Graph Algorithms

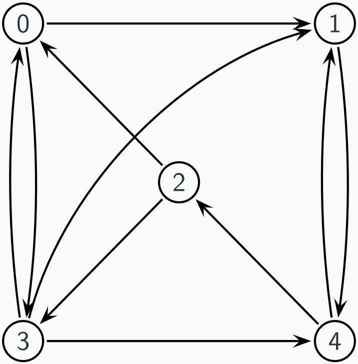
Advanced Algorithms and Data Structures - Lecture 6

Venanzio Capretta

Thursday 5 November 2020

School of Computer Science, University of Nottingham

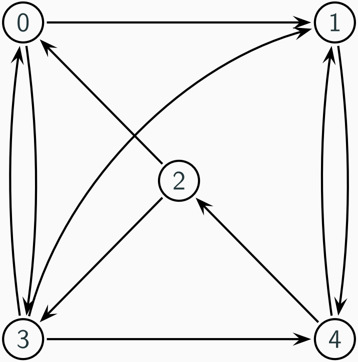
Directed Graphs



A (directed) graph consists of

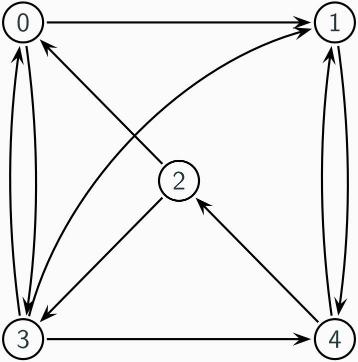
- a set of vertices: $\{0, 1, 2, 3, 4\}$
- a set of edges between the vertices: $\{(0,1),(0,3),(1,4),(2,0),(2,3),(3,0),(3,1),(3,4),(4,1),(4,2)\}$

Edge Representation



There are several ways of representing graphs as data structures:

Edge Representation



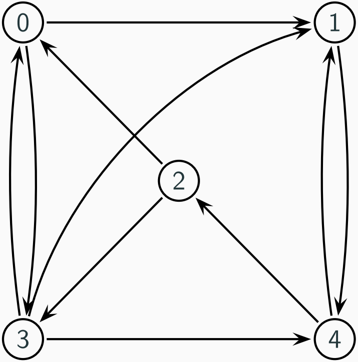
There are several ways of representing graphs as data structures:

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

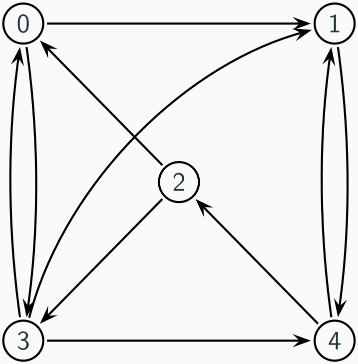


$0 \to [1, 3]$
$1 \to [4]$
$2 \to [0, 3]$
$3 \to [0, 1, 4]$
$4 \to [1, 2]$

Adjacency List:
For every vertex $i \to$ 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



	0	1	2	3	4
0	false	true	false	true	false
1	false	false	false	false	true
2	true	false	false	true	false
3	true	true	false	false	true
4	false	true	true	false	false

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 djacency 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 djacency 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 djacency 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

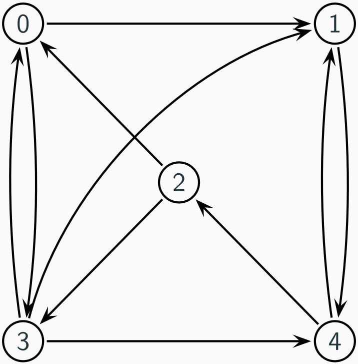
Which one is more convenient depends on the number of edges:

- 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: $0 \to 1 \to 4 \to 2 \to 3$

But the direct path has length 1: $0 \to 3$

Dynamic Programming for Minimum Path

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:

$$\underbrace{i \overset{\pi_1}{\rightsquigarrow} k \overset{\pi_2}{\rightsquigarrow} j}_{\pi}$$

If π is a minimum path from i to j, then

π_1 is a minimum path from i to k and

π_2 is a minimum path from k to j

Overlapping Subproblems

Overlapping Subproblems

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

$$i \dashrightarrow k \dashrightarrow j$$

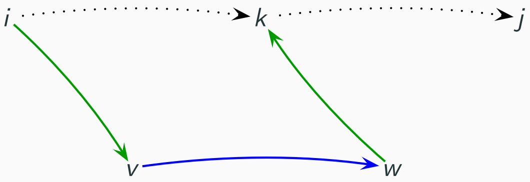
I'm trying to find a minimum path from i to j

I use an intermediate vertex k; subprobems: $i \rightsquigarrow k$, $k \rightsquigarrow j$

Overlapping Subproblems

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

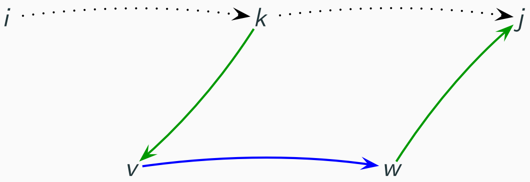
I use an intermediate vertex k; subprobems: $i \rightsquigarrow k$, $k \rightsquigarrow j$

Computing $i \rightsquigarrow k$ may involve paths going from v to w

Overlapping Subproblems

Overlapping Subproblems

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



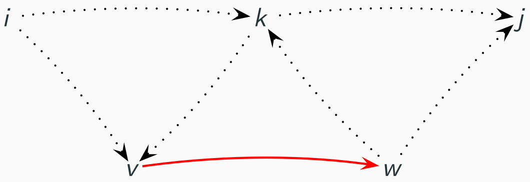
I'm trying to find a minimum path from i to j
I use an intermediate vertex k; subprobems: $i \rightsquigarrow k$, $k \rightsquigarrow j$

Computing $i \rightsquigarrow k$ may involve paths going from v to w
Computing $k \rightsquigarrow j$ may also involve paths going from v to w (not both)

Overlapping Subproblems

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; subprobems: $i \rightsquigarrow k$, $k \rightsquigarrow j$

Computing $i \rightsquigarrow k$ may involve paths going from v to w

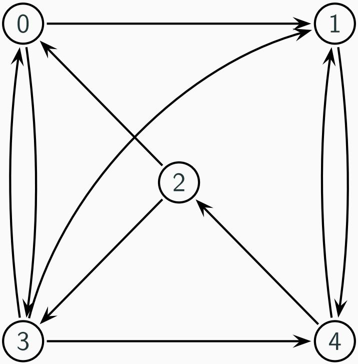
Computing $k \rightsquigarrow j$ may also involve paths going from v to w (not both)

The subproblem $v \rightsquigarrow w$ is recomputed several times

Exercise: Write a DP algorithm to solve the shortest path problem

Longest Path Problem

Similar problem: Find the longest simple path between two nodes (simple = contains no cycles)



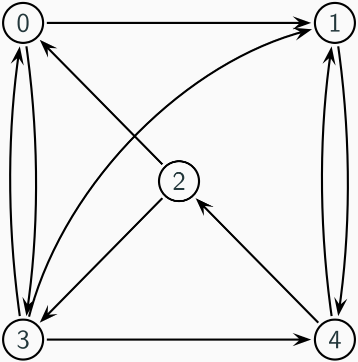
Longest Path from 0 to 3, length 4: $0 \to 1 \to 4 \to 2 \to 3$

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

$0 \to 1 \to 4 \to 2 \to 0 \to 1 \to 4 \to 2 \to 3$

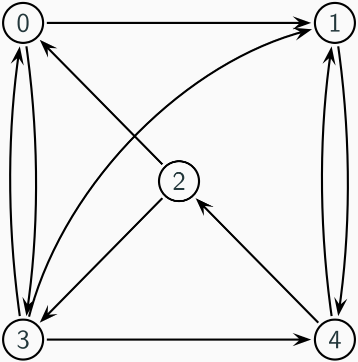
DP for Maximum Length?

Can DP also be applied to this problem? Optimal Substrcture?



DP for Maximum Length?

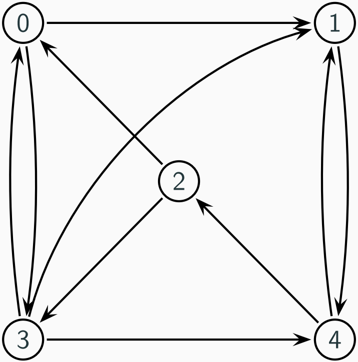
Can DP also be applied to this problem? Optimal Substrcture?



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

DP for Maximum Length?

Can DP also be applied to this problem? Optimal Substrcture?



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

  We can't put the subproblem together: cycles!

No DP for Maximum Length

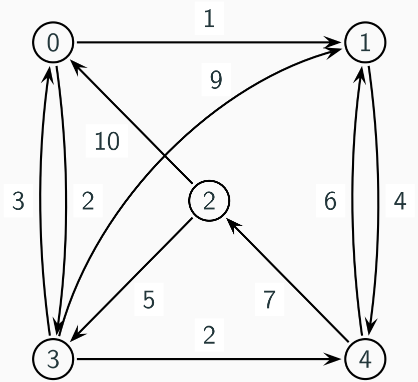
The Maximum Length Problem does not have Optimal Substrcture

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:



Every edge is assign 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

$$
\begin{aligned}
0 &\rightarrow [(1, 1.0), (3, 2.0)] \\
1 &\rightarrow [(4, 4.0)] \\
2 &\rightarrow [(0, 10.0), (3, 5.0)] \\
3 &\rightarrow [(0, 3.0), (1, 9.0), (4, 2.0)] \\
4 &\rightarrow [(1, 6.0), (2, 7.0)]
\end{aligned}
$$

$$
\begin{array}{l}
[[(1, 1.0), (3, 2.0)] \\
\;[(4, 4.0)] \\
\;[(0, 10.0), (3, 5.0)] \\
\;[(0, 3.0), (1, 9.0), (4, 2.0)], \\
\;[(1, 6.0), (2, 7.0)]]
\end{array}
$$

Weighted Graph Representations

- Adjacency List

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

$$
\begin{aligned}
0 &\rightarrow [(1, 1.0), (3, 2.0)] \\
1 &\rightarrow [(4, 4.0)] \\
2 &\rightarrow [(0, 10.0), (3, 5.0)] \\
3 &\rightarrow [(0, 3.0), (1, 9.0), (4, 2.0)] \\
4 &\rightarrow [(1, 6.0), (2, 7.0)]
\end{aligned}
$$

$$
\begin{array}{l}
[[(1, 1.0), (3, 2.0)] \\
\ [(4, 4.0)] \\
\ [(0, 10.0), (3, 5.0)] \\
\ [(0, 3.0), (1, 9.0), (4, 2.0)], \\
\ [(1, 6.0), (2, 7.0)]]
\end{array}
$$

- Adjecency Matrix

  The entries in the matrix are weights instead of Booleans

	0	1	2	3	4
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 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, findind 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 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, findind 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, findind 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

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 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 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 dist_i without using the vertex k
  (That is, our estimate of dist_i uses paths that don't include k)
- If at one point we found the minimum distance dist_k,
  (so dist_i is just an estimate, while dist_k is the correct value)
- We can use k to update the estimate dist_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 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 dist_i without using the vertex k
  (That is, our estimate of dist_i uses paths that don't include k)
- If at one point we found the minimum distance dist_k,
  (so dist_i is just an estimate, while dist_k is the correct value)
- We can use k to update the estimate dist_i

RELAXATION: If $\text{dist}_k + w_{k,i} < \text{dist}_i$ then update $\text{dist}_i \leftarrow \text{dist}_k + w_{k,i}$

Priority Queues

In our algorithm we will keep a queue of vertices
whose distance dist_i has been estimated but not yet fixed

Priority Queues

In our algorithm we will keep a queue of vertices
whose distance 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

Priority Queues

In our algorithm we will keep a queue of vertices
whose distance 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

Priority Queues

In our algorithm we will keep a queue of vertices
whose distance 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 dist that, for every vertex i, contain an approximation 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 dist_i
  - Relax the distances of all remaining elements of Q using i

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

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

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 $\text{dist}_{i,j}^{(k)}$ 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 $\{0, \dots, n-1\}$
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 $\text{dist}_{i,j}^{(k)}$ of a path from i to j that uses only elements of V_k as intermediate vertices

- $\text{dist}_{i,j}^{(0)} = w_{i,j}$ ($\text{dist}_{i,j}^{(0)} = \infty$ 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 $\text{dist}_{i,j}^{(k+1)} = \text{dist}_{i,j}^{(k)}$
  - 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 $\text{dist}_{i,j}^{(k+1)} = \text{dist}_{i,k}^{(k)} + \text{dist}_{k,j}^{(k)}$
- So $\text{dist}_{i,j}^{(k+1)} = \min(\text{dist}_{i,j}^{(k)}, \text{dist}_{i,k}^{(k)} + \text{dist}_{k,j}^{(k)})$

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 $\text{dist}_{i,j}^{(k)}$ of a path from i to j that uses only elements of V_k as intermediate vertices

- $\text{dist}_{i,j}^{(0)} = w_{i,j}$ ($\text{dist}_{i,j}^{(0)} = \infty$ 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 $\text{dist}_{i,j}^{(k+1)} = \text{dist}_{i,j}^{(k)}$
  - 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 $\text{dist}_{i,j}^{(k+1)} = \text{dist}_{i,k}^{(k)} + \text{dist}_{k,j}^{(k)}$
- So $\text{dist}_{i,j}^{(k+1)} = \min(\text{dist}_{i,j}^{(k)}, \text{dist}_{i,k}^{(k)} + \text{dist}_{k,j}^{(k)})$

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