

Smallest path

edgeTo[] 记录最短路径的上一个节点, distTo[] 记录距离的累加

relax 加入边的操作, 如果初始到v的距离加上v到w的距离小于之前最短到w的距离, 最短的距离等于到v的距离加w的距离, edgeTo的w的父节点改成v -> w边, 这里数组不是用一个节点, 而是用边来追踪路径.

```
private void relax(DirectedEdge e){
    int v = e.from(), w = e.to();
    if(distTo[w] > distTo[v] + e.weight()){
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```

Dijkstra's algorithm 有向图

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
5→7	6.0

So and a way we're going to do that is take the next vertex, add it to the tree

如图, 起点0, 先找最短的edge 0->1 因为他是0出发最短的, 所以不会有比他更短的了, 然后1->7,2,3 如果 $\text{distTo}(1) + \text{edge}(1-k)$ 小于 $\text{distTo}(k)$,就是relax所有的边, 如果小于就更新. 然后0-1检查过了, 继续下一个最短的, (0-7) 把7所有的边relax.

这里有一个decreaseKey的方法, 建议使用 `PriorityQueue.remove(Obj o)`然后再使用`PriorityQueue.add(Obj o)`的方法 简单一些.

实现

Dijkstra's algorithm: Java implementation

```
public class DijkstraSP
{
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;

    public DijkstraSP(EdgeWeightedDigraph G, int s)
    {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<Double>(G.V());

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        pq.insert(s, 0.0);
        while (!pq.isEmpty())
        {
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }
}
```

← relax vertices in order of distance from s

34

Dijkstra's algorithm: Java implementation

```
private void relax(DirectedEdge e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
        if (pq.contains(w)) pq.decreaseKey(w, distTo[w]); ← update PQ
        else pq.insert(w, distTo[w]);
    }
}
```

复杂度

Dijkstra's algorithm: which priority queue?

Share

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
unordered array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way heap (Johnson 1975)	$d \log_d V$	$d \log_d V$	$\log_d V$	$E \log_{d+1} V$
Fibonacci heap (Fredman-Tarjan 1984)	1 †	$\log V$ †	1 †	$E + V \log V$

† amortized

Bottom line.

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.

拓扑排序, 有向无环图, 无负值 **acyclic**(无环)

如果有负数的图, 0->3贪心的最短路径不会考虑到2->3是负数

Dijkstra. Doesn't work with negative edge weights.



复杂度

Single source shortest-paths implementation: cost summary

algorithm	restriction	typical case	worst case	extra space
topological sort	no directed cycles	$E + V$	$E + V$	V
Dijkstra (binary heap)	no negative weights	$E \log V$	$E \log V$	V
Bellman-Ford	no negative cycles	$E V$	$E V$	V
Bellman-Ford (queue-based)		$E + V$	$E V$	V

Remark 1. Directed cycles make the problem harder.

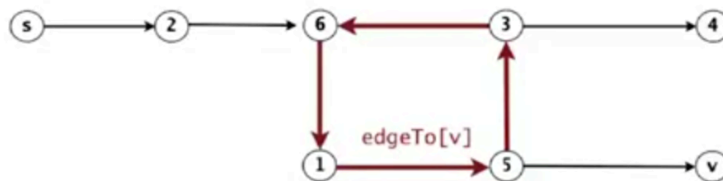
Remark 2. Negative weights make the problem harder.

Remark 3. Negative cycles makes the problem intractable.

如果最后一次还有更新,那么他肯定存在着环

Finding a negative cycle

Observation. If there is a negative cycle, Bellman-Ford gets stuck in loop, updating `distTo[]` and `edgeTo[]` entries of vertices in the cycle.



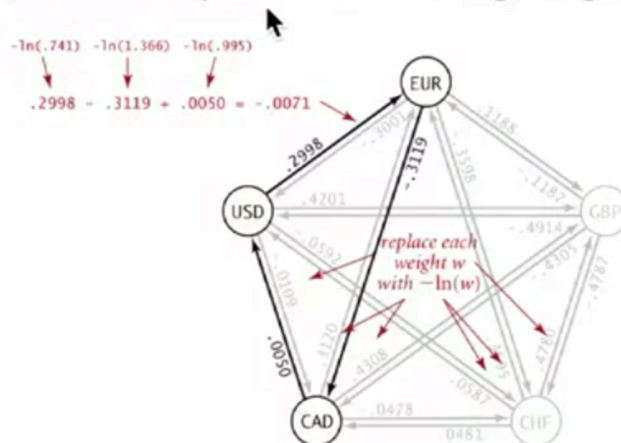
Proposition. If any vertex v is updated in phase v , there exists a negative cycle (and can trace back `edgeTo[v]` entries to find it).

汇率问题转化成negative cycle

Negative cycle application: arbitrage detection

Model as a negative cycle detection problem by taking logs.

- Let weight of edge $v \rightarrow w$ be $-\ln$ (exchange rate from currency v to w).
- Multiplication turns to addition; > 1 turns to < 0 .
- Find a directed cycle whose sum of edge weights is < 0 (negative cycle).



work schedule转换成shortest path

Critical path method

CPM. To solve a parallel job-scheduling problem, create edge-weighted DAG:

- Source and sink vertices.
- Two vertices (begin and end) for each job.
- Three edges for each job.
 - begin to end (weighted by duration)
 - source to begin (0 weight)
 - end to sink (0 weight)
- One edge for each precedence constraint (0 weight).

job	duration	must complete before		
0	41.0	1	7	9
1	51.0	2		
2	50.0			
3	36.0			
4	38.0			
5	45.0			
6	21.0	3	8	
7	32.0	3	8	
8	32.0	2		
9	29.0	4	6	

