Theory of Computing

Lecture 7

MAS 714

Hartmut Klauck

The single-source shortest-path problem with negative edge weights

- Graph G, weight function W, start vertex s
- Output: a bit indicating if there is a negative cycle reachable from s
  AND (if not) the shortest paths from s to all v

Bellman-Ford Algorithm

- Initialize: $d(s)=0$, $\pi(s)=s$, $d(v)=\infty$, $\pi(v)=NIL$ for other v
- For i=1 to n-1:
  - Relax all edges (u,v)
- For all (u,v): if $d(v)>d(u)+W(u,v)$ then output: „negative cycle!“

- Remark: d(v) and $\pi(v)$ contain distance from s and predecessor in a shortest path tree

Running time

- Running time is O(nm)
  - n-1 times relax all m edges

Correctness

- Assume that no cycle of negative length is reachable from s
- **Theorem:** After n-1 iterations of the for-loop we have $d(v)=\delta(s,v)$ for all v.

- **Lemma:** Let $v_0,\ldots,v_k$ be a shortest path from $s=v_0$ to v_k. Relax edges $(v_0,v_1)\ldots.(v_{k-1},v_k)$ successively. Then $d(v_k)=\delta(s,v_k)$. This holds regardless of other relaxations performed.

Correctness

- Proof of the theorem:
  - Let v denote a reachable vertex
  - Let s, …,v be a shortest path with k edges
  - k$\leq$ n-1 can be assumed (why?)
  - In every iteration all edges are relaxed
  - By the lemma d(v) is correct after k$\leq$ n-1 iterations
- For all unreachable vertices we have d(v)=∞ at all times
- To show: the algorithm decides the existence of negative cycles correctly
- No neg. cycle present/reachable: for all edges (u,v):
  - $d(v)=\delta(s,v)\leq\delta(s,u)+W(u,v)=d(u)+W(u,v)$, pass test

Correctness

- If a negative cycle exists:
  - Let $v_0,\ldots,v_k$ be a (reachable) path with negative length and $v_0=v_k$
  - Assume the algorithm does NOT stop with error message, then
    - $d(v_i) \leq d(v_{i-1})+W(v_{i-1},v_i)$ for all i=1...k
    - Hence

$$\sum_{i=1}^{k} d(v_i) \leq \sum_{i=1}^{k} \left(d(v_{i-1}) + W(v_{i-1}, v_i) \right)$$

Correctness

$$\sum_{i=1}^{k} d(v_i) \le \sum_{i=1}^{k} d(v_{i-1}) + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- $v_0 = v_k$, so

$$\sum_{i=1}^{k} d(v_i) = \sum_{i=1}^{k} d(v_{i-1})$$

- $d(v_i) < \infty$ in the end for all reachable vertices, hence

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$$

The Lemma

Lemma: Let $v_0,\ldots,v_k$ be a shortest path from $s=v_0$ to v_k. Relax edges $(v_0,v_1)\ldots.(v_{k-1},v_k)$ successively. Then $d(v_k)=\delta(s,v_k)$. This holds regardless of other relaxations performed.

Proof:

By induction. After relaxing (v_{i-1}, v_i) the value $d(v_i)$ is correct.

Base: $i=0$, $d(v_0)=d(s)=0$ is correct.

Assume $d(v_{i-1})$ correct. According to an earlier observation after relaxing (v_{i-1},v_i) also $d(v_i)$ correct.

Once $d(v)$ is correct, the value stays correct.

d(v) is always an upper bound

Application of Bellman Ford

- Graph is a distributed network
  - vertices are processors that can communicate via edges
- We look for distance/shortest path of vertices from s
- Computation can be performed in a distributed way, without
  - global control
  - global knowledge about the network
- Dijkstra needs global knowledge
- Running time: n-1 phases, vertices compute (in parallel)

All-pairs shortest path

- Given a graph
  - Variants:
    - directed/undirected
    - weighted/unweighted/pos./neg. weights
- Output: For all pairs of vertices u,v:
  - Distance in G (APD: All-pairs distances)
  - Shortest Paths (APSP: All-pairs shortest-path)

APSP

- APD: n^2 outputs, running time at least n^2
- Can just use adjacency matrix
- APSP: problem: how to represent n^2 paths?
  - Easy to construct a graph, such that for $\Omega(n^2)$ vertex pairs the distance is $\Omega(n)$
  - Simply writing paths requires output length n^3

APSP output convention

- Implicit representation of shortest paths as a *successor matrix*
- Successor matrix S is n by n, S[i,j]=k for the neighbor k of i, which is first on the shortest path from i to j
- Easy to compute the shortest path from i to j using S:
  - e.g. S[i,j]=k, S[k,j]=l, S[l,j]=a, S[a,j]=j

APSP: some observations

- Edge weights ≥ 0: use n times Dijkstra, running time: $O(nm+n^2 \log n)$
  - Unweighted graphs: n times BFS for time $O(nm+n^2)$
- For dense graphs $m=\Omega(n^2)$ and we get $O(n^3)$
- Can we save work?

Floyd-Warshall Algorithm

- Input: G, directed graph with positive and negative weights, no negative cycles
- $O(n^3)$ algorithm based on *Dynamic Programming*
- Compute shortest paths (from u to v) that use only vertices 1… k

Floyd-Warshall Algorithm

- Definition:
  - d[u,v,k]= length of the shortest path from u to v that (besides u,v) uses vertices {1,…,k} only
- d[u,v,0]=W(u,v)
  - is $=\infty$ if (u,v) is no edge
- Recursion:
  - d[u,v,k]= minimum of
    - d[u,v,k-1] paths using only 1,…,k-1
    - d[u,k,k-1] + d[k,v,k-1] paths also using k

Floyd-Warshall Algorithm

- Initialize d[u,v,0]=W(u,v) for all u,v
- For k=1,…,n
  – compute d[u,v,k] for all u,v
- Total running time: $O(n^3)$

Floyd-Warshall Algorithm

- Computing the paths: exercise
- Note that this algorithm is very simple, no fancy datastructures, so constant factors are small

Dynamic Programming

- The values d[u,v,0] are given immediately
- The values d[u,v,n] are the solution to the problem
- We can easily compute all d[u,v,k] once we know all d[u,v,k-1]
- This process of computing solutions bottom up is called *dynamic programming*
- Note the difference to computing top down by recursion!

Dynamic Programming

- There is a recursive solution
  - E.g. d[u,v,k]=min{d[u,v,k-1],d[u,k,k-1]+d[k,v,k-1]
- The total number of different sub-problems is bounded
  - only n^3 sub-problems d[u,v,k]
- Sub-problems have a parameter (e.g. k)
- So we compute all of them “bottom up”
- Compare this to recursion top down

Dynamic Programming

- Top down solution:
  - To compute d(u,v,n) we get T(n)=3T(n-1)+O(1)
    - Exponential time!
  - Recursion solves the same sub-problems over and over
- Dynamic programming solves each sub-problem once, and stores the solution

Example Dynamic Programming

- Fibonacci numbers:
  - F(0)=1, F(1)=1, F(n)=F(n-1)+F(n-2)
- Recursive algorithm:
  - Compute recursively like the definition
  - This needs time F(n) to compute F(n)
  - F(n) grows like 1.618^n
- Dynamic programming solution:
  - F=0, G=1, For i=2…n: {H=F+G, F=G, G=H}
  - Time: O(n) additions

Another example

- Longest Common Subsequence (LCS)
- A sequence $z_1,\ldots,z_k$ (over some alphabet) is a *subsequence* of $x=x_1,\ldots,x_m$, if there are $i_1<i_2<\cdots< i_k$ and all $x(i_j)=z_j$
- Input: sequences $x=x_1,\ldots,x_m$ and $y_1,\ldots.,y_n$
- Output: a longest sequence Z that is a subsequence of both X,Y

LCS

- Brute force approach: enumerate all subsequences (2^m)
- Dynamic Programming idea:
- **Theorem:** Let $x=x_1,\ldots,x_m$ and $y=y_1,\ldots,y_n$, and $z=z_1,\ldots,z_k$ be an LCS for x,y
  1. If $x_m=y_n$: $z_k = x_m = y_n$ and $z_1 \ldots z_{k-1}$ is LCS of $x_1,\ldots,x_{m-1}$ and $y_1,\ldots,y_{n-1}$
  2. If $x_m \neq y_n$ and $z_k \neq x_m$ then z is an LCS of $x_1,\ldots,x_{m-1}$ and y
  3. If $x_m \neq y_n$ and $z_k \neq y_n$ then z is an LCS of $y_1,\ldots,y_{n-1}$ and x

The recursion

- Denote $x(i)=x_1,\ldots,x_i$
- c[i,j] is the LCS length of x(i) and y(j)
- Recursion:
  - c[0,j]=0 and c[i,0]=0
  - c[i,j]=c[i-1,j-1]+1 if $x_i=y_j$ and i,j>0
  - max{ c[i,j-1] , c[i-1,j] } otherwise

- There are only mn subproblems c[i,j] and we can compute them starting from c[0,0], row by row
  - i,j viewed as indices in a matrix

LCS: the length

- LCSLength(X[1..m], Y[1..n])
  - **for** i=0...m
    - C[i,0] = 0
  - **for** j=0...n
    - C[0,j] = 0
  - **for** i=1...m
    - **for** j=1...n
      - **if** X[i] = Y[j] then C[i,j] := C[i-1,j-1] + 1
        else C[i,j] := max(C[i,j-1], C[i-1,j])

LCS: the sequence

- Create an array B of arrows during the computation
  - X[i]=Y[j]: left and up ↖
  - X[i]≠Y[j]:
    - C[i,j] = C[i,j-1] : left ←
    - C[i,j] =C[i-1,j] : up ↑
- Follow the arrows starting at B[m,n]
  - ↖ arrows are at elements of the LCS

LCS: example

	Ø	B	A	B	A	C
Ø	0	0	0	0	0	0
C	0	0	0	0	0	1
A	0	0	1	1	1	1
B	0	1	1	2	2	2
C	0	1	1	2	2	3

Algorithm design paradigms

- Divide and Conquer
- Dynamic Programming
- Greedy
- More:
  - Randomization
  - Recursion
  - Branch and Bound
  - etc.

APSP and APD faster?

- It seems that we are still doing a lot of work twice at running times like n^3 or nm for APSP
- Consider the adjacency matrix A of graph G
- For now settle for connectivity information: is v reachable from u?

- Consider A^2, with the standard matrix product
  - $A^2[u,v] > 0$ iff there is a path of length 2 from u to v

Connectivity by Matrix Multiplication

- Set A[u,u]=1
- Now: $A^t[u,v] > 0$ iff there is a path of length at most t from u to v
- Compute A^{n-1}

- Naive approach:
  n-1 matrix multiplications

Connectivity by Matrix Multiplication

- Assume $2^{k-1} \leq n \leq 2^k$
- Compute 2^k–th power of A
- Repeated squaring
  - Compute A, A^2, A^4, A^8, A^{16} etc.
  - Finish after k multiplications
  - $k \leq \log n+1$
- Best algorithm for matrix multiplication needs time $O(n^\gamma)$. It is known that $2 \leq \gamma \leq 2.3729$
- Running time is $O(n^\gamma \log n)$

Connectivity by Matrix Multiplication

- Problem: we can decide connectivity for all pairs, but have not solved APD or APSP!

Some results:

1. Can solve APD in time $O(n^{\gamma} \log n)$ for unweighted undirected graphs
2. APSP in time $O(n^{\gamma} \log^2 n)$ for unweighted undirected graphs via a *randomized* algorithm
3. APSP for directed graphs with polynomial size nonnegative weights:
   Approximation ratio $(1+\varepsilon)$ in time $O(n^{\gamma}/\varepsilon \ \log^3 n)$
4. APSP for weighted undirected graphs:
   Approximation ratio 3 in time $O(n^2)$