Data Structures (2028C) **Lab 9**

Fall 2020Topics covered***: Heaps***

(2 weeks: 11/9-20/20)

**Objective:**

The objective of this Lab is to explore creating scheduling applications using heaps and heap sort

BACKGROUND

Scheduling of Machines [Sahni, *Data Structures, Algorithms, and Applications in C++*, Ch. 6 & 9] –

Consider a factory that has *m* identical machines. We have *n* jobs that need to be processed. The processing time required by job *i* is *ti.* This time includes the time needed to set up and remove the job on/from a machine. A schedule is an assignment of jobs to time intervals on machines such that a) no machine processes more than one job at any time, b) no job is processed by more than one machine at any time, and c) each job *i* is assigned for a total *ti* units of processing.

Each machine is assumed to be available at time 0. The **finish time** or length of a schedule is the time at which all jobs have completed. In a non-preemptive schedule, each job *i* is processed by a single machine from some start time *si* to its completion time *si*+ *ti.*. We shall concern ourselves only with non-preemptive schedules.

If the given task is to write a program that constructs a **minimum**-finish-time m-machines schedule for a given set of n jobs, this task would be very hard. In fact, no one has ever developed a polynomial time algorithm to construct a minimum-finish-time schedule:

This problem is a member of the infamous class of NP-hard (**NP** stands for **non-deterministic polynomial**) problems. The NP-hard and NP-complete problem classes contain problems for which no one has developed a polynomial-time algorithm. The problem in the class NP-complete are decision problems. That is, for each problem instance, the answer is either yes or no.

Our machine-scheduling problem is not a decision problem as the answer, for instance, it is an assignment of jobs to machines such that the finish time is minimum. We may formulate a related Machine-Scheduling problem in which, in addition to the tasks and machine, we are given a *TMin* and are asked to determine whether or not there is a schedule with finish time *TMin* or less. For this related problem, the answer to each instance is either yes or no. This related problem is a decision problem that is NP-complete. NP-hard problems may or may not be decision problems.

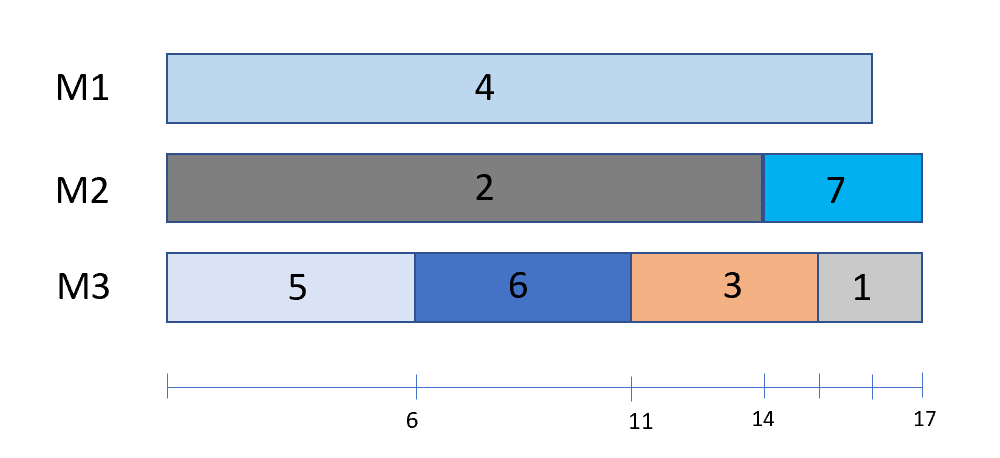
Thousands of problems of practical interest are NP-hard or NP-complete. If anyone discovers a polynomial-time algorithm for an NP-hard or NP-complete problem, then heshe would have simultaneously discovered a way to solve all NP-complete problems in polynomial time.

Although we are unable to prove that NP-complete problems cannot be solved in polynomial time, common wisdom very strongly suggests that this is the case. As a result, optimization problems that are NP-hard are often solved by **approximation algorithms**. Although approximation algorithms do not guarantee to obtain optimal solutions, they guarantee solutions “close” to optimal.

In case of our scheduling problem, we can generate schedules whose lengths are at most 4/3 – 1/(3m) of optimal by employing a simple scheduling strategy called **longest processing time** **first** (LPT). In LPT, jobs are assigned to machines in descending order or their processing time requirements *ti*. When a job is being assigned to a machine, it is assigned to the machine that becomes idle first. Ties are broken arbitrarily.

To illustrate, suppose that there are three machines ( *m* = 3) for seven jobs (n = 7) with processing requirements in time units (2, 14, 4, 16, 6, 5, 3). The machines are labeled M1, M2, and M3.

We can construct an LPT schedule by first sorting the jobs into descending order of processing times. The job order is (4, 2, 5, 6, 3, 7, 1). At time 0, all three machines are available, job 4 may be assigned to any machine. Suppose we assign it to M1. Now M1 is unavailable until time 16. Job 2 is next assigned; we can assign it to either machine 2 or 3, as both become available at the same time (i.e., time 0). Assume we assign job 2 to M2, now machine 2 becomes unavailable until time 14). Next, we assign job 5 to M3 from time 0 to 6. Job 6 is assigned next. The first available machine is M3. It becomes available at time 6. Following the assignment of job 6, from time 6 to 11 on this machine, the availability of M3 becomes 11. Job 3 is next considered for scheduling. The first machine to become available is machine 3 at time 11, we assign job 3 to this machine. At time 14, M2 becomes available, and the next job is 7, we assign job 7 to M2, from time 14 to 17, and at time 15, M3 is available, then job 1 is assigned to M3. The list of jobs is now empty.



To construct LPT schedule, the jobs are sorted in ascending order, as a min heap, and jobs are assigned to available machines in the reverse of this order. To determine which machine a job is to be assigned to, we need to determine which machine becomes available first. To make this determination, we maintain a min heap of the *m* machines. Each element on this min heap is of type MachineNode (see below) with ID - the machine identifier and avail the time at which the machine becomes available. A DeleteMin() is used to extract the machine that becomes available first. The machine’s availability time is increased, and it is inserted back into the min heap. This min heap is initialized by inserting a node for each machine. Since all machines are initially available at time 0, the ‘avail’ value for each of these machines is 0.

https://github.com/AlejandroPena813/LPT-ArrMinHeap-LeftistTree-Pointers-/blob/master/LPT.hv

class JobNode {

friend void LPT(JobNode \*, in int);

friend void main(void);

public:

operator int () const {return time;}

private:

int ID, //Job identifier

time;

};

Im creating a template class T for MinHeap…

class MachineNode {

friend void LPT(JobNode \*, int, int);

public:

operator int () const {return avail;}

private:

intLove 9 ID, // machine identifier

avail; // time that machine becomes available

}

template <class T>

void LPT(T a[], int n, int m) { // construct an m machine LPT schedule

if (n <= m) { cout << “Schedule one job per machine.” << endl; return; }

HeapSort(a, n); // sort the jobs into ascending order – a min heap

MinHeap<MachineNode> H(m); // initialize m machines and the min heap

MachineNode x;

For (int i=1; i<=m; i++) {

x.avail=0; x.ID = I;’ H.Insert(x);

}

// construct schedule

For (int I = n; I >= 1; i--) {

H.DeleteMin(x); // get the first free machine

Cout << “Schedule Job “ << a[i].ID << “ on machine “ << x.ID

<< “ from “ << x.avail << “ to “ << (x.avail + a[i].time) << endl;

x.avail += a[i].time; // new avail time

H.insert(x);

}

};

**Task 1:** ~~Design the necessary code for creating a max (min) heap, and implement the basic operations for inserting a node, and deleting Max (Min) node from the heap.~~

**~~Task 2:~~** ~~Using the LPT schedule approach, construct the schedule with the sample data provided above.~~

**~~Task 3:~~** ~~Generate another example of machine scheduling, with n = 10, and m = 5, using a different testing sample data set, created by using random number generator~~

**~~Lab Submission:~~**

1. ~~Write a lab report including the following information:~~ 
   1. ~~A description of the objectives/concepts explored in this assignment including why you think they are important to this course and a career in CS and/or Engineering.~~
   2. ~~The sections from each task indicated to be included in the lab report.~~
2. ~~Include all source code from all tasks, input and output files (if any), and any special instructions to compile and run those programs.~~

**~~Lab Grading:~~**

1. ~~20% - Lab attendance~~
2. ~~30% - Task 1 has been correctly implemented and meets all requirements.~~
3. ~~30% - Task 2 has been correctly implemented and meets all requirements.~~
4. ~~20% - Task 3 has been correctly implemented and meets all requirements.~~

~~If program fails to compile, 0% will be given for that Task.~~

~~LPT FUNCTION~~

~~I think the LPT function is actually printing out whats happening. So, that should be enough. You are using the same example anyway.~~

~~MIN & MAX HEAPS~~

~~[Yesterday 21:13] Lolla, Madhav (lollasv)~~

~~Implement both min and max heap~~

~~(1 liked)~~

~~​~~

~~[Yesterday 21:13] Lolla, Madhav (lollasv)~~

~~In a way, much of what goes on inside min and max heap's are same~~

​

[Yesterday 21:14] Lolla, Madhav (lollasv)

its just comparisons being made that differ

[Yesterday 21:16] Lolla, Madhav (lollasv)

Max heap heapsort basically outputs an ascending order of the list "in-place".

​

[Yesterday 21:16] Lolla, Madhav (lollasv)

If both are opposite of each other, its not unimaginable using them interchangeably, I donno if that makes sense

Min heap does a heapsort "in-place" in a decreasing order (non-increasing order, technically)

[Yesterday 21:21] Lolla, Madhav (lollasv)

It's not that they have to be numbered or anything. But there has to be one attribute for every node, that can be used as a number so that heaps can be rearranged (min or max)

(1 liked)

​

[Yesterday 21:21] Lolla, Madhav (lollasv)

Thats why you make a Heap of MachineNode here in this assignment

(1 liked)

MACHINE NODE

[Yesterday 21:24] Lolla, Madhav (lollasv)

On track, basically the machine that frees up first should be on the top of the machine heap...

​

[Yesterday 21:25] Lolla, Madhav (lollasv)

Well, the implementation is written down for LPT anyway. So all focus can be on implementing the heaps accurately

[Yesterday 21:31] Lolla, Madhav (lollasv)

Not exactly, initially time-to-free-up is 0 for all machines. Both heaps are still independent. But what you will do is heapsort the process heap so that you get an array, and assign longest process to the machine at root node (this assigning is basically adding up the avail time attribute). Since machine heap is also a minheap, you will then rearrange and get the next free machine to the top.

​

[Yesterday 21:31] Lolla, Madhav (lollasv)

Somewhere along you are also counting the total processtime (which is being done in the program already)

template <class T>

class **MinHeap**

{

private:

T data[100];

int capacity, currentSize;

void **bubbleUp**()

{

int child = currentSize - 1;

int parent = getParent(child);

while(data[child]<data[parent] && child >=0 && parent >=0)

{

swap(child, parent);

child = parent;

parent = getParent(child);

}

}

void **bubbleDown**()

{

int parent = 0;

while(1)

{

int left = getLeftChild(parent);

int right = getRightChild(parent);

int largest = parent;

if(left<currentSize && data[left]<data[largest])

{

largest = left;

}

if(right <currentSize && data[right]<data[largest])

{

largest = right;

}

if(largest!=parent)

{

swap(largest, parent);

parent = largest;

}

else

break;

}

}

void **swap**(int child, int parent)

{

T temp;

temp = data[child];

data[child]=data[parent];

data[parent]=temp;

}

int **getLeftChild**(int parent){return 2\*parent+1;}

int **getRightChild**(int parent){return 2\*parent+2;}

int **getParent**(int child)

{

if(child%2==0)

{

return(child/2)-1;

}

return child/2;

}

public:

**MinHeap**(int c)

{

capacity = c;

currentSize = 0;

}

**MinHeap**()

{

capacity = 0 ;

currentSize=0;

}

void **Insert**(T a)

{

currentSize++;

data[currentSize-1]=a;

bubbleUp();

}

T **DeleteMin**()

{

int child = currentSize - 1;

swap(child,0);

T value = data[child];

currentSize--;

bubbleDown();

return value;

}

int **getSize**()

{

return currentSize;

}

};