Algoritmos Voraces

Problema 5 - Problema del electricista

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

19 de mayo de 2015

Introducción

- En esta presentación se proporciona una solución para el ejercicio 5.
- El código, los resultados de las ejecuciones, las gráficas y los pdf asociados se puede encontrar en GitHub.

Explicación del problema

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

Resumen del enunciado

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, $i = 1, \dots, n$. 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.

Explicación del problema

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

Análisis del problema

- Estructura similar al **problema de los electricistas**. Conocido como **scheduling problem**.
- Cambia la función objetivo a optimizar: tiempo medio de espera vs tiempo total de los trabajos.
- Es un problema NP-Completo.

Análisis del problema

- Queremos conseguir la mejor solución exponencial posible.
- Si $n \leq m$, podemos asignar un trabajo a cada máquina. El tiempo es el máximo de los t_i.
- Podemos suponer entonces $n > m$.

Análisis del problema

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

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

Demostración:

- A cada trabajo se asigna una máquina, pudiendo repetirse.
- Para cada trabajo tenemos m posibilidades disponibles. Total: m^n.

Análisis del problema

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

Primer algoritmo ($\theta(m^n)$)

```
def algoritmo1(k,tiempos, solucion_actual, max_tiempo
  if k < len(tiempos):
    sol = Inf
    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])))
      solucion_actual[i] -= tiempos[k]
    return sol
  else:
    return max_tiempo
```

Análisis del problema

Mejorando el primer algoritmo

- No nos importa cuál sea la máquina que realiza un determinado trabajo. Todas son igual de eficientes.
- **Problema**: el algoritmo anterior calcula por separado permutaciones que dan soluciones equivalentes.
- **Solución**: si quedan máquinas sin tareas solo llamamos al algoritmo recursivo para la primera.

Análisis del problema

Segundo algoritmo

```
def algoritmo2(k, tiempos, solucion_actual, max_tiemp
  if k < len(tiempos):
    sol = Inf
    for i in range(0,len(solucion_actual)):
      # Si la máquina anterior no tiene asignado ento
      # trabajo a la actual (sería la misma rama que
      if i == 0 or solucion_actual[i-1] > 0:
        solucion_actual[i] += tiempos[k]
        sol = min(sol, algoritmo2(k+1,tiempos, soluci
            max(max_tiempo, solucion_actual[i])))
        solucion_actual[i] -= tiempos[k]
    return sol
  else:
    return max_tiempo
```

Análisis del problema

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

Mejorando el segundo algoritmo

- No queremos que una máquina esté sin realizar trabajo alguno ya que estaríamos perdiendo tiempo de trabajo.
- **Condición de poda:** si hay k máquinas libres y solo quedan por asignar k trabajos, un trabajo va a cada máquina libre.

Análisis del problema

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

Tercer algoritmo

```
def algoritmo3(k, tiempos, solucion_actual, max_tiemp
  if k < len(tiempos):
    # Comprobamos que hay más trabajos libres que máq
    if maquinas_libres < len(tiempos)-k:
      sol = Inf
      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]
```

Análisis del problema

Tercer algoritmo

```
      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
```

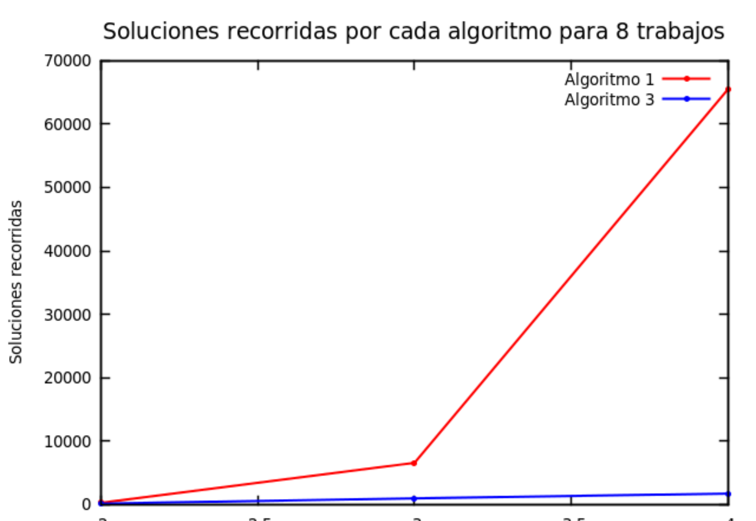
Análisis del problema

Soluciones recorridas por el tercer algoritmo

- $a_i :=$ número de soluciones con todas las máquinas ocupadas para i máquinas con $i = 1, \ldots, m$.
- $a_1 = 1$
- $a_i = i^n - \sum_{j=1}^{i-1} \binom{i}{j} a_j \ \forall i = 1, \ldots, m$

Análisis del problema

Comparación de soluciones recorridas



Análisis del problema

Mejorando el tercer algoritmo

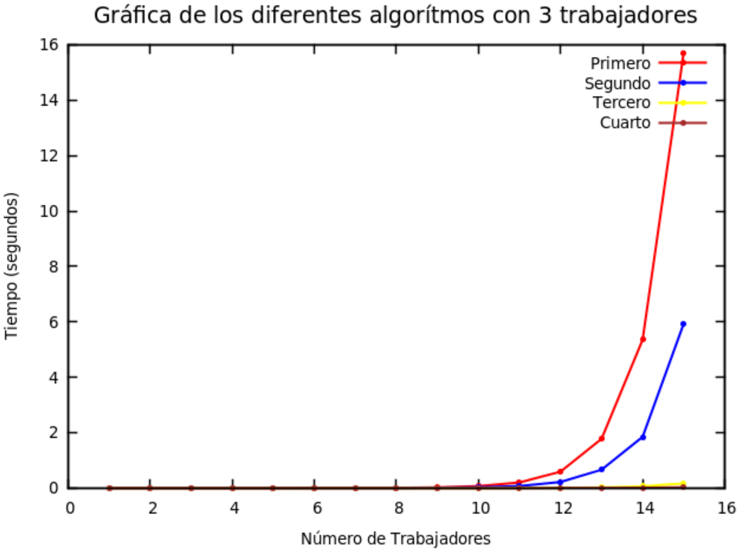
Criterio de poda habitual en backtracking: una vez se ha conseguido una solución se comprueba en cada momento si la rama actual puede conseguir una solución mejor que la mejor obtenida o no. En caso negativo se deja la rama sin terminar de visitar.

Análisis del problema

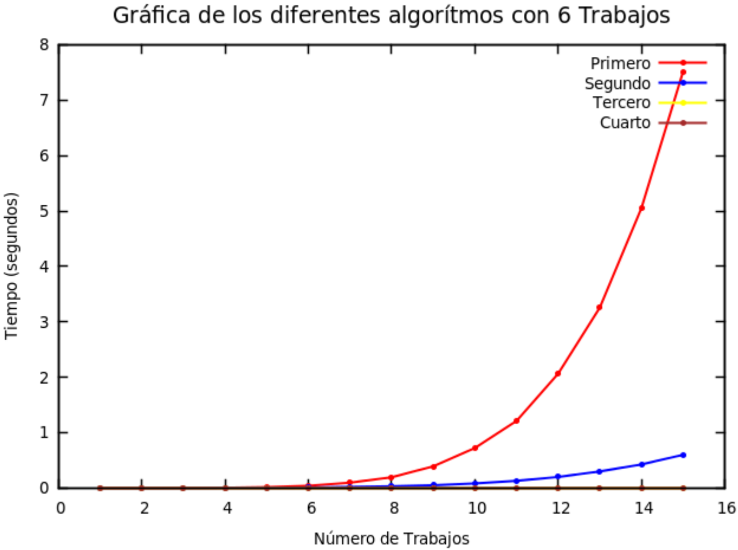
Algoritmo final

```
def algoritmo2(k, tiempos, solucion_actual, max_tiemp
  if k < len(tiempos) and max_tiempo < mejor_solucion
    # Comprobamos que hay más trabajos libres que máq
    if maquinas_libres < len(tiempos)-k:
      for i in range(0,len(solucion_actual)):
        if i == 0 or solucion_actual[i-1] > 0:
          solucion_actual[i] += tiempos[k]
          mejor_solucion = min(mejor_solucion, algori
              max(max_tiempo, solucion_actual[i]), ma
              0 if solucion_actual[i] != 0 else 1, me
          solucion_actual[i] -= tiempos[k]
      return mejor_solucion
    # Si la comprobación devuelve falso a cada máquin
    else:
```

Análisis empírico

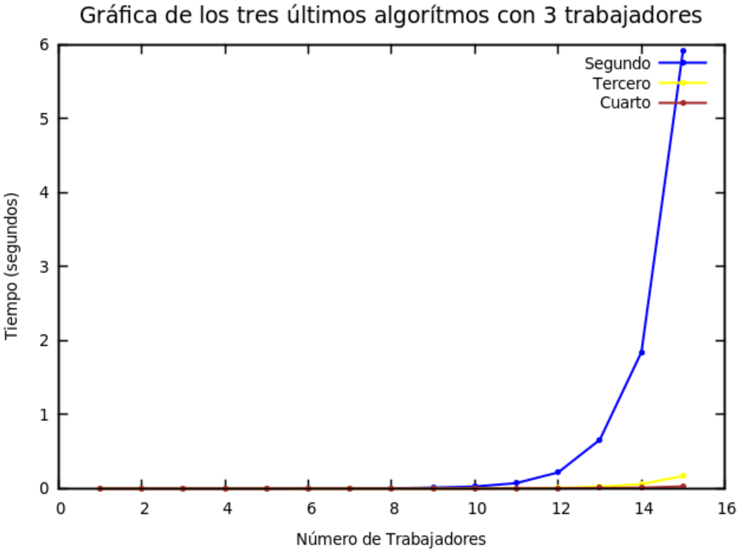


Análisis empírico

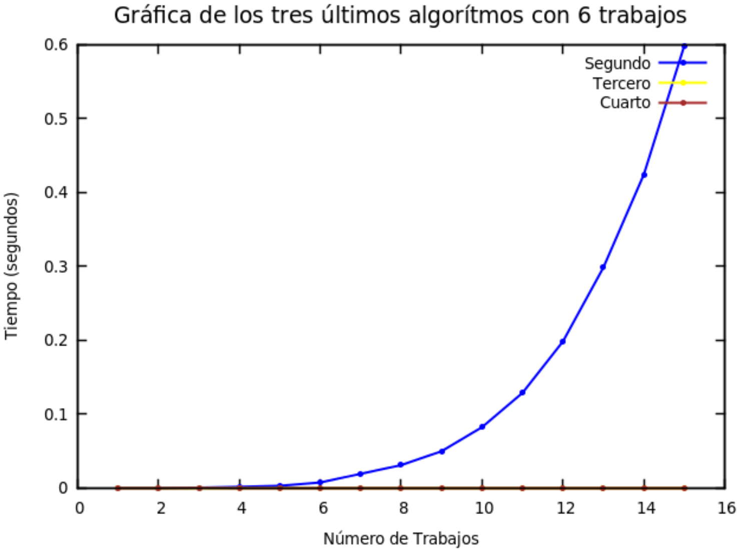


Análisis empírico

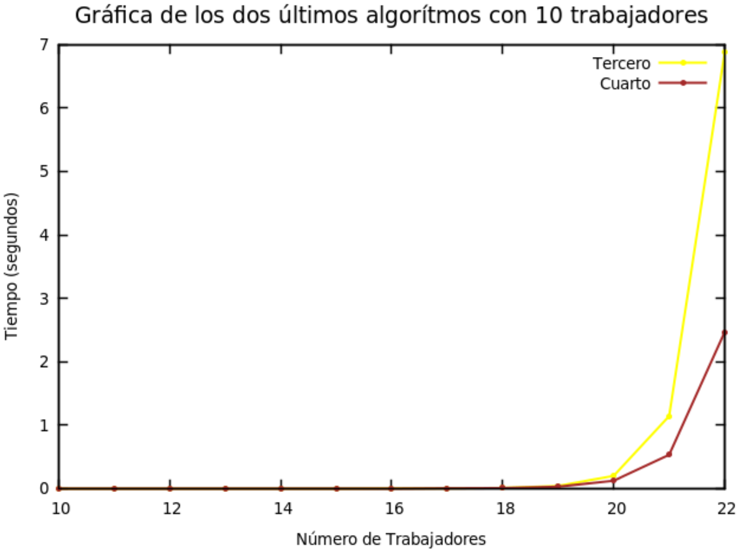
A. Herrera, A. Moya, I. Sevillano, J.L. Suarez



Análisis empírico



Análisis empírico



Análisis empírico