Algorithms and Complexity. Exercise session 5

Flows. Reductions

Altered flow

a) Describe an efficient algorithm that finds a new maximum flow if the capacity of a particular edge *increases* by one unit.

   Algorithm time complexity should be linear, ie $O(|V| + |E|)$.

b) Describe an efficient algorithm that finds a new maximum flow if the capacity of a particular edge *decreases* by one unit.

   Algorithm time complexity should be linear, ie $O(|V| + |E|)$.

Solution to Altered flow

a) Suppose the edge from u to v increases its capacity by one. From previous maximum flow Φ, just make a new iteration of Ford-Fulkerson algorithm with the modified graph: The residual flow graph increases the capacity of edge (u, v) by one. Make a graph search (in time $O(|V|+|E|)$) to see if there is any path in the residual flow graph along which the flow can increase. If there is one, it must be a flow of size one (because all flows are integers). If there is no flow in the residual flow graph, Φ is still the maximum flow.

b) Suppose the edge from u to v decreases its capacity by one. If previous maximum flow Φ didn't use the full capacity of (u, v) the decrease doesn't count at all. Otherwise the flow is updated as follows:

   Since the flow entering u is one unit more than the flow leaving u and the flow entering v is one unit less than the flow leaving v, there is no way to transfer a unit flow from u to v. Thus, the search in the residual flow graph for a path from u to v along which the flow can increase by one. This is done by a graph visit in time $O(|V| + |E|)$. If such path exists, we update Φ with the flow.

   If there is no such path, we must reduce the flow from s to u and from v to t by one unit. We do this by finding a path from u to s in the residual flow graph along which the flow can increase by one and a path from t to v along which flow can increase by one. (There must be such paths, because we had a flow from s to t via (u, v).) Then, update Φ with these two flows.

$\square$

Quick bin packing *Bin packing* is the following problem. You are given n objects, each weighting between 0 and 1 kg, and a number of boxes to place the objects in. The goal is to find the minimum number of boxes needed to store n objects with no box containing more than 1 kg.

This is a well-known problem which is difficult to solve exactly (it's a so called *NP-complete problem*). Therefore, it maight be sufficient to find a solution which is not optimal, by using the following simple algorithm:

Assume both objects and boxes are numbered from 1 to n. Pick one object at a time (sequentially) and put it in the first box which can handle it (ie the box with a remaining weight which can handle the object).

Your task is to describe how this algorithm can be implemented so that it runs in time $O(n \log n)$ (in the worst case with unit cost). To achieve this, you will have to build a heap-like data structure in which you can quickly look up the first box that should hold the current object.

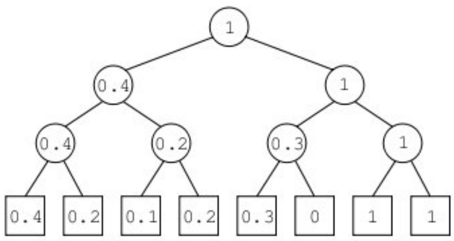
Solution to Quick bin packing
Since n objects should be placed in the boxes in time $O(n \log n)$ we need a way to put each object in a box in time $O(\log n)$. As the number of boxes can be up to n the algorithm must reject half of the boxes at each step. If so, we have rejected all boxes except one in $\log n$ steps, hence we know which box to put the object.

There are two criteria for rejecting boxes:

1. If the box can not accommodate the object
2. If the box has a higher number than the first box that can accommodate the object.

In order to reject half of the boxes in one search, we have to keep track of the weight of the heaviest object that can be placed in the first half of boxes and in the second half of boxes. This should be done recursively for each half.

The data structure thus becomes a complete binary tree of $\log n$ levels, where the tree leaves are the boxes. In each leaf we store the weight that the corresponding box holds (initially 1). In each internal tree node, we store the largest of the son's values. The data structure now looks like a heap with the largest value on top. Here is an example of eight boxes that are filled with 0.6, 0.8, 0.9, 0.8, 0.7, 1, 0 and 0 kg:



The algorithm puts an object with weight x; the first box that accommodates it becomes now:

```
void FindBin(double x, int i)
{ if (i >= n)                   /* Is this a leaf (ie. a box)? */
    H[i] = H[i] - x;
  else {
    if (H[2*i] >= x)            /* Can left son accommodate x? */
      FindBin(x, 2*i);          /* Yes, go to the left subtree. */
    else
      FindBin(x, 2*i+1);        /* No, go to the right subtree. */
    H[i] = max(H[2*i],H[2*i+1]); /* Update the current node. */
  }
}
```

The procedure starts with `FindBin(x,1)`. □

Negative reduction In previous exercise we described an algorithm that finds an approximate solution to bin packing problem. The algorithm works by placing each object in the first box that can handle it. The goal was to implement the algorithm in time $O(n \log n)$. Show that $\Omega(n \log n)$ is a lower bound for the algorithm time complexity.

Solution to Negative reduction
As usual for lower bounds in $\Omega(n \log n)$, we construct a reduction of the problem of sorting n numbers to our problem. We know it is impossible to sort n numbers by means of comparisons faster than $\Omega(n \log n)$. This applies even if the n numbers are permuted integers from 1 to n, and even if we allow to make an initial linear re-scaling of numbers. Let us show now how we can use quick bin packing algorithm to sort these numbers.

Idea: We rescale the numbers to be sorted by a factor of $1/(2n)$ so that they lie between $1/(2n)$ and $1/2$. Then we construct an input instance, containing objects that will fill the boxes, such that we can fit exactly the numbers from $1/(2n)$ to $1/2$ (in order from box 1 to box n). If we can place the objects and rescale them according to the algorithm, the numbers will be sorted.

Suppose that $v[1..n]$ are the objects to be sorted and the key field is *key*. Furthermore, suppose the algorithm returns, for each box, a list of indexes of the objects that it contains.

Sort($v[1..n]$) =
  $p \leftarrow 1/(2n)$
  for $i \leftarrow 1$ **to** n **do** $x[i] \leftarrow 1 - i \cdot p$
  for $i \leftarrow n+1$ **to** $2n$ **do** $x[i] \leftarrow v[i-n] \cdot p$
  $L[1..n] \leftarrow$FirstFit($x[1..2n]$)
  for $i \leftarrow 1$ **to** n **do** $res[i] \leftarrow L[i][2]$ // take the second object out of each box
  return $res[1..n]$

Function Sort reduces the sorting problem to FirstFit. Reduction (not counting the call to FirstFit) takes time $O(n)$, so if you could implement FirstFit in time less than $\Omega(n \log n)$, it would be possible to sort n numbers faster than $\Omega(n \log n)$, which is impossible. □

Positive reduction A useful way to solve problems is by finding a reduction to a problem which you already know how to solve. You should use this method to solve the following problem.

> INPUT: A connected undirected graph $G = (V, E)$ and a positive integer K between 1 and $|V|$.
> PROBLEM: Is it possible to remove K edges from graph G to make it disconnected (ie. divided into connected components)?

Solution to Positive reduction
Suppose X is the minimal number of edges whose removal makes G disconnected. (This means that no strict subset of X makes G disconnected.) Then G consists of two components, and all edges of X go between these two components. (Otherwise, X can not be minimal.) Thus, X corresponds to a cut in the graph (a division of the vertices in two parts). The number of edges in X is the cut size. This means that the minimum number of edges that we must remove so that G becomes disconnected — call this $\lambda(G)$ — is equal to the size of a minimum cut (V_1, V_2) in G.

Minimum cut is the same as the maximum flow. We shall therefore try to reduce our problem problem to a maximum flow between two vertices s and t in a graph. We extend G to a flow graph G', by giving each bidirectional edge capacity 1 and find $\lambda(G)$ by calculating the maximum flow from s to t for different s and t. However, we don't need to vary both. If we choose an arbitrary s, it must belong to one of the sets V_1 and V_2 above. Varying t over all verteces in addition to s, we are guaranteed to reach a vertex in the second set. Therefore, the answer is an arbitrary vertex s.

$$\lambda(G) = \min_{t \in V - \{s\}} \{\text{MaxFlow}(G', s, t)\}.$$

Now it remains to determine if $K \geq \lambda(G)$. We answer NO if $K < \text{MaxFlow}(G', s, t)$ for all $t \in V - \{s\}$ and YES otherwise.

The flow algorithm is called $|V|-1$ times. Each run takes time $O(|V|^3)$. So then, the complexity of our algorithm is $O(|V|^4)$. □

Reduction between decision-, optimization- and construction problems Assume that the algorithm GraphColouring(G,k) in time $T(n)$ (where n is the number of vertices in G) is 1 iff the vertices of G can be colored with k colors and no edge has both ends of the same color.

a) Construct an algorithm that given a graph G with n vertices determines the minimum number of colors needed to color G. The time complexity should be $O(\log n \cdot T(n))$.

b) Construct an algorithm that given a graph G with n vertices colors each vertex with the minimum number of colors in time $O(P(n)T(n))$, where $P(n)$ is a polynomial.

Solution to Reduction between decision-, optimization- and design problems

a) We know that the colors are between 1 and n. Do binary search in this interval by using the algorithm GraphColouring to find a k so that GraphColouring$(G, k) = 1$ and GraphColouring$(G, k-1) = 0$. This procedure requires minimal coloring to have k colors. We need at most $\log n$ iterations to get down to 1. The time complexity is therefore $O(\log n \cdot T(n))$.

b) Find the minimum number of colors k with the method above. We color the vertices of G with colors from 1 to k using the following algorithm:

$CreateColouring(G = (V, E),\ k)$=
$u \leftarrow$ first vertex of V
$C \leftarrow \{u\}; u.colour \leftarrow k$
foreach $v \in V - \{u\}$ **do**
  if $(u, v) \notin E$ **then**
    if GraphColouring$((V, E \cup \{(u, v)\}), k) = 1$ **then** $E \leftarrow E \cup \{(u, v)\}$
    else $C \leftarrow C \cup \{v\}; v.colour \leftarrow k$
if $k > 0$ **then** $CreateColouring((V - C, E), k - 1)$

GraphColouring is called at most once for each pair of vertices in the graph. Time complexity of the algorithm is therefore $O(\log n \cdot T(n) + n^2 \cdot T(n)) = O(n^2 \cdot T(n))$. □
