algoritmi bidirezionali

Filippo Magi

February 19, 2022

1 Ottimizzazione sugli ultimi livelli

1.1 FlowFordFulkerson

1.2 DoBfs

Algorithm 1 Ricerca del flusso massimo

Require: rete (G, u, s, t)
Ensure: valore del flusso massimo

1: $vuotiSouce \leftarrow$ pila di nodi
2: $vuotiSink \leftarrow$ pila di nodi
3: $fMax \leftarrow 0$
4: $vuotiSouce.push(s)$
5: $vuotiSink.push(t)$
6: **while** TRUE **do**
7: $(f, n) \leftarrow$ DoBfs(G, vuotiSource,vuotiSink)
8: **if** $f = 0$ **then**
9: **break**
10: **end if**
11: $vuotiSouce$.Clear()
12: $vuotiSink$.Clear()
13: n.flussoPassante $\leftarrow n$.flussoPassante + f
14: $momSource \leftarrow n$
15: $momSink \leftarrow n$
16: **while** $momSource \neq s$ **do**
17: $momSource$.previousEdge.addFlow(f)
18: **if** $u_f(momSource$.previousEdge $) < 0 \vee f(momSource$.previousEdge $)< 0$ **then**
19: $vuotiSource$.Clear()
20: $flowError \leftarrow$GetFlow(s, n).flussoPassante
21: $mom \leftarrow n$
22: **while** $mom \neq momsource$ **do**
23: mom.flussoPassante $\leftarrow mom$.flussoPassante - flowError
24: mom.PreviousEdge.addFlow($flowError$)
25: $mom \leftarrow mom$.previousNode
26: **end while**
27: $vuotiSource$.push(momSource)
28: $momSource.valid \leftarrow$ false
29: $f \leftarrow f + flowError$
30: **else if** $u_f(momSource$.previousEdge$) = 0$ **then**
31: $momSource.valid \leftarrow$ false
32: $vuotiSource.push(momSource)$
33: **end if**
34: $momSource$.flussoPassante $\leftarrow momSource$.flussoPassante-f
35: $momSource \leftarrow momSource$.previousNode
36: **end while**

```
37:   while momSink ≠ t do
38:     momSink.nextEdge.addFlow(f)
39:     if u_f(momSink.nextEdge)< 0 ∨ f(momSink.nextEdge)< 0 then
40:       vuotiSink.Clear()
41:       flowError ←GetFlow(t, n).flussoPassante
42:       mom ← n
43:       while mom ≠ momsink do
44:         mom.flussoPassante ← mom.flussoPassante - flowError
45:         mom.nextEdge.addFlow(flowError)
46:         mom ← mom.nextNode
47:       end while
48:       mom ← n
49:       while mom ≠ s do
50:         mom.flussoPassante ← mom.flussoPassante - flowError
51:         mom.PreviousEdge.addFlow(flowError)
52:         mom ← mom.previousNode
53:       end while
54:       vuotiSink.Push(momSink)
55:       momSink.valid ← false
56:       f ← f + flowError
57:     else if u_f(momSink.nextEdge) = 0 then
58:       momSink.valid ← false
59:       vuotiSource.push(momSource)
60:     end if
61:     momSink.flussoPassante ← momSink.flussoPassante −f
62:     momSink ← momSink.nextNode
63:   end while
64:   fMax ← fMax + f
65: end while
66: return fMax
```

Algorithm 2 DoBfs con ottimizzazione sugli ultimi livelli

Require: rete (G, u, s, t), $noCapsSource, noCapsSink$, cioè pile di nodi contenenti nodi non più raggiungibili attraverso il cammino trovato

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 ← coda di nodi vuota
2:  codaSink ← coda di nodi vuota
3:  codaEdgeSink ← coda di archi vuota
4:  codaEdgeSource ← coda di archi vuota
5:  if ¬noCapsSource.isEmpty then
6:     p ← null
7:     repaired ← true
8:     while ¬noCapsSource.isEmpty do
9:        noCapSource ← noCapsSource.pop()
10:       GetFlow(p, noCapSource)
11:       p ← noCapSource
12:       Repair(noCapSource)
13:       if non riesco a riparare noCapSource then
14:          noCapsSource.Push(noCapSource)
15:          repaired ← false
16:          break
17:       end if
18:    end while
```

```
19:   if ¬noCapsSink.isEmpty ∧ repaired then
20:     for all n ∈ LastSinkNodes |n.valid  do
21:       GetFlow(noCapSource, n) {da n cerco di retrocedere verso noCapSource, aggiornando ricorsivamente le informazioni dei nodi in modo opportuno (sopratutto per quanto riguarda n)}
22:       if GetFlow ha trovato un percorso∧n.flussoPassante ≠ 0 then
23:         if edge.reversed then
24:           return (min(n.flussoPassante, f(edge)), n)
25:         else
26:           return (min(n.flussoPassante, u_f(edge)), n)
27:         end if
28:       end if
29:     end for
30:   end if
31:   if ¬repaired then
32:     if noCapSource = s then
33:       codaSource.enqueue(noCapSource)
34:     else if noCapSource ∈ LastSinkNodes then
35:       codaSource ← LastSourceNodes {nodi collegati ai nodi di LastSinkNodes}
36:     else
37:       for all n ∈ V(G)|n.sourceSide ∧ n.label + 1 = noCapSource.label do
38:         codaSource.enqueue(n)
39:       end for
40:       for all n ∈ V(G)|n.SourceSide ∧ n.label ≥ noCapSource.label do
41:         n.reset()
42:       end for
43:     end if
44:   end if
45: end if
```

```
46: if ¬noCapsSink.isEmpty then
47:   repaired ← true
48:   p ← null
49:   while ¬noCapsSink.isEmpty do
50:     noCapsSink ← noCapsSink.pop()
51:     GetFlow(p, noCapSink)
52:     p ← noCapSink
53:     Repair(noCapSink)
54:     if non riesco a riparare noCapSink then
55:       noCapsSink.push(p)
56:       repaired ← false
57:       break
58:     end if
59:   end while
60:   if repaired ∧ noCapsSource.isEmpty then
61:     for all n ∈ LastSinkNodes|n.valid do {nodo di confine valido}
62:       if n.previousEdge.reversed then
63:         sourceFlow ← min(n.previousNode.inFlow, f(n.previousEdge))
64:       else
65:         sourceFlow ← min(n.previousNode.inFlow, u_f(n.previousEdge))
66:       end if
67:       GetFlow(p, n)
68:       if è stato trovato un percorso tra p ed n ∧n.flussoPassante ≠ 0 ∧
    sourceFlow > 0 then
69:         return (min(n.flussoPassante, sourceFlow), n)
70:       end if
71:     end for
72:   end if
73:   if ¬repaired then
74:     if noCapSink = t then
75:       codaSink.enqueue(noCapSink)
76:     else
77:       for all n ∈ V(G)|n.label + 1 = noCapSink.label do
78:         codaSink.enqueue(n)
79:       end for
80:       for all n ∈ N(G)|¬n.sourceSide ∧ n.label ≥ noCapSink.label do
81:         n.reset()
82:       end for
83:     end if
84:   end if
85: end if
```

86: **while** $\neg codaSink.isEmpty \vee \neg codaSource.isEmpty$ **do**
87: **if** $\neg codaSource.isEmpty \wedge (codaEdgeSource.isEmpty \vee (codaEdgeSink.isEmpty \wedge codaSink.isEmpty \wedge \neg noCapsSink.isEmpty))$ **then**
88: $elementSource \leftarrow codaSource$.dequeue()
89: **if** $\neg elementSource.sourceSide \wedge \neg elementSource.valid \wedge elementSource.flussoPassante = 0$ **then**
90: **continue**
91: **end if**
92: **for all** $e \in \delta(elementSource)|$ il nodo collegato potrà essere esplroato **do** {dato l'arco x, $(x.NextNode = elementSink \wedge x.Capacity > 0 \wedge (x.PreviousNode.InFlow = 0 \vee x.PreviousNode.SourceSide)) \vee (x.PreviousNode = elementSink \wedge x.Flow > 0 \wedge (x.NextNode.InFlow = 0 \vee x.NextNode.SourceSide))$}
93: $codaEdgeSource$.enqueue(e)
94: **end for**
95: **end if**
96: **if** $\neg codaSink.isEmpty \wedge (codaEdgeSink.isEmpty \vee (codaEdgeSource.isEmpty \wedge codaSource.isEmpty \wedge \neg noCapsSource.isEmpty))$ **then**
97: $elementSink \leftarrow codaSink$.dequeue()
98: **if** $\neg elementSink.sourceSide \wedge \neg elementSink.valid \wedge elementSink.flussoPassante = 0$ **then**
99: **continue**
100: **end if**
101: **for all** $e \in \delta(elementSink)|$ il nodo collegato potrà essere esplorato **do** {dato l'arco $x, (x.NextNode = elementSink \wedge x.Capacity > 0 \wedge (x.PreviousNode.InFlow = 0 \vee x.PreviousNode.SourceSide)) \vee (x.PreviousNode = elementSink \wedge x.Flow > 0 \wedge (x.NextNode.InFlow = 0 \vee x.NextNode.SourceSide))$}
102: $codaEdgeSink$.enqueue(e)
103: **end for**
104: **end if**

```
105:  while (¬codaEdgeSource.isEmpty ∨ noCapsSource.isEmpty) ∧
      (¬codaEdgeSink.isEmpty ∨ noCapSink.isEmpty) do
106:    if ¬codaEdgeSource.isEmpty then
107:      e ← codaEdgeSource.dequeue()
108:      p ← e.previousNode
109:      n ← e.nextNode
110:      if elementSource = p ∧ u_f(e) > 0 then
111:        if n.flussoPassante ≠ 0 then
112:          if ¬n.sourceSide then
113:            f ← min(n.flussoPassante, p.flussoPassante, u_f(e))
114:            if f ≠ 0 then
115:              continue
116:            end if
117:            n.update(p, edge)
118:            addLast(n)
119:            e.reversed ←false
120:            return (f, n)
121:          end if
122:        else
123:          n.update(p, edge)
124:          e.reversed ←false
125:          codaSource.enqueue(n)
126:        end if
127:      else if elementSource = n ∧ f(e) > 0 then
128:        if p.flussoPassante ≠ 0 then
129:          if ¬p.sourceSide then
130:            f ← min(p.flussoPassante, n.flussoPassante, f(e))
131:            if f = 0 then
132:              continue
133:            end if
134:            p.upate(n, e)
135:            addLast(p)
136:            e.reversed ← true
137:            return (f, p)
138:          end if
139:        else
140:          p.update(n, e)
141:          e.reversed ← true
142:          codasource.enqueue(p)
143:        end if
144:      end if
145:    end if
```

```
146:      if ¬codaEdgeSink.isEmpty then
147:        e ← codaEdgeSink.dequeue()
148:        p ← e.previousNode
149:        n ← e.nextNode
150:        if elementSink = n ∧ u_f(e) > 0 then
151:          if p.flussoPassante ≠ 0 then
152:            if ¬p.sourceSide then continue
153:            else
154:              f ← min(p.flussoPassante, n.flussoPassante, u_f(e))
155:              if f = 0 then
156:                continue
157:              end if
158:              n.update(p, e)
159:              e.reversed ← true
160:              addLast(n)
161:              return (f, n)
162:            end if
163:          end if
164:          p.update(n, e)
165:          e.reversed ← true
166:          codaSink.enqueue(p)
167:        else if elementSink = p ∧ f(e) > 0 then
168:          if n.flussoPassante ≠ 0 then
169:            if ¬n.sourceSide then continue
170:            else
171:              f ← min(p.flussoPassante, n.flussoPassante, f(e))
172:              if f = 0 then
173:                continue
174:              end if
175:              p.update(n, e)
176:              e.reversed ← true
177:              addLast(p)
178:              return (f, p)
179:            end if
180:          end if
181:          n.update(p, e)
182:          e.reversed ← true
183:          codaSink.enqueue(n)
184:        end if
185:      end if
186:    end while
187: end while
188: return (0, null)
```