Albert-Ludwigs-Universität, Institut für Informatik
Prof. Dr. Fabian Kuhn
M. Ahmadi, P. Schneider, J. Uitto

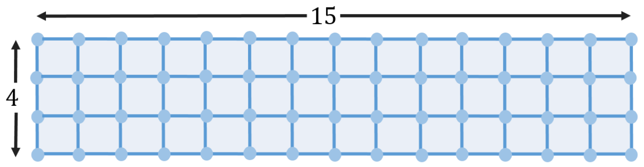
November 30, 2017

Algorithm Theory - Winter Term 2017/2018
Exercise Sheet 2

Hand in by Thursday 10:15, November 16, 2017

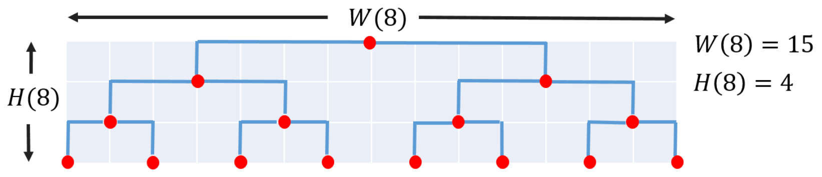
Exercise 1: Tree Embedding into Grids (4+6 Points)

A $n \times m$ grid graph is a graph $G = (V_G, E_G)$ with nodes $V_G := \{(i, j) \mid i \in \{1, \dots, n\}, j \in \{1, \dots, m\}\}$. These nodes are embedded in the Euclidian plane $\mathbb{R}^2$ and connected with edges as exemplified by the following 4×15 grid graph.



An embedding of a tree $T = (V_T, E_T)$ into a grid graph $G = (V_G, E_G)$ is defined as a one to one mapping of V_T to a subset of V_G, which satisfies the following condition. There exists a set P_G of vertex-disjoint paths[1] in G such that for each $\{u, v\} \in E_T$, there is a path $p \in P_G$ connecting u' to v', when u is mapped to u', and v is mapped to v'.

(a) We can embed a complete binary tree with n leaves into a grid, such that the nodes with height i of the tree are placed in the i^{th} row of the grid. Below you see the embedding of a tree with 8 leaves into a 4×15 grid as an example.



By this way of embedding, show that we need a grid of size[2] $\Theta(n \log n)$ to embed a complete tree with n leaves. To do so, write down the recurrence relations for the width $W(n)$ and the height $H(n)$ of the grid.

(b) Find a more efficient way of embedding a complete binary tree and show that it needs a grid of size $\Theta(n)$, if the tree has n leaves. Write down the recurrence relations for the width $W(n)$ and the height $H(n)$ of the grid.

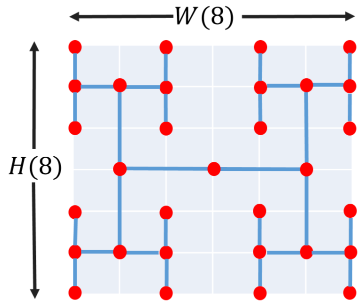
[1]We define vertex-disjoint paths in G as paths that may only have common endpoints but are disjoint otherwise.
[2]The size of a $n \times m$ grid graph is simply $n \cdot m$.

Sample Solution

(a) By doubling the number of the nodes the width of the grid doubles and the height of the grid is increased by one. Therefore, we can have the following recurrence relations for the width and the height of the grid.

$W(n) = 2W(n/2) + \Theta(1) \Rightarrow W(n) = \Theta(n)$
$H(n) = H(n/2) + \Theta(1) \Rightarrow H(n) = \Theta(\log n)$
Therefore, size of the grid is $\Theta(n \log n)$.

(b) To provide a more efficient way of embedding, we can avoid to place the nodes with same height in the same row of the grid. You can see below how we can embed a complete binary tree with 16 leaves in a 7×7 grid.



Here, we have $W(n) = H(n) = 2W(n/4) + \Theta(1)$. Therefore, $W(n) = H(n) = \Theta(\sqrt{n})$. Hence, the size of the grid is $\Theta(n)$.

Exercise 2: Polynomial to the power of k (4+6 Points)

Given a polynomial $p(x)$ of degree n in coefficient representation and an integer $k \geq 2$, the goal of this problem is to compute the k^{th} power $p^k(x)$ of $p(x)$ in an efficient way. For simplicity, we assume that k is a power of 2, that is, $k = 2^\ell$ for some integer $\ell \geq 1$.

(a) Describe an efficient algorithm to compute $p^k(x)$ polynomial using the fast polynomial multiplication procedure from the lecture.

(b) What is the asymptotic runtime of your algorithm in terms of k and n? Explain your answer.

Sample Solution

(a) We compute the k^{th} power of the polynomial $p(x)$ in coefficient representation by iterative multiplication, as

$$p(x)^k = (\dots((p(x)^2)^2 \dots)^2$$

where square is taken $\log k$ times (assuming k is a power of 2) with the procedure using the FFT from the lecture.

Algorithm 1 FastPotenciation(p)

```
$q \leftarrow p$
for i from 1 to $\ell$ do
    $q \leftarrow$ FastPolynomialMultiplication($q, q$)        ▷ procedure from the lecture using FFT
return q
```

(b) We know that using the FFT algorithm from the lecture, two polynomials of degree n can be multiplied in $\mathcal{O}(n \log n)$ time. Notice that in every iterative step i of the algorithm, we need to multiply two polynomials of degree $(2^{i-1} \cdot n)$ and get a polynomial of degree $(2^i \cdot n)$. Unwrapping the definition of the $\mathcal{O}$-Notation, we know that for large n and for a $c > 0$, we can compute the product of two polynomials of degree $(2^{i-1} \cdot n)$ in $c \cdot 2^i \cdot n \cdot \log(2^i \cdot n)$. To compute $p(x)^k$ (where $k = 2^\ell$ for some ℓ), there will be $\log k$ such iterations.

Therefore, the asymptotic running time of the algorithm (i.e., iterative multiplication of two same degree polynomials) would be:

$$\sum_{i=0}^{\log k - 1} c \cdot 2^i \cdot n \cdot \log(2^i \cdot n) \qquad \text{[where } c \text{ is the constant in the } \mathcal{O}(n \log n) \text{ bound]}$$

Let us compute this sum:

$$\begin{aligned}
\sum_{i=0}^{\log k - 1} c \cdot 2^i \cdot n \cdot \log(2^i \cdot n) &= cn \cdot \sum_{i=0}^{\log k - 1} 2^i \cdot (i + \log(n)) \\
&= cn \cdot \sum_{i=0}^{\log k - 1} i \cdot 2^i + cn \cdot \log(n) \cdot \sum_{i=0}^{\log k - 1} 2^i \\
&\leq cn \cdot \Big((\log(k) - 1) \cdot 2^{\log(k)}\Big) + cn \cdot \log(n) \cdot \Big(2^{\log(k)} - 1\Big) \\
&\leq cnk \cdot \log(nk)
\end{aligned}$$

Hence, the asymptotic running time of the algorithm is $\mathcal{O}(nk \cdot \log(nk))$.

Exercise 3: Greedy Algorithm (10 Points)

In the following, a *unit fraction* is a fraction where the numerator is 1 and the denominator is some integer larger than 1. For example 1/4 or 1/384 are unit fractions.

It is well-known that every rational number $0 < q < 1$ can be expressed as a sum of pairwise distinct unit fractions, e.g., we can write $\frac{4}{13}$ as

$$\frac{4}{13} = \frac{1}{5} + \frac{1}{13} + \frac{1}{32} + \frac{1}{65}.$$

Interestingly such a decomposition into distinct unit fractions can be computed using a simple greedy algorithm.

In the following, assume that you are given two positive integers a and b such that $b > a$. Design a greedy algorithm to compute integers $0 < c_1 < c_2 < \cdots < c_k$ such that

$$\frac{a}{b} = \frac{1}{c_1} + \frac{1}{c_2} + \cdots + \frac{1}{c_k}.$$

Prove that your greedy algorithm always works and that it decomposes $\frac{a}{b}$ into at most a unit fractions.

You can assume that your algorithm can deal with arbitrarily large integer numbers. Note that for the fraction $\frac{4}{13}$, the standard greedy algorithm computes a decomposition which is different from the one given above.

Sample Solution

Algorithm Description:

Here we introduce a greedy algorithm which solves the problem in at most a steps for any input fraction a/b. In the i^{th} step, the algorithm uses the remaining fraction from the previous step and outputs c_i and a remaining fraction a_i/b_i.

Algorithm 2 `UnitFracDecomp`(a, b)

```
$i \leftarrow 0, a_0 \leftarrow a, b_0 \leftarrow b$
while $a_i \neq 1$ do
    $i \leftarrow i + 1$
    Let $\frac{1}{c_i}$ be the largest unit fraction $\leq \frac{a_{i-1}}{b_{i-1}}$        $\triangleright$ $c_i := \lceil \frac{b_i}{a_i} \rceil$ does the trick.
    $\frac{a_i}{b_i} \leftarrow \frac{a_{i-1}}{b_{i-1}} - \frac{1}{c_i}$
return $(\frac{1}{c_1}, \ldots, \frac{1}{c_i}, \frac{1}{b_i})$
```

To calculate c_i and a_i/b_i in each step the algorithm does the following. If the remaining fraction from the previous step is a unit fraction ($a_{i-1} = 1$) then $c_i := b_{i-1}$ and the algorithm stops. Otherwise, the algorithm calculates the largest unit fraction $\frac{1}{c_i}$ smaller than $\frac{a_{i-1}}{b_{i-1}}$ from the previous step,

$$\frac{1}{c_i} < \frac{a_{i-1}}{b_{i-1}} < \frac{1}{c_i - 1}.$$

Then, it calculates the remaining fraction as follows.

$$\frac{a_i}{b_i} := \frac{a_{i-1}}{b_{i-1}} - \frac{1}{c_i},$$

where c_i is a positive integer.

Example:

Let us go through an example to see the algorithm more clearly. Consider the example in the exercise, $a/b = 4/13$.
1st step, $a/b = 4/13$: The largest unit fraction smaller than 4/13 is 1/4. Then,

$$c_1 = 4, a_1/b_1 = 4/13 - 1/4 = 3/52.$$

2nd step, $a_1/b_1 = 3/52$: The largest unit fraction smaller than 3/52 is 1/18. Then,

$$c_2 = 18, a_2/b_2 = 3/52 - 1/18 = 1/468.$$

Since the remaining fraction is a unit fraction, $c_3 = 1/468$, the algorithm stops and decomposition into sum of unit fractions is complete:

$$\frac{4}{13} = \frac{1}{4} + \frac{1}{18} + \frac{1}{468}$$

Analysis:

Now, let us show that the above proposed algorithm always stops in a finite number of steps (at most a steps) and works properly. To do this, we mention two claims.

Claim 1. In each step, the numerator of the remaining fraction is positive and strictly smaller than the numerator of the remaining fraction from previous step.
Proof. Fix some step with input fraction of x/y and the generated unit fraction of $1/c$. Then the remaining fraction is $\frac{xc-y}{yc}$. To prove the claim we should show that $xc - y$ is positive and strictly smaller than x. We assumed that $1/c$ is smaller than x/y and also it is the largest possible unit fraction. Based on the algorithm description we have

$$\frac{1}{c} < \frac{x}{y} \Rightarrow xc - y > 0 \qquad (1)$$

$$\frac{1}{c-1} > \frac{x}{y} \Rightarrow xc - y < x \qquad (2)$$

(1) and (2) prove claim 1.

Claim 2. If the algorithm stops after $t \le a$ steps there does not exist $i \ne j \in \{1, \dots, t\}$ such that $c_i = c_j$.
Proof. Fix some step r during the execution and let us assume that for round $r-1$ the remaining fraction is a_{r-1}/b_{r-1} and the calculated integer is c_r. Then the remaining fraction for round r is

$$\frac{a_r}{b_r} = \frac{a_{r-1}}{b_{r-1}} - \frac{1}{c_r} = \frac{a_{r-1}c_r - b_{r-1}}{b_{r-1}c_r}.$$

Since $1/c_r$ is the largest unit fraction smaller than a_{r-1}/b_{r-1} we have

$$\begin{aligned}\frac{1}{c_r - 1} > \frac{a_{r-1}}{b_{r-1}} &\Rightarrow a_{r-1}c_r - a_{r-1} < b_{r-1}\\ &\Rightarrow a_{r-1}c_r < a_{r-1} + b_{r-1} < 2b_{r-1}\\ &\Rightarrow \frac{a_{r-1}}{b_{r-1}} - \frac{1}{c_r} < \frac{1}{c_r}\\ &\Rightarrow \frac{a_r}{b_r} < \frac{1}{c_r}.\end{aligned}$$

Therefore, c_r never be selected for any round $r' \ge r$.

Based on claim 1, the numerator of the remaining fractions strictly decreases after each step and it is always positive. Therefore, there exists some step with remaining fraction of numerator 1. As a result the algorithm stops after at most a steps and based on claim 2 it generates the desired sequence of integers.

Exercise 4: Matroids (6+4 Points)

(a) For a graph $G = (V, E)$, a subset $F \subseteq E$ of the edges is called a forest iff (if and only if) it does not contain a cycle. Let $\mathcal{F}$ be the set of all forests of G. Show that $(E, \mathcal{F})$ is a matroid.

Hint: A forest with k edges and n nodes has $n - k$ connected components.

(b) For a matroid (E, I), a maximal independent set $S \in I$ is an independent set that cannot be extended. Thus, for every element $e \in E \setminus S$, the set $S \cup \{e\} \notin I$.

What are the maximal independent sets of the matroid in (a)?

Sample Solution

(a) We verify the three properties of a matroid $(E, \mathcal{F})$ with ground set E and independent sets $\mathcal{F}$:

(i) *Independence of empty set:* $\emptyset \in \mathcal{F}$.

(ii) *Hereditary property:* For all $F \subseteq E$ and $F' \subseteq F$: $F \in \mathcal{F} \Longrightarrow F' \in \mathcal{F}$.

(iii) *Augmentation property:* $F_1, F_2 \in \mathcal{F}$: $|F_1| > |F_2| \Longrightarrow \exists e \in F_1 \setminus F_2$ with $F_2 \cup \{e\} \in \mathcal{F}$.

To (i): $\emptyset$ does not have a cycle and is therefore a forest, i.e. $\emptyset \in \mathcal{F}$.

To (ii): Let $F \subseteq E$ and $F' \subseteq F$. This means we obtain F' by removing edges from F. Given that $F \in \mathcal{F}$ is acyclic and since removing edges does not create any cycles, we have that F' is also acyclic, i.e., $F' \in \mathcal{F}$.

To (iii): Let $F_1, F_2 \in \mathcal{F}$ with $|F_1| > |F_2|$. From the *Hint* we know that F_1 has fewer connected components than F_2. Hence there exists at least one component C_1 in F_1 whose vertices belong to at least two different components of F_2. Since C_1 is connected, there exists an edge $e \in F_1$ connecting the two aforementioned components of F_2. Hence adding e to F_2 does not create any cycle in F_2 and therefore $F_2 \cup \{e\} \in \mathcal{F}$.

(b) The maximal independent sets are spanning forests. I.e., a set of spanning trees for *all*(!) connected components of G is maximal independent.