TP 1: DAGs

General Guidelines

The purpose of the programming exercises you will find below is to (i) give you an opportunity to practice **implementing** some of the algorithms we have seen in class, improving your understanding, and to (ii) **experimentally verify** the theoretical complexity analysis about their running time. As a result, most exercises will have the following parts:

1. You will be reminded of a basic graph problem for which we have seen some algorithms and asked to program this algorithm (or some slight variation).
2. You will be given suggestions for the construction of test instances. These will consist of some small graphs which you can use to verify the correctness of your program. However, these will often be general constructions which can also give graphs of order n, sharing the same structure, for any desired n.
3. Using the above, you will be asked to perform some computational experiments, observing the running time of your program as a function of n. From this behaviour you will be asked to verify if the theoretical analysis was correct (or, more often, if your implementation is correct!). This part will be particularly interesting when you have implemented several algorithms for the same problem.

The intended audience for these exercises are students with some programming experience in Python or C. If you have never programmed in either language, you can either try to follow one of the numerous tutorials found online to start programming in either one, or use another language of your choice (but we may be less able to help you in this case). The exercise descriptions will often include language-specific advice, for C and Python, which may make them easier to solve.

Feel free to implement your algorithms in either Python or C. In fact, you are encouraged to try **both**. This will allow you to see for yourself that (i) Python programs are easier to write but (ii) C programs are significantly faster, and the low-level nature of C makes it easier to predict the running time of a program.

The spirit of the exercises below is that you should try to program them from scratch, using only basic data structures available in each language (e.g. lists or dictionaries in Python). There are of course numerous graph libraries available in both C and Python, but the objective here is to understand how these algorithms work, not to use them as a black box. You should, therefore, avoid `import`ing or `#include`ing non-standard packages.

1 Counting Paths on a DAG

The objectives of this programming exercise are the following:

- Implement the algorithm of TD4 which counts the number of $s \to t$ paths in a DAG.
- Experimentally verify that the assumption that arithmetic operations (such as addition) take time $O(1)$ is sometimes **false** and leads to an incorrect complexity analysis.

The reason that addition will not take $O(1)$ time in the context of this exercise is that we will be dealing with exponentially large values. It will therefore be much **easier** to solve this exercise in Python, where integers are unbounded (by default there are no overflows). However, one of the points of this exercise is that this can be **bad** in the sense that it makes non-trivial operations seem atomic. If you implement this exercise in C, things are more complicated, because the numbers we need to calculate will quickly overflow C-type integers. For this you have two choices: either only implement the modular counting part (in which case, all integers are bounded by 12345), or use the GMP library (more information below), which allows unbounded integers[1].

1.1 Counting the number of paths in a DAG

To begin this exercise we ask you to write a short function implementing the main part of the path-counting algorithm of TD4 (i.e. the part following the topological sorting).

More precisely, your function will take as input a topologically ordered DAG on n vertices, in adjacency list form. That is, your function's argument will be a list (or array) of lists. We will assume without loss of generality that the vertices of the DAG are numbered $0, \ldots, n-1$. Your function needs to calculate the number of distinct paths from 0 to $n-1$.

1.2 Testing

Construct two test cases:

1. A function that takes an argument n and returns the adjacency list of a transitive tournament on n vertices. A transitive tournament is a DAG on n vertices such that there is an arc between any pair of vertices and arcs are oriented towards higher-numbered vertices.
2. A function that takes an argument n and returns the adjacency list of a DAG constructed as follows: there are arcs from vertex 0 to vertices 1 and 2; there are arcs from vertices 1 and 2 to vertices 3 and 4; there are arcs from vertices 3 and 4 to vertices 4 and 5; and so on, until there are arcs from $n-3$ and $n-2$ to $n-1$. **Assume** n is always even for this construction.

Recall that we proved in TD4 that the number of paths from the source to the sink of a transitive tournament is 2^{n-2}. Use this fact to check that your function for counting paths is correct. Compute by hand the number of paths in the second family of DAGs and verify that your program computes the correct values.

1.3 Complexity Analysis

We now arrive at the interesting part of this exercise. By inspecting your code for the path-counting algorithm, you can probably infer that your program performs $O(n+m)$ arithmetic operations. Does this mean that the running time is $O(m+n)$?

In this course we typically assume that arithmetic operations take $O(1)$ time. This is, in fact, slightly inaccurate: the complexity of adding two integers should somehow depend on their sizes. For instance, using the elementary-school algorithm to compute the sum $x+y$ should take $O(\log x + \log y)$ steps (proportional to the number of digits of x, y). This slight inaccuracy is usually not important, as long as we are dealing with integers of moderate values.

[1] A third, more fun choice is to program large integer operations in C yourself!

In this exercise, however, as we saw, the integers we are dealing with have values exponential in n, therefore need $\Theta(n)$ digits to be stored. This is something that needs to be taken into account in the complexity analysis, if you program in Python; or something that will show up in the form of integer overflows, if you program in C (for more information about how to fix this, see below).

Once your program works correctly even for large values of n, do the following:

1. By executing your algorithm on transitive tournaments for different values of n, estimate its asymptotic running time. Recall that such graphs have $m = \Theta(n^2)$. Is your algorithm quadratic in n, as would be expected if arithmetic operations were $O(1)$ time?

2. Repeat the previous experiment, but for the second class of DAGs (which have $O(n)$ arcs). Is your algorithm linear in n, or worse?

1.4 Small numbers

In case you are not yet convinced that arithmetic operations on large integers are what is causing the slowdown in your program, consider the following variation of the problem. We are given a DAG G and an integer p and we want to compute the number of paths from 0 to $n-1$ **modulo** p.

Use the fact that $((a \bmod p) + (b \bmod p)) \bmod p = (a+b) \bmod p$ to modify your algorithm so that all values computed are between 0 and $p-1$. As an experiment, repeat the above tests, but this time calculate the number of paths modulo 12345. Did the behavior of your algorithm change?

Practical tips

Python uses large integers by default, so you don't have to worry about overflows in your calculations. However, after a certain threshold you may find that python refuses to print integers which are too long. To avoid this, your can `import sys` and then use the function `set_int_max_str_digits()` to define the maximum number of digits allowed in an integer to be converted to a string.

In order to measure the time taken by the execution of your program you can use the Unix command `time`. However, this has the drawback that you will measure the execution time of your whole program (including the generation of the test DAG). An alternative method is to `import time`, store the current time before the path-counting algorithm with `start = time.time()` and then compute the time of execution as `time.time()-start`.

C programs do not (by default) allow large integers, so a straightforward implementation will be incorrect starting from some moderate values of n. This is, however, not a problem for the modular counting part, where integer values are bounded. In order to also handle the general case, you are free to use the GMP library `https://gmplib.org/`. The basic functionality you need can be coded using the following:

- Add `#include <gmp.h>` to your program and compile with something like `gcc pathcount.c -lgmp` (i.e. link with the gmp library when compiling).
- Large integers have type `mpz_t`
- An `mpz_t` variable `x` can be initialized to a (small) integer value by `mpz_set_ui(x,17)` (to set `x` to 17)
- The function `mpz_add(x,y,z)` ensures that `x` becomes equal to `y+z` (where `x,y,z` not necessarily distinct).
- The call `mpz_get_str(NULL, 10, x)`, where `x` has type `mpz_t` returns a string representing integer `x` in base 10.

For more information, please refer to the libGMP documentation.

2 Topological Sorting

As you recall, a topological sorting of a DAG on n vertices, is a numbering of the vertices $0, \ldots, n-1$ so that all arcs point towards their higher endpoint. We have seen several algorithms for solving this problem (in class using DFS, and by updating indegrees in TD4). For this exercise we will ask you to experimentally compare these methods to the naive method which repeatedly finds a source in the DAG and removes it.

2.1 Algorithms

Program the following three algorithms, where in all cases the input is a directed graph (promised to be a DAG) in adjacency list form:

1. An algorithm which topologically sorts the vertices by repeatedly identifying a source and then marking it as removed from the graph.
2. A more sophisticated version of the previous algorithm, which updates the indegrees of all vertices each time a vertex is removed (seen in TD4).
3. A DFS-based version, as seen in class.

The output of all three functions should be an array `rank` which at position `i` contains the rank of vertex `i` in the topological sorting produced by your algorithm. Note that it is not mandatory that all three functions produce identical outputs, as a DAG may admit several distinct topological orderings.

Also program two helper functions:

1. A function which takes the ranks computed by the functions above and produces a sorted list of vertices respecting these ranks. (easy)
2. A verifier which checks if the outputs of the previous functions are correct, that is, if the `rank` array returned is a permutation of $\{0, \ldots, n-1\}$ and all arcs of the input graph point towards higher-ranked vertices.

2.2 Testing

Construct two test cases:

1. (Sparse DAG) A function that takes an argument n, which you may assume to be a multiple of 3, and produces a DAG with $n/3$ blocks of 3 vertices $B_0, \ldots, B_{n/3-1}$. The block B_i contains vertices $\{i, n/3+i, 2n/3+i\}$ and all vertices of B_i have arcs to all vertices of B_{i+1}, for all i.
2. (Dense DAG) A function that takes an argument n and produces a DAG with 4 blocks, A, B, C, D. A contains all vertices $i \in \{0, \ldots, n-1\}$ such that $i \bmod 7 \in \{1, 3\}$; B contains all vertices i such that $i \bmod 7 \in \{2, 5\}$; C contains vertices i such that $i \bmod 7\{0, 4\}$; and D the remaining vertices. We have arcs $A \to B \to C \to D$, that is, all vertices of A have arcs to all vertices of B, all vertices of B have arcs to all vertices of C, and so on.

2.3 Complexity Analysis

Execute your algorithms on several test instances for small values of n, to ensure that your output is correct. Also, verify your output with your checker function.

Once you are confident that your algorithms are correct, experimentally test their performance. Explain the behavior your observe. Would the naive algorithm be different if at every step we remove a sink instead of a source?

Practical tips

For the linear-time algorithms, you are likely to need to go beyond $n = 100000$ before the running time can be accurately measured. If you implement DFS with a recursive call, as we did in class, this is likely to **crash** in both C and Python, because the depth of recursive calls will be too high (stack overflow). You will get messages like `RecursionError: maximum recursion depth exceeded in comparison` or `Segmentation fault (core dumped)`.

In order to fix this you have to tell your system to allow a larger stack. On Linux systems this can be done by the terminal command `ulimit -s unlimited`. Executing this command before your program should be sufficient, if you program in C. If you program in Python, you will also need to tell the Python interpreter that you wish to allow a larger recursion depth. For this you need to `import sys` and use the command `sys.setrecursionlimit(10000000)` (or some other suitable large number).