Combinatorial Assignment 2

Henrique Lopes s2655349
Werner Lootsma s1914227

December 19, 2014

1 Assignment Problem

1.1 Problem description

The problem can be seen as the problem of finding a minimal perfect matching in a bipartite graph. If a bipartite graph formed by two vertex sets A and B is built being A the set of candidates for the committees and B the committees, edges can be added between candidates and committees follows:

- A candidate's first preference has weight 0;
- A candidate's second preference has weight 1;
- A candidate's third preference has weight 3;
- The weight of assigning a candidate to a committee he has not listed is infinite (in this case, the number 99 was used in the algorithm).

After that, there is a complete graph that connects the group of persons to the group of desired committee and an algorithm to find a perfect matching with the minimal total edge cost can be applied over the graph. In this case, the hungarian algorithm was chosen. The algorithm was built according to this description given in the lecture slides. The code to find a matching over the graph (using the max-flow algorithm), the code to adjust the weights and the code to find the minimal vertex cover can be found in Appendix A.

1.2 Results

With the code provided in Appendix A, the following results were found:

```
Person  1 is assigned to the committee  3. His preferences were, in order, [8, 3, 12]
Person 16 is assigned to the committee  2. His preferences were, in order, [2, 15, 6]
Person  3 is assigned to the committee 19. His preferences were, in order, [19, 5, 6]
Person 11 is assigned to the committee 11. His preferences were, in order, [11, 19, 9]
Person  5 is assigned to the committee  4. His preferences were, in order, [18, 4, 10]
Person  6 is assigned to the committee  7. His preferences were, in order, [7, 17, 2]
Person  7 is assigned to the committee 13. His preferences were, in order, [13, 20, 1]
Person  8 is assigned to the committee 10. His preferences were, in order, [10, 14, 7]
```

```
Person 13 is assigned to the committee 14. His preferences were, in order, [6, 9, 14]
Person 10 is assigned to the committee  1. His preferences were, in order, [1, 16, 14]
Person 12 is assigned to the committee 16. His preferences were, in order, [3, 16, 18]
Person 15 is assigned to the committee  6. His preferences were, in order, [6, 13, 12]
Person 14 is assigned to the committee 12. His preferences were, in order, [12, 7, 4]
Person 17 is assigned to the committee  9. His preferences were, in order, [9, 17, 16]
Person 18 is assigned to the committee  5. His preferences were, in order, [5, 1, 12]
Person 19 is assigned to the committee 20. His preferences were, in order, [10, 20, 9]
Person  4 is assigned to the committee 17. His preferences were, in order, [11, 17, 20]
Person  2 is assigned to the committee  8. His preferences were, in order, [8, 2, 5]
Person 20 is assigned to the committee 18. His preferences were, in order, [18, 17, 4]
Person  9 is assigned to the committee 15. His preferences were, in order, [15, 14, 10]
```

2 Sahni

The Sahni heuristic with $k = 3$ has a worst case performance ratio of $\frac{k}{k+1} = \frac{3}{4}$. Pseudocode of this algorithm is as follows (copied from the lecture notes), where GREEDY is Greedy Heuristic as discussed in class:

SAHNI(3)
1. **begin**
2. sort E according to non-increasing order of p/w ratios;
3. set S $\leftarrow \emptyset$, $\Pi \leftarrow 0$;
4. for (each M $\subset$ E) with$|M| \leq 3$ and $\sum_{e_j \in M} w_k \leq C$) **begin**
5. set $T^* \leftarrow$ GREEDY$(E \setminus M, C - \sum_{e_j \in M} w_k)$;
6. set T $\leftarrow M \cup T^*$;
7. if $(\sum_{e_j \in M} p_k > \Pi)$
8. set $S \leftarrow T$, $\Pi \leftarrow \sum_{e_j \in M} p_k$;
9. **end;**
10. **end.**

This heuristic has, as stated before, a worst case performance ratio of $\frac{k}{k+1}$.

Proof[1]: Let Y be the set of items inserted into the knapsack in the optimal solution. If $|Y| \leq k$, then **SAHNI(k)** gives the optimal solution, since all combinations of $|Y|$ are tried. Hence, assume $|Y| > k$. Let $\hat{M}$ be the set of the first k items of highest profit in Y, and denote the remaining items of Y with $j_1, ..., j_r$, assuming $\frac{p_{j_i}}{w_{j_i}} \geq \frac{p_{j_{i+1}}}{w_{j_{i+1}}} (i = 1, ..., r-1)$. Hence, if z is the optimal solution value we have

$$p_{j_i} \leq \frac{z}{k+1} \qquad \text{for } i = 1, ..., r \tag{1}$$

Consider now the iteration of SAHNI(k) in which $M = \hat{M}$, and let i_m be the first item of $\{j_1, ...j_r\}$ not inserted into the knapsack by SAHNI(k). If no such

[1]Silvano Matello and Paolo Toth, *Knapsack Problems. Algorithms and Computer Implementations*, England 1990, p.51-52

item exists then the heuristic solution is optimal. Otherwise we can write z as

$$z = \sum_{i \in \hat{M}} p_i + \sum_{i=1}^{m-1} p_{j_i} + \sum_{i=m}^{r} p_{j_i}, \tag{2}$$

while for the heuristic solution value returned by SAHNI(k) we have

$$z^H \geq \sum_{i \in \hat{M}} p_i + \sum_{i=1}^{m-1} p_{j_i} + \sum_{i \in Q} p_i, \tag{3}$$

where Q denotes the set of those items of $E \setminus \hat{M}$ which are in the heuristic solution but not in $\{j_1, ..., j_r\}$ and whose index is less than j_m. Let $c^* = c - \sum_{i \in \hat{M}} w_i - \sum_{i=1}^{m-1} w_{j_i}$ and $\bar{c} = c^* - \sum_{i \in Q} w_i$ be the residual capacities available, respectively, in the optimal and the heuristic solution for the items of $E \setminus \hat{M}$ following j_{m-1}. Hence, from (2),

$$z \leq \sum_{i \in \hat{M}} p_i + \sum_{i=1}^{m-1} p_{j_i} + c^* \frac{p_{j_m}}{w_{j_m}}, \tag{4}$$

by definition of m we have $\bar{c} < w_{j_m}$ and $\frac{p_i}{w_i} \geq \frac{p_{j_m}}{w_{j_m}}$ for $i \in Q$, so

$$z < \sum_{i \in \hat{M}} p_i + \sum_{i=1}^{m-1} p_{j_i} + p_{j_m} + \sum_{i \in Q} p_i. \tag{5}$$

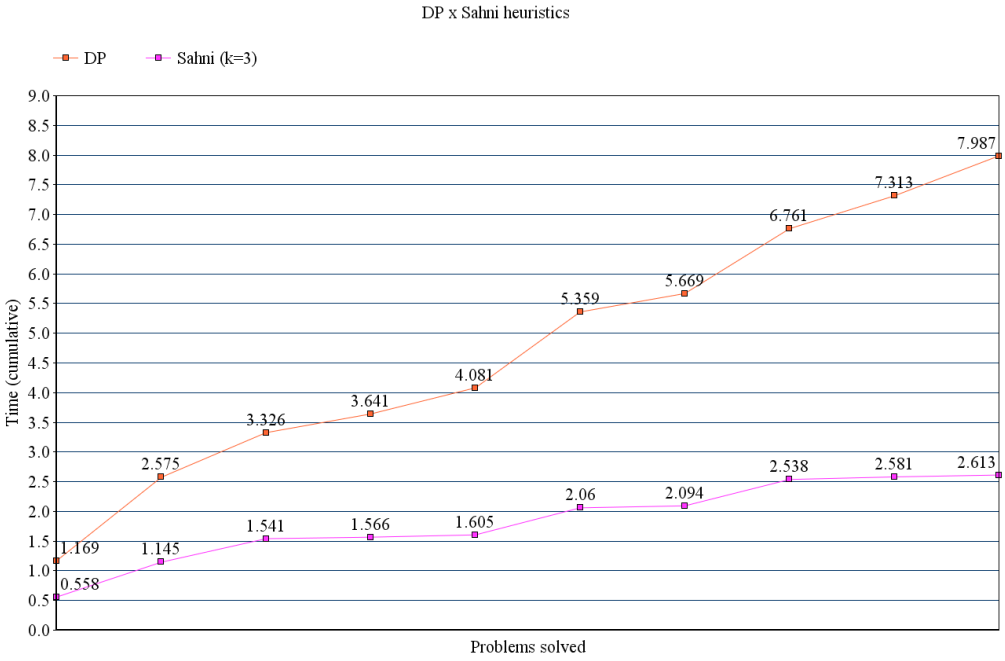
Hence, from (3), $z < z^H + p_{j_m}$ and then using (1) results in

$$\frac{z^H}{z} > \frac{k}{k+1}. \tag{6}$$

And since we use SAHNI(3) we have a worst preformance ratio of 0.75.

2.1 Coding the heuristics

Both heuristics were built using the Ruby programming language and their code can be found in Appendix B. The comparison between both heuristics was made by measuring the time to compute each problem (generated randomly using as a seed the student number "2655349") separately and then drawing a line using these measurements in a cumulative way. The resultant graph can be found below:



In the graph above, the red slope refers to the measurements of the optimal heuristic, which, due to the large amount of memory used to store the dynamic programming matrix, takes more time than the Sahni heuristic. Although the Sahni heuristic have to deal with combinations of a set, for $k = 3$ this computation still does not require large amounts of time, which is not the case when k is increased to 4.

2.2 Results

Since the results are listed in a large file, they were given apart from this report. They can be found in the file *results.txt*.

3 Appendix 1 - code for the matching problem

3.1 Matching problem: main.rb

```
require './matching/maxmatch.rb'
require './matching/minimalcover.rb'

$PROBLEM_SIZE = 0

#uses the class Graphmatch to find a minimum vertex cover for the 0-weighted graph
#takes as input the 0-weighted graph and its minimal matching
def find_minimum_vertex_cover matching, edges
  minimal_cover = Graphmatch::MinimalVertexCover.new matching, edges
```

```
  return minimal_cover.minimal_vertex_cover
end

#filters the cost matrix and return only the 0-weighted edges
def find_zero_weighted_edges(cost_matrix)
  edges = Hash.new
  (0..$PROBLEM_SIZE-1).each do |person|
    (0..$PROBLEM_SIZE-1).each do |committee|
      edges[person] = Hash.new if edges[person].nil?
      if cost_matrix[person][committee] == 0
        edges[person]["committee#{committee}".to_sym] = 0
      end
    end
  end
  return edges
end

#tests a matching to see if it is a perfect one
def perfect_matching? matching
  return matching.size == $PROBLEM_SIZE
end

def hungarian_algorithm adjacency_matrix
  edges = find_zero_weighted_edges(adjacency_matrix)
  left  = Array(0..$PROBLEM_SIZE-1)
  right = Array(0..$PROBLEM_SIZE-1)

  matching = Graphmatch.match(left, right
  .collect{|c| c = "committee#{c}".to_sym}, Marshal.load(Marshal.dump(edges)))


  #repeats the iteration until a perfect matching is found
  until perfect_matching? matching

    vertex_cover = find_minimum_vertex_cover matching, edges

    #finds the minimum value in the edges with weight > 0
    min = 99
    (0..$PROBLEM_SIZE - 1).each do |i|
      (0..$PROBLEM_SIZE - 1).each do |j|
        if(adjacency_matrix[i][j] < min \
           && vertex_cover.index(i).nil? \
           && vertex_cover.index(j+20).nil?)
          min = adjacency_matrix[i][j]
        end
      end
    end

    #applies the corrections in the cost matrix
    (0..$PROBLEM_SIZE - 1).each do |i|
```

```
      (0..$PROBLEM_SIZE - 1).each do |j|
        if(vertex_cover.index(i).nil? && vertex_cover.index(j+20).nil?)
          adjacency_matrix[i][j] -= min
        elsif(!vertex_cover.index(i).nil? && !vertex_cover.index(j+20).nil?)
          adjacency_matrix[i][j] += min
        end
      end
    end

    #filter the corrected cost matrix
    edges = find_zero_weighted_edges(adjacency_matrix)
    left  = Array(0..$PROBLEM_SIZE-1)
    right = Array(0..$PROBLEM_SIZE-1)
    matching = Graphmatch.match(left, right
    .collect{|c| c = "committee#{c}".to_sym}, Marshal.load(Marshal.dump(edges)))

  end
  return matching
end

#
#
# MAIN PROGRAM
#
#

#First, an array has to be created to store the preferences of every person
preferences = Array.new

#opens the input file in reading mode
File.open('chair.txt', 'r') do |file|

  #extracts the raw text and splits it by line,
  #then splits every line by the space character
  preferences = file.read.split("\n").collect{|line| line = line.split(' ')}

  #removes every first character from the arrays (so "C12" becomes "12" f. ex.)
  #it will help turning them into numbers later
  preferences = preferences.collect{|line| line = line
  .collect{|committee| committee = committee[1..-1]}}

  #since the array index can be easily used instead of the chairman number,
  #the latter becomes useless, so it can just be dropped
  preferences.collect!{|person| person.drop(1)}

  #finally, convert everything into numbers
  preferences = preferences.collect{|line| line = line
  .collect{|committee| committee = committee.to_i}}

  #the final result is an array of length 20 consisting of small arrays
```

```
  #of length 3 containing the ordered preferences of every person
end

$PROBLEM_SIZE = preferences.length

#creates a weight matrix where every element can be 0, 1, 2 or 99
cost_matrix = Array.new(20)
cost_matrix.collect!{|line| line = Array.new(20).collect{|c| c = 99}}

#brings the values from the preferences matrix to the cost matrix
(0..preferences.length-1).each do |person|
  cost_matrix[person][preferences[person][0]-1] = 0
  cost_matrix[person][preferences[person][1]-1] = 1
  cost_matrix[person][preferences[person][2]-1] = 2
end


#row minima is already subtracted, so let's subtract the col minima
(0..19).each do |i|
  min = 99
  (0..19).each do |j|
    min = cost_matrix[j][i] if(cost_matrix[j][i] < min)
  end
  (0..19).each do |j|
    cost_matrix[j][i] = cost_matrix[j][i] - min
  end
end

result = hungarian_algorithm(cost_matrix)
File.open('result', 'w') do |file|
  result.each_pair do |k, v|
    file.printf "Person %2d is assigned to the committee %2d. ",
    k + 1, v.to_s[9..-1].to_i + 1
    file.puts "His preferences were, in order, #{preferences[k].inspect}"
  end
end
```

3.2 maxmatch.rb

```
class Graphmatch
#returns a hash where the keys are the persons and the values are the committees
  def self.match(left_vertices, right_vertices, edges, search = :shortest_path)
    vertices = left_vertices + right_vertices + [:sink, :source]

    edges[:sink] = {}
    edges[:source] = {}

    left_vertices.each { |lv| edges[:source][lv] = 0 }
    right_vertices.each { |rv| edges[rv] = { :sink => 0 } }
```

```
    graph = { vertices: vertices, edges: edges }

    matching = Maxflow.best_matching! graph, search = search
  end
end

require './matching/maxflow.rb'
```

3.3 maxflow.rb

```
class Graphmatch::Maxflow
  #returns a hash with keys being the persons and values being the committees
  def self.best_matching!(graph, search, source = :source, sink = :sink)
    loop do
      path = augmenting_path graph, search
      break unless path
      augment_flow_graph! graph, path
    end
    matching_in graph, source, sink
  end

  # Finds an augmenting path in a flow graph.
  #
  # Returns the path from source to sink, as an array of edge arrays, for
  # example [[:source, 'a'], ['a', 'c'], ['c', :sink]]
  #
  # Search parameter will switch it from a max-flow to min-cost max-flow search
  def self.augmenting_path(graph, search = :shortest_path,
    source = :source, sink = :sink)
    if search == :shortest_path
      parents = Graphmatch::BFS.search graph, source, sink
    elsif search == :min_cost
      distance, parents = Graphmatch::BellmanFord.search graph, source
    end

    return nil unless parents[sink]

    # Reconstruct the path.
    path = []
    current_vertex = sink
    until current_vertex == source
      path << [parents[current_vertex], current_vertex]
      current_vertex = parents[current_vertex]

      if path.length > parents.length
        raise "Cannot terminate. Use integral edge weights."
      end
    end
    path.reverse!
  end
```

```
  # Augments a flow graph along a path.
  def self.augment_flow_graph!(graph, path)
    # Turn normal edges into residual edges and viceversa.
    edges = graph[:edges]
    path.each do |u, v|
      edges[v] ||= {}
      edges[v][u] = -edges[u][v]
      edges[u].delete v
    end
  end

  # The matching currently found in a matching graph.
  # @return [Hash] assignment hash of left_vertices => right_vertices
  def self.matching_in(graph, source = :source, sink = :sink)
    Hash[*((graph[:edges][sink] || {}).keys.map { |matched_vertex|
      [graph[:edges][matched_vertex].keys.first, matched_vertex]
    }.flatten)]
  end
end

require './matching/bellman-ford.rb'
require './matching/bfs.rb'
```

3.4 Bellman-ford.rb

```
class Graphmatch::BellmanFord
  # Implementation of the Bellman-Ford algorithm
  #
  # Finds the min-cost between the starting vertex and all other vertices.
  # returns distance, parent [Hash, Hash] distance from :source,
  #parents of each vertex
  def self.search(graph, source = :source)
    distance = {}
    parent = {}

    graph[:vertices].each do |v|
      distance[v] = (v == source) ? 0 : Float::INFINITY
      parent[v] = nil
    end

    # Bellman-Ford edge relaxation
    graph[:vertices].each do |vertex|
      graph[:edges].map do |u, neighbors|
        graph[:edges][u].map do |v, w|

          if distance[u] + w < distance[v]
            distance[v] = distance[u] + w
            parent[v] = u
          end
```

```
        end
      end
    end

    # Run once more to check for negative-weight cycles
    graph[:edges].map do |u, neighbors|
      graph[:edges][u].map do |v, w|
        if distance[u] + w < distance[v]
          raise "Graph contains negative-weight cycle"
        end
      end
    end

    return distance, parent
  end
end
```

3.5 Bfs.rb

```
class Graphmatch::BFS
  # Implementation of Breadth-First Search
  def self.search(graph, source = :source, sink = :sink)
    parents = { source => true }
    queue = [source]

    until queue.empty?
      current_vertex = queue.shift
      break if current_vertex == sink
      (graph[:edges][current_vertex] || {}).each do |new_vertex, edge|
        next if parents[new_vertex]
        parents[new_vertex] = current_vertex
        queue << new_vertex
      end
    end

    #return
    parents
  end
end
```

3.6 Minimal-vertex-cover.rb

```
class Graphmatch::MinimalVertexCover


  def initialize(maximal_matching, zero_weighted_graph)

    #transforms the hash into an array
    #every person has a number from 0 to 19
    #every committee has a number from 20 to 39
```

```
#the array is a list of adjacencies with 'true'
#for the edges that belong to the minimal matching
#and false to those that do not
edges = Array.new
(0..39).each do |i|
  edges[i] = Array.new
end

maximal_matching.each_pair do |key,value|
  edges[key] << [value.to_s[9..-1].to_i + 20, true]
  edges[value.to_s[9..-1].to_i + 20] << [key, true]
end

zero_weighted_graph.each_pair do |key, value|
  value.each_pair do |k, v|
    if(edges[key].index([k.to_s[9..-1].to_i + 20, true]).nil?)
      edges[key] << [k.to_s[9..-1].to_i + 20, false]
      edges[k.to_s[9..-1].to_i + 20] << [key, false]
    end
  end
end

#looks for uncovered (unmatched) vertices
matched = Array.new(40, false)
(0..39).each do |i|
  if(edges[i] != [])
    edges[i].each do |edge|
      if(edge[1] == true)
        matched[i] = true
      end
    end
  end
end

#edges_to_be_analyzed = uncovered vertices
edges_to_be_analyzed = Array.new
(0..39).each do |i|
  if matched[i] == false
    edges_to_be_analyzed << [i,true]
  end
end

#keeps track of the analysed vertices
visited = Array.new(40, false)

mincover = Array.new
delete = Array.new

#vertices reached by no edges are automatically
#added to the minimal cover
```

```
    edges_to_be_analyzed.each do |edge|
      if edges[edge[0]] == []
        mincover << edge[0]
        visited[edge[0]] = true
        delete << edge
      end
    end
    delete.each do |e|
      edges_to_be_analyzed.delete(e)
    end


  #walks through alternating paths adding
  #nodes to the minimal cover
   while edges_to_be_analyzed.size > 0
      edge = edges_to_be_analyzed.shift
      visited[edge[0]] = true
      if edge[1] == true
        edges[edge[0]].each do |connected_edge|
          if connected_edge[1] == false
            mincover << connected_edge[0]
            edges_to_be_analyzed << connected_edge
          end
        end
      elsif edge[1] == false
        edges[edge[0]].each do |connected_edge|
          if connected_edge[1] == true
            edges_to_be_analyzed << connected_edge
          end
        end
      end
   end


#adds the nodes that do not belong to
#alternating paths to the minimal cover
    (0..19).each do |person|
      edges[person].each do |v|
        if v[1] == true && visited[person] == false
          mincover << person
        end
      end
    end

    mincover &= mincover
    @minimal_cover = mincover
  end

  def minimal_vertex_cover
    @minimal_cover
```

```
  end
end
```

4 Appendix B - code for the heuristics problem

```
$K = 4

class Knapsack
        def initialize size, profits, weights, capacity, filename
                @size = size
                @profits = profits
                @weights = weights
                @capacity = capacity
                @output = File.new('results', 'a+')
                @name = filename

        end

        def output_header
                @output.puts "\n\n---------------------PROBLEM #{@name} -------
                --------------------"
        end

        def optimal_heuristic
                #initializes the main matrix
                @DP_matrix = Array.new(@size+1)
                .collect{|e| e = Array.new(@capacity+1, -1)}

                #fills the first row/col with zeros
                @DP_matrix[0].collect!{|e| e = 0}
                for i in 0..@size
                        @DP_matrix[i][0] = 0
                end

                #fills the matrix
                for i in 1..@size
                        for w in 1..@capacity
                         if(w < @weights[i-1])
                                 @DP_matrix[i][w] = @DP_matrix[i-1][w]
                         else
                                 @DP_matrix[i][w] = [@DP_matrix[i-1][w], @profits[i-1] \
                                 + @DP_matrix[i-1][w - @weights[i-1]]].max
                         end
                        end
                end

                items = Array.new

                i = @size
```

```
        k = @capacity
        while i >= 0 && k >= 0
                if(@DP_matrix[i][k] != @DP_matrix[i-1][k])
                        items << i
                        k -= @weights[i-1]
                end
                        i -= 1
        end

        total_weight = 0
        items.each do |i|
                        total_weight += @weights[i-1]
                end

        #sometimes a last item is added for no known reason. we are sorry
        if total_weight > @capacity
                @output.puts "############### OPTIMAL HEURISTIC ################"
                @output.puts "problem size: #{@size}"
                @output.puts "Total profit: #{@DP_matrix.last.last}"
                @output.print "items: {"
                items[0..-2].each do |i|
                 @output.print "(p: #{@profits[i-1]}, w: #{@weights[i-1]}) "
                end
                @output.puts "}"
                @output.print: "knapsack weight:"
                @output.print: "#{total_weight - @weights[items.last]}"
                @output.print: "(max capacity: #{@capacity})"
                @output.puts "################################################"
        else
                @output.puts "############### OPTIMAL HEURISTIC ################"
                @output.puts "problem size: #{@size}"
                @output.puts "Total profit: #{@DP_matrix.last.last}"
                @output.print "items: {"
                items[0..-2].each do |i|
                        @output.print "(p: #{@profits[i-1]}, w: #{@weights[i-1]})
                end
                @output.puts "}"
                @output.print "knapsack weight: #{total_weight} "
                @output.puts "(max capacity: #{@capacity})"
                @output.puts "################################################"
        end
end

def sahni_heuristic
        #sorts the profits/weights arrays in
        #non-incresing order for ratio profit/weight
        custo_beneficio = Array.new(@size)
        @size.times do |i|
                custo_beneficio[i] = @profits[i].to_f / @weights[i]
        end
```

```
custo_beneficio_sorted = custo_beneficio.sort

profits_sorted = Array.new
weights_sorted = Array.new
@size.times do |i|
        index = custo_beneficio_sorted.index(custo_beneficio[i])
        profits_sorted[i] = @profits[index]
        weights_sorted[i] = @weights[index]
end

@weights = weights_sorted
@profits = profits_sorted

#initial settings
_S = Array.new
_PI = 0

#gets all the combinations up to k elements
combinations = Array.new
for i in 1..$K do
        combinations = combinations +
                Array(0..@size-1).combination(i).to_a
end

#for each combination, gets its total profit and weight
combinations.each do |combination|
        combination_w_sum = 0
        combination_p_sum = 0
        combination.each do |i|
                combination_p_sum += @profits[i]
                combination_w_sum += @weights[i]
        end
        #if total w doesnt exceed capacity, applies GREEDY
        if combination_w_sum <= @capacity
                remaining_capacity = @capacity - combination_w_sum
                j = 0
                _T_S = Array.new
                _T_PI = 0

                while j < @size && remaining_capacity > 0
                        #ignores elements in the combination
                        if @weights[j] <= remaining_capacity \
                        && combination.index(j).nil?
                                _T_S << j
                                _T_PI += @profits[j]
                                remaining_capacity -= @weights[j]
                        end
                        j += 1
                        #if the found sum is higher than the current max
```

```
                                        #sum, then saves it
                                        if combination_p_sum + _T_PI > _PI
                                                _PI = combination_p_sum + _T_PI
                                                _S = combination + _T_S
                                        end
                                end
                        end

                end

                @output.puts "#################### SAHNI HEURISTIC ####################"
                @output.puts "problem size: #{@size}"
                @output.puts "total profit: #{_PI}"
                @output.print "items: {"
                result = 0
                _S.each do |i|
                        result += @weights[i]
                        @output.print "(p: #{@profits[i]}, w: #{@weights[i]}) "
                end
                @output.puts "}\ntotal weight: #{result} (max capacity: #{@capacity})"

                @output.puts "############################################################"
        end
end


problems = Array.new
#=begin
entries =  Dir.glob('*.txt')
entries.each do |filename|
        File.open(filename, 'r+') do |file|
                plaintext = file.read
                plaintext = plaintext.split("\r\n")
                problems << Knapsack.new(plaintext[0].to_i,
                                        plaintext[1].split(" ").collect{|x| x = x.to_i},
                                        plaintext[2].split(" ").collect{|x| x = x.to_i},
                                        plaintext[3].to_i,
                                        filename)
        end
end

f = File.new('benchmarks', 'w')

f.puts "times for optimal_heuristic:"
problems.each_with_index do |problem, i|
        time = Time.now
        problem.output_header
        problem.optimal_heuristic
        f.puts "#{Time.now - time}"
end
```

```
$K = 3
f.puts "times for sahni_heuristic with k=#{$K}"
problems.each_with_index do |problem, i|
        time = Time.now
        problem.output_header
        problem.sahni_heuristic
        f.puts "#{Time.new - time}"
end
```