CH-231-A

Algorithms and Data Structures

ADS

Lecture 40

Dr. Kinga Lipskoch

Spring 2022

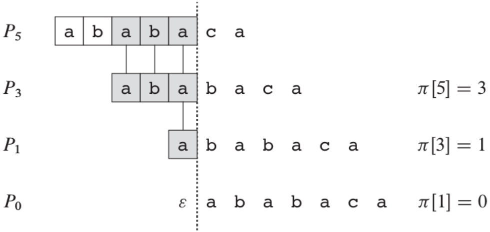
Prefix Function

Given a pattern $P[1..m]$, the prefix function for the pattern P is the function $\pi : \{1, 2, ..., m\} \rightarrow \{0, 1, ..., m-1\}$ such that $\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$.

Example: $P = ababaca$

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

KMP-Matcher

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0                                        // number of characters matched
for i = 1 to n                               // scan the text from left to right
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]                             // next character does not match
    if P[q + 1] == T[i]
        q = q + 1                            // next character matches
    if q == m                                // is all of P matched?
        print "Pattern occurs with shift" i − m
        q = π[q]                             // look for the next match
```

Compute Prefix

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

KMP Example (1)

Position	1	2	3	4	5	6	7	8	9
Pattern:	a	b	a	b	c	a	b	a	b
π	0	0	1	2	0	1	2	3	4

Position:	*0*	*1*	*2*	*3*	*4*	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Text:	*a*	*b*	*a*	*b*	**a**	b	c	b	a	b	a	b	c	a	b	a	b	c	a	b	...
Pattern:	*a*	*b*	*a*	*b*	**c**	a	b	a	b												

KMP Example (2)

Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Text:	a	b	*a*	*b*	a	b	c	b	a	b	a	b	c	a	b	a	b	c	a	b	...
Pattern:			*a*	*b*	a	b	c	a	b	a	b										

Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Text:	a	b	a	b	*a*	*b*	*c*	**b**	a	b	a	b	c	a	b	a	b	c	a	b	...
Pattern:			a	b	*a*	*b*	*c*	**a**	b	a	b										

KMP Example (3)

Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Text:	a	b	a	b	a	b	c	b	a	b	a	b	c	a	b	a	b	c	a	b	...
Pattern:								a	b	a	b	c	a	b	a	b					

Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Text:	a	b	a	b	a	b	c	**b**	a	b	a	b	c	a	b	a	b	c	a	b	...
Pattern:								**a**	b	a	b	c	a	b	a	b					

KMP Example (4)

Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Text:	a	b	a	b	a	b	c	b	a	b	a	b	c	a	b	a	b	c	a	b	...
Pattern:									a	b	a	b	c	a	b	a	b				

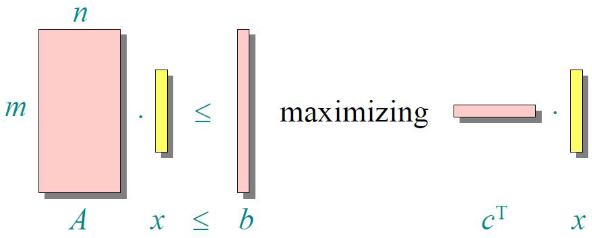
Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Text:	a	b	a	b	a	b	c	b	a	b	a	b	c	a	b	a	b	c	a	b	...
Pattern:														a	b	a	b	c	a	b	...

Time Complexity

- First, line 4 starts k at 0, and the only way that k increases is by the increment operation in line 9, which executes at most once per iteration of the for loop of lines $5 - 10$.
- Thus, the total increase in k is at most $m - 1$.
- Second, since $k < q$ upon entering the for loop and each iteration of the loop increments q, we always have $k < q$.
- Therefore, the assignments in lines 3 and 10 ensure that $\pi[q] < q$ for all $q = 1, 2, ..., m$, which means that each iteration of the while loop decreases k.
- Third, k never becomes negative.
- Therefore, the total decrease in k is $m - 1$.
- COMPUTE-PREFIX-FUNCTION runs in time $\Theta(m)$.
- Similarly, KMP-MATCHER runs in $\Theta(n)$.

Excurse: Linear Programming

Linear programming problem:
Let A be matrix of size $m \times n$, b a vector of size m, and c a vector of size n.
Find a vector x of size n that maximizes $c^T x$ subject to $Ax \leq b$, or determine that no such solution exists.



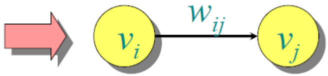
Example: Difference Constraints

Linear programming example, where each row of A contains exactly one 1 and one -1, other entries are 0.

$$\left.\begin{array}{l} x_1 - x_2 \le 3 \\ x_2 - x_3 \le -2 \\ x_1 - x_3 \le 2 \end{array}\right\} \quad x_j - x_i \le w_{ij}$$

Goal: Find 3-vector x that satisfies these inequations.
Solution: $x_1 = 3$, $x_2 = 0$, $x_3 = 2$.
Build constraint graph (matrix A of size $|E| \times |V|$):

$$x_j - x_i \le w_{ij}$$



Case 1: Unsatisfiable Constraints

Theorem:
If the constraint graph contains a negative-weight cycle, then the constraints are unsatisfiable.
Proof:
Suppose we have a negative-weight cycle:

$$v_1 \to v_2 \to \cdots \to v_k \to v_1.$$

Then,

$$\begin{aligned} x_2 - x_1 &\le w_{12} \\ x_3 - x_2 &\le w_{23} \\ &\vdots \\ x_k - x_{k-1} &\le w_{k-1,k} \\ x_1 - x_k &\le w_{k1} \end{aligned}$$

Summing the inequations delivers: $LHS = 0$, $RHS < 0$.
Hence, no x exists that satisfies the inequations.

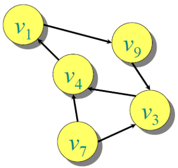
Case 2: Satisfiable Constraints (1)

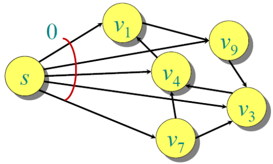
Theorem:

If no negative-weight cycle exists in the constraint graph, then the constraints are satisfiable.

Proof:

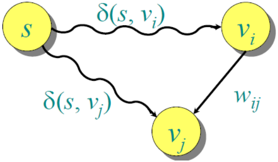
Add a vertex s with a 0-weight edge to all vertices. Note that this does not introduce a negative-weight cycle.





Case 2: Satisfiable Constraints (2)

Show that the assignments $x_i = \delta(s, v_i)$ for $i = 1, ..., n$ solve the constraints.
Consider any constraint $x_j - x_i \leq w_{ij}$.
Then, consider the shortest path from s to v_j and v_i.
The triangle inequality delivers $\delta(s, v_j) \leq \delta(s, v_i) + w_{ij}$.
Since $x_i = \delta(s, v_i)$ and $x_j = \delta(s, v_j)$, constraint $x_j - x_i \leq w_{ij}$ is satisfied.



Bellmann-Ford for Linear Programming

Corollary:
The Bellman-Ford algorithm can solve a system of m difference constraints on n variables in $O(m \cdot n)$ time.

Remark:
Single-source shortest paths is a simple linear programming problem.

All-Pairs Shortest Paths

Problem:

- So far, we considered the (single-source) shortest paths problem of finding the shortest paths from a source vertex $s \in V$.
- Now, we would like to extend this to finding all-pairs shortest paths.
- The input is, again, a directed graph $G = (V, E)$ with an edge-weight function $w : E \to \mathbb{R}$.
- Let $V = \{1, ..., n\}$.
- The output shall be an $n \times n$-matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.

Use Single-Source Shortest Paths

- Idea:
  Run the single-source shortest paths algorithm for each vertex $s \in V$ being the source once.
- Dijkstra's algorithm (for non-negative weights):
  Computation time $= O(|V| \cdot (|E| + |V|) \cdot lg(|V|))$ [min-heap]
  Worst-case $= \Theta(|V|^3 \cdot lg(|V|))$
- Bellman-Ford algorithm (for general case):
  Computation time $= O(|V|^2 \cdot |E|)$
  Worst-case $= \Theta(|V|^4)$

Dynamic Programming for All-Pairs Shortest Paths (1)

Consider the substructure:
$d_{ij}^{(m)} =$ weight of a shortest path
from i to j that uses at most m edges.

Theorem:

- Initially ($m = 0$), we have

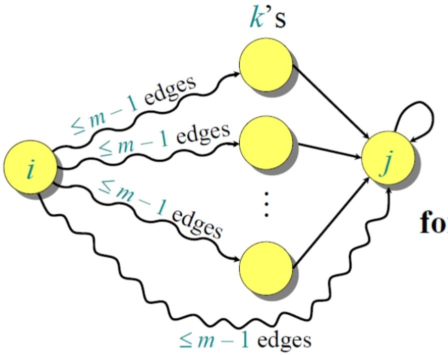
$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j; \end{cases}$$

- Then, for $m = 1, ..., n-1$, we have
$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$
where $A = (a_{ij})$ is the adjacency matrix

Dynamic Programming for All-Pairs Shortest Paths (2)

Proof:

$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$



for $k \leftarrow 1$ **to** n
  do if $d_{ij} > d_{ik} + a_{kj}$
    then $d_{ij} \leftarrow d_{ik} + a_{kj}$

Remark

- The dynamic programming strategy is to start with $m = 0$ and successively increase m until we reach $n - 1$.
- If we have no negative-weights cycles, we are done after $n - 1$ steps, i.e.,
  $\delta(i, j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \dots$

Implementation (1)

- The expression $d_{ij}^{(m)} = \min_k\{d_{ik}^{(m-1)} + a_{kj}\}$ updates all entries of the $n \times n$-matrix $D^{(m)} = (d_{ij}^{(m)})$ from the $n \times n$-matrices $D^{(m-1)}$ and A.
- We can use a matrix multiplication notation $D^{(m)} = D^{(m-1)} \cdot A$, where the typical operations "$+$" and "$\cdot$" are mapped to the operations "min" and "$+$".
- $D^{(0)}$ is the respective identity matrix

$$I = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix} = D^{(0)} = (d_{ij}^{(0)})$$

Implementation (2)

- The introduced matrix multiplication is associative and it can be shown that it forms a closed semi-ring (assuming real numbers).
- Hence, the dynamic programming algorithm executes the following computation steps:
  $D^{(1)} = D^{(0)} \cdot A = A^1$
  $D^{(2)} = D^{(1)} \cdot A = A^2$
  $\ldots$
  $D^{(n-1)} = D^{(n-2)} \cdot A = A^{n-1}$
  where the result is stored in
  $D^{(n-1)} = (\delta(i,j))$

Analysis

- Since we are executing $n-1$ matrix multiplications for matrices of size $n \times n$, the computation time is $\Theta(n \cdot n^3) = \Theta(n^4)$.
- Since $n = |V|$, this is not better than running n times the Bellman-Ford algorithm.
- However, we can exploit the generalized power-of-a-number recursion, which reduces the time complexity to $\Theta(n^3 \cdot \lg n)$.
- Note that n does not need to be a power of 2, as $A^{n-1} = A^n = A^{n+1} = \dots$

Summary

- Directed and undirected graphs
- Adjacency matrix vs. adjacency lists
- Graph search: BFS or DFS in $\Theta(|V| + |E|)$
- MST: Prim in $O(|E| \lg(|V|))$ for min-heap
- Single-source Shortest Paths:
  - Dijkstra for non-negative weights in $O((|V| + |E|) \lg(|V|))$ for min-heap
  - BFS for non-weighted edges in $\Theta(|V| + |E|)$
  - Bellman-Ford for all cases in $\Theta(|V| \cdot |E|)$