DISTANCE PROBLEMS IN NETWORKS - THEORY AND PRACTICE

Classical Shortest-Path Algorithms

Nesrine Damak

October 10, 2010

Abstract

In this work, we present four algorithms for the shortest path problem. The first is a greedy approach that yields the optimal solution when all arc weights are positiv. If some weights are negative but no negative cycles exist, Bellman-Ford algorithm may be used. For all pairs shortest path-problem is Floyd-Warshall and Johnson algorithm the appropriate ones.

Contents

1 Introduction

Definition 1.1. *A graph G(V,E) is a set V of vertices, a set E of edges, and a real-valued weight function* $f : E \longrightarrow R$.

Definition 1.2. *A path P in G is a sequence of vertices* $(v_1, .., v_n)$ *such that* $(v_i, v_{i+1}) \in E$ *for all* $1 \leq i < n$ *and* $v \in V$.

Definition 1.3. *The weight of a path P in G is the sum of the weights of its edges:* $w(P) = \sum_i w(v_{i-1}, v_i)$.

Definition 1.4. *For nodes* $u, v \in V$, *the shortest path weight from u to v is defined to be:* $\delta(u, v) = min(w(P))$ *if such a P exists or infinity otherweise.*

Definition 1.5. *For nodes* $u, v \in V$ *a shortest path from u to v is a path with* $w(P) = \delta(u, v)$

The problem of finding shortest paths in a graph has a surprising variety of applications:

- The most obvious applications arise in transportation or communications, such as finding the best route to drive between Berlin and Munich or figuring how to direct packets to a destination across a network.

- Consider the problem of image segmentation[1], that is, separating two characters in a scanned, bit-mapped image of printed text. We need to find the separating line between two points that cuts through the fewest number of black pixels. This grid of pixels can be modeled as a graph, with any edge across a black pixel given a high cost. The shortest path from top to bottom defines the best separation between left and right.

- A major problem in speech recognition is distinguishing between words that sound alike (homophones), such as to, two, and too. We can construct a graph whose vertices correspond to possible words, with an edge between possible neighboring words. If the weight of each edge measures the likelihood of transition, the shortest path across the graph defines the best interpretation of a sentence.

- Suppose we want to draw an informative picture of a graph. The center of the page should correspond to the "center" of the graph, whatever that means. A good definition of the center is the vertex that minimizes the maximum distance to any other vertex in the graph. Finding this center point requires knowing the distance (i.e. shortest path) between all pairs of vertices.

2 Single-Source Shortest Paths

Input: A graph $G = (V, E)$, an edge weight function w and a node $s \in V$.
Output: A shortest path P from s to all other vertices $v \in V \ \{s\}$.

[1] In computer vision, segmentation refers to the process of partitioning a digital image into multiple segments (sets of pixels, also known as superpixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze.

2.1 Breadth-first search (BFS): unweighted case

The breadth-first search is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.
The graph is unweighted, that means: $w(u, v) = 1$ for all $uv \in E$. We use a simple FIFO queue,so that the time complexity is: $O(V + E)$

BFS(G)
while $Q \neq empty$ **do**
  $u \leftarrow DEQUEUE(Q)$;
  for each $v \in adj(u)$ if$dist(v) = \infty$ $dist(v) = dist(u) + 1$;
  $ENQUEUE(Q, v)$;
end

Algorithm 1: Breadth-first search

2.2 Dijkstra-algorithm for no negativ edge-weights

Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1956 and published in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with no negative edge path costs, producing a shortest path tree.
Algorithm starts at the source vertex, s, it grows a tree, T, that ultimately spans all vertices reachable from S. Vertices are added to T in order of distance, i.e., first S, then the vertex closest to S, then the next closest, and so on. Following implementation assumes that graph G is represented by adjacency lists. We also define $dist(x)$ as the distance from the initial node to x.

Data: G=(V,E)
Result: Shortest path
dist(s) $\leftarrow 0$;
for $v \in V - \{s\}$ **do**
  dist(v)$\leftarrow \infty$;
end
$S \leftarrow empty$;
$Q \leftarrow V$ (Q is a priority queue maintaining V-S) ;
while $Q \neq empty$ **do**
  $u \leftarrow Extract - Min(Q)$;
  $S \leftarrow S \cup \{u\}$;
  for $v \in adj(u)$ **do**
    if $dist(v) > dist(u) + w(u, v)$ **then**
      $dist(v) \leftarrow dist(u) + w(u, v)$;
    end
  end
end

Algorithm 2: Pseudocode of DIJKSTRA-algorithm

At each iteration the algorithm computes the length of the path to every unsolved node through only solved nodes. The unsolved node with the shortest path length is added to the solved set. The process terminates when a spanning tree is obtained. Since a node is added to the tree at each step the algorithm requires $V - 1$ iterations. The algorithm works because of the assumption of no negative arc lengths.
Dijkstra's algorithm computes the shortest paths to all nodes in the Graph. If only one shortest path is needed to a target node t aus V , the algorithm can stop if the target t is removed from the priority queue in line 4.
Proof of correctness:

Let dist(v) be the label found by the algorithm and let $\delta(v)$ be the shortest path distance from s to v. We want to show that $dist(v) = \delta(v)$ for every vertex v at the end of the algorithm, showing that the algorithm correctly computes the distances. We prove this by contradiction via the following theorem:

Theorem 1. *Dijkstra's Algorithm terminates with* $dist(v) = \delta(s, v) for all v \in V$.

Proof. it suffices to show that $dist(v) = \delta(s, v) for every v \in V$ when v is added to S. Suppose u is the first vertex added to S for which $dist(u) > \delta(s, u)$. Let y be the first vertx in $V - S$ along a shortest path from s to u and let x be its predecessor. Since u is the first vertex violating the claimed invariant, we have $dist(x) = \delta(s, x)$. When x was added to S, the edge xy was relaxed, which implies that $dist(y) = \delta(s, y) \leq \delta(s, u) < dist(u)$. But $dist(u) \leq dist(y)$ by our choice of u : Contradiction! □

With a binary heap, the algorithm requires $O((|E| + |V|)log|V|)$ time (which is dominated by $O(|E|log|V|)$, assuming the graph is connected), and the Fibonacci heap improves this to $O(|E| + |V|log|V|)$.

Definition 2.1. *Fibonacci heap is a heap made of a forest of trees. The amortized cost of the operations create, insert a value, decrease a value, find minimum, and merge or join (meld) two heaps, is a constant* $\Theta(1)$. *The delete operation takes O(log n).*

Proof of the time complexity (Fibonacci-heaps):

Initialization	$O(\|V\|)$
Extract-Min	$\|V\|O(log\|V\|)$
DecreaseKey	$\|E\|O(\|V\|)$

so is the time complexity of Dijkstra :$O(|E| + |V|log|V|)$

2.3 The general case: Bellman-Ford algorithm

Bellman–Ford is used primarily for graphs with negative edge weights, it is named after its developers, Richard Bellman and Lester Ford, Jr. The algorithm initializes the distance to the source vertex to 0 and all other vertices to infinity. It then does $V - 1$ passes (V is the number of vertices) over all edges relaxing, or updating, the distance to the destination of each edge. Finally it checks each edge again to detect negative weight cycles, in which case it returns false.

The main idea is that if the edge uv is the last edge of the shortest path to v, the cost of the shortest path to v is the cost of the shortest path to u plus the weight of $w(u, v)$. Thus, if we iterate through every edge uv in the graph and call relax(u, v), then we would have found the shortest path for every vertex whose shortest path consists of only one edge. If we build upon this solution and call relax(u, v) again for every edge, then we would have found the shortest path for every vertex whose shortest path consists of two edges or less. Continuing in this fashion, after the kth iteration, we would have found the shortest path for every vertex whose shortest path consist of k edges or less. Since the shortest path in a graph with V vertices and E edges has at most $V-1$ edges, we only need to repeat the process at most V-1 time.

```
Data: G=(V,E)
Result: Shortest path
Step 1: initialize graph ;
for each v ∈ V do
    if v is source then
        dist(v) := 0 ;
    end
    dist(v) := ∞ ;
    pred(v) := null;
end
Step 2: relax edges repeatedly [a] ;
for i from 1 to V-1 do
    for each edge uv ∈ E do
        if dist(u) + w(u, v) < dist(v) then
            dist(v) := dist(u) + w(u, v) ;
            pred(v) := u ;
        end
    end
end
Step 3: check for negative-weight cycles ;
for each edge uv ∈ E do
    if dist(u) + w(uv) < dist(v) then
        error "Graph contains a negative-weight cycle" ;
    end
end
```

Algorithm 3: Pseudocode of Bellman-Ford-algorithm

[a] replaces equality constraints in the original problem with an inequality constrain

Proof of correctness:

The correctness of the algorithm can be shown by induction. The precise statement shown by induction is:

Lemma 2.2. *After i repetitions of for cycle: If dist(u) is not infinity, it is equal to the length of some path from s to u; If there is a path from s to u with at most i edges, then dist(u) is at most the length of the shortest path from s to u with at most i edges.*

Proof. For the base case of induction, consider i=0 and the moment before for cycle is executed for the first time. Then, for the source vertex, $dist(s) = 0$, which is correct. For other vertices u, $dist(u) = infinity$, which is also correct because there is no path from source to u with 0 edges.

For the inductive case, we first prove the first part. Consider a moment when a vertex's distance is updated by $dist(v) := dist(u) + w(uv)$. By inductive assumption, $dist(u)$ is the length of some path from source to u. Then $dist(u) + w(u, v)$ is the length of the path from source to v that follows the path from source to u and then goes to v. For the second part, consider the shortest path from source to u with at most i edges. Let v be the last vertex before u on this path. Then, the part of the path from source to v is the shortest path from source to v with at most $i - 1$ edges. By inductive assumption, $dist(v)$ after $i - 1$ cycles is at most the length of this path. Therefore, $w(uv) + dist(v)$ is at most the length of the path from s to u. In the ith cycle, $dist(u)$ gets compared with $w(uv) + dist(v)$, and is set equal to it if $w(uv) + dist(v)$ was smaller. Therefore, after i cycles, $dist(u)$ is at most the length of the shortest path from source to u that uses at most i edges. If there are no negative-weight cycles, then every shortest path visits each vertex at most once, so at step 3 no further improvements can be made. Conversely, suppose no improvement can be made.

Then for any cycle with vertices $v_0, ..., v_{k-1}$, $dist(v_i) <= dist(v_{(i-1)modk}) + w(v_{(i-1)modk}, v_i)$ Summing around the cycle, the $dis(v_i)$ terms and the $dist(v_{i-1(modk)})$ terms cancel, leaving $0 <= sumfrom1tokofw(v_{i-1(modk)}, v_i)$, i.e., every cycle has no negative weight.

□

Bellman–Ford runs in $O(|V|.|E|)$ time (at least quadratic), where $|V|$ and $|E|$ are the number of vertices and edges respectively.

3 All Pairs Shortest Paths

Input: A connected graph G=(V,E) and an edge weight function w.
Output: For all pairs $u, v \in V$ of nodes a shortest path from u to v.

3.1 Definition and naive implementation

The all-pairs shortest path problem can be considered the mother of all routing problems. It aims to compute the shortest path from each vertex v to every other u. Using standard single-source algorithms, you can expect to get a naive implementation of $O(VE + V^2logV)$ if you use Dijkstra for example – i.e. running a $O(E + VlogV)$ process $|V|$ times. Likewise, if you use the Bellman–Ford algorithm on a dense graph, it will take about $O(V^2E)$, but handle negative edge-weights too. Storing all the paths explicitly can be very memory expensive indeed, as you need one spanning tree for each vertex. This is often impractical in terms of memory consumption, so these are usually considered as all-pairs shortest distance problems, which aim to find just the distance from each to each node to another. The result of this operation is an $n \times n$ matrix, which stores estimated distances to the each node. This has many problems when the matrix gets too big, as the algorithm will scale very poorly.

3.2 Floyd Warshall

Floyd-Warshall is an algorithm to solve the all pairs shortest path problem in a weighted, directed graph by multiplying an adjacency-matrix representation of the graph multiple times.It takes the dynamic programming approach: This essentially means that independent subproblems are solved and the results are stored for later use. The edges may have negative weights, but no negative weight cycles.

FLOYD-WARSHALL(G,V)

Data: G=(V,E)
Result: Shortest path
for $i = 1toV$ **do**
  for $j = 1toV$ **do**
    if *there is an edge from i to j* **then**
      $dist^{(0)}(i,j) = thelengthoftheedgefromitoj$;
    end
    $dist^{(0)}(i,j) = \infty$;
  end
end
for $k = 1toV$ **do**
  for $i = 1toV$ **do**
    for $j = 1toV$ **do**
      $dist^{(k)}(i,j) = min(dist^{(k-1)}(i,j), dist^{(k-1)}(i,k) + dist^{(k-1)}(k,j))$;
    end
  end
end

Algorithm 4: Pseudocode of Floyd-Warshall-algorithm

How do we compute $dist^{(k)}(i,j)$ assuming that we have already computed the previous matrix $DIST^{(k-1)}$?
We don't go through k at all: Then the shortest path from i to j uses only intermediate vertices $\{1,...,k-1\}$ and hence the length of the shortest path is $dist^{(k-1)}(i,j)$.
We do go through k: we can assume that we pass through k exactly once (no negative cycles). That is, we go from i to k, and then from k to j: we should take the shortest path from i to k, and the shortest path from k to j (optimality).
Each of these paths uses intermediate vertices only in $\{1,2,..,k-1\}$. The length of the path is : $dist^{(k-1)}(i,k) + dist^{(k-1)}(k,j)$.
This suggests the following recursive rule for computing $DIST^{(k)}$:
$dist^{(0)}(i,j) = w(i,j)$ and $dist^{(k)}(i,j) = min(dist^{(k-1)}(i,j), dist^{(k-1)}(i,k) + dist^{(k-1)}(k,j))$
The algorithm runs in $\Theta(V^3)$ time, is simple to code and efficient in practice.

3.3 Johnson algorithm

It's an algorithm to solve the all pairs shortest path problem in a sparse weighted, directed graph. First, it adds a new node with zero weight edges from it to all other nodes, and runs the Bellman-Ford algorithm to check for negative weight cycles and find $h(v)$, the least weight of a path from the new node to node v. Next it reweights the edges using the nodes $h(v)$ values. Finally for each node, it runs Dijkstra's algorithm and stores the computed least weight to other nodes, reweighted using the nodes $h(v)$ values, as the final weight. Johnson's algorithm consists of the following steps:

Data: G=(V,E)
Result: Shortest path
Compute G', where $V[G'] = V[G] \cup s$;
$E[G'] = E[G] \cup (s, v) : v \in V[G]$;
for *all* $v \in V[G]$ **do**
  $w(s, v) = 0$;
end
if $BELLMAN - FORD(G', w, s) = FALSE$ **then**
  print the input graph contains a negative weight cycle ;
end
for *each vertex* $v \in V[G']$ **do**
  set $h(v)$ to the value of $\delta(s, v)$ computed by the Bellman-Ford alg.;
end
for *each edge* $(u, v) \in E[G']$ **do**
  $\hat{w}(u, v) \leftarrow w(u, v) + h(u) - h(v)$;
  for *each vertex* $u \in V[G]$ **do**
    run DIJKSTRA $(G, \hat{w}, u)$ to compute $\delta'(s, v)$ for all $v \in V[G]$;
    for *eachvertexv* $\in V[G]$ **do**
      $d(u, v) \leftarrow \delta'(s, v) + h(v) - h(u)$;
    end
  end
end
return D ;

Algorithm 5: Pseudocode of Johnson-algorithm

First, a new node s is added to the graph, connected by zero-weight edges to each other node. Second, the Bellman-Ford algorithm is used, starting from the new vertex s, to find for each vertex v the least weight $h(v)$ of a path from q to v. If this step detects a negative cycle, the algorithm is terminated.
Next the edges of the original graph are reweighted using the values computed by the Bellman-Ford algorithm: an edge from $u$to v, having length $w(u, v)$, is given the new length $\hat{w} = w(u, v) + h(u) - h(v)$.
The $\hat{w}$ is non-negativ because $\delta(s, u) + w(u, v) \geq \delta(s, v)$ otherwise $s \to u \to v$ would be shorter than the shortest path $s \to v$.
Finally, s is removed, and Dijkstra's algorithm is used to find a shortest path from each node to every other vertex in the reweighted graph.
The reweighting preserves the shortest-path-weight. We have:
$\hat{w}(p) = \sum \hat{w}(v_i, v_{i+1}) = w(v_1, v_2) + \delta(s, v_1) - \delta(s, v_2) + ... + w(v_{k-1}, v_k) + \delta(s, v_{k-1}) - \delta(s, v_k)$
$= w(p) + \delta(s, v_1) - \delta(s, v_k)$ with $\delta(s, v_1) - \delta(s, v_k)$ a value that depends only on the endpoints, not the path. The time complexity of this algorithm, using Fibonacci heaps in the implementation of Dijkstra's algorithm, is $O(V^2 logV + VE)$: the algorithm uses $O(VE)$ time for the Bellman–Ford stage of the algorithm, and $O(V \log V + E)$ for each of V instantiations of Dijkstra's algorithm.
Thus, when the graph is sparse, the total time can be faster than the Floyd–Warshall algorithm, which solves the same problem in time $O(V^3)$.

References

www.wikipedia.de
Videoslectures of Massachusetts Institute of Technology