

Shortest paths

Algorithms and Networks 2010/2011



Contents

- The shortest path problem:
 - Statement
 - Versions
- Applications
- Algorithms (for *single source* sp problem)
 - Reminders: relaxation, Dijkstra, Variants of Dijkstra, Bellman-Ford, Johnson ...
 - Scaling technique (Gabow's algorithm)
- Variant algorithms: A*, bidirectional search
- Bottleneck shortest paths



Notation

- In the entire course:
 - $n = |V|$, the number of vertices
 - $m = |E|$ or $m = |A|$, the number of edges or the number of arcs



1

Definition and Applications



Shortest path problem

- (Directed) graph $G = (V, E)$, **length** for each edge e in E , $w(e)$
- **Distance** from u to v : length of shortest path from u to v
- Shortest path problem: find distances, find shortest paths ...
- Versions:
 - All pairs
 - Single pair
 - Single source
 - Single destination
 - Lengths can be
 - All equal (unit lengths) (BFS)
 - Non-negative
 - Negative but no negative cycles
 - Negative cycles possible



Applications

- Subroutine in other graph algorithms
- Route planning
- Difference constraints
- Allocating Inspection Effort on a Production Line

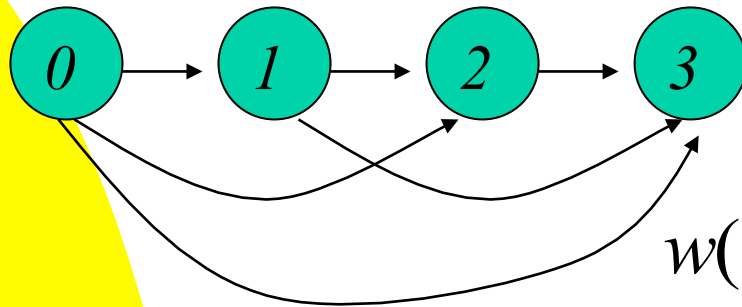


Allocating Inspection Efforts on a Production Line

- **Production line**: ordered sequence of n production stages.
- Items are produced in **batches** of $B > 0$ items.
- **Probability** that stage i produces a **defect** item is a_i .
- **Manufacturing cost** at stage i : p_i .
- **Cost of inspecting** at stage j , when last inspection has been done at stage i :
 - f_{ij} per batch, plus
 - g_{ij} per item in the batch
- **When should we inspect to minimize total costs?**



Solve by modeling as shortest paths problem



$$w(i, j) = f_{ij} + B(i)g_{ij} + B(i) \sum_{k=i+1}^j p_k$$

Where $B(i)$ denotes the expected number of non-defective items after stage i

$$B(i) = B \prod_{k=1}^i (1 - a_k)$$



Application: Difference constraints

- Tasks with precedence constraints and running length
- Each task i has
 - Time to complete $b_i > 0$
- Some tasks can be started after other tasks have been completed:
 - Constraint: $s_j + b_j \leq s_i$
- First task can start at time 0. When can we finish last task?
- Shortest paths problem on directed acyclic graphs!



Modeling as longest paths problem

- Take one vertex v_0 (models time 0)
- Vertex for each task
- Arc (v_0, i) for each task vertex i
- For each precedence constraint $s_j + b_j \leq s_i$ an arc (j, i) with length b_j
- Optimal: start each task i at time equal to length of longest path from v_0 to i .



Difference constraints as shortest paths

- The longest path problem can be solved in $O(n+m)$ time, as we have a directed acyclic graph.
- Transforming to shortest paths problem: multiply all lengths and times by -1 .



2

Algorithms for shortest path problems (reminders)



Basis of single source algorithms

- Source s .
- Each vertex v has variable $D[v]$
 - Invariant: $d(s,v) \leq D[v]$ for all v
 - Initially: $D[s]=0$; $v \neq s$: $D[v] = \infty$
- Relaxation step over edge (u,v) :
 - $D[v] = \min \{ D[v], D[u] + w(u,v) \}$



Maintaining shortest paths

- Each vertex maintains a pointer to the 'previous vertex on the current shortest path' (sometimes NIL): $p(v)$
- Initially: $p(v) = \text{NIL}$ for each v
- Relaxation step becomes:

p-values build paths of length $D(v)$
Shortest paths tree

Relax (u, v, w)

If $D[v] > D[u] + w(u, v)$

then $D[v] = D[u] + w(u, v)$; $p(v) = u$



Dijkstra

- Initialize
- Take priority queue Q , initially containing all vertices
- While Q is not empty,
 - Select vertex v from Q of minimum value $D[v]$
 - Relax across all outgoing edges from v
 - *Note: each relaxation can cause a change of a D -value and thus a change in the priority queue*
 - *This happens at most $|E|$ times*



On Dijkstra

- Assumes all lengths are non-negative
- Correctness proof (done in 'Algoritmiek')
- Running time:
 - Depends on implementation of priority queue
 - $O(n^2)$: standard queue
 - $O(m + n \log n)$: Fibonacci heaps
 - $O((m + n) \log n)$: red-black trees, heaps
 - ...



Negative lengths

- What if $w(u,v) < 0$?
- Negative cycles, reachable from s ...
- Bellman-Ford algorithm:
 - For instances without negative cycles:
 - In $O(nm)$ time: SSSP problem when no negative cycles reachable from s
 - Also: detects negative cycle



Bellman-Ford

*Clearly:
 $O(nm)$ time*

- Initialize
- Repeat $|V|-1$ times:
 - For every edge (u,v) in E do: Relax(u,v,w)
- For every edge (u,v) in E do
 - If $D[v] > D[u] + w(u,v)$
 - then There exists a negative circuit! Stop
- There is no negative circuit, and for all vertices v : $D[v] = d(s,v)$.



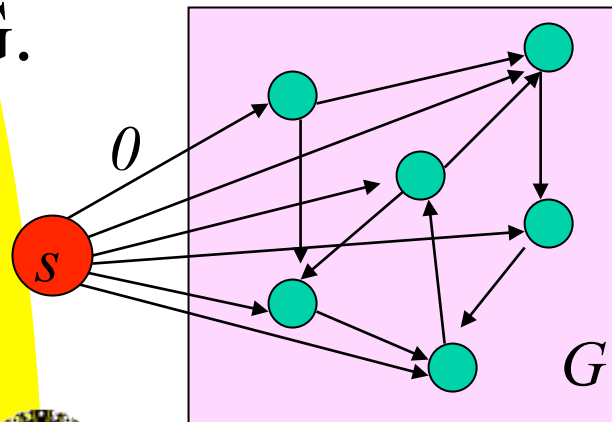
Correctness of Bellman-Ford

- Invariant: if no negative cycle is reachable from s , then after i runs of main loop, we have:
 - If there is a shortest path from s to u with at most i edges, then $D[u]=d[s,u]$, for all u .
- If no negative cycle reachable from s , then every vertex has a shortest path with at most $n - 1$ edges.
- If a negative cycle reachable from s , then there will always be an edge with a relaxation possible.



Finding a negative cycle in a graph

- Reachable from s :
 - Apply Bellman-Ford, and look back with pointers
- Or: add a vertex s with edges to each vertex in G .



All pairs

- Dynamic programming: $O(n^3)$ (Floyd, 1962)
- Johnson: improvement for sparse graphs with **reweighting technique**:
 - $O(n^2 \log n + nm)$ time.
 - Works if no negative cycles
 - Observation: if all weights are non-negative we can run Dijkstra with each vertex as starting vertex: that gives $O(n^2 \log n + nm)$ time.
 - What if we have negative lengths: reweighting...



Reweighting

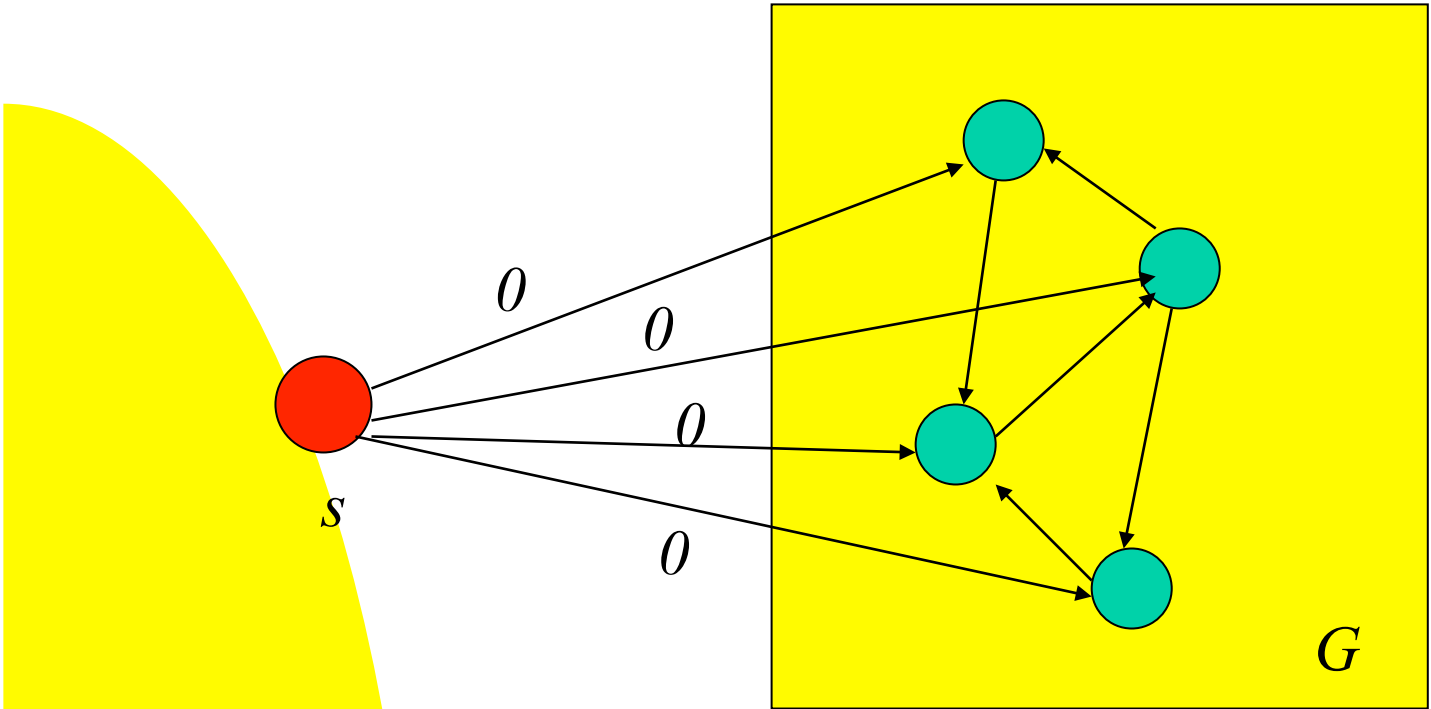
- Let $h: V \rightarrow \mathbb{R}$ be any function to the reals.
- Write $w_h(u, v) = w(u, v) + h(u) - h(v)$.
- Lemmas: *Potentials*
 - Let P be a path from x to y . Then $w_h(P) = w(P) + h(x) - h(y)$.
 - P is a shortest path from x to y with lengths w , if and only if it is so with lengths w_h .
 - G has a negative-length circuit with lengths w , if and only if it has a negative-length circuit with lengths w_h .



What height function h is good?

- Look for height function h with
 - $w_h(u, v) \geq 0$, for all edges (u, v) .
- If so, we can:
 - Compute $w_h(u, v)$ for all edges.
 - Run Dijkstra but now with $w_h(u, v)$.
- Special method to make h with a SSSP problem, and Bellman-Ford.





Choosing h

- Set $h(v) = d(s, v)$ (*in new graph*)
- Solving SSSP problem with negative edge lengths; use Bellman-Ford.
- If negative cycle detected: stop.
- Note: for all edges (u, v) : $w_h(u, v) = w(u, v) + h(u) - h(v) = w(u, v) + d(s, u) - d(s, v) \geq 0$



Johnson's algorithm

- Build graph G' (as shown)
- Compute with Bellman-Ford $d(s, v)$ for all v
- Set $w_h(u, v) = w(u, v) - d(s, u) + d(s, v)$ for all edges (u, v) .
- For all u do:
 - Use Dijkstra's algorithm to compute $d_h(u, v)$ for all v .
 - Set $d(u, v) = d_h(u, v) + h(v) - h(u)$.

$O(n^2 \log n + nm)$ time



3

Shortest path algorithms “using the numbers” and scaling



Using the numbers

- Back to the single source shortest paths problem with non-negative distances
 - Suppose Δ is an upper bound on the maximum distance from s to a vertex v .
 - Let L be the **largest length** of an edge.
 - Single source shortest path problem is solvable in $O(m + \Delta)$ time.



In $O(m+\Delta)$ time

- Keep array of doubly linked lists: $L[0], \dots, L[\Delta]$,
- Maintain that for v with $D[v] \leq \Delta$,
 - v in $L[D[v]]$.
- Keep a *current minimum* μ .
 - Invariant: all $L[k]$ with $k \leq \mu$ are empty
- **Changing $D[v]$ from x to y** : take v from $L[x]$ (with pointer), and add it to $L[y]$: $O(1)$ time each.
- **Extract min**: while $L[\mu]$ empty, $\mu++$; then take the first element from list $L[\mu]$.
- Total time: $O(m+\Delta)$



Corollary and extension

- SSSP: in $O(m+nL)$ time. (Take $\Delta=nL$).
- Gabow (1985): SSSP problem can be solved in $O(m \log_R L)$ time, where
 - $R = \max\{2, m/n\}$
 - L : maximum length of edge
- Gabow's algorithm uses **scaling** technique!



Gabow's algorithm

Main idea

- First, build a scaled instance:
 - For each edge e set $w'(e) = \lfloor w(e) / R \rfloor$.
- Recursively, solve the scaled instance.
- Another shortest paths instance can be used to compute the correction terms!



How far are we off?

- We want $d(s, v)$
- $R * d_w(s, v)$ is when we scale back our scaled instance: what error did we make when rounding?
- Set for each edge (x, y) in E :
 - $Z(x, y) = w(x, y) - R * d_w(s, x) + R * d_w(s, y)$
 - Works like height function, so the same shortest paths!
 - Height of x is $-R * d_w(s, x)$



A claim

- For all vertices v in V :
 - $d(s, v) = d_Z(s, v) + R * d_{w'}(s, v)$
- As with height functions (telescope):
 - $d(s, v) = d_Z(s, v) + h(s) - h(v) = d_Z(s, v) - R * d_{w'}(s, s) + R * d_{w'}(s, v)$
 - And $d_{w'}(s, s) = 0$
- Thus, we can compute distances for w by computing distances for Z and for w'

Gabow's algorithm

- Base case: use $O(m+nR)$ algorithm if $L \leq R$.
Otherwise:
- For each edge e : set $w'(e) = \lfloor w(e) / R \rfloor$.
- Recursively, compute the distances *but with the new length function w'* . Set for each edge (u,v) :
 - $Z(u,v) = w(u,v) + R * d_{w'}(s,u) - R * d_{w'}(s,v)$.
- Compute $d_Z(s,v)$ for all v and then use
 - $d(s,v) = d_Z(s,v) + R * d_{w'}(s,v)$



A Property of Z

- For each edge $(u, v) \in E$ we have:
 - $Z(u, v) = w(u, v) + R * d_{w'}(s, u) - R * d_{w'}(s, v) \geq 0$,
because
 - $w(u, v) \geq R * w'(u, v) \geq R * (d_{w'}(s, v) - d_{w'}(s, u))$.
- So, a variant of Dijkstra can be used to compute distances for Z.



Computing distances for Z

- For each vertex v we have
 - $d_Z(s, v) \leq nR$ for all v reachable from s
 - Consider a shortest path P for distance function w' from s to v
 - For each of the less than n edges e on P , $w(e) \leq R + R * w'(e)$
 - So, $d(s, v) \leq w(P) \leq nR + R * w'(P) = nR + R * d_{w'}(s, v)$
 - Use that $d(s, v) = d_Z(s, v) + R * d_{w'}(s, v)$
- So, we can use $O(m + nR)$ algorithm (Dijkstra with doubly-linked lists) to compute all values $d_Z(v)$.

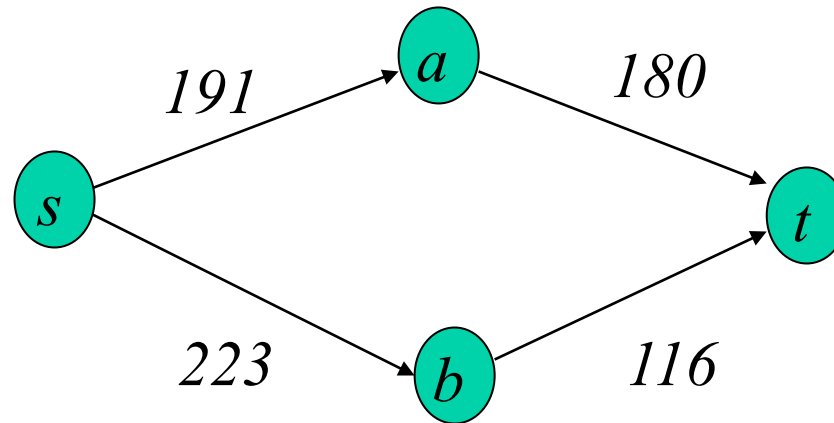


Gabow's algorithm (analysis)

- Recursive step: costs $O(m \log_R L')$ with $L' = \lfloor L/R \rfloor$.
- SSSP for Z costs $O(m + nR) = O(m)$.
- Note: $\log_R L' \leq (\log_R L) - 1$.
- So, Gabow's algorithm uses $O(m \log_R L)$ time.



Example



4

Variants: A^* and bidirectional search



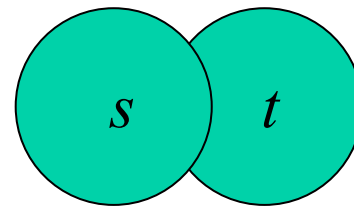
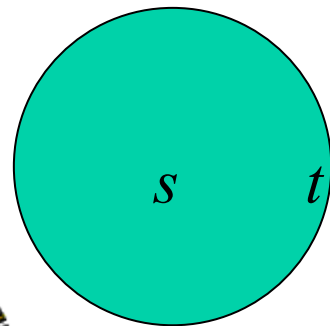
A* (simplified exposition)

- Practical importance: A* algorithm
- Can be explained in terms of height functions
- Consider a route planning problem with geographic data, and distances of arcs at least Euclidian distance of vertices
- Suppose we have a single pair shortest path problem, with source s and target t
- Use height function - $h(v) =$ - the Euclidian distance from v to t (notate: $|vt|$)
- For all arcs $(v, w) \in E$:
 - $w'(v, w) = w(v, w) - |vt| + |wt| \geq |vw| - |vt| + |wt| \geq 0$
- Note: arcs towards target get smaller length and arcs away from target larger length
- Algorithm is faster in practice but still correct



Bidirectional search

- For a single pair shortest path problem:
- Start a Dijkstra-search from both sides simultaneously
- Analysis needed for stopping criterion
- Faster in practice
- Combines nicely with A^*



5

Bottleneck shortest paths



Bottleneck shortest path

- Given: weighted graph G , weight $w(e)$ for each arc, vertices s, t .
- Problem: find a path from s to t such that the **maximum weight** of an arc on the path is as small as possible.
 - Or, reverse: such that the minimum weight is as large as possible: *maximum capacity path*



Algorithms

- On directed graphs: $O((m+n) \log m)$,
 - Or: $O((m+n) \log L)$ with L the maximum absolute value of the weights
 - Binary search and DFS
- On undirected graphs: $O(m+n)$ with divide and conquer strategy



Bottleneck shortest paths on undirected graphs

- Find the median weight of all weights of edges, say r .
- Look to graph G^r formed by edges with weight at most r .
- If s and t in same connected component of G^r , then the bottleneck is at most r : now remove all edges with weight more than r , and repeat (recursion).
- If s and t in different connected components of G^r : the bottleneck is larger than r . Now, contract all edges with weight at most r , and recurse.
- $T(m) = O(m) + T(m/2)$



5

Conclusions



Conclusions

- Applications
- Several algorithms for shortest paths
 - Variants of the problem
 - Detection of negative cycles
 - Reweighting technique
 - Scaling technique
- A*, bidirectional
- Bottleneck shortest paths

