PW3 - Dynamic Programming

Experimental comparison of the complexities of two algorithms solving the same problem

Objective

Measure and compare the execution times of a combinatorial optimization algorithm in its naive recursive version, and its dynamic programming version. The problem solved by the algorithm has been (or should have been) studied in tutorial classes: it consists of finding an optimal path in the assembly lines of a car factory. An additional objective is to discover some basic tools used in computer science experimentations for: measuring time, drawing curves, producing a report, etc.

Work to be done

The program in both versions (recursive and iterative) is already implemented. It is provided to you as source files so you can directly proceed with the time measurements. You have to perform the following tasks.

1. Download from Moodle the resource files for this PW: `srcPW3-ProgDyn.zip`
2. Compile the source code of the program. The file to compile is named `factory.c`. You can simply compile it by typing `'gcc -o factory factory.c'`. Usage of the program is as follows: `factory seed (d or r) factory-length`. Choose your own seed, for example a number between 1 and 18, the value in itself is of no importance. A given seed only guarantees that the same random numbers will be generated at each execution. To call the program for example with seed 9, with 20 stations per assembly line and in recursive version, type `factory 9 r 20`. For the dynamic programming version type `factory 9 d 20`.
3. Instrument the program with instructions for measuring the execution times of selected portions of code. You can for example compare the current dates in entry and exit of the chosen code portion. To this end, you can call the `gettimeofday()` command and use the `timeval` structure, after having included the `sys/time.h` header file. See the manual of that command by typing `man gettimeofday` to learn how it works.
4. Measure the successive execution times of both versions of the program with a number of stations growing from 2 to 40. You will rapidly see that measuring the actual execution times of the recursive version with 30 or more stations becomes very long, so you will have to *imagine* what these times would be: compare how time has previously grown at each station added, and extrapolate the result.
5. Write the obtained results in a text column file with in first column the factory size (number of stations), in second column the solution returned by the program, in third column the execution time of the iterative version, and in fourth column the execution time of the recursive version.
6. By observing how time grows in the recursive version each time a station is added, you have been able (see point 4 above) to estimate the times that were not measured explicitly because they were too long to get. See how it fits with the complexity computed in tutorial classes. N.B. If that complexity has not been studied, you still can guess what it is by giving the function that characterizes how time grows with respect to the number n of stations in the factory.
7. Draw the performance curves with `gnuplot`. This tool can either be used in command line or in a script. An example of a `gnuplot` script can be found in the file `factory.plot` (the comments are in french). The aim is to draw curves that show how time grows with the number of stations. Please complete the caption and produce the file `perfFactory.pdf`, for further inclusion in the LaTeX document `factory.tex`.
8. Redraw the curves after adding as a fifth column in `factory.txt` the binary logarithm of the fourth column. This way you can confirm experimentally the recursive's solution complexity computed in tutorial classes (or guessed at point 6 above).