Tutorial 10: Heaps

Algorithms - I

By Kanishk Singh

Table of Contents

Problem Statement

- Consider a refrigerator servicing facility that can handle just one refrigerator at a time. Suppose that on a particular day there are n-refrigerators awaiting repair. Arrival of refrigerator-i is specified by a start time s_i, and a processing time p_i. You are allowed to suspend the repairing of refrigerator and resume it later. We will refer to each refrigerator servicing as a job.

Problem Statement

- Consider a refrigerator servicing facility that can handle just one refrigerator at a time. Suppose that on a particular day there are n-refrigerators awaiting repair. Arrival of refrigerator-i is specified by a start time s_i, and a processing time p_i. You are allowed to suspend the repairing of refrigerator and resume it later. We will refer to each refrigerator servicing as a job.
- A schedule specifies for each unit time interval, the unique job that is run during that time interval. In a feasible schedule, every job J_i has to be run for exactly p_i time units ($jobLength$) after time s_i ($startTime$).

Problem Statement

- Consider a refrigerator servicing facility that can handle just one refrigerator at a time. Suppose that on a particular day there are n-refrigerators awaiting repair. Arrival of refrigerator-i is specified by a start time s_i, and a processing time p_i. You are allowed to suspend the repairing of refrigerator and resume it later. We will refer to each refrigerator servicing as a job.
- A schedule specifies for each unit time interval, the unique job that is run during that time interval. In a feasible schedule, every job J_i has to be run for exactly p_i time units ($jobLength$) after time s_i ($startTime$).
- The completion time C_i for job J_i is the earliest time when J_i has been run for p_i time units. You are asked to implement a scheduler that minimizes :

$$\sum_{j=1}^{n} C_j$$

Understand the plot

We want to schedule in an online fashion and it would look as follows :
$t = 0$
while there are jobs left not completely scheduled

- Among those jobs J_i such that $s_i \leq t$.
- pick a job J_m to schedule at time t according to some rule.
- increment t.

A Working Example

- You are given the number of jobs and *jobId*, *startTime* and *jobLength* values for each job.
- Number of jobs :$n = 4$
- The input format is given below :

jobId	startTime	jobLength
1	0	21
2	1	3
3	2	6
4	3	2

A Working Example

- You are given the number of jobs and *jobId*, *startTime* and *jobLength* values for each job.
- Number of jobs :$n = 4$
- The input format is given below :

jobId	startTime	jobLength
1	0	21
2	1	3
3	2	6
4	3	2

- Jobs scheduled at each timestep are:
- 1 2 2 2 4 4 3 3 3 3 3 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Continued...

- You are also asked to print the average turnaround time.

Continued...

- You are also asked to print the average turnaround time.
- Let's denote the *startTime* value for a job by s_i and the time when it first starts running on the processor by k_i. The average turnaround time is given by :

$$\frac{\sum_{i=1}^{n}(k_i - s_i)}{n}$$

Continued...

- You are also asked to print the average turnaround time.
- Let's denote the *startTime* value for a job by s_i and the time when it first starts running on the processor by k_i. The average turnaround time is given by :

$$\frac{\sum_{i=1}^{n}(k_i - s_i)}{n}$$

- The individual turnaround times for the jobs are 0, 0, 4 and 1 respectively.
  Hence Average Turnoaround Time is : 1.25

Table of Contents

Can you think of an approach?

Have $2 - 3$ minutes to think.

Greedy Approach

- Let $y_{i,t}$ be the total time that job J_i has been run before time t.

Greedy Approach

- Let $y_{i,t}$ be the total time that job J_i has been run before time t.
- Pick J_m to be a job that has minimal remaining processing time, that it, that has minimal $p_i - y_{i,t}$ (remLength).

Greedy Approach

- Let $y_{i,t}$ be the total time that job J_i has been run before time t.
- Pick J_m to be a job that has minimal remaining processing time, that it, that has minimal $p_i - y_{i,t}$ (remLength).
- Ties may be broken based on unique jobId, so that lower jobId is preferred.

Greedy Approach

- An efficient implementation of this greedy algorithm can be achieved by the scheduler maintaining a priority queue of currently remaining jobs.

Greedy Approach

- An efficient implementation of this greedy algorithm can be achieved by the scheduler maintaining a priority queue of currently remaining jobs.
- When a job is executed on the processor, its remLength value is decremented by 1 after each timestep.

Greedy Approach

- An efficient implementation of this greedy algorithm can be achieved by the scheduler maintaining a priority queue of currently remaining jobs.
- When a job is executed on the processor, its remLength value is decremented by 1 after each timestep.
- When the currently executing job finishes (remLength value 0), the scheduler removes the job with the least remLength value from the queue and schedules it to run on the processor.

Greedy Approach

- An efficient implementation of this greedy algorithm can be achieved by the scheduler maintaining a priority queue of currently remaining jobs.
- When a job is executed on the processor, its remLength value is decremented by 1 after each timestep.
- When the currently executing job finishes (remLength value 0), the scheduler removes the job with the least remLength value from the queue and schedules it to run on the processor.
    - This job now becomes the currently executing job.

Greedy Approach

- When a new job x arrives, its job duration is checked with the remLength value of the currently executing job y. Two cases are possible:

Greedy Approach

- When a new job x arrives, its job duration is checked with the remLength value of the currently executing job y. Two cases are possible:
  - If the job duration of x is less than the remLength value of y, y (with its current remLength value) is added to the scheduling queue and x becomes the currently executing job.

Greedy Approach

- When a new job x arrives, its job duration is checked with the remLength value of the currently executing job y. Two cases are possible:
  - If the job duration of x is less than the remLength value of y, y (with its current remLength value) is added to the scheduling queue and x becomes the currently executing job.
  - Otherwise, x is added to the scheduling queue.

Greedy Approach

- When a new job x arrives, its job duration is checked with the remLength value of the currently executing job y. Two cases are possible:
  - If the job duration of x is less than the remLength value of y, y (with its current remLength value) is added to the scheduling queue and x becomes the currently executing job.
  - Otherwise, x is added to the scheduling queue.
- If there are two jobs with the same remLength value, the one with the lower job id is to be chosen.

Let's Rework The Example

jobId	startTime	jobLength
1	0	21
2	1	3
3	2	6
4	3	2

Continued...

Table of Contents

Scheduling queue Implementation

- First, let's define a struct to represent a job.

```
#define MAXSIZE
typedef struct job{
    int jobId;
    int startTime;
    int jobLength;
    int remLength;
} job;
```

Scheduling queue Implementation

- First, let's define a struct to represent a job.

```
#define MAXSIZE
typedef struct job{
    int jobId;
    int startTime;
    int jobLength;
    int remLength;
} job;
```

- heap of jobs then could be represented as:

```
typedef struct heap{
job list[MAX_SIZE];
int numJobs;
} heap;
```

Scheduling queue Implementation

- First, let's define a struct to represent a job.

```
#define MAXSIZE
typedef struct job{
    int jobId;
    int startTime;
    int jobLength;
    int remLength;
} job;
```

- heap of jobs then could be represented as:

```
typedef struct heap{
job list[MAX_SIZE];
int numJobs;
} heap;
```

- Following are the helper functions to implement the queue.

```
void initHeap(heap *H): (sets numJobs to 0)
void insertJob(heap *H, job j): (inserts the job j in heap)
int extractMinJob(heap *H, job *j): (returns minimum element)
void decreaseKey(heap *H, job *j): (decreases key value of job j)
```

Scheduling queue Implementation

- First, let's define a struct to represent a job.

```
#define MAXSIZE
typedef struct job{
    int jobId;
    int startTime;
    int jobLength;
    int remLength;
} job;
```

- heap of jobs then could be represented as:

```
typedef struct heap{
job list[MAX_SIZE];
int numJobs;
} heap;
```

- Following are the helper functions to implement the queue.

```
void initHeap(heap *H): (sets numJobs to 0)
void insertJob(heap *H, job j): (inserts the job j in heap)
int extractMinJob(heap *H, job *j): (returns minimum element)
void decreaseKey(heap *H, job *j): (decreases key value of job j)
```

Scheduler

Algorithm 1 ScheduleJobs

```
1:  initHeap(H)
2:  sort(jobList)
3:  time ← 0
4:  while jobLeft do
5:      if time == someJob.startTime then
6:          insertJob(H, someJob)
7:      end if
8:      currJob ← extractMinJob(H,temp)
9:      currJob.remLength ← currJob.remLength − 1
10:     time ← time + 1
11: end while=0
```

What if there is a job dependency

- Now suppose that all the jobs are not independent, and there exist some pairs of jobs (x, y) such that if job x finishes before y starts, it can reduce the running time (remLength value) of y by 50%, if it has not started yet.

What if there is a job dependency

- Now suppose that all the jobs are not independent, and there exist some pairs of jobs (x, y) such that if job x finishes before y starts, it can reduce the running time (remLength value) of y by 50%, if it has not started yet.
- Relationship is one way only, y depends on x does not imply x depends on y, so (x, y) and (y, x) are different pairs.

What if there is a job dependency

- Now suppose that all the jobs are not independent, and there exist some pairs of jobs (x, y) such that if job x finishes before y starts, it can reduce the running time (remLength value) of y by 50%, if it has not started yet.
- Relationship is one way only, y depends on x does not imply x depends on y, so (x, y) and (y, x) are different pairs.
- Your job is to schedule the jobs using the same policy, subject to the change mentioned above. However, note that in this case, the duration of a job already in the heap can change in the middle if another job that it depends on finishes.

Changes Required?

- We need to decrease the key but how do we know the position in the heap ?

Changes Required?

- We need to decrease the key but how do we know the position in the heap ?
- We can do that by storing the position of particular jobID in an array which would be updated after calling the heap functions.

```
typedef struct heap{
job list[MAX_SIZE];
int jobPos[MAX_SIZE];
int numJobs;
} heap;
```

Changes Required?

- We need to decrease the key but how do we know the position in the heap ?
- We can do that by storing the position of particular jobID in an array which would be updated after calling the heap functions.

```
typedef struct heap{
job list[MAX_SIZE];
int jobPos[MAX_SIZE];
int numJobs;
} heap;
```

- Implement a helper function *decreaseKey* that would change the *remLength* value to *remLength*/2.

```
void decreaseKey(newheap *H, int jid);
```

Changes Required?

- We need to decrease the key but how do we know the position in the heap ?
- We can do that by storing the position of particular jobID in an array which would be updated after calling the heap functions.

```
typedef struct heap{
job list[MAX_SIZE];
int jobPos[MAX_SIZE];
int numJobs;
} heap;
```

- Implement a helper function *decreaseKey* that would change the *remLength* value to *remLength*/2.

```
void decreaseKey(newheap *H, int jid);
```

- Let's see the modified scheduler!!

Modified Scheduler

Algorithm 2 ScheduleJobs

```
1:  initHeap(H)
2:  sort(jobList)
3:  while jobLeft do
4:      if time == someJob.startTime then
5:          insertJob(H, someJob)
6:      end if
7:      currJob ← extractMinJob(H,temp)
8:      currJob.remLength ← currJob.remLength − 1
9:      if currJob.remLength ≤ 0 then
10:         someJob ← dependsOn(currJob)
11:         decreaseKey(H, someJob)
12:     end if
13:     time ← time + 1
14: end while=0
```

More Problems

Solve these problems on your own -

- Monk and Multiplication
- Killing Zombies
- Monk And Champions League

Thank You!

Any Further Questions?