Bellman-Ford Algorithm for shortest paths

Input: Directed graph $G = (V, E)$, edge weights $w : E \mapsto \mathbb{R}$, source $s \in V$.

Output: For each $u \in V$, $u.distance = \delta(s, u)$, shortest path distance from s to u, and, $u.parent$ = predecessor of u in such a path. If G has a negative cycle, algorithm returns $false$, otherwise $true$.

Idea: Dynamic program of the following recursive algorithm.

Define $d_k(u)$ to be the length of a shortest path from s to u that uses at most k edges. When $k = 0$, $d_0(u) = \infty$, if $u \neq s$, and, $d_0(s) = 0$. Recurrence for d_k:

$$d_k(u) = \min\{d_{k-1}(u), \min_{(p,u)\in E}\{d_{k-1}(p) + w(p, u)\}\}.$$

If G does not have a negative cycle, then $d_{|V|-1}(u) = \delta(s, u)$, because a simple shortest path has at most $|V| - 1$ edges. In addition, if $d_k(u) = d_{k-1}(u)$, for all $u \in V$, then the recursion can be stopped at k. If G has a negative cycle, then $d_{|V|}(u) \neq d_{|V|-1}(u)$, for some $u \in V$, and the algorithm returns $false$.

Dynamic program to compute d_k: Take 1

```
// Store d_k(u) in array d[ ] defined in Vertex class.
// Solve problems in increasing values of k to avoid recursive calls.
for u ∈ V do
    u.d[0] ← ∞;   u.parent ← null
s.d[0] ← 0
// Invariant: u.d[k − 1] = d_{k−1}(u), for all u ∈ V.
for k ← 1 to |V| do
    nochange ← true
    for u ∈ V do
        u.d[k] ← u.d[k − 1]
        for edge e = (p, u) ∈ E do
            if u.d[k] > p.d[k − 1] + w(e) then
                u.d[k] ← p.d[k − 1] + w(e)
                u.parent ← p
                nochange ← false
    if nochange then
        for u ∈ V do u.distance ← u.d[k]
        return true
return false // G has a negative cycle
```

Dynamic program to compute d_k: Take 2

Recurrence for d_k is guaranteed to be feasible, and therefore all elements of $u.d[\,]$ can be overlaid on the same location, thus replacing the array by a scalar, $u.distance$. In addition, all edges are relaxed in each iteration of k. Edges of the graphs can be relaxed in any order.

```
Bellman-Ford(Graph G = (V, E), Vertex s)
for u ∈ V do
    u.distance ← ∞
    u.parent ← null
s.distance ← 0
for k ← 1 to |V| do
    nochange ← true
    for edge e = (u, v) ∈ E do
        if v.distance > u.distance + w(e) then
            v.distance ← u.distance + w(e)
            v.parent ← u
            nochange ← false
    if nochange then
        return true
return false // G has a negative cycle
```

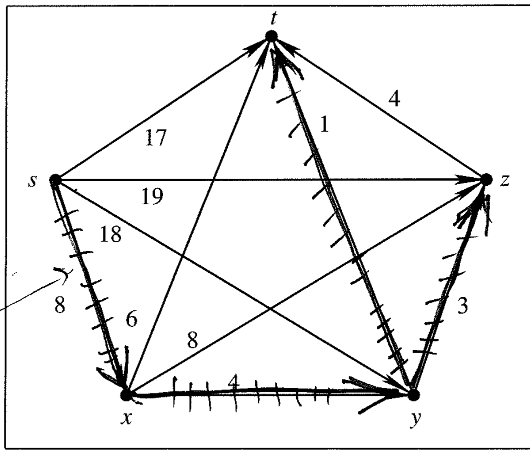
Faster algorithm: Take 3

Process edges out of u only when $u.distance$ changes. Keep track of how many times a node has been processed in field *count*. Worst-case RT is $O(|E||V|)$, but actual RT for many graphs is significantly less than the algorithm in Take 2.

```
Create a queue q to hold vertices waiting to be processed
for u ∈ V do
    u.distance ← ∞; u.parent ← null; u.count ← 0; u.seen ← false
s.distance ← 0; s.seen ← true; q.add(s)
while q is not empty do
    u ← q.remove(); u.seen ← false // no longer in q
    u.count ← u.count + 1
    if u.count ≥ |V| then return false // Negative cycle
    for Edge e = (u, v) ∈ u.Adj do
        if v.distance > u.distance + w(e) then
            v.distance ← u.distance + w(e)
            v.parent ← u
            if not v.seen then
                q.add(v); v.seen ← true
return true
```

Shortest path worksheet

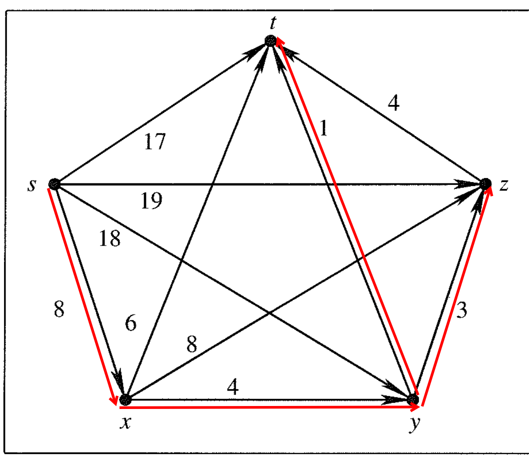
Shortest path tree



u	$u.d$	$u.\pi$
s	0	–
x	~~∞~~ 8	s
y	~~∞~~ ~~18~~ 12	~~s~~ x
z	~~∞~~ ~~19~~ ~~16~~ 15	~~s~~ ~~x~~ y
t	~~∞~~ ~~17~~ ~~14~~ 13	~~s~~ ~~x~~ y

DAG shortest path: Top order: ~~s~~ ~~x~~ ~~y~~ z t

Shortest path worksheet : Dijkstra's algorithm



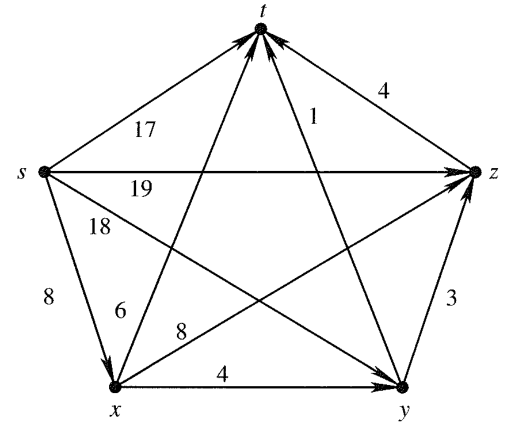
u	$u.d$	$u.\pi$
s	0	
x	~~∞~~ 8	s
y	~~∞~~ ~~18~~ 12	~~s~~ x
z	~~∞~~ ~~19~~ ~~16~~ 15	~~s~~ ~~x~~ y
t	~~∞~~ ~~17~~ ~~14~~ 13	~~s~~ ~~x~~ y

Removal order from Priority Queue: s x y t z ⟶ 根据u. π得到的结果

Bellman Ford - Take 2

Shortest path worksheet

t
z
y
x
s



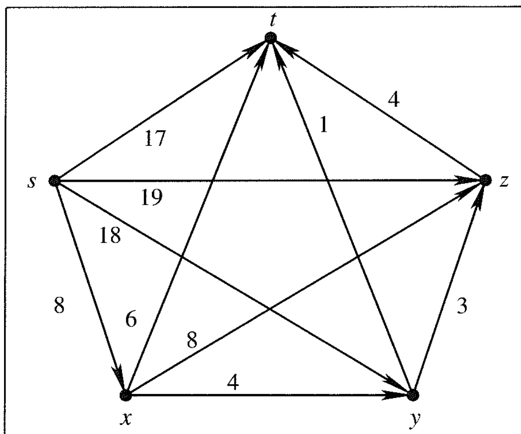
u	K=0 $u.d$ K=1 K=2 K=3 K=4	$u.\pi$
s	0 0 0 0	
x	~~∞~~ 8 8 8	s
y	~~∞~~ ~~18~~ 12 12	~~s~~ x
z	~~∞~~ ~~19~~ ~~16~~ 15	~~s~~ ~~x~~ y
t	~~∞~~ ~~17~~ ~~14~~ 13	~~s~~ ~~x~~ y

K

© Balaji Raghavachari 211 Algorithm Design and Analysis

Bellman-Ford Take 3

Shortest path worksheet



u	$u.d$	$u.\pi$
s	0	
x	~~∞~~ 8	s
y	~~∞~~ ~~18~~ 12	~~s~~ x
z	~~∞~~ ~~19~~ ~~16~~ 15	~~s~~ ~~x~~ y
t	~~∞~~ ~~17~~ ~~14~~ 13	~~s~~ ~~x~~ y

Queue: ~~s~~ ~~s~~ ~~t~~ ~~z~~ ~~y~~ x ~~t~~ ~~z~~ ~~y~~ ~~z~~ ~~t~~

Constrained shortest path problems

Input: $G=(V,E)$, $w: E \to \mathbb{R}$, limit k on the number of edges in a path.

Output: Find a shortest path from s to $t \in V$ that uses at most k edges.

Take 1 of Bellman-Ford.

P2: Find a s.p from s to t using exactly k edges.

P2 + ~~single path~~ difficult

P2 + ~~simple path~~

Take 1 of Bellman-Ford +

$$d_k(u) = \min_p \{ \cancel{d_{k-1}(u)} + d_{k-1}(p) + w(p,u) \}$$

Accelerated versions of both algorithms are possible. } Double # of edges in each iteration
- idea is similar to computing x^n.

Enumeration problems

- Commonly used in testing

Ex: Permutations and Combinations

nP_k — all permutations of k things out of n distinct objects

nC_k — all combinations of k things out of n distinct objects

Ex: $A = \{1, 2, 3, 4\}$

$k = 2$

$4C_2$	1 2	1 3	1 4	2 3	2 4	3 4
$4P_2$	1 2	1 3	1 4	2 3	2 4	3 4
	2 1	3 1	4 1	3 2	4 2	4 3

$$|nP_k| = n(n-1)(n-2)\dots(n-k+1) \approx n^k$$

$$|nC_k| = \frac{nP_k}{k!} \approx \frac{n^k}{k!}$$

Permutations: Algorithm nPk: **Input**: n distinct elements in an array A[n], integer k $\in$ [0, n]. **Goal**: Visit all permutations of k elements out of n. Initial call: permute(k). In algorithms below, A, n, k are class fields.

```
// Already selected: A[ 0···d−1 ]. Need to select c more elements from A[ d···n−1 ], where d = k−c.
permute( c ):
    if c = 0 then  visit permutation in A[ 0···k−1 ]
   else
      d ← k−c
      permute( c−1 )                                    // Permutations having A[d] as the next element
      for i ← d+1 to n−1 do
         tmp ← A[ d ];  A[ d ] ← A[ i ];  A[ i ] ← tmp    // After swap, A[ i ] = tmp
         permute( c−1 )                                 // Permutations having A[ i ] as the next element
         A[ i ] ← A[ d ];   A[ d ] ← tmp                // Restore elements where they were before swap
```

Heap's algorithm: A faster algorithm for generating all n! permutations, using just one swap for generating each permutation from the previous one. Initial call: heap(n). Proving correctness is tricky.

```
heap( g ):  // g elements to go ( A[ 0···g−1 ] ).  A[ g···n−1 ] are done.
   if g = 1 then  visit permutation A[ 0···n−1 ]
   else
      for i ← 0 to g−2 do
         heap( g−1 )
         if g is even then Exchange A[ i ] ↔ A[ g−1 ]
         else Exchange A[ 0 ] ↔ A[ g−1 ]
      heap( g−1 )
```

Knuth's L algorithm (invented by Narayana Pandita in 14th century, as per Wikipedia): This algorithm generates permutations in lexicographic order, and is useful when the input elements are not distinct.
Input: Sorted array: A[0] $\le$ A[1] $\le$ $\cdots$ $\le$ A[n−1].

```
Visit permutation given by A
while A is not in descending order do
   Find max index j such that A[ j ] < A[ j+1 ]
   Find max index k such that A[ j ] < A[ k ]      // k > j
   Exchange A[ j ]  ↔  A[ k ]
   Reverse A[ j+1···n−1 ]      // After this step, A[ j+1···n−1 ] is in descending order
   Visit permutation given by A
```

Combinations: Algorithm nCk: Initial call: combination(0, k). Uses array chosen[k] for output.

```
combination( i, c ):  // Choose c more items from A[ i···n−1 ].  Already selected: chosen[ 0···k−c−1 ].
   if c = 0 then  visit combination in chosen[ 0···k−1 ]
   else
      chosen[ k−c ] ← A[ i ]                  // Choose A[ i ]
      combination( i+1, c−1 )
      if n−i > c then
         combination( i+1, c )   // Skip A[ i ] only if there are enough elements left
```

Sample run of Knuth's L algorithm: in each permutation, j is shown in red and k in blue:
j = rightmost index with A[j] < A[j + 1], k = rightmost index with A[j] < A[k]. Input: 1 2 2 3 3 4

P_1 = 1 2 2 3 3 4	P_2 = 1 2 2 3 4 3	P_3 = 1 2 2 4 3 3	P_4 = 1 2 3 2 3 4	P_5 = 1 2 3 2 4 3
P_6 = 1 2 3 3 2 4	P_7 = 1 2 3 3 4 2	P_8 = 1 2 3 4 2 3	P_9 = 1 2 3 4 3 2	P_{10} = 1 2 4 2 3 3
P_{11} = 1 2 4 3 2 3	P_{12} = 1 2 4 3 3 2	P_{13} = 1 3 2 2 3 4	P_{14} = 1 3 2 2 4 3	P_{15} = 1 3 2 3 2 4
P_{16} = 1 3 2 3 4 2	P_{17} = 1 3 2 4 2 3	P_{18} = 1 3 2 4 3 2	P_{19} = 1 3 3 2 2 4	P_{20} = 1 3 3 2 4 2
P_{21} = 1 3 3 4 2 2	P_{22} = 1 3 4 2 2 3	P_{23} = 1 3 4 2 3 2	P_{24} = 1 3 4 3 2 2	P_{25} = 1 4 2 2 3 3
P_{26} = 1 4 2 3 2 3		etc.		
P_{176} = 4 3 2 3 1 2	P_{177} = 4 3 2 3 2 1	P_{178} = 4 3 3 1 2 2	P_{179} = 4 3 3 2 1 2	P_{180} = 4 3 3 2 2 1

Other interesting enumeration problems:

- Permutations consistent with a given set of precedence constraints (equivalent to enumerating topological orders of a DAG):

  Given a set of elements A[$0 \cdots n-1$], and a set of precedence constraints (x_i, y_i), i = $1 \cdots k$, output only those permutations in which x_i precedes y_i.

  Example: A = { 1, 2, 3 ,4 }, with the constraints { (1, 3), (2, 4), (3, 4) }. Output is { 1234, 1324, 2134 }.

  The problem is modeled as a graph problem, by creating a directed graph G in which, elements are represented by vertices, and constraints are represented by directed edges (from x_i to y_i, i = $1 \cdots k$). If the graph has a cycle, then there are no permutations consistent with the precedence constraints. Otherwise, the list of permutations to be output are exactly the different topological orders of G.

- Count or enumerate shortest paths in graphs: it is possible to count the number of shortest paths from a source vertex s to each vertex of a graph G efficiently, if all cycles of G have positive weight.

  The idea behind the algorithm is the following. Consider the subgraph H of G that includes only those edges e = (u, v) of G, such that δ(s, v) = δ(s, u) + (u, v).weight. If G is undirected, then orient the edge from u to v. If all cycles of G have positive length, then it can be shown that H is a DAG, and for any vertex u, all paths from s to u in H are shortest paths from s to u in G. Therefore all we need to do is count the number of paths in H from s to u, for all vertices u. This problem is easily solved in DAGs by writing a recurrence for count(u), the number of paths in H from s to u, and computing count(u) in topological order of the vertices.

  The problem of enumerating all shortest paths is not solvable in polynomial time, because the number of paths can be exponential in the size of G, but using ideas from solutions to the problem of enumerating topological orders, it is possible to design an algorithm for this problem, whose running time is proportional to the number of paths in the output.