

ALGORITMOS Y ESTRUCTURAS DE DATOS III

Trabajo Práctico N°3

De Sousa Bispo Mariano	389/08	marian_sabianaa@hotmail.com
Grosso Daniel	694/08	dgrosso@gmail.com
Livorno Carla	424/08	carlalivorno@hotmail.com
Raffo Diego	423/08	enanodr@hotmail.com

Junio 2010

Índice

Introducción	2
1. Situaciones de la vida real	2
2. Algoritmo exacto	3
2.1. Explicación	3
2.1.1. Optimizaciones	3
2.2. Detalles de la implementación	4
2.3. Complejidad temporal	5
2.4. Conclusiones	5
3. Heurística constructiva	6
3.1. Explicación	6
3.2. Detalles de la implementación	7
3.3. Desventajas	8
3.4. Complejidad temporal	9
4. Búsqueda local	11
4.1. Explicación	11
4.2. Detalles de la implementación	11
4.3. Desventajas	13
4.4. Complejidad temporal	13
5. Tabu-Search	15
5.1. Explicación	15
5.2. Detalles de la implementación	15
5.3. Desventajas	19
5.4. Complejidad temporal	19
6. Resultados	21
6.1. Parametros de la heurística tabú	21
6.2. Comparacion de tiempos	21
6.3. Comparacion de calidad	21
7. Mediciones	21
8. Compilación y ejecución de los programas	21

Introducción

Este trabajo tiene como objetivo la aplicación de diferentes técnicas algorítmicas para la resolución de tres problemas particulares, el cálculo de complejidad teórica en el peor caso de cada algoritmo implementado, y la posterior verificación empírica.

El lenguaje utilizado para implementar los algoritmos de todos los problemas fue C/C++

1. Situaciones de la vida real

El problema del Clique Máximo puede usarse como modelo para diversas situaciones de la vida real en ámbitos muy variados.

Aplicación N° 1: Por un lado tenemos los problemas que involucren personas (como nodos) y las relaciones entre ellos (los ejes) en distintas materias. El ejemplo mas cotidiano (al menos para todos nosotros) es el de las redes sociales, y las 'amistades' entre las distintas personas. En este caso, puede ser útil para hacer pruebas de mercado, como dar productos gratis para promocionarlos, y dado que el costo de cada producto puede ser elevado se trata de entregar la menor cantidad, asegurandose la máxima promoción posible, entonces se busca el grupo de 'amigos' mas grande intentando que todos se enteren del producto en cuestión.

Aplicación N° 2: Un terrorista quiere infectar a la población con un virus. Supongamos que el virus es de transmisión aérea, y dado el enorme costo de fabricacion del virus, solo se pudieron fabricar un par de cientos de ejemplares. El terrorista utilizaría una modificacion de Max Clique para elegir sus blancos para que la probabilidad de contagio sea mayor.

2. Algoritmo exacto

2.1. Explicación

El Algoritmo busca todas las formas de armar una clique utilizando la técnica de backtracking. Para esto, inicia la clique una vez desde cada vértice probando todas las combinaciones que lo incluyan, agregando vértices tal que forman un completo con los ya incluidos. Se necesita empezar una vez por cada nodo ya que la solución final podría no incluir el nodo inicial. De esta forma, se genera un árbol de backtracking teórico para cada vértice inicial. Mediante podas, evita recorrer el árbol por completo, siendo la solución final la máxima de las cliques encontradas. Como el algoritmo busca todas las cliques del grafo, la solución es la clique máxima del grafo.

2.1.1. Optimizaciones

Dado que se trata de un algoritmo de backtracking, la optimización se basa en podar las ramas en las que estamos seguros que no va a aparecer el óptimo. Para esto tenemos que poder predecir, dado un estado actual, si es posible mejorar el óptimo encontrado hasta el momento.

Por un lado, podemos las ramas que no forman un grafo completo, ya que no es solución.

Por otro lado, evaluamos en cada paso del algoritmo la cantidad de vértices que falta explorar. Es decir, calculamos el tamaño de la clique máxima que podríamos formar considerando los vértices que ya están incluidos en la solución actual. Si la cantidad de vértices que todavía no fueron evaluados más la cantidad de vértices ya pertenecientes a la clique actual es menor a la cantidad de vértices de la clique máxima encontrada hasta el momento, no tiene sentido seguir explorando esa rama ya que el tamaño de la clique máxima que se puede encontrar por ese camino es menor al tamaño de la máxima encontrada. Por este motivo, podemos esta rama.

Además, para cada vértice que inicia la clique se intenta agregar los de mayor numeración tal que forman un completo. Por lo tanto, se evita repetir combinaciones. Supongamos que tenemos una clique de tres vértices, siendo estos el $\langle 1, 2, 3 \rangle$. Con esta optimización, nunca han de analizarse los casos $\langle 2, 3, 1 \rangle$; $\langle 2, 1, 3 \rangle$; $\langle 3, 1, 2 \rangle$ y $\langle 3, 2, 1 \rangle$.

2.2. Detalles de la implementación

Almacenamos las relaciones entre los vértices en una matriz de $n \times n$, donde n es la cantidad de vértices. Cada posición (i, j) de la matriz contiene un *uno* si existe la arista (i, j) y un *cero* en caso contrario. De esta forma se le asigna un número a cada vértice.

A continuación, se muestra el pseudocódigo del algoritmo exacto.

```
exacto(matriz_adyacencia,n)
  solucion  $\leftarrow \emptyset$ 
  for  $i$  to  $n$ 
    buscar clique máxima desde el vértice  $i$ 
    if tiene más vértices que solucion
      solucion  $\leftarrow$  clique encontrada
  return  $\#(solucion)$ 
```

El algoritmo de backtracking recorre todos los vértices. Para cada uno de estos verifica si es adyacente con todos los vértices de la solución actual y si todavía no pertenece a la misma. Si es así lo agrega y repite este procedimiento (avanza). Si no, significa que recorrió todos los vértices y no logró formar una clique mayor a la encontrada, por lo que comienza a retroceder.

Cuando retrocede, saca el último vértice v que agregó a la solución y prosigue la búsqueda desde el vértice siguiente a v en numeración. Si no hay un vértice que se pueda agregar, es decir, si ninguno de los siguientes forma una clique, el algoritmo sigue retrocediendo.

2.3. Complejidad temporal

El algoritmo exacto utiliza la técnica de backtracking, con lo que genera un árbol donde cada rama es una posible solución, podando cuando considere que por ese camino no encuentra solución, o no encuentre una mejor a la actual. Dar la cota de complejidad de este algoritmo es complicado si tuviésemos en cuenta las podas que se realizan en el backtracking. Si quisiéramos ajustar la cota, deberíamos buscar un grafo particular tal que las minimice (las podas), maximizando así la cantidad de vértices del árbol de backtracking. Por este motivo decidimos plantear un caso hipotético donde el árbol de backtracking se genere sin podas, verificando para cada nodo i las posibles cliques que lo contengan, intentando incluir sólo los de mayor numeración para no repetir soluciones. Como dijimos, esto es un caso hipotético, donde el algoritmo tiene un comportamiento menos eficiente que el real, y de esta forma conseguimos una cota superior poco ajustada, pero que a los fines prácticos nos da una idea de que aún en el peor caso, el algoritmo será más eficiente respecto a la cota calculada.

De esta forma el algoritmo verifica para cada vértice i , con $0 < i < n$, las cliques posibles que lo contengan, es decir, todas las combinaciones que tengan a i más otros vértices entre i y n tal que forman una clique.

Así, para cada vértice i genera un árbol donde cada rama representa una clique que lo incluye, por lo que tendremos un árbol de $n-i$ factorial vértices.

Como este procedimiento se repite con i desde 1 hasta n nos queda planteada la siguiente sumatoria: $\sum_{i=1}^n i! = 1! + 2! + 3! + \dots + n!$

Entonces la complejidad viene dada por: $O(1! + 2! + 3! + \dots + n!) = O(\max(1!, 2!, 3!, \dots, n!)) = O(n!)$

2.4. Conclusiones

3. Heurística constructiva

3.1. Explicación

Como primera heurística, en este caso constructiva, desarrollamos un algoritmo goloso para resolver el problema **MAX-CLIQUE** de manera aproximada. El algoritmo parte del vértice de mayor grado en el grafo (el cual es considerado parte de la clique). En cada paso, agrega a la clique el vértice de mayor grado que todavía no pertenece a la misma tal que forma un completo con los ya pertenecientes. Este procedimiento se repite hasta que no se puedan agregar más vértices a la clique. (Figura 1)

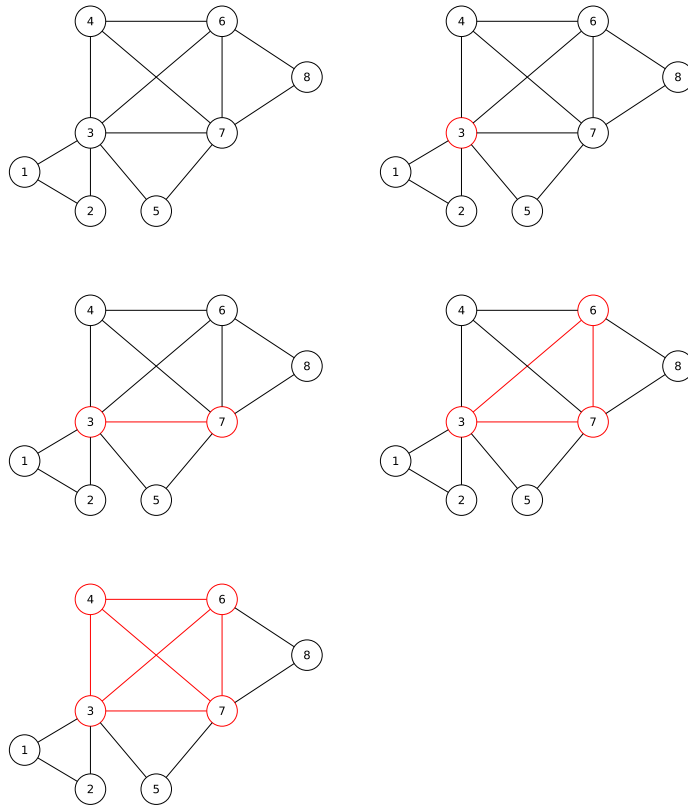


Figura 1: Ejemplo de la heurística constructiva

Se eligió este criterio goloso pensando en que el vértice con mayor grado

tiene más probabilidad de pertenecer a la clique máxima ya que el grado de un vértice perteneciente a la misma condiciona la mayor clique que podrá contenerlo, es decir, si un vértice que está en la clique tiene grado x no se podrá conseguir así una clique de tamaño mayor a $x + 1$.

3.2. Detalles de la implementación

Almacenamos las relaciones entre los vértices en una matriz de $n \times n$, donde n es la cantidad de vértices. Cada posición (i, j) de la matriz contiene un *uno* si existe la arista (i, j) y un *cero* en caso contrario. De esta forma se le asigna un número a cada vértice.

El algoritmo funciona de la siguiente manera:

Sea *grados* un arreglo de tamaño n , donde n es la cantidad de vértices del grafo. En cada posición $j \forall 1 \leq j \leq n$ del arreglo está el grado correspondiente al vértice j . Para construir una clique ordenamos *grados* en forma decreciente. El primer vértice del arreglo, es decir, el de mayor grado del grafo se considera parte de la solución final del algoritmo. Para completar la clique recorreremos *grados* en forma completa, y cada vértice que forma un completo con la solución parcial se agrega a la misma. Al terminar de recorrer *grados* el algoritmo termina, siendo la solución parcial, el resultado final.

A continuación, se muestra el pseudocódigo de la heurística constructiva.

Sea n la cantidad de vértices del grafo.


```

constructivo(matriz_adyacencia)
    grados[n] ← ordenar_grados(matriz_adyacencia)
    solucion[n]
    solucion[0] ← grados[0]
    tamanyo ← 1
    for i to n
        if ¬solucion[i]
            completo ← forma_completo(solucion, i, matriz_adyacencia)
            if completo
                solucion[i] ← true
                tamanyo ← tamanyo + 1
    return tamanyo

```

- **ordenar_grados:** En la implementación, el arreglo *grados* es de tipo tupla donde la primer componente representa el vértice y la segunda el grado. Dicho arreglo está ordenado según la segunda componente en forma decreciente. Lo ordenamos con el algoritmo de Quick Sort de *STL*.

Para setear el grado de un vértice *i* tenemos un contador inicializado en *cero*. Recorremos la columna de la matriz de adyacencia correspondiente a dicho vértice e incrementamos el contador por cada posición (*i, j*) igual *uno*.

- **forma_completo:** Para saber si agregar un vértice *v* determina una solución al problema debemos verificar que forme un completo con los vértices ya incluídos. Para esto recorremos todos los vértices del grafo y para cada uno que pertenezca a la solución parcial chequeamos que sea adyacente a *v*. Si esto ocurre podemos agregar el vértice *v* y agrandar la clique.

3.3. Desventajas

Dado que el vértice de mayor grado pertenece a la clique resultado, si la clique máxima no incluía dicho vértice, la solución puede ser muy mala.

Al ser un algoritmo goloso, en este problema como en tantos otros, no devuelve necesariamente el óptimo. Particularmente, la clique está condi-

cionada al vértice de mayor grado, y no necesariamente la solución óptima lo contiene.(Figura 2)

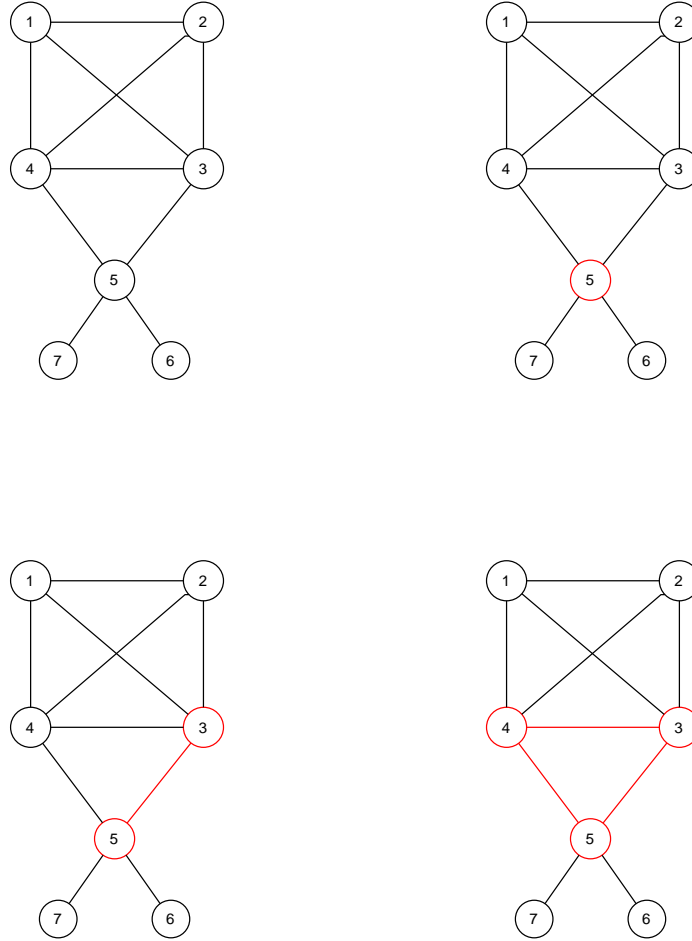


Figura 2: Caso malo para la heurística constructiva

3.4. Complejidad temporal

El algoritmo empieza inicializando el arreglo *grados* lo que tiene un costo de n^2 ya que para cada vértice recorre la columna correspondiente en la matriz de adyacencia (diseñada como un arreglo de arreglos).

Como mencionamos anteriormente, *grados* es ordenado con un algoritmo

de QuickSort dado por la librería estándar de C++. El costo del algoritmo es n^2 .

El algoritmo constructivo, una vez realizadas las operaciones antes mencionadas, entra a un ciclo *for* que itera desde 0 hasta n . En cada iteración, debe en primera instancia, analizar una guarda *if*. De ser *falsa*, procederá a la siguiente iteración del ciclo (teniendo así costo constante). De ser *verdadera* (es decir, el vértice i no forma parte de la solución parcial), analizará si puede formar una nueva clique mayor, ahora con el vértice i (en nuestro pseudo-código, la función *forma_completo* es la encargada de analizarlo). Con el resultado de *forma_completo* la iteración del *for* principal analiza una guarda *if* más de costo constante (en el peor caso realiza dos asignaciones y una suma). Sabemos entonces que el costo de este ciclo será como mínimo n . Analizaremos a continuación el costo de *forma_completo*, para concluir el costo total del algoritmo.

forma_completo, como ya dijimos antes, recorre todos los vértices del grafo analizando que, los vértices que pertenecen a la solución parcial, sean adyacentes al que queremos agregar. Esta función itera por todos los vértices del grafo, teniendo así un costo lineal en función de la cantidad de vértices.

Como el ciclo *for* del algoritmo constructivo, en el peor caso, llamaría una vez por cada una de las n iteraciones a la función *forma_completo*, el costo del ciclo entonces es cuadrático en función de la cantidad de vértices.

Tenemos entonces en el algoritmo constructivo, la inicialización del arreglo con costo de n^2 , el QuickSort de STL, costo n^2 y el ciclo *for* también con costo a lo sumo cuadrático. La complejidad asintótica del problema viene dada entonces por la suma de las complejidades anteriormente descriptas. Podemos afirmar que el costo es $O(n^2)$, ya que $O(n^2 + n^2 + n^2) = O(3 * n^2) = O(n^2)$.

4. Búsqueda local

4.1. Explicación

La heurística de búsqueda local actúa a partir de una solución inicial S , en este caso, a partir de la solución dada por la heurística constructiva. El algoritmo busca en la vecindad de la solución dada, $N(S)$, una solución mejor que ésta. Si no encuentra ninguna mejor, nos encontramos en un óptimo local (de la vecindad) que tomamos como solución del algoritmo.

Decidimos revisar toda la vecindad $N(S)$ en cada iteración y quedarnos con el vecino máximo. De esta manera llegamos al óptimo local por la 'máxima pendiente'. La otra opción era quedarse con el primer vecino que es mayor que el actual, pero la complejidad es la misma y creemos que revisar todos da mejores resultados.

La vecindad $N(S)$ que elegimos en este problema es el conjunto de soluciones tales que no tienen uno y sólo uno de los vértices pertenecientes a S , es decir, $S \setminus \{v\} \cup L$ donde L es un conjunto de vértices tal que $u \in L \iff S \setminus \{v\} \cup \{u\}$ forma un completo.

Para revisar la vecindad, sacamos un vértice de la solución óptima actual e intentamos agregar los vértices que no pertenecen a la clique. Luego, comparamos el tamaño de la clique que logramos contruir de esta manera con el tamaño de la clique correspondiente a la mejor solución vista de la vecindad. Si esta nueva solución es mejor, es decir, el tamaño de la clique es mayor al tamaño de la actual (mejor de la vecindad), dicha solución pasa a ser la mejor de la vecindad. Una vez revisada toda la vecindad comparamos la mejor solución encontrada en dicha vecindad con la mejor solución encontrada hasta el momento. Si es mejor, actualizamos el óptimo actual. (Figura ??)

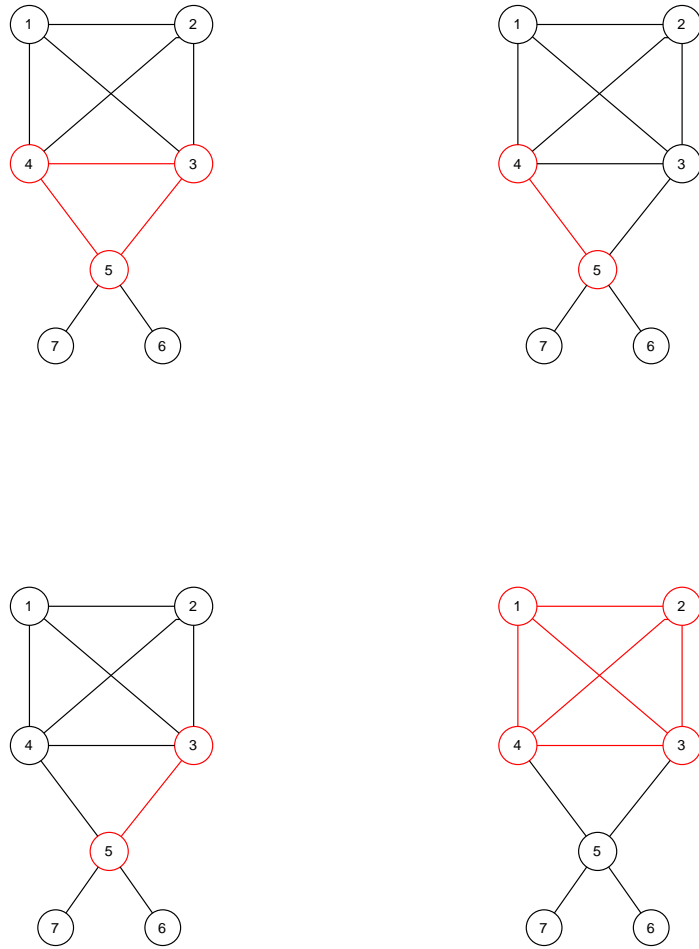


Figura 3: Ejemplo de la heurística de búsqueda local

4.2. Detalles de la implementación

Almacenamos las relaciones entre los vértices en una matriz de $n \times n$, donde n es la cantidad de vértices. Cada posición (i, j) de la matriz contiene un *uno* si existe la arista (i, j) y un *cero* en caso contrario. De esta forma se

le asigna un número a cada vértice.

A continuación, se muestra el pseudocódigo de la heurística de búsqueda local.

Sea n la cantidad de vértices del grafo.

```
busqueda_local(solucion,tamanyo,matriz_adyacencia)
  if  $tamanyo == 1$  or  $tamanyo == n$ 
    return  $tamanyo$ 
  grados[n]  $\leftarrow$  ordenar_grados(matriz_adyacencia)
  inicializar_estructuras
  while  $mejore$ 
    for  $i$  to  $n$ 
      sacar_de_clique(actual,  $i$ )
      agrandar_clique(actual, matriz_adyacencia)
      if  $tam\_actual > tam\_mejor\_vecindad$ 
        actualizar_mejor_vecindad
      else
        reconstruir(actual)
    if  $tam\_mejor\_vecindad > tamanyo$ 
       $mejore \leftarrow true$ 
      actualizar_solucion
  return  $tamanyo$ 
```

La primer cláusula **if** verifica si la solución constructiva encontró la clique tanto completa como la de un elemento. En ambos casos no tiene sentido aplicar la búsqueda local ya que, si encontró el completo, esta solución no podrá ser mejorada, al contener todos los vértices. Si sólo encontró un vértice, implica que el de grado mayor en el grafo es de grado cero, por lo tanto, todos sus vértices son de grado cero.

- **inicializar_estructuras**: Consiste en hacer dos copias de la solución, una para mejor vecindad y otra para actual, y setear dos variables que representan el tamaño de la clique de cada uno de los arreglos.
- **sacar_de_clique**: Esta función setea en *falso* la posición i del arreglo

actual, es decir, excluye el vértice i de la solución actual. Además, decrementa el tamaño de la clique, variable *tam_actual* y resetea una variable *nodo* que indica si despues logra agregar algún vértice, esto sirve para reconstruir la solución en caso de ser necesario (conseguir un tamaño de clique igual al tamaño de la clique desde la que partió).

- **agrandar_clique:** Esta función se encarga de buscar entre los vértices que actualmente no pertenecen a la clique e intenta agregarlos (agrega todo vértice que forma un completo con los ya pertenecientes), con el objeto de conseguir una de mayor tamaño. Por otro lado, setea *nodo* con el valor del último vértice que agrega.
- **reconstruir:** En el caso donde el tamaño de la clique que logro construir es igual al tamaño de la clique inicial, como queremos obtener el mejor de la vecindad, reconstruimos la clique anterior y continuamos, es decir, eliminamos *nodo* y volvemos a agregar i .
- **actualizar_mejor_vecindad y actualizar_solucion:** al encontrar una solución que es mejor a la que el algoritmo posee hasta ese momento, dependiendo del caso, la salva en uno de estos dos arreglos (*mejor_vecindad* o *actual*).

4.3. Desventajas

El algoritmo es estrictamente dependiente de la numeración inicial de los vértices. Para vértices de igual grado, al ordenarlos tiene prioridad el de menor numeración. Sea el siguiente gráfico, un ejemplo en el que esto sucede:

PONER EJEMPLINGUI QUE NO ANDABAAAAA era un caso de 7 nodos q se qda con dos vertices en vez de tres.

La clique dada por la heurística constructiva es el $\langle 1, 2 \rangle$. Cuando el algoritmo (búsqueda local) saca el vértice *uno* de la clique, se mueve a la clique $\langle 2, 4 \rangle$ ya que todos sus adyacentes tienen grado dos y el cuatro es el de menor numeración. Como no logra mejorar (no encuentra una clique que posea al dos y al cuatro, y además algún otro vértice), vuelve a la inicial de $\langle 1, 2 \rangle$. Prueba entonces sacando el dos, sin poder mejorar. Por lo tanto, la solución final es la que viene dada por el algoritmo constructivo.

4.4. Complejidad temporal

En un principio el algoritmo inicializa el arreglo de los *grados* y lo ordena aplicando QuickSort, lo que tiene un costo de n^2 . Luego, obtiene la solución inicial mediante la heurística constructiva que como ya vimos tiene también un costo de n^2 e inicializa los arreglos *actual* y *mejor_vecindad* con costo lineal.

La función *sacar_de_clique* y *reconstruir* tienen costo constante ya que consta sólo de indexaciones, asignaciones y otras operaciones elementales.

La función *agrandar_clique* tiene costo n^2 porque recorre todos los vértices y para cada uno de los que todavía no pertenece a la clique, verifica si puede agregarlo. Para esto, recorre nuevamente todos los vértices y corrobora que sea adyacente a cada uno de los pertenecientes a la clique.

La complejidad final del algoritmo es n^4 porque el ciclo **while** itera a lo sumo n veces (la solución final puede verse mejorada a lo sumo n veces, ver observación) y por cada una de estas el ciclo **for** itera exactamente n veces. En cada iteración del **for** hay una llamada a la función *sacar_de_clique* y *agrandar_clique* las cuales tienen un costo de n^2 . Eventualmente hay una llamada a *reconstruir* lo que no altera la complejidad, ya que su costo es constante.

Observación: El costo lineal en función de la cantidad de vértices del ciclo **while** está acotado por el caso hipotético de comenzar por una clique trivial, y en cada iteración, aumentar la clique en uno. Esto no puede pasar ya que se trataría de un grafo completo, que el algoritmo constructivo encontraría. En ese caso, el algoritmo termina antes de la primera iteración del **while**.

5. Tabu-Search

5.1. Explicación

Finalmente implementamos una metaheurística, es decir, una heurística que guía otra heurística, en este caso la búsqueda local del ejercicio anterior.

El **Tabu-Search** permite evitar que la heurística de búsqueda local se estanque en óptimos locales cuando en realidad fuera de la vecindad existía una solución óptima global (mejor que la local). Para lograrlo, permite al algoritmo perseguir una solución peor que la mejor obtenida mediante búsqueda local, por una cantidad máxima de iteraciones. Pasada esta cantidad, consideramos que el algoritmo ya buscó lo suficiente y la mejor solución encontrada hasta el momento debe ser la óptima.

Para no revisar las vecindades que se revisaron anteriormente (que son muy cercanas a la vecindad actual), cada vez que decidimos movernos a otra vecindad porque tenemos un nuevo aspirante a óptimo (más allá de que sea peor que la mejor solución que encontramos hasta el momento, pero lo llamamos así por su similitud con el mismo en la búsqueda local) prohibimos revertir el cambio que hicimos para llegar del aspirante anterior al nuevo, o sea, prohibimos volver a agregar el vértice que sacamos.

Cada vez que logramos mejorar, es decir, encontramos una clique de tamaño mayor a la máxima clique vista hasta el momento actualizamos la solución. Repetimos este procedimiento hasta agotar la máxima cantidad de iteraciones permitidas sin mejorar. Una vez que ocurre esto, el algoritmo termina, siendo la solución final la solución actual, es decir, la clique de mayor tamaño que logro encontrar.

5.2. Detalles de la implementación

Almacenamos las relaciones entre los vértices en una matriz de $n \times n$, donde n es la cantidad de vértices. Cada posición (i, j) de la matriz contiene un *uno* si existe la arista (i, j) y un *cero* en caso contrario. De esta forma se le asigna un número a cada vértice.

A continuación, se muestra el pseudocódigo de la heurística de búsqueda tabú.

Sea n la cantidad de vértices del grafo.

```
busqueda_tabu(solucion,tamanyo,matriz_adyacencia)
  if  $tamanyo == 1$  or  $tamanyo == n$ 
    return  $tamanyo$ 
   $grados[n] \leftarrow ordenar\_grados(matriz\_adyacencia)$ 
  while mejore
    for  $c$  to  $tamanyo$ 
      inicializar_estructuras
      rotar_clique
      while puedo seguir
        sacar_de_clique
        poner_tabu
        formar_completo(actual, matriz_adyacencia)
        if  $tam\_actual > tamanyo$ 
          agrandar_clique(actual, lista_tabu)
          actualizar_solucion
           $c \leftarrow tamanyo$ 
        else
          restar_tabu
    return  $tamanyo$ 
```

En la implementación mantenemos la solución actual tanto en una lista de vértices. La lista inicia ordenada de mayor a menor según los grados, esto lo hacemos para empezar a sacar desde el de menor grado ya que consideramos que es el que tiene más probabilidades de estar condicionando la clique. Además, tenemos un arreglo de tipo bool de tamaño n donde el índice representa a los vértices y esta seteado en *verdadero* si y sólo si el vértice pertenece a la clique actual. Mantenemos ambas estructuras porque usamos la lista para determinar el orden en que se eliminan (rotarla tiene costo constante) y el arreglo para verificar la pertenencia de un vértice a la clique (ya que esta operación en esta estructura de datos tiene costo constante). Por otro lado, mantenemos un arreglo de vértices (los índices representan los vértices), donde para cada uno guardamos la cantidad de iteraciones que falta para que deje de ser tabú una vez que son eliminados de la clique actual (el algoritmo lleva la cuenta de las iteraciones). Esto lo hacemos para no revertir los cambios recientemente hechos y cuando revisamos las vecindades

de un aspirante, evitamos aquellas donde la modificación implica agregar un vértice tabú.

La primer cláusula **if** verifica si la solución de búsqueda local encontró la clique tanto completa como la de un elemento. En ambos casos no tiene sentido aplicar el tabú search ya que, si encontró el completo, esta solución no podrá ser mejorada, al contener todos los vértices. Si sólo encontró un vértice, implica que el de grado mayor en el grafo es de grado cero, por lo tanto, todos sus vértices son de grado cero.

El valor de verdad de la guarda del **while** *puedo_seguir* viene dado por la conjunción entre $tam_actual \neq 1$, $\neg meiore$ e $iteracion < n$. Pedimos que el tamaño de la clique actual sea distinto de *uno* ya que nos interesa movernos a soluciones vecinas. Si el tamaño es *uno* en esa iteración saca el último vértice de la clique por lo que se pierde referencia a la misma moviéndose inmediatamente al primer vértice segun la numeración que no esté tabú. Por otro lado, el **while** itera mientras no logre mejorar para forzar la salida del ciclo cuando encuentre una clique de mayor tamaño que la actual y así empezar a sacar vértices desde la primer rotación (ya que también se fuerza la salida del **for**). La última condición es para asegurar la salida del ciclo, ya que podría no mejorar nunca y ciclar entre diferentes cliques. Además, esto determina la cantidad de iteraciones que le permitimos buscar sin lograr mejorar, es un parámetro que ajustamos de la siguiente manera AJUSTEMOSLO ALGUN DIAAAAAA!!!!!!ukuyfuyjtjft5umj76fu7666m,6b,ib7,ki7k,i87fk7k,KUFMKUTJYTDMHYTMJU

- **rotar_clique:** Dado que fijar la pertenencia de un vértice a la clique condiciona el resultado final, el orden en que se eliminan los vértices puede hacer la diferencia entre un buen resultado y uno malo, a pesar de encontrar un buen criterio para hacerlo. Por este motivo, decidimos empezar eliminando de menor a mayor grado, y en cada iteración rotar la lista para sacar los vértices en otro orden.
- **sacar_de_clique:** Esta función saca de la lista el último elemento (el de menor grado entre los vértices con una misma 'antigüedad' en la clique) y setea en *falso* la posición correspondiente en el arreglo (dejando tabú la operación inversa (agregarlo a la clique) tantas iteraciones como el tamaño de la clique con la que empieza a sacar), es decir, excluye el vértice de la solución actual. Además, decrementa la variable que indica el tamaño de la clique.

LA CANTIDAD DE ITERACIONES QUE QUEDA TABU ES UN

PARAMETRO ARREGLARLO

- **formar_completo:** Esta función se encarga de buscar entre los vértices que actualmente no pertenecen a la clique e intenta agregarlos (agrega todo vértice que forma un completo con los ya pertenecientes), con el objeto de conseguir una de mayor tamaño. Para saber si agregarlo determina una solución al problema debemos verificar que forme un completo con los vértices ya incluidos. Para esto recorreremos todos los vértices del grafo y para cada uno que pertenezca a la solución parcial chequeamos que sea adyacente al que pretendemos agregar. Si esto ocurre podemos agregarlo y agrandar la clique. Elegimos el vértice a agregar de mayor a menor grado.

Observaciones:

- Al agregar condicionamos la clique resultante al igual que pasa al sacar sin hacer rotaciones (depende del orden en que lo hagamos la calidad de la solución). Es decir, encontrar una mejor solución depende del orden en que agreguemos los vértices, podríamos también hacer rotaciones para agregar pero esto aumentaría en n la complejidad. Buscando un equilibrio entre eficiencia y calidad de la solución, decidimos que hacer ambas rotaciones (agregar, sacar) tenía una complejidad mayor a la que pretendemos aceptar, no hacer ninguno implica perdernos de encontrar mejores soluciones y obtener así soluciones muy precarias. Entonces elegimos arbitrariamente hacer las rotaciones sólo para sacar.
 - Si el algoritmo vuelve a la solución inicial y todavía le restan iteraciones del **while** anidado, queremos evitar que repita exactamente el mismo procedimiento pasando nuevamente por soluciones ya visitadas por lo que forzamos la salida y aplicamos una rotación a dicha solución para explorar nuevas posibilidades.
- **agrandar_clique:** Esta función itera los vértices que están en la lista tabú e intenta agregarlos a la clique actual. Esto es porque como ya conseguí una solución mejor lograr agregar algún vértice que esta tabú contribuye aún más a la solución.
 - **restar_tabu:** Para cada vértice tal que tiene tabú mayor a *cero* decrementa la cantidad de iteraciones que va a permanecer tabú. Si al decrementarlo deja de ser tabú lo elimina de *lista_tabu*.

Tanto en *formar_completo* como en *agrandar_clique* los vértices que agregamos a la lista de la clique actual los ponemos al principio de la misma, es una forma de poner tabú al menos tantas iteraciones como vértices había en la clique previo a agregarlos la operación inversa, sacarlos (ya que se saca siempre el último de la lista).

5.3. Desventajas

La numeración de los nodos toma un papel crucial a la hora de obtener una solución, es así que en casos patológicos puede pasar que:

El algoritmo visite soluciones anteriormente exploradas ya que como se mueve constantemente de solución, puede eventualmente volver a la solución inicial (ejemplo: la búsqueda local nos da un clique de dos vértices que se encuentran en un ciclo, y este ciclo está unido a un completo de tres o más vértices). Si los vértices del completo tienen mayor numeración a los del ciclo, el algoritmo ciclaría tomando en cada iteración (mientras no se cumpla la restricción de iteraciones pasadas como parámetros) como solución dos de los vértices pertenecientes al ciclo, sin ver el completo.

Para evitar pasar varias veces por las mismas soluciones, por cada iteración del `while` anidado se verifica si vuelve a la solución inicial, de ser así, se fuerza la salida y se procede a la siguiente rotación de la clique. A pesar de solucionar este problema para casos particulares como el que mencionamos (donde se parte de una solución y a través eliminar y agregar vértices se llega nuevamente a la solución inicial), pueden existir casos donde se repitan soluciones desde distintas rotaciones de la clique inicial. Estos casos no son advertidos por el algoritmo.

5.4. Complejidad temporal

En un principio el algoritmo inicializa el arreglo de los *grados* y lo ordena, lo que tiene un costo de n^2 ya que se utiliza el algoritmo de ordenamiento *QuickSort*. Luego, obtiene la solución inicial mediante la heurística de búsqueda local que como ya vimos tiene un costo de n^4 .

La función *rotar_clique*, *sacar_de_clique* y *poner_tabu* tienen costo constante ya que constan sólo de indexaciones, asignaciones y operaciones elementales sobre listas.

La función *formar_completo* itera por todos los vértices y para cada uno de estos verifica si es factible agregarlo a la clique, lo cual tiene un costo de

n^2 ya que dicha verificación consiste en recorrer todos los vertices del grafo y ver que el vértice que se pretende agregar es adyacente a cada uno de estos. Además, se concatena la lista de los vértices agregados con la lista de la clique actual lo que tiene un costo constante.

La función *agrandar_clique* tiene costo n^2 ya que no es más que una llamada a la función *formar_completo* con todos los vértices permitidos, es decir, previo a la llamada se resetea el arreglo *tabu*.

La cantidad de iteraciones del primer ciclo **while** se puede acotar por n (seria un caso hipotetico en el que la mejora sea sólo de un vértice, es decir, se inicie con la clique trivial (tamaño *uno*) y en cada iteración de este ciclo se logre incrementar en uno el tamaño de la clique). En cada una de estas iteraciones hay un ciclo **for** que se ejecuta a lo sumo tantas veces como el tamaño de la clique, también puede ser acotado por n , dentro de este ciclo se resetean las estructuras (la lista y los arreglos) lo que tiene un costo lineal. Además, hay un ciclo **while** (dentro del **for**) que se ejecuta a lo sumo n veces y en cada una de las iteraciones hay una llamada a la función *sacar_de_clique* con costo constante y *formar_completo* con costo n^2 , en caso de lograr mejorar hay una llamada a la función *agrandar_clique* con costo n^2 , en caso contrario, hay una llamada a la función *restar_tabu* de costo lineal (recorre todos los vértices decrementando la cantidad de iteraciones que permanecieran tabú). Entonces la complejidad del **while** anidado es: $O(n * (O(\text{sacar_de_clique}) + O(\text{formar_completo}) + O(\text{agrandar_clique}) + O(\text{restar_tabu}))) = O(n * (1 + n^2 + n^2 + n)) = O(n^3)$.

Finalmente, se deduce que la complejidad final del algoritmo es n^5 ya que acotamos tanto la cantidad de iteraciones del ciclo **while** como del **for** por n , entonces tenemos $O(n * n * O(\text{while_anidado})) = O(n^5)$.

Observación: En la función *formar_completo* para ver si es factible agregar un vértice a la clique en vez de recorrer todos los vértices del grafo como hacemos, se podría sólo recorrer los vértices pertenecientes a la clique ya que están en una lista, pero la complejidad sería la misma y deberíamos ir agregando a la lista a medida que se decide agregar un vértice (para considerarlo en las proximas verificaciones) y esto no nos permite hacerlo en el orden deseado (ya que queremos agregar al principio de la lista de la clique actual de menor a mayor grado y los recorreremos de mayor a menor grado). Como ya mencionamos esto no mejoraria la complejidad asintotica y aumentaria la complejidad de comprensión del código.

6. Resultados

6.1. Parametros de la heurística tabú

6.2. Comparacion de tiempos

6.3. Comparacion de calidad

7. Mediciones

Para contar la cantidad aproximada de operaciones definimos una variable inicializada en *cero* la cual incrementamos luego de cada operación. En el código puede verse la llamada a la función $O(x)$, donde x es un valor positivo entero, el cual es sumado a la cantidad de operaciones parcial para una vez terminado el algoritmo, poseer la cantidad de operaciones que realiza para una instancia dada.

Preferimos contar operaciones en vez de medir tiempo porque a pesar de que es aproximado el resultado, el error es siempre el mismo y así podemos hacer una mejor comparación entre las instancias.

Midiendo tiempo, el error para cada instancia varía, ya que es el sistema operativo el que ejecuta nuestro programa, al "mismo tiempo" que otras tareas.

8. Compilación y ejecución de los programas

Para compilar los programas se puede usar el comando `make` (Requiere el compilador `g++`). Se pueden correr los programas de cada ejercicio ejecutando `./exacto`, `./constructivo`, `./busqueda_local` y `./tabu_search` respectivamente.

Los programas leen la entrada de `stdin` y escriben la respuesta en `stdout`. Para leer la entrada de un archivo `Tp1EjX.in` y escribir la respuesta en un archivo `Tp1EjX.out` se puede usar:

```
./(ejecutable) <Tp1EjX.in >Tp1EjX.out
```

Para contar la cantidad de operaciones: `./(ejecutable) count`. Devuelve para cada instancia el tamaño seguido de la cantidad de operaciones de cada instancia.