Algorítmica - Practica 4: Backtracking

A. Herrera, A. Moya, I. Sevillano, J.L. Suarez

16 de mayo de 2015

Contents

1 Organización de la práctica

La práctica 4 consiste en el desarrollo de algoritmos basados en backtracking que consigan la solución óptima de los problemas propuestos. Usualmente las soluciones obtenidas serán exponenciales ya que intentaremos resolver problemas NP. La poda nos permitirá conseguir en la práctica resultados aceptables.

Nuestro grupo debe resolver el problema 5.

Para cada problema resuelto se sigue la siguiente estructura:

- Enunciado del problema
- Resolución teórica del problema (con una subsección por algoritmo)
- Análisis empírico. Análisis de la eficiencia híbrida

En este último apartado se proporcionan gráficas con los resultados de los algoritmos y un análisis de la eficiencia híbrida para los mismos.

Los algoritmos se han ejecutado sobre un ordenador con las siguientes características:

- **Marca:** Toshiba
- **RAM:** 8 GB
- **Procesador:** Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz

El código, los resultados de las ejecuciones, las gráficas y los pdf asociados se pueden encontrar en GitHub.

2 Problema 5: Estación de ITV

2.1 Enunciado del problema

Una estación de ITV consta de m líneas de inspección de vehículos iguales. Hay un total de n vehículos que necesitan inspección. En función de sus características, cada vehículo tardará en ser inspeccionado un tiempo t_i, $i = 1, \ldots, n$. Se desea encontrar la manera de atender a los n vehículos y acabar en el menor tiempo posible. Diseñar e implementar un algoritmo vuelta atrás que determine cómo asignar los vehículos a las líneas. Mejoradlo usando alguna técnica de poda. Realizar un estudio empírico de la eficiencia de los algoritmos.

2.2 Solución teórica

La formulación del problema recuerda al problema 5 de la práctica de algoritmos voraces. Tenemos varias máquinas que realizan trabajos que requieren un tiempo predeterminado. Este tipo de problemas suelen clasificarse como **scheduling problems**. Sin embargo, la función objetivo a optimizar es diferente. En el problema de la práctica anterior se pedía optimizar el tiempo medio de espera de los clientes mientras que en este caso debemos realizar todos los trabajos en el menor tiempo posible.

El problema a resolver es NP Hard [1]. Por tanto, queremos conseguir la mejor solución exponencial posible. Antes de nada, si $n <= m$ es fácil darse cuenta de que la solución será el máximo de los t_i pues basta asignar un trabajo a cada máquinal. Podemos suponer entonces que $n > m$.

Hacemos la siguiente observación, no nos importa el orden en el que una máquina realice sus trabajos asignados ya que en cualquier caso el tiempo en el que la máquina está trabajando es el mismo. Esto da lugar a la siguiente proposición:

Proposición 1.
El número de formas en las que se pueden asignar los trabajos a las máquinas es m^n.

Demostración. Cada posible asignación podemos verla como que a cada trabajo se le asigna una máquina, pudiendo asignar una máquina a varios trabajos. Esto es, para cada trabajo elegimos entre m posibilidades disponibles. En total, m^n asignaciones.

■

A raíz de esta proposición aparece un algoritmo de backtracking directo que consiste en recorrer todas las asignaciones posibles.

$ **Algoritmo 1.** Primer algoritmo de backtracking con eficiencia $\theta(m^n)$. $

```
# Parametros:
# - k : índice del trabajo a asignar.
# - tiempos : vector con los tiempos de los m trabajos.
# - solucion_actual : vector con el tiempo de trabajo
#   asignado a cada máquina.
# - max_tiempo : Máximo de solucion_actual
def algoritmo1(k,tiempos, solucion_actual, max_tiempo):
  if k < len(tiempos):
    sol = 0
    for i in range(0,len(solucion_actual)):
      solucion_actual[i] += tiempos[k]
      sol = min(sol, algoritmo1(k+1,tiempos, solucion_actual, \
          max(max_tiempo, solucion_actual[i])))
```

[1] Scheduling Job Problem

```
      solucion_actual[i] -= tiempos[k]
    return sol
  else:
    return max_tiempo
```

Basta llamar al algoritmo de la siguiente forma: `algoritmo1(0,tiempos,[0 for i in range(0, m)], 0)`. Realiza una búsqueda en profundidad del árbol con todas las soluciones posibles. Cada nodo consta de m hijos, uno por máquina a la cual se puede asignar el trabajo. Su eficiencia es claramente $\theta(m^n)$. Sin embargo, también es manifiestamente mejorable. Será nuestro punto de partida pero iremos mejorándolo reduciendo el espacio de soluciones a comprobar y aplicando criterios de poda.

Debemos observar que no nos importa cuál sea la máquina que realiza un determinado conjunto de trabajo. Por ejemplo, si tuviésemos dos máquinas, A y B, y dos trabajos, 1 y 2, da igual que la máquina A haga el trabajo 1 y la máquina B haga el trabajo 2 a que A realice el 2 y B el 1. Las máquinas son igual de eficientes por lo que el tiempo en el que se habrán terminado todos los trabajos será el mismo en ambos casos.

Por ejemplo, si cada máquina hace un conjunto de trabajos no vacío podemos aplicar las $m!$ permutaciones posibles obteniendo soluciones que terminan en el mismo tiempo pero que son calculadas por separado en el Algoritmo 1. Tratamos de evitar de la siguiente forma. Si quedan máquinas sin una tarea por asignar no llamamos al algoritmo recursivamente para cada una de estas máquinas sino solo para la primera de ella pues los otros estados son equivalentes. Este razonamiento conduce al Algoritmo 2.

Algoritmo 2. Mejora sobre el Algoritmo 1 que evita soluciones equivalentes.

```
# Parametros:
# - k : índice del trabajo a asignar.
# - tiempos : vector con los tiempos de los m trabajos.
# - solucion_actual : vector con el tiempo de trabajo
#   asignado a cada máquina.
# - max_tiempo : Máximo de solucion_actual
def algoritmo2(k, tiempos, solucion_actual, max_tiempo):
  if k < len(tiempos):
    sol = 0
    for i in range(0,len(solucion_actual)):
      # Si la máquina anterior no tiene asignado entonces no se asigna
      # trabajo a la actual (sería la misma rama que la anterior).
      if i == 0 or solucion_actual[i-1] > 0:
        solucion_actual[i] += tiempos[k]
        sol = min(sol, algoritmo2(k+1,tiempos, solucion_actual, \
            max(max_tiempo, solucion_actual[i])))
        solucion_actual[i] -= tiempos[k]
    return sol
  else:
    return max_tiempo
```

Un detalle importante es que no queremos que una máquina esté sin realizar trabajo alguno ya que estaríamos perdiendo tiempo de trabajo. Cada máquina debe tener asignado al menos un trabajo. Esto nos da una condición de poda para el algoritmo anterior. Si hay k máquinas libres y solo quedan por asignar k trabajos, un trabajo va a cada máquina libre (recordemos que nos da igual el orden). Podemos aplicarlo obteniendo el Algoritmo 3.

Algoritmo 3. Mejora sobre el Algoritmo 2 que evita soluciones con máquinas libres.

```
# Parametros:
# - k : índice del trabajo a asignar.
# - tiempos : vector con los tiempos de los m trabajos.
# - solucion_actual : vector con el tiempo de trabajo
#   asignado a cada máquina.
# - max_tiempo : Máximo de solucion_actual
# - maquinas_libres : Número de máquinas que no tienen asignado un trabajo.
def algoritmo2(k, tiempos, solucion_actual, max_tiempo, maquinas_libres):
  if k < len(tiempos):
    # Comprobamos que hay más trabajos libres que máquinas libres
    if maquinas_libres < len(tiempos)-k:
      sol = 0
      for i in range(0,len(solucion_actual)):
        if i == 0 or solucion_actual[i-1] > 0:
          solucion_actual[i] += tiempos[k]
          sol = min(sol, algoritmo2(k+1,tiempos, solucion_actual, \
              max(max_tiempo, solucion_actual[i]), maquinas_libres + \
              0 if solucion_actual[i] != 0 else 1))
          solucion_actual[i] -= tiempos[k]
      return sol
    # Si la comprobación devuelve falso a cada máquina se le asigna un trabajo
    else:
      return max(max_tiempo, max(tiempos[k:]))
  else:
    return max_tiempo
```

En este punto cabe preguntarse cuántas soluciones el espacio de soluciones (denotémoslo S) debe recorrer el Algoritmo 3. Nótese que obvia las equivalentes o aquellas que dejan alguna máquina libre. El espacio de soluciones consta de m^n elementos, todos ellos recorridos por el Algoritmo 1. Por tanto, para saber este número en primer lugar tenemos que restarle a m^n las soluciones que dejan alguna máquina libre. Llamamos a_m a este resultado. Podemos abstraer la definición para denotar a_i al número de soluciones con todas las máquinas ocupadas para i máquinas en lugar de m con $i = 1, \ldots, m$.

En esta situación podemos observar que para $i \in \{1, \ldots, m\}$ el número de soluciones de S que deja exactamente i máquinas ocupadas es

$$\binom{m}{i} a_i$$

Por tanto, se tiene que:

$$i^n = \sum_{j=1}^{i} \binom{i}{j} a_j \ \forall i = 1, \ldots, m$$

Esta relación permite calcular a_m de la siguiente forma:

- $a_1 = 1$
- $a_i = i^n - \sum_{j=1}^{i-1} \binom{i}{j} a_j \ \forall i = 1, \ldots, m$

Podemos calcular a_2, después a_3 y así sucesivamente.

Una vez hemos calculado a_m tenemos que tener en cuenta que las soluciones equivalentes (aquellas que son una permutación de la actual) no nos interesan. Dividimos pues a_m entre $m!$ para quedarnos con un único representante de las clases de equivalencia.

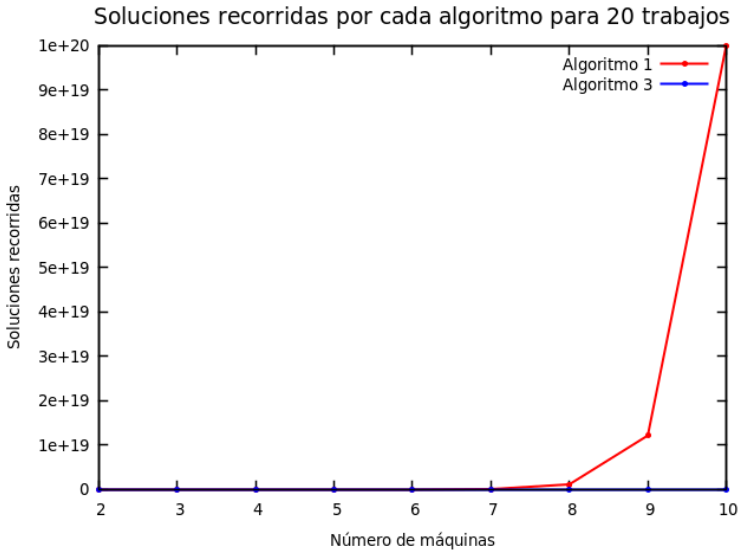

Imagen 1. Número de soluciones recorridas por cada algoritmo.