Module 3: Greedy Algorithms

Overview

Module Overview
Greedy algorithms are class of simple and intuitive *heuristic* algorithms used to solve various optimization problems. This algorithmic paradigm builds solutions, one piece at a time, following certain choice mechanisms that favor immediate and obvious benefits. Greedy algorithms are best suited for solving optimization problems where finding a locally optimal solution leads to the globally optimal solution. Common applications of the greedy algorithmic paradigm include scheduling problems, shortest path problems, and finding minimum spanning trees. In this module, you will learn the fundamentals of the greedy paradigm, and several greedy algorithms for interval scheduling, interval partitioning, scheduling to minimize lateness, finding the shortest path in a graph (Dijkstra algorithm), and finding the minimum spanning tree (Kruskal and Prim algorithms). Each lesson discusses basic definitions, examples, algorithm properties, running times, and implementation using Python.
Learning Objectives
• Explain the definition, principles, and engineering applications of greedy algorithms. • Solve and analyze greedy algorithms for solving scheduling, partitioning, and clustering problems. • Implement Dijkstra algorithm for solving shortest path problems. • Implement Prim and Kruskal algorithms for solving minimum spanning tree (MST) problems.
Course Outcomes
• Reading(s) • Lesson(s) • Discussion • Assignment • [Add_activity]

Reading and Resources	
Optional Resources	**Description**
Jon Kleinberg and Eva Tardos. Algorithm Design. USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN: 0321295358.	Chapter 4
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Third Edition. The MIT Press, 2009. ISBN: 0262033844.	Chapter 16

Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. Data Structures and Algorithms in Python, First Edition. Wiley Publishing, 2013. ISBN: 1118290275.	Chapter 14

Lesson 3.1: Fundamentals Of Greedy Algorithms

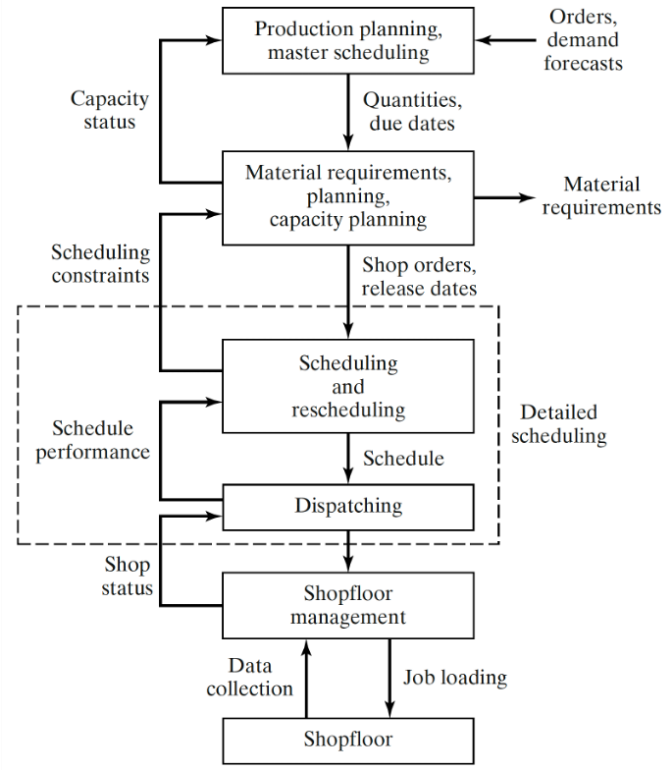
Consider a set of Automated Guided Vehicles (AGVs) tasked with visiting several locations in a warehouse to pickup/deliver different items during each trip. To reduce backlogs and improve process efficiency, the warehouse engineer is looking for a solution that minimizes the amount of time taken by each AGV to fetch or deliver the items. This can be achieved by identifying the best sequence of visits to the desired locations that yields the "shortest path" taken by each AGV during a trip. Given the size of the warehouse, however, examining all possible choices is overkill. This is a scenario where a **greedy algorithm** could be of great help. As the name suggests, a greedy algorithm always makes the choice that looks best at the moment without considering its global optimality. In our AGV example, a greedy algorithm would choose the nearest warehouse location for each AGV to visit next, in the hope that these choices will eventually lead to the minimum (or near-minimum) path length. Greedy algorithms do not necessarily yield optimal solutions; however, in many cases they are able to quickly find near-optimal solutions.

Brief History of Greedy Algorithms. Greedy algorithms were introduced in the 1950s. A pioneer in this area, Esdger Djikstra developed a greedy algorithm in 1959 to find the shortest path from Rotterdam to Groningen in Netherlands, based on a simple intuition: Once you are in a city (e.g., Amsterdam), you clearly know how long it will take to travel to the next connected cities on your map (e.g., Utrecht, Zwolle, Almere). Around the same time, Robert Prim and Joseph Kruskal devised optimization strategies for minimizing path costs on weighed routes. In In the 1970s, Thomas Cormen, Ronald Rivest, and Clifford Stein introduced a recursive sub-structuring of greedy algorithms. This algorithmic paradigm was recognized as one of the optimization strategies by the U.S. National Institute of Standards and Technology (NIST) in 2005. Greedy algorithms are still at the heart of many engineered systems such as web-based protocols, data management systems, project management systems, routing and navigation systems, among others.

Why Use Greedy Algorithms? Two key properties must be taken into consideration before designing a greedy algorithm for solving a particular optimization problem: (1) We need to determine if the problem has an **optimal substructure**; that is, an optimal solution can be efficiently constructed from optimal solutions to sub-problems. In Djikstra's shortest-path problem from Rotterdam to Groningen, for example, we have the luxury of assuming that we arrived at a sub-problem (e.g., an intermediate city) by having made the greedy choice (i.e., different paths) in the original problem. The challenge is to show that an optimal solution to the sub-problem, identified by the greedy choice, yields an optimal solution to the original problem. (2) Greedy is an algorithmic paradigm that always searches for the most obvious and immediate benefit based on the choices made so far, without considering its impact on future choices and sub-problems. That is, a greedy strategy usually progresses in a *top-down* fashion, making one greedy choice after another, reducing each given problem instance to a smaller one. Hence, we will also need to show that if we make a **greedy choice**, then only one sub-problem remains, and that it is always safe to make greedy choices. In the following lessons, you will learn the fundamentals of greedy algorithms and their applications for solving various engineering problems such as scheduling, task partitioning, and path optimization.

Lesson 3.2: Interval Scheduling

Scheduling is a fundamental process in various engineering systems for optimizing and controlling workloads on processors (e.g., production equipment, servers) such that the overall time or cost is minimized. In a manufacturing system, for example, orders have to be translated into "jobs" with specific characteristics such as arrival times, processing times, precedence relations, and due dates. Accordingly, the jobs need to be processed on process equipment or workstations in a given sequence. They may be delayed or preempted, and the workers, equipment, or other resources may be unavailable due to maintenance or breakdowns. Accordingly, finding the optimal schedule is key to ensure overall efficiency of the system.



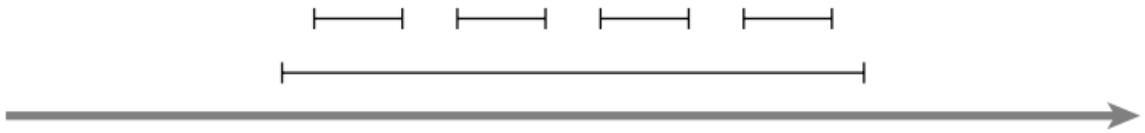
Interval Scheduling Problem. Consider n jobs each with a start time $s(i)$ and finish time $f(i)$, $s(i) < f(i)$, $\forall i \in 1, ..., n$. Jobs i and j are considered to be *compatible* if their requested processing intervals do not overlap: that is , either $f(i) \leq s(j)$ or $f(j) \leq s(i)$. A subset A of jobs is thus considered compatible if all pair of jobs $i, j \in A$, $i \neq j$ are compatible. The goal of interval scheduling is to select as large a compatible subset of jobs of maximum possible size. Compatible sets of maximum size will be called optimal.

Add Interactive Media Here

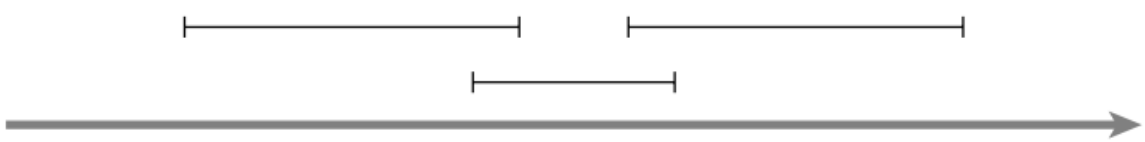
Sample interactive media for Interval Scheduling from Instructors resources folder - interval scheduling demo.ppt

Designing a Greedy Algorithm for Interval Scheduling. The goal is to first select a job i_1 and reject all other jobs that are not compatible with i_1, and then select the next job i_2 and reject all the jobs that are not compatible with i_2, and carry on this process until there are no jobs left. *The challenge, however, is in deciding which choice rule will yield the best solution.* Consider, for example, the following three choice rule "heuristics":

1. A potential approach could be choosing the job with the minimum start time $s(i)$ as the first job in each step. This, however, may not lead to an optimal solution because if the processing time of the earliest job is relatively long, we might end up missing several subsequent jobs with shorter processing times, while our goal is to select as many jobs as possible. In the worst case scenario where the finish time $f(i)$ of the first job is the maximum among all the jobs, the accepted job i will keep the processing resource occupied for the entire time, leading to the selection of only one job (see below).



2. An alternative approach would be to accept the job that requires the smallest processing time; i.e., the job with minimum $f(i) - s(i)$. Though this approach seems to be better than the first greedy approach; we may still end up with a sub-optimal solution as depicted below—here the job in the middle with the smallest interval is not letting us accept the other two.



3. Now consider another greedy approach which counts the number of other orders that are not compatible and chooses the job with the minimum number of incompatible jobs. With this approach which is proven to be optimal (see below), the above examples can be optimized to accommodate three jobs in the compatible subset A of jobs as shown below.

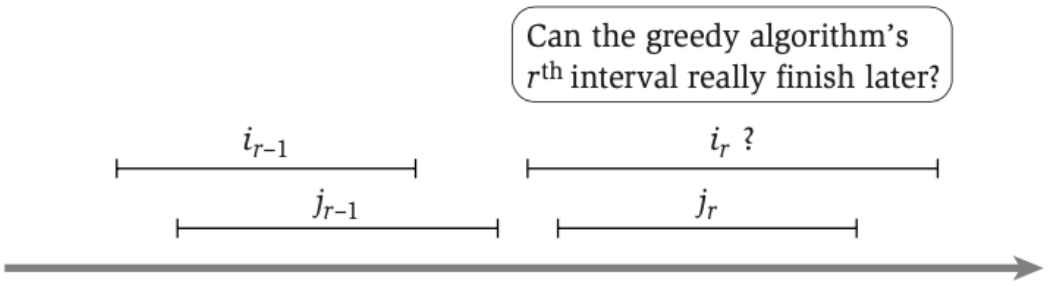
Algorithm 1: Greedy Algorithm for Interval Scheduling

Let R be the set of all jobs and let A be an empty set
while R *is not empty* **do**
  Choose a job $i \in R$ that has the smallest finishing time
  Add job i to A
  Delete all jobs that are not compatible with job i from R
end
Return A as the set of accepted jobs

Analyzing the Greedy Algorithm for Interval Scheduling. Algorithm 1 is proven to yield an optimal solution for interval scheduling, as proven below.

Theorem 1. A is a compatible set of jobs [1].

Proof. Let O be the optimal set of jobs and let $i_1, ..., i_k$ be the set of jobs in A in the order they were added to A. Now let the set of jobs in O be denoted by $j_1, ..., j_m$ in the ascending order of their processing times. We need to show that A contains the same number of jobs as O; i.e., $k = m$. Our greedy rule in Algorithm 1 guarantees that $f(i_1) \leq f(j_1)$ (see the next Theorem). Thus, we now prove that for each $r \geq 1$, the r^{th} accepted job in the algorithm's schedule finishes no later than the r^{th} job in the optimal schedule. Given the optimality of O, we can argue that $k = m$. □



Theorem 2. For all indices $r \leq k$, we have $f(i_r) \leq f(j_r)$ [1].

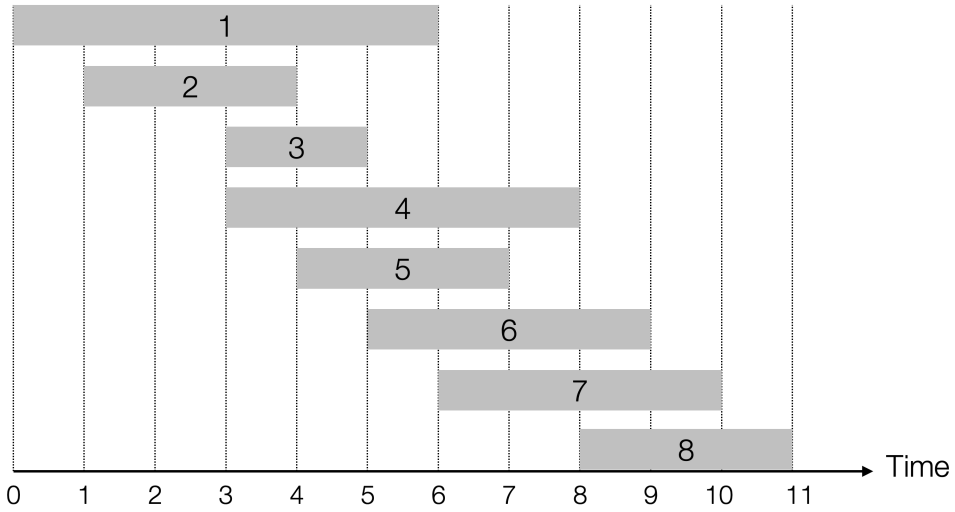
Proof. For $r = 1$, the statement is clearly true: the algorithm starts by selecting job i_1 with minimum processing time. Now let $r > 1$. We will assume as our induction hypothesis that the statement is true for $r1$, and we will try to prove it for r. The induction hypothesis concludes that $f(i_{r1}) \leq f(j_{r1})$. For the Algorithm 1's r_{th} interval not to finish earlier than O, it would need to "fall behind" as shown in the figure above. However, greedy Algorithm 1 always has the option of choosing j_r over i_r. We know that $f(j_{r1}) \leq s(j_r)$ (since O consists of compatible intervals) and that $f(i_{r1}) \leq f(j_{r1})$ (from the induction hypothesis); thus, we get $f(i_{r1}) \leq s(j_r$. The interval j_r is therefore in set R of available intervals at the time when the greedy algorithm chooses i_r. Since j_r is one of these available intervals, we have $f(i_r) \leq f(j_r)$. □

Theorem 3. The greedy Algorithm 1 returns an optimal set A [1].

Proof. By contradiction: If A is not optimal, then an optimal set O must contain more jobs; i.e., $m > k$. Applying the above theorem with $r = k$, we get that $f(i_k) \leq f(j_k)$. Since $m > k$, there is a job j_{k+1} in O which starts after job j_k ends, and hence after i_k ends. Therefore, after deleting all jobs that are incompatible with $i_1, ..., i_k$, the set of possible jobs R still contains j_{k+1}. However, greedy Algorithm 1 stops only when R is empty. □

Algorithm Running Time and Implementation. Sorting all the jobs in the order of finishing time and labelling them in that order will take approximately $O(n \log(n)$ time. We also need additional $O(n)$ time to construct an array $S[1...n]$ with the property that $S[i]$ contains the value $s(i)$: We select jobs by processing the intervals in ascending order of $f(i)$. We always select the first interval and then iterate through the intervals until reaching the first interval j for which $s(j) \geq f(1)$. In this way, we implement the greedy algorithm analyzed above in one pass through the intervals, spending constant time per interval. Thus, this part of the algorithm takes $O(n)$ time.

Example 1: Consider the following eight jobs with the specified start and finish times. The optimal interval schedule for these jobs can be computed by implementing Algorithm 1. You may find the schedule manually or using the following Python script for interval scheduling.



My screen recording programming and explaining the following Python script on Google Colab.

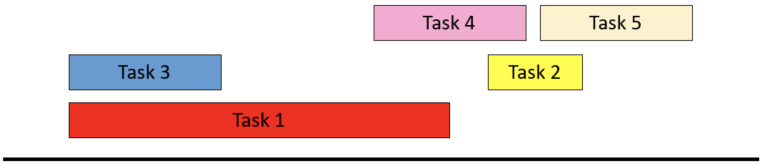
Python Script. Here is a Python script for interval scheduling using greedy Algorithm 1.

```
def interval_scheduling(stimes, ftimes):
    index = list(range(len(stimes)))
    # Sort jobs according to finish times
    index.sort(key=lambda i: ftimes[i])

    # Initate compatible subset A
    compatible_subset = set()
    prev_finish_time = 0
    for i in index:
        if stimes[i] >= prev_finish_time:
            compatible_subset.add(i+1)
            prev_finish_time = ftimes[i]

    return compatible_subset
```

Use the above Python code along with the script below to find the optimal interval schedule for the jobs in Example 1.

```
# Driver Code
stimes = [0, 1, 3, 3, 4, 5, 6, 8]
ftimes = [6, 4, 5, 8, 7, 9, 10, 11]

ans = interval_scheduling(stimes, ftimes)
print('An optimal compatible subset of jobs is', ans)
```

CYK. Applying Algorithm 1 for generating the interval schedule of the following five tasks, which item shows the correct departure order of the tasks from set R?



a 1, 3, 2, 4, 5
b 5, 4, 2, 1, 3
c 3, 1, 2, 4, 5
d 2, 4, 5, 3, 1
e 4, 5, 2, 1, 3

Correct answer: d

Lesson 3.3: Interval Partitioning

The interval scheduling problem involves a single processor for processing multiple jobs with different processing times. Naturally, the interval schedule will only accept a subset of compatible jobs and reject the rest. Now consider a system with multiple processors, such as a CNC machine shop where multiple identical CNC mill or turning machines that can process different machining jobs in parallel. Considering the average daily demand of the shop, the fixed costs of purchasing the machines and the variable costs of operating and maintaining them, an important practical problem would be to identify the minimum number of machines and the optimal partitioning of the job intervals across the machines such that the demand is fulfilled while the costs of operating the shop are minimized.

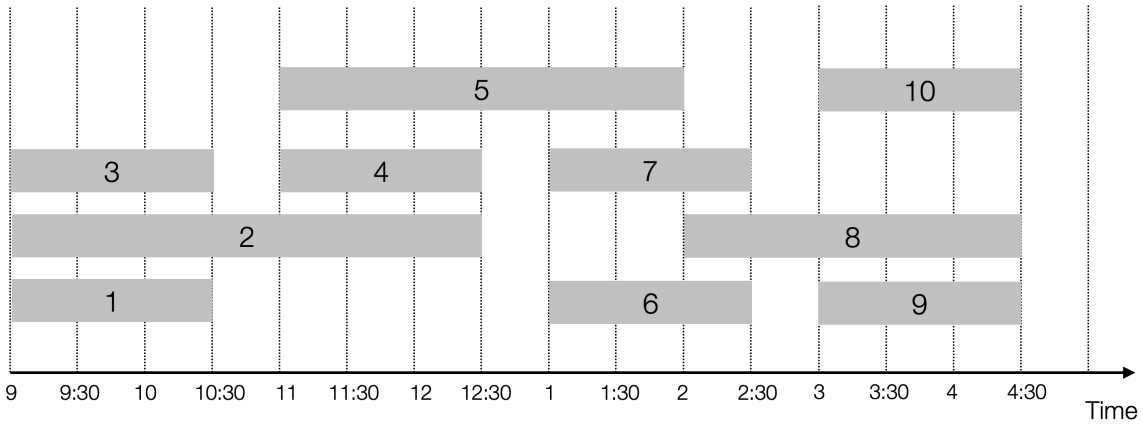
Interval Partitioning Problem. Consider n jobs each with a start time $s(i)$ and finish time $f(i)$, $s(i) < f(i)$, $\forall i \in 1, ..., n$. The goal of interval partitioning is to identify the minimum number of processors and their respective schedules such that all jobs $1, ..., n$ are processed. The main constraint is that any two overlapping jobs must be scheduled on different processors. The problem is also referred to as the **Interval Coloring Problem** where the jobs assigned to different processors are given distinct colors.

Theorem 4. In any instance of interval partitioning, the number of processors needed is at least the *depth* of the set of intervals (i.e., jobs) [1]; i.e., the maximum number of intervals that overlap at any point in the timeline.

Proof. Suppose a set of intervals has depth d, and let $I_1, ..., I_d$ all pass over a common point on the

timeline. Each of these intervals must therefore be scheduled on a different processor. Hence, the minimum number of processors needed is d. □

Example 2: In the below figure, we need at least four processors to process all the jobs.



Designing a Greedy Algorithm for Interval Partitioning. Let d be the depth of the set of intervals. We need to assign a label to each interval from the set of numbers $1, 2, ..., d$. The assignment has the property that overlapping intervals must be labeled with different numbers. This approach should yield the desired solution, because we can interpret each number as the ID of a processor, and the label of each interval as the name of the processor to which it is assigned. The algorithm follows a simple one-pass greedy strategy that orders intervals by their starting times. We go through the intervals in this order, and try to assign to each interval we encounter a label that has not been assigned to any previous overlapping interval yet. Specifically, we have the following description.

Algorithm 2: Greedy Algorithm for Interval Partitioning

```
Sort the intervals by their start times. Break ties arbitrarily
Let I_1, I_2, ..., I_n denote the intervals in this order
for j = 1, 2, ..., n do
    for each interval I_i that precedes I_j in sorted order and overlaps it do
        Exclude the label of I_i from consideration for I_j
    end
    if there is any label from 1, 2, ..., d that has not been excluded then
        Assign a non-excluded label to I_j
    else
        Leave I_j unlabeled
    end
end
```

Analyzing the Greedy Algorithm for Interval Partitioning. Algorithm 2 is proven to yield an optimal solution for interval scheduling, as proven below.

Theorem 5. Greedy Algorithm 2 assigns every interval a label ensuring that no two overlapping intervals receive the same label [1].

Proof. Consider one of the intervals I_j, and suppose there are t intervals earlier in the sorted order that overlap it. These t intervals together with I_j form a set of $t + 1$ intervals that all pass over a common point on the timeline. Therefore, $t \leq d1$. It follows that at least one of the d labels is not excluded by this set of t intervals. Thus, there is a label that can be assigned to I_j. That proves that no interval will remain unlabeled. Now consider any two intervals I and I that overlap, and suppose I precedes I in the sorted order. When I is considered by the algorithm, I is in the set of intervals whose labels are excluded from consideration. As a consequence, the algorithm will not assign to I the label used for I. □

Algorithm Running Time and Implementation. A preferred way to store the processors is in a priority queue. Priority is given to processors that have been free the longest. This algorithm runs in $O(n \log(n)$ time. For each of the intervals, we must choose a processor to assign them to. This choice takes at worst $O(n \log(n)$ time, assuming we implement the priority queue as a heap, since each time we add to a processor, we need to remove it and reinsert it into the priority queue since its key for the priority queue has been changed.

Example 1 (revisited): Recall the eight job intervals from Example 1. If we were to process all the jobs, how many processors would be needed? The optimal interval partition for these jobs can be computed by implementing Algorithm 2. You can use the following Python script for answering these questions.

My screen recording programming and explaining the following Python script on Google Colab.

Python Script. Here is a Python script for interval partitioning using greedy Algorithm 2.

```
def sortJobs(jobList):
    # Assume job list looks like [{"start": 10, "end": 10.75}, {"start":
    ↪  10.5, "end": 1}, ...]
    return sorted(jobList, key=lambda x: x["start"])
```

```
def Equipment(jobList):
    sortedJobList = sortJobs(jobList)
    equipment = []
    for time in sortedJobList:
        assigned = False
        for processor in equipment:
            if processor['end'] <= time['start']:  # This processor is free!
                assigned = True
                processor['end'] = time['end']
                break

        if not assigned:
            newProcessor = {"end": time['end']}
            equipment.append(newProcessor)
    return len(equipment)
```

Use the above Python code along with the script below to find the minimum number of processors needed for processing all the jobs in Example 1.

```
# Driver Code
jobs = [{"start": 0, "end": 6}, {"start": 1, "end": 4}, {"start": 3, "end":
↪  5}, {"start": 3, "end": 8}, {"start": 4, "end": 7}, {"start": 5, "end":
↪  9}, {"start": 6, "end": 10}, {"start": 8, "end": 11}]

print("The minimum number of processors needed is",  Equipment(jobs))
```

CYK. Interval partitioning allows for multiple feasible assignments of intervals to processors. That is, there could be multiple feasible interval partitioning solutions in which a particular interval is assigned to different processors.

a True
b False

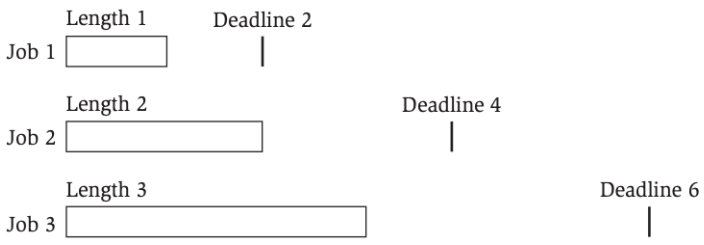
Correct answer: a

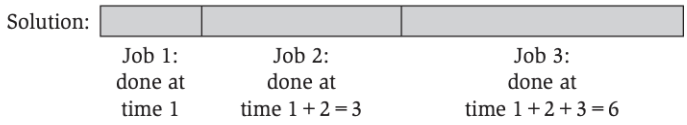
Lesson 3.4: Scheduling To Minimize Lateness

Consider a more advanced single-processor scheduling scenario: A mechanical assembler is tasked with the final assembly of multiple different engine parts in a week. Each assembly job can be processed at any point during the week without interruption, and has an estimated processing time and pre-specified *deadline*. The operator must try to meet the deadlines or the next stages of assembly will go idle. How can the operator decide which engine part to assemble next?

Scheduling to Minimize Lateness Problem. Consider a single processor with starting at time s on the timeline, and n jobs each with a deadline $d(i)$ and processing time interval of $t(i) = f(i) - s(i)$, where $s(i)$ and $f(i)$ are the start time and finish time of job i. All jobs can be initiated as early as s and must be assigned to non-overlapping intervals. Although the plan is to meet the deadlines of all orders, we are allowed to let certain orders run late (i.e., $f(i) \geq d(i)$). The lateness of an order i is defined as $l(i) = \max\{0, f(i) - d(i)\}$. The goal of our optimization problem will be to schedule all orders with no overlapping intervals, so as to minimize the maximum lateness, $L = \max_{i=1,\ldots,n} l(i)$.

Example 2: Consider the following three jobs and their specified deadlines. Job 1 has length $t(1) = 1$ and deadline $d(1) = 2$, Job 2 has $t(2) = 2$ and $d(2) = 4$, and Job 3 has $t(3) = 3$ and $d(3) = 6$. What is the optimal schedule for these three jobs such that the maximum lateness is minimized? It is not hard to check that scheduling the jobs in the order of 1-2-3 yields a maximum lateness of zero. In more complex practical problems, however, the answer may not be so obvious.





Designing a Greedy Algorithm for Scheduling to Minimize Lateness. The goal is to schedule all jobs with no overlap such that the maximum lateness is minimized. Consider, for example, the following three choice rule "heuristics" for deciding which job to schedule next:

1. One possible approach is to order the jobs in increasing order of processing times $t(i)$. The intuition is to get the shorter jobs out of the way as quickly as possible. Although this approach sounds promising on the surface, it completely ignores the deadlines. Consider a two-job example where the first job has $t(i) = 1$ and $d(1) = 100$, and the second job has $t(2) = 10$ and $d(2) = 10$. Clearly, the second job has to be started right away if the goal is to minimize $L = \max_{i=1,2} l(i)$.

2. An alternative approach would be to schedule the jobs in an ascending order of available slack time $d_t - t_t$. That is, prioritize the jobs with tight deadlines to attain minimal delay. Consider a two-job example again, where the first job has $t(1) = 1$ and $d(1) = 2$, and the second job has $t(2) = 10$ and $d(2) = 10$. Sorting by increasing slack would place the second job first in the schedule, which results in a lateness of 9 for the first job. If we schedule the first job first, however, it finishes on-time and the second job incurs a lateness of only 1. Thus, this heuristic approach may not yield an optimal schedule either.

3. Now, let's consider a basic greedy algorithm that always produces an optimal solution. The algorithm simply sorts the jobs in increasing order of their deadlines d_i, and schedules them in that order. This rule is often called **Earliest Deadline First** or **Earliest Due Date First**. The intuition is that we should make sure that jobs with earlier deadlines get completed earlier. Let s be the start time for all jobs, and let the jobs be labeled in the order of their deadlines (i.e., $d_1 \leq \ldots \leq d_n$). Job 1 will start at time $s = s(1)$ and end at time $f(1) = s(1) + t_1$, Job 2 will start at time $s(2) = f(1)$ and end at time $f(2) = s(2) + t_2$, and so forth. It might be hard to believe that this algorithm yields optimal solutions, especially because it never looks at the job processing times. But we will see how this simple algorithm always works optimally despite throwing half of the data away.

Algorithm 3: Greedy Algorithm for Scheduling to Minimizing Lateness

Label the jobs in order of their deadlines: $d(1) \leq ... \leq d(n)$
Set $f = s$
for $i = 1, 2, ..., n$ **do**
  Assign job i to the time interval from $s(i) = f$ to $f(i) = f + t_i$
  $f(i) = f + t_i$
end
Return the set of scheduled intervals $[s(i), f(i)]$ for $i = 1, 2, ..., n$

Analyzing the Greedy Algorithm for Scheduling to Minimize Lateness. The optimal algorithm for scheduling to minimize lateness should avoid *idle times*—the times when the processor is idle yet there are jobs waiting. The schedules produced by Algorithm 3 have no idle time and also are always optimal.

Theorem 6. Algorithm 3 yields an optimal schedule to minimize lateness, with no idle time [1].

Proof. Considering an optimal schedule O. We need to show that there is a way to gradually modify O, preserving its optimally at each step, and transform it into the schedule found by the greedy Algorithm 3. We say that a schedule found by Algorithm 3 has an *inversion* if a job i with deadline $d(i)$ is scheduled before job j with earlier deadline, i.e., $d(j) < d(i)$. By definition, the schedule produced by our Algorithm 3 has no inversions. If two different schedules have neither inversions nor idle time, then they might not produce exactly the same order of jobs, but they can only differ in the order in which jobs with identical deadlines are scheduled. Consider such a deadline d. In both schedules, the jobs with deadline d are all scheduled consecutively, after all jobs with earlier deadlines and before all jobs with later deadlines. Among the jobs with deadline d, the last one has the greatest lateness, and this lateness does not depend on the order of the jobs. The key step in showing the optimally of Algorithm 3 is to establish that there is an optimal schedule that has no inversions and no idle time. To this end, we will start with an optimal schedule O having no idle time, and then convert it into a schedule with no inversions without increasing its maximum lateness. Thus, the resulting scheduling after this conversion will be optimal as well. □

Algorithm Running Time and Implementation. The jobs can be ordered with respect to their deadlines in $O(n \log n)$ time, and the assignment of job i to the time interval takes $O(n)$ time. Thus, the total running time would be $O(n \log n)$ time.

Example 3: Consider the following six jobs with the specified processing times and deadlines which need to be processed by a single processor. You can compute the optimal schedule to minimize lateness using Algorithm 3. You can use the following Python script to compute the optimal schedule.

job (i)	1	2	3	4	5	6
$t(i)$	3	2	1	4	3	2
$d(i)$	6	8	9	9	14	15

My screen recording programming and explaining the following Python script on Google Colab.

Python Script. Here is a Python script for scheduling to minimize lateness using greedy Algorithm 3.

```
from operator import itemgetter
def get_minimum_lateness():
    schedule =[]
    max_lateness = 0
    t = 0
    sorted_jobs = sorted(jobs,key=itemgetter(2))
    for job in sorted_jobs:
        job_start_time = t
        job_finish_time = t + job[1]
        job_id = job[0]
        t = job_finish_time
        if (job_finish_time > job[2]):
          max_lateness =  max (max_lateness, (job_finish_time - job[2]))
        schedule.append((job_id, job_start_time, job_finish_time))
    return max_lateness, schedule
```

Use the above Python code along with the script below to find the optimal schedule for the jobs in Example 1.

```
# Driver Code
jobs = [(1, 3, 6), (2, 2, 9), (3, 1, 8), (4, 4, 9), (5, 3, 14), (6, 2, 15)]

max_lateness, sc = get_minimum_lateness()
print("Maximum lateness will be " + str(max_lateness))
print("The optimal schedule is as follows (job ID, start time, finish
↪  time):", sc)
```

Lesson 3.4: Shortest Path In A Graph

In Module 2, we learned about the fundamentals of graphs and basic graph-based algorithms. An important problem in any system that can be recast as a graph is finding the shortest path between each pair of vertices on the graph by traversing a sequence of intermediate vertices. Consider the warehouse AGV example discussed in Lesson 3.1, where each AGV is tasked with visiting several locations in a warehouse to pickup or deliver different items. The question we are interested in is: Given a start location, what is the shortest path from it to other destinations of the AGV in the warehouse? The most popular greedy algorithm that can help answer this question is Dijkstra algorithm, which finds the shortest path from a fixed "source" vertex s in a graph to to all other vertices in the graph, generating a **shortest-path tree**.

Dijkstra Algorithm. To identify the shortest-path tree from a source vertex s in a graph with set of vertices V, Dijkstra algorithm offers a simple yet powerful solution. The algorithm creates a set S of vertices u for which there is a shortest-path distance $d(u)$ from s. Initially $S = \{s\}$ and $d(s) = 0$. For each vertex $v \in V - S$, the algorithm constructs the shortest path that traverses along a path through the explored part S to some $u \in S$, followed by the single edge (u, v). That is, we choose vertex $v \in V - S$ which minimizes $d'(v) = min_{e=(u,v):u \in S} d(u) + l_e$, where l_e is the length of edge $e = (u, v)$, and add v to S.

Algorithm 4: Dijkstra Algorithm

Let S be the set of explored nodes
Initially $S = s$ and $d(s) = 0$
while $S \neq V$ **do**
  Select a vertex $v \notin S$ *with at least one edge from* S *for which* $d'(v) = min_{e=(u,v):u \in S} d(u) + l_e$
  Add v *to* S *and define* $d(v) = d'(v)$
end

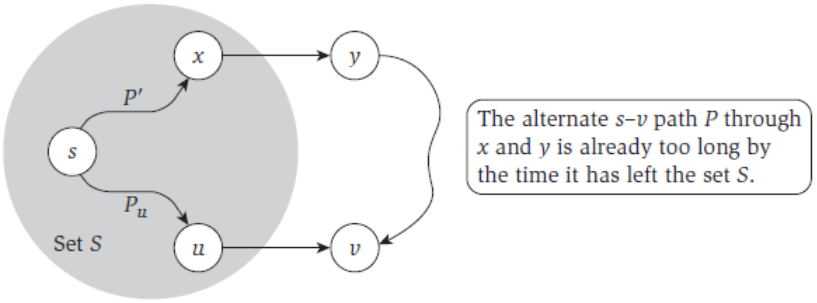
Add Interactive Media Here

Sample interactive media for Dijkstra algorithm
`https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html`

Analyzing Dijkstra Algorithm for Finding Shortest-Path Tree. How does Algorithm 3 always identify the shortest path in a graph? The answer lies in the following property of this algorithm.

Theorem 7. Consider set S at any point in the execution of Dijkstra algorithm. For each $u \in S$, the path P_u is the shortest s-u path [1].

Proof. The answer for the initial case $S = \{S\}$ is obvious, because $d(s) = 0$. Now suppose the above claim also holds when $|S| \geq 1$. Let (u, v) be the final edge on our s-v path P_v. By induction, we know that P_u is the shortest s-u path for all $u \in S$. We want to prove that any other s-v path P is at least as long as P_v. In order to reach v, this path P must leave set S at some point. Let y be the first node on P that is not in S, and let $x \in S$ be the node just before y (see the figure below). P cannot be shorter than P_v because it is already at least as long as P_v by the time it has left set S. Indeed, the algorithm must have considered adding node y to S via the edge (x, y) rather than adding v. Thus, there is no path from s to y through x that is shorter than P_v. However, the sub-path of P up to y is such a path, and therefore this sub-path is at least as long as P_v. Since edge lengths are non-negative, the full path P is at least as long as P_v.



□

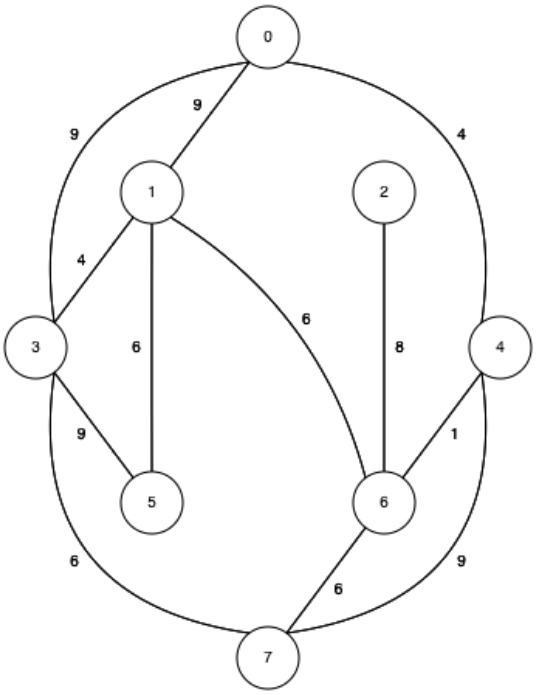
You may have notices an implicit condition of Dijkstra algorithm that all paths must be non-negative. In fact, the algorithm is completely incapable of finding shortest paths on graphs with

negative edge length for a very simple reason: It relies on the fact that if all edges are non-negative, adding a new edge always makes a path longer, and thus, picking the shortest immediate edge always yields the global optimal solution. This claim does not hold if the graph comprises negative edges. In such cases, a more complex and versatile version of Dijkstra algorithm known as the **Bellman-Ford algorithm** will be required. We will discuss this algorithm in future modules.

Dijkstra Algorithm Running Time and Implementation. For a graph with n vertices, the While loop in Algorithm 3 is iterated $n-1$ times, each time adding a new vertex to S. Each iteration would have to consider all vertices $v \notin S$ to compute $min_{e=(u,v):u\in S} d(u) + l_e$ and select the best vertex v. For a graph with m edges, computing this function would take $O(m)$ time; and therefore, the total running time of Algorithm 3 would be $O(mn)$.

Example 4: In the following weighted undirected graph, the shortest path from Vertex 1 to all other vertices can be computed using Dijkstra algorithm as follows:

From vertex	Path	Distance
0	1-0	9
1	1	0
2	1-6-2	14
3	1-3	4
4	1-6-4	7
5	1-5	6
6	1-6	6
7	1-3-7	10



My screen recording programming and explaining the following Python script on Google Colab.

Python Script. Here is a Python script for Dijkstra algorithm.

```
# The program is for adjacency matrix representation (contributed by
↪  Divyanshu Mehta).
import sys

class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printSolution(self, dist):
        print("Vertex tDistance from Source")
        for node in range(self.V):
            print(node, "t", dist[node])

    # A utility function to find the vertex with minimum distance value, from
    ↪  the set of vertices not yet included in shortest path tree
    def minDistance(self, dist, sptSet):

        # Initilaize minimum distance for next vertex
        min = sys.maxsize

        # Search not nearest vertex not in the shortest path tree
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]
                min_index = v

        return min_index
```

```
    # Funtion that implements Dijkstra's single source shortest path
    ↪  algorithm for a graph represented using adjacency matrix
    ↪  representation
    def dijkstra(self, src):

        dist = [sys.maxsize] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):
            # Pick the minimum distance vertex from the set of vertices not
            ↪  yet processed. u is always equal to src in first iteration
            u = self.minDistance(dist, sptSet)

            # Put the minimum distance vertex in the shotest path tree
            sptSet[u] = True

            # Update dist value of the adjacent vertices of the picked vertex
            ↪  only if the current distance is greater than new distance and
            ↪  the vertex in not in the shotest path tree
            for v in range(self.V):
                if self.graph[u][v] > 0 and sptSet[v] == False and dist[v] >
                ↪  dist[u] + self.graph[u][v]:
                  dist[v] = dist[u] + self.graph[u][v]

        self.printSolution(dist)
```

Use the above Python code along with the script below to find the shortest path for all the vertices in Example 1.

```
# Driver Code
g = Graph(8)
g.graph = [[0, 9, 0, 9, 4, 0, 0, 0],
           [9, 0, 0, 4, 0, 6, 6, 0],
           [0, 0, 0, 0, 0, 0, 8, 0],
           [9, 4, 0, 0, 0, 9, 0, 6],
           [4, 0, 0, 0, 0, 0, 1, 9],
           [0, 6, 0, 9, 0, 0, 0, 0],
           [0, 6, 8, 0, 1, 0, 0, 6],
           [0, 0, 0, 6, 9, 0, 6, 0]
          ]
g.dijkstra(1)
```

CYK. Consider a weighted undirected graph and assume that the shortest path from a source s to a destination t is correctly calculated using Dijkstra algorithm. Is the following statement true? If we increase the weight of every edge by 1, the shortest path always remains the same.

a True
b False

Correct answer: b

CYK. Now consider the same weighted undirected graph and shortest path from s to t calculated by Dijkstra algorithm. Is the following statement true? If we modify the graph such that weights of all edges are doubled, then the shortest path remains the same.

a True
b False

Correct answer: a

CYK. Given a weighted graph with unique weights for all edges, there is *always* one unique shortest path from a source s to a destination t.

a True
b False

Correct answer: b

Lesson 3.5: Minimum Spanning Tree

A telecommunication network company has initiated a project to provide Internet access to several rural areas. The company has already installed one Internet tower in each area. Internet communication cables can be installed between the central access point and each tower, as well as between each pair of towers. The costs of connecting the towers vary depending on their distance from the central access point and from the other towers. The telecommunication network company faces a strategic problem: How can they interconnect all rural areas to provide Internet access to all as cheaply as possible? The desired telecommunication network must be in the form of a **spanning tree**; because if it is not a tree, the company can always remove redundant connections to save money. Greedy algorithms can help find the Minimum Spanning Tree (MST) in a graph to solve engineering problems like this, as well as other problems such as constructing broadcasting in computer networks, image registration, segmentation, and feature extraction in computer vision, cluster analysis, and circuit design, among others.

Minimum Spanning Tree (MST) Problem. Consider the above example of the Internet communication network design. Suppose the company has installed a set of Internet towers $V = v_1, v_2, ..., v_n$. The company is examining the suitability of building a direct link between each pair of towers v_i and v_j at the cost of $c(v_i, v_j) > 0$. The set of possible links can be represented by a graph $G = (V, E)$, with a positive cost c_e associated with each edge $e = (v_i, v_j)$. The problem is then to find a subset of the edges $T \in E$ such that (1) the graph (V, T) is connected, and (2) the total cost $\sum_{e \in T} c_e$ is the minimum.

Theorem 8. Let T be a minimum-cost solution to the Internet network design problem defined above. Then, (V, T) is a tree.

Proof. By contradiction: By definition, we know that the graph (V, T) must be connected. If the graph contains a circle C, then $(V, T - e)$, $\forall e \in C$, must be connected. That is, any path that used to go through e must now take the longer way through the remainder of the edges. This implies

that the graph $(V, T - e)$ is a valid solution to the problem with lower total cost than the graph (V, T) because $c_e > 0$. This is a contradiction. □

The MST (Minimum Spanning Tree) can be found following two simple strategies that construct the tree by vertices and edges. The most efficient greedy algorithms in this context are Kruskal algorithm and Prim algorithm, as described next.

Kruskal Algorithm. This greedy algorithms begins with no edges and progressively builds an MST by adding edges from E in an increasing order of edge costs. As the algorithm progresses, it adds the next minimum cost edge as long as it does not create a cycle with the edges already added to the tree. If the next minimum cost edge creates a cycle, it is discarded and the algorithm proceeds to the next minimum cost edge. There is also a **backward** version of Kruskal algorithm, which starts with a full graph and progressively removes edges in a decreasing order of edge costs as long as deleting the next maximum cost edge does not disconnect the graph. This greedy algorithm is also known as the **reverse-delete algorithm**.

Sample interactive media for Kruskal algorithm
`https://www.cs.usfca.edu/~galles/visualization/Kruskal.html`

Kruskal algorithm is based on a data structure called **union-find**: Union-find is a disjoint-set data structure that stores a collection of disjoint (non-overlapping) sets. Equivalently, it stores a partition of a set into disjoint subsets. It provides operations for adding new sets, merging sets, and finding a representative member of a set. The latter allows to find out if any two elements are in the same or different sets. The union find data structure allows edges to be stored separately then merged with each other.

Algorithm 5: Kruskal Algorithm for Finding MST

Initialize $T = (V, \varnothing)$ **for** *each edge* (u, v) *in this order* **do**
  if u *and* v *are in different components of* T **then**
    Add (u, v) to T
  end
end
Return T

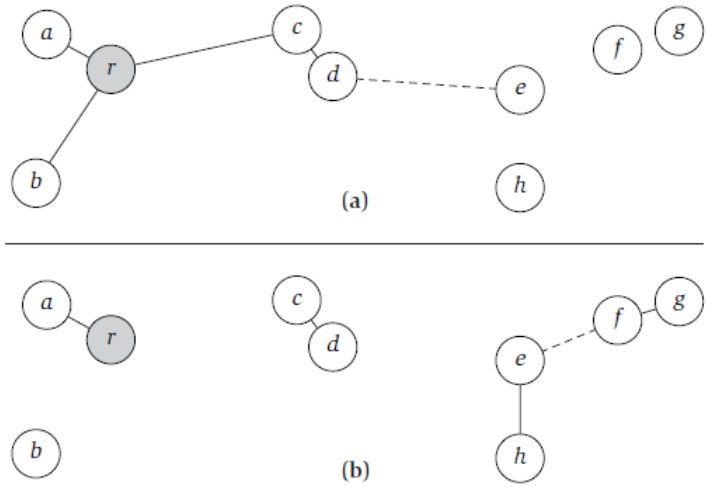
Prim Algorithm. This algorithm starts with a root node s and progressively grows a tree from s outward. The algorithm adds the vertex with minimum cost to the partial tree already constructed. More concretely, a set $S \in V$ is maintained on which a spanning tree has been constructed so far. Initially, $S = s$. In each iteration, the algorithm grows S one vertex at a time, adding vertex v with minimal cost $min_{e=(u,v):u \in S} c_e$, and includes the edge $e = (u, v)$ that achieves this minimum in the spanning tree.

Sample interactive media for Prim algorithm
`https://www.cs.usfca.edu/~galles/visualization/Prim.html`

Algorithm 6: Prim's algorithm

Initialize T to be an empty stack
Initialize U as an empty list
while $U \neq V$ **do**
  Let (u, v) *be the lowest cost edge such that* $u \in U$ *and* $v \in V - U$
  Add (u, v) *to* T $U = U \cup v$
end
Return T

Example 5: Figure below compares the procedures of Kruskal (a) and Prim (b) algorithms. The first four edges added to the spanning tree are indicated by solid lines, and the next edge to be added is a dashed line.



Analyzing Kruskal and Prim Algorithms. Both algorithms progressively add (or remove) edges to (from) the tree. A key question to consider is: How do we "safely" add an edge to ensure that the graph remains a spanning tree? A key property of both algorithms that addresses this question is the following (presented without proof). Let $S \neq \varnothing$ be a subset of the set of vertices V, where $S \neq V$, and let edge e be the minimum cost edge with one vertex in S and another vertex in V. Then every MST must contain e. Following this property, we can now prove the optimality of both algorithms.

Theorem 9. Kruskal algorithm produces an MST for a connected graph $G = (V, E)$ [1].

Proof. Consider any edge $e = (v, w)$ added so far, and let S be the set of all vertices to which v has a path just before e is added. Thus, $w \notin S$ since e should not create a cycle. Further, there is no edge connecting S and VS yet, since any such edge could have been added without creating a cycle. Thus, e is the cheapest edge with one end in S and the other in VS. By the aforementioned property, it should belong to every MST. Therefore, if we show that the output (V, T) of Kruskal algorithm is a spanning tree of G, the proof will be complete. We know that (V, T) contains no cycles, and also if (V, T) were not connected, there would exist a nonempty subset of nodes S with no edge to VS. However, since G is connected, there must be at least one edge between S and VS, and the algorithm will always add the cheapest of those edges.

□

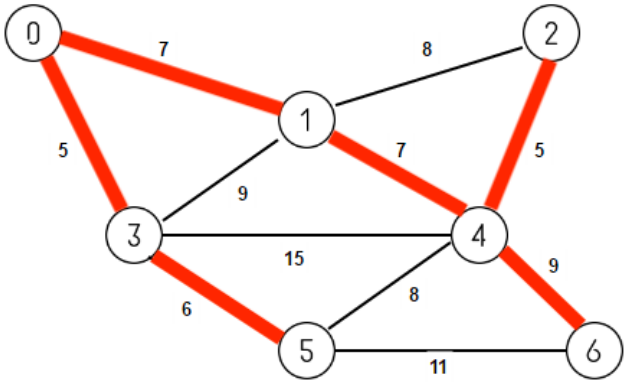
Theorem 10. Prim algorithm produces an MST for a connected graph $G = (V, E)$ [1].

Proof. In each iteration, there exists a set $S \subseteq V$ that constructs a partial spanning tree, and a vertex v and an edge e are added that minimize $min_{e=(u,v):u \in S} c_e$. We know that e is the cheapest edge with one end in S and the other in VS. By the aforementioned property, it is therefore in every MST. Prim algorithm cane easily proven to produce a spanning tree of G, and that concludes the proof.

□

Running Times and Implementation of Kruskal and Prim Algorithms. Kruskal algorithm first sorts the edges by their costs, which takes $O(mlogm)$ time, where m is the number of edges. It then applies the *Union-Find* data structure to maintain the connected components of (V, T). That is, it computes $Find(u)$ and $Find(v)$ when considering any edge $e = (v, w)$ to see if v and w belong to different components. It uses $Union(Find(u), Find(v))$ to merge the two components if edge e is chosen to be included in T. In sum, that includes a total of $2m$ $Find$ operations and $n1$ $Union$ operations, where n is the number of vertices. Accordingly, Kruskal algorithm takes a total of $O(mlogn)$ time to implement. Using the *priority queue* data structure, Prim algorithm can be implemented in $O(m)$ time in addition to the time for n $ExtractMin$ operations and m $ChangeKey$ operations.

Example 6: The MST for the following graph can be identified using Kruskal or Prim algorithm as follows:



Add Instructor's Video Here

My screen recording programming and explaining the following Python script on Google Colab.

Python Script. Here is a Python script for Kruskal algorithm.

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])
    # Search function

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def apply_union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    #  Applying Kruskal algorithm
    def kruskal_algo(self):
        result = []
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.apply_union(parent, rank, x, y)
        for u, v, weight in result:
            print("(%d, %d): %d" % (u, v, weight))
```

Use the above Python code along with the script below to find the MST for the graph in Example 5 using Kruskal algorithm.

```
# Driver Code
g = Graph(7)
g.add_edge(0, 1, 7)
g.add_edge(0, 3, 5)
g.add_edge(1, 0, 7)
g.add_edge(1, 2, 8)
g.add_edge(1, 3, 9)
g.add_edge(1, 4, 7)
g.add_edge(2, 1, 8)
g.add_edge(2, 4, 5)
g.add_edge(3, 0, 5)
g.add_edge(3, 1, 9)
g.add_edge(3, 4, 15)
g.add_edge(3, 5, 5)
g.add_edge(4, 1, 7)
g.add_edge(4, 2, 5)
g.add_edge(4, 3, 15)
g.add_edge(4, 5, 8)
g.add_edge(4, 6, 9)
g.add_edge(5, 3, 6)
g.add_edge(5, 4, 8)
g.add_edge(5, 6, 11)
g.add_edge(6, 4, 9)
g.add_edge(6, 5, 11)

print("The MST edges and corresponding weights for this graph are")
g.kruskal_algo()
```

My screen recording programming and explaining the following Python script on Google Colab.

Python Script. Here is a Python script for Prim algorithm.

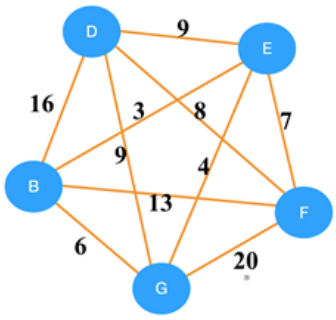
```
# The program is for adjacency matrix representation of the graph
↪  (contributed by Divyanshu Mehta)
import sys

class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    # A utility function to print the constructed MST stored in parent[]
    def printMST(self, parent):
        print("Edge \tWeight")
        for i in range(1, self.V):
            print(parent[i], "-", i, "\t", self.graph[i][parent[i]])

    # A utility function to find the vertex with minimum distance value, from
    ↪  the set of vertices not yet included in shortest path tree
    def minKey(self, key, mstSet):
        # Initilaize min value
        min = sys.maxsize
        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v

        return min_index

    # Function to construct and print MST for a graph represented using
    ↪  adjacency matrix representation
    def primMST(self):

        # Key values used to pick minimum weight edge in cut
        key = [sys.maxsize] * self.V
        parent = [None] * self.V # Array to store constructed MST
        # Make key 0 so that this vertex is picked as first vertex
        key[0] = 0
        mstSet = [False] * self.V
        parent[0] = -1 # First node is always the root of

        for cout in range(self.V):

            # Pick the minimum distance vertex from the set of vertices not
            ↪  yet processed. u is always equal to src in first iteration
            u = self.minKey(key, mstSet)
```

```
            # Put the minimum distance vertex in the shortest path tree
            mstSet[u] = True

            # Update dist value of the adjacent vertices of the picked vertex
            ↪  only if the current distance is greater than new distance and
            ↪  the vertex in not in the shotest path tree
            for v in range(self.V):

                # graph[u][v] is non zero only for adjacent vertices of m
                ↪  mstSet[v] is false for vertices not yet included in MST.
                ↪  Update the key only if graph[u][v] is smaller than key[v]
                if self.graph[u][v] > 0 and mstSet[v] == False and key[v] >
                ↪  self.graph[u][v]:
                        key[v] = self.graph[u][v]
                        parent[v] = u

        self.printMST(parent)
```

Use the above Python code along with the script below to find the MST for the graph in Example 5 using Prim algorithm.

```
# Driver Code
g = Graph(7)
g.graph = [[0, 7, 0, 5, 0, 0, 0],
           [7, 0, 8, 9, 7, 0, 0],
           [0, 8, 0, 0, 5, 0, 0],
           [5, 9, 0, 0, 15, 6, 0],
           [0, 7, 5, 15, 0, 8, 9],
           [0, 0, 0, 6, 8, 0, 11],
           [0, 0, 0, 0, 9, 11, 0]
          ]
print("The MST edges and corresponding weights for this graph are")
g.primMST()
```

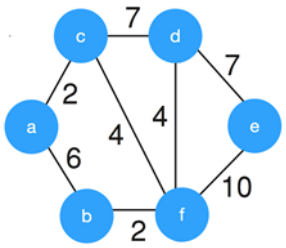
CYK. Which edge of the following graph will be selected first using Kruskal algorithm?



a DE
b BG
c GF
d BE

Correct answer: d

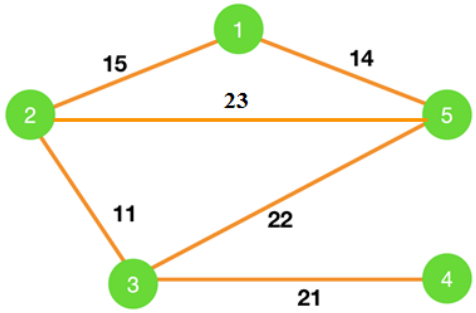
CYK. What is the weight of the MST for the following graph identified using Kruskal algorithm?



a 15
b 19
c 24
d 23

Correct answer: b

CYK. Which of the following edges form the MST for the graph below using Prim algorithm, starting from Vertex 4?



a (4-3)(3-5)(5-1)(1-2)
b (4-3)(5-3)(2-3)(1-2)
c (4-3)(3-2)(2-1)(1-5)
d (4-3)(3-5)(5-2)(1-5)

Correct answer: c