Algoritmos para el problema de la mochila binaria (0/1 knapsack problem)

Asignatura: Algoritmos

Docentes:

Pablo Factorovich y Emanuel Delgadillo

Federico Aloi

Julio 2016

Objetivo

El objetivo de este trabajo es desarrollar sencillos algoritmos Branch and Bound y metaheurística GRASP para el problema de la mochila binaria, eligiendo de manera experimental los criterios de corte y los distintos parámetros.

Dicha experimentación estará basada en un conjunto de 12 instancias de prueba con distinta cantidad de elementos y capacidad de la mochila, que fue provista por los docentes.

Descripción de los algoritmos

GRASP

Para la heurística Greedy Random decidí ordenar por mayor tasa y luego elegir un elemento al azar entre el primero y el n-ésimo, siendo *n = cantidad_de_elementos * 10%*.

Una vez obtenida esa solución, se la intenta mejorar con la heurística de búsqueda local elegida y se retorna el resultado cuando se cumple alguna de las condiciones de corte: tiempo máximo de ejecución, cantidad máxima de iteraciones totales o cantidad de iteraciones sin mejorar.

El código más relevante puede verse en el siguiente fragmento:

```
def compute_solution_for(instance)
  iterations = 1
  without_improving = 0
  best_solution = Knapsack::Solution.empty self, instance

  Timeout::timeout(@max_minutes_to_run * 60) do
    begin
      while iterations < @max_iterations && without_improving < @max_without_improving
        iterations += 1
        new_solution = solve_iteration_for instance

        if new_solution > best_solution
          best_solution = new_solution
        else
          without_improving += 1
        end
      end
    rescue Timeout::Error
      # ignored
```

```
    end

    best_solution.items
  end
end

def solve_iteration_for(instance)
  greedy_solution = Knapsack::Solvers::GreedyRandom.new.solve_for instance
  best_combination = @local_search_heuristic.try_to_improve self, instance,
greedy_solution.items, greedy_solution.remaining_capacity

  Knapsack::Solution.new self, instance, best_combination
end
```

Probé esto con dos heurísticas de búsqueda local diferentes, a saber:

Reemplazar un elemento de la solución inicial por entre 0 y 2 de los que habían quedado afuera, que mejoren la ganancia

Se ve que quitar un elemento sin agregar nada no mejora la solución, y también asumí que reemplazar uno por otro tampoco tenía sentido, ya que la solución inicial se parecía mucho a la Greedy y no iba a ser posible mejorarla de esa forma. Por lo tanto en la implementación siempre se reemplaza uno por dos.

```
def try_to_improve(grasp, instance, initial_items, remaining_capacity)
  remaining_items = instance.items - initial_items
  remaining_items.combination(2) do |pair|
    index_to_remove = 0

    while index_to_remove < initial_items.length
      item_to_remove = initial_items[index_to_remove]

      if can_be_replaced(item_to_remove, pair, remaining_capacity) &&
should_be_replaced(item_to_remove, pair)
        new_items, capacity_delta = replace_items_in_solution initial_items,
index_to_remove, pair

        return try_to_improve grasp, instance, new_items, remaining_capacity -
capacity_delta
      else
        index_to_remove += 1
      end
    end
  end

  fill_remaining_capacity(initial_items, remaining_capacity, remaining_items)
```

```
  initial_items
end
```

Reemplazar dos elementos de la solución inicial por cualquier cantidad de los que habían quedado afuera, utilizando un algoritmo Greedy para elegirlos

En esta opción sí puede darse el caso de reemplazar uno por otro y aún así mejorar, ya que las soluciones rápidamente se alejan de la Greedy Random inicial.

```
def try_to_improve(grasp, instance, initial_items, remaining_capacity)
  remaining_items = (instance.items - initial_items).sort_by(&:rate).reverse

  initial_items.combination(2) do |items_to_remove|
    capacity_to_fill = remaining_capacity + items_to_remove.sum(&:weight)
    next_index = 0
    items_to_add = []

    while capacity_to_fill > 0 && next_index < remaining_items.size
      next_item = remaining_items[next_index]

      if next_item.weight <= capacity_to_fill
        items_to_add << next_item
        capacity_to_fill -= next_item.weight
      end

      next_index += 1
    end

    if items_to_add.sum(&:gain) > items_to_remove.sum(&:gain)
      new_items = replace_items_in_solution initial_items, items_to_remove,
items_to_add

      return try_to_improve grasp, instance, new_items, capacity_to_fill
    end
  end

  fill_remaining_capacity(initial_items, remaining_capacity, remaining_items)

  initial_items
end
```

Branch and bound

Como criterio de subdivisión de problemas, decidí seguir la estrategia de incluir o no incluir un determinado ítem en la solución. La relajación elegida fue la mochila fraccionaria, y la heurística primal el algoritmo Greedy.

Para decidir el orden de los nodos, probé con dos criterios: elegir el de peor tasa - suponiendo que esto haría que se poden rápidamente las ramas que lo incluyan en su solución - o elegir el de mejor tasa - suponiendo que esto haría que se poden rápidamente las ramas que no lo incluyan en su solución.

El código para ambas soluciones es similar (sólo cambia el orden inicial de los elementos), por lo cual sólo incluiré el de una de ellas. Tuve que agregar también la posibilidad de terminar la ejecución por timeout, ya que el algoritmo no terminaba en tiempos razonables para las instancias más grande.

```
def compute_solution_for(instance)
  to_visit = [TreeNode.new(sort_items_by_best_rate(instance), instance.capacity, 0)]
  lower_bound = 0

  begin
    Timeout::timeout(@max_minutes_to_run * 60) do
      while to_visit.any?
        current_node = to_visit.shift

        next unless current_node.has_many_items?
        next if current_node.capacity < 0
        next if current_node.upper < lower_bound

        if current_node.lower > lower_bound
          lower_bound = current_node.lower
        end

        if current_node.lower != current_node.upper
          to_visit += current_node.next_level
        end
      end
    end
  rescue Timeout::Error
    # ignored
  end

  lower_bound
end
```

Mediciones

Todas las mediciones fueron realizadas sobre un equipo con las siguientes características:

CPU - fragmento de la salida de lshw
product: Intel(R) Core(TM) i7-3517U CPU @ 1.90GHz
physical id: 4
size: 1900MHz
capacity: 1900MHz
width: 64 bits

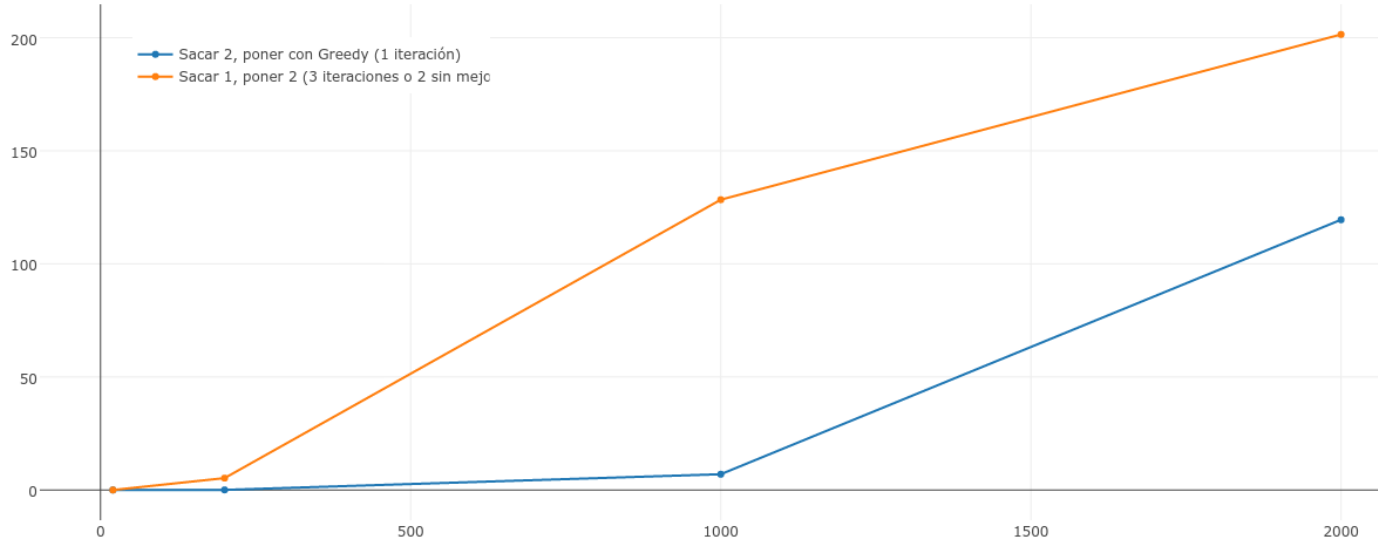
Memoria (x2, total 8GiB) - fragmento de la salida de lshw
description: SODIMM DDR3 Synchronous 1600 MHz (0.6 ns)
product: AM1U16BC4P2-B19C
vendor: Fujitsu
size: 4GiB
width: 64 bits
clock: 1600MHz (0.6ns)

Sistema operativo - extraido de /proc/version
Linux version 3.13.0-68-generic (buildd@lgw01-46) (gcc version 4.8.2 (Ubuntu 4.8.2-19ubuntu1)) #111-Ubuntu SMP Fri Nov 6 18:17:06 UTC 2015

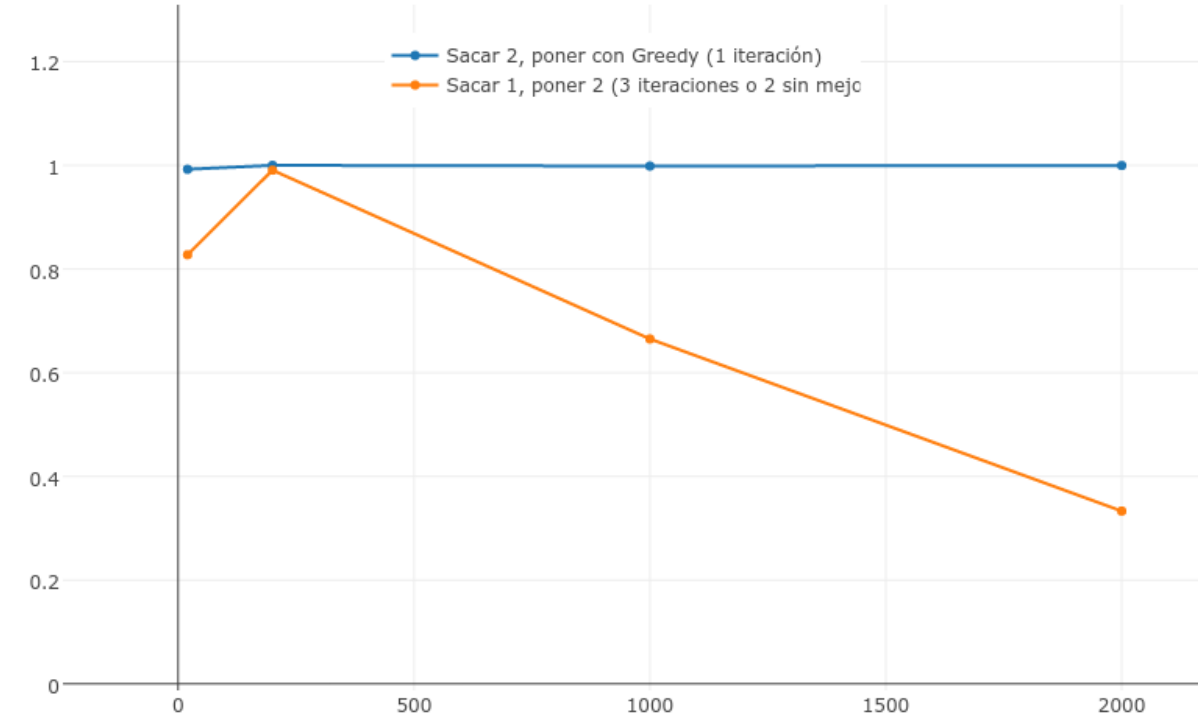
Para que todos las ejecuciones terminen se estableció un timeout de 5 minutos por ejecución de instancia (excepto en el Backtracking), provocando esto que la salida de algunas corridas produjera 0 como resultado.

GRASP

Cantidad de elementos X Tiempo de ejecución (segundos)



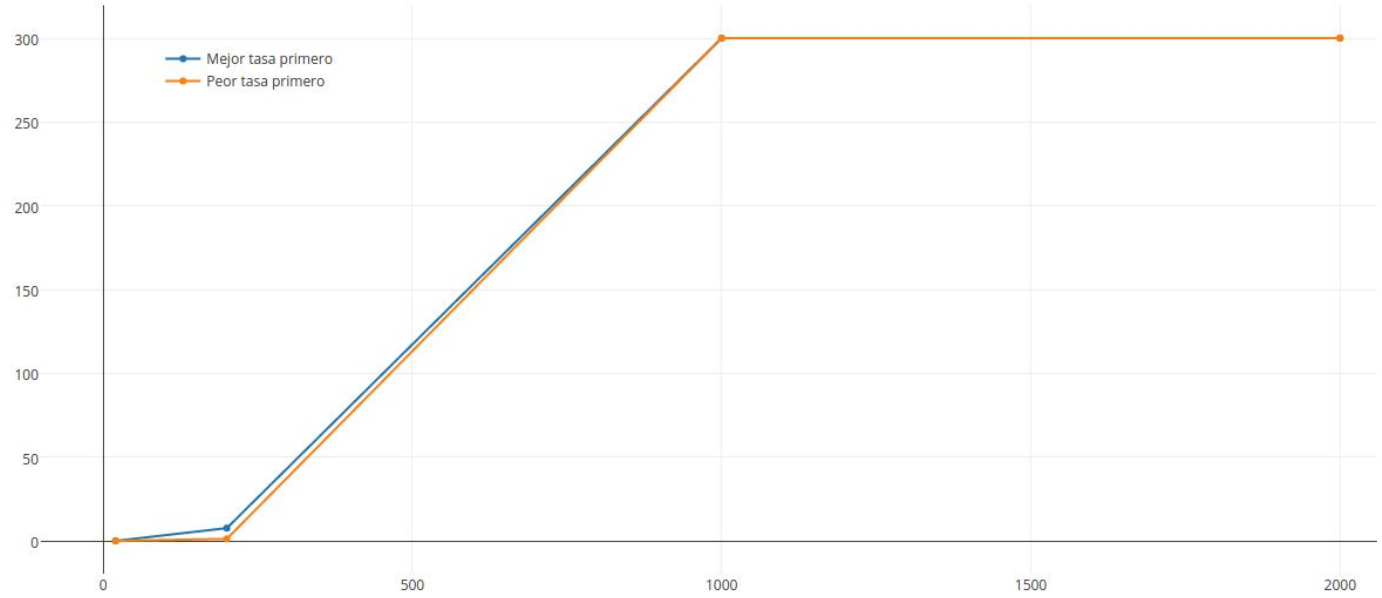
Cantidad de elementos X Calidad de la solución



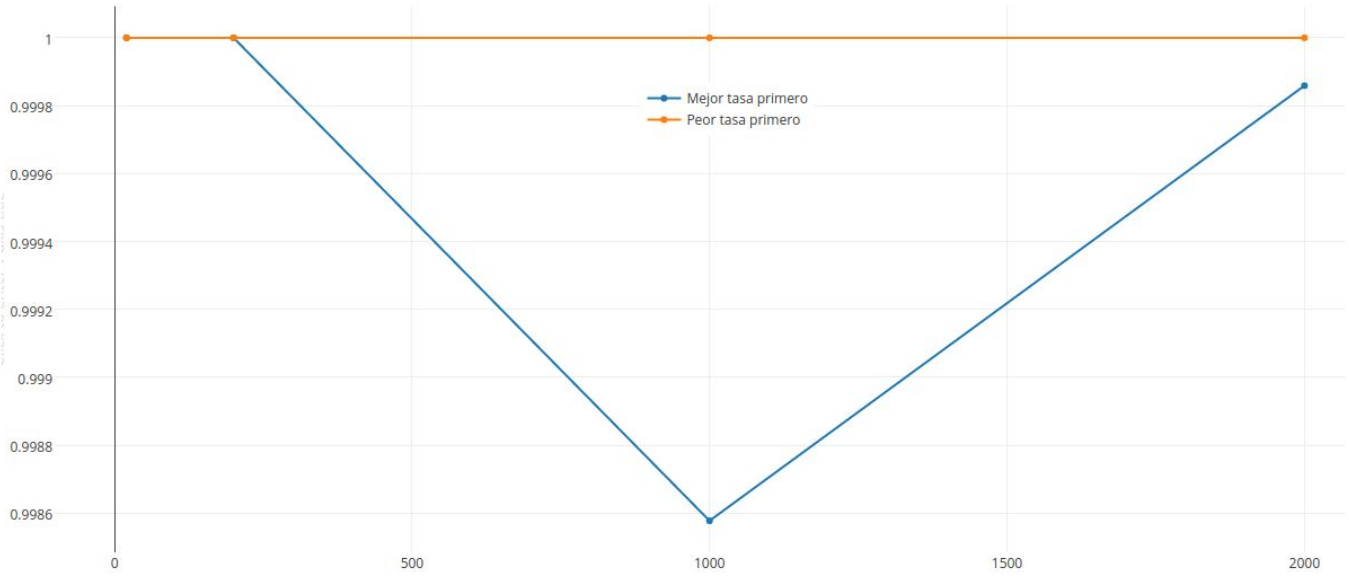
Del análisis de los gráficos surge que claramente la búsqueda local que saca dos elementos y rellena con Greedy produce soluciones en menor tiempo y con mejor calidad (prácticamente en todas las instancias encontró el óptimo).

Branch and bound

Cantidad de elementos X Tiempo de ejecución (segundos)



Cantidad de elementos X Calidad de la solución

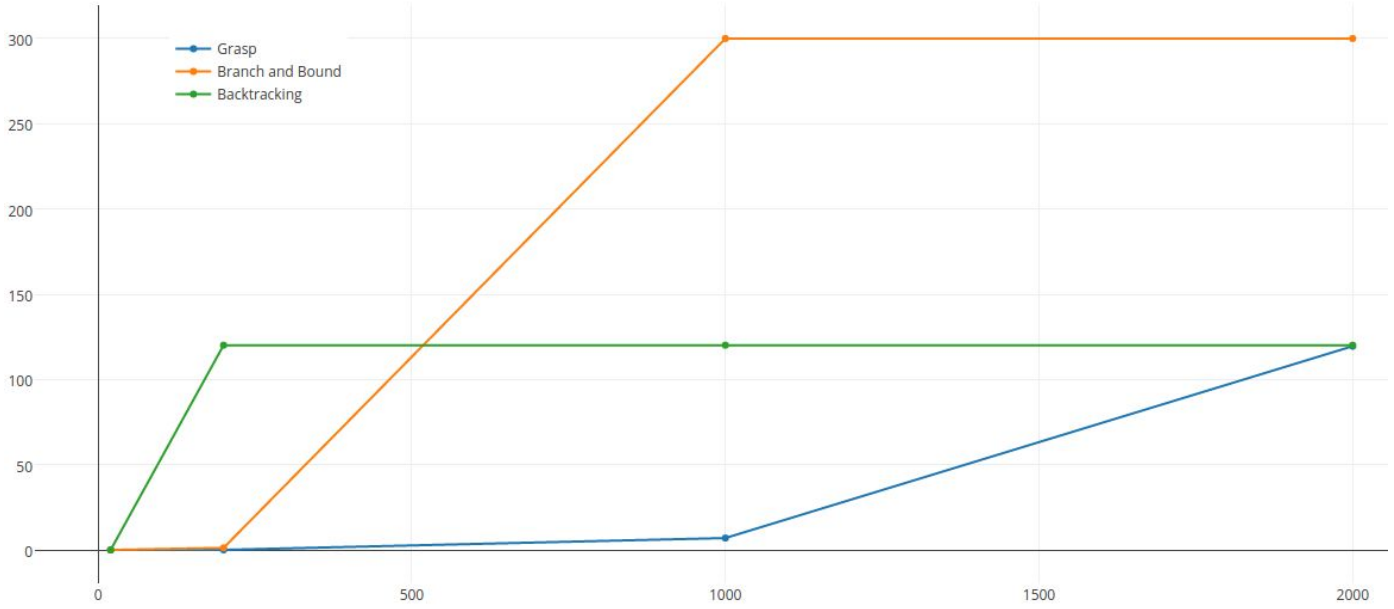


Del análisis de los gráficos surge que el criterio que elige los elementos con peor tasa primero es levemente más rápido que el otro, y que en todos los casos llega al óptimo - aún en aquellas instancias donde su ejecución fue interrumpida por timeout.

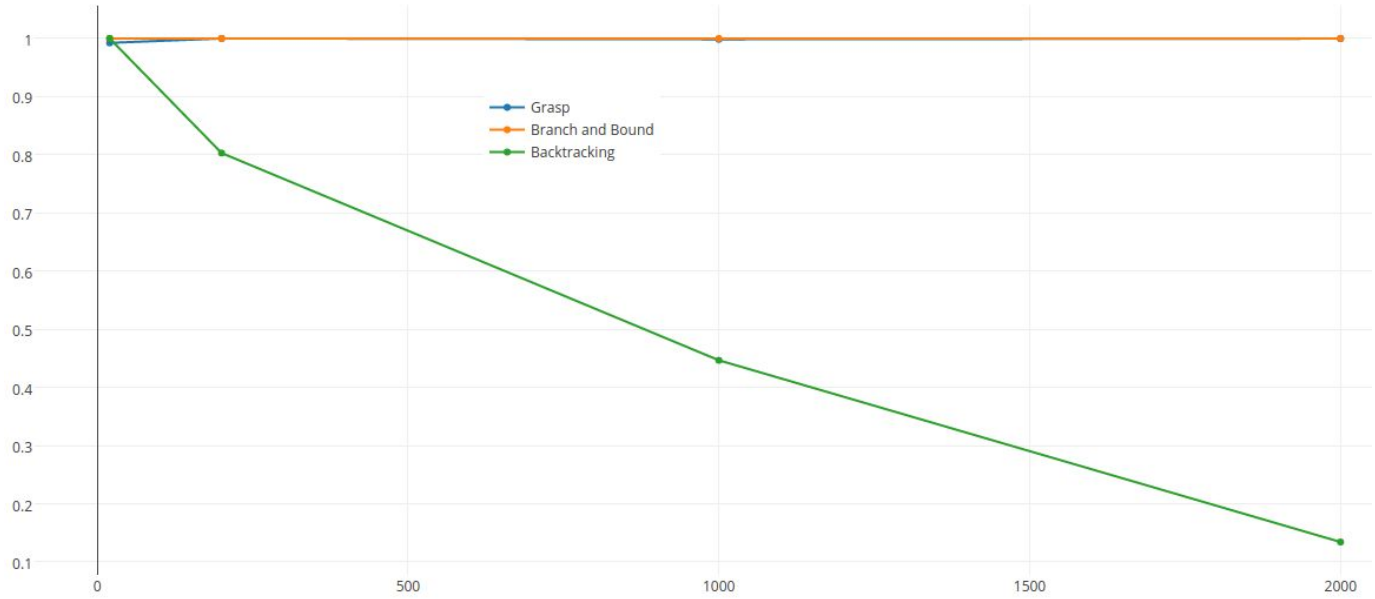
Comparación GRASP, Branch and Bound, Backtracking

Nota: para las instancias de Backtracking el timeout fue reducido a 2 minutos en vez de 5, ya que de otra forma la medición hubiese demorado casi una hora en completarse.

Cantidad de elementos X Tiempo de ejecución (segundos)



Cantidad de elementos X Calidad de la solución



Conclusiones

Se pudo comprobar de manera empírica que la metaheurística GRASP produce soluciones en tiempos muchísimo más razonables que los que emplean los algoritmos Branch and Bound y Backtracking, existiendo incluso casos en que estos últimos dos no terminan antes de los 5 minutos y deben ser abortados.

En cuanto a la calidad, se logró con GRASP llegar al óptimo en 9 de las 12 instancias, quedando las restantes ubicadas entre el 97% y el 99% de calidad con respecto al óptimo.