{(k)}_{k,j}$
  
- So $dist^{(k+1)}_{i,j} = \min(dist^{(k)}_{i,j}, dist^{(k)}_{i,k} + dist^{(k)}_{k,j})$

**Floyd-Warshall Algorithm:** Use the previous recursive equations to construct a sequence of matrices $(dist^{(k)}_{i,j})_{i,j=0 \ldots n-1}$ for $k = 0 \ldots n$

Return $(dist^{(n)}_{i,j})_{i,j=0 \ldots n-1}$