Scheduling Project

André Rossi

December 2021

This project must be completed by a team of two students, and released by email to `andre.rossi@dauphine.psl.eu` *by Monday February 7, 2022 at 13:00.*

1 Minimizing the weighted number of late tasks, $1||\sum w_i u_i$

1.1 Introduction

The purpose of this exercise is to implement the Fully Polynomial Time Approximation Scheme proposed in the scheduling class for problem $1||\sum w_i u_i$. The implementation should be done in C language. The `gcc` compiler called with the `-Wall` and `-pedantic` flags should not return any error nor warning.

Six problem instance files can be found in `instances.tar.gz`. These text files are all built under the same model as `i6.dat`, the instance used in class (see Figure 1). The first line contains n, the number of tasks (here, $n = 6$). The next six lines contain three integers: p_i, w_i, d_i that describe task T_i. For example, `4  2  5` describes the first task, and indicates that $p_1 = 4$, $w_1 = 2$ and $d_1 = 5$. The last line indicates that $p_6 = 5$, $w_6 = 2$ and $d_6 = 17$. Note that the tasks are already sorted by increasing due dates in all these files.

```
6
4 2 5
2 6 5
5 4 14
5 3 14
6 3 14
5 2 17
```

Figure 1: The instance file `i6.dat`

1.2 Dynamic programming algorithm

The program implementing the dynamic programming algorithm will be called from a terminal with one argument, which is an instance file (like `i6.dat`). It is recommended (but not mandatory) to define a structure `TASK` having the following members for storing the data related to a task:

- An unsigned integer `id` whose value will be in the set $\{1, \ldots, n\}$, for storing the index of the task (T_1 is the first task, its `id` is one)
- An unsigned integer `p` for storing the processing time of the task
- An unsigned long long `w` for storing the weight of the task
- An unsigned integer `d` for storing the due date of the task

We assume that `task` is a pointer to a `TASK` structure, that will be used to store the input data related to the `n` tasks. `n`, `task` and `w_max` (the maximum weight) are initialized by calling the function `load_data` that reads the instance file passed as an argument to the program. This function should read `n` from the instance file, perform a memory allocation for `task`, populate the array `task` and compute `w_max`.

Question 1: Write the C function `void load_data(char filename[], unsigned int *n, TASK **task, unsigned long long *w_max)`, where `filename` will contain the name of the instance file passed as an argument to the program.

Question 2: Write the C function `void display_data(unsigned int n, TASK *task, unsigned long long w_max)` for displaying all the input data (it may be useful for debugging purposes).

Question 3: Write the C function `unsigned int Moore(unsigned int n, TASK *task)` that returns the number of on-time tasks by applying the Moore-Hodgson algorithm.

Question 4: Write the C function `unsigned long long compute_WI(unsigned int n, TASK *task)` for computing W_I, an upper bound on the total profit (this function will call the `Moore` function).

Question 5: Write the C function `void populate_table(unsigned int n, unsigned long long WI, TASK *task, unsigned int ***P)` for computing $P_i(w)$ for all $i \in \{1, \ldots, n\}$ and for all $w \in \{0, \ldots, W_I\}$. The table is a two-dimensional array of unsigned integers `P` that is passed by address to this function. The memory allocation for `P` is also dealt with in function `populate_table`.

Question 6: Write the C function `void find_sol(unsigned int n, unsigned long long WI, TASK *task, unsigned int **P)` for displaying the solution on the screen, as well as the total profit.

Question 7: Write the C function `main` for solving problem $1||\sum w_i u_i$ by using the dynamic programming approach presented on slide 61 of the scheduling booklet.

1.3 Fully Polynomial Time Approximation Scheme

The proposed FPTAS is to use the same dynamic programming algorithm as for the original problem, but by replacing the original w_i values by $w'_i = \left\lceil \frac{w_i}{k} \right\rceil$ for all $i \in \{1, \ldots, n\}$, with $k = \frac{\varepsilon w_{max}}{n}$.

Theoretical part

Question 8: What is the range of k if the number of entries in the table of the dynamic programming algorithm should be less in the approximate problem than in the original problem?

Question 9: What is the range of ε if the approximate problem shall return a solution with an actual performance guarantee?

Question 10: Compute these ranges for k and ε in the case of instance file `i6.dat`, and conclude about the use of the FPTAS for that instance.

Implementation part

Question 11: Update the C program of the dynamic programming algorithm in order to produce an implementation of the FPTAS. In particular, this program is now taking two arguments: an instance file, and the value for k (as a `double`). The program should display ε, the number of elements to compute in the table of the dynamic programming algorithm, the solution, and its objective value. It should also display the total amount of time spent solving the approximate problem (see Section 3).

Question 12: Run the FPTAS on all the instance files in `instances.tar.gz` for different values of k. Analyze the instances and the results of FPTAS, and conclude on the kind of instances that can take advantage of FPTAS.

2 Addressing the bi-objective scheduling problem $1||\sum c_i, L_{max}$

This part is to implement the construction of all Pareto-optimal solutions of the problem $1||\sum c_i, L_{max}$. The instance file format is the same as in the first exercise, but task T_i is characterized by two integers only, p_i and d_i. The data structures of the previous exercise may be reused, and the program should be tested on many instances, including the one solved in class to illustrate the solution process of $1||\sum c_i, L_{max}$.

Question 13: Write a C program that solves the problem $1||\sum c_i, L_{max}$.

3 Appendix

3.1 Measuring the running time of a C program

The following self-explanatory piece of code can be used to measure the CPU time required by your program.

```
#include<stdio.h>
#include<sys/timeb.h>

int main(int argc, char *argv[])
{
struct timeb t0, t1; /* Require sys/timeb.h */
float cpu_time;      /* Running time in seconds */
ftime(&t0);          /* Start of the timing */
/* Place your code here */
ftime(&t1);          /* End of the timing */
cpu_time = (float)(t1.time - t0.time) + (float)(t1.millitm-t0.millitm)/1000.0;
printf("\nCPU time: %.3f s.\n", cpu_time);
return 0;
}
```

The compiled program may run faster if the flag `-O3` of `gcc` is used.

3.2 Accessing the largest unsigned integer value

The largest value that the type `unsigned int` can store is the constant `UINT_MAX`. The library `limits.h` is required to use this constant, as in:

```
#include<stdio.h>
#include<limits.h>

int main(int argc, char *argv[])
{
/* The largest unsigned integer can be referred to as UINT_MAX */
printf("UINT_MAX = %u\n", UINT_MAX);
return 0;
}
```

3.3 Debugging memory leaks

You can install and use `valgrind`, in order to make sure that your program is free of memory leaks, or to track bugs related to memory management that may cause your program to crash. `valgrind` can be installed by invoking `sudo apt install valgrind`, then you should compile your C program with `gcc -Wall -g myprog.c -o myprog`. Finally, you can call `valgrind` to analyze your program with `valgrind --leak-check=yes ./myprog arg1 arg2`

More information on `valgrind` is available at:
`http://www.valgrind.org/docs/manual/quick-start.html`