1 Ottimizzazione con propagazione della malattia

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() {elimino il contenuto della pila, in maniera da averla vuota}
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) {invio una quantità di flusso f nell'arco indicato}
18: **if** $u_f(momSource$.previousEdge $) < 0 \vee f(momSource$.previousEdge $) < 0$ **then**
19: $vuotiSource$.Clear()
20: $flowError \leftarrow$GetFlow(s, n).flussoPassante {si fa notare che il valore di flowError deve essere negativo}
21: $mom \leftarrow n$
22: **while** $mom \neq momsource.previousNode$ **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.nextNode 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

Require: rete (G, u, s, t), pila *noCapsSource* con i nodi non più raggiungibili dall'ultima iterazione esplorati da Source, pila *noCapsSink*, con i nodi non più raggiungibili dall'ultima iterazione esplorati da Sink

Ensure: valore del flusso inviabile, nodo intermedio (o di confine) con le informazioni di indirizzamento per l'invio del flussoPassante

1: {per rendere delle istruzioni leggibili abbrevierò flussoPassante in fP}
2: $codaSource \leftarrow$ coda di nodi vuota
3: $codaSink \leftarrow$ coda di nodi vuota
4: $malati \leftarrow$ coda di nodi vuota
5: $sourceRepaired \leftarrow noCapsSource.isEmpty$
6: $sinkRepaired \leftarrow noCapsSink.isEmpty$
7: {analizzo tutti i nodi malati di source, cercando di ripararli, nel caso non riesca li inserisco nella coda di malati. Nel caso sia riuscito a ripararli tutti, cerco un percorso tra l'ultimo nodo riparato e i nodi di confine}
8: **if** $\neg noCapsSource.isEmpty$ **then**
9: $repaired \leftarrow true$
10: $firstNoCapSource \leftarrow$ null
11: $p \leftarrow$ null
12: **while** $\neg noCapsSource.isEmpty$ **do**
13: $noCapSource \leftarrow noCapsSource$.Pop()
14: GetFlow($p, noCapSource$) {aggiorna ricorsivamente inFlow dal nodo p fino al nodo noCapSource}
15: $p \leftarrow noCapSource$
16: **if** $\neg$ RepairNode($G, noCapSource$,false) **then**
17: $malati$.enqueue($noCapSource$)
18: $repaired \leftarrow$ false
19: **if** $firstNoCapSource =$ null **then**
20: $firstNoCapSource \leftarrow noCapSource$
21: **end if**
22: **end if**
23: **end while**
24: **if** $repaired \wedge sinkRepaired$ **then**
25: **for all** $n \in V(G) | n.previousEdge \neq null \wedge n.nextEdge \neq null \wedge sourceValid \wedge sinkValid$ **do**
26: GetFlow(p, n)
27: **if** p è raggiungile da $n \wedge n.flussoPassante > 0$ **then**
28: **if** $n.nextEdge.reversed$ **then**
29: **return** $(\min(n.fP, n.nextNode.fP, f(n.nextEdge)), n)$
30: **else**
31: **return** $(\min(n.fP, n.nextNode.fP, u_f(n.nextEdge)), n)$
32: **end if**
33: **end if**
34: **end for**
35: **end if**

1.3 SourceSickPropagation

{eseguo propagazione della malattia in source,se trova un nodo di confine e lo ripara, lo restituisco (con sinkRepaired = true), altrimenti cerco tra i nodi di confine un nodo valido e che abbia un percorso tra s e il nodo designato}
$malato \leftarrow$ null
while $\neg malati.isEmpty$ **do**
  $m \leftarrow$ SourceSickPropagation($G, malati$.dequeue(), $codaSource$)
  if $m \neq$null $\wedge malato \neq null$ **then**
    $malato \leftarrow m$
  end if
end while
if $malato \neq null \wedge sinkRepaired$ **then**
  if $malato.nextEdge.reversed$ **then**
    return $(\min(f(malato.nextEdge), malato.nextNode.fP, malato.fP), malato)$
  else
    return $(\min(u_f(malato.nextEdge), malato.nextNode.fP, malato.fP), malato)$
  end if
end if
if $sinkRepaired$ **then**
  for all $n \in V(G) | n.sourceValid \wedge n.nextNode \neq null \wedge n.nextNode.flussoPassante > 0 \wedge n.previousNode \neq null \wedge n.nextEdge.reversed \wedge f(n.nextEdge) > 0$ **do**
    GetFlow(s, n)
    if ho trovato un percorso tra s e $n \wedge n.flussoPassante > 0$ **then**
      return $(\min(n.flussoPassante, f(n.nextEdge), n.nextNode.flussoPassante), n)$
    end if
    $n.flussoPassante \leftarrow \min(f(n.nextEdge), n.nextNode.flussoPassante)$
  end for
  for all $n \in V(G) | n.sourceValid \wedge n.nextNode \neq null \wedge n.nextNode.flussoPassante > 0 \wedge n.previousNode \neq null \wedge \neg n.nextEdge.reversed \wedge u_f(n.nextEdge) > 0$ **do**
    GetFlow(s, n)
    if ho trovato un percorso tra s e $n \wedge n.flussoPassante > 0$ **then**
      return $(\min(n.flussoPassante, f_f(n.nextEdge), n.nextNode.flussoPassante), n)$
    end if
    $n.flussoPassante \leftarrow \min(u_f(n.nextEdge), n.nextNode.flussoPassante)$
  end for
end if
$sourceRepaired \leftarrow repaired$ {inizializzazione coda di source}
if $\neg repaired \wedge codaSource.isEmpty$ **then**
  if $firstNoCapSource = s$ **then**
    $codaSource$.enqueue($firstNoCapSource$)
  else if $\neg firstNoCapSource.SourceSide$ **then**
    for all $n \in V(G) | n.previousNode \neq null \wedge n.nextNode \neq null$ **do**
      $codaSource$.enqueue(n)
    end for

```
75:       else
76:         for all n ∈ V(G)|n.sourceSide ∧ n.label = firstNoCapSource.label − 1 do
77:           codaSource.enqueue(n)
78:         end for
79:         for all n ∈ V(G)|n.sourceside ∧ n.label ≥ firstNoCapSource.label do
80:           n.flussoPassante ← 0 {indica che il nodo dovrà essere esplroato}
81:         end for
82:       end if
83:     end if
84: end if
85: { {riparazione nodi e, nel caso riesco a ripararli tutti, invio il flusso trovato}}
86: if ¬noCapsSink.isEmpty then
87:     repaired ← true
88:     firstNoCapSink ← null
89:     p ← null
90:     while ¬noCapsSink.isEmpty do
91:       noCapSink ← noCapsSink.Pop()
92:       GetFlow(p, noCapSink)
93:       p ← noCapSink
94:       if RepairNode(G, noCapSink,true) then
95:         malati.enqueue(noCapSink)
96:         repaired ← false
97:         if firstNoCapSink = null then
98:           firstNoCapSink ← noCapSink
99:         end if
100:      end if
101:    end while
102:    if repaired ∧ sourceRepaired then
103:      for all n ∈ V(G)|n.sourceValid ∧ n.sinkValid ∧ n.previousNode ≠ null ∧ n.nextNode ≠ null do
104:        if n.previousEdge.reversed then
105:          sourceFlow ← min(n.previousNode.fP, f(n.previousEdge))
106:        else
107:          sourceFlow ← min(n.previousNode.fP, u_f(n.previousEdge))
108:        end if
109:        GetFlow(p, n)
110:        if p è raggiungibile da n ∧ sourceFlow > 0 ∧ n.fP > 0 then
111:          return (min(n.flussoPassante, sourceFlow), n)
112:        end if
113:      end for
114:    end if
```

```
115:  {propagazione della malattia, inizializzazione della coda di sink}
116:  malato ← null
117:  while ¬malati.isEmpty do
118:    mom ←SinkSickPropagation(G, malati.dequeue(), codaSink)
119:    if mom ≠ null ∧ malato = null then
120:      malato ← mom
121:    end if
122:  end while
123:  if malato ≠ null then
124:    if malato.previousEdge.reversed then
125:      return (min(malato.fP, f(malato.previousEdge), malato.previousNode.fP), malato)
126:    else
127:      return (min(malato.fP, u_f(malato.previousEdge), malato.previousNode.fP), malato)
128:    end if
129:  end if
130:  for all n ∈ V(G)|n.previousNode ≠ null ∧ n.nextNode ≠
      null ∧ n.sinkValid ∧ n.previousNode.flussoPassante > 0 ∧
      n.previousEdge.reversed ∧ f(n.previousEdge) > 0 do
131:    GetFlow(t, n)
132:    if t è raggiungibile da n ∧ n.flussoPassante > 0 then
133:      return (min(n.flussoPassante, f(n.previousEdge), n.previousNode.flussoPassante), n)
134:    end if
135:  end for
136:  for all n ∈ V(G)|n.previousNode ≠ null ∧ n.nextNode ≠
      null ∧ n.sinkValid ∧ n.previousNode.flussoPassante > 0 ∧
      ¬n.previousEdge.reversed ∧ u_f(n.previousEdge) > 0 do
137:    GetFlow(t, n)
138:    if t è raggiungibile da n ∧ n.flussoPassante > 0 then
139:      return (min(n.flussoPassante, u_f(n.previousEdge), n.previousNode.flussoPassante), n)
140:    end if
141:  end for
142:  sinkRepaired ← repaired
143:  if ¬repaired ∧ codaSink.isEmpty then
144:    if firstNoCapSink = t then
145:      codaSink.enqueue(firstNoCapSink)
146:    else
147:      for all n ∈ V(G)|¬n.sourceside∧n.label = firstNoCapSink.label−
      1 do
148:        codaSink.enqueue(n)
149:      end for
150:      for all n ∈ V(G)|¬n.sourceside ∧ n.label ≥ firstNoCapSink.label
      do
151:        n.flussoPassante ← 0
152:      end for
153:    end if
```

```
154:    end if
155: end if
156: {esplorazione dei nodi}
157: while ¬codaSink.isEmpty ∧ ¬codaSource.isEmpty do
158:    if ¬codaSource.isEmpty ∧ ¬sourceRepaired then
159:       element ← codaSource.dequeue()
160:       if ¬element.sourceSide ∨ ¬element.sourceValid ∨ element.flussoPassante = 0 then
161:          continue
162:       end if
163:       for all e ∈ δ(element) do
164:          p ← e.previousNode
165:          n ← e.nextNode
166:          if element = p ∧ u_f(e) > 0 then
167:             if n.sourceSide ∧ n.flussoPassante = 0 then
168:                e.reversed ← false
169:                n.update(p, e) {aggiorno flussoPassante,label,indirizzamento e la validità della parte esplorata}
170:                codaSource.enqueue(n)
171:             else if n.flussoPassante ≠ 0 ∧ ¬n.sourceSide ∧ n.sinkValid then
172:                f ← min(n.flussoPassante, p.flussoPassante, u_f(e))
173:                if f = 0 then
174:                   continue
175:                end if
176:                e.reversed ← false
177:                n.updatePath(p, e) {aggiorno indirizzamento, sourcevalid, inserisco il nodo indicato in LastNodesSinkSide, e i nodi a lui collegati esplorati da source in LastNodesSourceSide }
178:                return (f, n)
179:             else if n.flussoPassante = 0 ∧ ¬n.sourceSide ∧ sinkRepaired then
180:                {nodo non valido dalla parte di sink, cerco di recuperare ripararlo, ma non essendo valido non ha una label a cui fare riferiemento}
181:                for all edge ∈ δ(n) do
182:                   if edge.previousNode = n ∧ edge.nextNode.flussoPassante > 0∧¬edge.nextNode.sourceSide∧u_f(edge) then
183:                      edge.reversed ← false
184:                      n.update(edge.nextNode, edge)
185:                      break
```

```
186:                else if edge.nextNode = n ∧ edge.previousNode.flussoPassante > 0∧¬edge.previousNode.sourceSide∧ f(edge) > 0 then
187:                    edge.reversed ← true
188:                    n.update(edge.previousNode, edge)
189:                    break
190:                end if
191:            end for
192:            if n.flussoPassante > 0 then
193:                f ← min(n.flussoPassante, p.flussoPassante, u_f(e))
194:                if f = 0 then
195:                    continue
196:                end if
197:                e.reversed ← false
198:                n.updatePath(p, e) {aggiorno indirizzamento e sourceValid}
199:                return (f, n)
200:            end if
201:            {dato che non è possibile ripararlo, cerco l'ultimo nodo valido nel path che collega a t e obbligo la parte di sink a svolgere una nuova ricerca}
202:            while mom ≠ t do
203:                if mom.nextEdge.reversed then
204:                    if f(mom.nextEdge) = 0 then
205:                        malato = mom
206:                    end if
207:                else
208:                    if u_f(mom.nextEdge) = 0 then
209:                        malato = mom
210:                    end if
211:                end if
212:            end while
213:            sinkRepaired ← false
214:            for all node ∈ V(G)|node.label = malato.label − 1 do
215:                codaSink.enqueue(node)
216:            end for
217:            for all node ∈ V(G)|node.label ≥ malato.label do
218:                node.flussoPassante ← 0
219:            end for
220:        end if
```

221: {stesse valutazione prima fatte, parte che cerca nel grafo dei residui gli archi "inversi", cioè dove riduco il flusso e aumento la capacità}
222: **else if** $element = n \wedge f(e) > 0$ **then**
223: **if** $p.sourceSide \wedge p.flussoPassante = 0$ **then**
224: $e.reversed \leftarrow$ true
225: p.$update$(n,e)
226: $codaSource$.enqueue(p)
227: **else if** $p.flussoPassante \neq 0 \wedge \neg p.sourceSide \wedge p.sinkValid$ **then**
228: $f \leftarrow \min(p.flussoPassante, n.flussoPassante, f(e))$
229: **if** $f = 0$ **then**
230: **continue**
231: **end if**
232: $e.reversed \leftarrow$ true
233: p.updatePath(n, e)
234: **return** (f, p)
235: **else if** $p.flussoPassante = 0 \wedge \neg p.sourceSide \wedge sinkRepaired$ **then**
236: **for all** $edgein\delta(p)$ **do**
237: **if** $edge.previousNode = p \wedge edge.nextNode.flussoPassante > 0 \wedge \neg edge.nextNode.sourceSide \wedge u_f(edge) > 0$ **then**
238: $edge.reversed \leftarrow$ false
239: p.update($edge.nextNode, edge$)
240: **break**
241: **else if** $edge.nextNode = p \wedge edge.previousNode.flussoPassante > 0 \wedge \neg edge.previousNode.sourceSide \wedge f(edge)$ **then**
242: $edge.reversed \leftarrow true$
243: p.update($edge.previousNode, edge$)
244: **break**
245: **end if**
246: **end for**
247: **if** $p.flussoPassante > 0$ **then**
248: $e.reversed \leftarrow$ true
249: p.updatePath(n, e)
250: **return** (f, p)
251: **end if**
252: $mom \leftarrow p$
253: $malato \leftarrow null$

```
254:            while mom ≠ t do
255:              if mom.nextEdge.reversed then
256:                if f(mom.nextEdge) = 0 then
257:                  malato ← mom
258:                end if
259:              else
260:                if u_f(mom.nextEdge) = 0 then
261:                  malato ← mom
262:                end if
263:              end if
264:              mom ← mom.nextNode
265:            end while
266:            sinkRepaired ←false
267:            for all node ∈ V(G)|node.label = malato.label − 1 do
268:              codaSink.enqueue(node)
269:            end for
270:            for all node ∈ V(G)|node.label ≥ malato.label do
271:              node.flussoPassante ← 0
272:            end for
273:          end if
274:        end if
275:      end for
276:    else if ¬codaSink.isEmpty ∧ ¬sinkRepaired then {esplorazione della
  parte di t}
277:      element ← codaSink.dequeue()
278:      if element.sourceSide ∨ ¬element.sinkValid ∨
  element.flussoPassante = 0 then
279:        continue
280:      end if
```

```
281:      for all e ∈ δ(element) do
282:        p ← e.previosuNode
283:        n ← e.nextNode
284:        if element = n ∧ u_f(e) > 0 then
285:          if p.flussoPassante ≠ 0) then
286:            if ¬p.sourceSide ∨ ¬p.sourceValid then
287:              continue
288:            else
289:              f ← min(p.flussoPassante, n.flussoPassante, u_f(e))
290:              if f = 0 then
291:                continue
292:              end if
293:              e.reversed ← false
294:              n.updatePath(p, e)
295:              return (f, n)
296:            end if
297:          end if
298:          e.reversed ← false
299:          p.update(p, e)
300:          codaSink.enqueue(p)
301:        else if element = p ∧ f(e) > 0 then
302:          if n.flussoPassante ≠ 0 then
303:            if ¬n.sourceSide ∨ ¬n.sourceValid then
304:              continue
305:            else
306:              f ← min(p.flussoPassante, n.flussoPassante, f(e))
307:              if f = 0 then
308:                continue
309:              end if
310:              e.reversed ← false
311:              p.updatePath(n, e)
312:              return (f, p)
313:            end if
314:          end if
315:          e.reversed ← true
316:          n.update(p, e)
317:          codaSink.enqueue(n)
318:        end if
319:      end for
320:    end if
321:  end while
322:  return (0, null)
```

Algorithm 3 SourceSickPropagation

Require: rete (G, u, s, t), nodo *malato*, coda di nodi *codaSource*
Ensure: nodo "di confine" che è sourceValid, null altrimenti
  malati ← coda di nodi
  malati.enqueue(*malato*)
  while $\neg malati.isEmpty$ **do**
    $m \leftarrow malati$.dequeue()
    if $m.sourceSide \vee LastNodesSinkSide$.contains($m$) **then** {se è esplroato da source o è un nodo di confine, il nodo di confine è ottenuto da graph}
      RepairNode($G, m, false$)
      if non sono riuscito a riparare m **then**
        for all $e \in delta^+(m) | e.nextNode.PreviousEdge = e$ **do** {per ogni arco che esce da m e lo ha come predecessore (o meglio, che ha l'arco considerato come arco predecessore)}
          $malati$.enqueue($e.nextNode$)
        end for
        for all $e \in delta^-(m) | e.previousNode.previousEdge = e$ **do** { per ogni arco che entra in m e ha m come nodo predecessore }
          $malati$.enqueue($e.previousNode$)
        end for
      else if $m.nextEdge \neq null \wedge m.nextNode.flussoPassante > 0 \wedge ((m.nextEdge.reversed \wedge f(m.nextEdge) > 0) \vee ((\neg m.nextEdge.reversed \wedge u_f(m.nextEdge) > 0)))$ **then**
        return m
      else
        $codaSource$.enqueue(m)
      end if
    end if
  end while
  return null

1.4 SinkSickPropagation

Algorithm 4 SinkSickPropagation

Require: rete (G, u, s, t),nodo *malato*, coda di nodi *codaSink*
Ensure: nodo "di confine" che è sinkValid, null altrimenti
  malati ←coda di nodi
  malati.enqueue(*malato*)
  while $\neg malati.isEmpty$ **do**
    $m \leftarrow malati$.dequeue()
    if $\neg m.sourceSide$ **then**
      RepairNode$(G, m, true)$
      if non sono riuscito a riaparare m **then**
        for all $e \in delta^+(m) | e.nextNode.nextEdge = e$ **do**
          $malati$.enqueue$(e.nextNode)$
        end for
        for all $e \in delta^-(m) | e.previousNode.nextEdge = e$ **do**
          $malati$.enqueue$(e.previousNode)$
        end for
      else if $m.previousEdge \neq null \wedge m.previousNode.flussoPassante > 0 \wedge ((m.previousEdge.reversed \wedge f(m.previousEdge) > 0) \vee (\neg m.previousEdge.reversed \wedge u_f(m.previousEdge) > 0))$ **then** {il nodo è stato riparato e ha un predecessore valido}
        return m
      else
        $codasink$.enqueue(m)
      end if
    end if
  end while
  return null

1.5 RepairNode

Algorithm 5 RepairNode

Require: una rete (G, u, s, t), nodo da riparare *node*, booleano per comprendere, nel caso il nodo sia di confine, se riparare sul lato esplorato da source (true) o in quello esplorato da sink (false) $onlySinkExploration$

Ensure: booleano che mi conferma se il nodo è stato riparato o meno

if $node = s \vee node = t$ **then**
  return false
end if
$pe \leftarrow node.previousEdge$
$ne \leftarrow node.nextEdge$
$pn \leftarrow node.previousNode$
$nn \leftarrow node.nextNode$
if $(pe \neq null \wedge ne \neq null \wedge pn.sourcevalid \wedge nn.sinkValid \wedge ((pe.reversed \wedge f(pe) > 0) \vee (\neg pe.reversed \wedge u_f(pe) > 0)) \wedge ((ne.reversed \wedge f(ne) > 0) \vee (\neg ne.reversed \wedge u_f(ne) > 0))) \vee (pe \neq null \wedge pn.sourceValid \wedge ((pe.reversed \wedge f(pe) > 0) \vee (\neg pe.reversed \wedge u_f(pe) > 0)) \wedge ne = null) \vee (ne \neq null \wedge nn.sinkValid \wedge pe = null \wedge ((ne.reversed \wedge f(ne) > 0) \vee (\neg ne.reversed \wedge u_f(ne) > 0)))$ **then** {il nodo non ha bisogno di essere riparato,viene prima anallizato il nodo di confine, poi il nodo esplorato da source e infine il nodo esplorato da sink (non di confine)}
  return true
end if
if *node.sourceSide* **then**
  for all $e \in \delta(node)$ **do**
    $p \leftarrow e.previousNode$
    $n \leftarrow e.nextNode$
    if $p.sourceSide \neq n.sourceSide$ **then** {verrano anallizati per i nodi di confine con onlySinkExploration true}
      continue
    end if
    if $node = n \wedge u_f(e) > 0 \wedge p.label = node.label - 1 \wedge p.sourceValid \wedge previous.flussoPassante > 0$ **then**
      $e.reversed \leftarrow$false
      $node$.update(p, e)
      return true
    else if $node = p \wedge f(e) > 0 \wedge n.label = node.label - 1 \wedge n.sourceValid \wedge n.flussoPassante > 0$ **then**
      $e.reversed \leftarrow$true
      $node$.update(n, e)
      return true
    end if
  end for

```
else
  for all e ∈ δ(node) do
    p ← e.previousNode
    n ← e.nextNode
    if p.sourceSide ≠ n.sourceSide ∧ ¬onlySinkExploration then
      if n = node ∧ u_f(e) > 0 ∧ p.sourceValid ∧ p.sourceSide ∧
p.flussoPassante > 0 then
        e.reversed ← false
        node.update(p, e)
        return true
      end if
      if p = node ∧ f(e) > 0 ∧ n.sourceValid ∧ n.sourceSide ∧
n.flussoPassante > 0 then
        e.reversed ← true
        node.update(n, e)
        return true
      end if
    else if onlySinkExploration ∧ n.sourceSide = p.sourceSide then
      if node = p ∧ u_f(e) > 0 ∧ n.sinkValid ∧ node.label = n.label + 1 ∧
n.flussoPassante > 0 then
        e.reversed ←false
        node.update(n, e)
        return true
      end if
      if node = n ∧ f(e) > 0 ∧ p.sinkValid ∧ node.label = p.label + 1 ∧
p.flussoPassante > 0 then
        e.reversed ←true
        node.update(p, e)
        return true
      end if
    end if
  end for
end if
if node.sourceSide ∨ ¬onlySinkExploration then
  node.sourceValid ←false
else
  node.sinkValid ← false
end if
return false
```