Flow Report

Adam Schønemann

Results

Our implementation successfully computes a flow of 163 on the input file, confirming the analysis of the American enemy.

We have analysed the possibilities of descreasing the capacities near Minsk. Our analysis is summaries in the following table:

case	4W-48	4W-49	Effect on Flow
1	30	30	no change (original)
2	30	20	no change
3	20	30	no change
4	20	20	no change

In case 4 we have a new bottleneck, namely 10-25, 11-25, 11-24, 12-23, 18-22, 20-23, 20-21, 27-26, 28-30

The comrade from Minsk is advised to reduce the capacity, since it does not affect the maximum throughput of backdoored iPhones we can ship to the capitalists!.

Implementation details

We use Schroeder et al's [1] algorithm for finding maximum flows of undirected graphs. The algorithm creates two residual edges for each undirected edge - one in each direction. Initially, each edge in the residual graph has a capacity equal to their corresponding edge in the actual graph. If you push x flow in one direction through an edge, the residual edge that goes in the other direction has its capacity incremented by x, while the residual edge that goes in the same direction has its capacity decremented by x.

[1] Schroeder et al, 2004

Augmented paths are found using a standard breadth-first search.

Implementation

Nodes contain an id, a label and adjacency lists

```
  trait Node {
    val id:Int
    val nodesOut:Seq[Int]
    val nodesIn:Seq[Int]
  }
```

Since the rail network is undirected, our concrete implementation of the trait is

```
  case class FlowNode(id:Int, label:String, neighbours:List[Int] = Nil) extends Node {

    lazy val nodesOut = neighbours
    lazy val nodesIn = neighbours
    ...
  }
```

The graph stores the capacities and flows in `Map[(Int,Int),Int]`

```
case class RailNetwork(nodes:IndexedSeq[FlowNode], capacities:Capacities = Map[(Int,Int),Int](),
                       flow:Flow = Map[(Int,Int),Int]())
             extends FlowNetwork[FlowNode] {

    ...
    def capacity(a:Int, b:Int):Int =
      capacities.get((a,b)).orElse(capacities.get((b,a)))
        .map(c => if (c < 0) Integer.MAX_VALUE else c).get

    def flowFrom(a:Int,b:Int):Int =
      flow.get((a,b)).getOrElse(0)

    def increaseFlow(ida:Int, idb:Int, inc:Int):RailNetwork = {
      val flowNow = flowFrom(ida, idb)
      val flow2 = flow.updated((ida, idb), flowNow + inc) // increase flow
      val flow3 = flow2.updated((idb, ida), -flow2((ida, idb))) // decrease flow in other dir
      copy(flow = flow3)
    }
    ...
}
```

The residual graph is implemented as a wrapper on the original graph, that recomputes capacities according to the specification, and filters out edges with 0 capacity.

```
case class Residual(nw:RailNetwork) extends FlowNetwork[FlowNode] {

  def node(id:Int) = nw.nodes(id)

  def edges = nw.edges

  val sourceId = nw.sourceId
  val sinkId = nw.sinkId

  def nodesOut(id:Int) = node(id).nodesOut.filter(id2 => capacity(id, id2) > 0)
  def nodesIn(id:Int) = node(id).nodesIn.filter(id2 => capacity(id2, id) > 0)

  def capacity(a:Int, b:Int):Int = nw.capacity(a,b) - nw.flowFrom(a,b)

  def bottleneck(path:Path):Int = bottleneckpairs(path2pairs(path))

  private def bottleneckpairs(lst:List[(Int,Int)]):Int =
    lst.map({ case (a,b) => capacity(a,b) }).filter(_ > 0).min

  def augment(path:Path, nw:RailNetwork):RailNetwork = {
    val pairs = path2pairs(path)
    val b = bottleneckpairs(pairs)
    pairs.foldLeft (nw) ((acc, e) => e match {
      case (from,to) => acc.increaseFlow(from, to, b)
    })
  }
}
```

Running time

Let n be the number of nodes in the graph and m be the number of edges. Let $C = \sum_{e \text{ out of } s} c_e$. Since each iteration of the algorithm finds an augmented path that augments the value of the flow with at least 1, the algorithm cannot run more iterations than C. The residual graph G_f has at most $2m$ edges. The edges are stored in adjacency lists, so looking up the edges of nodes is a constant time operations. Augmenting paths are found using breadth-first search which is $O(m + n) = O(m)$ since $m \geq n/2$. The capacities and flows are stored in hashmaps, so queries are expected constant time. `augment` takes $O(n)$ time as the path has at most $n-1$ edges. The Residual graph G_f is just a wrapper over the graph G that recomputes capacities (constant time) based on the original

capacities and the flow (all $O(1)$). All in all, since there are C iterations and each step takes $O(m)$ time, the total running time is $O(mC)$.