Modelos y Optimización I

Facundo Torraca

Junio 2020

1 Introducción

Durante el transcurso de este informe, se relatara con detalles como fue el procedimiento utilizado para encontrar una solucion aproximada a un problema de decisión calificado como *NP-Hard*, asi como también una explicacion detallada de como se logro llegar a los métodos de resolución.

El problema que motivo la implementación de ciertos algoritmos mencionados mas adelante en el informe consiste en determinar como agrupar un conjunto de prendas de ropa en diferentes lavados, de manera que se logre minimizar el tiempo total de lavado. Cada prenda posee ciertas restricciones con otras prendas, es decir, no es posible agruparlas en el mismo lavado. Como hipótesis inicial, se asumió que no existe una cantidad mínima o máxima de prendas por lavado.

2 Modelización

En esta sección explicaremos como se realizo la modelización, ya que fue un factor común en ambos problemas planteados.

La idea principal fue la creacion de un grafo no dirigido G, en que los vértices V(G) eran las prendas. Estos vértices estaban conectados mediante el conjunto de aristas E(G) tales que:

$$(u, v) \in E(G) \iff \text{u es una prenda incompatible con v}$$

A su vez, cada vértice de $v \in V(G)$ posee un peso $w \in W(V)$, tal que w es el tiempo de lavado del vértice v.

Este concepto matemático se represento mediante una matriz binaria en que 1 representa una arista y 0 que no hay conexion y una lista ordenada en que cada posición representa el peso del vértice.

Todos los tiempos mencionados son estimativos, ya que se considera que los algoritmos no fueron corridos en condiciones optimas. El tiempo esta medido en base a una implementación en *Python 3.7.5*, en un computador con procesador *Intel(R) Core(TM) i5-2467M CPU @ 1.60GHz* y 4 gigabytes de memoria.

3 Búsqueda de la solucion

3.1 Primer Problema

La primera aproximación de solucion fue probada con una versión reducida del problema, con apenas 20 prendas y 210 restricciones. El pequeño numero de prendas y restricciones permitía buscar la solucion mediante algún algoritmo exacto mediante alguna técnica de *bactracking* o *fuerza bruta*.

Si bien como mencionado anteriormente el tamaño del problema no imposibilitaba la resolución mediante algoritmos exactos, se considero que era mas relevante la implementación de un algoritmo que permitiese luego escalar a problemas de mayor tamaño. Por ese motivo, se decidió implementar una heurística *greedy* para la elección de la siguiente prenda.

Siendo el modelo un grafo, la heurística genera n conjuntos C de vértices que no estan conectados entre si, es decir, son compatibles, donde n representa el numero de lavados. Por lo tanto:

$$u, v \in C \to (u, v), (v, u) \notin E(G)$$

La primera heurística planteada y la que dio mejores resultados para este problema consistía en cada iteración ir eligiendo el vértice mas pesado, es decir, la prenda que mas tiempo demora en lavarse, y tratar de agregarlo a algún subconjunto de vértices ya creado previamente o crear uno nuevo caso el vértice elegido sea incompatible en todos los conjuntos.

El pseudo-código de seria:

Algorithm 1 Heurística: Próximo vértice mas pesado

```
procedure AgruparVertices
    grupos ← {}
    while V(G) ≠ ∅ do
        v := proximoVerticeMasPesado(V(G))
        while v no agregado do
            C := proximoGrupo(grupos)
            if C ≡ None then
                grupos ← grupos ∪ {v}
            else
                if esCompatible(v, C) then
                    C ← C ∪ v
    return grupos
```

Si bien es una heurística muy simple, los resultados fueron muy aceptables, obteniendo un tiempo de lavado de 63 en apenas 0.44 milisegundos, cuando el óptimo es de 61. Este resultado se logro posteriormente mejorarlo (de 63 a 62), pero no significativamente, simplemente haciendo que la funcion *proximoVerticeMasPesado* retornara el vértice con mayor cantidad de restricciones en caso de que el peso coincidiera.

Otras heurísticas fueron probadas, tales como elegir los vértices en orden decreciente de grados, por orden creciente de grados y por el menor peso primero, pero ninguna obtuvo mejores resultados.

3.2 Segundo Problema

,

Dado un nuevo problema, era necesario ver como funcionaba el algoritmo programado previamente. Este problema imponía un nuevo desafió, ya que su tamaño no permitía que todas las soluciones planteadas pudieran llegar a un resultado en un tiempo razonable. El problema consistía en este caso de 200 prendas con 38190 restricciones. Claramente una solucion por fuerza bruta ya no era una opcion frente a las millones de combinaciones posibles.

Felizmente, comenzamos nuestra implementación con el primer problema con un enfoque *greedy*. Esto permitió correr exactamente el mismo algoritmo que corrimos la primera vez con tiempos mas que aceptables y un resultado para nada decepcionante, dado que la heurística es extremadamente simple y rápida. En apenas 45 milisegundos se obtuvo un tiempo de lavado de 476. De igual manera, rápidamente surgieron alternativas muy superiores, por lo que se decidió migrar a alguna solucion un poco mas interesante.

Una solucion propuesta por uno de los compañeros de clase fue la utilización de un algoritmo de coloreo de grafos con un enfoque también greedy, permitiendo asi continuar aun con tiempos de ejecucion razonables, llamado *DSatur*.

3.2.1 DSatur para el coloreo de grafos

El algoritmo de *DSatur*, creado por Daniel Brélaz en 1979 fue el primer algoritmo documentado para el coloreo de grafos de forma *greedy*, permitiendo la resolver el problema del coloreo, el cual esta clasificado como *NP-Completo* en un tiempo razonable, comparado con algunos otros algoritmos de coloreo exactos como *Randall-Brown*.

EL algoritmo de *DSatur* se basa en el concepto de *grado de saturación* de un vértice, que corresponde a la cantidad de colores diferentes a los que un vértice es adyacente en un grafo parcialmente coloreado.

Mediante el concepto de *grado de saturación*, el algoritmo colorea primeros los vértices con mayor grado de saturación. El vértice inicial se suele elegir el vértice de mayor grado, asi como también cuando todos los vértices restantes tienen grado de saturación cero.

Si bien es un algoritmo *greedy*, esta demostrado que es un algoritmo exacto para grafos bipartitos.

El pseudo-código del algoritmo seria:

Algorithm 2 DSatur

```
procedure DSATUR(G)
    colores ← {}
    v := obtenerVerticeMayorGrado(V(G))
    colorear(v, colores)
    while not todosColoreados(V(G)) do
        v := proximoVerticeMayorGradoSaturacion(V(G))
        if v ≡ None then
            v := proximoVerticeConMayorGrado(V(G))
        colorear(v, colores)
```

Un detalle importante a mencionar es que la funcion *colorear* buscara colorear el vértice siempre con el color "mas pequeño" (en caso que se lo represente con números) y caso no sea posible, agregara un nuevo color al set de colores.

3.2.2 Aplicando DSatur al problema

Habiendo explicado previamente el funcionamiento del algoritmo, detallaremos su utilización para resolver el problema.

Si aplicamos el algoritmo al grafo generado por las prendas, claramente estaremos asignando un color a cada lavado, ya que las prendas adyacentes, es decir que son incompatibles, no pueden tener el mismo color. Si bien el algoritmo ignora los pesos, tratara de minimizar la cantidad de lavados posibles, o dicho de otra manera, minimizar la coloración del grafo, resultando asi en una muy buena aproximación de la solucion del problema.

Si bien no hemos encontrado la solucion óptima al problema, mediante algunos cálculos que explicaremos posteriormente de las cotas inferiores, asi como también en base a los resultados obtenidos con la heurística del primer problema, los resultados de aplicar DSatur al segundo problema fueron mas que satisfactorios, obteniendo el mejor resultado de 272 con un tiempo de 81312.87 milisegundos. Si bien los tiempos de ejecucion fueron de hasta 150 veces mas lentos que con otras heurísticas, se logro reducir el tiempo de lavado en hasta un 25% del resultado inicial.

4 Cotas inferiores

Una vez planteado el problema y encontrado una solucion, decidimos buscar una cota inferior para la solucion exacta del problema, es decir, un valor que nos indique un valor del cual es imposible bajar.

Encontrar una cota inferior tampoco es un problema trivial, y al igual que encontrar la solucion exacta, esta clasificado como un problema del tipo *NP-Hard*. Por ese motivo, se decidió implementar un nuevo algoritmo basándonos en la metodología *greedy*.

Es necesario tener extremo cuidado al encontrar una cota inferior, ya que por mas que no se logre encontrar el valor óptimo (lo que es equivalente a resolver el problema), se debe estar seguro que la solucion nunca podrá ser menor que la cota encontrada, por lo que los vértices algoritmos greedy utilizados no deben tratar de resolver el problema.

El primer algoritmo que utilizamos fue un algoritmo con una heurística muy similar al utilizado por primera vez en el primer problema. Esta heurística consistía en buscar los vértices del grafo por orden decreciente de peso y tratar de armar un subconjunto de vértices que todos estén conectados entre si, o lo que quiere decir, un subconjunto de prendas todas incompatibles entre si.

Volviendo al planteo de un grafo, el algoritmo buscara entonces encontrar el subgrafo $G \subset Kn$ mas pesado, también llamado *clique*.

El pseudo-código del algoritmo seria:

Algorithm 3 Cota Inferior con heurística: Próximo vértice mas pesado

procedure ENCONTRARCLIQUE
    $clique \leftarrow \{\}$
    while $V(G) \neq \emptyset$ **do**
        $v :=$ *proximoVerticeMasPesado(V(G))*
        $Nv := obtenerVecinos(v)$
        if $clique \subset Nv$ **then**
            $clique \leftarrow clique \cup v$
    return *clique*

Este algoritmo encontró una cota inferior de 208 para el segundo problema en apenas 27.53 milisegundos. Lo que es un resultado muy bueno dado el poco tiempo de ejecucion que requiere y su simplicidad. Para el primer problema tampoco decepciono, encontrando una cota inferior de de 58 en 0.17 milisegundos, un resultado excelente sabiendo que el resultado óptimo era de 61.

Llegado este punto, se encontró un nuevo algoritmo que prometía encontrar el *clique* mas pesado posible en un numero de iteraciones dado y de manera *greedy*, lo que era casi imprescindible dada la magnitud del problema. Este algoritmo es conocido como *FastWClq*.

4.1 FastWClq y su idea greedy

El problema MWCP (*maximum weight clique problem*) es mucho mas complicado que el problema MCP (*maximum clique problem*) y las técnicas para MCP no son aplicables o son muy poco efectivas para el problema MWCP.

Desde una mirada global, el algoritmo trabaja mediante un ciclo principal que frena cuando el numero de iteraciones llega a su fin o cuando una solucion exacta es encontrada. Dentro del ciclo, se construye un *clique*, extendiendo un set de vértices con vértices de otro set de candidatos. Se utilizan algunas técnicas, que explicaremos posteriormente, para evitar la construcción de *cliques* que se pueden comprobar previamente que no van a superar en peso al mas pesado creado hasta el momento.

El pseudo-código del algoritmo es:

Algorithm 4 Fast Weight Clique

procedure FASTWCLQ(G, N)
 $set\_inicial \leftarrow V(G)$
 $mejor\_C := \emptyset$
 $k := k_0$
 $iter := 0$
 while $iter < n$ **do**
  if $set\_inicial \equiv \emptyset$ **then**
   $set\_inicial \leftarrow V(G)$
   $ajustarNumeroBMS(k)$
  $u :=$ sacar un vértice random de set_inicial
  $C := \{u\}$
  $candidatos := vecinos(u)$
  while $candidatos \neq \emptyset$ **do**
   $v := elegirVerticeParaAgregar(candidatos, k)$
   if $peso(C) + peso(v) + peso(vecinos(v) \cap candidatos) \leq peso(mejor\_C)$ **then**
    break
   $C := C \cup \{v\}$
   $candidatos := candidatos - \{v\}$
   $candidatos := candidatos \cap vecinos(v)$
  if $peso(C) > peso(mejor\_C)$ **then**
   $mejor_C := C$
   $G := reducirGrafo(G)$
   $set\_inicial \leftarrow V(G)$
   if $estaVacio(G)$ **then**
    return $mejor_C$
 return $mejor_C$

Una de las funciones mas importantes del algoritmo es la funcion *elegirVerticeParaAgregar*. Esta funcion elige el siguiente vértice para agregar al *clique* mediante una estimación del beneficio del agregado del vértice y también mediante una heurística de BMS dinámico (ambos conceptos estan explicados en detalle en las secciones 4.1.1 y 4.1.2).

Un pseudo-código de la funcion se puede ver a seguir:

Respecto a la reducción del grafo, la idea es que mediante ciertas reglas, se pueda reducir el grafo manteniendo la solucion óptima. Esto es muy deseable, ya que permitirá acelerar los tiempos de ejecucion. El detalle de como es la reducción del grafo se puede en detalles en la sección 4.1.3.

Algorithm 5 Elegir vértice para agregar

procedure ELEGIRVERTICEPARAAGREGAR(CANDIDATOS, K)
    if $|candidatos| < k$ **then**
        return un vértice v $\in$ *candidatos*con el mayor beneficio
    $v :=$ vértice random $\in$ candidatos
    for $iter := 1$ **to** $K - 1$ **do**
        $u :=$ vértice random $\in$ candidatos
        if $beneficio(u) > beneficio(v)$ **then**
            $v := u$
    return v

4.1.1 Beneficio de agrega un vértice

El beneficio del agregado de un vértice se define como:

$$beneficio(v) = peso(C_f) - peso(C)$$

Donde C_f es el *clique* final que se logra con el agregado de v y C es el *clique* actual.

Como es imposible determinar el peso de C_f, nos basamos en dos consideraciones para calcular el beneficio:

- Si agregamos el vértice v al clique C, el peso de C se incrementa en $peso(v)$, por lo que seria la cota inferior del clique seria $peso(C) + peso(v)$
- Si agregamos el vértice v al clique C, el incremento máximo posible del peso de C seria $peso(v) + peso(vecinos(v) \cap candidatos$, por lo que la cota superior del clique seria $peso(C) + peso(v) + peso(vecinos(v) \cap candidatos)$.

Entonces aproximamos el beneficio como:

$$\widehat{beneficio}(v) = \frac{peso(v)+peso(v)+peso(vecinos(v)\cap candidatos)}{2}$$

4.1.2 Heurística de BMS dinámico

La elección de vértice es además guiada por un BMS dinámico (*best from multiple selection*). La heurística BMS original es una estrategia probabilística que retorna el mejor elemento de muchas muestras. Otra gran ventaja es que la heurística BMS dinámica nos permite controlar el *greediness* del algoritmo.

En al algoritmo, se comienza con un valor de k bajo, de manera que trabaje mas rápido y a medida que el *set_inicial* se achica, el valor de k crece.

4.1.3 Reducción del Grafo

La **regla** utilizada para la reducción del grafo G consiste en: Dado un grafo G = (V, E) y un *clique* C talque que $C \in G$, si existe una cota superior $UB(v)$ / $UB(v) \leq peso(C)$, entonces removemos del grafo el vértice v, asi como todas sus aristas incidentes.

El algoritmo utiliza dos cotas superiores, UB_0 y UB_1. La segunda requiere un poco mas de tiempo computacional pero es mas ajustada. Por ese motivo, utilizamos primeramente UB_0 y caso no podamos borrar el vértice, aplicamos la regla utilizando UB_1. Las formulas para calcularlas son:

- $UB_0 = peso(vecinos(v))$
- $UB_1 = peso(v) + peso(u) + peso(vecinos(v) \cap vecinos(u))$ donde u es el vecino mas pesado de v.

El pseudo-codigo del algoritmo de reduccion es el siguiente:

Algorithm 6 Reducción del grafo G

procedure ReducirGrafo(G, C)
    $ColaBorrar \leftarrow \emptyset$
    for all $v \in V(G)$ **do**
        if **then**$UB_0(v) \leq peso(C)$ **or** $UB_1(v) \leq peso(C)$
            $ColaBorrar.agregar(v)$
    while $ColaBorrar \neq \emptyset$ **do**
        $u := ColaBorrar.desencolar()$
        borrar u y sus aristas incidentes de G
        for all $v \in V(G)$ **do**
            if **then**$UB_0(v) \leq peso(C)$ **or** $UB_1(v) \leq peso(C)$
                $ColaBorrar.agregar(v)$
     return G

4.2 Utilizando FastWClq para encontrar la cota inferior

Luego de la explicación detallada del funcionamiento del algoritmo, mencionaremos como fue utilizado para encontrar una cota inferior y sus resultados.

Cuando aplicamos el algoritmo *FastWClq* a nuestro grafo de prendas, lo que obtenemos es el grafo conexo mas pesado encontrado, o dicho de otra manera, el conjunto de prendas incompatibles entre si que demoran mas tiempo en ser lavadas. Obviamente esto es una cota inferior ya que si todas las prendas son incompatibles, todas deben estar en lavados distintos.

La máxima cota inferior obtenida fue de 225, en un tiempo de 1363682.78 milisegundos y un tope de w0000 iteraciones, siendo el tiempo un mas de un orden de magnitud superior al tiempo que demora en obtener una solucion el algoritmo *DSatur*. Sin embargo, continúa siendo un tiempo realmente aceptable dado el tamaño del problema.

5 Conclusión

References

[1] Shaowei, Cai and Jinkun, Lin (2014). *Fast Solving Maximum Weight Clique Problem in Massive Graphs*, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 568–574, 7.
https://www.ijcai.org/Proceedings/16/Papers/087.pdf

[2] Brélaz, Daniel (1979). *New Methods to Colorthe Vertices of a Graph*, Commun. ACM,*22*(4),251–256, 6.
https://dl.acm.org/doi/pdf/10.1145/359094.359101