Algorithms & Models of Computation

CS/ECE 374, Fall 2017

Dynamic Programming: Shortest Paths and DFA to Reg Expressions

Lecture 18

Thursday, November 2, 2017

Part I

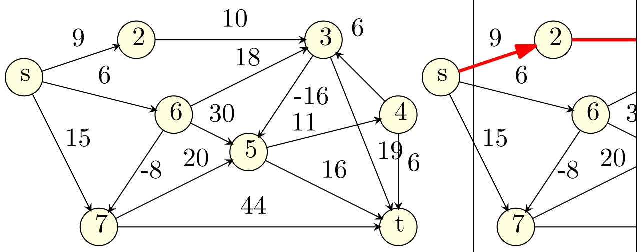
Shortest Paths with Negative Length Edges

Single-Source Shortest Paths with Negative Edge Lengths

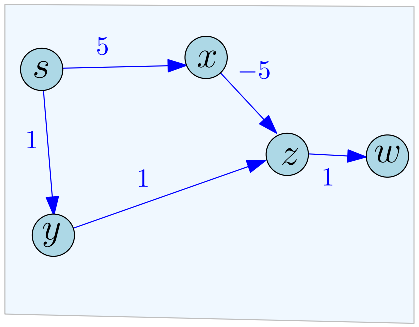
Single-Source Shortest Path Problems

Input: A *directed* graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.



What are the distances computed by Dijkstra's algorithm?



The distance as computed by Dijkstra algorithm starting from s:

(A) $s = 0$, $x = 5$, $y = 1$, $z = 0$.

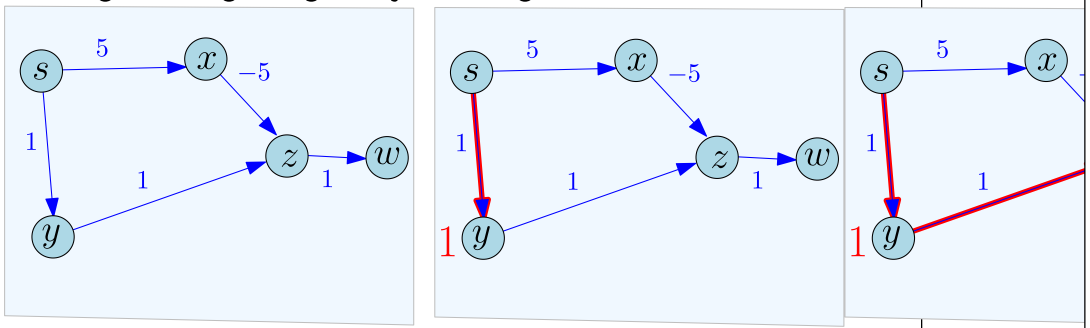
(B) $s = 0$, $x = 1$, $y = 2$, $z = 5$.

(C) $s = 0$, $x = 5$, $y = 1$, $z = 2$.

(D) IDK.

Dijkstra's Algorithm and Negative Lengths

With negative length edges, Dijkstra's algorithm can fail

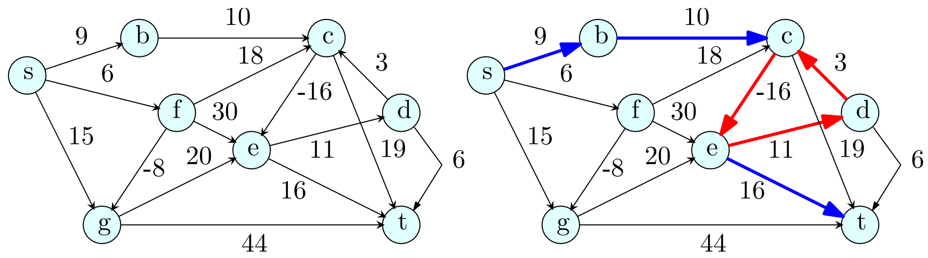


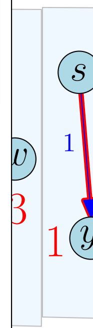
False assumption: Dijkstra's algorithm is based on the assumption that if $s = v_0 \to v_1 \to v_2 \ldots \to v_k$ is a shortest path from s to v_k then $dist(s, v_i) \leq dist(s, v_{i+1})$ for $0 \leq i < k$. Holds true only for non-negative edge lengths.

Negative Length Cycles

Definition

A cycle C is a negative length cycle if the sum of the edge lengths of C is negative.





Shortest Paths and Negative Cycles

Given $G = (V, E)$ with edge lengths and s, t. Suppose

1. G has a negative length cycle C, and
2. s can reach C and C can reach t.

Question: What is the shortest ***distance*** from s to t?
Possible answers: Define shortest distance to be:

1. undefined, that is $-\infty$, OR
2. the length of a shortest ***simple*** path from s to t.

Lemma

If there is an efficient algorithm to find a shortest simple $s \to t$ path in a graph with negative edge lengths, then there is an efficient algorithm to find the longest simple $s \to t$ path in a graph with positive edge lengths.

Finding the $s \to t$ longest path is difficult. **NP-Hard**!

Alternatively: Finding Shortest Walks

Given a graph $G = (V, E)$:

1. A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$.
2. A walk is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. Vertices are allowed to repeat.

Define $dist(u, v)$ to be the length of a shortest walk from u to v.

1. If there is a walk from u to v that contains negative length cycle then $dist(u, v) = -\infty$
2. Else there is a path with at most $n - 1$ edges whose length is equal to the length of a shortest walk and $dist(u, v)$ is finite

Helpful to think about walks

Shortest Paths with Negative Edge Lengths

Problems

Algorithmic Problems

Input: A directed graph $G = (V, E)$ with edge lengths (could be negative). For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

Questions:

1. Given nodes s, t, either find a negative length cycle C that s can reach or find a shortest path from s to t.
2. Given node s, either find a negative length cycle C that s can reach or find shortest path distances from s to all reachable nodes.
3. Check if G has a negative length cycle or not.

Shortest Paths with Negative Edge Lengths

In Undirected Graphs

Note: With negative lengths, shortest path problems and negative cycle detection in undirected graphs cannot be reduced to directed graphs by bi-directing each undirected edge. Why?

Problem can be solved efficiently in undirected graphs but algorithms are different and more involved than those for directed graphs. Beyond the scope of this class. If interested, ask instructor for references.

Why Negative Lengths?

Several Applications

1. Shortest path problems useful in modeling many situations — in some negative lengths are natural
2. Negative length cycle can be used to find arbitrage opportunities in currency trading
3. Important sub-routine in algorithms for more general problem: minimum-cost flow

Negative cycles

Application to Currency Trading

Currency Trading

Input: n currencies and for each ordered pair (a, b) the *exchange rate* for converting one unit of a into one unit of b.

Questions:

1. Is there an arbitrage opportunity?
2. Given currencies s, t what is the best way to convert s to t (perhaps via other intermediate currencies)?

Concrete example:

1. 1 Chinese Yuan $= \mathbf{0.1116}$ Euro
2. 1 Euro $= \mathbf{1.3617}$ US dollar
3. 1 US Dollar $= \mathbf{7.1}$ Chinese Yuan.

Thus, if exchanging $1\ \$ \rightarrow$ Yuan $\rightarrow$ Euro $\rightarrow \$$, we get: $\mathbf{0.1116 * 1.3617 * 7.1 = 1.07896\$}$.

Reducing Currency Trading to Shortest Paths

Observation: If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.

Create currency trading *directed* graph $G = (V, E)$:

1. For each currency i there is a node $v_i \in V$
2. $E = V \times V$: an edge for each pair of currencies
3. edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ can be negative

Exercise: Verify that

1. There is an arbitrage opportunity if and only if G has a negative length cycle.
2. The best way to convert currency i to currency j is via a shortest path in G from i to j. If d is the distance from i to j then one unit of i can be converted into 2^d units of j.

Reducing Currency Trading to Shortest Paths

Math recall - relevant information

1. $\log(\alpha_1 * \alpha_2 * \cdots * \alpha_k) = \log \alpha_1 + \log \alpha_2 + \cdots + \log \alpha_k$.
2. $\log x > 0$ if and only if $x > 1$.

Shortest Paths with Negative Lengths

Lemma

Let G be a directed graph with arbitrary edge lengths. If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ is a shortest path from s to v_k then for $1 \le i < k$:

1. *$s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ is a shortest path from s to v_i*
2. *False: $dist(s, v_i) \le dist(s, v_k)$ for $1 \le i < k$. Holds true only for non-negative edge lengths.*

Cannot explore nodes in increasing order of distance! We need other strategies.

Shortest Paths and Recursion

1. Compute the shortest path distance from s to t recursively?
2. What are the smaller sub-problems?

Lemma

Let G be a directed graph with arbitrary edge lengths. If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ is a shortest path from s to v_k then for $1 \le i < k$:

1. *$s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ is a shortest path from s to v_i*

Sub-problem idea: paths of fewer hops/edges

Hop-based Recursion: Bellman-Ford Algorithm

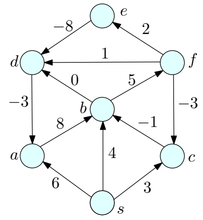
Single-source problem: fix source s.
Assume that all nodes can be reached by s in G
Assume G has no negative-length cycle (for now).

$d(v,k)$: shortest walk length from s to v using at most k edges.
Note: $dist(s,v) = d(v, n-1)$. Recursion for $d(v,k)$:

$$d(v,k) = \min \begin{cases} \min_{u \in V}(d(u,k-1) + \ell(u,v)). \\ d(v,k-1) \end{cases}$$

Base case: $d(s,0) = 0$ and $d(v,0) = \infty$ for all $v \neq s$.

Example



Bellman-Ford Algorithm

for each $u \in V$ **do**
  $d(u,0) \leftarrow \infty$
$d(s,0) \leftarrow 0$

for $k = 1$ to $n-1$ **do**
  for each $v \in V$ **do**
    $d(v,k) \leftarrow d(v,k-1)$
    for each edge $(u,v) \in in(v)$ **do**
      $d(v,k) = \min\{d(v,k), d(u,k-1) + \ell(u,v)\}$

for each $v \in V$ **do**
  $\mathrm{dist}(s,v) \leftarrow d(v,n-1)$

Running time: $O(mn)$ Space: $O(m+n^2)$
Space can be reduced to $O(m+n)$.

Bellman-Ford Algorithm

for each $u \in V$ **do**
  $d(u) \leftarrow \infty$
$d(s) \leftarrow 0$

for $k = 1$ to $n-1$ **do**
  for each $v \in V$ **do**
    for each edge $(u,v) \in in(v)$ **do**
      $d(v) = \min\{d(v), d(u) + \ell(u,v)\}$

for each $v \in V$ **do**
  $\mathrm{dist}(s,v) \leftarrow d(v)$

Running time: $O(mn)$ Space: $O(m+n)$
Exercise: Argue that this achieves same results as algorithm on previous slide.

Bellman-Ford: Negative Cycle Detection

Check if distances change in iteration n.

```
for each u ∈ V do
    d(u) ← ∞
d(s) ← 0

for k = 1 to n − 1 do
        for each v ∈ V do
            for each edge (u, v) ∈ in(v) do
                d(v) = min{d(v), d(u) + ℓ(u, v)}
(* One more iteration to check if distances change *)
for each v ∈ V do
    for each edge (u, v) ∈ in(v) do
        if (d(v) > d(u) + ℓ(u, v))
            Output ``Negative Cycle''

for each v ∈ V do
        dist(s, v) ← d(v)
```


Correctness of the Bellman-Ford Algorithm

Via induction: For each v, $d(v, k)$ is the length of a shortest walk from s to v with *at most* k hops.

Lemma

Suppose G does not have a negative length cycle reachable from s. Then for all v, $\mathrm{dist}(s, v) = d(v, n-1)$. Moreover, $d(v, n-1) = d(v, n)$.

Proof.

Exercise. □

Corollary

Bellman-Ford correctly outputs the shortest path distances if G has no negative length cycle reachable from s.

Correctness: Detecting negative length cycle

Lemma

G has a negative length cycle reachable from s if and only if there is some node v such that $d(v, n) < d(v, n-1)$.

Lemma proves correctness of negative cycle detection by Bellman-Ford algorithm.

The only if direction follows from Lemma on previous slide. We prove the if direction in the next slide.

Correctness: detecting negative length cycle

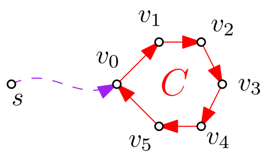
Lemma

Suppose G has a negative cycle C reachable from s. Then there is some node $v \in C$ such that $d(v, n) < d(v, n-1)$.

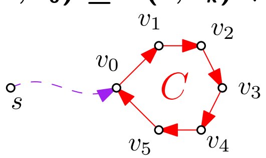
Proof.

Suppose not. Let $C = v_1 \to v_2 \to \ldots \to v_h \to v_1$ be negative length cycle reachable from s. $d(v_i, n-1)$ is finite for $1 \le i \le h$ since C is reachable from s. By assumption $d(v, n) \ge d(v, n-1)$ for all $v \in C$; implies no change in nth iteration; $d(v_i, n-1) = d(v_i, n)$ for $1 \le i \le h$. This means $d(v_i, n-1) \le d(v_{i-1}, n-1) + \ell(v_{i-1}, v_i)$ for $2 \le i \le h$ and $d(v_1, n-1) \le d(v_n, n-1) + \ell(v_n, v_1)$. Adding up all these inequalities results in the inequality $0 \le \ell(C)$ which contradicts the assumption that $\ell(C) < 0$. □

Proof in more detail...



$$
\begin{aligned}
d(s, v_1) &\leq d(s, v_0) + \ell(v_0, v_1)\\
d(s, v_2) &\leq d(s, v_1) + \ell(v_1, v_2)\\
&\dots\\
d(s, v_i) &\leq d(s, v_{i-1}) + \ell(v_{i-1}, v_i)\\
&\dots\\
d(s, v_k) &\leq d(s, v_{k-1}) + \ell(v_{k-1}, v_k)\\
d(s, v_0) &\leq d(s, v_k) + \ell(v_k, v_k)
\end{aligned}
$$



$$\sum_{i=0}^{k} d(s, v_i) \leq \sum_{i=0}^{k} d(s, v_i) + \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0)$$

$$0 \leq \sum^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0) = \mathrm{len}(C).$$

Finding the Paths and a Shortest Path Tree

How do we find a shortest path tree in addition to distances?

- For each v the $d(v)$ can only get smaller as algorithm proceeds.
- If $d(v)$ becomes smaller it is because we found a vertex u such that $d(v) > d(u) + \ell(u, v)$ and we update $d(v) = d(u) + \ell(u, v)$. That is, we found a shorter path to v through u.
- For each v have a $prev(v)$ pointer and update it to point to u if v finds a shorter path via u.
- At end of algorithm $prev(v)$ pointers give a shortest path tree oriented towards the source s.

Negative Cycle Detection

Negative Cycle Detection

Given directed graph G with arbitrary edge lengths, does it have a negative length cycle?

1. Bellman-Ford checks whether there is a negative cycle C that is reachable from a specific vertex s. There may negative cycles not reachable from s.
2. Run Bellman-Ford $|V|$ times, once from each node u?

Negative Cycle Detection

1. Add a new node s' and connect it to all nodes of G with zero length edges. Bellman-Ford from s' will fill find a negative length cycle if there is one. Exercise: why does this work?
2. Negative cycle detection can be done with one Bellman-Ford invocation.

Part II

Shortest Paths in DAGs

Shortest Paths in a

Single-Source Shortest Path Problems

Input A directed acyclic graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.

Simplification of algorithms for DAGs

1. No cycles and hence no negative length cycles! Hence can find shortest paths even for negative length edges
2. Can order nodes using topological sort

Algorithm for s

1. Want to find shortest paths from s. Ignore nodes not reachable from s.
2. Let $s = v_1, v_2, v_{i+1}, \ldots, v_n$ be a topological sort of G

Observation:

1. shortest path from s to v_i cannot use any node from $v_{i+1}, \ldots, v_n$
2. can find shortest paths in topological sort order.

Algorithm for s

```
for i = 1 to n do
        d(s, v_i) = ∞
d(s, s) = 0

for i = 1 to n − 1 do
        for each edge (v_i, v_j) in Adj(v_i) do
            d(s, v_j) = min{d(s, v_j), d(s, v_i) + ℓ(v_i, v_j)}

return d(s, ·) values computed
```

Correctness: induction on i and observation in previous slide.
Running time: $O(m + n)$ time algorithm! Works for negative edge lengths and hence can find *longest* paths in a DAG.

Bellman-Ford and s

Bellman-Ford is based on the following principles:

- The shortest walk length from s to v with at most k hops can be computed via dynamic programming
- G has a negative length cycle reachable from s iff there is a node v such that shortest walk length reduces after n hops.

We can find hop-constrained shortest paths via graph reduction. Given $G = (V, E)$ with edge lengths $\ell(e)$ and integer k construction new layered graph $G' = (V', E')$ as follows.

- $V' = V \times \{0, 1, 2, \ldots, k\}$.
- $E' = \{((u, i), (v, i + 1) \mid (u, v) \in E, 0 \le i < k\}$, $\ell((u, i), (v, i + 1)) = \ell(u, v)$

Lemma

Shortest path distance from $(u, 0)$ to (v, k) in G' is equal to the shortest walk from u to v in G with exactly k edges.

Layered DAG: Figure

Part III

All Pairs Shortest Paths

Shortest Path Problems

Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths (or costs). For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.
3. Find shortest paths for *all* pairs of nodes.

Single-Source Shortest Paths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.

Dijkstra's algorithm for non-negative edge lengths. Running time: $O((m + n) \log n)$ with heaps and $O(m + n \log n)$ with advanced priority queues.

Bellman-Ford algorithm for arbitrary edge lengths. Running time: $O(nm)$.

All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Find shortest paths for all pairs of nodes.

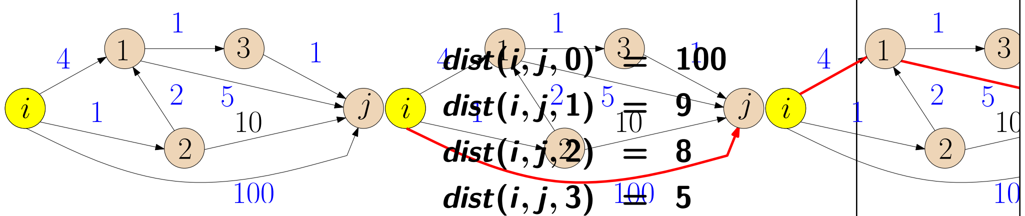
Apply single-source algorithms n times, once for each vertex.

1. Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.
2. Arbitrary edge lengths: $O(n^2 m)$.
   $\Theta(n^4)$ if $m = \Omega(n^2)$.

Can we do better?

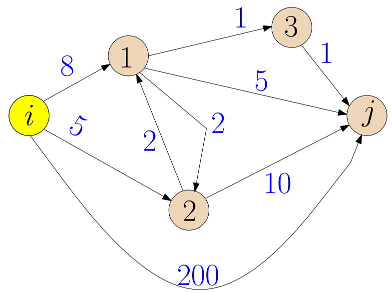
All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
2. $dist(i, j, k)$: length of shortest walk from v_i to v_j among all walks in which the largest index of an *intermediate node* is at most k (could be $-\infty$ if there is a negative length cycle).



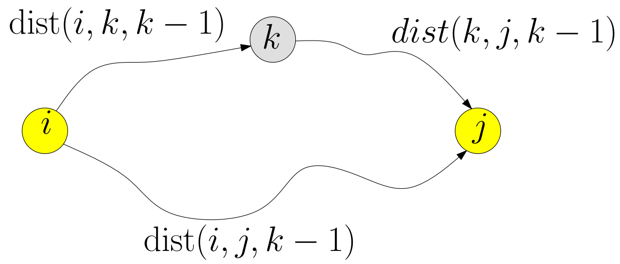
$dist(i, j, 0) = 100$
$dist(i, j, 1) = 9$
$dist(i, j, 2) = 8$
$dist(i, j, 3) = 5$

For the following graph, dist(i, j, 2) is...



(A) 9
(B) 10
(C) 11
(D) 12
(E) 15

All-Pairs: Recursion on index of intermediate nodes



$$\mathit{dist}(i,j,k) = \min \begin{cases} \mathit{dist}(i,j,k-1) \\ \mathit{dist}(i,k,k-1) + \mathit{dist}(k,j,k-1) \end{cases}$$

Base case: $\mathit{dist}(i,j,0) = \ell(i,j)$ if $(i,j) \in E$, otherwise ∞
Correctness: If $i \to j$ shortest walk goes through k then k occurs only once on the path — otherwise there is a negative length cycle.

All-Pairs: Recursion on index of intermediate nodes

If i can reach k *and* k can reach j *and* $\mathit{dist}(k,k,k-1) < 0$ then G has a negative length cycle containing k and $\mathit{dist}(i,j,k) = -\infty$.

Recursion below is valid only if $\mathit{dist}(k,k,k-1) \geq 0$. We can detect this during the algorithm or wait till the end.

$$\mathit{dist}(i,j,k) = \min \begin{cases} \mathit{dist}(i,j,k-1) \\ \mathit{dist}(i,k,k-1) + \mathit{dist}(k,j,k-1) \end{cases}$$

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

```
for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = ℓ(i, j)  (* ℓ(i, j) = ∞ if (i, j) ∉ E, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i, j, k) = min { dist(i, j, k − 1),
                                { dist(i, k, k − 1) + dist(k, j, k − 1)
for i = 1 to n do
    if (dist(i, i, n) < 0) then
        Output that there is a negative length cycle in G
```

Running Time: $\Theta(n^3)$, Space: $\Theta(n^3)$.
Correctness: via induction and recursive definition

Floyd-Warshall Algorithm: Finding the Paths

Question: Can we find the paths in addition to the distances?

1. Create a $n \times n$ array Next that stores the next vertex on shortest path for each pair of vertices
2. With array Next, for any pair of given vertices i, j can compute a shortest path in $O(n)$ time.

Floyd-Warshall Algorithm

Finding the Paths

```
for i = 1 to n do
    for j = 1 to n do
        dist(i,j,0) = ℓ(i,j)
(* ℓ(i,j) = ∞ if (i,j) not edge, 0 if i = j *)
        Next(i,j) = −1
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            if (dist(i,j,k−1) > dist(i,k,k−1) + dist(k,j,k−1)) then
                dist(i,j,k) = dist(i,k,k−1) + dist(k,j,k−1)
                Next(i,j) = k
for i = 1 to n do
    if (dist(i,i,n) < 0) then
        Output that there is a negative length cycle in G
```

Exercise: Given *Next* array and any two vertices i, j describe an $O(n)$ algorithm to find a i-j shortest path.

Summary of results on shortest paths

Single source		
No negative edges	Dijkstra	$O(n \log n + m)$
Edge lengths can be negative	Bellman Ford	$O(nm)$

All Pairs Shortest Paths

No negative edges	n * Dijkstra	$O(n^2 \log n + nm)$
No negative cycles	n * Bellman Ford	$O(n^2 m) = O(n^4)$
No negative cycles (*)	BF + n * Dijkstra	$O(nm + n^2 \log n)$
No negative cycles	Floyd-Warshall	$O(n^3)$
Unweighted	Matrix multiplication	$O(n^{2.38})$, $O(n^{2.58})$

Summary of results on shortest paths

More details

(*): The algorithm for the case that there are no negative cycles, and doing all shortest paths, works by computing a potential function using **Bellman-Ford** and then doing **Dijkstra**. It is mentioned for the sake of completeness, but it outside the scope of the class.

Part IV

DFA to Regular Expression

Back to Regular Languages

We saw the following two theorems previously.

Theorem

For every NFA N over a finite alphabet Σ there is DFA M such that $L(M) = L(N)$.

Theorem

For every regular expression r over finite alphabet Σ there is a NFA N such that $L(N) = L(r)$.

We claimed the following theorem which would prove equivalence of NFAs, DFAs and regular expressions.

Theorem

For every DFA M over a finite alphabet Σ there is a regular expression r such that $L(M) = L(r)$.

DFA to Regular Expression

Given DFA $M = (Q, \Sigma, \delta, q_1, F)$ want to construct an equivalent regular expression r.

Idea:

- Number states of DFA: $Q = \{q_1, \ldots, q_n\}$ where $|Q| = n$.
- Define $L_{i,j} = \{w \mid \delta(q_i, w) = q_j\}$. Note $L_{i,j}$ is regular. Why?
- $L(M) = \cup_{q_i \in F} L_{1,i}$.
- Obtain regular expression $r_{i,j}$ for $L_{i,j}$.
- Then $r = \sum_{q_i \in F} r_{1,i}$ is regular expression for $L(M)$ – here the summation is the or operator.

Note: Using q_1 for start state is intentional to help in the notation for the recursion.

A recursive expression for $L_{i,j}$

Define $L^k_{i,j}$ be set of strings w in $L_{i,j}$ such that the highest index state visited by M on walk from q_i to q_j (not counting end points i and j) on input w is at most k.

From definition

$$L_{i,j} = L^n_{i,j}$$

Claim:

$$L^0_{i,j} = \begin{cases} \{a \in \Sigma \mid \delta(q_i, a) = q_j\} & \text{if } i \neq j \\ \{a \in \Sigma \mid \delta(q_i, a) = q_j\} \cup \{\epsilon\} & \text{if } i = j \end{cases}$$

$$L^k_{i,j} = L^{k-1}_{i,j} \cup \left(L^{k-1}_{i,k} \cdot (L^{k-1}_{k,k})^* \cdot L^{k-1}_{k,j} \right)$$

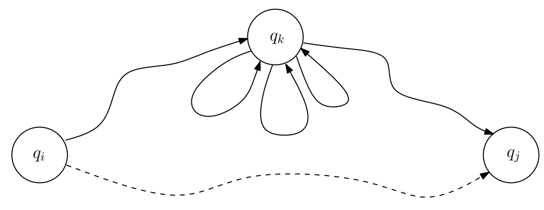
A recursive expression for $L_{i,j}$

Claim:

$$L^0_{i,j} = \begin{cases} \{a \in \Sigma \mid \delta(q_i, a) = q_j\} & \text{if } i \neq j \\ \{a \in \Sigma \mid \delta(q_i, a) = q_i\} \cup \{\epsilon\} & \text{if } i = j \end{cases}$$

$$L^k_{i,j} = L^{k-1}_{i,j} \cup \left(L^{k-1}_{i,k} \cdot (L^{k-1}_{k,k})^* \cdot L^{k-1}_{k,j} \right)$$

Proof: by picture



A recursive expression for $L_{i,j}$

$$L_{i,j} = L^n_{i,j}$$

Claim:

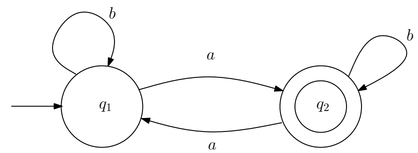
$$L^0_{i,j} = \{a \in \Sigma \mid \delta(q_i, a) = q_j\}$$

$$L^k_{i,j} = L^{k-1}_{i,j} \cup \left(L^{k-1}_{i,k} \cdot (L^{k-1}_{k,k})^* \cdot L^{k-1}_{k,j} \right)$$

From claim, can easily construct regular expression $r^k_{i,j}$ for $L^k_{i,j}$. This leads to a regular expression for

$$L(M) = \cup_{q_i \in F} L_{1,i} = \cup_{q_i \in F} L^n_{1,i}$$

Example



$$L(M) = L^2_{1,2}$$
$$r^2_{1,2} = r^1_{1,2} + r^1_{1,2}(r^1_{2,2})^* r^1_{2,2}$$
$$r^1_{1,2} = r^0_{1,2} + r^0_{1,1}(r^0_{1,1})^* r^0_{1,2}$$
$$r^1_{2,2} = r^0_{2,2} + r^0_{2,1}(r^0_{1,1})^* r^0_{1,2}$$

$$r^0_{1,1} = r^0_{2,2} = (b + \epsilon)$$
$$r^0_{1,2} = r^0_{2,1} = a$$

Correctness

Similar to that of Floyd-Warshall algorithms for shortest paths via induction.

The length of the regular expression can be exponential in the size of the original DFA.

Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

1. How to come up with the recursion?
2. How to recognize that dynamic programming may apply?

Some Tips

1. Problems where there is a *natural* linear ordering: sequences, paths, intervals, DAGs etc. Recursion based on ordering (left to right or right to left or topological sort) usually works.
2. Problems involving trees: recursion based on subtrees.
3. More generally:
   1. Problem admits a natural recursive divide and conquer
   2. If optimal solution for whole problem can be simply composed from optimal solution for each separate pieces then plain divide and conquer works directly
   3. If optimal solution depends on all pieces then can apply dynamic programming if *interface/interaction* between pieces is *limited*. Augment recursion to not simply find an optimum solution but also an optimum solution for each possible way to interact with the other pieces.

Examples

1. Longest Increasing Subsequence: break sequence in the middle say. What is the interaction between the two pieces in a solution?
2. Sequence Alignment: break both sequences in two pieces each. What is the interaction between the two sets of pieces?
3. Independent Set in a Tree: break tree at root into subtrees. What is the interaction between the subtrees?
4. Independent Set in an graph: break graph into two graphs. What is the interaction? Very high!
5. Knapsack: Split items into two sets of half each. What is the interaction?