CS 218: Assignment 1

Ankit Kumar Misra	Devansh Jain	Harshit Varma	Richeek Das
190050020	190100044	190100055	190260036

February 1, 2021

Contents

Question 1

Objective: To analyze 9 given strategies (other than the standard earliest-finish-first algorithm) aimed at solving the interval-scheduling problem, and to determine which of them give correct optimal solutions. We first give counter-examples for those that are incorrect, and then we prove those that are correct.

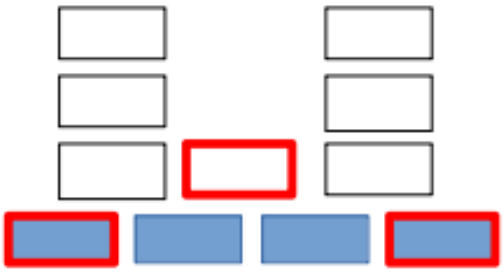
Of the algorithms (a)-(i), only (c), (h), and (i) are correct. The rest can be disproved using simple counterexamples, as shown below. The blue-filled boxes represent an optimal solution, and the red-outlined boxes represent a solution produced by the faulty algorithm.

(a) **Strategy**: Choose the course x that ends last, discard classes that conflict with x, and recurse.
Counterexample:

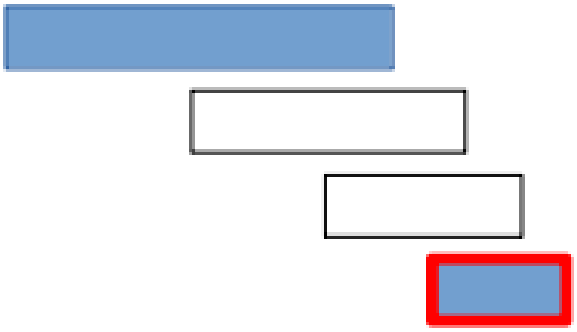
(b) **Strategy**: Choose the course x that starts first, discard all classes that conflict with x, and recurse.
Counterexample:

(d) **Strategy**: Choose the course x with shortest duration, discard all classes that conflict with x, and recurse.
Counterexample:

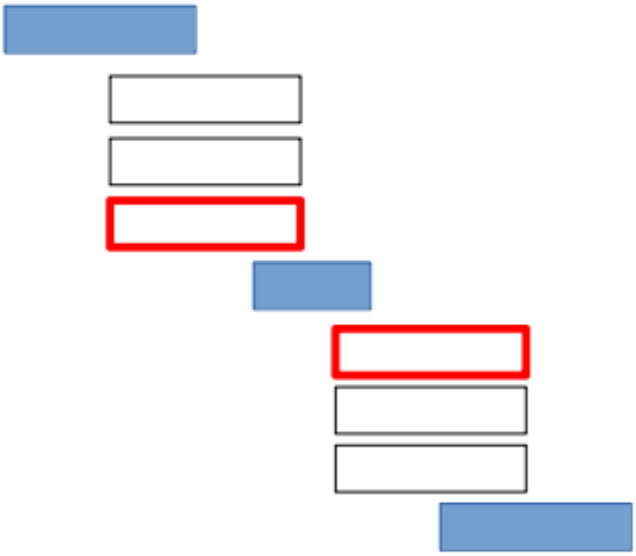
(e) **Strategy**: Choose a course x that conflicts with the fewest other courses, discard all classes that conflict with x, and recurse.
Counterexample:



(f) **Strategy**: If no classes conflict, choose them all. Otherwise, discard the course with longest duration and recurse.
Counterexample:



(g) **Strategy**: If no classes conflict, choose them all. Otherwise, discard a course that conflicts with the most other courses and recurse.
Counterexample:



In all the cases shown above, there is an optimal solution consisting of exactly one more interval than the number of intervals comprising the solution obtained from the corresponding algorithm/strategy. This

proves that they are all faulty.

Now we prove the correctness of the three remaining strategies, i.e., (c), (h), and (i).

(c) **Strategy**: Choose the course x that starts last, discard all classes that conflict with x, and recurse.
Proof: Let ALG be the list of intervals obtained by the above strategy, and OPT be the list of intervals forming an optimal solution. Let each list be indexed in the reverse order, i.e., the last interval is numbered 1, the second-last is numbered 2, and so on. So, we have:

$$\text{ALG} = [i_m, i_{m-1}, \ldots, i_2, i_1] \tag{1}$$
$$\text{OPT} = [j_n, j_{n-1}, \ldots, j_2, j_1] \tag{2}$$

Since OPT is optimal, we know that $|\text{OPT}| \geq |\text{ALG}|$. To prove that ALG also gives an optimal solution, we must show that $|\text{OPT}| = |\text{ALG}|$.

Claim: For all k such that $1 \leq k \leq m$, we have $s(i_k) \geq s(j_k)$.
This can be proved using induction on k.
Base case: $k = 1$. We have $s(i_1) \geq s(j_1)$, by the strategy's choice of i_1.
Inductive step: Suppose $s(i_k) \geq s(j_k)$ for some $k \geq 1$. Since intervals within OPT cannot overlap, we also have $f(j_{k+1}) < s(j_k)$. Thus, $f(j_{k+1}) < s(i_k)$, i.e., when ALG picks its $(k+1)$-th job (in reverse order), then j_{k+1} is available for it to choose. Our greedy algorithm is going to pick i_{k+1} as either j_{k+1} or something better. Thus, $s(i_{k+1}) \geq s(j_{k+1})$.

Clearly, the choices made by ALG at any step are going to be at least as good as those made by OPT. An interval selected by OPT can also be picked by ALG at any given time. Thus, ALG provides an optimal solution.

Note that this strategy is similar to the "earliest-finish-first" strategy, applied in the reverse direction.

(h) **Strategy**: Let x be the class with the earliest start time, and let y be the class with the second earliest start time.

- If x and y are disjoint, choose x and recurse on everything but x.
- If x completely contains y, discard x and recurse.
- Otherwise, discard y and recurse.

Proof: We prove that this strategy produces exactly the same results as the standard "earliest-finish-first" strategy discussed in class, which we know to be correct. Let x and y (as defined above) be the intervals (s_1, f_1) and (s_2, f_2). By definition, $s_1 \leq s_2$, and for any of the other intervals (s, f), we have $s_2 \leq s$. Consider each of the above cases:

- Case 1: x and y are disjoint. This means $s_1 < f_1 \leq s_2 < f_2$. Now $s_2 < f_2$, and $s_2 \leq s < f$ for any other intervals (s, f). Thus, $f_1 < f_2$, and $f_1 < f$ for other intervals (s, f), i.e., x is the interval having the earliest finish time. Choosing x and then recursing is consistent with the "earliest-finish-first" strategy.
- Case 2: x completely contains y. This means $s_1 \leq s_2 < f_2 \leq f_1$. In this case, the interval with the earliest finish time is either y itself, or some other interval (s, f) such that $s_2 \leq s$ (because y is defined this way) and $f \leq f_2$ (for it to have an earlier finish time than y). Either way, the interval having the earliest finish time is going to be contained within x, and therefore is going to conflict with x. Thus, discarding x is consistent with the "earliest-finish-first" strategy.

- Case 3: The only remaining case is $s_1 \leq s_2 \leq f_1 \leq f_2$. In this case, the interval with the earliest finish time is either x itself, or it's some other interval (s, f) such that $s_2 \leq s$ (because y is defined this way) and $f \leq f_1 \leq f_2$ (for it to have an earlier finish time than x). Either way, the interval having the earliest finish time is going to conflict with y. Therefore, discarding y is consistent with the "earliest-finish-first" strategy.

Thus, we conclude that this strategy produces the same results as our standard "earliest-finish-first" strategy, which proves that it is correct.

(i) **Strategy**: If any course x completely contains another course, discard x and recurse. Otherwise, choose the course y that ends last, discard all classes that conflict with y, and recurse.
Proof: Firstly, note that, if an interval x completely contains another interval y, and if any optimal solution contains x, then removing x and replacing it with y also gives an optimal solution. This is clear because any intervals not overlapping with x cannot overlap with y either, and because of the fact that at most one of x and y can be present in any solution (since they are overlapping).

Due to the above observation, the process of "discarding intervals x which completely contain some other interval" does not result in any losses, and we can do this first. If such an interval was going to be part of an optimal solution, then it can simply be replaced by an interval contained within it. If not, we can discard it anyway.

Once this has been done, we are left with a set of intervals where none is contained within another. This means the interval with the latest ending time is also the interval with the latest starting time! This is because, if the interval that ends the latest was (s, f), and the one that starts the latest was (s', f'), then we'd have $s \leq s' < f' \leq f$, and so the latter would be contained within the former, resulting in a contradiction. Thus, this strategy is equivalent to choosing the interval that starts the latest, discarding conflicts, and recursing.

We have already proved that the "latest-start-first" strategy is correct, in part (c). Therefore, this strategy is correct.

Question 2

Objective: Give an algorithm that designs an ordering of the answer sheets to be sent to the computer so that the overall correction time (i.e. the time by which both the parts of all the answer sheets are corrected) is minimised.

Let the set of n answer sheets be denoted by $A = \{A_1, A_2, \ldots, A_n\}$, the corresponding teachers be denoted by $T = \{T_1, T_2, \ldots, T_n\}$, and let C denote the computer.

Each teacher has *exactly* one answer sheet to correct, they can work independently of each other and can correct the sheet immediately after the sheet has passed through the computer (i.e Part-1 has been corrected)

Any ordering of the answer sheets requires *at least* $\sum_{i=1}^{n} t_{i,1}$ total time, since *all* answer sheets need to be corrected, and the computer can process the answer sheets only sequentially. Another way to see this is to set $t_{i,2} = 0\ \forall i$ and check the total time taken.

Thus, the major thing that affects the total time is the time taken by the teachers.
Since teachers can correct in parallel, intuitively, we need to utilize this parallelization effectively.

We will show that only (b) is optimal, and provide counter examples for the rest.

Strategy (b): *Sort the jobs in non-increasing order of* $t_{i,2}$

Definition: Let $u_O(i)$ denote the time taken, from the start, for the i^{th} answer sheet to be *completely* checked in a given ordering O. Then,

$$u_O(i) = \sum_{j=1}^{i} t_{j,1} + t_{i,2}$$

Because $\sum_{j=1}^{i} t_{j,1}$ is the time when the i^{th} answer sheet in an ordering exits the computer[1] and $t_{i,2}$ is the time taken by the corresponding teacher to correct it, immediately after it exits the computer.

Thus, the total time (τ_O) taken for an ordering O is:

$$\tau_O = \max(\{u_O(i)\}_{i=1}^{n})$$

Claim: Sorting the jobs in non-increasing order of $t_{i,2}$ is optimal
Proof: (Induction on n)

Base case: $n = 1$
Exactly one ordering exists, which is also optimal

Induction step:
Induction hypothesis: Our strategy yields an optimal ordering for $n = k$, $k \geq 1$

To prove: Our strategy yields an optimal ordering for $n = k + 1$
Proof: (Exchange Argument)

Let O be an optimal ordering for $n = k + 1$
Let A_m be an answer sheet in O that has the least $t_{i,2}$
Now, the set $O \backslash \{A_m\}$ has a cardinality of n
Let O_r be the ordering obtained by sorting the elements in $O \backslash (A_m)$ in non-increasing order of $t_{i,2}$
By the induction hypothesis, O_r is optimal.
Now, lets introduce A_m back into O_r

[1] Note that the computer works independently of the teachers

Claim: Introducing A_m at the end is optimal
Proof: (Via Contradiction)
For the sake of contradiction, assume:
Inserting A_m somewhere in the middle of O_r, but not at the end, (IIM Strategy) leads to strictly lesser total time than inserting at the end (IAE Strategy), i.e:

$$\max\left(\{u_{IIM}(i)\}_{i=1}^{n}\right) < \max\left(\{u_{IAE}(i)\}_{i=1}^{n}\right)$$

Let $[n] = \{1, 2, \ldots, n\}$ and let $t_{a,b}$ denote the time taken to correct the b^{th} part of the a^{th} answer sheet in the ordering given by IAE strategy, thus:

$$t_{i,2} \geq t_{j,2} \ \forall \ i \leq j$$

We will show that:

$$\forall i \in [n] \ \exists j \in [n] \ u_{IIM}(j) \geq u_{IAE}(i)$$

Let A_m be inserted just after the p^{th} position in O_r ($p \neq n-1$)
Proof: (By Cases)
Case 1: $1 \leq i \leq p$
$u_{IIM}(i) = u_{IAE}(i)$, thus $j = i$
Case 2: $p < i < n$
$u_{IIM}(i+1) \geq u_{IAE}(i)$, as $u_{IIM}(i+1) = u_{IAE}(i) + t_{n,1}$
Case 3: $i = n$
$u_{IIM}(n) = \sum_{i=1}^{n}(t_{i,1}) + t_{n-1,2}$ and $u_{IAE}(n) = \sum_{i=1}^{n}(t_{i,1}) + t_{n,2}$, as $t_{n-1,2} \geq t_{n,2}$, $u_{IIM}(n) \geq u_{IAE}(n)$, thus $j = i = n$

As $\forall i \in [n] \ \exists j \in [n] \ u_{IIM}(j) \geq u_{IAE}(i)$, $\max\left(\{u_{IIM}(i)\}_{i=1}^{n}\right) \geq \max\left(\{u_{IAE}(i)\}_{i=1}^{n}\right)$, thus a contradiction.

Therefore, after introducing A_m at the end, we end up with an ordering for $n = k+1$ elements that is consistent with the ordering given by our strategy.

Thus, by induction, our strategy is optimal.

Counterexamples:

(a)

Strategy: *Sort the jobs in non-increasing order of* $t_{i,1}$
Counterexample:
Let $n = 2$ and $A = \{(1,2),(2,1)\}$
Then the ordering suggested by the strategy will be (A_2, A_1)
This takes a total time of 5 units.

1. ($T = 0$) C takes 2 units to correct A_2 ($T = 2$)
2. ($T = 2$) C corrects A_1 and the T_2 corrects A_2 ($T = 3$)
3. ($T = 3$) T_1 corrects A_1 ($T = 5$)

(In these walkthroughs, T denotes the "global" time at the start of a step, and at the end of the step)
Whereas the optimal ordering will be (A_1, A_2)
This takes a total time of 4 units.

1. ($T = 0$) C takes 1 unit to correct A_1 ($T = 1$)
2. ($T = 1$) C corrects A_2 and the T_1 corrects A_1 ($T = 3$)
3. ($T = 3$) T_2 corrects A_2 ($T = 4$)

(c)

Strategy: *Sort the jobs in non-increasing order of* $t_{i,1} + t_{i,2}$

Counterexample:
Let $n = 2$ and $A = \{(1, 2), (3, 1)\}$

Then the ordering suggested by the strategy will be (A_2, A_1)
This takes a total time of 6 units.

1. $(T = 0)$ C takes 3 units to correct A_2 $(T = 3)$
2. $(T = 3)$ C starts correcting A_1 and the T_2 corrects A_2 $(T = 4)$
3. $(T = 4)$ T_1 corrects A_1 $(T = 6)$

Whereas the optimal ordering will be (A_1, A_2)
This takes a total time of 5 units.

1. $(T = 0)$ C takes 1 unit to correct A_1 $(T = 1)$
2. $(T = 1)$ C starts correcting A_2 and T_1 corrects A_1 $(T = 3)$
3. $(T = 3)$ C still requires 1 more unit to complete correcting A_2 $(T = 4)$
4. $(T = 4)$ T_2 corrects A_2 $(T = 5)$

(d)

Strategy: *Sort the jobs in non-decreasing order of* $t_{i,1}$

Counterexample:
Let $n = 2$ and $A = \{(1, 1), (2, 2)\}$

Then the ordering suggested by the strategy will be (A_1, A_2)
This takes a total time of 5 units.

1. $(T = 0)$ C takes 1 unit to correct A_1 $(T = 1)$
2. $(T = 1)$ C starts correcting A_2 and T_1 corrects A_1 $(T = 2)$
3. $(T = 2)$ C still requires 1 more unit to complete correcting A_2 $(T = 3)$
4. $(T = 3)$ T_2 corrects A_2 $(T = 5)$

Whereas the optimal ordering will be (A_2, A_1)
This takes a total time of 4 units.

1. $(T = 0)$ C takes 2 units to correct A_2 $(T = 2)$
2. $(T = 2)$ C corrects A_1 and T_2 starts correcting A_2 $(T = 3)$
3. $(T = 3)$ T_2 still requires 1 more unit to complete A_2 and T_1 corrects A_1 $(T = 4)$

(e)

Strategy: *Sort the jobs in non-decreasing order of* $t_{i,2}$

Counterexample: Same as strategy (d)

Question 3

Objective: There are n tasks where i-th task has an execution time t_i and deadline d_i . There are also precedence constraints amongst the tasks, specified by a directed acyclic graph having the tasks as vertices. If there is an edge from task i to task j, then task j cannot start before task i has been completed. Describe a polynomial-time algorithm to determine whether all tasks can be completed before their respective deadlines, on a single machine.

Algorithm: We can divide our algorithm into **two** parts:
Bottom-Up task scheduling based on the deadlines

1. Keep an empty **stack** which will store the order of execution.
2. Among the **leaf** tasks(tasks with no successors) find the task with **latest deadline**.
3. **Remove** the task and the edges associated with it and push it **into the stack**.
4. Repeat Step **2** and **3** until the stack has all the nodes or the precedence graph is empty.
5. The final filled stack shows the optimal execution order which is one of the best capable order for honouring the deadlines.

Execute tasks and check if every task can honour its deadline

1. Set a `flag` variable to **true**.
2. Execute the tasks one by one and keep on calculating the completion time of each task according to the given execution order in the stack.
3. If `completion time > deadline` for any task then set `flag=false` and **break**.
4. **If** flag is true after executing the loop in steps **2** and **3 then** all tasks can be completed before their respective deadlines **else** no.

Proof of correctness: We will state some **claims** and prove them -

Claim 1: Our algorithm honours the precedence graph

Proof: In the first part of our algorithm, we choose the task with the latest deadline from the set of tasks which have no successors. Because that is the complete set of tasks from which we can choose the one to be scheduled next(from the end).

This means either it was a task with least precedence or all the tasks with a lower precedence than it have already been scheduled to be executed after it(pushed into the stack).

Thus, we never execute some task with lower precedence before one with higher precedence.

Claim 2: Scheduling task with the latest deadline at the end, among the tasks available is an optimal strategy.

Proof: (Via Exchange Argument)
We can restate our claim as: "Any optimal strategy can be rearranged to form an ordering consistent with our strategy of keeping latest deadline at the end." Lets construct this:

Let there be two tasks: **Task1** with execution time t_1 and deadline d_1, **Task2** with execution time t_2 and deadline d_2. Let $d_2 > d_1$ and assume both have no successors(i.e they are available to be scheduled next). Let the time at the start of execution be s.

Latest Deadline End(LDE) Strategy: Execute Task2 after Task1.

Other Strategy: Execute Task1 after Task2.

Exchange Argument:

Let the schedule designed by the "Other Strategy" satisfy the deadlines. Then:

$$s + t_1 \leq d_2 \tag{1}$$
$$s + t_1 + t_2 \leq d_1 \tag{2}$$

The above two equations imply the following equations:

$$s + t_2 \leq d_1 \tag{3}$$
$$s + t_2 + t_1 \leq d_1 < d_2 \tag{4}$$

Therefore **If** "Other Strategy" satisfies the deadlines, **then** LDE Strategy satisfies the deadlines as well.

We can rewrite its contrapositive statement as well: **If** LDE Strategy fails to satisfy the deadlines, **then** any "Other Strategy" fails to satisfy the deadlines as well.

Therefore, we have shown that if any optimal solution exists, then we can rearrange it (moving later deadlines after early deadlines) without making things worse. Therefore our strategy of scheduling the tasks is optimal at every step. These two claims prove our greedy strategy for selecting an optimal schedule. Once we have the optimal schedule, execution and checking is pretty straightforward.

Pseudo Implementation and Complexity Analysis:

```
   Input: Adjacency list of the precedence DAG ⟶ adjlist
   Output: Boolean value stating if all the tasks can meet their deadlines
   /* Assume we have precedence graph as an adjacency list                    */
 1 stack schedule;
 2 maxheap heap ;                          // Max-heap based on the deadline parameter
 3
   /* This loop has a complexity of O(NlogN)                                  */
 4 for every leaf-task t do
 5 |  heap <-PUSH- t ;                    // Pushing into heap has a complexity of O(logN))
 6 end
 7
   /* Pick out one element every iteration, this loop runs for N iterations   */
 8 while heap is not empty do
 9 |  storetop <-STORE- heap<-POP-;
10 |  schedule <-PUSH- storetop;
11 |  adjlist <-REMOVE- storetop ; // Visit each parent and remove the task from adjlist O(N)
12 |
   |  /* The loop below will be discussed later                               */
13 |  for parent-tasks of storetop ⟶ pt do
14 |  |  if pt has no successor then
15 |  |  |  heap <-PUSH- pt;
16 |  |  end
17 |  end
18 end
19
   /* Straightforward O(N) implementation                                     */
20 for task_i in schedule do
21 |  completion-time+= t_i;
22 |  if completion-time>d_i then
23 |  |  <-RETURN FALSE-
24 |  end
25 end
26 <-RETURN TRUE-
```

Note: Even though the loop at line **13** seems to have a complexity of `O(NlogN)` at **every** iteration, it is good to note that we are just pushing N tasks into the heap in total. So the overall complexity of that loop at line **13** summed over all the iterations must be `O(NlogN)`.

Therefore, the overall complexity of the proposed pseudo implementation can be written as:

$$\text{Overall complexity} = \texttt{O(NlogN)}(\text{line 4}) + \texttt{O}(N^2)(\text{line 8}) + \texttt{O(NlogN)}(\text{line 13}) + \texttt{O(N)}(\text{line 20}) = \texttt{O}(N^2)$$

Question 4

(a)

Problem: Find a minimum weight subset of edges to be removed so that there are no cycles in G.
Algorithm:

1. Let the Graph be (V, E, W)
2. Find maximum weight present - say, maxw.
3. Replace all weights w by (maxw + 1 - w).
4. Find the Minimum Spanning Tree T = (V, E') using Kruskal's Algorithm or Prim's Algorithm [1]
5. E\E' is the required edge set

Time Analysis:
Finding maxw and updating all weights is O(E).
Prim's Algorithm has running time O((V+E)logV) (from slides).
We can find the complement of E' in O(E).
Overall complexity is O((V+E)logV).
Proof of correctness:
We have proved the correctness of Prim's Algorithm in class (using Cut property).
As we have reversed the ordering of edges based on weights by replacing w by (maxw + 1 - w) (we maintained the positive weight criterion), E' is actually is the subset of E representing the Maximum Spanning Tree.
What we require is to find the subset of E such that no cycles are present. We are given a undirected connected graph, so what we would essentially require is to form a spanning tree.
As we need to need the minimum weight subset that can be removed, we find the Maximum Spanning Tree and the complement of it would be the required subset.

(b)

Problem: Suppose R is a specified subset of the vertices of G. It is required to connect every vertex not in R to some vertex in R by a path. Determine the minimum weight subset of edges required to achieve this.
Algorithm and Implementation: Inspired from Kruskal's Algorithm

1. Let the Graph be (V, E, W). We are given $R \subset V$.
2. All vertices in $R' = V \backslash R$ are marked as white.
3. All edges in E are marked white if they join two vertices in R' otherwise black.
4. For every vertex in R', the edge to any vertex in R with minimum weight is marked as white.
5. Initialize an empty collection S of edges which represent the required subset of edges.
6. We form clusters of vertices in R', initially all of them are independent clusters, and then we merge them as we proceed. And this cluster has a unique edge (if exists) joining a vertex in R' with a vertex in R associated with it.

[1] Prim's algorithm runs faster in dense graphs. Kruskal's algorithm runs faster in sparse graphs.
Ref: `https://www.geeksforgeeks.org/difference-between-prims-and-kruskals-algorithm-for-mst/`

7. All the edges are sorted according to weights.
8. For every edge taken in ascending order, if it black, move ahead.
9. If the chosen edge is white, it can be of two types - joining two vertices in R' or joining a vertex in R with a vertex in R'.
10. Case 1: White edge $\{u, v\}$ where $u, v \in R'$
    (a) Both vertices are white.
    Let the clusters associated with u and v be V_u and V_v.
    Merge the clusters into one with associated edge as the one with lower weight of the edges associated with V_u and V_v. Mark the other one as black.
    Mark all edges between V_u and V_v as black.
    Add $\{u, v\}$ to the collection S.
    (b) One is black and other is white (w.l.o.g. u is black, v is white).
    Merge the clusters into one with associated edge as the edge associated with V_u. Mark the edge associated with V_v as black.
    Mark all edges between V_u and V_v as black.
    Mark all vertices in V_v as black.
    Add $\{u, v\}$ to the collection S.
11. Case 2: White edge $\{u, v\}$ where $u \in R', v \in R$
    (a) u is white.
    Let the cluster associated with u be V_u.
    Mark all vertices in V_u and $\{u, v\}$ as black.
    Add $\{u, v\}$ to the collection S.

Proof of correctness:

1. The algorithm would stop.
   As the number of edges are finite and we are iterating over them, the algorithm must end.
2. The edges marked as black won't be needed in the formation of the required subset.
   Edges between vertices of R are redundant as even without them, the condition is unaffected. So, they were marked as black.
   For every vertex in R', only the minimum weight edge from it to a vertex R is needed as all we need to do is connect it at least one of the vertex in R. So, others were initialized as black.
   Once two clusters are joined, we don't require any edges between them as they are all connected. So, the edges between them are marked as black.
   When a white cluster is merged with black cluster, we discard the edge associated with the white cluster as the vertices are now connected to some vertex in R via a path through the black cluster. So, it is also marked as black.
   There were no other operation which marked the edge as black.
3. As the graph is connected, no white cluster can remain, as it would have an edge either to some black cluster or to a vertex in R.

Time Analysis:
Initialization takes $O(|V| + |E|)$.
Sorting of edges - $O(|E|log|E|)$.
All edges are marked as black, exactly once - $O(|E|)$. —(1)

All vertices in R' are marked as black, exactly once - O($|V|$). —(2)
We merge clusters of vertices and copy the smaller of the cluster into the bigger one - O($|V|log|V|$).
As in every step, at least one of (1) or (2) occur - O($|E| + |V|$).
Overall complexity is O($|V|log|V| + |E|log|E|$).

(c)

Problem: Find the second minimum weight spanning tree in G. Assume the weights are distinct.
Algorithm: Inspired from Kruskal's Algorithm

1. Sort the edges by weight and find MST using Kruskal's Algorithm.
2. For each edge in the MST, remove it from the set and apply Kruskal's Algorithm on the remaining.
3. The new Spanning Tree has total weight increased by the difference of the weights of the edge removed and added.
4. Do step 2 for all edges in MST and take minimum of all.

Time Analysis:
Sorting the edges takes O($|E|log|E|$) time.
Kruskal's Algorithm takes O($|E|log|V|$) time using Disjoint Set Union [1] or O($|E|log|V| + |V|^2$) in simplest implementation.
It would take O(E) for applying Kruskal's Algorithm to finding the last edge.
This is repeated $|V| - 1$ times as MST contains V-1 edges.
Overall complexity is O($|E|log|E| + |E|log|V| + |V|^2 + |E| \cdot |V|$) = O($|E| \cdot |V|$).
($|E| \geq |V| - 1$ as the given graph is connected).
Proof of correctness:
Spanning trees have exactly $|V| - 1$ edges. So, second minimum spanning tree must have at least one edge that is not in the minimum spanning tree. If a second minimum spanning tree has exactly one edge, say (x, y), that is not in the minimum spanning tree, then it has the same set of edges as the minimum spanning tree, except that (x, y) replaces some edge, say (u, v), of the minimum spanning tree. In this case, $T' = (T - \{(u, v)\}) \cup \{(x, y)\}$.
Thus, all we need to show is that by replacing two or more edges of the minimum spanning tree, we cannot obtain a second minimum spanning tree.
Proof by contradiction:
Let T be the minimum spanning tree of G, and suppose that there exists a second minimum spanning tree T' that differs from T by two edges. There are at least two edges in edge set $T - T'$, and let (u, v) be the edge in $T - T'$ with minimum weight. If we were to add (u, v) to T', we would get a cycle c. This cycle contains some edge (x, y) in $T - T'$ (since otherwise, T would contain a cycle).
We claim that $w(x, y) > w(u, v)$. We prove this claim by contradiction, so let us assume that $w(x, y) < w(u, v)$. (edge weights are distinct, no need for equality).
If we add (x, y) to T , we get a cycle c', which contains some edge (u', v') in $T - T'$ (since otherwise, T' would contain a cycle). Therefore, the set of edges $T' = T - \{(u', v')\} \cup \{(x, y)\}$ forms a spanning tree, and we must also have $w(x, y) > w(u', v')$, since otherwise T'' would be a spanning tree with weight less than $w(T)$. Thus, $w(u', v') < w(x, y) < w(u, v)$, which contradicts our choice of (u, v) as the edge in $T - T'$ of minimum weight. Since the edges (u, v) and (x, y) would be on a common cycle c if we were to add (u, v) to T', the set of edges $(T' - \{(x, y)\}) \cup \{(u, v)\}$ is a spanning tree, and its weight is less than $w(T')$. Moreover, it differs from T (because it differs from T' by only one edge). Thus, we have formed a spanning

[1] Reference: `https://cp-algorithms.com/graph/mst_kruskal_with_dsu.html`

tree whose weight is less than $w(T')$ but is not $w(T)$. Hence, T' was not a second minimum spanning tree.

Now, that we have proven that the second minimum spanning tree has all edges from MST except one, we iterate through all possible combinations of removing an edge ($|V| - 1$) combinations, and find the minimum of them all.

More about it:
This is a fairly famous modification of MST problem. On searching about it, I came across several algorithms.
My above described algorithm is similar to the first algorithm described on `https://cp-algorithms.com/graph/second_best_mst.html`.
The algorithm with basic implementation has $O(|V| \cdot |E|)$ time complexity and it can be implemented by modelling it as Lowest Common Ancestor Problem and using those properties bringing the complexity down to $O(|E|log|V|)$.
Another implementation I found, quite similar to mine but using Prim's algorithm using Fibonacci Heap implementation of priority queue is described on `https://viterbi-web.usc.edu/~shanghua/teaching/Spring2010/public_html/files/HW2_Solutions_A.pdf`. This has time complexity of $O(|V|^2)$