2nd tutorial

Jamie Fravel
jfravel@iuuk.mff.cuni.cz

Asymptotics

1. Comparing functions Determine the asymptotic relationships between each pair of functions f and g in the following table (i.e. whether $f \in \mathcal{O}(g)$, $f \in \Omega(g)$ or $f = \Theta(g)$).

Lets start with a few different definitions of $\mathcal{O}$, Ω and Θ.

Intuitively, $f \in \mathcal{O}(g)$ if f grows at *most* as fast as g; $f \in \Omega(g)$ if f grows at most as *least* as g and $f \in \Theta(g)$ if f grows *about* as fast as g.

Another intuition could be $f \in \mathcal{O}(g)$ if $g(n)$ never becomes irrelevantly small when compared to $f(n)$; $f \in \Omega(g)$ if $f(n)$ never becomes irrelevantly small when compared to $g(n)$; grows at *most* as fast as g and $f \in \theta(g)$ if f and g never become irrelevant compared to each other.

Formally, $\mathcal{O}$, Ω and Θ are defined:

- $f \in \mathcal{O}(g)$ if there exist constants $c, m > 0$ such that $f(n) \le cg(n)$ for all $n > m$.
- $f \in \Omega(g)$ if there exist constants $c, m > 0$ such that $f(n) \ge cg(n)$ for all $n > m$.
- $f \in \Theta(g)$ if $f \in \mathcal{O}(g)$ and $f \in \Omega(g)$. That is $\Theta = \mathcal{O} \cap \Omega$.

These are the definitions we expect you to know and use for this class. However, it is often useful to consider a slight alteration of these definitions:

- $f \in \mathcal{O}(g)$ if $\frac{f(n)}{g(n)}$ is bounded from above as n increases (as $n \to \infty$).
- $f \in \Omega(g)$ if $\frac{f(n)}{g(n)}$ remains strictly positive as n increases (as $n \to \infty$).
- $f \in \Theta(g)$ if $\frac{f(n)}{g(n)} \in (0, \infty)$ as n increases (as $n \to \infty$).

These are equivalent when you notice that c should be a real, finite, positive number.

	$f(n)$	$g(n)$	Ω	$\mathcal{O}$	Θ
(1)	$n^2 - 30n + 5$	$0.7n^2 - 20n + 15$	★	★	★
(2)	$100n^3 + 40n^2 - n$	$0.5n^4 - 1000n^3$	○	★	○
(3)	$5n^2 - n$	$30n + 4$	★	○	○
(4)	n	$\sqrt{n}$	★	○	○
(5)	$n^{\frac{3}{4}}$	$\sqrt{n}$	★	○	○
(6)	$\log_2 n$	$\ln n$	★	★	★
(7)	$n(\log_2 n)^5$	$n\sqrt{n}$	○	★	○
(8)	2^n	2^{2n}	○	★	○
(9)	e^n	2^n	★	○	○
(10)	$n!$	n^n	○	★	○
(11)	$n!$	2^n	★	○	○
(12)	2^n	2^{n+1}	★	★	★

Below are some simple proofs of the above results.

(1) You may know to drop all but the highest order terms when comparing ratios of polynomials. This comes from L'Hôspital's rule:

$$\lim_{n\to\infty} \frac{n^2 - 30n + 5}{0.7n^2 - 20n + 15} = \lim_{n\to\infty} \frac{2n - 30}{1.4n - 20} = \lim_{n\to\infty} \frac{2}{1.4} = \frac{10}{7} \in (0, \infty).$$

We can also use this constant in place of c in the formal definitions:

$$\begin{aligned}
\mathcal{O}: \quad & n^2 - 30n + 5 \overset{?}{\leq} \frac{10}{7}(0.7n^2 - 20n + 15) && \rightarrow \quad 7n^2 - 210n + 35 \overset{?}{\leq} 7n^2 - 200n + 150) \\
& && \rightarrow \quad 0 - 10n - 115 < 0 \qquad \forall\, n > 0. \\
\Omega: \quad & n^2 - 30n + 5 \overset{?}{\geq} \frac{1}{3}(0.7n^2 - 20n + 15) && \rightarrow \quad 3n^2 - 90n + 15 \overset{?}{\geq} 0.7n^2 - 20n + 15) \\
& && \rightarrow \quad 2.3n^2 - 70n - 0 > 0 \qquad \forall\, n > 31.
\end{aligned}$$

Thus f is in both $\mathcal{O}$ and Ω of g.

(2) In this case, we get a limit of 0, so we only have $\mathcal{O}$. We can also see that

$$\begin{aligned}
\Omega: \quad 100n^3 + 40n^2 - n \overset{?}{\geq} c(0.5n^4 - 1000n^3) \quad & \rightarrow \quad 100n^2 + 40n^1 - 1 \overset{?}{\geq} \frac{c}{2}n^3 - 1000cn^2 \\
& \rightarrow \quad \left(2000 + \frac{200}{c}\right)n^2 + \frac{80}{c}n - \frac{2}{c} \overset{?}{\geq} n^3
\end{aligned}$$

In general, we find that the polynomial with the higher degree dominates.

(3) f has higher degree, so we get Ω.

(4) With fractional degree $\frac{n}{\sqrt{n}} = \sqrt{n}$ is unbounded, but positive (for positive n). Thus Ω.

(5) Similar to above. The higher degree still dominates.

(6) Recall that $\log_2 n = \frac{\ln n}{\ln 2}$ so the ratio reduces to $\ln 2$, a positive constant. Θ.

(7) After some simple algebra, we can compare $\log_2 n$ and $n^{\frac{1}{10}}$. The derivative of $\log_2 n$ is $\frac{1}{n \ln 2}$, so L'Hôspital's rule gives a limit of zero.

(8) Consider the ratio: $\frac{2^n}{2^2 n} = \frac{1}{2^n}$. This is clearly not bounded below (by a positive constant).

(9) Notice that $\mathrm{e}^n = (2 + 0.71\ldots)^n = 2^n + 2^{n-1}0.71\ldots$ which is clearly unbounded relative to 2^n.

(10) The ratio

$$\frac{n!}{n^n} = \frac{n(n-1)(n-2)\ldots 1}{n^n} < \frac{n^{n-1}}{n^n} = \frac{1}{n}$$

is not bounded from below by a positive constant.

(11) If $n > 4$, then $n! = \prod_{i=1}^{n} i > \prod_{i=2}^{n} 2 = 2^n$. Thus $c = 1$ and $m = 4$ is sufficient to satisfy the definition of Ω.

(12) $\frac{2^n}{2^{n+1}} = \frac{2^n}{2 \cdot 2^n} = \frac{1}{2}$ is positive and bounded. Hence Θ.

2. Properties of asymptotic notation

Let $f(n)$ and $g(n)$ be asymptotically positive functions (which means that $\lim_{n\to\infty} f(n) > 0$ and $\lim_{n\to\infty} g(n) > 0$). Prove or disprove each of the following conjectures

(1) $f(n) + g(n) \in \Theta(\min(f(n), g(n)))$.

(2) $f(n) + g(n) \in \Theta(\max(f(n), g(n)))$.

(3) $f(n) \in \mathcal{O}((f(n))^2)$.

(4) $f(n) \in \Omega((f(n))^2)$.

(5) $f(n) \in \mathcal{O}(g(n))$ implies $g(n) \in \Omega(f(n))$.

If the tree (and hence tree edges) represent a single path from the root to every other vertex, which edges constitute a second path?

(1) **false** — consider $f(n) = n$ and $g(n) = n^2$. $n^2 \notin \mathcal{O}(n)$.

(2) **true** — $\max\big(f(n), g(n)\big) \leq f(n) + g(n) \leq 2\max\big(f(n), g(n)\big)$

(3) **true** — $f(n) \leq f^2(n)$ wherever $f(n) \geq 1$.

2nd tutorial (continued)

(4) **false** — consider $f(n) = n$.

(5) **true** — $f(n) \leq cg(n)$ implies that $g(n) \geq \frac{1}{c} f(n)$.

Interlude: Bridges. Jamie will use SYGA to demo an algorithm for finding "bridges" in undirected graphs. See page 122 of https://iuuk.mff.cuni.cz/ koutecky/pruvodce-en-wip.pdf for a reference. Note that translation of this document into English is still a work in progress.
This also serves as our introduction to the SYGA (See Your Graph Algorithm) visualizer.

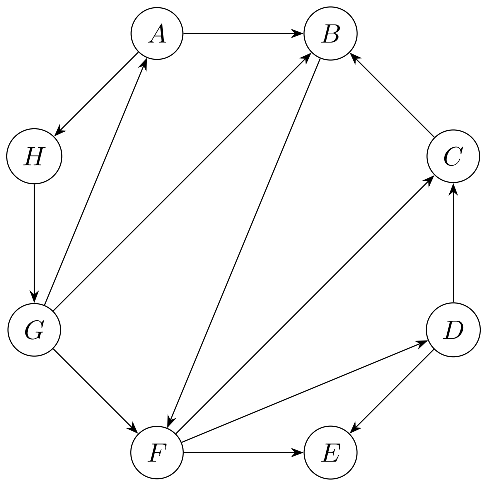
Depth First Search

3. DFS Runs — SYGA Starting at vertex A, perform depth-first search on the following graph in SYGA:

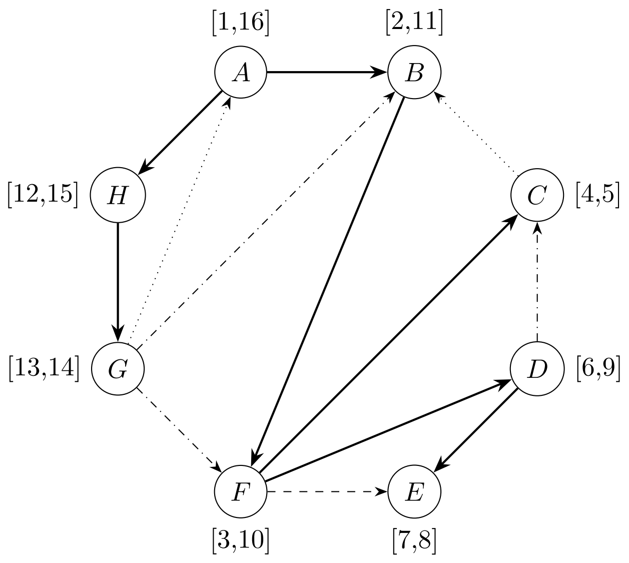
syga.space/exercises/week-2/practice-depth-first-search.

Whenever there's a choice of vertices, proceed in alphabetical order. First, find the tree edges, then classify the non-tree edges into back, forward and cross edges.

Finally you can compare your solution to one generated algorithmically. Feel free to change the graph (at the top of the script) or explore the algorithm.



The result can be seen in the following figures. Tree edges are thick, forward edges are dashed, back edges are dotted, and cross edges are dash dot. Each node is labeled with a pair of numbers, the first number is the "in" time (starting with 1), the second number is the "out" time.



4. Singly Connected Graphs — SYGA A directed graph G is *singly connected* if for every pair of vertices u and v there is at most one path from u to v in G. Follow this link to the SYGA exercise:

syga.space/exercise/week-2/singly-connected-graphs

Adjust the algorithm to determine whether or not a directed graph is singly connected. Your finished algorithm should set `flag = 1` whenever you find a counterexample.

There are three graphs at the top of the script for you to try your solution on. The first two are *not* singly connected.

Tree edges represent one path from the root to every other (reachable) node in the graph. What kinds of non-tree edges are there?

Both cross and forward edges imply additional paths to parts of the tree. Back edges imply cycles, but not paths. Simply set `flag = 1` when there is a cross or forward edge in DFS. In the second graph, DFS rooted at "A" will never reach "C", will miss a forward edge and erroneously return "true". To get a true result, the process must reach *every*. In particular, running from "C" gives the correct result of "false".

```
Algorithm SinglyConnectedTF($G$)
  for $v \in$ Vertices do
      Run DFS rooted at $v$
      if a cross or forward edge is found then
          return false
      end if
  end for
  return true
```

Complexity: $\Theta(n(n+m))$ where $n = |V|$ and $m = |E|$.

5. Longest Path In a DAG Given a directed, acyclic graph (DAG) $G = (V, E)$ and two of its vertices s and t, we want to compute the length of a longest path from s to t.

Proposition 1.1. *Running DFS on an acyclic graph will never produce a back edge.*

Back edges necessarily imply a cycle.

Our goal will be to compute, for every node $u \in V$, the value the length $\ell(u)$ of the longest path from s to u. The result is then $\ell(t)$.

Given a vertex u, let pred(u) represent the in-neighborhood (or predecessors) of u. We have then, that

$$\ell(u) = 1 + \max_{v \in \text{pred}(u)} \ell(v).$$

If pred$(u) = \emptyset$ (so that there is no edge from s to u), we will say $\ell(u) = -\infty$. In fact, we should initialize with $\ell(v) = -\infty$ for every vertex v.

From there, run DFS on G (rooted at s with $\ell(s) = 0$) and compute $\ell(v)$ in place as pred(v) is explored. By Proposition 1.1, there can be no back edges, and we only need to traverse the tree once. If we memoize the following, we get linear time in the number of vertices.

```
Algorithm LongestPathDAG($G, s, t$)
  if $s = t$ then
      return 0
  else
      $\ell \leftarrow -\infty$
      for $v \in$ pred$(t)$ do
          $k \leftarrow 1 +$ LongestPathDAG($G, s, v$)
          if $\ell < k$ then
              $\ell \leftarrow k$
          end if
      end for
  end if
  return $\ell$
```

Note that it is possible to reverse this concept—to recursively compute the longest paths from u to t—and get an equivalent algorithm.

6. Count Paths In a DAG Given a DAG $G = (V, E)$ and two of its vertices s and t, design an algorithm that calculates the number of distinct paths from s to t.

Our goal will be to compute, for every node $u \in V$, the number $c(u)$ of distinct paths from u to t.

Given a vertex u, let $\mathrm{succ}(u)$ represent the out-neighborhood (or successors) of u. We have then, that

$$c(u) = \sum_{v \in \mathrm{succ}(u)} c(v).$$

If $\mathrm{succ}(u) = \emptyset$ (and there is no path from u to t), then $c(u) = 0$.

For the moment, suppose that we have already run DFS rooted on s and found a topological ordering of the vertices. Then, traversing the graph in reverse-topological order would allow us to use the above formula to compute $c(u)$ for every node u very simply.

Ergo, we arrive at the following variant of DFS which, if properly memoized, runs in linear time in the number of vertices.

```
Algorithm PathCountDAG($G, s, t$)
  if $s = t$ then
      return 1
  else
      $c \leftarrow 0$
      for $v \in \mathrm{succ}(s)$ do
          $c \leftarrow c +$ PathCountDAG($G, v, t$)
      end for
  end if
  return $c$
```

Note that it is possible to reverse this concept—to recursively compute the number of paths from s to u—and get an equivalent algorithm.