HW 6

Willis Allstead

CS477-1001 – Professor Monica Nicolescu

11/21/17

**1.**

As is given in the question, each job has a maximum waiting time of *pi* on the supercomputer, which means that we should focus on minimizing the amount of time *fi* needed on each PC for the all the jobs to finish, which is called the *completion time*. With a greedy technique in solving this problem we should consider tackling the longest job, in terms of *fi*, first. This should result in a minimized total *completion time*. To focus on the longest job first, we need to sort the jobs in order of *fi* for each job, with largest *fi* first.

Example of Algorithm (pseudocode using Swift syntax):

/\* function takes and returns array of jobs \*/

func indexRecompilation(searchIndex: [Job]) -> [Job] {

/\* sortJobs performs algorithm like RANDOMIZED-QUICKSORT with all

elements of array, sorting based on given data member fi of each job. \*/

let sortedJobs = sortJobs(searchIndex)

/\* because we use RANDOMIZED-QUICKSORT, the expected time requirement

of sortJobs is O(nlogn). \*/

for (job in sortedJobs) { // for each job in sortedJobs

superComputer(job) {

/\* on completion of preprocess of job on super computer \*/

somePC(job) // finish job on pc

}

}

}

The indexRecompilation algorithm is greedy because of the way only the fi of each job is considered. This is using the concept of choosing the optimal job based on how it looks locally. To prove that this greedy strategy yields the optimal solution, we have to look at how it utilizes the greedy-choice property. To do this, we need to talk about the substructure of the algorithm used. If the algorithm used to order the following jobs after its predecessor finishes ordering is equally efficient, there is no structural problem using this algorithm. In use, this algorithm can be proven to be the optimal greedy solution because if every job was compared to the job after it, and that comparison was used to order the jobs from longest to shortest *completion time*, and that result was used to order that pair and move on, repeating until the optimal solution was found, there is no way it cannot find the optimal greedy solution if those comparisons don’t affect the time complexity of the algorithm. And as long as the algorithm uses some sorting algorithm to sort the jobs that is O(nlogn), then the final time complexity would be O(nlogn + n) or simply O(nlogn).

**2. a.** “There exists a valid schedule if and only if each stream *i* satisfies *bi* ≤ *rti*” :

As long as the constraint “*For each natural number t > 0, the total number of bits you send over the time interval from 0 to t cannot exceed rt*” is met, the schedule is valid. This constraint says nothing of the time interval requirements of a specific stream, only the overarching schedule. Therefore, *bi* could be greater than *rti* as long as the average of *bi* at most equals the total amount of time \* r.

To prove this, an example set of streams can be used. If only two are used, let’s say the first two of the example, (*b1*, *t1*) = (2000, 1)and (*b2*, *t2*) = (6000, 1) with an *r* = 5000. So the first stream would satisfy *bi* ≤ *rti*, but the second would not since 6000 ≤ 5000\*1. But the total bits would be 2000+6000 = 8000 and the total time would be 1+1 = 2 so taking those we need to check if 8000 ≤ 5000\*2, which it does so we see that this has created a valid schedule.

**b.** “Give an algorithm that takes a set of *n* streams, each specified by its number of bits *bi* and its time duration *ti*, as well as the link parameter *r*, and determines whether there exists a valid schedule” :

The algorithm will use the fact that the substructure will need to be able to schedule all *n* streams such that all *n*-1 streams correctly follow the fact that the total bits is less than or equal to the constant r times the total time of the streams. First we will order the streams by *bi*/*ti* in order to find the lowest bit rate needed. Then we will send those streams in that order.

[Algorithm is on following page]

Example of Algorithm (pseudocode using Swift syntax):

/\* function takes and returns array of jobs \*/

func calcSchedule(streams: [Stream]) -> [Stream] {

var bitRates: [Float] // to store bitrates of each stream

var bitRateTotal: Float

var sortedStreams: [Stream]

for (Int i = 0; i++; i < streams.count) { // for each stream

let thisStream = streams[i]

bitRates[i] = thisStream.bitRate/ thisStream.time

sortedStreams = sortStreams(streams, bitRates, bitRateTotal)

/\* sortStreams would fail if bitRateTotal was ever set to be

greater than r. Because we use RANDOMIZED-QUICKSORT, the expected

time requirement of sortStreams is O(nlogn). \*/

superComputer(job) {

/\* on completion of preprocess of job on super computer \*/

somePC(job) // finish job on pc

}

}

return sortedStreams // returns sorted streams that make up schedule.

}