4-Shortest Paths Problems

Bruno MARTIN,
University of Nice - Sophia Antipolis
`mailto:Bruno.Martin@unice.fr`
`http://deptinfo.unice.fr/~bmartin/MathMods`

Shortest-Paths Problems on Digraphs

Given a route map, we may be interested in questions like:
"What is the fastest way to get from city x to city y?"

The Shortest Path between x and y : G. Dantzig

"What is the fastest way to get from city x to every other city?"

The Single-Source Shortest-Paths Problem : G. Dantzig

"What is the fastest way to get from every city to every other?"

All Pairs Shortest Paths : R. W. Floyd

Construct a graph G in which each vertex represents a city and each directed edge a route between cities. The label on edge $x \to y$ is the time to travel from one city to the other.

The Shortest-Path Problem : G. Dantzig

Problem: Find the SP $s \rightsquigarrow t$ in $G = (\{v_1, ..., v_n\}, E)$ valuated

$dist[v_i]$ (**array**) stores the **the shortest path length** from s to v_i.
Let S be the set of elements on which $dist$ is defined
$weight(i, j)$ a **function** which gives the value of $i \to j$ if it exists
$pred[v_i]$ stores the predecessor of v_i or nil

Initially $dist[s] = 0$; $\forall v \neq s$ $dist[v] = +\infty$, $pred[v] = nil$ ($S = \{s\}$)

First step: iterate on the adjacency list of s. We keep the vertex $v \notin S$ so that the value of the edge $s \to v$ is minimum and update $dist[v]$. The set S now contains s and v

The Shortest-Path Problem : G. Dantzig (continued)

At step k

- $dist$ is defined on k vertices $v_1, ..., v_k$
- $\forall v_j \in S$, iterate on its **adjacency list** in order to find the **edge towards a vertex** $w_j \notin S$ with the **smallest distance**
- find the **index** j st $dist[v_j] + weight(v_j, w_j)$ is **minimum**
- update $dist[w_j]$ with this value and insert w_j in S
- **stop** as soon as we reach t

Complexity : v_j is taken in **constant time**. What remains are the k comparisons to choose w_j. The **maximum number of comparisons** is $1 + 2 + ... + V = V(V-1)/2$.

Relaxation

An edge $u \to v$ is *tense* if

$$dist[u] + weight(u, v) < dist[v]$$

If $u \to v$ is tense, the tentative (current) SP $s \rightsquigarrow u \to v$ is shorter. The algorithm finds a tense edge in G and *relaxes* it:

```
relax($u \to v$)
    $dist[v] = dist[u] + weight(u, v)$
    $pred(v) = u$
```

Detecting an edge which can be relaxed is like a graph traversal with a set S a vertices, initially containing $\{s\}$.
When taking u out of S, we scan its outgoing edges for something to relax. When we relax an edge $u \to v$, we put v in S.
Contrarily to traversal, the same vertex can be visited many times.

Why Dantzig works ?

There can't be a SP $s \rightsquigarrow v_j$ shorter than the one chosen by the algorithm

- $dist[v_j]$ chosen as the SP whose intermediate vertices are in S
- Suppose it exists a shorter path containing vertices not in S
    - $\exists v \notin S, so\ that\ s \rightsquigarrow v \rightsquigarrow w_j$ is shorter than $s \rightsquigarrow w_j$
    - In that case we should have selected v in the algorithm

If the shortest paths are unique, they form a tree (*spanning tree*). Observe that any subpath of a SP is also a SP. If there are multiple shortest paths to the same vertices, we can always chose a path to each vertex so that the union of the path is a tree.

The Single-Source Shortest-Paths Problem : G. Dantzig

You don't stop when reaching t
You continue until every vertices are in the set S
You have computed the **single-source shortest-paths for** s

This algorithm is attributed to **Dijkstra**

All this kind of algorithms are special cases of an algorithm proposed by Ford in 1956 or independently by Dantzig in 1957.

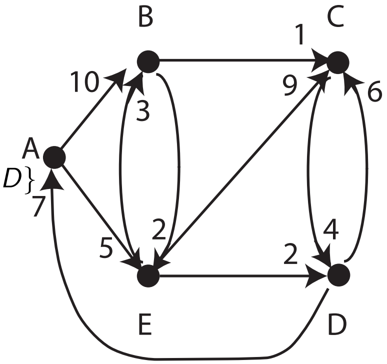
Single source SP algorithm

```
initSSSP(s)
  dist[s]=0
  pred[s]=nil
  forall vertices v != s
    dist[v]=infinite
    pred[v]=nil
```

```
SSSP(s)
  initSSSP(s)
  S={s}
  while not empty?(S)
    take u from S
    forall edges (u,v)
      if dist[u]+weight(u,v)<dist[v]
        dist[v]=dist[u]+weight(u,v)
        pred[v]=u
        S= S union {v}
```

Example SSSP from A

A	B	C	D	E	
0	∞	∞	∞	∞	$S=\{A\}$
0	$10A$	∞	∞	$5A$	$S=\{B,E\}$
0	$8E$	$14E$	$7E$	$5A$	$S=\{B,C,D\}$
0	$8E$	$13D$	$7E$	$5A$	$S=\{B,C\}$
0	$8E$	$13D$	$7E$	$5A$	$S=\{B\}$
0	$8E$	$9B$	$7E$	$5A$	$S=\varnothing$



Matrix multiplication algorithm

Here's the structure of the problem for $u, v \in V$

1. if $u = v$, then the SP from u to v is 0
2. oherwise, decompose $P = u \rightsquigarrow x \rightsquigarrow v$ where $P' = u \rightsquigarrow x$ contains at most k edges and is the SP from u to x

A recursive solution: Let d_{ij}^k the minimum weight of any path from i to j that contains at most k edges.

1. if $k=0$ then $d_{ij}^0 = \begin{cases} 0 \text{ if } i=j \\ \infty \text{ if } i \neq j \end{cases}$
2. Otherwise, for $k \geq 1$, d_{ij}^k is computed from d_{ij}^{k-1} and the weights adjacency matrix A:

$$d_{ij}^k = \min\left\{d_{ij}^{k-1}, \min_{1\leq \ell \leq n}\{d_{i\ell}^{k-1} + A(\ell, j)\}\right\}$$

All-Pairs Shortest-Path problem : R. W. Floyd

Problem: Find for each ordered pair of vertices $(v,\ w)$ the length of the SP from v to w in the digraph $G(V,E)$

Obvious solution: run the previous SSSP from every vertex.
In this case, this leads to a $O(V^3)$ algorithm with complex data structures and $O(V^3 \log V)$ with classical data structures.

All-Pairs Shortest-Path problem : R. W. Floyd

Problem: Find for each ordered pair of vertices $(v,\ w)$ the length of the SP from v to w in the digraph $G(V,E)$

$A[V \times V]$ is a matrix; $A[i,j]$ stores the length of the SP from i to j
A function *weight*(i,j) gives the value of the edge between i and j if it exists ∞ otherwise

Initially A stores the *weight* of each existing edge, ∞ otherwise and 0 on the diagonal

All-Pairs Shortest-Path problem : R. W. Floyd (continued)

We iterate on the vertices of the graph
At the k^{th} iteration:

- $A[i,j]$ is the shortest path from i to j that passes only through vertices $\{1, ..., k-1\}$
- $A[i,j] = min(A[i,j],\ A[i,k] + A[k,j])$
- If we need to retrieve the path to go from i to j use an additional matrix ($Path[i,j] = k$ if relevant)

The Floyd's algorithm

```
floyd
  for  i = 0 to numberOfVertices
    for j = 0 to numberOfVertices
      if (weight(i,j) != nil) A[i,j] = weight(i,j)
      else A[i,j] = Infinite
      Path[i,j]=-1;
  for i = 0 to numberOfVertices
      A[i,i] = 0
  for k = 0 to numberOfVertices
    for i = 0 numberOfVertices
      for j = 0 to numberOfVertices
        if (A[i,k]+A[k,j] < A[i,j])
          A[i,j] = A[i,k] + A[k,j]
          Path[i,j]=k
```

How Floyd's algorithm works

For each vertex k in V, we run through the entire matrix A
Before the iteration for the vertex k, the existing $A[i,j]$ does not pass through the vertex k
If it is faster to go from i to j by passing through k, we take $A[i,k] + A[k,j]$ as the new $A[i,j]$ value
The running time is clearly $O(V^3)$ three nested loops

Floyd's algorithm in Ruby

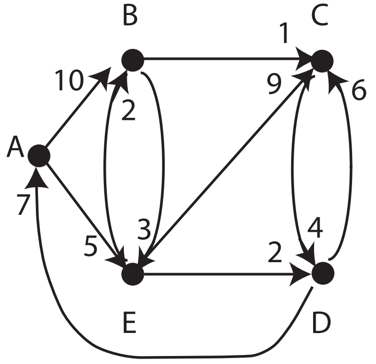
```
def floyd
    @graph.each_index do |k|
      @graph.each_index do |i|
       @graph.each_index do |j|
        if (@graph[i][j] == "inf.") && (@graph[i][k] != "inf."
              && @graph[k][j] != "inf.")
            @graph[i][j] = @graph[i][k]+@graph[k][j]
            @pre[i][j] = @pre[k][j]
        elsif (@graph[i][k] != "inf." && @graph[k][j] != "inf.")
                 && (@graph[i][j] > @graph[i][k]+@graph[k][j])
            @graph[i][j] = @graph[i][k]+@graph[k][j]
            @pre[i][j] = @pre[k][j]
          end
        end
      end
    end
  end
```

Floyd Example

k	0	*B*	*C*	*D*	*E*	*A*
AB	10	10	10	10	7	7
AC	∞	11	11	11	8	8
AD	∞	∞	15	15	7	7
AE	5	5	5	5	5	5
BA	∞	∞	∞	12	12	12
BC	1	1	1	1	1	1
BD	∞	∞	5	5	5	5
BE	3	3	3	3	3	3
CA	∞	∞	∞	11	11	11
CB	∞	∞	∞	∞	∞	18
CD	4	4	4	4	4	4
CE	∞	∞	∞	∞	∞	16
DA	7	7	7	7	7	7
DB	∞	∞	∞	∞	∞	14
DC	6	6	6	6	6	6
DE	∞	∞	∞	∞	∞	12
EA	∞	∞	∞	9	9	9
EB	2	2	2	2	2	2
EC	9	3	3	3	3	3
ED	2	2	2	2	2	2

$$\begin{pmatrix} 0 & 10 & \infty & \infty & 5 \\ \infty & 0 & 1 & \infty & 3 \\ \infty & \infty & 0 & 4 & \infty \\ 7 & \infty & 6 & 0 & \infty \\ \infty & 2 & 9 & 2 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 10 & \infty & \infty & 5 \\ \infty & 0 & 1 & \infty & 3 \\ \infty & \infty & 0 & 4 & \infty \\ 7 & \infty & 6 & 0 & \infty \\ \infty & 2 & 9 & 2 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 7 & 11 & 7 & 5 \\ \infty & 0 & 1 & 5 & 3 \\ \infty & \infty & 0 & 4 & \infty \\ 7 & 17 & 6 & 0 & 12 \\ 9 & 2 & 3 & 2 & 0 \end{pmatrix}$$



Warshall's algorithm

```
warshallAlgorithm
 for i = 0 to numberOfVertices
   for j = 0 to numberOfVertices
      if (weight(i,j) != nil) then A[i,j] = TRUE
      else A[i,j] = FALSE
 for k = 0..numberOfVertices
   for i = 0..numberOfVertices
     for j = 0..numberOfVertices
       if (A[i,j] == FALSE) then
         A[i,j] = A[i,k] && A[k,j]
```

Transitive Closure : Warshall's Algorithm

In some problems we may need to know only whether there exists a path from vertex i to vertex j in the digraph $G(V, E)$

We specialize Floyd's algorithm

- $weight(i, j) = TRUE$ if there is an edge from i to j, $FALSE$ otherwise
- We wish to compute the matrix A such that $A[i, j] = TRUE$ if there is a path from i to j and $FALSE$ otherwise
- A is called the **transitive closure** for the adjacency matrix

How it works

- For each vertex $k \in V$, we run through the entire matrix A
- If there is no path from i to j ($A[i, j] = FALSE$), we test if there is a path from i to j going through k ($A[i, k]$ and $A[k, j]$) and we update A if needed

Improvement by repeated squaring

Inside k loop, each A_k matrix contains the SP of at most k edges. What we were doing: "Given the SP of at most length k, and the SP of at most length 1, what is the SP of at most length $k + 1$?" Repeated squaring method: "Given the SP of at most length k, what is the SP of at most length $k + k$?" The correctness of this approach lies in the observation that the SP of at most m edges is the same as the shortest paths of at most $n - 1$ edges for all $m > n - 1$. Thus:

$$\begin{array}{l} A_1 = W \\ A_2 = W^2 = W \cdot W \\ A_4 = W^4 = W^2 \cdot W^2 \\ \vdots \\ A_{2\lceil\log(n-1)\rceil} = W^{\lceil\log(n-1)\rceil} \cdot W^{\lceil\log(n-1)\rceil} \end{array}$$

With repeated squaring, we run the algorithm $\lceil\log(n - 1)\rceil$ times