**MTH6105 – Algorithmic Graph Theory** **Spring 2024**

**Problem Sheet 4** **F. Fischer**

---

You are expected to **attempt all exercises** before the seminar and to **actively participate** in the seminar itself.

---

1. (a) For functions $f, g, h : \mathbb{N} \to \mathbb{R}^+$ show the following:
   (i) $f(n) + g(n)$ is $O(\max\{f(n), g(n)\})$.
   (ii) if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

   (b) Let $f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3$, where $a_0, a_1, a_2, a_3 \in \mathbb{Z}$ are constants. Show that $f(n)$ is $O(n^3)$.

   (c) Show that $2^{2n}$ is *not* $O(2^n)$. To this end, you may want to assume that it was and derive a contradiction.

   **Note:** This type of question looks intimidating, but really it is all bark and no bite. Recalling the relevant definition, all we need to do to show that $f(n)$ is $O(g(n))$ is find constants $c$ and $n_0$ with certain properties and verify that they indeed satisfy these properties.

   **Solution:**

   (a) (i) Let $n_0 = 0$ and $c = 2$. Then $f(n) + g(n) \leq c \cdot \max\{f(n), g(n)\}$ for all $n \geq n_0$, so $f(n) + g(n)$ is $O(\max\{f(n), g(n)\})$.
   (ii) Since $f(n)$ is $O(g(n))$, there exist $c > 0$ and $n_0 \geq 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$. Since $g(n)$ is $O(h(n))$, there exist $d > 0$ and $n_1 \geq 0$ such that $g(n) \leq dh(n)$ for all $n \geq n_1$. Let $e = cd$ and $n_2 = \max\{n_0, n_1\}$. Then $f(n) \leq cg(n) \leq cdh(n) = eh(n)$ for all $n \geq n_2$, so $f(n)$ is $O(h(n))$.

   (b) Let $c = |a_0| + |a_1| + |a_2| + |a_3|$ and $n_0 = 1$. Then, for $n \geq n_0$, $f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 \leq |a_0| + |a_1| n + |a_2| n^2 + |a_3| n^3 \leq (|a_0| + |a_1| + |a_2| + |a_3|) n^3 = cn^3$. Thus $f(n)$ is $O(n^3)$.

   (c) Assume for contradiction that $2^{2n}$ is $O(2^n)$, i.e., that there are constants $c$ and $n_0$ such that for all $n \geq n_0$, $2^{2n} \leq c2^n$. Let $m = \max\{n_0, \log_2(c) + 1\}$. Then $m \geq n_0$, $2^m > c$, and $2^{2m} = 2^m 2^m > c2^m$, which is a contradiction.

   **Bonus question:** Is it true for all functions $f : \mathbb{N} \to \mathbb{R}^+$ and $g : \mathbb{N} \to \mathbb{R}^+$ that $f(n)$ is $O(g(n))$ or $g(n)$ is $O(f(n))$?

---

2. Euclid's algorithm determines the greatest common divisor $\gcd(a, b)$ of two non-negative integers $a \geq b$ by setting $r_0 = a$, $r_1 = b$, and then repeating the following steps for rounds $n = 2, 3, 4, \ldots$:

   - If $r_{n-1} = 0$ then stop, output $\gcd(a, b) = r_{n-2}$.
   - Find $q_n$ and $r_n$ such that $r_{n-2} = q_n r_{n-1} + r_n$ and $0 \leq r_n < r_{n-1}$.

Note that $(r_n)_{n\geq 0}$ is a decreasing sequence of non-negative integers and the algorithm thus stops after a finite number of rounds. Note further that it makes sense to say that the size of the input of the algorithm is $\log_2 a + \log_2 b$, because this is the number of digits of $a$ and $b$ as binary numbers.

(a) If $r_n$ were to decrease by 1 in each round, how many rounds would the algorithm run for?

(b) Show that $r_n$ in fact decreases significantly over two consecutive rounds of the algorithm, namely $r_n < r_{n-2}/2$. You may want to distinguish among the three cases where $r_{n-1} = r_{n-2}$, $r_{n-2}/2 < r_{n-1} < r_{n-2}$, or $r_{n-1} \leq r_{n-2}/2$.

(c) Give an upper bound on the maximum number of rounds of the algorithm in terms of the size of the input to the problem. You may want to argue that after a certain number $k$ of rounds, $r_k < 1$ and the algorithm must therefore have stopped.

**Solution:**

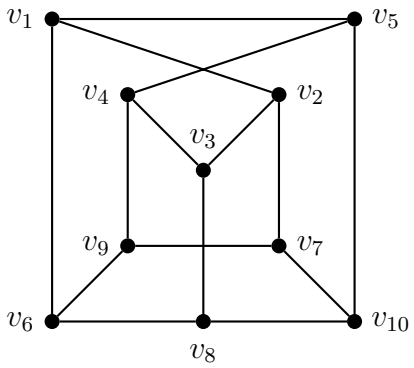
(a) If $r_n$ were to decrease by 1 in each round, the algorithm would run for $a$ rounds. Its running time would therefore be exponential in the size of the input!

(b) If $r_{n-1} = r_{n-2}$, then $r_n = 0 < r_{n-2}/2$. If $r_{n-2}/2 < r_{n-1} < r_{n-2}$, then $r_n = r_{n-2} \bmod r_{n-1} = r_{n-2} - r_{n-1} < r_{n-2}/2$. Finally, if $r_{n-1} \leq r_{n-2}/2$, then $r_n = r_{n-2} \bmod r_{n-1} < r_{n-1} \leq r_{n-2}/2$.

(c) We have shown that $r_n < r_{n-2}/2$, so $r_n < r_0/2^{\lfloor n/2 \rfloor} = a/2^{\lfloor n/2 \rfloor}$. Assume that the algorithm has not terminated after $k$ rounds. Then $r_k \geq 1$, so $2^{\lfloor k/2 \rfloor} < a$ and thus $\lfloor k/2 \rfloor < \log_2 a$. The algorithm therefore cannot run for more than $2\log_2 a + 2$ rounds. This is $O(\log_2 a)$, linear in the size of the input.

---

3. Consider the following graph.




(a) Apply breadth-first search to the graph, starting from $v_1$. Give the tree $T$ after each iteration of the algorithm.

(b) Apply depth-first search to the graph, starting from $v_1$. Give the tree $T$ after each iteration of the algorithm.

(c) Which of the algorithms can be used to find a spanning tree of the graph? Draw such a spanning tree.

(d) Which of the algorithms can be used to find a shortest $v_1-v_8$-path in the graph? Give such a path.

**Solution:**

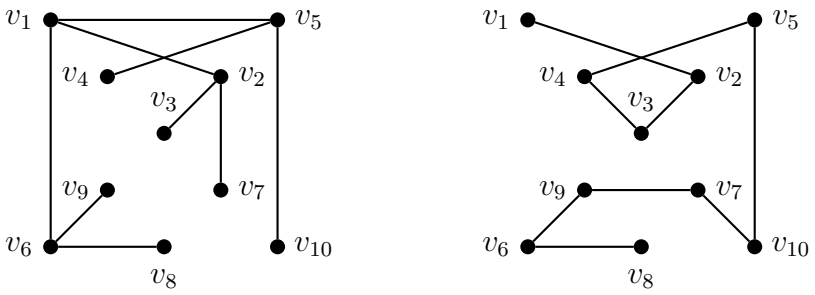
(a) When started from $v_1$, breadth-first search constructs a spanning tree $T$ of the connected component containing $v_1$. Initially $V(T) = \{v_1\}$ and $E(T) = \emptyset$. In each iteration the algorithm considers for inclusion any edge that has exactly one endpoint in $V(T)$, and among these edges chooses one that has been considered for a maximum number of past iterations. It may thus add edges in the order

$$v_1v_2, v_1v_5, v_1v_6, v_2v_3, v_2v_7, v_5v_4, v_5v_{10}, v_6v_8, v_6v_9.$$

(b) Depth-first search proceeds in a similar fashion as breadth-first search, but among the edges considered for inclusion chooses one that has been considered for a minimum number of past iterations. It may thus add edges in the order

$$v_1v_2, v_2v_3, v_3v_4, v_4v_5, v_5v_{10}, v_{10}v_7, v_7v_9, v_9v_6, v_6v_8.$$

(c) Both algorithms produce a spanning tree of the connected component containing the vertex they are started from, and thus a spanning tree of the whole graph when the graph is connected. For breadth-first search and depth-first search as described above, these spanning trees look as follows.




(d) The spanning tree $T$ constructed by breadth-first search has the special property that it contains a shortest $v-u$-path for every vertex $u$, where $v$ is the vertex the algorithm is started from. The path $v_1, v_6, v_8$ is thus a shortest $v_1-v_8$-path.

**Bonus question:** Depth-first search finds a longest path in the above example, but it does not do so in every graph. In fact, the problem of finding a longest path is NP-hard, so no efficient algorithm for it may exist. Can you give a graph where depth-first search does not find a longest path? Why does NP-hardness of finding a longest path not contradict the fact a longest path is easy to find in some graphs?