Subgrafo K-Conexo Minimo

Andrés Felipe Movilla Obregón
Fundación Universidad del Norte
movillaf@uninorte.edu.co
Noviembre, 2, 2017

Terminologia:

1. **Vertice:** Representacion de una informacion.
2. **Arista:** Representacion de conexion entre dos vertices.
3. **Grafo:** Conjunto de vertices unidos por aristas.
4. **Camino:** Conjunto de aristas con comienzo en un vertice y fin en otro vertice.
5. **Grafo conexo:** Grafo que para cualquier dos vertices existe un camino que los conecte.
6. **Conexidad:** Propiedad de un grafo que implica la cantidad de vértices o aristas que al ser removidos el grafo deja de ser un grafo conexo.
7. **Grafo k-conexo:** Grafo conexo que tras remover 'k' vértices, se desconecta.

Resumen:
Dado un entero positivo k y un grafo G k-conexo, y un subgrafo de expansión k-conexo de G con un número mínimo de vertices. Este problema es conocido por ser NP-complete. Khuller y Raghavachari presentaron el primer algoritmo con una relación de rendimiento menor a 2 para todos k. Resultaron un límite superior de 1.85 para el rendimiento relativo de su algoritmo. Luego se demostro que la relación de rendimiento de su algoritmo es menor que 1.7 para k lo suficientemente grande, y que es como máximo 1.75 p ara todo k. Entonces se mejoran las relaciones más conocidas para cualquier k >= 4, en particular, para k = 4 de 1.75 a 1.65, y para k = 5 de 1.733 ... a 1.68. Por último, mostramos que el mínimo subgrafo de expansión k-conexo es MAX SNP-hard, incluso para k = 2. Todo esto se ve afectado por que el solo computar la conexidad de vértices de un grafo G tiene un tiempo de procesamiento polinomial, con una complejidad de $\min(k^3+n,kn)*m$.

El presente artículo tiene como fin analizar y presentar una solución al problema NP-complete conocido como "Subgrafo K-Conexo Minimo". El cual, como dicho antes, tiene la complicación que el solo revisar la conexidad de un grafo es un problema de tiempo polinomial donde se busca encontrar un camino conectando al menos un vértice a todos los otros vértices. A continuación, se hará una presentación con el índice de términos, el planteamiento del problema por parámetros, los antecedentes del problema, su desarrollo en algoritmo con sus respectivas funciones de tiempo T(¿?) y complejidad O(¿?).

Planteamiento:

Dado un grafo G = (V, E) y enteros positivos K ≤ |V| y B ≤ |E|,

$Existe\ un\ subconjunto\ \ de\ vértices\ E'\ contenido\ en\ E\ con\ |E'| \leq\ B\ tal\ que\ G' = (V, E')es\ K - conexo?$

Algoritmos:

Para hallar K,

```
for (int i = 0; i < this.allSetups.length; i++) {
      if (this.allSetups[i].getK() == -1) {
            this.setup(i);
      i = 0;
      boolean connected = true;
while (i < N && connected) {
      connected = findPathsForVertex(i);
      i++;
      }
      currentSetup.setK(connected? -1 : 0);
}
}
for (int i = 0; i < this.allSetups.length; i++) {
      GraphSetup set = this.allSetups[i];
      if (set.getK() == -1) {
            int minDist = N;
            for (int j = 0; j < this.allSetups.length; j++) {
                  GraphSetup other = this.allSetups[j];
                  if (i != j && !other.isConnected()) {
                        minDist = Integer.min(minDist,
set.distanceTo(other));
                  }
            }
            set.setK(minDist);
      }
}
```

Para hallar el minimo subgrafo,

```
int maximumC = -1;
int index = -1;
for (int i = 0; i < this.allSetups.length; i++) {
      GraphSetup setup = this.allSetups[i];
      if (setup.getK() == K) {
            if (setup.getC() > maximumC) {
                  maximumC = setup.getC();
                  index = i;
            }
      }
}
if (index != -1) {
      this.setup(index);
} else {
      this.setup(0);
```

```
}
```

Algoritmo Concreto:

```
boolean checkConnection = false;
boolean[] donePaths = new boolean[N];

for (int j = 0; j < N; j++)
      donePaths[j] = false;

boolean done = false;
int goalID = 0;
int changes = 0;
Vertex goal = V[goalID];
Vertex start = V[a];

if (!start.isActive())
      return findPathsForVertex(a+1);

while (!done) {
      donePaths[goalID] = true;
      if (start.getP()[goalID].getGoal() != goal && goal.isActive()) {
            int currentPathIndex = -1;
            int currentLength = N+1;
            for (int j = 0; j < N; j++) {
                  if (start.getP()[j].getGoal() == V[j]) {
                        Vertex midVertex = start.getP()[j].getGoal();
                        if (midVertex.isActive() &&
midVertex.getP()[goalID].getGoal() == goal) {
                              int possibleLength =
start.getP()[j].getLength() + midVertex.getP()[goalID].getLength() -
1;
                              if (possibleLength < currentLength) {
                                    currentPathIndex = j;
                                    currentLength = possibleLength;
                              }
                        }
                  }
            }
            if (currentPathIndex != -1) {

      start.getP()[goalID].add(start.getP()[currentPathIndex]);

      start.getP()[goalID].add(start.getP()[currentPathIndex].getGoal()
.getP()[goalID]);
                  donePaths[goalID] = true;
                  changes++;
            }
      }
}
```

Analisis

Tras recibir los vertices y la matriz de adyacencia del grafo G, el algoritmo procede a generar todas las posibles combinaciones de vértices, estando cada uno removido o no removido. Despues, se establecen las conexiones iniciales dictadas por la matriz de adyacencia, y se recorren todas las combinaciones del grafo, verificando que todo vértice tenga un camino a todo otro vértice con un algoritmo implementado de Bellman Ford. Si se encuentra que un vértice no tiene camino a algún otro vértice, se concluye que el grafo es disconexo y se dice que su K es 0. De lo contrario, se resuelve como K -1. Tras estableces esto, procede a recorrer las combinaciones de nuevo y hallar las verdaderas K para las combinaciones conexas. Estas K se hallan buscando la combinación del grafo con menor distancia a la combinación actual, de tal forma que si se remueven K vértices, se vuelve una combinación del grafo que es disconexa. Finalmente, cuando el usuario pide ver un subgrafo del grafo original G con una K en particular, solo se recorren las combinaciones del grafo buscando una combinación con la respectiva K, y luego se revisa que ese subgrafo, o combinación del grafo original, sea aquel con la mayor cantidad de vértices, tal que necesitas remover la menor cantidad de vértices para cumplir la K dada por el usuario.

Para resolver todos los K:

$$\boldsymbol{T(n) = \left(2^{n^2}\right) + (2^n * n^3) + 1}$$

$$\boldsymbol{O(n) = \left(2^{n^2}\right)}$$

Algoritmos en pseudo codigo:

Para hallar K,

```
para (i = 0, this.allSetups.length) haga
      si (this.allSetups[i].getK() == -1) entonces
            this.setup(i)
      i = 0;
      boolean connected = true;
mientras que (i < N && connected) haga
      connected = findPathsForVertex(i);
      i = i + 1;
      fin mientras que
            si (connected) entonces
                  currentSetup.setK(-1)
            si no
                  currentSetup.setK(0)
            fin si
      fin si
Fin para
para (i = 0, this.allSetups.length) haga
      GraphSetup set = this.allSetups[i]
      si (set.getK() == -1) entonces
            entero minDist = N
            for (j = 0,  this.allSetups.length) entonces
                  GraphSetup other = this.allSetups[j]
                  si (i != j && !other.isConnected()) entonces
                        si (minDist < set.distanceTo(other)) entonces
                              minDist = set.distanceTo(other)
                        fin si
                  fin si
            fin para
            set.setK(minDist);
      fin si
fin para
```

Para hallar el minimo subgrafo,

```
entero maximumC = -1;
entero index = -1;
para (i = 0, this.allSetups.length) haga
      GraphSetup setup = this.allSetups[i]
      si (setup.getK() == K) entonces
            si (setup.getC() > maximumC) entonces
                  maximumC = setup.getC();
                  index = i;
            fin si
      fin si
fin para
si (index != -1) entonces
```

```
      this.setup(index)
si no
      this.setup(0)
fin si
```

Algoritmo Concreto en pseudo codigo:

```
booleano checkConnection = falso;
booleano[] donePaths = new booleano[N];

para (j = 0, N) haga
     donePaths[j] = false;
fin para

booleano done = false;
entero goalID = 0;
entero changes = 0;
Vertex goal = V[goalID];
Vertex start = V[a];

si (!start.isActive()) entonces
     return verdadero;
fin si

mientras que (!done) {
     donePaths[goalID] = true;
     si (start.getP()[goalID].getGoal() != goal && goal.isActive())
entonces
          entero currentPathIndex = -1;
          entero currentLength = N+1;
          para (int j = 0; j < N; j++) haga
               si (start.getP()[j].getGoal() == V[j]) entonces
                    Vertex midVertex = start.getP()[j].getGoal();
                    si (midVertex.isActive() &&
midVertex.getP()[goalID].getGoal() == goal) entonces
                         entero possibleLength =
start.getP()[j].getLength() + midVertex.getP()[goalID].getLength() -
1;
                         si (possibleLength < currentLength)
entonces
                              currentPathIndex = j;
                              currentLength = possibleLength;
                         fin si
                    fin si
               fin si
          fin para

          si (currentPathIndex != -1) entonces

     start.getP()[goalID].add(start.getP()[currentPathIndex]);

     start.getP()[goalID].add(start.getP()[currentPathIndex].getGoal()
.getP()[goalID]);
               donePaths[goalID] = true;
```

```
                changes = changes + 1;
            fin si
        fin si
fin mientras que
```

Accion de mejora

No revisar todo vértice con todo otro vértice para revisar conexidad, sino revisar un vértice con todo otro vértice.

Algoritmos Mejorados:

Para hallar K,

```
for (int i = 0; i < this.allSetups.length; i++) {
     if (this.allSetups[i].getK() == -1) {
          this.setup(i);
     currentSetup.setK(findPathsForVertex(0)? -1 : 0);
}
}
for (int i = 0; i < this.allSetups.length; i++) {
     GraphSetup set = this.allSetups[i];
     if (set.getK() == -1) {
          int minDist = N;
          for (int j = 0; j < this.allSetups.length; j++) {
               GraphSetup other = this.allSetups[j];
               if (i != j && !other.isConnected()) {
                    minDist = Integer.min(minDist,
set.distanceTo(other));
               }
          }
          set.setK(minDist);
     }
}
```

Para hallar el minimo subgrafo,

```
int maximumC = -1;
int index = -1;
for (int i = 0; i < this.allSetups.length; i++) {
     GraphSetup setup = this.allSetups[i];
     if (setup.getK() == K) {
          if (setup.getC() > maximumC) {
               maximumC = setup.getC();
               index = i;
          }
     }
}
if (index != -1) {
     this.setup(index);
} else {
     this.setup(0);
}
```

Algoritmo Concreto Mejorado:

```
boolean checkConnection = false;
boolean[] donePaths = new boolean[N];

for (int j = 0; j < N; j++)
      donePaths[j] = false;


boolean done = false;
int goalID = 0;
int changes = 0;
Vertex goal = V[goalID];
Vertex start = V[a];

if (!start.isActive())
      return findPathsForVertex(a+1);

while (!done) {
      if (start.getP()[goalID].getGoal() != goal && goal.isActive()) {
            int currentPathIndex = -1;
            int currentLength = N+1;
            for (int j = 0; j < N; j++) {
                  if (start.getP()[j].getGoal() == V[j]) {
                        Vertex midVertex = start.getP()[j].getGoal();
                        if (midVertex.isActive() &&
midVertex.getP()[goalID].getGoal() == goal) {
                              int possibleLength =
start.getP()[j].getLength() + midVertex.getP()[goalID].getLength() - 1;
                              if (possibleLength < currentLength) {
                                    currentPathIndex = j;
                                    currentLength = possibleLength;
                              }
                        }
                  }
            }

            if (currentPathIndex != -1) {
                  start.getP()[goalID].add(start.getP()[currentPathIndex]);

      start.getP()[goalID].add(start.getP()[currentPathIndex].getGoal().getP(
)[goalID]);
                  donePaths[goalID] = true;
                  changes++;
            }

      } else {
            donePaths[goalID] = true;
      }
}
```

Analisis

Tras recibir los vertices y la matriz de adyacencia del grafo G, el algoritmo procede a generar todas las posibles combinaciones de vértices, estando cada uno removido o no removido. Despues, se establecen las conexiones iniciales dictadas por la matriz de adyacencia, y se recorren todas las combinaciones del grafo, verificando que el primer vértice no removido en la combinación tenga una conexión a todo otro vértice no removido con un algoritmo implementado de Bellman Ford. Si se encuentra que un vértice no tiene camino a algún otro vértice, se concluye que el grafo es disconexo y se dice que su K es 0. De lo contrario, se resuelve como K -1. Tras estableces esto, procede a recorrer las combinaciones de nuevo y hallar las verdaderas K para las combinaciones conexas. Estas K se hallan buscando la combinación del grafo con menor distancia a la combinación actual, de tal forma que si se remueven K vértices, se vuelve una combinación del grafo que es disconexa. Finalmente, cuando el usuario pide ver un subgrafo del grafo original G con una K en particular, solo se recorren las combinaciones del grafo buscando una combinación con la respectiva K, y luego se revisa que ese subgrafo, o combinación del grafo original, sea aquel con la mayor cantidad de vértices, tal que necesitas remover la menor cantidad de vértices para cumplir la K dada por el usuario.

Para resolver todos los K:

$$\boldsymbol{T(n) = \left(2^{n^2}\right) + (2^n * n^2) + 1}$$

$$\boldsymbol{O(n) = \left(2^{n^2}\right)}$$

Algoritmos Mejorados en pseudo codigo:

Para hallar K,

```
para (i = 0, this.allSetups.length) haga
      si (this.allSetups[i].getK() == -1) entonces
            this.setup(i)
            si (findPathsForVertex(0)) entonces
                  currentSetup.setK(-1)
            si no
                  currentSetup.setK(0)
            fin si
      fin si
Fin para
para (i = 0, this.allSetups.length) haga
      GraphSetup set = this.allSetups[i]
      si (set.getK() == -1) entonces
            entero minDist = N
            for (j = 0,  this.allSetups.length) entonces
                  GraphSetup other = this.allSetups[j]
                  si (i != j && !other.isConnected()) entonces
                        si (minDist < set.distanceTo(other)) entonces
                              minDist = set.distanceTo(other)
                        fin si
                  fin si
            fin para
            set.setK(minDist);
      fin si
fin para
```

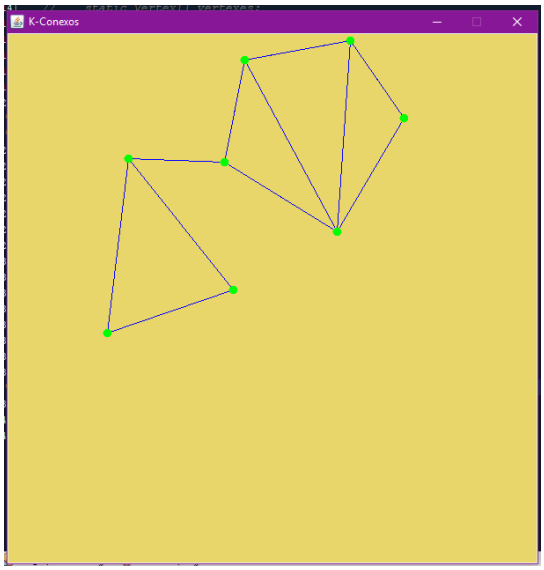
Para hallar el minimo subgrafo,

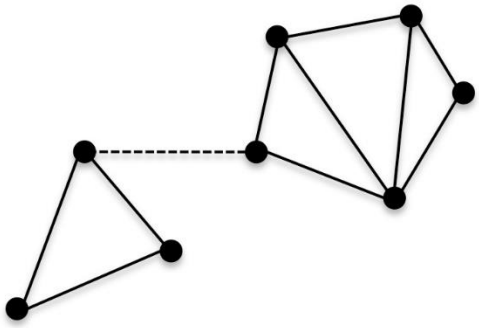
```
entero maximumC = -1;
entero index = -1;
para (i = 0, this.allSetups.length) haga
      GraphSetup setup = this.allSetups[i]
      si (setup.getK() == K) entonces
            si (setup.getC() > maximumC) entonces
                  maximumC = setup.getC();
                  index = i;
            fin si
      fin si
fin para
si (index != -1) entonces
      this.setup(index)
si no
      this.setup(0)
fin si
```

Algoritmo Concreto Mejorado en pseudo codigo:

```
booleano checkConnection = falso
booleano[] donePaths = new booleano[N]
para (j = 0, N) haga
      donePaths[j] = falso
fin para

booleano done = falso
entero goalID = 0
entero changes = 0
Vertex goal = V[goalID]
Vertex start = V[a]

si (!start.isActive()) entonces
      return findPathsForVertex(a+1)
fin si

mientras que (!done) haga
      donePaths[goalID] = verdadero
      si (start.getP()[goalID].getGoal() != goal && goal.isActive()) entonces
            entero currentPathIndex = -1
            entero currentLength = N+1
            para (j = 0, N) haga
                  si (start.getP()[j].getGoal() == V[j]) entonces
                        Vertex midVertex = start.getP()[j].getGoal();
                        if (midVertex.isActive() &&
midVertex.getP()[goalID].getGoal() == goal) entonces
                              entero possibleLength =
start.getP()[j].getLength() + midVertex.getP()[goalID].getLength() - 1;
                              if (possibleLength < currentLength) entonces
                                    currentPathIndex = j;
                                    currentLength = possibleLength;
                              fin si
                        Fin si
                  Fin si
            Fin para

            si (currentPathIndex != -1) entonces
                  start.getP()[goalID].add(start.getP()[currentPathIndex])

      start.getP()[goalID].add(start.getP()[currentPathIndex].getGoal().getP(
)[goalID])
                  donePaths[goalID] = verdadero
                  changes = changes + 1
            fin si
      fin si
fin mientras que
```

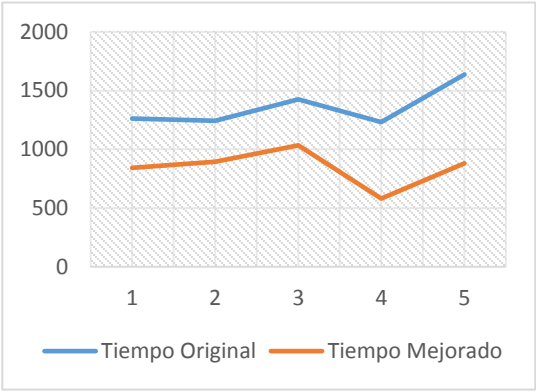
Prueba:





Pruebas de Rendimiento		
Prueba	Tiempo Original (microsegundos)	Tiempo Mejorado (microsegundos)
1	1262.44	841.623
2	1241.91	894.726
3	1426.65	1033.955
4	1231.64	580.568
5	1637.06	880.446

Matriz de Adyacencia								
	A	B	C	D	E	F	G	H
A	0	1	1	0	0	1	0	0
B	1	0	1	0	0	0	0	0
C	1	1	0	0	0	0	0	0
D	0	0	0	0	1	1	0	1
E	0	0	0	1	0	0	1	1
F	1	0	0	1	0	0	0	1
G	0	0	0	0	1	0	0	1
H	0	0	0	1	1	1	1	0



Comparacion de Rendimiento:
La función original de tiempo,

$$\boldsymbol{T^{original}(n) = \left(2^{n^2}\right) + (2^n * n^3) + 1}$$

Mejoró con la reducción de grado en 1,

$$\boldsymbol{T^{mejorado}(n) = \left(2^{n^2}\right) + (2^n * n^2) + 1}$$

Que se puede ver reflejado en la reducción de las medidas de tiempo en el caso mostrado en la sección de pruebas.

Por otro lado, la complejidad general del algoritmo no se vio afectada por esta mejora:

$$\boldsymbol{O^{original}(n) = \left(2^{n^2}\right)}$$

$$\boldsymbol{O^{mejorado}(n) = \left(2^{n^2}\right)}$$

Aunque si se separan los pasos del algoritmo completo si se veria reflejada la mejora en el orden de complejidad de la primera parte del algoritmo, representada en la función de tiempo como el segundo polinomio.

Conclusiones:
Se pueden ver aplicaciones de este algoritmo en la optimización de costos de redes de telecomunicación, fluido eléctrico, flujo de gas y flujo de agua. Tras establecer la red como un grafo se puede analizar que zonas, por ejemplo, quedarían sin servicio si se llegase a averiar una estación, o dos, o tres. Con esta información, los proveedores de estos servicios pueden establecer mas conexiones o agregar estaciones para fortalecer el suministro del servicio a la zona y prevenir un posible inconveniente en el futuro.

De la misma forma, se puede utilizar para asegurarse que una zona esté recibiendo una cantidad considerable o aceptable del servicio que se esta siendo brindado, asi mejorando el servicio, su calidad como tal, al igual que la calidad de la zona y de la vida de los que allí habitan. Esto causaría un impacto positivo en la comunidad afectada por esa zona, y, muy posiblemente, las zonas circundantes también. La mejora de la calidad de vida y la seguridad del préstamo de servicios en una zona pueden servir como impulso de la economía de esta misma, permitiendo a los habitantes de la zona a sentirse a gusto y querer emprender con ideas ahora posibles gracias a esta mejora causada por el uso del algoritmo aquí desarrollado.

Ademas, se puede usar este algoritmo para mejorar las conexiones de líneas de vuelo, asegurándose que haya mas de uno, dos, K, aeropuertos del cual se pueda partir y llegar a un país o conjunto de países. Asi mejorando la comunicación de esta parte del mundo con el resto de este, impulsando el esfuerzo de globalización al igual que el turismo de todo país y ciudad que se encuentre en la zona que tenia limitado acceso previo al uso del algoritmo. Se podría ver afectada la economía de estas ubicaciones causado por el ingreso de turistas e ingreso de moneda extranjera al país.

Antecedentes y Referencias:

Se presentan papers, informes y demas articulos hablando del problema "Subgrafo K-Conexo Minimo", o de su equivalente en aristas, "Subgrafo K-Arista-Conexo", expresados en ingles como "K-Connected Spanning Subgraph" y "K-Edge-Connected Spanning Subgraph".

1. "On the Structure of Minimum-Weight k-Connected Spanning Networks" [Online] Available: http://epubs.siam.org/doi/abs/10.1137/0403027 (Daniel Bienstock, Ernest F. Brickell, and Clyde L. Monma, 1988)
   Complejidad: $O((p (\log n))^{1/2}/k)$

2. "A Better Approximation Ratio for the Minimum Size k-Edge-Connected Spanning Subgraph Problem" [Online] Available: https://pdfs.semanticscholar.org/1ec3/62164016964b14589b288c02384505361343.pdf (Cristina G Fernandes, 1997)

3. "On Approximability of the Minimum-Cost k-Connected Spanning Subgraph Problem" [Online] Available: https://www.hni.uni-paderborn.de/publikationen/publikationen/?tx_hnippview_pi1%5Bpublikation%5D=1533&tx_hnippview_pi1%5Bfelder%5D%5Blade%5D=394 (Artur Czumaj and Andrzej Lingasy, 1999)
   Complejidad: $O(c^2k^4)$, $O(c \log W)$

4. "Approximation algorithms for minimum-cost k-vertex connected subgraphs" [Online] Available: https://dl.acm.org/citation.cfm?id=509955&dl=ACM&coll=DL&CFID=995825419&CFTOKEN=36763894 (Joseph Cheriyan, Adrian Vetta and Santosh Vempala, 2002)

5. "Approximating Minimum-Size k-Connected Spanning Subgraphs via Matching" [Online] Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.8937&rep=rep1&type=pdf (Joseph Cheriyan and Ramakrishna Thurimella, 2006)

6. "The k-edge connected subgraph problem: Valid inequalities and Branch-and-Cut" [Online] Available: https://limos.isima.fr/IMG/pdf/rr-07-01.pdf (F. Bendali, I. Diarrassouba, M. Didi Biha, A. R. Mahjoub, and J. Mailfert, 2007)

7. "Approximating minimum cost connectivity problems" [Online] Available: http://www.crab.rutgers.edu/~guyk/pub/book/nabs.pdf (Guy Kortsarz and Zeev Nutov, 2008)

8. "Minimum Cost ≤k Edges Connected Subgraph Problems" [Online] Available: http://www.sciencedirect.com/science/article/pii/S1571065310000053 (Firdovsi Sharifov and Hakan Kutucu, 2010)

9. “On the minimum-cost λ-edge-connected k-subgraph problem” [Online] Available: https://link.springer.com/article/10.1007/s10287-016-0260-7 (Elham Sadeghi Neng Fan, 2016)