Slovak Universiry of Technology in Bratislava
Faculty of Informatics and Information Technologies

Analysis and Complexity of Algorithms

Practical Assignment

Implementation of Scheduling with Deadlines, Scheduling with Deadlines using Disjoint Set Data Structure III, Greedy approach to assignment n jobs to n persons and Hungarian algorithm to same problem.

Dávid Truhlář ID: 120897 xtruhlar@stuba.sk

Table of Contents

The assignment

Task 1: Scheduling with Deadlines

Design and implement an algorithm to solve a problem for scheduling with deadlines.

Consider the following jobs, deadlines, and profits, and use the scheduling with deadlines algorithm to maximize the total profit. Show the optimal sequence of jobs with max profit on the screen.

Job	Deadline	Profit
1	2	40
2	4	15
3	3	60
4	2	20
5	3	10
6	1	45
7	1	55

Table 1 Jobs, Deadlines and Profits table

Task 2: Scheduling with Deadlines using Disjoint Set Data Structure III

Consider the procedure schedule in the Scheduling with Deadlines algorithm (Algorithm 4.4). Let d be the maximum of the deadlines for n jobs. Modify the procedure so that it adds a job as late as possible to the schedule being built, but no later than its deadline. Do this by initializing d+1 disjoint sets, containing the integers 0, 1, ..., d. Let small(S) be the smallest member of set S. When a job is scheduled, find the set S containing the minimum of its deadline and n. If small(S) = 0, reject the job. Otherwise, schedule it at time small(S), merge S with the set containing small(S)-1. Assuming we use Disjoint Set Data Structure III in Appendix C, show that this version is $\theta(n \lg m)$, where m is the minimum of d and n.

Task 3: Greedy approach and Hungarian algorithm to assignment n jobs to n persons.

Suppose we assign n persons to n jobs. Let C_{ij} be the cost of assigning the i^{th} person to the j^{th} job.

a) Use a greedy approach to write an algorithm that finds an assignment that minimizes the total cost of assigning all n persons to all n jobs. Analyze your algorithm and show the results using order notation.
b) Use the Hungarian algorithm for the same problem. Analyze your algorithm and show the results using order notation.

For illustration, consider the following matrix of costs:

	Job1	Job2	Job3
Person 1	10	5	5
Person 2	2	4	10
Person 3	5	1	7

Table 2 Jobs table

Marking table

1. Scheduling with deadline algorithm implementation	3
Correct output on a screen based on input from the Table 1	1
2. Algorithm Modification – code description	2
Use of Disjoint Set Data Structure III	3
Implementation of the modification	6
Correct output on a screen based on input from the Table 1	1
Modified algorithm analysis	4
3. Use a Greedy approach to design and implement the algorithm	3
Use Hungarian algorithm to solve same problem	4
Complexity analysis and comparison of both approaches	4

Implementation

Task 1: Scheduling with Deadlines

Problem

- Identify the optimal schedule with the highest total profit, considering that each job yields a profit only when scheduled by its respective deadline.

Inputs

- n: The total number of jobs.
- The vector of Job structures containing Job ID, Deadline and Profit.
- The vector of integers, which stores the optimal sequence.

Outputs

- The sequence of jobs with maximal profit.
- The maximal profit.

Scheduling with deadline algorithm implementation

To address the given problem, the core logic of the program is grounded in the pseudocode provided in the assignment. An integral addition to the program is the incorporation of the `isFeasible()` function. This function, taking two arguments, `Job &a` and `Job &b`, evaluates and returns the superior of the two.

The `isFeasible()` function plays a crucial role in the *C++* sort function, enabling the sorting of jobs based on their deadlines and profits in accordance with the defined criteria.

When it comes to complexity analysis the complexity of `sort()` implemented in *C++* has to be addressed as well. In *C++*, the `sort()` function is a powerful tool for sorting elements within a vector or array that supports random access. Typically, it takes two parameters: the starting point of the array or vector where sorting should commence and the length up to which the array or vector should be sorted. Optionally, a third parameter can be employed, especially when sorting elements lexicographically or according to a specific criterion. The default behavior of `sort()` is to arrange elements in ascending order.

The time complexity of the `sort()` function is **O(n log(n))**, making it an efficient sorting algorithm. If the desire is to sort elements in descending order, the function can be customized by providing a third parameter, such as using the `isFeasible()` function, which compares elements in a way that places more feasible elements before others. (1)

Correct output on a screen based on input from the Table 1 Jobs, Deadlines and Profits table**Table 1**:

Within the `main()` function, a vector is defined, drawing its values from Table 1.

```
vector<Job> jobs = {{1, 2, 40},{2, 4, 15},{3, 3, 60},{4, 2, 20},{5, 3, 10},{6, 1, 45},{7, 1, 55}};
```

Subsequently, the desired output is generated and displayed in the console.

```
Optimal sequence of jobs:
7 1 3 2
Maximum Profit:
170
```

Task 2: Scheduling with Deadlines using Disjoint Set Data Structure III

Problem

- Modify the scheduling algorithm (Algorithm 4.4) for Scheduling with Deadlines to add a job as late as possible to the schedule being built, but no later than its deadline. Implement Disjoint Data Structure III from Appendix C (2).

Inputs

- n, the number of jobs
- The vector of Job structures containing Job ID, Deadline and Profit.
- The vector of integers, which stores the optimal sequence.

Outputs

- An optimal sequence J for the jobs.
- The maximal profit.

Algorithm Modification – code description

The primary change involves scheduling jobs as late as possible, but no later than their respective deadlines. This modification aims to maximize the total profit by considering the deadline constraints for each task. The code is structured to handle the initialization of sets, finding representative sets with path compression, merging sets based on their depths, and determining the smallest element in a set. Additionally, the algorithm sorts jobs based on their deadlines and profits to optimize the scheduling process.

Use of Disjoint Set Data Structure III

Initialization of Sets:
Function `void initializeSet()`, initialize a set for given element by setting its parent pointer to itself, depth to 0, and smallest element to the element itself.
Finding Representative Sets with Path Compression:
`SetPointer findSet()` finds the representative set of an element while applying path compression. It ensures that the parent pointers along the path are updated to point directly to the representative set, optimizing future find operations.
Merging Sets Based on Depths:
Function `void mergeSets()`, merges two sets based on their depths, ensuring that the set with a smaller smallest element becomes the representative set. The depths of the sets are updated accordingly.
Getting the smallest element in a set:
`int smallestElement()` is responsible for retrieving the smallest element in a given set. It utilizes the smallest information stored in the set structure within the Disjoint Set Data Structure III.

Implementation of the modification

The implementation of the modification is in the file **task2.cpp**. The file structure is organized to handle all aspects of the modification, such as initializing sets, finding representative set, merging sets, identifying the smallest element, and optimizing the schedule by sorting jobs based on their deadlines and profits.

Correct output on a screen based on input from the Table 1 Jobs, Deadlines and Profits table:

The program utilizes the same input data as provided in Table 1 for Task 1: Scheduling with Deadlines. The `vector<Job> jobs` is shared between task1 and task2, ensuring consistency in the dataset.

```
Optimal sequence of jobs:
7 1 3 2
Maximum Profit:
170
```

Modified algorithm analysis

The initialization of disjoint sets involves a loop for each element, resulting in a linear time complexity of **O(n)**. Sorting tasks based on completion deadlines employs the sort function from the <algorithm> library, with a best-known average-case time complexity of **O(n log n),** where n is the number of tasks. The traversal of tasks, occurring in a loop, has a time complexity of **O(n). Overall, the program's time complexity can be approximated as O(n log n).**

Task 3: Greedy approach and Hungarian algorithm to assign n jobs to n persons.

Problem

- Implement a greedy algorithm and Hungarian algorithm to efficiently allocate n individuals to n job positions, considering C_{ij} as the cost associated with assigning the i^{th} person to the j^{th} job.

Inputs

- The matrix of costs as, representing the cost of assigning each person to each job.

Outputs

- Optimal pairs of jobs and persons.
- Total cost

Greedy approach

To address the given problem using the greedy approach algorithm, various segments of the program were implemented.

Within the `greedyApproach()` function, taking a parameter `of vector<vector<int>>& costs`, the first step involves calculating the value of 'n' based on the size of the cost matrix. Subsequently, the variable `totalCost` is initialized to zero, and a `vector<int> assignedJobs(n, -1)` is created to store pairs of jobs and persons.

The algorithm then iterates through each person, evaluating the following condition for each job:

```
if (costs[P][J] < minCost && assignedJobs[J] == -1) {
```

When this condition holds true, it indicates that the current job has the lowest cost, and it has not been assigned to another person. After checking all jobs for the current person, the `minCost` is added to `totalCost`, and the current person is assigned to the job with the minimum cost in the `assignedJobs` vector.

Finally, the pairs in `assignedJobs` are printed to the console, along with the total cost of this combination.

Output based on Table 2

```
Job     Person  Cost
--------------------
 J1       P2      2
 J2       P1      5
 J3       P3      7
--------------------
   Total Cost: 14
```

Hungarian Algorithm

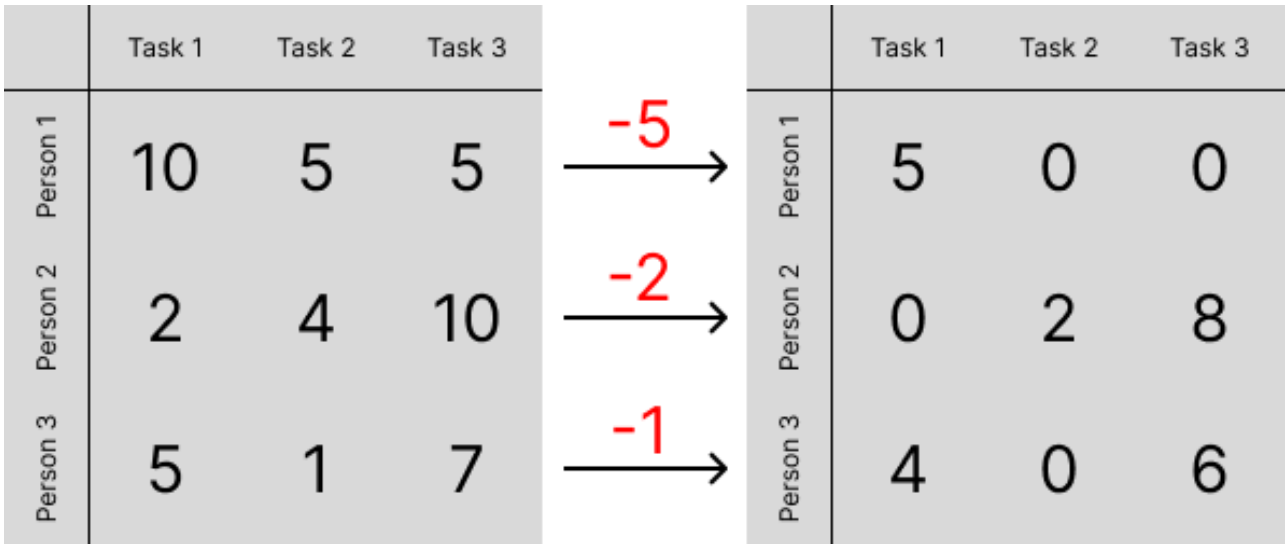
The Hungarian algorithm is an allocation algorithm, that is commonly used to solve our problem. There are exact steps that should be follow to achieve the wished result – optimal allocation of tasks to persons.

In preparation for the implementation of this algorithm, a crucial source is a YouTube video (3), which I meticulously followed, applying each recommended rule and step.

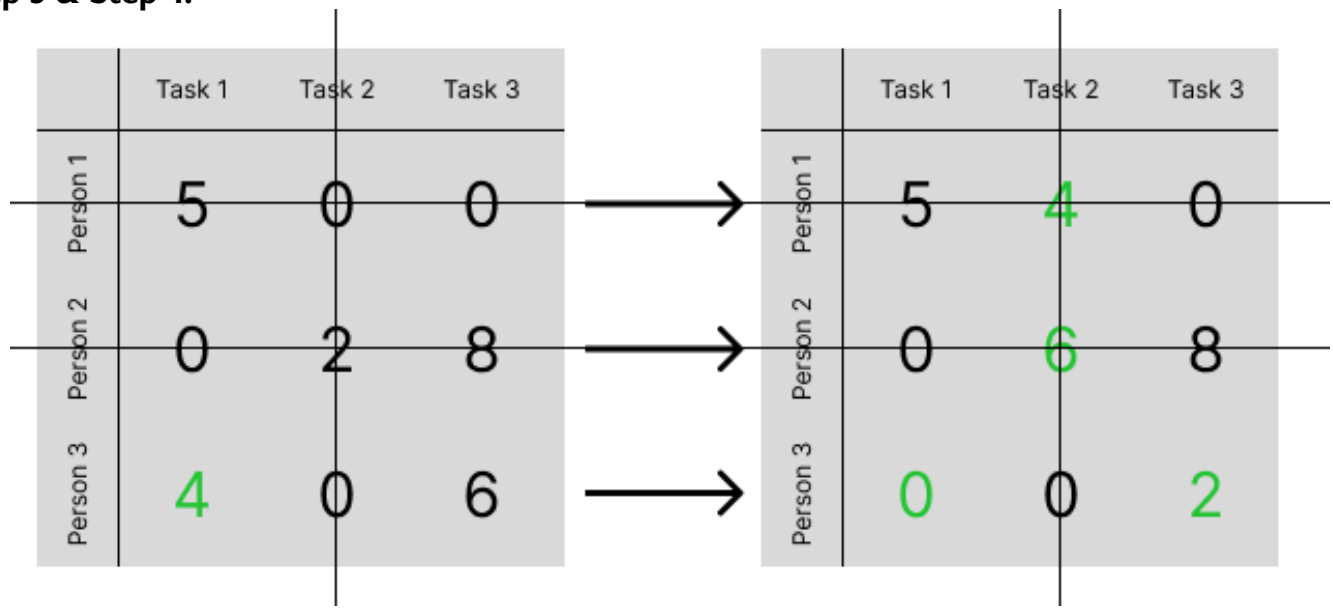
The algorithm steps:

1. Find the minimal cost in each row and subtract it from each task cost.
2. If there is no zero in the column, repeat the process for each column.
3. Use minimal horizontal and vertical lines to cross all zeros.
4. Identify the minimal cost that is not crossed, subtract this number from every uncrossed tile. If tiles are crossed with multiple lines, add this number to the tile.
5. Count all zeros in each row. If there is only one zero, use that Cij as the assigned task.
6. Cross out all zeros in the given column.
7. Repeat the process until n jobs are assigned.

Step 1 & Step2:

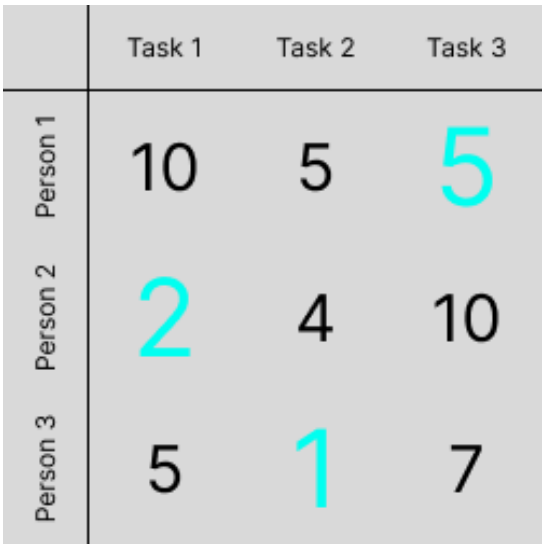


Step 3 & Step 4:



Step 5 & Step 6:

Step 7: - Result



Complexity analysis and comparison of both approaches

A critical aspect of this assignment is the time complexity analysis of both approaches. Beginning with the greedy approach, the program can be divided into the following parts:

a) Initialization of assignedJobs has a time complexity of O(n).

b) The outer loop runs n – 1 times for each person, and the inner loop runs n – 1 times for each job. Therefore, the time complexity is $O(n^2)$.

c) In the inner loop, there are n comparisons, resulting in a time complexity of O(n).

d) Finally, a for loop is used to print the assignedJobs, which has a time complexity of O(n).

This implies that the overall time complexity of the program is $O(n^2)$.

The time complexity of the Hungarian algorithm can be divided into seven steps for better understanding:

a) Step 1: The entire matrix needs to be checked, involving two nested for loops that each run n – 1 times. The time complexity is $O(n^2)$.

b) Step 2: Similar to Step 1, each column is checked, resulting in a time complexity of $O(n^2)$.

c) Step 3: The entire matrix is searched to find zeros, contributing to a time complexity of $O(n^2)$.

d) Step 4: The program includes a while loop in which there are nested loops, each running n – 1 times. In the worst-case scenario, the while loop itself can run n times. The time complexity is $O(n^3)$.

e) Step 5: More complex functions are implemented to find zeros. For each row, the number of zeros is calculated $O(n^2)$, and the matrix is sorted to prioritize rows with one zero at the top $O(n^2)$. Additionally, mapping the original matrix is performed to retain correct indexes $O(n^2)$.

f) Step 6: To assign the correct task to persons, nested loops are used, involving three nested for loops $O(n^3)$.

g) Step 7: Mapping the correct assigned task involves using the map of the original matrix O(n). and printing the correct solution into the console utilizes a loop that runs n times O(n).

This implies that the time complexity of the Hungarian algorithm implementation is $O(n^3)$.

Conclusion

In conclusion, the greedy approach, with a time complexity of O(n^2), provides a practical and efficient solution for task assignment, suitable for smaller datasets. On the other hand, the Hungarian algorithm, with a higher time complexity of O(n^3), excels in finding optimal solutions for larger and more complex problems. The choice between the two approaches should be based on the specific requirements and scale of the assignment problem. The greedy approach offers speed, while the Hungarian algorithm prioritizes optimality.

References

1. **GeeksforGeeks.** Sort in C++ Standard Template Library (STL). [Online] September 23, 2023. [Cited: November 11, 2023.] https://www.geeksforgeeks.org/sort-c-stl/.
2. **Neapolitan, Richard.** *Foundations Of Algorithms.* Sudbury : Jones and Bartlett Publishers, Inc., 2014. Vol. 5th.
3. **farnboroughmaths.** *Hungarian Algorithm.* YouTube : s.n., January 24, 2013.

Table of tables