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 will 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 will be linear, ie $O(|V| + |E|)$.

Solution to Altered flow

a) Suppose that the edge from u to v increases its capacity by one. From the 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) with 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, there 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 that the edge from u to v decreases its capacity by one. If the previous maximum flow Φ didn't use the full capacity of (u, v) such change doesn't count at all. Otherwise we update the flow as follows:

Since the flow entering u is one unit more than the flow leaving it and the flow entering v is one unit less than the flow leaving it, there is no way we must transfer a unit flow from u to v. Thus, 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 there is such a path we update Φ with the flow.

If there is no such a path, we must reduce the flow from s to u and from v to t with 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 pieces of metal gadgets, each weighting between 0 and 1 kg. Moreover, you are given a number of large but short boxes to place such gadgets in. The goal is to find the minimum number of boxes needed to store n metal gadgets with no box containing more than 1kg.

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

Assume that both metal gadgets and boxes are numbered from 1 to n. Pick one gadget at a time (sequentially) and put it in the first box which can handle it (ie the box with the lowest weight which can handle the gadget).

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 holds the current gadget.

**Solution to Quick bin packing**

Since n gadgets should be placed in the boxes in time $O(n \log n)$ we need a way to put a gadget 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, you have rejected all boxes except one in $\log n$ steps, and you know which box to put the gadget.

There are only two criteria for rejecting boxes:

1. If the box can not accommodate the gadget

2. If the box has a higher number than the first box that accommodates the gadget.

In order to reject half of the boxes with one search, we have to keep track of the weight of the heaviest gadget, which can be placed in the first half of the boxes and in the second half of the boxes. We must keep track of 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 amount that the corresponding box holds (initially 1). In each internal node in the tree, we store the largest of the son's values. The data structure now looks like a heap with the largest value at the 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:
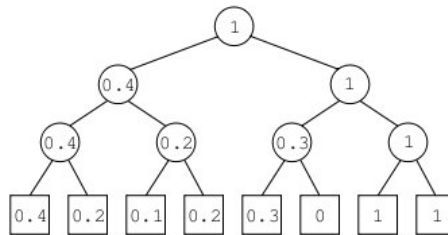
The algorithm puts a gadget with weight x; the first box that accommodates it becomes now:

```
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 the last exercise we described an algorithm that finds an approximate solution to the bin packing problem. The algorithm works by placing each gadget 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 the lower bounds that are $\Omega(n \log n)$, 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 the numbers. Let us show you how we can use the 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 using gadgets that will fill the boxes so that we can fit exactly the numbers from $1/(2n)$ to $1/2$ (in order from box 1 to box n). If you can place the gadgets and rescale them according to the algorithm, the numbers will be sorted.

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

```
Sort(v[1..n]) =
    p ← 1/(2n)
    for i ← 1 to n do x[i] ← 1 − i · p
    for i ← n + 1 to 2n do x[i] ← v[i − n] · p
    L[1..n] ← FirstFit(x[1..2n])
    for i ← 1 to n do res[i] ← L[i][2] // take the second gadget 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 to find 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 enough to remove K edges from the graph G to make it disconnected (ie. divided into connected components)?

Solution to Positive reduction

First, we should try to understand the problem. Suppose X is the minimal number of edges whose removal makes G disconnected. (This means that no strict subset of X make 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.

Minimal 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. If 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 meet 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 $O(|V|^3)$ (which you do not need to know by heart). So then, the complexity of our algorithm is $O(|V|^4)$. $\quad\square$

Reduction between decision-, optimization- and design problems Assume that the algorithm GraphColouring(G,k) at 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 will 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 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 want to color the vertices of G with colors from 1 to k. The following algorithm does it:

$CreateColouring(G = (V, E),\ k)=$
$u \leftarrow$ 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. The time complexity of the algorithm is therefore $O(\log n \cdot T(n) + n^2 \cdot T(n)) = O(n^2 \cdot T(n))$. $\quad\square$