algoritmi bidirezionali

Filippo Magi

February 8, 2022

1 Ottimizzazione sugli ultimi livelli

1.1 FlowFordFulkerson

Algorithm 1 Ricerca del flusso massimo

Require: rete (G, u, s, t)
Ensure: valore del flusso massimo
1: $vuotoSouce \leftarrow s$
2: $vuotoSink \leftarrow t$
3: $(fmax, node) \leftarrow$ FirstBfs(G)
4: sendFlow(node,fMax)
5: **while** TRUE **do**
6: $(f, n) \leftarrow$ DoBfs(G, vuotoSource,vuotoSink)
7: **if** $f = 0$ **then**
8: **break**
9: **end if**
10: $fMax \leftarrow fMax + f$
11: $(vuotoSource, vuotoSink, b) \leftarrow sendFlow(n, f)$
12: **end while**
13: **return** $fMax$

1.2 sendFlow

1.3 FirstBfs

Algorithm 2 sendFlow

Require: rete (G, u, s, t), nodo n intermedio con le indicazioni riguardanti dove inviare il flusso, quantità di flusso da inviare$f$

Ensure: ultimo nodo trovato (quindi il più vicino a s o t dalla parte di source e di sink non più raggiungibile causa capacità residua = 0, booleano che mi informa se un arco selezionato $\forall e \in$percorso selezionato $u(e) > f \wedge f(e) > f$

n.flussoPassante $\leftarrow$ n.flussoPassante + n.flussoPassante

$momSource \leftarrow n$

$momSink \leftarrow n$

while $momSource \neq s$ **do**

$momSource$.previousEdge.addFlow(f)

if $u(momSource$.previousEdge) < 0 OR $f(momSource$.previousEdge)< 0 **then**

$vuotoSource \leftarrow momsource$

$momsource$.revert() {faccio tornare i dati a come erano prima dell'aggiornamento (fMax, valori di flusso passante e archi modificati)}

$fMax \leftarrow fMax - f$

return ($vuotoSource, null$,true)

else

if $u(momSource$.previousEdge) $= 0$ **then**

$momSource$.valid = false

$vuotoSource \leftarrow momSource$

break

end if

$momSource$.flussoPassante $\leftarrow$ $momSource$.flussoPassante-f

$momSource \leftarrow momSource$.previousNode

end if

end while

while $momSink \neq t$ **do**

$momSink$.nextEdge.addFlow(f)

if $u(momSink$.nextEdge)< 0 OR $f(momSink$.nextEdge)< 0 **then**

$vuotoSink \leftarrow momSink$

$momSink$.revert() {anche quello già fatto dalla parte di source va fatto tornare come era prima}

return ($null, vuotoSink, false$)

else

if $u(momSink$.nextEdge) $= 0$ **then**

$momSink$.valid $\leftarrow$ false

$vuotoSink \leftarrow momSink$

$fMax \leftarrow fMax - f$

end if

$momSink$.flussoPassante $\leftarrow$ $momSink$.flussoPassante $-f$

$momSink \leftarrow momSink$.nextNode

end if

end while

return ($vuotoSource, vuotoSink$,false)

Algorithm 3 BFS che inizialliza tutti i nodi e che indica se devono essere esplorati da source o da sink, oltre a trovare un percorso tra questi due

Require: rete (G, u, s, t)
Ensure: quantità di flusso inviabile,nodo intermedio tra i nodi esplorati da source e quelli esplorati da sink

```
codaSource ← coda di nodi
codaSource.enqueue(s)
codaSink ← coda di nodi
codaSink.enqueue(t)
codaEdgeSource ← coda di archi
codaEdgeSink ← coda di archi
returnvalue ← 0
while ¬codaSink.isEmpty ∨ ¬codaSource.isEmpty do
  if (¬codaSource.isEmpty ∧ codaEdgeSource.isEmpty) ∨ (codaSink.isEmpty ∧ codaEdgeSink.isEmpty then
    elementSource ← codaSource.dequeue()
    codaEdgeSource.enqueue(δ+(elementSource))
  end if
  if (¬codaSink.isEmpty ∧ codaEdgeSink.isEmpty) ∨ (codaSource.isEmpty ∧ codaEdgeSink.isEmpty) then
    elementSink ← codaSink.dequeueu
    codaEdgeSink.enqueue(δ−(elementSink)
  end if
  while ¬codaEdgeSource.isEmpty ∧ ¬codaEdgeSink.isEmpty do
    es ← codaEdgeSource.dequeue
    ps ← es.previousNode
    ns ← es.nextNode
    if elementSource = ps ∧ uf(es) > 0 then
      if ns.inFlow ≠ 0 then {è già stato visitato}
        if ns.SourceSide then {esplorato da source}
          continue
        else if returnvalue = 0 then
          returnvalue ← min(ps.flussoPassane, ns.flussoPassante, uf(es))
          ns.update(ps, es)
          returnNode ← ns
        end if
      else
        ns.upate(ps, es)
        codaSource.enqueue(ns)
      end if
    end if
```

```
      et ← codaEdgeSink.dequeue
      pt ← et.previousNode
      nt ← et.nextNode
      if elementSink = nt ∧ u_f(et) > 0 then
        if pt.flussoPassante ≠ 0 then
          if ¬pt.SourceSide then
            continue
          else if returnvalue = 0 then
            returnvalue ← min(pt.flussoPassante, nt.flussoPassante, f(et)
            nt.update(pt, et)
            returnNode ← nt
          end if
        else
          pt.update(et, nt)
          codaSink.enqueue(pt)
        end if
      end if
    end while
  end while
  return (returnvalue, returnNode)
```

1.4 DoBfs

Algorithm 4 DoBfs con ottimizzaione sugli ultimi livelli

Require: rete (G, u, s, t), $noCapSource, noCapSink$, cioè nodi,rispettivamente della parte sorgente e della parte destinazione, che non sono più raggiungili dal percorso deciso precedentemente

Ensure: valore del flusso inviabile, nodo appartenente LastSinkNodes, cioè tutti i nodi che sono intermedi che fanno da ponte tra le due ricerche.

1: $codaSource \leftarrow$ coda di nodi vuota
2: $codaSink \leftarrow$ coda di nodi vuota
3: **if** $noCapSource \neq$ null **then**
4: Repair($noCapSource$
5: **if** riesco a riparare $noCapSource$ **then**
6: **if** $noCapSink$ =null **then**
7: **for all** $n \in$ LastSinkNodes $|n.valid$ **do**
8: GetFlow($noCapSource, n$) {da n cerco di retrocedere verso noCapSource, aggiornando ricorsivamente le informazioni dei nodi in modo opportuno (sopratutto per quanto riguarda n)}
9: **if** percorso legale tra n e $noCapSource$ AND $\forall e|e \in$ percorso tra n e $noCapSource, u(e) > 0$ **then**
10: **return** (n.flussoPassante,n)
11: **end if**
12: **end for**
13: **else**
14: $sourceRepaired \leftarrow$ true
15: **end if**
16: **end if**
17: **if** $noCapSource = s$ **then**
18: $coda$.enqueue($noCapSource$)
19: **else if** $noCapSource \in$ LastSinkNodes **then**
20: $codaSource \leftarrow$ LastSourceNodes {nodi collegati ai nodi di LastSinkNodes}
21: **else**
22: $codaSource \leftarrow$ nodi esplorati da source con label = noCapSource.label-1
23: **for all** $n \in N(G)|$esplorati da source AND $n.label \geq noCapSource.label$ **do**
24: n.reset()
25: **end for**
26: **end if**
27: **end if**

```
28: if noCapSink ≠ null then
29:   Repair(noCapSink)
30:   if riesco a riparare noCapSink then
31:     sinkRepaired ← true
32:     if sourceRepaired OR noCapSource = null then
33:       for all n ∈ N(G)|n.valid AND n ∈LastSinkNodes  do
34:         if sourceRepaired then
35:           if da n posso ricorsivamente retrocedere verso noCapSource
    (GetFlow) then
36:             sourceFlow ← n.flussoPassante
37:           else
38:             continue
39:           end if
40:         else
41:           sourceFlow ← min(n.previousNode.flussoPassante,u(n.PreviousEdge)
    {nel caso in cui arco.reversed = true, devo controllare il flusso e non la
    capacità residua} )
42:         end if
43:         if sourceFlow > 0 AND n può retrocedere ricorsivamente verso
    noCapSink(Getflow) AND n.flussoPassante > 0  then
44:           retrun (min(n.flussoPassante, sourceFlow), n)
45:         end if
46:       end for
47:     end if
48:   end if
49:   if noCapSink = t then
50:     codaSink.enqueue(noCapSink)
51:   else
52:     codaSink ← nodi esplorati da sink con label = noCapSink.label-1
53:     for all n ∈ N(G)| esplorati da sink AND n.label ≥ noCapSink.label
    do
54:       n.reset()
55:     end for
56:   end if
57: end if
```

```
58: while ¬codaSink.isEmpty OR ¬codaSource.isEmpty do
59:   if ¬codaSource.isEmpty AND (noCapSource ≠ null OR ¬sourceRepaired then
60:     element ← codaSource.dequeue()
61:     if ¬element.sourceSide OR ¬element.valid  then
62:       continue
63:     end if
64:     for all  edge ∈ element.Edges do
65:       p ← edge.previousNode
66:       n ← edge.nextNode
67:       if element = pAND u(edge) > 0 then
68:         if n.visited then
69:           if n.sourceside (esplorato da source) then
70:             continue
71:           else{in questo caso ho le due parti che si incontrano}
72:             f ← min(n.flussoPassante, p.flussoPassante, u(edge))
73:             if f = 0 then
74:               continue
75:             end if
76:             n.update(p, edge)
77:             edge.reversed ← false
78:             LastNodesSinkSide.add(n) {di conseguenza inserisco tutti i
    nodi collegati direttamente a n che fanno parte di SourceSide in LastN-
    odesSourceSide}
79:             return (f, n)
80:           end if
81:         end if
82:         if ¬n.sourceSide AND n ≠ t then
83:           sinkRepaired ← false
84:           for all  node ∈ N(G)|¬node.sourceSide AND node.label =
    (n.label − 1) do
85:             codaSink.enqueue(node)
86:           end for
87:           for all node ∈ N(G)|¬node.sourceSide AND node.label ≥
    n.label do
88:             node.reset()
89:           end for
90:           continue
91:         end if
92:         n.update(p, edge)
93:         codaSource.enqueue(n)
```

```
94:         else if element = n AND f(edge) > 0 then
95:           if p.visited then
96:             if p.sourceside then
97:               continue
98:             else
99:               f ← min(n.flussoPassante, p.flussoPassante, f(edge))
100:               if f = 0 then
101:                 continue
102:               end if
103:               p.update(n,edge)
104:               edge.reversed ← true
105:               return (f, p)
106:             end if
107:           end if
108:           if ¬p.sourceSide AND p ≠ t then
109:             sinkRepaired ← false
110:             for all node ∈ V(G)|¬node.sourceNodeANDnode.label =
   (p.label − 1) do
111:               codaSink.enqueue(node)
112:             end for
113:             for all node ∈ V(G)|¬node.sourceSide AND node.label ≥
   p.label do
114:               node.reset()
115:             end for
116:             continue
117:           end if
118:           p.update(n, edge)
119:           edge.reversed ← true
120:           codaSource.enqueue(p)
121:         end if
122:       end for
123:     end if
```

```
124:  if ¬codaSink.isEmptyAND (noCapSink ≠ null OR ¬sinkRepaired) then
125:    element ← codaSink.dequeue()
126:    if element.sourceSide OR ¬element.valid then
127:      continue
128:    end if
129:    for all edge ∈ element.Edges do
130:      p ← edge.previousNode
131:      n ← edge.nextNode
132:      if element = n AND u(edge) > 0 then
133:        if p.visited then
134:          if ¬p.sourceSide then
135:            continue
136:          else
137:            if sourceRepaired AND n può retrocedere ricorsivamente verso noCapSink(GetFlow) AND n.flussoPassante> 0 then
138:              f ← min(n.nextNode.flussoPassante, n.flussoPassante)
139:              f ← min(f, n.PreviousNode.flussoPassante, u(n.nextEdge), u(n.previuosEdge))
                  {qui se edge.reversed = true va considerato il flusso f e non la capacità residua u}
140:              if f > 0 then
141:                return f
142:              end if
143:            end if
144:            f ← min(n.flussoPassante, p.flussoPassante, u(edge))
145:            if f = 0 then
146:              continue
147:            end if
148:            n.update(p, edge)
149:            edge.reversed ←false
150:            return (f, n)
151:          end if
152:        end if
153:        if p.sourceSide AND noCapSink ≠ t then
154:          continue
155:        end if
156:        p.update(n, edge)
157:        edge.reversed ← false
158:        codaSink.enqueue(p)
```

```
159:        else if element = pAND f(edge) > 0 then
160:          if p.visited then
161:            if ¬p.sourceSide then
162:              continue
163:            else
164:              if sourceRepaired AND da p riesco a raggiungere noCap-
    Source(GetFlow) then
165:                return (p.flussoPassante, p)
166:              end if
167:              f ← min(p.flussoPassante, n.flussoPassante, f(edge))
168:              if f = 0 then
169:                continue
170:              end if
171:              p.update(n, edge)
172:              edge.reversed ← true
173:              return (f, p)
174:            end if
175:          end if
176:          if n.sourceSide AND n ≠ t then
177:            continue
178:          end if
179:          n.update(p, edge)
180:          edge.reversed ← true
181:          codaSink.enqueue(n)
182:        end if
183:      end for
184:    end if
185: end while
186: return (0, null)
```