K-Truss massimali in un grafo

Marco Miani

Laboratorio Computazionale

Indice

1 Il problema affrontato **2**

2 L'algoritmo risolutivo **2**
2.1 Preprocessing 2
2.2 Ciclo 5
2.3 Assegnamento finale 6

3 I risultati Sperimentali **6**
3.1 Esempi numerici 7

4 Extra **7**

1 Il problema affrontato

Il problema assegnatomi riguardava la ricerca delle k-truss massimali all'interno di un grafo, dove una k-truss è definita come un grafo connesso non orientato in cui ogni spigolo è incidente ad almeno K triangoli.
Formalmente dato un grafo $G = (V, E)$ allora questo è una k-truss *sse*

$$|T_{(u,v)}| = |\{w \in V \text{ t.c } (w, u) \in E \wedge (w, v) \in E\}| \geq k \qquad \forall (u, v) \in E$$

La k-truss è un rilassamento della k-cricca (grafo con k nodi in cui ogni coppia di vertici è collegata), in particolare ogni k-cricca è anche una $(k-2)$-truss. Le k-truss sperimentalmente si sono rivelate uno strumento molto potente per identificare reti coesive in grafici del mondo reale come i social network.

2 L'algoritmo risolutivo

Nel seguito $G = (V, E)$ sarà un grafo con $|V| = n$ e $|E| = m$.
L'algoritmo proposto dall'articolo, in particolare, per ogni arco trova la dimensione della k-truss massimale che lo contiene (parametro salvato come τ). Il programma all'inizio chiede di inserire un intero K_{max}, troverà quindi tutti gli archi con $\tau < K_{max}$ e assegnerà $\tau = K_{max}$ a tutti gli altri.
Per fare ciò dobbiamo tenere in memoria una struttura dati randomizzata, l'algoritmo si articola in vari step che spiego brevemente allegando anche alcune parti di codice (in C) esplicative.

2.1 Preprocessing

Step 1

Per iniziare fissiamo il valore della costante intera $L = cK_{max} \log n$, dove c è una costante assegnata a priori, maggiore è c maggiore sarà la probabilità di ottenere una soluzione corretta.

```
int L = (int) c*Kmax*log((double)numNodi);
```

Fatto ciò generiamo L sottoinsiemi random $X_1, \ldots, X_L \subseteq V$, in modo che ogni vertice appartenga a ciascun X_i con probabilità $0.01/K_{max}$. Per ottimizzare la memoria utilizzata li salviamo come array, ci servono ordinati perché poi dovremo farci la ricerca binaria.
Per ogni nodo $v \in V$ e per ogni $l \in \{1, \ldots, L\}$ lanciamo la funzione *randomSubset* per decidere se v apparterrà o meno a X_l

```
int randomSubset (int Kmax) {
        if (random() % (100*Kmax)) return 0;
        return 1;
}
```

Step 2

Utilizzeremo una struttura dati che, per ogni arco $e = (u, v)$, salva:

$$G_e(l) = \sum_{w \in N(u) \cap N(v) \cap X_l} w \qquad \forall l \in \{1, \ldots, L\}$$

$$\Delta_e = \sum_{w \in N(u) \cap N(v)} 1$$

Per farlo, definiamo la seguente struct:

```
typedef struct edge {
        int nodo1, nodo2;
        int Delta;
        int* G;
        int tau;
} edge_t;
```

Valutiamo quindi questi valori per ciascun arco $e = (u, v) \in E$. Innanzitutto sia $K_{max} = m^a$, sia $b \in [0, 1 - a]$ e sia $s = m^b$.
(il programma chiederà di inserire un valore C_{onstB} così da fissare $b = aC_{onstB}$).
Dividiamo i nodi $v \in V$ in Heavy (se $N(v) > s$) e Light (se $N(v) \leq s$).
Dividiamo i triangoli in Light (con tutti e 3 i vertici Light) e Heavy (altrimenti).
Procediamo quindi al calcolo di $G(l)$ e Δ separatamente per Light e Heavy.
Calcolo Light
Procediamo per enumerazione, stando attenti a non sommare più volte gli stessi triangoli

```
for (v=0;v<numNodi;v++)
if (nodes[v].numAdiacenti <= s) {
  for (provv1=nodes[v].inizioListaAdiacenti; provv1!=NULL; provv1=provv1->next)
  for (provv2=provv1->next; provv2!=NULL; provv2=provv2->next) {
      u = provv1->indice;
      w = provv2->indice;
      if (edges[numNodi*u+w] != NULL)
       if ((nodes[u].numAdiacenti<=s && u>v) || nodes[u].numAdiacenti>s)
        if ((nodes[w].numAdiacenti<=s && w>v) || nodes[w].numAdiacenti>s) {
            edges[numNodi*v+u]->Delta++;
            edges[numNodi*v+w]->Delta++;
            edges[numNodi*u+w]->Delta++;
            for (l=0;l<L;l++) {
              if (ricercaBinaria(v,X[l],0,Xlenght[l]))
                 edges[numNodi*u+w]->G[l] += v;
              if (ricercaBinaria(u,X[l],0,Xlenght[l]))
                 edges[numNodi*v+w]->G[l] += u;
              if (ricercaBinaria(w,X[l],0,Xlenght[l]))
                 edges[numNodi*u+v]->G[l] += w;
} }     }    }
```

Calcolo Heavy
$\forall l \in \{1, \ldots, L\}$ generiamo due matrici A e A'. Per ogni $v \in V$ e $w \in V \cap X_l$

$$A(v,w) = \begin{cases} 1 & \text{se } (v,w) \in E \\ 0 & \text{se } (v,w) \notin E \end{cases} \qquad A'(v,w) = \begin{cases} w & \text{se } (v,w) \in E \\ 0 & \text{se } (v,w) \notin E \end{cases}$$

Si osserva quindi che $\forall e = (u,v) \in E$

$$G_e(l) = \left(A'A^T\right)_{u,v}$$

e quasi allo stesso modo si può calcolare anche Δ_e, riporto solo questa parte di codice per brevità.
Il programma inizia facendo un binding degli indici dei nodi Heavy.

```
int* bindingHeavy = malloc(numNodi*sizeof(int));
int numNodiHeavy;
counter=0;
for (v=0;v<numNodi;v++)  if (nodes[v].numAdiacenti > s) {
    bindingHeavy[counter++] = v;
}
numNodiHeavy = counter;
```

Genera le matrici

```
for (i=0;i<numNodiHeavy;i++)
 for(j=0;j<numNodiHeavy;j++) {
    v = bindingHeavy[i];
    w = bindingHeavy[j];
    if (edges[numNodi*v+w] != NULL) {
      Atrasp[numNodiHeavy*j+i] = 1;
      Aprimo[numNodiHeavy*i+j] = 1;
    }
    else {
      Atrasp[numNodiHeavy*j+i] = 0;
      Aprimo[numNodiHeavy*i+j] = 0;
    }
 }
```

Le moltiplica

```
matrixMultiplication(Aprimo, Atrasp, Result, numNodiHeavy);
```

E aggiorna i risultati

```
for (i=0;i<numNodiHeavy;i++)
 for(j=i+1;j<numNodiHeavy;j++) {
  v = bindingHeavy[i];
  u = bindingHeavy[j];
  if (edges[numNodi*v+u] != NULL)
    edges[numNodi*v+u]->Delta += Result[numNodiHeavy*i+j];
}
```

2.2 Ciclo

Man mano che assegnamo valori di τ a degli archi, eliminiamo questi ultimi dal grafo e continuiamo l'algoritmo sul grafo residuo. Il programma si ferma o quando il grafo residuo è vuoto o quando viene raggiunto il K_{max} inserito.
Per questo procedimento abbiamo bisogno di uno heap, che ho implementato.

```
for (k=1; k<=Kmax; k++) {
  while (getMin(heap)!=NULL && getMin(heap)->Delta < k) {
      e = extractMin(heap);
      ...
```

Step 3

Assegnamo il valore di τ all'arco appena estratto

```
e->tau = k-1;
```

Step 4

Creiamo la lista di tutti i triangoli che coinvolgono l'arco e, ci servirà poi. La salviamo come lista di nodi. Per ogni l controlliamo il valore di $xl = G_e(l)$, e guardiamo se il nodo corrispondente a quell'indice è collegato ad entrambi gli estremi di $e = (u, v)$, in caso affermativo lo aggiungiamo alla lista (a meno che non ci sia già). Ovviamente questo procedimento elenca tutti i triangoli solo se ognuno di questi è *da solo* in almeno un X_i, per questo abbiamo bisogno che L sia molto grande.

```
v = e->nodo1;
u = e->nodo2;
trianglesInvolvingE = NULL;
for (l=0;l<L;l++) {
  xl = e->G[l];
  if (xl>=0 && xl<numNodi)
    if (edges[numNodi*xl+v]!=NULL && edges[numNodi*xl+u]!=NULL) {
      giaInLista=0;
      for (provv=trianglesInvolvingE; provv!=NULL; provv=provv->next)
        if (xl == provv->indice) giaInLista=1;
      if (!giaInLista)
        trianglesInvolvingE = newElementoLista(xl,trianglesInvolvingE);
    }
}
```

Step 5

Rimuoviamo l'arco e dal grafo residuo

```
edges[numNodi*u+v] = edges[numNodi*v+u] = NULL;
```

Step 6

Aggiorniamo i valori di Δ nel grafo residuo

```
for (provv=trianglesInvolvingE; provv!=NULL; provv=provv->next) {
  w = provv->indice;
  edges[numNodi*v+w]->Delta--;
  aggiusta(heap, edges[numNodi*v+w]->posizioneNellHeap);
  edges[numNodi*u+w]->Delta--;
  aggiusta(heap, edges[numNodi*u+w]->posizioneNellHeap);
}
```

e di $G(l)$

```
for (l=0;l<L;l++) {
  if (ricercaBinaria(v,X[l],0,Xlenght[l]))
    for(provv=trianglesInvolvingE; provv!=NULL; provv=provv->next) {
      w = provv->indice;
      edges[numNodi*v+w]->G[l] -= u;
    }
  if (ricercaBinaria(v,X[l],0,Xlenght[l]))
    for(provv=trianglesInvolvingE; provv!=NULL; provv=provv->next) {
      w = provv->indice;
      edges[numNodi*u+w]->G[l] -= v;
    }
}
```

2.3 Assegnamento finale

Finito il ciclo **for**, assegnamo i valori di τ anche agli archi rimanenti nel grafo residuo, se ce ne sono

```
while (getMin(heap)!=NULL)
    extractMin(heap)->tau = k-1;
```

3 I risultati Sperimentali

Per fare delle prove ho utilizzato alcuni grafi scaricati da *snap.stanford.edu*, nello specifico quelli relativi a Facebook. Li allego nella cartella "grafi".
Il parametro c influisce solo sul numero L di sottoinsiemi $X_1, \ldots, X_L \subseteq V$ che vengono creati. Maggiore sarà c e maggiori saranno la probabilità di ottenere una soluzione corretta e il tempo di esecuzione. Facendo alcune prove ho riscontrato che un valore di $c = 100$ garantisce pressoché sempre una soluzione corretta.

3.1 Esempi numerici

Riporto un paio di esempi, con rispettivi parametri da inserire, per farle verificare potenzialità e limiti del programma:

Potenzialità: Un grafo contenente 347 nodi e 2519 archi
Nome del file: "grafi/0.edges"
K_{max}: 20
c: 100
C_{onstB}: 0.7
Il programma trova tutte le k-truss massimali (il k massimo è 15)
Tempo impiegato dal mio pc: qualche secondo.

Limiti: Un grafo contenente 1912 nodi e 26749 archi
Nome del file: "grafi/107.edges"
K_{max}: 10
c: 100
C_{onstB}: 0.55
Il programma assegna i rispettivi valori di $0 \leq \tau < 10$ ai rispettivi archi, assegna $\tau = 10$ anche ai 22220 archi rimanenti nell'heap che avranno τ effettivo ≥ 10.
Tempo impiegato dal mio pc: un paio di minuti

4 Extra

Ho pensato anche ad un metodo per verificare la correttezza (o meglio,verificare l'incorrettezza oppure essere più sicuri della correttezza) della soluzione.
L'ho lasciato in commento in fondo al programma e lo riporto qui sotto. Alla fine dell'algoritmo (se il K_{max} inserito non ha "esaurito tutto il grafo") mi faccio stampare il grafo residuo in un file. Rilanciando poi l'algoritmo su quel file, se la soluzione era corretta necessariamente dovrò ottenere valori di τ tutti maggiori o uguali al K_{max} limite che avevo messo prima. (con il vantaggio che il nuovo grafo ha meno nodi e archi, quindi il preprocessing sarà meno gravoso. Il che mi permette di scegliere un c maggiore e avere la soluzione corretta con maggiore probabilità).

```
FILE *controllo = fopen("Sottografo_di_controllo.txt","w");
while (getMin(heap)!=NULL)                    {
    e = extractMin(heap);
    fprintf(controllo,"%d %d\n",
        e->nodo1+IndiceNodoMin-1,e->nodo2+IndiceNodoMin-1);
}
```