Dynamic Programming: Shortest Paths and DFA to Reg Expressions

Lecture 18
March 28

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.

- Given nodes \( s, t \) find shortest path from \( s \) to \( t \).
- 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 \( s = v_0 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow v_k \) is a shortest path from \( s \) to \( v_k \) then \( \text{dist}(s, v_i) \leq \text{dist}(s, v_{i+1}) \) for \( 0 \leq i < k \). Holds true only for non-negative edge lengths.

Shortest Paths and Negative Cycles

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

- \( G \) has a negative length cycle \( C \), and
- \( s \) 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:

- undefined, that is \( -\infty \), OR
- the length of a shortest simple path from \( s \) to \( t \).

**Lemma**

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

Finding the \( s \rightarrow t \) longest path is difficult. **NP-Hard!**

Alternatively: Finding Shortest Walks

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

- 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 \( 1 \leq i \leq k - 1 \).
- 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 \( \text{dist}(u, v) \) to be the length of a shortest walk from \( u \) to \( v \).

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

Helpful to think about walks.
Shortest Paths with Negative Edge Lengths

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.

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 = 0.1116 Euro
2. 1 Euro = 1.3617 US dollar
3. 1 US Dollar = 7.1 Chinese Yuan.

Thus, if exchanging 1 $ → Yuan → Euro → $, we get: 
\[0.1116 \times 1.3617 \times 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 \( \text{exch}(i, k_1) \times \text{exch}(k_1, k_2) \times \ldots \times \text{exch}(k_h, j) \) units of \( j \).

Create currency trading directed graph \( G = (V, E) \):
- For each currency \( i \) there is a node \( v_i \in V \)
- \( E = V \times V \): an edge for each pair of currencies
- edge length \( \ell(v_i, v_j) = -\log(\text{exch}(i, j)) \) can be negative

**Exercise:** Verify that
- There is an arbitrage opportunity if and only if \( G \) has a negative length cycle.
- 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 \( j \).

Math recall - relevant information

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

**Lemma**

Let \( G \) be a directed graph with arbitrary edge lengths. If 
\( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \) is a shortest path from \( s \) to \( v_k \) then for \( 1 \leq i < k \):
- \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_i \) is a shortest path from \( s \) to \( v_i \).
- False: \( \text{dist}(s, v_i) \leq \text{dist}(s, v_k) \) for \( 1 \leq i < k \). Holds true only for non-negative edge lengths.

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

**Lemma**

Let \( G \) be a directed graph with arbitrary edge lengths. If 
\( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \) is a shortest path from \( s \) to \( v_k \) then for \( 1 \leq i < k \):
- \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_i \) 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: $\text{dist}(s, v) = d(v, n - 1)$. Recursion for $d(v, k)$:

$$d(v, k) = \min \left\{ \min_{u \in V} (d(u, k - 1) + \ell(u, v)), d(v, k - 1) \right\}$$

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

Bellman-Ford Algorithm

```python
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 $(u, v) \in \text{in}(v)$ do
            $d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\}$

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

Running time: $O(mn)$ Space: $O(m + n^2)$

Space can be reduced to $O(m + n)$.

Example

Bellman-Ford Algorithm

```python
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 $(u, v) \in \text{in}(v)$ do
            $d(v) = \min\{d(v), d(u) + \ell(u, v)\}$

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

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 $n$. 

```plaintext
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
            $d(v) = \min\{d(v), d(u) + \ell(u, v)\}$
        
    /* One more iteration to check if distances change */
    for each $v \in V$ do
        for each edge $(u, v) \in in(v)$ do
            if ($d(v) > d(u) + \ell(u, v)$)
                Output "Negative Cycle"
for each $v \in V$ do
    $dist(s, v) \leftarrow d(v)$
```

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 $G$ does not have a negative length cycle reachable from $s$. Then for all $v$, $dist(s, v) = d(v, n - 1)$. 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**

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 $C = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_h \rightarrow v_1$ be negative length cycle reachable from $s$. $d(v_i, n - 1)$ is finite for $1 \leq i \leq h$ since $C$ is reachable from $s$. By assumption $d(v, n) \geq d(v, n - 1)$ for all $v \in C$; implies no change in $n$th iteration; $d(v_i, n - 1) = d(v_i, n)$ for $1 \leq i \leq h$. This means $d(v_i, n - 1) \leq d(v_{i-1}, n - 1) + \ell(v_{i-1}, v_i)$ for $2 \leq i \leq h$ and $d(v_1, n - 1) \leq d(v_h, n - 1) + \ell(v_h, v_1)$. Adding up all these inequalities results in the inequality $0 \leq \ell(C)$ which contradicts the assumption that $\ell(C) < 0$. 

$\square$
Proof in more detail...

\[ d(s, v_1) \leq d(s, v_0) + \ell(v_0, v_1) \]
\[ d(s, v_2) \leq d(s, v_1) + \ell(v_1, v_2) \]
\[ \ldots \]
\[ d(s, v_i) \leq d(s, v_{i-1}) + \ell(v_{i-1}, v_i) \]
\[ \ldots \]
\[ d(s, v_k) \leq d(s, v_{k-1}) + \ell(v_{k-1}, v_k) \]
\[ d(s, v_0) \leq d(s, v_k) + \ell(v_k, v_0) \]
\[
\sum_{i=0}^{k} d(s, v_i) \leq \sum_{i=0}^{k} d(s, v_i) + \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0).
\]

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 \( \text{prev}(v) \) pointer and update it to point to \( u \) if \( v \) finds a shorter path via \( u \).
- At end of algorithm \( \text{prev}(v) \) pointers give a shortest path tree oriented towards the source \( s \).

Negative Cycle Detection

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

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

Negative Cycle Detection

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

Shortest Paths in DAGs

Shortest Paths in a DAG

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.

- Given nodes $s, t$ find shortest path from $s$ to $t$.
- Given node $s$ find shortest path from $s$ to all other nodes.

Simplification of algorithms for DAGs

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

Algorithm for $s$

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

Observation:

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

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.

Algorithm for $s$

```plaintext
for i = 1 to n do
    d(s, vi) = \infty
    d(s, s) = 0

for i = 1 to n - 1 do
    for each edge (vi, vj) in Adj(vi) do
        d(s, vj) = min(d(s, vj), d(s, vi) + \ell(vi, vj))

return d(s, \cdot) values computed
```

Sariel Har-Peled (UIUC) CS374 Fall 2017
Bellman-Ford and DAG

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.

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

**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.

Part III

All Pairs Shortest Paths

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.

- Given nodes $s, t$ find shortest path from $s$ to $t$.
- Given node $s$ find shortest path from $s$ to all other nodes.
- 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.

- Given nodes \( s, t \) find shortest path from \( s \) to \( t \).
- 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.

- Find shortest paths for all pairs of nodes.

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

- Non-negative lengths. \( O(nm \log n) \) with heaps and \( O(nm + n^2 \log n) \) using advanced priority queues.
- 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

- Number vertices arbitrarily as \( v_1, v_2, \ldots, v_n \).
- \( \text{dist}(i, j, k) \): 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).

For the following graph, \( \text{dist}(i, j, 2) \) is...

(A) 9  
(B) 10  
(C) 11  
(D) 12  
(E) 15
All-Pairs: Recursion on index of intermediate nodes

\[
\begin{align*}
dist(i, j, k) &= \min \left\{ dist(i, j, k - 1), dist(i, k, k - 1) + dist(k, j, k - 1) \right\} \\
\text{Base case: } dist(i, j, 0) &= \ell(i, j) \text{ if } (i, j) \in E, \text{ otherwise } \infty
\end{align*}
\]

Correctness: If \( i \rightarrow j \) shortest walk goes through \( k \) then \( k \) occurs only once on the path — otherwise there is a negative length cycle.

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

```plaintext
for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = \ell(i, j) (* \ell(i, j) = \infty if (i, j) \notin 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 \left\{ dist(i, j, k - 1), dist(i, k, k - 1) + dist(k, j, k - 1) \right\}

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**

```plaintext
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 cycles   | \( n \times \text{Dijkstra} \) | \( O(n^2 \log n + nm) \) |
| No negative cycles (\*) | \( n \times \text{Bellman Ford} \) | \( O(n^4) = O(n^4) \) |
| No negative cycles (\*) | BF + \( n \times \text{Dijkstra} \) | \( O(nm + n^4 \log n) \) |
| No negative cycles (\*) | Floyd-Warshall         | \( O(n^4) \)          |
| Unweighted           | Matrix multiplication  | \( O(n^{2.38}), O(n^{2.58}) \) |

### 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 $\Sigma$ there is DFA $M$ such that $L(M) = L(N)$.

**Theorem**

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

---

**DF A 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?
- $L(M) = \bigcup_{q_i \in F} L_{1,i}$.
- Obtain regular expression $r_{i,j}$ for $L_{i,j}$.
- Then $r = \sum_{q_i \in F} r_{i,i}$ is regular expression for $L(M)$ – here the summation is the $\lor$ 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 $L_{i,j}^k$ 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

$$L_{i,j} = L_{i,j}^n$$

**Claim:**

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

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

**Proof:** by picture
A recursive expression for $L_{i,j}$

$$L_{i,j} = L_{i,j}^n$$

Claim:

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

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

From claim, can easily construct regular expression $r_{i,j}^k$ for $L_{i,j}^k$. This leads to a regular expression for

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

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.

Example

$$L(M) = L_{1,2}^2$$

$$r_{1,2}^2 = r_{1,2}^1 + r_{1,2}^0 (r_{2,2}^1)^* r_{2,2}^1$$

$$r_{1,2}^1 = r_{1,2}^0 + r_{1,1}^0 (r_{1,1}^0)^* r_{1,2}^0$$

$$r_{1,2}^0 = r_{2,2}^0 + r_{2,1}^0 (r_{2,1}^0)^* r_{2,2}^0$$

$$r_{1,1}^0 = r_{2,2}^0 = (b + \epsilon)$$

$$r_{1,2}^0 = r_{2,1}^0 = a$$

Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

- How to come up with the recursion?
- How to recognize that dynamic programming may apply?
Some Tips

- 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.
- Problems involving trees: recursion based on subtrees.
- More generally:
  - Problem admits a natural recursive divide and conquer
  - If optimal solution for whole problem can be simply composed from optimal solution for each separate pieces then plain divide and conquer works directly
  - 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

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