Design and Analysis of Algorithms

L23: Job Scheduling

Dr. Ram P Rustagi
Sem IV (2019-H1)
Dept of CSE, KSIT/KSSEM
rprustagi@ksit.edu.in

Resources

- Text book 2: Sec 4.1, 4.3, 4.4
- Text book 1: Sec 9.1-5.4 - Levitin
- R1: Introduction to Algorithms
    - Cormen et al.

Example Case

- In college fest which starts at 9:00am, there are a number of available events as below to participate, and each event takes 1 unit of time (e.g. 1hr).
  - Each event has different awards values
  - Each event has its own closing timeline.

Event	Closing	Award
`Mimcry`	12:00	200
`Drama`	11:00	100
`Painting`	12:00	90
`Dance`	10:00	50
`JAM`	11:00	125
`Singing`	10:00	60

Deadline
3
2
3
1
2
1

Q: What is the max award you can get?

Greedy Job Scheduling

- A set of n jobs to run on a computer
- Each job i has a deadline $d_i \geq 1$ and profit $p_i \geq 0$
- There is only one computer
- Each job takes one unit of time (simplification)
- Profit is earned when job is completed by deadline
- Find the subset of jobs that maximizes the profit, i.e.

  Maximize $\Sigma_{i \in J}\ P_i$

Note: It belongs to subset paradigm since we are looking at subset of jobs.

Example: Job Scheduling

Job	Profit	Dead -line
1	100	2
2	10	1
3	15	2
4	27	1

Feasible Solutions	Profit
1	100
2	10
3	15
4	27
1,2	110
1,3	115
1,4	127
2,3	25
3,4	42

Optimal Solution: 1,4

Job Scheduling: Greedy Approach

- What should be the optimization measure to schedule the next job?
- First attempt:
  - Choose $\Sigma_{i \in J} P_i$ as the optimization measure
  - i.e. choose a job that increases this value maximum
    - Subject to constraint of the deadline i.e. J (set of jobs) should be feasible solution.
  - How to choose jobs:
    - Order jobs in decreasing order of profit
    - Choose job one at a time as per this order and add to the solution if solution remains feasible.

Job Scheduling: Greedy Approach

Job	Profit	Deadline
1	100	2
2	10	1
3	15	2
4	27	1

- Application of First Greedy approach
  - Job 1 is added to J. Feasible `{1}`
  - Next: Job 4 is considered as per order.
    - Is set `J={1,4}` feasible.
      - Yes if 4-1, No if 1-4
    - Thus `{1,4}` is feasible solution.
  - Next: Job 3 is considered, `{1,4,3}` is infeasible, thus J remains `{1,4}`
  - Next: Job 2 is considered `{1,4,2}` is infeasible thus J remains `{1,4}`
  - The max profit is `127` for `J={1,4}`
- Time complexity :
  - `n!` to evaluate feasibility for a given set

Job Scheduling: Feasible Solution

- How to determine that a given set of jobs constitute feasible solution.
- Try out all possible permutations in jobs J
  - Check for each permutation if jobs can be scheduled meeting the deadlines.
- Easy to check or a given permutation $\sigma = i_1, i_2, \ldots, i_k$
  - Job i_q must be completed by time q, $1 \le q \le k$
  - If for some job i_q, $q > d_{i_q}$, then job i_q is not completed by d_{i_q}.
- When $|J| = k$, all $k!$ permutations must be checked
- Can we find one permutation that meets the need?
  - Order the jobs in non-decreasing order of deadlines

Proof for Feasible Solution

- Theorem 1:
  - Let J be the set of k jobs and $\sigma = i_1, i_2, \ldots, i_k$ is a permutation of jobs in J such that $d_{i_1} \leq d_{i_2} \leq \ldots \leq d_{i_k}$.
    Then J is a feasible solution if and only if (*iff*) the jobs in J can be processed in the order σ without violating any deadline.
- Theorem 2:
  - The greedy method describes above always obtains an optimal solution to the job scheduling problem.

Algo High Level

```
Algo GreedyJob(int d[], set J, int n) {
    // J is set of jobs that can be completed in deadlines d[]
    J={1}
    for i=2 to n {
        if all jobs in J U {i} can be completed, then
            // by their deadlines
            J = J U {i}
    }
}
```

Algo-1: Job Scheduling

int `JobSchedule2`(*int* `d[]`, *int* `j[]`, *int* `n`) {

//`n`≥`1`, and deadlines `d[i]`≥`1`, `1`≤`i`≤`n`

//Jobs are ordered such that their profits are in non-increasing order i.e. `p[1]`≥`p[2]`≥…≥`p[n]` .

//`J[i]` is the i^{th} job in the optimal solution with `k`≤`n` jobs

// At algo termination, `d[J[i]]`≤`d[j[i+1]],  1`≤`i<k`

// Initialize

`d[0] = 0` // fictitious job with deadline of 0

// allows for job insertion at position `1` later.

`J[0] = 0` // this job is boundary and can't be scheduled

`J[1] = 1` // start with job `1` with highest profit

`k = 1` // job set size is `1` to start with

Algo1: Job Scheduling

```
for(i=2; i≤n; i++) {
  // consider jobs in non-increasing order of p[i]
  // find pos for J[i] and check for feasibility of insertion
  int r = k //job set size
  while ((d[J[r]]>d[i]) &&(d[J[r]!=r))
      r–; //find position where job i can be considered.
  if ((d[J[r]]≤d[i]) &&(d[J[r]>r)){
      //insert i into J[]
      for (int q=k; q≥(r+1); q--)
          J[q+1]=J[q] // increase deadline of jobs by 1.
      J[r+1]=i
      k++ // since job i is feasible, increase the set size.
  }//end if
}//end for i
return k
}
```

Algo-1: Time Complexity

- For loop run n times.
  - Each job needs to be considered.
- if K is the value of max deadline, then
  - Inside while loop plus for loop (for shifting slots) may run K times.
- Time complexity: $O(nK)$
- Considering K is of order of n (if all jobs can be scheduled)
- Time complexity: $O(n^2)$

Algo-2: Job Scheduling

```
//Approach: schedule a job in the slot where it meets deadline.
// If no slot is available before deadline, then job is not scheduled.
// jobs are ordered in non-increasing order as per deadlines.
int JobSchedule-1(int d[], int j[], int n) {
  //n≥1, and deadlines d[i]≥1, 1≤i≤n
  //Jobs are ordered such that their profits are in non-
  increasing order i.e. p[1]≥p[2]≥...≥p[n] .
  //Job[i] is ith job in the optimal solution with k≤n jobs
  // At algo termination, d[Job[i]]≤d[job[i+1]],
  1≤i<k
  // Initialization
  k=0;  // size of Job schedule
  for i=1 to n
     slot[i]=False // all slots are initialized to false
```

Algo-2: Job Scheduling

```
for (i=1; i≤n; i++) {
  // consider jobs in non-increasing order of p[i]
  //check if any slot available before deadline
  while (j=d[i]; j>0; j—) {
      //find position where job i can be considered.
      if (slot[j] == False{
        //Add jobs to the slot
        slot[j] = True;
        Job[j] = i;
        k++;
        break
      }//end if
  }//end while
 } // end for
 return k
}
```

Algo-2: Time Complexity

- For loop run n times.
  - Each job needs to be considered.
- if `K` is the value of max deadline, then
  - while loop may run `K` times.
- Time complexity: `O(nK)`
- Considering K is of order of n (if all jobs can be scheduled)
- Time complexity: $O(n^2)$

Union-Find Approach

- Using set based Union-Find approach
  - It is almost O(n)
- Union-Find approach
- Consider the set of `n` elements which are known.
- All these elements put in an array and their id can be the array index i.e.
  - $a[i] = x_i$ # i^{th} element is s_i.
- Elements are divided into groups (sets)
  - Initially, each element is a group by itself.
- Two kinds of operations:
  - Find the group to which element belongs
  - Merge the two groups

Union-Find Approach

- Two operations given below are performed in arbitrary order
  - *Find(i)*: return the group id containing element xi.
  - *Union(A, B)*: Combine the (set) group A with (set) group B to form a new group.
    - Give a unique name to this group. All elements of this new group will have this group id.
    - This could be one of earlier groups as well i.e.
      - The new names should conflict with other names.
- Goal: Design an efficient data structure that will support any sequence of these two operations.

Union-Find Approach

- Approach 1:
  - Keep ***Find***(i) efficient. Since all elements are accessible at the i^{th} index in array,
    - This can take $O(1)$ time. Essentially, a trivial operation.
  - ***Union***(A, B) is expected to take more time.
    - Either change the id of all elements of A to that of B or vice versa.
    - Typically, take the smaller set and change group identity of these elements to that of larger set.
- Time complexity: $O(n\log_2 n)$
  - Each time an element's group is changed, group size at least doubles.

Union-Find Approach: Improved

- Approach 2:
  - Make ***Union***(`A,B`) efficient at the cost of ***Find***(`i`).
    - Make *Union* operation takes constant time and improve upon the time taken by ***Find***.
    - Use the indirect addressing for union to make it in `O(1)` time.
    - Each entry in the array has two parts
      - Identity of element i.e. group id or value.
      - Pointer which is initially `Nil`.
    - Union(A,B) is performed by making the pointer of B to point to A.
    - After several union operations, data structures becomes like a tree.

Union-Find Approach: Improved

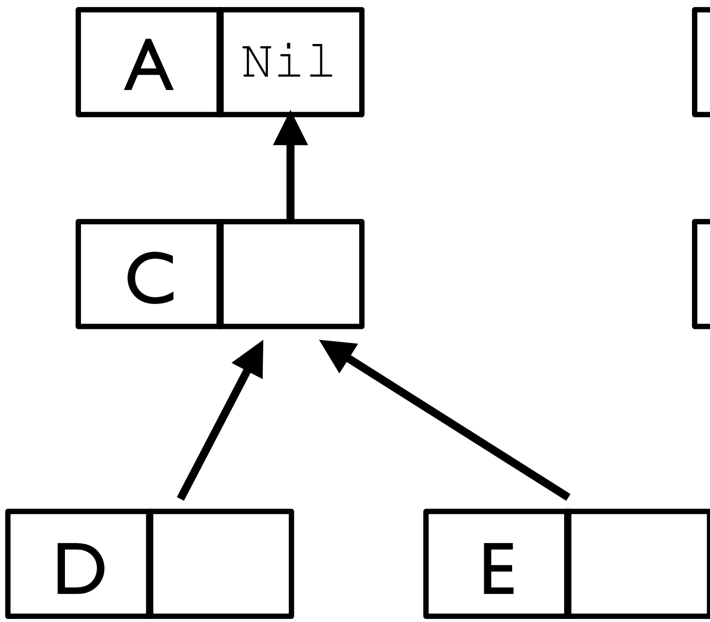
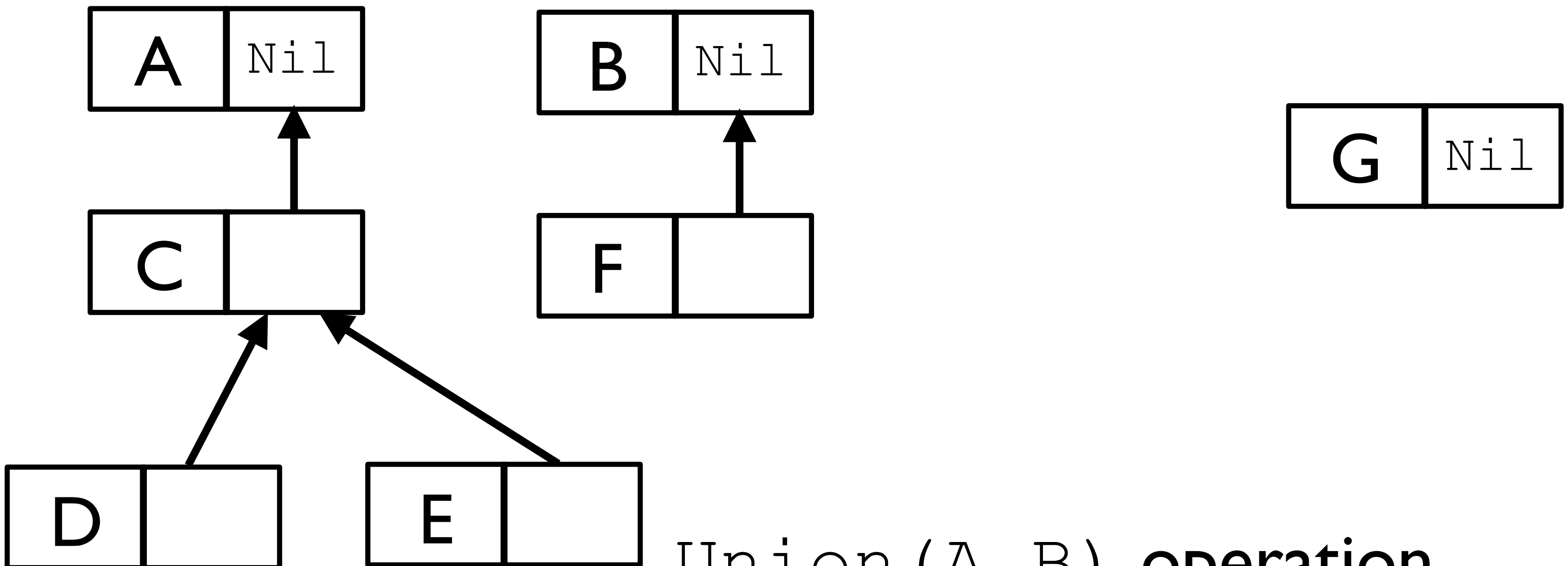


Fig: Representation for Union Find problem

- The element at root of the Tree is the identify of group.
- To find the group of an element, follow the path till the root of the tree.

Union-Find Approach: Improved



`Union(A,B)` operation

- Take the tree with smaller number of nodes to point to root of the tree of larger number of nodes.
- This requires storing the number of elements in the tree at the root as well.
- On Union operation, update the pointer of smaller tree and count at the larger tree.
    - Break the tie arbitrarily

Union-Find Approach: Improved

- Basic idea: Balance and collapse the tree
  - Union operation still takes `O(1)` time
    - Changing the pointer `O(1)` time
    - Updating the count `O(1)` time
  - Thus, `O(1) + O(1) = O(1)`

Union-Find Approach: Improved

- Theorem: When balancing is used, the tree of height h will contain at least 2^h nodes.
- Proof outline
  - First union operation results in tree of height 1 with two elements.
  - Consider A is height h_A and B is of height h_B.
    - Let A is larger tree. Thus, on merging B, root of B points to root of A.
    - If $h_A > h_B$, then A's height remains h_A i.e. unchanged
    - Otherwise, height of tree becomes h_B+1
    - Thus, with increase in height of 1, the size of tree has at least doubled.
- Thus, time taken for a *Find*(X) operation is $O(\log_2 n)$

Union-Find Approach: Improved

- Further improvement
- Any time we do a find operation, change the pointers of all the nodes in the path to directly point to the root of the tree.
  – This is called **path compression**.
- Traversing the path takes only double the number of steps, and thus
  – Time complexity of find remains the same.
- Time complexity with path compression for m operations is given by
  $O(m\log^* n)$, where $\log^* n$ is iterated logarithm function

Iterated Logarithm

- Iterated logarithm function is defined as

$\log^* n = 1 + \log^*(\lceil \log_2 n \rceil)$

$\log^* 2 = 1$ (Given)

$\log^* 4 = \log^* 2^2$

$= 1 + \log^*(\lceil \log_2 2^2 \rceil)$

$= 1 + \log^* 2 = 1 + 1 = 2$

$\log^* 16 = \log^* 2^4$

$= 1 + \log^*(\lceil \log_2 2^4 \rceil)$

$= 1 + \log^* 4 = 1 + 2 = 3$

$\log^* 65536 = \log^* 2^{16}$

$= 1 + \log^*(\lceil \log_2 2^{16} \rceil)$

$= 1 + \log^* 16 = 1 + 3 = 4$

$\log^* 2^{65536} = 1 + \log^* 65536 = 5$ (very large n)

Fast Job Scheduling (Union-Find)

- Let `i` denote the timeslot `i`
  - At the start time, each time slot is its own set
- There are m timeslots, where

  $m = \min(n, \max(d_i))$

  - i.e. the latest deadline
- Each set of `k` slots has a value `F(k)` for all slots `i` set `k`
  - `F(k)` : Stores highest free timeslot before this time
  - `F(k)` : Defined only for root node in set
- Initially all slots are free

Summary

- ?