Dijkstra

Referências

- Directed graphs (S&W)
- slides (S&W)

Vídeos

- Directed graphs (S&W)

Caminhos mínimos

O **custo/peso de um caminho** é a soma dos custos/pesos do seus arcos. Um caminho `P` é um caminho mínimo se o custo de `P` é menor ou igual ao custo de todo caminho com a mesma origem e término.

Problema

Problema dos caminhos mínimos com origem fixa (*Sigle-source shortest paths problem*): Dado um vértice `s` de um digrafo com custos **não-negativos** nos arcos, encontrar, para cada vértice `t` que pode ser alcançado a partir de `s`, um caminho mínimo de `s` a `t`.

Arborescência de caminhos-mínimos

Uma arborescência com raiz `s` é de **caminhos mínimos** (=*shortest-paths tree* = SPT) se para todo vértice `t` que pode ser alcançado a partir de `s`,

> o único caminho de `s` a `t` na arborescência é um caminho mínimo

Problema: Dado um vértice `s` de um digrafo com custos **não-negaticos** nos arcos, encontrar uma SPT com raiz `s`.

Classe DijkstraSP

```
public class DijkstraSP {
    private double[] distTo;        // distTo[v] = distance  of shortest s->v path
    private DirectedEdge[] edgeTo;  // edgeTo[v] = last edge on shortest s->v path
    private IndexMinPQ<Double> pq;  // priority queue of vertices

    // Computes a shortest-paths tree from the source vertex s to every other
    // vertex in the edge-weighted digraph  G.
    public DijkstraSP(EdgeWeightedDigraph G, int s) {
        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];
```

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

    // relax vertices in order of distance from s
    pq = new IndexMinPQ<Double>(G.V());
    pq.insert(s, distTo[s]);
    while (!pq.isEmpty()) {
        int v = pq.delMin();
        for (DirectedEdge e : G.adj(v)) {
            // relax(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]);
                else                pq.insert(w, distTo[w]);
        }
    }
}

// relax edge e and update pq if changed
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]);
        else                pq.insert(w, distTo[w]);
    }
}

// returns the length of a shortest path from the source vertex s to vertex v.
public double distTo(int v) {
    return distTo[v];
}

// returns true if there is a path from the source vertex s to vertex v.
public boolean hasPathTo(int v) {
    return distTo[v] < Double.POSITIVE_INFINITY;
}

// returns a shortest path from the source vertex s to vertex v.
public Iterable<DirectedEdge> pathTo(int v) {
    if (!hasPathTo(v)) return null;
    Stack<DirectedEdge> path = new Stack<DirectedEdge>();
    for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()]) {
        path.push(e);
    }
    return path;
```

```
}


// check optimality conditions:
// (i) for all edges e:            distTo[e.to()] <= distTo[e.from()] + e.weight()
// (ii) for all edge e on the SPT: distTo[e.to()] == distTo[e.from()] + e.weight()
private boolean check(EdgeWeightedDigraph G, int s) {

    // check that edge weights are nonnegative
    for (DirectedEdge e : G.edges()) {
        if (e.weight() < 0) {
            System.err.println("negative edge weight detected");
            return false;
        }
    }

    // check that distTo[v] and edgeTo[v] are consistent
    if (distTo[s] != 0.0 || edgeTo[s] != null) {
        System.err.println("distTo[s] and edgeTo[s] inconsistent");
        return false;
    }
    for (int v = 0; v < G.V(); v++) {
        if (v == s) continue;
        if (edgeTo[v] == null && distTo[v] != Double.POSITIVE_INFINITY) {
            System.err.println("distTo[] and edgeTo[] inconsistent");
            return false;
        }
    }

    // check that all edges e = v->w satisfy distTo[w] <= distTo[v] + e.weight()
    for (int v = 0; v < G.V(); v++) {
        for (DirectedEdge e : G.adj(v)) {
            int w = e.to();
            if (distTo[v] + e.weight() < distTo[w]) {
                System.err.println("edge " + e + " not relaxed");
                return false;
            }
        }
    }

    // check that all edges e = v->w on SPT satisfy distTo[w] == distTo[v] + e.weight()
    for (int w = 0; w < G.V(); w++) {
        if (edgeTo[w] == null) continue;
        DirectedEdge e = edgeTo[w];
        int v = e.from();
        if (w != e.to()) return false;
        if (distTo[v] + e.weight() != distTo[w]) {
            System.err.println("edge " + e + " on shortest path not tight");
            return false;
        }
```

```
        }
        return true;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        int V = distTo.length;
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1)
    }
}
```

Classe AcyclicSP

```
// The {@code AcyclicSP} class represents a data type for solving the
// single-source shortest paths problem in edge-weighted directed acyclic
// graphs (DAGs). The edge weights can be positive, negative, or zero.
public class AcyclicSP {
    private double[] distTo;          // distTo[v] = distance  of shortest s->v path
    private DirectedEdge[] edgeTo;    // edgeTo[v] = last edge on shortest s->v path

    // Computes a shortest paths tree from {@code s} to every other vertex in
    // the directed acyclic graph {@code G}.
    public AcyclicSP(EdgeWeightedDigraph G, int s) {
        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];

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

        // visit vertices in toplogical order
        Topological topological = new Topological(G);
        for (int v : topological.order()) {
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }

    // relax edge e
    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;
        }
    }

    // returns the length of a shortest path from the source vertex {@code s} to vertex {@cod
    public double distTo(int v) {
        validateVertex(v);
        return distTo[v];
    }

    // is there a path from the source vertex {@code s} to vertex {@code v}?
    public boolean hasPathTo(int v) {
        return distTo[v] < Double.POSITIVE_INFINITY;
    }

    // returns a shortest path from the source vertex {@code s} to vertex {@code v}.
    public Iterable<DirectedEdge> pathTo(int v) {
        if (!hasPathTo(v)) return null;
```

```
    Stack<DirectedEdge> path = new Stack<DirectedEdge>();
    for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()]) {
        path.push(e);
    }
    return path;
}
```

PQ com chaves mutáveis

Não sei se *PQ com chaves mutáveis* é um bomo nome para o que S&W chamam de *index priority queues*.

Em algumas aplicações é razoável permitirmos que o cliente altere a prioridade de um item que já esta na fila.

Uma maneira de lidar com isso é associar um único índice a cada item. Já comentamos essa estratégia quando tratamos de *union-find*.

API

```
public class IndexMinPQ<Item extends Comparable<Item>>
-------------------------------------------------------
         IndexMinPQ(int maxN)
    void insert(int k, Item item)  insere um item associa a k
    void change(int k, Item item)  muda o item associado com k para item
 boolean contains(int k)           k está associado a algum item
    void delete(int k)             remove k e o item associado
    Item min()                     retorna o menor item
     int minIndex()                retorna o índice do menor item
     int delMin()                  retorna o menor item e retorna o seu índice
 boolean isEmpty()                 a fila está vazia?
     int size()                    número de itens
```

Implementação

Exrecício 2.4.33 e 2.4.34 do S&W.

Alterar de `MaxPQ.java` para `IndexMinPQ.java`. Note que `swim()` e `sink()` não foram alterados, mas `exch()` e `less()` foram :-).

```
public class IndexMinPQ<Item extends Comparable<Item>>{
    private int n = 0; // heap fica em pq[1..n]

    private int[] pq; // heap binário
    private int[] qp; // inversa de pq: qp[pq[i]] = pq[qp[i]] = i
    private Item[] itens;

    public IndexMinPQ(int maxN) { // construtor
        itens = (Item[]) new Comparable[maxN+1];
        pq   = new int[maxN+1];
        qp   = new int[maxN+1];
        for (int i = 0; i <= maxN; i++) {
            qp[i] = -1;
        }
    }

    public boolean isEmpty() {
        return n == 0;
    }
```

```
public boolean contains(int k) {
    return qp[k] != -1;
}

public int size() {
    return n;
}

public void insert(int k, Item item) {
    n++;
    pq[n] = k;
    itens[k] = item;
    qp[k] = n;
    swim(n);
}

public void change(int k, Item item) {
    itens[k] = item;
    swim(qp[k]);
    sink(qp[k]);
}

public Item minKey() {
    return itens[pq[1]];
}

public int minIndex() {
    return pq[1];
}

public int delMin() {
    int indexOfMin = pq[1];
    exch(1, n--);
    sink(1);
    itens[pq[n+1]] = null; // avoid loitering
    qp[pq[n+1]] = -1;
    return indexOfMin;
}

public void delete(int k) {
    int j = pq[n];
    exch(qp[k],n--);
    // heapfy
    sink(qp[j]);
    swim(qp[j]);
    // destroi o rastros de k
    itens[k] = null;
    qp[k] = -1;
}
```

```
    // não altera :-)
    private void swim(int k) {
        while (k > 1 && less(k/2, k)) {
            exch(k/2, k);
            k = k/2;
        }
    }

    // não altera :-)
    private void sink(int k) {
        while (2*k <= n) {
            int j = 2*k;
            if (j < n && less(j, j+1)) j++;
            if (!less(k, j)) break;
            exch(k, j);
            k = j;
        }
    }

    // altera
    private boolean less(int i, int j) {
        return itens[pq[i]].compareTo(itens[pq[j]]) > 0;
    }

    // altera
    private void exch(int i, int j) {
        int t = pq[i];
        pq[i] = pq[j];
        pq[j] = t;
        // para consistência
        qp[pq[i]] = i;
        qp[pq[j]] = j;

    }
}
```