algoritmi bidirezionali

Filippo Magi

February 11, 2022

1 Senza alcuna ottimizzazione

1.1 FlowFordFulkerson

1.2 DoBfs

Algorithm 1 Ricerca del flusso massimo

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

1: $fMax \leftarrow 0$
2: $vuotoSource \leftarrow$ true
3: $vuotoSource \leftarrow$ true
4: **while** TRUE **do**
5: $(f, nodo) \leftarrow$ DoBfs(G,vuotoSource,vuotoSink)
6: **if** $f = 0$ **then**
7: **break**
8: **end if**
9: $vuotoSource \leftarrow$ false
10: $vuotoSource \leftarrow$ false
11: $fMax \leftarrow fMax + f$
12: $mom \leftarrow n$
13: **while** $n \neq s$ **do**
14: n.PreviousEdge.AddFlow(f)
15: **if** $u(n$.PreviousEdge$)= 0$ **then**
16: $vuotoSource \leftarrow$ true
17: **end if**
18: n.update(f) {n.InFlow -=f}
19: $n \leftarrow n$.previousNode
20: **end while**
21: **while** $mom \neq t$ **do**
22: n.nextEdge.addFlow(f)
23: **if** $e(n$.nextEdge$) = 0$ **then**
24: $vuotoSink \leftarrow$ true
25: **end if**
26: n.update(f) $\{n.InFlow- = f\}$
27: $n \leftarrow n$.nextNode
28: **end while**
29: **end while**
30: **return** $fMax$

Algorithm 2 DoBfs : Ricerca un path tra s e x[], e da x[] a t, dove t[] sono i nodi intermedi dove si incontrano i due path

Require: rete (G, u, s, t), *sourceSide* e *sinkSide*, che sono dei booleani che chiariscono in quale parte si dovrà operare

Ensure: valore del flusso inviabile, nodo intermedio, cioè che tiene in memoria sia il nodo successivo, sia il nodo precedente

```
1: codaSource ← coda di nodi
2: codaSink ← coda di nodi
3: if sourceSide then
4:    for all n ∈ V(G)|n è stato esplorato da s do
5:       n.Reset()
6:    end for
7:    codaSource.enqueue(s)
8: end if
9: if sinkSide then
10:   for all n ∈ V(G)|n è stato esplorato da t do
11:      n.Reset()
12:   end for
13:   codaSink.enqueue(t)
14: end if
15: {per motivi prestazioni si controlla nel codice se si hanno sia sinkSide sia
    sourceSide positivi per non analizzare tutti i nodi 2 volte}
16: while ¬codaSource.isEmpty OR ¬codaSink.iSEmpty  do
17:   if codaSource.isEmpty then
18:      element ← codaSource.dequeue()
19:      for all edge ∈ n.Edges do
20:         p ← edge.previousNode
21:         n ← edge.nextNode
22:         if element = p AND u(edge) > 0 then
23:            if n.visited then
24:               if n.sourceSide then
25:                  continue
26:               else
27:                  f ← min(u(edge), p.flussoPassante, n.flussoPassante)
28:                  if  f = 0 then
29:                     continue
30:                  end if
31:                  n.update(p, edge, n)
32:                  edge.reversed = false
33:                  return (f, n)
34:               end if
35:            end if
36:            n.update(p, edge)
37:            edge.reversed = false
38:            codaSource.enqueue(n)
39:         end if
```

```
40:         if element = n AND f(edge) > 0 then
41:           if p.visited then
42:             if p.sourceSide then
43:               continue
44:             else
45:               f ← min(n.flussoPassante, p.flussoPassante, f(edge)
46:               if f = 0 then
47:                 continue
48:               end if
49:               p.update(n, edge, p)
50:               edge.reversed = true
51:               return (f, p)
52:             end if
53:           end if
54:           p.update(n, edge)
55:           edge.reversed = true
56:           codaSource.enqueue(p)
57:         end if
58:       end for
59:     end if
60:     if ¬codaSink.isEmpty then
61:       element ← codaSink.dequeue()
62:       for all edge ∈ element.Edges do
63:         p ← edge.previousNode
64:         n ← n.nextNode
65:         if element = n AND u(edge) > 0 then
66:           if p.visited then
67:             if ¬p.sourceSide then
68:               continue
69:             else
70:               f ← min(p.flussoPassante, u(edge), n.flussoPassante)
71:               if f = 0 then
72:                 continue
73:               end if
74:               n.update(p, edge, n)
75:               edge.Reversed = false
76:               return (f, n)
77:             end if
78:           end if
79:           p.update(n, edge)
80:           edge.Reversed = false
81:           codaSink.enqueue(p)
82:         end if
```

```
83:         if element = p AND f(edge) > 0 then
84:           if n.visited then
85:             if ¬n.sourceSide then
86:               continue
87:             else
88:               f ← min(n.flussoPassante, p.flussoPassante, f(edge)
89:               if f = 0 then
90:                 continue
91:               end if
92:               p.update(n, edge, p)
93:               edge.reversed = true
94:               return (f, p)
95:             end if
96:           end if
97:           n.update(p, edge)
98:           edge.reversed = true
99:           codaSink.enqueue(n)
100:        end if
101:      end for
102:    end if
103: end while
104: return (0, null)
```