Exercise 1, Discrete Mathematics for Bioinformatics

Sascha Meiers, Martin Seeger

Winter term 2011/2012

1.1 MST Approximation

a) The following complete graph with six nodes satisfies the triangle inequality. Marked in red are the edges that are selected by the Kruskal algorithm.

The following algorithm (borrowed from Wikipedia) delivers a 2-approximation:

1. Construct the minimum spanning tree.
2. Duplicate all its edges. This gives us an Eulerian graph.
3. Find a Eulerian cycle in it. Clearly, its length is twice the length of the tree.
4. Convert the Eulerian cycle into the Hamiltonian one in the following way: walk along the Eulerian cycle, and each time you are about to come into an already visited vertex, skip it and try to go to the next one (along the Eulerian cycle).

Carrying out this algorithm on the above graph visits the vertices in the following order:

$$v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to v_6$$
$$\to v_5 \text{ (skipped)} \to v_4 \text{ (skipped)} \to v_3 \text{ (skipped)} \to v_2 \text{ (skipped)} \to v_1.$$

The resulting approximation to the TSP is displayed below. It happens to be the exact solution.

b) A 2-approximation is worse by at most a factor of two than the optimal solution for any input (i.e. for any graph). The Eulerian trail constructed in the intermediate step has two times the weight of a minimal spanning tree. By repeatedly taking shortcuts, its weight can only be reduced, hence

$$w(\text{approximation}) \leq w(\text{Eulerian trail}) = 2w(\text{MST}) \leq 2w(\text{optimal solution}).\square$$

1.2 Landau Symbols

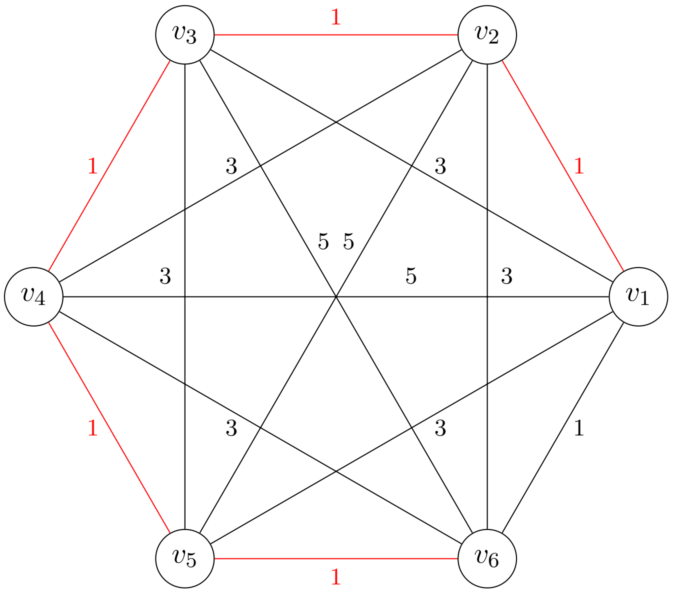
a) Let $k, l \in \mathbb{Z}$, $k > l$. $f = o(g)$ holds iff

$$\lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| = 0. \tag{1}$$

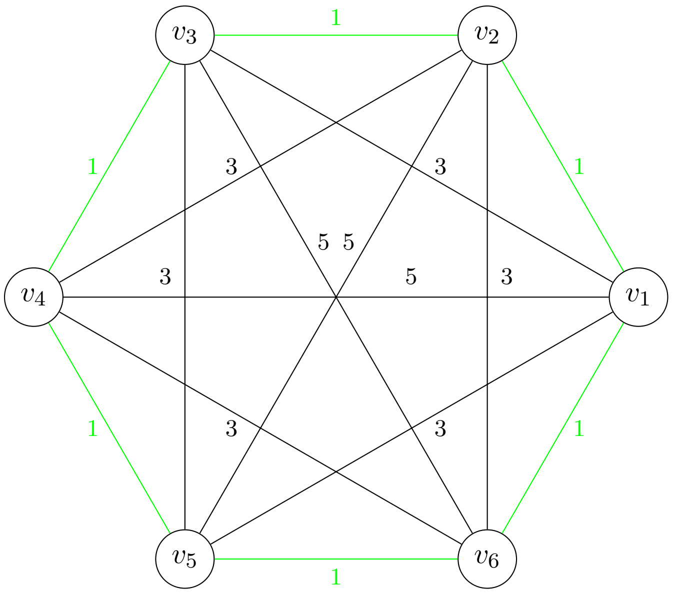
In our case,

$$\lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| = \lim_{n\to\infty} \left|\frac{n^l}{n^k}\right| = \lim_{n\to\infty} \left|\frac{1}{n^{k-l}}\right| = 0, \tag{2}$$

whence it follows that $n^l = o(n^k)$.$\square$



v_1
v_2
v_3
v_4
v_5
v_6
1
1
1
1
1
1
3
3
3
3
3
3
3
3
5
5
5



v_1
v_2
v_3
v_4
v_5
v_6
1
1
1
1
1
1
3
3
3
3
3
3
3
3
5
5
5

b) Let $k, l \in \mathbb{N}$, $k > l$. In general, $f = \Theta(g)$ iff $f = O(g)$ and $g = O(f)$. We use the definition $f = O(g)$ iff

$$0 \leq \limsup_{n\to\infty} \left| \frac{f(n)}{g(n)} \right| < \infty. \tag{3}$$

In our case,

$$\limsup_{n\to\infty} \left| \frac{n^k + n^l}{n^k} \right| = \limsup_{n\to\infty} \left| 1 + \frac{1}{n^{k-l}} \right| = 1, \tag{4}$$

and

$$\limsup_{n\to\infty} \left| \frac{n^k}{n^k + n^l} \right| = \limsup_{n\to\infty} \left| 1 - \frac{n^l}{n^k + n^l} \right| = \limsup_{n\to\infty} \left| 1 - \frac{1}{n^{k-l}+1} \right| = 1. \square \tag{5}$$

c) Counterexample: $f(n) = 2^{cn}$ with $c > 1$ is clearly $2^{O(n)}$. However,

$$\limsup_{n\to\infty} \left| \frac{2^{cn}}{2^n} \right| = \limsup_{n\to\infty} 2^{(c-1)n} = \infty, \tag{6}$$

hence $f \neq O(2^n)$.$\square$

1.3 Amortized Analysis

The array size must be multiplied by a constant factor in order to guarantee an armotized constant runtime. We would prefer to double the size whenever the push operation is called and the array is already full.

1.3.1 The accounting method

Let's say each push operation gains us 2 EUR and copying the array costs 1 EUR per element that is copied. We want to show that no matter how many push operations are performed, our credit never falls below zero.

Let's assume an array has just been doubled to $2n$ and the credit is ≥ 0. Now we can perform the next n push operations, each one adding 2 EUR to our account. When the number of 2n is reached, we own at least $2n$ EUR. Then the complete array with all its $2n$ elements is copied again, which costs exactly $2n$ EUR. Afterwards the account remains ≥ 0.

1.3.2 The potential method

We take $\phi = 2k - n$ as the potential function, where k is the number of elements in the array and n the allocated size. For the empty data structure D_0, this potential is zero ($k = n = 0$). Further values are $D_1 = 1$ ($n = k = 1$), $D_2 = 2$, $D_3 = 2$, $D_4 = 4$, $D_5 = 2$, $D_6 = 4$, ...

One can easily see that this function never falls below zero, since the array is always at least half filled. So we have $D_i \geq D_0 \; \forall i$, which a proper potential function must hold. Now we'll have a look at the amortized costs of a push operation, differing two cases:

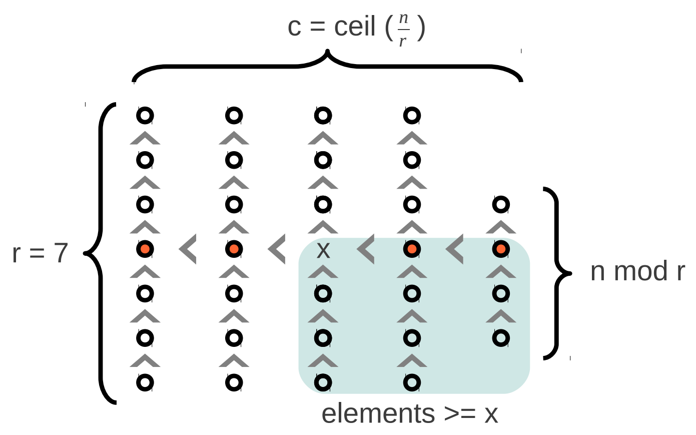
a) step i does not double the array. Then the actual cost is $c_i = 1$ and the amortized cost

$$\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + (2k - n) - (2k - 2 - n) = 3$$

b) step i doubles the array size. Then the actual cost is $c_i = 1 + k$ because k elements have to be copied into a new array. The amportized cost is

$$\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + k + (2k - 2n) - (2k - 2 - n) = 3 - k + n$$

But the array was only doubled because it was completely filled, i.e. $k = n$. This leads to a constant amortized runtime of 3 even when the array is doubled.



1.4 Analysis of Select algorithm

The Select algorithm is given a list A of n numbers (or elements of a completely ordered universe) and a rank $k \leq n$. It returns the k-th smallest element of A.

The deterministic Select algorithm guarantees a worst-case complexity of $\mathcal{O}(n)$ by smartly choosing a Pivot element. This is achieved by dividing the elements of the list into groups of 5 (in general r, meaning *row* as related to the figure) and calculating the medians of each group in constant time (since a group contains only constant many elements). The same routine is used recursively to determine the median of these medians. Finally the whole list is split up like in Quicksort, using the median of medians for Pivot element, and the search is continued in only one of both parts depending on k. As is shown in the additional material, splitting at this point drops always enough elements for the runtime to be linear in n. Here we want to investigate wether the linear-time property holds for other values of r, too.

The runtime of the Select algorithm can be written recursively as

$$T(n) = T(\left\lceil \frac{n}{r} \right\rceil) + T(g(n)) + an$$

- $T(\lceil \frac{n}{r} \rceil)$ is the runtime for the recursive call of Select in order to compute the median of medians.
- $c \cdot n$ is the runtime of dividing the list into groups of r, finding their medians (which works in $\mathcal{O}(r\ log\ r)$, i.e. in $\mathcal{O}(1)$ for each group) and finally splitting up the list.
- $T(g(n))$ is the runtime of the recursive call of Select on a smaller list. $g(n)$ is an upper bound for the length of this list and by association the key variant in enabling a linear overall runtime. For $r = 5$ we've seen that $g(n) \leq \frac{7}{10}n + 6$.

The figure shows the partial ordering we obtain by determining the median of medians. At least the elements in the lower right box are larger than or equal to the Pivot element x, but there can be more in the upper right or lower left quarter. On the other side there is at least the upper left quarter of elements that are lower than or equal to x. Let's assume r to be odd (since we then can clearly determine a median). Then the lower right quarter consists of $\lceil \frac{n}{2r} \rceil$ columns with each $\frac{r+1}{2}$ elements except for the last columns, wich contains at least one element. So we will not count the last column and we will also deduct the column of x itself (the same is done in the additional material). So the maximum number of elements in the next recursive call for an odd number r is

$$\begin{aligned} g(n) &\leq n - \left(\left\lceil \frac{n}{2r} \right\rceil - 2 \right) \cdot \frac{r+1}{2} \\ &\leq n - \left(\frac{n}{2r} - 2 \right) \cdot \frac{r+1}{2} \\ &\leq (0.75 - \frac{1}{4r}) \cdot n + (r+1). \end{aligned}$$

Due to monotonicity of the runtime function, $T(n)$ is now determined by

$$T(n) \leq T\left(\frac{n}{r} + 1\right) + T\left((0.75 - \frac{1}{4r}) \cdot n + (r+1)\right) + an$$

Now assume that $T(n) \leq c \cdot n$ for large n and a constant factor c. Then we have

$$\begin{aligned} T(n) &\leq c \cdot \left(\frac{n}{r}\right) + c + c \cdot \left((0.75 - \frac{1}{4r}) \cdot n + (r+1)\right) + an \\ &= cn\left(\frac{1}{r} + 0.75 - \frac{1}{4r}\right) + an + (r+2)c \\ &= cn - cn\left(\frac{r-3}{4r}\right) + an + (r+2)c \end{aligned}$$

In order to complete the proof, we have to show that this recursively defined runtime is at most cn, too. This depends on the term after cn: If it is at most zero, the runtime is linear. We have to differ two cases: If $r \leq 3$, then there are only positive addends in the term and the inequality obviously does not hold. Otherwise, if $r > 3$, the fraction $\frac{r-3}{4r}$ is larger than zero but less than one and we can find the factor c as needed:

$$\begin{aligned} -cn\left(\frac{r-3}{4r}\right) + an + (r+2)c &\leq 0 \\ c\left(-n\frac{r-3}{4r} + r + 2\right) &\leq -an \\ c &\geq \frac{an}{n\frac{r-3}{4r} - r - 2} && \| \text{ for } n > r+2 \end{aligned}$$

This inequality can be solved for any $r > 3$, if n is chosen large enough (for example $n > 2(r+2)$). If we choose $r = 5$, we can exactly reproduce the example from the additional material:

$$\begin{aligned} c &\geq \frac{an}{n\frac{5-3}{4\cdot 5} - 5 - 2} && \| \text{ for } n > 5+2 \\ c &\geq \frac{10an}{n - 70} \\ c &\geq 20a && \| \text{ for } n > 140, \text{ because } \frac{n}{n-70} \leq 2 \end{aligned}$$

Finally we found a constant factors c (which of course depends on the factor a for splitting the list etc.) such that the overall runtime is linear. During the derivation we found that the linear runtime only works for $r > 3$, which answers the initial question.