Algorithmics SAT1 Project: Modularised algorithm design principles and weighted interval scheduling problems

Daniel Lawrence

June 1, 2025

Contents

Abstract

An online tutor wants to maximise her earnings for different online tutoring jobs that are available from her agency. Each available tutoring job has a start and finishing time and earnings based on the year level and subject of the student. Tutoring jobs are incompatible if they overlap. The goal for the tutor is to find the maximum earnings they can achieve on a given day of her availability of h hours with mutually compatible online tutoring jobs.

This report will fulfil the problem solving methodology to analyse the classic weighted interval scheduling problem in a real-world application, defining an appropriate model with ADTS, and applying the algorithmic design principles to formulate a justified pseudocode algorithm to solve the problem. All topics covered in this assignment conform with the VCE Algorithmics 3/4 Study Design for all Area of Studies in Unit 3, adhering to and mentioning the outcomes for each chapter of the report.

1 Problem specification and modelling

Outcome 1: Define a flexible model to hold the data information using simple ADTs and combination ADTs from the Algorithmics course for holding all the relevant information and features of the online tutor real world problem. Consider the features that will be necessary for solving this problem that can accept any tutoring jobs available for any given hours 'h' of the tutor's time allocation to this work and provide the tutor with a personalised timetable of tutoring jobs to take to maximise their earnings.

Outcome 2: Justify the selection of each simple and combination ADT you have selected in part A) for the information model you create and describe how the features of the real-world problem map to your data model.

Outcome 3: In pseudocode create a structured module that accepts as input parameters the jobs available using the data structures you have defined and the h hours that the tutor has allocated to do this work that will be used to set up the information in an Algorithm. Your pseudocode must show all the actions required to create your entire information model as you have defined in part A) according to the signatures of the simple and combination ADTs that you have selected.

1.1 Defining a model with ADTs

In the situation of the tutor working for an online tutoring agency, the problem of figuring out the optimal weighted interval scheduling problem comes down to the materials that the agency gives the tutor. In this case, we shall assume we are given a schedule highlighting starting times, ending times, the year level and the subject of the tutoring job being offered. This can first be visually represented using a simplified schedule (Figure 1) which has a pre-calculated pricing as well as ordering based on ID and not starting or finishing times, since both of these steps will be executed in the initialisation stage:

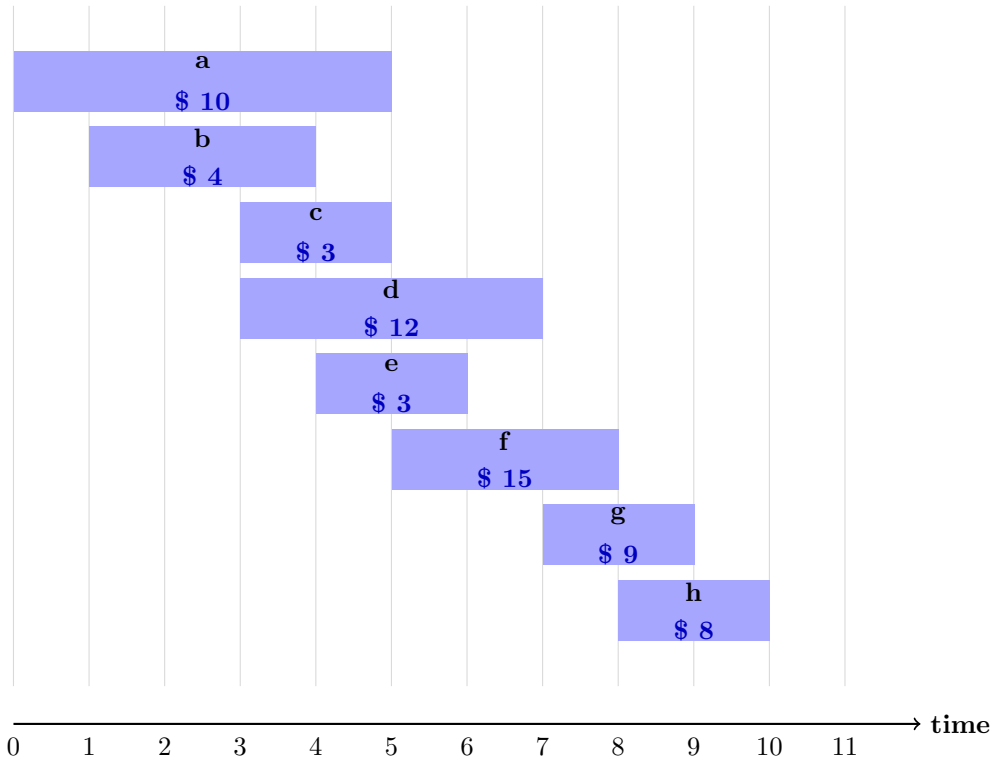

Figure 1: Visual model representation of the job timelines with pay.

To further abstract this current model for suitable usage in the algorithm, we convert each job in the timeline into a dictionary with 4 keys (start, end, year, subject) which each represent starting time, ending time, year level and subject of the tutoring job respectively. A list is then used to contain each dictionary into one data type, combining the usage of lists and dictionaries to model the data from the job schedules. We can ignore the ID of the job since our data model uses the index of each item as the list in lieu of an ID, and the cost parameter isn't included since that will be calculated using a linear model.

1.2 Justification of ADTs

For the data modelling stage, multiple combinations of ADTs were considered such as usage of a graph with its relevant ADTS to formula a solution using graph algorithms. The main problem with this approach arises when trying to implement the pseudocode into other programming languages, since it'd be harder to implement a graph structure than a normal string-based ADT into a coding language like Python3, making

this selection of ADTs simplistic whilst still adhering to the criterion of the problem.

The ADT's key signatures for the dictionary ADT is shown below:

`name`	*dictionary*
`import`	*key, value, integer, boolean, list*
`operations`	
	`newDict`: *dictionary*
	`insertDict`: *dictionary* $\times$ *key* $\times$ *value* $\rightarrow$ *dictionary*
	`inDict`: *dictionary* $\times$ *key* $\rightarrow$ *boolean*
	`removeDict`: *dictionary* $\times$ *key* $\rightarrow$ *dictionary*
	`lookupDict`: *dictionary* $\times$ *key* $\rightarrow$ *value*
	`updateDict`: *dictionary* $\times$ *key* $\times$ *value* $\rightarrow$ *dictionary*
	`lengthDict`: *dictionary* $\rightarrow$ *integer*
	`keysDict`: *dictionary* $\rightarrow$ *list*

The dictionary has been chosen to hold the majority of information regarding the job since each key and value pair can be easily represented when inputting parameters, similar to a non-abstracted job schedule. The only problem with this ADT is that its harder to implement an algorithm that works with dictionaries instead of a list, so a dictionary to list conversion is necessary for a coherent algorithm to work properly.

The ADT's key signatures for the list ADT is shown below:

`name`	*list*
`import`	*element, boolean, position, integer*
`operations`	
	`newList`: $\rightarrow$ `list`
	`first`: *list* $\rightarrow$ *element*
	`last`: *list* $\rightarrow$ *element*
	`rest`: *list* $\rightarrow$ *list*
	`prepend`: *list* $\times$ *element* $\rightarrow$ *list*
	`contains`: *list* $\times$ *element* $\rightarrow$ *boolean*
	`append`: *list* $\times$ *element* $\rightarrow$ *list*
	`isEmpty`: *list* $\rightarrow$ *boolean*
	`delete`: *list* $\times$ *position* $\rightarrow$ *list*
	`length`: *list* $\rightarrow$ *integer*

The list has been chosen to envelop all of the job cells holding their information into one data type, so that information can be easily accessed using indexing. This indexing also allows us to classify a job's ID by simply using the index of the list. This type of ADT is prefered over something like a set because values in a list can be changed and edited to better suit the problem's conditions, such as removing jobs that are outside of the range of h hours.

1.3 Structured pseudocode module

To correctly initialise the starting conditions of the input parameters to correctly solve the weighted interval scheduling problem in the main algorithm, modular functions must be employed to correctly edit the input list of dictionaries to suit the problem's needs. Below is the structured initialisation algorithm in pseudocode deploying usage of abstraction and modularisation:

Algorithm 1 JobsInitialisation

Input: A list of dictionaries *jobs* where each index represents a dictionary cell with certain job info, and an integer h which determines the available jobs whose ending times are within the range $[0, h]$.
Output: A list of lists *init_jobs* which is abstracted and simplified to fit the main algorithm's criterion.

```
function DICT_TO_LIST(jobs)                          ▷ Converts a list of dictionaries to a list of lists
    jobs_list ← [ ]
    for all dict in jobs do
        job_info ← [ ]
        for all key in dict do
            append dict[key] to job_info             ▷ Extracts the value from the dictionary key
        end for
        append job_info to jobs_list
    end for
    return jobs_list
end function

function TIME_PERIOD(jobs, h)                        ▷ Removes all jobs that aren't within the h hour range
    for all j in jobs do
        if j[end] > h then
            remove j from jobs                       ▷ Deletion method to simplify job selection
        end if
    end for
    return jobs
end function

function CALCULATE_PAY(jobs)                         ▷ Uses the year and subject variables to calculate the pay
    for all j in jobs do
        year ← j[year]
        subject ← j[subject]
        year_value ← ⌈year/3⌉                        ▷ Accepted year levels: year ∈ ℕ ∩ [6, 12]
        subject_dict ← {"maths", "english": 3, "science", "technology": 2, "humanities", "music": 1}
        subject_value ← subject_dict[subject]        ▷ Accepted subjects: seen in subject_dict
        hourly_rate ← 30 + 5 × (year_value + subject_value)   ▷ Linear model for calculating pay
        total_pay ← hourly_rate × (j[end] − j[start])
        remove subject, year from j
        j[pay] ← total_pay
    end for
    return jobs
end function

subset_jobs ← TIME_PERIOD(jobs, h)
calculated_jobs ← CALCULATE_PAY(subset_jobs)
list_jobs ← DICT_TO_LIST(calculated_jobs)
init_jobs ← list_jobs sorted by EFT                  ▷ Sorts the converted list by eearliest finishing time
return init_jobs
```

2 Algorithm design for real-world problem

Outcome 1: Discuss and compare some algorithmic approaches that will be suitable for designing an algorithm for solving this problem using the information model that you have defined in part A). Consider and discuss the design patterns and algorithms that have been studied in Unit 3 and their applicability to this particular problem for finding the best solution.

2.1 Algorithmic design approaches

To fully explore the rel-world application of the WIS problem, several algorithm design approaches must be compared to determine the most appropriate solution. The brute force method provides a foundational baseline, as it systematically evaluates every possible subset of jobs to identify the optimal selection. While this guarantees that no potential solution is overlooked, it becomes impractical where some sub-problem branches are clearly redundant. The greedy algorithm, which always selects the job with the highest immediate value, offers a more intuitive and straightforward strategy. However, for weighted interval scheduling, this approach is unsuitable, as it can miss combinations of jobs that collectively produce a higher total value, demonstrating that greedy selection fails to account for the cumulative impact of job choices over the entire schedule. To address these limitations, the decrease and conquer strategy can be considered. This approach involves reducing the sample size of the main problem by removing jobs that are incompatible or clearly suboptimal, then recursively solving the reduced problem.

We can also combine the recursion with memoization (using an array to store values). This unique method begins by sorting the jobs by their finish times, and then for each job, the algorithm considers two possibilities: including the job or excluding it. By recursively applying this decision process and storing the results of previously solved sub-problems in an array, the algorithm avoids redundant calculations and ensures that every relevant combination is considered.

NOTE: This also incorporates elements of a divide and conquer algorithmic approach, however for the purpose of this report we can primarily classify the algorithm to be decrease and conquer to stick within the bounds of the topic's exploration.

To reconstruct the actual set of jobs that make up the optimal solution, backtracking is used after the main computation. This involves tracing the decisions made during the recursive process to explicitly identify which jobs were included. The integration of sorting, recursion, memoization, and backtracking exemplifies a sophisticated combination of algorithm design patterns, tailored specifically to the weighted interval scheduling problem and adhering to the real-world scenario.

3 Algorithmic solution for real-world problem

Outcome 1: Create a modular algorithm in structured pseudocode that has meaningful variable names and comments to solve the online tutor problem using the design structures and graph algorithms where appropriate that have been identified in Stage 1. Your algorithm needs to accept any given hours h of the tutor's time allocation and consider all the available tutoring jobs they can take and provide the tutor with a personalised timetable of tutoring jobs to take to maximise their earnings.

Outcome 2: For the entire algorithm that you have created in part A) create a call graph showing how all the modules call each other within your algorithm.

3.1 Structured pseudocode algorithm

Continuing on from the initialisation module above in Section 1 where the given output is a list of lists, we can use a modularised algorithm to find the maximal pay the tutor can earn for the real-world scenario, and also give a 'schedule' outlining which jobs to take in order to achieve the maximal earnings. The algorithm shown below achieves said tasks, implementing the output from the initialisation:

Algorithm 2 WeightedIntervalScheduling

Input: A list of lists *jobs* where each index represents a list cell with certain job info, and an integer j which is the index targeting a specific job in *jobs*.

Output: An integer index of an array $max[j]$ that has the maximal income up to a specific job, and a list *schedule* which puts the best jobs to take in order based on their index.

```
function OPTIMAL_JOB(j)        ▷ Recursive decrease and conquer function to calculate the maximal pay
    if j == 0 then
        return 0                                   ▷ Checking for the base case and returning 0
    else if max[j] ≠ −1 then
        return max[j]                  ▷ Returns the value if it is already defined in the array
    else
        max[j] ← max(jobs[j][2] + OPTIMAL_JOB(NEXT_JOB(j)), OPTIMAL_JOB(j − 1))
        return max[j]          ▷ Returning the array value after undergoing the recursive step
    end if
end function

function SEARCH_SOLUTION(j)                ▷ Backtracking to find the optimal jobs j to take
    schedule ← [ ]                          ▷ Initialising an empty list to hold the schedule
    if j > 0 then
        if jobs[j][2] + max[p(j)] ≥ max[j-1] then
            append jobs[j] to schedule                ▷ Add to the list upon meeting condition
            SEARCH_SOLUTION(NEXT_JOB(j))
        else
            SEARCH_SOLUTION(j − 1)
        end if
    end if
    return schedule                          ▷ Outputs final schedule after recursion terminates
end function

function NEXT_JOB(j)              ▷ Bisection method iterative function to find next available job
    low ← 1
    high ← j − 1
    result ← 0
    while low ≤ high do
        mid ← ⌊(low+high)/2⌋
        if jobs[mid][1] ≤ jobs[j][0] then      ▷ Checks if the jobs overlap to find the next starting point
            result ← mid
            low ← mid + 1
        else
            high ← mid - 1
        end if
    end while
    return result
end function

max ← [-1] × length of jobs          ▷ Initialising array with the smallest values of -1; global variable
return OPTIMAL_JOB(jobs[-1])                  ▷ Outputs the maximal profit of the chosen jobs
return SEARCH_SOLUTION(jobs[-1])                ▷ Schedule of optimal chosen jobs to take
```

3.2 Modular call graph

The following call graph represents the simplified function calls and dependencies in the Weighted Interval Scheduling algorithm after an initial iteration:

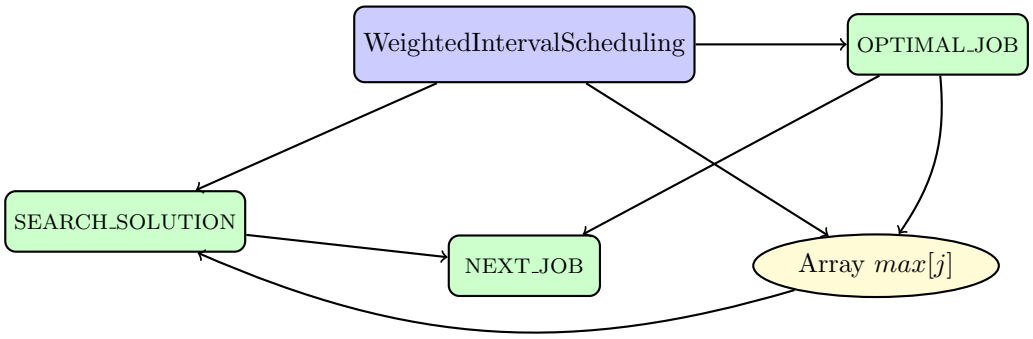

Figure 2: Simplified function call graph of the $WeightedIntervalScheduling$ algorithm.

In the modular call graph above (Figure 2), blue nodes represent the primary algorithm, green nodes indicate helper functions within in the main algorithm, and yellow nodes represent ADTs implemented and used throughout the algorithm.

NOTE: Although ADTs aren't typically mentioned in 'function' call graphs, I felt it was important to highlight how they were being interacted with each helper function, to better visualise the processes of the algorithm and help demonstrate it in the context of the real-world scenario.

When the algorithm is first run, it initialises the *max* array, hence why an arrow indicates the main node interacting with the ADT node. Then the first helper function, OPTIMAL_JOB is called. Upon calling this function, it repeatedly updates the *max* array as well as call the NEXT_JOB function to calculate the appropriate sub-problem size to work on in its next recursive call, justifying the two arrows coming out of the first helper node into the ADT node and the centre helper node.

After OPTIMAL_JOB has finished running and a maximal output has been printed, the next helper function SEARCH_SOLUTION is called, using similar recursive properties to the first helper function. However, since this function doesn't update the *max* array and only extracts values from it, an arrow is instead drawn from the ADT node to the helper node. Thus the only arrow going outward from SEARCH_SOLUTION is into the last helper function which is NEXT_JOB, serving as the foundation for the other two functions.

4 Justification of solution

Outcome 1: In a discussion format provide a justification for the suitability of the algorithm that you have created in Stage 2 part A) demonstrating its suitability by use of a few examples, its coherence by highlighting its readability and clarity and clear communication of the algorithm to the reader and the overall effectiveness of the algorithm for solving the problem.

4.1 Suitability of algorithm

The proposed $WeightedIntervalScheduling$ algorithm provides a comprehensive solution to the weighted interval scheduling problem through a recursive decrease and conquer approach that demonstrates clear understanding of the optimisation challenge. This algorithm is highly suitable for addressing real-world scheduling scenarios where maximising profit while avoiding temporal conflicts is essential, as well as being without the tutor's h hour schedule.

WeightedIntervalScheduling effectively tackles the fundamental decision-making aspect of weighted interval scheduling: determining whether to include or exclude each job to achieve maximum total profit. The recursive formulation (Figure 3) demonstrates this decision process by comparing the profit of including the current job (plus optimal profit from subsequent non-overlapping jobs) against the optimal profit achieved by excluding it entirely. This approach demonstrates coherence between problem analysis and algorithmic design, as well as aspects which help simplify the problem effectively:

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(p(j)),\ OPT(j-1)\} & \text{otherwise} \end{cases}$$

Figure 3: The original recursive formula for solving the WIS problem.

NOTE: This recursive formula uses different variable names because this is the standard accepted for most solutions that tackle the WIS problem, however I changed the variable names to make it more applicable to the real-world scenario in the pseudocode modules.

To show the proof of correctness for the algorithm and thus showing that the algorithm would provide an optimal response for all inputs of the *jobs* data type, we must primarily convey the proof of optimality of the main recursive function OPT(j) which is used in this algorithm as well as most WIS solution implementations. This proof is conveyed using the mathematical principle of strong induction, conveying the optimality of the *WeightedIntervalScheduling* algorithm:

- The proposition P(j): That $OPT(j)$ returns the optimal solution up to the j^{th} job.
- The base case P(0): Where we check the optimal solution for $j = 0$.
  - Given that the recursive formula for $OPT(j)$ is given by:

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(p(j)),\ OPT(j-1)\} & \text{otherwise} \end{cases}$$

  - We can see that when $j = 0$, the formula returns 0 which is correct.
- The assumption P(k) where $0 \leq k \leq j$: Where $OPT(k)$ is assumed to return the optimal value for every value of k from 0 up to j.
- The inductive step P(k+1): Showing that $OPT(k+1)$ works when $OPT(k)$ works.
  - If $OPT(k)$ is optimal where $k > 0$, it either returns $v_k + OPT(p(k))$ or $OPT(k-1)$.
  - **Case 1:** Where $OPT(k+1) = v_{k+1} + OPT(p(k+1))$
    - p(k+1) must return a value between 0 and k, hence by strong induction we know OPT(p(k+1)) must be optimal.
    - Since $OPT(k+1)$ is the only remaining compatible job, adding its weight v_{k+1} will complete the optimal solution for $OPT(k+1)$.
  - **Case 2:** Where $OPT(k+1) = OPT((k+1)-1)$
    - $OPT(k+1)$ would return $OPT(k)$ which we already know is optimal.
- Hence by the principle of strong mathematical induction, the proposition P(j) is proven true, hence proving the optimality of the recursive formula $OPT(j)$.

Following the proof of optimality as seen in the visual representation of the structured pseudocode algorithm (*WeightedIntervalScheduling*), the usage of comments have been deployed to show the major purpose of each helper function, enhancing its readability and usability when debugging or implementing into another programming language. Additionally, simple variable names have been utilised to avoid any confusion between what each ADT does within the program. All of these aspects of the algorithm help convey an effective solution to the Weighted Interval Scheduling problem.