Algorithms and Data Structures

Shortest Paths

Marcin Sydow

Web Mining Lab
PJWSTK

Topics covered by this lecture:

- Shortest Path Problem
- Variants
- Relaxation
- DAG
- Nonnegative Weights (Dijkstra)
- Arbitrary Weights (Bellman-Ford)
- (*)All-pairs algorithm

Example: Fire Brigade

Consider the following example. There is a navigating system for the fire brigade. The task of the system is as follows. When there is a fire at some point f in the city, the system has to quickly compute the shortest way from the fire brigade base to the point f. The system has access to full information about the current topology of the city (the map includes all the streets and crossings in the city) and the estimated latencies on all sections of streets between all crossings.

(notice that some streets can be unidirectional. Assume that the system is aware of this)

The Shortest Paths Problem

INPUT: a graph $G = (V, E)$ with weights on edges given by a function $w : E \rightarrow R$, a starting node $s \in V$

OUTPUT: for each node $v \in V$

- the shortest-path distance $\mu(s, v)$ from s to v, if it exists
- the parent in a shortest-path tree (if it exists), to reconstruct the shortest path

The shortest path may not exist for some reasons:

- there may be no path from s to v in the graph
- there may be a *negative cycle* in the graph (there may be no lower bound on the length in such case)

Variants

When we design the best algorithm for the shortest paths problem, we can exploit some special properties of the graph, for example:

- the graph G is directed or undirected
- the weights are non-negative or integer, etc. (each case can be approached differently to obtain the most efficient algorithm)
- the graph is acyclic, etc.

We will see different algorithms for different variants

Shortest Paths - properties

If $G = (V, E)$ is an arbitrary graph and $s, v \in V$, we consider the following convention for the value of $\mu(s, v)$, the shortest-path distance from s to v in G:

- $\mu(s, v) = +\infty$ (when there is no path from s to v)
- $\mu(s, v) = -\infty$ (when there exists a path from s to v that contains a negative cycle)
- $\mu(s, v) = d \in R$ (else)

(we will use the denotation $\mu(v)$ instead of $\mu(s, v)$ if s is clear from the context)

Notice the following important property of shortest paths. A subpath of a shortest path is itself a shortest path (i.e. if $(v_1, ..., v_{k-1}, v_k)$ is a shortest path from v_1 to v_k then $(v_1, ..., v_{k-1})$ must be a shortest path from v_1 to v_{k-1} – simple proof by contradiction)

The idea of computing shortest paths

The general idea is similar to BFS from a source node s. We keep two attributes with each node v:

- `v.distance`: that keeps the shortest (currently known) distance from s to v
- `v.parent`: that keeps the predecessor of v on the shortest (currently known) path from s

Initialisation: `s.distance = 0`, `s.parent = s`, and all other nodes have `distance` set to infinity and parent to `null`.

The attributes are updated by "propagation" via edges: it is called a *relaxation* of an edge.

The Concept of Edge Relaxation


```
relax((u,v))              # (u,v) is an edge in E
  if u.distance + w(u,v) < v.distance
    v.distance = u.distance + w(u,v)
    v.parent = u
```

The algorithms relax edges until the shortest paths have been found or a negative cycle has been discovered.

Relaxation has some important properties, eg:
(assuming the prior initialisation as described before)
- after any sequence of relaxations, $\forall v \in V$ `v.distance` $\geq \mu(v)$
(simple proof by induction on the number of edge relaxations) (the distance discovered by relaxations never drops below the shortest distance)

It can be proven (by induction) that the relaxations indeed are the right tool to compute the shortest paths.

Correctness of Relaxation

Lemma

After doing a sequence R of edge relaxations that contains (as a subsequence) a shortest path $p = (e_1, ..., e_k)$ from s to v, $v.distance = \mu(s, v)$.

Proof: Because p is a shortest path, we have $\mu(v) = \sum_{1 \leq j \leq k} w(e_j)$. Let v_i denote the target node of e_i, for $0 < i \leq k$ ($v_0 = s$). We show, by induction, that after the i-th relaxation $v_i.distance \leq \sum_{1 \leq j \leq i} w(e_j)$. It is true at the beginning (s.distance == 0). Then, after the i-th relaxation, $v_i.distance \leq v_{i-1}.distance + w(e_i) \leq \sum_{1 \leq j \leq i} w(e_j)$ (by relaxation definition and induction). Thus, after k-th relaxation $v.distance \leq \mu(v)$. But $v.distance$ cannot be lower than $\mu(v)$ (previous slide), so that $v.distance == \mu(v)$ holds.

Shortest Paths in a Directed Acyclic Graph (case 1)

A directed acyclic graph (DAG) $G = (V, E)$ is a simple case for computing shortest paths from a source s.

First, any DAG can be topologically sorted (e.g. by DFS) in $O(m + n)$ time (where $m = |E|$ and $n = |V|$, and edge traversing is the dominating operation), resulting in a sequence of vertices $(v_1, ..., v_n)$.
Then, assuming $s = v_j$, for some $0 < j \leq n$, we can relax all edges outgoing of v_j, then all edges outgoing from v_{j+1}, etc.

Thus, each edge is relaxed at most once. Since each relaxation has constant time, the total time complexity is $O(m + n)$.

Dijkstra's Algorithm: only non-negative weights (case 2)

Another case is when there are no negative weights on edges (thus, there are no negative cycles, but a cycle can exist so that topological sorting is not possible).

The idea is as follows. If we relax edges in the order of non-decreasing shortest-path distance from the source s, then each edge is relaxed exactly once.

high-level pseudo-code of Dijkstra's algorithm:

```
initialise
while(there is an unscanned node with finite distance)
  u = minimum-distance unscanned node
  relax all edges (u,v)
  u becomes scanned
```

In the following proof of correctness, for any node v we will denote $v.distance$ as $d(v)$, for short.

Correctness of the Dijkstra's Algorithm

It is quite easy to observe that each node v reachable from s will be scanned by the Dijkstra's algorithm (e.g. by induction on the unweighted length of the shortest path from s to v)

To complete, we prove that when any node v is scanned, $d(v) = \mu(v)$. By condtradiction, let t be the first moment in time when any node v is scanned with $d(v) > \mu(v)$ (i.e. v contradicts what we want to prove). Let $(s = v_1, ..., v_k = v)$ be the shortest path from s to this node and let i be the smallest index on this path such that v_i was not scanned before the time t. Since $v_1 = s$ and $d(s) = 0 = \mu(s)$, it must be that $i > 1$. By the definition of i, v_{i-1} was scanned before the time t and $d(v_{i-1}) = \mu(v_{i-1})$ when it is scanned (due to the definition of t). Thus, when v_{i-1} is scanned, $d(v_i)$ is set to $d(v_{i-1}) + w(v_{i-1}, v_i) = \mu(v_{i-1}) + w(v_{i-1}, v_i)$ (because the path is a shortest one). To summarise, at time t we have: $d(v_i) = \mu(v_i) \leq \mu(v_k) < d(v_k)$ so that v_i should be scanned instead of v_k (and they are different due to the sharp inequality) – a contradiction.

Algorithm (Dijkstra)

(pq - a priority queue with decreaseKey operation, priority is the value of distance attribute)

```
s.distance = 0
pq.insert(s)
s.parent = s

for-each v in V except s:
   v.distance = INFINITY
   v.parent = null

while(!pq.isEmpty())
   scannedNode = pq.delMin()
   for-each v in scannedNode.adjList:
      if (v.distance > scannedNode.distance + w(scannedNode, v))
         v.distance = scannedNode.distance + w(scannedNode, v)
         v.parent = scannedNode
         if (pq.contains(v)) pq.decreaseKey(v)
         else pq.insert(v)
```

(to efficiently implement contains and decreaseKey operations of priority queue, an additional dictionary can be kept for mapping from nodes to their positions (pointers) in priority queue)

Dijkstra - analysis

data size: n = |V|, m = |E|

dominating operation: comparison of priorities (inside priority queue), attribute update

initialisation: $O(n)$

loop: O(n $\times$ (delMin + insert) + m $\times$ decreaseKey)

$O(nlogn) + O(mlogn) =$ **O((n + m)logn)** (if binary Heap is used)

(*)Faster implementations of Dijkstra's algorithm

If we use Fibonacci Heap we can accelerate $O((n+m)logn)$ complexity to:

$O(m + nlogn)$

Fibonacci Heap is a fast implementation of priority queue that guarantees amortised O(1) time of `decreaseKey` operation.[1]

Further, if the weights are integers bound by a constant C, we can reduce time complexity of the Dijkstra's algorithm even to:

$O(m + nC)$

by applying so called *bucket queue* as a *monotone* integer priority queue[2]

[1] Fibonacci Heap is not discussed in this lecture

[2] There are implementations of priority queue that exploit the fact that new-coming priorities are non-decreasing integers to accelerate the operations.

Bellman-Ford's Algorithm: works for any weights (case 3)

If the input graph is acyclic (case 1) or has only non-negative weights (case 2), there exist algorithms that compute shortest paths with at most m edge relaxations (as was demonstrated). If the weights are arbitrary, however, it suffices to do $O(nm)$ relaxations, anyway.

The idea is as follows (Bellman-Ford's algorithm):
In each of $n-1$ iterations, all the m edges are relaxed. Since any shortest path has at most $n-1$ edges, it must be included (as a subsequence) in any sequence of $m(n-1)$ such relaxations. Afterwards, unreachable nodes will still have `v.d == ` ∞ .

The only remaining problem now is to detect the nodes v with $\mu(v) == -\infty$. But we can easily detect such nodes v by testing whether $u.d + w(e) < v.d$ for any edge (u, v). Further, if any node v satisfies that condition, also any node w reachable from it has $\mu(w) == -\infty$.

Bellman-Ford's Algorithm


```
(initialise as in Dijkstra)

for(i = 1; i <= (n-1); i++)
   for each e in E
      relax(e)

for each e=(u,v) in E
   if (u.distance + w(u,v) < v.distance)
      identifyNegativeCycle(v)

***

identifyNegativeCycle(v)
   if (v.distance > -infinity)
      v.distance = -infinity
      for each w in v.adjList
         identifyNegativeCycle(w)
```

(*) All-Pairs Shortest Paths (for graphs with no negative cycles)

If we need to compute the shortest paths for all the possible pairs of nodes in a graph, it is possible to run any single-source-shortest-path algorithm from each of the nodes (as a source) which results in $O(n^2 m)$ time complexity (n times Bellman-Ford's algorithm) if nothing can be assumed about the weights or acyclicity of the graph.

However, for a graph without negative cycles, it is possible to save some redundant work in such naive approach and reduce it to $O(nm + n^2 logn)$ time complexity by running one universal single-source variant of algorithm plus n times non-negative variants on a properly modified graph. This is done by a notion of *node potential*.

(*) All-Pairs Shortest Paths (no negative cycles): Using node potentials to avoid negative weights

Lemma

Let $G = (V, E)$ be a graph with an edge-weight function $w : E \to R$ without negative cycles and $pot : V \to R$ be a function on vertices (called potential). We call $w' : E \to R$ a reduced weight function if, for any $e = (u, v) \in E$ it is defined as
$w'(e) = w(e) + pot(v) - pot(w)$.

- *weight reduction preserves shortest paths*
- *if all nodes can be reached from some node s, and potential is defined as $pot(v) = \mu(s, v)$ (shortes path distance from s) for all nodes v, the reduced weights are non-negative*

(*) All-Pairs Shortest Paths (no negative cycles): A pseudo-code of an algorithm


```
allPairsShortestPathsWithNoNegCycles(){
add an artificial node s with zero-weight edges (s,v) to all nodes v

use Bellman-Ford's algorithm to compute shortest paths from s

for each node v:
   define pot(v) as the shortest path computed above

for each node v:
   run Dijkstra's algorithm from v on reduced weights
   (as the reduced weights are non-negative)

for each edge e=(u,v):
   translate back from reduced distances (**)
   (as the shortest paths are preserved)
}
```

(**)The translation is done as follows:

$\mu(v, w) = \mu_{reduced}(v, w) + pot(w) - pot(v)$

Summary

- Shortest Path Problem
- Variants
- Relaxation
- DAG
- Nonnegative Weights (Dijkstra)
- Arbitrary Weights (Bellman-Ford)
- (*)All-pairs algorithm

Thank you for attention